Machine Learning for Source-code Plagiarism Detection

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science in Computer Science and Engineering
by
Research

by

Jitendra Yasaswi Bharadwaj Katta
201407566
jitendra.katta@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500032, INDIA
July 2018

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Machine Learning for Source-code Plagiarism Detection" by Jitendra Yasaswi Bharadwaj Katta, has been carried out under our supervision and is not submitted elsewhere for a degree.

Date	Adviser: Prof. C.V. Jawahar
Date	Co-Adviser: Dr. Suresh Purini

To

My Family

Acknowledgements

My journey in IIIT-Hyderabad has been a wonderful experience. As I submit my MS thesis, I wish to extend my gratitude to all those people who helped me in successfully completing this journey.

I would first like to thank my thesis advisor Prof. C.V. Jawahar for all the guidance and support. I could not have imagined having a better advisor and mentor for my MS. He has been a source of continuous inspiration throughout my MS journey. His guidance has not only helped me become a good researcher but also a better person. Thank you for pushing me beyond my limits.

I would also like to extend my special gratitude towards my co-advisor Dr. Suresh Purini for accepting to work with me. It was both an honour and a great privilege to work with him.

Working at CVIT and ML lab was great fun. Thank you Aditya, Priyam, Praveen, Yashaswi, Anand Mishra, Aniketh, Suriya, Pritish, Saurabh, Jobin, Vijay and all other CVITians for the wonderful interactions and fun work sessions. I am grateful to Siva, R. S. Satyanarayana, Varun, Rajan and Silar for all the support. Special thanks to Satwik, Soumyajit, Pramodh and Harish for sharing your knowledge and for those cheerful experiences in IIIT-H.

My life at IIIT-H would not be complete without my dearest friends Ganesh, Vijay and Sukanya. Its truly priceless to find friends like you. I am thankful to you guys for giving me a ton of happy memories during my stay at IIIT-H. Thanks Sukanya for your constant support, motivation, fruitful discussions and arguments which made me a better person. This journey would not have been complete without people like you.

I could not have accomplished it without the support and understanding of my parents and my family. They have given me continuous encouragement throughout my years of study and through the process of researching and writing this thesis. Last, but not the least, thanks to IIIT-H community for giving me an inspiring environment and loads of opportunities to grow.

Abstract

This thesis presents a set of machine learning and deep learning approaches for building systems with the goal of source-code plagiarism detection. The task of plagiarism detection can be treated as assessing the amount of similarity presented within given entities. These entities can be anything like documents containing text, source-code etc. Plagiarism detection can be formulated as a fine-grained pattern classification problem. The detection process begins by transforming the entity into feature representations. These features are representatives of their corresponding entities in a discriminative high-dimensional space, where we can measure for similarity. Here, by entity we mean solution to programming assignments in typical computer science courses. The quality of the features determine the quality of detection

As our first contribution, we propose a machine learning based approach for plagiarism detection in programming assignments using source-code metrics. Most of the well known plagiarism detectors either employ a text-based approach or use features based on the property of the program at a syntactic level. However, both these approaches succumb to code obfuscation which is a huge obstacle for automatic software plagiarism detection. Our proposed method uses source-code metrics as features, which are extracted from the intermediate representation of a program in a compiler infrastructure such as gcc. We demonstrate the use of unsupervised and supervised learning techniques on the extracted feature representations and show that our system is robust to code obfuscation. We validate our method on assignments from introductory programming course. The preliminary results show that our system is better when compared to other popular tools like MOSS. For visualizing the local and global structure of the features, we obtained the low-dimensional representations of our features using a popular technique called t-SNE, a variation of Stochastic Neighbor Embedding, which can preserve neighborhood identity in low-dimensions. Based on this idea of preserving neighborhood identity, we mine interesting information such as the diversity in student solution approaches to a given problem. The presence of well defined clusters in low-dimensional visualizations demonstrate that our features are capable of capturing interesting programming patterns.

As our second contribution, we demonstrate how deep neural networks can be employed to learn features for source-code plagiarism detection. We employ a character-level Recurrent Neural Network (char-RNN), a character-level language model to map the characters in a source-code to continuous-valued vectors called embeddings. We use these program embeddings as deep features for plagiarism

detection in programming assignments. Many popular plagiarism detection tools are based on n-gram techniques at syntactic level. However, these approaches to plagiarism detection fail to capture long term dependencies (non-contiguous interaction) present in the source-code. Contrarily, the proposed deep features capture non-contiguous interaction within n-grams. These are generic in nature and there is no need to fine-tune the char-RNN model again to program submissions from each individual problem-set. Our experiments show the effectiveness of deep features in the task of classifying assignment program submissions as copy, partial-copy and non-copy.

As our final contribution, we demonstrate how to extract local deep features from source-code. We represent programs using local deep features and develop a framework to retrieve suspicious plagiarized cases for a given query program. Such representations are useful for identification of *near-duplicate* program pairs, where only a part of the program is copied or certain lines, blocks of code may be copied etc. In such cases, obtaining local feature representations for a program is more useful than representing a program with a single global feature. We develop a retrieval framework using Bag of Words (BoW) approach to retrieve susceptible plagiarized and partial-plagiarized (near-duplicate) cases for a given query program.

Contents

List of Figures

List of Tables

Listings

Chapter 1

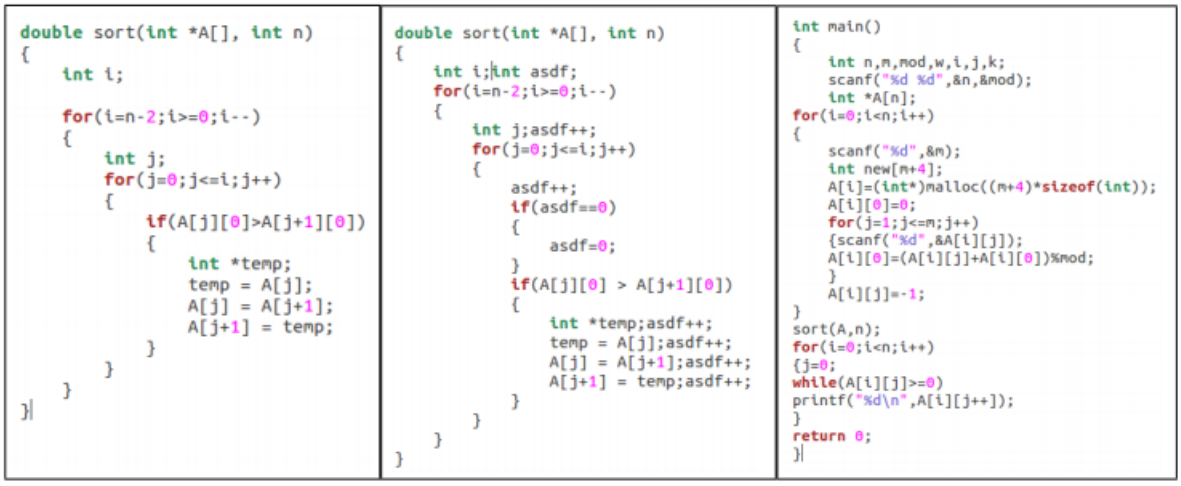
Introduction

1.1 Problem and Scope

Martins et al., [29] define plagiarism as "the usage of work without crediting its authors". The easy and cheap access to enormous web content has turned plagiarism into a serious problem be it for researchers, publishers or educational institutions. One of the most common forms of plagiarism in academia is the textual plagiarism at document level, where documents are typically essays or reports and scientific papers. According to a recent survey [42], 16 percent of the original articles published in leading surgical journals could be considered redundant publication. However, plagiarism applies not only to text documents but also to source-code as well. Source-code plagiarism can be defined as trying to pass off (parts of) source code written by someone else as one's own (i.e., without indicating which parts are copied from which author) [17].

In academic environment, source-code plagiarism arise in programming assignments. With the intention of achieving good grades with less or almost no effort, students often try to copy the assignments from their friends. Freshmen who plagiarize in their courses are more likely to continue this malpractice in their later courses. Therefore this malpractice needs to be curbed at its initial stages. The instructor of a course can receive a false feedback about the level of the course and performance of the students. This makes the problem of assignment plagiarism detection an important task. It is hard to manually inspect and (decide whether a submission is genuine or plagiarized) detect similar student submitted solutions in a large class. Though manual inspection is effective, it is laborious and time consuming.

One possible way to address this is to seek the help of automated code comparison tools like Measure Of Software Similarity (MOSS) [43] , JPlag [39] which help in identifying similar submission pairs. Most of the well known automatic code comparison tools rely solely on the text-based approach or use the features solely based on the property of the assignments at a syntactic level to detect plagiarism. However, both these approaches succumb to code obfuscation [31], which is a huge obstacle to automatic software plagiarism detection. Often students use clever techniques to obfuscate the code and evade from being detected.



```c
double sort(int *A[], int n)
{
    int i;

    for(i=n-2;i>=0;i--)
    {
        int j;
        for(j=0;j<=i;j++)
        {
            if(A[j][0]>A[j+1][0])
            {
                int *temp;
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}
```

(a) CODE SNIPPET-1

```c
double sort(int *A[], int n)
{
    int i;int asdf;
    for(i=n-2;i>=0;i--)
    {
        int j;asdf++;
        for(j=0;j<=i;j++)
        {
            asdf++;
            if(asdf==0)
            {
                asdf=0;
            }
            if(A[j][0] > A[j+1][0])
            {
                int *temp;asdf++;
                temp = A[j];asdf++;
                A[j] = A[j+1];asdf++;
                A[j+1] = temp;asdf++;
            }
        }
    }
}
```

(b) CODE SNIPPET-2

```c
int main()
{
    int n,m,mod,w,i,j,k;
    scanf("%d %d",&n,&mod);
    int *A[n];
for(i=0;i<n;i++)
{
    scanf("%d",&m);
    int new[m+4];
    A[i]=(int*)malloc((m+4)*sizeof(int));
    A[i][0]=0;
    for(j=1;j<=m;j++)
    {scanf("%d",&A[i][j]);
    A[i][0]=(A[i][j]+A[i][0])%mod;
    }
    A[i][j]=-1;
}
sort(A,n);
for(i=0;i<n;i++)
{j=0;
while(A[i][j]>=0)
printf("%d\n",A[i][j++]);
}
return 0;
}
```

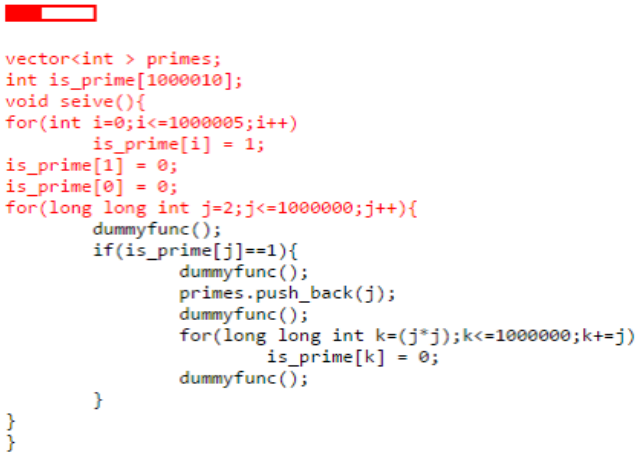
(c) CODE SNIPPET-3

Figure 1.1: Sample code-snippets showing the code obfuscated with code insertion technique. Carefully notice the usage of the variable asdf in code snippet-2.

In the context of programming assignments (from an introductory computer science course) few examples of code obfuscation are statement reordering, code insertion (dead code injection), careful conversion of variable names, control replacement etc. The figure 1.1 shows the code obfuscated with code insertion technique. Popular plagiarism detection tools like MOSS accepts as input a set of programs and outputs pairs of similar programs, the matched lines and percentage of matched code. Usually when teaching assistants evaluate the student submitted solutions, they consider only the solution pairs as copy cases where similarity score is above some threshold (say 80%). This threshold varies from assignment to assignment depending on the type of problem asked to solve in the assignment. However, MOSS falls prey to code obfuscation as it outputs a very low similarity value when the code is obfuscated. For example, consider the code snippets shown in figure 1.2 and 1.3 . The similarity score given by moss is only 40%. But surely the second code is copied from the first code.

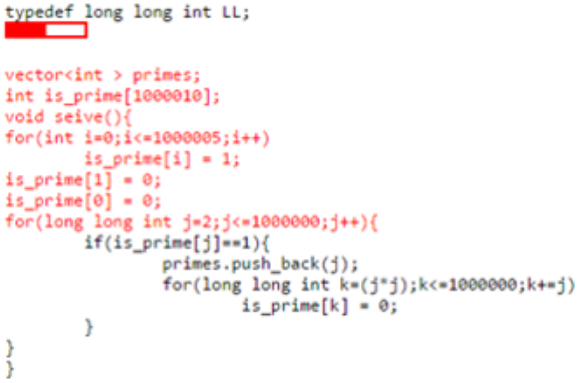
In this work, our focus is on building a tool powered by machine learning algorithms, to detect plagiarism in programming assignments. We demonstrate the use of unsupervised and supervised learning techniques and show that our system is robust to code obfuscation. We propose and use novel features that can alleviate the effect of code obfuscation and be able to detect the plagiarized student solutions. We use source code metrics as feature representations which can capture the variations observed in the code as belonging to three distinct themes: *structure, syntax* (syntax refer to the tokens that occur within basic blocks) and *presentation*. Later, we demonstrate how to learn embeddings for programs using deep neural networks and use these embeddings as features for source-code plagiarism detection.

A large number of student submissions, the variations in student submitted codes, the techniques employed by students to obfuscate the code makes this a challenging problem. To the best of our



```
vector<int > primes;
int is_prime[1000010];
void seive(){
for(int i=0;i<=1000005;i++)
        is_prime[i] = 1;
is_prime[1] = 0;
is_prime[0] = 0;
for(long long int j=2;j<=1000000;j++){
        dummyfunc();
        if(is_prime[j]==1){
                dummyfunc();
                primes.push_back(j);
                dummyfunc();
                for(long long int k=(j*j);k<=1000000;k+=j)
                        is_prime[k] = 0;
                dummyfunc();
        }
}
}
```

Figure 1.2: Sample code snippet-1. The code highlighted in red color is detected as plagiarized content by MOSS.



```
typedef long long int LL;

vector<int > primes;
int is_prime[1000010];
void seive(){
for(int i=0;i<=1000005;i++)
        is_prime[i] = 1;
is_prime[1] = 0;
is_prime[0] = 0;
for(long long int j=2;j<=1000000;j++){
        if(is_prime[j]==1){
                primes.push_back(j);
                for(long long int k=(j*j);k<=1000000;k+=j)
                        is_prime[k] = 0;
        }
}
}
```

Figure 1.3: Code snippet-1 obfuscated with code insertion. The code highlighted in red color is detected as plagiarized content by MOSS.

knowledge, our work is one of the first few to incorporate modern machine learning techniques into detecting plagiarism (specifically for programming assignments).

1.2 Related Work

1.2.1 Plagiarism Detection

In previous works, researchers have used program similarity metrics to identify plagiarism such as MOSS [43] , JPlag [39] etc. However, most of the well known automatic code comparison tools employ a text-based approach or use the features based on the property of the assignments at a syntactic level to detect plagiarism. However, both these approaches succumb to code obfuscation [31] which is a huge obstacle to automatic software plagiarism detection. MOSS is based on the property of the assignments at a syntactic level that use winnowing [43], a local fingerprinting algorithm. MOSS fingerprint selection is not very accurate (selects the fingerprint with minimum value in a window). On top of this fingerprint, a longest common sequence search is performed.

JPlag [39] is another popular tool for plagiarism detection. JPlag uses greedy string tiling on each pair of submissions to find the longest common sequences on the tokenized version of source code. JPlag is similar to MOSS in working. However, by focusing on common blocks of tokenized constructs, JPlag still overlooks some key features such as the formatting and style of the submitted code when making its comparison.

Likewise, other approaches have tried analyzing program dependence graphs to detect plagiarism. In [26], the authors proposed a new plagiarism detection tool, called GPlag, which detects plagiarism by

mining program dependence graphs (PDGs). A PDG is a graphic representation of the data and control dependencies within a procedure. Because PDGs are nearly invariant during plagiarism, GPlag is more effective for plagiarism detection.

In [40], the authors propose a representation of source code pairs by using five high level features; namely: lexical features, stylistic feature,comments feature, programmer's text feature, and structure feature. Particularly, for the lexical, comments and programmer's text features, they represent source code as a set of characters n-grams. These features are more oriented to detect aspects that the programmers leave in natural language more than in a particular programming language.

In [11], the authors proposed a system that is based on properties of assignments that course instructors use to judge the similarity of two submissions. They proposed 12 features (out of which MOSS similarity score itself is a feature) like similarity in comments, white-space etc. However, both MOSS and JPlag filter out these features when performing their analysis. Their main motivation is to use these features as cues in plagiarism detection. This system uses neural network techniques to measure the relevance of each feature in the assessment. However, their focus is on detecting plagiarized pairs within a single problem set. However, our proposed method is independent of problem set.

1.2.2 Grading Programs

Although not specifically related to plagiarism detection in assignments, there is quite a bit of research that attempts to use program similarity metrics as features. The advent of Massive Open-Access On-line Courses (MOOCs) has provided data at very large scale in the form of assignment submissions. Researchers have successfully applied *Machine Learning techniques* to unravel interesting patterns from these large datasets. These works include grading open-ended questions, feed-back and clustering [14], modeling student solutions [37] and coding style [41].

Rogers et al. [41] introduced ACES – Automated Coding Evaluation of Style – a module that automates grading for the composition of Python programs. ACES, given certain constraints, assesses the composition of a program through static analysis, conversion from code to an Abstract Syntax Tree, and clustering (unsupervised learning), helping streamline the subjective process of grading based on style and identifying common mistakes. Further, they created visual representations of the clusters to allow readers and students understand where a submission falls, and what are the overall trends. Moreover, the authors extrapolate grading results from the tool, including what common mistakes students make and what features are considered more important.

Brooks et al. [8] used clustering to grade open-ended short answer questions quickly and at scale. They proposed a cluster-based interface that allowed teachers to read, grade, and provide feedback on large groups of answers at once. Glassmen et al. [14] used a clustering approach, but with a focus on feature engineering and specifically for the grading of code. However their main aim was to provide a feedback and alternate solutions to the students. They performed a two-level hierarchical clustering of student solutions: first they partitioned them based on the choice of algorithm, and then partitioned

solutions implementing the same algorithm based on low-level implementation details. They found that in order to cluster submissions effectively, both abstract and concrete features needed to be extracted and clustered.

1.2.3 Detecting Duplicate Document Pairs

The identification of near-duplicate entity pairs in a large collection is a significant problem with wide-spread applications. Two source-codes are said to be *near-duplicate* if they differ from each other in a very small portion. Previous studies on near-duplicate detection can be roughly divided into two directions: document representation and efficient detection. This problem of near-duplicate detection has been well studied for documents and web pages. One of the popular approach is shingling [7], where each document is broken down into overlapping fragments called shingles. For each document, all the shingles or word sequences of adjacent words are extracted. If two documents contain same set of shingles, they can be termed as near-duplicates. Based on the ratio of magnitude of the shingle union and intersection, the degree to which two documents resemblance each other is calculated. Other popular approach for near-duplicate document detection is based on creating signatures from n-grams. In [49] the authors proposed a novel method to identify which n-grams should be used to create signatures. The n-gram selection is based on *SpotSigs* method which combines stop-word antecedents with short chains of adjacent content terms.

Indyk and Motwani [21] proposed the notion of Locality Sensitive Hashing (LSH) and applied it to sublinear-time similarity searching. LSH maintains a number of hash tables, which each of is parameterized by the number of hashed dimensions. Points close to each other in some metric space have the same hash value with high probability. Kołcz et al. [18] proposed a method based on the idea of improved similarity measure. They represent each document as a real-valued sparse n-gram vector, where the weights can be learned to optimize for a specified similarity function, such as the cosine similarity. Near-duplicate documents can be reliably detected through this improved similarity measure. In addition, these vectors can be mapped to a small number of hash-values as document signatures through the LSH scheme for efficient similarity computation. Most of the above mentioned methods for near-duplicate detection focus on document level, partial duplicates can not be dealt well with these approaches.

Many of the methods proposed in this thesis are inspired by the successful applications of machine learning techniques in the domain of natural language processing and computer vision. In this chapter, section 1.3.1 details general machine learning formulation and section 1.3.2 mentions about the deep learning and recurrent neural networks used to learn program embeddings.

1.3 Background

1.3.1 Machine Learning Formulation

"Machine learning is the science of getting computers to act without being explicitly programmed" (Arthur Samuel, 1959). Machine learning explores the study and construction of algorithms that can learn from and make predictions on data. Such algorithms operate by building a model from example inputs in order to make data-driven predictions or decisions, rather than following strictly static program instructions. It is employed in a range of computing tasks where designing and programming explicit algorithms is in-feasible. Machine learning technology powers many aspects of modern society, web searches to content filtering on social platforms to recommendations on e-commerce sites. Machine learning is so ubiquitous today that we probably use it dozens of times a day without knowing it. Machine learning has given us practical speech recognition systems, effective web search, self-driving cars, and a vastly improved understanding of the human genome.

Machine learning tasks are typically classified into three broad categories, depending on the nature of the learning "signal" or "feedback" available to a learning system. These are:

1. **Supervised learning :** The computer is presented with example inputs and their desired outputs, given by a teacher. The goal is to learn a general rule/function that maps inputs to outputs. Classification and regression belongs to this category.

2. **Unsupervised learning :** Labels are not provided to the learning algorithm, leaving it on its own to discover the structure in its input. Clustering algorithms fall under this category.

3. **Reinforcement learning :** A computer program interacts with a dynamic environment in which it must perform a certain goal (such as playing a game against an opponent), without explicitly telling it whether it has come close to its goal. The program is provided feedback in terms of rewards and punishments as it navigates its problem space.

This thesis deals with two categories(Supervised and Unsupervised learning). In this work, our focus is on building a tool powered by machine learning algorithms, to detect plagiarism in programming assignments. Many popular machine learning algorithms like Support Vector Machines (SVM), Nearest Neighbor and Artificial Neural Networks (ANN), require lots of 'labeled' or 'annotated' data for building accurate models. Especially in our plagiarism detection task, labels are required for a pair of programs. These labels are referred as pairwise labels. Enumerating all possible pairs from individual data points and annotating them is both extremely expensive and time consuming. The burden of annotating data required for training the models can be significantly reduced using *Active Learning*. In real world scenario, it is common to have imbalanced datasets where the number of instances of one class far exceeds the other. In such cases, instead of treating the problem as classification problem, one can treat it as an *Anomaly Detection* problem and solve it. These techniques are discussed below.

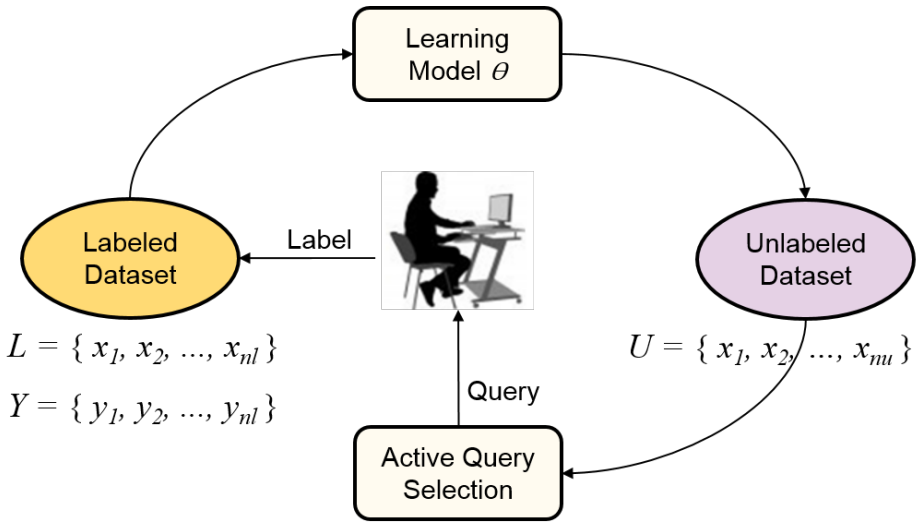


Figure 1.4: A typical Active Learning scenario. At first, the learning model is trained using few training samples L, Y from a labeled dataset. Next, a large unlabeled dataset U is passed to a learned model to get the predictions. Then, query selection strategy is used for selecting the best query that helps to minimize the needed input from a human annotator. After the sample is labeled by annotator, it is added to the labeled dataset and a procedure is repeated again.

Active Learning

Active learning is a sub-field of machine learning, a special case of semi-supervised machine learning. The task of Active Learning is to search for a small set of informative samples that restricts search space as much as possible. The learner in active learning is encouraged to be *curious* so that it interactively chooses which data points to label. The learner will eventually perform better with less training. In active learning, the algorithm tries to both learn the task and tell us what labels would be most useful at the current state. The user can then label just those data points, so that the manual effort is reduced significantly and directed in labeling useful data points thus making problem solving via machine learning more practical. The figure 1.4 below summarizes active learning strategy

The strategy for query selection can be based on different criteria. Few popular strategies proposed in the literature are Uncertainty sampling, Query by committee, Expected model change, Expected error reduction, Variance reduction, Balance exploration and Exponentiated gradient exploration for active learning [6].

Anomaly Detection

Most supervised machine learning algorithms works best when the number of instances belonging to each class are roughly equal. However, in practice it is extremely common to have imbalanced datasets where the number of instances of one class far exceeds the other. Anomaly detection is the task of

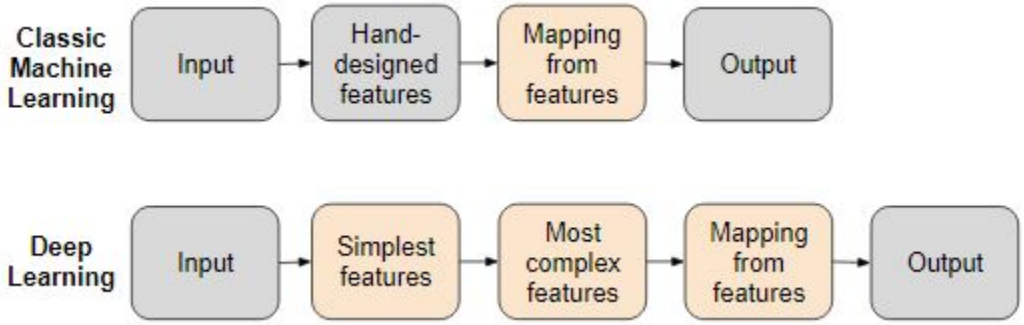


Figure 1.5: Figure demonstrating the distinction between traditional machine learning pipeline and deep learning pipeline. Orange color in the diagram represents the learning stage in the pipeline.

identifying instances or data points which do not agree to an expected pattern or other instances in a dataset. The key difference to the standard classification task is the inherent unbalanced nature of dataset.

Consider the scenario where the total number of instances in a class of data (positive) is far less than the total number of instances in another class of data (negative). In our task of plagiarism the *positives* correspond to *copy* cases and *negatives* corresponds to *non-copy* cases. Sometimes the positive instances will make up only 1% of the entire dataset. Since, there are not enough positive examples for learning algorithm to get a sense of what positive examples are like and future positive examples may look nothing like anomalous examples seen so far, it is not good to treat this as a standard supervised learning problem and train a classifier. Hence, we can treat this as a anomaly detection problem.

In anomaly detection task, the first step is to construct a training set consisting of majority of the negative instances only, a cross-validation set consisting of the negative instances and few positive instances, a test set consisting of the negative instances and few positive instances. The next step is to fit a known distribution 'p' (Gaussian), on training dataset. Next, on the cross-validation, we are going to think of the anomaly detection algorithm as trying to predict the value of y (a label) depending on a threshold ϵ. The value of ϵ is decided using cross-validation. Given input x in test set, the anomaly detection algorithm predicts the y as positive if $p(x)$ is less than ϵ. Predicting that it is an anomaly. And the algorithm is predicting that y is a negative, if $p(x)$ is greater then or equals ϵ. Predicting those normal example if the $p(x)$ is reasonably large.

1.3.2 Deep Learning

Increasingly machine learning technology is using a class of techniques called Deep Learning [5]. The models used in chapter 3 of this thesis comes under the umbrella of Deep Learning. Traditional machine learning was limited to its ability to process data in the raw form or heavily relied on features which needed careful hand-engineering which took years to build. For a long time, most machine learning technology was based on carefully hand-engineered features which needed considerable domain

expertise in order to design feature extractor which transformed the raw data to a suitable feature vector on which a learning system can work on. The choice of learning system depends on the task at hand like object recognition, sequence classification or regression tasks on complicated objects like trees etc.

The goal of *representation learning* is to find an appropriate representation of the data in order to perform a machine learning task. This concept is exploited by deep learning by its innate nature. For example, in a neural network, each hidden layer maps its input data to an inner representation that tends to capture a higher level of abstraction. These learned features are increasingly more informative through layers towards the machine learning task that we intend to perform (e.g. classification). Deep-learning attempt to learn multiple levels of representation of increasing complexity/abstraction. These deep features are hierarchical in nature and learned through a series of non-linear transformations modeled by deep neural networks. Figure 1.5 demonstrates the difference in the way features are obtained in traditional machine learning and deep learning. Recently, by combining the complex representations that deep neural networks can learn with the goal-driven learning of a Reinforcement Learning (RL) agent, computers have accomplished some amazing feats, like beating humans at over a dozen Atari games, and defeating the Go world champion.

1.3.3 Recurrent Neural Networks (RNN)

Traditional feed forward neural networks, can not predict an event based on its past experiences. This is contrary to the way humans think. Also, humans are good at processing sequences. RNNs are inspired from the way humans think. They build upon the knowledge which persists across time. Unlike, traditional neural networks, RNNs have loops in them which allow the information to persist. RNNs is are blessed with "memory" to captures information about what has been encountered in the past. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations [2].

As shown in figure 1.6, RNNs when unrolled reveal a chain-like nature and are intimately related to sequences and list. One of the appeals of the RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the sky," we do not need any further context its pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that its needed is small, RNNs can learn to use the past information. But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France I speak fluent French." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France from further back. It is entirely possible for the gap between the relevant information and the point where it is needed to become very large.

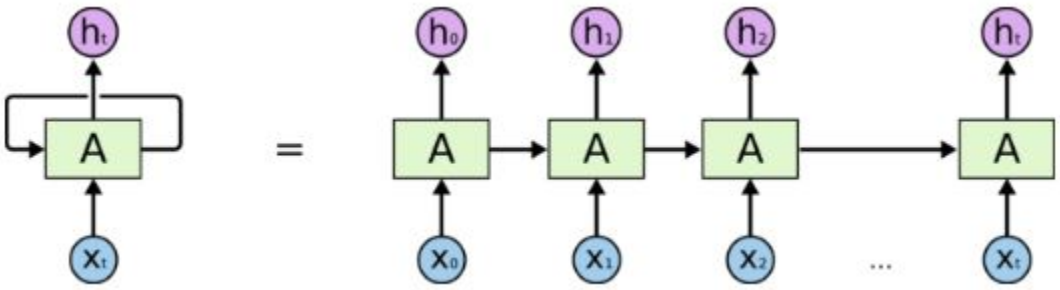


Figure 1.6: Unrolling a recurrent neural network. [1]

Long Short Term Memory (LSTM) Networks

RNNs are capable of holding an information over small time steps. However, they can not do that over a longer time steps. LSTM [20] are special kind of RNNs that are capable of learning long-term dependencies. Their innate behaviour is to remember information for longer periods of time. This is enabled by the special gate mechanisms and interactions among them, that are present in the LSTM units. All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single *tanh* layer. However, in LSTM, instead of having a single neural network layer, there are four which interact in a very special way.

The key to LSTMs is the cell state. The cell state is like a conveyor belt. It runs straight down the entire chain, with minor linear interactions, which helps the information to flow along it unchanged. The LSTM cell have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of sigmoid neutral net layer and a point-wise multiplication operation. These gates allows LSTM memory cells to store and access information over long period of time, thereby avoiding the vanishing gradient problem.

It can be seen from figure 1.7 there are four interacting layers in LSTM cell. Initially, the "forget gate" with sigmoid activation decides what information is to be thrown out from the cell state. Later, step decides what new information to store in the cell state. This has two parts, a) a *sigmoid* layer called the "input gate layer" decides which values we'll update and b) a *tanh* layer creates a vector of new candidate values that could be added to the state. In next step, these two are combined to create an update to the cell state. Final step is to decide what we're going to output. Then, we run a *sigmoid* layer which decides what parts of the cell state were going to output. Then, we put the cell state through *tanh* (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

LSTM have been used in applications like image caption generation [51], video summarization and machine translation [46] etc.

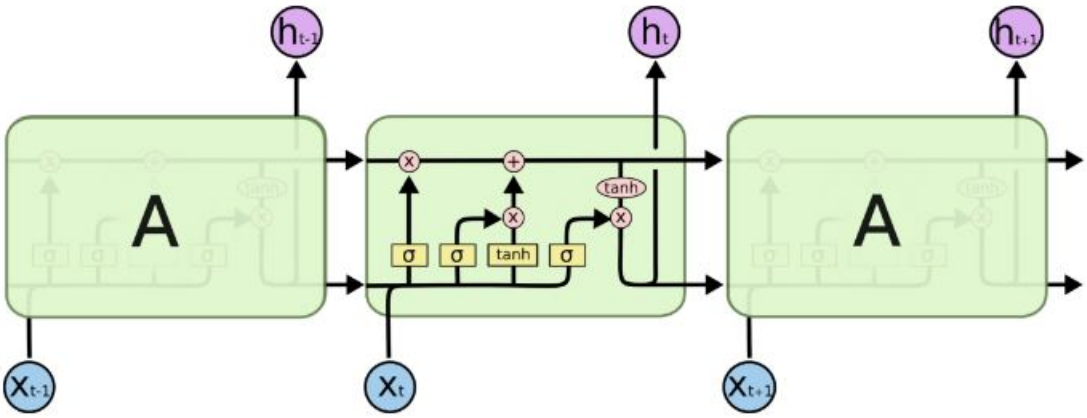


Figure 1.7: The repeating module in a standard LSTM contains four interacting layers. [1]

1.4 Contribution and Thesis Overview

We focus on source-code plagiarism detection and retrieval of plagiarized cases from a given database of student submitted programs. The main contributions of this thesis are:

1. **Problem:** *To design new feature representations for programs that are robust to code obfuscation and can detect plagiarized programming assignment submissions.*

   Most of the well known plagiarism detectors either employ a text-based approach or use features based on the property of the program at a syntactic level. However, both these approaches succumb to code obfuscation which is a huge obstacle for automatic source-code plagiarism detection. We propose and use source-code metrics as features, which are extracted from the intermediate representation of a program in a compiler infrastructure such as gcc. We demonstrate the use of unsupervised and supervised learning techniques on the extracted feature representations and show that our features are robust to code obfuscation.

2. **Problem:** *Tackle the limitations of previously used textual features and code metrics which were hand engineered, by learning the features from data.*

   Major limitations of designing hand-engineered features is the amount of time involved throughout the design and experimentation cycle. Still this required lot of domain knowledge and has limitation on number of features based no domain experience. We propose a method for detecting plagiarism in source-codes using deep features. The embeddings for programs are obtained using a character-level Recurrent Neural Network (char-RNN), which is pre-trained on Linux Kernel source-code. Many popular plagiarism detection tools are based on n-gram techniques at syntactic level. However, these approaches to plagiarism detection fail to capture long term dependencies (non-contiguous interaction) present in the source-code. Contrarily, the proposed deep features capture non-contiguous interaction within n-grams.

3. Proposed an efficient scheme for retrieval of plagiarized and near-duplicate student submissions, which is analogous to Bag of Words (BoW) representation. Our method is highly problem-set independent. The efficiency of proposed method is shown experimentally on twenty-two programming problem sets.

1.4.1 Organization of Thesis

The thesis is organized as follows. In chapter 2, we introduce source-code metrics as features and demonstrate the use of unsupervised and supervised learning techniques for plagiarism detection. In chapter 3, we imbibe motivations from deep learning and apply recurrent neural networks to learn embeddings for programs. We introduce deep features and demonstrate their superiority over source-code metrics. In chapter 4, we propose a system to retrieve potential plagiarized cases from the database of student submitted solutions to introductory programming assignments. We use bag-of-local deep features inspired by bag-of-words method, to detect near-duplicate source-codes. Finally in chapter 5, we summarize the contributions of this work along with the possible scope for the work in future and final conclusions.

Chapter 2

Plagiarism detection in Programming Assignments Using Source-code Metrics

In this chapter, our focus is on building a system powered by machine learning algorithms, to detect source-code plagiarism in programming assignments. We demonstrate the use of unsupervised and supervised learning techniques and show that our system is robust to code obfuscation. We propose and use novel features that can alleviate the effect of code obfuscation and be able to detect the plagiarized student solutions. We use source-code metrics as feature representations which can capture the variations observed in the code as belonging to three distinct themes: *structure, syntax* (syntax refers to the tokens that occur within basic blocks) and *presentation*.

For visualizing the local and global structure of the features, we obtained the low-dimensional representations of our features using a popular technique called t-SNE, a variation of Stochastic Neighbor Embedding, which can preserve neighborhood identity in low-dimensions. Based on this idea of preserving neighborhood identity, we mine interesting information such as the diversity in student solution approaches to a given problem. The presence of well defined clusters in low-dimensional visualizations demonstrate that our features are capable of capturing interesting programming patterns

Our key contributions in this chapter are:

1. Use of source code metrics (static code-based features) extracted during code compilation as feature representations of the student solutions to the given programming assignments.

2. Machine learning based approach to detect potential plagiarized cases.

A large number of student submissions, the variations in student submitted codes, the techniques employed by students to obfuscate the code makes this a challenging problem. To the best of our knowledge, our work is one of the first few to incorporate machine learning techniques into detecting plagiarism (specifically for programming assignments).

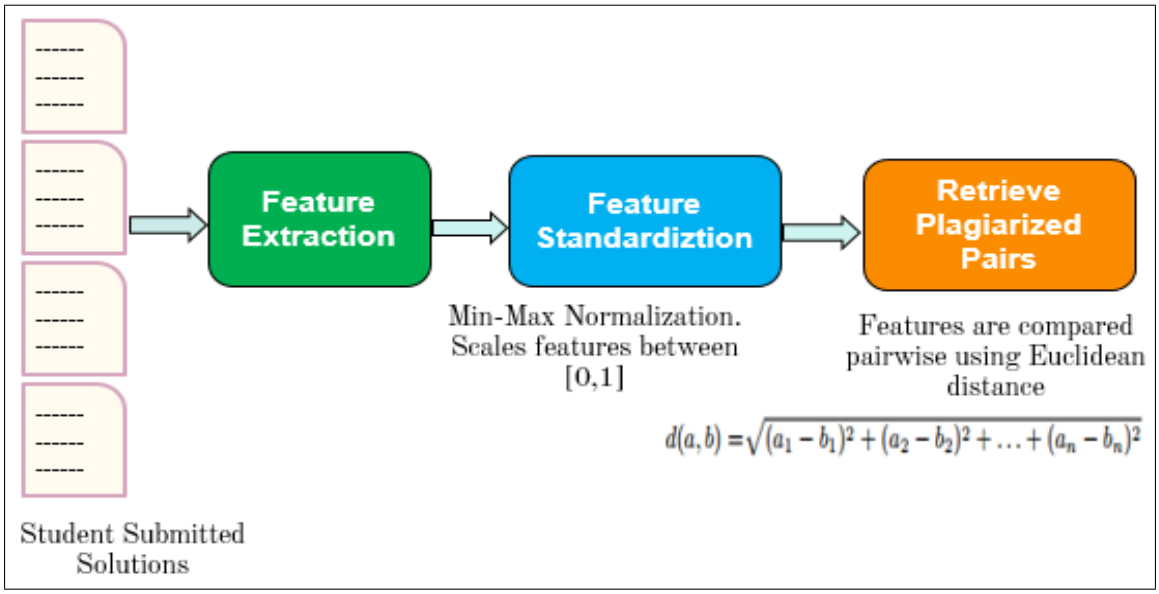


Figure 2.1: An overview of the proposed pipeline for unsupervised learning based approach.

2.1 Unsupervised Learning Based Approach

This section discusses our first approach to plagiarism detection in programming assignments using an unsupervised approach. Given a programming problem to solve (from an introductory computer science course) and a set of corresponding correct student submitted solutions written in C language, the task is to automatically detect all the plagiarized program submission pairs. Our proposed method automatically detects the solution pairs that are most susceptible to be the plagiarized pairs or the cheating cases.

Our method accepts as input a set of correct student solutions. Let $(X_1, X_2, ..., X_m)$ be the student submissions for a given problem, one submission per student. We extract source code metrics from the student solutions, use them as feature representations so that each student solution is mapped to a point in an n-dimensional space The feature representations from the solutions are then compared pairwise (computing for each pair a total similarity value). We consider student solutions that lie close to each other to be possible plagiarized cases. The closeness or similarity is defined by the *Euclidean distance* measure between the candidate solution pairs. Based on the pairwise Euclidean distance, we cluster together the similar solutions. If the pairwise euclidean distance value for a pair of solutions is less than or equal to some threshold (δ), then these pairs are more likely to belong to the same cluster. For acceptable values of δ (say δ = 0), the pairs are either full copy case or partial copy cases. Related experiments are discussed in section 2.4.2.

2.2 Supervised Learning Based Approach

When we began our work, we were attempting to automatically retrieve the plagiarized submissions in an unsupervised manner. However, we were also interested to use supervised learning approach. Using meaningful features, we can learn from these past student submissions and their relative labels (copy, not copy and partial copy) to predict labels for a pair of coding submissions from different students in the future which have not been graded by human graders. The labels make sense only for a pair of submissions and not for each individual student submission.

Explicitly enumerating all possible pairs for submissions in each problem set and annotating them is unfeasible. We needed an intelligent way to label only few pairs and still be able to train a classifier on these pairs of submissions. The scheme we followed is as mentioned below. There are 22 problem sets and from each problem set around top 100 pairs are picked (based on MOSS scores) for annotation purpose. And then for each feature, the top pairs which are not in top 100 pairs (that are picked earlier) are also picked and added to the dataset and annotated. The assigned labels are 0, 1, 2, where label 0 implies that a pair is *not a copy*, label 1 implies that a pair *partial copy* and label 2 implies that a pair is *surely a copy*. In total there are 3,600 solution pairs. Of this, 75% were used as training cases and 25% were held out for testing. The details of the dataset are given in table 2.2.

After having labeled data in hand, we train a Support Vector Machine (SVM) Classifier in a variety of settings using various types of features. Along with our source-code metrics discussed in the next section 2.3, we also use few text-based representations [11] extracted at syntactic level of a source code. We train SVM classifier using full feature set (source code based compiler level representations + text-based representations) and using only text-based representations. We also treat the classification problem as:

- a binary classification problem (including only copy/non-copy cases).
- a multiclass classification (includes partial copy cases also).

Related experiments covering unsupervised, supervised approaches are clearly discussed in section 2.4.

2.3 Feature Representations

Instead of using the source-code (raw representation), we want to engineer some features and use them as representations for a student submission. For our task, a good feature is one that helps in detecting plagiarism, even in presence of code obfuscation at the syntactic level. Hence, our idea is to design and use features that can alleviate the effect of code obfuscation and be able to detect the plagiarized student solutions.

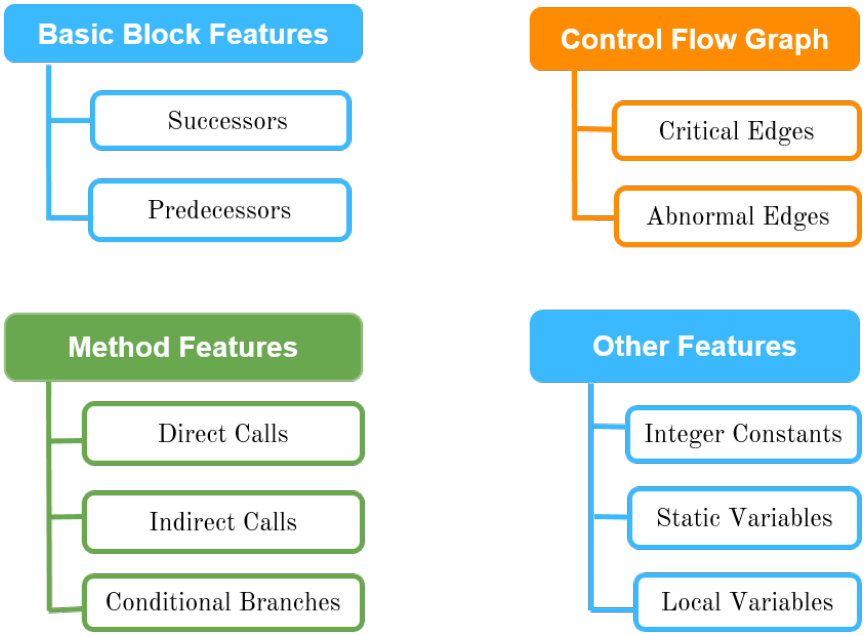


Figure 2.2: High-level overview of our proposed features.

With this intention in mind, we arrived at the idea of using source code metrics as feature representations for student submitted programming solutions. In order to avoid getting caught as plagiarized cases, students change the text of the code as that is the simplest thing that can be done with very less effort. However, the internal logic and implementation is many times the same and it can be captured by the intermediate representations after compilation (like edges in control flow graph). Instead of using text based approach, here we use source code metrics as static code-based features that are extracted using MILEPOST GCC [3] plug-in with O2 optimization. MILEPOST framework transforms GCC into a powerful machine learning enabled research infrastructure suitable for adaptive computing. It uses program feature extractor to modify internal optimization decisions.

2.3.1 Source-code based features

We have extracted fifty-five significant features. Each student submission is represented as a 55 dimensional feature vector. Each element in this vector captures a small characteristic of a submission that is easy to measure. The program features are typically a summary of the internal program representation and characterize the essential aspects of a program needed to distinguish copy and non-copy cases.

The features in our feature vector can be roughly divided into four subsets depending on the type of program characteristic they capture. These subsets can be found in figure 2.2. The first subset consists of twenty three features which are *basic block features*. A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. They describe a given

program based on the number of basic blocks, basic blocks with successors, predecessors, instructions etc.

The second subset consists of three features that can be termed as *control flow graph features*. These three features describe edges, critical edges and abnormal edges in the control flow graph. A critical edge is a control-flow edge from a basic block with multiple successors to a basic block with multiple predecessors. An abnormal edge is a control-flow edge whose destination is unknown. For example, exception handling produces these abnormal edges.

The third subset of eighteen features are *method features*. They contribute in capturing information related to methods. Some of these features are number of indirect calls (i.e. done via pointers) in the method, number of switch instructions the method, unary operations in the method etc. The fourth subset contain eleven features that capture characteristics like occurrence of integer constant zero, integer constant one static/local variables etc. For a complete list of features and their description please refer here [4]. The complete list of features with description is shown in table 2.1
An example showing code snippet along with few features are shown in the figure 2.3.

Feature Name	Description
ft1	Number of basic blocks in the method
ft2	Number of basic blocks with a single successor
ft3	Number of basic blocks with two successors
ft4	Number of basic blocks with more then two successors
ft5	Number of basic blocks with a single predecessor
ft6	Number of basic blocks with two predecessors
ft7	Number of basic blocks with more then two predecessors
ft8	Number of basic blocks with a single predecessor and a single successor
ft9	Number of basic blocks with a single predecessor and two successors
ft10	Number of basic blocks with a two predecessors and one successor
ft11	Number of basic blocks with two successors and two predecessors
ft12	Number of basic blocks with more then two successors and more then two predecessors
ft13	Number of basic blocks with number of instructions less then 15
ft14	Number of basic blocks with number of instructions in the interval [15, 500]
ft15	Number of basic blocks with number of instructions greater then 500
ft16	Number of edges in the control flow graph
ft17	Number of critical edges in the control flow graph
ft18	Number of abnormal edges in the control flow graph
ft19	Number of direct calls in the method
ft20	Number of conditional branches in the method
ft21	Number of assignment instructions in the method
ft22	Number of binary integer operations in the method

ft23	Number of binary floating point operations in the method
ft24	Number of instructions in the method
ft25	Average of number of instructions in basic blocks
ft26	Average of number of phi-nodes at the beginning of a basic block
ft27	Average of arguments for a phi-node
ft28	Number of basic blocks with no phi nodes
ft29	Number of basic blocks with phi nodes in the interval [0, 3]
ft30	Number of basic blocks with more then 3 phi nodes
ft31	Number of basic block where total number of arguments for all phi-nodes is in greater then 5
ft32	Number of basic block where total number of arguments for all phi-nodes is in the interval [1, 5]
ft33	Number of switch instructions in the method
ft34	Number of unary operations in the method
ft35	Number of instruction that do pointer arithmetic in the method
ft36	Number of indirect references via pointers (”*” in C)
ft37	Number of times the address of a variables is taken (”&” in C)
ft38	Number of times the address of a function is taken (”&” in C)
ft39	Number of indirect calls (i.e. done via pointers) in the method
ft40	Number of assignment instructions with the left operand an integer constant in the method
ft41	Number of binary operations with one of the operands an integer constant in the method
ft42	Number of calls with pointers as arguments
ft42	Number of calls with the number of arguments is greater then 4
ft44	Number of calls that return a pointer
ft45	Number of calls that return an integer
ft46	Number of occurrences of integer constant zero
ft47	Number of occurrences of 32-bit integer constants
ft48	Number of occurrences of integer constant one
ft49	Number of occurrences of 64-bit integer constants
ft50	Number of references of a local variables in the method
ft51	Number of references (def/use) of static/extern variables in the method
ft52	Number of local variables referred in the method
ft53	Number of static/extern variables referred in the method
ft54	Number of local variables that are pointers in the method
ft55	Number of static/extern variables that are pointers in the method

Table 2.1: Complete list of features with description

```c
int main()
{
        int T;
        scanf("%d",&T);
        while(T--)
        {
                int n;
                scanf("%d",&n);
                int x,min=0;
                int count=0;
                int i;
                scanf("%d",&x);
                n--;
                min = x;
                for(i=0;i<n;i++)
                {
                        scanf("%d",&x);
                        if(x>=min)
                                count++;
                        else
                                min = x;
                }
                printf("%d\n",count);
        }
        return 0;
}
```

Figure 2.3: Consider the sample code snippet show in here. For this code snippet, the total number of basic blocks are 12, the number of basic blocks with a single successor are 6 and the number of basic blocks with a single predecessor are 7 respectively. As there is no exception handling in the code, there are no abnormal edges in its control flow graph. However, the number of edges in the control flow graph is 14. Coming to method features, the number of instructions are 24, the number of switch instructions in the method is 0 as there are no switch case statements and the number of direct calls in the method are 5. The number of local variables referred in the method are 13. From the code we can verify that the number of occurrences of integer constant zero is 4.

2.3.2 Text based features

Along with our source-code metrics discussed above, we also use few text-based representations [11] extracted at syntactic level of a source code. These features are inherently pairwise features. The features are described below:

- Difference in Length of Submissions (DLS) - Captures the difference between the lengths of each student submission.
- Similarity as Measured by Diff (SMD) - Measures the number of lines of common code in original submissions. Extracted using *diff* command.
- Similarity of Comments (SOC) - Measures the number of comments in common. Extract comments from student submissions and use 'diff'.
- Similarity in String Literals (SSL) - Measures the similarity between the two sets of literals one from each submission.

- Edit Distance (ED) - Treat each student submission as a string and measure the similarity between two strings. Edit distance is the number of deletions, insertions, or substitutions required to transform source string 's' into target string 't'.

2.3.3 Feature Standardization

Standardization of datasets is a common requirement for many machine learning estimators. The features might behave badly if they are not standardized. For example, in case of Support Vector Machines (SVM), if the features are not standardized and a feature has a variance that is orders of magnitude larger than others, it might dominate the objective function and make the SVM estimator unable to learn from other features correctly as expected. One way to standardize the features is to scale the features to lie between a given minimum and maximum value. We used min-max scaling which scales the features between [0,1], or so that the maximum absolute value of each feature is scaled to unit size.

2.3.4 Creating Pairwise Features

We extract the features that represent each individual student solution. Each feature vector is of sixty dimensions. The first fifty-five elements in each vector are features extracted from intermediate compiler representation of a source code. The last five elements are the text-based representations extracted at syntactic level of a source code. We took the individual point-wise feature representation of each solution in a pair reported by MOSS and computed an element-wise difference between them so that the resultant vector can be used as a pair-wise representation of the solution pair. Text based features are inherently pairwise features.

2.4 Experiments

2.4.1 Dataset and Evaluation Measures

The dataset we adopted is a collection of student submitted solutions to assignments from an introductory C programming course. We call this dataset CP assignments dataset. The CP dataset is formed out of 22 problem-sets or questions. In each problem-set there are 70 to 250 student submissions. Totally there are about 4,700 submissions. Out of these around 80 pairs are plagiarized, 110 pairs are partial-copy and the rest are non-copy. The questions asked in the problem-sets range from more specific (as in case of tree-traversal) to diverse. Each problem set is of varying difficulty, with student solutions ranging from 50 to 400 lines of code. The details of the dataset are given in table 2.2.

Dataset	Problem Sets	Submissions	Language	Average LOC
CP Assignments	22	4.7K	C	106

Table 2.2: Dataset details

Evaluation Measures

To analyze the classifier performance in supervised learning approaches, we compute *precision*, *recall* and *f1-score* of each label in a dataset. Suppose a label y_i is present in ground-truth of m_1 instance pairs and it is predicted for m_2 pairs while testing out of which m_3 instance pairs are correct. Then its precision will be = m_3/m_2 and recall will be = m_3/m_1. Using these two measures, we get the percentage F1-score F1 = 2.P.R/(P+R), where P is precision and R is recall. F1-score takes care of the trade-off between precision and recall.

2.4.2 Experiment - 1: Retrieve Plagiarized Pairs - Unsupervised Approach

For a given query program submission P_i, we need to find the neighbors P_j which are at a distance δ. The feature representations from the solutions are extracted and then compared pairwise (computing for each pair a total similarity value). We consider student solutions that lie close to each other to be possible plagiarized cases. The closeness or similarity is defined by the Euclidean distance measure between candidate solution pairs. Based on the pairwise Euclidean distance, we cluster together the similar solutions. Using cross-validation We have identified that the solution pairs as copy cases by using a distance threshold δ = 0. With the current threshold we are able to identify and retrieve exact copy cases as well as some partial copy cases. With our current threshold we observe that the retrieved pairs at-least have some similarity in the logic.

The results are shown in figure 2.4. From results, we can observe the results of our method compared with MOSS. When comparing our method with MOSS, we have to decide two thresholds namely: a threshold on pairwise Euclidean distance and a threshold on the obtained MOSS scores. Depending on the domain knowledge (here the type of problem asked to solve in the problem set) these thresholds vary. The threshold on pairwise Euclidean distance selected we used is zero (δ = 0). The thresholds on MOSS scores is 90% and 80% respectively. The red and yellow bars in figure 2.4 represent the MOSS performance when the threshold on score is 90% and 80% respectively. The green colored bars are the ground truths provided by teaching assistants of the course. Depending on the distance threshold (δ), we can report more confident and refined accurate results. The performance of our method with different values of δ is shown in the figure 2.5.

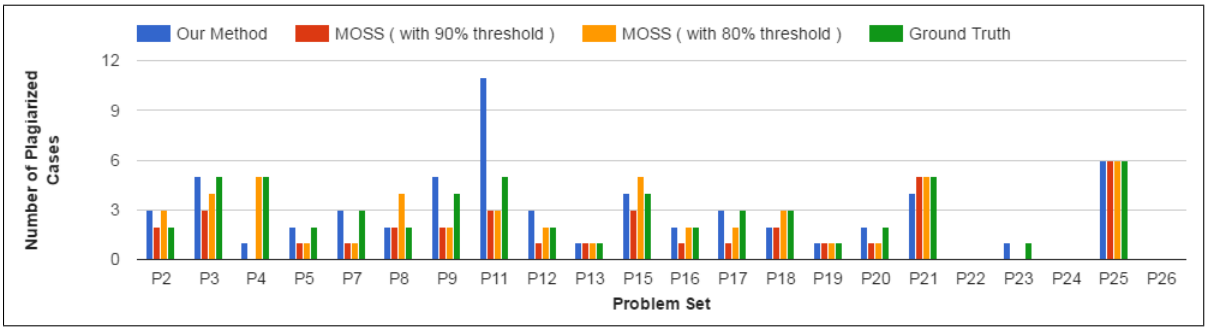


Figure 2.4: A plot showing the performance of our method when compared with MOSS. The blue colored bars represent the performance of our method, the red and yellow bars represent the MOSS performance when the threshold on score is 90% and 80% respectively. The green colored bars are out ground truths provided by teaching assistants of the course. Our method has a recall of 0.924 whereas MOSS with 80% and 90% threshold has a recall of 0.819 and 0.631 respectively.

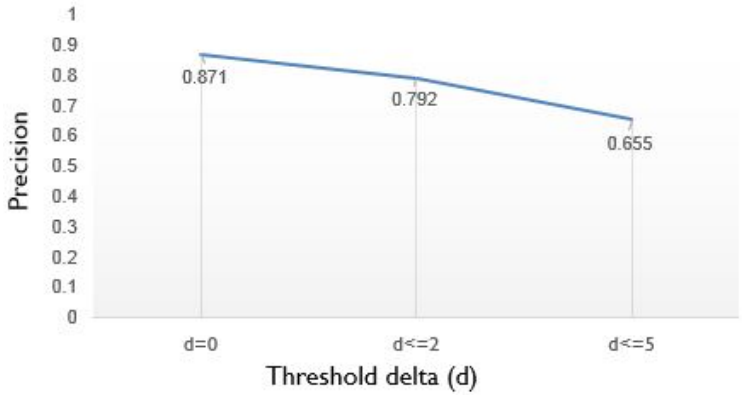


Figure 2.5: The effect of delta (δ) on the precision value of our method.

2.4.3 Experiment - 2: Binary Classification

In supervised learning based approach our first idea is to pose the problem as a binary classification problem. So we removed the instances belonging to the class of partial copy cases from the dataset. At this point, we trained a *Support Vector classifier*, using *Sci-kit Learn* Python library. Table 2.3 shows the number of instances belonging to each class in train and test dataset respectively. From there, we opted for 4-fold cross-validation, training on 3/4 of the dataset and testing on the remaining 1/4 for all 4 folds was run on approximately on 3,400 instances. However, even after regularizing we found that our classifier was choosing to predict the class with the highest frequency. We can notice from table 2.3 that our dataset suffers from class-imbalance problem.

Class Label	No. of Train Instances	No. of Test instances
0 (Not Copy)	2515	839
2 (Copy)	57	19

Table 2.3: Table showing number of instances belonging to each class in train and test dataset

Improvements - Assign class-weights

Our first thought was to use a *class weighing* scheme to alleviate the effect of class imbalance. The class weighing scheme forces the classifier to learn the decision boundary in such a way that the misclassification of an instance from an under-represented is more costlier than the misclassification of an instance from an over-represented class. Here, the class weights are inversely proportional to class frequencies in the dataset. A Support Vector Classifier with *'rbf'* kernel is trained on the training dataset. Using 4-fold cross validation the values of the parameters *C* and *gamma* were chosen to be *100* and *1.0* respectively. The classifier was trained to improve the *macro recall* score. The cross-validation *recall* score was *0.79*.

After the classifier is trained, it is tested on the test dataset. The performance of the classifier on the test dataset is measured using the metrics *precision, recall, f1-score*. On the test set the *f1-score* was *0.783*. Table 2.4 shows the various metrics used to measure the per-label performance of the classifier. The table 2.5 shows the confusion matrix for the support vector classifier.

Class Label	precision	recall	f1-score
0	0.99	0.98	0.99
2	0.50	0.68	0.58

Table 2.4: Table showing per label performance of the Support Vector classifier on the test dataset using full feature set.

True Label	**Predicted Label**	
	0 (Not Copy)	**2 (Copy)**
0 (Not Copy)	826	13
2 (Copy)	6	13

Table 2.5: Table showing confusion matrix for Support Vector Classifier on test dataset using full feature set.

2.4.4 Experiment - 3: Binary Classification Using Text Based Features

As mentioned earlier, our feature set consists of text-based representations [11] which are extracted at the syntactic level of a source code. A shell script, which extracts features of the code and produces a text based feature vector per pair of submission, was run across all annotated pairs of submissions. These features are inherently pairwise features. We trained a classifier using the text features alone. This provided us valuable insights on the role of compiler level features in the full feature set (intermediate compiler representations and text-based features).

A Support Vector Classifier with *'rbf'* kernel is trained on the training dataset. Using 4-fold cross validation the values of the parameters *C* and *gamma* were chosen to be *500* and *8.0* respectively. The classifier was trained to improve the *macro recall* score. The cross-validation *f1-score* was *0.673*. After the classifier is trained, it is tested on the test dataset. The performance of the classifier on the test dataset is measured using the metrics *precision, recall, f1-score*. On the test set the *f1-score* was *0.652*. Table 2.6 shows the various metrics used to measure the per-label performance of the classifier. The table 2.7 shows the confusion matrix for the support vector classifier.

Class Label	precision	recall	f1-score
0	0.99	0.93	0.96
2	0.19	0.79	0.31

Table 2.6: Table showing per label performance of the Support Vector classifier on the test dataset text based features only.

True Label	**Predicted Label**	
	0 (Not Copy)	**2 (Copy)**
0 (Not Copy)	777	62
2 (Copy)	4	15

Table 2.7: Table showing confusion matrix for Support Vector Classifier on test dataset using text based features only.

2.4.5 Experiment - 4: Multi-class Classification

As mentioned earlier, we also wanted to include the instances from the class of partial copy cases to our dataset. Thus, we pose the problem as a multi-class classification problem. Table 2.8 shows the number of instances belonging to each class in train and test dataset respectively. We can notice that the dataset suffers from extreme class-imbalance problem.

Class Label	No. of Train Instances	No. of Test Instances
0 (Not Copy)	2516	838
1 (Partial Copy)	83	30
2 (Copy)	58	18

Table 2.8: Table showing the number of instances belonging to each class in train and test dataset respectively

We proceed by training a support vector classifier in a one-versus-rest fashion. Using 4-fold cross validation the values of the parameters *C* and *gamma* were chosen to be *15* and *2.0* respectively. The classifier was trained to improve the *macro recall* score. The cross-validation *recall* was *0.63*. After the classifier is trained, it is tested on the test dataset. The performance of the classifier on the test dataset is measured using the metrics *precision, recall, f1-score*, which is shown in the tables 2.9 and 2.10 . On the test set the *recall* was *0.573*.

Class Label	Precision	Recall	f1-score
0 (Not Copy)	0.97	0.84	0.90
1 (Partial Copy)	0.07	0.27	0.11
2 (Copy)	0.28	0.61	0.38

Table 2.9: Table showing per label performance of the Support Vector classifier on the test dataset using full feature set.

True Label	Predicted Label		
	0 (Not Copy)	1 (Partial Copy)	2 (Copy)
0 (Not Copy)	707	106	25
1 (Partial Copy)	18	08	04
2 (Copy)	05	02	11

Table 2.10: Table showing confusion matrix for Support Vector Classifier on test dataset using full feature set.

2.4.6 Experiment - 5: Multi-class Classification Using Text Based Features

A Support Vector Classifier with *'rbf'* kernel is trained on the training dataset. Using 4-fold cross validation the values of the parameters *C* and *gamma* were chosen to be *1000* and *512.0* respectively. The classifier was trained to improve the *macro recall* score. The cross-validation *recall* was *0.532*. After the classifier is trained, it is tested on the test dataset. The performance of the classifier on the test dataset is measured using the metrics *precision, recall, f1-score*. Table 2.11 shows the various metrics used to measure the per-label performance of the classifier. The table 2.12 shows the confusion matrix for the support vector classifier. On the test set the *recall* was *0.49*.

Class Label	Precision	Recall	f1-score
0 (Not Copy)	0.96	0.83	0.89
1 (Partial Copy)	0.05	0.20	0.08
2 (Copy)	0.23	0.44	0.30

Table 2.11: Table showing per label performance of the Support Vector classifier on the test dataset using text based features only.

True Label	Predicted Label		
	0 (Not Copy)	1 (Partial Copy)	2 (Copy)
0 (Not Copy)	694	119	25
1 (Partial Copy)	22	06	02
2 (Copy)	08	02	08

Table 2.12: Table showing confusion matrix for Support Vector Classifier on test dataset using text based features only.

2.4.7 Experiment - 6: Feature Visualization

Having proposed source-code metrics as feature representations of student solutions, we are interested in answering the question "*What information is captured by the extracted static code-based features and what it is not capturing*?". For this purpose, we used t-SNE [28], which can help us in developing and evaluating our feature representations. The intuition to use t-SNE came from its inherent ability to preserve neighborhood identity and semantic similarity in low-dimensions. We claim that if our feature representations are good enough, clustering using these representations should give us well defined clusters, each cluster containing all the student solutions that used similar logic to solve a

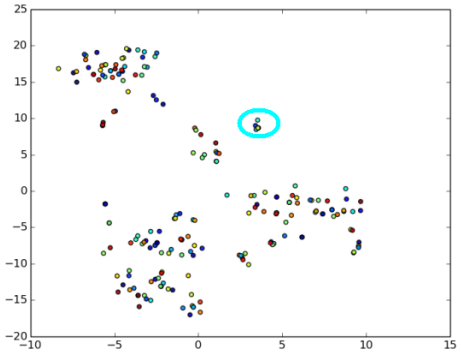


Figure 2.6: Low-dimensional embedding of the submissions for swapping problem, obtained by using t-SNE

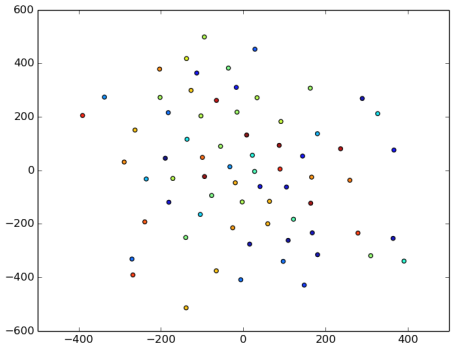


Figure 2.7: Low-dimensional embedding of the submissions for a heap problem, obtained by using t-SNE

given problem. We arrived at some interesting conclusions like finding out explicit diversity in student solution approaches i.e. different ways in which students developed a solution to the given problem. The presence of well defined clusters in low-dimensional visualizations demonstrate that our features are capable of capturing this interesting result. For example, consider the scatter plots shown in figure 2.6 and 2.7 . The mapped student solutions in the cluster circled in blue color used two variables to achieve swapping. Similarly, they make use of register variables to speed up sorting. While, all other solutions used three variables to achieve swapping. The student solutions corresponding to figure 2.7 are more diverse. The spread of the data in figure 2.6 along both the axis is more when compared to spread of data in figure 2.7 The spread of the features in the two-dimensional map itself reveals the nature of the solution submitted (the scatter plot looks cluttered for a problem which can be solved in a limited number of ways and wide spread for a problem which can be solved in multiple ways).

2.4.8 Discussion

In most of the cases, our results are consistent with scores from MOSS. The potential plagiarized solution pairs detected by our method falls in top-five cases detected by MOSS. There are solution pairs where MOSS does not show any similarity and the pairwise distance is comparatively very low. However, our proposed method does a better job in most cases. From figure 2.4, we can observe the results of our method compared with MOSS.

We analyze our results in the cases presented below:

Case1: *Absence of plagiarized cases*

From figure 2.4, we can observe that there are no reported copy cases for problem sets P22, P24 and P26. Similarly, for these problem sets, the similarity scores of MOSS are also very low (less than 50%).

Case2: *Interchanging if-else code*

Measure	Ours	MOSS (90%)	MOSS (80%)
Precision	0.871	1.000	0.920
Recall	0.924	0.631	0.819
F1 Score	0.896	0.773	0.866

Table 2.13: Performance of our method in comparison to MOSS with various thresholds.

If the code contains *if-else* blocks and if the conditions are interchanged then MOSS does not give us any acceptable similarity score as a plagiarized case. However, our feature vector does not differ in this case.

Case-3: *Type define the frequently called functions*

In one of code if the frequently called functions like *printf* and *scanf* are type defined, then MOSS shows only similarity score of 50-60% but it should be a 100% copy case. In this scenario, our method works perfectly well, resulting in a pairwise distance value of zero.

Case-4: *Presence of dead code*

If dead code is added to one of the codes the MOSS score is very low. In such cases, compiler optimization can be used to extract the features which can successfully eliminate the effect of dead code. The example in figure 1.1 (b), shows a code snippet containing dead code.

Case-5: *Interchange the position of functions*

If function code is interchanged MOSS shows 60-70% similarity for copy pairs where there is an exact match of the code. However, our measured pairwise distance is zero which makes it easy to detect the copy cases.

All the above five cases provide enough proof about the robustness of our features and its ability to detect plagiarized cases even in obfuscated solution pairs. Table 2.13 demonstrates the superiority of our method when compared to MOSS.

Role of Text Based Features

The inclusion of compiler level features improves the overall f1-score of the classifier by 13% in binary setting and 4% in multiclass classification setting. In multiclass classification setting, the distinction between instances belonging to class 1 and 2 is subjective.

2.4.8.1 Importance of Features

We assess the importance of each feature using Mutual Information (MI). MI measures the dependency between two features and assigns a non-negative value. The higher the MI value, the higher the dependency. The figure 2.8, we can observe that almost 6 features are of no use (last 6 features from

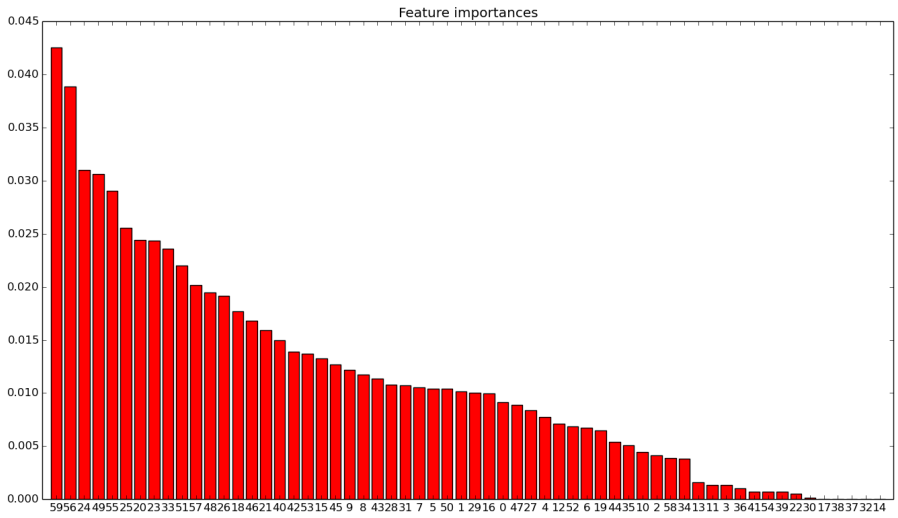


Figure 2.8: Feature importance using Mutual Information (MI).

the figure). These corresponds to the features describing the basic blocks, address of a functions and phi-nodes etc. All the text based features appear within top 15 important features. Three out of five text based features appear in top five important features. The features 'Average of number of instructions in basic blocks' and 'Number of references of a local variables in the method ' are the most important among source-code based compiler level features.

2.5 Summary

Static program features extracted from the intermediate representations during the various phases of compilation process are used successfully to address the compiler phase optimization problem. In this work, we explored the possibility of using those features for plagiarism detection and, mining programming patterns and the associated visualizations. Our experiments suggests that the approach is very promising and can be used in many different ways.

Chapter 3

Plagiarism Detection in Programming Assignments Using Deep Features

3.1 Introduction

The task of plagiarism detection can be treated as assessing the amount of similarity presented within given entities. These entities can be anything like documents containing text, source-code etc. Plagiarism detection can be formulated as a fine-grained pattern classification problem. The detection process begins by transforming the entity into feature representations. These features are representatives of their corresponding entities in a discriminative high-dimensional space, where we can measure for similarity. Here, by entity we mean solution to programming assignments in typical computer science courses. The quality of the features determine the quality of detection.

Popular source-code plagiarism detection tools such as MOSS [44] or JPlag [38] rely on analysis at textual level, and compare two student submitted assignment solutions for signs of plagiarism. These tools are based on n-gram techniques and create fingerprints to measure for similarity between submissions. Moreover, these tools ignore some hints like similar comments and white-spaces which are important cues for plagiarism [12]. Moreover, these approaches to plagiarism detection can not capture non-contiguous interaction present in the code and they can only tolerate small local changes. Simple obfuscation, such as noise injection, can evade from being detected as copy cases [32].

We aim at automatically learning feature representations for solutions submitted by students to programming assignments that can capture non-contiguous interactions. We believe that features which capture such interactions can detect plagiarism even if the source-code is obfuscated. However, programs are highly structured in nature and applying machine learning directly to data in the form of programs is difficult. We overcome this difficulty by exploiting the fact that Programming Language Processing (PLP) can be treated similar to Natural Language Processing (NLP). Utilizing the "naturalness" in source-code [19] and treating programs as natural language, the traditional n-gram models may not be sufficient to model the non-consecutive interaction within n-grams present in the programs. This demands for powerful models that can capture long-term interactions.

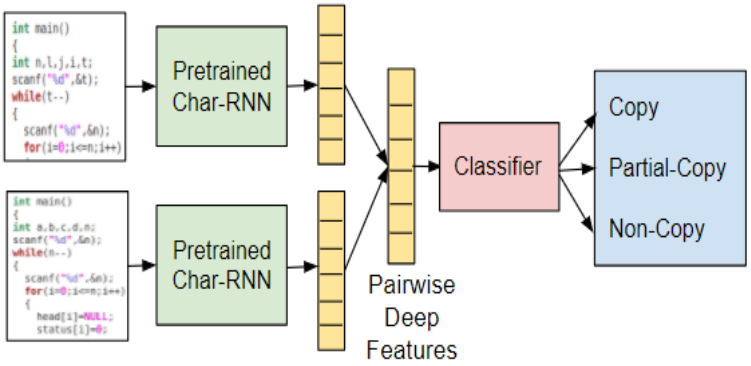


Figure 3.1: Our model, consists of a char-RNN followed by a SVM classifier. From two individual program submissions, pairwise deep features are obtained and they are classified as copy/partial-copy/non-copy.

Inspired by recent success of deep neural networks for learning features in other domains like Computer Vision (CV) and NLP, we are motivated to use such networks that enable us to learn the representations for the programs. Tasks like image classification [24], sequence prediction [15], image-captioning [51] and machine translation [46] are witnessing impressive results by leveraging the ability of deep neural networks to learn features from large amount of data. Similarly, deep learning has been used in domains of and Programming Languages (PL) and Compilers. Areas like program synthesis [35], induction, iterative compilation and compiler optimization [10] have benefited significantly with the use of deep neural networks.

In this work, we employ a character-level language model to map the characters in a source-code to continuous-valued vectors called *embeddings*. We use these program embeddings as *deep features* for plagiarism detection in programming assignments. These deep features can capture the non-consecutive interaction within n-grams present in programs at a syntactic level. These features are hierarchical in nature and learned through a series of non-linear transformations modeled by deep neural networks. These features are non-linear, *generic* which can generalize well [5]. We demonstrate the fact that these deep features are generic. By generic we mean that although they are learned using the task of character level sequence-prediction, they are directly applied on different dataset and to a different task like code plagiarism detection without the need to fine-tune on each individual problem-set.

Recently, character-level Recurrent Neural Networks (char-RNN) have been shown to generate text [45] and Linux C code [23]. Recent works also show that RNNs are particularly good at modeling syntactical aspects, like parenthesis pairing, indentation, etc. We prefer RNNs since they are inherently suitable for modeling sequences, attributed to their iterative nature. However, RNNs are difficult to train because of the *vanishing gradient* and *exploding gradient* [36] problems. To alleviate these problems, the Long short-term memory (LSTM) [20] units have been proposed. These LSTM units have special gated mechanism that can retain the previous state and memorize new information at the current time step. With the help of memory and different gates in LSTM, it enables us to capture non-consecutive interactions

at syntactic level [52]. Hence, we use LSTM units in char-RNN to learn feature representations for programs. The detailed architecture of char-RNN model is described in section 3.4.2. The contributions of our work mentioned in this chapter are as follows:

- We learn features that can capture non-contiguous interactions among n-grams present in the source-code, using deep neural networks.
- We demonstrate the generic nature, robustness and the superiority of our deep features when compared to source-code metrics and textual features in the task of plagiarism detection.

We evaluate the performance of the proposed features to detect plagiarism in programming assignments. Comparing our proposed features with popular handcrafted features (both source-code metrics and textual features), we report *f1-score* improvement of 9.5% for binary classification (copy/non-copy) and 5% for three-way classification (copy/partial-copy/non-copy) tasks respectively. Figure 3.1 demonstrate our approach.

3.2 Background and Related Work

3.2.1 Language Modeling

The char-RNN used in our approach is an implicit statistical language model. A statistical language model is a probability distribution over sequences of words in a sentence. The goal of statistical language modeling is to predict the next word in textual data given context [30]. Traditional language models use the chain rule to model joint probabilities over word sequences as:

$$p(w_1, \ldots, w_N) = \prod_{i=1}^{N} p(w_i | w_1, \ldots, w_{i-1}) \tag{3.1}$$

Many NLP tasks are performed using these statistical language models. Very recently, they have been put to use for many Software Engineering (SE) tasks such as code suggestions and fixing programming errors [16]. Hindle et al. [19] were the first to apply NLP techniques to source-code. They demonstrated that code written by humans is also likely to be repetitive and predictable like natural language. Hence, the code written by humans can be successfully modeled by statistical language models. Such models can be used to assist humans in code suggestions and design.

3.2.2 Related Work

Existing code plagiarism detection techniques can be put into four categories [9]. They are metric based, token based, tree based and Program Dependency Graph (PDG) based. However, each of these methods has some limitations. Metric based approaches fail because most of the metrics are highly

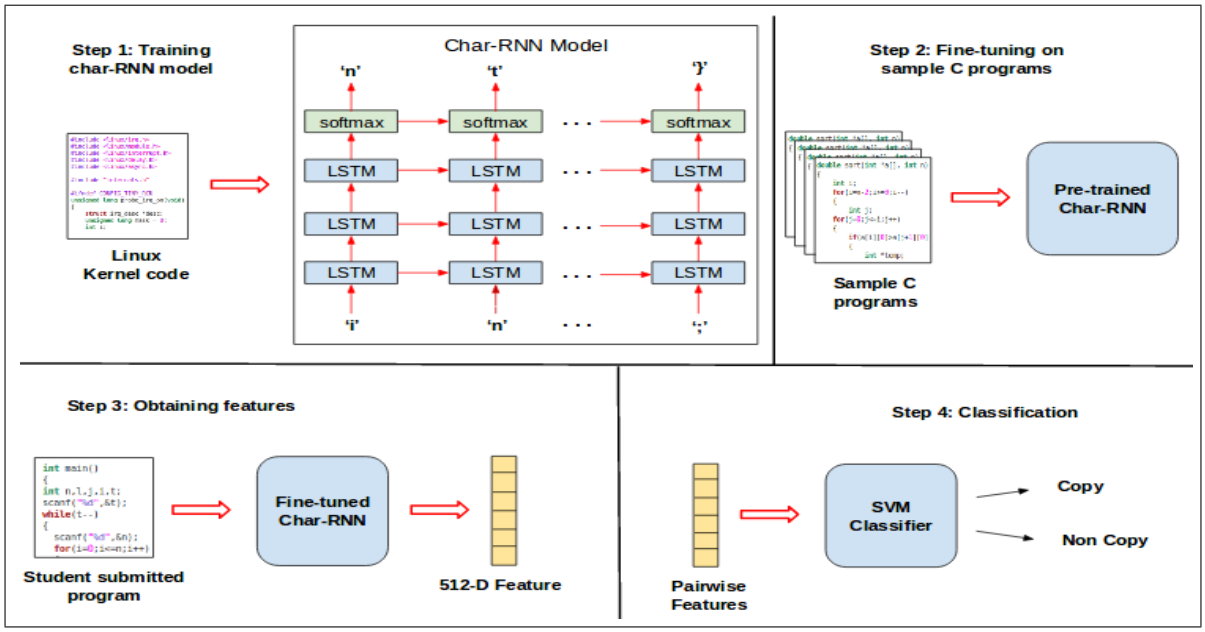


Figure 3.2: Our proposed approach step by step. First, we train a char-RNN model on the Linux Kernel source-code, then we fine-tune it to some sample C programs. Later, we use this fine-tuned model to obtain embeddings for programming solutions submitted by students and use these embeddings as features to detect plagiarized cases.

sensitive to minor edits. Token based approaches cannot be applied to a repository containing multiple program submissions from different problem-sets. Abstract Syntax Tree (AST) based approaches produce many false-positives because of abstraction of the original code. PDG based approaches are costly as they involve comparing graphs.

In [12], the authors proposed a system that is based on properties of assignments that course instructors use to judge the similarity of two submissions. They proposed 12 textual features like similarity in comments and white-space etc. However, both MOSS and JPlag filter out these features when performing their analysis. Their main motivation is to use these features as cues in plagiarism detection. This system uses neural network based techniques to measure the relevance of each feature in the assessment. Their focus is on detecting plagiarized pairs within a single problem set. However, our proposed method is independent of problem set. The approaches proposed in [13] are based on searching similar n-grams or small character sequences (strings) between two source-codes. However, n-grams have their own limitations. Since they are simply frequency count of term co-occurrences, they are limited by the amount of context they consider [22].

In all the papers mentioned above, the main idea is to hand-craft certain features to capture similarity between two code submissions by exploiting either syntactical and stylistic (spaces, comments) aspects. Some of these are highly based on knowledge from generic programming constructs. Importantly, such handcrafted features are suitable only for specific tasks. This feature engineering itself is a fundamental

point of difference of our work, when compared with other works. Here, we learn discriminative features from the data, which are generic in nature as opposed to specifying and engineering a set of specific features. A recent paper [34] is closely related to ours, in which the authors proposed a Tree-based Convolutional Neural Network (TBCNN), that can be applied on program AST. They use this architecture to classify programs by functionality and also to detect code snippets of certain patterns. However, features extracted from AST alone cannot be used for plagiarism detection, since we may loose some important cues like misspelled comments and white spaces [12].

3.3 Approach

Our proposed approach can be summarized into four steps as demonstrated in figure 3.2.

3.3.1 Step 1: Train a char-RNN model

The first step in our approach is to exploit the "naturalness" in source-code by treating it as natural language and training the char-RNN [23] model on *Linux Kernel* source-code. A char-RNN is a character-level language model that accepts a sequence of characters as input and the model is trained to predict the next character in sequence at each time step. Consider there are V unique characters in Linux Kernel source-code. Each of the characters from the source-code is encoded as one-hot-vectors of V-dimensions. Now, a sequence of one-hot vectors $x = \{ x_1, x_2, \ldots, x_T \}$ are fed as input to a recurrent neural network, where T is the last time-step of the sequence. The model first computes a sequence of hidden vectors at each hidden layer. Let $h^L = \{ h_1, h_2, \ldots, h_T \}$ be the sequence of N-dimensional hidden vectors computed by the model at last layer. The last layer activations from the model are projected using a $[V \times N]$ weight matrix W_y to a sequence of output vectors $y = \{ y_1, y_2, \ldots, y_T \}$ to predict for next character in the sequence. However, these output vectors hold unnormalized log probability for next character in the sequence, which is normalized by employing the softmax function at the output layer. The model is trained using cross-entropy loss and we use LSTM units in place of vanilla RNN in our char-RNN. The parameter settings of the model and other training details are mentioned in section 3.4.2.

3.3.2 Step 2: Fine-tuning char-RNN model

The second step is to use this pre-trained char-RNN model to fine-tune it to sample C programs. For this task, we have considered programming submissions from 4 problem-sets picked from 104-class program dataset [34]. In this step, the model is adapted to less complex C programs (unlike Linux kernel source-code). More details on fine-tuning are presented in section 3.4.2.

Dataset	#problem sets	#code submissions	language
D1 (*this work*)	22	4,700	C
D2 [34]	104	52,000	C

Table 3.1: Overall information of datasets used. The dataset D2 refers to the 104-class program dataset

3.3.3 Step 3: Obtaining Program Embeddings

The third step is to use this fine-tuned model to obtain feature representations for programming solutions submitted by students. The programs used here are picked from the dataset we created (see section 3.4.1) and importantly, these are completely different from the ones used to fine-tune the model in the previous step. We consider the hidden-vectors from the last LSTM layer as the feature representations of the programs. A program is represented by the average of last layer LSTM hidden vectors from each character (i.e at hidden vector at each time step) similar to [48]. The embeddings obtained from the model are 512-D vectors. More details on features are given in section 3.4.3.

3.3.4 Step 4: Classification

From the individual program feature representations, we construct pair-wise features and classify the submissions as copy/partial-copy/non-copy cases. Figure 3.3 demonstrates the process of construction of pair-wise features from individual program pair. These details are mentioned in section 3.4.3. However, plagiarism cases only make up a small percentage student submissions in our dataset. The dataset is highly class imbalanced. To overcome class imbalance we use standard techniques like class-weighing scheme. These details are mentioned in section 3.4.5.

3.4 Experiments

3.4.1 Datasets

The dataset we appropriately adopted is a collection of assignment solutions submitted by students from an introductory C programming course. We call this dataset as D1. Our dataset is formed out of 22 problem-sets. In each problem-set there are 70 to 250 student submissions. Program solutions falling in the same problem-set are functionally consistent. In total, there are about 4,700 submissions. The questions asked in the problem-sets range from more specific (as in case of tree-traversal) to diverse. Each problem-set is of varying difficulty, with student solutions ranging from 50 to 400 lines of code.

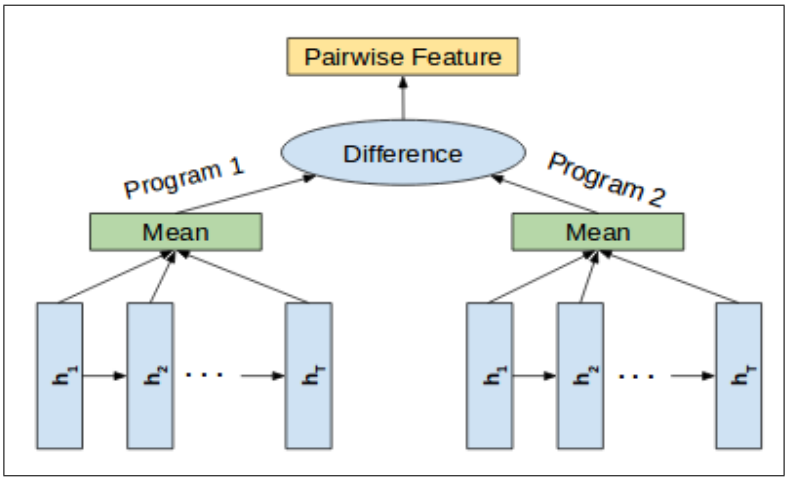


Figure 3.3: A schematic diagram showing our pairwise feature construction from two program submissions. The hidden vectors are from the last layer of char-RNN model.

The programs in our dataset are obfuscated in the following ways: variable name changing, careful conversion of while loop into for loop, dead code injection etc.

To train a classifier for plagiarism detection, data pairs (pairs of student's submissions) are needed. The labels make sense only for a pair of submissions and not for individual student submission. Explicitly enumerating all possible pairs for submissions in each problem set and annotating them is not feasible. Hence, we relied on MOSS scores and handcrafted features [53] for creating pairs. The constructed data pairs were annotated by teaching assistants. In total, there are about 3,600 program pairs. Out of these around 80 pairs are plagiarized, 110 pairs are partial-copy and the rest are non-copy. Table 3.1 provides the information about the datasets used.

For training our character-RNN model, we used the source-code of *Linux Kernel*. All the files are shuffled and concatenated to form a 6.2 Million character long dataset. The maximum sequence length used for training LSTM is 50. During preprocessing stage, all the unicode characters are removed from the dataset. For fine-tuning our pre-trained char-RNN model, we used submissions from 104-class program dataset.

3.4.2 Char-RNN Training and Fine-tuning

We trained the char-RNN model on the source-code of *Linux Kernel* for sequence prediction task. The size of vocabulary is 96. Cross-entropy loss was used to train the model. Our char-RNN model consists of three hidden LSTM layers stacked on top of each other. Each of the layer consists of 512 LSTM cells/units. We used mini-batch stochastic gradient descent with RMSProp [50]. All the other hyper-parameters are same as mentioned in [23]. This pre-trained model was then *fine-tuned* to adapt it to the less complicated C/C++ program submissions, taken form from 4 problem-sets from 104-class

program dataset. We call this dataset as D2. During fine-tuning, we freeze the weights of first two hidden layers and only the last hidden layer weights are allowed to update. Early stopping is used based on the validation performance.

3.4.3 Constructing Pairwise Representations

Given the feature representations of two individual programs from a problem-set, the pairwise features are constructed by taking the element-wise difference between individual program feature representations. Figure 3.3 demonstrates the process of construction of pair-wise features from individual program pairs.

3.4.4 Evaluation Measures

To analyze the classifier performance in supervised learning approach, we compute per-label *precision*, *recall* and *f1-score*. Suppose a label y is present in the ground-truth of m_1 instance pairs and it is predicted for m_2 pairs while testing out of which m_3 instance pairs are correct. Then its precision will be = m_3 /m_2 and recall will be = m_3 /m_1. Using these two measures, we get the percentage F1-score as $F1 = 2.P.R/(P + R)$, where P is precision and R is recall. After calculating precision, recall and f1-score metrics for each label individually, we calculate their unweighted mean.

3.4.5 Results

After obtaining pairwise feature representations for given program pairs, we train a Support Vector Machine (SVM) classifier in a variety of settings namely using deep features, full feature set (source-code metrics + textual features) and also using only textual features. We also treat the classification task as:

- a binary classification task (including only copy and non-copy cases).
- a three-way classification (includes partial-copy cases along with copy and non-copy cases).

3.4.5.1 Feature Comparison

We compare the proposed deep features with various other hand-crafted features like source-code metrics [53] and also few textual features [12]. The textual features are as follows: Difference in Length of Submissions (DLS) - the difference between the lengths of each student submission, Similarity as Measured by Diff (SMD) - the number of lines of common code in original submissions, Similarity of Comments (SOC) - the number of comments in common and Similarity in String Literals (SSL) - the similarity between the two sets of literals. To these features, we also add Edit Distance (ED) between

Features	Precision	Recall	F1-score
Textual features [12]	0.590	0.860	0.640
Source-code metrics [53]	0.670	0.800	0.715
Textual features + Source-code metrics	0.745	0.830	0.785
Deep features (*this work*)	0.840	**0.940**	0.880
Deep features + Textual features + Source-code metrics (*this work*)	**0.855**	**0.940**	**0.890**

Table 3.2: Results of binary classification on our dataset. Observe the significant boost in f1-score by using our proposed deep features.

two programs as an extra feature. These features are extracted at syntactic level of a source-code and are inherently pairwise features. The source-code metrics are typically a summary of the internal program representation. They consist of various elements that can capture basic block characteristics, method , control flow characteristics, frequency of variables and constants. Basic block features describe a program based on the number of basic blocks, basic blocks with successors, predecessors etc. Method features capture information related to calls in the methods. However, both textual features and source-code metrics fail to capture some of the essential aspects of a program like non-contiguous interactions at syntax level, needed to detect plagiarized program pairs.

3.4.5.2 Binary Classification

In the first task, we consider plagiarism detection as a binary classification problem. So we removed the instances belonging to the class of partial-copy cases from the dataset and trained an SVM classifier. Then, we opted for 4-fold cross-validation, training on 3/4 of the dataset and testing on the remaining 1/4. However, even after regularizing we found that the classifier was choosing to predict the class with the highest frequency. We use a class weighing scheme to alleviate the effect of class imbalance. The performance of classifier on the test-set using different features is shown in table 3.2. Using our proposed deep features, we observe an improvement of 9.5% in f1-score when compared to source-code metrics + textual features and an improvement of 24% in f1-score when compared to textual features alone.

Table 3.4 shows per-label performance using deep features along with textual features and source-code metrics. Given below are the details of f1-score on per-label basis namely for copy and non-copy cases. The f1-score values obtained using textual features alone are 0.960 and 0.310 for non-copy and copy cases respectively. Using source-code metrics alone, the f1-score values are 0.980 and 0.450 for non-copy and copy cases respectively. By combining the textual features and source-code metrics, the f1-score values slightly increased and are 0.990 and 0.580 for non-copy and copy cases respectively.

Features	Precision	Recall	F1-score
Textual features [12]	0.413	0.490	0.423
Source-code metrics [53]	0.430	0.570	0.460
Textual features + Source-code metrics	0.440	0.573	0.463
Deep features (*this work*)	0.470	0.653	0.513
Deep features + Textual features + Source-code metrics (*this work*)	**0.490**	**0.660**	**0.543**

Table 3.3: Results of three-class classification on our dataset. Observe the significant boost in f1-score by using our proposed deep features.

Class label	Precision	Recall	F1-score
Not copy	1.000	0.990	0.990
Copy	0.710	0.890	0.790

Table 3.4: Per-label performance in binary classification setting using Deep features + Textual features + Source-code metrics.

Class label	Precision	Recall	F1-score
Not copy	0.970	0.890	0.930
Partial copy	0.110	0.300	0.170
Copy	0.390	0.790	0.530

Table 3.5: Per-label performance in three-way classification setting using Deep features + Textual features + Source-code metrics.

However, using deep features alone, we observed a significant boost in the f1-score values for copy cases. The f1-scores are noted to be 0.990 and 0.770 respectively.

3.4.5.3 Three-way Classification

As mentioned earlier, we also wanted to include the instances from the class of partial copy cases to our dataset. We treat the problem as a three-way classification problem and proceed by training a SVM classifier. The performance of classifier on the test-set using different features is shown in table 3.3. Using our proposed deep features, we observe an improvement of 5% in f1-score when compared to

source-code metrics + textual features and an improvement of 9% in f1-score when compared to textual features alone.

Table 3.5 shows per-label performance using deep features along with textual features and source-code metrics. Given below are the details of f1-score on per-label basis namely for copy, partial-copy and non-copy cases. The f1-score values obtained using textual features alone are 0.890, 0.080 and 0.300 for non-copy, partial-copy and copy cases respectively. Using source-code metrics alone, the f1-score values are 0.910, 0.110 and 0.390 for non-copy, partial-copy and copy cases respectively. However, using deep features alone, we observed a significant boost in the f1-score values for copy cases. The f1-scores are noted to be 0.920, 0.140 and 0.480 respectively.

3.5 Summary

In this work, we have explored the possibility of learning generic representations for source-code. Although the proposed deep features are learned using the task of sequence-prediction, they can be directly applied on different dataset and to a different task like code plagiarism detection without the need to fine-tune on each individual problem-set. Our experiments suggest that these features are very promising.

Chapter 4

Retrieval Based Approach for Source-code Plagiarism Detection

Retrieval of potential plagiarized cases from a database containing student submitted assignment programs is a challenging problem. There are three primary dimensions to this problem: (i) How to represent the programs? (ii) How to match/compare two program representations? and (iii) How to retrieve copy cases efficiently and accurately when the size of the database grows? All these problems are relatively easy when the representation is text.

In previous chapter, we have demonstrated how to learn features (deep features) for the structured data in the form of programs. However, these deep features are global representations of a program, where a single global feature can represent/describe a whole program. Unlike this, we can also represent a program by multiple local features that are capable of representing/describing a small local block (few lines) within a program. This idea is analogous to local image descriptors like SIFT [27] that describe a patch within an image. Such representations are useful for identification of *near-duplicate* program pairs, where only a part of the program is copied or certain lines, blocks of code may be copied etc. In such cases, obtaining local feature representations for a program is more useful than representing a program with a single global feature.

In this chapter, we aim at representing programs using local deep features and develop a framework to retrieve susceptible plagiarized cases for a given query program. This work proposes a simple instance retrieval pipeline based on encoding the features of char-RNN using the bag of words aggregation scheme (BoW). We demonstrate the suitability of the BoW representation based on local deep features for retrieval of susceptible plagiarized programs. Figure 4.1 demonstrates our proposed bag of local deep features pipeline.

To retrieve plagiarized programs, we use representation that is similar to Bag of Visual Words (BoVW) representation used for retrieval of images. This is motivated by multiple factors (i) Bag of Words (BoW) representation has been the most popular representation for document (text) retrieval. There are scalable (and even distributed software) solutions available. (ii) BoW method has shown to perform excellently for recognition and retrieval tasks in images and videos [25, 33]. In BoW, a program (source-code) is represented by an unordered set of non-distinctive discrete words. Bag of words encod-

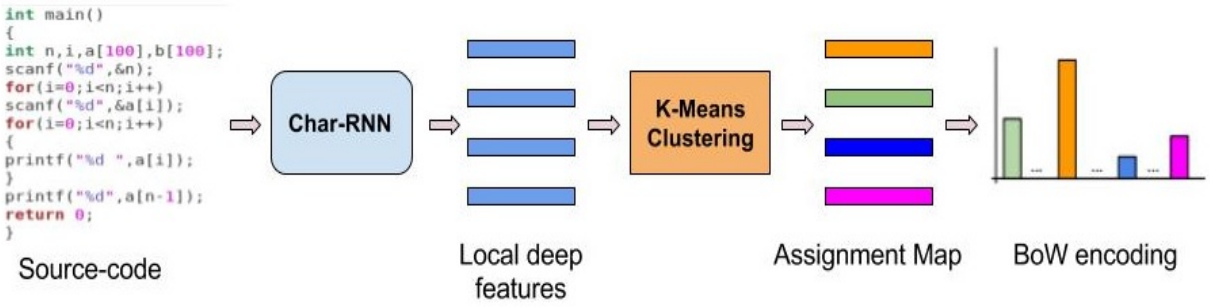


Figure 4.1: The Bag of Local Deep Features pipeline. Assigning each local deep features to a code-word produces an assignment map, a compact representation that relates regions/blocks of a source-code with a code-word.

ings produce sparse high-dimensional codes that can be stored in inverted indices, which are beneficial for fast retrieval. Moreover, BoW-based representations are more compact, faster to compute and easier to interpret. In retrieval phase, a program is retrieved by computing the histogram of word frequencies, and returning the program, with the closest (measured by the cosine of the angles) histogram. This can also be used to rank the returned programs. A benefit of this approach is that, matches can be effectively computed. Therefore, programs can be retrieved with no delay.

4.1 Bag of Words Approach

In text domain, document retrieval is very extensively studied. One of the popular method used for document retrieval is based on Bag of Words (BoW) approach. It provides a mechanism to represent documents wherein the frequency of occurrence of code-word is used as a feature representation of the document. However, documents are represented as an unordered collection of words. BoW is generally divided into followings stages: Preprocessing, Stemming and Vocabulary/code-word selection. Using the selected vocabulary, a document is indexed using inverted file index and retrieval is performed using term frequency inverse document frequency (*tf - idf*).

In preprocessing stage, a text document is tokenized into stream of words. All the punctuations are striped off and tabs and other non-text characters are replaced by white spaces. These tokenized words are then used for further processing. The set of different words obtained by merging words from all the text documents is termed as a dictionary or vocabulary. After, obtaining initial vocabulary, the size of vocabulary can be reduced by filtering and stemming. Initial filtering is done by removing stop words like articles, conjunctions, prepositions, etc. Removing stop words bear little or no content information on document representation. Furthermore, words that occur extremely often can be said to be of little information content to distinguish between documents and words that occur very rare are likely to be of no statistical relevance and can be removed from the vocabulary. In stemming, we try to build basic word forms. This is done by replacing plurals, verb forms of words by their root words or stem.

To further decrease the number of words that should be used keyword selection algorithms can be used. In this case, only the selected keywords are used to describe the documents. A simple method for keyword selection is to extract keywords based on their entropy. Entropy gives a measure how well a word is suited to separate documents by keyword search. Once the vocabulary has been fixed, each document is represented as a vector with integer entries of length V. If this vector is x then its jth component x_{j} is the number of appearances of word j in the document. The length of the document is n = $\sum_{j=1}^{V} x_{\mathrm{j}}$. For documents, n is much smaller than V and xj = 0 for most words j. This gives sparse representation of a given document.

Next step is to index these documents using created vocabulary. Indexing is mostly done using inverted file index, which is most popular and efficient way to index these vectors. In inverted file index consists entry for all words and each word is linked with all the documents where the word is present and also its frequency is stored. While is retrieval phase, for a given query corresponding word in probed in index and all documents which are having given query words is retrieved. These retrieved documents are ranked using *tf - idf*, which is defined as:

$$W_{ij} = f_{ij} \times (log\ N - log\ d_j) \tag{4.1}$$

where W_{ij} is weight of the term j in document i, f_{ij} is the frequency of the term j in document, *N* id the number of documents in the collection, and d_j is the number of documents containing the term j. Thus, the weight W_{ij} quantifies the extent of relevance of the document of the given word. Relevant documents usually have high *tf - idf* weights relative to other documents.

4.2 Bag of Visual Words

The BoVW model is inspired by the success of using BoW in text classification and retrieval. Document is formally represented with the help of frequency of occurrences of the words in the vocabulary. These histograms are then used to perform document classification and retrieval. Analogously, an image is represented by an unordered set of non-distinctive discrete visual features. The set of these discrete visual features is called vocabulary. By representing an image as a histogram of visual words, one can obtain certain level of in-variance to the spatial location of objects in the image.

The typical bag-of-visual words approach consist following steps:

- Extraction of local image features
- Visual vocabulary generation
- Histogram generation

Extraction of local image features

The first step is to compute the local features, which captures the local characteristics of an image. Scale Invariant Feature Transform (SIFT) [27] is one of the popular local feature used in computer vision. At first, interest points like corners and blobs in an image are detected. At each of these interest points, we extract a SIFT descriptor to describe the local information as a vector of gradients.

Visual vocabulary generation

Once the local descriptors are computed on the given images, the next step is to create a code-book from them. Since the space of SIFT feature descriptors is continuous, we create code-book/vocabulary by discretizing the space by clustering SIFT vectors (often with k-means) obtained from a small collection of images. The size of vocabulary is a parameter that depends on the dataset, and is generally determined experimentally.

Histogram generation

The created visual vocabulary is then used to quantize the extracted features by simply assigning the label of the closest cluster centroid. This is carried out by rolling down the sample from the root to the leaf of the vocabulary tree. The final representation for an image is the frequency counts or histogram of the quantized SIFT features $[f_1, f_2, ...f_i, ..., f_k]$, where f_i is the number of occurrences of i^{th} visual word in the image and k is the vocabulary size. To account for the difference in the number of interest points between images (due to size etc.), the BoVW histogram is normalized to have unit L1 norm.

Once, the features representations of an image are obtained by following the above mentioned three steps, the pipeline generally ends by learning suitable models (classifiers) like SVM for a classification problem.

4.3 Retrieval System Overview

In this section we describe our retrieval system. Figure 4.2 shows the overview of the system. The system is divided into two parts i.e., indexing and retrieval. This is in addition to the one time computation of the vocabulary. Indexing comprises of three steps as follows: (i) Features are extracted from submitted programs, (ii) Histograms are created by vector quantization, and (iii) Database is created by indexing programs using an inverted file index. In retrieval process, first two steps are similar to the indexing. Then histogram is finally given to the index structure and images are retrieved in a ranked manner.

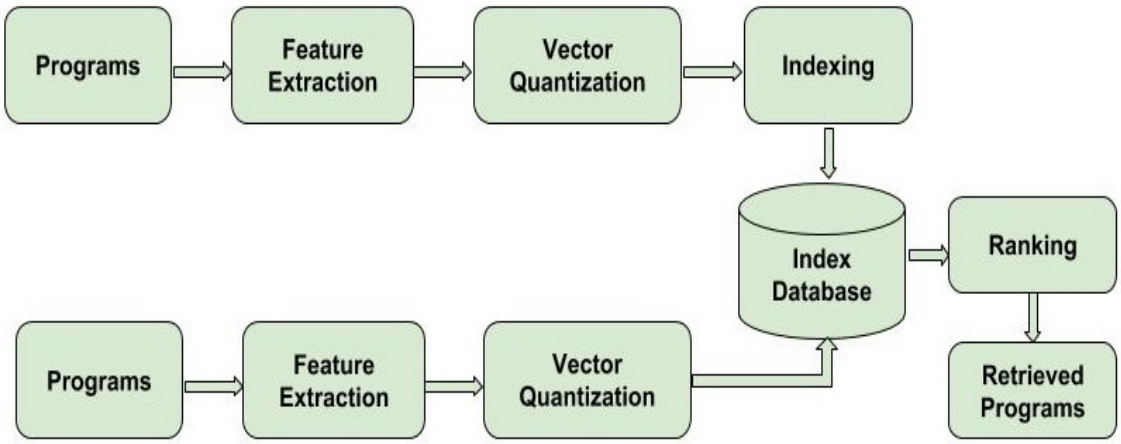


Figure 4.2: Overview of the retrieval system.

4.3.1 Local Feature Extraction - Sliding Window Approach

To extract local features from programs (source-code), we follow a sliding window approach. We treat the program as sequence of characters. We have a chosen to use a window of size 100 characters with a stride of 1. Stride describes how much a window is shifted at each step. Local features within the window are extracted using the pre-trained char-RNN with LSTM units.

Suppose a programs contains a total of N characters including white spaces and let the size of window be W. Then, the total number of local features that can be extracted is given by (N - W + 1). Figure 4.3 demonstrates the extraction of local features from sliding windows. Since, all these local features are continuous real-valued vectors, histogram creation and vector quantization is necessary to create vocabulary / code-book. We use a vocabulary of 1000 and clustering solution based on K-means.

This generated code-words/vocabulary is then used to quantize the extracted features by simply assigning the label of the closest cluster centroid. This is carried out by rolling down the sample from the root to the leaf of the vocabulary tree. The final representation for a program is the frequency counts or histogram of the quantized local deep features $[f_1, f_2, ...f_i, ..., f_k]$, where f_i is the number of occurrences of i^{th} visual word in the image and k is the vocabulary size. To account for the difference in the number of local features between programs (due to size etc.), the histogram is normalized to have unit L1 norm.

4.3.2 Retrieval

In retrieval phase, local features are extracted on new given programs using char-RNN model. A program is retrieved by computing the histogram of code-word frequencies, and returning the program, with the closest (measured by the cosine of the angles) histogram. This can also be used to rank the returned programs.

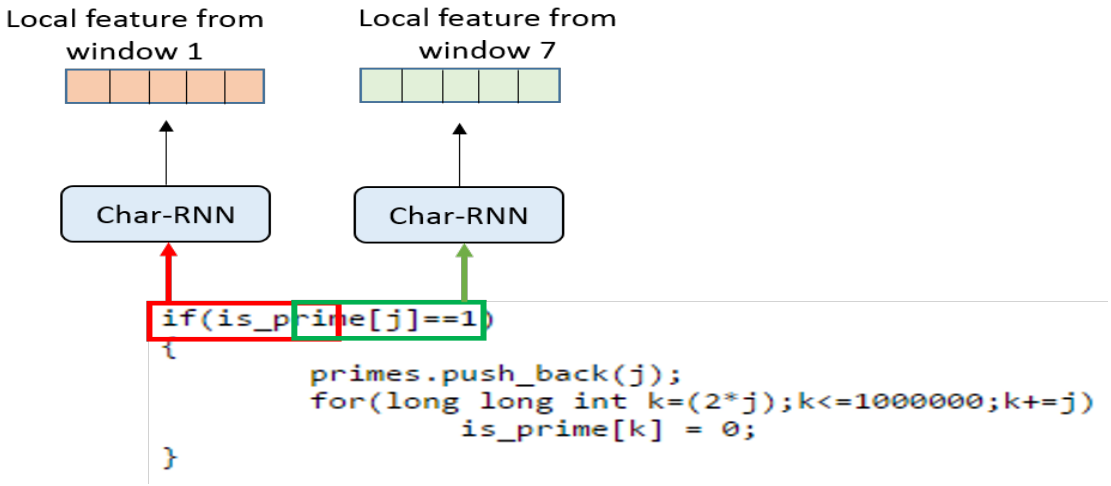


Figure 4.3: Extraction of local features from source-code using sliding window approach. A stride of size one is used originally. For sake of clarity, we used a stride of six in the figure. Finally the entire source-code is represented as a bag of these local features.

Dataset	Candidate set size	Query set size	Prec @ 5	Prec @ 10
Ours	4,700	80	0.8638	0.8638

Table 4.1: Precision @5 and Precision @10 statistics on our dataset.

The ranking is computed using the cosine similarity between the BoW vector of the query program and the BoW vectors of all the programs in the database. The image list is then sorted based on the cosine similarity of its elements to the query.

4.4 Experiments and Results

4.4.1 Dataset Details

The dataset we appropriately adopted is a collection of assignment solutions submitted by students from an introductory C programming course. We call this dataset as D1. Our dataset is formed out of 22 problem-sets. In each problem-set there are 70 to 250 student submissions. Program solutions falling in the same problem-set are functionally consistent. In total, there are about 4,700 submissions. The questions asked in the problem-sets range from more specific (as in case of tree-traversal) to diverse. Each problem-set is of varying difficulty, with student solutions ranging from 50 to 400 lines of code. The programs in our dataset are obfuscated in the following ways: variable name changing, careful conversion of while loop into for loop, dead code injection etc. Creation of program pairs and labeling

them is carried out in same fashion as mentioned in in section 3.4.1. The candidate set and query set are created from the pairs that are labeled as copy cases. Overall, the candidate set contains 4K programs. Table 3.1 provides the information about the datasets used.

4.4.2 Results

In this section, we present results to demonstrate the utility and scalability of the proposed approach. The proposed method of plagiarized code retrieval is tested on a collection of student submitted programs on 22 different types of problem sets of varying difficulty. The results are summarized in table 4.1. We verify our method on around 4K annotated program pairs from 22 problem sets. In order to demonstrate the utility of the method further, we conduct experiment on these program pairs and achieve a Precision@5 and Precision@10 of 0.8638 across the entire collection. For retrieved entities (programs), we can provide the cosine similarity score as a confidence measure.

Listing 4.2 and 4.1 shows an example of partial copy cases where only some lines/block of code is copied. Notice that in both the codes, most of the lines in first ten lines are same. Here listing 4.1 is the query program and listing 4.2 is the retrieved program which is ranked second in the ranking list, with a similarity score of 0.746.

```
1  int power(int a,int x);
2  int main()
3  {
4  int t,n,temp,r;
5
6  scanf("%d",&t);
7  while(t--)
8  {
9  scanf("%d",&n);
10 if(n==1)
11 printf("0\n");
12 else{
13 temp= log2(n);
14 r= n-power(2,temp);
15 if(r>(power(2,temp-1)-1))
16 {
17 temp= 2*temp;
18 printf("%d\n",temp);
19 }
20 else
21 {
```

```
temp= (2*temp)-1;
printf("%d\n", temp);
}}

}
return 0;
}
int power(int a,int x)
{int i;
int ans=a;
for(i=1;i<x;i++)
ans=ans*a;
if(x==0)
return 1;
else
return ans;
}
```

Listing 4.1: Query Program

```
int main(){
int t,n,temp,r;
scanf("%d",&t);
while(t--){
scanf("%d",&n);
temp= log2(n);
r= n-pow(2,temp);
if(r >(pow(2,temp-1)-1)){
temp= 2*temp;
printf("%d\n",temp);
}
else{
temp= (2*temp)-1;
printf("%d\n", temp);
}

}
return 0;
```

```
}
```

Listing 4.2: Retrieved program

4.4.3 Discussion

Our proposed system takes 0.3 seconds to retrieve and rank the near-duplicate or plagiarized programs. This shows the effectiveness of our system. As the size of candidate dataset increases, for scalability purpose, we can use tf-idf based indexing scheme for effective search and retrieval of plagiarized programs from a collection of student submitted programs. In retrieval phase, local features are extracted on new given programs using char-RNN model. A program is retrieved by computing the histogram of code-word frequencies, and returning the program, with the closest tf-idf score. Hence, for retrieval we only need to refer to the inverted list.

The notion of partial-copy or near-duplicate is very subtle. There can be two individual programs in which has few lines (say 10 out of 100 lines) in common. This does not essentially make them partial-copy cases. Also, some of syntactical elements like keywords, loop structures etc. in code are common in two given programs that belong to the same problem set. Especially, it becomes even more difficult to evaluate programs if they are small in length (only very few lines of code). Consider the programs shown in the listing 4.3 and 4.4. The similarity score for this pair of programs is around 0.60. However, if we closely observe the similarity lies in the lines 10 to 20 in both these codes which are mostly loop structures. It is also important to note that the problem these programs are solving is too specific in nature. In such cases, the domain knowledge (type of problem-set etc.) along with similarity score could assist one to decide whether a pair of programs is really near-duplicates or not.

```
int main()
{
int n,i,j,k;
scanf("%d",&n);
while(n!=0)
{
int a[n];
int c=0;
for(i=0;i<n;i++)
scanf("%d",&a[i]);
for(i=0;i<n;i++)
{
for(j=i+1;j<n;j++)
{
for(k=i+1;k<j;k++)
```

```
{
if(a[k]>a[i] || a[k]>a[j])
{
c++;
break;
}
}
}
}
printf("%d\n",c);
scanf("%d",&n);
}
return 0;
}
```

Listing 4.3: A query program to demonstrate that the notion of near-duplicate is sometimes subtle

```
int main()
{
int height[500010],n,i,j,k,count;
while(1)
{
count=0;
scanf("%d",&n);
if(n==0)
break;
for(i=0;i<n;i++)
scanf("%d",&height[i]);
for(i=0;i<n;i++)
{
for(j=i+1;j<n;j++)
{
for(k=i+1;k<j;k++)
{
if(height[k]>height[i]||height[k]>height[j])
{
count++;
break;
}
```

```
}
}
}
printf("%d\n",count);
}
return 0;
}
```

Listing 4.4: A retrieved program corresponding to the query program shown in listing 4.3

4.5 Summary

We have presented a retrieval system based on BoW scheme. Our method is highly problem-set independent and scalable. The efficiency of proposed method is shown experimentally on twenty-two different problem-sets. Our future work includes (i) Learning problem-set-specific local descriptors (ii) Use a re-ranking strategy to rank the retrieved results.

Chapter 5

Conclusions and Future Work

5.1 Summary and Conclusion

In this thesis we have explored how to apply machine learning techniques to the structured data in the form of programs (source-code). Specifically, we have proposed machine learning and deep learning approaches for automatic source-code plagiarism detection. We have proposed novel features (source-code metrics) and demonstrated their utility. Later, we exploited the naturalness in source-code and demonstrated how to learn embeddings for programs. We trained a char-RNN model for sequence prediction task on source-code and learned program embeddings. We use these embeddings as deep features and proved that they are useful in source-code plagiarism detection. Later, we introduced local deep features and presented a retrieval system based on BoW model. The preliminary retrieval results show that our system is better at identifying plagiarized cases when compared to other popular tools like MOSS.

5.2 Future directions

- **Dynamic features:** Identify and use additional dynamic features along with static source-code metrics, that could boost the chances of plagiarism detection.
- **Attention Based Models:** At a finer level, one could attempt to learn deep features using char-RNN with attention mechanism. These models are capable of highlighting / attending regions in source-code which is responsible for current character output.
- It might be possible to build retrieval systems which could actually retrieve source-code from large code repositories which are similar at higher level semantics. Use of tree-structured LSTM [47], recursive recurrent neural networks can facilitate us to learn some representation to develop such retrieval systems.

Related Publications

1. **Jitendra Yasaswi**, Sri Kailash, Anil Ch, Suresh Purini ans C. V. Jawahar, " *Unsupervised Learning Based Approach for Plagiarism Detection in Programming Assignments*", ACM ISEC 2017.

2. **Jitendra Yasaswi**, Suresh Purini ans C. V. Jawahar, "*Plagiarism Detection in Programming Assignments Using Deep Features*", ACPR 2017

Bibliography

[1] Christopher Olah - Understanding LSTM Networks http://colah.github.io/posts/2015-08-Understanding-LSTMs/. xi, 10, 11

[2] http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-intro 9

[3] http://ctuning.org/wiki/index.php/CTools:MilepostGCC. 16

[4] http://ctuning.org/wiki/index.php/CTools:MilepostGCC:StaticFeatures:MILEPOST_V2.1. 17

[5] Y. Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2009. 8, 30

[6] D. Bouneffouf. Exponentiated gradient exploration for active learning. *Computers*, 5(1):1, 2016. 7

[7] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8-13):1157–1166, 1997. 5

[8] M. Brooks, S. Basu, C. Jacobs, and L. Vanderwende. Divide and correct: using clusters to grade short answers at scale. In *Proceedings of the first ACM conference on Learning@ scale conference*, pages 89–98. ACM, 2014. 4

[9] M. Chilowicz, E. Duris, and G. Roussel. Syntax tree fingerprinting: a foundation for source code similarity detection. In *ICPC 2009*. 31

[10] C. Cummins, H. Leather, P. Petoumenos, and R. Mayr. Deep learning for compilers. *ICSA*, 2017. 30

[11] S. Engels, V. Lakshmanan, and M. Craig. Plagiarism detection using feature-based neural networks. In *ACM SIGCSE Bulletin*, volume 39, pages 34–38. ACM, 2007. 4, 15, 19, 23

[12] S. Engels, V. Lakshmanan, and M. Craig. Plagiarism detection using feature-based neural networks. In *SIGCSE*, 2007. 29, 32, 33, 36, 37, 38

[13] E. Flores, A. Barrón-Cedeño, P. Rosso, and L. Moreno. Towards the detection of cross-language source code reuse. *Natural Language Processing and Information Systems*, 2011. 32

[14] E. L. Glassman, R. Singh, and R. C. Miller. Feature engineering for clustering student solutions. In *Proceedings of the first ACM conference on Learning@ scale conference*, pages 171–172. ACM, 2014. 4

[15] A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013. 30

[16] R. Gupta, S. Pal, A. Kanade, and S. Shevade. Deepfix: Fixing common c language errors by deep learning. In *AAAI*, 2017. 31

[17] J. Hage, P. Rademaker, and N. van Vugt. A comparison of plagiarism detection tools. *Utrecht University. Utrecht, The Netherlands*, 28, 2010. 1

[18] H. Hajishirzi, W.-t. Yih, and A. Kolcz. Adaptive near-duplicate detection via similarity learning. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, pages 419–426, New York, NY, USA, 2010. ACM. 5

[19] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *ICSE, 2012*. 29, 31

[20] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 1997. 10, 30

[21] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998. 5

[22] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016. 32

[23] A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015. 30, 33, 35

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012. 30

[25] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, volume 2, pages 2169–2178. IEEE, 2006. 40

[26] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881. ACM, 2006. 3

[27] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004. 40, 43

[28] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008. 25

[29] V. T. Martins, D. Fonte, P. R. Henriques, and D. da Cruz. Plagiarism detection: A tool survey and comparison. In *OASIcs-OpenAccess Series in Informatics*, volume 38. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014. 1

[30] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010. 31

[31] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu. Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection. *IEEE Transactions on Reliability*, 65(4):1647–1664, 2016. 1, 3

[32] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu. Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection. *IEEE Transactions on Reliability*, 2016. 29

[33] E. Mohedano, K. McGuinness, N. E. O'Connor, A. Salvador, F. Marqués, and X. Giró-i Nieto. Bags of local convolutional features for scalable instance search. In *Proceedings of the 2016 ACM on International Conference on Multimedia Retrieval*, pages 327–331. ACM, 2016. 40

[34] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, 2016. 33, 34

[35] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016. 30

[36] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *ICML*, 2013. 30

[37] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 153–160. ACM, 2012. 4

[38] L. Prechelt et al. Finding plagiarisms among a set of programs with jplag. *J. UCS*, 2002. 29

[39] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *J. UCS*, 8(11):1016, 2002. 1, 3

[40] A. Ramírez-de-la Cruz, G. Ramírez-de-la Rosa, C. Sánchez-Sánchez, H. Jiménez-Salazar, C. Rodríguez-Lucatero, and W. A. Luna-Ramírez. High level features for detecting source code plagiarism across programming languages. In *FIRE Workshops*, pages 10–14, 2015. 4

[41] S. Rogers, D. Garcia, J. F. Canny, S. Tang, and D. Kang. *ACES: Automatic evaluation of coding style*. PhD thesis, Master's thesis, EECS Department, University of California, Berkeley, 2014. 4

[42] M. Schein and R. Paladugu. Redundant surgical publications: tip of the iceberg? *Surgery*, 129(6):655–661, 2001. 1

[43] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003. 1, 3

[44] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *ACM SIGMOD*, 2003. 29

[45] I. Sutskever, J. Martens, and G. E. Hinton. Generating text with recurrent neural networks. In *ICML*, 2011. 30

[46] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 2014. 10, 30

[47] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015. 51

[48] D. Tang, B. Qin, and T. Liu. Document modeling with gated recurrent neural network for sentiment classification. In *EMNLP*, 2015. 34

[49] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 563–570. ACM, 2008. 5

[50] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012. 35

[51] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015. 10, 30

[52] S. Y et al. Deep lstm based feature mapping for query classification. In *HLT-NAACL 2016*. 31

[53] J. Yasaswi, S. Kailash, A. Chilupuri, S. Purini, and C. V. Jawahar. Unsupervised learning based approach for plagiarism detection in programming assignments. ISEC '17. ACM, 2017. 35, 36, 37, 38