Filipe Colla David

A WebApp Graphical Interface for Reliability Detection on Social Media

Curricular Internship Report

Tutor: Álvaro Pedro de Barros Borges Reis Figueira

Co-Tutor: Nuno Ricardo Pinheiro da Silva Guimarães

Department of Computer Science

Faculty of Science of University of Porto

June 2022

Content

1 Introduction

Social media has become a structural pillar of our society with the number of users continuing to grow every day. Social media allows for every user to share their lives, ideas, and thoughts globally at a touch of a button, reaching thousands or even millions other users. Almost every business, news outlet, and government institution have a social media presence and it has become a standard practice to share news with the public through these online platforms. With this ease of information sharing, the term "fake news" became inextricably linked to social media, as the latter served as a platform for the spread of fake news. The term fake news represents various misleading information, from sharing conspiracy theories to influencing voters' perception on political views. For example, at the 2016 United States presidential election it has been proven that fake news influenced some of the voters into a certain political view[1]. This continues to be an issue in social media and it's not limited to politics, for instance, more recently with the rise of the Covid-19 pandemic, there was also a rise in Covid related misinformation [2]. It has become clear how social media and fake news can have a negative impact in our society, as people cannot rely on everything they read on these platforms. Fact-checking became a standard solution. However, most people can't be bothered to verify the reliability of the content as it can be very time-consuming and sometimes exhausting, as a huge amount of information needs to be filtered. Moreover, due to the large amounts of unreliable content on the internet[3], this solution can lead to wrong results if one does not have the right tools or knowledge, leading to confirmation biased, meaning they end up confirming their wrong believes even more than when they started. This problem led to the appearance of platforms that provide information on the reliability of the content. These platforms are mainly divided into systems that verify the content manually, without the assistance of machine learning algorithms and systems that verify the content using machine learning algorithms. The systems that use machine learning algorithm to verify the content are usually black boxes, as they do not provide any insight on how the result is calculated or give hints that help the user understand what makes the content reliable or unreliable. These types of systems are already available to the public, they vary depending on the content being verified. For instance, one can check whether a certain user on social media is a real person or a bot or classify the veracity of a statement. The results in these systems are usually a score or a simple classification, regarding the reliability of the content. However, these systems, don't have any graphical interface with extra information regarding the reliability of the content being analyzed [4].

This internship takes the results from a series of experimental studies regarding the reliability of content in social media conducted in [5] and incorporates them on a Web application (WebApp). This WebApp serves the purpose of providing the user with tools to analyze the reliability of a tweet or a small text while also providing some hints that may aid in the interpretation of the result outputted. These hints come in the form of graphical visualizations that represent several features extracted from the text. To achieve this goal, this internship was divided into three main parts: 1) study of the different technologies that could be used in this project, 2) study of different features that could be extracted from the text and how to represent them graphically and 3) the implementation of the WebApp. The development of this WebApp also resulted in the writing and submission of a research article.

2 State of the Art

Fake News/Unreliable content detection approaches presented in the state of the art, either use machine learning algorithms or rely on humans to determine the reliability of the content.

As stated in [4] approaches that rely on machine learning algorithms are usually divided between verifying if the content being produced by real humans or bots, or whether the content is reliable or not. The state of the art for verifying the authenticity of the content being produced is either to detect/classify a user as being a real person or to analyze how bots are involved with certain topics being discussed online. The systems for verifying if a user is real normally analyze an account and, based on certain features return a score that determine whether the account is real or not. An example of this system can be found in [6], that given a twitter handle, returns a score that determines the likelihood of that user being a bot. The systems for analyzing the presence of bots in certain topic usually return a network composed of accounts and their connection with the addition some metrics that give some insight on the influence of bots in a certain subject. An example of this system can be found in [7], this system allows the user to search for a certain topic and returns a graph with information that can show the influence of bots in the neighborhood of users that are debating the topic. The state of the art for analyzing the reliability of content using machine learning are usually simple interfaces that return a text label with the reliability of the content submitted. An example of this interface can be found in [8]. This system allows users to submit a small text for analysis with the result been whether the text is reliable or not. These systems are great tools for analyzing online content as they allow for a fast and reliable verification. However, due to the lack of explainability on how these algorithms predict certain results the user is left with little or no information about the result.

Solutions that analyze the reliability of content without the use of machine learning vary depending on the content needed for verification. For instance, for website reliability, as stated in [9], they are usually browser add-ons/extensions that flag different content in social media in different categories such as clickbait, bias, conspiracy theory and junk science. An example of this system is the bs-detector, that uses curated list of dubious websites to make its evaluation. For verifying different types of statements, whether it is news headlines or something a person said (either online or not), there are several websites that take these statements and verify them manually. This is usually done by journalists/experts that investigate different sources and use them to verify the reliability of the statement. The result is then shown to the user with the several sources that support the result. An example of these can be seen on websites such as Snopes[10] or Politifact[11]. There's also a Portuguese website that is highly regarded upon the community Poligrafo[12]. For instance, this system classifies the content as True, True but …, Imprecise, Out of Context, Manipulated, False or Pepper on the Tongue. These systems give a very detailed explanation on the result, and, in contrast with the systems that use machine learning, they are much more detailed and based on verified sources. However, the user is limited to content previously verified, not allowing for content to be analyzed and verified on demand. In addition, the process of verifying this content can be very time-consuming since an investigation is needed for validating the different statements.

The work conducted in this internship proposes an innovative way of verifying content, as it tries to solve the issues related with both methods mentioned above, allowing the user to verify, on demand, any written statement using machine learning algorithms, while trying to fulfill the need for more information that can support these results. To achieve this goal, the proposed solution is an interface in the form of a WebApp. This WebApp allows the user to submit a short text for analysis with the classification outputted by the system presenting the user with several hints that serve the purpose of increasing the explainability of the classification.

To better understand this system, this WebApp can be divided in several components:

- Backend: the model, the features that will be use as the additional information to the model's result, the connection to the TwitterAPI, and endpoints that connect the backend with frontend.
- Frontend: the web interface that will allow the user to check for the reliability of a certain content and see the different visualizations for the features extracted.

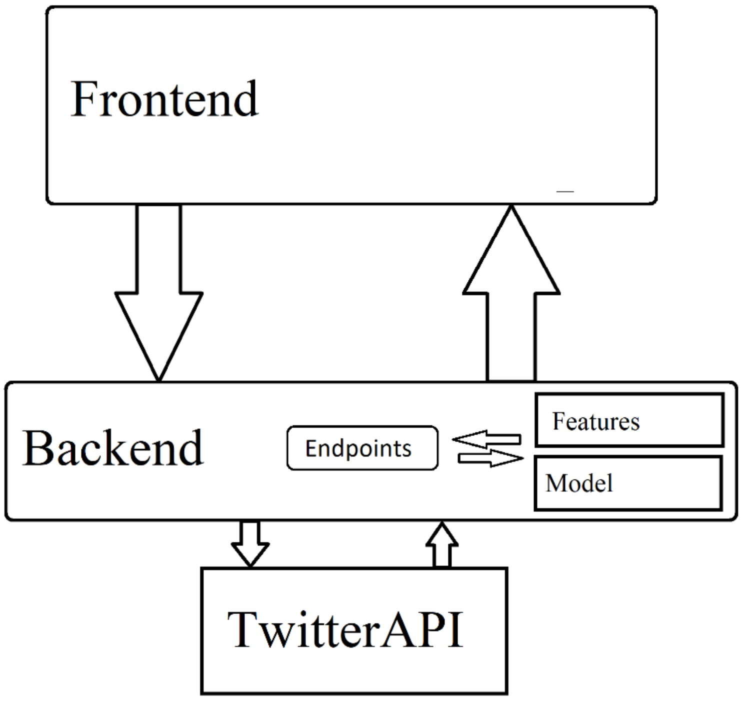


Figure 1 - Structure of the WebApp

2.1 Backend

As mentioned in the section above, the backend can be divided in:

- **Twitter API**: used to retrieve, the data relevant for the analyses, and some features that are used as hints for the analyses of the result. The text content of the tweet and some of its statistics like the reply count, likes and retweets. Some of the author data, like the twitter handler, the twitter username the verified status and URL of the profile picture.
- **Features**: this component consists of all the functions that are used to extract the all the features that are used for the graphical interface in the frontend.
- **Model**: this component contains the function returns the machine learning model prediction regarding text classification
- **Endpoints**: this component contains the different endpoints that connect the components above with the frontend.

The backend is also linked with a database of previously classified tweets.

To achieve this implementation, an extensive study of the different technologies was conducted to determine what would be best fit for the project. As the main existing infrastructure of this project was implemented in Python, to avoid any extra steps when linking the different components of this project, the options were filtered to technologies implemented in Python or with Python support. The technologies considered for this project were, Flask[13], Django[14] and FastAPI[15]. All three were suitable candidates for this project. However, FastAPI is regarded as the fastest one of the three[16], creates automatic documentation, allows for a very intuitive infrastructure for testing, and has less of a learning curve when compared to the other three technologies. Thus, we chose this framework for this project.

FastAPI was used to create the communication between the frontend and the backend. This framework allows for the implementation of very customizable endpoints that use *pydantic* models[17] to build their response. It also allows for the validation of almost all Python types[18]. As for the link between the backend and the TwitterAPI, a developer account was created[19]. To access all the different tweet features in Python, Tweepy[20] was used. Tweepy is a python library that simplifies the access to the TwitterAPI.

2.2 Frontend

The frontend component of this project is the interface that allows the user to input a small text or tweet, receive the reliability classification associated to it and visualize the different features extracted.

After a study of the different technologies available, the initial plan was to build this interface from the server-side, meaning the HTML layout would be generated in the different access-points and injected with data and the different visualizations to fulfill the different requests from the client. This would be done using just HTML, JavaScript and Jinja2[21]. Jinja2 is a fast and extensible templating engine that allows to write special placeholders in the template, allowing to inject code similar to Python. However, after some testing, structuring the project in such a way would be a challenging task. Therefore, it was built using ReactJS[22] with the addition of other libraries for the graphical representation of features. ReactJS is a JavaScript library maintained by Facebook frequently used for single-page web applications. This library is component and state based, meaning that allows the programmer to divide the app in several components, and switch them based on defined states. As for the graphical representation of the features, an extensive study of the different technologies for representing data was made, and after some testing the following open-source JavaScript libraries were selected:

- **ChartJS**[23]: is a library for plotting charts, this library was used for the radar plot and doughnut plot;
- **React-gauge-chart**[24] is a library for creating gauge-meter representations, this type of representation was used when representing the model result and the sentiment analysis.
- **Wordcloud**[25]: is a library for creating custom word clouds, was used in keywords and empath.

For design and the webpage styling, Semantic UI[26] was tested. However, when making the change to ReactJS, the use of this library became less relevant, as React-Bootstrap[27], the technology chosen, apart from having a unique modern styling, with a lot of features that allowed customizable interface, allows for a seamless integration with ReactJS, as all of its styled elements are implemented as ReactJS components.

2.3 Use of the App

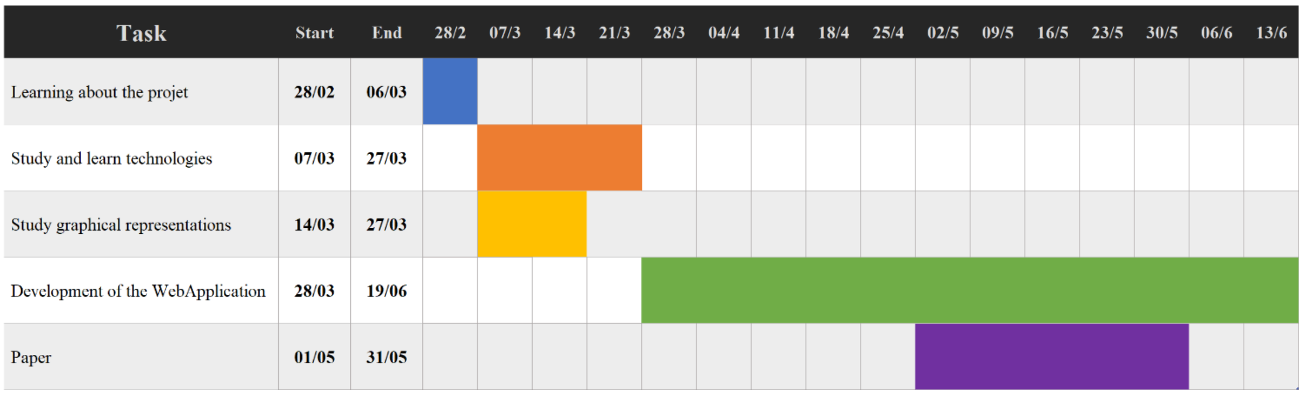
With the implementation of the WebApp using the technologies mentioned above, the user must be able capable of receiving a string input, either a simple text or a tweet id in the form of a URL. In case the input comes in the form of a tweet id, in addition to the text part of the tweet, the different features of the tweet, such as the profile pic and the verified status of the author, the different tweet statistics, such as the like, reply and retweet/quote count, will be shown to the user. The user will also be able to see the result of the machine learning model regarding the reliability of the text submitted for analysis, and other features extracted from text:

- The sentiment of the text – whether it has a positive or a negative sentiment.
- The empath categories identified in the text.
- The emotions identified in the text, such as joy, surprise, sadness, anger, fear and disgust.
- The keywords extracted from the text
- The Entities present in the text, can be entities related to persons, organizations, time, locations, quantities, and groups.
- Some text statistics such as the uppercase percentage or the text score.
- What and how many emojis/emoticons are present in the text.

3 Description of the internship

3.1 Chronological Description

This part of the report will describe, in chronological order, the different stages of this internship and the different goals achieved in these stages. This internship was divided into several main tasks, as shown on Figure 2. In the following subsections, a summary of each task is presented.



Task	Start	End	28/2	07/3	14/3	21/3	28/3	04/4	11/4	18/4	25/4	02/5	09/5	16/5	23/5	30/5	06/6	13/6
Learning about the projet	28/02	06/03	🟦															
Study and learn technologies	07/03	27/03		🟧	🟧	🟧												
Study graphical representations	14/03	27/03		🟨	🟨													
Development of the WebApplication	28/03	19/06					🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Paper	01/05	31/05										🟪	🟪	🟪	🟪	🟪		

Figure 2 - The tasks in this project

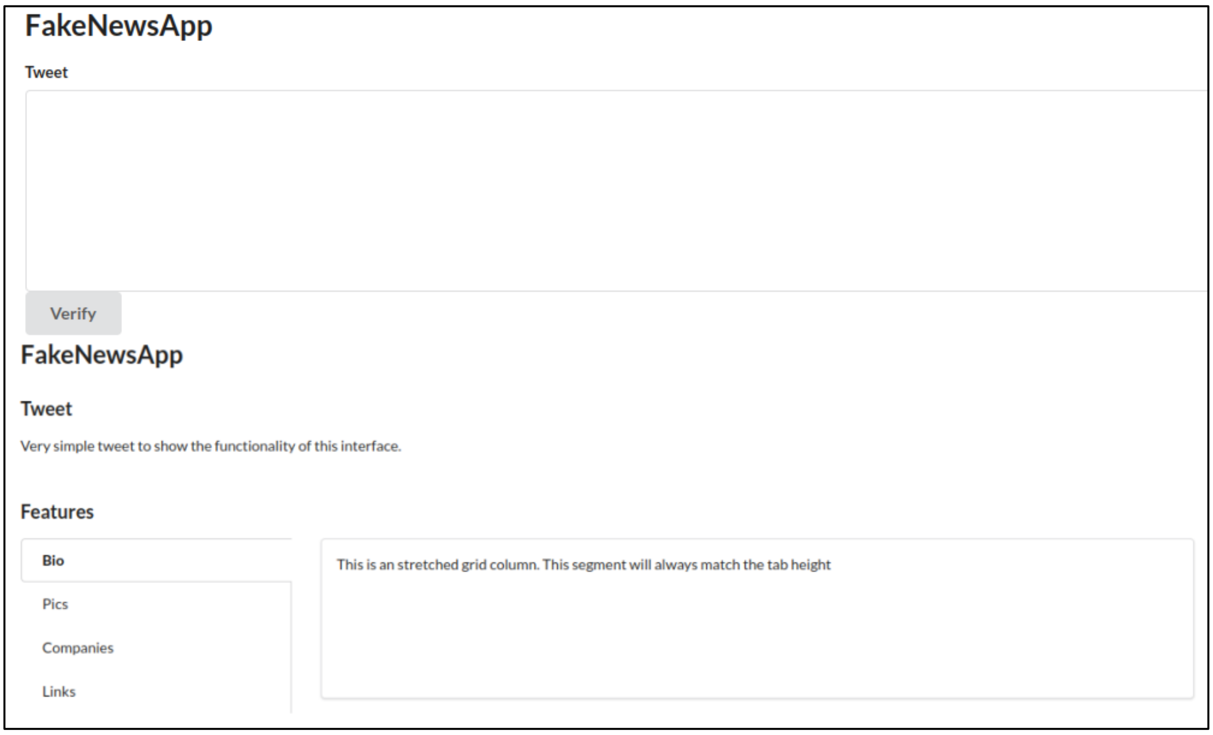
3.1.1 Learning about the project

In the first week of the internship a meeting was arranged with my tutors to briefly introduce the project, discuss its current state, and what would be my contribution to it. Then a set of objectives were defined. Throughout this week, a study of the existing structure was conducted, in order to get comfortable with the concept and domain of this project.

3.1.2 Study and Learn Technologies

My initial plan for this project was to have the backend part generate all the frontend components. This choice allowed for a more contained system since 1) all the computations occurred in one place and 2) easier communications from the client to the backend and inside the backend itself, from the different access points and the existing interface, as it didn’t require to convert data for it to be compatible with other technologies. With that plan in mind, the study of the available technologies was focused on backend solutions with some extra libraries to generate the layout for the frontend. An extensive study on the available technologies to serve this goal was made and FastAPI was chosen to implement the backend component of this project. By the end of the first week, a basic interface of a text input and a second page with that same content was created.

However, after some further clarification and study of the project needs, it became clear that would be very challenging to create the entire interface in this manner as it would rely on a higher level of Python programming that I was not comfortable nor had the time to achieve during the course of this internship.



Very simple tweet to show the functionality of this interface.

This is an stretched grid column. This segment will always match the tab height

Figure 3 - Early prototype using Backend layout generation

To solve this problem, the project had to be divided into two separate components, the backend, which would be in Python with the previously chosen FastAPI, and the frontend. After an extensive study of the different technologies available for the implementation of the frontend component, ReactJS was chosen. As ReactJS is based on JavaScript, a language that I'm very acquainted with, the learning curve required to understand the different concepts behind this technology would be less steep than any other. Moreover, ReactJS, as of the writing of this report, is one of the more requested technologies in the web development market, serving as a good addition to my skill set as a programmer.

In the following week, I started to study this technology and consolidate my knowledge. This task was possible with the help of the documentation and the tutorials made available by the React developers[28].

3.1.3 Study Graphical Representations

Alongside the study of the technologies suitable for this project, a study of different graphical representations was made to decide how the features extracted from the text would be represented in the interface. This influenced the choices of the technologies suitable for this project as some candidates would represent challenges on the implementation of some of the visualizations required. The following was retrieved from a shared document with my tutors where we choose the different visualizations for the different features extracted.

As shown in *Table 1*, each feature has a Python function from the existing infrastructure of this project. The visualizations were chosen based on the output of this functions and their respective description. Then, after some debating and some testing on the different choices, we'd come to an agreement on which visualization would be best for each feature. For instance, the visualization for the Psycholinguistic Features/Categories initially was a WordCloud. However, this visualization had some issues regarding the possible user interpretation and was changed to a radar plot. This was not the only modification, as will be shown further in this report.

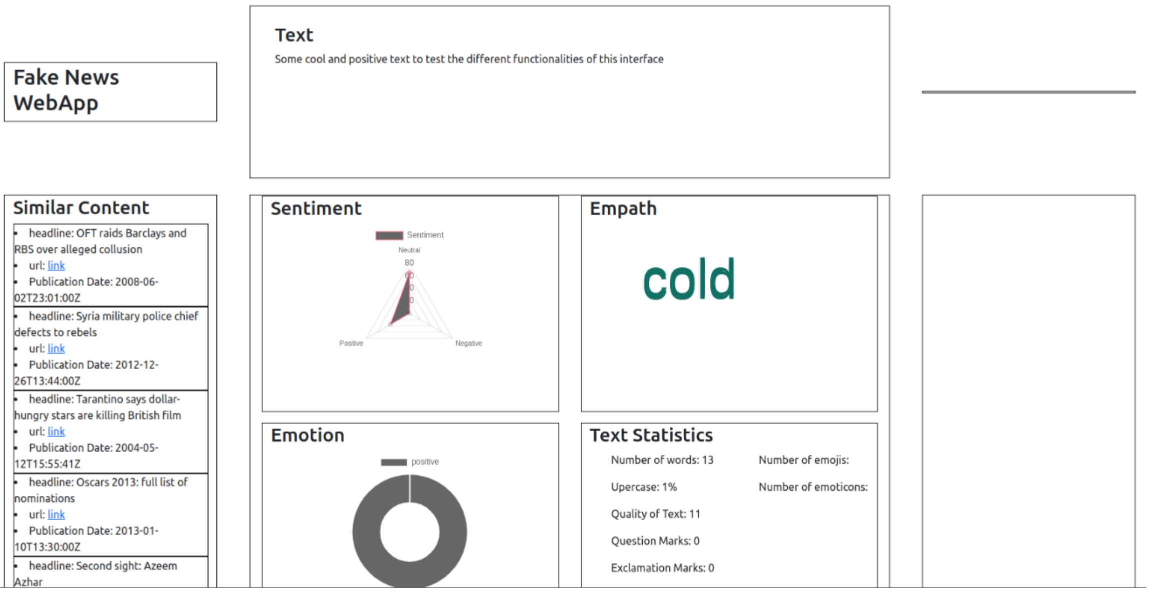
Table 1 - Initial attribution of graphical representations for some of the features used.

Function	Feature	Input	Output	Description	Visualization
`similarContent`	Similar Content	Clean Text	List of dicts with headline, publication date and news url	Uses the Guardian API to extract similar reliable news to the text inputted	List to the side
`getKeywords`	Keywords	Clean Text	Dict with keyword and scores	Uses Yake to extract the most relevant keywords from the text	Annotation/Highlight text (https://genius.com/)
`getEmojisTweets`	Emojis	Text	Dict with emojis frequency	Outputs the emojis identified in text	Text Statistics
`getEmoticonsTweets`	Emoticons	Text	Dict with emoticons frequency	Outputs the emoticons identified in the text	Text Statistics
`uppercasePercDescription`	Uppercase percentage	Clean Text/ text	Dict with uppercase mean	Outputs the percentage of uppercase letters in text	Text Statistics
`GetEntities`	Entities	text	List of dicts with entity and category	Uses NLTK to identify basic entities in the text	Annotation/Highlight text (https://genius.com/) Alternate between entities and POSTags. Highlight words according to class in the text. List them in a table (number of columns = number of categories)
`getPOSTags`	Parts of Speech	text	Dict with the number of POS identified for each category	Uses NLTK to identify basic POS in the text	
`getTweetsRead`	Readability	text	Dict with the readability score	Identifies the readability score of the text. The higher the score the more well written is the tweet	Text Statistics -> use a table
`GetEmpathTweets`	Psycholinguistic Features/ Categories	text	Dict with the Psycholinguistic features identified	Uses Empath to identify psycholinguistic categories for each text	Word Cloud -> Radar plot
`getSentiment`	Sentiment	text	Dict with the sentiment scores for each category	Uses Vader to identify the positive, negative, and neutral sentiment of text	Radar Chart -> meter
`GetEmotion`	Emotion	text	Dict with the emotion scores for each category	Uses NRC to identify the 8 basic emotions in the text	doughnut
`numberofQuestionMarks`	Number of question marks in text	text	Dict with number of question marks	Count the number of question marks	Text Statistics -> use a cell in a table
`numberofExclamationMarks`	Number of exclamation marks in text	text	Dict with number of exclamation marks	Count the number of exclamation marks	Text Statistics -> use a cell in a table
`countWords`	Number of Words	text	Dict with number of words	Count the number of words	Text Statistics -> use a cell in a table

3.1.4 Development of the WebApp

The development of the WebApp, as stated in the State of The Art (2), consisted in the implementation of the backend and the frontend. The development began once all the features necessary were discussed and settled. The initial idea for the WebApp only considered the graphical representations shown in *Table 1*. However, as mentioned in the section above (3.1.3), after some testing, some of the representations changed, and some extra features were added. These changes will be further explained in the results section (4).

In the initial stages of this task, the backend and the frontend were developed simultaneously. The early development of this components was very experimental as I was still learning the full range of options that these technologies had to offer. This led to a lot of trial and error since I was trying to figure out the best way to implement some parts. This resulted in an early frontend prototype that had little to no styling, as shown in Figure 4.


Fake News WebApp

Text

Some cool and positive text to test the different functionalities of this interface

Similar Content

- headline: OFT raids Barclays and RBS over alleged collusion
- url: link
- Publication Date: 2008-06-02T23:01:00Z

- headline: Syria military police chief defects to rebels
- url: link
- Publication Date: 2012-12-26T13:44:00Z

- headline: Tarantino says dollar-hungry stars are killing British Film
- url: link
- Publication Date: 2004-05-12T15:55:41Z

- headline: Oscars 2013: full list of nominations
- url: link
- Publication Date: 2013-01-10T13:30:00Z

- headline: Second sight: Azeem Azhar

Sentiment

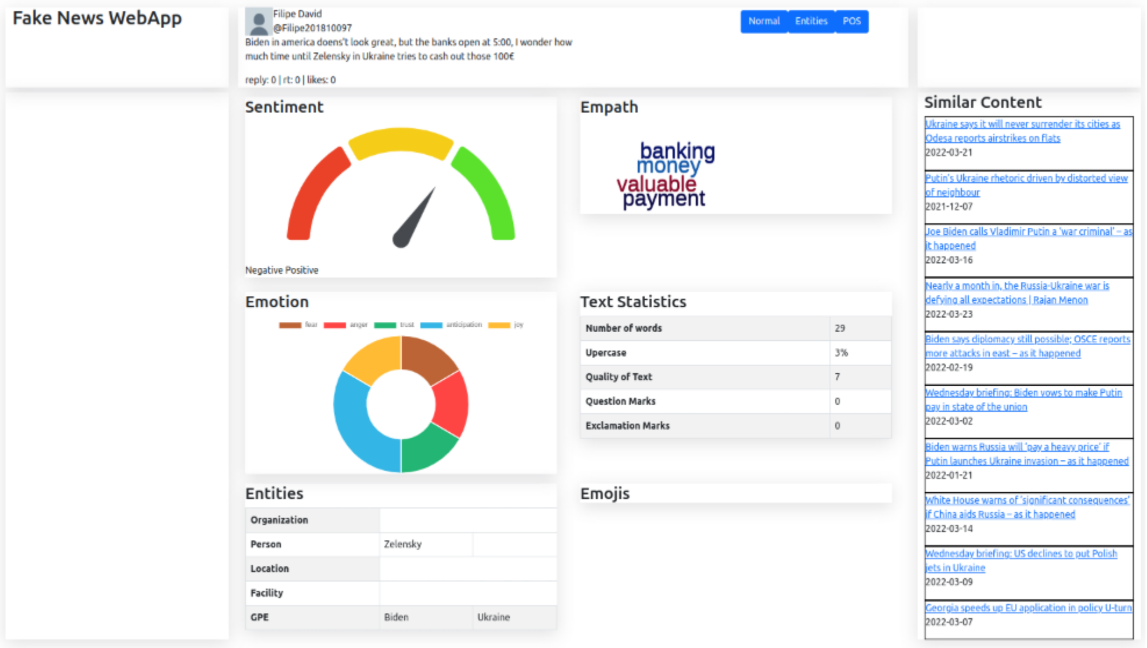
Empath

Emotion

Text Statistics

Number of words: 13
Upercase: 1%
Quality of Text: 11
Question Marks: 0
Exclamation Marks: 0
Number of emojis:
Number of emoticons:

Figure 4 - Early prototype without any styling


Fake News WebApp

Filipe David
@Filipe201810097
Biden in america doens't look great, but the banks open at 5:00, I wonder how much time until Zelensky in Ukraine tries to cash out those 100€
reply: 0 | rt: 0 | likes: 0

Sentiment

Empath

Emotion

Text Statistics

Number of words	29
Upercase	3%
Quality of Text	7
Question Marks	0
Exclamation Marks	0

Entities

Organization		
Person	Zelensky	
Location		
Facility		
GPE	Biden	Ukraine

Emojis

Similar Content

Ukraine says it will never surrender its cities as Odesa reports airstrikes on flats
2022-03-21

Putin's Ukraine rhetoric driven by distorted view of neighbour
2021-12-07

Joe Biden calls Vladimir Putin a 'war criminal' – as it happened
2022-03-16

Nearly a month in, the Russia-Ukraine war is defying all expectations | Rajan Menon
2022-03-23

Biden says diplomacy still possible; OSCE reports more attacks in east – as it happened
2022-02-19

Wednesday briefing: Biden vows to make Putin pay in state of the union
2022-03-02

Biden warns Russia will 'pay a heavy price' if Putin launches Ukraine invasion – as it happened
2022-01-21

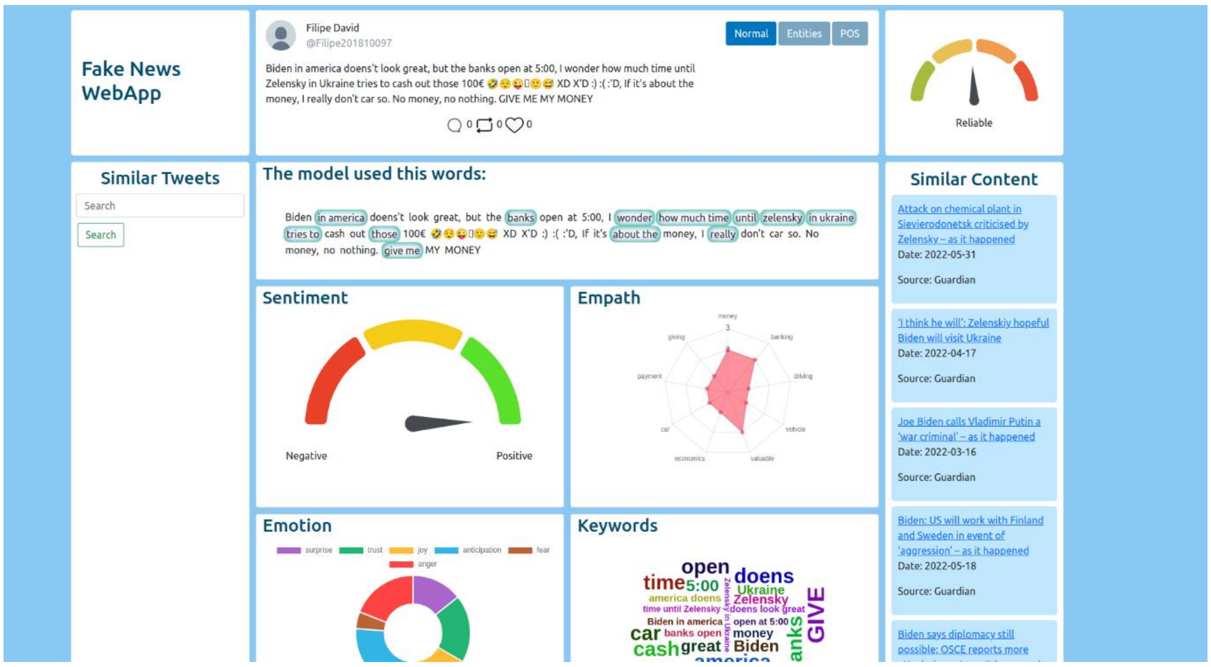
White House warns of 'significant consequences' if China aids Russia – as it happened
2022-03-14

Wednesday briefing: US declines to put Polish jets in Ukraine
2022-03-09

Georgia speeds up EU application in policy U-turn
2022-03-07

Figure 5 - Same prototype as Figure 4 but with styling applied

Once the backend had a final structure that did not need any more implementation (apart from minor improvements and later implementation of new features), the styling of the frontend began and the WebApp began to be more visually appealing. This, however, did not meant that the logical structure of the frontend remained the same until the end of this internship. As I became more confident with React, I would do certain adjustments to the code to improve its structure and readability and fix minor bugs. Some features were also added later such as the possibility of searching for previously classified tweets, new hints like a word cloud that would show the user the keywords extracted from the text, and a new visualization that allowed the user to see what words were used by the word2vec model. These changes can be seen from Figure 4 to Figure 6.



Fake News WebApp

Filipe David
@Filipe201810097

Biden in america doens't look great, but the banks open at 5:00, I wonder how much time until Zelensky in Ukraine tries to cash out those 100€ 🤣😌😜▯🙂😅 XD X'D :) :(:'D, If it's about the money, I really don't car so. No money, no nothing. GIVE ME MY MONEY

The model used this words:

Biden in america doens't look great, but the banks open at 5:00, I wonder how much time until zelensky in ukraine tries to cash out those 100€ 🤣😌😜▯🙂😅 XD X'D :) :(:'D, If it's about the money, I really don't car so. No money, no nothing. give me MY MONEY

Similar Content

Attack on chemical plant in Sievierodonetsk criticised by Zelensky – as it happened
Date: 2022-05-31
Source: Guardian

'I think he will': Zelenskiy hopeful Biden will visit Ukraine
Date: 2022-04-17
Source: Guardian

Joe Biden calls Vladimir Putin a 'war criminal' – as it happened
Date: 2022-03-16
Source: Guardian

Biden: US will work with Finland and Sweden in event of 'aggression' – as it happened
Date: 2022-05-18
Source: Guardian

Biden says diplomacy still possible: OSCE reports more

Figure 6 - Prototype with extra features

3.1.5 Paper

Due to the innovation of this project, a paper was written and submitted to the DatSA'22 – Workshop[29] for the 2022 CENTERIS - International Conference. This paper was a joint effort with my tutors, to describe the entire system that composes the WebApp. It describes the *Unreliable Detection Model*, the *Reliability Analysis Cues* (described in this project as the hints to complement the result of the classification) and how it was all built into the WebApp. My contribution to this paper was based on the description of the WebApp interface and its implementation. This paper, as of the writing of this report, has been submitted and awaits decision.

3.2 Methods

As mentioned in the State of the Art (2), this WebApp follows a standard *fullstack* structure where the backend is separated from the frontend, meaning all the data processing is done on the backend, sent to the frontend and (in some cases) transformed into the graphical visualizations.

The frontend part of the system represents the interface that the user will use for submitting the text for analysis. The interface, as mentioned in the State of The Art (2), was built using ReactJS.

A ReactJS webapp usually consists of several components, and each component can also be further divided into other components [see Diagram 1, Diagram 2, *Diagram 3*].The components are usually defined in different files. However, is not mandatory to do so, as multiple components can be defined in a single file. These components are independent, and they cannot access data defined outside themselves. To transfer data between components, props are used[30]. To have a dynamic page, the components are rendered conditionally[31] using *state*[32] to alternate between different components. The change of state in a ReactJS WebApp is responsibility of the programmer since it is he who defines and/or invokes the different functions that allow the change of state. There are several ways of defining *state* in ReactJS, but in this WebApp the method used was hooks[33]. Hooks were introduced in 2018 as a simpler and more legible way to use *state* in the ReactJS WebApps. The hooks used in this internship were *useState()* and *useEffect()* [34]. To use *useState()*, a new variable is declared as state [], with this declaration coming in the form of a list with two elements. The first element is the name of the variable that holds the state and the second is a function used to set the new value for the state. An example of the use of *useState()* can be found in this WebApp when switching between the forms of input allowed, *TwitterId* or a simple text [Code Snipet 8]. The hook *useEffect()* accepts a function, usually defined as an anonymous function that contains imperative, possible effectful code that runs after the rendering of components. This hooks, if contains effectful code, forces the component to re-render with the changes caused by the function inside *useEffect()*. The *useEffect() function* does the computation needed for a certain component. However, in this project, *useEffect()* and *useState()* are mainly used combined. An example of this can be found when rendering a component that relies on data from the backend. A ternary operator using a state variable as the conditional value is defined to determine which component to render. In this particular case, the state variable describes the data loading state and defines whether the component to render is the loading component (true) or the dashboard component with all the graphical representations (false). This variable is defined using *useState()* and it is initially set to true. Then, using *useEffect()*, an anonymous function is given as parameter to fetch data. When the data is ready this function changes the state variable to false, as the data is no longer loading causing the component to re-render, and changing what component is being rendered.

In the following diagrams (Diagram 1, Diagram 2, *Diagram 3*) we can see how the WebApp components render conditionally depending on state. These components, as writen in ReactJS, are identified by <Component/> with the state induced conditions appearing as the label for the dotted lines. Actions such as, Query, are represented as the label of a full line.

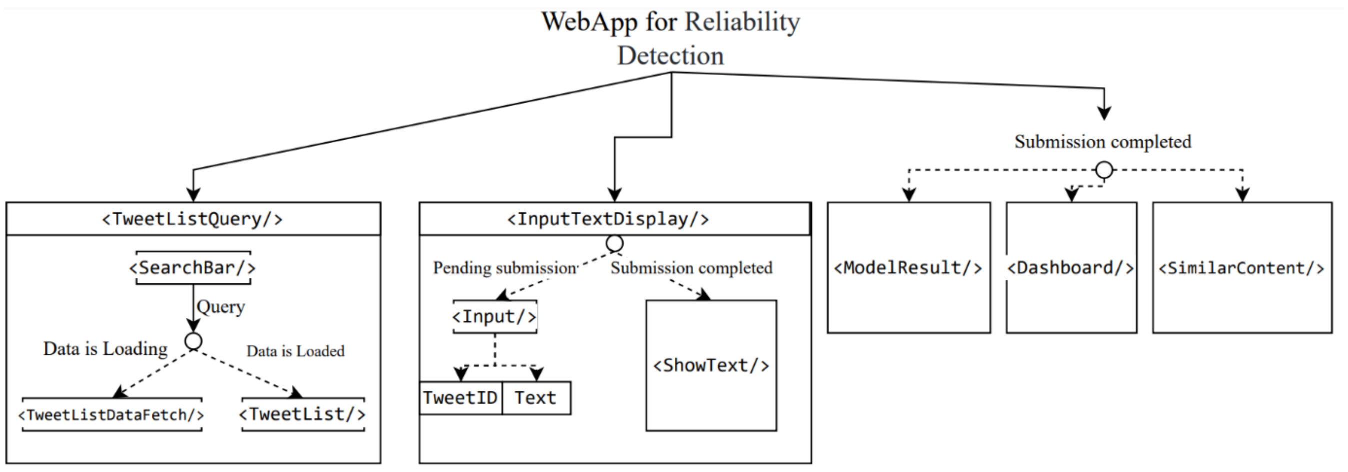


Diagram 1 - WebApp structure

For instance, in the Diagram 1, we can observe how in the component TweetListQuery, initially only the component SearchBar is rendered. Once a query has been made, it triggers the render of TweetListDataFetch, that doesn't change until the data requested has been loaded completely. Once the data is loaded, the component rendered switches from TweetListDataFetch to TweetList, where the data will be represented. Same procedure is followed for the InputTextDisplay component, that at its initial state is rendering the Input component that alternates between the TweetID or the Text component, however, once a submission is made for analyzing the text, the component ShowText (Diagram 2) is rendered. Note that once the submission is completed other components like the ModelResult, Dashboard and SimilarContent start being rendered.

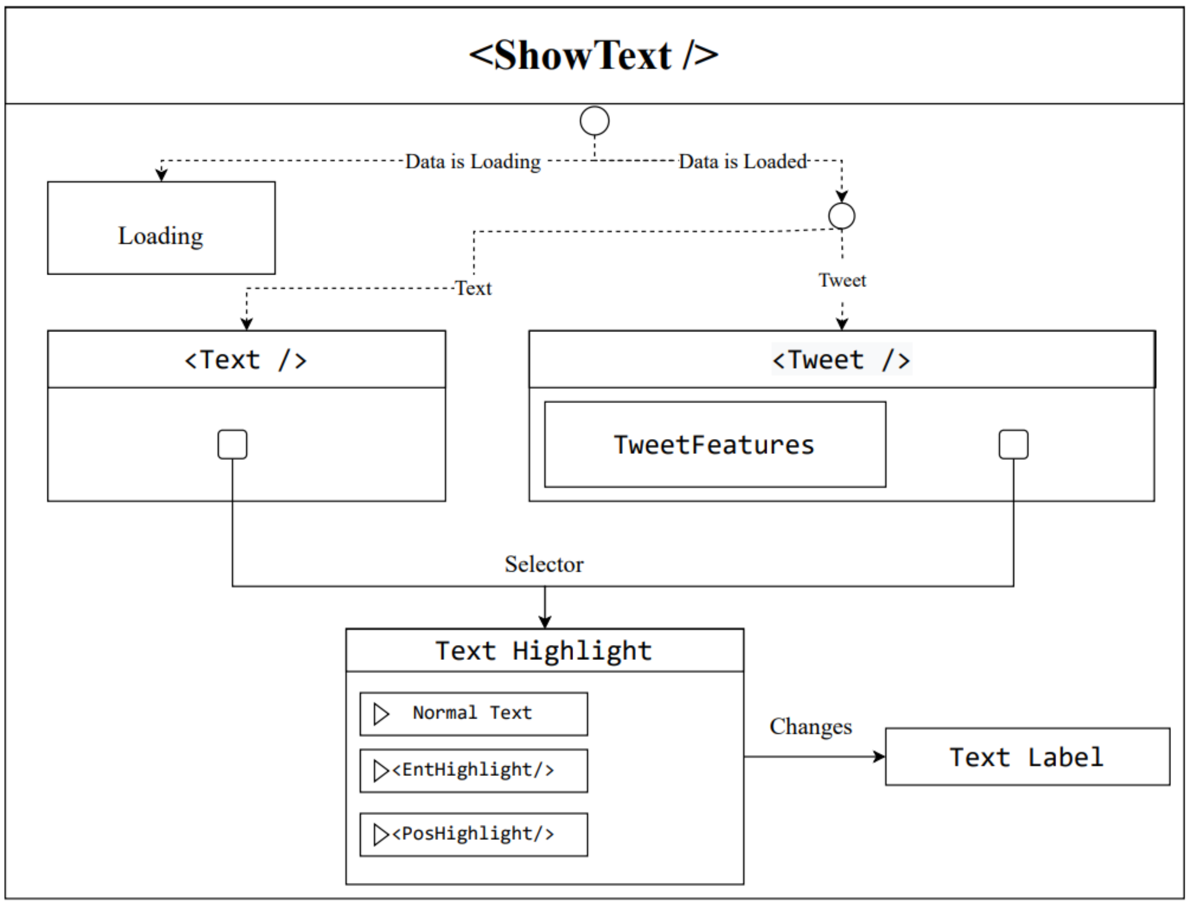


Diagram 2 - ShowText component

In *Diagram 2* we can see another type of use of *useState()*. Instead of using a state to determine whether the data is loading, we can hold different components inside a state. For instance, in this case, the state variable TextLabel alternates between the components Normal Text (in this case is not a component is just the string of the text), <EntHighlight/> and <PosHighlight />, these components highlight different features identified in the text.

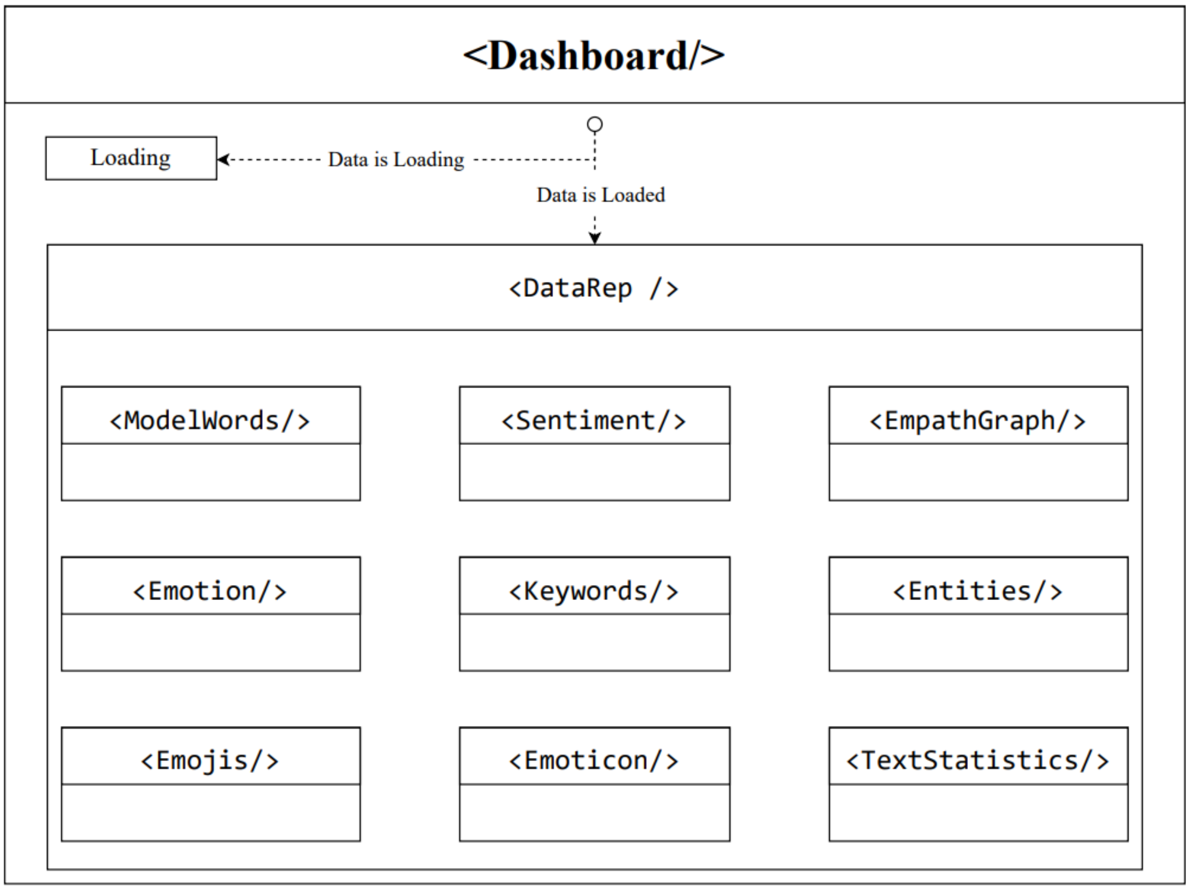


Diagram 3 - Dashboard component

The graphical visualizations used in this WebApp are, in essence, open source ReactJS components developed by other programmers that receive data and configuration values as *props*. Then, they are executed as normal ReactJS components. The data received by these components is the data processed by the backend. However, in some cases, this data needs to be processed once more to fit the configurations set by the graphical visualization's components. Regarding the choice of each individual visualization, we follow the guidelines presented in[35], an effective visual communication must have a clear purpose, show the data clearly and make the message obvious. These visualizations were implemented in accordance with these guidelines and what we believed was the best graphical representation for the data.

The backend component implementation of this project can be divided in three main parts, the model interface, the features interface (this includes the connection with the data base with previously classified tweets), and the different endpoints responsible for the communication between the frontend and the backend. The model interface is where the classification of the text is computed, and the features interface consists of the functions that extract features from the text. The functions used from these interfaces were the following:

- **getModelResult:** this function returns the model prediction regarding of the text analyzed
- **highlightWordsFromWord2Vec:** this function returns a list of words that were in the word2vec vocabulary, and the model used for its prediction. The word2vec vocabulary is the result from a word2vec algorithm that takes a text corpus as input and produces the word vectors as output[36] .
- **getEntities:** this function uses two libraries, NLTK[37] and Stanza[38] to extract the different entities present in the text. These entities can be divided into 9 categories. However, in the WebApp they were mapped as shown in Table 2.
- **getPosTags:** using NLTK, this function extracts the different Parts of Speech Tags (POS) from the text, this includes Adjective, Ad position, Numeral, Adverb, Verb, Conjunction, Particle, Determinant, Noun, Proper Noun.

- **getEmpathTweets:** this function uses Empath, a Python library[39], to identify psycholinguistic categories for each text.
- **getSentiment:** this function uses Vader sentiment rules to identify the positive, negative, and neutral sentiment of text. The result used in the WebApp takes the compound value of the sentiment analyzes and transfers it to range between -1 and 1 to be compatible with the representation used in the frontend.
- **getEmotion:** uses NRC to identify the eight basic emotions in the text, however positive and negative were taken out as they were already being used for the Sentiment analysis.
- **uppercasePercDescription:** returns the uppercase percentage in the text
- **getTweetsRead:** identifies the readability score of the text. The higher the score the more well written is the tweet.
- **numberofQuestionMarks, numberofExclamationMarks, countWords:** returns the number of question marks, exclamation marks and words present in the text.
- **getEmojisTweetsWeb, getEmoticonsTweetsWeb:** extracts the different emojis, emoticons and their respective quantity from the text.
- **getKeywords:** Using YAKE keywords extractor[40], extracts the keywords from the text and a score associated with lower values meaning more relevant keywords.
- **getHighlighting:** returns an HTML block with the text highlighted. This function receives the text to be highlighted and either the word "entities" (for Entities) or "pos" (for POS) to get the respective highlighted text.
- **similarContent:** this function takes a query consisting of the entities extracted from the text and trough the GuardianAPI[41] retrieves the news headlines and dates of publications filtered by that query. The GuardianAPI is a public web service for accessing all the content that the Guardian creates.
- **searchLocalTweets:** this function receives a query as argument and retrieves a selected number of previously classified tweets. These tweets were previously annotated and stored on a database, more information regarding the extraction and annotation of these tweets can be found in[42].

Table 2 - Grouping of recognized entities into entity types

Categories	***Who***	***When***	***Where***	***Quantity***	***Groups***	***Other***
Entities Types	Persons Organizations	Date Time	Geo-Political Entities Locations Events	Money Percentage Cardinal Ordinal	NORP	Other

The server endpoints take the data from the TwitterAPI, the model result, and the functions that extract the different features and sends them to the frontend. Each endpoint uses a different *pydantic* model to build the response required to satisfy the needs of each request.

4 Results

The main result of this internship is a working prototype of the WebApp. For the sake of explainability, the results are divided between backend and frontend.

4.1 Backend – server endpoints

As mentioned in the Methods section of this report, to satisfy every request from the frontend, several endpoints were implemented:

- **Model result** [Code Snipet 1]**:** this endpoint receives the text submitted for classification and returns the classification result. This response consists of a *pydantic* model, with two fields: one for the probability of the text being reliable and one for the probability of being unreliable. Both values are between 0 and 1, with one being the remaining of the other. These values are then used by the frontend in a gauge meter.
- **Words used by the model** [Code Snipet 2]**:** this endpoint receives the text submitted for analysis and uses the function **highlightWordsFromWord2Vec** that returns the words of the tweet that belong to the word2vec vocabulary and were part of the information that the model used to predict the result. The response takes the result from a function in the model interface and converts it to *JSON*. This result consists of a python dictionary of pairs (String, Boolean), where each word used by the model is marked as true, and the rest as false.
- **The features extracted from the text** [Code Snipet 3]**:** this endpoint receives the text submitted for analysis and returns all the features that are represented in the graphical area of the WebApp. This response takes results from the model interface and builds them into a *pydantic* model previously defined[code]. This includes the functions: getEntities, getPosTags, getEmpathTweets, getSentiment, getEmotions, uppercasePercDescription, getTweetsRead, numberofQuestionMarks, numberofExclamationMarks, countWords, getEmojisTweetsWeb, getEmoticonsTweetsWeb and getKeywords.
- **The Text Highlighting** [Code Snipet 4]**:** this endpoint receives the text submitted for analysis and returns a *JSON* defined by a *pydantic* model that consists of four fields: two are HTML blocks that highlights the words extracted as Entities, and POS. This HTML blocks are obtained using the function getHighlighting.
- **Similar Content** [Code Snipet 5]**:** this endpoint receives the text submitted for analysis and returns a list of Guardian headlines and date of information that are related to the text submitted using the function similarContent.
- **Similar Tweets** [Code Snipet 6]: this endpoint receives a query from the user and using the function getSimilarTweets, returns a list of classified tweets related to the query requested by the user.
- **Tweet Features** [Code Snipet 7]: this endpoint is responsible for linking the backend with the TwitterAPI to extract the different features of the tweet submitted for analysis. The response was built on a previously defined *pydantic* model that contains the Twitter handler, the screen name, the profile picture URL, the verified status, the text content and other statistics such as the retweet, quote, like and reply count.

4.2 Frontend – description

This interface can be divided in four different components: input, model result, visualization of reliability analysis cues, and the “Similar Content” and Query component.

4.2.1 The Input

The input part of the system is where the user can start the analyses of the text. This part is divided in two types of input:

a) **a Twitter URL for a specific tweet:** this method of input uses the Twitter API to extract the tweet text as well as the Twitter handle, username and verified status of the author. In addition, other tweet information such as reply, like and retweet count is also extracted.

b) **a textbox where the user can write a small text:** this method of input accepts a small text written by the user (could be a headline or a claim).

After the content has been submitted for analysis, either as type a) or type b), the user will be shown a dashboard with the model's result, several different widgets the serve as reliability analysis cues, and a list of Guardian headlines related to content submitted for analysis.

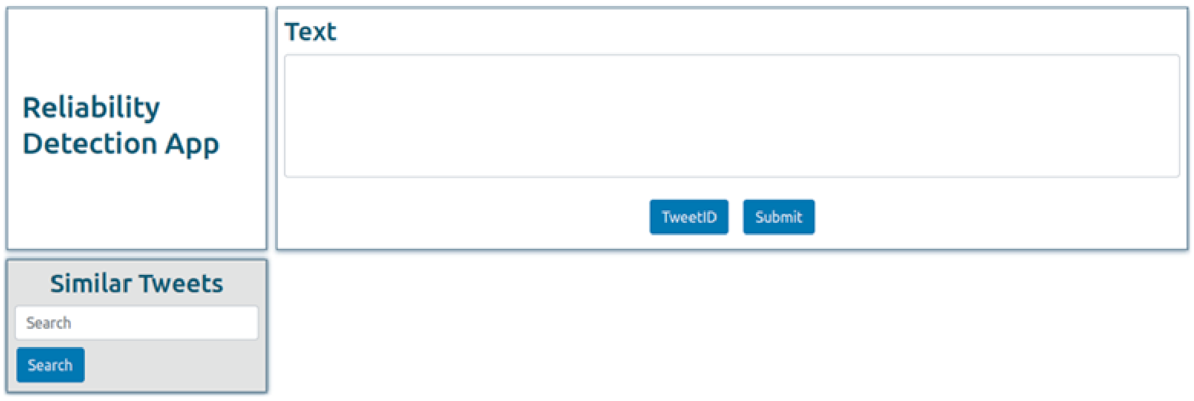
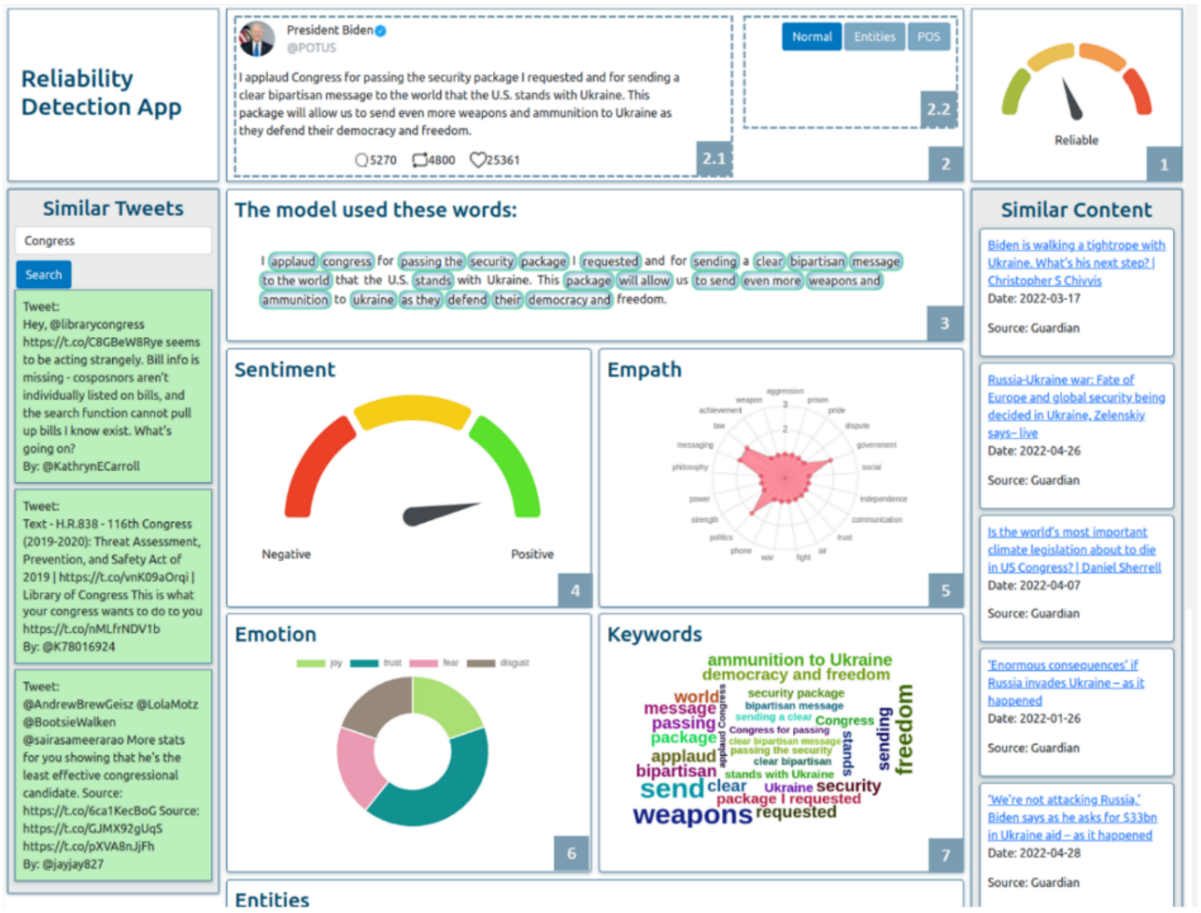


Figure 7 - The App's initial input boxes. For the Tweet Id a single line text box is displayed for entering the tweet's URL.

4.2.2 Widgets - Graphical representations

For a better description of the WebApp's panels and elements we numbered each one using small boxes in the lower-right corner which we will reference as #<number>. This representation can be seen on Figure 8.



President Biden
@POTUS

I applaud Congress for passing the security package I requested and for sending a clear bipartisan message to the world that the U.S. stands with Ukraine. This package will allow us to send even more weapons and ammunition to Ukraine as they defend their democracy and freedom.

5270 4800 25361

The model used these words:

I applaud congress for passing the security package I requested and for sending a clear bipartisan message to the world that the U.S. stands with Ukraine. This package will allow us to send even more weapons and ammunition to ukraine as they defend their democracy and freedom.

Similar Tweets

Congress

Tweet:
Hey, @librarycongress https://t.co/C8GBeW8Rye seems to be acting strangely. Bill info is missing - cosposnors aren't individually listed on bills, and the search function cannot pull up bills I know exist. What's going on?
By: @KathrynECarroll

Tweet:
Text - H.R.838 - 116th Congress (2019-2020): Threat Assessment, Prevention, and Safety Act of 2019 | https://t.co/vnK09aOrqi | Library of Congress This is what your congress wants to do to you https://t.co/nMLfrNDV1b
By: @K78016924

Tweet:
@AndrewBrewGeisz @LolaMotz @BootsieWalken @sairasameerarao More stats for you showing that he's the least effective congressional candidate. Source: https://t.co/6ca1KecBoG Source: https://t.co/GJMX92gUqS https://t.co/pXVA8nJjFh
By: @jayjay827

Similar Content

Biden is walking a tightrope with Ukraine. What's his next step? | Christopher S Chivvis
Date: 2022-03-17
Source: Guardian

Russia-Ukraine war: Fate of Europe and global security being decided in Ukraine, Zelenskiy says– live
Date: 2022-04-26
Source: Guardian

Is the world's most important climate legislation about to die in US Congress? | Daniel Sherrell
Date: 2022-04-07
Source: Guardian

'Enormous consequences' if Russia invades Ukraine – as it happened
Date: 2022-01-26
Source: Guardian

'We're not attacking Russia,' Biden says as he asks for $33bn in Ukraine aid – as it happened
Date: 2022-04-28
Source: Guardian

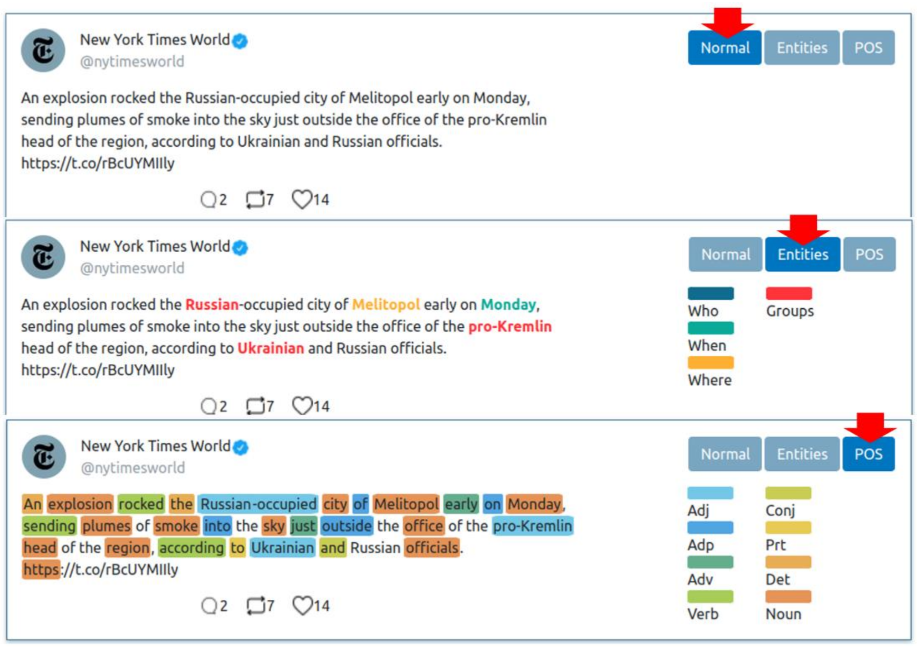
Figure 8 - Reliability Detection App's interface (top of the page)

4.2.3 Model Result

The model result #1 outputs he reliability or unreliability of the text/tweet submitted for analysis. This result is represented by a gauge meter where the needle increases with the unreliability percentage outputted by the model. This representation is based on the concept that a text that is unreliable accumulates various factors building up to determine its unreliability. An example of this type of representation can be seen on Poligrafo[12].

4.2.4 Visualizations of analysis cues

The visualizations of analysis cues are distributed in eleven panels, each one representing a different feature. The middle panel of the first line #2 shows the analyzed text and its Twitter features on the left side #2.1 (in case the method of input chosen was Twitter URL). On the right side #2.2 there are three buttons that allow the user to switch between different types of highlighted text: normal, entities, and POS tags. For the entities, we have highlighted the foreground of each word while for the POS tags we've highlighted the background of each word, thus having a distinct highlight for each category. This type of visualization allows the user to have a more in-depth view on how each element, being entities or the different parts of speech, are distributed in the text, as it can be seen in *Figure 9*.


New York Times World
@nytimesworld

An explosion rocked the Russian-occupied city of Melitopol early on Monday, sending plumes of smoke into the sky just outside the office of the pro-Kremlin head of the region, according to Ukrainian and Russian officials.
https://t.co/rBcUYMIlly

2 7 14

Figure 9 - Normal text (top), verification of recognized mentioned entities (middle) and parts of speech (bottom).

The panel "The model used these words:" (#3) highlights the words or expressions that the model used to calculate its result. In this case, the highlight method is different from the previous ones as it covers the words/expressions in an elliptical shape, differentiating the words/expressions used by the model from the rest of the text.

The "Sentiment" (#4) panel represents the aggregated sentiment detected in the text, that ranges from –1 to 1, where 0 is neutral, negative values represent a negative sentiment and positive values represent a positive sentiment. For this data a gauge meter was used, as it gives the user an intuitive visual cue of this range. The colors used are the standard and most agreed on, where negative values are associated with the red color, values closer to zero (neutral), are associated with yellow color, and positive values are represented by the green color.

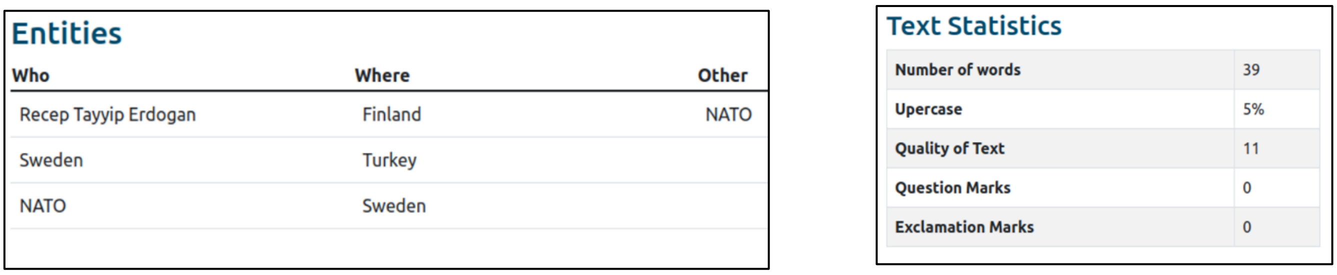
The "Empath" (#5) panel represents a list of categories with their respective weights on the text. The data is represented in a radar plot where each category is displayed at the end of each axis and the point in that axis represents the weight of that category. The scale of this visualization is represented by several radius, incremented by a variable unit, that crosses each axis. In contrast with other visualizations that could have been used for this representation, such as a word cloud or bar plot, the use of a radar plot transmits to the user the polarization of the text and how it may gravitate towards certain categories.

The “Emotion” (#6) panel represents the different emotions detected and their respective percentage in the text. The graphical representation chosen for this feature was a pie chart (normally used to show the different parts of a whole). This representation divides the circle in several portions, each one representing the percentage of each emotion. Each emotion has a color associated that tries to transmit the emotion its associated with. In addition, the emotions are displayed on the pie chart according to a certain order, with the intent of separating positive emotions from negative ones.

The “Keywords” (#7) panel represents the multiple keywords identified and their importance on the text. Each keyword is outputted with the score representing its relevance in the text analyzed. To display this data, we’ve used a world cloud as it allows the user to have a visual interpretation of the different keywords and their importance in the text.

The “Entities” panel displays all the different entities identified in the text, grouped by

- “Who” - entities categories that represent individuals or organizations.
- “When” - entities categories that represent time.
- “Where” - entities categories related with location.
- “Quantity” - entities categories that represent quantity.
- “Groups” - entities that represent groups.
- “Other” - where all the other entities categories were grouped in



Entities

Who	Where	Other
Recep Tayyip Erdogan	Finland	NATO
Sweden	Turkey	
NATO	Sweden	

Text Statistics

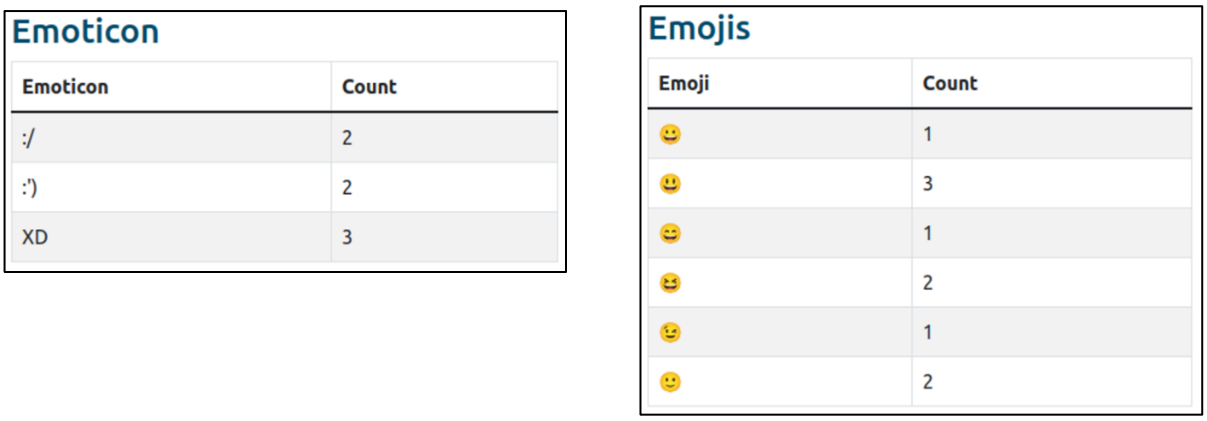
Number of words	39
Upercase	5%
Quality of Text	11
Question Marks	0
Exclamation Marks	0

Figure 10 - Entities and Text statistics

The “Text Statistics” panel shows different text statistics such as the number of words, the percentage of letters that are uppercase, a text score (calculated based on the readability of the text), and the number of question marks and exclamation marks. These two panels are represented in *Figure 10*.

This type of representations complements the highlight representation of the tweet when the button entities is pressed (#2.2), as it displays the data in a more compact manner.

The “Emojis” panel shows the different emojis identified and its respective number of occurrences in the text. Likewise, the “Emoticon” panel serves the same purpose as the “Emojis” but shows the different emoticons identified in the text (both represented in *Figure 11*). The representation chosen for this panels, “Emojis”, “Emoticon” and “Text Statistics” is a table as this data represents more statistical features of the text.



Emoticon

Emoticon	Count
:/	2
:')	2
XD	3

Emojis

Emoji	Count
😀	1
😃	3
😄	1
😆	2
😉	1
🙂	2

Figure 11 - The Emojis and Emoticons panels

The “Similar Content” panel (column on the right of the page) uses an API provided by the English newspaper Guardian and displays a list of different headlines related to the text analyzed. Having the option to see headlines from known and trustworthy sources gives the user more information on the subject that can be used to formulate its opinion.

4.2.5 Query – “Similar Tweets”

The “Similar Tweets” panel (located as the left column of the interface) shows the result of a query where the user can search for classified tweets.

5 Conclusion

All the objectives defined at the beginning of this internship were successfully achieved. Furthermore, an additional set of objectives, proposed throughout the course of this internship, were also successfully achieved. The concretization of these objectives resulted in a fully working WebApp, that satisfies and exceeds the requirements necessary for an interface that allows the classification of the reliability of small statements and further complements that classification with additional information in the form of visual hints.

The WebApp, is fully ready to be deployed and made available for the public to use.[1] However, this does mean there's no room for improvement, a topic to be further explored in the section regarding Future Work (5.2).

5.1 Reflection of the work achieved in this internship

This internship was my first interaction with developing interfaces for the field of Data Science, and although I didn't work directly with the development of the machine learning classification models used in this WebApp, I can say that I've consolidated and expanded my knowledge regarding this field. My previous knowledge regarding this field was trough the Artificial Intelligence course, that is a mandatory component of my degree. The content lectured in this course had a small practical component, however, was very limited, when compared to the actual work and research that goes into creating these types of models. Moreover, I had the privilege of working with active members in the scientific community that gave me some insight on how the investigation is conducted in these field. Another field where I improved my skills was in data visualizations, as the process of choosing and testing the different graphical representations used this project allowed me to learn the best practices regarding this matter.

Learning the ReactJS library and FastAPI framewok was a valuable addition to my skill set and increased my current knowledge regarding the implementation of a fullstack WebApplications. I can now say that I'm very comfortable with the concepts behind ReactJS. The level of complexity regarding the implementation of the backend component was not as high as the frontend component, which led me to not exploring the full capabilities of the FastAPI framework, however, I'm very comfortable working with it, and if required, I would easily acquire the knowledge to build a more complex backend components using this technology.

As for the project itself, I've accomplished all the tasks successfully with a high level of autonomy overcoming the obstacles that would appear on my path. When I couldn't be fully autonomous in solving some of the obstacles, my tutors were very helpful in giving me tools that would help me succeed in my tasks.

5.2 Future Work

This section will describe the future work that could be done on the WebApp, for such, this section will be divided into two categories, improvements, and new features.

5.2.1 Improvements

Although the WebApp is stable and ready for being deployed to the public, it can be improved in some components.

In the representation of the different highlighted features, the process of highlighting the text is done in the backend, as explained in the Methods section of this report. The method followed for this implementation is not encouraged by the React development team, as it can represent a security

[1] Minus some possible adjustments needed for the integration in a web server. These adjustments may vary depending on the platform were the

issue for some WebApplications, however this is not the case for this WebApp, as it only returns a static HTML block. However, there is already a fix for this issue in the WebApp in the visualization “The Model used these words:” and will be implemented as soon as possible.

The compatibility of this interface in smaller devices wasn’t considered for this WebApp, and is a very welcome improvement, since a large amount of internet browsing is done in mobile devices with smaller screens. There are two possibilities for this improvement, either develop an entire mobile application that could be installed on the device or create a new WebApp layout that would be compatible with smaller screens.

Since the priority in this internship was to have a fully working prototype, the animations were not implemented in this WebApp. The implementation of some animations would significantly improve the user experience while using the WebApp.

5.2.2 New Features

The addition of new features was suggested in every step of this internship. Some were implemented in the WebApp, some were not feasible or relevant to the project, however, some were good additions to the project, however, due to the time limitation they were put to the side.

The strong connection between this WebApp and Twitter always suggested for a Twitter bot to developed. This bot would be tagged in a tweet and would proceed to compute a full analysis of the text. Once the analysis was completed, it would respond to the tagged tweet with a simple text classification and a link the WebApp were the full analysis would be available.

The machine learning model used in this WebApp is not unique, as there are some variations trained with different algorithms. This new feature would allow for the integration of different machine learning models in this WebApp. This integration would allow us to show the differences in results between the different models, and after some research, would also allows us to try to show the meaning behind these differences.

Bibliography

1. Bovet, A. and H.A. Makse, *Influence of fake news in Twitter during the 2016 US presidential election.* Nature Communications, 2019. **10**(1): p. 7.
2. Rocha, Y.M., et al., *The impact of fake news on social media and its influence on health during the COVID-19 pandemic: A systematic review.* Journal of Public Health, 2021: p. 1-10.
3. Zhang, X. and A.A. Ghorbani, *An overview of online fake news: Characterization, detection, and discussion.* Information Processing & Management, 2020. **57**(2): p. 102025.
4. Figueira, A., N. Guimaraes, and L. Torgo. *Current State of the Art to Detect Fake News in Social Media: Global Trendings and Next Challenges*.
5. Guimarães, N., Á. Figueira, and L. Torgo, *Can Fake News Detection Models Maintain the Performance through Time? A Longitudinal Evaluation of Twitter Publications.* Mathematics, 2021. **9**(22).
6. Yang, K.-C., E. Ferrara, and F. Menczer, *Botometer 101: Social bot practicum for computational social scientists.* arXiv preprint arXiv:2201.01608, 2022.
7. Shao, C., et al., *Hoaxy: A Platform for Tracking Online Misinformation.* WWW '16 Companion: Proceedings of the 25th International Conference Companion on World Wide Web, 2016.
8. Torabi Asr, F. and M. Taboada, *Big Data and quality data for fake news and misinformation detection.* Big Data & Society, 2019. **6**(1): p. 2053951719843310.
9. Figueira, Á., N. Guimaraes, and L. Torgo, *A brief overview on the strategies to fight back the spread of false information.* Journal of Web Engineering, 2019. **18**(4): p. 319-352.
10. *Snopes*. [cited 2022 30-06]; Available from: https://www.snopes.com/.
11. *Politifact*. [cited 2022 30-06]; Available from: https://www.politifact.com/.
12. *Polígrafo*. [cited 2022 30-06]; Available from: https://poligrafo.sapo.pt/autor/poligrafo-sic.
13. *Flask*. [cited 2022 30-06]; Available from: https://flask.palletsprojects.com/en/2.1.x/.
14. *Django*. [cited 2022 30-06]; Available from: https://www.djangoproject.com/.
15. *FastAPI*. [cited 2022 30-06]; Available from: https://fastapi.tiangolo.com/.
16. *FastAPI - Performance*. [cited 2022 30-06]; Available from: https://fastapi.tiangolo.com/#performance.
17. *Pydantic Models*. [cited 2022 30-06]; Available from: https://pydantic-docs.helpmanual.io/.
18. *FastAPI - Validation*. [cited 2022 30-06]; Available from: https://fastapi.tiangolo.com/features/#validation.
19. *TwitterAPI*. [cited 2022 30-06]; Available from: https://developer.twitter.com/en/portal/dashboard.
20. *Tweepy*. [cited 2022 30-06]; Available from: https://www.tweepy.org/.
21. *Jinja2*. [cited 2022 30-06]; Available from: https://jinja.palletsprojects.com/en/3.1.x/.
22. *ReactJS*. [cited 2022 30-06]; Available from: https://reactjs.org/.
23. *ChartJS*. [cited 2022 30-06]; Available from: https://www.chartjs.org/.
24. *React-gauge-Chart*. [cited 2022 30-06]; Available from: https://martin36.github.io/react-gauge-chart/.
25. *Wordcloud*. [cited 2022 30-06]; Available from: https://react-wordcloud.netlify.app/.
26. *Semantic UI*. [cited 2022 30-06]; Available from: https://semantic-ui.com/.
27. *React Bootstrap*. [cited 2022 30-06]; Available from: https://react-bootstrap.github.io/.
28. *ReactJS - Tutorials*. [cited 2022 30-06]; Available from: https://reactjs.org/tutorial/tutorial.html.
29. *CENTERIS - International Conference*. in *CENTRIS*. 2022.
30. *ReactJS - Props*. [cited 2022 30-06]; Available from: https://reactjs.org/docs/components-and-props.html.
31. *ReactJS - Conditional Rendering*. [cited 2022 30-06]; Available from: https://reactjs.org/docs/conditional-rendering.html.

32. *ReactJS - State*. [cited 2022 30-06]; Available from: https://reactjs.org/docs/state-and-lifecycle.html.
33. *ReactJS - Hooks*. [cited 2022 30-06]; Available from: https://reactjs.org/docs/hooks-intro.html.
34. *ReactJS - useState() and useEffect()*. [cited 2022 30-06]; Available from: https://reactjs.org/docs/hooks-effect.html.
35. Vandemeulebroecke, M., et al., *Effective visual communication for the quantitative scientist.* CPT: pharmacometrics & systems pharmacology, 2019. **8**(10): p. 705-719.
36. *Google - word2vec*. [cited 2022 30-06]; Available from: https://code.google.com/archive/p/word2vec/.
37. *NLTK*. [cited 2022 30-06]; Available from: https://www.nltk.org/.
38. *Stanza*. [cited 2022 30-06]; Available from: https://stanfordnlp.github.io/stanza/.
39. *Empath - Python library for psycholinguistic categories indetification*. [cited 2022 30-06]; Available from: https://pypi.org/project/empath/.
40. Campos, R., et al., *YAKE! Keyword extraction from single documents using multiple local features.* Information Sciences, 2020. **509**: p. 257-289.
41. *GuardianAPI*. [cited 2022 30-06]; Available from: https://open-platform.theguardian.com/.
42. Guimarães, N., Á. Figueira, and L. Torgo. *Contributions to the Detection of Unreliable Twitter Accounts through Analysis of Content and Behaviour*.

Code Snippets

```python
class Result(BaseModel):
    reliable: float
    unreliable: float


@app.post("/model_result", response_model=Result)
async def textResult(content: Text):
    pr = model_in.getModelResult(content)
    result = Result(
        reliable=pr["reliable"],
        unreliable=pr["unreliable"],
    )
    return result
```

Code Snipet 1 - Model Result Endpoint

```python
@app.post("/model_used_words")
async def modelwords(content: Text):
    mw = feat_in.highlightWordsFromWord2Vec(content.text)
    mwd = json.dumps(mw)

    mw_json = jsonable_encoder(mwd)

    return mw_json
```

Code Snipet 2 - Model used these words endpoint

```python
class TextStatistics(BaseModel):
    upperCase: dict
    read_score: dict
    nQuestionMarks: dict
    nExclamationMarks: dict
    countWords: dict
    emojis: dict
    emoticons: dict
class Data(BaseModel):
    text: str
    empath: dict
    sentiment: dict
    emotion: dict
    textStatistics: TextStatistics
    entities: list
    pos: list
    keywords: dict
@app.post("/text_classify", response_model=Data)
async def readText(content: Text):

    ent = feat_in.getEntities(content.text)
    pos = feat_in.getPosTags(content.text)

    data = Data(
        text=content.text,
        empath=feat_in.getEmpathTweets(content.text),
        sentiment=feat_in.getSentiment(content.text),
        emotion=feat_in.getEmotions(content.text),
        textStatistics=TextStatistics(
            upperCase=feat_in.uppercasePercDescription(content.text),
            read_score=feat_in.getTweetsRead(content.text),
            nQuestionMarks=feat_in.numberofQuestionMarks(content.text),

nExclamationMarks=feat_in.numberofExclamationMarks(content.text),
            countWords=feat_in.countWords(content.text),
            emojis=feat_in.getEmojisTweetsWeb(content.text),
            emoticons=feat_in.getEmoticonsTweetsWeb(content.text)
        ),
        entities=jsonable_encoder(ent),
        pos=jsonable_encoder(pos),
        keywords=jsonable_encoder(feat_in.getKeywords(content.text))
    )
    print(jsonable_encoder(data))
    return data
```

Code Snipet 3 - The features extracted from the text

```
class DataHiglight(BaseModel):
    entH: str
    posH: str
    entities: list
    pos: list

@app.post("/text_highlighting", response_model=DataHiglight)
async def readText(content: Text):
    ent = feat_in.getEntities(content.text)
    pos = feat_in.getPosTags(content.text)
    data = DataHiglight(
        entH=feat_in.getHighlighting(content.text, "entities"),
        posH=feat_in.getHighlighting(content.text, "pos"),
        entities=jsonable_encoder(ent),
        pos=jsonable_encoder(pos),
    )

    return data
```
Code Snipet 4 - Highlight endpoint

```
@app.post("/similar_content")
async def getSimilarContent(content: Text):
    sm = feat_in.similarContent(content.text)
    smd = json.dumps(sm)

    sm_json = jsonable_encoder(smd)

    print(sm_json)

    return sm_json
```
Code Snipet 5 - Similar Content

```
@app.get("/similar_tweets={query}")
async def getSimilarTweets(query: str):

    st = jsonable_encoder(tl.searchLocalTweets(query))

    random.shuffle(st)

    print(st)

    return st
```
Code Snipet 6 - Similar Tweets

```python
@app.get("/id_info={id}", response_model=TweetData)
async def readId(id: str):
    client = tweepy.Client(bearer_token=bearerToken)
    # Get tweet info and metrics
    tweet_metric = client.get_tweet(
        id, tweet_fields=['public_metrics'], expansions=['author_id'])
    if(tweet_metric.errors):
        # Error handling
        raise HTTPException(
            status_code=404, detail="Invalid ID, Tweet not found")
    else:
        id = tweet_metric.data.author_id
        user_ids = [id]
        # Get tweet author info
        response = client.get_users(ids=user_ids, user_fields=[
                                    "verified,profile_image_url,name,url"])
        if(response.errors):
            # Error handling - Can't happen
            raise HTTPException(status_code=404, detail="Invalid user ID")
        else:
            for user in response.data:
                username_r = user.username
                pic_url_r = user.profile_image_url
                verified_r = user.verified
                name_r = user.name
            data = TweetData(
                username=username_r,
                name=name_r,
                pic_url=pic_url_r,
                verified=verified_r,
                rt_count=tweet_metric.data.public_metrics['retweet_count'],
                q_count=tweet_metric.data.public_metrics['quote_count'],
                reply_count=tweet_metric.data.public_metrics['reply_count'],
                like_count=tweet_metric.data.public_metrics['like_count'],
                text=tweet_metric.data.text
            )
            return data
class TweetData(BaseModel):
    username: str
    name: str
    pic_url: str
    verified: Boolean
    rt_count: int
    q_count: int
    reply_count: int
    like_count: int
    text: str
```

Code Snipet 7 - Tweet Features

```
{!toggleTT && (
        <Form>
          <Form.Group className="mb-3" controlId="textContent">
            <h3>Text</h3>
            <Form.Control
              id="text_box"
              as="textarea"
              rows={5}
              cols={10}
              style={{ resize: "none" }}
              name="textContent"
              value={textContent}
              onChange={(e) => setTextContent(e.target.value)}
            />
          </Form.Group>
        </Form>
      )}
      {toggleTT && (
        <>
          <Form>
            <Form.Group className="mb-3">
              <h3>TweetID</h3>
              <Form.Control
                id="id_box"
                as="textarea"
                rows={1}
                cols={10}
                style={{ resize: "none" }}
                name="textContent"
                value={textContent}
                onChange={(e) => setTextContent(e.target.value)}
              />
            </Form.Group>
          </Form>
        </>
      )}
```

Code Snipet 8 - Text and Tweet switch