Proceedings of the International MultiConference of Engineers and Computer Scientists 2011 Vol I,
IMECS 2011, March 16 - 18, 2011, Hong Kong

Plagiarism Detection Based on SCAM Algorithm

Daniele Anzelmi, Domenico Carlone, Fabio Rizzello,
Robert Thomsen, D. M. Akbar Hussain *

Abstract—**Plagiarism is a complex problem and considered one of the biggest in publishing of scientific, engineering and other types of documents. Plagiarism has also increased with the widespread use of the Internet as large amount of digital data is available. Plagiarism is not just direct copy but also paraphrasing, rewording, adapting parts, missing references or wrong citations. This makes the problem more difficult to handle adequately. Plagiarism detection techniques are applied by making a distinction between natural and programming languages. Our proposed detection process is based on natural language by comparing documents. A similarity score is determined for each pair of documents which match significantly. We have implemented SCAM (Standard Copy Analysis Mechanism) which is a relative measure to detect overlap by making comparison on a set of words that are common between test document and registered document. Our plagiarism detection system, like many Information Retrieval systems, is evaluated with metrics of precision and recall.**

Keywords: Plagiarism, SCAM, WordNet, Apache Lucene

1 Introduction

A document may be seen as a list of terms, list of keywords, and set of related words or concepts. It seems obvious that it is not the same to have just simple words as the basic items of the representation, that to have a set of words (may be structured someway) with a representation of their meaning [1]. The analysis on documents' contents can be semantic or statistical. Regarding the document representation, a commonly used Information Retrieval approach is the adoption of the Vector Space Model, where the similarity score between two documents is calculated using the cosine formula, resulting the cosine of the angle between the two corresponding vectors. This representation is also called a "bag-of-words", since the list of word positions are not maintained, hence relationships between words are missed [2]. This seems not to be appropriate for a plagiarism detection system that works on text documents, also cosine formula has issues when documents differ in size. In natural language, a sentence may be seen as the fundamental part of a discourse and the minimum unit to express a concept. We considered a good norm to build a sentence-level system to compare documents using semantic analysis. Sentence boundaries allow us to keep track of meaning and contest of terms, maintaining information about their position and mutual relationships. A phrase is extracted from each document every time a particular punctuation mark is met; the vocabulary of terms is expanded with synonyms through Wordnet, trying to cover paraphrasing. To search for source containing suspicious phrases a search engine is required. Usually plagiarism detection involves Internet sources and web search engines which is free, easy and fast way of detecting plagiarism. The user can copy and paste or type in suspicious phrases taken from suspected plagiarized work into a search engine in an attempt to find on-line material containing the suspicious phrases" [3]. Unfortunately they are not open source and working with them means that there is no possibility to tweak the code according to your requirements and consequently, user lacks complete control about elaboration and results. In order to avoid such limitations we decided to use our own search engine based on Apache Lucene java library. This is a major benefit of using an open source search engine, since one can tweak the calculation of the score for a document to the required specifications. The scoring and similarity calculations are transparent and one can build similarity classes that are appropriate for required domain [2]. The dataset used in our system is restricted to some local documents in text format. For the search algorithm we combined Lucene functionalities with our own comparison procedure. For each sentence in each document, several searches are launched, trying to cover all the possible forms of a plagiarized phrase. The similarity matches are obtained with (using) SCAM algorithm and they are displayed on a GUI. The idea is to present a list of plagiarized documents ordered by similarity score. Thus the user has the possibility to visualize in detail the compared parts. To evaluate the effectiveness of the detection system, "precision and recall" are implemented.

*Manuscript submitted January, 2011 Dr. M. Akbar Hussain is member of IEEE, IDA, IAENG and works at the Department of Electronic Systems Aalborg University, Niels Bohrs Vej 8, 6700 Esbjerg, Denmark. Email: akh@es.aau.dk

2 Implementation

In this section we present an implementation of our plagiarism detection system based on Lucene search engine. The main tools and utilities used to develop our application are Apache Lucene and WordNet. The abstract system architecture is shown in figure 1, however, details of the database structure and the main classes of code carrying out the execution are also described in this section. The programming language chosen is java, the motivation is given by the fact that the native version of Lucene is written in this language and this can easily help us to manage and possibly modify it.

2.1 Database

The creation of the database has a simple scheme to keep all the information ordered in a uniform way. As mentioned above, keeping some data structures in database tables saves memory from the RAM and makes the execution faster. The database architecture is pretty simple and we use only three tables. A representation of the

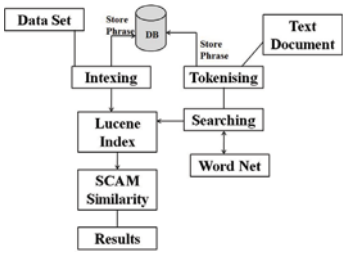

Figure 1: System Architecture

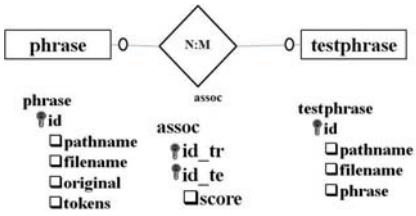

Figure 2: E-R Scheme

E-R scheme and the definition of tables and attributes can be seen in Figure 2. The phrase table stores the phrases obtained from dataset documents. The attribute *id* is a unique identifier or primary key; it is an auto generated incremental number, which acts as surrogate key. The attribute original is the column where we store the phrase extrapolated from the test document. The others columns store information about the path of the file, the name of the file and the list of tokens contained in the phrase. The table *testphrase* is the table where we store the phrases obtained from the test document. The table *assoc* is the table connecting phrases from phrase and testphrase with the relative matching score. We have used Mysql as it is the most common open source relational database management system available on the net [4]. It provides support for almost all the SQL syntax and it is compatible with many major programming languages including Java.

2.2 Detection Steps

In this section we report a detailed overview on the main methods implemented and the steps followed to achieve a detection system. There are four main steps:

- Indexing the dataset: The dataset is composed of several documents which are preprocessed and indexed.
- Processing the test document: The test document given in input is processed in order to tokenize it in a list of words; stemming and stop-word removal are applied.
- Searching on the index: The index is questioned in order to retrieve a match between the test document and the documents belonging to the dataset.
- Evaluating similarity: Using SCAM formula, a similarity measure is calculated between test document and the documents belonging to the dataset.

The code is divided in five Java packages:

- Core: Handles the project execution; all the Java classes implementing the above four main steps are included in this package, which are: IndexTrain.java, ProcessTest.java, SearchDocs.java, ScamEvaluation.java.
- Utils: Includes some Java Classes for example, Config.java: handles information about the configuration and relative paths contained in the file Config.dat, LuceneTools.java: contains methods to support extraction of tokens from raw text, WordnetUtils.java: makes the application using WordNet dictionary.
- Db: Includes all objects related to the database and its operations, DatabaseConnection.java: permits the connection to database, QueryExecution.java: contains methods to query the database, ScriptExecution.java: starts the execution of database scripts, creates database, creates tables and SQL views.
- HtmlResults: Contains the Java class in charge of showing detailed results in html format.
- Gui: Contains Graphical User Interface.

2.3 Indexing Dataset

The first step is to create an index of the dataset documents; with this index it will be possible to retrieve useful information that is going to be used during the evaluation step. The Java class handling this process is IndexTrain from the package core. Lucene provides a class called IndexWriter, the main class in charge of creating the index.

new IndexWriter (Directory dir, Analyzer Analyzer, boolean option).

The constructor (IndexWriter) takes in three parameters, the first parameter is an object of class Directory from Lucene that has been set with the path of the directory in which we want to write the index. The second parameter is an object of the abstract class Analyzer, this is an important passage in order to obtain satisfactory results because the terms generated by the analyzer are

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

the only way a document can be questioned and retrieved. There are several implementations of Analyzer, most of them directly provided by Lucene project; furthermore it would take not too much effort if it is demanded to implement your own analyzer; in our project we use an instance of StandardAnalyzer: it provides the common pre-processing steps as stop-word removal and stemming. We also considered others analyzers for example nGramAnalyzer but discarded because n-grams technique divides the text in character sequences and it is not possible to treat tokens generated as single words which means we lose the meaning of terms; we need to keep track of the meanings because we have to work on them and locate the synonyms to discover paraphrasing plagiarism. The third parameter of IndexWriter is a Boolean value, needed to indicate the indexer how to behave with the index directory; with a value set to true the index will be rewritten each time the constructor will be launched; with a value set to false, the indexer will open the already existing index and will modify it; we want to write an index so we pass a value of true. The IndexWriter has a method called addDocument that permits to add a document to the index. The class Document is passed to that method; each document has several fields, thus we can add as much fields as we want, depending on how much information we want to store about the document. Unfortunately Lucene does not permit to retrieve the piece of text within the document that best matches the query, it only retrieves the whole document containing the best match; it does this without giving out more specific information because the query returns a set of document objects. What we did is that we have splited each dataset document in phrases and consider each phrase as a document object (Figure 3). One of the methods helping us to split text has been the use of regular expressions. The easiest way to divide text is to locate a phrase each time a particular punctuation mark is met. This design choice

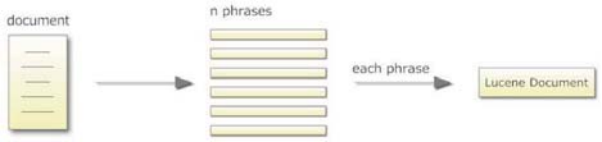

Figure 3: Document Split

increments the number of index choices; however, it permits us to be more precise later in retrieving the match among exact phrases that have been plagiarized. The information each document contains inside are basically three:

- id: It is a progressive numerical identifier; id is not indexed (Field.Index.NO) but it is stored (Field.Store.YES) since the aim of the id is to find the corresponding chunk term in frequency vector and the corresponding row in database.
- filename: The name of the document's chunk, also this field is not indexed but it is stored because we wanted to retrieve the name to show in the results; which means filename is the name of file plus the number of inner chunk.
- content: The content of document's chunk, indexed in form of tokens. We wanted the indexer to create a term frequency vector with option Field.TermVector.YES; the vector will be used in the last step of our application to calculate the similarity formula. The Lucene class Document is added to the index by the method addDocument from the class IndexWriter. In this class (IndexTrain) we store each phrase in table *phrase* after splitting process.

2.4 Processing the Test Document

In this step input data is processed and it is called a test document. The class in charge of performing this task is *ProcessTest*. The main method in this class is extractTokens; it has two main objectives: fill a data structure (a Java HashMap) and fill the table *testphrase*. The method uses the same Analyzer which is used to index the dataset; this is very important in order to keep consistency in the way the text is processed and in order to achieve better results. The test document is splited in the same way the train documents have been splited by using regular expressions and the same logic is used to trim a phrase each time a particular punctuation mark is met. The phrase is stored in the database with a unique *id*. Using the static method tokensFromAnalysis from the class LuceneTools in the package utils, we transformed the text in a bag-of-words according to the analyzer passed as parameter. It is important to remember that we use StandardAnalyzer which performs stemming and stop-word removal. The logic behind this has been to consider each phrase generated as a single document, so for each generated phrase we created a term frequency vector, like the code below: `if (tdocFreq.containsKey(term))`
`tdocFreq.put(term, tdocFreq.get(term).intValue() + 1);`
`else`
`tdocFreq.put(term, 1);` This data structure is a simple term frequency vector associating each term to the number of its occurrences in text. For example if we had: *"Every man dies, not every man really lives."*, the derived structure will be as shown in Table 1. It is related to the phrase and to keep track of this relation we associate each phrase through its own id and term frequency vector. In the same way we operated in indexing process: we consider and work only on phrases having more than five tokens. To launch a more sophisticated searching, there is a second method in the class called expandWordnet. This method associates the terms with their own synonyms creating and filling another data structure, a Java HashMap. This method uses the class WordnetUtils in the package utils which makes possible to query the dictionary to obtain a list of synonyms. For example for the same text of above we would have a data structure filled in this way as shown in table 2. This structure will be passed to a method querying the created index.

Table 1: Term Frequency: Test Document

Term	Synonyms
Every	2
Man	2
Die	1
Really	1
Lives	1

2.5 Searching on the Index

In this step which is performed by the class SearchDocs, the index is questioned to retrieve documents containing

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

terms also in the test document. Lucene class permitting to query an index is IndexSearcher:

IndexSearcher is = new IndexSearcher(directory);

where directory is an object of class Directory containing the path of the index previously written. Within SearchDocs two methods having the same name are declared: queryDocs; the first one takes only the structure with test phrase *id* associated to its term frequency vector (tdocFreq, Table 1); the second method takes also the WordNet structure (Table 2). In both cases we scanned this structure and for each test phrase it is operated in this way:

for each term
create a query
question the index
for each result obtained (for each document)
get the term frequency vector
fill two data structures:
- docVectors: document - vector
- docSumOfFreq: document - sum of the term frequencies
end for each
end for each

Creating a query with Lucene has been pretty simple; first we created the TermQuery with the Term we wanted to include; in this case we specify that we wanted to search a term, which is taken from the test document and stored in a variable (in field contents). Indexed the dataset, we create this field and put inside the content of the phrase. A BooleanQuery has been created and passed to the method search of IndexSearcher, like in code:

TermQuery tq = new TermQuery
(new Term("contents", term));
BooleanQuery theQuery = new BooleanQuery();
theQuery.add(tq, BooleanClause.Occur.SHOULD);
Hits hits = is.search(theQuery); Hits is a list of Document resulting from the query; for each Document we took the respective term frequency vector, as we specified to IndexWriter. We did this using methods getIndexReader and getTermFreqVector from class IndexSearcher, specifying *id* and *field* to retrieve.

TermFreqVector tfVector =
is.getIndexReader().getTermFreqVector(Integer.parseInt(id),
"contents");
We associated each retrieved document with respective term frequency vector in a data structure called docVectors.
docVectors.put(filename, tfVector);

Table 2: Term Synonyms WordNet

Term	Occurrence
Man	adult male, homo, human being, human...
Die	decease, perish, pass away, expire...
Really	truly, actually, in truth...
Lives	survive, endure, exist, alive...

Term frequency vector is a structure similar to the one shown in table 3, used for test document, every term in the text is associated with its occurrence number. In the other data structure we associated the document with a number resulting from the following formula:

$$\sum_{j=1} tf_j^2 \quad (1)$$

This structure supports the SCAM formula: for each

Table 3: Document Vectors

Document	Vector
d1.txt	animal=1, jungle=1, africa=3, ...
d2.txt	stadium=1, soccer=2, ball=4, ...
d3.txt	Denmark=2, queen=2, snow=3, ...
.....	.....
.....	.....
.....	.....

term in the vector we calculated the sum of squares of occurrence number. In equation 1, n is the total number of terms in the vector and tf is the number of occurrences in the text. The code implementation is:

for (int j = 0; j ¡ frequencies.length; j++)
sumOfFreq += (frequencies[j] * frequencies[j]);

The overloaded method queryDocs takes the WordNet structure associating term - synonyms. The logic behind this is really simple:

if the query by term does not return results
create a new query with the term's synonyms.

A practical example could be applied to the previous sentence: *"Every man dies, not every man really lives"* Searching for plagiarism in the following sentence using synonyms: *"Every human decease, actually not every human lives."* We would not find a good match between the two sentences even if it is clearly a plagiarized part. In fact, the query results of the following terms: human, decease and actually, would not return anything. But taking in consideration the synonyms, results can be improved; scanning out the synonyms' data structure we would find that man, die and really are synonyms of human, decease and actually. The following code implements the case, if previous query has no result, get the synonyms of that term and for each synonym create a new query. We decided to create WordNet search to provide an optional functionality. In the end we have a data structure: in which for each test phrase *id* is associated with the respective list of documents retrieved.

if(hits.length()==0)
if(expandedTokens.containsKey(term))
Iterator<String> it = expandedTokens.get(term).iterator();
while(it.hasNext())
String new$_t erm = it.next()$;
TermQuery tq = new TermQuery(new
Term("contents",new$_t erm$));
BooleanQuery theQuery = new BooleanQuery();
theQuery.add(tq, BooleanClause.Occur.SHOULD);
hits = is.search(theQuery);

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

2.6 Evaluating Similarity with SCAM

Detecting plagiarism is not a simple string match; it should give a positive result for example by indicating either the registered document is a superset or a subset of the test document. Simple cosine similarity measure typically used is not enough to recognize overlap between documents; SCAM formula is a relative measure to detect overlap, irrespective of the differences in document sizes [2]. We have implemented the SCAM formula to detect similarity among documents. This similarity formula returns a high value when the content of test document is either a subset or a superset of the registered document. It is an alternative to the cosine similarity formula and it works on a set of words that are in common between test and registered document, a word w_i is included in the set C, if the following condition is true:

$$\epsilon - \left(\frac{f_i(R)}{f_i(T)} + \frac{f_i(T)}{f_i(R)} \right) > 0 \qquad (2)$$

ϵ is a constant value greater than 2; a large vale of ϵ expands the above set including words sometimes not relevant, a lower value of ϵ reduces the ability to detect minor overlap, since some words can be excluded. $f_i(R)$ and $f_i(T)$ are the number of times w_i occurs in registered documents (R) and test document (T). The score for the measures is given by:

$$S(T,R) = \frac{\sum_{w_i \epsilon C} f_i(R) \times f_i(T)}{\sum_{i=0}^{N} f_i(T) \times f_i(T)} \qquad (3)$$

Equation 3 returns the degree to which R overlaps T, normalized with the document T alone. The numerator works, as we said, on the frequencies of words in the pair of document from the set C. The relative similarity which is limited to the range 0 to 1, is given by:

$$similarity(T,R) = max\left[S(T,R), S(R,T)\right] \qquad (4)$$

where $S(R,T)$ is the same formula with reversed operands as;

$$S(R,T) = \frac{\sum_{w_i \epsilon C} f_i(R) \times f_i(T)}{\sum_{i=0}^{N} f_i(R) \times f_i(R)} \qquad (5)$$

Now we have all the information and data structures to be used in the implementation of SCAM formula. The class ScamEvaluation has evaluation as the main method. For each test phrase, according to equation 3, we calculated first the denominator then the determine sum of the squares of occurrences number of every term in test phrase. We call the denominator denom1 and this is how it is calculated:

```
while (iterator.hasNext())
Map.Entry¡String, Integer¿ element = iterator.next();
denom1 += (element.getValue().intValue())
(element.getValue().intValue());
```

This code iterates on occurrences' structure and sums the value of each term in denom1. The implementation of the formula follows this logic:

```
for each retrieved Document
get the terms
for each term
if the term appear in the test Document
get the test term frequency
get the term frequency
if condition EPSILON
calculate S(T, R)
calculate S(R, T)
end if
end if
end for
end for
```

The outer loop iterates on docVectors as shown in table 3 (a structure containing all retrieved documents associated to term frequency vector).

```
String[] terms = tfVector.getTerms();
int[] freqs = tfVector.getTermFrequencies();
Here is the second loop iterating on each term:
for (int i = 0; i < terms.length; i++)
if (tdocFreq.containsKey(terms[i]))
int f1 = tdocFreq.get(terms[i]);
int f2 = freqs[i];
```

For each term in test phrase, tdocFreq is the data structure enclosing the bag-of-words from test phrase, (table 1); f2 is the frequency of test term and f1 is the frequency of the term we are processing. According to SCAM formula, now we can calculate:

```
if ((EPSILON - ((f1 / f2) + (f2 / f1))) > 0)
double delta = 1.0 * f1 * f2 / denom1;
docScores1.put(filename, docScores1.get(filename) + delta);
delta = 1.0 * f1 * f2 / denom2;
docScores2.put(filename, docScores2.get(filename) + delta);
```

EPSILON is a constant with a value of 2.5, declared private static final double EPSILON = 2.5. If the condition is true, we calculate S(T; R) and S(R; T), with denom1 already found and denom2 contained in the data structure created previously (each document retrieved was associated with the sum of its terms frequencies, figure 2. At the end of the loop we have two data structures containing the documents with relative score in both cases S(T, R) and S(R, T). The similarity measure is the maximum of the values associated to the document:

```
double score1 = docScores1.get(filenam);
double score2 = docScores2.get(filenam);
double score = ((score1 > score2) ? score1 : score2);
if (score > THRESHOLD)
/ * the document is probably plagiarized * /
```

THRESHOLD is a value we set to 0.9, declared as private static final double THRESHOLD = 0.90;

Our goal has been to fill a database table, precisely *assoc* shown in figure 2 where we associated train phrase with the test phrase with their relative score. All needed elements to fill the table *assoc*, are now available:

- id_{tr} : from Lucene field *id*.

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

- id_{te} : from test phrase *id*.
- *score*: from SCAM similarity measure.

The table *assoc* contains all the information to show it to the user; a screen shot of this table is given in figure 4. We can see that id_{tr} 392 (which corresponds to a text phrase in the table *phrase*), and id_{te} 7 (which corresponds a text phrase in the table *testphrase*) have a score of 100 %, and so on.

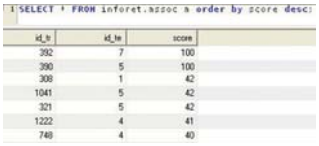

Figure 4: assoc

2.7 Graphical User Interface

We developed a simple GUI allowing the user to accomplish two main tasks as indexing a dataset and checking for plagiarism. It has been chosen a windows interface, common and easy to understand; the screen shot in Figure 5 represents the main window. The index window includes the path of the directory containing the documents to index (Figure 6). The list of indexed documents and time are shown to the user. Figure 7 shows some phrases from the selected document and some suspected plagiarized phrases.

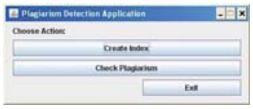

Figure 5: Screen-Shot Main Window

3 Conclusion

The two fundamental concepts for the evaluation of an information retrieval system are:

- Precision: Expresses the number of retrieved documents that are relevant over the number of retrieved documents, a percentage of how many documents are relevant among the retrieved ones.
- Recall: Expresses the number of retrieved documents that are relevant over the number of relevant documents present in the data source, a percentage of all relevant documents that are retrieved.

These two parameters are necessary for establishing the effectiveness of plagiarism detection. The copy detection technique we developed works good in finding exact or partial plagiarism copies; the algorithm is able to precisely detect copy of entire phrases or parts of them. Rewording, in terms of changing words by synonyms were also identified relatively easily. The use of the WordNet module, in order to expand the vocabulary of terms with similar words, increases the effectiveness of our algorithm; it must be clarified that it is an attempt to cover simple forms of paraphrasing; a more complex linguistic analysis should be necessary to be able to discover more complex rewrites. The analysis of suspicious documents and phrases involves a great amount of comparisons, especially when the synonyms' substitution is taking place for each sentence; however it has been noticed that the behavior of the system is constant and acceptable, in terms of performance. About accuracy, from a Precision and Recall perspective, best results are obtained starting with a similarity threshold higher than 70% - 80%.

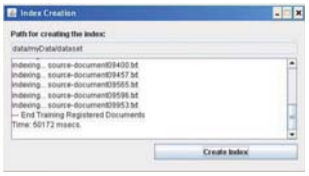

Figure 6: Screen-Shot Create Index Window

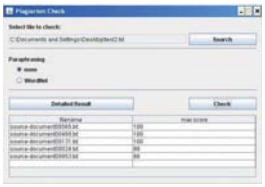

Figure 7: Screen-Shot Detailed View in Browser

References

[1] C. Justicia de la Torre, Maria J. Martn-Bautista, Daniel Sanchez, and Mara Amparo Vila Miranda. Text mining: intermediate forms on knowledge representation.

[2] Manu Konchady. Building Search Applications: Lucene, Lingpipe, and Gate. Mustru Publishing, Oakton, Virginia, 2008.

[3] Higher Education Academy ICS (Information and Computer Sciences) University of Ulster. Plagiarism prevention and detection. (n.d.). Retrieved March 17, 2010, from http://www.ics.heacademy.ac.uk/resources/assessment/plagiarism/detectplagiarism.html.

[4] Oracle Corporation. Mysql, 2010. Retrieved April 12, 2010, from http://www.mysql.com.

[5] Eduard Montseny and Pilar Sobrevilla, editors. Proceedings of the Joint 4th Conference of the European Society for Fuzzy Logic and Technology and the 11th Rencontres Francophones sur la Logique Floue et ses Applications, Barcelona, Spain, September 7-9, 2005. Universidad Polytecnica de Catalunya, 2005.

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)