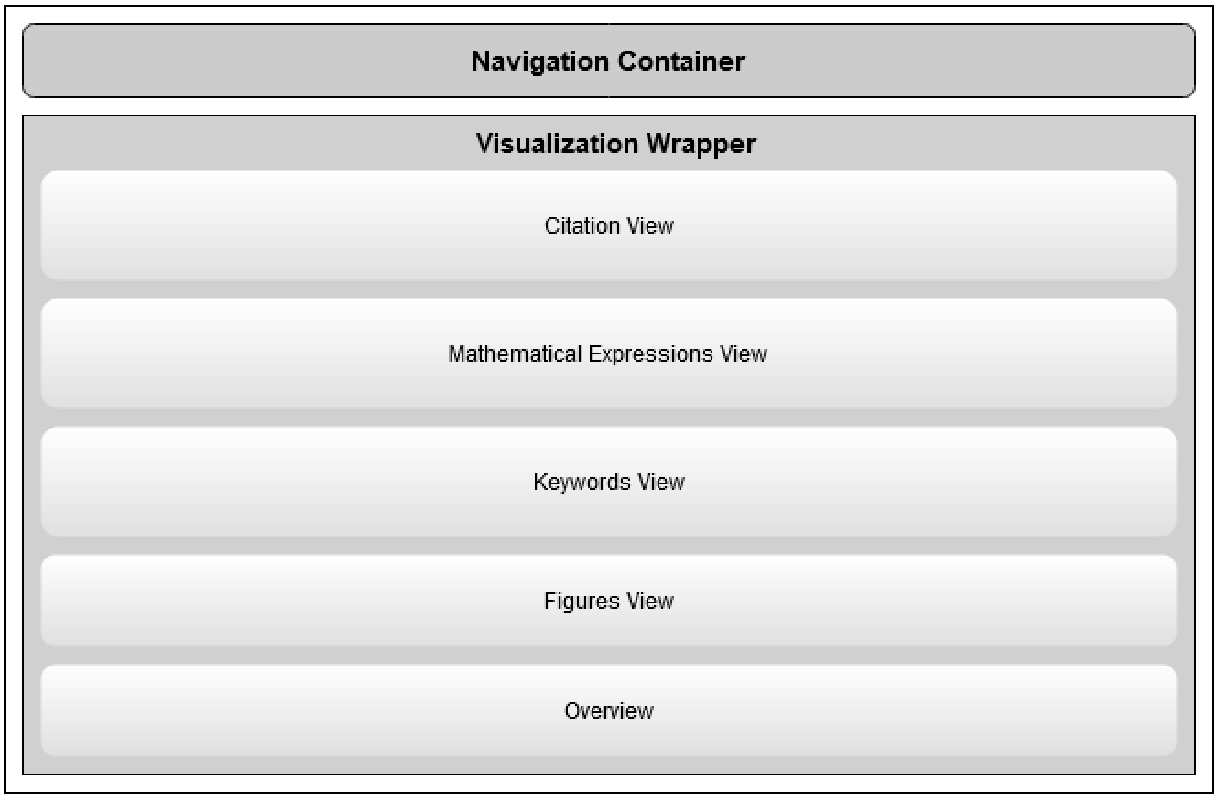
RecVis Detailed Comparison View Technical Documentation

Detailed.html



The detailed comparison view is divided into two main sections. The first section is the navigation container, the second section is a wrapper for the feature visualizations. Each feature visualization approach receives its container within the wrapper. The layout implementation can be found in the detailed.html file, where all the structural information about the detailed view is implemented.

To only display one feature view at a time, the CSS property display: none is being used. All views, other than the one which is being inspected, will have this attribute set. An HTML element with a style of display: none is treated by the browser as if it does not exist during the layouting. Thus, it is not considered when the browser calculates the width and height of each element. By doing this, the container of the feature which is considered active can take the entire space of the visualization wrapper, making it look as if there is only a single container on the page.

Detailed-style.css

This file contains all the styling information which are required by the html elements to be positioned properly. The styling is not only used for layouting purposes, but also to create pseudo-elements which can be used to assists the user in understanding the proposed visualization approaches. An example for such pseudo-elements are the semi-circles in the citation view: . Besides the detailed-style file, there is also the detailed-text-style.css. This stylesheet contains styles which are solely relevant for the keywords view and thus have been separated from the other styles.

Detailed.js

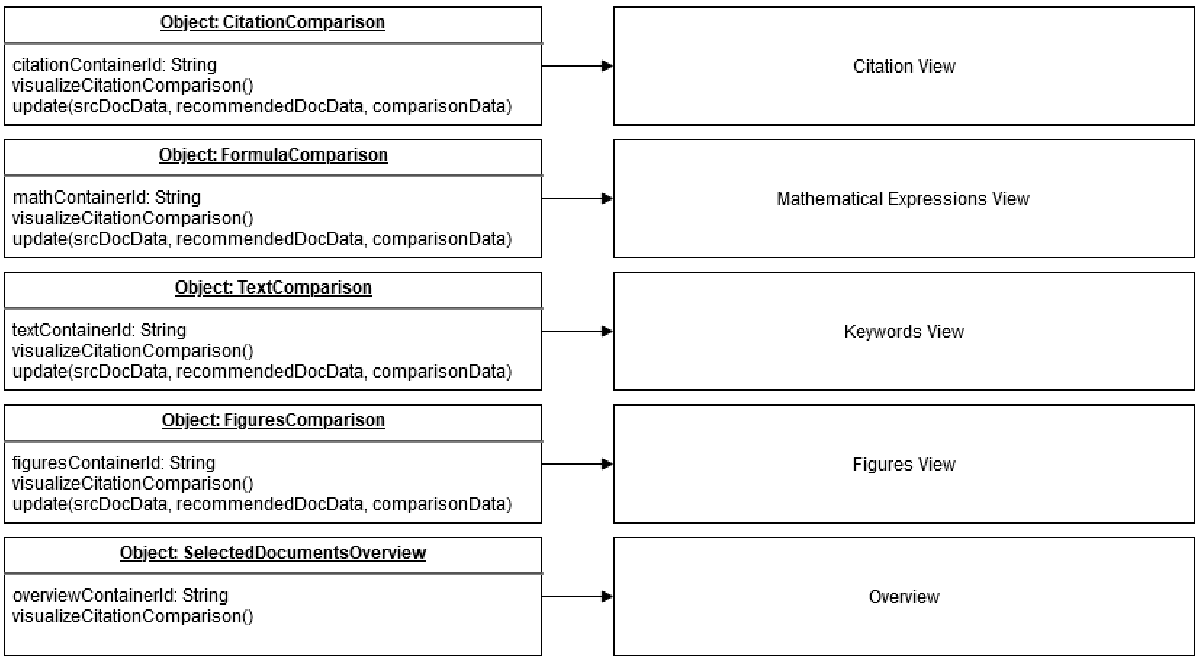
The detailed.js file contains the script which does the most work. To enable the navigation as well as the visualization approaches, it first requests all necessary data. Once the DOM content is loaded, four asynchronous callback-based requests are being sent to the RecVis backend.

The first callback retrieves information regarding which documents the user has saved under their collected documents list in the analysis view. This data is necessary to dynamically fill the dropdown menu of the navigation container with document entries which can then be compared against the selected document. Once the callback is resolved, the navigation container can be initialized.

Before the visualization approaches can be displayed, all necessary data needs to be available. This data is fetched by the other three asynchronous requests. These requests need to be joined before proceeding. They retrieve the source document data, the recommendation document data, and the comparison data of the two documents. To wait for the asynchronous calls to complete, two steps are required. First, a variable that keeps track of the total amount of callbacks that need to be resolved is initialized. Second, each callback function calls a mediator function which is called callbackResolved. This function decrements the count of the callback count variable. If the decremented value reaches zero, all callbacks have been resolved and the function either instantiates or updates a variety of objects. These objects are responsible for updating the contents within the feature visualization views.

The detailed.js file is the **last script** loaded in the head of the detailed.html file. Therefore, it can instantiate the objects which require classes that are declared in different script files as they have already been loaded.

Each view container in the visualization wrapper has a corresponding visualization class. When the page is first loaded, an instance of each class is being created by this script. The initial set of data is being provided to the instantiated object of each class via the constructor. Afterward, each view's container content is being updated dynamically by calling the visualize method of the corresponding object. Once instantiated, the update function of the objects can be used to insert and visualize new comparison data.


Object: CitationComparison
citationContainerId: String
visualizeCitationComparison()
update(srcDocData, recommendedDocData, comparisonData)
Citation View
Object: FormulaComparison
mathContainerId: String
visualizeCitationComparison()
update(srcDocData, recommendedDocData, comparisonData)
Mathematical Expressions View
Object: TextComparison
textContainerId: String
visualizeCitationComparison()
update(srcDocData, recommendedDocData, comparisonData)
Keywords View
Object: FiguresComparison
figuresContainerId: String
visualizeCitationComparison()
update(srcDocData, recommendedDocData, comparisonData)
Figures View
Object: SelectedDocumentsOverview
overviewContainerId: String
visualizeCitationComparison()
Overview

Detailed-citation.js

This script provides the definition for the CitationComparison class. Even though we use the word class, all the comparison classes are actually functions which can be used just like classes in JS. When first instantiated by the detailed.js script, this script initializes the visualization for the citation view.

Initially, the citation matches returned by all four algorithms are used for the visualization. The comparison data which was retrieved from the backend is an object which has the algorithm names as keys and an array of their data as the value. To get all the relevant data, we create a string array consisting of the algorithm names of interest. In the next step, we iterate over that array, using the entry at the current index as our key for the comparison data object. This way it is possible to access the corresponding comparison data for each algorithm. Each match is then pushed into a detection results array. Once the loop is finished, a set is being created from the detection results array to eliminate duplicate entries. To allow the user to select which algorithms to consider when creating the visualization, the checkboxes have an onChange handler which is called whenever a checkbox is checked or unchecked.

Once a set of unique detection results has been created, we identify the surrounding citing sentences for each match. This means that a match consists of the source document citation sentence and the recommended document citation sentence. We then create a span around the citation with a certain class name which will in the next step allow us to highlight the relevant citation matches within the identified citing sentences. This processed data is what we call our comparison data.

The next step is to visualize the generated data. This is where d3.js comes into play. First, we create a pack layout. This layout allows the hierarchical grouping of data into circles, also known as enclosure diagrams. Since the layout expects a d3 hierarchy, we need to mask our created comparison data accordingly. This allows us to pack our data, allowing d3 to generate x, y and r values for each data node. These values can now be used to draw the circles.

Each node corresponds to a citation match which consists of the source match sentence, the recommendation match sentence, as well as their longest string length. To achieve the desired visualization, we need to draw two semicircles for each node. This is achieved by using the SVG path element. The shape of a path is decided by its d element. Using d3, we can draw an arc into this d element. An arc has a start angle, as well as an end angle. If the start- and end angle differentiate by 180° or π, the shape is a semicircle. To draw the two semicircles, for each node, two arcs are being drawn. The gap between these two arcs is achieved by adding them to two different g elements. The first g element is for all the upper semicircles, the second for the lower semicircles. The first g elements y coordinate is transformed vertically to be positioned 15px higher than the lower group.

Placing the text on top of the semicircles requires a little trickery. It is not possible to append a text element to a path. The implemented solution makes use of the foreignObject[1] element. This element allows the use of elements from different namespaces. In our case, it is nothing more than a div element. For each semicircle, a corresponding div is being created. These divs can then be filled with the citation match data, the computed sentences. Since they contain a span, we set the divs innerHTML to equal our source and recommendation sentence, instead of using the innerText attribute. To properly display the text, the container still needs to be sized and positioned accordingly. This is done by calculating the maximum possible width and height that a rectangle can have to fit into the semicircle of radius r. Next, the x and y coordinates are being transformed to center the rectangle within the semicircle.

[1] https://developer.mozilla.org/en-US/docs/Web/SVG/Element/foreignObject

With the div being created and the text placed inside, one issue remains. The text needs to fit into the container. Setting the min-height instead of the height property would ruin our layout for longer text, making the previous calculation pointless. Setting the property overflow: hidden would simply cut off the rest of our text if it does not fit. So, the font-size needs to be scaled with the text length. The longer a text, the smaller the font-size. This scaling cannot be linear, as that would rapidly shrink the font size.

The formula which is currently used is $\frac{r}{\sqrt{n}}$, where r is the radius calculated for the node by the circle packing layout and n the total amount of characters in the sentence. It returns very decent results for sentences of normal size but scales too strong for sentences of extraordinary length.

To be able to better read the sentences which are put on top of the semicircles, the visualization supports simple zooming functionality. Each semicircle receives an onClick handler. When one of them is clicked, it calls the handler with its id, as well as its x and y coordinates. If the view is not zoomed in already, the handler then selects the element which made the call and translates the center towards its x and y coordinates with a scaling factor of two, thus making it easier to read the content. If zoomed in, clicking on a semicircle will reset the zoom. To be able to also zoom out when clicking the canvas of the SVG, an absolutely positioned rectangle with opacity zero is put on top of the entire SVG. This invisible rectangle has an onClick handler which resets the zoom when zoomed in.

Detailed-formulas.js

This script provides the definition for the FormulaComparison class. This class initializes the navigation and visualization of the math view. It works with dummy data. First, an input event listener is added to the math similarity threshold slider which defines a function that handles a value change of the slider. Formulas that do not meet the minimum similarity score will be removed from the list of formulas to display and the first formula which meets the new similarity score is being displayed.

To achieve this behavior, within the event handler, a working copy of the mock data is created by removing every match which does not meet the requirement of the minimum similarity set by the slider. Once the mock data is updated to meet the slider requirements, the handler first deletes the old visualization and then triggers the visualize function of the math view which redraws the visualization for the newly filtered data. This works unless the slider value is higher than the highest score assigned to the mock data, in that case, the formula match with the highest score is displayed, even though it does not match the filter value.

Transforming the textual formulas of the dummy data into HTML and CSS is done by using MathJax. On top of the detailed comparison HTML document, we first set up the configuration of MathJax before loading it, as it immediately starts the configuration when it is loaded. When updating the formulas within our formula containers, we put the textual representation of the formulas back into the elements. To tell MathJax to typeset these formulas again, we can notify it to check our content again. The following code snippet tells MathJax to typeset the document with the id src-formula:

```
MathJax.Hub.Queue(["Typeset", MathJax.Hub, "src-formula"]);
```

The highlighting of certain identifiers within a formula is achieved by adding classes within the formulas. These class declarations are then being processed by MathJax and added to the class list of the created HTML elements. These classes can then be used to style the identifiers.

Detailed-text.js

The class of this script is used to manipulate the keywords view. First, event listeners are being added to the help button to show a legend when hovering over it and to hide it when the mouse moves outside of the buttons range.

The keywords view is distributed into 13 columns. The first four columns represent the keywords of the selected document. The last four columns represent the keywords of the recommended document. The five columns in between represent the shared keywords between the two documents. In this context, keywords are words that can be used to identify the contents of a document. They are the words that appeared most frequently within a document. The words with higher frequency end up in columns with a bigger font-size to visualize the difference in relevance.

To achieve this visualization result, the first step is to process the document data to determine the keywords for each section. First, both texts need to be partitioned into an array of words. This procedure is done for both the source text as well as the recommendation text. In the next step, the word occurrences are being counted for each word. After the counting, a simple pluralization search is being done. Next, we filter certain stop words from our results by using a non-exhaustive list of the most used English stop words. As explained earlier, the keywords will be divided into columns depending on the occurrence frequency. To efficiently be able to do this, the processed keywords are sorted in descending order by their occurrence frequency.

Now that both the source document keywords map and the recommendation document keywords map are sorted, their top entries are compared to retrieve the most common keywords. The implementation either compares each word of the source document array with the top 500 entries of the recommendation document array or if the arrays have less than 500 entries, their maximum length is being used. This interval is set to make sure that not every single word is being compared, no matter the frequency and relevance. Whenever a match is found, it is pushed to a map that uses the word as its key and the sum of the indices as the value. This value will be used to sort the map. The lower the value, the more relevant were the words for both documents, as they appeared early on within the sorted maps, thus having a high occurrence frequency. In this case, the map will need to be sorted ascending.

Next, all the identified common keywords are being removed from the recommendation document- and source document maps. We now determine the maximum possible number of keywords per column by setting the size which a keyword will require in relation to the maximum container height and adding some margin.

Now that we know the maximum number of words within a single column, we can begin to place them. For the source and recommendation keywords, this matter is trivial. We iterate over all the columns. For each column we iterate over the keywords array and append a div to the column with the keyword as its text. The start and end index within the keywords array depend on the column we are in. The end index is further being decremented by the column number times a fixed value. This way, the columns with less important entries also receive a lower amount of entries.

For the identical keywords' columns, the procedure needs to be modified. First, the most important words are being placed in the center column. Next, the column to the left and right are being filled. Afterward, the two outer columns on the left and right are being filled. The general algorithm is the same, the only difference being that two columns are being filled at the same time now, thus requiring the start and end index to add a factor of two. To make sure that the words are distributed evenly when filling two columns at the same time, a modulo two operation is being used to alternate between the columns that are being filled.

The columns have the display property set to flex. In combination with the flex-direction: column and justify-content: center CSS properties, the browser does all the layouting for us.

Detailed-figures.js

The FigureComparison class initializes and manipulates the figures view. It also works with dummy data. When first instantiated, the minimum figure similarity slider is initialized. It has an event handler added which triggers our handle function on change. The handle function then first creates a new working copy of the mock data by filtering out all entries that do not meet the minimum similarity requirement. Next, it removes all children of the visualization container and then calls the visualize function which puts the first data entry in the working copy in the corresponding containers. This way the slider impact can be seen in real-time. There is only one exception. If the user chooses a minimum similarity which is higher than what the data provides, then the entry with the highest similarity which does not meet the requirement is being displayed.

On the bottom left of the figures view there is a checkbox that shows a similarity region on top of both figures when it is checked. These similarity regions are also randomized dummy data and used to explain a concept. Using d3.js, when the checkbox is checked, we first create an SVG which is absolutely positioned on top of the figures container. Next, we create a radial gradient. We then place a circle on both the source document figure and the recommended document figure which are filled with the previously created radial gradient.

The figure match container has the CSS property overflow-x set to auto and overflow hidden. A single figure match takes 100% of the visualization container. Thus, only a single match can be displayed in the container itself at a time. The other matches are also on the horizontal axis but cannot be seen due to the hidden overflow. To properly display the absolutely positioned heatmap circles on top of the figure matches, the left attribute of the parent SVG might need to be set. For instance, when looking at figure match three, we need a left property set to 200% to begin at the left border of the third citation match.

Detailed-overview.js

This script provides the definition for the RecommendationOverview class. This classes' approach was inspired by the d3 zoomable circle packing example[2]. It manipulates the overview view. This class also

uses dummy data. The data is being randomized to display a different result for the different comparison documents.

[2] *https://observablehq.com/@d3/zoomable-circle-packing*

To randomize data, we need to programmatically create an object. The object needs to follow the d3 hierarchy rules for us to be able to use the pack layout as previously shown for the detailed-citation.js file. The creation of this data can be seen in the following code snippet:

```
let overviewData = {};
overviewData.name = "root";
overviewData.children = [{
        name: this.recommendationDocumentData.title,
        children: getDocumentOverviewData(25), "size": 1
        }];
```

The getDocumentOverviewData function creates randomized data for the different features and their algorithms. This way, a randomly distributed hierarchy is being created. Using d3.js we can then generate nodes for our data which also consist of the x and y coordinates, as well as a nodes radius.

When hovering over an outer circle, the corresponding document title is being displayed on top of it. This is achieved in two steps. First, both the circle and its corresponding text are being appended to the SVG and grouped. By setting the display property of the corresponding nodes to inline, this allows for the text to be centered within the circle. This is not the desired behavior for the nodes other than the leaves and those should only be displayed when they are being clicked at.

To achieve the wanted behavior, we first set the opacity to 0 for all text. Next, we transform the y coordinates of all text nodes which are not leaf nodes, by subtracting their nodes radius. This pushes them on top of the corresponding circles. Now we need to enable displaying the text on hover as for now, it will always be invisible. This is achieved by adding mouse listeners to the corresponding circles. When the mouse hovers over a circle, its corresponding text node opacity is set to 1, if the node is at depth one or two. Leaf nodes are at depth 3 in our hierarchy. When the mouse cursor leaves the node, the text opacity is set back to 0.

The current setup allows us to hover over none leaf nodes to see the text on top but when zooming in on the leaf nodes, there is no text displayed. This functionality is now being implemented in the zoom function. Whenever a node is clicked, the zoom function is called to zoom to the node that has just been clicked. Using the following code, we check if a transition has been made to a node at depth 2, whose children are leaf nodes for which we want the text to display without hover:

```
transition.selectAll("text")
    .filter(function (d) {
        return d.parent === focus || this.style.display === "inline";
    })
    .style("fill-opacity", function (d) {
        //only display text at depth 3 within the bubbles
        return d.parent === focus && d.depth === 3 ? 1 : 0;
    })
```

The focus variable is the latest node on which the user has zoomed. First, we select all the text nodes which are direct children of the node that has been clicked last. If these nodes are at depth 3, they are leaf nodes and their text is being displayed.