e-Informatica Software Engineering Journal, Volume 15, Issue 1, 2021, pages: 163–184, DOI 10.37190/e-Inf210108

Multi-view learning for software defect prediction

Elife Ozturk Kiyak*, Derya Birant**, Kokten Ulas Birant**
*Graduate School of Natural and Applied Sciences, Dokuz Eylul University, Izmir, Turkey
**Department of Computer Engineering, Dokuz Eylul University, Izmir, Turkey
elife.ozturk@deu.edu.tr, derya.birant@deu.edu.tr, ulas.birant@deu.edu.tr

Abstract

Background: Traditionally, machine learning algorithms have been simply applied for software defect prediction by considering single-view data, meaning the input data contains a single feature vector. Nevertheless, different software engineering data sources may include multiple and partially independent information, which makes the standard single-view approaches ineffective.
Objective: In order to overcome the single-view limitation in the current studies, this article proposes the usage of a multi-view learning method for software defect classification problems.
Method: The Multi-View k-Nearest Neighbors (MVKNN) method was used in the software engineering field. In this method, first, base classifiers are constructed to learn from each view, and then classifiers are combined to create a robust multi-view model.
Results: In the experimental studies, our algorithm (MVKNN) is compared with the standard k-nearest neighbors (KNN) algorithm on 50 datasets obtained from different software bug repositories. The experimental results demonstrate that the MVKNN method outperformed KNN on most of the datasets in terms of accuracy. The average accuracy values of MVKNN are 86.59%, 88.09%, and 83.10% for the NASA MDP, Softlab, and OSSP datasets, respectively.
Conclusion: The results show that using multiple views (MVKNN) can usually improve classification accuracy compared to a single-view strategy (KNN) for software defect prediction.

Keywords: Software defect prediction, multi-view learning, machine learning, k-nearest neighbors

1. Introduction

Predicting defects (bugs) in source codes is one of the most valuable processes of the software development life cycle (SDLC) to achieve high quality and reliability in software. The defects that exist in the software lead to not only waste time and money but also severe consequences during deployment. Therefore, detecting and fixing defects in the initial stages of SDLC are crucial requirements to produce robust and effective software systems. For this purpose, so far, many studies on software defect prediction have been conducted by utilizing machine learning techniques. For instance, software modules have been classified as buggy or non-buggy, which refers to the binary classification, or the number of bugs predicted, which refers to the regression problem [1].

Software defect prediction (SDP) generally includes the following stages to recognize defect-prone software components. (i) First, software modules are obtained from software repositories. (ii) After that, the features (software metrics) are extracted from the software modules and each module is labeled to indicate whether the module contains a defect or not. (iii) A classification model is constructed on the training labeled data. (iv) Finally, the constructed model is utilized to estimate the defect-proneness of new unseen software modules.

The standard SDP studies have used the traditional machine learning techniques which are basically working on single-view data. However, the SDP problems can involve data with multiple views (i.e., multiple feature vectors). The conventional classification algorithms (i.e., k-nearest

Submitted: 22 March 2021; Revised: 10 August 2021; Accepted: 10 August 2021; Available online: 3 November 2021

neighbors – KNN) simply concatenate all multiple views into a single view for learning. However, this simple view-concatenation approach may produce undesirable prediction results since each view has its own specific characteristics. Therefore, multi-view learning (MVL) methods are needed to individually explore diverse information from several different feature vectors and, hence, to increase learning performance by taking into account the diversity of various views. For this purpose, in this work, we propose the usage of a multi-view learning approach for software defect prediction problems.

The fundamental contributions of this paper can be pointed out as follows. (i) This study uses the *Multi-View k-Nearest Neighbors (MVKNN)* algorithm in the software engineering area. (ii) It compares the MVKNN and KNN algorithms for software defect prediction. (iii) This study is also original in that it investigates the effects of the number of neighbors (the parameter k) on the software defect prediction performance.

In the experimental studies, we demonstrated the effectiveness of the multi-view learning approach on the SDP. Our MVKNN method was tested on 50 datasets obtained from different software bug repositories. The experiment results show that our MVKNN algorithm achieved higher accuracy values than the KNN algorithm on most of the datasets.

The remainder of this article is basically organized in the following manner. In Section 2, we give an overview of the previous studies related to the topic. In Section 3, we explain the methods used in this study. In Section 4, we describe the main characteristics of the datasets, present the experimental studies, and discuss the results. The last part (Section 5) provides concluding remarks and intended future works.

2. Related work

In *single-view learning*, a classification method is applied to the entire feature set or the specific part of the feature set [2]. In *multi-view learning*, various distinct feature sets (views) are evaluated [3]. These views can be collected from diverse sources, or a single raw data can be separated into different feature sets. For example, in web mining, the textual content can be considered as one view, and image data can be represented as another view. Similarly, a single document can have multiple representations (views) in different languages. Another typical example is the music emotion recognition, in which lyrics (view 1) and song (view 2) can be considered as multiple views. The views can be multiple measurement modalities such as jointly represented audio signals + video signals in television broadcast, biological + chemical data in drug discovery, or images from different angles.

Recently, multi-view learning (MVL) has been combined with different machine learning paradigms such as active learning [4], ensemble learning [5, 6], multi-task learning [7], and transfer learning [8]. In the literature, many studies on MVL have been focused on classification and/or regression problems [4, 5, 8]; however, recently, some studies have focused on a clustering task [3, 6]. Until now, MVL has been used in different fields such as health [5, 9], finance [6], and transportation [7]. However, MVL studies are considerably limited in software engineering, especially for software defect prediction.

2.1. Related studies on single-view learning for software engineering

Researchers have concentrated on machine learning (ML) methods in software defect prediction. However, they have applied ML techniques on the whole feature set or the selected feature subset. In the literature, a massive number of existing SDP studies have built classification models on specific features which have been determined by using a feature selection method. For example, Laradji et al. [10] used an ensemble classifier and claimed that the reduction of unimportant and irrelevant features improves the performance of defect prediction. They applied a greedy-forward selection method to obtain a subset of software metrics. Agarwal and Tomar [11] applied the F-score feature selection method which was utilized to determine the important software metric set that was distinctly affecting the defect classifi-

cation in software projects. The study conducted by Wang et al. [12] used the filter-based feature ranking techniques to determine how many metrics should be selected when constructing a defect prediction model. First, the top 10, 15, and 20 metrics were chosen according to their scores, and then three different classification models were constructed. Although different feature sets achieved high accuracy for different classifiers, their results showed that a predictor could be constructed with only three features.

In the literature, ML methods have been usually used to perform *within-project defect prediction (WPDP)*. However, when historical data was insufficient within a project, the *cross-project defect prediction (CPDP)* approach [13] has been applied to employ the information from other projects. Moreover, some studies used advanced ML paradigms in software defect prediction, such as ensemble learning [14], semi-supervised learning [15], and transfer learning [13].

All the aforementioned studies performed experiments on a single feature vector (view). However, in our study, we used several feature sets (views) to learn diverse information obtained from various data sources, and therefore to increase learning performance by taking into account the diversity of different views.

2.2. Related studies on multi-view learning for software engineering

Recently, lots of information about a subject have been acquired easily, as well as various kinds of data (i.e., image, audio, text, video) have been obtained from multiple sources. As many MVL studies [3] indicated that the information acquired by using data gathered from multiple sources can be more valuable than the information obtained from single-view data. Thus, multi-view learning has been used in a variety of areas for different purposes, such as text classification [16], image classification [17], face identity recognition [18], and speech recognition [19].

Multi-view learning in software engineering is concerned with the problem of learning from data that describe a particular SE problem and is represented by multiple distinct feature sets (called views). Many SE problems can be expressed with different data views. One of the most well-known examples is that software engineering data can consist of several views such as the module dependency graph (view 1), execution logs (view2), and the vocabulary (view 3) used in the source code. Since all of them collectively describe a software system, the integration of all these unique and complementary views can be jointly used for analysis [20]. A software system can also be investigated from different perspectives, relating to a variable group or representation scheme depicting that domain. For instance, evolutionary information is a set of group variables describing the co-changing frequency of software units. Another example in the software engineering area is the identification of programming language from different views. For example, source code classification can be performed by applying multi-view learning to the same piece of code data obtained as text (view1) and image (view2) [21]. In the software defect prediction area, as in this study, the software metrics extracted from source codes can be divided into different views, considering that they are obtained from different perspectives.

In the literature, only a limited number of studies have focused on multi-view learning for software defect classification. In 2017, Phan and Nguyen [22] applied a multi-view convolutional neural network to predict software defects from assembly instruction sequences. They represented each instruction with two views: the content (view 1) and the group (view 2). For each view, convolutional layers were applied and merged before feeding them to the fully-connected layers. In 2019, Chen et al. [23] proposed a multi-view NN-based heterogeneous transfer learning model built by partitioning features into groups for software defect prediction. However, in the former study, source code was used as a dataset, and in the latter study, different feature sets were evaluated. Our approach is leveraged by combining KNN algorithms separately implemented to each view.

2.3. Related studies on KNN for software defect prediction

By means of its simplicity and ease of implementation, the KNN algorithm has been implemented in many ML applications. For defect prediction, several studies have been developed using the KNN algorithm in various ways, such as weighted KNN [24], hybrid model using Naive Bayes and KNN [25], boosting-based KNN [26], and KNN regression [27]. Although these studies are related to defect prediction, experiments were conducted using single view data. Different from these previous studies, we used an improved version of KNN for multi-view software defect data.

Until now, the KNN algorithm has been employed in various multiple learning studies in different areas. For example, multi-label learning (ML-KNN) [28], multi-instance learning (FuzzyKNN) [29], multi-class learning (DEMST--KNN) [30], and multi-task learning (ssMTTL--KNN) [31]. Unlike these previous studies, we used the multi-view KNN method proposed in [32].

3. Material and methods

In this section, the KNN and MVKNN algorithms are briefly described.

3.1. *K*-nearest neighbors

K-nearest neighbors (KNN) is a typical supervised ML algorithm utilized in both classification and regression problems. The KNN algorithm works on labeled data samples and uses them to classify a new sample based on its similarity to the k closest neighbors. In this study, we chose KNN as a machine learning algorithm since it has many advantages such as simplicity, easy implementation, efficiency, and ease of understanding the results [33]. The advantages of KNN also include that it supports incremental data; therefore, re-training is not required for the newly-arrived observations. The other advantage of KNN is that it can be successfully used for nonlinear data. Moreover, it has the ability to predict both categorical and discrete variables and to deal with noisy data. Furthermore, it can be directly used for multi-class classification, in addition to binary classification. KNN has been proven to be an effective method in various studies [24–31] and thus, it has been widely used in many applications.

Considering k as the number of nearest neighbors and n labeled data samples, $D = \{s_1, s_2, \ldots, s_n\}$ be the training set in the form of $s_i = (x_i, c_i)$, where x_i is the feature vector of the sample with d-dimension, denoted by $x_i = (x_{i1}, x_{i2}, \ldots, x_{id})$ and comes from data space X, and c_i is the class label that s_i assigned to and it is from a set of classes $Y = \{c_1, c_2, \ldots, c_m\}$. A classifier is a function in the form of $f : X \to Y$ that maps a new data sample X onto an item of Y. The KNN algorithm starts with a new sample $s' = (x', ?)$ whose class label is unknown. Distance $\mathrm{dis}(x', x_i)$ is calculated between s' and all s_i in the dataset D. The most widely-used distance measures are Euclidean and Manhattan distance metrics. Then, the k closest neighbors to s' in the training samples are selected. Finally, class label c' is assigned to s' based on the majority class of neighbors.

3.2. Multi-view *k*-nearest neighbors

Our algorithm, called *multi-view k-nearest neighbors* (MVKNN), was proposed in [32]. It is an advanced version of KNN that combines individual models developed for each view.

Let $S^v = (X^v, c)$ be a sample of a view v, where X^v is the feature set of v such that $X^v = \{x_1^v, x_2^v, \ldots, x_d^v\}$ for $v = 1, 2, \ldots, V$ and c refers to the class label of the sample, where V is the number of views. The dataset D consists of n samples and denoted by $D = \{(X_i^1, X_i^2, \ldots, X_i^V, c_i), i = 1, 2, \ldots, n\}$, where X_i is the i^{th} sample and $c_i \,\epsilon\, Y$ is the class label of it and X_i^j refers to the i^{th} instance of the j^{th} view. Views are mutually exclusive, so they have different feature sets such that $\forall_{(p,q)} X^p \cap X^q = \emptyset$. Since views express different representations of the same object, each instance in different views has the same class label $c_i \,\epsilon\, Y$, where $Y = \{c_1, c_2, \ldots, c_n\}$. Firstly, in the *view-based classification*, a set of classifiers

are built for each view by using different k input parameters such that $f_j : X^j \rightarrow Y$. After that, in the *multi-view-based classification*, a number of view-based classifiers f_j are combined to determine the final prediction.

Figure 1 shows the general overview of the MVKNN method that can be used for software defect classification. The MVKNN method consists of three main steps: data preparation, view-based classification, and multi-view-based classification.

Step 1 – *Data preparation*: Raw data obtained from a software repository may require preparation before yielding results, so it has been prepared for view-based classification. To increase data quality and to acquire more accurate outcomes, various preprocessing techniques may be applied such as completing missing data or removing duplicate instances. Since the dataset consists of several views, data preprocessing should be performed for all views. It can be noted here that each view has a different feature set. In other words, each view is represented as disjoint features of the same object, so distinct features and class label of an object exist in each view.

Step 2 – *View-based classification*: This classification aims at creating a model learning from each view using an adaptive method in which each instance is classified with a different number of neighbors, rather than utilizing only a single value of k. For each view, weak KNN classifiers

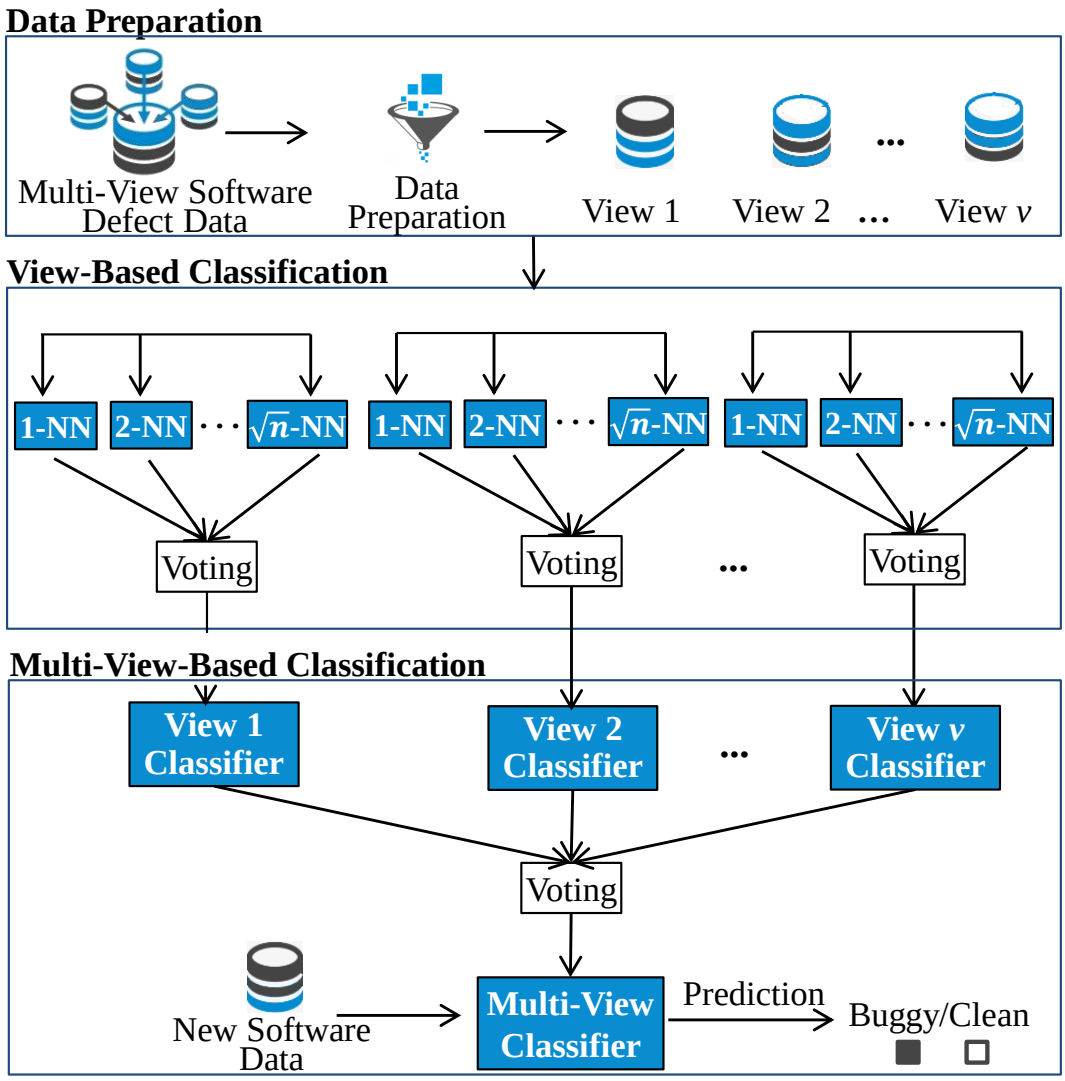

Figure 1: The general overview of the MVKNN method for software defect prediction

are constructed with different parameters (k), where k is ranged from 1 to the square root of the number of samples ($\sqrt{n}$). The classifiers (1-NN, 2-NN, 3-NN, ..., $\sqrt{n}$-NN) are combined to form a strong ensemble model for a specific view. In the end, the class label of a particular view is determined by using a voting strategy that selects the class having the highest number of votes, where n refers to the number of samples in the dataset.

Step 3 – *Multi-view-based classification*: In this step, a number of view-based learners are combined to form a general model for classification. A majority voting strategy is utilized to combine outcomes from each view and the final class label of a new instance is specified as *buggy* or *clean*.

The MVKNN method can be used for software detect prediction to provide many advantages as follows:

- A single-view software data is dependent on a single view-point, whereas multi-view software data usually contains complementary information since it typically includes many view-points. In multi-view learning, the lack of information of one view can be complemented by the sufficiency of other views. Thanks to this essential property of MVKNN, it eliminates the weaknesses of single-view software defect prediction.
- Compared to the traditional KNN algorithm which is substantially developed for single-view data, MVKNN is expected to yield more robust outcomes in the presence of noisy software defect data. Because noise in one view can be reduced by a voting mechanism among multiple views.
- Numerous software metrics can be extracted from software projects and so data can be high dimensional. The MVKNN method has the ability to handle a large number of features since it considers high-dimensional data as a union of multiple low-dimensional subspaces, called views. Software defect data can contain many metrics from different perspectives such as Halstead's measures and McCabe's measures. Therefore, considering high dimensional data containing a group of different feature sets, we can separate the data into appropriate views, each corresponding to a disjoint feature set.

In spite of numerous benefits, the MKKNN method considers correlations at the view level; however, it does not take into account implicit correlations between features in multiple views. In addition, MVKNN is computationally more expensive than KNN since it separately learns from each view dataset and runs the base learner many times to jointly learn from multiple k parameters rather than a single k parameter.

4. Experimental studies

In this section, software defect datasets are described, and several experiments conducted with the MVKNN method are presented. The MVKNN and KNN methods were compared for software defect prediction. The obtained results were validated utilizing statistical tests to ensure that the differences between the methods in the datasets were significant. For this purpose, we applied the Wilcoxon test which is a well-known non-parametric statistical test.

The MVKNN algorithm was implemented utilizing the WEKA machine learning library [34] and *C#* programming language. The implementation of the MVKNN method is available at the website https://github.com/elifeozturk/MVKNN. As an evaluation method, the 10-fold cross-validation technique was used, in which the dataset is divided into 10 parts and then each part is used once as a test set while the remaining parts form the training set. In this study, four evaluation metrics (Accuracy, Precision, Recall, and F1 Score) were used to evaluate the classification performances. *Accuracy* is the most widely-used performance measure that calculates the ratio between the number of correctly predicted instances and all instances. It is calculated as follows: $Accuracy = (TN + TP)/(TP + FP + TN + FN)$, where TN (true negatives) and TP (true positives) are correctly predicted as real labels. In other words, if the real label is "positive" and the predicted label is "positive" or the actual label is "nega-

tive" and the predicted label is "negative", it is TP or TN, respectively. Unlike correct estimates, FP and FN point out that instances did not correctly classified as actual labels. *Precision* shows the ratio of correctly predicted positive instances to the total predicted positive instances. Precision is calculated as follows: $Precision = TP/(TP + FP)$. *Recall* is the number of correct predictions divided by the number of all predictions in the actual class. It is calculated as follows: $Recall = TP/(TP + FN)$. *F1 Score* is the harmonic mean of the precision and recall. This evaluation measure is particularly preferred when datasets have uneven class distribution. F1 Score is calculated as follows: $F1\ Score = 2 \times (Recall \times Precision)/(Recall + Precision)$. Precision, recall, and *F1 Score* are useful metrics for evaluating "learning from imbalanced datasets".

4.1. Dataset description

In this work, we conducted experiments on 50 bug datasets from three different repositories available in the software engineering area: Tera-PROMISE Open Source Software Projects (OSSP), NASA MDP (Metrics Data Program), and Softlab that included 40, 5, and 5 datasets, respectively. Table A1 lists the main characteristics of the datasets, including their names, the groups to which the datasets belong, the number of samples in the datasets, and defect percentages (%). To be able to test MVKNN on imbalanced data, we especially used the NASA MDP datasets where defect percentages ranged between 7% and 23%. Furthermore, these datasets have been widely used in many machine learning studies [10, 14, 15]. MVKNN is designed for general purposes; therefore, it can be further applied to different datasets with different software engineering metrics when the research community has presented new ones.

More details about datasets are described below, and supplemental information and tables are included in Appendix A.

– *OSSP Datasets* [35]: The datasets in this group consists of 20 independent object-oriented source code metrics and one dependent defect variable that indicates buggy or not.
– *NASA MDP Datasets* [36]: The datasets (named cm1, jm1, kc1, kc2, pc1) in this group were obtained from NASA software projects. These datasets include 21 static code features that were extracted from a software product based on the McCabe metric, Basic Halstead measures, and Derived Halstead measures.
– *Softlab Datasets* [37]: The datasets were denoted by a Software Research Laboratory and collected from a Turkish white-goods manufacturer. The ar1 and ar6 datasets were collected from an embedded controller for white goods, while the other (ar5, ar4, ar3) datasets were obtained from a refrigerator, dishwasher, and washing machine, respectively. The datasets contain 29 static code attributes.

Table A2 presents the fundamental characteristics of datasets, containing the number of classes (i.e., buggy or clean), the number of views, and the number of features that belong to each view.

Tables A3, A4 and A5 show all the software metrics and their categories in each dataset group for NASA, OSSP and SOFTLAB datasets, respectively. The datasets have different views designed based on the previous studies [38–40]. The NASA MDP datasets have three views as given in [38]: McCabe, Basic Halstead, and Derived Halstead features with 4, 9, and 8 software metrics, respectively. The OSSP datasets contain object-oriented measures that indicate the characteristics of inheritance, coupling, cohesion, complexity, and line features of software programs [39]. The Softlab datasets consist of four views [40], including Halstead, McCabe, LOC, and Miscellaneous metrics. Halstead metrics show the program complexity obtained by analyzing the source code. McCabe metrics quantify the control flows in a program. LOC metrics refer to the measures related to lines of code, such as the number of executable lines and the number of comment lines. Finally, the rest software metrics are additionally included in the "Miscellaneous" group.

4.2. Experimental results

Table 1 shows the comparison of the KNN and MVKNN algorithms on the NASA MDP and Softlab datasets in terms of accuracy. According to the results, our MVKNN algorithm achieved 86.59% and 88.09% accuracy values on average for the NASA MDP and Softlab datasets, respectively. However, KNN reached only 86.46% and 86.60% accuracy values on average. It is clear that MVKNN outperformed KNN on four datasets (jm1, kc1, kc2, pc1) from the NASA MDP group, while only on one dataset (cm1) both algorithms have the same classification accuracy (90.16%). The results obtained from the Softlab datasets show that our MVKNN algorithm demonstrated better or equal accuracy on all datasets compared to the KNN algorithm.

According to the results, it is possible to say that quality assurance teams can effectively allocate limited resources for validating software products since the constructed defect prediction models provide satisfactory results (>86%) on average when identifying bug-prone software artifacts. This result indicates that the models can correctly predict defect-prone software modules with a rate of 86% before defects are discovered; thus, the predictive models can be used to prioritize software quality assurance efforts. The code areas that potentially contain defects can be predicted to help developers allocate their testing efforts by first checking potentially buggy code. As the size of software projects becomes larger, the constructed defect prediction models play a more critical role to support developers. Furthermore, they speed up time to market as well as with more robust software products.

Figure 2 shows the comparison of KNN and MVKNN on the OSSP datasets in terms of accuracy. The results show that our MVKNN algorithm has equal to or higher accuracy than the KNN algorithm on 34 out of 40 datasets. Therefore, our MVKNN algorithm is better than KNN on 85% of the datasets. In particular, the biggest accuracy differences between the methods were observed on the "berek" and "velocity 1.6" datasets, where our method increased the accuracy by over 6.98% and 6.37%, respectively. For example, our method (86.05%) achieved better performance than the existing method (79.07%) in the "berek" dataset in terms of accuracy. In the "velocity 1.6" dataset, accuracy of our method (72.05%) is higher than the accuracy of the existing method (65.68%). These results show that our method usually gives more accurate outputs for software defect prediction by using a different perspective.

Some datasets were obtained from different versions of a software project, such as Jedit 4.0, Jedit 4.1, Jedit 4.2, and Jedit 4.3. In this case, within-project defect prediction (WPDP) can be used to identify defect-prone modules in a forthcoming version of a software project. However, some datasets were obtained from a single version of a software project, such as the "arc" dataset. In this case, cross-project defect prediction (CPDP) can be applied since the target project may be a new project or does not have enough labeled modules.

In addition to accuracy, we evaluated the performances of the methods using the precision, recall, and *F1 Score* metrics. As can be seen in Table B1, when considering the NASA MDP datasets, MVKNN achieved equal or higher accuracy than KNN. In the Softlab datasets, it is clearly seen that the MVKNN model has better results than the KNN model on average. In addition, Table B2 shows the precision, recall, and *F1 Score* results for the OSSP datasets. According to the results, the MVKNN algorithm achieved the values of 0.79, 0.83, and 0.81 for the precision, recall, and *F1 Score* metrics, respectively; whereas, KNN only obtained the values of 0.77, 0.81, and 0.79 on average.

Though MVKNN achieved usually higher accuracy than KNN, all the results (Table 1 and Figure 2) were also validated using statistical tests. We utilized a well-known non-parametric statistical test: Wilcoxon Test, also known as Wilcoxon signed-rank test. Since it is used to analyze matched-pair data, it can be considered a rank-based alternative to the two-sample t-test [41]. Wilcoxon test does not rely on the assumption of data complying with any distribution. It considers the sign and magnitude of the distribution of cumulative observations. In

Table 1: Comparison of the KNN and MVKNN algorithms on the NASA MDP and Softlab datasets

NASA MDP			Softlab		
Dataset	KNN	MVKNN	Dataset	KNN	MVKNN
cm1	**90.16**	**90.16**	ar1	**92.56**	**92.56**
jm1	80.98	**81.08**	ar3	90.08	**90.48**
kc1	84.97	**85.30**	ar4	83.88	**85.98**
kc2	83.14	**83.33**	ar5	79.86	**83.33**
pc1	93.03	**93.06**	ar6	**85.15**	**85.15**
Avg.	86.46	**86.59**	**Avg.**	86.60	**88.09**

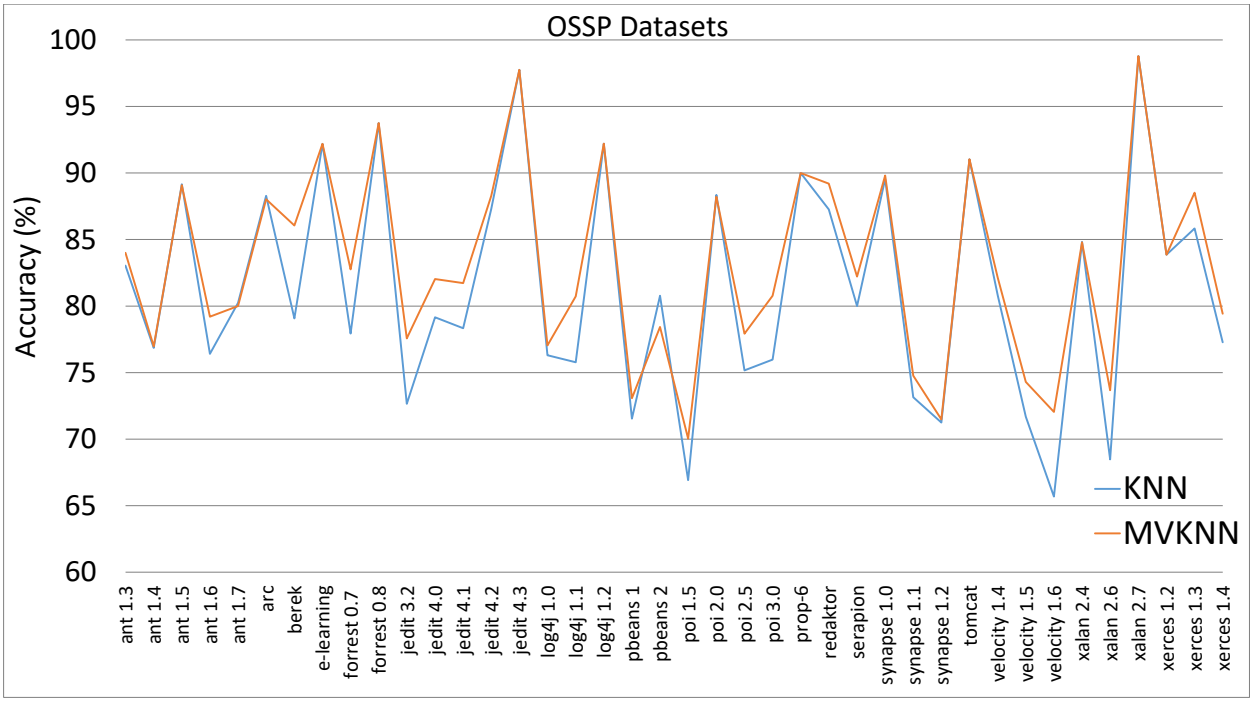

Figure 2: Comparison of the KNN and MVKNN algorithms on the OSSP datasets

this statistical test, the null hypothesis (H0) indicates that there is no difference or relationship between methods. The alternative hypothesis (H1) states that there is a significant difference or relationship between methods. A significance level (α) is usually specified as 0.05. The p-value is used to determine the presence of statistical significance since it shows the level of evidence of the difference. If the obtained p-value is lower than the threshold level ($p < 0.05$), the null hypothesis (H0) is rejected, which means that the difference is significant. Since the p-value obtained from the Wilcoxon test is 0.0000027 and it is smaller than the significance level, H0 is rejected. Therefore, it implies that the obtained results are statistically significant. As can be seen in Figure 3, MVKNN showed the median accuracy of 83.6%, while KNN had the median accuracy of 82.0%.

In order to show performance comparisons between KNN and MVKNN, supplemental tables and figures are presented in Appendix B. Figure B1 displays individual view-based performances. When each view in the datasets is examined separately, it can be seen that MVKNN has good performance on the view-based classification, but it does not always better than KNN. However, the multi-view-based classification results of MVKNN are either greater than or equal to the KNN accuracy values for all datasets.

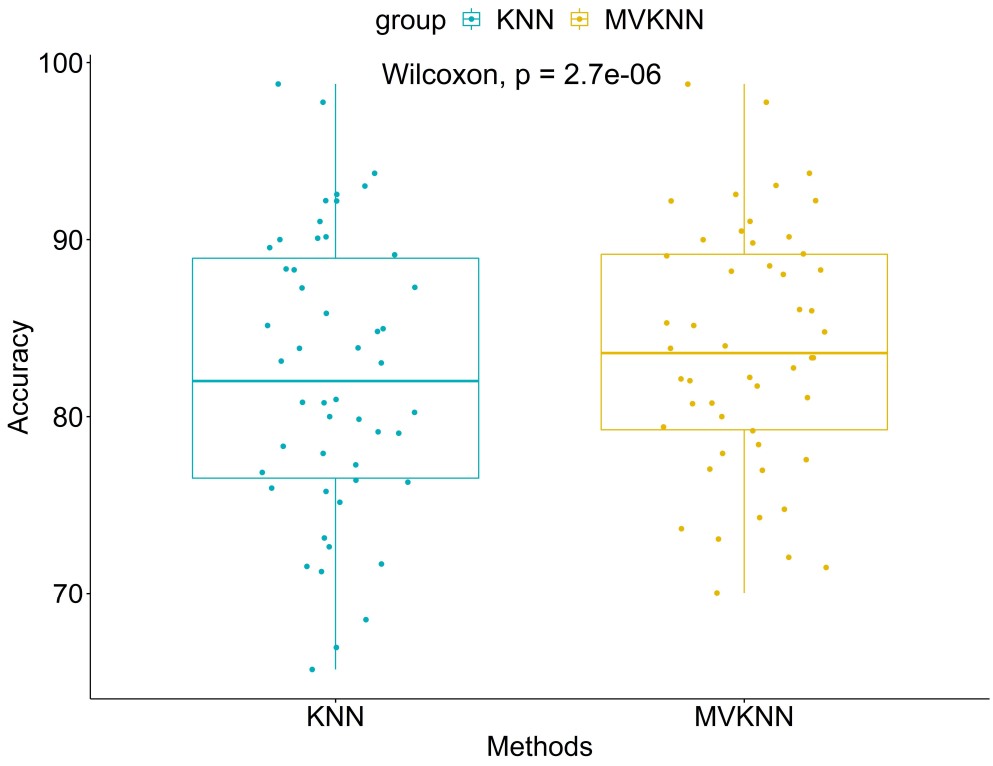

Figure 3: The spread of the accuracy scores for each algorithm

This is because of the complementary power of MVKNN.

In the traditional KNN algorithm, a single and fixed k value is used for classification. However, in a real dataset, data points may be distributed irregularly, or some of them can be noisy. To consider the density variations in the data, different k parameters can be used to benefit from both more neighbors in dense regions and less in sparse regions. Liu et al. pointed out that a fixed k value is not suitable for many test samples in a given training set [42]. For this reason, in our study, rather than just using a single and fixed value of k, the algorithm is run with various k values. MVKNN learns from k values, starting from 1 to $\sqrt{n}$, where n is the number of instances. The maximum k value was selected as $\sqrt{n}$ on the basis of many studies [43–45]. Choosing the maximum value of k as $\sqrt{n}$ is an appropriate decision since the probability of overfitting significantly increases if k is selected as too small or too large. Park and Lee [44] reported that setting k as the square root of the data size is a good empirical ground rule. Lall and Sharma [43] also proved this statement theoretically using a generalized cross-validation (GCV) score function. Mitra et al. [45] also stated that $k = \sqrt{n}$ is usually suitable for test samples. If the k value is small, then the results can be susceptible to noisy instances; otherwise, if the k value is large, the neighborhood may cover many instances from other classes. Therefore, the square root is a reasonable choice for the searching neighborhood. Figure B2 displays accuracy values obtained for different k values. It compares KNN and MVKNN performance on the k values starting from 1 to the square root of the number of samples for each dataset. For all the datasets, MVKNN starts with greater accuracy than KNN, and at the maximum k value, it is either greater than or equal to the KNN accuracy.

Table B3 separately shows the performances of the KNN and MVKNN algorithms for each view in the OSSP datasets. It is clearly observed that MVKNN has equal to or higher accuracy than KNN for 34 out of 40 datasets. It can be seen that our MVKNN method (83.10%) outperformed the traditional KNN method (81.37%)

in classification on average. According to the results given in Table B3, the proposed method (MVKNN) is more successful than the traditional KNN method in terms of accuracy. Thus, multi-view learning can achieve more accurate results than single-view learning in defect prediction since it benefits from different perspectives of data.

4.3. Validity

This section discusses the validity of the research, the threats, and countermeasures of the context under common guidelines given in [46].

i. **Construct validity**
Threats to construct validity are concerned with establishing correct measures for the concepts addressed in empirical analysis. The selection of performance measures is the basic limitation. In our study, the most commonly used performance measure (accuracy) was selected to overcome the threat of measure selection. In other words, the predictive validities of the constructed models were assessed by using the accuracy measure. According to the results, the proposed approach achieved 86.59%, 88.09%, and 83.10% accuracy values on average for the NASA MDP, Softlab, and OSSP datasets, respectively. Thereby, predictive validity was proven, since all average accuracy results are higher than the acceptable level (>80%).

ii. **Internal validity**
Internal validity is related to uncontrolled factors that can cause a difference in experimental measurements. To reduce this threat, we used the k-fold cross-validation technique in which the validation procedure is repeated k times until each of the k data subsets has served as a test set. In addition, we ran all the experiments in the same environment. Another internal threat is related to data collection. We tested our approach on public and most widely-used software engineering datasets in the literature [11, 13–15]. The information on data collection and its validity can be found in [35–37].

iii. **External validity**
External validity concerns the generalizability of a conclusion or experimental finding reached on the sample group under experimental conditions in various environments. In this study, our approach was tested on a total of 50 datasets from three different data repositories to reduce the threat of this kind of validity. In addition, since MVKNN is designed for general-purpose, it can be applied to various domains from transportation to medicine.

iv. **Conclusion validity**
Conclusion validity is concerned with the relationship between the treatment in an experiment and the actual outcome we observed. It is considered as the evaluation of statistical power, significance testing, and effect size. For this purpose, we used the Wilcoxon statistical test to ensure the differences in KNN and MVKNN performances are statistically significant. Since the p-value obtained from the statistical test (0.0000027) is smaller than the significance level (<0.05), it is possible to say that the results are statistically significant.

5. Conclusion and future work

The standard software defect classification studies work on single-view data. They do not utilize different feature sets, called views. However, a software defect prediction problem can involve data with multiple views in which the feature space includes multiple feature vectors. Therefore, in this study, the multi-view k-nearest neighbors (MVKNN) algorithm is used for software defect classification. Here, the software defect metrics are grouped under several views according to their feature extractors. MVKNN consists of two parts. First, base classifiers are constructed to learn from each view. Second, classifiers are combined to create a strong multi-view model.

In this study, several experiments were conducted on 50 bug datasets from different repositories to show the capability of the MVKNN method. It can be concluded from the results that

the MVKNN algorithm usually achieved better performance compared to the KNN algorithm.

As future work, the MVKNN method can be used for other software engineering problems such as software cost estimation, software effort prediction, readability analysis, refactoring, software clone detection, vulnerability prediction, and software design pattern mining.

References

[1] R. Ozakinci and A. Tarhan, "Early software defect prediction: A systematic map and review," *The Journal of Systems and Software*, Vol. 144, Oct. 2018, pp. 216–239.

[2] K. Bashir, T. Li, and M. Yahaya, "A novel feature selection method based on maximum likelihood logistic regression for imbalanced learning in software defect prediction," *The International Arab Journal of Information Technology*, Vol. 17, No. 5, Sep. 2020, pp. 721–730.

[3] J. Zhao, X. Xie, X. Xu, and S. Sun, "Multi-view learning overview: Recent progress and new challenges," *Information Fusion*, Vol. 38, No. 1, Nov. 2017, pp. 43–54.

[4] F. Liu, T. Zhang, C. Zheng, Y. Cheng, X. Liu, M. Qi, J. Kong, and J. Wang, "An intelligent multi-view active learning method based on a double-branch network," *Entropy*, Vol. 22, No. 8, Aug. 2020.

[5] Y. Chen, D. Li, X. Zhang, J. Jin, and Y. Shen, "Computer aided diagnosis of thyroid nodules based on the devised small-datasets multi-view ensemble learning," *Medical Image Analysis*, Vol. 67, No. 8, Jan. 2021.

[6] Y. Song, Y. Wang, X. Ye, D. Wang, Y. Yin, and Y. Wang, "Multi-view ensemble learning based on distance-to-model and adaptive clustering for imbalanced credit risk assessment in p2p lending," *Information Sciences*, Vol. 525, Jul. 2020, pp. 182–204.

[7] S. Cheng, F. Lu, P. Peng, and S. Wu, "Multi-task and multi-view learning based on particle swarm optimization for short-term traffic forecasting," *Knowledge-Based Systems*, Vol. 180, Sep. 2019, pp. 116–132.

[8] Y. He, Y. Tian, and D. Liu, "Multi-view transfer learning with privileged learning framework," *Neurocomputing*, Vol. 335, Mar. 2019, pp. 131–142.

[9] J. Li, L. Wu, G. Wen, and Z. Li, "Exclusive feature selection and multi-view learning for alzheimer's disease," *Journal of Visual Communication and Image Representation*, Vol. 64, Oct. 2019.

[10] I.H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Information and Software Technology*, Vol. 58, Feb. 2015, pp. 388–402.

[11] S. Agarwal and D. Tomar, "A feature selection based model for software defect prediction," *International Journal of Advanced Science and Technology*, Vol. 65, 2014, pp. 39–58.

[12] H. Wang, T.M. Khoshgoftaar, and N. Seliya, "How many software metrics should be selected for defect prediction?" in *Proceedings of the Twenty-Fourth International Florida Artificial Intelligence Research Society Conference*, R.C. Murray and P.M. McCarthy, Eds. Palm Beach, Florida, USA: AAAI Press, May 2011.

[13] W. Wen, B. Zhang, X. Gu, and X. Ju, "An empirical study on combining source selection and transfer learning for cross-project defect prediction," in *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*. Hangzhou, China: IEEE, 2019, pp. 29–38.

[14] A. Iqbal, S. Aftab, I. Ullah, M.S. Bashir, and M.A. Saeed, "A feature selection based ensemble classification framework for software defect prediction," *International Journal of Modern Education and Computer Science*, Vol. 11, No. 9, 2019, pp. 54–64.

[15] A. Arshad, S. Riaz, L. Jiao, and A. Murthy, "The empirical study of semi-supervised deep fuzzy c-mean clustering for software fault prediction," *IEEE Access*, Vol. 6, 2018, pp. 47 047–47 061.

[16] M.M. Mirończuk, J. Protasiewicz, and W. Pedrycz, "Empirical evaluation of feature projection algorithms for multi-view text classification," *Expert Systems with Applications*, Vol. 130, 2019, pp. 97–112.

[17] C. Zhang, J. Cheng, and Q. Tian, "Multi-view image classification with visual, semantic and view consistency," *IEEE Transactions on Image Processing*, Vol. 29, 2020, pp. 617–627.

[18] Z. Zhu, P. Luo, X. Wang, and X. Tang, "Multi--view perceptron: a deep model for learning face identity and view representations," in *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014*, Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger, Eds. Montreal, Quebec, Canada: Citeseer, Dec. 2014, pp. 217–225.

[19] S.R. Shahamiri and S.S.B. Salim, "A multi-views multi-learners approach towards dysarthric speech recognition using multi-nets artificial neu-

ral networks," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, Vol. 22, No. 5, Sep. 2014, pp. 1053–1063.

[20] A. Saeidi, J. Hage, R. Khadka, and S. Jansen, "Applications of multi-view learning approaches for software comprehension," *The Art, Science, and Engineering of Programming*, Vol. 3, No. 3, 2019.

[21] E.O. Kiyak, A.B. Cengiz, K.U. Birant, and D. Birant, "Comparison of image-based and text-based source code classification using deep learning," *SN Computer Science*, Vol. 1, No. 5, 2020, pp. 1–13.

[22] A.V. Phan and M.L. Nguyen, "Convolutional neural networks on assembly code for predicting software defects," in *2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)*. Hanoi, Vietnam: IEEE, Nov. 2017, pp. 37–42.

[23] J. Chen, Y. Yang, K. Hu, Q. Xuan, Y. Liu, and C. Yang, "Multiview transfer learning for software defect prediction," *IEEE Access*, Vol. 7, Jan. 2019, pp. 8901–8916.

[24] D. Ulumi and D. Siahaan, "Weighted k-NN using grey relational analysis for cross-project defect prediction," *Journal of Physics: Conference Series*, Vol. 1230, Jul. 2019, p. 012062.

[25] R. Sathyaraj and S. Prabu, "A hybrid approach to improve the quality of software fault prediction using naïve bayes and k-nn classification algorithm with ensemble method," *International Journal of Intelligent Systems Technologies and Applications*, Vol. 17, No. 4, Oct. 2018, pp. 483–496.

[26] L. He, Q.B. Song, and J.Y. SHEN, "Boosting-based k-NN learning for software defect prediction," *Pattern Recognition and Artificial Intelligence*, Vol. 25, No. 5, 2012, pp. 792–802.

[27] R. Goyal, P. Chandra, and Y. Singh, "Suitability of k-NN regression in the development of interaction based software fault prediction models," *IERI Procedia*, Vol. 6, No. 1, 2014, pp. 15–21.

[28] S.K. Srivastava and S.K. Singh, "Multi-label classification of twitter data using modified ML-kNN," in *Advances in Data and Information Sciences*, Lecture Notes in Networks and Systems, K. M., T. M., T. S., and S. V., Eds., Vol. 39. Singapore: Springer, Jun. 2019, pp. 31–41.

[29] P. Villar, R. Montes, A.M. Sánchez, and F. Herrera, "Fuzzy-citation-k-NN: A fuzzy nearest neighbor approach for multi-instance classification," in *2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. Vancouver, BC, Canada: IEEE, Jul. 2016, pp. 946–952.

[30] Y. Xia, Y. Peng, X. Zhang, and H.Y. Bae, "DEMST-KNN: A novel classification framework to solve imbalanced multi-class problem," in *Artificial Intelligence Trends in Intelligent Systems*, Advances in Intelligent Systems and Computing, R. Silhavy, R. Senkerik, Z.K. Oplatková, Z. Prokopova, and P. Silhavy, Eds., Vol. 573. Cham, Germany: Springer, Apr. 2017, pp. 291–301.

[31] S. Gupta, S. Rana, B. Saha, D. Phung, and S. Venkatesh, "A new transfer learning framework with application to model-agnostic multi-task learning," *Knowledge and Information Systems*, Vol. 49, No. 3, Feb. 2016, pp. 933–973.

[32] E.O. Kiyak, D. Birant, and K.U. Birant, "An improved version of multi-view k-nearest neighbors (MVKNN) for multiple view learning," *Turkish Journal of Electrical Engineering and Computer Sciences*, Vol. 29, No. 3, 2021, pp. 1401–1428.

[33] S. Li, E.J. Harner, and D.A. Adjeroh, "Random KNN feature selection – A fast and stable alternative to random forests," *BMC bioinformatics*, Vol. 12, No. 1, 2011, pp. 1–11.

[34] I.H. Witten, E. Frank, M.A. Hall, and C.J. Pal, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th ed., The Morgan Kaufmann Series in Data Management Systems. Cambridge, MA, USA: Elsevier Science, 2016.

[35] "Tera-promise data," accessed: 10.05.2020. [Online]. https://github.com/klainfo/DefectData/tree/master/inst/extdata/terapromise

[36] "NASA MDP data," accessed: 07.05.2020. [Online]. https://github.com/klainfo/NASADefectDataset/tree/master/OriginalData/MDP

[37] B. Turhan, T. Menzies, A.B. Bener, and J.D. Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, Vol. 14, No. 5, Jan. 2009, pp. 540–578.

[38] E. Borandag, A. Ozcift, D. Kilinc, and F. Yucalar, "Majority vote feature selection algorithm in software fault prediction," *Computer Science and Information Systems*, Vol. 16, No. 2, 2019, pp. 515–539.

[39] Z. Yao, J. Song, Y. Liu, T. Zhang, and J. Wang, "Research on cross-version software defect prediction based on evolutionary information," *IOP Conference Series: Materials Science and Engineering*, Vol. 563, Aug. 2019, p. 052092.

[40] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE transactions on software engineering*, Vol. 33, No. 1, Dec. 2006, pp. 2–13.

[41] R.F. Woolson, *Wilcoxon Signed-Rank Test*. Wiley Encyclopedia of Clinical Trials, 2008, pp. 1–3.
[42] H. Liu, S. Zhang, J. Zhao, X. Zhao, and Y. Mo, "A new classification algorithm using mutual nearest neighbors," in *2010 Ninth International Conference on Grid and Cloud Computing*. Nanjing, China: IEEE, Nov. 2010, pp. 52–57.
[43] U. Lall and A. Sharma, "A nearest neighbor bootstrap for resampling hydrologic time series," *Water Resources Research*, Vol. 32, No. 3, Mar. 1996, pp. 679–693.
[44] J. Park and D.H. Lee, "Parallelly running k-nearest neighbor classification over semantically secure encrypted data in outsourced environments," *IEEE Access*, Vol. 8, 2020, pp. 64 617–64 633.
[45] P. Mitra, C. Murthy, and S. Pal, "Unsupervised feature selection using feature similarity," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 24, No. 3, 2002, pp. 301–312.
[46] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, Vol. 14, No. 2, 2009, pp. 131–164.

A. Description of datasets

Table A1: The main characteristics of the datasets

Group	Dataset	Release	Number of Instances	Defect (%)
Open Source Software Projects (OSSP)	ant	1.3	125	16.00
		1.4	178	22.47
		1.5	293	10.92
		1.6	351	26.21
		1.7	745	22.28
	arc	–	234	11.54
	berek	–	43	37.21
	e-learning	–	64	7.81
	forrest	0.7	29	17.24
		0.8	32	6.25
	jedit	3.2	272	3.31
		4.0	306	24.51
		4.1	312	25.32
		4.2	367	13.08
		4.3	492	2.24
	log4j	1.0	135	25.19
		1.1	109	33.95
		1.2	205	92.20
	pbeans	1.0	26	76.92
		2.0	51	19.61
	poi	1.5	237	59.49
		2.0	314	11.78
		2.5	385	64.41
		3.0	442	63.57
	prop	6	661	9.98
	redactor	–	176	15.34
	serapion	–	45	20.00
	synapse	1.0	157	10.19
		1.1	222	27.03
		1.2	256	33.59
	tomcat	–	858	8.97

Table A1 continued

Group	Dataset	Release	Number of Instances	Defect (%)
OSSP	velocity	1.4	196	75.00
		1.5	214	66.36
		1.6	229	34.06
	xalan	2.4	723	15.21
		2.6	885	46.44
		2.7	909	98.79
	xerces	1.2	440	16.14
		1.3	453	15.23
		1.4	588	74.32
Softlab	ar1	–	121	7.44
	ar3	–	63	12.70
	ar4	–	107	18.69
	ar5	–	36	22.22
	ar6	–	101	14.85
NASA MDP	cm1	–	498	9.83
	jm1	–	10885	19.00
	kc1	–	2109	15.45
	kc2	–	522	20.49
	pc1	–	1109	6.94

Table A2: The number of classes, views, and features of each dataset group

Dataset Group	#Classes	#Views	#Features				
NASA MDP	2	3	McCabe	Basic Halstead		Derived Halstead	
			4	9		8	
OSSP	2	5	Coupling	Complexity	Cohesion	Inheritance	Scale
			5	3	3	3	6
SOFTLAB	2	4	Halstead	McCabe	LOC	Miscellaneous	
			12	3	5	9	

Table A3: Categories of software metrics in the NASA datasets

NASA MDP		
View	Symbol	Metric Full Name
MCCABE	v(g)	Cyclomatic complexity
	ev(g)	Essential complexity
	Iv(g)	Design complexity
DERIVED HALSTEAD	N	Total operators+ operands
	V	Volume
	L	Program length
	D	Difficulty
	I	Intelligence
	E	Effort to write code

Table A3 continued

NASA MDP		
View	Symbol	Metric Full Name
	B	Effort estimate
	T	Time estimator
BASIC HALSTEAD	IOCode	Line count
	IOComment	Comment count
	IOBlank	Blank line count
	IOCodeAnd Comment	# of code and comment lines
	Uniq_Op	# of unique operators
	UniqOpnd	#of unique operands
	Total_Op	#of total operators
	Total_Opnd	# of total operands
	branch_count	# of branch counts

Table A4: Categories of software metrics in the OSSP datasets

OSSP		
View	Symbol	Metric Full Name
COUPLING	ca	Afferent couplings
	cbm	Coupling between methods
	cbo	Coupling between object classes
	ce	Efferent couplings
	ic	Inheritance coupling
COHESION	lcom	Lack of cohesion in methods
	lcom3	Lack of cohesion in methods
	cam	Cohesion among methods of class
COMPLEXITY	amc	Average method complexity
	avg_cc	Average McCabe
	max_cc	Maximum McCabe
INHERITANCE	dit	Depth of inheritance
	moa	Measure of aggregation
	mfa	Measure of function abstraction
SCALE	loc	Lines of code
	noc	Number of children
	rfc	Response for a class
	npm	Number of public methods
	wmc	Weighted methods per class
	dam	Data access metric

Table A5: Categories of software metrics in the Softlab datasets

SOFTLAB		
View	Symbol	Metric Full Name
MCCABE	v(g)	Cyclomatic complexity
	iv(G)	Cyclomatic density
	Iv(G)	Design complexity
LOC	loc_total	Total lines of code
	loc_blank	number of blank lines
	loc_comments	number of comment lines
	loc_code_and_comment	number of code and comment lines
	loc_executable	number of lines of executable code
HALSTEAD	N1	number of operators
	N2	number of operands
	μ_1	number of unique operators
	μ_2	number of unique operands
	N	program length
	V	volume (program size)
	L	program level
	D	difficulty level
	I	content c
	E	effort to implement
	B	estimated number of bugs
	T	implementation time
MISCELLANEOUS	branch_count	number of branch counts
	call_pairs	number of calls to other functions
	condition_count	number of conditionals in a given module
	decision_count	number of decision points
	decision_density	Condition count / Decision count
	design_density	iv(G) / v(G)
	multiple condition count	number of multiple conditions
	normalized_cyclomatic_complexity	v(G) / number of lines
	formal parameters	Identifiers used in a method

B. Appendix B: Experimental Results

Table B1: Comparison of the KNN and MVKNN algorithms on the NASA MDP and Softlab datasets in terms of precision, recall, and F1 Score

NASA MDP						
	Precision		Recall		F1 Score	
Dataset	KNN	MVKNN	KNN	MVKNN	KNN	MVKNN
cm1	0.81	0.81	0.90	0.90	0.85	0.85
jm1	0.77	0.77	0.81	0.81	0.79	0.79
kc1	0.81	0.82	0.85	0.85	0.83	0.83
kc2	0.81	0.82	0.83	0.83	0.82	0.82
pc1	0.87	0.87	0.93	0.93	0.90	0.90
Avg.	0.81	**0.82**	**0.86**	**0.86**	**0.84**	**0.84**
Softlab						
	Precision		Recall		F1 Score	
Dataset	KNN	MVKNN	KNN	MVKNN	KNN	MVKNN
ar1	0.86	0.86	0.93	0.93	0.89	0.89
ar3	0.89	0.89	0.90	0.90	0.89	0.89
ar4	0.82	0.85	0.84	0.86	0.83	0.85
ar5	0.76	0.85	0.80	0.83	0.78	0.84
ar6	0.73	0.73	0.85	0.85	0.79	0.79
Avg.	0.81	**0.84**	0.86	**0.87**	0.84	**0.85**

Table B2: Comparison of the KNN and MVKNN algorithms on the OSSP datasets in terms of precision, recall, and F1 Score

OSSP							
		Precision		Recall		F1 Score	
Dataset	Release	KNN	MVKNN	KNN	MVKNN	KNN	MVKNN
ant	1.3	0.73	0.70	0.83	0.84	0.78	0.76
	1.4	0.62	0.60	0.77	0.77	0.69	0.67
	1.5	0.81	0.79	0.89	0.89	0.85	0.84
	1.6	0.74	0.78	0.76	0.79	0.75	0.78
	1.7	0.78	0.78	0.8	0.8	0.79	0.79
arc	–	0.81	0.78	0.88	0.88	0.84	0.83
berek	–	0.80	0.86	0.79	0.86	0.79	0.86
e-learning	–	0.85	0.85	0.92	0.92	0.88	0.88
forrest	0.7	0.71	0.85	0.78	0.83	0.74	0.84
	0.8	0.88	0.88	0.94	0.94	0.91	0.91
jedit	3.2	0.72	0.77	0.73	0.78	0.72	0.77
	4.0	0.78	0.82	0.79	0.82	0.78	0.82
	4.1	0.77	0.83	0.78	0.82	0.77	0.82
	4.2	0.85	0.88	0.87	0.88	0.86	0.88

Table B2 continued

OSSP							
		Precision		Recall		F1 Score	
Dataset	Release	KNN	MVKNN	KNN	MVKNN	KNN	MVKNN
jedit	4.3	0.96	0.96	0.98	0.98	0.97	0.97
log4j	1.0	0.73	0.75	0.76	0.77	0.74	0.76
	1.1	0.76	0.82	0.76	0.81	0.76	0.81
	1.2	0.85	0.85	0.92	0.92	0.88	0.88
pbeans	1.0	0.63	0.58	0.71	0.73	0.67	0.65
	2.0	0.69	0.64	0.8	0.78	0.74	0.70
poi	1.5	0.67	0.7	0.67	0.7	0.67	0.70
	2.0	0.8	0.78	0.88	0.88	0.84	0.83
	2.5	0.75	0.78	0.75	0.78	0.75	0.78
	3.0	0.77	0.81	0.76	0.81	0.76	0.81
prop 6	–	0.81	0.81	0.9	0.9	0.85	0.85
redactor	–	0.85	0.88	0.87	0.89	0.86	0.88
serapion	–	0.74	0.85	0.8	0.82	0.77	0.83
synapse	1.0	0.81	0.81	0.9	0.9	0.85	0.85
	1.1	0.68	0.73	0.73	0.75	0.70	0.74
	1.2	0.7	0.7	0.71	0.71	0.70	0.70
tomcat	–	0.83	0.83	0.91	0.91	0.87	0.87
velocity	1.4	0.8	0.84	0.81	0.82	0.80	0.83
	1.5	0.7	0.77	0.72	0.74	0.71	0.75
	1.6	0.63	0.72	0.66	0.72	0.64	0.72
xalan	2.4	0.77	0.72	0.85	0.85	0.81	0.78
	2.6	0.69	0.75	0.68	0.74	0.68	0.74
	2.7	0.98	0.98	0.99	0.99	0.98	0.98
xerces	1.2	0.7	0.87	0.84	0.84	0.76	0.85
	1.3	0.81	0.89	0.86	0.89	0.83	0.89
	1.4	0.74	0.78	0.77	0.79	0.75	0.78
Avg.		0.77	**0.79**	0.81	**0.83**	0.79	**0.81**

Table B3: Comparison of single-view and multi-view accuracy values of the KNN and MVKNN algorithms on the OSSP datasets

ID	Dataset Name	KNN						MVKNN					
		view1	view2	view3	view4	view5	All views	view1	view2	view3	view4	view5	All views
1	ant 1.3	80.80	84.00	84.00	84.00	82.40	83.04	80.80	83.20	83.20	83.20	82.40	**84.00**
2	ant 1.4	77.53	76.40	76.40	76.97	76.97	76.85	75.84	74.72	74.16	72.47	74.16	**76.97**
3	ant 1.5	88.74	89.08	88.74	89.08	90.10	**89.15**	89.08	89.08	88.40	88.4	91.13	89.08
4	ant 1.6	78.06	76.92	73.79	73.22	80.06	76.41	77.49	76.35	75.21	75.78	78.92	**79.20**
5	ant 1.7	80.67	80.94	79.60	78.52	81.48	**80.24**	80.81	80.94	79.46	78.93	80.27	80.00
6	arc	88.46	88.46	87.18	88.46	88.89	**88.29**	88.03	88.03	88.03	87.61	87.61	88.03
7	berek	81.40	83.72	76.74	72.09	81.40	79.07	72.09	90.70	86.05	72.09	88.37	**86.05**
8	e-learning	92.19	92.19	92.19	92.19	92.19	**92.19**	90.62	92.19	92.19	92.19	90.62	**92.19**
9	forrest 0.7	86.21	75.86	75.86	82.76	68.97	77.93	82.76	75.86	75.86	89.66	72.41	**82.76**
10	forrest 0.8	93.75	93.75	93.75	93.75	93.75	**93.75**	93.75	90.62	90.62	93.75	93.75	**93.75**
11	jedit 3.2	70.59	66.54	73.90	75.74	76.47	72.65	68.38	65.44	73.53	75.00	76.10	**77.57**
12	jedit 4.0	78.10	79.74	76.80	79.41	81.70	79.15	74.51	80.07	75.49	80.39	81.37	**82.03**
13	jedit 4.1	81.41	77.24	77.56	76.92	78.53	78.33	80.45	78.21	78.85	78.21	80.13	**81.73**
14	jedit 4.2	87.74	86.65	87.19	86.92	88.01	87.30	88.56	86.92	88.01	87.19	87.47	**88.28**
15	jedit 4.3	97.76	97.76	97.76	97.76	97.76	**97.76**	97.76	97.76	97.76	97.76	97.76	**97.76**
16	log4j 1.0	77.04	71.11	78.52	77.04	77.78	76.30	76.30	72.59	80.00	80.00	80.00	**77.04**
17	log4j 1.1	77.06	74.31	70.64	77.06	79.82	75.78	75.23	71.56	76.15	80.73	79.82	**80.73**
18	log4j 1.2	92.20	92.20	92.20	92.20	92.20	**92.20**	92.20	92.20	92.20	92.20	90.73	**92.20**
19	pbeans 1	73.08	69.23	69.23	76.92	69.23	71.54	76.92	65.38	80.77	76.92	73.08	**73.08**
20	pbeans 2	80.39	78.43	84.31	80.39	80.39	**80.78**	80.39	74.51	82.35	78.43	74.51	78.43
21	poi 1.5	72.15	57.38	65.82	69.2	70.04	66.92	70.04	65.82	64.98	68.78	69.62	**70.04**
22	poi 2.0	88.22	88.85	88.22	88.22	88.22	**88.35**	88.22	88.85	86.94	88.22	88.22	88.22
23	poi 2.5	78.96	67.27	76.36	73.51	79.74	75.17	76.36	71.69	77.92	72.21	81.30	**77.92**
24	poi 3.0	76.70	78.05	75.34	73.98	75.79	75.97	76.24	77.15	76.70	73.98	78.28	**80.77**
25	prop-6	90.00	90.00	90.00	90.00	90.00	**90.00**	89.85	90.00	90.00	90.00	90.00	**90.00**
26	redaktor	88.07	84.09	90.34	86.36	87.50	87.27	85.23	85.80	90.34	87.50	87.50	**89.2**
27	serapion	80.00	80.00	77.78	80.00	82.22	80.00	77.78	71.11	77.78	80.00	86.67	**82.22**
28	synapse 1.0	89.17	89.17	89.81	89.81	89.81	89.55	89.17	89.17	88.54	89.81	88.54	**89.81**
29	synapse 1.1	72.52	70.27	75.68	71.17	76.13	73.15	72.07	72.07	73.87	72.07	77.03	**74.77**
30	synapse 1.2	71.09	71.88	73.44	67.58	72.27	71.25	69.53	70.70	72.27	67.19	70.70	**71.48**
31	tomcat	91.03	91.03	91.03	91.03	91.03	**91.03**	91.14	90.91	90.91	90.79	91.49	**91.03**
32	velocity 1.4	80.10	75.51	84.69	83.67	80.10	80.81	81.63	74.49	84.69	84.69	77.04	**82.14**
33	velocity 1.5	73.83	73.36	70.09	66.82	74.30	71.68	70.56	71.50	72.90	69.63	72.90	**74.30**
34	velocity 1.6	64.19	62.01	71.18	64.19	66.81	65.68	64.19	68.12	69.87	67.25	67.25	**72.05**
35	xalan 2.4	84.79	84.79	84.79	84.79	84.92	**84.82**	84.09	84.09	84.65	84.92	84.65	84.79
36	xalan 2.6	72.43	71.30	65.88	61.13	71.64	68.48	70.17	71.86	69.27	69.49	74.12	**73.67**
37	xalan 2.7	98.79	98.79	98.79	98.79	98.79	**98.79**	98.79	98.79	98.79	98.79	98.79	**98.79**
38	xerces 1.2	83.86	83.86	83.86	83.86	83.86	**83.86**	83.86	83.64	83.86	83.86	83.64	**83.86**
39	xerces 1.3	86.75	84.55	86.75	83.44	87.64	85.83	86.53	84.33	88.52	85.65	87.64	**88.52**
40	xerces 1.4	71.94	77.89	92.18	72.45	71.94	77.28	72.45	77.89	92.52	72.45	71.60	**79.42**
	Average	81.94	80.36	81.71	80.79	82.02	81.37	81.00	80.36	82.34	81.45	82.19	**83.10**

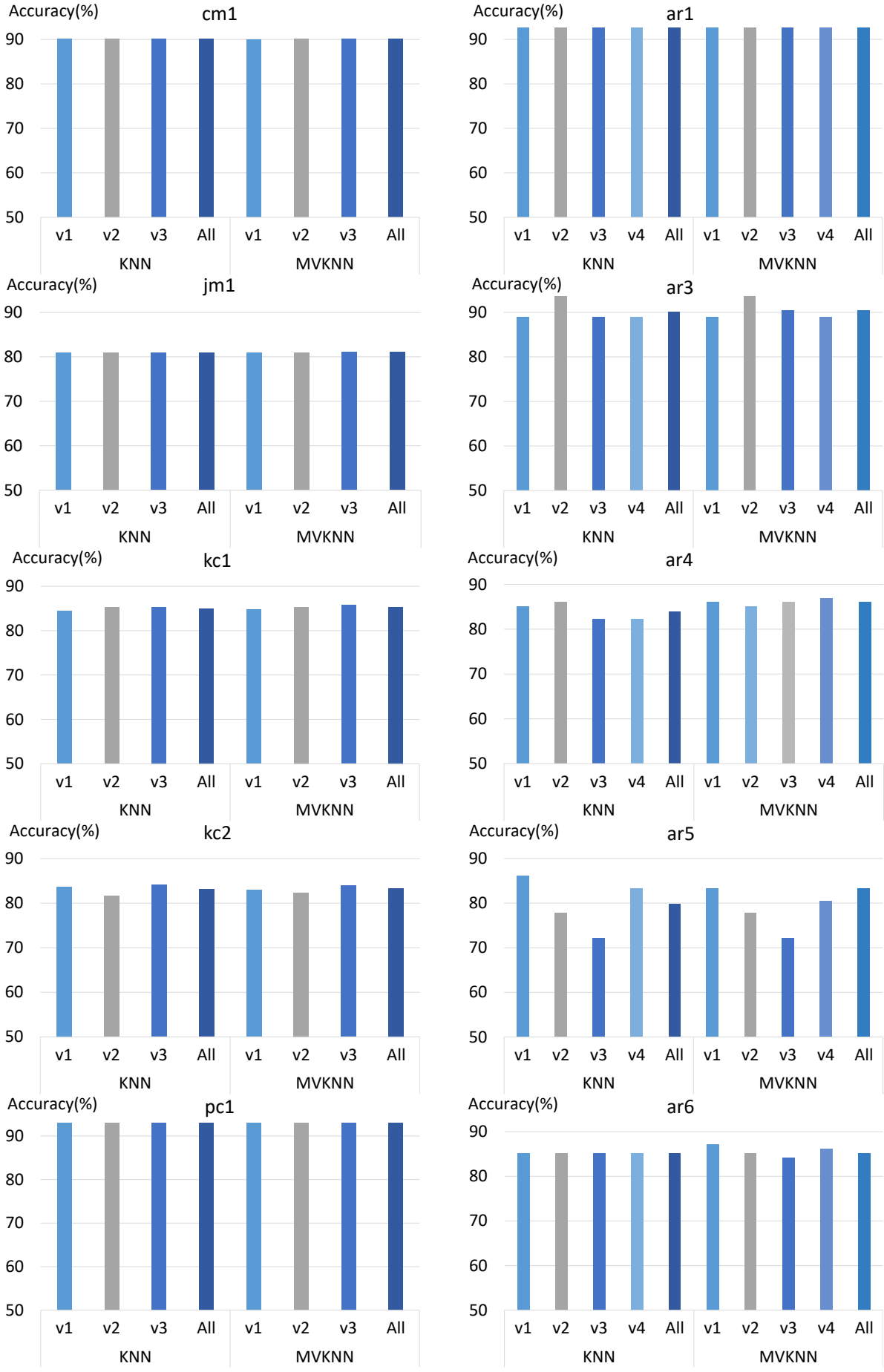

Figure B1: View-based comparison of the KNN and MVKNN algorithms

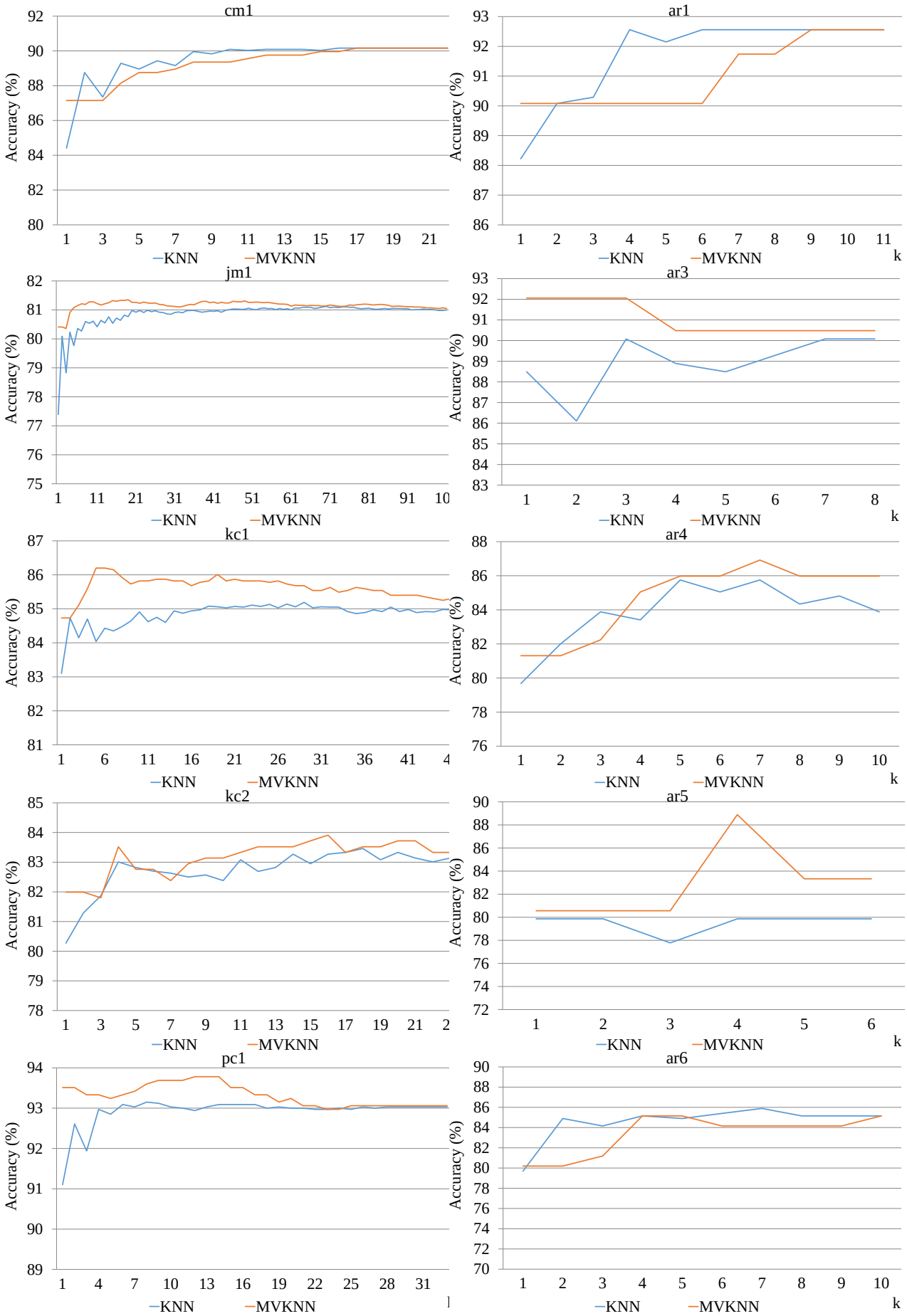

Figure B2: Comparison of single-view and multi-view versions of the KNN algorithm on various k values