ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/349382985

Software Forest: A Visualization of Semantic Similarities in Source Code using a Tree Metaphor

Conference Paper · February 2021

DOI: 10.5220/0010267601120122

CITATIONS
0

READS
195

7 authors, including:

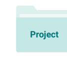
Maximilian Söchting
Hasso Plattner Institute
10 PUBLICATIONS **26** CITATIONS
SEE PROFILE

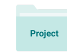
Willy Scheibel
Hasso Plattner Institute
29 PUBLICATIONS **68** CITATIONS
SEE PROFILE

Daniel Limberger
Hasso Plattner Institute
37 PUBLICATIONS **139** CITATIONS
SEE PROFILE

Jürgen Döllner
Hasso Plattner Institute
431 PUBLICATIONS **4,620** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:


Information Cartography View project


Mobility and Reachability Analytics View project

All content following this page was uploaded by Willy Scheibel on 09 March 2021.

The user has requested enhancement of the downloaded file.

Software Forest: A Visualization of Semantic Similarities in Source Code using a Tree Metaphor

Authors Version

Daniel Atzberger*, Tim Cech, Merlin de la Haye, Maximilian Söchting, Willy Scheibel*, Daniel Limberger*, and Jürgen Döllner*

This article is originally published as

D. Atzberger, T. Cech, M. de la Haye, M. Söchting, W. Scheibel, D. Limberger, and J. Döllner. Software forest: A visualization of semantic similarities in source code using a tree metaphor. In *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications – Volume 3: IVAPP*, IVAPP '21, pages 112–122. INSTICC, SciTePress, 2021. ISBN 978-989-758-488-6. doi:10.5220/0010267601120122

Bibtex entry:

```
@InProceedings{atzberger2021-softwareforest,
  author       = {Atzberger, Daniel and Cech, Tim and de la Haye, Merlin and S{\"o}chting, Maximilian and Scheibel, Willy and Limberger,
       Daniel and D{\"o}llner, J{\"u}rgen},
  title        = {Software Forest: A Visualization of Semantic Similarities in Source Code using a Tree Metaphor},
  booktitle    = {Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and
       Applications -- Volume 3: IVAPP},
  year         = {2021},
  series       = {IVAPP '21},
  publisher    = {SciTePress},
  organization = {INSTICC},
  doi          = {10.5220/0010267601120122},
  isbn         = {978-989-758-488-6},
  pages        = {112--122},
}
```


*The author is with the Hasso Plattner Institute, Digital Engineering Faculty, University of Potsdam

Software Forest: A Visualization of Semantic Similarities in Source Code using a Tree Metaphor

Daniel Atzberger[1], Tim Cech[2], Merlin de la Haye[2], Maximilian Söchting[2],
Willy Scheibel[1], Daniel Limberger[1], and Jürgen Döllner[1]
Hasso Plattner Institute, Digital Engineering Faculty, University of Potsdam, Germany

Keywords: Topic Modeling, Software Visualization, Source Code Mining

Abstract: Software visualization techniques provide effective means for program comprehension tasks as they allow developers to interactively explore large code bases. A frequently encountered task during software development is the detection of source code files of similar semantic. To assist this task we present *Software Forest*, a novel 2.5D software visualization that enables interactive exploration of semantic similarities within a software system, illustrated as a forest. The underlying layout results from the analysis of the vocabulary of the software documents using Latent Dirichlet Allocation and Multidimensional Scaling and therefore reflects the semantic similarity between source code files. By mapping properties of a software entity, e.g., size metrics or trend data, to visual variables encoded by various, figurative tree meshes, aspects of a software system can be displayed. This concept is complemented with implementation details as well as a discussion on applications.

1 INTRODUCTION

Program comprehension is of fundamental importance during the entire software development process, especially in the maintenance phase. Getting a quick overview over a program's structure and functionalities is necessary for further development. A question developers frequently encounter is whether there are already source code units of certain semantics in terms of features, concerns, and techniques. The identification of such code units can support decisions on feature location, encapsulation, and module architecture. Furthermore, this information helps to identify associated developers and could be used to, e.g., determine and assign reviewers. This is a particular challenge for historically grown legacy systems, systems developed from distributed teams, or systems depending on a large technology stack. In addition, the folder structure of a software project may deviate from its conceptual content over time. Various software visualization techniques support developers by providing expressive images whose layouts are inspired by maps known from daily life. They promote shared mental images that enable users to communicate more precisely and gain insight into abstract data.

We present *Software Forest*, a novel 2.5D software visualization technique for displaying semantic similarity of source code units. The semantic content of software files is described as distributions over latent topics hidden in the source code by applying Latent Dirichlet Allocation (LDA). Multidimensional Scaling (MDS) is used to compute a layout, where the distance of points reflect their semantic similarity. Using a tree metaphor, *Software Forest* introduces various visual variables for topic emphasis and additional data display, e.g., software metrics, and offers possibilities for displaying static as well as evolutionary data (Figure 1). The following contributions are made:

Figure 1: A software forest depicting 11 291 source code units (top). Each unit is positioned based on its semantic similarity w.r.t. to all other units. The unit's dominant topic is represented by the type of the tree (right, e.g., palm tree, deciduous tree, conifer) and additional attributes are be mapped to, e.g., tree health and tree growth (left).

1. We present a layout algorithm based on the application of LDA and MDS on source code files,

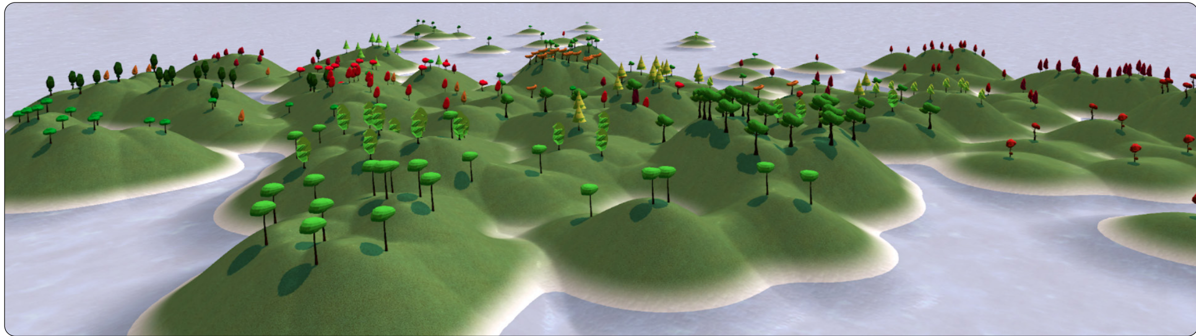


Figure 2: Visualization of the Microsoft Calculator open-source project in our framework. Tree types are chosen based on the topics, i.e., source code files with the same dominant topics are represented through the same tree type. Clusters of identical tree types, i.e., source code files with identical dominant topics, can be easily recognized in this example.

thereby mirroring the semantic relatedness of documents such as source code units.

2. We enlist and review visual variables inherent to the tree metaphor and discuss possibilities and applications for software visualization tasks.
3. We provide technical notes on the implementation of our visualization, discuss its features and limitations, and provide a web-based prototype for interactive exploration and evaluation.

The remainder of this paper is organized as follows. Section 2 discusses background and related work. In Section 3, we detail our layout algorithm. Section 4 presents our visualization concept using visual variables of trees (figuratively). Section 5 highlights implementation aspects regarding the data processing pipeline and the visualization framework. Section 6 discusses application scenarios and limitations of the current approach. Section 7 concludes this paper and outlines direction for future work.

2 RELATED WORK

Related work is mainly identified in two research areas: (1) preprocessing of software system data by means of topic extraction and dimension reduction and (2) thematic maps, map-based visualizations, and visualization metaphors in general.

Data Processing. Since a large part of a developer's knowledge of the project is directly encoded in the source code through comments and identifier names, a layout origin from the vocabulary could reflect the project's semantic structure. This approach was pursued for the first time by Kuhn et al. (Kuhn et al., 2008). They examine the natural language found in source code to generate software visualization. In a first stage source files are parsed and a term-document-matrix is created, which stores the frequency of each word in a document. The high-dimensional document-word-vectors, i.e., the columns of the term-document-matrix, are projected to the plane by applying Latent Semantic Indexing (LSI) (Deerwester et al., 1990) followed by MDS (Cox and Cox, 2008). By creating a height field which represents the LOC, island-like landscapes are created which allow semantic affiliations to be deduced. Another visualization of natural language texts, which utilizes a cartographic metaphor, can be found in (Skupin, 2004). The layout, which is created by using a self-organizing map (Kohonen, 1997) on the term-document-matrix, is enriched with a height field. In their investigations the authors limit themselves to abstracts from publications of the geographic domain. For the visualization of source code this can be extended by suitable preprocessing.

The main difference between our work and the previously presented is that our layout is based on a probabilistic topic model, LDA specifically. It is known that the pure Bag-of-Words (BOW) model is not able to recognize synonymy and polysemy. Although LSI can recognize synonyms, polysemy is still an issue. Probabilistic models, e.g., LDA can help to overcome both problems. Besides the layout, we can also map properties of the topic distribution to visual variables of the trees (Figure 2). Since LDA has been used for numerous software mining tasks, it is not surprising that the obtained results were represented by dimension reductions. For instance, Linstead et al. used LDA and the author-topic model (Rosen-Zvi et al., 2004) to detect semantic similarity between documents and developers (Linstead et al., 2009). The resulting document-topic distributions are displayed by applying MDS to reduce them to two dimensions. Nevertheless, this neglects the fact that the similarity between topics among each other might differ. We circumvent this issue by first applying a dimension re-

duction to the topic distributions and then aggregating the document vectors as linear combinations.

Another problem which adresses the question of positionining elements in a way that reflects their semantic similarity was presented in (Barth et al., 2014). Barth et al. presented three novel layout algorithms for word clouds, where the distance between the words should capture their semantic relatedness. Our considerations neglect the similarity of words and address the representation of semantic similarity of documents.

Visualization Metaphors. A majority of the underlying layouts of software visualizations are based on the hierarchical folder structure (Holten et al., 2005; Cornelissen et al., 2008). Treemaps are a widely used technique in hierarchy visualization (Scheibel et al., 2020) and are prominently used for software visualization as they provide a wide variety of visual variables (Limberger et al., 2019).

Besides classic visual variables like block height, texture or color a 2.5D treemap offers creative ways for visualizing dynamic properties of a software system. An example is presented by Würfel et al., who showed how natural phenomena such as fire and rain can be used to visualize the evolution of source code artifacts (Würfel et al., 2015). In this context, *CodeCity* (Wettel and Lanza, 2007) and the *Software City* (Steinbrückner and Lewerentz, 2013) are well known metaphors. By mapping source code metrics, e.g., Lines-of-Code (LOC), to visual variables of cuboids, e.g., the height, visualizations are created, which remind optically of modern cities. The approach was further developed in later work and integrated into IDEs (Balogh et al., 2015). Next to rectangular layouts, map-like shapes are used in hierarchy visualization as well (Auber et al., 2013). With focus on software visualization, the map metaphor is extended to derive layouts from dependencies (Hawes et al., 2015). The map metaphor is further used in conjunction with oceans and islands that are derived from Voronoi diagrams (Schreiber and Misiak, 2018). Another approach for visualizing software metrics inspired by nature are *Software Feathers*, a visualization approach for object-oriented entities like classes and interfaces with feathers (Beck, 2014). The structure of feathers is determined by numerous parameters, e.g., their size, shape, or texture, which provides many possible mappings of software metrics. Similar to our forest and landscape visualization, Štěpánek proposes a visualization primarily composed of trees, wood cabins and islands (Štěpánek, 2020). The layout of source code units is calculated based on a weighted graph, with files that reference each other having a stronger pull toward one another. Different file types are represented with different metaphors, e.g., classes are represented with trees while value types (structs) are visualized using small cabins.

Kleiberg et al. proposed another visualization which mainly bases on the tree metaphor (Kleiberg et al., 2001). The authors generate a botanic tree whose branches and fruits display hierarchical structures. Our work completely neglects the underlying hierarchical structure of a software system provided by its folder structure. Additionaly a single tree is used to display a single file rather than an entire project.

3 LAYOUT COMPUTATION

The underlying layout of a software forest is constructed to reflect the semantic similarity of the source code units, in our case files contained in a source code repository. Thus, we analyze the use of natural language in the source code. We first preprocess the corpus $C = \{d_1, \ldots, d_m\}$ of software files $d_1, \ldots, d_m$ to get rid of the words in the vocabulary that do not carry a semantic meaning. We then compute a formal description of the documents as distributions over latent topics using LDA. The resulting high-dimensional vectors describing topics are reduced by MDS to two components, which finally aggregate to the positions of document vectors.

3.1 Preprocessing Data

Although source code resembles natural language in its vocabulary, various preprocessing steps are required to get rid of words that do not carry relevant semantic meaning. For this, we implement an algorithm, processing all documents as follows:

1. **Removal of Non-text Symbols:** All special characters such as dots and semicolons are replaced with white spaces to avoid accidental connection of words not meant to be combined. This includes the splitting of identifier names, e.g., the word *foo.bar* gets split into *foo* and *bar*.
2. **Split of Words:** Identifiers are split according to delimiters and the camel case convention, e.g., *FooBar* is split into *foo* and *bar*, and stripped from redundant white space subsequently.
3. **Removal of Stop Words:** Stop words based on natural language and keywords of programming language are removed as they carry no semantic content. Additionally, we filter the input based on a hand-crafted list comprising domain specific stop words, data types, type abstractions etc.
4. **Lemmatization:** To avoid grammatical diversions, all words are reduced to their basic form, e.g., *said*

Topic #1		Topic #2		Topic #3	
Term	**Probability**	**Term**	**Probability**	**Term**	**Probability**
std	0.070	thread	0.132	address	0.115
transaction	0.031	time	0.070	model	0.108
fee	0.027	queue	0.064	table	0.065
tx	0.026	std	0.054	label	0.051
ban	0.024	callback	0.040	qt	0.033
str	0.023	run	0.037	index	0.030
handler	0.016	call	0.025	dialog	0.024
output	0.016	mutex	0.021	column	0.024
bitcoin	0.015	scheduler	0.020	ui	0.021
reason	0.015	wait	0.018	role	0.019

Table 1: Three exemplary topics extracted from *Bitcoin Core* (github.com/bitcoin/bitcoin) source code with $K = 50$.

and *saying* are reduced to *say*.

After preprocessing, we store all documents as BOW, i.e., the order in which the words occur is neglected and only their frequency is stored. We are not introducing new identifiers for this notation and, when using the term *documents*, always refer to their representation as BOW. The term-document-matrix stores these BOW vectors as its columns.

3.2 Latent Dirichlet Allocation

Given a corpus $\mathcal{C} = \{d_1, \ldots, d_m\}$ of documents that share a vocabulary $\mathcal{V} = \{w_1, \ldots, w_N\}$, a probabilistic topic model extracts semantic clusters in its vocabulary, so-called *topics*, as multinomial distributions over the vocabulary $\mathcal{V}$. Additionally each document is described as a mixture of the extracted topics, which describe its document's semantic structure. Given a number K of topics specified as a hyperparameter together with Dirichlet priors $\alpha = (\alpha_1, \ldots, \alpha_K)$, where $\alpha_i > 0$ for all $1 \leq i \leq K$ and $\beta = (\beta_1, \ldots, \beta_N)$, where $\beta_j > 0$ for all $1 \leq j \leq N$ that capture the document-topic-distributions and the topic-term-distribution respectively, LDA defines a fully generative model for the creation of a corpus (Blei et al., 2003). The generation process for a document d operates as follows:

1. choose a distribution over topics $\theta \sim \text{Dirichlet}(\alpha)$
2. for each word w in d
   (a) choose a topic $z \sim \text{Multinomial}(\theta)$
   (b) choose the word w according to the probability $p(w|z, \beta)$

Table 1 exemplarily shows the top 10 words of 3 manually selected topics applied on the corpus derived from the source code of the *Bitcoin Core*. These often suggest the concept underlying the topic, e.g., the third topic covers user interaction. The number of topics was chosen such that the extracted topics were easily interpretable by humans.

3.3 Dimension Reduction

Given the extracted topics $\phi_1, \ldots, \phi_K$ and the document-topic distributions $\theta_1, \ldots, \theta_m$, we first project the topic vectors to two dimensions. We adapt the approach by Sievert et al. (Sievert and Shirley, 2014), which is given by applying MDS to the disimilarity matrix derived from the Jensen-Shannon divergence. MDS yields projections $\bar{\phi}_1, \ldots, \bar{\phi}_K$, such that the euclidean distance between projected points reflects the dissimilarity between the high-dimensional points. A document d_i, $1 \leq i \leq m$ viewed as K-length vector $\theta_i = (\theta_i^{(1)}, \ldots, \theta_i^{(K)})$, is then represented by the

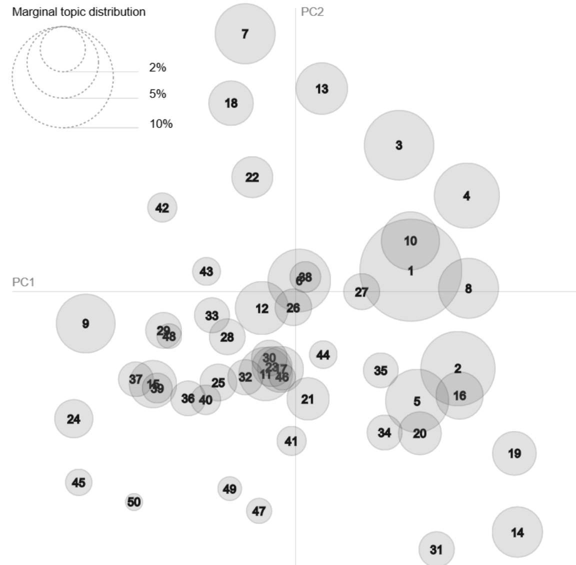


Figure 3: LDAvis applied on *Bitcoin* with $K = 50$. The centers of the circles indicate the position of the topic-word distributions after being reduced with MDS, the radii display the relative importance of the topic for the entire corpus.

convex linear combination $\bar{d}_i$, precisely

$$\bar{d}_i = \sum_{j=1}^{K} \theta_i^{(j)} \bar{\phi}_j. \tag{1}$$

A direct application of MDS to the document-topic vector as done in (Linstead et al., 2009) neglects the similarity of topics to each other as shown in Figure 3.

4 VISUALIZATION APPROACH

Since software has no intrinsic gestalt, our visualization must project the properties of the processed topic model data onto graphical primitives. This can be done by utilizing its visual variables and either providing an explanation, how the mapping of information to the variables is done, or by making use of metaphors that are known to the users (or both).

Tree Metaphor. Our visualization applies a tree metaphor: each source code unit is represented by an individual tree. All source code units together make up the software project and are depicted as a forest. The trees provide different visual variables that can be utilized, such as their type and height, the size of their trunk and crown, or the amount and color of their leaves. The position and displayed proximity of two trees allows drawing conclusions about topic similarity easily. In addition, we can rely on easy-to-understand concepts such as health, age, or season of trees in order to convey common software metrics such as test coverage, change frequency, or size. Anomalies in the source code files, such as many occurring errors or rapidly changing topics, can be displayed by tree stumps, or fallen or burning trees.

In our prototype, we decided to allow users to choose their own mappings by listing available visual variables and properties of the used source code entities. As an example, users can visualize the dominant topic with the tree type, as can be seen in Figure 2, and thus recognize which files contain similar topics. Users can explore the 3D forest by rotating, pan, and zoom using the virtual camera by means of a world-in-hand navigation. To reduce rendering cost and maintain visual clarity we opted for simpler, low-poly tree geometries instead of realistic tree rendering—we integrated and experimented with a technique for large scale forest rendering first (Bruneton and Neyret, 2012) but found it difficult to increase the visual expressiveness of individual trees. The resulting visualization can be categorized as $\mathcal{A}^3 \oplus \mathcal{R}^2$, i.e., three-dimensional graphical primitives on a two-dimensional reference space (Dübel et al., 2014).

Figure 4: Visualization of the *notepad-plus-plus*[1] project. The dominant topic of each source code file is visualized with the tree type. The cursor hovers over a light green tree and identical trees, i.e., similar source code files, are highlighted. For the selected tree, information on the corresponding source code file, such as file path, lines of code and its dominant topics, is shown above. In this example, the pre-computed topics have been labeled manually to provide comprehensible topic descriptions.

The visualization is also intended to support software systems of more than 10 000 source code units such as *TensorFlow*. This raises the issue that the visualization of so many files and thus so many trees can get confusing. To alleviate this issue, our visualization emphasizes dense tree clusters by making use of the terrain's height. Each tree is assigned to a terrain height depending on how many trees can be found in its surroundings and trees in denser clusters then can be found on hills. To be able to differentiate various clusters, a water surface with adjustable height can be used to separate different hills from one another. Furthermore, users can receive information about each tree via hovering over it. Depending on the zoom level of the visualization, a tooltip is shown for the hovered tree, containing the name and details of the represented software code file or project, such as software metrics or information about its topic distribution (Figure 4). Trees with the same identical dominant topic are highlighted to emphasize topic clusters.

Tree Geometry. We use handcrafted geometry from Sketchfab, rather than designing our own models or generating geometry dynamically (Figure 5). While the latter approach could introduce numerous variations, handcrafted geometry keeps the visualization framework easily customizable and reduces the overall complexity. Generating 3D trees at runtime would allow data to be displayed in an arbitrary exact fashion. However, rendering highly individual meshes increases the complexity for and requirements on web-based implementations. In contrast, handcrafted mod-

Figure 5: Examples of two handcrafted 3D tree model sets: top row shows a pine tree, modeled and hand painted in various stages, sizes, and health conditions. The bottom row shows low-poly meshes of various tree types in color variation. The models were purchased on Sketchfab: "HandPainted Pine Pack" by ZugZug and "Low Poly Nature Pack" by NONE.

els tend to increase the visual quality more and can easily be instanced and rendered.

The model files are expected to contain multiple variations of trees using scene-graph semantics. Furthermore, for every tree, multiple variation of the tree must be associated to a specific combination and range of visual variables and data respectively, e.g., tree growth requires information on which variations represent correspondingly large or small grown trees.

Each model file is stored with a description file that captures each tree variant and its attributes used in the visualization. Although automatic creation of this description file is imaginable, e.g., through use of heuristics or rule-based approaches, we manually created our description files. In detail, each description file contains the following information: Each file contains a list of tree types and variants along with an overview of meta-information about the corresponding model. The overview stores a reference to the model file it describes, a list of the identifiers used for all visual variables labeled and information on how to scale the whole scene to create a consistent visualization. By storing an explicit reference to the model file, multiple descriptions can be used for a single asset file, e.g., when asset files contain multiple, different 3D models.

The list of distinct variants follows a two-level structure: On the top level, models are grouped by their tree type, as this property separates fundamentally different trees. Within each tree type, a generic representative is named and a list of tree variants is recorded, e.g., trees of type "birch" with different growth states. Every tree variant assigns concrete manifestations in form of numeric values to each of its visual variables.

[1]https://github.com/notepad-plus-plus/notepad-plus-plus

Models in the description file are referenced by their scene identifier from the model file scene graph. With the description and the model file, data points in the framework can be visualized using its models and all visual variables of the file get shown to the user for attribute mapping. Users can freely assign data attributes to tree variables, e.g., by assigning the age of a software project to the growth state of the trees. As each set of models provides only a finite set of tree variants, not every value combination can be expressed directly, especially if multiple visual variables are used. We employed a model selection algorithm to choose the most suited tree model for each data point. To facilitate this, the algorithm rates each tree model based on the difference between encoded values of each of the used variables and their attributes.

Data Input. The data processing on source code repositories can take anything between hours and days, thus we decoupled data processing and visualization by storing the results in a structured, intermediate data format with the following constraints:

- Each data point will be represented by one tree.
- Each data point has attributes that represent a 2D position to place the tree in the terrain.
- Each data point might have additional attributes, e.g., dominant topics, file name or metrics.

This separation allows for the visualization, to be used with data from topic models as well as any input that complies to these constraints, e.g., data from other domains such as geo-referenced data.

5 IMPLEMENTATION

The implementation is separated into two distinct component: data processing and visualization. The interface between both components are structured files where one file represents one data set. Thereby, the structured data set uses the common CSV file format. The data processing pipeline and the visualization approach are implemented as follows.

5.1 Data Processing

We use the Python package environment and the *GitHub API for Python* for gathering our corpus. For preprocessing, we use the *nltk*[2] and *spacy*[3] library for lemmatization and obtaining a stop word list for the English language. One of the bottle necks of our data pipeline is the main memory. Large projects like *TensorFlow* may exhaust the main memory especially during lemmatization. Therefore, we chose to process projects in batches and process them sequentially, optimizing the preprocessing steps for this kind of distributed computing. The results of this step were cached and saved in text files that can be loaded during runtime to avoid redoing this step for a project.

After finishing preprocessing, we used the *gensim*[4] implementation for LDA. It provides a convenient optimization pipeline with random search as well as dealing properly with relatively large amounts of data. For optimization and the dimension reduction algorithms, we used the *scikit-learn* library[5].

5.2 Visualization

The visualization component is implemented as a web-based visualization system that processes the CSV files. Thereby, the component is implemented using TypeScript and WebGL to be used in modern web browsers. As the browser integration and rendering is based in the open-source framework *webgl-operate*[6]. It provides some abstraction in accessing WebGL rendering functions, including some very high-level functionality such as a ready-to-use label rendering pass or glTF[7] scene loading functionality. On top of this, the terrain and tree rendering functionality are implemented as extension to the framework.

[2]https://www.nltk.org/

[3]https://spacy.io/

[4]https://radimrehurek.com/gensim/

[5]https://scikit-learn.org/stable/index.html

[6]https://webgl-operate.org

[7]https://www.khronos.org/gltf/

Tree Geometry. The visualization component handles 3D models, i.e., the trees, encoded using the glTF file format. To describe the models contained in the scene graph, a hand-managed description file in JSON format is provided by the visualization designer, associating visual variables with a number 3D models in the scene graph. Each tree model is represented by a scene graph node within the while glTF file. Since trees may inherit a translation in the scene graph, the translation is normalized and adjusted to the tree position in the terrain for this visualization. Each tree has a corresponding data structure in memory and on the GPU as vertex buffers that contains all its attributes, given through the current attribute mapping. When the mapping changes, i.e., the user selects a different CSV column for a tree attribute, the attributes are updated and the vertex buffers are updated accordingly. After a change in its attributes, the tree model for a data point is determined by a best-fit approach among the rule set of the description file.

Terrain Generation. When loading a data set, the terrain is calculated based on the tree density. For this, the terrain is subdivided into a grid with a fixed resolution. For each tree, a Gaussian kernel is applied at its position, increasing the terrain height there and in the vicinity. In cases where many trees are next to few trees, the height is flattened with a non-linear fall-off, resulting in a flatter, more plain terrain. The goal is that the densest cluster on the whole terrain will reach maximum terrain height while a single tree will still have a clearly visible terrain height of ca. 30-50% of the maximum terrain height. In addition to the hills and beaches of the terrain, a static water texture is rendered to give a sense of cohesion to clustered tree groups. The water level is configurable in the user interface and may be used for height-filtering of data points (Limberger et al., 2018).

Tree Placement. Stating no additional assumptions on the characteristics of the 2D layout of the data processing pipeline, data points in the 2D layout may overlap or may be positioned very close to each other. Since the tree models have an actual and visible volume, these positions are not directly suitable for a clear visualization and may lead to trees standing within another. In order to prevent this, a basic collision reduction algorithm is applied. First, the terrain is subdivided into a grid with a fixed resolution. Then, each tree is assigned the closest position on this grid. If the closest position is already taken, alternative positions in the vicinity are checked until a free position is found. The alternative positions are looked up based on a fixed circle kernel in a clockwise direction, incre-

Figure 6: Four different assets comprised of different tree geometries and styles used for data mapping. The mapping, the number of visual variables, and the overall expressiveness of the forest is highly dependent on the assets and its description.

mentally increasing the radius until the search kernel is exhausted. In order to not make the clusters seems artificial because of the introduced circle-like structures, a random offset based on the original position is introduced. This results in the natural looking clusters in comparison to sole square or circle kernels.

Rendering. The tree rendering itself is implemented using WebGL 2 and its instanced rendering capabilities, i.e., the `drawElementsInstanced()` API. This makes it possible to efficiently render a model multiple times in the scene and allows to depict a large number of data points. However, tree types and their visual variables might change dynamically, thus, it is necessary to adjust the buffers sent to the GPU frequently since each tree might change its visual representation on every change in the attribute mapping.

6 RESULTS & DISCUSSION

We built a renderer that runs in modern web browsers and can thus be used on a variety of desktop and mobile devices (Figure 7). Moderately sized software projects (>10 000 entities) can be explored interactively with an integrated graphics chip. With a dedicated graphics card, even bigger projects (>100 000 entities) can be visualized with interactive frame rates. In the visualization, different presets are available that store a

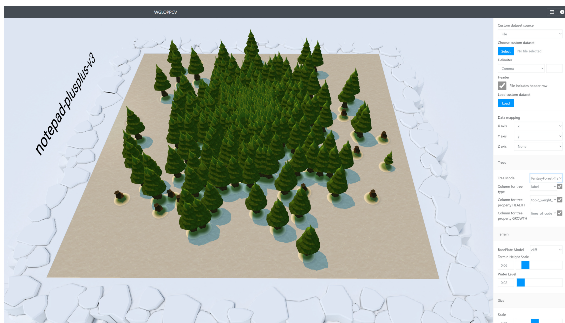


Figure 7: GUI of our visualization running in a browser.

configuration for the major user interface options and can be used to get a first impression of the framework or to get a similar view on different data sets quickly. Apart from that, every option can be adjusted to customize the visualization: The data sets can be chosen from a list of uploaded data sets or own data sets can be uploaded. Additionally, the currently used model files can be changed during runtime. The available visual variables of the model file are displayed in the user interface and can be assigned to data attributes. The set of example use cases shown in this work underline the capabilities of the proposed framework but are not hard-wired in the structure of the framework. Instead it is possible to easily adjust the visualization for other needs if the constraints on the data format are met, custom data files can be used and visualized.

6.1 Examples

Apart from the currently provided assets and their descriptions, other glTF scenes can be used to visualize the provided data (Figure 6). This effectively means swapping trees with other objects and provide appropriate visual variables, e.g., buildings, street furniture, rocks, flowers, etc. The listing 1 shows an excerpt of the description file used for Figure 1. Figure 8 shows a forest of the *notepad-plus-plus* project. The resulting topics of the software files are mapped to tree types and 2D position. A cohesive cluster in the center made up of tall, light green trees can be recognized. When inspecting the trees of this cluster using the mouse cursor, tooltips reveal that they share a dominant topic. Analyzing the words this topic comprises, the files can be related to user interface code. The topic model is correct in this judgment since the files displayed in this cluster indeed deal with the user interface of *notepad++*.

In Figure 9, 40 different open-source projects are visualized. The projects have been processed and their topic weights have been determined. The *Software Forest* allows to recognize similar projects, e.g., deep learning projects. The cluster of red maple trees is

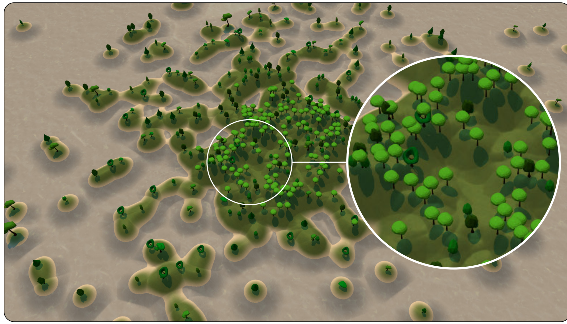


Figure 8: Software forest of *notepad-plus-plus* project.

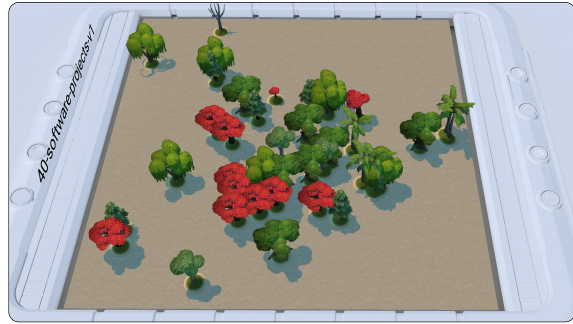


Figure 9: Software forest of 40 open-source projects.

composed of deep learning projects such as *CNTK*, *caffe*, *Pytorch*, and others.

6.2 Limitations

For now, we manually labeled the topics in order to give more context to users beyond the numerical topic index and word vectors. Since the topics returned by LDA are latent, they have no inherit label mechanism (Blei et al., 2003). This labeling is an essential part for making our visualization interpretable. Therefore, efficient ways to label topics automatically and consistently should be researched, also drawing from previous work and literature about this known problem of latent topic modeling (Magatti et al., 2009).

The proposed work could benefit from the usage of more data sources. Similar to Linstead et al. bug reports could be considered for additional input as well (Linstead et al., 2007). We did some first experiments using so called joint embeddings (Yu et al., 2017), which can be used for the integration of more diverse data sources, but more work needs to be done to conclude whether or not this is a promising approach.

When exploring large projects, dense clusters of trees occur frequently. To improve the visual clarity of the system, it would be possible to recognize clusters and aggregate similar trees in the visualization. Hovering over these trees or zooming into their region, the trees clusters could be expanded again, or tooltips with additional details could be shown.

```
1  {
2      "modelFile": "BiomeTrees.glb",
3      "attributes": [ "health", "growth" ],
4      "modelScale": 1.0,
5      "types": [ ...
6          {
7              "name": "Tree_Palm",
8              "baseModel": "Tree_Palm_02",
9              "variants": [
10                 {
11                     "name": "Tree_Palm_03",
12                     "health": 0.16,
13                     "growth": 0.16
14                 }, {
15                     "name": "Tree_Palm_02",
16                     "health": 0.5,
17                     "growth": 0.83
18                 }, ... ]
19         }, ... {
20             "name": "Tree_Birch",
21             "baseModel": "Tree_Birch_03",
22             "variants": [
23                 {
24                     "name": "Tree_Birch_04",
25                     "health": 0,
26                     "growth": 0.83
27                 }, ... ]
28         } ]
29 }
```

Listing 1: JSON configuration of a mapping from attribute values to 3D model.

7 CONCLUSIONS

The approach to visualize software data, i.e., a prominent example of data without intrinsic gestalt, using metaphors is often-tried but only scarcely used outside of research communities. We try to address application issues by proposing a metaphor close to nature and a dedicated visualization approach. As such, we propose *Software Forest*, a novel 2.5D software visualization to interactively explore a software system illustrated as a forest. The visual metaphor allows for multiple visual variables, including, but not limited to, tree type, growth state, height, and color. The layout of the forest is derived from a source code document analysis using Latent Dirichlet Allocation and Multidimensional Scaling. This layout reflects similarity in semantics of source code by proximity, and, thus, allows for an alternative mapping and view than traditional software visualization approaches such as treemaps. The layout is used for deriving a terrain and locations for the trees for further attribute mapping. The rendering approach takes different parameterization, different visual variables, and scalability into account. As such, using different 3D models is feasible and requires only the composition of a glTF scene graph and a description file with information on individual models and specific

attribute mapping. The approach is exemplified using mid-sized open-source software development projects. An application to the data sets indicates both an effective layout base on semantic similarity of source code and an explorative visualization approach of the resulting maps.

Future work can be discussed separately for layout generation and the visualization approach. For layout improvements, software similarity (Al-msie'deen et al., 2013) and suitable dimensionality reduction methods (Cha, 2007; Vernier et al., 2020) are target for further research. The visualization approach needs to be evaluated regarding readability using user studies. Regarding variability of the 3D models, tooling on the glTF models would simplify the integration of other 3D models, even for applications outside of forest metaphors. As the input for the prototypical viewer are CSV files with explicit layout and additional attributes as columns, we already imagine easy visualization prototyping and adaption in other visualization domains as well. Using a broader perspective, the proposed visualization approach allows for visualization of structured data sets with scatter-plot layout and dynamic model mapping for a wide-spread use of thematic mapping – the topic maps.

ACKNOWLEDGEMENTS

We want to thank the anonymous reviewers for their valuable comments and suggestions to improve this article. Further, this work is part of the "Software-DNA" project, which is funded by the European Regional Development Fund (ERDF – or EFRE in German) and the State of Brandenburg (ILB) as well as the "TASAM" project, which is funded by the German Federal Ministry for Economic Affairs and Energy (BMWi, ZIM).

REFERENCES

Al-msie'deen, R., Seriai, A.-D., Huchard, M., Urtado, C., and Vauttier, S. (2013). Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In *Proc. 14th International Conference on Information Reuse & Integration*, IRI '13, pages 586–593. IEEE.

Auber, D., Huet, C., Lambert, A., Renoust, B., Sallaberry, A., and Saulnier, A. (2013). Gospermap: Using a gosper curve for laying out hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 19(11):1820–1832.

Balogh, G., Szabolics, A., and Beszédes, Á. (2015). Codemetropolis: Eclipse over the city of source code. In *Proc. 15th International Working Conference on Source Code Analysis and Manipulation*, SCAM '15, pages 271–276. IEEE.

Barth, L., Fabrikant, S. I., Kobourov, S. G., Lubiw, A., Nöllenburg, M., Okamoto, Y., Pupyrev, S., Squarcella, C., Ueckerdt, T., and Wolff, A. (2014). Semantic word cloud representations: Hardness and approximation algorithms. In *Latin American Symposium on Theoretical Informatics*, pages 514–525. Springer.

Beck, F. (2014). Software feathers - figurative visualization of software metrics. In *Proc. 5th International Conference on Information Visualization Theory and Applications - Volume 1: IVAPP*, IVAPP '14, pages 5–16. INSTICC, SciTePress.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022.

Bruneton, E. and Neyret, F. (2012). Real-time realistic rendering and lighting of forests. *Computer Graphics Forum*, 31(2pt1):373–382.

Cha, S.-H. (2007). Comprehensive survey on distance/similarity measures between probability density functions. *International Journal of Mathematical Models and Methods in Applied Sciences*, 1(4):300–307.

Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., Deursen, A., and van Wijk, J. (2008). Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81:2252–2268.

Cox, M. A. and Cox, T. F. (2008). Multidimensional scaling. In *Handbook of Data Visualization*, pages 315–347. Springer.

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407.

Dübel, S., Röhlig, M., Schumann, H., and Trapp, M. (2014). 2d and 3d presentation of spatial data: A systematic review. In *Proc. VIS International Workshop on 3DVis*, 3DVis '14, pages 11–18. IEEE.

Hawes, N., Marshall, S., and Anslow, C. (2015). Codesurveyor: Mapping large-scale software to aid in code comprehension. In *Proc. 3rd Working Conference on Software Visualization*, VISSOFT '15, pages 96–105. IEEE.

Holten, D., Vliegen, R., and van Wijk, J. (2005). Visual realism for the visualization of software metrics. In *Proc. 3rd International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '05, pages 1–6. IEEE.

Kleiberg, E., van De Wetering, H., and Van Wijk, J. J. (2001). Botanical visualization of huge hierarchies. In *Information visualization, IEEE symposium on*, pages 87–87.

Kohonen, T. (1997). Exploration of very large databases by self-organizing maps. In *Proc. International Conference on Neural Networks*, ICNN '97, pages 1–6. IEEE.

Kuhn, A., Loretan, P., and Nierstrasz, O. (2008). Consistent layout for thematic software maps. In *Proc. 15th Working Conference on Reverse Engineering*, WCRE '08, pages 209–218. IEEE.

Limberger, D., Scheibel, W., Döllner, J., and Trapp, M. (2019). Advanced visual metaphors and techniques for software maps. In *Proc. 12th International Symposium on Visual Information Communication and Interaction*, VINCI '19, pages 11:1–11:8. ACM.

Limberger, D., Trapp, M., and Döllner, J. (2018). Interactive, height-based filtering in 2.5d treemaps. In *Proc. 11th International Symposium on Visual Information Communication and Interaction*, VINCI '18, pages 49–55. ACM.

Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., and Baldi, P. (2009). Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336.

Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., and Baldi, P. (2007). Mining eclipse developer contributions via author-topic models. In *Proc. 4th International Workshop on Mining Software Repositories*, MSR '07, pages 30–34. IEEE.

Magatti, D., Calegari, S., Ciucci, D., and Stella, F. (2009). Automatic labeling of topics. In *Proc. 9th International Conference on Intelligent Systems Design and Applications*, ISDA '09, pages 1227–1232. IEEE.

Rosen-Zvi, M., Griffiths, T., Steyvers, M., and Smyth, P. (2004). The author-topic model for authors and documents. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, pages 487–494. AUAI Press.

Scheibel, W., Trapp, M., Limberger, D., and Döllner, J. (2020). A taxonomy of treemap visualization techniques. In *Proc. 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications – Volume 3: IVAPP*, IVAPP '20, pages 273–280. INSTICC, SciTePress.

Schreiber, A. and Misiak, M. (2018). Visualizing software architectures in virtual reality with an island metaphor. In *Proc. International Conference on Virtual, Augmented and Mixed Reality: Virtual, Augmented and Mixed Reality: Interaction, Navigation, Visualization, Embodiment, and Simulation*, VAMR '18, pages 168–182. Springer.

Sievert, C. and Shirley, K. (2014). Ldavis: A method for visualizing and interpreting topics. In *Proc. Workshop on Interactive Language Learning, Visualization, and Interfaces*, pages 63–70. ACL.

Skupin, A. (2004). The world of geography: Visualizing a knowledge domain with cartographic means. *Proceedings of the National Academy of Sciences*, 101(suppl 1):5274–5278.

Steinbrückner, F. and Lewerentz, C. (2013). Understanding software evolution with software cities. *Information Visualization*, 12(2):200–216.

Štěpánek, A. (2020). Procedurally generated landscape as a visualization of C# code. Technical report, Masaryk University, Faculty of Informatics. Bachelor's thesis.

Vernier, E. F., Garcia, R., Silva, I. P. d., Comba, J. L. D., and Telea, A. C. (2020). Quantitative evaluation of time-dependent multidimensional projection techniques. *Computer Graphics Forum*, 39(3):241–252.

Wettel, R. and Lanza, M. (2007). Visualizing software systems as cities. In *Proc. International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '07, pages 92–99. IEEE.

Würfel, H., Trapp, M., Limberger, D., and Döllner, J. (2015). Natural phenomena as metaphors for visualization of trend data in interactive software maps. In *Proc. Conference on Computer Graphics and Visual Computing*, CGVC '15, pages 69–76. EG.

Yu, J., Jian, X., Xin, H., and Song, Y. (2017). Joint embeddings of chinese words, characters, and fine-grained subcharacter components. In *Proc. Conference on Empirical Methods in Natural Language Processing*, pages 286–291. ACL.

View publication stats