SDE-Project Report

****
****

Team Members:

Swetha Kumari Kerahalli (M20AIE317) ->Worked on the implementation section and report and ppt review.
Akshata A. Kulkarni (M20AIE209) ->Worked on the report section.

****
****

Youtube Link: https://youtu.be/3Ug8IHW8V4E
Git repo:
Git clone "git@github.com:swetha-kerahalli-iitj/Swetha_M20AIE317_SDE_PRJ.git"
https://github.com/swetha-kerahalli-iitj/Swetha_M20AIE317_SDE_PRJ.git

****
****

Introduction:

Paper Chosen:
"Deep Learning Based Code Smell Detection", by Hui Liu; Jiahao Jin; Zhifeng Xu; Yanzhen Zou; Yifan Bu; Lu Zhang
https://ieeexplore.ieee.org/document/8807230

Before getting into a brief introduction about the paper, we need to understand what code smell is.

Code Smells are the structures which will be present in the source code in which there will be scope for refactoring the source code and we developers will be able to identify these refactoring bits by detecting the code smells.

In computer programming and software design, code refactoring is the process of restructuring existing computer code—changing the factoring—without changing its external behavior. Refactoring is intended to improve the design, structure, and/or implementation of the software, while preserving its functionality.

In this paper, they have proposed number of approaches to identify code smells automatically or semi-automatically, and as it is difficult to manually select the best features as most of these approaches rely on manually designed heuristics so as to map the selected code metrics into predictions manually, so they have proposed deep learning techniques-based approach for detecting code smells.

One of the advantages of deep neural networks and advanced deep learning techniques is that they could select best features from source code automatically for code smell detection and can build the complex mapping between the selected features and predictions of the same.

But there is a limitation in using deep learning approaches, which is that deep learning requires a lot of training data in order to fine tune a large number of parameters within the existing deep neural network whereas existing datasets for code smell detection are small. So, to overcome this, they have proposed an automatic approach to generate labeled training data for the neural network-based classifier which can be achieved without any human intervention.

As part of code smells detection, they applied the deep neural techniques approach on four common and famous code smells which are feature envy, long method, large class, and misplaced class. From the proposed approach , they were able to observe that there was significant improvisation with respect to state-of-art which was evaluated based on open-source applications.

Summary of Work done:

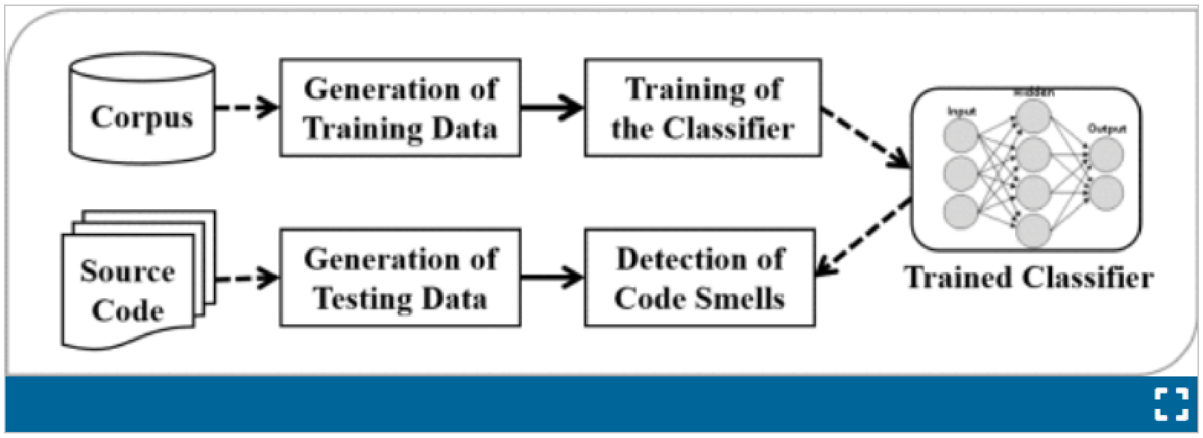

Corpus
Generation of Training Data
Training of the Classifier
Input
Hidden
Output
Trained Classifier
Source Code
Generation of Testing Data
Detection of Code Smells

Fig. 1.
Overview of the proposed approach.

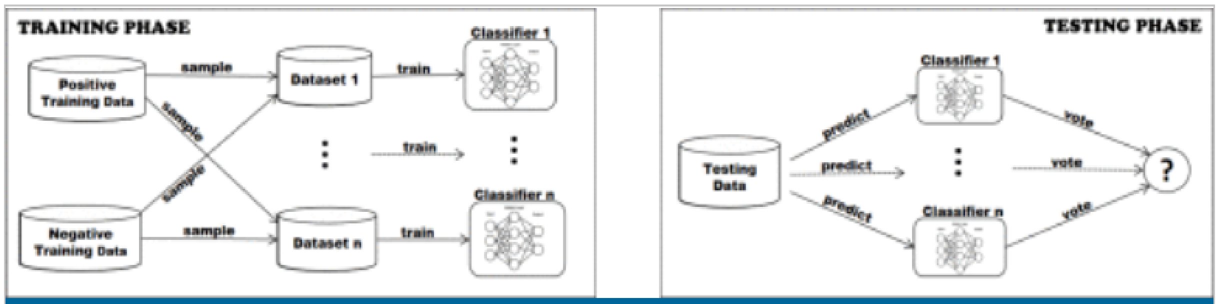


Fig. 2.
Common framework for deep learning based smell detection.

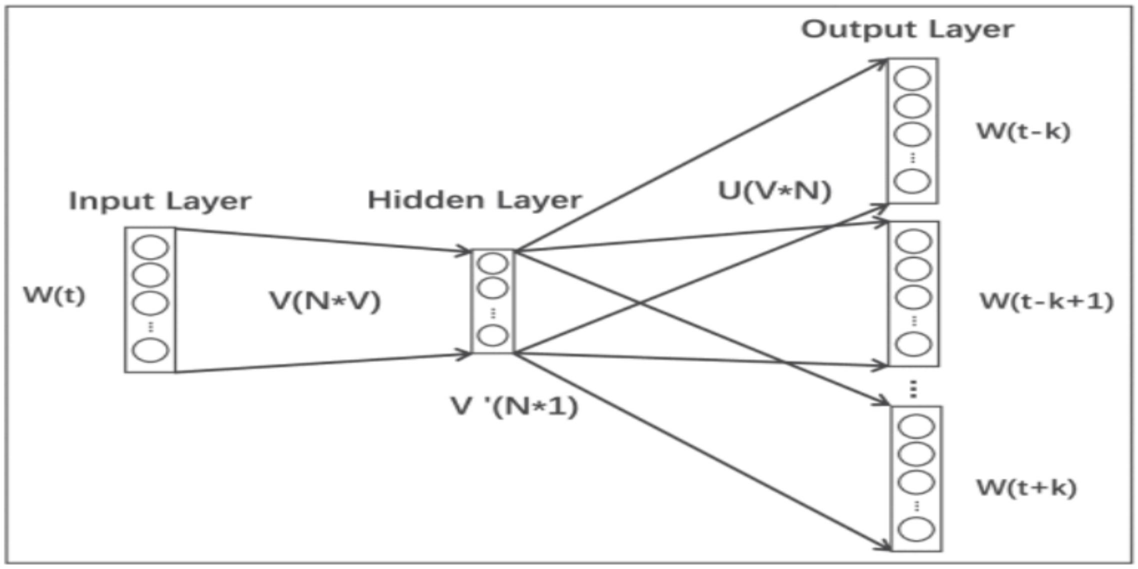


Fig. 3.
Model of Word2Vector.

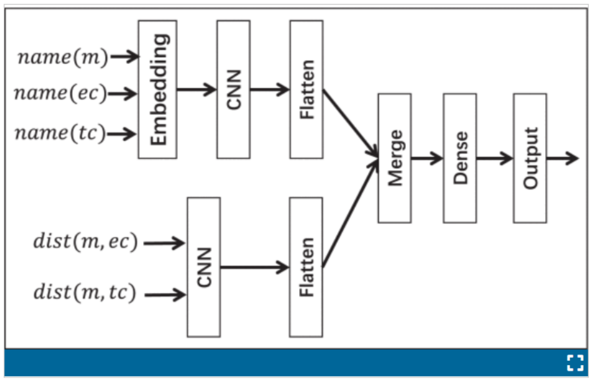


Fig. 4.
Classifier for feature envy detection.

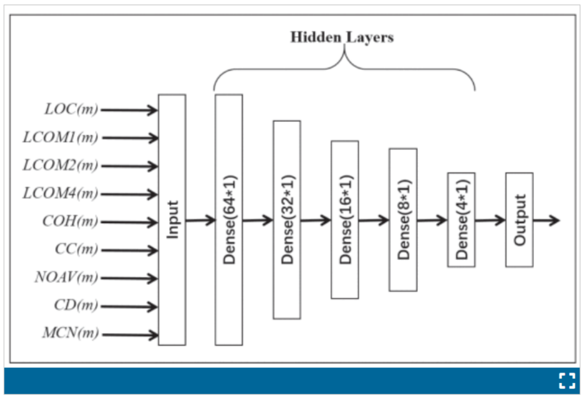


Fig. 5.
Classifier for long method detection.

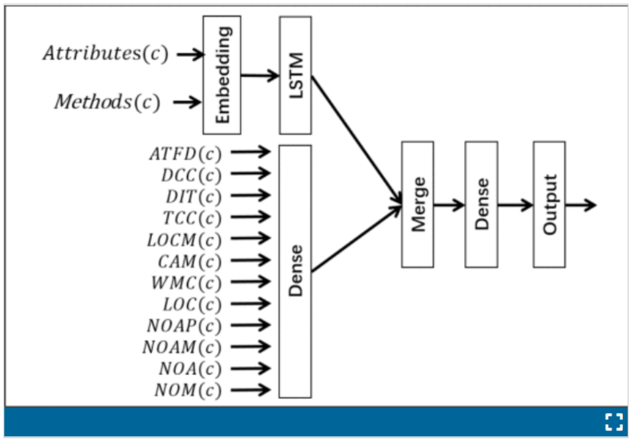


Fig. 6.
Classifier for large class detection.

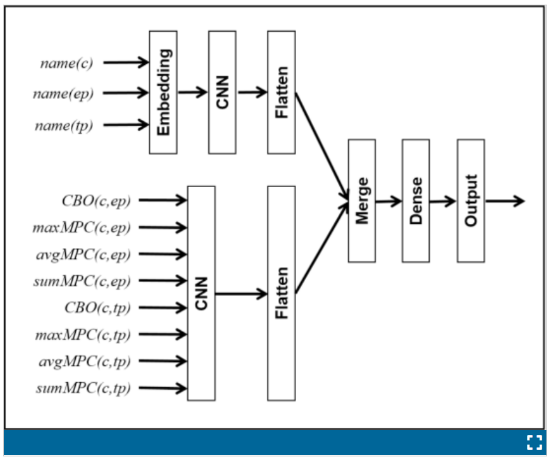


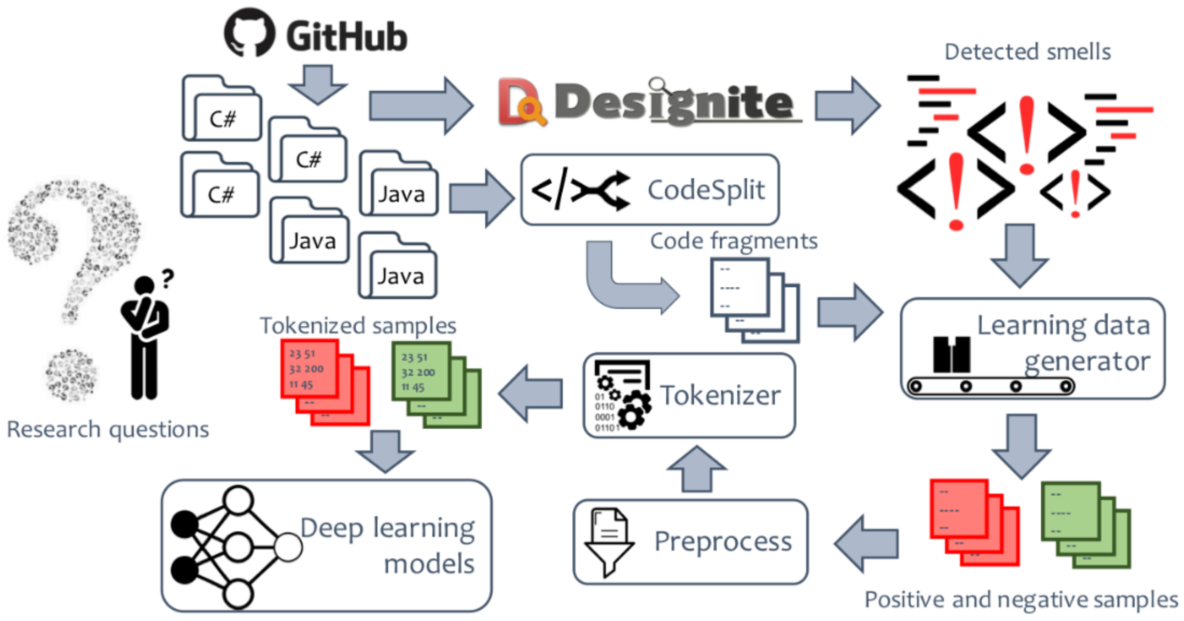
Fig. 7.
Classifier for misplaced class detection.

The method they followed as in Paper:

1. **Generating learning data:** For a given corpus, they have generated positive training data by smell-introducing refactoring and on the same corpus of subject application, they have generated negative training data.
2. **Deep learning based Smell detection:** From positive and negative data , they randomly sample a subset to form a training dataset and train a neural network based classifier with the dataset and iterate the process for sampling and training for n times.
3. **Code Smells:** They have applied DNN to four common and extensively used Code Smells which are to be feature envy, long method, large class, misplaced class and provided how they have evaluated the accuracy of each Code Smell detections and their respective results over different subject applications.

Note: The above figures have been taken from the original paper itself.

Implementation:



The method we followed as follows:

1. **Download repositories:**

   We used the following protocol to identify and download our subject systems. We download repositories containing C# and Java code from GitHub. We use RepoReapers to filter out low-quality repositories. RepoReapers analyzes GitHub repositories and provides scores for nine dimensions of quality. These dimensions are architecture, community, continuous integration, documentation, history, license, issues, and unit tests.

2. **Splitting code fragments:**

   CodeSplit are utilities to split the methods or classes written in C# and Java source code into individual files. The utilities can parse the code correctly (using Roslyn for C# and Eclipse JDT for Java), and emit the individual methods or classes fragments into separate files following hierarchical structure (i.e., namespaces/packages become folders). CodeSplit for Java is an open-source project that can be found on GitHub. CodeSplit for C# can be downloaded freely online.

3. **Smells detection:**

   We use Designite to detect smells in C# code. Designite is a software design quality assessment tool for projects written in C#. It supports detection of eleven implementations, 19 designs, and seven architecture smells. It also provides commonly used code metrics and other features such as trend analysis, code clones detection, and dependency structure matrix to help us assess the software quality. A free academic license can be requested for all the academic purposes. Similar to the C# version, we have developed DesigniteJava which is an open-source tool to analyze Java code. We use DesigniteJava to detect smells in the Java codebase. The tool supports detection of 17 design and ten implementation smells.

We use console version of Designite (version 2.5.10) and DesigniteJava (version 1.1.0) to analyze C# and Java code respectively and detect design and implementation smells in each of the downloaded repositories.

4. **Generating learning data:**

The learning data generator requires information from two sources:
A list of detected smells for each analyzed repository and the path to the folder where code fragments are stored corresponding to the repository.

The program takes a method (or class in case of design smell) at a time and checks whether the given smell has been detected in the method (or class). If yes, the program puts the code fragment into the positive folder corresponding to the smell, otherwise into the negative folder. all_smells_files is a collection of files containing information about detected smells.

For implementation smells, it is a list of projectN_implSmells.csv; similarly, for design smells, it is a list of projectN_designSmells.csv[Ex: rnn_rq1_ComplexConditionalfinal_17112022_0007.csv].
Implementation_smell_name | Namespace_name | Class_name | File_path | Method_name | Description |

Each project1_designSmells.csv has the following columns:
Design_smell_name | Namespace_name | Class_name | File_path | Description |

5. **Tokenizing learning data:**

Machine learning algorithms including neural networks take vectors of numbers as input. Hence, we need to convert source code into vectors of numbers honoring the language keywords and other semantics. Tokenizer is an open-source tool to tokenize source code into integer vectors, symbols, or discrete tokens. It supports six programming languages currently including C# and Java.

6. **Data format:**

For 1D format, each sample is stored in a line.
For 2D format, two samples are separated by one new line

7. **Data preparation:**

The stored samples are read into numpy arrays, preprocessed, and filtered. We first perform bare minimum preprocessing to clean the data. For both 1D and 2D samples, we scan all the samples for each smell and remove duplicates if they exist.

We balance the number of samples for training by choosing the smaller number from positive and negative sample count for training. We discard the remaining training samples from the larger side. We figure out the maximum input length (or, maximum input height and width in case of 2-D samples) for an individual sample. To filter out the outliers, we read all the samples into a numpy array and compute mean and standard deviation. We discard all the samples where the length of the sample is greater than mean + standard deviation. This filtering helps us keep the training set in reasonable bounds and avoids waste of memory and processing resources. Finally, we shuffle the array of input samples along with its corresponding labels array.

Note:We fine tuned the hyper parameters for data processing, for 1D and 2D ,but results are based on less number of epoch since it was taking time and my system got crashed multiple times while executing and was not able to capture the results, and we have processed data processing for C# code smell though we got generated data for both C# and Java source code.And we have used only three Code Smell detections which are to be feature envy, Complex method which is long method and Complex condition which is large class as though we are able to generate the training data, we were unable to process for data processing due to system crash and deadline.

Implementation Comparison:

Below are the brief details of implementation comparison :

Proposed Paper	**Implemented Improvements**
For categorizing smells they have used ● Categorized and found refactoring opportunities ● randomly select one and apply refactoring ● Undo applied refactoring	Used Designate tool to identify smells. Since Designate smells uses in-depth analysis like trend analysis and other features. The quality of smells is better compared to conventional methods of smell detection.
Traditional methods used for code split and tokenization	Used CodeSplit tool and tokenization Tool for tokenization
Generates labeled training data by randomly selecting smell -introducing factors	Automatically generated training data to detect code smells
Apply conventional tokenization method of Word2Vector. Always generated 1D data.	tokenize source code into integer vectors, symbols, or discrete tokens as applicable. Generates 1D and 2D tokenized data.
Basic preprocessing or Removal of Duplicates of data done. No 1D or 2D samples generated	The stored samples are read into numpy arrays, preprocessed, and **filtered**. We first perform bare minimum preprocessing to clean the data. For both **1D and 2D** samples, we scan all the samples for each smell and remove **duplicates** if they exist. We **balance** the number of samples for training by choosing the smaller number from positive and negative sample count for training. We **discard** the remaining training samples from the larger side. We figure out the **maximum input length (or, maximum input height and width in case of 2-D samples)** for an individual sample. To filter out the outliers, we read all the samples into a numpy array and **compute mean and standard deviation**. We discard all the samples where the length of the s**ample is greater than mean + standard** deviation. This filtering helps us keep the

	training set in reasonable bounds and avoids waste of memory and processing resources. Finally, we **shuffle the array** of input samples along with its corresponding labels array
conventional CNN applied	Convolution 1D/2D used Normalization Applied MaxPooling applied Two layers of dense applied(relu and sigmoid) Batch Sizes of 32,64,128 and 256 are used.

Results:

Observations of the implementation are as follows:

DNN Approaches	Feature Envy AUC Results [%]	Complex Method AUC Results [%]	Complex Conditional AUC Results [%]
CNN 1D	50.00	83.72	73.16
CNN 2D	Not done	89.48	50.00
RQ1 RNN Embedded LSTM [Performance based]	Not done yet	85.44	82.82
RQ2 RNN Embedded LSTM [Accuracy based]	73.89	74.56	76.33

Results Updated:

Observations of the implementation are as follows:
Note: Not all algorithm/smell combinations were executed during the updated results and can be referred for in the results section.

Smell	Accuracy(%)	TestAccuracy(%)	DNN Approaches
ComplexMethod	82.74228067	85.76765656	rq1_cnn_1d
ComplexMethod	83.31662822	84.52336192	rq1_cnn_1d
ComplexMethod	83.60686461	90.74790478	rq1_rnn_emb_lstm_1d
ComplexMethod	88.5909	86.670112	rq1_cnn_2d
ComplexConditional	86.08766928	82.19563365	rq1_rnn_emb_lstm_1d

FeatureEnvy	83.24231072	71.55659795	rq1_rnn_emb_lstm_1d

Comparison Results:

Code Smell detection	Paper AUC Results[%]	Implementation AUC Results[%]
Feature Envy	76.35	73.89
Long Method	79.24	89.48
Large Class Method	75.77	82.82
Misplaced Class	99.09	Not done

Comparison Results Updated:

Code Smell detection	Paper AUC Results[%]	Implementation AUC Results[%]
Feature Envy	76.35	83.24
Long Method	79.24	83.60
Large Class Method	75.77	86.08
Misplaced Class	99.09	Not done

Conclusion:

We have observed from the results that, though we have taken less number of epochs for training data and applied very minimal approaches, significant improvisation can be achieved by applying different approaches and fine tuning the hyper parameters for training data.

Reference:

https://ieeexplore.ieee.org/document/9914414
https://ieeexplore.ieee.org/document/9761969
https://ieeexplore.ieee.org/document/9724775
https://ieeexplore.ieee.org/document/9724436
https://ieeexplore.ieee.org/document/9596476
https://dl.acm.org/doi/10.1145/3477535
https://onlinelibrary.wiley.com/doi/10.1002/smr.2369
https://github.com/tushartushar/DeepLearningSmells

https://link.springer.com/article/10.1007/s11432-019-2830-8
https://dl.acm.org/doi/10.1145/3387940.3392191
https://journals.sagepub.com/doi/10.1177/1063293X20958932
https://link.springer.com/article/10.1007/s13369-020-04365-1
https://ieeexplore.ieee.org/document/9529815
https://ieeexplore.ieee.org/document/9392959
https://link.springer.com/chapter/10.1007/978-981-15-7984-4_37
https://ieeexplore.ieee.org/document/9424815
https://ieeexplore.ieee.org/document/9359294
https://ieeexplore.ieee.org/document/9054826
Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. Empirical Software Engineering 22, 6 (01 Dec 2017), 3219–3253. https://doi.org/10.1007/s10664 - 017- 9512- 6