Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

CAI Centre for Artificial Intelligence

Bachelor thesis (Computer Science)

Claim Extraction

Authors	Pascal Isliker Christoph Mathis Karin Birle
Main supervisor	Mark Cieliebak
Sub supervisor	Pius von Däniken
Date	09.06.2023

Formula

DECLARATION OF ORIGINALITY
Bachelor's Thesis at the School of Engineering

By submitting this Bachelor's thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party. (Group works: the performance of the other group members are not considered as third party).

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Bachelor thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations

City, Date:	**Name Student**:
Winterthur, 09.06.2023	Pascal Isliker
Winterthur, 09.06.2023	Christoph Mathis
Winterthur, 09.06.2023	Karin Birle

Abstract

In recent years widespread fake news have become a problem. To tackle this issue, it is necessary to fact-check information. However, manual fact-checking is a time-consuming process. In order to minimize the amount of information requiring fact-checking, it is important to automatically identify relevant claims. The objective of this project is to develop a system that can automatically identify relevant claims. The Conference and Labs of the Evaluation Forum (CLEF) organizes yearly competitions, one of them being the CheckThat! competition, which focuses on the issue of fake news. Task 1B of the CheckThat!2023 competition focused on identifying relevant claims. We participated in the competition and focused on task 1B. The CheckThat! lab provided data that can be utilized to train a system on the task of predicting checkworthiness. We have built a system by analysing this data and doing experiments. To accomplish this, we analyzed the given data and experimented with various transformer models like BERT and ELECTRA. We also explored the usage of other models such as word2vec. For our final system, we employed a meta-estimator to consolidate all our predictions into a single output. We utilized this system to participate in the CheckThat!2023 competition, placing 10th out of 11 teams with an F-score of 0.767. Following the deadline of the CheckThat! competition we further improved our system by adding predictions from ClaimBuster and OpenAI using their APIs. Additionally, we applied hyperparameter search to the transformer models we used. The F-score of our final model using the Logistic Regression with cross validation as a meta-estimator was 0.817.

Zusammenfassungen

In den letzten Jahren sind weit verbreitete Fake News zu einem Problem geworden. Um dieses Problem in den Griff zu bekommen, ist es notwendig, Informationen auf ihre Richtigkeit hin zu überprüfen. Die manuelle Überprüfung von Fakten ist jedoch ein zeitaufwändiger Prozess. Um die Menge der zu prüfenden Informationen zu minimieren, ist es wichtig, relevante Behauptungen automatisch zu identifizieren. Ziel dieses Projekts ist es, ein System zu entwickeln, das automatisch relevante Behauptungen identifizieren kann. Das Conference and Labs of the Evaluation Forum (CLEF) organisiert jährlich Wettbewerbe. Einer davon ist der CheckThat!-Wettbewerb, der sich mit dem Thema Fake News beschäftigt. Aufgabe 1B des CheckThat!2023-Wettbewerbs konzentrierte sich auf die Identifizierung relevanter Behauptungen. Wir haben an diesem Wettbewerb teilgenommen und uns auf Aufgabe 1B fokussiert. Das CheckThat!2023-Labor lieferte Daten, die genutzt werden können, um ein System für die Vorhersage der Prüfwürdigkeit zu trainieren. Wir haben ein System entwickelt, indem wir diese Daten analysiert und Experimente durchgeführt haben. Dazu analysierten wir die gegebenen Daten und experimentierten mit verschiedenen Transformer Modellen wie BERT und ELECTRA. Wir haben auch die Verwendung anderer Modelle wie word2vec untersucht. Für unser endgültiges System setzten wir einen Meta-Estimator ein, um alle unsere Vorhersagen in einer einzigen Ausgabe zusammenzufassen. Mit diesem System nahmen wir am Wettbewerb CheckThat!2023 teil und belegten mit einem F-score von 0.767 den 10. Platz von 11 Teams. Nach Ablauf der Frist für den CheckThat!-Wettbewerb haben wir unser System weiter verbessert, indem wir weitere Vorhersagen mithilfe der APIs von ClaimBuster und OpenAI hinzugefügt haben. Ausserdem wendeten wir die Hyperparametersuche auf die von uns verwendeten Transformer-Modelle an. Der F-score unseres endgültigen Modells betrug 0.817. Dabei haben wir den Logistic Regression mit Cross-Validation als Meta-Estimator verwendet.

Acknowledgements

We would like to thank everyone who has supported us throughout the process of creating our bachelor thesis. In particular, we want to thank our supervisors Mark Cieliebak and Pius von Däniken.

We would like to acknowledge the valuable assistance of ChatGPT, an advanced language model developed by OpenAI. ChatGPT was employed for the purpose of rephrasing sentences and providing grammatical improvements to enhance the clarity and coherence. The suggestions contributed to the readability and overall quality of our paper.

Contents

1 Introduction

1.1 Starting Position

Social media's widespread use for news has exposed the rapid spread of misinformation, as seen in the 2016 US presidential election [1] and the COVID-19 pandemic [2]. To address this issue, information must be fact-checked. However, fact-checkers face limitations in manually examining all content. To aid the process, automatic claim extraction can help to identify claims requiring verification, allowing fact-checkers to focus on evaluating potentially deceptive content.

A claim is the key component of an argument [3], "an assertion that deserves our attention" [4]. For example "Yesterday it was raining in London." is a claim that is usually not a relevant (not checkworthy) claim. But "In 2022, the unemployment rate in Great Britain was estimated at around 3.70 percent." is a checkworthy claim. "How do you do?" and "I'd like to mention one thing." are not claims.

1.2 Task Definition

In this project we provide an overview of existing models and according literature for claim extraction out of text and detection of checkworthiness. Based on this knowhow a model is trained, to detect checkworthy claims. To get data to train the model and evaluate how well the model performs, we work with the datasets from the CheckThat! lab challenge of the current year.

1.3 CheckThat! Lab

The Conference and Labs of the Evaluation Forum (CLEF) [5] organizes yearly challenges. Ever since 2018 there has been a lab regarding claim extraction. We participated in task 1B [6] of this year's challenge. The goal of task 1B was to predict the checkworthiness of tweets and transcriptions of speeches and debates in English, Spanish and Arabic. We focused on the English part of the task 1B which consists exclusively of transcriptions of speeches and debates.

1.4 Definition of checkworthiness

In last years (2022) CheckThat! competition, checkworthiness was described as follows:

"checkworthiness: Do you think that a professional fact-checker should verify the claim in the tweet? This question asks for a subjective judgment. Yet, its answer should be based on whether the claim is likely to be false, is of public interest, and/or appears to be harmful. Note that we stress the fact that a professional fact-checker should verify the claim, ruling out claims that are easy to fact-check by a layperson." [7]

Therefore, checkworthiness of a Claim is affected by

- Likeliness to be false
- Relevance / Public interest
- Harmfulness

1.5 Overview

Section 2 Related Work references promising approaches and Section 3 explains important models and provides some theoretical foundations for this work. The datasets we utilized are described in Section 4 and in Section 5 we explain how we comprehend the task and its complexity. We accomplished this by manually labeling a small amount of data and comparing our labels to each other, the labels generated by the models, and the gold labels. This allowed us to get a better understanding as to how the data was labeled and assess the level of agreement among ourselves and determine if our performance surpassed that of the models. In Section 6 Methods we show how we solve the task by training different models like the recently best performing transformer models BERT, ELECTRA, DistilBERT as well as basic models such as word2vec and n-gram on the provided training dataset. After that, we will let those models as well as the OpenAI API model and the ClaimBuster API model make predictions on a secondary dataset. With all those predictions, we will then train a meta-estimator to get our final model. This final model we use to predict the labels of the test data to see how well the meta-estimators perform on this dataset. As previous work indicates that even bad performing models can contribute to combined decisions [8] we expect that the final model performs best, but we will compare all the models to get a ranking. Based on that ranking, we will decide which model we will use for the CheckThat! submission. Section 7 Results presents the intermediate results on the test data, and shows the scores we achieved on the submission dataset. Afterwards we will analyse our results, improve the model according to our learnings and give ideas for further improvement in Section 8 Conclusion and Discussion.

2 Related Work

Current research on claims can be separated in three categories as shown in Figure 1. [9]

- Claim detection: Claim detection is the process of extracting claims from a given context. [10]
- Claim checkworthiness: Detected claims can be manually fact-checked. However, fact-checking claims manually (by professional fact-checkers) is very time-consuming. checkworthiness is introduced to reduce the number of claims that need to be checked. [11]
- Claim verification: Claim verification is the task of checking if a claim is factual / trustworthy. [12]

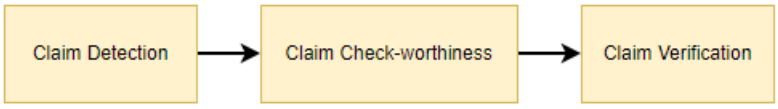

Figure 1: Shows the three research categories the claims are separated in.

In this paper we will focus on claim checkworthiness.

2.1 Checkworthiness for speeches and debates

The ClaimBuster [13] system was developed to target checkworthiness. Their system was specifically built based on data from U.S. presidential debates. The system was built to classify sentences into one of three categories: Non-Factual, Unimportant Factual and Check-worthy Factual. Later on a system called ClaimRank [11] based on [14] was introduced. ClaimRank was also trained on political debates. The ClaimRank system can predict the checkworthiness of a sentence on a scale from 0 (not checkworthy) to 1 (checkworthy).

2.2 Previous Approaches

In previous iterations of the CheckThat! lab there have already been tasks that focused on predicting checkworthiness. So far they have all focused either on tweets or political debates and speeches. Initial approaches on solving the problem of checkworthiness involved models such as Random Forest, Support Vector Machines (SVM), Multinomial Bayes, and features based on Term Frequency-Inverse Document Frequency (TF-IDF) representation, part-of-speech tags, sentiment scores, and named entities. [13], [15]. In the case of check-worthy detection of claims, Recurrent Neural Networks (RNN) produced the best unofficial run in 2018 [15], and in 2019, a Neural Network (NN) with Long Short-Term Memory (LSTM) also performed the best. In more recent competitions transformer models have become the state-of-the-art model for predicting checkworthiness. Starting in 2020 [16] they have been used to achieve the best

results in the CheckThat! competition [7], [17]. The task of predicting checkwothiness in text inherently includes claim detection. This is because if a text fragment is not a claim, it is never considered checkworthy.

2.3 Previous Competitions

In this section, we will introduce the methods that achieved the best results in each year's respective CheckThat! competition in the according checkwothyness task.

2.3.1 CheckThat!2020

Task 1: Checkworthiness estimation for tweets [18]

Task 1 of the CheckThat!2020 competition involved estimating the checkworthiness of tweets. The topic and a stream of potentially related tweets were given and had to be ranked according to their checkworthiness regarding that topic. The topic of focus was COVID-19, and the manually annotated tweets were collected from March 2020 [18]. Most of the participating groups used pre-trained models such as BERT and RoBERTa. However, there were also groups that used traditional models such as SVM and Logistic Regression. The official evaluation measure was mean average precision (MAP) complemented with further metrics. The best performing team Accenture [19] used a model based on RoBERTa. They reached a MAP score of 0.806. Nearly as well performed team Team_Alex with a MAP score of 0.803 by also using RoBERTa.

2.3.2 CheckThat!2021

In the CheckThat!2021 competition there were two tasks regarding checkworthiness. In one task, the dataset consisted of tweets, while in the other task, the participants had to classify sentences of political debates and speeches.

Task 1A: Checkworthiness of tweets [17]

Subtask 1A was offered in Arabic, Bulgarian, English, Spanish, and Turkish. The tweets focused on COVID-19, vaccines, and politics, and were crawled and manually annotated between January 2020 and March 2021. The evaluation was performed per language, but it was possible to use multilingual approaches that leverage information from all available datasets. 15 teams participated in subtask 1A, the most successful approaches used transformers or a combination of embeddings, manually engineered features, and NNs. The top performing team on the English dataset was NLP&IR@UNED with a MAP score of 0.224 using several pre-trained transformer models. BERTweet, the model they used, performed the best on the development set and was trained using RoBERTa on 850 million English tweets and 23 million COVID-19 English tweets. The second-best team, Fight for 4230, achieved a MAP score of 0.195. They utilized BERTweet with a dropout layer as their primary model. Moreover, they implemented pre-processing techniques and data augmentation methods as part of their approach.

Task 1B: Checkworthiness of debates or speeches [17]

In Subtask 1B previously fact-checked political debates and speeches in English from PolitiFact were provided. This task has evolved from the first edition of the CheckThat! lab and in each new edition (2018, 2019, 2020) more training data from increasingly diverse sources has been added. The original task was defined as follows: "Given a transcript of a speech or a political debate, rank the sentences in the transcript according to the priority with which they should be fact-checked." [17]

The task got submissions from two teams and only one of them, team Fight for 4230 with a MAP score of 0.402, performed better than the n-gram baseline using a RoBERTa model that was fine-tuned on tweets_hate_speech_detection dataset with one dropout- and one classifier layer. The baseline had a MAP score of 0.235 [20].

2.3.3 CheckThat!2022

Task 1A: Checkworthiness of tweets [7]

Subtask 1A involved labeling tweets with "yes" or "no" in response to the question "Do you think a professional fact-checker should verify the claim in the tweet?". They provided manually annotated datasets for Arabic, Bulgarian, Dutch, English and Turkish, which were also used for the other subtasks, while the Spanish dataset was only used for subtask 1A and provided in a larger-scale dataset. The tweets were collected in the timeframe from January 2020 until March 2021 by specifying the language and a set of COVID-19 keywords. Retweets, replies, duplicates and tweets with less than five words were removed and the most frequently liked and retweeted tweets were selected for annotation. To rank the teams, the F1-measure was utilized with respect to the positive class ("yes") to account for class imbalance. The best performing team for the English dataset was AI Rational with a F-score of 0.698 [7]. They conducted experiments with various transformer models and decided to utilize RoBERTa specifically for the English dataset. To increase the amount of training data AI Rational used a data augmentation process based on back-translation [21]. Team Zorros submitted the second-best performing system, which achieved an F-score of 0.667, based on an ensemble approach combining BERT and RoBERTa.

2.4 Context

Regarding the identification of checkworthy claims in texts, the previous CheckThat! labs have been our most important sources of reference. Our most important principles for the construction and implementation of transformer models [22], [23], NNs [24], and OpenAI [25] are compiled in the following Section 3 Theoretical Foundations. The most important source on how even bad performing models can contribute to meta-estimators was [8]. Building upon this foundation, our study leverages a diverse array of established models to consistently achieve optimal prediction outcomes.

3 Theoretical Foundations

3.1 Neural Network

A Neural Network (NN) is a method in artificial intelligence that teaches computers to process data. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain. NNs are a set of algorithms that are designed to recognize patterns. The patterns they recognize are numerical vectors, into which all real-world data, for example text, has to be translated.

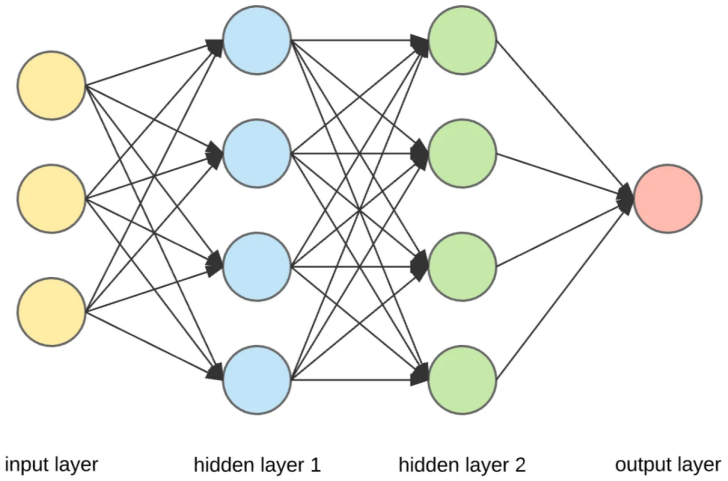

Figure 2: ANN with two hidden layers [26]

Artificial Neural Networks (ANNs) are structured as a series of interconnected layers composed of nodes like shown in Figure 2. These layers consist of an input layer, which receives external data, one or more hidden layers responsible for intermediate computations and an output layer that produces the final results. Within this network, each node, or artificial neuron, is linked to others and possesses a weight and a threshold. When the output of a node exceeds its threshold, it becomes activated and transmits information to the subsequent layer. Conversely, if the output does not surpass the threshold, no data is propagated to the following layer. Feedforward Neural Network (FNN) are ANNs wherein connections between the nodes do not form a cycle. They are also known as multi-layer perceptrons (MLPs). Convolutional Neural Networks (CNNs) share similarities with FNNs, but they are predominantly employed for tasks such as image recognition, pattern recognition, and computer vision. CNNs leverage principles from linear algebra, specifically matrix multiplication, to effectively detect patterns within an image. Recurrent Neural Networks (RNNs) are distinguished by their characteristic feedback loops, which allow them to process sequential and time-series data. These learning algorithms are particularly well-suited for tasks that involve making predictions about future outcomes based on historical information [24].

3.2 Feature Extraction

Feature extraction is a process that involves transforming raw data into a more compact representation, a so-called feature, that captures relevant information from the original data. It aims to reduce the dimensionality of the data while preserving or enhancing the discriminatory information. The raw data, such as images, text, or audio signals, often contain high-dimensional and redundant information. Feature extraction identifies and extracts meaningful patterns, structures, or characteristics that can represent the data effectively. The extracted features should possess certain properties, such as relevance, discriminatory power, and robustness, to be useful for subsequent machine learning tasks. Feature extraction helps to improve the efficiency and effectiveness of subsequent algorithms by providing more informative and manageable representations of the data [27].

3.3 Transformer model

Transformer models were introduced in 2017 by Google [22]. They are used to solve various NLP (natural language processing) tasks [23]. The key concept behind transformers is the attention mechanism as described in [22]. As language models are trained on tokens that consist of text pieces that often occur together and not on raw text this has an impact on how they perceive text and prompts as those are sets of tokens. GPT-style models utilize tokenization methods like Byte Pair Encoding (BPE). Those map all input bytes to token IDs in a greedy manner. The model is mapping a query and a set of key-value pairs to an output. The query, the keys, the values, and the output are all described vectors. The output is the weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

3.3.1 BERT

The Bidirectional Encoder Representation Transformer (BERT) is the first bi-directional pre-trained language model. Learning the deep meaning of words and contexts using self-supervised learning, the pre-trained model can be adapted to different tasks as well as different datasets with minimal adjustment.

BERT uses a Masked Language Model (MLM) pre-training objective. It randomly masks some tokens from the input, so the original vocabulary identifier of the masked word has to be predicted only based on its context. In addition to the MLM, a "next sentence prediction" task that jointly pre-trains text-pair representations is used. BERT is widely used as a baseline, but training can be computationally expensive [28].

3.3.2 DistilBERT

DistilBERT is a distilled version of BERT. It is a smaller, faster, cheaper and lighter version of BERT [29]. BERT is a large model with 12 or 24 transformer layers, depending on the variant (BERT-base or BERT-large) and has a number of parameters, typically ranging in the hundreds of millions or even billions. Designed to have a smaller memory footprint and

faster inference times DistilBERT typically has 6 transformer layers and significantly fewer parameters. DistilBERT does not optimize the performance of BERT, but it reduces the size and enhances the speed, so it is often used in scenarios where computational resources are limited or when faster inference is desired [29].

3.3.3 ELECTRA

The Efficiently Learning an Encoder that Classifies Token Replacements Accurately (ELECTRA) was introduced it as follows: "Instead of masking the input, our approach corrupts it by replacing some tokens with plausible alternatives sampled from a small generator network. Then, instead of training a model that predicts the original identities of the corrupted tokens, we train a discriminative model that predicts whether a token in the corrupted input was replaced by a generator sample or not." [30] That means while BERT is a bidirectional model trained with masked language modeling (MLM) and next sentence prediction objectives, Electra is trained to discriminate between original and replaced tokens using a replaced token detection task. So Electra achieves comparable or better performance than BERT while requiring fewer resources.

3.4 OpenAI

3.4.1 GPT-3

GPT-3 [31] is a large language model (LLM) that uses deep learning to generate human like text. Its base is a Generative Pretrained Transformer (GPT) which uses a transformer neuronal network to generate text. The capability of LLMs increases as the size of their input datasets and parameter space increases. GPT models were first launched in 2018 by OpenAI as GPT-1 [32]. All GPT models use transformer architectures [22].

3.4.2 ChatGPT

ChatGPT is the next iteration of GPT-3 and is trained to be conversational. ChatGPT is using Reinforcement Learning from Human Feedback (RLHF) and uses the same methods as InstructGPT with slightly different data collection setups. The RLHF consists of tree steps described in detail in [25]. In step 1 the Supervised Fine-Tuning (SFT) Model collects demonstration data and trains on a supervised policy. In step 2 the Reward Model the responses from the SFT model are taken and ranked by labelers form best to worst output. The combinations of the rankings are served to the model as batch datapoint. In step 3 the Reinforcement Learning Model receives a new prompt from the dataset, the policy generates an output which will get a reward calculated by the reward model. This reward is used to update the policy using Proximal Policy Optimization (PPO) introduced by OpenAI in 2017 [33].

3.4.3 Prompt Engineering

In order to engage in meaningful communication with a LLM, it is necessary to formulate a written query known as a "prompt." Prompt engineering refers to the process of constructing

a well-designed prompt that increases the probability of the LLM effectively addressing and solving the intended task. The prompt can have a range of variations, spanning from a partial sentence that requires completion by the LLM to an entire paragraph that necessitates summarization by the LLM. The prompt can have explicit instructions regarding the methodology or approach for solving the given task, as well as incorporate questions that should be answered by the LLM. Additional contextual information can be provided within the prompt to guide the LLM in formulating a response that aligns with a specific framing or perspective. Zero-shot prompting involves presenting the task or objective to the LLM without providing any explicit instructions or examples, relying solely on the LLM's ability to infer the desired response based on its training and understanding of the given task. In one-shot prompting, a single example of the expected response is provided to the LLM as a reference, allowing the LLM to generate subsequent responses based on that given example. On the other hand, few-shot prompting involves presenting multiple examples of the desired response type to the LLM, enabling it to learn and generalize patterns from the given examples to generate appropriate responses. Only the Few-shot prompting method was used, as the zero-shot and one-shot approaches have been shown to be mostly worse [31].

The Prompt Engineering Guide[1] serves as a valuable resource for additional readings on the topic.

3.5 Fine-Tuning

Pre-trained models often need to be fine-tuned [34]. This means taking the pre-trained model and training it further on a specific task with task-specific labeled data. This process adapts the model's knowledge to the target task and typically leads to improved performance compared to training from scratch.

[1] https://www.promptingguide.ai/

4 Datasets and Baselines

The CheckThat!2023 lab provides us with data in Spanish, Arabic and English. In Spanish and Arabic the data provided consists of tweets and in English the data consists of sentences from debates and speeches.

Initially three datasets Train, Dev and Dev-Test were provided for each language. The number of checkworthy and non-checkworthy sentences per dataset can be seen in the "Yes" (checkworthy) and "No" (not checkworthy) columns of Table 1. There are significantly more sentences that are not checkworthy ("No") rather than checkworthy ("Yes"). In general there are about three times as many sentences labeled with "No" than there are labeled with "Yes". That means that the data provided for the CheckThat! lab has a bias towards sentences that were considered not to be checkworthy.

After the CheckThat!2023 competition had concluded, we received the Test dataset, which was used to evaluate the performance of the submitted models. This means that we were not able to use the Test dataset to improve or evaluate our models in preparation for the CheckThat!2023 competition. The Test dataset has a significantly higher ratio of "Yes" labels, compared to the datasets that were provided originally, as shown in Table 1.

Language	**Type**	**Split**	**Yes**	**No**	**Total**
English	Speech / Debate	Train	4058	12818	16876
		Dev	1355	4270	5625
		Dev-Test	238	794	1033
		Test	108	210	318
Spanish	Tweets	Train	2208	5280	7490
		Dev	299	2161	2500
		Dev-Test	704	4296	5000
Arabic	Tweets	Train	1758	4301	6060
		Dev	485	789	1274
		Dev-Test	411	682	1094

Table 1: Provides an overview of the data that was provided for the CheckThat!2023 competition. The English data consists of speeches and debates while the data for Spanish and Arabic consists of tweets. The data is separated in the datasets: Train, Dev, Dev-Test and Test (English only) per language.

4.1 Baselines

The CheckThat!2023 lab provides three baselines for task 1B. The results of these baselines across all languages are shown in Table 2. Overall the n-gram baseline has the best performance with an F-score of 0.821 on the Dev-Test dataset.

Model	**1B - Arabic**	**1B - Spanish**	**1B - English**
Random Baseline	0.364	0.153	0.220
Majority Baseline	0.000	0.000	0.000
n-gram Baseline	0.202	0.546	0.821

Table 2: Shows the performance (F-score) of the baseline models (Random, Majority, n-gram) on the Dev-Test dataset.

4.1.1 Random Baseline

The random baseline predicts the labels based on random chance, which means over a large sample size 50 % of the sentences will be predicted to be checkworthy ("Yes").

4.1.2 Majority Baseline

The majority baseline predicts sentences based on the label that was most frequent in the training dataset. In this case the Train dataset had more "No" labels. Therefore, the majority baseline predicted all sentences not to be checkworthy ("No").

The F-score is calculated as follows:

$$F - score = \frac{(precision \cdot recall)}{(precision + recall)}, precision = \frac{TP}{TP + FP}, recall = \frac{TP}{TP + FN}$$

Both recall and precision are zero because there are no True-Positives (TP). Therefore, the F-score is $0 = 0/0$ for all datasets as depicted in Table 2.

4.1.3 n-gram Baseline

The n-gram baseline uses the TF-IDF for feature extraction and a SVC for classification.

- Feature Extraction: TF-IDF [35] is used to convert the sentences (features) into a numerical vector representation.
- Classification: To classify the sentences a linear SVC [36] is used.

4.2 Speeches and Debates

The data provided for task 1B consists of transcripted sentences from speeches and debates. There is no information about previous and following sentences. Therefore, there is no contextual information that could be used for the prediction of the true labels (also referred to as gold labels).

4.2.1 Example data

The English datasets that were provided contain sentences ("Text") with the corresponding gold label ("class_label") and id ("Sentence_id"). Table 3 provides a few examples taken from the Train dataset.

Sentence_id	Text	class_label
30313	And so I know that this campaign has caused some questioning and worries on the part of many leaders across the globe.	No
19099	Now, let's balance the budget and protect Medicare, Medicaid, education and the environment.	No
33964	I'd like to mention one thing.	No
16871	I must remind him the Democrats have controlled the Congress for the last twenty-two years and they wrote all the tax bills.	Yes
28916	I'm proud of the fact that violent crime is down in the State of Texas.	Yes
22058	If we're $4 trillion down, we should have everything perfect, but we don't.	Yes

Table 3: Shows example data from the English Train dataset. The "class_label" column shows the gold label corresponding to the sentence ("Text"). "Yes" means the sentence is checkworthy and "No" means the sentence is not checkworthy.

4.2.2 Comparing Dev and Dev-Test

We used the given baselines that were trained on the Train dataset and evaluated with the Dev and Dev-Test dataset separately. As shown in Table 4 the n-gram baseline model performs significantly better on the Dev-Test (0.821) than on the Dev (0.601) dataset.

Model	**1B - English - Dev**	**1B - English - Dev-Test**
Random Baseline	0.241	0.194
Majority Baseline	0.000	0.000
n-gram Baseline	0.601	0.821

Table 4: Shows the performance (F-score) of the baseline models (Random, Majority, n-gram) on the Dev and Dev-Test datasets.

5 Data Analysis

5.1 Labeling

In order to gain a more comprehensive understanding of how the data was labeled, we undertook some labeling ourselves. We created a script in order to efficiently label our data. The script

- selects 300 random sentences to label,
- prints a single sentence at a time to request the user's prediction and
- saves all predicted labels in a single *.tsv file.

Each of us labeled 300 randomly selected sentences both from the Dev and the Dev-Test dataset. We then analyzed the results to find the answers to the following:

- Labeling performance: How good were our labels compared to the gold labels?
- Inter-Annotater-Agreement: How much did we agree with each other?
- Model comparison: Were our labels better than the predictions from our models?

5.1.1 Labeling performance

We combined all our annotations into a single annotation using a majority voting. The goal was to find our best prediction or rather the prediction the majority of us (at least 2 out of 3) agreed on. To do that for a given sentence, we used the label that was selected by at least two annotators. The resulting predictions compared to the true labels are shown in Figure 3.

Table 5 provides an overview of how well we labeled the sentences in comparison to the provided gold labels. It shows that each of us managed to label the Dev dataset with an accuracy of at least 0.8.

Looking at the F-scores of our combined labels (Majority) for both the Dev and the Dev-Test dataset we can see that those are very similar to the scores of the n-gram Baseline in Table 4. The fact that we were only able to label the data as well as the given n-gram baseline shows that we didn't managed to label the data very well.

Table 5 also shows that we managed to label the Dev-Test data much better than the Dev data. This was also the case in the given baselines and will also be the case in the models we used going forward. It seems that the gold labels of the sentences in the Dev-Test dataset were easier to predict than the gold labels of other datasets.

Annotater	F-score		Accuracy	
	Dev	Dev-Test	Dev	Dev-Test
Pascal	**0.686**	0.829	0.820	0.907
Christoph	0.574	**0.839**	0.827	**0.923**
Karin	0.420	0.558	0.807	0.847
Majority	0.612	0.832	**0.843**	**0.923**

Table 5: Provides an overview of how well we labeled the sampled data both from the Dev and Dev-Test dataset. The performance measures used are the F-score and Accuracy. The best scores are marked as bold.

Figure 3 consists of all confusion matrices corresponding to our labeling task for both the Dev and the Dev-Test dataset. It shows that Karin (Dev: 8.67 % "Yes" labels) and Christoph (Dev: 16 % "Yes" labels) were much more conservative with labeling a sentence as checkworthy ("Yes") than Pascal (Dev: 32.67 % "Yes" labels). This is indicated by the sum of values on the right side of each matrix.

Figure 3: Shows the confusion matrices of our labels compared to the gold labels for both the Dev and Dev-Test dataset. The confusion matrix shows that all annotaters had a similar accuracy of at least 0.8 for both datasets. It also shows that Pascal labeled the most sentences as checkworthy while Karin labeled fewest sentences as checkworthy.

Table 6 provides a more simple overview of our labeling task. It shows how many sentences were labeled incorrectly by at least one person, by the majority and by all of us. On the Dev dataset there were 102 out of 300 sentences that were labeled incorrectly by at least one person. However, only 15 sentences were classified incorrectly by all of us.

	Dev	**Dev-Test**
Wrong by at least one person	102	71
Wrong by majority	47	23
Wrong by everyone	15	3

Table 6: Shows an overview of how many sentences (out of 300) were classified incorrectly by the labelers on both the Dev and Dev-Test dataset.

We went through several example sentences together for further analysis. When we had differences, we could come to an agreement after a short discussion in most cases. However, even after going through the sentences together and looking at the gold label we didn't agree with all of them. Table 7 shows two examples of sentences we did not agree with the gold label.

- The first sentence we did not consider to be checkworthy because the term "gone bad" seems to be difficult to fact check.
- The second sentence we labeled as checkworthy because it is clearly a claim that can be checked and we considered war to be an important topic.

Gold	**Labeled**	**Sentence**
Yes	No	Our housing programs have uh - gone bad.
No	Yes	Well, we, the living Americans, have gone through four wars.

Table 7: Shows two sentences that were labeled incorrectly by all of the annotaters. The column "Gold" shows the gold label and the column "Labeled" shows the label we consider to be appropriate.

5.1.2 Inter-Annotater-Agreement

The Inter-Annotater-Agreement (IAA) is a measure to show how well multiple annotaters agreed when labeling the same data. There are multiple metrics to calculate the annotater agreement as shown in Table 8. The most relevant score is the Fleiss Kappa as it is designed to be used with more than two annotaters. The Cohen Kappa and the Scott's Pi measure the agreement between raters. However, it is possible to calculate the agreement score for Cohen Kappa and Scott's Pi by calculating the average of all pairwise agreements (which is what we did to obtain the score for Cohen Kappa and Scott's Pi). [37], [38]

The score can range from -1 to 1. A score above 0 means there is an agreement (more than random) and a score of 1 means there is a perfect agreement. Table 9 shows one interpretation of the scores in Table 8. The scores ranging from 0.37 to 0.41 on the Dev and 0.54 to 0.55 on the Dev-Test dataset indicate a moderate to fair agreement between all the annotaters.

Metric	Score (Dev)	Score (Dev-Test)	Agreement
Cohen Kappa	0.41	0.55	Moderate
Fleiss Kappa	0.39	0.55	Fair-Moderate
Krippendorffs Alpha	0.38	0.55	Fair-Moderate
Scott's Pi	0.37	0.54	Fair-Moderate

Table 8: Shows an overview of IAA scores across different metrics for both the Dev and Dev-Test data we labeled. The IAA scores on the Dev data range from 0.37 to 0.41 and the IAA scores on the Dev-Test data are all either 0.54 or 0.55. Which means there is a moderate to fair agreement between the annotaters.

Kappa	Agreement
< 0	Less than chance
0.01 - 0.20	Slight
0.21 - 0.40	Fair
0.41 - 0.60	Moderate
0.61 - 0.80	Substantial
0.81 - 0.99	Almost perfect

Table 9: Shows an interpretation of the IAA Kappa scores [39]. This is just one interpretation for IAA scores, there are others as shown in [40].

5.1.3 Model comparison

After we had trained all individual models, we went ahead and compared their performance on the Dev dataset with our labels. To do that we plotted the ROC-curve as shown in Figure 4. It shows that all transformer models managed to produce better predictions, while the n-gram model showed a similar performance and only the word2vec model performed worse.

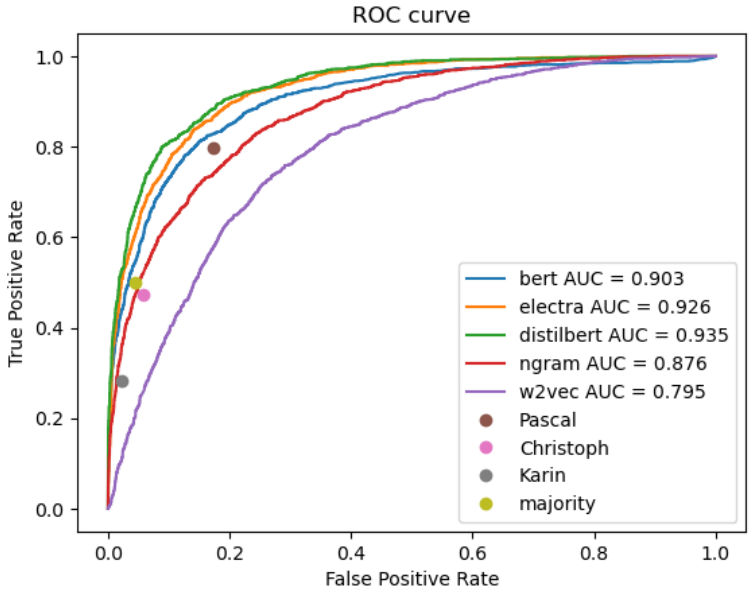

Figure 4: Provides a comparison between performance of the models we used and our own (Pascal, Christoph, Karin) labeling performance.

5.1.4 Learnings and Conclusion

In Section 5.1.1 we have shown that we weren't able to accurately predict the gold labels manually. Additionally, we have compared our performance to the performance of the transformer models we employed, which showed that the transformer models were able to make better predictions than we made when manually labeling the data, as outlined in 5.1.3. Lastly, we showed that there was only a fair to moderate agreement between us.

Labeling the data proved to be a difficult task, as we weren't able to make better predictions than either of the transformer models we used. We tried predicting the labels of a few sentences in pairs by discussing the sentences, but even then we were unable to predict some labels correctly.

6 Methods

In this section, we are going to discuss the methods we used throughout this project.

6.1 Approach

In the first part of this section, we present an overview of the steps we took throughout our process, arranged in chronological order. Subsequently, in the following subsections, we delve into a more comprehensive explanation of our actions. In Section 6.9 (Final model) we then present the model we used for the CheckThat!2023 submission and our final model.

1. Baselines: Initially we ran the baseline models provided for the CheckThat!2023 competition. This allowed us to establish a benchmark for comparison and evaluation.

2. Prototype: To tackle the task at hand, we began by developing a prototype system. For our prototype we used a pre-trained transformer model.

3. Labeling: To get a better idea as of how the given datasets were labeled we decided to take on the task of labeling ourselves on a subset of the data, as described in Section 5.1 (Labeling).

4. Transformer models: In order to explore different possibilities and improve upon the baseline results, we experimented with other pre-trained transformer models, to evaluate their suitability for the CheckThat!2023 competition, as explained in Section 6.4 (Transformer models).

5. Fine-tuning: In order to enhance the performance of the pre-trained models for our specific task, we engaged in fine-tuning the models using the training data provided for the CheckThat!2023 competition, as described in Section 6.4 (6.4).

6. Analysis and Comparison: Once we had trained and fine-tuned multiple transformer models, we conducted a performance analysis to compare their effectiveness, as explained in Section 6.8 (Analysis and Comparison).

7. Meta-Estimators: At this point we had three models that provided good results for the task. Our analysis had shown, that the different models made different mistakes. Therefore, we thought it might be a good idea to combine the results. To do this we experimented with different meta-estimators. Further explanation follows in Section 6.7 (Meta-Estimators).

8. Additional predictions: To improve the predictions of our meta-estimator, we added additional predictions. We added the n-gram baseline and a word2vec model, as well as predictions from OpenAI and ClaimBuster, as described in Section 6.6 (API models) and 6.3 (Basic models).

9. Hyperparameter optimization: Finally, we applied hyperparameter optimization to the transformer models we used as described in Section 6.4.5 (Hyperparameter Optimization).

10. Retraining of meta-estimator: In order to incorporate the enhanced hyperparameters, it was necessary to conduct a retraining process for the transformer models, utilizing these specific parameters. Additionally, the meta-estimator had to undergo retraining as well, incorporating the improved predictions from the transformer models, along with additional predictions from the preceding two steps.

6.2 Evaluation

To evaluate our models we utilized the Dev and Dev-Test datasets. In this section we have sometimes also included the Test dataset to provide a comparison. However, the Test dataset was not available prior to the CheckThat!2023 deadline and we were not able to use it to evaluate our results during our development process. The metric we used to measure the performance of the models was the F-score. This is because the CheckThat!2023 competition also uses the F-score for their rankings.

6.3 Basic models

6.3.1 n-gram

An n-gram model, introduced in section 4.1.3, predicts the next word by considering n preceding words. This model served as a baseline for the CheckThat! competition. Table 10 displays the performance of the n-gram model on the Dev, Dev-Test, and Test datasets.

Model	**F-Score**		
	Dev	Dev-Test	Test
n-gram	0.601	0.821	0.561

Table 10: Shows the performance of the n-gram model on the Dev, Dev-Test and Test dataset. The measurement that was used is the F-score.

6.3.2 word2vec

Word2vec, short for word to vector, is an algorithm introduced by Google in 2013 [41], [42] to generate vector representations of words from text. It begins by constructing a vocabulary from the training text data and then learns vector representations for each word based on their context. Words with similar meanings are represented by vectors that are close together, while words with different meanings are represented by vectors that are far apart.

To represent each sentence from a speech or debate transcript, we aggregated the word embeddings of individual words into a feature vector. This was done by computing the average of all individual word vectors within the sentence. We then trained a SVC classifier using these sentence vectors as input features and the corresponding outcomes as labels.

For our word2vec model we utilized the word2vec module from the gensim library[2]. We trained the word2vec model with C-SVC on the Train dataset using the library SVM from Sklearn, which is based on LIBSVM [43], for classification We used this model to predict the labels and their probabilities for the testing data.

Table 11 presents the performance of the word2vec model evaluated on the Dev, Dev-Test and Test datasets.

Model	**F-Score**		
	Dev	Dev-Test	Test
word2vec	0.542	0.706	0.615

Table 11: Shows the performance of the word2vec model on the Dev, Dev-Test and Test dataset. The F-score was utilized as the measurement for evaluating performance.

6.4 Transformer models

In general, we extensively relied on the Huggingface transformer library[3] throughout our project. This powerful library provided us with a wide range of transformer models, tokenizers, and utilities, simplifying the implementation and experimentation process.

For all transformer models, including BERT, ELECTRA, and DistilBERT, the prediction process involves initial tokenization of the input sentence using the respective tokenizer associated with the specific model. This tokenization step converts the sentence into vectors. Subsequently, these vectors are passed through the model to generate the desired prediction, an example can be found in the Appendix 3 . The same sequence of tokenization and feeding the vectors into the model is employed during the fine-tuning process of pre-trained models as well. In our case, all the models were trained on the Train dataset consistently.

6.4.1 BERT

We employed the BERT model [28], specifically bert-base-uncased [44], primarily as a vectorizer and for generating predictions in conjunction with a LinearSVC classifier. This approach was guided by a tutorial[4] that provided a framework for the implementation. As our work progressed, we discovered a more suitable pre-trained BERT model called textattack/bert-base-uncased-yelp-polarity [45], specifically designed for sequence classification tasks. This model demonstrated improved performance on the Dev-Test dataset compared to the bert-base-uncased model as shown in Table 12.

[2]https://github.com/RaRe-Technologies/gensim/blob/develop/gensim/models/word2vec.py
[3]https://huggingface.co/docs/transformers/index
[4]https://towardsdatascience.com/build-a-bert-sci-kit-transformer-59d60ddd54a5

Model	**F-Score**		
	Dev	Dev-Test	Test
bert-base-uncased with SVC	0.709	0.883	0.808

Table 12: Shows the F-Scores that were achieved on the Dev, Dev-Test and Test dataset by using the bert-base-uncased model with a SVC.

The performance of the fine-tuned BERT transformer model are shown in Table 13.

Model	**F-score**		
	Dev	Dev-Test	Test
BERT	0.707	0.893	0.758

Table 13: Shows the performance of the BERT transformer model on the Dev, Dev-Test and Test dataset. To produce these scores the fine-tuned BERT model was utilized. The measurement that was used is the F-score.

6.4.2 ELECTRA

In a subsequent development, we transitioned from utilizing the BERT model to leveraging the ELECTRA model. This decision was driven by the fact that the ELECTRA model, being newer, exhibited superior performance across various sequence classification tasks. We implemented an initial solution that involved fine-tuning the ELECTRA model, specifically the electra-base-emotion [46] variant, using the Train dataset. Subsequently, we discovered that the pre-trained model had six output labels, which did not align with our specific interest in only two labels. To address this, we made modifications to the output labels and retrained the model to cater to our specific classification requirements.

The performance of the fine-tuned ELECTRA transformer model are shown in Table 14.

Model	**F-score**		
	Dev	Dev-Test	Test
ELECTRA	0.732	0.939	0.783

Table 14: Shows the performance of the ELECTRA transformer model on the Dev, Dev-Test and Test dataset. To produce these scores the fine-tuned ELECTRA model was utilized. The F-score was utilized as the measurement for evaluating performance.

6.4.3 DistilBERT

In order to perform a comparative analysis with the existing models, we incorporated a third alternative model to assess its performance in relation to the previously implemented models. To diversify our approach from the prevalent use of RoBERTa in previous CheckThat! competitions, we conducted research to identify alternative models specifically suited for sequence classification tasks. The selection of the DistilBERT model distilbert-base-uncased [47] was based on its smaller size compared to the BERT model and its faster training capabilities. This advantage would allow us to conduct a greater number of experiments.

The performance of the fine-tuned DistilBERT transformer model are shown in Table 15.

Model	**F-score**		
	Dev	Dev-Test	Test
DistilBERT	0.757	0.926	0.821

Table 15: Shows the performance of the DistilBERT transformer model on the Dev, Dev-Test and Test dataset. To produce these scores the fine-tuned DistilBERT model was utilized. The F-score was utilized as the measurement for evaluating performance.

6.4.4 Fine-Tuning

To fine-tune the transformer models we used the Trainer API[5] from Huggingface. Before initiating the Trainer, it is necessary to create instance of TrainingArguments[6], which serve as input parameters for configuring the training process. The TrainingArguments are responsible for defining the training settings and specifications that dictate how the Trainer will train the model. Additionally, they determine the location where the trained model will be stored after the training process is completed. If you intend to release your models, an alternative option is to upload them directly to HuggingFace.

During the training process, we enabled the truncation and padding functionalities of the tokenizer. This ensures that input sequences are appropriately truncated or padded to a consistent length. Upon concluding the project, it came to our attention that we did not utilize truncation and padding during the prediction phase. It is possible that this omission has influenced the resulting scores.

The initial model we fine-tuned was the ELECTRA model. The ELECTRA model necessitated fine-tuning as the initial pre-trained model was specifically designed for sentiment analysis on tweets and generating emotion-based labels [46]. Following the fine-tuning process using the Train dataset, it was observed that only the first two labels exhibited higher values compared to the remaining labels. Initially, we disregarded the other four labels and directed our focus solely on the two relevant labels. Subsequently, we discovered that this approach would alter

[5]https://huggingface.co/docs/transformers/main/en/main_classes/trainer
[6]https://huggingface.co/docs/transformers/v4.29.1/en/main_classes/trainer#transformers.TrainingArguments

the output of the transformer-interpreter library and in general make the predictions worse. However, we later realized that by training the model anew and adjusting the num_labels parameter during the training process, we could effectively modify the model's output to just 2 labels. This modification resulted in an enhancement of the model's performance and led to an improved output from the transformer-interpreter library.

As depicted in the Table 12, it is evident that fine-tuning the models for a specific dataset may not always yield significant benefits or be worth the expended effort. In the BERT SVC model, we simply fitted the SVC onto the Train dataset without making any modifications to the underlying BERT model itself. By solely training the BERT model itself without incorporating an SVC, we noticed a minimal decrease in the model's performance compared to when it was used in conjunction with an SVC. The code for fine-tuning the models can be found in the Appendix 10.4.3 Fine-Tuning and Hyperparameter Optimization

6.4.5 Hyperparameter Optimization

To enhance the performance of the transformer models, we employed hyperparameter optimization techniques on the Train dataset. We used the hyperparameter search from the Trainer API of Huggingface[7] to improve the fine-tuning of the pre-trained models. We focused on the following four hyperparameters:

- learning-rate: is set to control how the gradients are updated
- per_device_train_batch_size: sets the size of the batch per training device e.g. CPU/GPU
- num_train_epochs: determines the number of iterations we go through the provided trainings dataset
- weight_decay: is set to control how the gradients are updated

We choose these particular hyperparameters because they can be seamlessly integrated into the TrainingArguments of the Trainer API, simplifying the implementation and configuration of the training process. Our objective was to optimize for the maximum F-score, which we calculated and used as a performance metric during the optimization process. For each model we conducted 20 trials to explore various configurations and settings. The parameters we ended up with are shown in Table 16.

[7] https://huggingface.co/docs/transformers/hpo_train

Hyperparameter	**ELECTRA**	**Distilbert**	**Bert**
Learning Rate	2.650e-05	2.251e-05	9.459-05
Weight Decay	91.942e-04	50.479e-04	2.737e-04
Batch Size	32	128	64
Number of Epochs	5	5	4

Table 16: Shows the values of each hyperparameter which was used to train the transformer models.

By utilizing the optimized hyperparameters, we were able to reduce the training time by a factor of 10. Table 17 shows the performance of our transformer models before and after the hyperparameter optimization. The specific parameters used for model fine-tuning are listed in Table 16. In most cases the hyperparameter optimization resulted in a higher F-score. The largest improvement can be observed on the Test dataset. The Electra model went from an F-score of 0.783 up to 0.857 and the BERT model went from an F-score of 0.758 up to 0.789. In the Dev-Test dataset the score of the BERT and DistilBERT models went up, while the score of the ELECTRA model dropped. This shows that despite conducting a hyperparameter search and employing improved training techniques, there is no guarantee that the final model will consistently outperform previous iterations. After completing the training of the transformer models, all the meta-estimators were retrained.

Model	**Data**	**F-score**	**F-score (Hyperparameter)**
DistilBERT	Dev	0.757	**0.765**
ELECTRA		0.732	**0.738**
BERT		0.707	**0.711**
DistilBERT	Dev-Test	0.926	**0.945**
ELECTRA		**0.939**	0.918
BERT		0.893	**0.904**
DistilBERT	Test	**0.821**	0.819
ELECTRA		0.783	**0.857**
BERT		0.758	**0.789**

Table 17: Shows the performance of transformer models with and without optimized hyperparameters on Test dataset

The code for hyperparameter optimization for the models can be found in the Appendix 10.4.3 Fine-Tuning and Hyperparameter Optimization.

Future Improvements

Given more time, we could increase the number of trials conducted, allowing us to explore a wider range of combinations and potentially discover better parameter settings. This expanded search space increases the likelihood of finding optimal hyperparameters for improved model performance. Additionally, we have the flexibility to fine-tune other hyperparameters or extend the hyperparameter search to the fitting function of the meta-estimators.

6.5 Word attributions

Transformer models are rather complex, and it is difficult to understand why a model predicted a certain text to be checkworthy. However, there are methods that allow us to show which words were important for the models decision. To determine this we used a library called transformer-interpret [48]. By getting a better understanding of the models we used, we wanted to find out why these models make mistakes. The goal was to find ways to improve the models based on our findings.

6.5.1 Transformer-Interpret

Transformer-interpret is built on top of the Captum [49] package, which is designed for model interpretability. Transformer-interpret is a library developed to provide transformer models with explainability. It provides an attribution score for each word in a given sentence and model. The attribution score indicates how much a word contributed to the prediction. To obtain these scores the model, a tokenizer and the text need to be provided. An example of these attribution scores are shown Listing 1 for the sentence: "we're consuming 50 percent of the world's cocaine.". A positive score indicates that token contributed towards the predicted label, which in this case is "Yes".

```
// [(token1, score1), (token2, score2),...]
[
    ('[CLS]', 0.0),
    ('we', 0.11201885420961634),
    ("'", 0.25647434238794753),
    ('re', 0.05324198527316812),
    ('consuming', -0.020812193503298385),
    ('50', 0.6058883371072958),
    ('percent', 0.6342749327159223),
    ('of', 0.33569119162550515),
    ('the', 0.03863532323688133),
    ('world', 0.08869171513703551),
    ("'", 0.00959359866329527),
    ('s', 0.11580983965201863),
    ('cocaine', 0.05045720677664899),
    ('.', 0.10441309212831708),
    ('[SEP]', 0.0)
```

```
]
```

Listing 1: Shows the attribution scores of the sentence "we're consuming 50 percent of the world's cocaine."

To visualize these attribution scores we utilized the visualizer from transformer-interpret as shown in Figure 5 for the previously used sentence. Positive scores are highlighted with a green background and negative scores with a red background. Therefore, green means that the word contributed positively towards the predicted label and red means a word contributed negatively towards the predicted label. The more intensive the color the higher the contribution. The attribution score shown in 5 is equivalent to the sum of all attribution scores of this sentence.

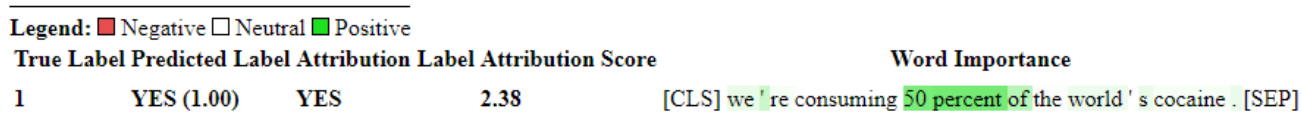

Figure 5: Shows the word attribution visualization provided by the transformer-interpret visualizer. The true label is the label that was to be predicted. The attribution score is equivalent to the sum of all scores. The scores are shown as colors in the "Word Importance" column. Green means that the word contributes positively towards the predicted label and red means the opposite.

To get a better overview and comparison between the models we used, we implemented our own visualization, as shown in Figure 6. In addition to what the visualization from transformer-interpret shows, we also included the model that was used. The idea was to be able to directly compare the visualization of all the transformer models.

id	label	model	prediction	probability	attribution score	word attribution (colored)
80	Yes	electra	Yes	0.984	3.33	[CLS] we ' re consuming 50 percent of the world ' s cocaine . [SEP]
80	Yes	bert	Yes	0.997	1.74	[CLS] we ' re consuming 50 percent of the world ' s cocaine . [SEP]
80	Yes	distilbert	Yes	0.998	2.38	[CLS] we ' re consuming 50 percent of the world ' s cocaine . [SEP]

Figure 6: Shows an example of our word attribution visualization. Shows the gold label ("label"), the "model" that was used, the predicted label "prediction" and its corresponding probability. It also shows the attribution score (sum of all attribution scores of the sentence) and the word attribution visualization.

Captum

In our case the LayerIntegratedGradients [8] algorithm from Captum was applied. The IntegratedGratients algorithm was originally mentioned in [50].

6.5.2 Word attribution analysis

To analyze these attributions we have made a script that creates a Pandas dataframe with an overview of information per token. The code that was used to produce this overview is shown in

[8] https://captum.ai/docs/extension/integrated_gradients

Section 10.3 of the Appendix. The overview as visualized in Figure 7 and 8 shows the following information:

- *count_neg*: number of occurrences of the token with a negative score (< 0)
- *avg_neg*: average score of tokens with a negative score (< 0)
- *count_pos*: number of occurrences of the token with a positive score (> 0)
- *avg_pos*: average score of tokens with a positive score (> 0)
- *count_all*: number of occurrences of the token, includes tokens with neutral score ($= 0$)
- *avg*: average score of all tokens
- *avg_abs*: average of absolute score of all tokens

Negative numbers indicate that the token contributed towards the label "No" (not checkworthy) and positive numbers indicate that a token contributed towards the label "Yes" (checkworthy). Figure 7 shows that the token "?" heavily indicates that a sentence is not checkworthy. This makes sense considering that a question is not a claim.

Token	avg_neg	count_neg	avg_pos	count_pos	avg_abs	avg	count_all
?	-0.72	153	0.42	2	0.70	-0.69	158
admire	-0.68	6	0.00	0	0.58	-0.58	7
appreciate	-0.61	8	0.18	1	0.56	-0.52	9
attitude	-0.58	4	0.00	0	0.46	-0.46	5
need	-0.51	128	0.22	10	0.46	-0.43	149
essential	-0.42	5	0.00	0	0.42	-0.42	5
feeling	-0.42	5	0.00	0	0.42	-0.42	5
worry	-0.41	6	0.00	0	0.41	-0.41	6
cannot	-0.41	26	0.00	0	0.41	-0.41	26
vital	-0.40	6	0.00	0	0.40	-0.40	6
question	-0.45	75	0.10	3	0.41	-0.40	84
##ful	-0.50	4	0.00	0	0.40	-0.40	5
let	-0.50	143	0.11	13	0.41	-0.40	177
thank	-0.44	27	0.13	1	0.40	-0.39	30
hope	-0.45	45	0.19	4	0.42	-0.39	50

Figure 7: Shows the statistics from our token attributions on the Dev-Test dataset with the DistilBERT model. The information in the red box shows statistics across all words with a negative score, the green box shows all information with a positive score and the blue box shows statistics across all scores. The table is sorted by the column "avg" (ascending). The "avg" column shows the average of all scores of the token in this row. The token with the most negative average score is "?". Only tokens with at least 5 occurrences are shown in this table.

Figure 8 shows tokens that indicate a positive contribution towards the label "Yes" (checkworthy). The token "democrat" for example generally indicates that a sentence is checkworthy.

Token	avg_neg	count_neg	avg_pos	count_pos	avg_abs	avg	count_all
funding	0.00	0	0.32	6	0.19	0.19	10
bottom	0.00	0	0.32	6	0.19	0.19	10
democrat	-0.09	1	0.22	10	0.21	0.19	11
indicated	0.00	0	0.40	7	0.17	0.17	16
met	-0.04	3	0.30	11	0.18	0.17	19
russians	0.00	0	0.30	10	0.17	0.17	18
stood	-0.11	1	0.21	9	0.17	0.15	12
leaving	-0.26	1	0.29	6	0.20	0.15	10
worked	-0.27	6	0.40	15	0.26	0.15	30
recently	0.00	0	0.52	4	0.14	0.14	15
##ot	0.00	0	0.16	11	0.14	0.14	13
difference	-0.08	14	0.31	22	0.19	0.13	42
east	-0.01	2	0.21	18	0.14	0.13	28
##est	0.00	0	0.44	4	0.13	0.13	13
seniors	-0.05	3	0.24	12	0.13	0.12	23

Figure 8: Shows the statistics from our token attributions on the Dev-Test dataset with the DistilBERT model. The information in the red box shows statistics across all words with a negative score, the green box shows all information with a positive score and the blue box shows statistics across all scores. The table is sorted by the column "avg" (descending). This column shows the average of all scores of the token in this row.

6.5.3 Learnings

We have shown that valuable information about the behavior of a transformer model can be derived by using Explainable AI [51]. More specifically it is possible to show why a certain sentence is labeled as checkworthy by looking at the word attributions. By analyzing these word attributions over an entire dataset we were able to show which tokens generally contribute to which label.

6.6 API models

6.6.1 OpenAI

When working with OpenAI, we employed various methods to generate predictions regarding the check-worthiness of the provided sentences.

Chat

Initially, we utilized the web interface of ChatGPT to generate predictions. We attempted the approach of inserting sentences in large batches after providing an initial instruction to consistently generate responses in the same format. Completion models exhibit a tendency to be verbose and generate responses that surpass the specific scope of the prompt. We started with few shot prompting right away as this seemed to be the most promising approach.

The initial segment of the prompt served as an introductory statement outlining our intended objectives. Following that, we included the description for evaluating the checkworthiness, which was provided by CheckThat!2022 [7] to give it more context. Subsequent, we proceeded to include a demonstration of how the prompt and the appropriately formatted response should

appear. Concluding the prompt, we included a statement for the AI agent to generate, indicating its comprehension and understanding of the given task. The complete prompt can be found in section 10.5.1 ChatGPT Prompt of the Appendix.

The limited context provided by a single chat history proved insufficient for handling large batches of data. On our initial attempt to input the first batch of the Dev dataset into the AI model, we encountered a limitation wherein the desired format was not preserved when attempting to process more than 100 sentences simultaneously. Consequently, we transitioned to using the API, which facilitated the processing of smaller batches or even individual sentences with greater ease.

API

Initially, our focus was on utilizing the gpt-3.5-turbo model [9] for chat completion, as it represents the latest and most cost-effective option available from OpenAI. The pre-existing instruction segment developed for ChatGPT was utilized as the system message, and two exemplar user queries along with their respective AI model responses were incorporated as seen in section 10.5.2 API code of the Appendix. Following numerous iterations with the API, we accomplished the objective of obtaining a consistent output that adhered to the desired JSON format.

The first few predictions have been consistently successful, generating meaningful results in the desired format that align with our expectations. However, a misinterpretation occurred when the gpt-3.5-turbo model mistakenly took the sentence "That answer was about as clear as Boston harbor" as a request to modify the initial system prompt. The model deviated from adhering to the desired JSON format and instead attempted to defend the correctness of the initially submitted prompt. This phenomenon is called prompt injection, wherein the system prompt is disregarded following a new user input, is well described here [52]

Due to the complexity of resolving this issue, we opted to transition to utilizing the text-davinci-003 model as an alternative approach. The text-davinci-003 model exhibited resilience against adversarial attacks, as all requests were successfully processed. In some cases the model replied in the incorrect format which caused an error in our script and we just had to restart it and to continue after the last successful sentence. The predictions generated by the API exhibited relatively poor scores in terms of F-score. The F-score on the Dev-Test dataset was 0.632 as shown in Table 18, is worse than the performance of the baseline n-gram model. In the Section 8.2.5 OpenAI API of the Conclusion and Discussion we will delve into potential strategies for enhancing and improving the obtained results.

[9] https://platform.openai.com/docs/guides/gpt/chat-completions-api

Model	**F-score**		
	Dev	Dev-Test	Test
OpenAI API	0.504	0.632	0.479

Table 18: Shows the F-scores that were achieved on the Dev, Dev-Test and Test dataset by using the OpenAI API.

6.6.2 ClaimBuster API

ClaimBuster provides a system that was trained on US Debates, Twitter tweets and other sources and is able to predict checkworthiness of sentences [53]. ClaimBuster provides an API endpoint ("/api/v2/score/text/<input_text>") that predicts how check-worthy a text is. We used this API to predict the checkworthiness of the Dev dataset sentences. To make these predictions the endpoint uses the adversarially trained BERT ClaimSpotter algorithm [54].

Table 19 shows the performance of the predictions obtained by using the ClaimBuster API. On the Dev dataset the F-score was 0.574 which is worse than F-score of the n-gram baseline. In section 6.7 we will show how utilized these scores.

Model	**F-score**		
	Dev	Dev-Test	Test
ClaimBuster API	0.574	0.706	0.694

Table 19: Shows the F-scores that were achieved on the Dev, Dev-Test and Test dataset by using the ClaimBuster API.

6.7 Meta-Estimators

By comparing the results of different models we experimented with, we noticed that the models made different mistakes. Therefore, we thought it might make sense to somehow combine these models to obtain a single best prediction.

The first approach to this was to take the binary prediction of all models and use a majority voting classifier to make a final prediction. Our second approach was to use the resulting probabilities of all models to train another classifier. Hence, it was imperative to ensure that every preceding classifier generated a probability distribution associated with the assigned label for a given sentence.

We experimented with these four different classifiers:

- Bagging Classifier: sklearn.ensemble.BaggingClassifier [10]

[10] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html

- Random Forest Classifier: sklearn.ensemble.RandomForestClassifier [11]
- Logistic Regression: sklearn.linear_model.LogisticRegression [12]
- Logistic Regression with Cross-Validation: sklearn.linear_model.LogisticRegressionCV [13]

The Bagging Classifier is an ensemble algorithm that operates by creating subsets of the original training set and subsequently combining the individual predictions from these subsets to generate a final prediction. In the Bagging Classifier, we utilized a Logistic Regression Classifier as the base estimator, while maintaining the default settings for the remaining parameters. The Random Forest Classifier (RFC) is another ensemble algorithm that combines multiple individual predictions from different models or decision trees. Due to the absence of the random_state parameter setting in our approach, certain results may exhibit slight variations and become more challenging to reproduce consistently. The inherent randomness within the classifier's initialization process contributes to these subtle differences in outcomes. The Logistic Regression and the Logistic Regression with cross-validation have both been initialized with the default settings for the parameters.

Due to the meta-estimators' rapid fitting capability on the predictions generated by other models, we aimed to conduct an extensive testing phase, exploring as many combinations and configurations as feasible.

Table 20 shows the performance of our meta-estimator on the Dev-Test dataset. The performance of the meta-estimator is similar to the performance of the DistilBERT and ELECTRA models. The Bagging Classifier has an F-score of 0.932 which is 0.006 lower than the F-score of the ELECTRA model.

Meta-Estimator	**F-score**
Random Forest Classifier	0.915
Bagging Classifier	**0.932**
Logistical Regression	0.928
Logistical Regression with cross validation	0.930

Table 20: Presents a comparison of the meta-estimators' performance on the Dev-Test dataset.

As depicted in Table 21, it is evident that the performance of the meta-estimators does not consistently improve with an increasing number of estimators. The impact of increasing the number of estimators on performance is dependent on the dataset. As depicted in Table 22, it is evident that the number of estimators employed can indeed have a substantial impact

[11] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
[12] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
[13] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV

on the performance of the meta-estimators as it can lead to a notable improvement in the F-score, potentially increasing it by up to 0.02. Given that the performance improvement on one dataset outweighed the performance decline on the other dataset, we maintain the belief that it is beneficial to include as many estimators as possible. The cumulative effect of incorporating multiple estimators is anticipated to yield an overall improvement in the F-score across various datasets.

Meta-Estimator	**Transformers**	**Transformers + word2vec + n-gram**	**All**
Random Forest Classifier	**0.944**	0.938	0.942
Bagging Classifier	0.947	**0.949**	0.943
Logistical Regression	**0.949**	**0.949**	0.943
Logistical Regression with cross validation	**0.949**	0.947	0.943

Table 21: Presents a comparison of the meta-estimators' performance on the Dev-Test dataset. It showcases the performance achieved when trained using different inputs, including: solely the predictions from transformer models, predictions from both transformer models and basic models (word2vec and n-gram), and all predictions including API predictions.

Meta-Estimator	**Transformers**	**Transformers + word2vec + n-gram**	**All**
Random Forest Classifier	0.793	0.800	**0.806**
Bagging Classifier	0.804	0.804	**0.824**
Logistical Regression	0.804	0.811	**0.824**
Logistical Regression with cross validation	0.804	0.791	**0.817**

Table 22: Presents a comparison of the meta-estimators' performance on the Test dataset. It showcases the performance achieved when trained using different inputs, including: solely the predictions from transformer models, predictions from both transformer models and basic models (word2vec and n-gram), and all predictions including API predictions.

6.8 Analysis and Comparison

In this section we first provide information about the analysis steps we have taken throughout the project. Additionally we provide and overview of the models we utilized and compare their performance.

6.8.1 Analysis

Initially we tried analyze the results of our transformer models manually. To do so, we separated our predictions into four subsets: True-Positive (TP), False-Positive (FP), True-Negative (TN) and False-Negative (FN). Our primary focus was on sentences that were predicted incorrectly (FP and FN). During this process, we took notes and documented common characteristics among the incorrectly classified sentences. For instance, we observed a notable occurrence of

sentences with indirect facts, such as "He said that...," in false predictions from both ELECTRA and BERT.

Reviewing these predictions and taking notes gave us a better overall understanding of how the data was labeled and the types of mistakes made by the models. However, it did not offer insights into the reasons behind these mistakes. Therefore, we decided to utilize a library called transformer-interpret, as detailed in Section 6.5.1 to help us understand why a sentence was classified in a particular manner.

Additionally, we compared the overlap of the predictions from different models. We observed that different models made different mistakes. Table 23 demonstrates that the ELECTRA model (742) and the BERT model (774) made over 700 incorrect predictions each. However, the number of sentences that were predicted incorrectly by both models (489) was below 500. Consequently, more than 35 % of the incorrect predictions could potentially be eliminated, if both model could be combined perfectly. This conclusion led us to experiment with meta-estimators, which are explained in Section 6.7.

	Dev - Incorrectly predicted sentences (FP + FN)
BERT	774
ELECTRA	742
BERT + ELECTRA	489

Table 23: Shows the number of sentences that were predicted incorrectly by the BERT model, ELECTRA model and both. The predictions were done on the Dev dataset.

6.8.2 Comparison

In the Dev dataset, the overall best performance was observed among the transformer models. Among them, DistilBERT achieved the highest F-score of 0.765, followed by ELECTRA with 0.738, and BERT with a score of 0.711. All other models performed at least 0.1 worse than the BERT model. On the Dev-Test dataset DistilBERT again performed best with an F-score of 0.945, followed by the meta-estimators with 0.942 and 0.943. On the Test dataset ELECTRA outperformed all other models by a margin of 0.03. Logistic Regression followed closely, slightly ahead of DistilBERT.

Model	**F-score**		
	Dev	Dev-Test	Test
BERT	0.711	0.904	0.789
ELECTRA	0.738	0.918	**0.857**
DistilBERT	**0.765**	**0.945**	0.819
n-gram	0.601	0.821	0.561
word2vec	0.542	0.706	0.615
OpenAI API	0.504	0.632	0.479
ClaimBuster API	0.574	0.706	0.694
Random Forest Classifier		0.942	0.806
Bagging Classifier		0.943	0.811
Logistical Regression		0.943	0.824
Logistical Regression with cross validation		0.943	0.817

Table 24: Shows all the F-scores for each model on all datasets. On the Test dataset the performance of the ELECTRA model was the best with 0.857 while the OpenAI API was the worst with 0.497

In Figure 9 we compare the confusion matrices of all the best and worst models with each other. We are showcasing the OpenAI API Model (gpt) as it was the worst model of them all with a difference of almost 0.4 to the best-performing ELECTRA model. In the confusion matrix you can see, that the OpenAI API model misclassified slightly over 30 % of all sentences with 10 % labeled as wrongly checkworthy and almost 20 % as wrongly not checkworthy. This indicates that either the LLM did not fully comprehend the task or it was unable to fulfill it effectively. Comparing the ELECTRA and DistilBERT models, which were the top performers among the transformer models, we find that they exhibit similar performance to the Logistic Regression meta-estimator. While the Logistic Regression model mislabeled fewer sentences as wrongly checkworthy (FP) compared to the transformer models, it labeled more sentences as wrongly negative (FN). Considering an unseen dataset, the Logistic Regression model is likely to provide more consistent predictions compared to the other models, as it consistently ranks among the top performers across all datasets.

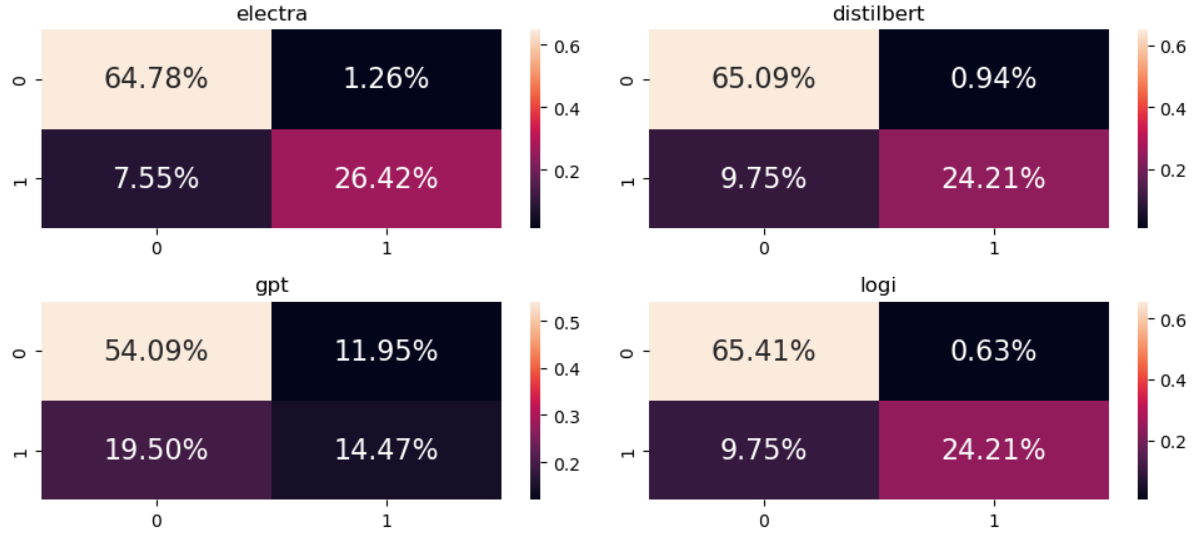

Figure 9: Shows the confusion matrix of the different models on the Dev-Test dataset

6.9 Final model

In this section we are going to provide an overview of our final model.

6.9.1 Overview

Figure 10 shows the training process we used to obtain our final model.

1. Train models with Train dataset to obtain trained / fine-tuned models.
   - Fine-tuning of transformer models (ELECTRA, BERT, DistilBERT)
   - Training of basic models (n-gram and word2vec)
2. Make predictions on the Dev dataset to obtain a single probability for each model.
   - Make predictions with fine-tuned transformer models
   - Make predictions with the basic models
   - Make predictions with OpenAI API
   - Make predictions with Claim-Buster API
3. Train meta-estimators based on all predicted probabilities to obtain the final model.
4. Make predictions on the Dev-Test dataset to see how the meta-estimators performed on the Dev-Test dataset.

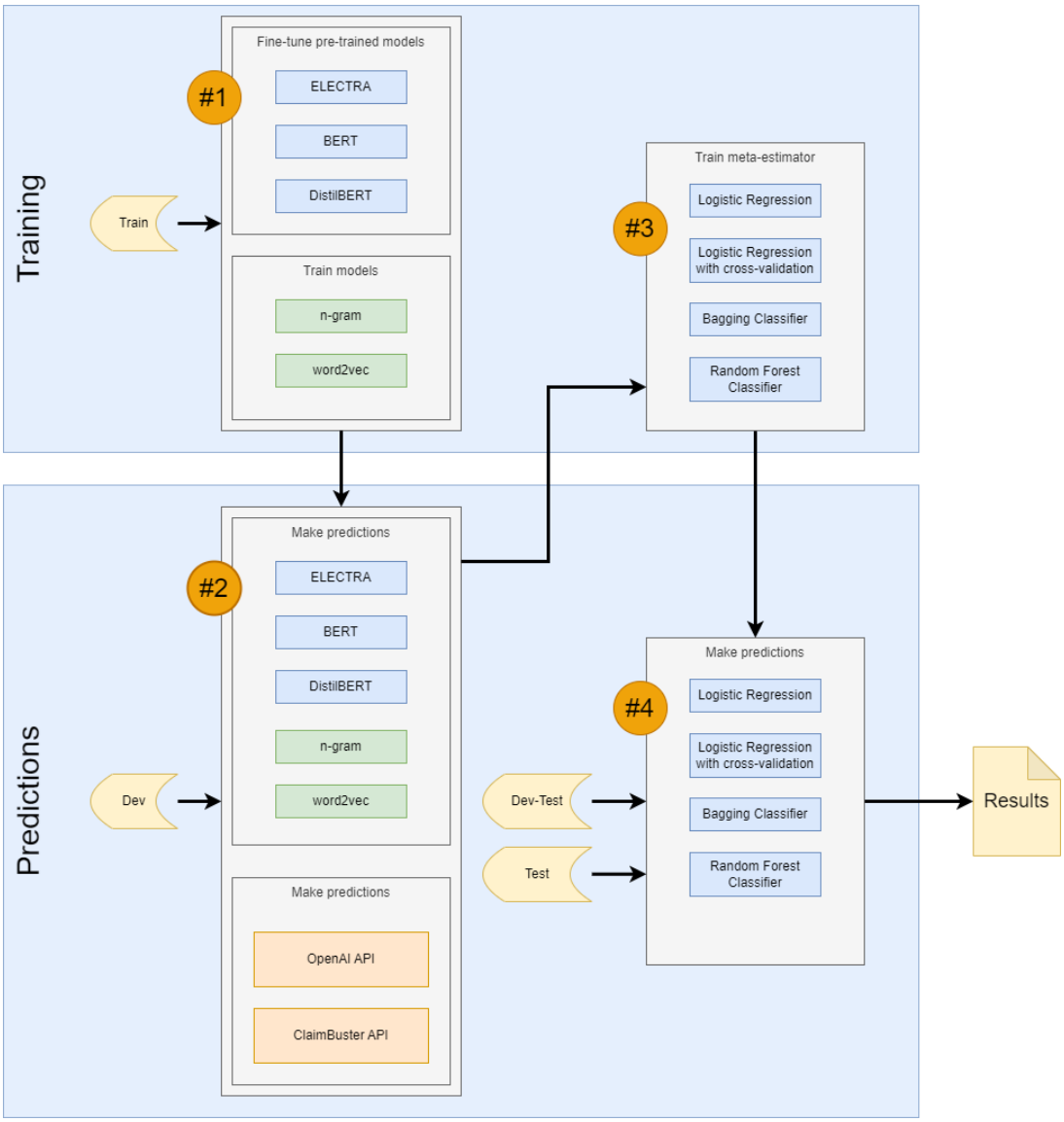

Figure 10: Shows the entire training and prediction process to obtain the final results. First the Train dataset is utilized for training and fine-tuning the models. Subsequently, the resulting models are employed to make predictions on the Dev dataset. The probabilities from these predictions are all passed on and used to train the meta-estimators. Lastly these meta-estimators are utilized to classify the sentences of the Dev-Test and Test dataset to obtain the final predictions.

7 Results

In this section we are going to present the results we were able to achieve in the CheckThat!2023 competition as well as our final results evaluated on the Test dataset.

7.1 CheckThat! Lab

Our Team name for the CheckThat!2023 submission was Pikachu. We selected that name because at the time of registration we have been heavily working on the ELECTRA model. The submission was evaluated on the Test dataset, which was released a week prior to the submission deadline. The Test dataset consists of 318 sentences and without the corresponding true labels. After the competition was finished the true labels have also be released.

The pipeline we used to achieve these results in the CheckThat!2023 lab did not include any prediction from API models as we had only done them after the CheckThat!2023 submission. The hyperparameter tuning was also done after the CheckThat!2023 submission. Aside from that we used the Pipeline as shown in Figure 10 of Section 6.9.

7.1.1 Competition results

There were 11 teams who participated in task 1B English. We managed to place 10th with an F-score of 0.767 as shown in 11, outperforming the random baseline (F-score of 0.462) by a margin of 0.305. The best performing team was OpenFact with an F-score of 0.898.

1B English

	Team	F1
1	OpenFact	0.898
2	Fraunhofer SIT	0.878
3	Accenture	0.860
4	NLPIR-UNED	0.851
5	ES-VRAI	0.843
6	Z-Index	0.838
7	CSECU-DSG	0.834
8	FakeDTML	0.833
9	DSHacker	0.819
10	Pikachu	0.767
-	UGPLN y SINAI	0.757
	Baseline	0.462

Figure 11: Shows the results and ranking of the CheckThat!2023 lab. We (Pickachu) received an F-score of 0.767 and placed 10th out of 11 teams. The best team OpenFact managed to achieve an F-score of 0.898.

7.1.2 All Submissions

The CheckThat!2023 competition allowed us to do multiple submissions. However only the last one would be considered for the final ranking, as shown in Figure 11. Table 25 shows the results of all the models we submitted to the CheckThat!2023. The last model we submitted was the logistical regression meta-estimator with cross validation (in bold). Had we submitted the DistilBERT predictions as our final submission, we would have achieved one rank higher in the competition standings.

Model Description	**Type**	**F1**
DistilBERT	Transformer	0.821
Electra	Transformer	0.783
Bagging Classifier	Meta-Estimator	0.767
Logistical Regression with cross validation	Meta-Estimator	0.767
Logistical Regression	Meta-Estimator	0.767
BERT	Transformer	0.758
Random Forest Classifier	Meta-Estimator	0.749
Word2vec	Basic Model	0.615
n-gram	Basic Model	0.561

Table 25: Shows the results of all submitted models to the CheckThat!2023 competition.

7.2 Final results

Table 26 shows all the models performances on the Test dataset ordered by score. On the Test dataset ELECTRA outperformed all other models. However this is only the case on the Test dataset, as depicted in Table 24 of Section 5.1.3, other models performed better on other datasets. This is why we would recommend to use the Logistic Regression meta-estimator model as it seems to be the most consistent performer.

Model Description	**Type**	**F-score**
ELECTRA	Transformer	**0.857**
Logistical Regression	Meta-Estimator	0.824
DistilBERT	Transformer	0.819
Logistical Regression with cross validation	Meta-Estimator	0.817
Bagging Classifier	Meta-Estimator	0.811
Random Forest Classifier	Meta-Estimator	0.806
BERT	Transformer	0.789
Word2vec	Basic Model	0.615
n-gram	Basic Model	0.561
Claimbuster API	API Model	0.694
OpenAI API	API Model	0.479

Table 26: Provides an overview of the performance of all models evaluated on the Test dataset ordered by F-score.

8 Conclusion and Discussion

8.1 Overview

In this paper, a system has been developed to predict the checkworthiness of sentences derived from political speeches and debates. Multiple models have been trained and fine-tuned specifically for the task of checkworthiness prediction. The predicted probabilities generated by these models are then utilized to generate a unified prediction through the implementation of meta-estimators.

As detailed in Section 7.1, our system attained the 10th place out of 11 teams in the CheckThat!2023 competition, with an F-score of 0.767. Following the submission of CheckThat!2023, we made additional enhancements to our system by applying hyperparameter optimization to the transformer models and using additional predictions from the ClaimBuster API and OpenAI API. This improvement resulted in a significant increase in the F-score, raising it from 0.767 to 0.817 for the Logistic Regression with cross validation model.

Nevertheless, it is important to acknowledge that the top-performing models employed for these predictions already had rather high F-scores. The best transformer model ELECTRA had an F-score of 0.857, which is significantly better than the prediction of the meta-estimator in our final system.

8.2 Potential improvements and Learnings

In this section we are going to discuss what could potentially be done to improve our results. A good starting point for improvements will also be the overview paper of all the other solutions of the CheckThat!2023 competition.

8.2.1 Preprocessing

In our system, we directly input the data into our models without applying any preprocessing steps to the given data. This approach was adopted because the transformer models already incorporate a built-in tokenizer that handles the necessary preprocessing tasks. To maintain consistency and facilitate comparisons across different methods, we decided not to preprocess the data for other models either.

Data augmentation could be used to generate additional training data, for example by exchanging words with synonyms or by applying back translation. This method has previously successfully been implemented for this task in [19]–[21].

As described in Section 6.4.4 we only used truncation and padding in the prediction process but not in the training process. This may have influenced our results. To improve this the truncation and padding should be applied both in the training and prediction process.

8.2.2 Cross validation

To train and evaluate our models we used the predefined datasets Train, Dev and Dev-Test as they were provided.

During our analysis, we observed that all models trained on the Train dataset exhibited significantly better performance when evaluated on the Dev-Test dataset compared to the Dev dataset. Therefore, we concluded that there is a significant difference between the two datasets Dev and Dev-Test. However, we have been unable to pinpoint the exact reason behind this observations. The fact that the results on the Dev-Test dataset were much better than on any other dataset lead us to believe that our predictions might have been overfitted. To mitigate the issue of overfitting and potentially enhance the performance of our models one approach could involve combining all three predefined datasets into a unified dataset and shuffle it. This new dataset could then be used in a k-fold cross validation procedure [55] to improve the models performance.

8.2.3 Meta Estimators

As describe in Section 6.7 we experimented with four different meta-estimators to obtain a single prediction from all predicted probabilities. We choose multiple meta-estimators to be able to compare their performance. However, it is worth noting that three out of the four estimators we employed utilized the same method, namely Logistic Regression. It would make sense to experiment with more different meta-estimators. Conducting further experimentation with a broader selection of meta-estimators would be reasonable.

We were able to improve the prediction of our meta-estimator by adding additional predictions. Even adding bad models helped to improve the prediction of our meta-estimators. Therefore it would make sense to add predictions from other models (transformer or basic models), that have not been utilized thus far.

8.2.4 Hyperparameter optimization

We have applied hyperparameter optimization to the transformer models. However, we did not have much time to experiment with too many parameters. Doing this could lead to an improvement in performance of our transformer models. Additionally, it would make sense to find optimal parameters not only for the transformer models but also for the n-gram and word2vec models, as well as for the meta-estimators.

8.2.5 OpenAI API

The performance of the OpenAI API has exhibited poor results. There are multiple approaches to improve the performance of the OpenAI API model. First you could look into adding more examples of sentences, create clearer instructions about the task or define the context around checkworthiness better. Second, it would make sense to use other prompting techniques as already mentioned in Section 3.4.3. We could also use Chain-of-Thought Prompting [56] as

this has also proven to be quite effective in making the decisions of LLMs better. We also did not try the ReAct [57] or the CARP [58] methods as these came to our attention at a later time, when we already had our predictions made. Given these alternative prompting techniques, it would be prudent to explore whether the gpt-3.5-turbo model is able to make predictions. Lastly, there would also be the possibility to fine-tune the OpenAI models itself. Considering the potential high costs involved, we made the decision not to pursue this approach.

8.2.6 Analysis

In order to gain more insight into the decision-making process of the transformer models we utilized, we employed transformer-interpret, a library specifically designed for this purpose. We built a script that summarizes these decisions per token as described in Section 6.5.1. To further enhance our the transformer models, it is recommended that this information be analyzed, and the acquired insights should then be utilized to improve the models.

8.2.7 Labeling

To obtain a more precise comprehension of the data labeling process, we took the initiative to label a portion of the data ourselves, as described in Section 5.1. Subsequently, we compared our labels with the gold labels and achieved an accuracy of approximately 0.8 and an F-score of around 0.6 when compared to the gold labels. These results indicate that the predictions from our final model were significantly better than own predictions when compared to the gold labels.

There were several occurrences where our team members held contrasting viewpoints regarding the classification of specific sentences, which is worth highlighting. This demonstrates the challenging nature of predicting checkworthiness, as it is not always clear when exactly a sentence should be considered checkworthy. Additionally, different annotators may not consistently agree on whether a sentence should be considered checkworthy.

9 Indices

References

[1] H. Allcott and M. Gentzkow, "Social Media and Fake News in the 2016 Election," *Journal of Economic Perspectives*, vol. 31, no. 2, pp. 211–236, May 2017, ISSN: 0895-3309. DOI: 10.1257/jep.31.2.211. (visited on 05/08/2023).

[2] J. Y. Cuan-Baltazar, M. J. Muñoz-Perez, C. Robledo-Vega, M. F. Pérez-Zepeda, and E. Soto-Vega, "Misinformation of COVID-19 on the Internet: Infodemiology Study," *JMIR Public Health and Surveillance*, vol. 6, no. 2, e18444, Apr. 2020. DOI: 10.2196/18444. (visited on 05/08/2023).

[3] J. Daxenberger, S. Eger, I. Habernal, C. Stab, and I. Gurevych, "What is the Essence of a Claim? Cross-Domain Claim Identification," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, Comment: Published at EMNLP 2017: http://www.aclweb.org/anthology/D/D17/D17-1217.pdf, 2017, pp. 2055–2066. DOI: 10.18653/v1/D17-1218. arXiv: 1704.07203 [cs]. (visited on 02/28/2023).

[4] S. E. Toulmin, *The Uses of Argument*, Updated ed., reprinted. Cambridge: Cambridge University Press, 2008, ISBN: 0-521-82748-5.

[5] N. Ferro, *>Conference and Labs of the Evaluation Forum (CLEF)*, https://www.clef-initiative.eu/. (visited on 03/07/2023).

[6] *CheckThat!* https://checkthat.gitlab.io/clef2023/task1/. (visited on 03/07/2023).

[7] P. Nakov, A. Barrón-Cedeño, R. Míguez, *et al.*, "Overview of the CLEF-2022 CheckThat! Lab Task 1\\ on Identifying Relevant Claims in Tweets,"

[8] M. Cieliebak, O. Dürr, and F. Uzdilli, "Meta-Classifiers Easily Improve Commercial Sentiment Detection Tools," in *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, Reykjavik, Iceland: European Language Resources Association (ELRA), May 2014, pp. 3100–3104. (visited on 05/11/2023).

[9] M. Sundriyal, A. Kulkarni, V. Pulastya, M. S. Akhtar, and T. Chakraborty, *Empowering the Fact-checkers! Automatic Identification of Claim Spans on Twitter*, Comment: Accepted at EMNLP22. 16 pages including Appendix, Oct. 2022. arXiv: 2210.04710 [cs]. (visited on 02/07/2023).

[10] R. Levy, Y. Bilu, D. Hershcovich, E. Aharoni, and N. Slonim, "Context Dependent Claim Detection," in *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, Dublin, Ireland: Dublin City University and Association for Computational Linguistics, Aug. 2014, pp. 1489–1500. (visited on 02/07/2023).

[11] I. Jaradat, P. Gencheva, A. Barron-Cedeno, L. Marquez, and P. Nakov, *ClaimRank: Detecting Check-Worthy Claims in Arabic and English*, Comment: Check-worthiness; Fact-Checking; Veracity; Community-Question Answering; Neural Networks; Arabic; English, Apr. 2018. DOI: 10.48550/arXiv.1804.07587. arXiv: 1804.07587 [cs]. (visited on 02/27/2023).

[12] S. Zhi, Y. Sun, J. Liu, C. Zhang, and J. Han, "ClaimVerif: A Real-time Claim Verification System Using the Web and Fact Databases," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17, New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 2555–2558, ISBN: 978-1-4503-4918-5. DOI: 10.1145/3132847.3133182. (visited on 02/27/2023).

[13] N. Hassan, C. Li, and M. Tremayne, "Detecting Check-worthy Factual Claims in Presidential Debates," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, Melbourne Australia: ACM, Oct. 2015, pp. 1835–1838, ISBN: 978-1-4503-3794-6. DOI: 10.1145/2806416.2806652. (visited on 05/09/2023).

[14] R. Mitkov and G. Angelova, Eds., *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2021)*. Held Online: INCOMA Ltd., Sep. 2021. (visited on 05/15/2023).

[15] P. Atanasova, L. Marquez, A. Barron-Cedeno, *et al.*, "Overview of the CLEF-2018 CheckThat! Lab on Automatic Identification and Verification of Political Claims. Task 1: Check-Worthiness,"

[16] M. Hasanain, F. Haouari, R. Suwaileh, *et al.*, "Overview of CheckThat! 2020 Arabic: Automatic Identification and Verification of Claims in Social Media,"

[17] S. Shaar, M. Hasanain, B. Hamdan, *et al.*, "Overview of the CLEF-2021 CheckThat! Lab Task 1 on Check-Worthiness Estimation in Tweets and Political Debates,"

[18] S. Shaar, A. Nikolov, N. Babulkov, *et al.*, "Overview of CheckThat! 2020 English: Automatic Identification and Verification of Claims in Social Media,"

[19] E. Williams, P. Rodrigues, and S. Tran, "Accenture at CheckThat! 2021: Interesting claim identification and ranking with contextually sensitive lexical training data augmentation,"

[20] X. Zhou, B. Wu, and P. Fung, "Fight for 4230 at CheckThat! 2021: Domain-Specific Preprocessing and Pretrained Model for Ranking Claims by Check-Worthiness,"

[21] A. Savchev, "AI Rational at CheckThat! 2022: Using transformer models for tweet classification,"

[22] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, *Attention Is All You Need*, Comment: 15 pages, 5 figures, Dec. 2017. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762 [cs]. (visited on 03/02/2023).

[23] F. A. Acheampong, H. Nunoo-Mensah, and W. Chen, "Transformer models for text-based emotion detection: A review of BERT-based approaches," *Artificial Intelligence Review*, vol. 54, no. 8, pp. 5789–5829, Dec. 2021, ISSN: 1573-7462. DOI: 10.1007/s10462-021-09958-2. (visited on 05/15/2023).

[24] K. Suzuki,
Artificial Neural Networks - Methodological Advances and Biomedical Applications.
Apr. 2011, ISBN: 978-953-307-243-2.

[25] L. Ouyang, J. Wu, X. Jiang, *et al.*,
Training language models to follow instructions with human feedback, Mar. 2022.
arXiv: 2203.02155 [cs]. (visited on 05/16/2023).

[26] A. Dertat, *Applied Deep Learning - Part 1: Artificial Neural Networks*,
https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6, Oct. 2017. (visited on 06/08/2023).

[27] H. Liu and H. Motoda, Eds., *Feature Extraction, Construction and Selection*.
Boston, MA: Springer US, 1998, ISBN: 978-1-4613-7622-4 978-1-4615-5725-8.
DOI: 10.1007/978-1-4615-5725-8. (visited on 06/09/2023).

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova,
BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,
May 2019.
DOI: 10.48550/arXiv.1810.04805. arXiv: 1810.04805 [cs]. (visited on 05/24/2023).

[29] V. Sanh, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter,"

[30] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning,
ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators,
Comment: ICLR 2020, Mar. 2020.
DOI: 10.48550/arXiv.2003.10555. arXiv: 2003.10555 [cs]. (visited on 03/01/2023).

[31] T. B. Brown, B. Mann, N. Ryder, *et al.*, *Language Models are Few-Shot Learners*,
Comment: 40+32 pages, Jul. 2020. arXiv: 2005.14165 [cs]. (visited on 05/26/2023).

[32] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving Language Understanding by Generative Pre-Training,"

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov,
Proximal Policy Optimization Algorithms, Aug. 2017.
DOI: 10.48550/arXiv.1707.06347. arXiv: 1707.06347 [cs]. (visited on 05/23/2023).

[34] *Fine-tuning a Neural Network explained*,
https://deeplizard.com/learn/video/5T-iXNNiwIs. (visited on 06/09/2023).

[35] Z. Yun-tao, G. Ling, and W. Yong-cheng, "An improved TF-IDF approach for text classification,"
Journal of Zhejiang University-SCIENCE A, vol. 6, no. 1, pp. 49–55, Aug. 2005,
ISSN: 1862-1775. DOI: 10.1007/BF02842477. (visited on 05/21/2023).

[36] R. Gandhi, *Support Vector Machine — Introduction to Machine Learning Algorithms*,
https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47, Jul. 2018. (visited on 05/21/2023).

[37] L. de Bruijn, *Inter-Annotator Agreement (IAA)*,
https://towardsdatascience.com/inter-annotator-agreement-2f46c6d37bf3, Jul. 2020.
(visited on 05/20/2023).

[38] *Inter-rater Reliability Metrics: An Introduction to Krippendorff's Alpha*, https://www.surgehq.ai//blog/inter-rater-reliability-metrics-an-introduction-to-krippendorffs-alpha. (visited on 05/20/2023).

[39] A. J. Viera and J. M. Garrett, "Understanding Interobserver Agreement: The Kappa Statistic," *Family Medicine*,

[40] S. AI, *Inter-Annotator Agreement: An Introduction to Cohen's Kappa Statistic*, Dec. 2021. (visited on 05/20/2023).

[41] *Google Code Archive - Long-term storage for Google Code Project Hosting.* https://code.google.com/archive/p/word2vec/. (visited on 06/09/2023).

[42] T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient Estimation of Word Representations in Vector Space*, Sep. 2013. arXiv: 1301.3781 [cs]. (visited on 06/09/2023).

[43] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 1–27, Apr. 2011, ISSN: 2157-6904, 2157-6912. DOI: 10.1145/1961189.1961199. (visited on 05/25/2023).

[44] *Bert-base-uncased · Hugging Face*, https://huggingface.co/bert-base-uncased, Jun. 2023. (visited on 06/07/2023).

[45] *Textattack/bert-base-uncased-yelp-polarity · Hugging Face*, https://huggingface.co/textattack/bert-base-uncased-yelp-polarity. (visited on 05/24/2023).

[46] *Bhadresh-savani/electra-base-emotion · Hugging Face*, https://huggingface.co/bhadresh-savani/electra-base-emotion. (visited on 06/03/2023).

[47] *Distilbert-base-uncased · Hugging Face*, https://huggingface.co/distilbert-base-uncased, Apr. 2023. (visited on 05/24/2023).

[48] C. Pierse, *Introducing Transformers Interpret — Explainable AI for Transformers*, https://towardsdatascience.com/introducing-transformers-interpret-explainable-ai-for-transformers-890a403a9470, Feb. 2021. (visited on 05/09/2023).

[49] PyTorch, *Introduction to Captum — A model interpretability library for PyTorch*, Mar. 2020. (visited on 05/21/2023).

[50] M. Sundararajan, A. Taly, and Q. Yan, *Axiomatic Attribution for Deep Networks*, Jun. 2017. arXiv: 1703.01365 [cs]. (visited on 06/05/2023).

[51] N. Ankarstad, *What is Explainable AI (XAI)?* https://towardsdatascience.com/what-is-explainable-ai-xai-afc56938d513, Jan. 2022. (visited on 06/09/2023).

[52] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, *Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection*, May 2023. arXiv: 2302.12173 [cs]. (visited on 06/01/2023).

[53] *ClaimBuster*, https://idir.uta.edu/claimbuster/. (visited on 05/15/2023).

[54] K. Meng, D. Jimenez, F. Arslan, J. D. Devasier, D. Obembe, and C. Li, *Gradient-Based Adversarial Training on Transformer Networks for Detecting Check-Worthy Factual Claims*, Comment: 11 pages, 4 figures, 6 tables paperswithcode, May 2020. arXiv: 2002.07725 [cs]. (visited on 03/01/2023).

[55] J. Brownlee, *A Gentle Introduction to k-fold Cross-Validation*, May 2018. (visited on 06/09/2023).

[56] J. Wei, X. Wang, D. Schuurmans, *et al.*, *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*, Jan. 2023. DOI: 10.48550/arXiv.2201.11903. arXiv: 2201.11903 [cs]. (visited on 06/08/2023).

[57] S. Yao, J. Zhao, D. Yu, *et al.*, *ReAct: Synergizing Reasoning and Acting in Language Models*, Comment: v3 is the ICLR camera ready version with some typos fixed. Project site with code: https://react-lm.github.io, Mar. 2023. arXiv: 2210.03629 [cs]. (visited on 05/26/2023).

[58] X. Sun, X. Li, J. Li, *et al.*, *Text Classification via Large Language Models*, Comment: Pre-print Version, May 2023. DOI: 10.48550/arXiv.2305.08377. arXiv: 2305.08377 [cs]. (visited on 05/26/2023).

List of Figures

List of Tables

10 Appendix

10.1 Software and Tools

To do our experiments and train our models we used the programming Language Python. The most important libraries we used were Pandas, Sklearn and Transformers (from HuggingFace). We also used Git to manage our code.

10.2 Guide

All our Transformer models are uploaded to Hugging Face: `https://huggingface.co/ba-claim`

Clone GitHub repository: `https://github.com/BA-Claim-Extraction/code`

Note: The repository was originally setup to be used for all languages. However, as we started focusing primarly on English, we didn't make sure everything was working for all languages anymore.

10.2.1 Setup environment

- Setup project locally
- Install Python 3.8
- Install Python plugin in Intellij
- Create virtual Python env (files -> project structure -> add sdk -> add python skd -> create virtual env and select Python 3.8)
- switch to the directory check_that_task1
- Install requirements: *pip install -r requirements.txt*
- Upgrade pip: *python.exe -m pip install –upgrade pip*
- create .env file with OpenAI API key and Claimbuster API key as seen in Appendix 10.2.3 Repository usage.

10.2.2 Repository structure

All our code and data is located in the check_that_task1 folder.

The following are the most important directories:

- baselines: Contains a class for each model (meta-estimators are grouped in a single class) to make predictions on a certain dataset.
- core: Contains the core classes
- data: Contains all data, the original data, our predictions, submissions...
- preprocessing: Contains preprocessing tasks.
- training: Contains classes to train / fine-tune the transformer models we used. The results of this training are saved as checkpoints (not saved on GitHub due to their size). These checkpoints are later used to make predictions (using the baseline classes).
- utils: Contains helper functions and classes.

10.2.3 Repository usage

trainer.py

Python script that can be run to train the transformer models. The main method contains several variables that act as parameters to decide which model train, what data to train on, etc. Running this might take several hours.

runner.py

Python script to make predictions on any model. Similarly to the trainer, the runner has a main method that can be used to select the model, etc. All results are saved in tsv files. If the override flag is set, the results are only shown and not predicted again.

labeler.py

Python script, that we used to label some randomly selected data ourselves.

analyzer.ipynb

This jupyter notebook uses our result files (saved when running the runner.py) to analyze the results and provide an overview. It shows several different graphics (ROC-Curve and confusion matrix) and overviews.

analyze_labels.ipynb

This jupyter notebook provides an overview of our labelings. It shows how good our labels were compared to the gold labels and shows how much we agreed with each other.

playground.ipynb

This jupyter notebook provides an overview of our methods to make predictions. It shows all the used methods and provides a place to test them.

trainer_script.py

Python script that can be run to fine-tune the transformer models and to do hyperparameter search. This was the second approach for fine-tuning. Now it also included the hyperparameter search and was easier to use then the trainer.py Its a CLI script which will ask you what you would like to do and you can select the modes. Running this might take several hours as the process of finding hyperparameters and fine-tuning can be quite resource intensiv. Parts of this file can be found in Appendix 10.4.3 Fine-Tuning and Hyperparameter Optimization.

/core/chatgpt_classification_model.py

Python class to make prediction over the OpenAI API. Includes the prompt and all the settings to use the API. With the methods classify_text or classify_with_chat you can switch between the text-davinci-003 and gpt-3.5-turbo model. This class can be found in the Appendix 10.5.3 OpenAI API Classifier Class.

/core/text_classification_model.py

Python class is able to make prediction of the label and the corresponding probability. It has to be initialized with a transformer model from Huggingface. It will also automatically generate the word attributions. This class can be found in the Appendix 10.4.2 Classifier Class.

.env

You can generate a new api key here: `https://platform.openai.com/account/api-keys`

You can find your OpenAI Org here: `https://platform.openai.com/account/org-settings`

```
OPENAI_API_KEY=<api_key>
OPENAI_ORG=<org_key>
CLAIMBUSTER_API_KEY=<api_key>
```

Listing 2: Environment File to access the OpenAI API and Claimbuster API

10.3 Word attribution calculation

This script analyzer_word_attr.ipynb can be used to produce an overview of the word attributions per token as shown in Figures 7 and 8. The following code is used for the calculations shown in section 6.5.2, which is also included in the analyzer_word_attr.ipynb script.

```
df_attr = pd.DataFrame(columns=['Token', 'avg_neg', 'sum_neg', 'count_neg',
    'avg_pos', 'sum_pos', 'count_pos', 'avg_abs', 'sum_abs', 'avg', 'sum', '
    count_all'])

for key, value in token_scores_dict.items():
```

```
    pos_val = np.array([score for score in value if score > 0])
    neg_val = np.array([score for score in value if score < 0])

    new_row = {
            'Token': key,
            'avg_neg': np.mean(neg_val),
            'sum_neg': np.sum(neg_val),
            'count_neg': np.size(neg_val),

            'avg_pos': np.mean(pos_val),
            'sum_pos': np.sum(pos_val),
            'count_pos': np.size(pos_val),

            'avg_abs': np.mean(np.array(np.abs(value))),
            'sum_abs': np.sum(np.array(np.abs(value))),

            'avg': np.mean(np.array(value)),
            'sum': np.sum(np.array(value)),

            'count_all': np.size(np.array(value)),
            }
    df_attr = df_attr.append(new_row, ignore_index=True)

df_attr = df_attr.fillna(0)
```

10.4 Transformer models

10.4.1 Make predictions

To make predictions using our the transformer models you can use this code from the utils/example_prediction.py.

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from scipy.special import softmax

tokenizer = AutoTokenizer.from_pretrained("ba-claim/electra")
model = AutoModelForSequenceClassification.from_pretrained("ba-claim/electra
    ")

text = "I don't want to go back to malaise and misery index."

inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True,
     max_length=60)
outputs = model(**inputs)

probabilities = softmax(outputs.logits[0].detach().numpy(), axis=0)
label_index = outputs.logits.argmax().item()
label = model.config.id2label[label_index]
print(f"label: {label} --- pro: {probabilities} --- text: {text}")
```

Listing 3: Script to use our Hugging Face model to create predictions for a sentence

10.4.2 Classifier Class

```
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin
from transformers_interpret.explainers.text import
    SequenceClassificationExplainer
from scipy.special import softmax


class TextClassificationModel(BaseEstimator, ClassifierMixin):
    def __init__(self, model, tokenizer, name):
        self.model = model
        self.tokenizer = tokenizer
        self.cls_explainer = SequenceClassificationExplainer(model,
    tokenizer)
        self.name = name
        self.embedding_func = lambda x: x[0][:, 0, :].squeeze()

    def classify_text(self, text, class_name=""):
        inputs = self.tokenizer(text, return_tensors="pt")
        outputs = self.model(**inputs)
        probabilities = softmax(outputs.logits[0].detach().numpy(), axis=0)
        label_index = outputs.logits.argmax().item()
        label = self.model.config.id2label[label_index]
        word_attributions = self.cls_explainer(text, class_name=class_name)
        print(f"label: {label} --- text: {text}, pred = {self.cls_explainer.
    predicted_class_name}")
        return label, probabilities[label_index], word_attributions

    ###
    # https://captum.ai/
    # https://towardsdatascience.com/introducing-transformers-interpret-
    explainable-ai-for-transformers-890a403a9470
    # https://pypi.org/project/transformers-interpret/
    ###
    def get_explanation(self, text, class_name):
        word_attributions = self.cls_explainer(text, class_name=class_name)
        print(word_attributions)
        self.cls_explainer.visualize()

    def fit(self, X, y):
        return self

    def predict(self, X):
        return [self._tokenize_and_predict_label(string) for string in X]

    def predict_proba(self, X, y=None):
        return [self._tokenize_and_predict_probability(string) for string in
    X]

    def _tokenize_and_predict_label(self, text: str):
        inputs = self.tokenizer(text, return_tensors="pt")
        outputs = self.model(**inputs)
        label_index = outputs.logits.argmax().item()
        print(f"label: {label_index} --- text: {text}")
        return label_index
```

```

    def _tokenize_and_predict_probability(self, text: str):
        inputs = self.tokenizer(text, return_tensors="pt")
        outputs = self.model(**inputs)
        probabilities = softmax(outputs.logits[0].detach().numpy(), axis=0)
    [0:2]
        print(f"label: {np.around(probabilities,2)} --- text: {text}")
        return probabilities
```

Listing 4: Class for making predictions with a transformer model

10.4.3 Fine-Tuning and Hyperparameter Optimization

For the hyperparameter search and the following fine-tuning we used the trainer_script.py. We only highlight the important part of the script here.

```
    if selected_model == 'distilbert':
        hyper= {'learning_rate': 2.250831270949427e-05, '
    per_device_train_batch_size': 128, 'num_train_epochs': 5, 'weight_decay':
     0.005047876524611274}
        tokenizer = DistilBertTokenizer.from_pretrained("distilbert-base-
    uncased")
        model_init = model_init_distilbert

    elif selected_model == 'bert':
        hyper= {'learning_rate': 9.458979436773355e-05, '
    per_device_train_batch_size': 64, 'num_train_epochs': 4, 'weight_decay':
    0.0002736731745285213}
        tokenizer = AutoTokenizer.from_pretrained("textattack/bert-base-
    uncased-yelp-polarity")
        model_init = model_init_bert

    else:
        hyper = {'learning_rate': 2.6496325134477252e-05, '
    per_device_train_batch_size': 32, 'num_train_epochs': 5, 'weight_decay':
    0.009194238497066992}
        tokenizer = AutoTokenizer.from_pretrained("bhadresh-savani/electra-
    base-emotion")
        model_init = model_init_electra
```

Listing 5: Hyperparameter settings used for the training

```
    dataset_train_tokens = dataset_train.map(
        lambda text: tokenizer.encode_plus(text["Text"], add_special_tokens=
    True, truncation=True, padding=True))
    dataset_dev_tokens = dataset_dev.map(
        lambda text: tokenizer.encode_plus(text["Text"], add_special_tokens=
    True, truncation=True, padding=True))

    data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Listing 6: Here we enable truncation and padding for the tokenization

```
    if selected_action == 'hyperparameter':
```

```
        training_args = TrainingArguments(per_device_train_batch_size=4,
                                          num_train_epochs=1,
                                          **default_args)
    else:
        print('training -> starts')
        training_args = TrainingArguments(
             learning_rate= hyper['learning_rate'],
             per_device_train_batch_size= hyper['per_device_train_batch_size
    '],
             num_train_epochs= hyper['num_train_epochs'],
             weight_decay= hyper['weight_decay'],
              **default_args)


    trainer = Trainer(model=None, args=training_args,
                      model_init=model_init,
                      eval_dataset=dataset_dev_tokens,
                      data_collator=data_collator,
                      train_dataset=dataset_train_tokens,
                      compute_metrics=compute_metrics,tokenizer=tokenizer)

    if selected_action == 'hyperparameter':
       best_trial = trainer.hyperparameter_search(
         direction="maximize",
         backend="optuna",
         hp_space=optuna_hp_space,
         n_trials=20,
       )
       print(f"best_trial: {best_trial}")
    else:
       trainer.train()
```

Listing 7: Initialization of the Trainer API with the TrainingArguments and running either the hyperparameter search or the training

```
def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    predictions = np.argmax(predictions, axis=1)
    return f1.compute(predictions=predictions, references=labels)
```

Listing 8: Shows the method which is used for calculating the f-score for evaluation between searching and training cycles

```
def model_init_electra(trial):
    return ElectraForSequenceClassification.from_pretrained("bhadresh-savani
    /electra-base-emotion", num_labels=2, ignore_mismatched_sizes=True).to("
    cuda")

def model_init_bert(trial):
    id2label = {0: "No", 1: "Yes"}
    label2id = {"No": 0, "Yes": 1}
    return BertForSequenceClassification.from_pretrained("textattack/bert-
    base-uncased-yelp-polarity", num_labels=2,

                    id2label=id2label, label2id=label2id).to("cuda")
```

```

def model_init_distilbert(trial):
    return DistilBertForSequenceClassification.from_pretrained("distilbert-base-uncased", num_labels=2).to("cuda")
```

Listing 9: Initializing each Huggingface model with the right amount of labels

```
def optuna_hp_space(trial):
    return {
        "learning_rate": trial.suggest_float("learning_rate", 1e-6, 1e-4, log=True),
        "per_device_train_batch_size": trial.suggest_categorical("per_device_train_batch_size", [16, 32, 64]),
        "num_train_epochs": trial.suggest_int("num_train_epochs", 2, 5),
        "weight_decay": trial.suggest_float("weight_decay", 4e-5, 0.01)
    }
```

Listing 10: Method which sets the hyperparameter space in where the search is made

10.5 OpenAI

10.5.1 ChatGPT Prompt

The following prompt demonstrates our approach to generate predictions using ChatGPT.

```
I would like you to simulate an AI model which is classifing if a sentence is Checkworthy.
This is how checkworthy is defined:
Check-worthiness: Do you think that a professional fact-checker should verify the claim in the tweet? This question asks for a subjective judgment. Yet, its answer should be based on whether the claim is likely to be false, is of public interest, and/or appears to be harmful. Note that we stress the fact that a professional fact-checker should verify the claim, ruling out claims that are easy to fact-check by a layperson.

Prompt:
30313 And so I know that this campaign has caused some questioning and worries on the part of many leaders across the globe.
19099 Now, let's balance the budget and protect Medicare, Medicaid, education and the environment.
33964 I'd like to mention one thing.
16871 I must remind him the Democrats have controlled the Congress for the last twenty-two years and they wrote all the tax bills.
13150 And to take a chance uh - now be - and not make every effort that we could make to provide for some control over these weapons, I think would be a great mistake. 13386 Well, what he is saying there in effect, we're going to have inflation.
28916 I'm proud of the fact that violent crime is down in the State of Texas.
10612 You know, you may have seen your health care premiums go up.
22058 If we're $4 trillion down, we should have everything perfect, but we don't.
18005 And I made some tough decisions.

```

```
Output:
30313 No chatgpt
19099 No chatgpt
33964 No chatgpt
16871 Yes chatgpt
13150 No chatgpt
13386 No chatgpt
28916 Yes chatgpt
10612 No chatgpt
22058 Yes chatgpt
18005 No chatgpt

if you understand just anwser with: 'i understand'
```

Listing 11: Shows the example prompt used with the ChatGPT Web interface

10.5.2 API code

```
#This was the system prompt

system_content = "I would like you to simulate an AI model which is
    classifing if a sentence is Checkworthy.This is how checkworthy is
    defined:\n Check-worthiness: Do you think that a professional fact-
    checker should verify the claim in the tweet? This question asks for a
    subjective judgment. Yet, its answer should be based on whether the claim
     is likely to be false, is of public interest, and/or appears to be
    harmful. Note that we stress the fact that a professional fact-checker
    should verify the claim, ruling out claims that are easy to fact-check by
     a layperson.\n You response always in Json, with the label and the
    probability for the Yes label. The User is only giving sentences to
    classify\n"

# These are the two given examples:

first_user_content = "And if it's rape, how do you draw moral judgments on
    that?"
fist_assistent_content = "{\"label\":\"No\",\"probability\":0.267}"
second_user_content = "And many of the public schools are meeting the call."
second_assistent_content = "{\"label\":\"Yes\",\"probability\":0.854}"

# And we had 3 different test phrases to test the output.
test_no_content = "We've had enough of that, ladies and gentlemen."
test_yes_content = "Well, Mr. Ford is - is uh changing uh considerably his
    previous philosophy."
test_attack_content = "That answer was about as clear as Boston harbor."
# api call to the chat completion endpoint
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=[
            {"role": "system", "content": f"{system_content}"},
            {"role": "user", "content": f"{first_user_content}"},
            {"role": "assistant", "content": f"{fist_assistent_content}"},
            {"role": "user", "content": f"{second_user_content}"},
            {"role": "assistant", "content": f"{second_assistent_content}"},
```

```
            {"role": "user", "content": f"{test_attack_content}"}
    ]
)
# api call to the completion endpoint
response = openai.Completion.create(
    model="text-davinci-003",
    prompt=f"{self.system_content}\nUser:{self.first_user_content}\n"
           f"{self.fist_assistent_content}\n"
           f"User:{self.second_user_content}\n"
           f"{self.second_assistent_content}\n"
           f"User:{text}\n",
    temperature=0.6,
    max_tokens=150,
    top_p=1,
    frequency_penalty=1,
    presence_penalty=1
)
```

Listing 12: Shows how the OpenAI API is used to make predictons

10.5.3 OpenAI API Classifier Class

```
import os
import time

from dotenv import load_dotenv
import openai
import json

class ChatGPTClassificationModel:
    system_content = "I would like you to simulate an AI model which is classifing if a sentence is Checkworthy.This " \
                     "is how checkworthy is defined:\n Check-worthiness: Do you think that a professional " \
                     "fact-checker should verify the claim in the tweet? This question asks for a subjective " \
                     "judgment. Yet, its answer should be based on whether the claim is likely to be false, " \
                     "is of public interest, and/or appears to be harmful. Note that we stress the fact that a " \
                     "professional fact-checker should verify the claim, ruling out claims that are easy to " \
                     "fact-check by a layperson.\n You response always in Json, with the label and the probability " \
                     "for the Yes label.\n"
    first_user_content = "And if it's rape, how do you draw moral judgments on that?"
    fist_assistent_content = "{\"label\":\"No\",\"probability\":0.267}"
    second_user_content = "And many of the public schools are meeting the call."
    second_assistent_content = "{\"label\":\"Yes\",\"probability\":0.854}"

    def __init__(self):
        load_dotenv()
```

```
24
25          OPENAI_API_KEY = os.getenv('OPENAI_API_KEY')
26          OPENAI_ORG = os.getenv('OPENAI_ORG')
27
28          if not OPENAI_API_KEY:
29              print('Error: OPENAI_API_KEY is not set in .env file')
30          if not OPENAI_ORG:
31              print('Error: OPENAI_ORG is not set in .env file')
32
33          openai.organization = OPENAI_ORG
34          openai.api_key = OPENAI_API_KEY
35
36      def adjust_probability(self, prob, label):
37          if label == 'Yes':
38              prob = (float(prob) / 2) + 0.5
39          else:
40              prob = 0.5 - (1-float(prob) / 2)
41              prob = 1-prob
42          return prob
43
44      def classify_text(self, text):
45          label, probability = self.classify_with_completion(text)
46          return label, self.adjust_probability(probability, label)
47
48      def classify_with_chat(self, text):
49          response = openai.ChatCompletion.create(
50              model="gpt-3.5-turbo",
51              messages=[
52                  {"role": "system", "content": f"{self.system_content}"},
53                  {"role": "user", "content": f"{self.first_user_content}"},
54                  {"role": "assistant", "content": f"{self.
    fist_assistent_content}"},
55                  {"role": "user", "content": f"{self.second_user_content}"},
56                  {"role": "assistant", "content": f"{self.
    second_assistent_content}"},
57                  {"role": "user", "content": f"{text}"}
58              ]
59          )
60          content = response["choices"][0]["message"]["content"]
61          json_res = json.loads(content)
62          return json_res['label'], json_res['probability']
63
64      def classify_with_completion(self, text):
65          time.sleep(2)
66          response = openai.Completion.create(
67              model="text-davinci-003",
68              prompt=f"{self.system_content}\nUser:{self.first_user_content}\n
    "
69                     f"{self.fist_assistent_content}\n"
70                     f"User:{self.second_user_content}\n"
71                     f"{self.second_assistent_content}\n"
72                     f"User:{text}\n",
73              temperature=0.6,
74              max_tokens=150,
75              top_p=1,
```

```
            frequency_penalty=1,
            presence_penalty=1
        )
        print(response)
        content = response["choices"][0]["text"]
        json_res = json.loads(content)
        return json_res['label'], json_res['probability']
```

Listing 13: Class for making predictions with the OpenAI API