

Received March 27, 2022, accepted April 11, 2022, date of publication April 13, 2022, date of current version April 20, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3167269

Improved Prioritization of Software Development Demands in Turkish With Deep Learning-Based NLP

VOLKAN TUNALI , (Member, IEEE)

Department of Software Engineering, Faculty of Engineering and Natural Sciences, Maltepe University, 34857 Istanbul, Turkey

e-mail: volkan.tunali@gmail.com


ABSTRACT Management of software development demands including bug or defect fixes and new feature or change requests is a crucial part of software maintenance. Failure to prioritize demands correctly might result in inefficient planning and use of resources as well as user or customer dissatisfaction. In order to overcome the difficulty and inefficiency of manual processing, many automated prioritization approaches were proposed in the literature. However, existing body of research generally focused on bug report repositories of open-source software, where textual bug descriptions are in English. Additionally, they proposed solutions to the problem using mostly classical text mining methods and machine learning (ML) algorithms. In this study, we first introduce a demand prioritization dataset in Turkish, which is composed of manually labeled demand records taken from the demand management system of a private insurance company in Turkey. Second, we propose several deep learning (DL) architectures to improve software development demand prioritization. Through an extensive experimentation, we compared the effectiveness of our DL architectures trained with several combinations of different optimizers and activation functions in order to reveal the best combination for demand prioritization in Turkish. We empirically show that DL models can achieve much higher accuracy than classical ML models even with a small amount of training data.

INDEX TERMS Software engineering, demand prioritization, bug prioritization, machine learning, text classification, deep learning.

I. INTRODUCTION

Software development is a very sophisticated endeavor that aims to address real-world problems which are inherently complex and difficult to solve [1]. Other factors like development tools and techniques, developer competency, and schedule pressure add an extra layer of complexity and difficulty to development processes. As a result of these complexities and difficulties, several types of defects are inevitably introduced into software in any stage of software development life cycle (SDLC) due to several reasons like inadequately collected and analyzed requirements, misunderstanding of requirements, bad architecture and design, bad coding practices, and so on [2]. In addition, poor testing approaches and quality assurance strategies allow for releasing software with unidentified defects. Most of the defects in released software can be identified and reported

by end-users, and some by other developers and technical support personnel. Regardless of the source, these defects are usually submitted to and managed in a kind of issue tracking system such as Jira, Bugzilla, or other proprietary or open-source systems. In addition to defect or bug tracking, these issue tracking systems are also used for tracking demands of small-scale new feature developments or change requests by end-users. For example, an end-user may request the display of an additional information regarding a customer on a report page, which was never on the initial requirements specification, and thus, its absence is not a kind of defect left by the development team. Therefore, in this paper we develop and use a unified terminology such that a software development demand is either a kind of defect- or bug-fix, or a new feature development, or a change request. In this sense, we call any kind of issue or bug tracking and management system a Demand Management System (DMS). A DMS, in this perspective, allows users and development teams to report, describe, prioritize, track, and resolve software

The associate editor coordinating the review of this manuscript and approving it for publication was Roberto Nardone .

development demands in a systematic manner via appropriate user-interfaces and work-flows [3].

A demand record in a DMS usually contains fields such as demand-id, date of submission, product, component, version, operating system, title, short description, long description as well as severity and priority. Severity means how serious is the issue or bug if it is a bug report. Priority, on the other hand, indicates how important a demand is for the software system to be resolved earlier. Some systems use categorical values for priority field as P1 to P5, P1 being the highest priority. Some employs a similar but simpler approach with categorical values of High, Medium, and Low priority.

It is our observation in the software industry that end-users usually submit their development demands with high priority even it is medium or low in order to make them scheduled and resolved earlier. We also observe that some end-users leave priority field blank. In either case, new demands must be carefully examined and evaluated by someone called triager to assign correct and meaningful priorities to them [3]. Another role of the triager is to assign the demand appropriate development person or team according to demand content. This type of triaging process assumes that the triager has the broad knowledge of the software system and the development team like a project manager. This may not always be the case especially for large and complex projects. Additionally, this is a very labor-intensive task where the triager has to meticulously examine large batches of demands periodically, and her judgment of priority may become biased or she may fail to assign appropriate priorities. It is of very much importance to prioritize demands correctly in order to effectively schedule their fixing and resolution and to allocate correct resources to them. Failure to prioritize them correctly might result in inefficient planning and use of resources as well as user or customer dissatisfaction.

In order to overcome the difficulty and inefficiency of manual demand triaging, many automated prioritization approaches have been proposed in the literature. We provide a concise yet comprehensive overview of these approaches in Section II. When we survey the literature, previous body of research is generally focused on bug prioritization on bug repositories in English language. In this research, however, we prioritize not only bugs but any kind of software development demands in the DMS of a private insurance company in Turkey. The company has a software development department and develops its own software with an in-house development strategy. A great majority of the development demands are submitted to the DMS by the internal end-users of the company. Hence, demand records are written in Turkish language, which imposes several additional difficulties to process due to the agglutinative nature of the language.

In our previous study, we tested and compared several classical Machine Learning (ML) models on the demand records taken from the DMS, applying several text pre-processing operations on the demand texts [4]. In that study, we achieved F-Score values of 0.54 and 0.53 at best

with Support Vector Machines and Naive Bayes algorithms, respectively. Over the past decade, we have been witnessing a paradigm shift in computer vision thanks to advances in Deep Neural Networks (DNN) and Deep Learning (DL). As well as computer vision, developments in Natural Language Processing (NLP) have gained great momentum with the introduction of DL architectures for sequence processing like RNN and LSTM, and with the release of large pretrained language models based on the transformer architecture like BERT (Bidirectional Encoder Representations from Transformers) and GPTs (Generative Pretrained Transformer) [5]. Therefore, in this current study, our main motivation was to apply state-of-the-art concepts and techniques in NLP to the very same problem and to propose effective DL architectures to improve demand prioritization on Turkish demand records. The main contributions and novelty of our work can be summarized as follows:

- 1) We introduce an original demand prioritization dataset in Turkish language, taken from the DMS of a private insurance company in Turkey. This dataset can serve as a reference to further similar studies in software engineering domain. In addition, this dataset can be useful to researchers who conduct general-purpose text classification research in a low-resource language like Turkish.
- 2) We propose several deep learning architectures to greatly improve software development demand prioritization over our previous research with the same dataset.
- 3) Through an extensive experimentation, we compare the effectiveness of our deep learning architectures trained with several combinations of different optimizers and activation functions in order to reveal the best combination for demand prioritization in Turkish.
- 4) We empirically show that deep learning models can achieve much higher accuracy than classical ML models even with a small amount of training data.

The rest of this paper is organized as follows. In Section II, we provide an essential background on prioritization of software development demands. In Section III, we establish a technical background on the morphology of Turkish language and Deep Learning methods. In Section IV, we present the materials and methods used in the study. We present our experimental results and discussion in Section V. Finally, in Section VI, we conclude the paper.

II. RELATED WORKS

Existing research usually considered the management of software development demands as a bug or defect management problem from various perspectives such as automating bug assignments, detecting duplicate or similar bugs, predicting bug fixing time, categorizing bugs, predicting severity and priority of bugs [6]. Besides, they usually approached the problem with supervised and unsupervised ML techniques, using mostly classical algorithms like K-Nearest Neighbor (KNN), Naive Bayes (NB), Support Vector Machines (SVM), and K-Means (KM). While some of them used textual

features extracted from text fields like title and description, some other utilized the categorical features in issue records as well. Another common pattern among previous studies is that they usually used bug tracking databases of open source software, taking advantage of the availability of such open-source repositories like Eclipse, GNOME, and OpenOffice. To the best of our knowledge, none of the previous studies apart from our previous study worked with a dataset in Turkish language. In this section, we provide a concise yet comprehensive overview of the related research under three categories as bug severity prediction, bug priority prediction, and bug fix time prediction, also summarizing them in Table 1.

A. RESEARCH ON BUG SEVERITY PREDICTION

Bug severity is defined as the degree of impact that a bug causes on the expected operation of a software component or a whole system [6]. Bug severity is an important measure in deciding when to fix a bug once it is reported. Depending on the bug tracking system, different levels of bug severities are employed such as from 1 to 5 or just severe and non-severe. In this subsection, we present important research on bug severity prediction.

Menzies and Marcus [8] proposed a system named SEVERIS to predict severity of issues identified during tests. They developed and tested the system on NASA datasets by taking into consideration textual descriptions of the issues. Applying common text mining operations and employing a rule learner, they achieved considerable accuracy.

Lamkanfi *et al.* [11] employed NB to predict the severity of bugs in categories as severe and non-severe. Their approach was based on textual features extracted bug descriptions of bug repositories of three open-source software, namely, Mozilla, Eclipse, and GNOME. In their follow-up study with the same methodology as the previous one, they compared NB, NBM (NB Multinomial), KNN, and SVM algorithms on bug reports of Eclipse and GNOME projects, concluding that NBM was best suited for the task with its speed and accuracy [13].

Chaturvedi and Singh [6] compared several ML algorithms including NB, KNN, SVM, NBM, DT, and RIPPER (Repeated Incremental Pruning to Produce Error Reduction) to predict severity of bugs in five levels from 1 to 5 on NASA datasets. They utilized text mining operations to extract features to feed the ML models, and conducted extensive experiments with different number of terms considered from textual descriptions of bugs.

In a study by Tian *et al.* [15], a bug severity prediction model was proposed to utilize a well-known text similarity function BM25. They used this function in a KNN search to obtain a severity prediction from the most similar records of the bug report. The proposed system was developed and tested on bug reports of OpenOffice, Mozilla, and Eclipse, and it was shown to produce comparable performance against the system proposed by Menzies and Marcus [8].

A novel approach to bug triage and bug severity prediction was proposed by Yang *et al.* [18]. They first performed topic modeling based on several features of bug report data. Then, they utilized KNN with KL divergence instead of Cosine or BM25 similarity indices over the topics to make a prediction about the severity of a new bug report. They showed the effectiveness of their approach on bug reports of Eclipse, Mozilla, and Netbeans. They extended this study by proposing a new similarity metric to find the similarity between a new bug report and historical data [20]. In addition, they included bug reports of GCC and OpenOffice to their experimental evaluations.

Sharma *et al.* [19] proposed a method based on feature selection from the document-term matrix generated from a collection of textual bug descriptions of Eclipse. Then, using only top 125 important terms, they applied and compared NBM and KNN to categorize severity of bugs in two categories as severe and non-severe.

B. RESEARCH ON BUG PRIORITY PREDICTION

Bug priority indicates how important a bug is for the software system to be fixed earlier. Priorities are usually managed as categorical values from P1 to P5, P1 being the highest priority. It is also very common to use a similar but simpler approach with categorical values of High, Medium, and Low priority. In this subsection, we present important research on bug priority prediction.

In the earliest study we consider by Podgurski *et al.* [7], an automated system for classifying software failure reports was proposed in order to support the priority prediction. The system was principally based on clustering and multivariate visualization. Quite interestingly, rather than using failure reports from real users, they executed the compilers GCC for C language, Jikes and javac for Java language to produce failures on self-validating tests of the respective compilers. Then, they analyzed the cluster contents for validating the effectiveness of the system.

In a study by Yu *et al.* [9], an ANN model was proposed to predict four levels of defect priorities, P1 to P4. The proposed neural network was trained on several categorical features from the defect reports of RIS 2.0 software such as milestone, category, module, main workflow, function, integration, frequency, severity, and tester.

Kanwal and Maqbool [10] applied a classification-based approach on the bug repository of Eclipse project. They utilized SVM for a five class classification for priority classes P1 through P5 using both categorical and textual features. They extended the study in a follow-up work by comparing SVM to NB [2]. They also proposed two new evaluation metrics based on precision and recall, namely, Nearest False Negatives (NFN) and Nearest False Positives (NFP), in order to better evaluate bug priority recommendation.

In order to build a predictive model for bug prioritization, Alenezi and Banitaan [16] investigated the effectiveness of NB, DT, and Random Forest (RF) algorithms with different feature combinations on bug tracking data of open-source

TABLE 1. Summary of the previous studies in chronological order.

Reference	Proposed Technique	Dataset	Prediction	Language	Feature
Podgurski <i>et al.</i> (2003) [7]	k-medoids, hier. MDS	GCC, Jikes, javac	Priority	EN	CT
Menzies & Marcus (2008) [8]	rule learner	NASA	Severity	EN	TX
Yu <i>et al.</i> (2010) [9]	ANN, NB	RIS 2.0	Priority	EN	CT
Kanwal & Maqbool (2010) [10]	SVM	Eclipse	Priority	EN	CT, TX
Lamkanfi <i>et al.</i> (2010) [11]	NB	Eclipse, GNOME, Mozilla	Severity	EN	TX
Giger <i>et al.</i> (2010) [12]	DT	Eclipse, GNOME, Mozilla	Fix Time	EN	CT
Lamkanfi <i>et al.</i> (2011) [13]	NB, NBM, KNN, SVM	Eclipse, GNOME	Severity	EN	TX
Kanwal & Maqbool (2012) [2]	SVM, NB	Eclipse	Priority	EN	CT, TX
Abdelmoez <i>et al.</i> (2012) [14]	NB	Eclipse, GNOME, Mozilla	Fix Time	EN	CT
Chaturvedi & Singh (2012) [6]	NB, KNN, SVM, NBM, DT	NASA	Severity	EN	TX
Tian <i>et al.</i> (2012) [15]	KNN with BM25	Eclipse, Mozilla, OpenOffice	Severity	EN	CT, TX
Alenezi & Banitaan (2013) [16]	DT, RF, NB	Eclipse, Firefox	Priority	EN	CT, TX
Tian <i>et al.</i> (2013) [17]	SVM, RIPPER, NBM	Eclipse	Priority	EN	CT, TX
Yang <i>et al.</i> (2014) [18]	Topic Model, KNN	Eclipse, Mozilla, Netbeans	Severity	EN	CT, TX
Sharma <i>et al.</i> (2015) [19]	NBM, KNN	Eclipse	Severity	EN	TX
Zhang <i>et al.</i> (2016) [20]	Topic Model, KNN	Eclipse, Mozilla, GCC, Netbeans, OpenOffice	Severity	EN	CT, TX
Choudhary & Singh (2017) [21]	ANN, NB	Eclipse	Priority	EN	TX
Kumari & Singh (2018) [22]	NB, DL	OpenOffice	Priority	EN	CT, TX
Tekin & Tunalı (2019) [4]	NB, NBM, SVM, RF, ROF	Inhouse Insurance	Priority	TR	TX
Umer <i>et al.</i> (2020) [23]	CNN	Eclipse	Priority	EN	TX

* CT: Categorical, TX: Textual, EN: English, TR: Turkish

projects Eclipse and Firefox. Instead of using the original priority labels P1 through P5, they mapped them to three-level priorities, where High corresponds to P1 and P2, Medium corresponds to P3, and Low corresponds to P4 and P5. They obtained best results with tree-based algorithms rather than NB.

Tian *et al.* [17] proposed a framework named DRONE that utilized linear regression with thresholding approach to handle imbalanced nature of bug report data. They compared their method to several ML algorithms including SVM, RIPPER, and NBM by exploiting both textual and categorical features of bug repository data of Eclipse.

In a study by Choudhary and Singh [21], they proposed an ANN model trained on textual features extracted via regular text preprocessing operations from bug descriptions of Eclipse. They provided a comparison between ANN and NB on the prediction of priority labels from P1 to P5.

Kumari and Singh [22] compared NB and DL models for bug priority prediction in a five-level priority setting P1 to P5. First, they calculated entropy of top 200 terms of textual descriptions of bug reports of OpenOffice. Then, using it as an individual feature, they combined it with other categorical features of bug reports to train NB and DL models. They showed DL with entropy outperformed NB with entropy.

In our previous study, we compared several classical ML models including NB, NBM, SVM, RF, and Rotation Forest (ROF) on the demand records taken from the DMS of a Turkey based insurance company [4]. We applied several text preprocessing operations on textual descriptions of the demands to generate matrices to feed the ML algorithms. In that study, we achieved F-Score values of 0.54 and 0.53 at best with SVM and NB, respectively. As far as we know, this is the only study that considered demand records in Turkish language.

In the most recent study we review in this section, Umer *et al.* [23] proposed a Convolutional Neural Network

(CNN) based approach to predicting priority of bug reports in Eclipse environment. Applying NLP techniques to textual data of bug reports to obtain semantic word vectors, they trained a CNN model to perform priority classification P1 through P5.

C. RESEARCH ON BUG FIX TIME PREDICTION

In a limited number of previous studies, whether a bug needs to be fixed earlier or it can be delayed to a later time was considered as a two-class classification problem. They artificially created these two classes from the time of report and time of fix of bug reports based on some statistics. In this sense, bug fix time prediction can be regarded as a special form of bug priority prediction.

Open-source software projects provide a convenient way to collect data related to software engineering practices. In the study by Giger *et al.* [12], Mozilla, Eclipse, and GNOME were again the subject. Their goal was to decide whether a bug should be fixed as soon as possible or it could be resolved within a larger period. Therefore, they used Decision Tree (DT) models trained on categorical features to make a binary prediction between fast and slow classes, showing the effectiveness of the approach over random classification.

A very similar study was conducted by Abdelmoez *et al.* [14], where they also used data from Mozilla, Eclipse, and GNOME. They trained NB models using the categorical features to prioritize which bugs to begin fixing and which to fix later.

D. EVALUATION OF PREVIOUS RESEARCH

We see that previous studies heavily rely on the availability of public bug repositories of only several open-source software. Therefore, these studies are centered around a limited number of different datasets, and they lack diversity in this sense. Another limitation of them is that they mostly utilize classical and popular ML algorithms with conventional text processing

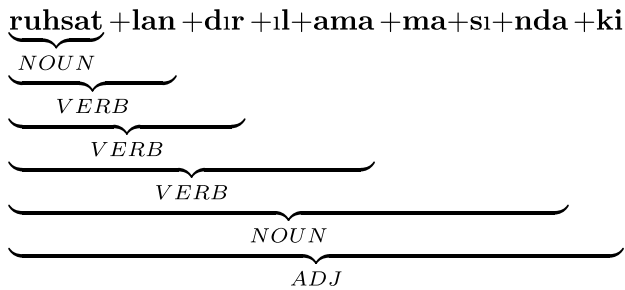


FIGURE 1. Illustration of multiple derivations in a Turkish word [24] (Reprinted with permission).

approaches. Furthermore, there is no previous study that considers demand records particularly in Turkish language (other than ours).

In our current study, we significantly differ from most of the existing studies in two important ways. First, rather than using public bug repositories of open-source software, we introduce and use a novel demand prioritization dataset in Turkish, which is a low-resource language for NLP studies. Second, unlike most of the previous studies, we perform model development by using modern and advanced deep learning NLP approaches to improve demand prioritization.

III. TECHNICAL BACKGROUND

In this section, we provide a technical background on the complex morphology of Turkish language and the Deep Learning methods used in our research.

A. COMPLEX MORPHOLOGY OF TURKISH

Belonging to the Turkic family of Altaic languages, Turkish language has certain challenging characteristics for natural language processing. It is an agglutinative language where words are formed by affixations of multiple suffixes to root words. That is, words in Turkish can take many inflexional and derivational suffixes to such an extent that a single word can simply correspond to a full sentence in English. For example, the word “yap+abil+ecek+si+n” in Turkish means “you will be able to do (it).” Similarly, it is not uncommon to see multiple derivations in a word. A very nice example is provided by Oflazer in Fig. 1, meaning “related to (something) not being able to acquire certification.” In this example, the root word is “ruhsat” (certification) and it becomes a noun modifier after five derivations [24].

B. DEEP LEARNING METHODS

1) DEEP LEARNING

Deep Learning (DL) is a subset of machine learning where algorithms are inspired by the connectivity patterns of human brain called Artificial Neural Network (ANN) [25]. There are different deep learning architectures that have been applied to fields such as computer vision, speech recognition, natural language processing, and so on. Convolutional Neural Network (CNN) is an architecture that has been used for image feature extraction. One other architecture is

Recurrent Neural Network (RNN) which has connections between its layers as a form of directed graph, allowing the information carried in layers to remember. Long Short-Term Memory (LSTM) is special type of RNN, capable of remembering long-term dependencies [26].

RNN and LSTM are more suitable for sequential data such as text, time series, financial data, speech, audio, video, and so on. Therefore, they are commonly used for tasks such as natural language processing and time series processing. CNN, on the other hand, is best suitable to work with spatial structures like images.

2) CONVOLUTIONAL NEURAL NETWORKS

Most crucial components of this architecture are the convolution and pooling operations. Convolution represents the direct application of any mathematical filter to a given input that results in an activation. Repeating the same process with the same filter results in a map of activations that is called feature maps, indicating the locations and strength of a detected characteristic of the given input. Pooling is used for reducing the spatial dimensions of mapped feature maps. Pooling layer operates on each feature map independently [26].

3) RECURRENT NEURAL NETWORKS

A typical ANN contains input layer followed by a number of hidden layers. This kind of network is considered as memoryless. In a Recurrent Neural Network (RNN) there is a loop at the hidden layers. In that manner, the state of the hidden layers depends not only on the input but also on the state of the hidden layer at the previous time step.

RNNs have difficulties in learning long-term dependencies. LSTM-based models represent an extension to RNNs, which are able to address the vanishing gradient problem in a remarkably precise way. LSTM models essentially extend the memory of RNNs to enable them to keep and learn long-term dependencies of inputs. Bidirectional LSTMs (BiLSTM) are an extension of LSTM layers, that is, two LSTMs are applied to input data from both directions, forward and backward.

4) TRANSFER LEARNING

Transfer learning is a machine learning technique to reuse pretrained models for new objectives. That is, layers and weights of a pretrained model are used as a starting point in model creation [26]. Transfer learning is usually used when there are insufficient number of samples to train a model from scratch. In this way, we make use of the information readily available in the parameters of a previously trained model. In addition, training of a new model takes considerably less amount of time.

IV. MATERIAL AND METHODS

A. DATASET

In this study, the dataset is composed of demand records taken from the database of a DMS used for tracking and resolving the demands of internal end-users by a software development

TABLE 2. Sample records from the dataset. Approximate English translations are provided in brackets beneath the Turkish phrases.

Title	Description	Priority
KATASTROFİK RISKLER (CATASTROPHICAL RISKS)	mali işler liste ve sorgular hazine raporları hazine raporları katastrofik riskler bireysel raporu uzun süre çalışıyor görünüyor ama raporu ekrana ve veya excele aktarmıyor (financial affairs lists and queries, treasury reports, catastrophic individual risks report run for long time but the report does not appear on the screen nor it is exported to excel)	Düşük (Low)
referans admin (reference admin)	merhaba, kullanıcısı referans admin ekranında görünmüyor. acil kontrolünüzü rica ederiz hello, user is not shown in reference admin screen. we ask for your urgent check	Orta (Medium)
Özel Güvenlik Zeyil basım hatası çok acil (Print error in Private Security Addendum very urgent)	y z no lu Özel Güvenlik poliçesi zeyli onaylanmasına rağmen basım alınamıyor. Zorunlu poliçe olması sebebiyle çok acil yardım rica ederiz. Gün içinde zeyli emniyete vermek zorundalar (Private Security policy addendum was approved but cannot be printed. Since it is a mandatory policy, we ask for your urgent help. They have to deliver the addendum to police department)	Yüksek (High)

department of a private insurance company. From these records, only the demand title, description, and priority class fields were used, and the other fields were completely ignored. Confidential information such as Turkish Identity Number, Tax Number, and various document numbers in the fields were cleared manually. The title and description fields of the records were combined and each record was saved as a separate text file in a separate folder with respect to priority classes.

Sample records from the dataset are shown in Table 2. Approximate English translations are provided in brackets beneath the Turkish phrases in the table. As seen in the sample records, there is no standard in the writing of text fields and many typographical errors are made by users. This is a major challenge that is often encountered and must be overcome in all text processing and NLP applications.

The records in the dataset had a skewed distribution over three priority classes: low, medium, and high. There was a total of 344 records in the dataset, 43 of which are low, 150 medium, and 151 high priority. In a classification study, a balanced distribution of the records in the dataset is desired because predictions produced by a model trained on imbalanced classes tend towards majority classes, which results in a decrease in correct prediction rate of the samples from minority classes. In order to increase the number of records in low priority class, we applied two different augmentation techniques together. First one is based on Word2Vec similarity [27] and the second one is based on an approach known as back-translation [28]. We applied augmentation to only randomly-picked 38 of the 43 records to generate additional records. For each of the 38 records, we generated three additional records by randomly employing one of the approaches each time. In the Word2Vec-based approach, we trained Turkish word embeddings of length 512 using Gensim [29] on Turkish Wikidump. Using these embeddings, for each word in a demand record, we obtained the most similar word and if the similarity score was higher than 0.5, we replaced the original word with the most similar one and left as is otherwise. In the back-translation-based approach, on the other hand, we first translated the record from Turkish to

English, and then from English to Turkish back using Google Translate service. In the end, we had 157 records of low priority after augmentation including the other five records that were not involved in the augmentation. We manually checked the original records and their augmented versions to confirm that we achieved the expected diversity. Finally, we had 458 demand records in the dataset with 157, 150, and 151 records in each priority class. In Table 3, we present three sample records and their augmented versions from the low priority class. When we inspect the augmented versions, we clearly see the effect of our augmentation process as some words are replaced by their different derivations or semantically close words. For example, the original word “çalışıyor” (meaning “[it is] working”) in the first sample is replaced with “çalışmaktadır” (meaning “[it is] working”) and “çalışıyorlar” (meaning “[they are] working”).

As one of the main contributions of this research, we released this dataset as a reference to further similar studies in software engineering domain. In addition, it can be useful to researchers who conduct general-purpose text classification research in a low-resource language like Turkish. We made it publicly available at <https://github.com/volkantunali/text-datasets>.

B. TEXT PREPROCESSING

Textual data written in natural language is considered unstructured because it lacks a pre-defined data model or data structure. In order to use textual data as an input to a ML method, it must be transformed into a structured form which is usually a tabular structure like a matrix. The end-to-end process to transform textual data into a structured form is called text preprocessing [30]. In addition to this transformation, some text preprocessing steps are required to reduce the size of the vocabulary for computational efficiency while keeping the meaning of remaining words in the data. For instance, Stop-words Removal is used to remove the words that do not contribute to the actual meaning of a text. Moreover, another important text preprocessing technique is called Stemming where each word in a text is converted to its basic (root) form. These two text preprocessing techniques require different approaches for

TABLE 3. Sample records from the dataset after augmentation.

Original	Augmented v1	Augmented v2	Augmented v3
KATASTROFİK RİSKLER mali işler liste ve sorgular hazine raporları hazine raporları katastrok riskler bireysel raporu uzun süre çalışıyor görünüyor ama raporu ekrana ve veya excele aktarmıyor	katastrok riskler mali işler liste birçoğu sorgular evkaf raporları hazine raporunu katastrok riskler bireysel raporu uzun süre çalışmaktadır görünüyor ama raporunda ekrana ve veya excele aktarmıyor	katastrok riskler finansal işler listede bazıları araştırır evkaf raporları hazine raporunu katastrok riskler bireysel raporu uzun süre çalışmaktadır görünürler ama raporunda ekranda ve veya excele aktarmıyor	katastrok maliyetler mali işlerini listeler fakat sorgular umumiye raporları vergi raporları bradikardi riskler bireysel iddianamede uzun süreden çalışıyorlar görünüyor zira soruşturması ekrandan onların hatta excele aktarmıyor
TARİH FORMATINDA HATA BT SİSTEM LİSTE SORGULAR POLİÇE HASAR LİSTESİ YENİ alanında tarih aralıkları ile rapor alındığında exelde ki tarih formatlarında hata olmakta ve düzeltilememkte düzeltilmesi için desteğinizi rica ederiz	tarih formatında hata bt sistem liste sorgular poliçe hasar listesi öneriler alanında tarih aralığında ile rapor alındığında exelde ki tarih formatlarında hata olmakta ve düzeltilememkte düzeltilmesi için desteğinizi rica edeceğimiz	tarih dvd hata transmiyon sistem liste sorgular poliçe hasar listesine öneriler alanında tarih aralığını ile raporda alındığında exelde ki etnoloji formatlarında hata olmaktadır ve düzeltilememkte düzeltilmesi isteyenlerin desteğinizi tebrik edeceğimiz	tarih formatında hataların bt sistemlerdir chart sorgular poliçe hasara listesi önceleri alanlarında tarih aralığının yla öneriler alındığında exelde ki tarih formatları hatalar olabilecek bir düzeltilememkte uygulanması zorundadırlar desteğinizi rica ediliyorsa
EXGRATIA HAVUZ LİSTESİ ALINAMIYOR VE EXCELE AKTARILAMIYOR	exgratia havuz listesi alınamıyor böylece excele aktarılamıyor	exgratia havuzlar listesi alınamıyor böylece excele aktarılamıyor	exgratia gölet listelenmiştir alınamıyor ve excele aktarılamıyor

different languages. As Turkish is an agglutinative language where words are derived by appending suffixes as mentioned before, stemming can be more important for Turkish than it is for other languages [31]. Finally, Padding is used to create fixed-size sequences of words to generate the final structured data. These techniques are described in the following subsections in more detail.

1) STOP-WORDS REMOVAL

A classical and effective technique widely-used in NLP tasks is to eliminate the words that do not contribute to the actual meaning of a text known as stop-words removal. In this technique, words like *a*, *the*, and *with* are removed from the text based on a list of specific words called stop-words. Each language has its own list of stop-words, thus, we used a commonly-used list of Turkish stop-words. Unlike NLP studies in languages like English, we applied stop-words removal before and after stemming because after stemming of Turkish words, the remaining root word may also be a stop-word [30].

2) STEMMING

Stemming is a technique to obtain the root form of a word in order to map related words to the same stem. For example, words *writer*, *writing*, and *writes* reduces to the stem *write*. For stemming Turkish words, there are several approaches that can be used considering the agglutinative nature of Turkish language as well as previous Turkish NLP studies. The first stemming approach we used in this research is a dictionary-based stemmer from TRNLP library [32]. TRNLP stemmer tries to find possible roots of a word by searching a dictionary for the parts of the word produced by stripping off a letter from the end. Dictionary-based stemmers are known to produce very useful stems in practice. As the second stemming method, we used the affix-stripping stemmer that reaches the stem of a word by a morphological analyzer modeled by a finite state machine [33]. It makes use of the

rule-based structure of Turkish language and it works without using any lexicon. Finally, the third approach we used is a pseudo stemmer called fixed-prefix stemmer, which works just by taking the first n character of a word as the stem of the word. Even though this naive approach does not use any linguistic information, stems it produces are empirically known to be effective at NLP tasks on Turkish texts like clustering and classification when n is equal to 5 [34].

3) PADDING

In NLP applications, processed text sequences can have variable lengths. In our case, for example, one development demand can be written in a number of words while the other in fewer or more words. However, the common practice is to use word sequences of fixed-sizes in NLP implementations based on deep learning. Sequences shorter than this fixed-size are padded with a particular word index zero to differentiate from the real vocabulary word indices. Likewise, longer sequences are truncated to this size. We illustrate this padding technique on an example corpus in Fig. 2, where sequence length is four. In the example, the first sequence is shorter than four, thus, it is padded with trailing zeros to make its length four. The second one is longer than four, then, the tokens whose positional index is greater than four are simply removed. Finally, the third one is already of length four, therefore, it remains unchanged.

In this research, we followed the same approach for padding. While there is no mathematical procedure to determine effective sequence length, it is possible to employ some statistical modeling to predict an appropriate value. The technique we applied is as follows. Firstly, we calculated the lengths of the demands after stemming and stop-words removal. Then, we plotted the histogram of demand lengths with kernel density as seen in Fig. 3. Next, we approximated the distribution to a Gamma distribution with $\alpha = 2.18$ and $\beta = 10.09$. We picked Gamma rather than Gaussian distribution for approximation because Gamma distribution

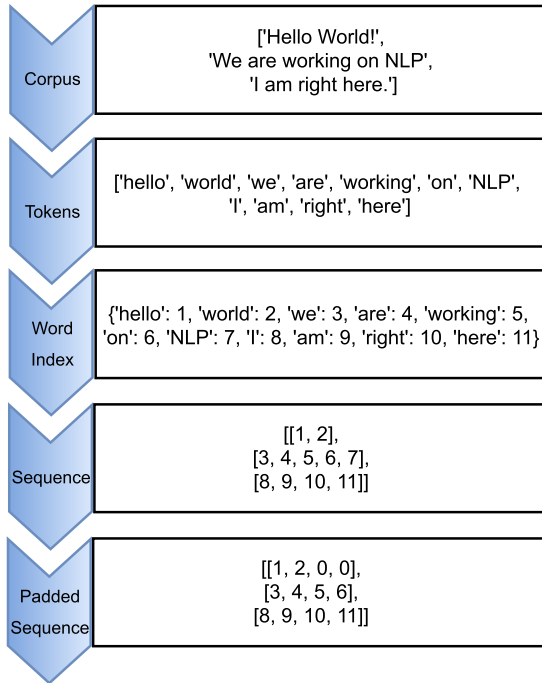


FIGURE 2. Illustration of padding on a sample corpus.

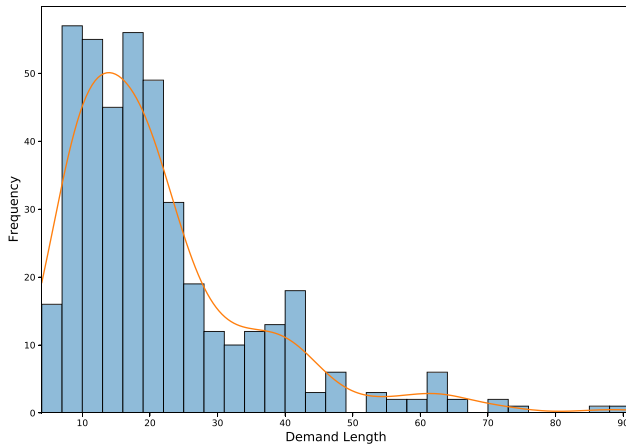


FIGURE 3. Distribution of demand lengths.

models variables that are always positive, being more appropriate than Gaussian to model sequence lengths in our problem. Therefore, we computed the mean of the approximated Gamma distribution as 22, which was our final sequence length.

C. PROPOSED DL ARCHITECTURES

One of the main contributions of this research was to design several DL architectures that could achieve considerable improvement over classical ML algorithms on an NLP task with limited number of training data points. The NLP task we dealt with was software development demand prioritization, and the number of available training records was 458, which is very small for successful model development in practice.

Therefore, we proposed four different DL architectures and compared their performances. We first began with a baseline ANN architecture where no feature extraction layer existed as in CNN and RNN architectures. Then, we proposed three enhanced architectures that contained combinations of convolution and BiLSTM components. The structure of the proposed DL architectures can be seen in Fig. 4. It should be noted that all of the proposed architectures, including the baseline ANN, began with a Word Embedding layer. Likewise, all of them ended with two fully connected layers with 128 and 3 neural units.

Word embeddings are a technique used in NLP to represent words as vectors in a continuous vector space. This representation allows words with similar meanings to be clustered together, and also allows for numerical calculations on words and word sequences. Unlike word embeddings that are based on traditional one-hot-encoding schemes like TF-IDF, it is found by training the neural network on a corpus of text data. We used 128-dimensional trainable embedding vectors in the embedding layers of our DL architectures.

1) BASELINE ANN

The possible simplest architecture for an NLP task could include fully connected layers right after the embedding layer. We stacked convolution and/or BiLSTM components over this baseline to obtain further improved architectures as explained in the following subsections.

2) CNN

What we proposed as a CNN was the simplest form of a CNN that could be designed. It simply consisted of a convolution and global max pooling layers to perform feature extraction from word embedding vectors. The convolution layer was a 1D-convolution with 32 filters and its kernel size was 7.

3) CONV+BiLSTM

The third architecture we proposed was an extension to CNN with an additional layer of BiLSTM between pooling and fully connected layers. In our architecture, number of LSTM units in BiLSTM layer was 128.

4) BiLSTM

Our BiLSTM architecture was a simple architectural improvement over baseline ANN obtained by inserting a BiLSTM layer with 128 units between word embedding and fully connected layers.

5) TRANSFER LEARNING WITH TRANSFORMERS

In addition to DL architectures with relatively simpler structures that we proposed, we tested the effectiveness of pretrained language models on prioritization of software development demands. These were very large models based on transformer architecture, which were trained on lots of text documents in Turkish. They were readily available to fine-tune on our dataset via transfer learning. Since the main purpose of our study was not to provide an exhaustive

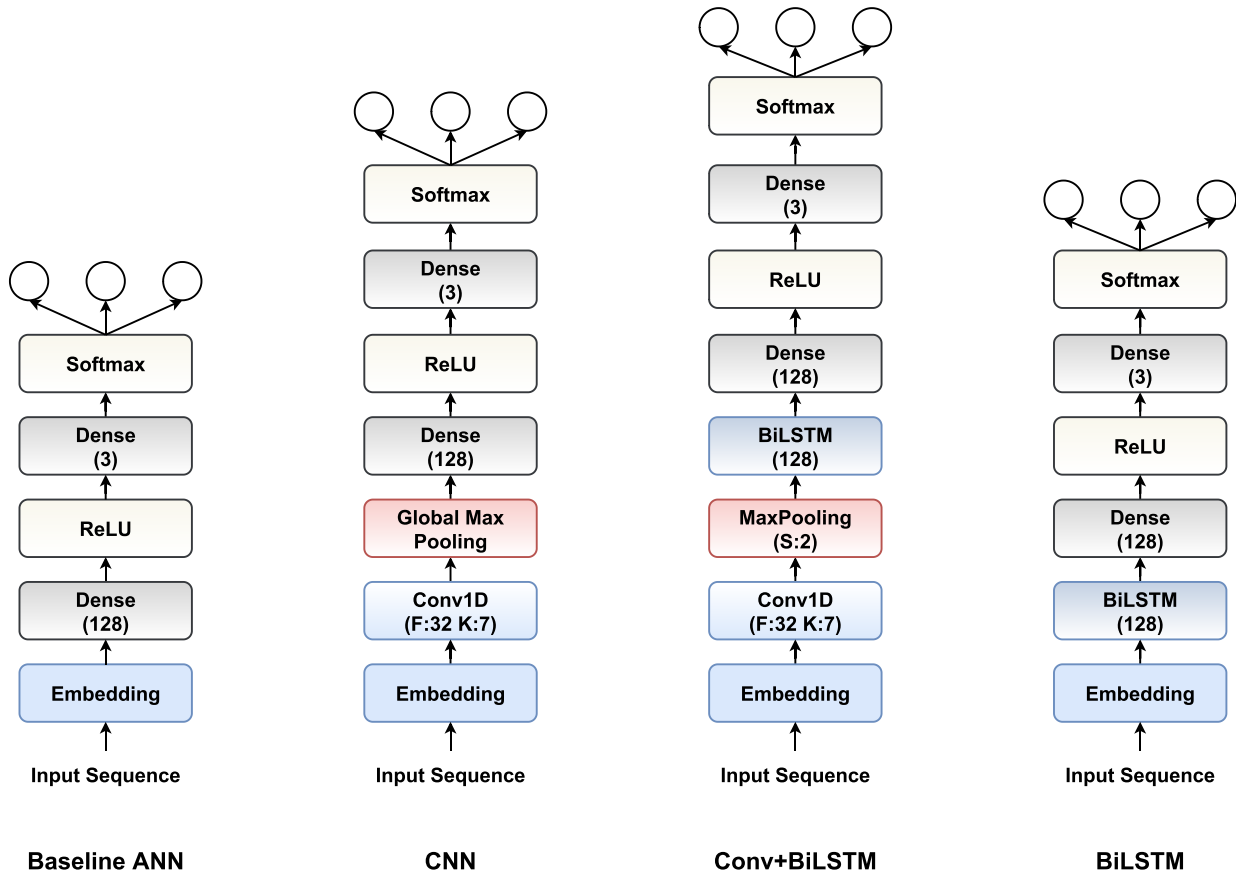


FIGURE 4. Overview of proposed DL architectures.

comparison of pretrained transformer models available for Turkish, we limited our study to evaluating only the most common and significant ones in Turkish NLP literature.

The first pretrained transformer model was **bert-base-turkish-128k-cased**, which was a BERT-based model [35]. It was trained on a large filtered and sentence-segmented corpus composed of several corpora such as the Turkish OSCAR corpus [36], a Wikipedia dump, and various OPUS corpora [37]. It is a cased model with a vocabulary size of 128k.

distilbert-base-turkish-cased was the second pretrained model based on DistilBERT architecture that was proposed to be a smaller model than BERT yet retaining almost the same performance as BERT [38]. It was also trained on the same corpus as **bert-base-turkish-128k-cased** model, and it is a cased model with a vocabulary size of 32k.

The other two pretrained transformer models were based on ELECTRA architecture proposed to require relatively less computation power to train large models [39]. First, **electra-base-turkish-cased-discriminator** was also trained on the very same corpus as the previous models, and it is a cased model with a vocabulary size of 32k. Second, **electra-small-turkish-cased-discriminator** was a different model trained on a different corpus composed of Turkish texts collected from online blogs, free e-books, newspapers, common crawl

TABLE 4. Optimizer parameters used for training the models.

Optimizer	Learning Rate	β_1	β_2	amsgrad
Adam	0.001	0.9	0.999	false
Rectified Adam	0.001	0.9	0.999	false
Yogi	0.01	0.9	0.999	NA
NovoGrad	0.001	0.9	0.999	false

corpora, Twitter, Wikipedia, and so on [40]. It is also a cased model with a vocabulary size of 32k.

D. MODEL TRAINING

We used GeForce GTX 1660 Ti and Tesla K80 GPUs for training all the models. Models were generated using TensorFlow 2.6.0 [41] and Keras 2.6.0 [42] in Python 3.8.8 environment.

In this research, rather than relying on the effectiveness of a single optimizer like Adam [43], we wanted to measure the effectiveness of other optimizer such as Rectified Adam (RAdam) [44], Yogi [45], and NovoGrad [46]. Our motivation was that depending on the task and training data, performance of the trained neural networks can vary greatly according to choice of the optimizer, as supported by the experimental results. In Table 4, we present the parameter values of the optimizer that we used for training the models.

E. MODEL EVALUATION METRICS

In order to measure the classification performance of the selected models, we used Accuracy, Precision, Recall, F-Score, and Cohen's Kappa Coefficient metrics. For each of these metrics, the higher the metric value, the higher the performance of a classifier is. We used scikit-learn [47] to generate desired metrics.

When we test a classifier, we obtain four different counts as True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN). Using these counts, it is possible to compute the above metrics as given in (1), (2), (3), (4), and (5), respectively.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

$$F - Score = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (4)$$

$$Kappa = \frac{Accuracy - P_e}{1 - P_e} \quad (5)$$

where

$$P_e = \frac{(TP+FP) \times (TP+FN) + (TN+FP) \times (TN+FN)}{(TP+TN+FP+FN)^2} \quad (6)$$

V. EXPERIMENTAL RESULTS AND DISCUSSION

We carried out extensive experiments for the DL architectures that we proposed and the pretrained transformer architectures on our dataset. For each architecture excluding the transformers, we trained and tested with all possible combinations of stemmers, activation functions, and optimizers with 5-fold cross validation. For the transformer architectures, on the other hand, we did not employ any specific stemmer, activation function, or optimizer because we transfer their existing knowledge to our specific task without manipulating their structure in any specific way. We collected the performance evaluation metrics of each model in terms of Accuracy, Precision, Recall, F-Score, and Cohen's Kappa Coefficient. We present experimental results of the architectures in the following subsections.

A. BASELINE ANN MODELS

We present the experimental results of Baseline ANN architecture in Table 5 and Fig. 5. Even with this simple architecture, we obtained promising results with the dictionary-based stemmer. It clearly outperformed the affix-stripping and fixed-prefix stemmers with all combinations of activation functions and optimizers.

Within the results of dictionary-based stemmer, Swish activation function and Adam optimizer pair achieved the best results (F1 = 0.688), closely followed by ReLU+Adam and ReLU+RAdam pairs (F1 = 0.670 and F1 = 0.667). When

it comes to other stemmers, there was no clear difference between them.

B. CNN MODELS

Unlike the Baseline ANN, CNN architecture attained consistently high performance with all combinations of stemmers, activation functions, and optimizers, having F-Scores between 0.649 and 0.737. We present the experimental results in Table 6 and Fig. 6. Although there was no clear winner among the stemmers, we obtained the best performance with the dictionary-based one with ReLU+Adam pair (F1 = 0.737). A very close performance was achieved by the affix-stripping+Swish+Adam combination (F1 = 0.718). We also observe that Adam was consistently the best optimizer with any stemmer and activation function when used in our CNN architecture.

C. CONV+BiLSTM MODELS

In our Conv+BiLSTM architecture, the fixed-prefix stemmer was always behind the dictionary-based and affix-stripping ones in terms of model performance. Apart from improving the overall performance, the addition of BiLSTM layer to the convolution layer had the opposite effect. The best F-Score achieved by this architecture was 0.650 with the affix-stripping+Mish+Adam combination. It was closely followed by affix-stripping+ReLU+Adam and dictionary-based+ReLU+RAdam combinations. These results are presented in Table 7 and Fig. 7.

D. BiLSTM MODELS

The architecture with BiLSTM without a preceding convolution layer produced better results especially with the affix-stripping stemmer. It achieved the highest F-Score of 0.710 with ReLU+Adam combination. Swish+Adam and ReLU+RAdam were the second and third best performers (F1 = 0.664 and F1 = 0.661). We give the experimental results in Table 8 and Fig. 8.

E. PRETRAINED TRANSFORMERS MODELS

Among the pretrained transformer models, **distilbert-base-turkish-cased** was the best model with the highest F-Score of 0.780. It was also better than the models that we proposed. The other BERT-based model, named **bert-base-turkish-128k-cased**, achieved the second best F-Score of 0.746 among all models including our DL architectures. The other two models based on ELECTRA architecture did not exhibit as high performance as BERT-based models. In fact, our CNN models did better than them in general. Experimental results can be seen in Table 9 and Fig. 9.

F. COMPARISON OF ARCHITECTURES

In Table 10 and Fig. 10, we present a comparison of the best models in all architectures we considered. The results clearly show that the transformer model **bert-base-turkish-128k-cased** outperformed all others with considerably high difference. The CNN model with dictionary-based stemmer,

TABLE 5. Performance scores of Baseline ANN models.

Stemmer	Activ.	Optimizer	Acc	Prc	Rec	F1	Kap
Dict-Based	ReLU	Adam	0.681	0.672	0.677	0.670	0.521
		RAdam	0.679	0.670	0.674	0.667	0.518
		Yogi	0.642	0.624	0.638	0.627	0.462
		NovoGrad	0.561	0.534	0.558	0.538	0.339
	Swish	Adam	0.694	0.691	0.691	0.688	0.541
		RAdam	0.670	0.650	0.667	0.654	0.504
		Yogi	0.626	0.610	0.623	0.611	0.439
		NovoGrad	0.550	0.514	0.547	0.523	0.323
	Mish	Adam	0.675	0.672	0.672	0.667	0.512
		RAdam	0.631	0.630	0.628	0.625	0.446
		Yogi	0.624	0.611	0.620	0.611	0.436
		NovoGrad	0.509	0.473	0.505	0.480	0.260
Affix-Strip	ReLU	Adam	0.397	0.348	0.400	0.303	0.102
		RAdam	0.404	0.385	0.402	0.330	0.104
		Yogi	0.395	0.459	0.394	0.424	0.092
		NovoGrad	0.581	0.555	0.577	0.554	0.369
	Swish	Adam	0.448	0.467	0.446	0.408	0.170
		RAdam	0.378	0.384	0.380	0.328	0.070
		Yogi	0.380	0.338	0.379	0.322	0.069
		NovoGrad	0.429	0.467	0.426	0.392	0.141
	Mish	Adam	0.399	0.396	0.400	0.365	0.101
		RAdam	0.369	0.373	0.372	0.340	0.058
		Yogi	0.399	0.349	0.397	0.350	0.098
		NovoGrad	0.447	0.402	0.443	0.388	0.168
Fixed-Prefix	ReLU	Adam	0.387	0.377	0.392	0.304	0.090
		RAdam	0.423	0.377	0.430	0.344	0.145
		Yogi	0.429	0.419	0.429	0.402	0.146
		NovoGrad	0.460	0.422	0.457	0.404	0.190
	Swish	Adam	0.399	0.368	0.401	0.308	0.102
		RAdam	0.421	0.417	0.421	0.352	0.251
		Yogi	0.418	0.412	0.420	0.341	0.131
		NovoGrad	0.474	0.471	0.472	0.419	0.212
	Mish	Adam	0.425	0.467	0.421	0.372	0.134
		RAdam	0.401	0.410	0.399	0.341	0.101
		Yogi	0.413	0.445	0.409	0.352	0.116
		NovoGrad	0.433	0.419	0.430	0.372	0.148

* Top three values in each column are marked in boldface.

TABLE 6. Performance scores of CNN models.

Stemmer	Activ.	Optimizer	Acc	Prc	Rec	F1	Kap
Dict-Based	ReLU	Adam	0.742	0.751	0.740	0.737	0.613
		RAdam	0.720	0.726	0.717	0.711	0.579
		Yogi	0.713	0.724	0.710	0.703	0.569
		NovoGrad	0.711	0.708	0.703	0.703	0.567
	Swish	Adam	0.707	0.706	0.702	0.695	0.558
		RAdam	0.683	0.678	0.680	0.666	0.525
		Yogi	0.679	0.670	0.676	0.658	0.518
		NovoGrad	0.677	0.675	0.674	0.663	0.515
	Mish	Adam	0.712	0.715	0.709	0.705	0.567
		RAdam	0.684	0.692	0.681	0.670	0.526
		Yogi	0.672	0.684	0.670	0.654	0.509
		NovoGrad	0.689	0.700	0.687	0.678	0.534
Affix-Strip	ReLU	Adam	0.692	0.705	0.689	0.692	0.538
		RAdam	0.686	0.689	0.683	0.681	0.529
		Yogi	0.681	0.687	0.678	0.676	0.522
		NovoGrad	0.681	0.684	0.677	0.674	0.521
	Swish	Adam	0.723	0.731	0.720	0.718	0.584
		RAdam	0.688	0.696	0.685	0.678	0.533
		Yogi	0.678	0.688	0.675	0.664	0.517
		NovoGrad	0.678	0.685	0.675	0.667	0.517
	Mish	Adam	0.714	0.717	0.711	0.708	0.571
		RAdam	0.698	0.703	0.695	0.689	0.547
		Yogi	0.691	0.697	0.687	0.679	0.536
		NovoGrad	0.694	0.704	0.691	0.687	0.541
Fixed-Prefix	ReLU	Adam	0.703	0.717	0.697	0.699	0.555
		RAdam	0.678	0.695	0.671	0.658	0.516
		Yogi	0.669	0.697	0.662	0.649	0.503
		NovoGrad	0.683	0.693	0.675	0.665	0.523
	Swish	Adam	0.713	0.712	0.707	0.702	0.570
		RAdam	0.685	0.677	0.677	0.669	0.526
		Yogi	0.688	0.680	0.680	0.670	0.530
		NovoGrad	0.679	0.674	0.672	0.665	0.518
	Mish	Adam	0.718	0.745	0.712	0.705	0.576
		RAdam	0.681	0.692	0.673	0.666	0.520
		Yogi	0.679	0.692	0.672	0.665	0.518
		NovoGrad	0.671	0.679	0.664	0.658	0.505

* Top three values in each column are marked in boldface.

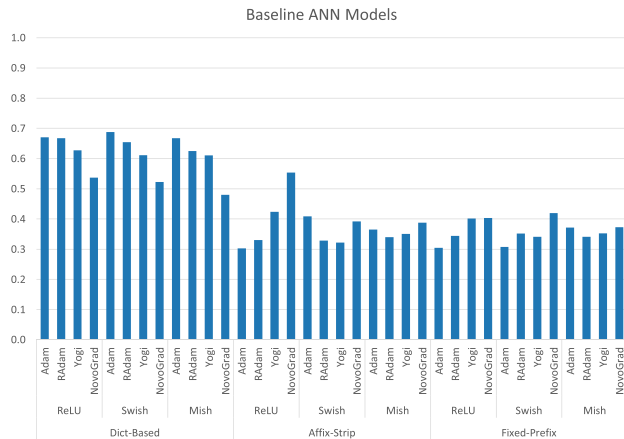


FIGURE 5. Comparison of Baseline ANN models in terms of F-Score.

ReLU activation, and Adam optimizer, was the model that showed the closest performance to the transformer. When we compare these two models in terms of architectural complexity, the CNN is much smaller and simpler than the transformer. Therefore, the CNN can be preferable to the transformer in environments with limited computational resources.

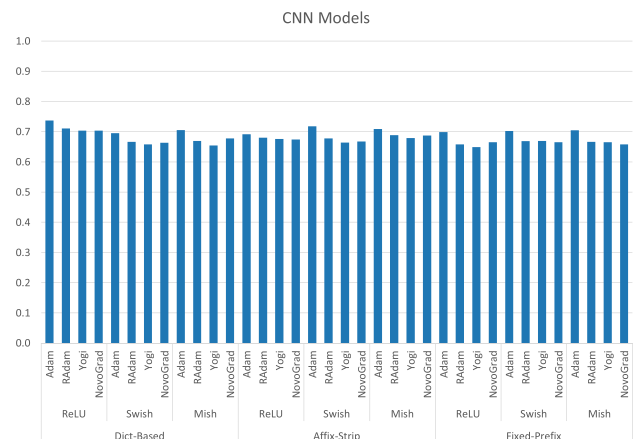


FIGURE 6. Comparison of CNN models in terms of F-Score.

No matter what DL architecture was used, Adam optimizer was present in the best models consistently. Therefore, Adam optimizer can be among the first optimizers to try in models that address similar tasks. Moreover, ReLU activation function along with Adam optimizer can make a good combination because they both were part of best two DL architectures we proposed, namely, CNN and BiLSTM.

TABLE 7. Performance scores of Conv+BiLSTM models.

Stemmer	Activ.	Optimizer	Acc	Prc	Rec	F1	Kap
Dict-Based	ReLU	Adam	0.640	0.668	0.637	0.615	0.460
		RAdam	0.654	0.661	0.651	0.632	0.481
		Yogi	0.647	0.657	0.644	0.621	0.471
		NovoGrad	0.644	0.645	0.641	0.623	0.466
	Swish	Adam	0.631	0.620	0.627	0.612	0.446
		RAdam	0.634	0.620	0.630	0.608	0.450
		Yogi	0.634	0.630	0.630	0.611	0.451
		NovoGrad	0.629	0.615	0.625	0.606	0.443
	Mish	Adam	0.626	0.621	0.624	0.584	0.440
		RAdam	0.635	0.640	0.631	0.616	0.451
		Yogi	0.627	0.635	0.623	0.605	0.439
		NovoGrad	0.633	0.635	0.629	0.615	0.448
Affix-Strip	ReLU	Adam	0.655	0.689	0.652	0.625	0.481
		RAdam	0.616	0.594	0.612	0.570	0.422
		Yogi	0.580	0.552	0.576	0.518	0.368
		NovoGrad	0.625	0.608	0.621	0.588	0.435
	Swish	Adam	0.633	0.695	0.629	0.589	0.449
		RAdam	0.641	0.665	0.637	0.609	0.461
		Yogi	0.629	0.654	0.624	0.592	0.442
		NovoGrad	0.653	0.669	0.649	0.626	0.479
	Mish	Adam	0.681	0.687	0.677	0.650	0.521
		RAdam	0.643	0.646	0.639	0.619	0.464
		Yogi	0.626	0.627	0.622	0.593	0.439
		NovoGrad	0.648	0.648	0.644	0.627	0.471
Fixed-Prefix	ReLU	Adam	0.615	0.627	0.606	0.593	0.422
		RAdam	0.596	0.594	0.587	0.564	0.391
		Yogi	0.577	0.556	0.568	0.538	0.361
		NovoGrad	0.593	0.585	0.585	0.558	0.388
	Swish	Adam	0.624	0.608	0.615	0.572	0.434
		RAdam	0.617	0.599	0.608	0.579	0.424
		Yogi	0.607	0.588	0.597	0.560	0.408
		NovoGrad	0.618	0.603	0.609	0.585	0.426
	Mish	Adam	0.639	0.619	0.629	0.590	0.456
		RAdam	0.627	0.616	0.618	0.592	0.440
		Yogi	0.631	0.616	0.622	0.593	0.445
		NovoGrad	0.630	0.616	0.621	0.599	0.444

* Top three values in each column are marked in boldface.

TABLE 8. Performance scores of BiLSTM models.

Stemmer	Activ.	Optimizer	Acc	Prc	Rec	F1	Kap
Dict-Based	ReLU	Adam	0.600	0.608	0.597	0.586	0.401
		RAdam	0.619	0.619	0.616	0.600	0.428
		Yogi	0.627	0.630	0.623	0.606	0.439
		NovoGrad	0.611	0.610	0.608	0.595	0.416
	Swish	Adam	0.614	0.614	0.587	0.600	0.401
		RAdam	0.583	0.595	0.579	0.536	0.373
		Yogi	0.574	0.573	0.571	0.566	0.361
		NovoGrad	0.583	0.582	0.580	0.576	0.374
	Mish	Adam	0.594	0.570	0.590	0.566	0.390
		RAdam	0.622	0.636	0.620	0.620	0.433
		Yogi	0.557	0.553	0.553	0.543	0.333
		NovoGrad	0.570	0.566	0.567	0.561	0.354
Affix-Strip	ReLU	Adam	0.716	0.723	0.723	0.710	0.574
		RAdam	0.674	0.675	0.671	0.661	0.510
		Yogi	0.671	0.675	0.668	0.657	0.506
		NovoGrad	0.673	0.672	0.670	0.661	0.509
	Swish	Adam	0.696	0.721	0.693	0.664	0.544
		RAdam	0.650	0.652	0.647	0.625	0.475
		Yogi	0.637	0.643	0.634	0.606	0.456
		NovoGrad	0.655	0.656	0.652	0.635	0.483
	Mish	Adam	0.624	0.623	0.620	0.586	0.436
		RAdam	0.636	0.640	0.632	0.613	0.453
		Yogi	0.616	0.620	0.613	0.586	0.424
		NovoGrad	0.645	0.651	0.642	0.627	0.468
Fixed-Prefix	ReLU	Adam	0.619	0.609	0.612	0.578	0.428
		RAdam	0.630	0.623	0.622	0.596	0.444
		Yogi	0.614	0.599	0.606	0.569	0.420
		NovoGrad	0.635	0.628	0.627	0.607	0.451
	Swish	Adam	0.643	0.650	0.634	0.616	0.463
		RAdam	0.640	0.644	0.632	0.620	0.459
		Yogi	0.631	0.631	0.622	0.608	0.445
		NovoGrad	0.637	0.639	0.628	0.618	0.454
	Mish	Adam	0.660	0.644	0.651	0.642	0.488
		RAdam	0.633	0.633	0.625	0.605	0.447
		Yogi	0.632	0.633	0.624	0.609	0.447
		NovoGrad	0.631	0.632	0.623	0.611	0.445

* Top three values in each column are marked in boldface.

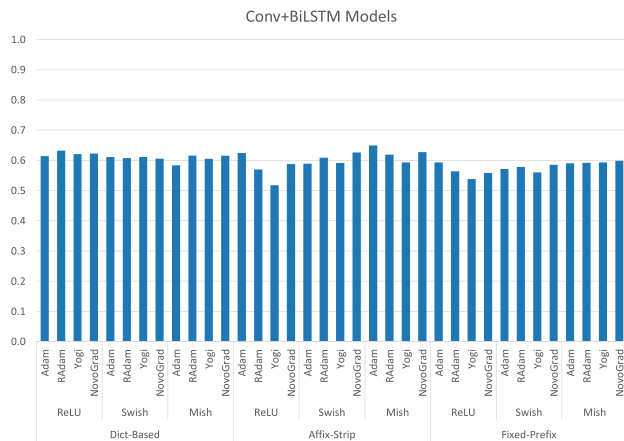


FIGURE 7. Comparison of Conv+BiLSTM models in terms of F-Score.

When we examine the experimental results from the perspective of the stemming method, dictionary-based stemmer was the most effective one with the Baseline ANN and CNN models. Affix-stripping stemmer, on the other hand, was most effective with the architectures that contain BiLSTM layer. Fixed-prefix stemming approach did never exhibit effectiveness in any of the experiments. Therefore, the logical

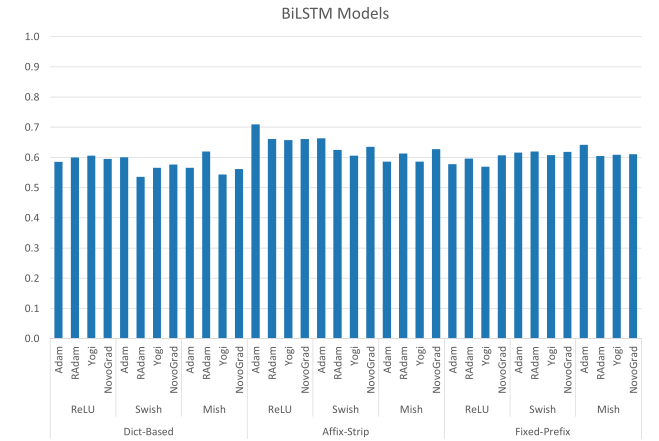


FIGURE 8. Comparison of BiLSTM models in terms of F-Score.

choice of stemmer in demand prioritization in Turkish would be to use either the dictionary-based one or the affix-stripping depending on the network architecture.

G. COMPARISON WITH OUR PREVIOUS STUDY

In our previous study [4], highest F-Score values we achieved were 0.54 and 0.53 with Support Vector Machines and

TABLE 9. Performance scores of pretrained transformer models.

Pretrained Transformer Model	Acc	Prc	Rec	F1	Kap
bert-base-turkish-128k-cased	0.756	0.747	0.767	0.746	0.601
distilbert-base-turkish-cased	0.824	0.809	0.786	0.780	0.722
electra-base-turkish-cased-disc	0.691	0.688	0.699	0.686	0.538
electra-small-turkish-cased-disc	0.706	0.690	0.698	0.693	0.558

* Top value in each column is marked in boldface.

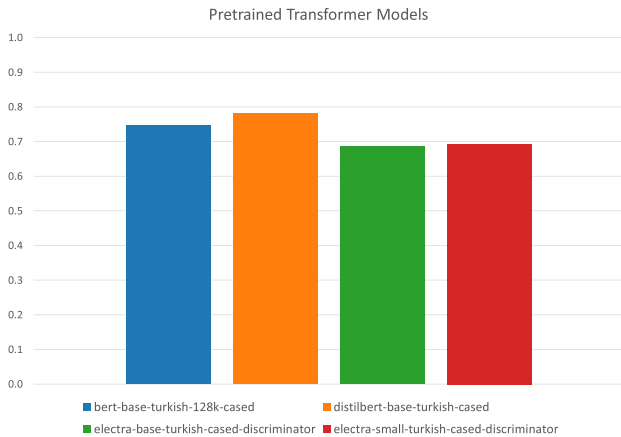


FIGURE 9. Comparison of pretrained transformer models in terms of F-Score.

Naive Bayes algorithms, respectively. In this current study, however, we achieved F-Score value of 0.78 with the pretrained transformer model called **distilbert-base-turkish-cased**. The CNN model with dictionary-based stemmer, ReLU activation, and Adam optimizer, was the model that showed the closest performance to the transformer with an F-Score value of 0.737. When compared to the transformer model, the CNN is much smaller and simpler in terms of architectural structure and complexity. Therefore, it can be a satisfactory alternative to the transformer in environments with limited computational resources. Moreover, our other models, apart from the Baseline ANN, showed better overall performance than the best scores of the previous study. As a result, DL-based NLP approaches are more feasible than the classical approaches for demand prioritization in Turkish language. Essentially, their success can be associated with the use of word embedding vectors, which keep the semantic relationships among the words much better than one-hot-encoding schemes like TF-IDF. On the other hand, feature extraction and sequence processing mechanisms provided by CNN and LSTM architectures contribute to their success. In conclusion, we can confidently state that DL can learn complex patterns in data that classical ML algorithms would not be able to learn.

H. MODELING FOR TURKISH LANGUAGE

NLP studies have usually been focused on model training with a monolingual corpus. Recently, we have seen some instances of multilingual models trained on multilingual corpora such as mBERT [35] and XLM [48]. However,

TABLE 10. Comparison of best models of all architectures.

Architecture	Best Model	Acc	Prc	Rec	F1	Kap
Baseline ANN	Dict-based+Swish+Adam	0.694	0.691	0.691	0.688	0.541
CNN	Dict-based+ReLU+Adam	0.742	0.751	0.740	0.737	0.613
Conv+BiLSTM	Affix-strip+Mish+Adam	0.681	0.687	0.677	0.650	0.521
BiLSTM	Affix-strip+ReLU+Adam	0.716	0.723	0.723	0.710	0.574
Transformers	distilbert-base-turkish-cased	0.824	0.809	0.786	0.780	0.722

* Top value in each column is marked in boldface.

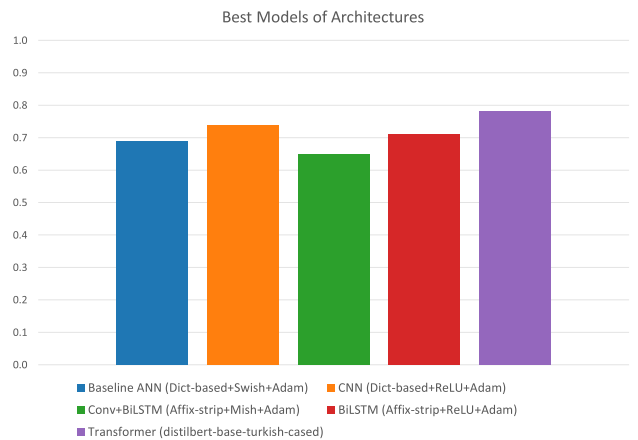


FIGURE 10. Comparison of best models of all architectures in terms of F-Score.

those multilingual models are known to perform well on downstream tasks for the languages with more training data. Similarly, low-resource languages like Turkish are not well represented in those models, resulting in a low performance on downstream tasks. Therefore, especially for low-resource languages, an individual monolingual model tends to perform better [49]. For this reason, in our study, we decided to develop and train (or fine-tune) our own monolingual models with our data coming from particularly insurance software domain.

One possible alternative to using multiple models for multiple languages would be to use translation, for example, from Turkish to English, and then to use a predictive model trained in English, thus, eliminating the need for training different models for different languages. This approach, nevertheless, would not be feasible because exact and acceptable translation across languages is still not possible. Besides, original textual data is usually full of grammatical or clerical errors in a real-world setting like our demand prioritization data. Translation of such data would probably produce results with loss of information, rendering the process useless.

VI. CONCLUSION

In this study, we applied state-of-the-art concepts and techniques in NLP to the problem of prioritization of software

development demands in Turkish, where we considered it as a text classification problem. In order to achieve this, we proposed several DL architectures to improve demand prioritization over our previous research. Through an extensive experimentation, we compared the effectiveness of our deep learning architectures trained with several combinations of different optimizers and activation functions in order to reveal the best combination for demand prioritization in Turkish. We empirically showed that deep learning models can achieve much higher accuracy than classical ML models even with a small amount of training data.

One of the main contributions of this study was the introduction of an original demand prioritization dataset in Turkish language, taken from the DMS of a private insurance company in Turkey. This dataset can serve as a reference to further similar studies in software engineering domain. In addition, this dataset can be useful to researchers who conduct general-purpose text classification research in a low-resource language like Turkish.

ACKNOWLEDGMENT

The author would like to thank Kaan Bıçakçı for his invaluable suggestions for the research and his assistance with conducting the experiments.

REFERENCES

- [1] V. Tunali and M. A. A. Tüysüz, "Analysis of function-call graphs of open-source software systems using complex network analysis," *Pamukkale Univ. J. Eng. Sci.*, vol. 26, no. 2, pp. 352–358, 2020.
- [2] J. Kanwal and O. Maqbool, "Bug prioritization to facilitate bug report triage," *J. Comput. Sci. Technol.*, vol. 27, no. 2, pp. 397–412, Mar. 2012.
- [3] J. Uddin, R. Ghazali, M. M. Deris, R. Naseem, and H. Shah, "A survey on bug prioritization," *Artif. Intell. Rev.*, vol. 47, no. 2, Feb. 2016, pp. 145–180.
- [4] M. C. Tekin and V. Tunali, "Yazılım geliştirme taleplerinin metin madenciliği yöntemleriyle önceliklendirilmesi," *Pamukkale Univ. J. Eng. Sci.*, vol. 25, no. 5, pp. 615–620, 2019.
- [5] A. A. Patel and A. Uppili, *Applied Natural Language Processing in the Enterprise*. Sebastopol, CA, USA: O'Reilly Media, 2021.
- [6] K. K. Chaturvedi and V. B. Singh, "Determining bug severity using machine learning techniques," in *Proc. CSI 6th Int. Conf. Softw. Eng. (CONSEG)*, Sep. 2012, pp. 1–6.
- [7] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proc. 25th Int. Conf. Softw. Eng.*, May 2003, pp. 465–475.
- [8] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2008, pp. 346–355.
- [9] L. Yu, W.-T. Tsai, W. Zhao, and F. Wu, "Predicting defect priority based on neural networks," in *Advanced Data Mining and Applications*, L. Cao, J. Zhong, and Y. Feng, Eds. Berlin, Germany: Springer, 2010, pp. 356–367.
- [10] J. Kanwal and O. Maqbool, "Managing open bug repositories through bug report prioritization using SVMs," in *Proc. 4th Int. Conf. Open-Source Syst. Technol. (ICOSST)*, Dec. 2010, pp. 22–24.
- [11] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Proc. 7th IEEE Work. Conf. Mining Softw. Repositories (MSR)*, May 2010, pp. 1–10.
- [12] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proc. 2nd Int. Workshop Recommendation Syst. Softw. Eng. (RSSE)*, 2010, pp. 52–56.
- [13] A. Lamkanfi, S. Demeyer, Q. D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *Proc. 15th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2011, pp. 249–258.
- [14] W. Abdelmoez, M. Kholief, and F. M. Elsalmy, "Bug fix-time prediction model using naive Bayes classifier," in *Proc. 22nd Int. Conf. Comput. Theory Appl. (ICCTA)*, Oct. 2012, pp. 167–172.
- [15] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *Proc. 19th Work. Conf. Reverse Eng.*, Washington, DC, USA, Oct. 2012, pp. 215–224.
- [16] M. Alenezi and S. Banitaan, "Bug reports prioritization: Which features and classifier to use?" in *Proc. 12th Int. Conf. Mach. Learn. Appl.*, Dec. 2013, pp. 112–116.
- [17] Y. Tian, D. Lo, and C. Sun, "DRONE: Predicting priority of reported bugs by multi-factor analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Washington, DC, USA, Sep. 2013, pp. 200–209.
- [18] G. Yang, T. Zhang, and B. Lee, "Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports," in *Proc. IEEE 38th Annu. Comput. Softw. Appl. Conf.*, Jul. 2014, pp. 97–106.
- [19] G. Sharma, S. Sharma, and S. Gujral, "A novel way of assessing software bug severity using dictionary of critical terms," *Proc. Comput. Sci.*, vol. 70, pp. 632–639, Jan. 2015.
- [20] T. Zhang, J. Chen, G. Yang, B. Lee, and X. Luo, "Towards more accurate severity prediction and fixer recommendation of software bugs," *J. Syst. Softw.*, vol. 117, pp. 166–184, Jul. 2016.
- [21] P. Choudhary, "Neural network based bug priority prediction model using text classification techniques," *Int. J. Adv. Res. Comput. Sci.*, vol. 8, no. 5, pp. 1315–1319, May/Jun. 2017.
- [22] M. Kumari and V. B. Singh, "An improved classifier based on entropy and deep learning for bug priority prediction," in *Proc. 18th Int. Conf. Intell. Syst. Design Appl. (ISDA)*, Dec. 2018, pp. 571–580.
- [23] Q. Umer, H. Liu, and I. Illahi, "CNN-based automatic prioritization of bug reports," *IEEE Trans. Rel.*, vol. 69, no. 4, pp. 1341–1354, Dec. 2020.
- [24] K. Oflazer, "Turkish and its challenges for language processing," *Lang. Resour. Eval.*, vol. 48, no. 4, pp. 639–653, Dec. 2014.
- [25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.
- [26] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, M. Hasan, B. C. Van Essen, A. A. S. Awwal, and V. K. Asari, "A state-of-the-art survey on deep learning theory and architectures," *Electronics*, vol. 8, no. 3, p. 292, Mar. 2019.
- [27] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. 1st Int. Conf. Learn. Represent.*, (ICLR), May 2013, pp. 1–12.
- [28] S. Edunov, M. Ott, M. Auli, and D. Grangier, "Understanding back-translation at scale," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2018, pp. 489–500.
- [29] R. Řehůřek and P. Sojka, "Software framework for topic modelling with large corpora," in *Proc. LREC Workshop New Challenges NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50.
- [30] V. Tunali and T. T. Bilgin, "PRETO: A high-performance text mining tool for preprocessing Turkish texts," in *Proc. 13th Int. Conf. Comput. Syst. Technol. (CompSysTech)*, 2012, pp. 134–140.
- [31] V. Tunali and T. T. Bilgin, "Examining the impact of stemming on clustering Turkish texts," in *Proc. Int. Symp. Innov. Intell. Syst. Appl.*, Jul. 2012, pp. 1–4.
- [32] E. M. Bayol. (2021). *TRNLP: Türkçe İçin Doğal Dil İşleme Araçları*. [Online]. Available: <https://github.com/brolin59/trnlp>
- [33] G. Eryigit and E. Adali, "An affix stripping morphological analyzer for Turkish," in *Proc. Int. Conf. Artif. Intell. Appl.*, 2004, pp. 299–304.
- [34] V. Tunali and T. T. Bilgin, "Türkçe metinlerin kümeleneşinde farklı kök bulma yöntemlerinin etkisinin araştırılması," in *Proc. Elektrik, Elektronik ve Bilgisayar Mühendisliği Sempozyumu (ELECO)*, 2012, pp. 598–602.
- [35] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, vol. 1, Jun. 2019, pp. 4171–4186.
- [36] P. J. Ortiz Suárez, L. Romary, and B. Sagot, "A monolingual approach to contextualized word embeddings for mid-resource languages," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, Jul. 2020, pp. 1703–1714.
- [37] J. Tiedemann, "Parallel data, tools and interfaces in OPUS," in *Proc. 8th Int. Conf. Lang. Resour. Eval. (LREC)*, May 2012, pp. 2214–2218.
- [38] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter," 2019, *arXiv:1910.01108*.
- [39] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: Pre-training text encoders as discriminators rather than generators," in *Proc. 8th Int. Conf. Learn. Represent. (ICLR)*, Apr. 2020, pp. 1–18.
- [40] Loodos. (2021). *Transformer Based Turkish Language Models*. [Online]. Available: <https://github.com/Loodos/turkish-language-models>

- [41] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, and M. Kudlur, "TensorFlow: A system for large-scale machine learning," in *Proc. Symp. Operating Syst. Design Implement.*, May 2016, pp. 265–283.
- [42] F. Chollet. (2021). *Keras*. [Online]. Available: <https://keras.io>
- [43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2015, *arXiv:1412.6980*.
- [44] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," in *Proc. 8th Int. Conf. Learn. Represent. (ICLR)*, Apr. 2020, pp. 1–14.
- [45] M. Zaheer, S. Reddi, D. Sachan, S. Kale, and S. Kumar, "Adaptive methods for nonconvex optimization," in *Proc. 32nd Conf. Neural Inf. Process. Syst. (NeurIPS)*, Dec. 2018, pp. 9815–9825.
- [46] B. Ginsburg, P. Castonguay, O. Hrinchuk, O. Kuchaiev, V. Lavrukhin, R. Leary, J. Li, H. Nguyen, Y. Zhang, and J. M. Cohen, "Stochastic gradient methods with layer-wise adaptive moments for training of deep networks," 2019, *arXiv:1905.11286*.
- [47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, and V. Dubourg, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, no. 10, pp. 2825–2830, Jul. 2017.
- [48] A. Conneau and G. Lample, "Cross-lingual language model pretraining," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst. (NeurIPS)*, Dec. 2019, pp. 7059–7069.
- [49] S. Wu and M. Dredze, "Are all languages created equal in multilingual BERT?" in *Proc. 5th Workshop Represent. Learn. (NLP)*, 2020, pp. 120–130.



VOLKAN TUNALI (Member, IEEE) was born in Bursa, Turkey, in 1978. He received the B.S. and M.S. degrees in computer engineering and the Ph.D. degree in computer and control education from Marmara University, Istanbul, in 2001, 2005, and 2012, respectively. From 2001 to 2012, he worked as a Software Engineer in a private software company in Istanbul. Since 2012, he has been an Assistant Professor with the Department of Software Engineering, Maltepe University, Istanbul. His research interests include network science, social network analysis, text mining, data science, and software engineering.

• • •