Accuracy of Unit Under Test Identification Using Latent Semantic Analysis and Latent Dirichlet Allocation

1st Matej Madeja
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Košice, Slovakia
matej.madeja@tuke.sk

2nd Jaroslav Porubän
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Košice, Slovakia
jaroslav.poruban@tuke.sk

***Abstract*—Identification of unit under test (UUT) from a test is often difficult and requires wider source code comprehension. By automating this process it would be possible to support the program comprehension and reduce software maintenance process. In this paper the Latent Semantic Analysis (LSA) and the Latent Dirichlet Allocation (LDA) were used which proved to be inaccurate in the UUT identification. The experiment was conducted on 5 popular projects where 1,093,730 similarity results were obtained. It was found out that the best topic number for the LSA model is from 7 to 10, the LDA model had big differences in this value, so it was not possible to define a stable value. The best UUT identification accuracy compared to manual testing has been obtained with the LSA model with result of 7.63% success, where documents were preprocessed using words splitting based on naming conventions and Java keywords removal. The accuracy of the LDA model was almost zero. Further 8 manual identification errors were discovered during the experiment.**

***Index Terms*—program comprehension, latent semantic analysis, latent dirichlet allocation, github mining, unit under test**

I. Introduction

Natural language processing (NLP) attempts to reduce the barriers in computer-to-human communication [1]. This process involves the correct text analysis, then the determination of its semantics and the execution of the required action. During the implementation of the program source code the programmer writes statements in the prescribed syntax of a non-natural language but many times expresses thoughts that are important to facilitate program comprehension in the future – either for himself/herself or for other programmers. These thoughts are expressed mainly by following naming conventions (e.g. Java Coding Style Guide [2]) which task is to simplify the representation of the code in the problem domain, e.g. by semantically correct naming of classes, methods, variables, etc. Based on the research by Butler et al. [3] it can be seen that the names of the identifiers have a significant impact on the information mining from the source code and its comprehension by the programmer.

In our previous research [4] we focused on word frequency analysis between test and production classes in 5 popular Android projects on Github[1] and general testing practices. Because the tests can be considered always up-to-date documentation of the production code we found out that the words used by the programmers in the tests and the production code are very similar. It was also found that 49% of the analyzed test titles included the full name of a particular unit under test (UUT) and even 76% the partial name. The production and test method bodies also used similar vocabulary. A lot of vocabulary is placed in comments in the form of natural language.

From the above it is possible to assume that each source code file uses its own vocabulary. The used words are not in the form of sentences so it is not possible to search for semantics between words within the final set of words (sentence). But each document contains a set of words that together characterize a whole, e.g. in Java the most common a single file will represent a class.

A general problem during the test code comprehension is the UUTs identification, especially when one test class tests multiple production classes. According to McGlauflin [5] in Java one production class should be tested by only one test class and the programmer is led to this convention also using an integrated development environment (IDE) tool. If this convention is followed it can be assumed that the test and production classes will have similar vocabulary.

The aim of this paper is to use 2 models of natural language processing (NLP) techniques: *Latent Semantic Analysis* (LSA) and *Latent Dirichlet Allocation* (LDA). These models are some of the least computationally complicated in the NLP field and could greatly help identify UUTs based on the vocabulary of test, thereby simplify program comprehension and prevent faults. In this paper the following research questions are discussed:

[1]https://github.com/

978-1-7281-3180-1/19/$31.00 ©2019 IEEE

RQ1: Is there a general topics number for processing source code files without the need of searching it?
RQ2: How exactly can UUT be identified from the test class vocabulary?
RQ3: How to preprocess source code documents for model training to the obtain best results?

In the Section II we briefly describe usage basics of LSA and LDA models to process natural text without mathematical details, since they are not important to the problem this paper deals with. The Section III describes the programming language selection, used libraries and data preparation for processing. The results are described in the Section IV and at the end of the paper threats to validity, related work, conclusions and future work are discussed.

II. LSA AND LDA MODELS

Both models are information retrieval (IR) algorithms that expect vectors as input, mainly because their nature is mathematical operations involving matrices. The input strings are therefore represented as vectors and this type of representation is called *Vector Space Model*. Based on these vectors a particular NLP model can make predictions. The aim of these algorithms is to train the model from the input data to minimize the occurrence of prediction errors.

There are different representations of text as a vector. The most straightforward representation is *bag-of-words* (BoW) and it is an orderless document representation, so only the counts of the words matter. This leads to loss of word order, syntactic relations, or morphology [6]. However, in our case most of the input data except comments will not have the form of natural text, so this is not critical. For most IR algorithms the frequency of word occurrence is sufficient for calculation. A slightly more sophisticated model is *tf-idf* (*term frequency-inverse document frequency*), which tries to encode two different kinds of information - term frequency and inverse document frequency [7]. Term frequency (*tf*) is the number of times the word appears in a document. It is possible to extend these models, for example using topic models. Both LSA and LDA can use *bag-of-words* model to obtain term-document matrix.

In the following subsections selected models without mathematical details are described. A detailed explanation of both models can be found in [8] and [9]. Cvitanic et al. [10] discuss models' differences in more detail.

A. Latent Semantic Analysis

LSA is an indexing and IR method that uses *Singular Value Decomposition* (SVD) to identify relationships between words in an unstructured text. The model is based on the assumption that words used in the same context have a similar meaning [8]. By extracting terms from the document's body it seeks to create relationships between individual documents. It is important to choose a right number of topics to generate because if too many topics are requested for a short document the algorithm returns also words that should not determine the resulting topic of the document and vice versa.

B. Latent Dirichlet Allocation

The model considers each document as a set of topics which characterize it [9]. Each topic consists of a set of words in a certain proportion. Based on the number of topics required the model attempts to rearrange the topics distribution within the documents to achieve the best composition. It is also very important to determine the right number of topics that the algorithm returns.

C. Number of topics

To evaluate the quality of a trained model, which is significant for results of both considered models, can be determined by topic coherence that is a measure used to evaluate topic models. The topic coherence is applied to the top N words from the topic and it is defined as the average/median of the pairwise word-similarity scores of the words in the topic [11]. A good model will generate coherent topics with high coherence scores. Good topics are those that can be described by a short label.

III. METHOD

The experiment was conducted on 5 popular Android projects from our previous research [4]. In order to know the success rate of particular NLP models in identifying a UUT from a test it is necessary to establish a link between the test and the production classes. Since we performed a manual analysis of 617 tests in [4] we can partially use the collected data for this experiment. We assume that manually created links are correct. The source codes of considered projects (see Table I) are from February 2019 to preserve consistency with manually collected data. The projects were selected on the assumption that the most popular projects will include tests (see more in [4]).

TABLE I
GENERAL STATS OF MANUALLY ANALYZED DATA. [4]

Project	Prod. classes	Prod. methods	Test classes	Test methods
plaid	37	71	39	180
ExoPlayer	49	98	53	323
Android-Clean Architecture	17	22	17	29
shadowsocks-android	6	7	6	8
iosched	16	40	16	77
SUM	**125**	**238**	**131**	**617**

The Table I shows that in most cases the convention that one UUT (production class) is tested by one test class has been fulfilled, so in 125 cases we can clearly establish the expected connection between the test and production class. For tests that test multiple production classes the most tested production class will be considered as correct UUT.

A. Programming language and library

For projects' source code analysis we chose Python language which is great for processing computationally difficult tasks. Also availability of *gensim* [12] library for this language

was crucial. The library is very popular in the NLP field and according to its author Řehůřek [13] *gensim* is the most robust, efficient and hassle-free piece of software to realize unsupervised semantic modeling from plain text.

B. Documents preparation

All analyzed projects are built on the Android platform and implemented in Java and/or Kotlin. For each project we recursively searched for files having the `.java` or `.kt` extension. Kotlin is designed to interoperate fully with Java so they use similar programming conventions and it is also suitable for our analysis. Of course, only the files of the project were included in the analysis, i.e. without dependencies and platform software development kit files. We already knew the names of the test classes from [4] so we divided the particular files into test and production set. The content of production classes served for model training and content of test classes for searching similarity.

File preprocessing was the same for both test and production classes. From the content of each file new line characters have been removed and result was saved in a project-depended training file. One line in this file represented one document for further processing.

Since the result is highly dependent on the quality of the input data and is governed by the idiom *"garbage in, garbage out"*, it is very difficult to assume in a non-natural text how it should be properly preprocessed. To find out how to prepare input documents in the best way (RQ3), we incrementally created 5 versions of document preprocessing:

1) *Full version* - original file version, removed only `\n` chars.
2) *Word split* - all `camelCase` or `snake_case` words has been split. Words out of base conventions, such as `ORMLite`, remained unsplit.
3) *Removed Java keywords* - all Java keywords have been discarded.
4) *Removed comments* - multi- and one-line comments discarded.
5) *Removed imports* - all Java imports removed.

Incremental preprocessing means that for example in the 4th iteration also *Word split* and *Removed Java keyword* have been included. Another preprocessing of documents that applied to all iterations was the removal of frequently occurring English words using *nltk* library, such as *and*, *a*, *the*, etc. At the same time stemming over the documents has been executed, where inflected or sometimes derived words to their word stem have been reduced, e.g. *cars* to *car*. The last step was to remove words that occurred only once in the corpus of training documents to eliminate their negative impact on results.

C. Model training

Creating a *bag-of-words* representation in the form of a dictionary (*id* + *word* pair) and creating a corpus of sparse vectors was relatively easy using the functions of *gensim*. As mentioned in the Section II, choosing the right number of topics is a much bigger challenge. Since the number of topics is dependent on the nature of the documents we searched for the highest coherence value for each iteration and project.

Training a large number of models is very time consuming task so in the early stages of the experiment we tried to obtain approximate range to try in. After multiple tests we decided to calculate coherence values from 7 to 50 for each model, project and iteration, and the model with the highest value has been chosen for the analysis.

To train the LSA model only the necessary parameters were supplied - number of topics, dictionary (BoW) and corpus (of vectors). For the LDA model training we also set the *alpha = auto* parameter which means the model learns asymmetric prior from provided corpus. From *alpha* attribute LDA model computes *theta* that decides how the topic distribution is drawn. The last special parameter set for the LDA model was *passes = 20* which express the number of passes through the corpus during training. In terms of statistics, more training means statistically more accurate results.

D. Evaluation of document similarity

When using LSA or LDA we focused on a single aspect of possible similarities, i.e. on apparent semantic relatedness of their texts (words), just a semantic extension over the boolean keyword match. Modern search engines also take into account random-walk static ranks, hyperlinks, etc. *Gensim* basically uses *cosine similarity* [14] to determine the similarity of two vectors and it is a standard measure in *Vector Space Modeling*.

The search query was made up of the content of a test class. We created *bag-of-words* for each document (test body) and converted it to the corresponding LSI/LSA space. Subsequently, an index has been created from the trained model against which the query was evaluated. Similarities to all production classes were calculated and we obtained the result as $(document\_no, similarity\_value)$ pairs where $similarity\_value \in \langle -1, 1 \rangle$. The greater *similarity_value* the more similar document. Every *document_no* has been paired with value stored in a relation database created during training document preparation to identify particular production class.

IV. Results

Altogether we analyzed 2221 production and 168 test classes in five projects (see Table II). In five iterations of document preprocessing a total of 1,093,730 similarity results between the tests and the production source code have been obtained. In the Table II also statistics of mean search time in prepared index are presented, as can be seen, searching in LDA index is mostly a little bit faster.

A. Optimal topic number and train times

Model training lasted the longest time especially because of searching for the optimal topic number. The tests were performed on 12-core *Intel Core i7-5820K* CPU with 12GB RAM and *Debain GNU/Linux 9* installed. Nevertheless, analysis was performed in one thread only to simulate the use in real environment, e.g. in the background of IDE. The average search times for the highest coherence value can be seen in the

TABLE II
GENERAL DOCUMENT STATISTICS IN ANALYZED PROJECTS.

#	Project	Number of files		Mean search time in index (1 query)	
		prod.	tests	LSA	LDA
1	plaid	679	39	49.73 ms	43.97 ms
2	ExoPlayer	954	53	78.52 ms	88.33 ms
3	Android-Clean Architecture	99	17	7.65 ms	3.53 ms
4	shadowsocks-android	157	6	11.67 ms	10.00 ms
5	iosched	332	16	18.89 ms	13.33 ms

Table III. The differences in individual LDA iterations were minimal, i.e. in tens of seconds, with the LDA model the differences were even a few minutes. The greatest decrease of training time in a particular NLP model was recorded in the 4th iteration when comments were removed. By removing them a lot of the training data have been lost so finding the best coherence value in the 4th iteration was faster: 2.17 times for LSA and 2.83 times for LDA. Despite the increased speed a large amount of potentially natural text in the source code has been lost.

TABLE III
AVERAGE TIMES OF SEARCHING BEST COHERENCE VALUES PER PROJECT IN MINUTES.

Project #	1	2	3	4	5
LSA	3.96m	4.46m	0.97m	2.03m	1.36m
LDA	21.31m	32.66m	2.26m	6.84m	7.18m

- to identify particular project pair with Table II.

To answer **RQ1** the Table IV was created. As can be seen the mean value for LSA is relatively stable. Although the maximum deviation of the LSA value is 40, this situation occurred in only one case, i.e. it was just an exception that could be neglect. For the source code analyses using the LSA model it is therefore possible to use a relatively stable value of topic numbers in the range 7–10.

TABLE IV
TOPIC NUMBER MODE VALUE, TOPICS NUMBER DIFFERENCES AND MODEL TRAINING TIME WITH BEST COHERENCE VALUE PER PROJECT.

Subject	Metric	Model	Project #				
			1	2	3	4	5
Topic No.	**mode**	**LSA**	7	7	7	10	10
		LDA	-	-	14	12	19
Topic number difference	**min**	**LSA**	0	0	0	0	0
		LDA	1	1	0	0	0
	max	**LSA**	1	6	40	6	5
		LDA	11	20	25	17	23
Best model train time (s)	**min**	**LSA**	0.09	0.14	0.02	0.03	0.03
		LDA	8.92	11.27	1	2.88	3.82
	avg	**LSA**	0.14	0.19	0.02	0.04	0.05
		LDA	12.35	24.31	1.29	4.27	5.3
	max	**LSA**	0.2	0.3	0.02	0.05	0.08
		LDA	17.86	44.74	1.5	8.37	7.23

- to identify particular project pair with Table II.

When using LDA the selected topic numbers were very diverse as the model is less stable. This can be obtained in multiple model training with the same data when the results vary slightly. That's why topic numbers are more diverse than in LSA. Most often the topic number for a project was in similar values and in the difference range of 10 units. However, the differences between the particular projects were large and based on this data it is not possible to determine the recommended topic number for LDA.

B. Accuracy of UUT identification

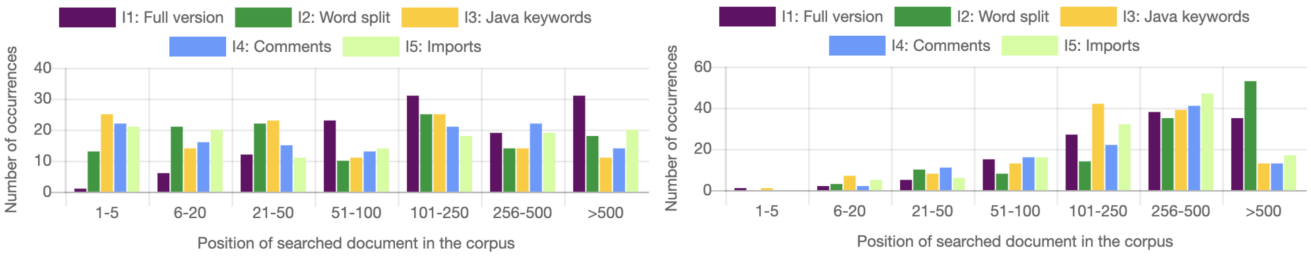
Since we assume that manually identified UUTs are correct it is possible to determine the accuracy of a particular model based on the order of production class in the search result. The Figure 1 shows the frequencies in the search queries for manually labeled production classes as UUT.

As can be seen, LSA performed much better than LDA. The LSA is based on the frequency of words in the documents and as was found in [4], the words between the test and UUT are very similar which positively influenced the result. The accuracy of the LDA model was very low, in the first five results the correct UUT appeared only 2 times. Although the LSA achieved 82 correct UUTs in the first five results for all iterations it is still only 13.33% success rate which is considerably inadequate. In response to **RQ2** from our results, only 5.20% of UUTs were marked correctly (all iterations) and solely by the LSA method. In the results it is necessary to take into account the fact that for the 6 test classes, which tested multiple production classes at once, the most tested class was chosen as the correct UUT (discussed in Section III).

A more detailed look at the best search results in each iteration is needed to respond the **RQ3** (see Table V). As can be seen, *word split* and *removal of java keywords* (I2 + I3) has the greatest impact on the accuracy of the results. Our expectation was that when comments are removed the results will get worse because there is a potential for sole natural language in the comments. In the Table V it can be seen that removing comments (I4) and imports (I5) had a negligible impact on accuracy. It also shows that the meaning in the code is most often expressed directly in the names of the identifiers, i.e. class, methods and variables names. Using word splitting were obtained the most accurate results, it was the fastest iteration in terms of model training, finding the best coherence value and search in the index (Table II).

TABLE V
POSITION FREQUENCY FOR FIRST 5 POSITIONS OF ANALYZED MODELS.

Iteration	Position frequency in the search											
	LSA						LDA					
	1	2	3	4	5	Σ	1	2	3	4	5	Σ
I1		1				1			1			1
I2	4	1	3	5		13						
I3	10	8	2	4	1	25			1			1
I4	9	6	4	1	2	22						
I5	9	6	2	3	1	21						
Σ	32	22	11	13	4		0	0	2	0	0	



(a) LSA

(b) LDA

Fig. 1. Position frequency of manually identified UUT for LSA and LDA models.

C. Detected errors of manual identification

In the experiment, we unexpectedly found that 8 classes manually labeled as UUT were not in the corpus. Manual testing was done in *Android Studio* IDE where we used references created directly by IDE. After a more detailed analysis we found that production classes that were not in the corpus were incorrectly labeled. Incorrect identification was due to references to generated source code that were not present in the file system of the project without run. The method can be therefore used to prevent such errors.

V. Threats to validity

Comparison the accuracy of LSA and LDA was relied on the fact that the manual identification of UUTs was performed correctly. If an error in manual identification happened this could have a negative effect on the results reported in this paper. The analysis was performed on only 5 popular Android projects and no other projects were included, e.g. less popular, proprietary, etc. At the same time, projects in other languages have not been analyzed and a particular language can affect the corpus of words by using naming conventions or language syntax.

While the LDA method is more accurate than the LSA (claim from the official description of the method, not from our results), on the other hand, this method also shows slight differences in document comparisons when training the LDA model multiple times with the same data, indicating some inaccuracy but statistically friendly.

The preprocessing of documents (the *garbage in, garbage out* idiom) has also a huge impact on the results. How to prepare source codes for such analysis was also one of the research questions of this paper. There exist a thread to validity because not all possible document preparation could be tried out.

As mentioned in Section II the choice of topics number also has a big impact on the accuracy of the methods. Despite finding the best value for this parameter a search range of 7 to 50 may not be sufficient. Also the use of coherence value may not be reliable at all times and there is no general recommendation on how to accurately determine this parameter, so there is no guarantee that the best values have been chosen with respect to the input data.

VI. Related work

The most similar research on improving of program comprehension was done by Maletic et al. [15], [16]. In their conclusions they argue that the LSA model can assist in supporting some of the activities of the program's comprehension process. However, they only analyzed one project in C, in our case a larger sample and Java and/or Kotlin languages are considered. They analyzed 269 files, we analyzed on a sample of 2221 files in all projects without 168 test files used as queries on index. They created code clusters of similar files trying to make it easier for the programmer to find related parts of the program. In our case we focus on the relationship between the test and the production class which can even be written in another language (e.g. tests in Java, production code in Kotlin; see *plaid* project) and assume that the UUT test will have more common vocabulary as 2 different classes.

Another type of research was performed by Thomas et al. [17], [18] in 2014 who mined software repositories using topic models to simplify the understanding of software changes during software evolution, especially for stakeholders. Although their experiments have not been verified on real stakeholders the results show that extraction of topics is sufficient and should therefore have a positive impact on simplification of understanding. In our research we focus more on developers, analyzing the source code and relationships within it.

Asuncion et al. [19] proposed an automated technique that combines traceability with topic modeling. They record traceability links during the software development process and learns a probabilistic topic model over different artifacts. From collected data they are able to categorize artifacts and create topical visualisations of particular system. They implemented several tools that support data collecting during software evolution. In our case, we are still in the early stages, so we found out whether it is beneficial to look for similarities between the test and the production code using NLP techniques.

VII. Conclusions and future work

In this paper the use of 2 NLP methods for UUT detection in popular open-source Github projects is discussed. Previous research shows that using these methods for source code

analysis can positively affect the program comprehension of developers.

Five popular Android projects were used for the analysis which included 131 test and 2221 production classes in total. In experiment *Latent Semantic Analysis* and *Latent Dirichlet Allocation* models have been used which are able to evaluate the similarities between documents. The source code of the production classes was used to train the models and the content of the test classes were used for search queries. In the paper is described how to find the best value of topic number. The main objective was to find out how exactly these models cans identify unit under test against manual identification. The experiment was conducted in five iterations and for every iteration training and search documents have been modified in different ways to determine how to preprocess source code documents for these models to get more accurate results .

It was found that a range of topic number from 7 to 10 is suitable for the LSA model and the choice of topic number was very varied for the LDA model, so a generally sufficient value could not be found. The accuracy of the LDA model was only 5.20%, after taking into account the small deviation only 13.33% records were found in the top five search results. The accuracy of the LDA model is almost zero, so it is not at all suitable for UUT identification. The fact that the LSA model was more successful in the experiment corresponds to the results of our previous research [4] that the vocabulary used in the test and UUT was often similar. The LSA method could therefore be a partial complement for more accurate UUT identification but not absolutely reliable.

The best results in terms of searching for the best coherence value, model training time, searching for similarity and UUT identification were recorded for 3rd iteration with the accuracy of 7.63%, taking into account the deviation of up to 5 records 19.08%. The mentioned iteration included: removing new lines; splitting words using camelCase and snake_case convention; removing Java keywords; removing English general words; stemming. At the same time, the class lookup notified us to 8 mislabeled UUTs during manual testing in which dynamically generated classes were identified as UUTs.

Despite the considerable failure of the methods the experiment also needs to be performed outside of Android projects and on a different sample of projects, e.g. in terms of length of maintenance, number of tests, language, etc. For example *Gherkin* which contains more of natural text the results could provide better results. In the future we will also look at comparing the results with other UUT identification techniques, e.g. observation of code co-evolution, helper methods, or naming conventions.

ACKNOWLEDGMENT

This work was supported by project VEGA No. 1/0762/19: Interactive pattern-driven language development.

REFERENCES

[1] C. D. Manning, C. D. Manning, and H. Schütze, *Foundations of statistical natural language processing*. MIT press, 1999.

[2] A. Reddy *et al.*, "Java™ coding style guide," *Sun MicroSystems*, 2000.

[3] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Mining java class naming conventions," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2011, pp. 93–102.

[4] M. Madeja and J. Porubän, "Tracing naming semantics in unit tests of popular github android projects," vol. 74, 2019. [Online]. Available: https://www2.scopus.com/inward/record.uri?eid=2-s2.0-85071071510&doi=10.4230%2fOASIcs.SLATE.2019.3&partnerID=40&md5=6f5044423719630d3eaa7ed15b351def

[5] B. McGlauflin, *Java Unit Testing Best Practices: How to Get the Most Out of Your Test Automation*. DZone Technical Library, 05 2019. [Online]. Available: https://dzone.com/articles/java-unit-testing-best-practices-how-to-get-the-mo

[6] W. B. Croft, D. Metzler, and T. Strohman, *Search engines: Information retrieval in practice*. Addison-Wesley Reading, 2010, vol. 520.

[7] D. Hiemstra, "A probabilistic justification for using tf×idf term weighting in information retrieval," *International Journal on Digital Libraries*, vol. 3, no. 2, pp. 131–139, Aug 2000. [Online]. Available: https://doi.org/10.1007/s007999900025

[8] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[9] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[10] T. Cvitanic, B. Lee, H. I. Song, K. Fu, and D. Rosen, "Lda v. lsa: A comparison of two computational text analysis tools for the functional categorization of patents," in *International Conference on Case-Based Reasoning*, 2016.

[11] J. H. Lau, D. Newman, and T. Baldwin, "Machine reading tea leaves: Automatically evaluating topic coherence and topic model quality," in *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, 2014, pp. 530–539.

[12] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[13] R. Řehůřek, *About Gensim*, 07 2019. [Online]. Available: https://radimrehurek.com/gensim/about.html

[14] A. Huang, "Similarity measures for text document clustering," in *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, vol. 4, 2008, pp. 9–56.

[15] J. I. Maletic and A. Marcus, "Using latent semantic analysis to identify similarities in source code to support program understanding," in *Proceedings 12th IEEE Internationals Conference on Tools with Artificial Intelligence. ICTAI 2000*, Nov 2000, pp. 46–53.

[16] J. I. Maletic and N. Valluri, "Automatic software clustering via latent semantic analysis," in *14th IEEE International Conference on Automated Software Engineering*. IEEE, 1999, pp. 251–254.

[17] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Studying software evolution using topic models," *Science of Computer Programming*, vol. 80, pp. 457–479, 2014.

[18] S. W. Thomas, "Mining software repositories using topic models," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1138–1139.

[19] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 95–104.

Comparison of Predictive Statistical Learning Accuracy with Computational Intelligence Methods

Dusan Marcek
Institute of Informatics
Silesian University in Opava
Opava, Czech Republic
dusan.marcek@fpf.slu.cz

Abstract— **Forecasting of high-frequency economic time series data is a complex problem, which has benefited from recent advancements and research in machine learning. To forecast time series data, two methodological frameworks of statistical and computational intelligence modelling are considered. The statistical methodological approach is based on the theory of invertible ARIMA (Auto-Regressive Integrated Moving Average) models. The Computational Intelligence (CI) models are based on neural networks (NN) and Support Vector Machines (SVM). The main work of this study is to compare the predictive accuracy level of the statistical methodological approach with NN and SVM on the large data set. We evaluate statistical ML (Maximum Likelihood) learning method, Back-Propagation (BP) and genetic algorithms (GA) for half-hourly 1-step-ahead electricity demand prediction using Australian electricity data. We showed that all ARIMA, NN, SVM models as prediction methods are reasonable and acceptable for use in forecasting systems that routinely predict values of variables in competitive energy markets.**

Keywords— ARIMA models, SVR, Neural networks, Learning algorithms, Roulette wheel.

I. INTRODUCTION

Electricity demand prediction is very important for the reliable and efficient operation of power systems. We consider predicting the electricity demand half hour ahead from previous half-hourly demands. This type of prediction is used for two main purposes: (1) to make decisions about dispatching generators and setting the minimum reserve during the daily operation of power systems and (2) to provide information to electricity market participants for their bidding in competitive energy markets. In both cases the goal is to ensure reliable electricity supply while minimizing the

There are three main groups of approaches for electricity demand prediction: the traditional causal (econometric), time series [1] and the more recent computational intelligence method, such as NN, SVM. In this paper we develop and compare the performance of the classic (that is of the perceptron type) RBF (Radial Basic Function) NN and RBF NN trained by GA with ARIMA models. The goal of this paper is to illustrate that the two distinct approaches, i.e. statistical models and computational networks may be used for financial high-frequency time series modelling.

The paper is organized as follows. In Section II we briefly describe the variants statistical ARIMA models (ARMA and seasonal ARMA model) and SVR model. In Section III we present the data, conduct some preliminary analysis of the time series and demonstrate the forecasting abilities of classic/seasonal ARMA and SVR models. Section IV describes design and application of RBF NN trained with BP and GA algorithms. Section V presents results and empirical comparison. Section VI briefly concludes.

II. STATISTICALTIME SERIES MODELS

A. ARIMA time series models

ARIMA time series model belongs to the group of Box and Jenkins [2] methods. The Auto-Regressive (AR) process is a linear combination of previous values, the Moving-Average (MA) process is a linear combination of previous errors, 'I' is an operator for differencing a time series. An *ARMA*(*p, q*) model of orders *P* and *Q* is defined by

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \ldots + \theta_q \varepsilon_{t-q} \quad (1)$$

where $\{\phi\}$ and $\{\theta_j\}$ are the parameters of the autoregressive and moving average parts respectively, and ε_t is white noise with mean zero and variance σ^2. We assume that ε_t is normally distributed, i.e. $\varepsilon_t \sim N(0, \sigma^2)$.

B. Seasonal ARMA Models

An extension of the ARMA process is a pure seasonal model abbreviated as $ARMA(P, Q)_s$ process in the form

$$y_t - \lambda_1 y_{t-s} - \lambda_2 y_{t-2s} - \ldots - \lambda_P y_{t-Ps} - \varepsilon_t = \gamma_1 \varepsilon_{t-s} + \gamma_2 \varepsilon_{t-2s} + \ldots + \gamma_Q \varepsilon_{t-Qs} \quad (2)$$

where $\{\lambda_i\}$ are the seasonal autoregressive parameters, $\{\gamma_j\}$ are the seasonal moving average parameters and the subscripts *s* denote nonzero parameters that are integer multiple of *s*. The pure seasonal models defined by (2) are often not realistic since they are completely decoupled from each other. That is, (2) represents *s* identical but separate models and we need to take into account the interactions or correlations between the time series values within each period. This can be done by combining the seasonal and regular effects into a single model. We will use a multiplicative seasonal autoregressive integrated moving average process of period *s* (*SARIMA(p, d, q)(P, D, Q)*, with regular and seasonal AR orders *p* and *P*, regular and seasonal MA orders *q* and *Q*, and regular and seasonal differences *d* and *D*. In typical application, *D* = 1, the model is defined in Section III.

C. SVR model

Support Vector Regression (SVR) model is an extension of the SVM algorithm for numeric prediction and is

978-1-7281-3180-1/19/$31.00 ©2019 IEEE