University of Stavanger

Faculty of Science and Technology

BACHELOR'S THESIS

Study program/ Specialization: Computer Science	Spring Semester 2021 Open access
Writers: KEVIN RATDAL MUSTAFA HERSI	Kevin Ratdal Mustafa Hersi Writers' Signatures
Faculty Supervisor: Vinay Setty	
Thesis title: **Smart text editor/Plugin to fact check**	
Credits (ECTS): 20	
Key Words • Fact checking • Chrome extension • RESTful API • Cosine similarity	Pages: 72 +enclosure: 0 Stavanger May 29, 2021

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

smrtfact - A smart fact checker

Bachelor's Thesis in Computer Science
by

Kevin Ratdal, Mustafa Hersi

Internal Supervisors

Vinay Setty

May 29, 2021

"The human brain has not evolved to perceive reality, it has evolved to create an illusion of reality. That's why an exciting lie gains more attention than a boring truth."

— Abhijit Naskar, I Vicdansaadet Speaking: No Rest Till The World is Lifted

Abstract

With the increase in social media activity, and people from all backgrounds being able to use it, it has become increasingly difficult to separate between what is true and what is false. When people post statements on the Internet, they might believe what they write is true, when in reality it might not be. The tool we have developed will help writers confirm the validity of their statements. It will display if what they're writing is true or not, before they put it out to the public.

Acknowledgements

A big thanks to Vinay Setty from the Department of Electrical Engineering and Computer Science at UiS for helping us with this thesis. He has given us tips and tools along the way, leading to this finished product.

Contents

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
JSON	**J**ava**S**cript **O**bject **N**otation
GIF	**G**raphics **I**nterchange **F**ormat
DOM	**D**ocument **O**bject **M**odel
UI	**U**ser **I**nterface
SPA	**S**ingle **P**age **A**pplication
URL	**U**niform **R**esource **I**dentifier
ID	**I**dentification

Chapter 1

Introduction

1.1 Motivation

Fake news and other different types of false information can take on many faces. It could be your favourite blogger or your neighbourhood politician. Either way, they can have major impacts on our lives. Because as we make decisions based on the details we gather, we might change an opinion on something after we've been fed with this information. Therefore, it's important that the facts we digest are correct. Having real and accurate information is very important - now more than ever. As we are all connected through the internet, and more specifically social media. According to Statista, the current number of smartphone users in the world today has far surpassed 3 billion [1]. This means more than 48% of the world's population owns a cell. Consequently, the phone application TikTok has over 600 million active monthly users [2]. And it would be incautious to proclaim TikTok itself has the capability to oversee every post by every user in case of fake news.

The fact that the spreading of fake news and information can come out of control when the biggest applications don't have built-in filters or mechanisms to prevent it, is a scary thought. The consequences of an even more rapid escalation of this issue can target and poison the entire society.

1.2 Problem Definition

When false information is posted online, there can be different reasons for it. Though there is little information behind why people deliberately do this, it is known that some people are driven by hope of a better self-esteem, or just hate for someone or something

[3]. It could also be for economical reasons. Whatever the motive, the rumours spread - or the misleading information posted - can target and affect businesses, organizations, politics and individuals. An example could be a restaurant, posting fake viewpoints and making false accusations on another restaurants' website. Stepping on a competitors feet like this could be beneficial in an economic way without much effort. This is what is called *disinformation*. Disinformation is when "fake or misleading stories are created and shared deliberately, often by a writer who might have a financial or political motive, e.g. propaganda" [4].

The task at hand, where it was asked to create a program that could check for false information, was focused more towards *misinformation* - "...fake and misleading stories, but in this case the stories may not have been deliberately created or shared with the intention to mislead." [4]. By creating an extension, people are given the option to look over what they - or any other person - have written, and eliminate falseness they might unintentionally be writing or seeing. The latest research tells us only just over 2% of young people can actually spot fake news, with more than 60 percent saying that fake news make them trust all news in general a lot less [4]. This is one of the reasons why it was important to create this application. The group was free to choose in what way to solve the given problem description.

1.3 Goals and inspiration

As mentioned earlier, social media user bases continues to grow as technology becomes more and more accessible, and people get more and more curious to what this "social media" talk is all about. More people entering the social media world, means more room for information spread, thus making more room for false information. When the false information waves are escalating, motivation and room is being created to solve the complications that come with it. Rightly so, there have been created applications out there that identify, fight and negates fake news and false information. Some examples of how these work are mentioned in Chapter 2.

Even though the Internet is flourishing with social media applications, it has been found that there are very few alternatives to srmtfact. The extension takes inspiration from the web browser extension Grammarly. Grammarly is not made to detect or negate any fake or false information. It it used for spellchecking your writing. It works by creating an empty document - a text editor - and filling it with text. It will scan through the document and point out any misspellings. Grammarly is comparable to the extension in the way that misspellings are swapped out with misinformation. smrtfact's end goal is to show what is correct information, and what is incorrect information. Though operated in

the somewhat same manner as with Grammarly, with smrtfact, you don't have to go into a new web page to do the work. It can either be done directly on all websites, although there is some additional support to three major web sites. As well as in a custom fashion by going inside the extension itself. You can put in whatever claim you want, and create a true/false output for this exact claim. In addition to the end goal, we wanted to make it an easy experience by making a browser extension out of it.

1.4 Use cases

The target audience of smrtfact is everyone, all ages. However, the project group think it will be most useful for youths and younger people. This is because young people are naturally more impulsive, and often express their thoughts in a less considered manner. Because of this, it will not only be useful for younger people, but it will be important for the application to reach out to this audience, so that they can make use of it. Additionally, as the members of the project group can be counted for as members of the "young people" group themselves, they can relate and back up why they think this is an accurate view.

smrtfact can be used in almost every scenario where you are about to publicise a statement, claim, fact, or just anywhere where the text has a true/false factor. On web sites likes the ones integrated with smrtfact - Facebook, Twitter and YouTube, this is a big concern. These web sites are social media web sites where people can post about things happening in their lives, things they believe in, and thoughts they have. They can even comment on other peoples or organizations posts. Even comment on others' comments! This is often where the "keyboard warriors" come forth. These are people that dare to state facts and opinions without regarding anything, thinking it will have no consequence because they sit safely in their home behind the computer screen. This gives rise to a lot of bias and false information.

Brands or organizations that have something to do with subjects like politics, the climate, or health, are often in the center of intense debates. When these organizations make a post, they easily wake peoples passion and feelings, creating environments for discussion. This is where people start commenting about what they think or how they think things work, or should work. And this is often backed up by facts, or at least they often think they are facts.

1.4.1 Intro example

If you were to visit either Facebook, YouTube or Twitter, it is easy work to find examples of where the extension might prove useful. Below is an example from Facebook, of

a person commenting on an opinion in the comment section of a post by the highly discussed vegan health movie "Game Changers".

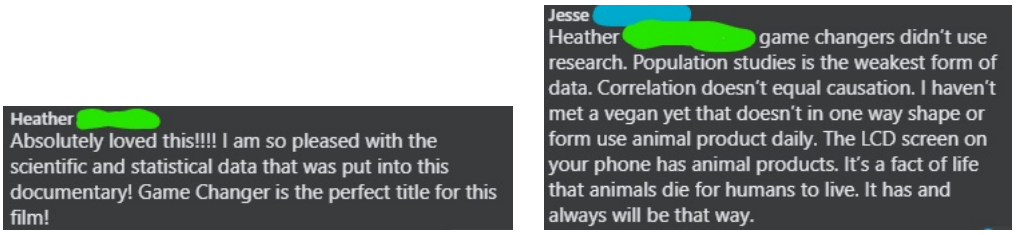

Figure 1.1: Example from the comment section under a post from the movie "Game Changers" on Facebook

It is a viable option to use smrtfact in this scenario. The first comment is an opinion. So no use or need to fact check anything here. But if the first sentence - *... game changers didn't use research ...* - of the second comment by "Jesse" is taken out and fact checked, it's possible to see how much of what this person says, is gibberish. It can be done by selecting whatever sentence you'd want out of the second comment. You then put this sentence into the application, and check for validity. The extension couldn't give an accurate result if you were to fact check the whole paragraph. You would need to point out the chunks of text where it's sensible to check for correctness. Putting the before-mentioned sentence into the extension yields the result below.

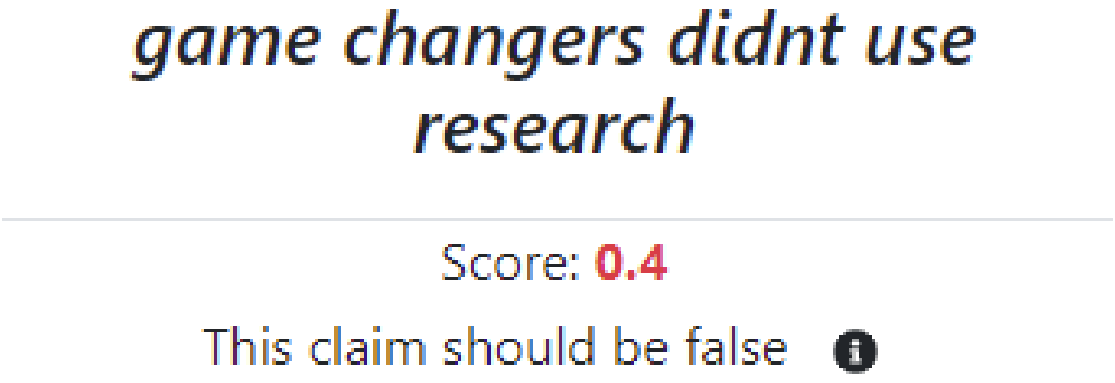

Figure 1.2: A fact check from a comment from Facebook

As one can see, it gets a score of 0.4, and receives an assessment of "*This claim should be false*". 0.4 is a relatively low score. From this result, one can already have prejudice against this person regarding how much "facts" "Jesse" comes with. Many people would be eager to invalidate the rest of her comments, just based on the first one getting such a low score.

1.5 Outline

The thesis is outlined as follows:

Chapter 1 Introduction to the thesis

Chapter 2 Related work and background, includes general and theoretical information

Chapter 3 Solution approach, includes the popup and it's functionality

Chapter 4 The fact check button, considering functionality injected into current website

Chapter 5 Implementations and experiments, includes measuring the time it takes to display claims

Chapter 6 Discussion and Future Directions, includes challenges during the development process and future ideas

Chapter 2

Related Work and Background

2.1 Extensions

Browser extensions are software add-ons for your web browser, which aims to make your browsing experience more trouble-free and/or "better" to use. Their use cases range from customising the background color of your browser, to scanning for coupons on different web sites, to blocking incoming advertisements. The first browser which supported extensions was Internet Explorer version 5 in 1999. Google Chrome got extension support in 2010, 2 years after Chromes' creation [5].

2.2 HTML

HTML, short for Hypertext Markup Language communicates with the web, and lays the base for creating a web application. It is the body - or chassis - of a car. The web browser obtains HTML documents from a web server, and puts them into the actual web site.[6]

2.3 CSS

Cascading Style Sheets (CSS) is the language that makes up the style and design of the web page. In other words, determines the type of leather in the seats of a car, or if it should have a mahogany or carbon fiber dashboard, or if it should be painted red or blue on the outside.

2.3.1 Bootstrap

Bootstrap is a CSS framework. A collection of code written in CSS, JavaScript and HTML. It is made for front-end development to simplify the process of styling fully responsive websites in a quick manner. So in essence, it saves the developers from spending too much time styling the websites, allowing them to focus on writing the functional and general code.

Since Bootstrap eases the process of styling, the well-known toolkit was made use of when making and designing the UI of smrtfact. Its benefits and convenience suited the project perfectly when creating the front-end of the extension.

2.4 JavaScript

JavaScript - or just JS - is a language in computer programming commonly used for adding functionality to web pages (and extensions in this case) there by making them interactive [7]. While CSS and HTML is used to style and structure the application, JS gives elements that engage the end users. Following the analogy of the car, JavaScript makes the car function. It gives interactions between the gas pedal and motor, or between the steering wheel and the axle. It makes the different tools engage with each other to create a fully functioning car.

It was initially designed to make front-end web development easier, but in later times it has expanded to cover programming in the back-end as well. It is also the only language that is native to the web browser, which is why it almost always becomes the chosen language to use when making browser extensions also.

2.5 Vue.js

Vue.js is a JavaScript framework for building front end UIs (User Interfaces). At it's core, it provides a way to build a component that encapsulates data or states in your JavaScript, and then in a reactive manner connects that state to a template in HTML. These components are called declarative views, because the same data inputs will always produce the same outputs in the visual UI.

When data is declared on this data object, it links or binds it to the HTML element in the template above. When the value of the data changes, the component will automatically re-render, or in other words be reactive. This data can be worked with in the template.

And thanks to Vue's HTML-based template syntax, a value or expression can interpolate using double braces around it.

2.5.1 Vue Life-cycle hooks

Life-cycle hooks allows you to take a peek at how the Vue library functions behind-the-scenes. Following this they allow the developer to know when a component is created, added to the DOM, is destroyed or updated.

This diagram from the Vue.js documentations shows how the Vue instance's life-cycle functions

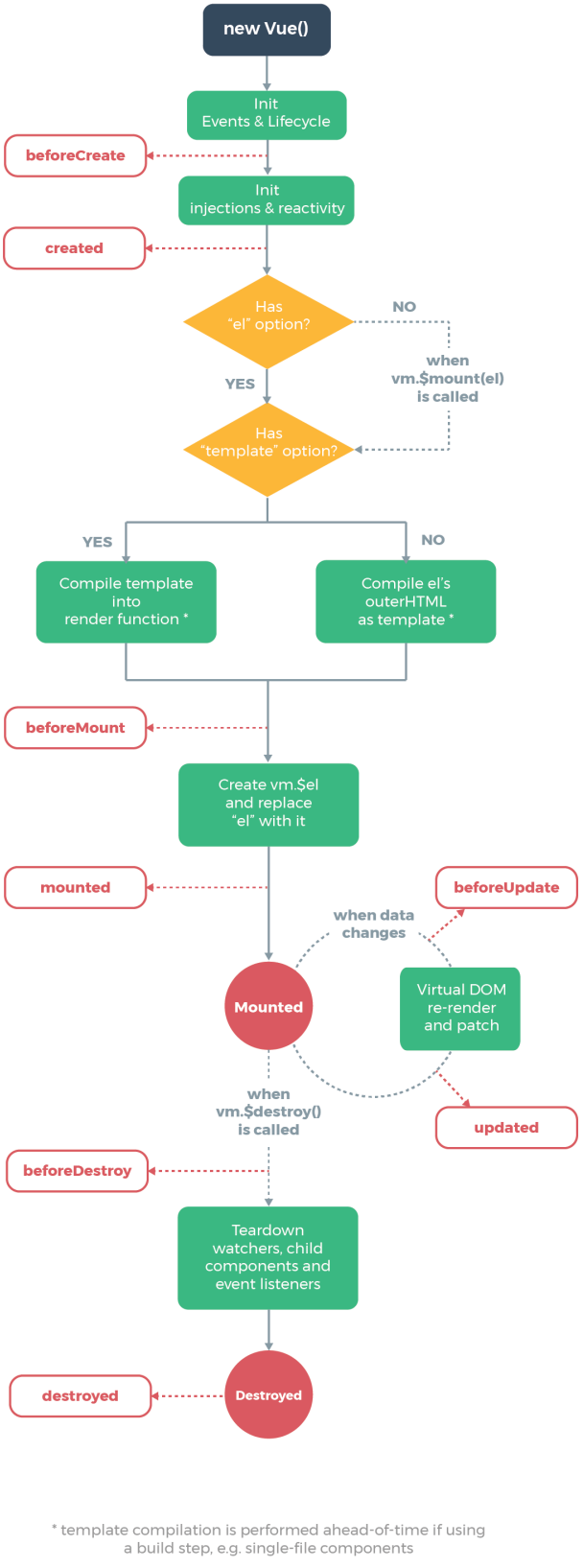

Figure 2.1: Vue life-cycle Diagram

2.5.2 Vue extensions

The Vue.js framework already has a lot of functionality, but it also allows for so called Vue extensions to expand this functionality. In the smrtfact extension, Vue router is used for simplifying a SPA setup.

2.6 Shadow DOM

Shadow DOM - Shadow Document Object Model - serves for encapsulation from the main document DOM tree. It allows a component to have its very own “shadow” DOM tree which can be interacted with and manipulated separately. This prevents changes from the main document to unintentionally affect these encapsulated components. It makes it so the component is able to keep the markup structure, style, and behavior hidden and separate from other code on the page. Essentially, it’s put together so that separate parts of the code don’t clash, all by letting you attach a hidden separated DOM to an element.

2.7 API And Chrome API

An API, short for Application Programming Interface, is a "software link" that allows two applications to communicate with each other. As an example, when you use an application on your mobile phone, the application connects to the Internet and sends data to a server. The server then retrieves that data, interprets it, performs the necessary actions and sends it back to your phone. The application then interprets that data and presents you with the information you wanted in a readable way. This is what an API is - all of this happens via API.

The Chrome suite of APIs contains many of these so called links, that allows interaction with different pieces of the google chrome application. Some of which proved to be quite essential in the application, and some of which didn’t. The range of functionality of the suite is quite broad. It includes features from time scheduled actions using the chrome.alarms API, to a local or synchronised storage system using the chrome.storage API. In smrtfact three of them are being used.

2.7.1 chrome.storage

The chrome.storage API allows user the access two areas of storage, Local storage and sync storage.

- The local storage area is located on the users device and does not transfer to other installation on different devices.
- The sync storage area is like the local storage are but it does sync to other devices. This means that if you have two computers, A and B, both with the same extension installed, you can store something on computer A, and have it available on computer B and vice versa.

These two areas share the same functionality, including storing data, retrieving data, removing data and event handling for changes and more.

2.7.2 chrome.tabs

The chrome.tabs API allows manipulation of tabs, and includes useful tools to send messages from the popup part of the extension to the content script. It also enables fetching of information about the currently active tab.

2.7.3 chrome.runtime

The chrome.runtime API contains lots of useful tools and functionality when developing a chrome extension. Some worth mentioning are communication between content scripts and background scripts, returning information from the manifest, and listening or responding to events in the app or extension life cycle. The runtime API also enables you to convert the relative path of URLs to fully-qualified URLs.

Below is an example that shows how a content script can add an image in the extension's package to the page that the content script has been injected into.

```
//// content.js ////

{ // Block used to avoid setting global variables
  let img = document.createElement('img');
  img.src = chrome.runtime.getURL('logo.png');
  document.body.append(img);
}
```

Listing 2.1: Simple extension code example

Within the chrome.runtime there also is a message system, which is useful when communicating between different layers of chrome extensions. Considering there is a "Popup" layer, a "background" layer and eventually a "content script" layer which all run separately, the message system allows for these to listen to and send messages to each other.

2.8 RESTful API

A RESTful API is an Architecturally type of API, which uses HTTP requests to access and use data between two applications. It's stateless, in the sense that it doesn't keep track of the clients state. It performs a set of operations that can be accessed by GET, PUT, POST, and DELETE. In this case the server is idle and standing by until the extension send a GET request with the claim as a parameter. It then computes the claim received, and sends back a JSON object. [8]

2.9 Cosine similarity

Cosine similarity is a metric - shown as the cosine of an angle - that is used to tell something about the similarity of two documents without regard to their length or size (Euclidean distance) [9]. This method of finding similarity is regarded as superior to only counting common words. When the size of the regarded documents increase, the amount of words they have in common also shoot up. And when finding the cosine similarity, the two documents can still be far apart by the Euclidean distance, but still have a small angle, giving an accurate computation.

To find the cosine similarity, you take the cosine of the angle between two vectors in a multi-dimensional space, where each dimension corresponds to a word in the respective document. The vectors are the two arrays that contain the count of all the words in each of the documents.

Say you have two blocks of text, M and N, and you choose to let the three words, A, B and C lay the base for how similar M and N are. The three words correspond to one axis each. Lets say the word count for word A, B and C in text M come out as [A=3, B=5, C=1], and [A=2, B=3, C=2] for text N. Two vectors are then received. Computing the cosine similarity between these two happens using the following formula:

$$cosine(M, N) = \frac{M \cdot N}{||M|| \cdot ||N||}$$

Plotting in the numbers provided above give us a value of 0.942908. This is the cosine similarity. Intuitively, if the variable a is the similarity number, the absolute values' range for this metric is $0 \leq a \leq 1$.

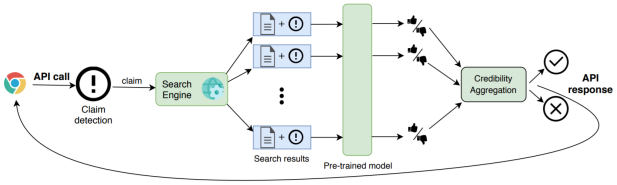

Figure 1: Block diagram of the server.

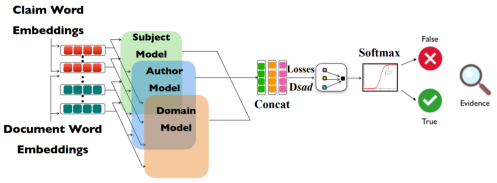

Figure 2: SADHAN Model

Figure 2.2: Server which claims are sent to

2.10 Automated Fake news detection using machine learning

Text or URL claims are sent to the server. The server then retrieves relevant articles from the WEB using search engines like Google and analyzes them by applying machine learning models. [10]

2.11 Existing Approaches/Baselines

Since the recent growth in technology, and more people making use of the Internet to read news, source criticism has become more important. People have picked up on this, and a simple Google search for "fact checker extension" provides several hits on different tools for checking for falsities, though not similar to smrtfact. A lot of them revolve around political bias. One of the bigger programs found in this search, called the "Media Bias/Fact Check Extension" by Mike Crowe[11], attaches a meter to posts on Facebook (among other sites). The meter shows how politically left- or right biased each newspaper is. The problem with this is that each newspaper has a prefixed reason on how left or right it is. Generalising means accuracy is lost.

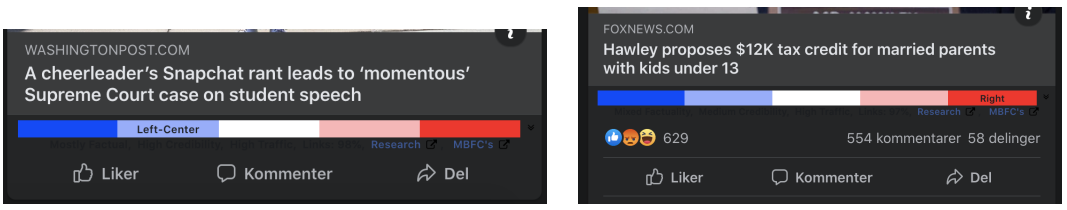

Figure 2.3: Crowes extension, examples from Facebook. Left is a Washington Post article judged as left-center-biased. Right is a Fox News article judged as right-biased

There is a site called Politifact.com, where there are claims already put into the web site, and the claim has a truth-o-meter attached to it, telling you how liable that exact claim is.

In theory, the smrtfact extension can also check for political correctness in addition to any general fact. By taking base in making custom claims an option, it can hit every category where it's important to get the facts right. Political, scientifically and intellectually based claims are all feasible options.

Chapter 3

Solution Approach

3.1 Introduction

The group were free to choose which tools to use and how to solve the task. Therefore the solution became making a browser extension. This seemed like the most practical solution to be able to check facts on a website without leaving the site. This is because extensions work as small software program which can enhance and/or customize your browsing experience, and allow additional functionality to successfully layer over web pages.

3.2 The browser of choice

Google Chrome became the browser of choice. There were several reasons for this. One was because of the fact that as of February 2021, Google Chrome accounted for 63.64% of overall internet browser market share worldwide [12]. Since it has such a large user base, and is such a matured product, the documentation has been well written and maintained. Google chrome also happened to be the browser of choice for the authors, which then helped settle the choice of platform.

3.3 Workflow

In this section the process of sending a claim in the extension will be described from start to finish. A flowchart illustrating the process is also included.

When the user enters a claim and presses the check button, a quick check and cleanup of the claim is initialized. The claim gets stripped of any trailing white space and characters not accepted by the API. Minimum length is also checked before proceeding. If this part fails, nothing more happens and a error is alerted. If it succeeds, the claim is sent to the content script using the chrome.tabs message system with a tag for the type of claim it is, and the popup renders its /loading view.

Within the content script there is a listener for messages which captures messages and sends them forward to the background page using the chrome.runtime message system. This step is included so that whether the claim is being sent from the popup, or from a text field on the website, it gets sent through the same process.

In the background layer, the claim gets passed into a cosine similarity check against existing claims, accessed from a claims key:value pair in a chrome.storage.local system. If it is deemed similar enough to an existing claim, a message with a tag for handling similar claims is sent out. This message is then captured in the popup and renders the /alias view, where the user has the ability to add it as a related claim in the similar claim, or add it as a separate claim.

if the user decides to add it as a separate claim it gets sent back to the background layer, and an asynchronous request to the fact checking API is made. During this the user is sent to the /loading view. If an error occurs in this step, the error then gets captured by an on Change listener for the storage system, and gets rendered on the loading view. After 10 seconds the user is redirected to the /tools view, and can make another claim. If the API request succeeds, the claim gets put into the storage system, and the user gets redirected by yet another on-change listener to the /result view for the current claim.

If the user on the other hand decides to add it as a similar claim, the claims gets sent back to background with a similar tag, which adds it within the similar claim in storage. This again triggers the on-change listener, and routes the user to the appropriate /result view.

Stepping back to the background layer, if the claim is not found to be similar to existing claims, it gets sent to the API in the same way as explained above. Lastly if the claim already exists within the storage, a message is sent, containing the id of the currently existing claim and a duplicate tag. This message gets picked up by the popup using a message listener, and the user gets redirected to the respective /result page.

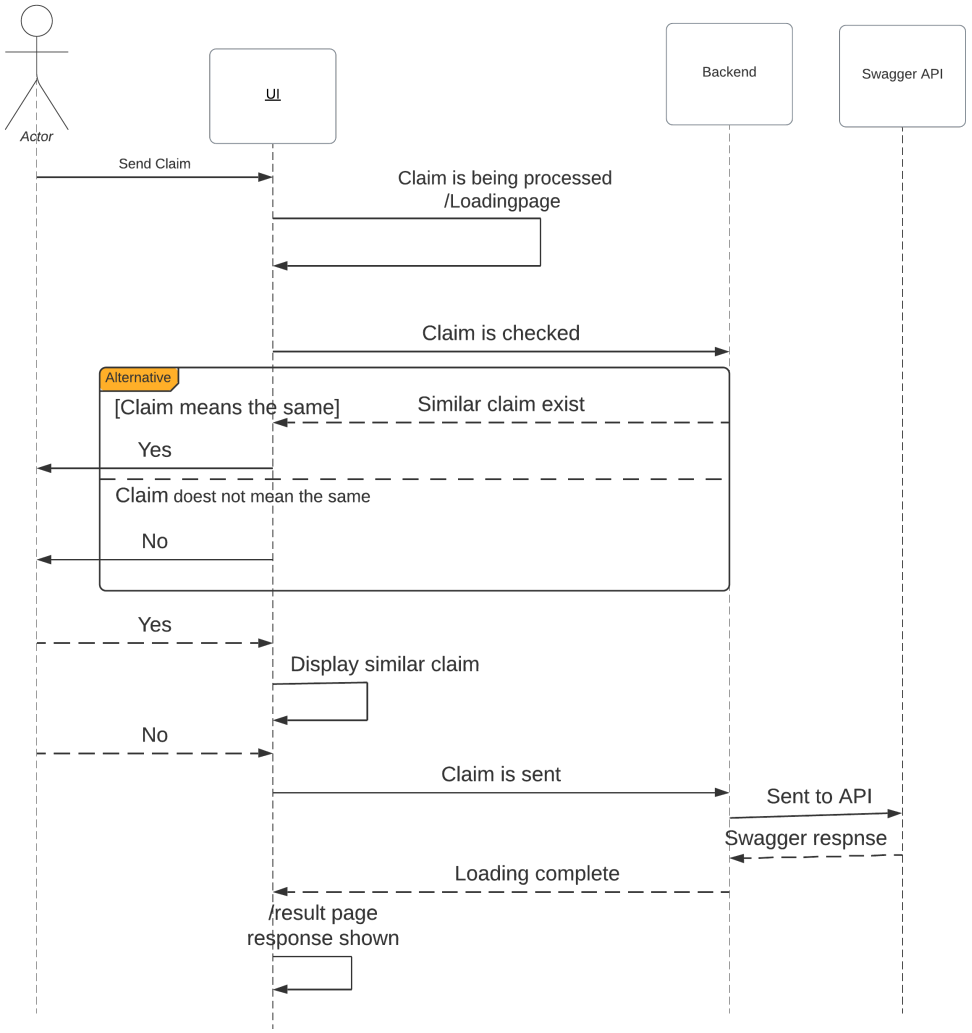

Figure 3.1: Flowchart

3.4 Server interaction

When the client sends a claim or a URL to the API, there is two possible routes being used. The first one is called *text-mining*, and accepts a string aptly named, *user_claim*, which contains the claims which is to be checked.

The other route, which accepts an article's URL is called *article-mining*. It accepts an string in URL format, *article_url*.

An API request for the claim "the earth is flat" would therefore look something like this:

```
\{API\_address\}/text-mining?user_claim=the%20earth%20is%20flat
```

An API request for the article URL "https://example.com/article1" would look something like this:

```
\{API\_address\}/article-mining?article_url=https://example.com/article1
```

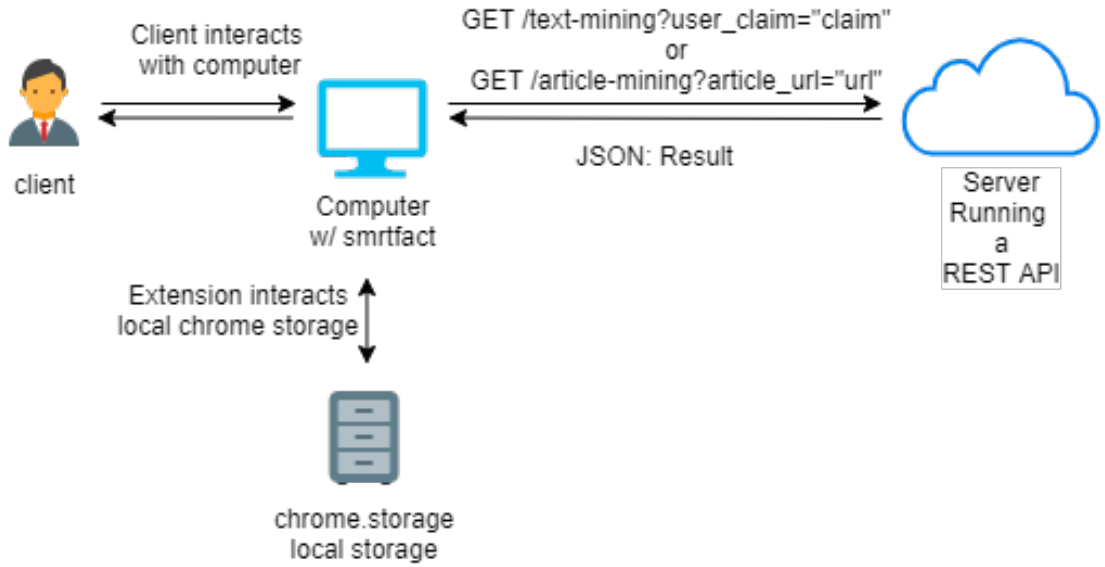

Figure 3.2: Client-server diagram

3.5 Chrome storage

An API that has been very important in the development of the extension, was the before-mentioned Chrome storage API. It is used in order to store, retrieve, delete, and track changes in user data. Though it doesn't offer data encryption, it doesn't matter, because the information it takes and processes is not confidential, and it does the job just fine.

A call to the Chrome storage API would look like this:

```
chrome.storage.local.set({key: value}, function() {
  console.log('Value is set to ' + value);
});

chrome.storage.local.get(['key'], function(result) {
  console.log('Value currently is ' + result.key);
});
```

3.5.1 Storing data

To be able to use the Storage API you must declare "storage" in the "permissions" list in the extension manifest as seen in manifest.json in chapter 3.6.3. When storing the data you can choose between sync and local "mode". Sync means the stored data will automatically be synced to any Chrome browser that the user is logged into, while local stores it on the computer. In the extension, it was decided to use local storage. Primarily because the limit on how much data can be stored is larger in local than in sync.

3.5.2 Managing history/related claims

Adding the content of our chrome storage to an array declared in data on our parent component. Enables the parent components data to be used in it's children components. Such that in the *history component* an array is created which is a deep clone of the data in the parent component.

```
this.desiredDisplay = [...this.$parent.allData]
```

As for related claims, a new key is created in the claim object, containing all the similar claims. This allows the user to access the similar claim, without waiting for the API to respond with a new claim, therefore speeding up the process.

3.5.3 Tracking changes in storage

The benefits of using the chrome.storage API is that it emits an "onChanged" event, whenever a change is made to the data object. As this event fires off every time there is a change, a listener is added to this event, thus being able to update the front end application to have the latest user claims.

3.6 Chrome extension

3.6.1 Extension file outline

The extensions' file outline looks like the following:

```
/extension
|-- background.js
|-- /components
|   |-- alias.js
|   |-- historyt.js
|   |--loadingPage.js
|   |-- navbar.js
|   |-- result.js
|   |-- settings.js
|   |-- tools.js
|-- contentScript.js
|-- /css
|   |-- bootstrap.min.css
|   |-- style.css
|-- generators.js
|-- /img
|   |-- Ellipsis.svg
|   |-- default.png
|   |-- extension.png
|   |-- fail.png
|   |-- icon.png
|   |-- loading.gif
|   |-- logo.png
|   |-- ...
|-- /js
|   |-- all.js
|   |-- bootstrap.bundle.min.js
|   |-- vue-router.js
|   |-- vue.js
|-- manifest.json
|-- popup.html
|-- popup.js
```

Listing 3.1: "File outline"

3.6.2 The back-end

The back-end of the application receives queries from the user interface in the browser. Here, the queries are handled, and returns the handled data to the front-end for the user to see. When it gets sent to the front-end, it has been "modified" to JSON - JavaScript Object Notation - format.

3.6.3 manifest.JSON

The JSON format is commonly used when transmitting data in between web applications. [13]

The manifest.json acts as a blueprint for metadata and permissions for the Chrome extension. This is also where all scripts and entry points are defined. Local resources are also defined here, like images one would want to have access to in a content script.

```
{
    "manifest_version": 2,
    "name": "smrtfact",
    "description": "a Smart fact checking extension for google chrome",
    "version": "2.0",
    "author": "STG0002664",
    "content_security_policy": "script-src 'self' 'unsafe-eval';
                                object-src 'self';",
    "browser_action": {
        "default_icon": "img/icon.png",
        "default_popup": "popup.html",
        "default_title": "smrtfact"
    },
    "background": {
        "scripts": [
            "background.js"
        ]
    },
    "content_scripts": [
        {
            "matches": [
                "http://*.example.org/*",
                "https://*/*",
                "http://*/*"
            ],
            "run_at": "document_idle",
            "js": [
                "contentScript.js",
                "generators.js"
            ]
        }
    ],
    "permissions": [
        "activeTab",
```

```
        "tabs",
        "http://*.{API IP}}/*/",
        "storage",
        "contextMenus"
    ],
    "web_accessible_resources": ["img/*.png","img/*.gif",
                                 "js/bootstrap.bundle.min.js",
                                 "css/bootstrap.min.css"]
}
```

Listing 3.2: "manifest.json"

Most APIs must be registered under the permissions field in order for the extension to be able to use them. The "activeTab" gives an extension temporary access to the currently active tab when the user invokes the extension. "tabs" are for interaction between the different tabs and windows. The "http://API IP/*/" is for accessing the server which is hosted on that URL. *API IP* is where the actual IP of the server should be, but is censored due to confidentiality. Storage also needs to be given permission for it to be used. Lastly contextMenus is used to add items to Google Chrome's context menu.

3.6.4 Content Scripts

For our extensions content scripts, the functionality was split into two scripts. These being contentScript.js and generators.js

contentScript.js contains the following functionality

- Handling of messages, and events
- Website detection
- Retrieving claim from websites
- Inserting button into field

generators.js contains the following functionality

- Generating elements which should be placed on the active website's DOM.
- Generating overlay page
- Generating button

3.6.5 Background page

A background page is loaded when it is needed, and unloaded when it goes idle. Some of the events are when it's installed for the first time or when it updates to a new version.

```
chrome.runtime.onInstalled.addListener( function (details) {
    let defaultValue = [];
    let defaultSettings = {
        filter: "default",
        false_boundary: 0.4,
        truth_boundary: 0.6,
        hist_sort: "newest",
        hist_filter: "all",
        hist_limit: 5,
    }
    chrome.storage.local.get({ claims: defaultValue }, function (data) {
        chrome.storage.local.set({ claims: data.claims,
                                   errorMessage: "OK" }, function () {
        });
    });
    chrome.storage.local.get({ settings: defaultSettings }, function (data) {
        chrome.storage.local.set({ settings: data.settings }, function () {
        });
    });
    if(details.reason == "install") {
        console.log("This is a first install")
    }else if (details.reason == "update"){
        var thisVersion = chrome.runtime.getManifest().version;
        console.log("Updated from " + details.previousVersion + " to "
                    + thisVersion + "!");
    }
})
```

Listing 3.3: "Install/update sequence"

Furthermore, it is used for listening to events and acting upon them. When a user sends a claim, a message is sent with an instruction and claim to the background script. A listener will pick up this message and swiftly send it to the API to be processed.

3.6.6 Communication

Content scripts, background scripts and the popup scripts all work separately. The content scripts specifically run in the context of a web page and is "injected" into the currently focused website, that is if it has permission to do so. Since they are independent and provide different limitations and possibilities, communication between these become important. The chrome.runtime message system comes in handy here, allowing these layers to communicate and transport data between themselves.

3.7 Accessing the popup

The popup of the extension can be accessed by clicking the smrtfact extension in the list of Chrome browser extensions as shown in Figure 3.3.

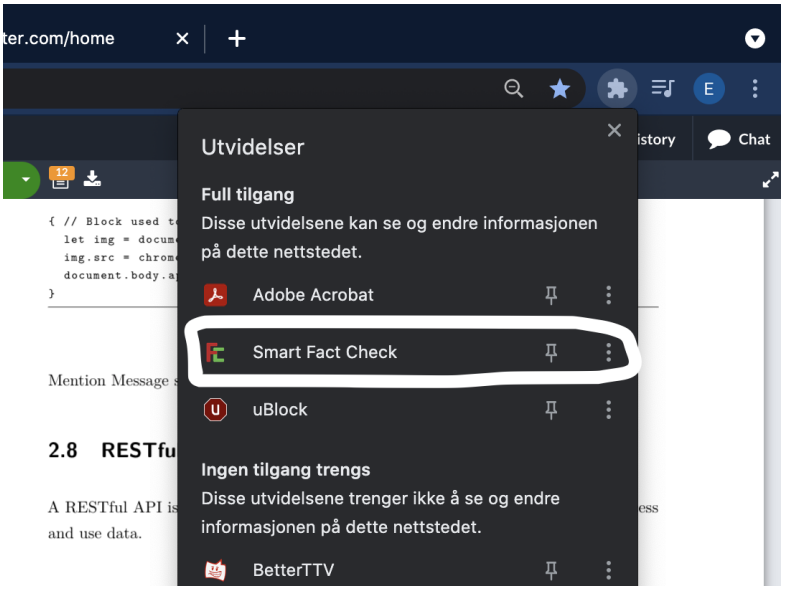

Figure 3.3: How to access the popup

Clicking it will bring up a popup where most of the functionality of the extension reside. It will start up in the Tools tab as shown in Figure 3.5 in chapter 3.9.1.

3.8 The Interface

When building the UI, a lot focus was on simplicity and functionality. All the tabs (Tools, Result, History and Settings) each have their own important purpose which is discussed later. Though inside one SPA, making four separate tabs, cost simplicity and sleekness. Initially, it was thought to have all components in one single tab, but with later work it was logical that they had to be separated. It wouldn't have been possible to display the outputs as desired if a single tab was to be the solution. In Figure 3.4 you can see how the tabs are displayed in the header of the popup.

3.9 Routes

The Extension popup window is built using Vue.js with a SPA layout in mind. In the popup window there is a toolbar, containing shortcuts to different "views" of the application.

Figure 3.4: The popup header

The components are all routed with Vue Router from a constant Array *routes* in the *popup.js* file as below.

```
const routes = [
    { path: '/', component: Tools },
    { path: '/tools', component: Tools },
    { path: '/settings', component: Settings },
    { path: '/result/:claim_id', component: Result, props: true },
    { path: '/result', component: Result, props: true },
    { path: '/history', component: Historyt },
    { path: '/loading', component: loadingPage },
    { path: '/alias', component: alias}
]

const router = new VueRouter({
    routes
})
```

Listing 3.4: Vue component routes.

By utilizing a Vue router, it enables the application to render different views from these defined routes. These views can then be rendered in a sub-window in the application, allowing the application to only re-render the needed components instead of redirecting to separate windows for each view. Therefore acting like a SPA instead of a multi page application.

3.9.1 Tools

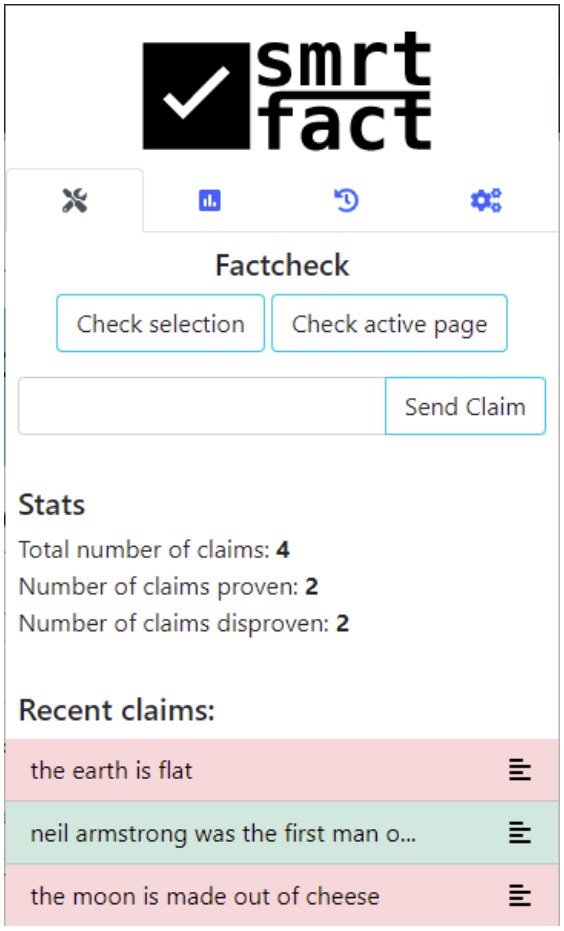

Figure 3.5: The tools tab

Tools component is where the primary functionality of the extension lies. At the top there are three actions you can choose between:

- Check selection
- Check active page
- Send Claim

Underneath there are stats, showing the amount of claims and their results. And clickable links to the three most recent claims.

To use the *Check Selection* button, the user must have the popup open, then mark a text in the browser by using the cursor, and then click *Check Selection*. The marked text will first be sent as a claim to the API. Secondly it will redirect to the loading page while the claim is being processed, and finally redirect to result page when the API returns the result. This process starts by the sendM method in tools.js, but the redirection after the loading page is handled by various event listeners in popup.js. For convenience sake *Check active page* has been added. This button - *Check active page*. When clicked, will retrieve the URL of the window you are in, and fact check that page.

```
sendM: function (e) {
            let self = this;
            chrome.tabs.query({ active: true, currentWindow: true },
                function (tabs) {
                chrome.tabs.sendMessage(tabs[0].id,
                    { instruction: e, data: self.claim },
                    function (response) {
                        self.$router.push({ path: 'loading' });
                    });
            });
        }
```

Listing 3.5: How Check Selection works

Optionally, you can write your claim directly into the input field and click the *Send Claim* button, which will trigger the checkClaimType function.

```
checkClaimType: function() {
            if (this.validateURL(this.claim)) {
                let temp = this.claim.split('.')
                if(temp[0] != "https://www") {
                    this.claim = "https://www." + this.claim }
                let e = "factCheckURL"
                this.sendM(e)
            }else {
                this.claim = this.claim.replace(/[^a-z0-9 \.,_-]/gim,"");
                if(this.claim.trim().length < 10) {
```

```
            return alert("Could not compute this claim,
                          please check if its correct")
        }else {
            this.claim = this.claim.trim()
            let e = "factCheckText"
            this.sendM(e)
        }
    }
```

Listing 3.6: checkClaimType function when pressing Send Claim

There have been implemented checks in order to see whether it was a URL entered, or if it was a text claim. Such that the claim can be sent to the appropriate API endpoint. Additionally these checks are used in a computed function, to deduce whether the user is trying to type a text claim or an URL claim and change the button's name between *Send Claim* and *Send URL*. Purpose of this is to clearly show the user that what they are writing is being interpreted as an URL or vice versa for a better user experience.

3.9.2 Related claims

If a claim turns out to be very similar to an already existing one, you will be redirected from tools to the *alias* component.

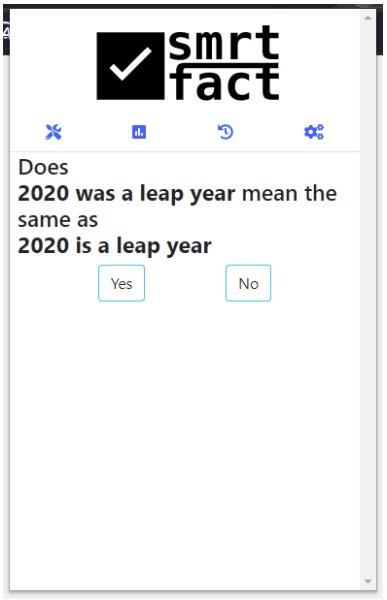

Figure 3.6: alias page

As seen in figure 3.6, you are presented with two options. For instance if clicked yes, it will automatically redirect to result, see figure 3.10 to see how result page looks with related claims. However if clicked no you will be redirected to loading page as a regular claim.

3.9.3 Loading ...

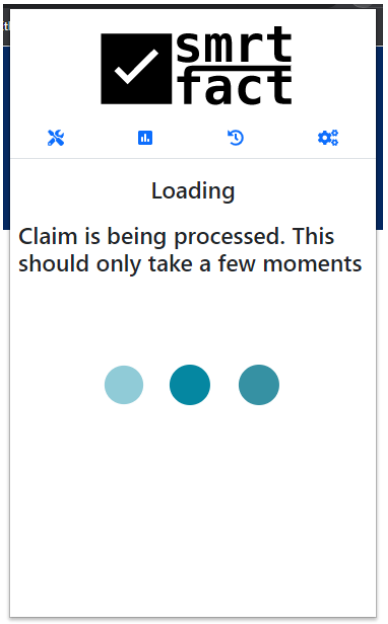

Figure 3.7: Loading page

The loading page is a route only accessible when you use one of the fact checking tools on the tools component which will redirect you to the loading page. Loading page consists of a message and a loading GIF. After this two things will happen, either the claim is successful and you will be directed to result. Or the claim is unsuccessful, meaning there is an error on the server side.

Error handling

When sending an asynchronous request to the fact checking API, it's within an await. An await causes other operations to pause until a promise is settled. This is essential as other operations can't proceed until a response is returned from the server. Inside the asynchronous request - *.then()* methods are chained, also called *composition*[14]. *.then()* methods return a promise, a promise represents the eventual completion or failure of an asynchronous function. As a result if the response fails, a *.catch()* method can be used to deal with rejected cases. Thus allowing errors to be handled. When the asynchronous function is unsuccessful, the Loading page will update and display the error message. This happens because error messages are stored as a key:value pair in storage. By having a chrome storage on-change event listener, errors will automatically appear in the popup as they arise.

The loading page will then run the function which displays the error and redirect to homepage/tools after 10 seconds.

3.9.4 Result

The *Result* component consists of two routes, one with a prop and one without. The result page will by default display the latest claim made, the route without a prop value. Whereas the route with a prop is used when the user wants to view a specific claim. For example if the user happens to click on one of the recent claims in the *tools* component, or other previously made claims that can be found in the *history* component.

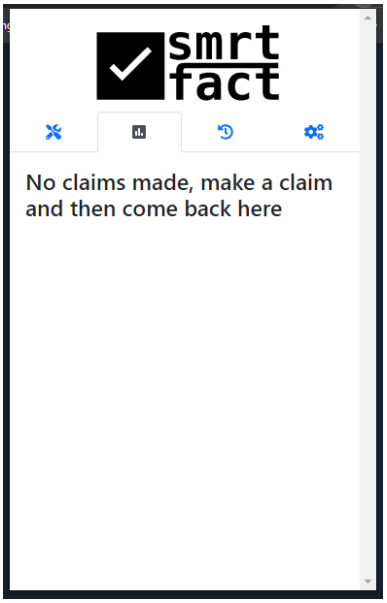

Figure 3.8: Result page when no claims have been made

If there are no claims made result page is empty like in Figure 3.8. However the moment a claim is returned from the server, it redirects to the result page and loads up the information.

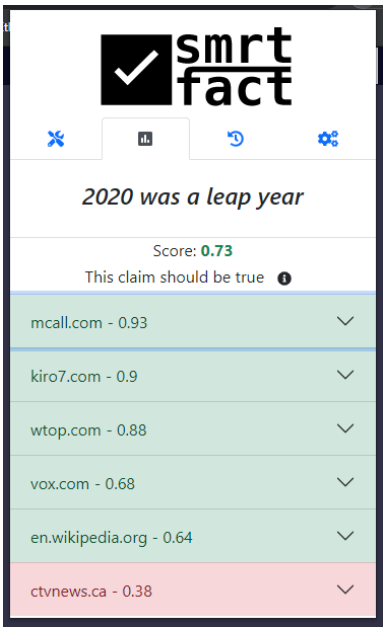

Figure 3.9: How the result page looks after a claim has been made

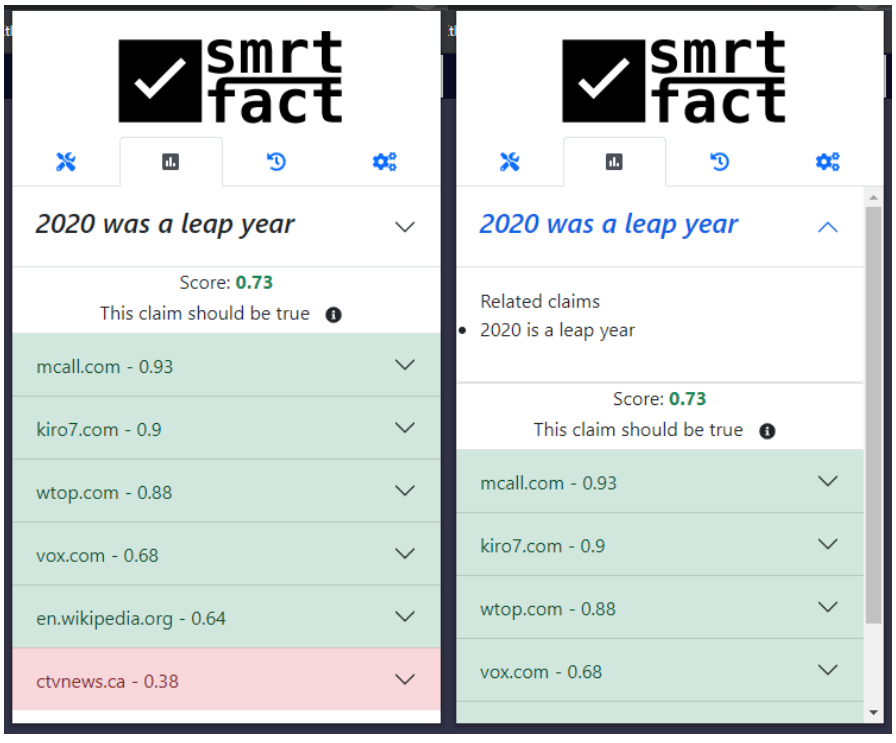

Figure 3.10: Result page with related claims

Loaded information needs to be concise. Therefore the information to display would be the final prediction - true or not, and the articles which the claim is basing its statement on. The articles which the user can view, contain a snippet - with a link to the website, and the date the article was written. To make it easier for the user, there is colouring to the articles. Green if they support the statement, or red if they disagree with the statement.

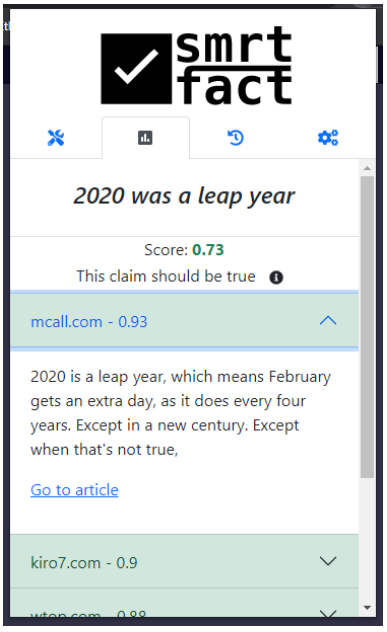

Figure 3.11: How the result page looks when article's clicked

3.9.5 History

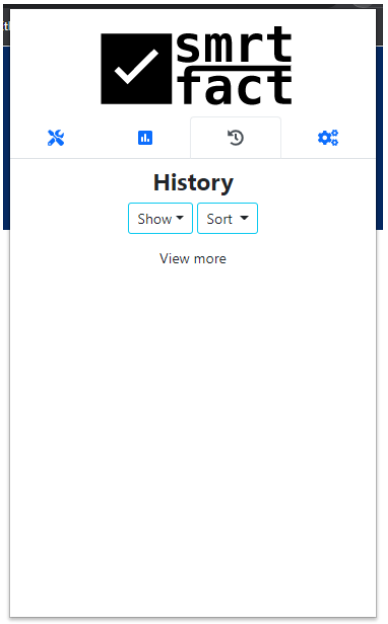

Figure 3.12: Empty history page

History component stores all claims made, and has two filtering functionalities.

- Show
- Sort

Firstly the *Show* drop down button lets you choose between simple text claims or URL claims. Secondly the *Sort* drop down button lets you choose in which order the claims should appear. By clicking on one of the claims, you will be taken to the result page using a prop to identify which claim to display.

```
<router-link v-bind:to="'/result/'+claims.id">
```

Listing 3.7: Router link to show specific results of specific claim

An id is passed as a prop so that the clicked claim will be retrieved in the result page. By letting the claims have their own id, the order of which they are being rendered, doesn't change. This is done because the rendered array of claims, isn't necessarily ordered the same way as the main claims array.

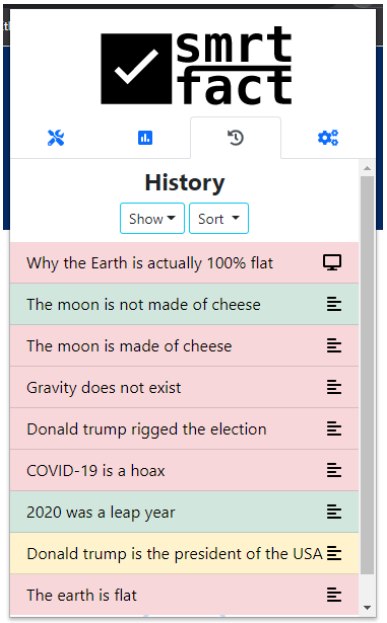

Figure 3.13: Expanded history page. NB: Computer icon symbols URL claims, while the others are text claims

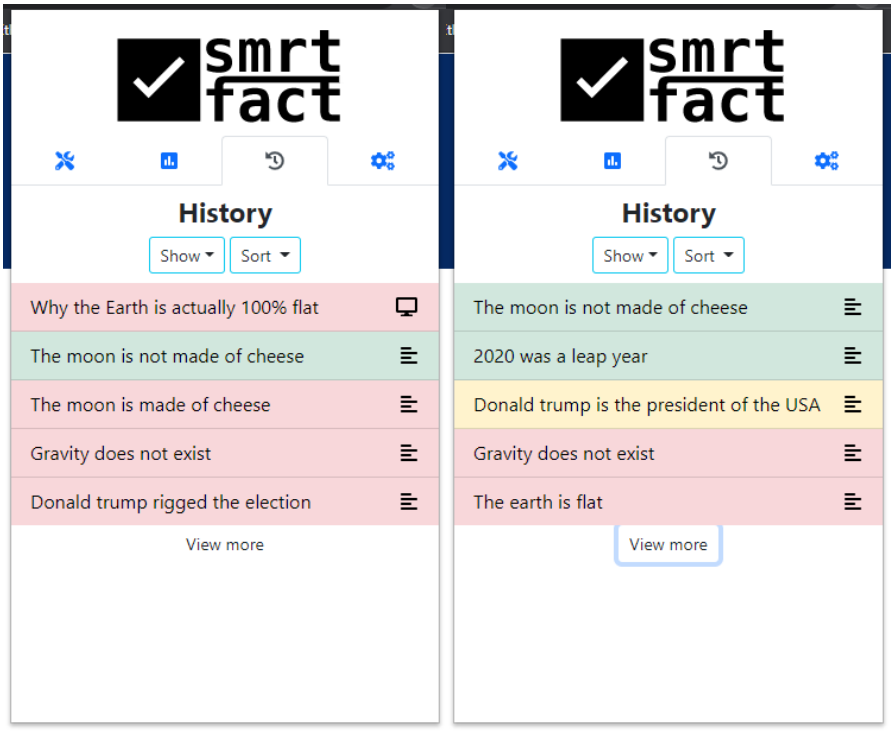

Figure 3.14: History page unsorted and sorted

3.9.6 Settings

Settings has many functionalities,

- Setting user desired truth and false boundary
- Setting persistent ordering of claims in history
- Setting persistent filtering of types of claim
- Setting persistent length of amount of claims to display
- Clear history
- Reset settings

Using persistent filtering enables a tailored user experience. For example the user is able to set their own requirements for how high or low the criteria must be for it be a true statement, or a false statement. Furthermore if they want the order of claims sorted differently or how many claims they want displayed, it can be set in settings. To achieve this chrome.runtime.onInstalled is used, because it runs whenever the extension is installed for the first time. Making it so that the default settings can be set in chrome storage, and later on be fine-tuned by the user.

Use case

Claims truthfulness are measured in a scale from 0-1, 1 being most likely true, and 0 being most likely false. A true statement requires a score of over 0.6, and a false statement requires a score of 0.4 and below. If the user for any reason deemed the boundaries for truth and false as too loose or too strict, the user can then easily adjust the boundaries to fit more into their requirements. For example setting truth boundary to be 0.8 will make it harder for claims to meet that criteria, and consequently less claims will return true.

Clearing history and resetting settings to default are accessible as buttons. As you can see from figure 3.17 there are two buttons; *Clear History* and *Reset Settings*. These buttons run methods which resets the array stored in chrome storage.

```
methods: {
        clearHistory: function () {
            chrome.storage.local.set({ "claims": [], "errorMessage": "OK" },
            function () {}
            );
        },
```

```
        resetSettings: function () {
            self = this

            chrome.storage.local.set({
                "settings": {
                        "false_boundry": 0.4,
                        "filter": "default",
                        "truth_boundry": 0.6,
                        "hist_sort": "newest",
                        "hist_filter": "all",
                        "hist_limit": 5,
                } }, function () {

                self.defaultSettings = true
            });
        }
    }
```

Listing 3.8: methods for clearing history and clearing settings

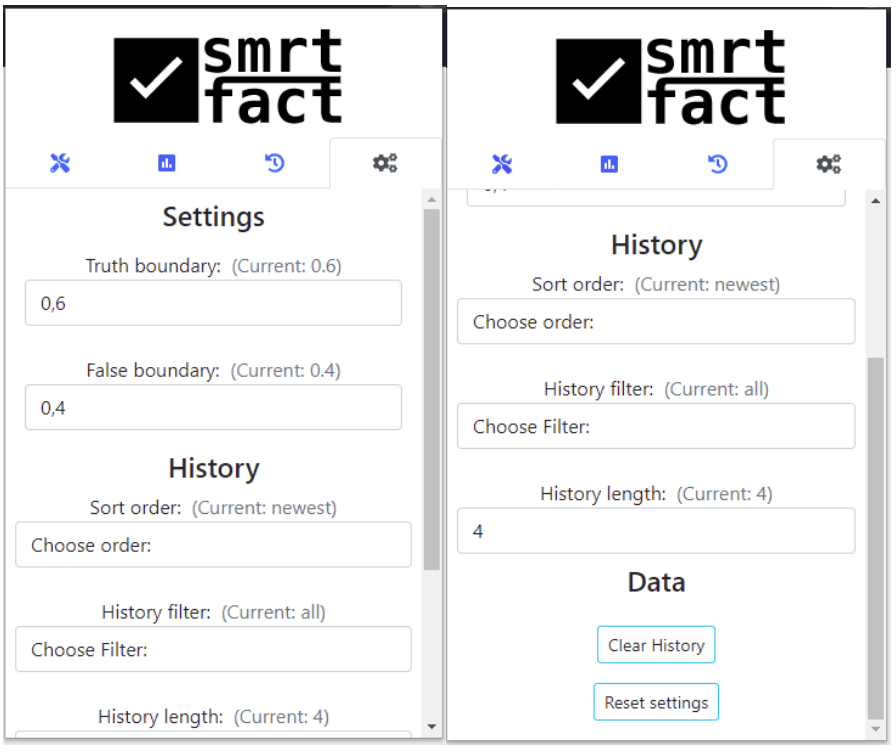

Figure 3.15: Settings route

Chapter 4

The fact check button

4.1 Introduction

To be able to easily fact check the chosen text, the smrtfact extension comes with a layover to the right in the text editor fields of the three web sites. The three chosen websites are Facebook, YouTube and Twitter. All three very renowned for being big factors when it comes to spreading information. By choosing these, a big user base is reached, as these applications - according to Statista - had billions of monthly users in 2020 respectively [15]. If billions of users with their billions on posts were to fact check their posts before posting, there would be a lot less fake information going around. Thus the fact check button was implemented on those sites. Looking at figure 4.1 you can see the fact check button before any text or claim is made in the editor. It looks very plain with only a light blue question mark button. It looks the same for all three of the web sites, and does not look any different whether you're about to make post or comment.

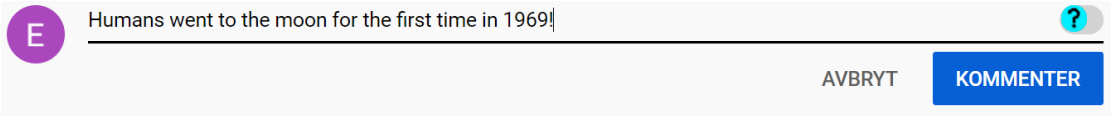

Figure 4.1: How the button looks in the editor field (from YouTube comment section)

4.2 Checking claims

When you type something in the editor field and it exceeds 10 characters, there is a timer that starts counting to 3 seconds. The timer is reset every time you make an input (as long as no claim is being processed). If the timer is not interrupted, the fact check button will send the text as a claim to the API. You will know that this happens when a

loading animation starts. This is made as a GIF with three dots moving up and down, as shown below. The user is free to choose whether they want to wait the 3 seconds before it starts checking, or if they want to instantly check the claim by clicking the question mark button manually.

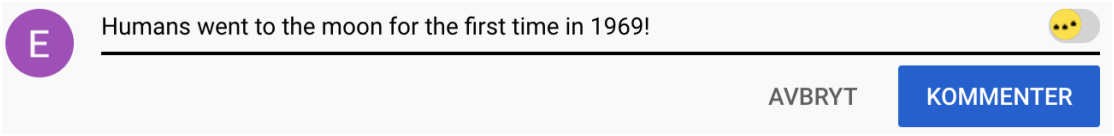

Figure 4.2: Loading animation after 3 seconds of inactivity

After it's done loading, there are four possibilities. The icon will turn red with a cross if the claim is false, or green with a check mark if it's true. There are also two cases for similar claims, but more on those later. In the case below, the overlay confirms the proposed claim to be true. Thus turning into a green check button like in Figure 4.3.

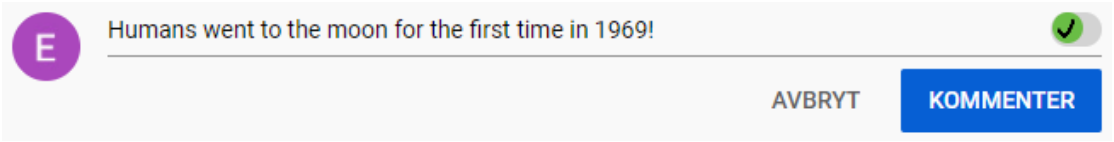

Figure 4.3: Green check button as a result of the fact being "true"

When hovering over the button, another blue button, "expand", is displayed. Clicking this will bring up a layout which refers to the Popup for more information as to why it got that result. An example when hovering over the button is displayed in Figure 4.4. Look to Figure 4.5 to see what the "expand" interface looks like.

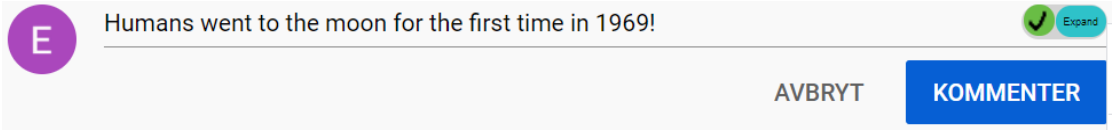

Figure 4.4: How the check button looks when hovering over it

Figure 4.5: Interface when clicking on the expand button

4.3 Alias for button

The button can also handle similarities and appears different when a similarity arises. As seen in figure 4.6 below. There is an exclamation mark surrounded by the color of the result the similar claim received.

Figure 4.6: Button when hovering over it

When the expand button is clicked a popup appears, showing the claim you just made and the claim it's similar to. If clicked yes, the window will close and the green check mark will appear. However if clicked no, the window will close and the claim will be handled as a new claim with the loading GIF appearing. Showing that the claim is being processed and will result in either a green check mark or a red x.

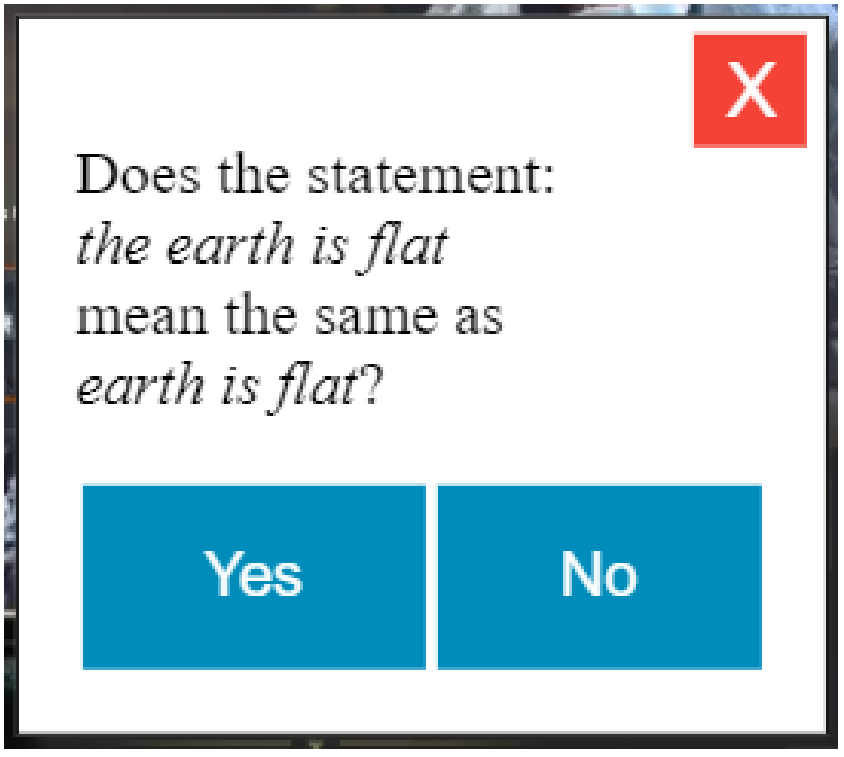

Figure 4.7: Interface when clicking on the expand button

And as seen in figure 4.1-4.7 it operates as the extension, albeit less information.

4.4 How the button is implemented

The button is created and designed in the *generators.js* file. Having these features.

- Receive claim
- Display result (true or false)
- Handle similar claims
- Popup with more information

For all three currently supported websites the same approach is used. Firstly determining which site the user is currently on and identifying the container which contains the field where one can post. More over these fields differ in their respective selectors. As a result these field's unique selectors have been extracted.

For the button to operate properly it needs to be inserted at the right time. For this reason a listener is used on the *focusin* event to accurately deduce when a text field is in focus. Next the focused in text field's selector is compared to the extracted selectors, and if they are the same a button will be spawned in. So long as there doesn't exist one already.

4.4.1 Shadow DOM

To create this button without common class names and IDs interfering with the button, it was attached to a shadow DOM. So regardless of which page its being rendered on its appearance wouldn't change.

```
buttonDiv = document.createElement("fact-analysis-button");
let shadowRoot = buttonDiv.attachShadow({ mode: "open" });
```

Listing 4.1: Creating a shadow root

Here the check button gets attached to a shadow root. the parameter "open" refers to whether you can access the shadow dom using JavaScript written in the main page context.

Chapter 5

Implementations and Experiments

5.1 Measurements

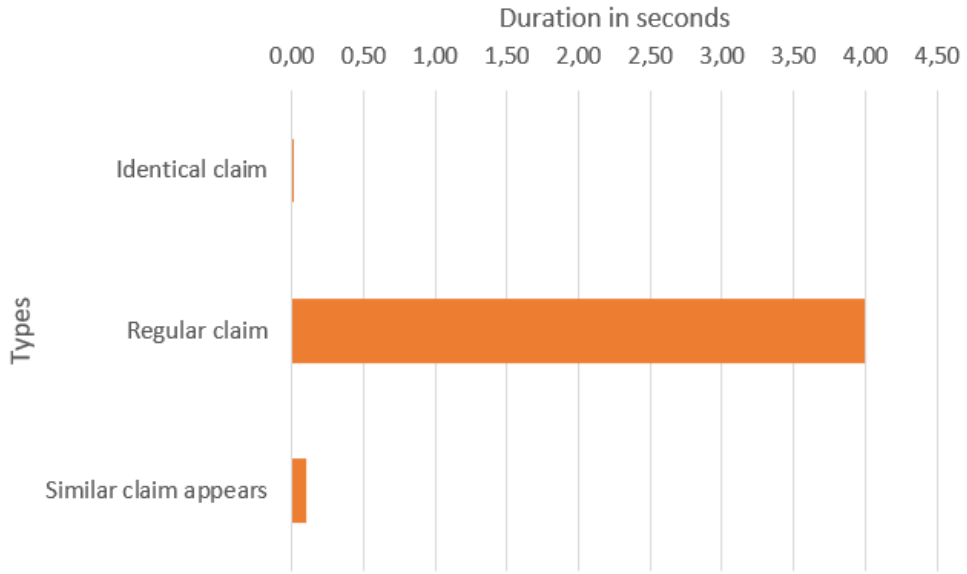

Figure 5.1: Measuring time from start to finish

Figure 5.1 shows a representation of the execution time, from the send claim button is pressed to the result is retrieved, of the three main cases for claims getting checked. Firstly if a claim is identical to an existing claim, it gets sent back within a few milliseconds. Secondly if a similar claim appears, and the cosine similarity check is triggered, the similar claim options will most often appear within a hundred milliseconds. Both these save time compared to making a new claim. This is a result of the API request itself. If the time spent waiting for the request to come back would have been neglected, claims could have been handled in a similar time frame as the similar claims.

5.2 How the front-end and back-end communicates

Communication between the front-end (Extension popup) and back-end (background-script and content-script) is handled through the chrome message and storage APIs.

The front-end is actively listening to changes made in the utilized local storage system. This means that if a new claim is made, an error is received, or a claim gets edited, that being a similar claim is added to it, the extension will respond with rendering the appropriate view.

The back-end on the other hand mostly listens to messages sent by the content-script itself or the extension popup. These messages usually contain an instruction named appropriately to their functions, and some data attached to itself.

5.3 Cosine similarity

In the beginning string comparison were performed on new claims against our storage to see if the claim already exists.

```
for(let i in allData) {
    if(allData[i].body.claim === request.data.toString()) {
        exists = true
        sendResponse({results: allData[i]})
        }
    }
```

Listing 5.1: "Direct string comparison"

However, upon the realization that essentially almost identical claims ended up being sent as new claims to the API. As a consequence of the claim being slightly different than the claim in storage, but in essence meant the same. For example "the earth is flat" and "earth is flat" will not pass a string comparison and essentially create an unnecessary API call. This lead to the implementation of cosine similarity. The cosine similarity can be found using this formula:

$$cosine(M, N) = \frac{M \cdot N}{||M|| \cdot ||N||}$$

5.3.1 Cosine similarity code

To compute the cosine similarity a set of functions was implemented. First thing is to split up the string and map the frequency of each word, and return an object with the mapping.

```
function mappingWordCount(str) {
    let words = str.split(' ');
    let wordCount = {};
    words.forEach((w) => {
        wordCount[w] = (wordCount[w] || 0) + 1;
    });
    return wordCount;
}
```

Listing 5.2: "mappingWordCount"

After that, create a dictionary of all the words that are present in both strings.

```
function addingWordsToDictionary(mappingWordCount, dict) {
    for (let key in mappingWordCount) {
        dict[key] = true;
    }
}
```

Listing 5.3: "addingWordsToDictionary"

Then change the dictionary into a vector. The dimension of the vectors will depend on the number of words there are in the dictionary

```
function mapToVector(map, dict) {
    let wordCountVector = [];
    for (let term in dict) {
        wordCountVector.push(map[term] || 0);
    }
    return wordCountVector;
}
```

Listing 5.4: "mapToVector"

Now that the string has been turned into a vector, it can be inserted into the formula.

```
function dotProduct(vecA, vecB) {
    let product = 0;
    for (let i = 0; i < vecA.length; i++) {
        product += vecA[i] * vecB[i];
    }
    return product;
}
```

Listing 5.5: "dotProduct"

```
function magnitude(vec) {
    let sum = 0;
    for (let i = 0; i < vec.length; i++) {
        sum += vec[i] * vec[i];
    }
    return Math.sqrt(sum);
}
```

Listing 5.6: "magnitude"

```
function cosineSimilarity(vecA, vecB) {
    return dotProduct(vecA, vecB) / (magnitude(vecA) * magnitude(vecB));
}
```

Listing 5.7: "cosineSimilarity"

Lastly the above functions are nested into one function to actually calculate the cosine similarity of two claims.

```
function cosineSimiliarityToText(claimA, claimB) {
    const wordCountA = mappingWordCount(claimA);
    const wordCountB = mappingWordCount(claimB);
    let dict = {};
    addingWordsToDictionary(wordCountA, dict);
    addingWordsToDictionary(wordCountB, dict);
    const vectorA = mapToVector(wordCountA, dict);
    const vectorB = mapToVector(wordCountB, dict);
    return cosineSimilarity(vectorA, vectorB);
}
```

Listing 5.8: cosineSimiliarityToText

While this did solve our problem, a new problem arose. For example "the earth is flat" and "the earth is not flat" mean two different things, but still score high on the cosine similarity. The solution was then to create an alternate state in both the extension popup, and the overlay button. In this state, the user gets the choice of sending the claim as a new separate claim, or adding it to an existing claim as a "related claim".

5.3.2 Cosine similarity execution time

Considering the code for the cosine similarity, it can be concluded that the big O notation is O(n), with n being the amount of unique words in the two compared claims. In reality this does not effect the total execution time of the claim checking process, since the average length of sentences typically is within the 15-25 words range

By measuring the time of the cosine similarity check in the extension, it was discovered that the cosine similarity check averaged around a fraction of a millisecond per claim.

5.4 Scoring

Scoring the claims is handled by fact checking API. This API provides the user a bunch of different values, including a final score, per article scoring and a final prediction. The final prediction is based on the final score, where if the score is 0.5 and larger it is 1, else it is 0. Due to the Boolean nature of this final prediction method, a margin of uncertainty has been added in the extensions implementation. This margin is configurable by the user in the settings page. By default 0.4 has been set as the boundary for false claims, while 0.6 has been set as the true claim margin. If the score of a claim is between these two margins, it gets considered an uncertain situation, and the decision is left to the user to determine whether the claim is true or false.

Chapter 6

Discussion and Future Directions

6.1 Conclusion

The goal of making a chrome extension that allows for quick and simple fact checking was in the end reached. Although we did not integrate with text editors like Google Docs or Microsoft word, the three websites we did integrate with are major social media platforms, susceptible to fake news. Nevertheless, since the plugin portion of the application works on any web page (with a few exceptions like chrome://*), we consider the usability and potential reach of the extension to be well within satisfaction.

6.2 User feedback

Throughout the development process, some feedback was requested from a group of test users. This feedback ranged from thoughts on the UI and functionality, to usefulness and the design. These pieces of feedback turned out to be very helpful, and lead to a redesign of the UI.

6.3 Challenges

Through out the project a number of different issues and challenges were discovered, and thus some workarounds and clever solutions had to be implemented.

Locating text input fields on different websites turned out to be a challenge in the beginning.

Deciding on when to render the check button took some iterations to improve. Starting out it appeared when the user clicked on the site, which turned into issues where it wouldn't render since the page hadn't loaded the text field yet. This eventually developed into it rendering on focus, with a check if the button already exists.

Where to attach the button on the supported sites also gave us some trouble, but taking inspiration from the Grammarly chrome extension[16], we decided to attach to the parent element of the text fields.

Trying to make the Popup open directly from the website was a wanted feature, but turned out to be not directly possible in stable builds of the google chrome browser. This functionality apparently is only available on development and canary builds of chrome.

Rendering the button once per page turned out to be a problem when we started implementing comment field support, most noticeably within Facebook, and YouTube. Since we only rendered it once per page, it would only render within the first field you commented on. Therefore implementing it to render once per text field turned out to be more appropriate.

Figuring out a way to implement automatic checking turned out to be a challenge too, where a couple of iterations from checking on key up, to checking on space characters, to eventually ending up on a 3 second timer.

6.4 Future Directions

Improve the storage solution being used, eventually add support for sync-storage between devices.

Make our own claim checking REST server, with possibility to cache previously checked claims for a determined duration

Implement full screen view of application, with better visibility over claims, settings, etc.

Simplify messaging system, removing the current dependency of the content script layer for the popup to work on all sites, including the chrome about page, chrome settings page, and local file pages.

Adding support for graphical word processing programs, such as Microsoft word or google docs. With "Grammarly " style highlighting that will underline sentences or claims either red or green based on the result.

Implement a more context aware comparison check between claims, instead of the current cosine similarity.

List of Figures

Appendix A

Contributions

Team Memeber	Area	Notable Specifics
Kevin Ratdal	All	Project management, Git setup Communication between layers Architectural design Back-end and Front-end focused work
Mustafa Hersi	All	Undertook many of the popups core functionality Communication within the extension Code reviews

The source code for the extension can be found at github.com/KevinRatdal/smrtfact[17]

Bibliography

[1] statista. Number of smartphone users worldwide from 2016 to 2023, 2021. URL https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. Retrieved April, 2021.

[2] Oberlo. 10 TIKTOK STATISTICS THAT YOU NEED TO KNOW IN 2021 [INFOGRAPHIC], 2021. URL https://www.oberlo.com/blog/tiktok-statistics. Retrieved April, 2021.

[3] Shalini Talwar, Amandeep Dhir, Dilraj Singh, Gurnam Singh Virk, and Jari Salo. Sharing of fake news on social media: Application of the honeycomb framework and the third-person effect hypothesis. *Journal of Retailing and Consumer Services*, 57:102197, 2020. ISSN 0969-6989. doi: https://doi.org/10.1016/j.jretconser.2020.102197. URL https://www.sciencedirect.com/science/article/pii/S0969698920306433. Retrieved April, 2021.

[4] BBC. What's so bad about fake news?, unknown. URL https://www.bbc.co.uk/bitesize/articles/zjykkmn. Retrieved April, 2021.

[5] Wikipedia. Google Chrome, 2021. URL https://en.wikipedia.org/wiki/Google_Chrome. Retrieved April, 2021.

[6] HTML. HTML, April 9, 2021. URL https://en.wikipedia.org/wiki/HTML. Retrieved May, 2021.

[7] hackreactor. What is JavaScript Used For?, 2018. URL https://www.hackreactor.com/blog/what-is-javascript-used-for. Retrieved April, 2021.

[8] RESTful. RESTfulapi, 2020. URL https://restfulapi.net/. Retrieved May, 2021.

[9] cosine. Cosine Similarity. URL https://www.machinelearningplus.com/nlp/cosine-similarity/. Retrieved May, 2021.

[10] Bjarte Botnevik, Eirik Sakariassen, and Vinay Setty. Brenda: Browser extension for fake news detection. In *Proceedings of the 43rd International ACM SIGIR*

Conference on Research and Development in Information Retrieval, SIGIR '20, page 2117–2120, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380164. doi: 10.1145/3397271.3401396. URL https://doi.org/10.1145/3397271.3401396.

[11] mediabias. Media bias Fact check extension. URL https://chrome.google.com/webstore/detail/media-biasfact-check-exte/ganicjnkcddicfioohdaegodjodcbkkh?hl=no. Retrieved May, 2021.

[12] marketshare. Market share of leading internet browsers in the United States and worldwide as of February 2021. URL https://www.statista.com/statistics/276738/worldwide-and-us-market-share-of-leading-internet-browsers/. Retrieved April, 2021.

[13] JSON. Working with JSON, Apr 27, 2021. URL https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON. Retrieved May, 2021.

[14] .then(). Promise.prototype.then() , May 5, 2021. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then. Retrieved May, 2021.

[15] Userstats. Most popular social networks worldwide as of January 2021, ranked by number of active users, Feb 9, 2021. URL https://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/. Retrieved May, 2021.

[16] Grammarly. Grammarly, 2021. URL https://www.grammarly.com/. Retrieved April, 2021.

[17] GitHub. smrtfact - smart-fact-checker , May 28, 2021. URL https://github.com/KevinRatdal/smrtfact. Retrieved May, 2021.