ISSN 1330-3651 (Print), ISSN 1848-6339 (Online)

https://doi.org/10.17559/TV-20180123005000
Original scientific paper

High Performance Twitter Sentiment Analysis Using CUDA Based Distance Kernel on GPUs

Ferhat BOZKURT, Önder ÇOBAN, Faruk Baturalp GÜNAY, Şeyma YÜCEL ALTAY

Abstract: Sentiment analysis techniques are widely used for extracting feelings of users in different domains such as social media content, surveys, and user reviews. This is mostly performed by using classical text classification techniques. One of the major challenges in this field is having a large and sparse feature space that stems from sparse representation of texts. The high dimensionality of the feature space creates a serious problem in terms of time and performance for sentiment analysis. This is particularly important when selected classifier requires intense calculations as in k-NN. To cope with this problem, we used sentiment analysis techniques for Turkish Twitter feeds using the NVIDIA's CUDA technology. We employed our CUDA-based distance kernel implementation for k-NN which is a widely used lazy classifier in this field. We conducted our experiments on four machines with different computing capacities in terms of GPU and CPU configuration to analyze the impact on speed-up.

Keywords: CUDA; *k*-NN; LDA; parallel computing; sentiment analysis; twitter

1 INTRODUCTION

Social media is a strong communication way that impacts the events and many fields directly. Today, users can state their feelings, thoughts, ideas, and experiences about any matter in various social media environments such as Facebook, Google+, and Twitter [1]. On the other hand, these platforms are used by a growing number of users for news sharing and organizing events [2]. In this respect, social media offers a rich data source that could be leveraged in research areas such as economy, trade, politics, and opinion mining. Sentiment analysis is a monitoring method used to detect sentiment on social media content and it is considered as a classical text classification problem. Identifying ideas related to political matters of communities, moods of users, and negative and positive comments about any product are some of the widely used functionalities of this field [3]. Despite the popularity of sentiment analysis, high dimensionality of feature space makes it harder in terms of time and performance. In this study, we are particularly interested in applying sentiment analysis in Turkish which is known as an agglutinative language. Specific characteristics of Turkish make it difficult to perform Natural Language Processing (NLP) tasks (e.g., Named Entity Recognition, Sentiment Analysis, Word Sense Disambiguation etc.) as it requires effective pre-processing and increases the feature space further [4], [5]. Structural processing in Turkish is more difficult than in English and performing sentiment analysis for Turkish is a challenging task due to its rich morphology and abundance of dialectal use in Twitter [6]. For instance, morphological suffixes may change the polarity of words and it is possible to produce many surface forms of a word from its root in Turkish [7].

Therefore, sentiment analysis on Turkish texts is not an easy task with the help of single threaded executions of learning algorithm. On the other hand, most of the existing classification methods (e.g., neural network based classifier with a high number of neurons, decision tree classifier with highly irregular shape of tree and variable number of nodes at run time) are not suited for efficient parallel processing and this also makes harder to handle large datasets [8], [9]. Single threaded executions often lead to underutilized computational power causing high executing times. Applying some of the algorithms such as sample-based classifiers (e.g., *k*-Nearest Neighbors) to a complex language (e.g., Turkish) exacerbate these performance problems and makes it even harder to tackle.

The execution time of *k*-Nearest Neighbors (*k*-NN) method increases dramatically when size of the input dataset and value of *k* is large. Nevertheless, *k*-NN algorithm can be easily executed in parallel mode [10]-[12], since the computation of similarity or distance between test and training samples is independent in this algorithm. Scalable solutions can also be achieved by leveraging the use of effective parallel computer architectures like GPUs (Graphics Processing Units). Nowadays, GPUs are broadly available and relatively cheap. Due to high parallel computation power, GPU technology is highly preferred for handling machine learning problems that require intensive computational tasks. There are many studies in the literature making use of GPUs for speeding up machine learning algorithms [13]-[16]. Currently, CUDA (Compute Unified Device Architecture) is a commonly used programming language for developing applications for GPUs.

In this study, we focus on speeding up the sentiment analysis task. In order to do so, we selected the *k*-NN as our classifier and parallelized its nearest neighbor search process which takes the longest time in it. Note that an approximate speed-up can also be achieved by using dimension reduction or feature selection techniques. However, these aspects of speed-up are out of the scope of this article. Calculation of the distances between vector pairs belonging to training and test samples can be performed independently which makes it possible to execute *k*-NN algorithm in parallel with CUDA GPUs [10]-[12]. Therefore, in this study, we implemented a CUDA-based distance kernel for *k*-NN classifier. Our main goal is to speed-up classification process in Turkish Twitter sentiment analysis which requires high computational power. For this purpose, we employed the *k*-NN method on four machines with different processors and graphics cards in serial and parallel fashion and monitored the impact of the parallel execution both in terms of time and speed-up on classification task. We also carried out serial and parallel tests on three different sub-sets of the dataset and observed the scalability of parallel kernel. Based on these observations, we determined the required optimum thread grid and block size to obtain maximum speed-up in parallel architecture. The contributions of this paper are summarized as follows: (1) we performed sentiment classification using LDA (Latent Dirichlet Allocation) for Twitter feeds; (2) CUDA based distance kernel for *k*-NN algorithm was implemented to overcome performance problems which caused by the high dimensionality of feature space; (3) according to studies in literature, there is no previous work which is similar to ours in the context of Turkish Twitter sentiment analysis using CUDA.

The rest of the paper is organized as follows. In Section 2, we discuss previous works which employ parallel implementations of machine learning algorithms on large datasets using CUDA architecture and Hadoop MapReduce paradigm. In Section 3, we describe our methodology which has sub-headings such as Twitter Text Processing and CUDA Parallel Computing Architecture. We present our experimental results in Section 4 and finally, we conclude our study in Section 5.

2 RELATED WORKS

Although there are studies on speeding up Twitter sentiment analysis in the literature, our study is the first of its kind targeting this problem for Turkish language. Some of the existing studies teach how to use parallel machine learning implementations for this problem by employing NVIDIA's CUDA technology and Hadoop MapReduce paradigm. In [17], authors implemented and compared parallel algorithms for pre-processing of Twitter feeds based on GPU and Hadoop MapReduce architectures. This paper presents the effectiveness and obstacles of using parallel algorithm methods for effective pre-processing (i.e., some tasks like string to vector conversion, elimination of trivial words and symbols, and frequency mapping etc.) of data. In another study [18], authors evaluated the scalability of Naive Bayes classifier on large-scale datasets. They performed sentiment mining on large datasets by using a Naive Bayes classifier with the Hadoop framework.

A novel parallel implementation of Naive Bayesian classifier is proposed by [19] to decrease the test time complexity while handling large datasets. In [20], authors introduced parallel implementations of several classification algorithms including *k*-NN, Naive Bayesian model, and decision tree based on MapReduce, which make these classifiers to be applicable to mine large datasets. A CUDA-based parallel implementation of *k*-NN algorithm for spam classification has been carried out by [21]. The performance of the *k*-NN search degrades dramatically for large datasets as the task is computationally intensive. In their study, the distances between the newly-processed email and each email in the training set are quickly calculated by using the GPU. In [10]-[12] authors worked on CUDA-based parallel implementation of *k*-NN algorithm to maximize the utilization of the GPU for scalable and fast *k*-NN computation. In [22], authors proposed a method for processing all *k*-NN queries in Hadoop. They decompose the given space into cells and execute a query using the MapReduce framework in a distributed and parallel manner.

3 MATERIALS AND METHODOLOGY

In this section, the dataset and methodology used in this paper are described under three sub-headings. First, we clarify text processing steps which we applied on Turkish Twitter Feeds (TTF) dataset. Then, we define the CUDA parallel computing architecture and our parallel distance calculation kernel respectively.

3.1 Selected Dataset as Evaluation Material

In this study, we used the TTF dataset created in [23]. This dataset contains randomly collected public Twitter feeds by sending emoticons as a query key [23], [24]. The TTF dataset has totally 20000 Twitter feeds (after pre-processing) which are equally distributed between negative and positive categories [25]. Each tweet in this dataset is labeled by using distant supervision approach that considers whether related tweet contains an emoticon from the following positive and negative groups:

- Positive group: ":-)", ":)", "=)", ":D"
- Negative group: ":-(", ":(", "=(", ";("

To assign category, it is enough to include one of the emoticons in the positive or negative group for a tweet. Note that tweets that include emoticons from both positive and negative groups are eliminated in crawling process and not included in the dataset.

3.2 Methodology

3.2.1 Twitter Text Processing

In the first stage of the methodology, the following steps are employed respectively for Twitter text processing.

Pre-processing: In pre-processing phase, each of Twitter messages is cleaned by removing incomprehensible contents such as words, characters, decimal values, and punctuation marks after the lowercase conversion. We did not apply term frequency and term length filter in order to avoid low feature space. Then, to get significant short term expressions, we applied the following steps:

- We removed the retweet, retweeted, and repetitive messages from the dataset,
- We coded the Twitter specific terms (i.e., usernames, URL, and hashtag) with specific terms and preserved thought to be effective in sentiment classification phase.

Note that we remove these terms after sentiment classification.

Sentiment Classification: In sentiment analysis, the major challenge is to decide which category is most accurate for a tweet. Therefore, different techniques such as using emoticons or lexicon-based linear classifiers are employed in the literature for this purpose [23], [24], [26]. In this study, we use topic based sentiment classification method which is proposed in [27]. Using this method, a topic model is created that comprises 2 topics by using the LDA algorithm. The most observed 50 words in these 2 topics were extracted and renamed each of them by hand as positive or negative according to the number of observed positive or negative words. Then, for each tweet in dataset we obtained a likelihood value that indicates which topic is closer for related tweet. Finally, we assigned category for each tweet by depending on its closeness to positive or negative topic [27] as shown in **Algorithm 1**. We used MALLET package to implement the LDA [28] algorithm that its notation is depicted in Fig. 1. LDA is a topic modelling approach which assumes the dataset includes T hidden topics. Each document m consisting of N_m words has a multinomial φ_m distribution on these T topics. A $z_{m,n}$ topic can be determined for each $w_{m,n}$ word which is observed in document m by this φ_m distribution. In this way, a $w_{m,n}$ word for $z_{m,n}$ topic can be sampled from obtained $\varphi_{z_{m,n}}$ document-topic distribution [29]. Here, the word-topic φ_m and document-topic $\varphi_{z_{m,n}}$ distributions have α and β dirichlet priority parameters. In this paper, Gibbs sampling algorithm is used for the purpose of determining these parameters [30].

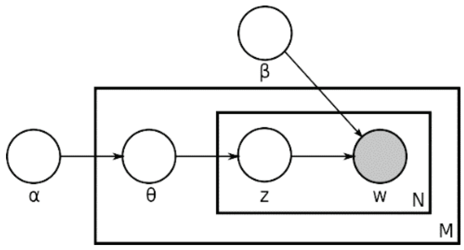

Figure 1 Latent Dirichlet Allocation

After the sentiment classification, we eliminated all of the Twitter specific terms to avoid biased results as these terms are most frequently observed terms in each category.

Algorithm 1 Sentiment classification using LDA [27]

```
procedure LABELTWEETS(Tweets)
    p = positive, n = negative
    topic = 2, iteration = 2000
    topicModel ← LDA(topic, iteration)
    for each topic t in topicModel do
        W = most observed terms in t
        nT = # of n terms in W
        pT = # of p terms in W
        if pT > nT then
            t.label = p
        else
            t.label = n
        end if
    end for
    for each tweet twt in Tweets do
        cTopic = topicModel.getCloserTopic(twt)
        if cTopic = p then
            twt.category = p
        else
            twt.category = n
        end if
    end for
    return Tweets
end procedure
```

Feature Extraction: In this step, we used Bag of Words (BOW) model to extract features from Twitter feeds. In this model, each document sample is usually represented by associating the word and its observed frequency [31]. The order of words in text content is also considered as to be not important [32]. After obtaining the BOW features, we applied removal of stopwords, stemming, and text normalization steps by using Zemberek and Lucene APIs as in [33].

Representation and Term Weighting: We represented each tweet, contained in TTF dataset, as a feature vector by using VSM (Vector Space Model) [34]. In VSM, each feature is represented as its term frequency or weight value using numbers. Therefore, term weighting is an important step which assigns a weight to indicate importance of each related feature in sample vector. In this step, we used *TF*IDF* (Term Frequency-Inverse Document Frequency) method which is most commonly used weighting scheme in text classification [35], [36]. The *TF*IDF* applies weighting to a feature based on its inverse document frequency and term frequency factors. This means that if in more tweets a term appears, the less important that term will be, and the weighting will be less. The *TF*IDF* can be formulated as Eq. (1):

$$W_{TF*IDF} = tf_{ij} \times \log\left(\frac{N}{n_j}\right) \tag{1}$$

where tf_{ij},N, and n_j represent the raw term frequency of term j in a tweet i, total number of tweets in the dataset, and the number of tweets that term i appears respectively.

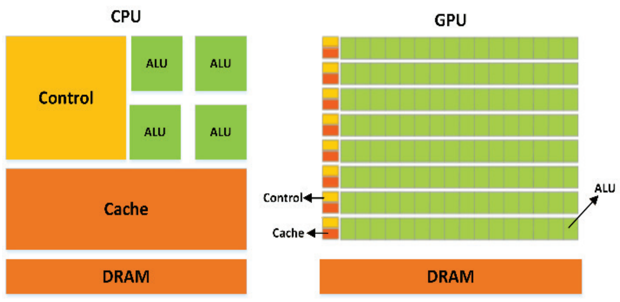

Figure 2 GPU allocates more transistors for data processing [41]

3.2.2 GPU Parallel Computing Architecture and CUDA Programming Model

CUDA is a parallel computing architecture which allows significant increases in computing performance by using GPU power of NVIDIA. As a result of the development of TESLA GPU architecture [37], NVIDIA has shown that GPUs can be programmed by considering like a processor. Previously, GPUs were mainly used for graphics applications. Research conducted on CUDA architecture revealed that the performance is higher when compared to CPU [38], [39]. The reason of this difference is that the GPU architecture has been developed for the operations requiring parallel calculations at high degrees and operation intensity such as graphics processing. Contrary to the operations requiring a flow control as in CPU, the GPU target parallel calculation applications such as data processing, image processing, 3D rendering, and signal processing consist of the repeating and arithmetically intense operations [40]. As shown in Fig. 2, GPU parallel calculation enables an architecture that has intense transistors devoted to data processing instead of cache memory and flow control mechanisms in CPU [41]. The excessive performance difference between multi-core GPUs and general-purpose multi-core CPUs arises from basic design difference between two processors. Compared to CPU, GPU has more ALU (see Fig. 2) and includes less components (e.g., cache memory and flow control). This is an important factor in terms of obtaining high arithmetic operation power and capacity that are required for performing parallel arithmetical operations.

GPU also allows to carry out same operations on different data elements in compliance with SIMD (Single Instruction Multiple Data) programming [39] and the small cache memory on GPU enables to control the bandwidth required by the application. Therefore, there is no need to turn back to DRAM (Dynamic Random Access Memory) for threads which are trying to have an access to data in the same memory area. Thus, results are obtained much faster for any application that has parallelizable functions by running on GPU [42], [43]. CUDA is quite useful and practical thanks to its easy thread management structure and being a software with shared memory that can be used on GPU. In addition, it enables the applications written in C programming language for CPU to be run by using multi-thread on graphics processor as it is a C based parallel programming language [44]. In this aspect, CUDA programming with NVIDIA GPUs also provides adequate API for non-graphical applications.

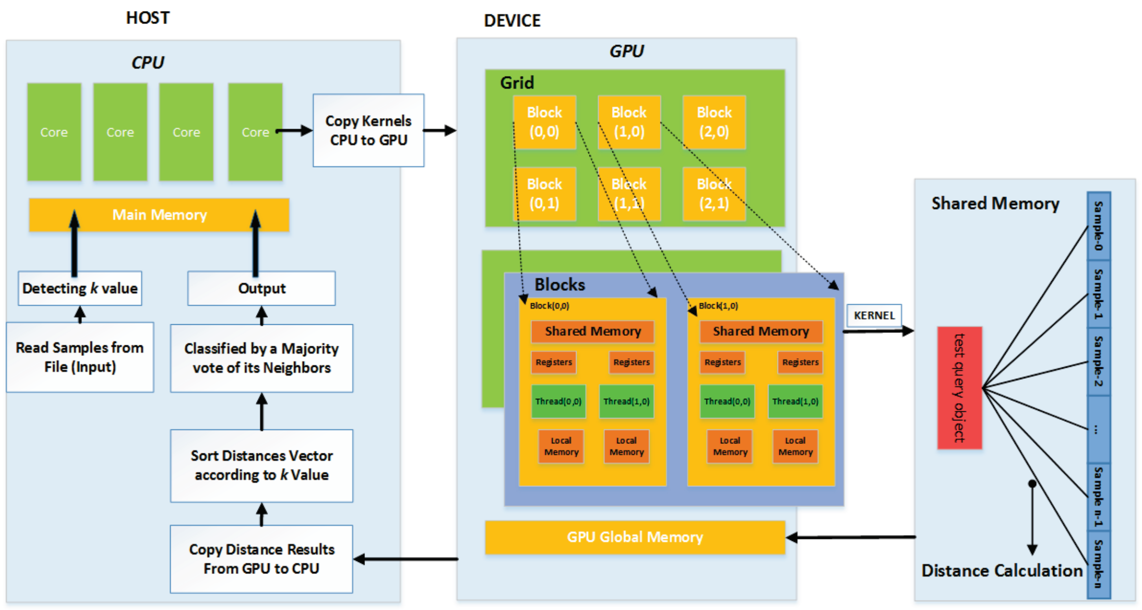

Figure 3 Flowchart of our CUDA-based parallel *k*-NN

Table 1 GPU and CPU configurations of four machines used in experiments

	Device configuration	Machine I (MI)	Machine II (MII)	Machine III (MIII)	Machine IV (MIV)
GPU	NVIDIA graphics card	GeForceGTX850M	GeForceGT650M	GeForce9600MGT	Does not have NVIDIA graphics card
	Total amount of global memory	4096 MBytes	4096 MBytes	512 Mbytes	
	# of CUDA cores	640	384	32	
	Warp size	32	32	32	
	Max. # of threads per multiprocessor	2048	2048	768	
	Max. # of threads per block	1024	1024	512	
	Max. size of each dimension of a thread block	1024×1024×64	1024×1024×64	512×512×64	
	Max. size of each dimension of a grid	2147483647×65535×65535	2147483647×65535×65535	65535×65535×1	
CPU	Processor (Intel)	i7-4700HQ	i7-3630QM	Core2Duo CPUP8400	i5-4260U
	Processor base frequency	2,40 GHz	2,40 GHz	2,26 GHz	1,40 GHz
	Cache	6 MB Smart Cache	6 MB Smart Cache	3 MB L2	3 MB Smart Cache
	RAM	16 GB	16 GB	2 GB	8 GB

CUDA-Enabled GPU consists of SIMD SMs (Stream Multiprocessors) clusters and each of these clusters includes 8 stream processors (SPs). Accordingly, CPU acts as a multi-core co-processor CUDA device and SMs form the graphics card hardware [10], [39], [45]. Each of the SM has a fast-shared memory and shares it with all other processors. And each SP includes a 32 byte local register cluster. SMs make contact by global/device memory and shared memory is explicitly managed by programmers [10]. When we consider the CUDA technology as a software, it actually comprises parallel running thread collection. In this way, software can run much faster by creating parallelism for arithmetical operations requiring heavy calculations. Any program written in CUDA is actually a serial program named as a kernel. This kernel identifies the operations to be carried out for a specific dataset [10], [39]. GPU runs thousands of copies of this kernel and makes it as parallel.

A system with shared memory on GPU is carried out by installing driver software that provides parallel programming support of the producing company. When the program is executed, CPU runs serial part of the code and GPU runs parallel CUDA code which requires intense calculations. GPU creates a separate kernel copy for each sample in dataset and these kernel copies are called as thread [41]. As shown in Fig. 3, thread groups are consisted by combination of thread structures. Thread groups create the blocks and the block groups are called as grid. Grids are comprised of thread blocks in computing organization and make the operations parallel by running copies of GPU kernel. SM performs one or more thread blocks at the same time. The threads in common block communicate by common shared memory and each thread includes a program counter, register, and state bar belonging to itself. In this context, the basis of CUDA technology is based on running GPU by many thread series. All of the threads can run same code and each thread has an ID. As CUDA is an extension of C language, it is generally not necessary to change its architecture to direct the programs to CUDA or transform them to multi-thread form. CUDA includes both C language extensions and runtime time library that provides API for GPU control [39], [42]. Therefore, millions of threads can be created simultaneously making parallel processing possible in various areas such as image and data processing [42].

3.2.3 Parallel Implementation

We selected the *k*-NN algorithm to implement in parallel fashion. Main reasons behind our selection are as follows:

- *k*-NN is generally the slowest algorithm among machine learning algorithms, since it often outperforms other well-known classifiers (e.g., Decision Tree, Naïve Bayes etc.) in terms of computation complexity or accuracy especially in text categorization,
- Compared to the other classification algorithms, this method is more suitable to run in parallel mode, as searching for nearest neighbors can be calculated independently and this task is the most time consuming part of the *k*-NN.

Note that implementing other classifiers is also possible, but each algorithm has its own calculation strategy and not all of the algorithms have the same parallelizable potential. For instance, compared to the *k*-NN, running Naïve Bayes algorithm in parallel fashion may not provide a huge performance difference when compared to its serial execution. Therefore, in this study, we selected the *k*-NN algorithm which is also known as lazy learner and requires intense calculation especially in text categorization. As distance calculation is the most time-consuming part in *k*-NN classification, we performed this calculation in parallel using our distance calculation kernel.

***k*-Nearest Neighbor Classifier:** *k*-NN is a supervised learning algorithm based on the distance (or similarity) between the samples [46]. In this method, the classification process is time consuming and it is hard to estimate the most suitable *k* value [47]. *k*-NN classifier finds the *k* closest neighbors in accordance with distance function and uses category weights of these neighbors to assign a category to test sample [48], [49]. The distance (e.g., Euclidean, Manhattan, and Minkowski etc.) or similarity function may be different [11], [46]. However, Euclidean distance is generally used to determine the closest neighbors in *k*-NN classifier [50]. Let d_0, $d_j \in k-NN(d_0)$ and C represent test sample, k closest neighbors and category set respectively. Then, assigning a category to a test sample using the category weights of its k closest neighbors is performed as in Eq. (2) and (3):

$$score(d_0, C_j) = \sum_{d_j \in k-NN(d_0)} Sim(d_0, d_j)(d_j, C_i) \tag{2}$$

$$C = argmax_{C_i}(score(d_0, C_i)) \tag{3}$$

In above equations, the (d_j, C_i) statement takes value of 1 if the category of document d_j is C_i, 0 otherwise. Finally, the classifier assigns the category which has higher weight to the test sample [51].

Table 2 The effect of pre-processing on TTF dataset

Feature reduction	# of features	Percent of original (%)
None	155603	100
(-)Hashtag{#}	1512	0,97
(-)URL{http://}	3463	2,22
(-)Username{@}	12436	7,99
(-)Emoticon{:),-),:(}	21944	14,1
(-)FilteredTerm	39130	25,1
(-)NormalizedTerm	2530	1,62
(-)All	81015	52,0

Distance Calculation Kernel: The most time-consuming components of *k*-NN algorithm are distance calculation and sorting process [52]. Even though this expense may be decreased with some indexing methods (e.g., K-D tree), it is observed that this operation can be performed faster with less expense by making it parallel [38]. Therefore, we focused on speed-up the distance calculation component in this study. In this way, the consistency between different threads is raised to top level in distance calculation. Also, the access to global memory is minimized as it has a delay. As the distance calculation between training and test vector pairs is independent, it can be calculated in parallel. Therefore, the *k*-NN algorithm is perfectly suitable to be implemented in parallel with GPUs [12]. In this study, firstly, we detected the optimum *k* value for *k*-NN classifier. Secondly, we applied the following steps:

- For each test sample, we calculated the distance between test and training samples in parallel mode by using GPUs (i.e., Distance Calculation Kernel),
- We sorted all training samples based on their distance and selected the *k* nearest neighbours by minimum distance,
- We assigned the category which has highest weight among the *k* nearest neighbours to the test sample.

In our CUDA-based *k*-NN implementation (see Fig. 3) which includes a parallel distance calculation kernel, the training and test data are read at first. Then, optimum *k* value to be used in classification process is detected in the serial part. However, the Euclidean distance between test and training samples is calculated by different threads by transferring data from CPU to GPU in parallel distance calculation kernel [10], [11], [52]. These distances are obtained quickly and independently from each other in parallel part. In this process, a vector pair is obtained from both test and training samples which are contained in global memory and placed on shared memory in each block. Then, the entire SPs in each block fetch data from common shared memory. Using this approach, the threads in the common blocks share reference data (i.e., a unique sub-set of training data) in the shared memory. There are many blocks and threads that perform this process as the training reference data is huge.

Table 3 Evaluation of the optimum *k* value for highest accuracy on MII

Evaluation metric	# of nearest neighbours (*k*)							
	1	3	5	7	9	11	13	15
Precision	0,66	0,65	0,65	0,66	0,65	0,63	0,63	0,64
Accuracy	0,77	0,81	0,81	0,82	0,83	0,83	0,84	0,85
Recall	0,53	0,61	0,61	0,63	0,66	0,66	0,69	0,70
Specifity	0,88	0,88	0,88	0,89	0,89	0,88	0,88	0,89
F-measure	0,59	0,63	0,63	0,64	0,65	0,64	0,66	0,67

Table 4 Obtained speed-up by MII on TTF dataset for the different *k* values and thread block sizes

Thread block size	# of nearest neighbours (*k*)																							
	1			3			5			7			9			11			13			15		
	CT	GP	SU	CT	GP	SU	CT	GP	SU	CT	GP	SU	CT	GT	SU	CT	GT	SU	CT	GT	SU	CT	GT	SU
1024×1024×1	300689	3765	79,86	300832	3788	79,41	300966	3745	80,36	302181	3729	81,03	302964	3742	80,96	304577	3790	80,36	309860	3870	80,06	314524	3860	81,48
512×512×1		3816	78,79		3800	79,16		3756	80,12		3778	79,98		3753	80,72		3785	80,46		3891	79,63		3870	81,27
256×256×1		3837	78,36		3809	78,97		3878	77,60		3787	79,79		3772	80,31		3787	80,42		3952	78,40		3882	81,02
128×128×1		3918	76,74		3830	78,54		4086	73,65		3810	79,31		3910	77,48		3801	80,13		4007	77,32		3912	80,39
64×64×1		3967	75,79		3872	77,69		4266	70,54		3903	77,42		4001	75,72		3811	79,92		4050	76,50		3949	79,64
32×32×1		4135	72,71		4023	74,77		4450	67,63		4295	70,35		4237	71,50		3915	77,79		4105	75,48		4055	77,66

Consequently, the distance calculation kernel is being parallel in compliance with parallel logic. The distances of each test sample to training samples are calculated respectively in conclusion with running of distance calculation kernel in threads. After that, the results of each distance calculations are stored in a vector in order to sort by *k* value. After the distances are calculated in parallel kernel, the vector including distance calculations is sent to CPU. Then, the samples are sorted by minimum distance and category weights of the *k* closest neighbours are obtained in CPU. Since the number of samples used in the voting phase is *k*, which is generally set to a small value in order to maintain accuracy and avoid over-fitting, the computation for this step is negligible. For this reason, we execute this step on the CPU.

4 EXPERIMENTAL RESULTS

In this section, the results for our CUDA- based parallel *k*-NN implementation are analyzed on TTF dataset. First, we describe the configuration and evaluation metrics that is employed and then, our results are presented.

4.1 Configuration and Evaluation Metrics

We conducted the experiments on four machines which have different CPU and GPU configurations. We present the detailed information about every single machine in Tab.1. For instance, MII has Intel Core i7-360QM processor (2.40 GHz), 6 MB cache, 16 GB main memory, GeForce GT 650M graphics card, and 384 CUDA cores. This machine also has CUDA 7.5 software and NVDIA 353.90 graphics card driver.We obtained the speed-up value that is equal to the execution time of CPU, divided by the execution time of GPU [53], [54]. In all experiments, we employed the Holdout [55] validation method by taking 90% and 10% of the dataset as training and test set respectively. Evaluation metrics are the key to understanding how classification model performs when applied to a test dataset. In other words, a metric evaluates the quality of an engine by comparing the engine's output (i.e., predicted result) with the original (i.e., actual result) label. Therefore, we report weighted average values of well-known evaluation metrics such as F1-score, precision, recall, and accuracy [56], [57].

4.2 Results

In this section, experimental results are presented. Our main goal is to investigate the speed-up of our CUDA-based *k*-NN implementation. For this purpose, we pre-processed the TTF dataset and transformed it into classification-ready structure. After the sentiment classification, we eliminated the emoticons and Twitter specific terms from the dataset. For this reason, we also removed some messages from the dataset as they do not have enough content. Thus, we obtained 4578 unique features from the remaining 18760 messages in TTF dataset. As seen from Tab. 2, this has enabled us to reduce the feature space at the rate of 52%. After the pre-processing, we investigated the speed-up in two different phases. Firstly, we conducted experiments on CPU for determining the optimum *k* value in terms of accuracy. As shown in Tab. 3, we detected that the optimum *k* value is 15 which gives the best accuracy at the rate of 85%. Then, we also used different threads per block sizes including 32×32×1, 64×64×1, 128×128×1, 256×256×1, 512×512×1, and 1024×1024×1 to detect optimum thread value in each block.

In this step, we conducted experiments by taking the serial and parallel running times to calculate the speed-up for each *k* value ranging from 1 to 15 again.We changed the number blocks on the grid and the number of threads in each block manually from within the program for the parallel distance kernel function. One point to mention here is that we set the number of blocks on the grid as equal to the number of samples and features in the dataset. When the number of samples exceeds the maximum number of blocks supported by graphics card, we accepted the maximum number of blocks size on the grid as maximum.

We show our results in Tab. 4 which indicates the effect our parallel distance calculation kernel on speed-up for the cases where number of threads and the number of nearest neighbors (i.e., *k* value) are different. The CT, GP, and SU represent CPU time (in *ms*), GPU time (in *ms*), and speed-up (CPU time/GPU time) respectively. According to Tab. 4, the speed-up was changed in range from 67 to 81 times for the number of threads per block that was between 32×32×1 and 1024×1024×1. We achieved the best speed-up till 81.48 times for the cases that the *k* and thread block size as equal to 15 and 1024×1024×1 respectively. Therefore, in our all subsequent experiments, we have taken the optimum values of both *k* and thread block size as equal to 15 and 1024×1024×1 respectively. Tab. 4 shows that as the *k* increases, the CPU time also increases. In addition, the difference between the total CPU time and the GPU time also tends to increase depending on the increase in *k* value. Therefore, the classification process is faster in the small values of *k,* but the resulting speed-up is higher for large values of *k*. The reason for this is that other parts of the *k*- NN classification take more time as *k* increases. In addition to these findings, we investigated the GPU running time and the speed-up depending on the number of threads for optimum *k* value. We obtained the results for this experiment as shown in Fig. 4. When we consider Fig. 4, we observe that as the number of threads increases, the speed-up can reach up to 81 times. In addition, we investigated the effect of the number of samples in dataset on speed-up. We also detected that there is a right proportion between the number of samples and speed-up as shown in Tab. 5. This shows the scalability of our parallel distance calculation kernel.

Table 5 Obtained speed-up by MII depending on the number of samples in three different subsets of the TTF dataset

GPU vs. CPU	# of samples (*n*)		
	6000	12000	18000
Speed-up	52,8	72,8	81,4

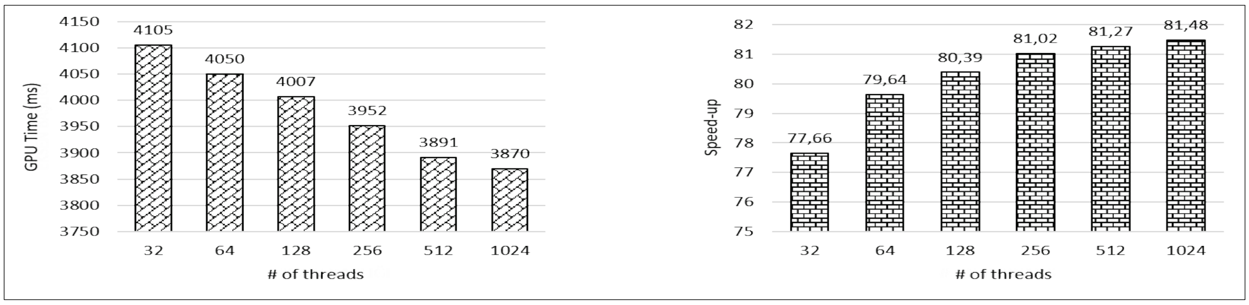

Figure 4 Effect of the number of threads for the TTF dataset on the GPU execution time and the speed-up of the *k*-NN on MII

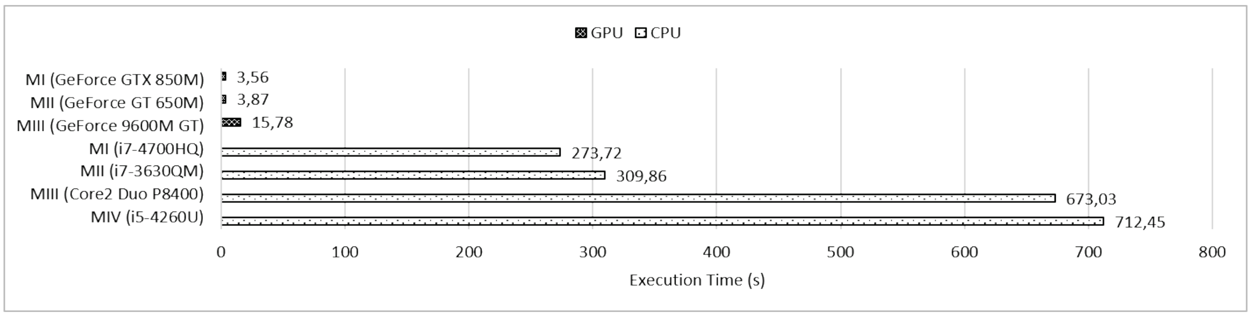

Figure 5 Comparison of CPU and GPU execution time on TTF dataset for k-NN classfication using four different machines listed in Tab.1

In the first stage, we performed all of these experiments on a single machine (MII) to prevent the difference in terms of hardware and software source and parameters. Secondly, we performed the *k*-NN classification both in serial and parallel on different machines which have different CPU and GPU configurations. We also investigated the effect of both processor and CUDA configuration on speed-up. Based on our results, we observed that while the classification process approximately takes 5-12 minutes with the CPU, this process takes 3-15 seconds (see Fig. 5) by using CUDA GPUs. In summary, we obtained the highest speed-up as 81,48 times in all of our experiments in which the classification results both in serial and parallel as identical.

5 CONCLUSIONS

The high dimensionality of feature space and the number of samples especially on large data sets are major problems in Twitter sentiment analysis as it is considered as a text classification task in general. This situation causes disadvantages in terms of time and performance. Moreover, it needs high calculation power when it uses a lazy classifier like *k*-NN. Therefore, in this study, it has been intended to speed-up Twitter sentiment analysis with CUDA based parallel *k*-NN classifier. For this purpose, the sentiment analysis techniques were applied on TTF dataset. Firstly, the sentiment classification of tweets was performed by using the LDA method and preprocessed tweets were transformed into the classification-ready structure. To achieve our goal, we performed different experiments for varying number of samples, thread block sizes, and *k* values to investigate the effect on results. In Tab. 6, we summarize all of our experiments which are composed of two phases. Based on all of our observations, in this paper, we have the following conclusions in summary:

- A large number of obtained features from the TTF dataset produce high feature space. This clearly shows that the feature space to be processed on large datasets would be much higher. Therefore, there is need to parallelize the sentiment analysis task especially when selected classifier (e.g., *k*-NN) requires intense calculations.
- For the *k*-NN algorithm, determining the optimum *k* value, sorting samples by minimum distance, and determining the category weights of the nearest neighbors do not provide any remarkable gain in terms of performance when they are performed in parallel fashion. This proves that the distance calculation

process is the most time consuming part of the *k*-NN classifier.

- It is possible to achieve 81,48% speed-up by employing CUDA in sentiment analysis even though the TTF is a small dataset. This shows that the speed-up will increase even more over larger datasets.
- The speed-up increases too when we increase the number of tweets and use higher values of *k* as these parameters cause to increase in search space when detecting nearest neighbors. Thread block size and the number of threads have also positive effect on speed-up through increasing these parameters also means having more workers to perform distance calculation.
- Machines that have i7 processor run serial code faster and the best performance is provided by GeForce GTX 850M driver among NVIDIA graphics cards. Achieved speed-up could be increased even further when running the parallel CUDA code on a more powerful machine (e.g., MI).

As a future work, we are planning to make the TTF dataset larger and compare NVIDIA's CUDA technology with Hadoop MapReduce paradigm in terms of the speed-up.

Table 6 The summary of our experiments

Experiments		Conducted on	
		CPU	GPU
First phase on single machine (Machine II)	Optimum *k* for higher accuracy	Tab. 3	-
	Optimum thread block size for higher speed-up	Tab. 4 (CPU and GPU)	
	The effect of the number of samples on speed-up in three different subsets of TTF dataset	Tab. 5 (CPU and GPU)	
	The effect of the number of threads time and the speed-up of the *k*-NN	Fig. 4 (CPU and GPU)	
Second phase on four machines (see Tab. 1)	Optimum thread block size for higher speed-up	Fig. 5 (CPU and GPU)	

Abbreviations and symbols

API	–	Application Programming Interface
ALU	–	Arithmetic Logic Unit
BOW	–	Bag of Words
CPU	–	Central Processing Unit
CT	–	CPU Time
CUDA	–	Compute Unified Device Architecture
DRAM	–	Dynamic Random Access Memory
GPU	–	Graphics Processing Units
GT	–	GPU Time
IDF	–	Inverse Document Frequency
k-DT	–	*k*-Dimensional Tree
k-NN	–	*k*-Nearest Neighbors
LDA	–	Latent Dirichlet Allocation
NLP	–	Natural Language Processing
SIMD	–	Single Instruction Multiple Data
SM	–	Stream Multiprocessor
SP	–	Stream Processor
SU	–	Speed-up
TF	–	Term Frequency
TTF	–	Turkish Twitter Feeds
VSM	–	Vector Space Model

6 REFERENCES

[1] Sommer, S., Schieber, A., Hilbert, A., & Heinrich, K. (2011). Analyzing customer sentiments in microblogs–A topic-model-based approach for Twitter datasets. *Proceedings of the Americas conference on information systems (AMCIS).*

[2] Michelson, M. & Macskassy, S. A. (2010). Discovering users' topics of interest on twitter: a first look. *Proceedings of the fourth workshop on Analytics for noisy unstructured text data*, ACM, 73-80. https://doi.org/10.1145/1871840.1871852

[3] Agarwal, A., Xie, B., Vovsha, I., Rambow, O., & Passonneau, R. (2011). Sentiment analysis of twitter data. *Proceedings of the workshop on languages in social media*.

[4] Karayiğit, H., Acı, Ç., & Akdağlı, A. A Review of Turkish Sentiment Analysis and Opinion Mining. *Balkan Journal of Electrical and Computer Engineering, 6*(2), 26-30. https://doi.org/10.17694/bajece.419547

[5] Yıldırım, E., Çetin, F. S., Eryiğit, G., & Temel, T. (2015). The impact of NLP on Turkish sentiment analysis. *Türkiye Bilişim Vakfı Bilgisayar Bilimleri ve Mühendisliği Dergisi, 7*(1), 43-51.

[6] Rosenthal, S., Farra, N., & Nakov, P. (2017). SemEval-2017 task 4: Sentiment analysis in Twitter. *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, 502-518. https://doi.org/10.18653/v1/S17-2088

[7] Dehkharghani, R., Saygin, Y., Yanikoglu, B., & Oflazer, K. (2016). SentiTurkNet: a Turkish polarity lexicon for sentiment analysis. *Language Resources and Evaluation, 50*(3), 667-685. https://doi.org/10.1007/s10579-015-9307-6

[8] Saldanha, L. B. & Bobda, C. (2015). An embedded system for handwritten digit recognition. *Journal of Systems Architecture, 61*(10), 693-699. https://doi.org/10.1016/j.sysarc.2015.07.015

[9] Srivastava, A., Han, E. H., Kumar, V., & Singh, V. (1999). Parallel formulations of decision-tree classification algorithms. *High Performance Data Mining*, Springer, Boston, MA, 237-261. https://doi.org/10.1007/0-306-47011-X_2

[10] Liang, S., Liu, Y., Wang, C., & Jian, L. (2010). Design and evaluation of a parallel k-nearest neighbor algorithm on CUDA-enabled GPU. *Web Society (SWS), 2010 IEEE 2nd Symposium on.* https://doi.org/10.1109/SWS.2010.5607480

[11] Arefin, A. S., Riveros, C., Berretta, R., & Moscato, P. (2012). Gpu-fs-knn: A software tool for fast and scalable knn computation using gpus. *PloS one, 7*(8), e44000. https://doi.org/10.1371/journal.pone.0044000

[12] Garcia, V., Debreuve, E., & Barlaud, M. (2008). Fast k nearest neighbor search using GPU. *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08.* https://doi.org/10.1109/CVPRW.2008.4563100

[13] Lopes, N. & Ribeiro, B. (2011). GPUMLib: An efficient open-source GPU machine learning library. *International Journal of Computer Information Systems and Industrial Management Applications*, 3, 355-362.

[14] Bekkerman, R., Bilenko, M., & Langford, J. (Eds.). (2011). Scaling up machine learning: *Parallel and distributed approaches*. Cambridge University Press. https://doi.org/10.1145/2107736.2107740

[15] Upadhyaya, S. R. (2013). Parallel approaches to machine learning—A comprehensive survey. *Journal of Parallel and Distributed Computing, 73*(3), 284-292. https://doi.org/10.1016/j.jpdc.2012.11.001

[16] Steinkraus, D., Buck, I., & Simard, P. Y. (2005, August). Using GPUs for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, IEEE, 1115-1120. https://doi.org/10.1109/ICDAR.2005.251

[17] Nirmal, V. J. & Amalarethinam, D. G. (2015). Parallel Implementation of Big Data Pre-Processing Algorithms for Sentiment Analysis of Social Networking Data. *International journal of fuzzy mathematical archive, 6*(2), 149-159.

[18] Liu, B., Blasch, E., Chen, Y., Shen, D., & Chen, G. (2013). Scalable sentiment classification for big data analysis using Naive Bayes Classifier. In *Big Data, 2013 IEEE International Conference on*, IEEE, 99-104. https://doi.org/10.1109/BigData.2013.6691740

[19] Katkar, V. D. & Kulkarni, S. V. (2013). A novel parallel implementation of Naive Bayesian classifier for Big Data. *Green Computing, Communication and Conservation of Energy (ICGCE), 2013 International Conference on*, IEEE, 847-852. https://doi.org/10.1109/ICGCE.2013.6823552

[20] He, Q., Zhuang, F., Li, J., & Shi, Z. (2010). Parallel implementation of classification algorithms based on MapReduce. *Rough Set and Knowledge Technology*, 655-662. https://doi.org/10.1007/978-3-642-16248-0_89

[21] Smithrud, J. M., McElroy, P., & Andonie, R. (2015). Massively Parallel kNN using CUDA on Spam-Classification. *MAICS.*

[22] Yokoyama, T., Ishikawa, Y., & Suzuki, Y. (2012). Processing all k-nearest neighbor queries in hadoop. *Web-Age Information Management*, 346-351. https://doi.org/10.1007/978-3-642-32281-5_34

[23] Çoban, Ö., Özyer, B., & Özyer, G. T. (2015). Sentiment analysis for Turkish Twitter feeds. *Signal Processing and Communications Applications Conference (SIU)*, 2015 23th. https://doi.org/10.1109/SIU.2015.7130362

[24] Pak, A. & Paroubek, P. (2010). Twitter as a corpus for sentiment analysis and opinion mining. *LREc, 10*(2010), 1320-1326.

[25] Sevli, O. & Küçüksille, E. U. (2017). Advertising Recommendation System Based On Dynamic Data Analysis On Turkish Speaking Twitter Users. *Tehnicki Vjesnik, 24*(2), 571-578. https://doi.org/10.17559/TV-20151020205558

[26] Kaya, M., Fidan, G., & Toroslu, I. H. (2012). Sentiment analysis of turkish political news. *Proceedings of the The 2012 IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technology*, *Volume 01* (pp. 174-180). IEEE Computer Society. https://doi.org/10.1109/WI-IAT.2012.115

[27] Çoban, Ö. & Özyer, G. T. (2016). Sentiment classification for Turkish Twitter feeds using LDA. *Signal Processing and Communication Application Conference (SIU), 2016 24th*. (pp. 129-132). IEEE. https://doi.org/10.1109/SIU.2016.7495693

[28] Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan), 993-1022.

[29] Godin, F., Slavkovikj, V., De Neve, W., Schrauwen, B., & Van de Walle, R. (2013). Using topic models for twitter hashtag recommendation. *Proceedings of the 22nd International Conference on World Wide Web*. https://doi.org/10.1145/2487788.2488002

[30] Carlo, C. M. (2004). Markov chain monte carlo and gibbs sampling. *Lecture notes for EEB*, *581*.

[31] Joachims, T. (1996). *A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization* (No. CMU-CS-96-118). Carnegie-mellon univ pittsburgh pa dept of computer science.

[32] Lewis, D. D. (1992). An evaluation of phrasal and clustered representations on a text categorization task. *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 37-50). ACM. https://doi.org/10.1145/133160.133172

[33] Çoban, Ö., Özyer, B., & Özyer, G. T. (2015, December). A comparison of similarity metrics for sentiment analysis on Turkish twitter feeds. In *Smart City/SocialCom/SustainCom (SmartCity), 2015 IEEE International Conference on* (pp. 333-338). IEEE. https://doi.org/10.1109/SmartCity.2015.93

[34] Salton, G., Wong, A., & Yang, C.-S. (1975). A vector space model for automatic indexing. *Communications of the ACM, 18*(11), 613-620. https://doi.org/10.1145/361219.361220

[35] Chen, K., Zhang, Z., Long, J., & Zhang H. (2016). Turning from TF-IDF to TF-IGM for term weighting in text classification, *Expert Systems with Applications 66 (Supplement C)*, 245-260. https://doi.org/10.1016/j.eswa.2016.09.009

[36] Polettini, N. (2004). The vector space model in information retrieval-term weighting problem. *Entropy,* 1-9.

[37] Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). NVIDIA Tesla: A unified graphics and computing architecture. *IEEE micro, 28*(2). https://doi.org/10.1109/MM.2008.31

[38] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., …, & Volkov, V. (2008). Parallel computing experiences with CUDA. *IEEE micro, 28*(4). https://doi.org/10.1109/MM.2008.57

[39] Kirk, D. B. & Wen-Mei, W. H. (2016). *Programming massively parallel processors: a hands-on approach.* Morgan kaufmann.

[40] Wei, W. & Huang, Y. (2011). Real-time flame rendering with gpu and cuda. *International Journal of Information Technology and Computer Science (IJITCS)*, 3(1), 40. https://doi.org/10.5815/ijitcs.2011.01.06

[41] Nvidia, C. U. D. A. (2011). Nvidia cuda c programming guide. *Nvidia Corporation, 120*(18), 8.

[42] Kankatala, S. (2015). Performance Analysis of kNN on large datasets using CUDA & Pthreads: Comparing between CPU & GPU.

[43] Zhang, H., Zhang, D.-f., & Bi, X.-a. (2012). Comparison and Analysis of GPGPU and Parallel Computing on Multi-core CPU. *International Journal of Information and Education Technology, 2*(2), 185. https://doi.org/10.7763/IJIET.2012.V2.106

[44] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., & Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. https://doi.org/10.1145/1345206.1345220

[45] Michailidis, P. D. & Margaritis, K. G. (2013). Accelerating kernel density estimation on the GPU using the CUDA framework. *Applied Mathematical Sciences, 7*(30), 1447-1476. https://doi.org/10.12988/ams.2013.13133

[46] Han, J., Pei, J., & Kamber, M. (2011). *Data mining: concepts and techniques.* Elsevier.

[47] Korde, V. & Mahender, C. N. (2012). Text classification and classifiers: A survey. *International Journal of Artificial Intelligence & Applications, 3*(2), 85. https://doi.org/10.5121/ijaia.2012.3208

[48] Yang, Y. & Liu, X. (1999). A re-examination of text categorization methods. *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 42-49). ACM. https://doi.org/10.1145/312624.312647

[49] Yağanoğlu, M., Bozkurt, F., & Günay, F. B. (2014). EEG tabanlı beyin-bilgisayar arayüzü sistemlerinde öznitelik çıkarma yöntemleri. *Mühendislik Bilimleri ve Tasarım Dergisi, 2*(3), 313-318.

[50] Khan, A., Baharudin, B., Lee, L. H., & Khan, K. (2010). A review of machine learning algorithms for text-documents

classification. *Journal of advances in information technology, 1*(1), 4-20. https://doi.org/10.4304/jait.1.1.4-20

[51] Tan, S. (2006). An effective refinement strategy for KNN text classifier. *Expert Systems with Applications, 30*(2), 290-298. https://doi.org/10.1016/j.eswa.2005.07.019

[52] Liang, S., Liu, Y., Wang, C., & Jian, L. (2009). A CUDA-based parallel implementation of K-nearest neighbor algorithm. In *Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. CyberC'09. International Conference on* (pp. 291-296). IEEE. https://doi.org/10.1109/CYBERC.2009.5399145

[53] Agarwal, N., Goyal, A., Maheshwari, G., & Dugtal, A. (2015). Parallel Implementation of Scheduling Algorithms on GPU using CUDA. *Architecture, 21*, 22. https://doi.org/10.5120/ijca2015906339

[54] Bozkurt, F., Yaganoglu, M., & Günay, F. B. (2015). Effective Gaussian Blurring Process on Graphics Processing Unit with CUDA. *International Journal of Machine Learning and Computing, 5*(1), 57. https://doi.org/10.7763/IJMLC.2015.V5.483

[55] Alpaydin, E. (2014). *Introduction to machine learning*. MIT press.

[56] Bozkurt, F., Köse, C., & Sarı, A. (2018). An inverse approach for automatic segmentation of carotid and vertebral arteries in CTA. *Expert Systems with Applications*, 93, 358-375. https://doi.org/10.1016/j.eswa.2017.10.041

[57] Sheela, L. J. (2016). A Review of Sentiment Analysis in Twitter Data Using Hadoop. *International Journal of Database Theory and Application, 9*(1), 77-86. https://doi.org/10.14257/ijdta.2016.9.1.07

Contact information:

Ferhat BOZKURT, PhD, Assistant Professor
(Corresponding author)
Department of Computer Engineering,
Faculty of Engineering, Ataturk University, Erzurum, 25240, Turkey
Phone: +904422316057
E-mail: fbozkurt@atauni.edu.tr

Önder ÇOBAN, PhD candidate
Department of Computer Engineering,
Faculty of Engineering, Adıyaman University, Adıyaman, 02040, Turkey
E-mail: ocoban@adiyaman.edu.tr

Faruk Baturalp GÜNAY, PhD candidate
Department of Computer Engineering,
Faculty of Engineering, Ataturk University, Erzurum, 25240, Turkey
E-mail: baturalp@atauni.edu.tr

Şeyma YÜCEL ALTAY, PhD candidate
Department of Computer Engineering,
Faculty of Engineering, Ataturk University Erzurum, 25240, Turkey
E-mail: seyma.yucel@atauni.edu.tr