TakeLab

Laboratorij za analizu teksta i inženjerstvo znanja
Text Analysis and Knowledge Engineering Lab
Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva
Unska 3, 10000 Zagreb, Hrvatska

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BSc THESIS No. 5328

Computational Analysis of the Similarity of Math Word Problems

Doria Šarić

Zagreb, June 2017

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
BACHELOR THESIS COMMITTEE

Zagreb, 3 March 2017

BACHELOR THESIS ASSIGNMENT No. 5328

Student:	**Doria Šarić (0036482792)**
Study:	Computing
Module:	Computer Science

Title: **Computational Analysis of the Similarity of Math Word Problems**

Description:

Computational analysis of math word problems is an interesting research area situated at the intersection of natural language processing and symbolic computation, with applications in education and teaching. Math word problems differ in various aspects, including their type, difficulty, and wording. A computational analysis of these aspects could make it possible to group similar problems together and develop a taxonomy, which could in turn be used for more efficient teaching of math.
The topic of this thesis is the clustering of math word problems, based on the analysis of a series of each problem's features, including its wording and step-by-step solution. Study the methods for unsupervised machine learning, clustering in particular. Devise and implement a method for grouping math problems which, among others, will rely on the trace produced by the symbolic solver developed by PhotoMath. Compile a suitable test collection with multiaspect similarity judgments. Carry out an experimental evaluation of the clustering method on the test collection. Apply the method on a larger collection of math problems provided by PhotoMath, and visualize the results. All references must be cited, and all source code, documentation, executables, and datasets must be provided with the thesis.

Issue date:	10 March 2017
Submission date:	9 June 2017

Mentor:

Associate Professor Jan Šnajder, PhD

Committee Chair:

Full Professor Siniša Srbljić, PhD

Committee Secretary:

Assistant Professor Tomislav Hrkać, PhD

I would like to thank my mentor, Jan Šnajder, for all the patience, good advice and putting me back on the right path when I felt a bit lost.
A thank you to PhotoMath for providing me with this idea and a chance to solve it.
Many thanks to my family, Ines and Luka for all the bad jokes during this last semester.

CONTENTS

1. Introduction

Mathematical knowledge has never been so reachable in a digital form like it is nowadays. In today's world, we are using software to solve mathematical problems and having all the answers in a matter of seconds. To come where we are standing today, much of the mathematical literature must have been digitalized. If we looked back on the last decade, we would notice that enormous amounts of world's data have been generated. That enabled machine learning algorithms to evolve and with the development of machine learning and artificial intelligence in general, software for solving mathematical tasks improved and became much more powerful and accurate.

In recent times, machine learning (henceforth also referred to as ML) and artificial intelligence, in general, gained a lot of attention and popularity due to its applicability in many fields. ML is a field of computer science that focuses on building systems that can learn by experience, without being explicitly programmed. That allowed programs to solve mathematical tasks similar to the ones in the knowledge base. About ten years ago, Wolfram Alpha was first introduced. Since then a lot has changed. Ten years is not really a long period of time, except when it comes to technology. Today we also have PhotoMath's application that we can use to solve various mathematical tasks by just scanning our notebooks with our mobile phones. This thesis is about classifying such computer-solvable math problems into different areas of mathematics.

Classification is the problem of identifying to which of a predefined set of labels a new data instance belongs. There are two main approaches to machine learning – supervised and unsupervised learning. In supervised classification, some examples in the data set are labeled (e.g., category is already known for some tasks). To predict unlabeled tasks, we first need to train the classifier on some labeled data. These labels are given to the model during the learning process. In unsupervised learning, by contrast, the model is not provided with any correct answers and groups data into classes based on some measure of similarity between them.

If all tasks would be mapped to corresponding categories, the knowledge could be appropriately organized into mathematical areas. Furthermore, the system could then give the user recommendations to solve similar tasks and strengthen their knowledge in this area. Also, if we could predict task category from the task itself, solving the task could be based on category prediction and become faster and simplified.

In the next chapter of this thesis, I will introduce you to the data I was working with. The third chapter will describe the models that I used in my solution. In chapter four results of this project will be presented and interpreted. Finally, in the last chapter I will shortly explain my thoughts and conclusions on this project.

2. Dataset

In 2014 a company named PhotoMath launched a software that scans photos of mathematical tasks, recognizes them using OCR and shows the user steps to solve them. Optical Character Recognition, or OCR, is a technology that makes it possible to convert different types of documents such as images, scanned media or PDFs into editable and searchable data. To provide users with a step-by-step solution, PhotoMath built their own solver. After parsing a mathematical task that has been scanned, PhotoMath app converts this string to some agreed prefix command form which is then being passed to the solver. If the solver managed to resolve the given task, it then outputs steps explaining to the user how it came to the solution.

For this thesis PhotoMath provided me with the dataset of 2829 labeled tasks. Each task belongs to a certain mathematical category. All possible categories are listed in the category taxonomy they have given me. These categories do not include all areas of mathematics, but have been defined manually by mathematicians working in PhotoMath. The taxonomy includes only math areas that PhotoMath can work with. Those categories are hierarchically related. Each task can be labeled with only one category. If the category that the task is labeled with is of depth 4 in a category tree, then this task also belongs to each category that is contained in the path from the root to the label.

Below is an example of one simple instance from the data set:

```
F.8.1.1;;Positive number within absolute value bars;;
(simplify_wrapper;(simplify;(abs;(abs_real;))));;abs(const(2));auto
```

Four columns describing the task are separated with a double semicolon.

```
category path ;; category name ;; solver steps ;; prefix command
```

First two columns are identifying the category that this particular task belongs to.

Each category is defined with a name and a path. Some categories in the taxonomy can have identical names. That is why the path is used to identify each category and additionally define relations between the categories.

Before explaining the remaining columns, it is important to mention that the category taxonomy is loaded from a separate file that contains all predefined mathematical categories. The category taxonomy is loaded into a tree structure. Each category path defines where this category is positioned in a tree of all existing categories. E.g., if the path to the category is F.8.1., it then means that this category is the first child of category F's 8th child. Each child node in the category tree is a subcategory, meaning parent categories are more general and define a mathematical area to a lesser extent. That is why all parent categories from the label to the root are also considered corresponding to the task.

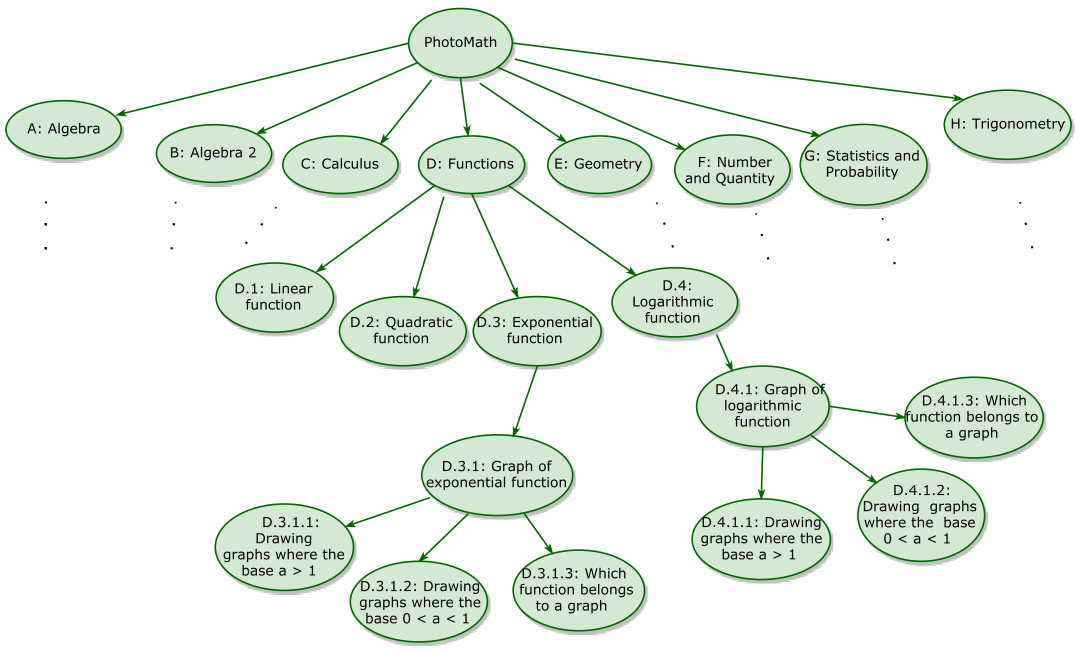

PhotoMath
A: Algebra
B: Algebra 2
C: Calculus
D: Functions
E: Geometry
F: Number and Quantity
G: Statistics and Probability
H: Trigonometry
D.1: Linear function
D.2: Quadratic function
D.3: Exponential function
D.4: Logarithmic function
D.3.1: Graph of exponential function
D.4.1: Graph of logarithmic function
D.4.1.3: Which function belongs to a graph
D.3.1.1: Drawing graphs where the base a > 1
D.3.1.2: Drawing graphs where the base 0 < a < 1
D.3.1.3: Which function belongs to a graph
D.4.1.1: Drawing graphs where the base a > 1
D.4.1.2: Drawing graphs where the base 0 < a < 1

Figure 2.1: An excerpt from the taxonomy for category "Functions".

Figure 2.1 shows us a part of the taxonomy. Besides the name of the category, each node also contains the appropriate path prepended to the category name.

The next or third column in the data set contains steps from the solver in a string form. The steps are written in pre-order tree traversal. Each step is surrounded by parenthesis and ends with a semicolon, except for some leaf nodes. Before parsing solver steps into ordered solver tree, the third column is being validated. If everything is in order, the solver tree is generated. The mentioned solver tree structure is the representation of the given mathematical problem. Since the data is not flat, but hierarchical, we will be doing multinomial classification on tree structures. This will be explained in detail in the next chapter.

Finally, the last column is not being used in this thesis project. However, this is a PhotoMath command for solving this task, written in prefix form. It may be involved in future work. For now, we have focused on category prediction based on solver trees.

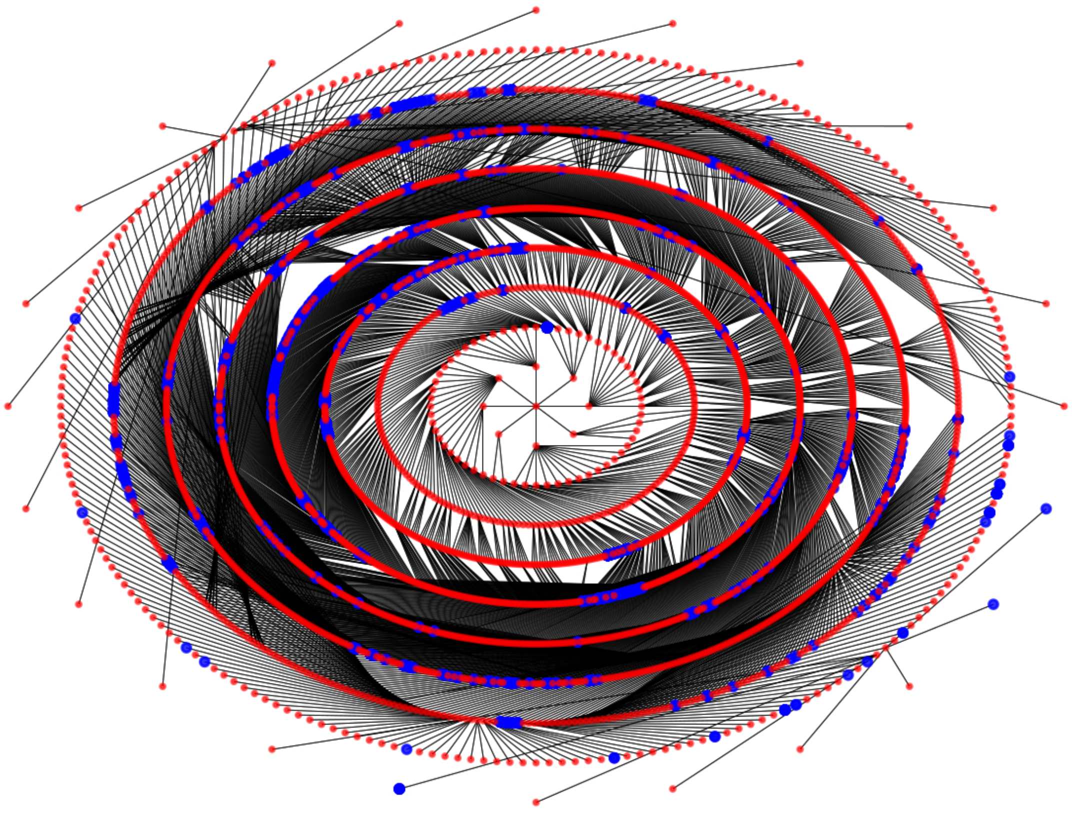


Figure 2.2: Sparsity of labels. Blue nodes represent labels that are existent in the data set.

However, besides working with hierarchical data there is another relevant obstacle to overcome. Only a subset of 565 labels from a set of 3707 possible categories is represented in this data set. This problem is known as the problem of “sparse labels”. Most of the categories here would not be present in the training phase and the system would never predict these categories because it wouldn’t know they existed.

In Figure 2.2, we can see the distribution of sparse categories in the data set. The graph is generated with the “networkx” library in Python. All labels that are present in the data set are coloured blue. Sparse labels are coloured red. Each ellipse is representing a certain depth in a categorytree.

To avoid this difficulty, I restricted the maximum depth (hereafter also called $level$) of training labels to n. That way the number of unrepresented labels in a data set would decrease significantly and categorization would be successful. A downside of this approach is that all categories of depth in a category tree that is larger than n would be generalized to a category of depth n, so the classification would be less precise. For example, if I chose a level of training labels to be three, all original labels will be transformed to categories of depth three or less.

3. Model

As we have already concluded, this problem demands hierarchical multiclass classification. “In machine learning, multiclass or multinomial classification is the problem of classifying instances into one of the more than two classes.” (Wikipedia, 2017) There are two main approaches to this problem. Trees are hierarchical data structures and deserve a different approach than flat data which is most often classified. The structure of the tree plays an important part in differentiating the data.

However, one approach is similar to the standard approach with flat data. The idea is to map instances from the data set to feature vectors. The structure of the data needs to be considered when building feature space. The other approach uses kernel methods and avoids directly operating on feature space.

3.1. Classifiers

In both approaches, I have used some already implemented classifiers from scikit-learn package in Python (Pedregosa et al., 2011). Some of the classifiers that I have tried out throughout this thesis are Support Vector Machine, LinearSVC, Naive Bayes and Random Forest.

3.1.1. Support Vector Machine

Support Vector Machine is a supervised learning model associated with algorithms for classification and regression. SVM maps instances from the data set so that the examples of the separate classes are divided by a widest possible gap. New instances are then mapped into the same space and predicted to a class (or a category) based on which side of the gap they fall. Besides being a linear classifier, SVM can be kernelized and used for hierarchical classification, implicitly mapping their inputs into high-dimensional feature spaces depending on the kernel function. In this thesis I have used SVM in both kernelized and standard way.

SVM Hyperparameters

SVM seeks to find a gap that separates all positive and negative examples. However, outliers – extremely unusual or mislabeled instances can strongly affect SVM's predictions in a negative way. That can lead to poorly fit models. To account for this there is a hyperparameter C that can be manually configured when working with SVM. The idea is to have a "soft margin" that allows some instances to be "ignored" or placed on the different side of the margin. Optimizing hyperparameter C often leads to a better overall fit. Parameter C controls the influence of each instance – each support vector in training the model. A small C means lower variance, but higher bias. Larger C means lower bias, but higher variance. In a way, we are trading error penalty for stability of our classifier.

The kernel parameter is simply a similarity measure. That will be explained in detail in subchapter 3.3. "Kernel Methods".

3.1.2. LinearSVC

LinearSVC is a version of SVM with parameter kernel='linear'. It has been implemented in a different way, so that it would work faster on large data sets and have more freedom in the choice of penalties and loss functions. There are significant differences between SVC with linear kernel and LinearSVC scikit implementations, since they always present different results when trained and evaluated on the same data. In my experiments, LinearSVC models performed really well and predicted results with the highest accuracy.

3.1.3. Naive Bayes

Naive Bayes classifier, hereafter NB, is based on Bayes' Theorem. The adjective "naive" is found here because of the strong "naive" assumptions that the features are independent. NB is a conditional probability model. For each instance, it assigns probabilities that it belongs to a certain class. In Figure 3.1, we can see the formula associated with these probabilities.

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots x_n \mid y)}{P(x_1, \ldots, x_n)}$$

Figure 3.1: NB assigns "posterior" probability to each instance from the data set

With naive assumptions for feature independence and constant value in the denominator, we get the formula which NB uses to make predictions:

$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

Figure 3.2: This formula is taken from Scikit-learn (2016).

I will not further describe hyperparameteres with NB, since I did not optimize them and NB hasn't really proved effective in this project.

3.1.4. Random Forest

Random Forest classifier works by generating decision trees in the training phase, using the random selection of features. Afterwards, it predicts the class that is the mode of the classes. Decision trees are a popular ML method and are invariant to scaling and many other transformations of feature values. However, trees that are deep tend to learn highly irregular patterns and overfit the model. Random forests try to combine multiple trees and average them, so that each tree is trained on a different part of the common training data.

Random Forest Hyperparameter

In this work, I have optimized the “n_estimators” parameter. This hyperparameter defines the number of trees in a forest.

3.2. Mapping Solver Trees to Feature Vectors

Most classification algorithms require data transformed into a numeric vectorized form, representing the values of the data's features in the feature space. That allows analyzing the data in the vector space using linear algebra. Since we are mapping trees to feature vectors, a hierarchy of the data needs to be preserved in the feature space. To do that, we will need to contain more information in feature vectors.

The proposed mapping of tree structures into feature vectors is inspired by Yang et al. (2005) and based on the binary tree representation of rooted ordered labeled trees. In a binary tree, every node has at most two children. The standard algorithm to transform an ordinary tree to its corresponding binary tree is through the left-child, right-sibling representation.

Conversion procedure:
1) Create the edges between all siblings in a tree
2) Delete all the edges between each node and its children except those that connect it with its first child
3) All leaves must be epsilon-leaves

In Figure 3.3 below, there is an example of a transformation to a binary tree. We can see how all siblings are connected and all leaves are epsilons. These representations are taken from Yang et al. (2005) and modified.

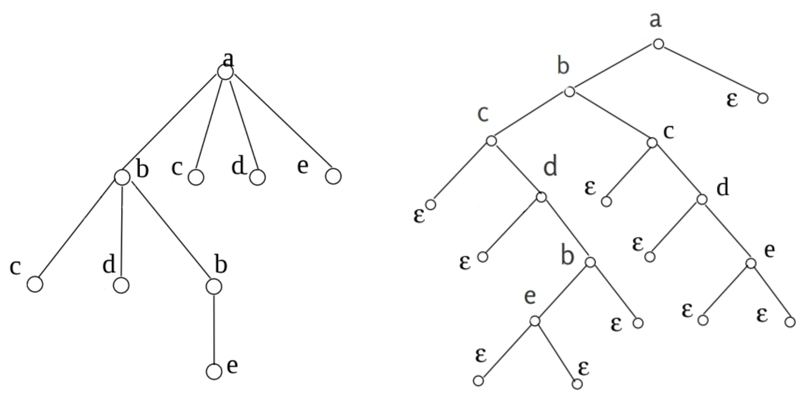
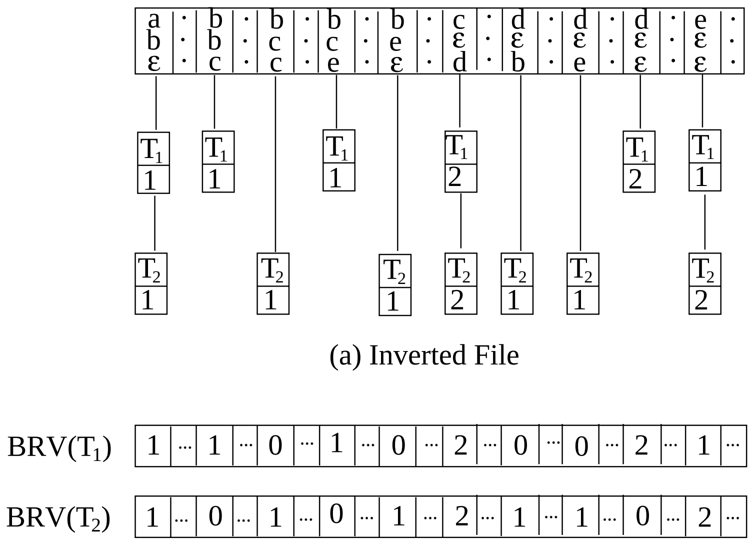


Figure 3.3: An example of ordinary rooted tree and a corresponding binary tree.

After all solver trees in the data set have been transformed to binary trees, next move is to traverse through each binary tree and build a vocabulary of binary branches. Binary branch is defined by two edges connecting the node with both of its children.

For each entry in the data we traverse again through its binary solver tree and count occurrences of each binary branch. We then represent this tree as a feature vector of same dimensions as the binary branch vocabulary. The output vector is mostly filled with zeros and other values represent the number of occurrences certain binary branches appeared in the tree.

In Figure 3.4, the dictionary of binary branches records numbers of occurences of a certain binary branch in every data instance represented by a corresponding binary tree T_i. Afterwards, these information are used to create feature vector for each instance in the data set.


BRV(T_1) 1 ··· 1 ··· 0 ··· 1 ··· 0 ··· 2 ··· 0 ··· 0 ··· 2 ··· 1 ···
BRV(T_2) 1 ··· 0 ··· 1 ··· 0 ··· 1 ··· 2 ··· 1 ··· 1 ··· 0 ··· 2 ···

(a) Inverted File

(b) Binary Branch Vectors

Figure 3.4: Creating feature vectors from dictionary of binary branches. This is taken from Yang et al. (2005).

In Figure 3.5 we can see how feature vectors generated this way do not strictly preserve information about the structure of the solver tree. This is a good fit for this thesis because most of the time small differences and changes in the order solver steps are arranged are not that significant for determining the task category correctly. We do need to roughly preserve the structure here, just not in that much detail.

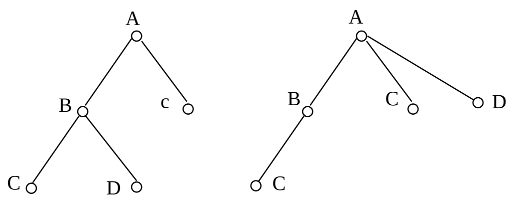


Figure 3.5: Trees with zero-distance when converted to binary trees (Yang et al., 2005)

.

Now that every instance has been mapped to a corresponding feature vector, we can choose any classifier and see how it behaves and how accurate it categorizes our tasks. The classifiers I have used are SVM, Random Forest, Linear SVM, and Naive Bayes. They have all been implemented in scikit-learn package I've been using in Python (Pedregosa et al., 2011). The results will be provided to the reader in the next chapter.

3.3. Kernel Methods

To preserve information about the structure in hierarchical data, dimensions of feature space must be increased for feature vectors to contain more information. This causes the effect known as “the curse of dimensionality”. The classification power decreases with an increase in the dimensionality of the input. With this being the main disadvantage of the first approach and considering that kernel methods do not operate directly on feature space, it is safe to say kernel methods are often preferred choice.

Kernel methods avoid transformation to vectors. They only require a kernel. A kernel is a measurement of similarity of each pair of instances in the data set. Tree kernel is defined as:

$$K : T \times T \to R$$

It operates on trees and returns a number that describes similarity between two given trees. Furthermore, the computational complexity of analyzing the feature space depends only on the complexity of the kernel function. Analyzing the feature space is independent of the feature space dimensions, so kernel methods do not suffer this curse of dimensionality. This is especially convenient for small data sets with huge feature space.

As this data set consists of 2829 finite examples, we can get a complete representation of a kernel by generating a kernel matrix of size 2829$\times$2829. The function for determining a kernel and building a kernel matrix is called the kernel function. Since the data is now represented with a kernel matrix instead of a feature matrix, all analysis will be performed over the kernel matrix. Many machine learning algorithms can be kernelized. I have focused on kernel methods with Support Vector Machine.

To kernelize SVM, it suffices to define a kernel function and pass it to the classifier. To be valid, the kernel function must satisfy Mercer’s condition. Validness of a kernel function, however, does not imply that such kernel is efficient or even effective. Instead, this means that the kernel is defined in a correct way and can be passed as an argument to the Support Vector Machine or some other kernelized classifier.

Mercer's condition only says that a kernel K is valid if:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} K(T_1, T_2)c_i c_j \geq 0$$

for all choices of real-valued coefficients c_i and c_j. This simply means that a kernel function must a positive semi-definite function. For a function to be positive semi-definite, it must fulfill two conditions:

1. $f(x) = 0$ for $x = 0$
2. $f(x) \geq 0$ for $x \in D$ and $x \neq 0$

where D is domain of the data we are working with. Therefore, our kernel matrix K needs to have all zeros on the top-left to bottom-right diagonal because the numbers on that diagonal are representing the similarity of each tree compared to itself. All other values in the matrix must be greater than or equal to zero.

I have implemented two tree kernel functions that I found appropriate for this case. One is based on converting trees into strings and it is basically a string kernel. The other one is based on comparing subpaths in trees. Both methods are explained in detail in (Causevic, 2017).

3.3.1. Tree Kernel Based on Converting Trees into Strings

The idea behind this tree kernel is to convert two trees into strings so that the strings encode the tree's structure. Afterwards, we apply string kernel to converted strings.

How Trees Are Converted to Strings

As we can see in Figure 3.6, to encode and preserve the tree's structure usually the parenthesis are used and the tree is presented in the pre-order traversal. PhotoMath saved their solver trees in this form; they were included in the data set as strings. Therefore, I found this kernel method convenient to implement.

Applying String Kernel Method

Kernel between two strings S_1 and S_2 is defined as:

$$K_{string}(S_1, S_2) = \sum_{s \in C} num_s(S_1) num_s(S_2)$$

where $num_s(S_x)$ is number of occurrences of substring s in string S_x.
Finally, C is defined as the set of all common substrings that appear in both S1 and S2.

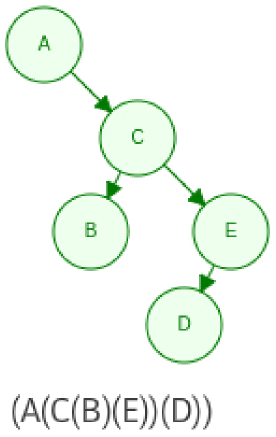


Figure 3.6: An example of encoding tree structure and content into string

Possible members of C are all solver commands from a given string. For example, if S_x is "(simplify;(simplify_wrapper(abs_real)))" then possible entries for C would be "simplify", "simplify_wrapper", and "abs_real".

Since support vector machine estimator in scikit-learn gets input matrix only as float, I input a matrix of task ids. The kernel function I've implemented gets a task solver tree from extracting information about the task from data. It then calculates a kernel by the given formula.

3.3.2. Tree Kernel Based on Subpaths

A subpath is a segment of a path from the root to one of the leaves. With this approach, we can compare trees based on subpaths of a certain length that they have in common.

Extracting Subpaths from a Tree

The first step of this approach would be to implement a function that finds the set of all subpaths in the given tree. For subpaths we are extracting we need to define a certain length l. To find this set of subpaths we first find paths from the root to each of the leaves. We then operate on these paths and take all their subsections of length l. In the Figure 3.7 we can see an example of extracted subpath sets for length l equal to two and three.

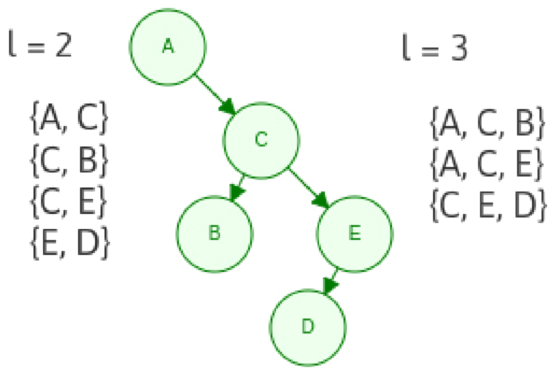


Figure 3.7: An example of extractring subpaths of length l from a tree.

Building the Kernel Matrix

Extracting subpaths of length l from both trees we are comparing preceeds applying the formula below:

$$K(T_1, T_2) = \sum_{sp \in S} num_{sp}(T_1) num_{sp}(T_2) w_s$$

where $num_{sp}(T_x)$ is the number of occurrences of subpath sp in tree T_x. A subpath set of tree T is the set of all subpaths included in the tree. In this formula S is an interception of subpath sets of trees T_1 and T_2. Therefore, S contains all common subpaths of two mentioned trees.

This kernel method operates vertically on trees, which means that we find similar only those trees that have similar vertical structures. The reason I chose this method is that it works quite differently than the first two approaches I tried. I wanted to see what makes some task belong to a certain category. I was looking for an answer to questions like – If we considered only one category, what would then be the best similarity measure between all solver trees within this category? To compare tree similarity based on their vertical structure or just mathematical operations that appear in the solution of these tasks? So I took three different methods to see which fits the problem of this thesis. The results are presented in the following chapter.

4. Results

Before interpreting final results, I will shortly introduce each reported metric to explain the intuition behind them.

Accuracy

Percentage of all predictions that are correct. Accuracy shows us how good a model is.

$$accuracy = \frac{\text{correct predictions}}{\text{all predictions}}$$

Precision

Percentage of all positive predictions that are correct or indeed positive.

$$precision = \frac{\text{positive predicted correctly}}{\text{all positive predictions}}$$

Recall

“Recall is intuitively the ability of the classifier to find all the positive samples.” (Albon, 2016). Recall alone is not enough as it is trivial to achieve recall of score 1 (ideal value).

$$recall = \frac{\text{predicted to be positive}}{\text{all positive instances}}$$

F1 score

The F1 score is defined as the harmonic mean of the precision and recall. In the multinomial classification this is the weighted average of the F1 score of each class.

$$\mathit{F1} = 2\frac{precision * recall}{precision + recall}$$

The motivation for using F1 score hides in the fact that it shows us some intuition on how precision and recall are balanced and how accurate our system really is. F1 score reaches its best score at 1 and worst score at 0.

All models shown below are optimized by cross-validated grid search over some or all of their parameters. The data set has been split randomly into train and test part. All models were trained on 75 percent of the data set and tested on 25. In cross-validation k number of folds was set to five. Table 4.1 shows us the parameters that the grid search was performed on. Only optimized values of corresponding parameters are shown for each classifier.

Classifier	**Parameters**
SVM	C: 1, kernel: linear
LinearSVC	C: 0.1
Random Forest	n_estimators: 35
Kernelized SVM	C: 1

Table 4.1: Optimized classifiers with corresponding parameters

4.1. Mapping trees to feature vectors

In Table 4.2 we can see that the classifier predicted labels of level 1 really well. That is expected though, because there are only 8 categories of depth less or equal to one. And we have 2829 instances.

Level 1			
Classifier	**Precision**	**Recall**	**F1 score**
SVC	0.930	0.924	0.924
LinearSVC	0.928	0.922	0.922
Random Forrest	0.921	0.917	0.917

Table 4.2: Results for each classifier, level 1

On label precision of level 2, there are 23 labels. In Table 4.3, we can see that linear SVC classifier had the best performance. Its parameter C has been optimized and its value is set to 1.

Level 2			
Classifier	**Precision**	**Recall**	**F1 score**
SVC	0.894	0.880	0.881
LinearSVC	0.904	0.896	0.896
Random Forrest	0.875	0.860	0.862

Table 4.3: Results for each classifier, level 2

In Table 4.4 are shown full results for each class showing the model performance with LinearSVC. The “support” column represents the number of instances in the test set that belonged to a certain mathematical category.

Level 2				
Label	**Precision**	**Recall**	**F1 score**	**support**
A.1	0.856	0.856	0.856	90
A.12	1.000	0.900	0.947	20
A.13	1.000	1.000	1.000	2
A.2	0.902	0.949	0.925	39
A.4	0.857	0.750	0.800	16
A.5	0.857	1.000	0.923	6
A.8	0.900	1.000	0.947	9
B.1	0.943	0.971	0.957	34
B.2	0.800	1.000	0.889	8
B.3	0.955	0.875	0.913	24
B.4	0.000	0.000	0.000	0
B.5	0.864	0.655	0.745	29
B.6	0.968	0.833	0.896	36
B.7	1.000	0.846	0.917	13
C.1	0.933	0.933	0.933	30
C.2	1.000	0.953	0.976	43
F.2	1.000	1.000	1.000	18
F.5	0.909	0.952	0.930	21
F.8	1.000	0.750	0.857	4
F.9	0.750	0.857	0.800	7
H.1	0.917	0.647	0.759	17
H.3	0.767	0.972	0.857	71
H.4	1.000	0.966	0.982	29
avg / total	**0.904**	**0.896**	**0.896**	**566**

Table 4.4: Full results for best classifier, LinearSVC

Some classes like A.13 and F.2 have all three metrics equal to one. That might be the result of obvious and routinary solver trees. For example, in the data set there is a lot of tasks that are labeled as some form of mathematical equations – category A. Their solver steps look really similar as solving an equation is always a routine procedure.

In Tables 4.5 and 4.6, we can see how scores are getting smaller while the number of possible categories increases. On $level = 3$ and $level = 4$ the LinearSVC has still proven to be the best classifier. At the maximum category depth of 4, the problem with sparsity reappears.

Level 3			
Classifier	**Precision**	**Recall**	**F1 score**
SVC	0.751	0.705	0.702
LinearSVC	0.783	0.751	0.748
Random Forrest	0.743	0.705	0.699

Table 4.5: Results for each classifier, level 3

Level 4			
Classifier	**Precision**	**Recall**	**F1 score**
SVC	0.635	0.613	0.590
LinearSVC	0.660	0.634	0.607
Random Forrest	0.620	0.618	0.583

Table 4.6: Results for each classifier, level 4

4.2. Tree Kernel Method Based on String Similarity

In Table 4.7 we can see results of tree kernel method that measures the similarity between the trees by looking at solver commands these trees have in common.

Level	**Precision**	**Recall**	**F1 score**
2	0.923	0.917	0.916
3	0.817	0.790	0.787
4	0.659	0.610	0.596

Table 4.7: Results per levels

We can see that this seems to overcome the first approach with mapping trees to feature vectors. This model achieves best results on all levels.

Level 2				
Label	**Precision**	**Recall**	**F1 score**	**support**
A.1	0.952	0.940	0.946	84
A.12	1.000	1.000	1.000	23
A.2	0.943	0.943	0.943	35
A.4	0.765	0.765	0.765	17
A.5	0.833	1.000	0.909	10
A.8	0.938	1.000	0.968	15
B.1	0.914	0.970	0.941	33
B.2	1.000	1.000	1.000	10
B.3	1.000	0.963	0.981	27
B.4	0.500	1.000	0.667	1
B.5	0.850	0.630	0.723	27
B.6	0.941	0.865	0.901	37
B.7	1.000	0.778	0.875	9
C.1	1.000	1.000	1.000	28
C.2	1.000	0.974	0.987	38
F.2	1.000	1.000	1.000	19
F.5	0.824	0.875	0.848	16
F.8	0.800	1.000	0.889	4
F.9	0.714	0.625	0.667	8
H.1	1.000	0.652	0.789	23
H.3	0.787	0.959	0.864	73
H.4	1.000	1.000	1.000	29
avg / total	**0.923**	**0.917**	**0.916**	**566**

Table 4.8: Full results for level 2, string kernels

Again, we can notice in Table 4.8 how some classes perform unexpectedly well, although there were almost 30 instances in the test set that were labeled with this category. To be sure what I said was true, I checked to which areas of mathematics these labels belong to. Above we can see categories C.1 and H.4, Trigonometric equations and Derivatives, accomplishing ideal values in all three metrics.

4.3. Tree Kernel Method Based on Subpaths

This last approach seems to be the least accurate one. Comparing the subpaths of length n to measure the similarity of trees proved to be inefficient in this case. Length n was optimized, but the results remained low as seen in Table 4.9.

Level	Precision	Recall	F1 score
3	0.531	0.488	0.470

Table 4.9: Results for subpaths kernel method, level 3

4.4. Shortly on Results

I am generally satisfied with the results achieved in this thesis. I must also be honest and say that I already see some potential for future improvement, especially in case task describitions would be provided. The first approach which consisted of mapping solver trees to feature vectors surprised me with such high accuracy due to its main disadvantage – direct transformation of data to feature space. Implementing all three methods helped me in figuring out what really matters and what should be the measure of similarity between solver trees. Although the string kernel method worked best, it is not safe to say that it's the most successful one. The data set was quite sparse and the structure should, however, still be taken into account. The achievement of the first approach indicated how the structure should be preserved along with the content. It also pointed out how the optimal way to encode structure into feature vectors should not be in a robust, strict way. It should be somehow flexible. Altogether, I am content with the result that were presented in this chapter and I hope I will improve them in the future, perhaps on a bigger data set.

5. Conclusion

The goal of this thesis was to build a system that can classify software-solvable mathematical tasks into predefined mathematical categories. The solution to this problem has been accomplished and the results achieved in this project are satisfactory.

The first approach included mapping solver trees to vectors in the feature space. This is also how we usually behave when data is flat. However, we had to preserve information about the structure of solver trees and were therefore obligated to increase the dimensions of the feature space. By increasing the dimensions of the feature space, our representation of the data set in the feature space became more sparse and influenced our predictions negatively. Although this approach may not be optimal, the results that were achieved seem to be quite successful. The cause of that might be that our transformation to feature space encoded the tree structure into feature vectors to the right extent. The structure was encoded as a vocabulary of binary branches. However, it could easily be the case that original structure has been a bit misinterpreted after conversion from the ordinary to the binary tree. That allowed us to encode tree structure into feature vectors more freely.

In contrast, the second approach avoids operating on feature space directly and uses kernel methods. We have implemented two kernel functions and passed their reference to Support Vector Machine. The first kernel function calculated a kernel based on the string similarities between trees encoded as strings. This method proved even more successful than the first approach with mapping solver trees to the feature space. Later on, I implemented a kernel function that generates a kernel matrix based on common subpaths in solver trees. Despite my expectations, this proved to be much less efficient.

Future Work

Despite the problems with data sparsity, some future work could be done on predicting the categories on all depths of the taxonomy. The classifier could start predicting at the maximum depth and label tasks if the level of confidence is high enough. It could then progress towards more general categories and label them the same way – if it is confident enough to do so. If not, it should leave it to the next iteration with more general categories. The data set should also contain more tasks from currently unrepresented labels. Especially since the taxonomy is still not complete. Various kernel methods could still be implemented. Likewise, it would be interesting to try and combine unsupervised with supervised learning approach and try to classify tasks based on the PhotoMath prefix commands which remained untouched in the current data set. Assuming that raw tasks in string form could be made available to us and with application of NLP algorithms, I believe it would work at least for more general categories. Nonetheless, it would undoubtedly be interesting to work on that.

BIBLIOGRAPHY

Chris Albon. Precision, recall and f1 scores, 2016. URL https://chrisalbon.com/machine-learning/precision_recall_and_F1_scores.html.

Dino Causevic. Tree kernels: Quantifying similarity among tree-structured data, 2017. URL https://www.toptal.com/machine-learning/structured-data-tree-kernels.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

Scikit-learn. Naive bayes, 2016. URL http://scikit-learn.org/stable/modules/naive_bayes.html.

Wikipedia. Multiclass classification — Wikipedia, the free encyclopedia, 2017. URL https://en.wikipedia.org/wiki/Multiclass_classification#cite_ref-3.

Rui Yang, Panos Kalnis, i Anthony K. H. Tung. Similarity evaluation on tree-structured data. U *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’05, stranice 754–765, New York, NY, USA, 2005. ACM. ISBN 1-59593-060-4. doi: 10.1145/1066157.1066243. URL http://doi.acm.org/10.1145/1066157.1066243.

Računalna analiza sličnosti matematičkih zadataka zadanih riječima

Sažetak

Cilj ovog rada bio je napraviti sustav za kategorizaciju matematičkih zadataka. Tvrtka PhotoMath prije nekoliko je godina uspješno izašla na tržiste sa istoimenom aplikacijom koja omogućuje korisnicima da mobitelom skeniraju zadatak te u kratkom vremenu dobiju detaljan postupak rješavanja. PhotoMath mi je dao priliku da riješim ovaj problem i osigurao vlastiti skup podataka za ovaj rad. Svaki matematički zadatak predstavljen je stablom naredbi odnosno postupaka koji čine rješenje zadatka. Svakom zadatku cilj je pridijeliti matematičku kategoriju. Sve su kategorije ranije definirane taksonomijom te svaka kategorija predstavlja neko matematičko područje. Implementirana su dva različita pristupa. Prvi je pristup obuhvaćao pretvorbu stabla postupaka u pripadajuće vektore značajki. Vektori su bili generirani tako da sadržavaju i informacije o strukturi podataka. Nad vektorima značajki zatim se radila klasifikacija podataka. Rezultati su se pokazali poprilično uspješnima. Drugi pristup sastojao se od izgradnje matrice međusobne sličnosti svih podataka u skupu. Isprobane su dvije metode mjerenja sličnosti između stabla. Najbolje rezultate postigla je mjera sličnosti koja ovisi o broju zajedničkih naredbi saržanih u oba stabla. Druga metoda pokazala se manje učinkovitom.

Ključne riječi: višeklasna klasifikacija, sličnost kod stabla, kernel metode, hijerarhijska klasifikacija, kategorizacija matematičkih zadataka, strojno učenje

Computational Analysis of Math Word Problems

Abstract

The task of this thesis was to build a system that can classify computer-solvable mathematical tasks into predefined mathematical categories. A few years ago, a company named PhotoMath released its homonymous app that is able to output solution steps to the user after recognizing the mathematical task scanned with a mobile phone. They provided me with the data set and the idea for this project. Each mathematical category represents some area of mathematics. The categories are hierarchically related. Each task is represented by a corresponding solver tree. Solver tree is a tree that consists of steps which explain task resolution to the user. There were two main ideas to implement. The first approach consisted of mapping solver trees from each task in the data set to a vector in the feature space. Although this approach may not be optimal, the results are quite successful. The cause of that might be that our transformation to feature space encoded the tree structure into feature vectors to the right extent. In contrast, the second approach avoids operating on feature space directly and uses kernel methods. I have implemented two kernel functions and passed their reference to Support Vector Machine. The first kernel function calculated a kernel based on the string similarities between trees encoded as strings. This method proved even more successful than the first approach with mapping solver trees to the feature space. Later on, I implemented a kernel function that generates a kernel matrix based on common subpaths in solver trees. Despite my expectations, this proved to be much less efficient.

Keywords: hierarchical multinomial classification, hierarchical data, tree similarity, mathematical categories, task categorization, classification, kernel, machine learning.