ArchiNet: A Concept-token based Approach for Determining Architectural Change Categories

Amit Kumar Mondal Banani Roy Sristy Sumana Nath Kevin A. Schneider
University of Saskatchewan, Canada
{amit.mondal, banani.roy, sristy.sumana, kevin.schneider }@usask.ca

***Abstract*—Causes of software architectural change are classified as *perfective, preventive, corrective, and adaptive*. Change classification is used to promote common approaches for addressing similar changes, produce appropriate design documentation for a release, construct a developer's profile, form a balanced team, support code review, etc. However, automated architectural change classification techniques are in their infancy, perhaps due to the lack of a benchmark dataset and the need for extensive human involvement. To address these shortcomings, we present a benchmark dataset and a text classifier for determining the architectural change rationale from commit descriptions. First, we explored source code properties for change classification independent of project activity descriptions and found poor outcomes. Next, through extensive analysis, we identified the challenges of classifying architectural change from text and proposed a new classifier that uses concept tokens derived from the concept analysis of change samples. We also studied the sensitivity of change classification of various types of tokens present in commit messages. The experimental outcomes employing 10-fold and cross-project validation techniques with five popular open-source systems show that the F1 score of our proposed classifier is around 70%. The precision and recall are mostly consistent among all categories of change and more promising than competing methods for text classification.**

***Index Terms*—Architectural change; text classification; concept extraction; code review**

I. Introduction

Software architecture is concerned with the partitioning of a software system into parts, with a specific set of relations among the parts [10]. A meaningful architectural document helps reduce the cognitive load and maintenance activities of the software development team [16]. Moreover, appropriate architectural formulation is becoming more critical to circumvent software bloat, scalability, and security backdoors [11]. However, elements of architecture can be changed [20] continuously as code components of a software system changes to support continuous development and maintenance [29] such as adding new features, restructuring the design models, and fixing flaws. Architectural change can affect many aspects of a software system and, for this, change analysis is a crucial task. Development team can group architectural changes leveraging change classification process based on the cause of the change, type of change, location of the change, the size of the code modification, and impact of change [40], [15]. For example, four major causes of architectural changes have been defined explicitly in the literature [40], [8], [29]: (i) *perfective* – adjusting new behaviour, (ii) *preventive* – prevent bad design, (iii) *corrective* – correct discovered problems, and (iv) *adaptive* – adapting to new platform.

DOI reference number: 10.18293/SEKE2021-060

Grouping causes of change is beneficial for post-release analyses, where design change activities are not explicitly annotated [8]. Change classification is also required for composing a developer's profile, building a balanced team, and handling anomalies in the development process [21]. Furthermore, code review process involving architectural change is complex than local or atomic change [38], which is dependent on determining change type. Moreover, an automated technique can be employed to produce design documentation for every release recording types of structural changes happened and associated components [17]. Automated architectural change classification technique [40], [32] can be used to develop strategies for implementing a system change, support continuous architecture, augment DevOps and Model-Driven Engineering tools [6], [12], [11]. Existing active software projects (even if we consider a tiny portion of the 100 million repositories in GitHub [1]) could immediately benefit if a structural change classification technique is available to help develop an architectural versioning schema.

However, while architectural change can be identified from source code change, identifying the design decision, reason, and categories of changes requires analyzing the development team's intention. The intention can be extracted from textual description of the developer's tasks and discussions [6], [29]. Literature has focused on classifying typical software changes, architectural design concerns and design solutions [41], [28], [14]. Yet, supporting architectural change classification is still in its infancy [29], [11], perhaps, due to lack of benchmark data and requirements of laborious human analysis. Nevertheless, a few of the studies explored for both manual [8], [32] and semi-automated [29] techniques for classifying architectural changes. In these studies, a small collection of samples is being experimented where challenges are not identified properly, which leads to developing infeasible models. Besides, the traditional text classification techniques [7], [42] might not handle the scenario when keywords are present among multiple concepts within the description of a task.

To address the shortcomings, we design a benchmark data and propose a text classifier called ArchiNet for architectural change classification. In particular, we focus on the two research questions: **RQ1:** How can source code properties that are independent of the description of project activities classify the rationale of architectural changes?, and **RQ2:** How can we improve text classification to predict the rationale of architectural changes leveraging commit descriptions?

To answer RQ1 and RQ2, we collect around 1,133 architectural change instances from 5K commits of five popular projects (shown in Table I). After extensive analysis of the created dataset, we have successfully identified the challenges of categorizing the architectural changes both from the source code and the texts. One of the challenges is that typical operations in the source code do not have a significant number of distinguishing patterns in various changes, and classification performance is not promising (F1 score is 33%). A major challenge in the commit description is that multiple concepts are presented, whereas only one or two concepts indicate the intention. Furthermore, many words are common for expressing the reasons for changes, such as keyword *update* is used to describe both *perfective* and *adaptive* changes. Such a phenomenon is not acute in many other text classification tasks [19]. All things considered, we propose a new technique for classifying the changes from the text where trained keywords from concept analysis of different changes play a crucial role. The training process of our proposed technique is different from the traditional NLP training process. For training, we first define the relevant concepts (contextual occurrence of words and tokens such as $\{update, API, version\}$ indicate *adaptive* change more confidently) within each sample. Next, all tokens' weights appeared within all the concepts for a relevant change class are calculated. These weights are distributed among all of the classes leveraging a statistical model. Thus, our technique does not consider all the words within a description. Finally, a given commit is predicted to one of the four classes using a probability model from the trained database. Experimental outcome of our classifier with different datasets shows that the F1 score is around 70% and promising compared to the competing techniques (including deep learning).

The paper continues as follows. In Section II we discuss the background of architectural change detection and classification. Section III describes our dataset creation process. Section IV explains the challenges of change classification. Our proposed classifier is presented in Section V. Section VI reports our experimental outcome and Section VII discusses threats to validity. Section VIII discusses related studies and Section IX concludes our paper with future direction.

II. Background

Architectural Change Instance: Studying typical changes from version control systems does not require a change detection strategy as it provides differences. However, architectural change detection [20], [24], even from the version control system (*diff*), is challenging. Some of the widely used change metrics are DSM [5], MoJo [39], MoJoFM [39], graph kernel structure [30], A2A [20], C2C [24] and include-symbol dependencies [24]. These metrics are calculated based on the following operations: *adding components, removing components, replacing components, splitting components, merging components, relocating, module dependency graph, and usage dependency*. We focus on intermediate-level architecture for collecting change samples and employ $A2A$ and $include{+}symbol$ dependency metrics for change detection. A2A considers component addition, removal and moves; $include + symbol$ dependency considers including/removing header file, program file, importing class, and importing interface. Causes for architectural changes are grouped as follows.

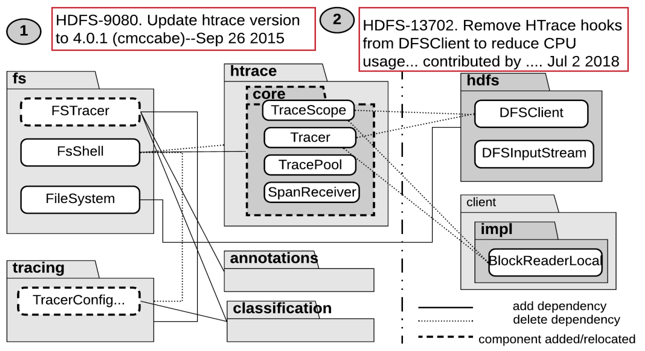

Fig. 1: Two commits of *Hadoop* where new components are added, dependency added and deleted.

Adaptive (A) change: This change would be a reflection [40], [22], [37] of system portability, adapting to a new platform such as commit① in Fig. 1. Adaptive change also happen for imposing new organisational and governmental policies.

Corrective (C) change: A corrective change is the reactive modification of a software product performed after deployment to correct discovered problems [40]. Specifically, this change refers to defect repair, and the errors in specification, design and implementation.

Preventive (PV) change: Preventive change [37], [40] refers to actionable means to prevent, retard, or remediate code decay. In other meanings, preventive changes happen to improve file structure or to reduce dependencies between software modules and components may later impact quality attributes such as understandability, modifiability, and complexity.

Perfective (PF) change: Perfective changes are the most common and inherent in development activities. This change mainly focuses on adjusting new behaviour or requirements changes [37]. Also, these changes are aimed at improving processing efficiency and enhancing the performance of the software (such as commit② in Fig. 1) that is both functional and non-functional optimizations.

This classification is essential to deal with various challenges (discussed in the Introduction) since different types of change influence them in different ways. Among the change categories, preventive and corrective changes are directly related to major design debt management. A few of the change types in the two commits in Hadoop is shown in Fig. 1. Commit descriptions simply express their intentions. Commit① is an adaptive change in 2015 and commit② is a perfective change in 2018. It is noticeable from commit② that a dependency change between two components ($htrace$ and $hdfs$) increases performance by reducing CPU usage, which is also an architectural change. Both of the changes happen almost a decade later of the first

TABLE I: Candidate projects for our study (in inspection time).

Project	All	A.	Domain	Source
Hadoop	22631	266	Distributed Computing	gt/apache/hadoop
HibernateORM	9811	261	Object/Relational Mapping	gt/hiber../hibernate-orm
LinuxTools	10630	265	C&C++ IDE for Linux	gt/eclipse/linuxtools
JavaClient	1477	136	Java bind for Appium Tests	gt/appium/java-client
JVMcouchbase	914	205	JVM core for Couchbase Server	gt/couchbase/couchbase-jvm-core
Total	45463	1133		

A: architectural changes in selected 1K commits; gt: github.com

release of Hadoop. Components of the *htrace* module are at the center of these two changes (commit① and commit②) although the second change occurred after three years of the occurrence of the first change.

III. Dataset Preparation

A few of the studies [8], [29] created datasets for architectural change classification from the development history. The recent dataset created by Mondal et al. [29] consists of 362 samples of four projects (26 of them are adaptive). This dataset might be insufficient for detecting some of the text classification challenges such as various concept tokens including code elements and framework name. Created dataset by Pixao et al. [32] contains architectural change only for new features and other categories are not annotated (recently they updated their dataset with fixing issues but not specifically annotated to four groups discussed widely in the literature [40], [8], [29]). Another dataset is constructed by Ding et al. [8] which is not publicly available (thanks to the authors for providing us 37 samples). Therefore, we prepare a new dataset containing a large collection of commits (shown in Table I).

1) Architectural Change Commits Filtering: We selected five open source projects that are widely experimented in literature for software change and architectural analysis [28], [20], [17], [32], ensuring a diversity of domains. We also ensure that the projects are in active development for at least several years. The selected projects are: Hadoop, Hibernate ORM, Linux Tools, Java Client, and Couchbase JVM Core have 45,463 commits which are infeasible to analyze manually. Since determining and categorising architectural change instances require huge human efforts, in our dataset creation process, we restrict primary selection of commit samples into 5K. We randomly choose 1K commits from each of the projects containing more than two words in the messages excluding stop words, non-alpha words (that contains non-letters such as *issue-110*) along with the words having *Change-Id:* or *Signed-off-by:* and so on as shown in Fig. 1. We separate the architectural change samples from the primary collection (around 5K) if $A2A$ and $include + symbol$ dependency metrics are changed. However, as suggested by the literature [25], we do not consider system library usage from native (Java, Python) framework for dependency change. In this way, we get around 1133 samples (distribution of them is shown in Table I) as architecturally changed commits.

2) Architectural Change Category Annotation: In the next step of our study, we manually label those samples by two authors independently into one of the four categories described in the existing studies [29], [8]. There are ambiguities in some of the descriptions of four types of changes. We review most of the relevant papers referred by [40], [8], [29] for more explanation to resolve the ambiguity (details are discussed in Section II). Our manual annotation process has two iterations. In the first iteration, two of the authors having three years of average software industry experience, categorized the samples separately. In this step, we get many samples mismatched in annotation. In the second iteration, we recheck the mismatch samples and resolve the disagreements by discussion. Total number of samples in each of the annotated categories from the candidate projects is shown in Table II. The finalization of our dataset took one month of two person-hours, indicating that manual change analysis is expensive. In the next section, we investigate the automatic change classification challenges.

TABLE II: Training and test samples in the golden set.

Split	Perfective	Corrective	Preventive	Adaptive	Total
Train	425	122	185	68	800
Test	173	49	73	39	333
Total	598	171	258	107	1133

IV. Change Classification Challenges

For examining the challenges of classification, we divide the samples into two parts: training and test sets. As empirical study [18] suggests that 30% test samples are ideal for real data, we split around 70% of the architectural commits for training purposes and around 30% for testing purposes with random sampling. However, we could not extract meaningful concepts (Section IV-B2) for some of the samples due to lack of information, and skip those during the training and testing phases. Distribution of change types in the train and test sets are shown in Table II. Both the train and test sets contain the conflicted samples accordingly.

A. Classification from Source Code

First, we explore classification options leveraging source code operations. Yamauchi et al. [41] cluster the change commits based on source code modifications: identifiers, method name, and class name into as many groups dependent on component-requirement relations. Their technique cannot be used for a fixed number of classes. The clustering basically groups the commits into related components attached to an implemented functional requirement, not the reason for changes. Therefore, we explore a technique utilizing the distribution of change operations of the architectural components (static). We examine the abstract operations (O) occurred in the source code of a commit: *import added* or *deleted*, *class file added* or *deleted*, *file* or *package rename*, and *function added* or *deleted* as properties of change classification since they are universal and independent of project context.

Considering these properties, we design a classifier using $C_i(w_O)$ in (1) as described in Section V to evaluate how significant the prediction is using these operations as metrics and has the following outcome with 10 fold cross-validation. The best F1 score (among different combinations of the operation types) for perfective, preventive, corrective, adaptive, and all combined are 0.33, 0.53, 0.08, 0.13, and 0.33 respectively. F1 for the corrective and adaptive classes are negligible. In summary, source code properties are not promising for architectural change classification; this answers our RQ1. In the next section, we explore existing change classification techniques from commits messages.

B. Change Classification from Text

1) Explored Models: Next, we examine the explored models of Mondal et al. [29], where the best model produces 39% F1 scores with our dataset. Following these approaches, we also develop a discriminating feature selection (DFS) model from the distribution of words in our training dataset. Similar to Mondal et al., our DFS model has many common keywords in the top list. Considering such overlapping of keywords, existing techniques based on the DFS model discussed in DPLSA [42], LLDA [34], and SemiLDA [9] predict more false positives since such a model also considers the words that might be irrelevant to the original intentions. With our new collection, the best DFS model produces 46% F1 score with precision 45.6% which is similar to the outcome of the best method in [29]. Our DFS model for the dataset in [29] produces an F1 score of 20% that is significantly lower than the previous model. In summary, the DFS models are not promising and possibly biased to the project contexts. Therefore, we focus on a more advanced classifier identifying the challenges within the textual descriptions. We discuss classifiers from traditional machine learning and neural word embedding models in Section VI.

2) Concept Analysis: As we have a large number of samples, we are able to identify the specific challenges within the message description. One of the significant challenges present in many commit messages is developers express more than one concept (*contextual occurrence of words*) for a single intention. An N-gram model might capture continuous sequences of n-words involved in such concepts within a sentence [35], [36]. However, in multiple iterations of our inspection, we find that concept words are scattered among multiple sentences in many commit descriptions. We also prioritize such scattered words while categorizing the commits. Traditional text classification techniques do not address this particular scenario (including the n-gram model). We also attempt to determine the dominating concepts from multiple concept tokens. Lets consider the corrective change message *"adding more support for services **down..**"*; here $adding\ support$ and $down$ keywords will influence to predict a category by $tf - idf$ [31], LLDA, SemiLDA, and DPLSA techniques. Unfortunately, $adding$ and $support$ keywords will measure more weight to other categories because they are present among the list of the top keywords. However, if we prioritize the $down$ keyword as the dominating concept, it is more likely to be a corrective category. We annotated such keywords for the dominating categories.

In the list from Section IV-B1, some dominating words (such as issue and leak) for this corrective category are hardly used for others. But, many samples contain negative words which are not meant faults, such as -*"...This changeset moves the responsibility of sending into the locators, which has two benefits:- **No** Node[] allocations since nothing needs to be signalled back.- The code **doesn't** need to iterate through the list again ..."* is more likely to be a preventive change despite too many negative words. This is significantly an opposite concept in the sentiment analysis [4], which would treat this as negative for such words. The existing techniques falsely classify such a description as a corrective one. However, in many samples, when the word ***not*** co-locates with the word $working$, it indicates flaws in the system. Therefore, we should not skip such keywords during concept extraction. Furthermore, some code elements are used for assuming a corrective concept such as NullPointerException and LinkedError. For the adaptive category, the dominating concept is indicated by mostly multiple words. From the example in Fig. 1, we notice that $update$, and $version$ form a dominating concept together where domain specific terms (such as htrace, API, library, and so on) need to be included. Again, these words are present in other categories. We have manually re-analyzed all the training samples to find such concepts containing the minimal number of words. Overall, there are ambiguities of concepts (and top keywords) among all the categories. The most ambiguities are found in the perfective category to define the related concept uniquely with the minimum number of words. While manual annotation is easier for the perfective categories, defining the dominating concept, as discussed previously, is the most difficult. We have seen that a word might indicate different concepts with co-occurring different terms as shown in Table III. In the next section, we describe our proposed solution based on this finding.

TABLE III: Ambiguity of concepts appeared in description.

Base words	Not failure	Faults
Not	complex	
Doesn't	need	work, release
Error	message	$-$network, $-$fix
Can't		change

Symbol '$-$' indicates located before the base word.

V. Our Proposed Classifier: ArchiNet

From the empirical observations, it is evident that handling overlapping words among the descriptions is the key to develop a promising solution. We conjecture that no word should be in the distinguishing list to a single category. Instead of the logic of the previous techniques, we assign a strength of a word for each of the categories. For example, for the strength of the words presented in Table IV for different concepts, if the words $add, support, down$ appear within a text description, then the total value for the category C_1 is $0.52 + 0.38 + 0 = 0.90$, and the total value for the category C_2 is $0.03 + 0 + 1 = 1.03$. As $1.03 > 0.90$, the sample would be for the category C_2. For simplicity, we explain with weight addition; more complex situations (with various token strengths) are handled with a probabilistic prediction technique described in Section V-3. Therefore, this gives more importance to the co-occurrence of the words add and $down$, and such a solution might handle the described challenges in a promising way. In our solution, the crucial point is to get the concept tokens and their weights distribution efficiently, and then predict a class confidently. We describe our proposed method in three steps.

TABLE IV: Strength of words within the concepts C_1 and C_2.

Word	Strength in C_1	Strength in C_2
add	0.52	0.03
support	0.38	0
down	0	1

1) Concepts Extraction: In this stage, we define and extract concepts from the commit messages of all the annotated samples that express the corresponding intention of a task (as discussed in Section IV-B2). Even, the top words (such as *support*) among the defined concepts contain many overlapping words. However, we have found some patterns in many samples for expressing different concepts when these terms are co-occurred with other tokens which are stop words, code elements, and API, library or framework name. Some of the examples are discussed in earlier sections. Before training, natural words are stemmed with PorterStemmer. In the next section, we discuss our training and weight distribution process from the extracted concept tokens.

2) Training Model Generation: In this phase, we train a model by assigning weights to the concept tokens using (1) from a set of preclassified commits into four change categories. This is motivated by the core idea of how the model for a word's sentiment is generated [2]. These weights represent the strengths while present within the concepts of the categories. The trained model produces a collection of unique concept tokens denoted by S having weights w_i to the classes C_i:

$$C_i(w_S) \Rightarrow \bigcup_{t\epsilon S} \frac{w_i(t)}{\sum_{1}^{i} w_i(t)} \cup C_i(w_O),\ S = \{t_w, t_p, t_s, t_a\} \tag{1}$$

$$w_i(t_w) = \frac{f(t_w)}{N_i}, w_i(t_p) = \frac{\boldsymbol{F}(t_p)}{N_i}, w_i(t_s) = \frac{\boldsymbol{F}(t_s)}{N_i}, \tag{2}$$

$$w_i(t_a) = \frac{\boldsymbol{F}(t_a)}{N_i}, w_i(t_o) = \frac{\boldsymbol{F}(t_o)}{N_i}\ \ where\ \ O = \{t_o\} \tag{3}$$

Here, $f(t)$ frequency of a token t within the concepts S of all samples in a category N_i, and $C_i(w_O)$ is the weight of the tokens defined with source code change operation types (O) associated with $S, C_i\epsilon\{PF, PV, C, A\}$. $w_i(t)$ can be calculated by adopting various metrics such as frequency value or $tfidf$. A concept S consists of various types of tokens such as t_w is natural words without stop words, t_p is some specific stop words such as negation words, t_s is some special code elements such as *NullPointerException* and *LinkedError*, t_a is api, library or framework name, and t_o is code operation types treated as tokens; each of these types has a collection of tokens. We utilized frequency normalized sum for calculating the probability value. We calculated the frequency values differently represented by bold **F** in (2) and (3): $\boldsymbol{F}(t_p)$ considers only inclusive stop words, $\boldsymbol{F}(t_s)$ consider the issue related token parts (such as Exception and Error from the mentioned tokens) extracted from a code element using camel case parsing, $\boldsymbol{F}(t_a)$ is calculated by converting all the api names into a unique token ("AA/BB" in our experiment), and $\boldsymbol{F}(t_o)$ consider one or more instances of each token in t_o as value 1 within a commit. All the values in (1) and (2) are adjusted when new concepts are defined with new trained samples. Then, we employ a classifier from these trained weights.

3) Classification: During the classification phase, the generated models (M) from the training phase (in (1)) are used to evaluate the probability ($P_m(C)$) that a given class C is associated with the commit m. Only the tokens identified in the concepts (S) from phase one and tokens as the source code operations (O) in Section IV-A are considered from commit message and code change. The classification score is then defined as follows.

$$P_m(C) = \frac{\sum\limits_{t\epsilon(S\cup O)\cap C} M(w(t))}{\sum\limits_{t\epsilon(S\cup O), C_i} M(w(t))} \tag{4}$$

where the numerator is computed as the sum of the token weights ($w(t)$) of all types that are contained in C, and the denominator is the sum of the token weights for all types for all classes (C_i). The probabilistic classifier for a given commit m will assign a higher score $P_m(C)$ to class C that contains several strong tokens for concept S and operation O. However, if the probabilities $P_m(C_i)$ are same for more than one class, ArchiNet considers the class which contains the highest weighted word.

VI. Performance Evaluation

Our created dataset contains both an architectural change set three times larger than that of [29] and a list of concept-words with strength. Our proposed classifier ArchiNet is designed to handle the overlapping words and includes various tokens discussed in Section IV-B2 within the change description. We compare the performance of ArchiNet based on recall (R) – quantitative correctness of retrieving relevant categories; precision (P) – the rate of accuracy among the predicted samples, and the F1 score – $2PR/(P + R)$ calculated from precision and recall. We have also compared with the published dataset and classifiers [29], [42]. The performance is also compared with other promising techniques in literature [14], [28], [36], [23] for text classification. These techniques include RCNN-LSTM the state-of-the-art Deep Neural Learning (DNL), Naive Bayes (NB), Bag-of-words (BoW) model, Decision Tree (DT), Random Forest (RF), DPLSA, LLDA, and SemiLDA. Our training model is significantly faster than RF and DNL, but we will not discuss time complexity since it is less critical if a model is built once for application. We evaluated the performance of ArchiNet in the following four phases.

1) Testing with the Golden Set: We train our proposed method (ArchiNet) and other methods with the training set. Train and test set partitioning is described in Section IV. Then, the classification performance is tested with the test set (from Table II); comparison of the outcome is presented in Fig. 2. Please note that only methods having close performance are shown here. The most promising method in the baseline work by Mondal et al. is DPLSA, where discriminating keywords for the individual classes are used as features in a probabilistic model. The difference in the percentage of F1 score between

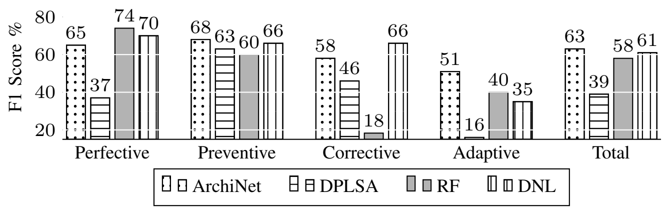

Fig. 2: F1 score comparison of ArchiNet with the most promising classifiers explored in [29], [36], [14].

ArchiNet and DPLSA for all classes is 24 points higher, while this difference is 35 points higher for the adaptive category. The F1 score of our model for the test data in [29] is 63% (shown in Table VI), which is 18 points higher compared to their best model (45% gain in performance). We have employed a DNL based text classifier [19], [36] with Google Tensorflow [3]. The DNL network where encoded words are embedded with the RCNN-LSTM strategy shows a 61% F1 score, which is 2 points lower than ArchiNet. The configuration of our DNL model has 64 layers, 64 units, epoch size 10, *relu* activation function, and cross-entropy as loss function [36].

Furthermore, we adopted the best algorithms suggested by Hindle et al. [14] to classify large change commits into five categories, and Soliman et al. [35] to classify architectural discussions. We also explore Naive Bayes (NB), Decision Trees (DT), and Random Forest (RF) [23], [14], [35] for our dataset with the WEKA [13] tool utilizing word-to-vector features [27]. Among them, the most promising classifiers such as NB, and DT have less than 55% F1. However, Random Forest (RF), which forms a group of DTs, produces around 58% F1 score for our dataset. The F1 score produced by our technique for the adaptive category is much higher than the competing methods. The ranges of precision and recall rate of ArchiNet among the individual categories are 42.4–77.8% and 64–73.7% respectively, which are more consistent than other classifiers. Notably, from the graph, we can see that F1 scores of RF and DNL for the perfective category is higher than ArchiNet, while significantly lower in the adaptive category because many samples from adaptive might be falsely predicted (high recall rate) into the perfective category (due to lack of handling mechanism of the overlapped concepts). We also see this pattern in the 10-fold validation phase. In this evaluation phase, the distribution of P, R, and F1 scores to all the classes with the test sets indicates a better and stable outcome of ArchiNet with the concept-words.

2) 10-folds Validation: In this phase, we show how our classifier is performing with cross-fold validation since it provides a more accurate evaluation against the over-fitting problem [14], [28]. However, we experiment with the promising methods proven in the first phase. We compare the performance of ArchiNet with DNL and RF by 10-fold cross-validation technique. In 10 iterations, we take 90% samples as the training set, and 10% as the test set exclusively for each of the iterations [28]. The performance comparison is presented in Table V. The F1 score of ArchiNet is around 69%, which is 7 points better

TABLE V: Performance (%) comparison of ArchiNet (A), Random Forest (RF), and Deep Neural Learning (DNL).

Metric	**Perfect**			**Correct**			**Prevent**			**Adapt**			**Combined**		
	A	RF	DNL	A	RF	DNL	A	RF	DNL	A	RF	DNL	A	RF	DNL
P	77.5	62.7	77.6	63.4	88	50.3	77.8	91	66.5	42.4	96	40.1	69.1	76	62
R	73.7	99	69	66.25	$\textcircled{19}$	62	63.8	43	62.1	64	$\textcircled{25}$	$\textcircled{28}$	69	67	62.1
F1	**76**	77	73	**63**	$\textcircled{30}$	$\textcircled{50}$	**70**	58	64	**51**	$\textcircled{40}$	$\textcircled{33}$	**69**	62.2	62

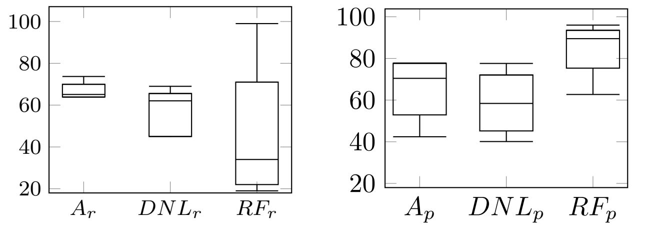

Fig. 3: Range of Recall (r) and precision (p) rate of all classes

than the two classifiers. Deep learning with RCNN-LSTM [19], [36] shows 62% F1 score; RF has a similar outcome as of DNL. F1 scores for some other classifiers are between 50 to 60% with the word-to-vector [27] features. From the median and range values in box plots in Fig. 3, it is observed that the precision and recall rate of ArchiNet (in the ranges 40.1–77.8% and 63.8–73.7% represented by A_p and A_r) are consistent with all the classes (recall is highly consistent than others indicated by DNL_r and RF_r). For the adaptive and corrective categories, the outcome of ArchiNet is significantly higher. Poor recall rate of RF and DNL (marked with circles in Table V) for the adaptive category and high recall rate for the perfective category indicates that many samples from adaptive are falsely retrieved into perfective by both of the classifiers. A similar trend is observed with the corrective category except for lower precision for DNL. Since RF and DNL do not distinguish and select words based on concepts/semantics, they produce more unstable outcomes. In summary, our proposed classifier has better performance for all metrics (F1, P, and R scores) compared to other classifiers because concept-words handle various influential tokens from a commit message efficiently. This exploration answers the research question RQ2.

3) Project-wise Validation: We also conduct cross-projects validation of our proposed approach. We train the classifier with four projects and test with the remaining project in each iteration for the five projects. The project-wise outcomes for both ArchiNet and DNL are presented in Fig 4. Combined F1 scores of each of the projects produced by ArchiNet are better than that of DNL. None of the project's F1 scores is below 60% for ArchiNet, while the highest is 69%. The highest precision is 85%, and the recall is 80% (for the perfective and preventive category) for our method. However, the precision and recall can be low for the adaptive category as can be seen in the Fig 4. On the other hand, the adaptive category's F1 score reaches 62 for the ArchiNet (whereas 23 for DNL). Performance of some of the projects is lower than 10-fold validation because of insufficient training data. Overall, Hadoop's outcome for

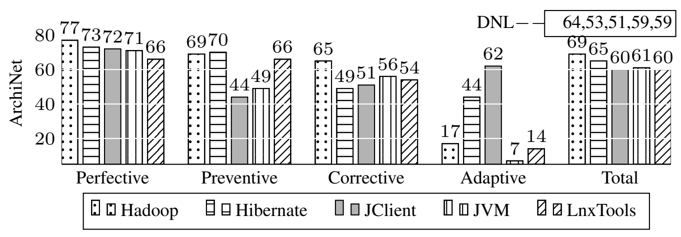

Fig. 4: F1 score for individual projects.

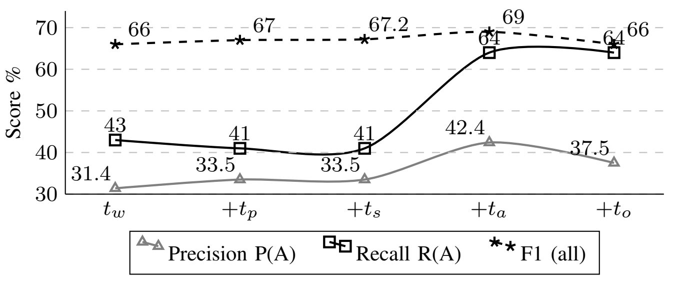

Fig. 5: Performance sensitivity of terms. '+' means including all others terms from left. *P,R* are from the Adaptive(A) class.

both ArchiNet and DNL is the most promising because the commit messages in Hadoop might contain a less ambiguous explanation compared to other projects.

4) Sensitivity of Tokens: The performance sensitivity of ArchiNet (for 10-folds) for various token-weights ($w(t)$) combination (in (1) and (2)) is shown in Fig. 5. The best performance is shown for the combination $\{t_w, t_p, t_s, t_a\}$ which is three points (69% F1) better than only considering natural terms (t_w) (66% F1). Including API, library, and framework name (t_a) increments the performance by two points as there is more likely to be an adaptive category for those compared to others. As can be observed from precision and recall in Fig. 5, the adaptive category is the most sensitive. However, we notice that combining source-code operations (t_o) affects the performance slightly negatively (66% F1, whereas it is 65% with t_w); therefore, source code operations are not promising features for classifying the architectural change.

VII. Threats to Validity

One of the greatest threats to the validity of our result is that annotating the intention of change is subject to human bias. To reduce this threat, two of the authors independently annotated, and then conflicts are resolved by discussion. Any classifier may suffer an over-fitting problem. To overcome this, we experimented with our classifier with a tenfold cross-validation technique and found a promising result. Another concern of our classification model is how general it predicts change from different programming languages and cross-projects. One of our test sets is collected from Mondal et al. [29] that also contains projects of Python language, and have similar outcome as shown in Table VI. A few of the projects such as Hadoop has

TABLE VI: F1 of ArchiNet with our data and data in [29].

Dataset	Perfective	Corrective	Preventive	Adaptive	All
Our data	65	58	**68**	**51**	**63**
Data [29]	55	**61**	**80**	16	**63**

substantial industrial participation [17]. Therefore, our study also mitigates generalizability threat to some extent.

Our model can be trained with different metrics. Therefore, for (2) in Section V, we also have trained our model with the *tf-idf* metrics. However, the result is not as promising as the direct probability value, but still shows a better result than DPLSA, LLDA, and SemiLDA. With this metric, the best F1 scores for the data in [29] are 47% and 51% for our benchmark data. Yan et al. [42] utilized DPLSA for predicting multiple categories of usual changes (three types). We found only a few of the samples in our data have multiple intentions when architectural changes happened. ArchiNet can handle such scenarios to some extent as we experiment on that mode; when the predicted sample is in Hit@2 [35], [33] (within the top 2 ranks), the F1 score is 83.5%. Notably, our proposed classifier is versatile and does not require parameter tuning, unlike others. Our dataset and trained models are available in *github.com/akm523/archinet* for further investigation.

VIII. Related Work

1) Architectural Design Issues and Solutions Classification: Yamauchi et al. [41] proposed a technique considering program identifiers to group the large commits into related components having relations with the functional requirements. An early approach of committed code classification was studied for architectural tactics (design solutions such as resource pooling, secure session management, and so on) [28] based on code identifiers (such as heartbeat) mapped with text description (heartbeat emitter and receiver) from a set of trained samples, and commits are predicted using a term-frequency based classifier. Solaiman et al. [35] reported Bayesian Network and Naive Bayes as the best algorithms to classify architectural discussions related to six ontology classes (such as technology) into three design steps focusing ambiguous concepts (such as $server$ has different meanings for different cases), concepts expressing reasons of architectural changes are different than those. However, although these classes were either subset or irrelevant to architectural changes, they were not specialized in four architectural changes. In our work, we explore both source code features and concept-token properties to predict the reasons for architecture changes.

2) Architectural Change Classification: We are aware of only one study by Mondal et al. [29] to categorize four architectural changes from the text. Their model was generated by popular discriminating feature selection techniques DPLSA [42], SemiLDA [9], and LLDA [34] originally proposed [42] for classifying all software changes into three groups, and none of the techniques could handle the twists and challenges of architectural change classification properly. Consequently, the outcome of their proposed technique is poor. Another study by Hindel et al. [14] close to ours explored various machine learning techniques for classifying large commits (commits with many files changed) into five groups. We also explore the promising classifiers reported by them: Naive Bayes, Decision Trees, and so on. However, Random Forest (RF), an advanced version of Decision Trees, produces promising outcomes with our dataset (but 7 points lower F1 than ArchiNet). Recently, word embedding technique that captures contextual and semantic information with deep learning is being successfully used for software artifacts analysis and classification [19], [36], [26]. However, due to the overlapping of concept words, deep learning might not produce the best outcome, which is mostly inexplicable when multiple intentions are required to extract from a single message. Our proposed classifier ArchiNet handles these concerns considering other tokens and gains in performance.

IX. Conclusion

In this paper, we present a dataset collected from five popular projects and a promising classifier for architectural change categorization from texts. Our study identifies the challenges of classifying changes from both source code properties and textual properties. We address those challenges with a concept analysis approach that indicate the developers' intentions. Both 10-fold cross-validation and cross-projects validation show that our technique is promising in all aspects compared to traditional methods (F1 score is 70%). We also explore the sensitivity of the performance of our classifier for various tokens. Besides, we extract around 237 keywords (with trained weights for each change category) from the training set. Given the success, many of the text analysis approaches to support the ten activities of software architecture discussed by Bi et al. [6] might be enhanced by adopting our proposed technique. In future, we will explore automatic design documentation generation and architectural versioning schema applying our change classification technique.

Acknowledgment

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by two Canada First Research Excellence Fund (CFREF) grants coordinated by the Global Institute for Food Security (GIFS) and the Global Institute for WaterSecurity (GIWS).

References

[1] Github projects:. https://thenextweb.com/news/github-now-hosts-over-100-million-repositories.

[2] Sentiwordnet: http://sentiwordnet.isti.cnr.it/. June, 2019.

[3] Tensorflow: www.tensorflow.org/tutorials/text.

[4] S. Baccianella, A. Esuli, and F. Sebastiani. Sentiwordnet 3.0: an enhanced lexical resource for sentiment analysis and opinion mining. In *Proc. of LREC*, 2010.

[5] C. Y. Baldwin and K. B. Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[6] T. Bi, P. Liang, A. Tang, and C. Yang. A systematic mapping study on text analysis techniques in software architecture. *JSS*, 2018.

[7] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3:993–1022, 2003.

[8] W. Ding, P. Liang, A. Tang, and H. Van Vliet. Causes of architecture changes: An empirical study through the communication in oss mailing lists. In *SEKE*, pages 403–408, 2015.

[9] Y. Fu, M. Yan, X. Zhang, L. Xu, D. Yang, and J. D. Kymer. Automated classification of software change messages by semi-supervised latent dirichlet allocation. *IST*, 57:369–377, 2015.

[10] D. Garlan, F. Bachmann, J. Ivers, J. Stafford, L. Bass, P. Clements, and P. Merson. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition, 2010.

[11] N. Ghorbani, J. Garcia, and S. Malek. Detection and repair of architectural inconsistencies in java. In *Proc. of ICSE*, 2019.

[12] P. Haindl and R. Plösch. Towards continuous quality: Measuring and evaluating feature-dependent non-functional requirements in devops. In *Proc. of ICSA-C*, pages 91–94. IEEE, 2019.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD*, 2009.

[14] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic classication of large changes into maintenance categories. In *Proc. of ICPC*, pages 30–39, 2009.

[15] A. Hindle, D. M. German, and R. Holt. What do large commits tell us? a taxonomical study of large commits. In *Proc. of MSR*, 2008.

[16] M. I.S.O. Systems and software engineering–architecture description. Technical report, ISO/IEC/IEEE 42010, 2011.

[17] R. Kazman, D. Goldenson, I. Monarch, W. Nichols, and G. Valetto. Evaluating the effects of architectural documentation: A case study of a large scale open source project. *Transactions on SE*, 2016.

[18] K. Korjus, M. N. Hebart, and R. Vicente. An efficient data partitioning to improve classification performance while keeping parameters interpretable. *PloS one*, 11(8):e0161788, 2016.

[19] S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent convolutional neural networks for text classification. In *Proc. of AAA*, 2015.

[20] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In *Proc. of MSR*, 2015.

[21] S. Levin and A. Yehudai. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proc. of PROMISE*, pages 97–106, 2017.

[22] I.-H. Lin and D. A. Gustafson. Classifying software maintenance. In *Proc. of CSM*, 1988.

[23] D. Liparas, Y. HaCohen-Kerner, A. Moumtzidou, S. Vrochidis, and I. Kompatsiaris. News articles classification using random forests and weighted multimodal features. In *Proc. of IRFC*, 2014.

[24] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *Proc. of ICSE*, 2015.

[25] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. of IWPC*, 1998.

[26] R. Messina and J. Louradour. Segmentation-free handwritten chinese text recognition with lstm-rnn. In *Proc. of ICDAR*, 2015.

[27] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv*, 2013.

[28] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar. A tactic-centric approach for automating traceability of quality concerns. In *Proc. of ICSE*, 2012.

[29] A. K. Mondal, B. Roy, and K. A. Schneider. An exploratory study on automatic architectural change analysis using natural language processing techniques. In *Proc. of SCAM*, 2019.

[30] T. Nakamura and V. R. Basili. Metrics of software architecture changes based on structural distance. In *Proc. of METRICS*, 2005.

[31] K. Oskina. Text classification in the domain of applied linguistics as part of a pre-editing module for machine translation. In *SPECOM*, 2016.

[32] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman. Are developers aware of the architectural impact of their changes? In *Proc. of ASE*, 2017.

[33] M. M. Rahman and C. K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proc. of ESEC/FSE*, 2018.

[34] D. Ramage, D. Hall, R. Nallapati, and C. D. Manning. Labeled lda: A supervised topic model for credit attribution in multi-labeled corpora. In *Proc. of EMNLP*, 2009.

[35] M. Soliman, A. R. Salama, M. Galster, O. Zimmermann, and M. Riebisch. Improving the search for architecture knowledge in online developer communities. In *Proc. of ICSA*, 2018.

[36] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

[37] E. B. Swanson. The dimensions of maintenance. In *Proc. of ICSE*, 1976.

[38] M. Wang, Z. Lin, Y. Zou, and B. Xie. Cora: decomposing and describing tangled code changes for reviewer. In *ASE*, 2019.

[39] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Proc. of IWPC*, 2004.

[40] B. J. Williams and J. C. Carver. Characterizing software architecture changes: A systematic review. *IST*, 2010.

[41] K. Yamauchi, J. Yang, K. Hotta, Y. Higo, and S. Kusumoto. Clustering commits for understanding the intents of implementation. In *Proc. of ICSME*, 2014.

[42] M. Yan, Y. Fu, X. Zhang, D. Yang, L. Xu, and J. D. Kymer. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *JSS*, 2016.