Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Master's thesis

Phishing detection using natural language processing

Bc. Radek Starosta

Supervisor: Ing. Jan Brabec

January 5, 2021

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Starosta Radek** Personal ID number: **423311**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software Engineering**

II. Master's thesis details

Master's thesis title in English:

Phishing detection using natural language processing

Master's thesis title in Czech:

Detekce phishingových e-mailů pomocí technik zpracování přirozeného jazyka

Guidelines:

This thesis addresses a problem of phishing detection from email content.
The concrete goals are:
1. Review the current phishing landscape and discuss its future trends. Particularly, focus on more sophisticated types such as targeted phishing and spear phishing.
2. Review the state-of-the-art approaches to phishing detection and natural language processing in general.
3. Design and implement a system for phishing detection. Describe various engineering constraints of such system and take them into account.
4. Evaluate the efficacy of the system on suitable datasets. Evaluate individual components of the system and when suitable perform ablation studies.
The student will work on this topic as part of a wider team of ML scientists and engineers.

Bibliography / sources:

[1] Jerry Shenk. SANS Spearphishing Survival Guide. 2015
[2] Wei Wang, Saghar Hosseini, Ahmed Hassan Awadallah, Paul Bennett, Chris Quirk. Context-Aware Intent Identification in Email Conversations. 2019
[3] Alberto Giaretta and Nicola Dragoni. Community Targeted Phishing: A Middle Ground Between Massive and Spear Phishing through Natural Language Generation. 2017
[4] Ashish Vaswani and Noam Shazeer and Niki Parmar and Jakob Uszkoreit and Llion Jones and Aidan N. Gomez and Lukasz Kaiser and Illia Polosukhin. Attention Is All You Need. 2017
[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2018

Name and workplace of master's thesis supervisor:

Ing. Jan Brabec, Department of Computer Science, FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **07.09.2020** Deadline for master's thesis submission: _____

Assignment valid until: **19.02.2022**

Ing. Jan Brabec Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt	Student's signature

Acknowledgements

I would first like to thank my supervisor, Jan Brabec, for his valuable guidance while writing this thesis and his excellent management skills throughout the course of this project's development.

I would also like to acknowledge my colleagues Marc Dupont, Tomáš Sixta, Miloš Lenoch, Filip Šrajer, and Martin Vejmelka, as well as the rest of the Cognitive Intelligence team. I thoroughly enjoy collaborating with you on this exciting project, and I could have never completed this thesis without your outstanding contributions.

I want to thank my family and friends for their support and care during these unusual times. Finally, I would like to thank my wonderful girlfriend, Nikola, who is the main reason for the timely completion of this thesis.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague on January 5, 2021 .....

Czech Technical University in Prague
Faculty of Electrical Engineering

Citation of this thesis

Starosta, Radek. *Phishing detection using natural language processing.* Master's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2021.

Abstrakt

V této diplomové práci představujeme architekturu systému pro detekci cílených phishingových útoků. Nejprve prozkoumáme současné trendy v oblasti phishingu, a identifikujeme techniky a vzorce chování, které se v těchto škodlivých zprávách vyskytují. Navrhneme detektor pro call-to-action, neboli potenciálně nebezpečné žádosti, které jsou jedním ze signálů pro identifikaci phishingových e-mailů. Na tomto detektoru ukážeme vývoj v oblasti zpracování přirozeného jazyka. Na této úloze následně evaluujeme několik klasifikačních algoritmů, od klasických metod strojového učení po nejnovější modely založené na neuronových sítích s architekturou Transformer. Dále v programovacím jazyce Python implementujeme obecný, rozšiřitelný systém pro klasifikaci e-mailových zpráv, zpřístupníme jeho funkcionalitu pomocí aplikačního rozhraní s architekturou REST, a navrhneme škálovatelnou infrastrukturu pro jeho nasazení v cloudové platformě Microsoft Azure. V neposlední řadě optimalizujeme výpočetní výkonnost detektorů založených na architektuře Transformer. Oddělíme detektor potenciálně škodlivých žádostí do samostatné komponenty, ve které zrychlíme inferenci modelu vybráním vhodné infrastruktury a architektury, a optimalizací výpočetního grafu pomocí akcelerátoru ONNX Runtime. Účinek navržených vylepšení ověříme na veřejném e-mailovém datasetu Enron, na kterém pozorujeme až pětinásobné zrychlení.

Klíčová slova phishing, zpracování přirozeného jazyka, klasifikace, neuronové sítě, nasazení modelů strojového učení

Abstract

In this thesis, we propose an architecture for an ensemble-based detection engine for targeted phishing attacks. We review the current phishing landscape and identify common phishing techniques and behaviors. Next, we develop a detector for call-to-action, a common phishing signal, and use it to showcase the impact of recent advances in natural language processing. We evaluate multiple classification algorithms, ranging from classic machine learning algorithms to state-of-the-art neural network Transformer models for language modeling, on the task of call-to-action detection. We implement an extensible e-mail classification pipeline using the Python programming language, expose its functionality through a REST API service, and design a scalable infrastructure setup for deploying this service in Microsoft Azure. Finally, we focus on optimizing the computational performance of Transformer-based detectors. We extract the call-to-action detector into a separate service, boost its performance by selecting suitable infrastructure and model architecture, and optimizing the computation graph with the ONNX Runtime accelerator. We validate the speedup from the proposed optimizations on the Enron e-mail dataset, where we observe a $5\times$ increase in service throughput.

Keywords phishing, natural language processing, classification, neural networks, deploying machine learning models

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the migration of critical infrastructure into the internet, the need for effective systems to counter cyberattacks is larger than ever. E-mail is commonly used to deliver malicious payloads, and according to IBM X-Force Threat Intelligence Index 2020, phishing is currently the most frequent initial attack vector [9]. However, many successful attacks do not require compromising the security of company infrastructure; a large portion of cybercrime cases consist of surprisingly simple frauds based on social engineering. Phishing poses risks for ordinary users, but also to government agencies and businesses, where these attacks are prevalent in the form of "business e-mail compromise" and were responsible for over $1.7 billion in losses reported in 2019 [10].

In this thesis, we present a system for detecting e-mail phishing attacks, with an emphasis on identifying sophisticated targeted phishing using natural language processing techniques.

The thesis is structured as follows. In Chapter 2, we describe the concept of phishing, categorize phishing attacks, and showcase common phishing techniques. We give a high-level introduction of a system for phishing detection and enumerate its system requirements.

In Chapter 3, we present an overview of relevant natural language processing techniques and their relation to the proposed ensemble-based phishing classification engine. We enumerate several classification and language modeling methods, emphasizing state-of-the-art neural network models, and describe multiple approaches to representing text for machine learning algorithms. Then, we introduce a call-to-action detector based on natural language processing, show its development progress, and demonstrate the impact of the selected classifiers and text representations on its predictive performance.

In Chapter 4, we give a detailed description of the classification engine workflow and show the pipeline for e-mail preprocessing, detection, and detection aggregation.

In Chapter 5, we show how we expose the phishing detection engine through a REST API service and highlight important implementation details.

In Chapter 6, we overview the cloud infrastructure used to deploy the API service and describe tools used to automate the deployment process.

In Chapter 7, we focus on optimizing the inference of neural network models. We demonstrate how inference time can be reduced significantly using a combination of suitable infrastructure, alternative model architectures, and model graph optimizations. Finally, we benchmark the impact of the proposed optimizations on the call-to-action detector service using a public e-mail dataset.

We conclude the thesis in Chapter 8 and present ideas for future work.

Chapter 2

Phishing

Phishing is a technique for acquiring sensitive data, such as bank account numbers, through a fraudulent solicitation in email or on a web site, in which the perpetrator masquerades as a legitimate business or reputable person [11]. Merriam-Webster give a broader definition of phishing as a fraudulent operation by which an e-mail user is duped into revealing personal or confidential information which can be used for illicit purposes [12].

Phishing attacks are popular because they are easy and cheap – sending e-mails is free and lists of e-mail adresses can be bought or collected from scraping the web. They also do not require direct contact with the victims and can be thoroughly premeditated.

2.1 Categorization

In practice, we identify two main categories of phishing e-mails – large-scale campaigns which target massive amounts of users with generic e-mails, and spearphishing attacks which are aimed at a specific person or company. The boundary between these two categories is fuzzy, and in general, there is a tradeoff between the required effort, quantity of contacted victims, and the effectivity of the attack.

2.1.1 Large-scale Phishing Campaigns

Unlike the common spam e-mails which contain unwanted advertisements or spread false information, large-scale phishing campaigns aim at actively compromising user accounts or stealing sensitive data such as credit card details. In the phishing e-mails, the attacker is posing as a trustworthy authority and tries to trick the users with a fraudulent message. According to Cofense 2019 Annual Phishing Report, 74% of phishing e-mail target credentials [13].

They often target online banking accounts and similar services (e.g., Paypal) which provide direct access to money once compromised. Phishing e-mails also tend to create a sense of urgency, e.g., threatening to terminate the users account if

he does not act quickly. The attackers will often use machine translation to reach a broader audience of potential victims, therefore grammatical errors and overall poor use of the language can be a giveaway of such e-mails. An example of a typical phishing e-mail aimed at bank customers can be seen in Figure 2.1.

Dear member,

we detected unusual activity on your Bank of America debit card on 09/22/2014. For your protection, please verify this activity so you can continue making debit card transactions without interruption.

Please sign in to your account at https://www.bankofamerica.com to review and verify your account activity, After verifying your debit card transactions we will take the necessary steps to protect your account from fraud. If you do not contact us, cretain limitations may be placed on your debit card.

© 2014 Bank of America Corporation. All rights reserved.

Figure 2.1: Example of a large-scale phishing campaign aimed at Bank of America customers. Note the unnatural emphasis and the presence of slight grammatical errors. Link is masqueraded and leads to a malicious destination (Section 2.2.3.1).

While these types of e-mails were very prevalent in the past, we now encounter fewer of them, as they become less effective with modern e-mail providers improving their phishing detection and filtering, and the users becoming more cautious and educated about phishing. The general bar of the security of computer systems nowadays is also higher, and with the presence of safeguards such as two factor authorization, it can be difficult to compromise the account even with the knowledge of user credentials.

2.1.2 Spearphishing

In recent years, we can see a shift towards more sophisticated techniques, as e-mail providers improve the detection of general phishing e-mail. Personalized, targeted phishing scams known as "spearphishing" are particularly dangerous. Kaspersky define spearphishing as an email or electronic communications scam targeted towards a specific individual, organization or business [14].

In spearphising attempts, the attacker possesses some knowledge about his victim, and he abuses it to trick him or her into doing some harmful action, such as opening a malicious attachment, authorizing a payment, or leaking internal data. Such attacks are difficult to detect automatically, as they don't repeatedly appear in the network data, and also extremely effective, because they target a single person with specific information.

For instance, if an employee receives an offer from a Nigerian prince offering to transfer his entire net worth to a chosen bank account, it will probably get ignored or filtered by the e-mail provider. However, if the same employee receives an e-mail from someone posing as his co-worker, containing an Excel file with data he needs to finish his overdue assignment, it is quite likely he will open it without a thorough examination.

2.1.3 Business E-mail Compromise

The spearphishing attacks are extremely dangerous in the environment of corporate networks. A significant portion of communication inside a typical company still happens over e-mail, and in huge companies with multiple subsidiaries, it can sometimes be the only option to communicate. If the attackers have knowledge of the company' organizational structure, they can easily come up with a scenario to compromise its security (i.e., by posing as a co-worker, an executive, or a business partner). This is called a "Business E-mail Compromise" (BEC), and according to Cofense 2019 Annual Phishing Report, over 8% of all observed phishing e-mails were BEC scam attempts [13]. Example of a BEC attempt can be seen in Figure 2.2.

```
Hi Joe,

Are you in the office? I need you to setup a $8,988.0 payment for
the new contractor.

Sort code: 30-61-22
Acc. number: 10436773
Beneficiary: Heidi Roberts

I will send the documents as soon as I'll sort out my stuff.

Regards
John Patterson
Sent from my iPhone.
```

Figure 2.2: Example of a business e-mail compromise attempt to force an employee to perform a money transfer. Adapted from [1].

BEC attempts become especially difficult to counter when a real business e-mail account is compromised. In this case, the defense depends on the employees ability to recognize the malicious intentions, which can be nearly impossible if the request is within the scope of his standard responsibilities. Attackers will often pose as executives who the employees are also less likely to oppose or question.

A successful BEC can result in significant damages to the company, e.g, through forcing a fraudulent money transfer, exfiltrating sensitive data out of the company, or executing a more sophisticated attack after gaining access to the company in-

frastructure. According to FBI 2019 Internet Crime Report, over 23 thousand cases of successful BEC attacks have been reported, with total losses of over $1.7 billion [10].

2.2 Phishing Techniques

To detect phishing e-mails, we first need to examine the techniques commonly used to disguise the malicious intentions of such e-mails. In this section, we explore some of the more common signals we encounter today, and also discuss possible future trends utilizing artifical intelligence. Note that while most of the behaviors we enumerate are not conclusive evidence on their own, their presence is highly suspicious, and detecting a combination of them can suffice to produce a final verdict with reasonable confidence.

2.2.1 Social Engineering

In this section, we analyze the textual content of phishing e-mails and show common persuasion techniques employed by attackers. One common factor of phishing e-mails is the use of social engineering. To gain something from the attack, the attacker first has to make his victim believe that the e-mail is coming from a trusted party. This impersionation alone is difficult to detect automatically – there are no written rules for what is considered suspicious in an e-mail conversation, and judging the situations correctly depends on the context. For broad bulk phishing campaigns, we could often notice poor use of the language stemming from the lack of proper language skill and the use of machine translation, but encountering such telling signs is rare in the more elaborate spearphishing attacks.

2.2.1.1 Call-to-Action

A successful impersonation is not in itself dangerous until the attacker succeeds in making the victim perform a malicious action. Therefore the e-mails we are most interested in detecting and stopping are the ones that are directly requiring the user to do something — i.e., provide his credentials, click a link, or open an attached file (Figure 2.3). We will be naming this "call to action," and a call to action detector will be one of the main components of our proposed phishing classifier.

2.2.1.2 Urgency

Phishing e-mails often evoke a sense of urgency to make the victim act quickly and without caution. Threats of impending account cancellation, missed due dates, or limited time offers are all a common occurrence. Urgency is also present in the example e-mail in Figure 2.3, where the CEO impersonator demands a response "as soon as possible".

Hello David,

hope you had a great weekend. **I'm still missing the financial report, please send it to me ASAP.**

Mr. CEO

Figure 2.3: Example of a call-to-action in the broader context of a sample e-mail. Our detector should place this sample in the "data input" category.

2.2.1.3 Confuseable Unicode Characters

With increasingly sophisticated phishing and spam detectors employing e-mail content analysis, attackers have come up with a technique to evade detection from these automated systems through the use of confuseable unicode characters. The message containing these characters will be fully understood by the victim reading the e-mail in the context of the surrounding words and characters, but the inclusion of mangled words can potentially disrupt the automated analysis of the message.

Examples of confuseable unicode characters can be seen in [15], other common cases of visual spoofing are enumerated in [16].

2.2.2 Sender Impersonation

The e-mail sender is possibly the most imporant indicator of e-mail legitimacy. Therefore, it is important for phishing e-mails to make users believe they are coming from a familiar source even if they do not have control over such e-mail account. In this section, we cover how this can be achieved using e-mail addresses resembling familiar senders or by spoofing the e-mails.

2.2.2.1 Using Misleading E-mail Addresses

A simple and common approach to sender impersonation is to register a free public domain e-mail account resembling the credentials of the impersonated company or person, hoping that the targeted user does not notice the use of an unofficial e-mail address. In the case of spearphishing, attackers will often claim the use of a private e-mail account of a coworker or business associate, and when paired with a believable and convincing message, this often suffices to fool an unsuspecting victim. Clever use of a suitable free e-mail host can increase the e-mails' credibility, such as in the case of a recent phishing scam impersonating the Czech postal service, where the e-mail came from an address registered with the `@post.cz` domain (Figure 2.4).

From:	Czech Post <intro-tracking-post@post.cz>
Subject:	Confirm your parcel payment

Figure 2.4: Example of a phishing e-mail using a misleading e-mail address inspired by a real recent attack [2].

2.2.2.2 Header Spoofing

Users are now often knowledgeable enough to check the origin of suspicious e-mails, so the e-mail sender must appear to be legitimate. Directly spoofing the sender address in the e-mail header is possible because of the lack of authentication in the core e-mail protocol. Therefore, anyone is capable of constructing and sending an e-mail similar to the example in Figure 2.5. Such e-mails are nowadays usually detected using a combination of DMARC, DKIM and SPF protocols [17, 18, 19] and filtered or displayed with a warning by most e-mail providers. However, sophisticated attacks can circumvent these authentication protocols, as shown by Chen et. al [20].

From:	doc. RNDr. Vojtěch Petráček, CSc. <Vojtech.Petracek@cvut.cz>
To:	prof. Mgr. Petr Páta, Ph.D. <dean@fel.cvut.cz>
Subject:	Scholarship Recommendation for Radek Starosta

Figure 2.5: Example of a spoofed e-mail header

2.2.3 URL Manipulation

Phishing e-mails are used to establish contact with the victim, but the critical part of the attack often happens on a malicious website to which the attackers try to lure the users using external URL links in the e-mail message. These sites can then contain forms to steal confidential data or credentials. In other cases, they will host malicious executables to avoid automatic scanning of e-mail attachments. In this section, we walk through several techniques used to hide true destination of these malicious links.

2.2.3.1 Link Masquerade

Link masquerade is a technique applicable to HTML e-mails, which abuses the display text of an <a> anchor tag. Anchor tags allow setting a custom display text for the link, and the attacker can set the display text to appear as an URL leading to a legitimate hostname, but the href attribute will instead lead the user to his malicious site. To defend against this attack, the targeted user would have to check the final destination in the web browser, which many will not do if the web page or the e-mail looks convincing.

```
<a href="http://attacker.com">http://google.com</a>
```

Figure 2.6: Example of a link masquerade – uses google.com as display name, but points to attacker.com

2.2.3.2 Open Redirect

Open redirects can be utilized similarly to link masquerade and again attempt to disguise suspicious-looking URLs from the user [21]. In this case, with the help of a redirection from a well-known service — the link will appear to lead to a trusted host (i.e., Google), but the real destination is passed in the request parameters. When the user visits the link which appears to point to a familiar website, he gets redirected to a different, malicious site by the legitimate web service. Using an open redirect can also help avoid automatic detections that would typically block links to such malicious servers.

```
https://google.com/url?sa=t&url=http://attacker.com/&usg=AOvV…
```

Figure 2.7: Example of open redirect on Google.com [3].

2.2.3.3 Typosquatting

Typosquatting is the use of a registered domain name which very closely resembles the domain of the impersonated brand. While the previously described techniques rely on the user failing to notice a malicious link destination or redirect in the e-mail, typosquatting makes this harder to detect even when checking the final domain, as they will be visually similar.

Different variants of this method can be seen in the wild – some use confusable characters or minor spelling differences (i.e., `bankofamereca.com`, `goggle.com`), in other cases the sites can only use a different top-level domain. Other complementary techniques like "combosquatting" [22] and "dash-phishing" make use of second-level domains and extra periods or hyphens (Figure 2.8).

```
https://google-com.attacker.com/
```

Figure 2.8: Example of dash-phishing on Google.com

Szurdi et al. [23] summarize typosquatting techniques and their monetization strategies, and develop a framework for typosquatting categorization based on clustering data collected from publicly available domain records and web crawlers.

2.2.4 Utilizing Artifical Intelligence

In the recent years, we have seen a major advancement in the field of natural language processing, and artificial intelligence in general. Although we have mainly seen the positive effect on these advancements in areas such as machine translation, they can also be utilized by malicious actors and have significant security consequences. For instance, a successful, fully-automated spear phishing system utilizing AI has been demonstrated by Seymour et al. [24].

With the availability of powerful neural network models like GPT-2 [25], and recently its successor GPT-3 [26], the potential for malicious use has only increased. These models are capable of producing generated text that is often indistinguishable from human-generated data, and even the authors of GPT models warn about the possibilities of malicous use for generating fake news, spam, but also e-mail phishing [27].

Giaretta et al. [28] describe "community targetted phishing", a possible future trend in phishing. This approach can be thought of as a middle-ground between large, untargetted campaigns and closely targetted spearphishing e-mails, and aims at creating phishing e-mails for a broader audience which remaining specific enough to stand out from basic phishing attempts. Authors first describe a template-driven approach, and show how it can be evolved using natural language generation techniques – both with relatively little effort compared to spearphishing. An example of a community targetted phishing e-mail which can target scholars and be easily adapted using publicly available data (i.e., from Google Scholar) is shown in Figure 2.9.

Dear **[Target Name]**,

I am Prof. **[Reputable Name]** from **[Reputable University]**, we are currently expanding our lab and we are evaluating some possible candidates. I have personally checked your Google Scholar account and noticed that you h-index is high with respect of the average in your field, and considerably grew over the last 5-years span. Therefore, we would like to propose you a position. You can enrol at the following link: **[Phishing Link]**

Figure 2.9: Example of a community targetted phishing e-mail template.

2.3 Project Introduction

In this thesis, we cover the development of a phishing classification engine which has been the main focus of our team for the past year. The engine we are developing is meant to integrate with and complement other products which already provide a layer of security against generic phishing. Therefore, our main goal is to detect

more sophisticated attacks such as spearphishing and BEC attempts using advanced machine learning techniques applied on e-mail content.

2.3.1 System Requirements

Azure ecosystem Our engine is provides an additional layer of security to Office 365 [29] mailboxes. This ties us to the Microsoft ecosystem and requires us to deploy the engine inside of Microsoft Azure [30].

Scalability There is an emphasis on scalability in our design process. As the number of managed mailboxes increases, we need to be able to scale our infrastructure appropriately and design the system in a way where we do not create processing bottlenecks.

Efficacy Our system has to provide solid and useful verdicts about previously undetected e-mails, and minimize the amount of false positives, as blocking legitimate e-mails is unacceptable in corporate communication.

Cost of Operation Finally, we need to take into account the cost of operation of our engine, and design an efficient system with reasonable costs. This requires selecting suitable Azure instances and utilizing them effectively.

Chapter 3

Phishing Detection Using Natural Language Processing

In this chapter, we introduce natural language processing and show how it relates to our task of phishing detection. We describe our idea of the phishing detection system based on ensembling and introduce a detector for call-to-action, on which we demonstrate multiple techniques for text classification. Starting from fundamental NLP approaches, we incrementally show our detector's evolution while introducing newer techniques and advancements in the field. Finally, we demonstrate their impact by comparing the predictive performance of the detector iterations.

3.1 Natural Language Processing

Natural Language Processing (NLP) is the automatic manipulation of natural language, such as speech, or in our case, written e-mail communication. Voice and text are our primary means of communication, and as such, it is crucial to understand and analyze this data. Understanding natural language is challenging, as languages can be difficult to bind by rules and often have nuances that even native speakers can misinterpret or fail to understand. It can be tackled from several angles, i.e., using the linguistic approach of analyzing and describing the syntax and semantics, but in this thesis, we will focus on statistical methods for NLP, more specifically on the task of classifying text data.

3.2 Phishing Detection

The goal of our project is to be able to identify potentially dangerous phishing e-mails. In its simplest form, this is a binary text classification task where we try to distinguish between phishing and benign e-mails.

We can observe a large number of diverse phishing techniques, and therefore independent signals of suspicious behavior in e-mail data. Thus, the approach we

have chosen to tackle the problem of detecting phishing e-mails is to construct an ensemble system with multiple, specialized detectors and then aggregating these signals to produce a final verdict. These individual detectors can be simple or complex and have different weights, but eventually, the combination of their detections will give us the binary classification result.

Although multiple types of signals can help us flag suspicious e-mail (i.e., by using information from e-mail headers), one of the more exciting research problems we attempt to solve is to discover the malicious intent in the body of the e-mail, which contains natural language. This chapter will focus on a detector for the behavior we have named a "call-to-action" and showcase multiple NLP techniques used to bring it to life.

3.3 Classification Algorithms

This section gives an overview of machine learning algorithms used to build learned detectors of our phishing classification engine. We briefly introduce naive Bayes and logistic regression classifiers as the representatives of classic machine learning algorithms and then cover neural network-based approaches, which currently dominate the field of machine learning, in greater detail.

3.3.1 Classification

Classification is the task of assigning one or more categories, or classes, to data based on a set of input variables.

We formalize classification in the context of supervised learning, where a training set of labeled data is available. Classifier is a function f_θ learned from a set of training samples $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\} \subset \mathbb{R}^d \times Y$, where $\mathbf{x}_i = (x_1, \ldots, x_d)$ are numeric feature vectors and y_i are classification labels from a discrete set of classes $Y = \{c_1, \ldots, c_k\}$. The classifier function then maps a given input vector $\mathbf{x}$ to the prediction result $\hat{y} = f_\theta(\mathbf{x})$ using the learned parameters θ.

Multi-label classification is a generalization of classification, where any number of classes can be assigned to the same instance.

In this thesis, we focus on classifying text extracted from e-mail messages. This requires us to perform feature extraction and transform the input data into a numeric feature vector representation, which we cover in Section 3.7.

3.3.2 Naive Bayes Classification

Naive Bayes classifier is a probabilistic classifier based on Bayes' theorem application, which naively assumes conditional independence between input features given a class label. It is extremely fast compared to more sophisticated classifiers, essentially requiring only a single pass through the training data to estimate its parameters, making it a very cheap starting model. Despite its simplicity, it has been

shown to work well in practice in the area of text classification, even with relatively small amounts of training data [31].

It can be formulated using the Equation 3.1, where $P(y)$ is the probability of class y estimated from its frequency in the training set, and $P(x_i|y)$ is the probability of feature x_i appearing in a sample belonging to class y.

$$\hat{y} = \arg\max_{y} \left(P(y) \prod_{i=1}^{n} P(x_i|y) \right) \tag{3.1}$$

There exist multiple variants of this algorithm, such as Gaussian or multinomial naive Bayes, which differ by the assumptions they make regarding the distribution of $P(x_i|y)$.

3.3.3 Logistic Regression

Logistic regression is a linear classification model that assumes a linear relationship between the predicted class and input features but adds a non-linear transform on the output using a logistic function. This logistic function (Equation 3.2) can squeeze any real-valued input into a value between 0 and 1. We can then obtain the binary classification by setting a decision boundary threshold (Equation 3.3).

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

$$\hat{y} = \begin{cases} 1 & \text{if } f(x) \geq 0.5 \\ 0 & \text{if } f(x) < 0.5 \end{cases} \tag{3.3}$$

3.4 Neural Networks

The recent rapid progress in neural network research has been driving most of modern AI successes. Neural network models have broad applications in computer vision, speech and pattern recognition, language modeling, and related tasks such as machine translation, fact-checking, or text classification. For most, if not for all of these tasks, they can outperform conventional machine learning algorithms and achieve state-of-the-art results. For instance, a summary of language modeling task results for recently published models is available at [32].

Artificial neural networks are inspired by real neural networks found in the human brain. They are composed of many connected units called neurons, which receive and transmit signals in a large network structure. In an artificial neural network, this signal is a real number. The neuron's output is computed by applying an activation function on a weighted sum of its inputs.

The original idea of neural networks is very old, with original papers exploring systems resembling neural networks dating back to the 1940s. However, the wide adoption of this approach to statistical modeling came only in recent decades, and it

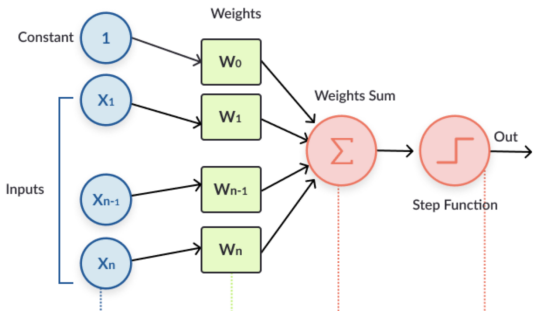

Figure 3.1: Demonstration of an artificial neuron in the perceptron classifier [4].

was largely driven by improvements in distributed and GPU computing, which provided enough computing power to support the training of large models with a deep, layered structure. This concept has been named "deep learning," and a thorough summary of its applications can be found in the Deep Learning book by Goodfellow et al. [33]. One of the main reasons behind the success of these models is their ability to scale and learn from huge datasets that have become available in recent years.

3.4.1 Perceptron

As we briefly discussed in the previous section, the building blocks of neural networks are artificial neurons (Figure 3.1). These neurons are connected, and each neuron has one or many input and output connections. The neuron has internal, learned weights, which it uses to compute a weighted sum from its input signals. Finally, to obtain the neuron's output signal, we use an activation function to transform this sum, which introduces non-linearity into these systems and makes them able to model highly complex relationships in the data accurately.

Before we dive into more complex architectures, we will first introduce the simplest neural network classifier — the perceptron. Perceptron is a binary classifier with a single artificial neuron that combines the feature vector with learned weights. It then applies the Heaviside step function (returns 0 for negative and 1 for positive arguments) as its activation function, which directly decides the output class (Equation 3.4).

$$\hat{y} = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \end{cases} \tag{3.4}$$

3.4.2 Multi-layer Perceptron

The real power of neural networks comes with the combination of many neurons in large, connected structures. A multi-layer perceptron model, or a feed-forward

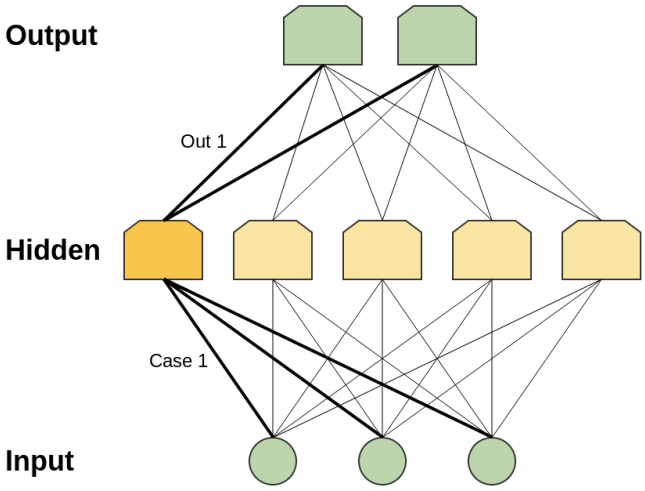

Figure 3.2: An example of a simple feed-forward neural network [5].

neural network, combines multiple perceptrons organized into a hierarchical, multi-layer structure.

Multi-layer perceptrons have an input layer (sized to accept signal from all input features), an output layer (sized according to the expected output, i.e., probabilities for individual classes in classification tasks). In between, there are one or many arbitrarily sized hidden layers. In the basic MLP neural network, layers are fully connected, which means that each neuron sends its output to all neurons in the next layer.

In Figure 3.2, we can see a simple multi-layer perceptron with three layers. We will encounter huge neural networks in practice, often with hundreds of layers and hundreds of millions of parameters. Because learning a deep network with a fully connected architecture is computationally expensive, and such networks are more prone to overfitting, the deeper and more complex models use different types of layers, which help reduce the number of model parameters. Examples of these are convolutional and pooling layers or techniques like dropout [34], which omits some randomly selected neurons during each training epoch.

3.4.3 Training Neural Network Models

Neural network models are trained using algorithms based on stochastic gradient descent (SGD). In this iterative process, the training data is passed through the network, the result of this computation is compared to the expected output, and an error is calculated using a selected loss function. We then use a "backpropagation" [35] technique to update the neural network weights — the error is propagated back through the network, gradually computing the gradient of the loss function used to update the weights according to how they contributed to the computed error.

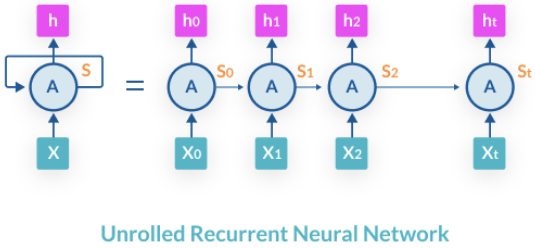

Figure 3.3: An example of a recurrent neural network architecture.

Because the datasets used for large-scale neural network models are often massive, it is not feasible to compute the gradient from all available data together. On the other hand, updating the network-based on only a single data point in every step or epoch, while much faster, leads to chaotic and unpredictable changes in the network. Therefore batch learning, essentially a middle ground between these two methods, is the standard approach — small, randomly selected batches are used to compute the gradient, bringing more stability to the learning while still being reasonably fast.

In practice, different variants of this algorithm are used to make the learning efficient, one notable example being the ADAM optimizer [36]. The training process can also be further adjusted using hyperparameters, such as learning rate, momentum, or decay.

3.5 Recurrent Neural Networks

Neural networks provide excellent performance in a huge number of diverse machine learning tasks, including language modeling and text classification, which will be our main area of focus in this chapter. Standard feed-forward neural networks are not suited for processing sequential data such as written text — they receive a set of inputs and analyze them together, but a proper understanding of a piece of text requires knowing the context of previous words or sentences. Their fixed-sized inputs are also limiting as written text, and sequences in general, are by nature of variable size. Unlike, i.e., images, they cannot be easily cropped or resized. Recurrent neural networks (RNNs) solve this problem by allowing input sequences of variable length and reusing the output or hidden state from one or many previous steps to compute the prediction for the current step. An example of inference in such a network can be seen in Figure 3.3 — processing an input sequence X using the state vectors S yields the output sequence h.

The network parameters are learned using an algorithm called "Back propagation through time" [37], which is similar to standard backpropagation but gradually computes the error for each time step in the sequence. The errors are then accu-

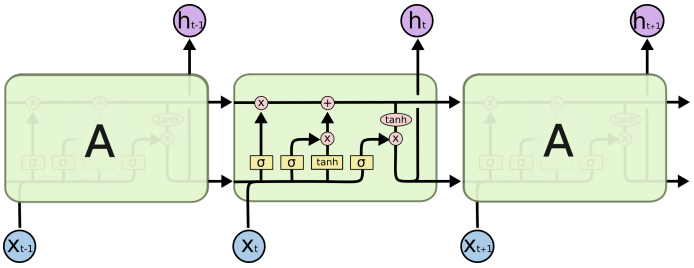

Figure 3.4: Overview of an LSTM cell [6].

mulated and used to update network weights. This process can be computationally expensive for longer sequences with many time steps.

RNN architecture can be naturally applied to language modeling, which is essentially the task of predicting the probability of words occurring in a sentence, given a previous piece of text. RNNs, with the addition of gating mechanisms like long short-term memory or gated recurrent units, have been essential for the rapid progress in the field of language modeling and other closely related tasks, such as machine translation or text synthesis. However, we have recently seen a significant shift towards transformer-based models built on the ideas presented in this section, but no longer directly utilize recurrence.

RNNs have also been shown to perform well even on data that is not sequential — Ba et al. [38] successfully use an RNN to classify image data using sequences of pixels.

3.5.1 Long short-term memory

Although the information about all previous input can theoretically be encoded in the state vectors of basic RNNs, in practice, they struggle with capturing long-term dependencies. Long short-term memory (LSTM) [39] is an architecture, which significantly increases the ability of RNNs to retain information in longer input sequences.

LSTMs use three gates to control the state vectors passed through the cells, which have the ability to both add and remove information (Figure 3.4). The input gate controls the significance of the new input value that flows into the cell and whether it should be retained in the state. The forget gate determines which old information should be discarded. Finally, the output gate selects the relevant parts of the state for the output of the current cell.

3.6 Transformers

The Transformer is a neural network architecture first proposed in the “Attention is all you need” article published in 2017 [7]. The original paper focuses on its application for machine translation, but the Transformer model has been highly

influential in the entire field of NLP, as it was the first successful model to stray away from sequence-based RNNs and convolutions, instead of relying on a mechanism called "attention." Attention, used in the form of scaled dot-product self-attention in the proposing paper, allows the model to utilize information from other words in the processed sequence when encoding the current word. In the sentence "The student did not work on his thesis today because he was too tired." the word "he" refers to the student — the attention mechanism can help the model focus on these relationships required to fully comprehend the sentence.

Transformer is based on the Encoder-Decoder architecture successfully used for machine translation with both RNNs [40] and LSTMs [41]. As proposed in these papers, the encoder part of the model reads the sequence, one timestep at a time, and encodes it into a fixed-sized context vector. The decoder then gradually extracts the output sequence from this context vector. However, the compression of the necessary information into a fixed-sized vector can be problematic for longer input sequences because the model eventually "forgets" the information from the early sequence parts.

The attention mechanism introduced in [42] helps to memorize information from these longer sequences by developing a context vector filtered specifically for each output time step. The Transformer model takes this approach even further, as the entire model is based on attention without using the recurrent architecture (Figure 3.5). It views the encoded representation of the input as a set of key-value pairs (K, V), obtained from the encoder's hidden states. In the decoder, the previous output is compressed into a query Q, and the next output is produced by mapping this query and the set of keys and values.

These vectors are packed into matrices and combined using a scaled dot-product self-attention mechanism (Equation 3.5), where d_k denotes the dimension of the key vectors.

$$\text{Attention}\ (Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3.5}$$

Abandoning the standard sequential approach of processing text left-to-right has been the most significant contribution of this model. Learning models with Transformer architecture are easier to parallelize. It, therefore, allows utilizing the computation power of current GPUs fully. Since the publishing of this article, a vast amount of models belonging to the Transformer family have been developed [43], and Transformers have become the go-to models for more complex NLP tasks.

A thorough visual demonstration of the Transformer architecture and the self-attention mechanism is available at [44]. For an overview of other attention models, see [45]. Recent progress on enhanced Transformer models is summarized at [46].

3.6.1 BERT

Bidirectional Encoder Representations from Transformers (BERT) is a language representation model developed by Google researchers in 2018 [47].

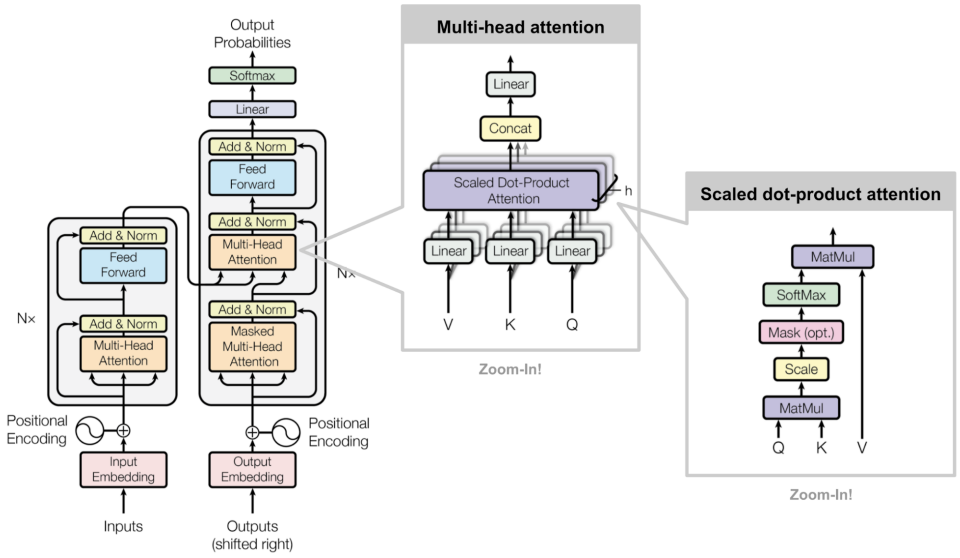

Figure 3.5: The Transformer architecture [7].

BERT directly applies the Transformer architecture to language modeling, and to do so, it utilizes two training strategies. Masked LM replaces some words in each sentence with a `[MASK]` token, and the model then tries to predict the masked words based on the other, non-masked words in the sentence. Next Sentence Prediction works with pairs of sentences for which the model attempts to predict whether they appear as two subsequent sentences in the text. Sentences are paired so that 50% of the input pairs are initially subsequent sentences, and the other 50% are randomly selected from the training corpus.

Both masked LM and next sequence prediction are used together during training, in which the optimizer minimizes the combined loss function of both strategies. Because neither strategy requires labeled data, the model can be trained in an unsupervised fashion on large corpora.

BERT has been a hugely successful language model, and many machine learning researchers have followed-up with related models, either focusing on further scaling the architecture and improving its performance (RoBERTa [48], XLNET [49]), or replicating its results using less expensive model architectures (ALBERT [50], DistilBERT [51]).

3.7 Text Representations

To make use of general machine learning algorithms introduced in Section 3.3, we need to represent our data using numerical feature vectors. In this section, we enumerate multiple options for building numerical representations from text data.

a	bag	of	words	has	inside	it	other
1	1	2	2	1	1	1	0

Table 3.1: Example of a bag-of-words representation

3.7.1 Bag-of-Words Representation

One of the common approaches is representing text using a bag-of-words model, where the vector represents word counts in a predefined dictionary. In this simplistic model, we completely ignore the words' positions and relationships and only work with an unordered set of words present in the sentence or text document.

An example representation of the sentence "A bag of words has words inside of it." is displayed in Table 3.1.

3.7.2 TF-IDF

Basic bag-of-words representations are not very efficient or robust. They do not consider the size of the document and carry a lot of unnecessary information about stop words — the most common words used in the language. These issues can be partially solved by scaling and filtering stop words, but a more advanced statistic called TF-IDF (term frequency-inverse document frequency) is often used instead.

TF-IDF captures the information about the importance of a word in a document. The value increases as the word appears more frequently in the document but is scaled down when the word is prevalent in many other observed documents. Therefore, the impact of stop words and common parts of speech that appear in all documents is reduced, and we can instead focus attention on the more defining set of words that carry the meaning of the specific document.

3.7.2.1 Term Frequency

Term frequency (Equation 3.6) captures the number of times a term occurs in a specific document. In its basic form, it equates to the unscaled bag of words approach mentioned previously.

$$tf(t, d) = \text{raw count of term } t \text{ in document } d \tag{3.6}$$

3.7.2.2 Inverse Document Frequency

Inverse document frequency measures how rarely the word appears across all documents. It is defined as an inverse fraction of the documents containing the term, which is scaled logarithmically.

$$idf(t, D) = \log \frac{|D|}{1 + |\{d \in D : t \in d\}|} \tag{3.7}$$

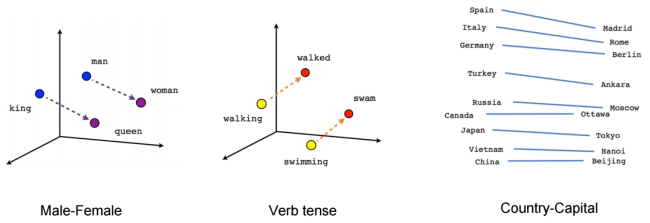

Figure 3.6: Similarity substructures in word2vec word embedding [8].

Equation 3.7 shows the formal definition where D is the set of documents (also called corpus) and $|\{d \in D : t \in d\}|$ is the number of documents where the term t appears. We adjust the denominator to avoid division by zero.

This value is small when the term t often appears across the whole set of documents D, and large when this term is only present in a small number of documents.

Finally, we obtain the TF-IDF statistic as a product of term frequency and inverse document frequency (Equation 3.8).

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D) \tag{3.8}$$

3.7.3 Word Embeddings

Word embeddings are learned numeric representations of text. Unlike simple bag-of-words models, they also aim to capture the meaning and the similarity of the words. Every word is mapped to a single fixed-sized real-valued vector with a dimensionality generally in the order of hundreds — depending on the size of the document, the resulting number of features can be much lower compared to bag-of-words models with large dictionary sizes.

Word embeddings are learned using neural network architectures, with the values in the resulting vectors obtained from the network's hidden layers.

The goal of the word embedding algorithms is to capture the similarity between words — similar words should have similar vector representations. Other interesting properties, such as noun genders, verb tenses, or information about capital cities, can also be captured by considering the context around the words, as demonstrated in Figure 3.6.

The first real breakthrough of this representation learning technique came with the publishing of the word2vec [52, 53] algorithm. The area of word embeddings is still an active research field and other notable algorithms such as GloVe [54] and ELMo [55] have been published since.

Overall, using word embeddings will give us richer feature vectors encapsulating additional information to improve the predictions of our call to action detector.

3.7.3.1 Word2vec

Word2vec [52, 53] is an algorithm developed by Mikolov et al. at Google, which is the de facto standard in the world of word embeddings. This algorithm's variants differ in the used learning model— continuous bag-of-words (CBOW) and skip-gram. In CBOW, the model learns the embedding by predicting the current word based on the words around it. The skip-gram model uses the opposite approach and learns by predicting the surrounding words given a current word.

Both have their advantages; the CBOW model is faster to train, but the skip-gram model tends to work better for infrequent words.

Additional tricks like hierarchical softmax or negative sampling can be used to make the algorithm more computationally efficient [56].

3.7.3.2 GloVe

Global Vectors (GloVe) is a word embedding algorithm developed at Stanford by Pennington et al. [54]. It aims the combine the context-based learning used in word2vec with the usage of global statistics found in classical representations by training on a word-word co-occurrence matrix computed across the whole training corpus.

It demonstrates the same desirable properties found in word2vec and outperforms it on word analogy, word similarity, and named entity recognition tasks in the evaluation of the proposing article.

3.7.4 Sentence Embeddings from Transformers

Because of the way a BERT neural network model (Section 3.6.1) is trained, quality sentence embeddings can be obtained from its hidden states, and the bi-directional nature of Transformer models can finally bring the desired positional and contextual properties into the vector representation.

A popular implementation of this approach is `bert-as-service` [57], which is available as a full-fledged client-server application. It obtains the embedding from the second-to-last hidden layer of BERT, which should avoid the embedding being too specific to the Masked LM and Next Sentence Prediction tasks that the model is trained on while still maximizing the amount of information about sentence meaning.

Sentence-BERT [58] is a modification of BERT specifically tailored for obtaining sentence embeddings, which is able to directly output a fixed-sized vector for the input sentence by adding a pooling layer. Authors are able to significantly speed up inference using smart batching while still maintaining the accuracy of BERT — it shows superior performance to both GloVe and bert-as-service embeddings in the SentEval benchmark [59] which evaluates the quality of sentence representations using a large set of NLP tasks [60]. The authors provide a Python library and pre-trained models [61], making it very easy to experiment with.

3.8 Call-to-Action Detector

While our goal is to classify the received e-mail into phishing and not phishing categories, this problem is difficult for a single, end-to-end classifier, and we also lack labeled data for this approach. Instead, we decided to classify the e-mails using an ensemble of smaller, focused detectors. Call-to-action detector, which aims to detect requests to perform potentially dangerous actions, is one of the components in our ensemble.

3.8.1 Call-to-Action Categories

Our call-to-action detector is essentially a text classifier working on the level of sentences in the e-mail message. We have defined a set of sentence categories that we want to identify, such as requests to open e-mail attachments or visit links, and a *not a request* category for regular sentences and edge cases that do not contain any behavior that we want to detect. A full list of categories paired with example sentences seen in the Enron [62] e-mail dataset can be viewed in Table 3.2.

Call-to-action category	**Example sentence**
contact request	For any inquiries, contact customer service.
data input request	Please fill in your card information now to avoid extra upgrade fees being withdrawn from your account later on.
link visit request	Please follow the link below and renew your account information.
open attachment request	You can find more information in the attached document.
other suspicious request	To restore your account access, please take the following steps to ensure that your account has not been compromised.
not a request	Make sure you never provide your password to fraudulent persons.

Table 3.2: Examples of labeled sentences for the call-to-action detector

For some of these sentences, we also add additional tags when encountering interesting parts of speech. These can be either conditional sentences that generally make the request a bit weaker, or urgency, forcing rash decisions by putting time pressure on the users. Secondary sentence tags are listed in Table 3.3.

One interesting observation about our data is that multiple of these categories and tags can appear in the same sentence. This fact brings some additional complexity to labeling, data handling, and the machine learning classifier's implementation, which should ideally produce a classification verdict with multiple categories instead of merely selecting the most probable one.

Call-to-action tag	Example sentence
contact request	For any inquiries, contact customer service.
data input request	Please fill in your card information now to avoid extra upgrade fees being withdrawn from your account later on.

Table 3.3: Examples of secondary sentence tags

3.8.1.1 Labeling Data

Because the task of detecting call-to-action in written communication is very specific, no labeled datasets are available. Therefore, we needed to label the data ourselves. Input sentences were taken from two public datasets — Enron [62] (fairly standard company e-mails) and Nazario [63] (dataset of spam and phishing e-mails). For this tedious process, we an open-source annotation tool called Doccano[64], which we packaged using Docker [65] and deployed on our private Kubernetes cluster.

3.8.2 Naive Bayes Classification on TF-IDF

In the first development stage of the call-to-action detector, we wanted to start with a simple classifier to set a baseline for the classification results. For our initial detector, we decided to use a naive Bayes classifier (Section 3.3.2 with a TF-IDF feature representation (Section 3.7.2).

Our codebase is in Python, so for implementations of classic machine learning algorithms, we use the widely adopted `scikit-learn` library. Despite its simplicity, this original implementation showed decent results during evaluation and gave us confidence that we would be able to identify call-to-action sentences with reasonable accuracy. However, as we show in the final comparison in Section 3.9.2, it cannot compete with the more sophisticated classifiers which we describe next.

3.8.3 Using Word Embeddings in Call-to-Action Detector

In the next iteration of the call-to-action detector, we decided to use the GloVe word embedding to construct the feature vectors. Although the training process of GloVe is complex, in practice, it is rarely necessary to perform this step because pre-trained word vectors computed on huge and diverse datasets are publicly available to download. We use a set of vectors trained on the Common Crawl corpus of text data collected from crawling the web, with a vocabulary size of about 1.1 million.

3.8.3.1 Aggregating Word Vectors

We face a new problem with the introduction of word embeddings into our detector is the aggregation of word vectors. GloVe embeddings are fixed 300-dimensional

vectors, but the sentences have a variable length. Therefore, it is necessary to combine these vectors to produce a final uniform-sized embedding for the whole sentence.

Although many sophisticated methods for creating sentence embeddings out of word vectors exist, simple aggregation schemes such as averaging the vectors produce decent results on real-world data [66, 67].

We can imagine the average of the word vectors as finding the center of mass — if many words from the sentence are projected to a similar area of the vector space, they probably give a good representation of the information in this sentence. Another similar technique is to take the maximum value for every dimension of the vectors.

It is important to note that although using word embeddings gives our feature vectors new information about the meaning of present words, we still ignore the context and word ordering in the sentence by condensing them into a single vector using averaging.

3.8.3.2 Finding Suitable Classifiers

In the early stages of development, we wanted to start with simple, readily available classifiers. Our codebase is in Python, so for implementations of classic machine learning algorithms, we use the widely adopted `scikit-learn` library.

`scikit-learn` contains implementations of many types of ML classifiers, such as logistic regression, SVMs, or random forests, and these algorithms often allow tuning through additional hyperparameters. Finding the best classifier by hand-tuning would be a long and tedious task, so we experimented with the `auto-sklearn` toolkit [68] to automatically find a suitable classifier and hyperparameter configuration on our labeled dataset.

After weighing the properties of tested methods and our experiments' evaluation results, we decided to stick with logistic regression as our baseline classification model.

Logistic regression is fundamentally a binary classifier, which was not a problem during the early experimentation. Our labeled data was still in the format of binary labels — `request` and `not a request` sentences. Later with the introduction of more specific categories shown in Section 3.8, we were required to switch to multi-class classification. We eventually solved this using a one-vs-rest strategy that transforms the problem into multiple tasks of predicting whether the data belongs to a selected class or one of the other remaining ones.

3.8.4 Using Sentence Embeddings in Call-to-Action Detector

In the next evolution of our call-to-action detector, we decided to use Sentence-BERT embeddings (Section 3.7.4) to improve our baseline logistic regression classifier's accuracy.

With this iteration, we also performed a relabeling of our data and moved from a binary `request` / `not a request` classification to the multi-class categories shown in Section 3.8, and started to evaluate the performance for the individual classes as well. This required using the classifier in a one-vs-rest fashion, as described in Section 3.8.3.2.

Note that while training of the simple classifier remains trivial, with only the feature vectors' size increasing slightly, this approach is much more expensive than the previous GloVe averaging. To obtain the feature vectors from training data, we have to run the full model inference on the input sentence instead of simply looking up pre-trained word vectors in a dictionary. However, we have seen a noticeable improvement in prediction performance, which led us to believe a neural network-based classifier is necessary to bring the call-to-action detector to a production-ready state.

3.8.5 Call-To-Action BERT Classifier

After experimenting with BERT sentence embeddings, we wanted to see how a full-fledged end-to-end classifier would perform on the task of call-to-action detection. For neural network models in the scale of BERT, it is very expensive to perform the whole training process, as these model architectures require learning a huge amount of parameters on massive datasets, often with billions of data points. Instead, a technique called transfer learning is often employed — we start from a base pre-trained model and adapt it to our specific task. BERT is a general language model that we want to use to solve a classification problem. Therefore, we need to modify its structure to suit our specific task of classifying call-to-action sentences.

To achieve this, we add a fully connected layer after the last hidden layer of BERT, sized to have one node for each call-to-action category. To shape the values into probabilities, we require an activation function. In standard multi-class classification, a softmax function is typically used in this final layer — all outputs will sum to 1, and each node then represents the probability of that class. However, we require our model to perform multi-label classification, which means that all output values need to be independent probabilities between 0 and 1, and multiple of these can be high or close to 1. For instance, it is quite common for the call-to-action sentences we encountered to also contain urgency — these types of sentences should have high probability values for both of these labels. For this reason, we use a sigmoid function, which we have already seen in logistic regression, as the activation function in the output layer.

To implement this custom neural network classifier, we used a BERT implementation from the `transformers` library [69]. `transformers` add a layer of abstraction on top of the popular Pytorch framework [70], providing efficient implementations of popular Transformer architectures and training loops, as well as fast tokenizers written in Rust. Additional classification layers were added directly with the use of the Pytorch library.

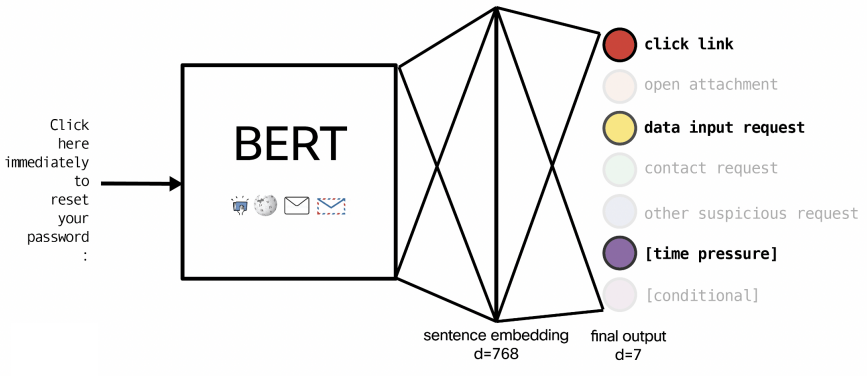

Figure 3.7: Overview of Call-to-Action BERT classifier. Credits go to my colleague Marc Dupont.

For learning the model, we transform the labels in our training data into one-hot encoded vectors — each dimension represents one category, we insert 1 for every tagged label; the rest are filled with zeros. This is the desired result of our output layer — for the sentence displayed in Figure 3.7, this vector would be $[1, 0, 1, 0, 0, 1, 0]$.

Because the model has been thoroughly trained on a large English language dataset, we retain the general understanding of the language in the representation from BERT layers and then fine-tune our additional classification layers to solve the task of call-to-action detection. This requires far less computation work than a situation where we would start training the model from the ground up instead of reusing the pre-trained weights, and decent results can be obtained after only a couple of training epochs.

With this technique, even though our dataset size was only in the order of tens of thousands data points, which is fairly small for a neural network classifier of this scale, we were still able to improve the performance significantly, as we show in Section 3.9.2.

3.8.6 Summary of Call-to-Action Detector Evolution

In this section, we summarize the evolution of our call-to-action detector. Initially, we started with a baseline naive Bayes classifier using a simplistic TF-IDF representation. To inject word meaning into our representation, we switched to a representation using the GloVe word embedding and trained a logistic regression classifier on sentence vectors obtained through averaging. To further enrich the representation with information about sentence context, we used a sentence embedding from the Sentence-BERT Transformer model. Finally, we upgraded our detector to utilize the full power of neural networks, and developed an end-to-end multi-label classifier using a fine-tuned BERT model.

3.9 Evaluation

Finally, we define a suitable set of evaluation metrics and show the evaluation results of the call-to-action detector, which demonstrate the differences in the prediction power of the described classifiers.

3.9.1 Evaluation Metrics

Firstly, let us focus on evaluating the performance for a binary classification task, where we define one class which we aim to detect as positive, and the other as negative. In our phishing detection task, the positive class would denote phishing e-mails and the negative class benign e-mails.

Given a labeled set of data, we can divide the predictions from our classifier into four categories:

True positive (TP) A positive sample correctly identified as positive.

False positive (FP) A negative sample incorrectly identified as negative.

True negative (TN) A negative sample correctly identified as negative.

False negative (FN) A positive sample incorrectly identified as negative.

Using these values, which form a confusion matrix, we can compute several evaluation metrics. For evaluating the multi-class output of the call-to-action detector, we can use a similar approach to measure performance separately for each class.

Accuracy (the ratio of correctly identified samples) and precision (the ratio of positive predictions that were truly positive) are widely used for evaluating statistical models. However, when used alone, they are unsuitable for evaluation on highly imbalanced data such as e-mails where the benign class has a strong majority compared to the phishing e-mails. Imagine that out of 1000 received e-mails, one is a phishing e-mail. Suppose we construct a trivial classifier, which flags every e-mail as benign. In that case, we will achieve a seemingly satisfying accuracy of 99.9%, even though our classifier fails to detect any phishing e-mails. Similarly, a classifier can achieve high precision by flagging only obvious phishing cases, but such a model will produce a large number of false negatives.

The shortcomings of these traditionally used metrics for evaluation on imbalanced data are thoroughly described in [71], and the authors present more resilient approaches in [72].

3.9.1.1 True Positive Rate

True positive rate (TPR), also called recall or sensitivity, tells us the proportion of positive samples that were correctly identified as positive (Equation 3.9). Note that a classifier can trivially achieve 100% recall by predicting the positive class for every

sample. Therefore, it is often paired with precision to provide a better overview of predictive performance.

$$TPR = \frac{TP}{TP + FN} \tag{3.9}$$

3.9.1.2 False Positive Rate

False positive rate (FPR) tells us the proportion of negative samples that were incorrectly identified as positive (Equation 3.10).

$$FPR = \frac{FP}{TP + FN} \tag{3.10}$$

3.9.1.3 ROC Curve

Given that our underlying classification model provides the probability of the output class, we can use thresholding to plot TPR and FPR in various settings. The resulting plot is called a receiver operating characteristic (ROC) curve and helps us visualize the tradeoff between the number of correct positive predictions and the number of false positives.

Area under the curve (AUC) can be used to provide a single value characteristic of the model, which is unaffected by class imbalance. However, we often want to restrict the area to a smaller region, as classifiers with high FPR are often useless in practice.

3.9.2 Evaluation of Call-to-Action Detector

For evaluation of the call-to-action detector, we select several operating points on the ROC curve, which displays the relation between true and false positive rates. We calculate the area under the ROC curve for a region between 0 and 10% FPR and also compute the exact TPR values at 10%, 1%, and 0.1%. For simplicity, our main evaluation metric to show the results of the described classification methods will be the AUC of the ROC curve truncated at 10% FPR and scaled to provide a characteristic value between 0 and 1.

We measure the model efficacy on a dataset of hand-labeled sentences extracted from Enron and Nazario datasets (Section 3.8.1.1). For a more robust evaluation, we use 10-fold cross-validation with stratified splits and average the AUC metric overall cross-validation runs.

In Table 3.4, we show the performance of linear classifiers with multiple feature representations introduced in this chapter. This evaluation is performed for the binary call-to-action classification task, where all of the call-to-action categories (Table 3.2) are collapsed into a single call-to-action positive class. With increasingly complex sentence representations, we can see a significant efficacy boost, and the final classifier learned on BERT sentence embeddings provides the best predictive performance in this benchmark.

Model	ROC AUC (truncated at 10% FPR)
TF-IDF Naive Bayes	0.615
GloVe Linear Regression	0.650
Sentence-BERT Linear Regression	0.776

Table 3.4: Comparison of classic machine learning classifiers on the call-to-action detection task using multiple sentence representations. ROC AUC measured on a labeled subset of sentences from Enron and Nazario datasets and averaged over 10 cross-validation folds split as 90% train and 10% test sets.

Call-to-action class	Sentence-BERT LR	BERT NN
Open attachment	0.778	0.936
Link visit	0.788	0.881
Data input	0.574	0.716
Contact	0.826	0.850
Other suspicious request	0.494	0.572

Table 3.5: Comparison of a linear regression classifier on Sentence-BERT sentence embeddings (Sentence-BERT LR) and BERT neural network classifier (BERT NN) performance on the call-to-action detection task. The compared metric is the ROC AUC trucated at 10% FPR measured on a labeled subset of sentences from Enron and Nazario datasets and averaged over 10 cross-validation folds split as 90% train and 10% test sets.

In Table 3.5, we compare the efficacy of the best performing linear classifier to a fine-tuned neural network classifier on the task of multi-class call-to-action detection where we aim to identify the specific request category. For the linear regression classifier, we use a one-vs-rest approach, whereas the output of the BERT neural network is inherently multi-class. We can observe that the neural network classifier dominates the linear regression classifier on every class, even when this model works with sentence embeddings from a similar BERT language model.

Chapter 4

Classification Workflow

In this chapter, we introduce the workflow of our classification engine. First, we give a brief overview of the entire processing pipeline. Then, we describe its stages in more detail and discuss related challenges and our proposed solutions.

4.1 Overview

Our classification engine analyzes e-mail files using a set of detectors and aggregates the detections into a classification verdict (Figure 4.1). First, we need to preprocess the e-mail into a suitable data structure – we parse and properly decode the e-mail, extract the relevant text content, and tokenize it into words and sentences. Then, we analyze this tokenized document using multiple detectors and collect the detections, with an option to preemptively stop the pipeline based on certain properties of the input data. Finally, we aggregate the collected detections using a scoring mechanism and produce a verdict.

4.2 Preprocessing

In the preprocessing stage, we transform the input e-mail file into a tokenized document structure, which abstracts the access to sentences and word tokens.

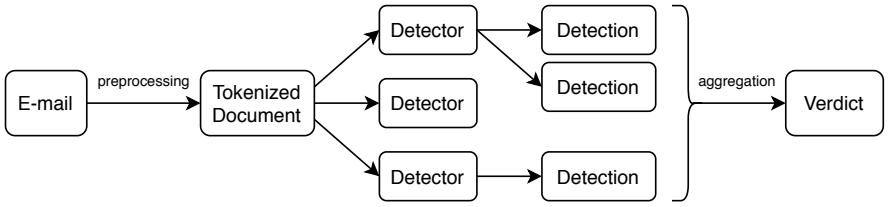

Figure 4.1: Diagram of the phishing classification workflow.

4.2.1 E-mail file format

Although multiple e-mail formats exist, we focus on describing the standard EML file format defined in RFC-822 [73] and later enhanced in RFC-2822 [74], as this is the format we expect as the input to our classification engine.

EML files contain two sections – headers and message body. Headers contain the e-mail metadata in the form of a list of key-value pairs. They carry information about the e-mail sender and recipients, the e-mail subject, message timestamp, and potentially other custom data.

The message body contains the primary information of the e-mail. It consists of payloads described by their `Content-Type` – these can contain different types of data such as text, HTML content, or file attachments. Multi-part e-mails contain multiple payloads that are nested in a tree structure. An example of a multi-part e-mail is available at [75].

4.2.2 Parsing

The first preprocessing step is to parse the input e-mail file into a Python data structure. For this, we use the standard Python parsing library [76].

The main complexity of e-mail parsing comes from content decoding. Payloads can be encoded using two types of encoding – content transfer encoding (i.e., using `base64` or `quoted-printable`), and character encoding (i.e., `ASCII` or `UTF-8`). To process the payload, we first check for the content transfer encoding header and decode the payload if necessary. Using the specified character encoding (charset), we then transcode the message to the standard Unicode UTF-8 encoding, which is the default for Python strings. Given a correctly annotated e-mail payload, this gives us uniform, decoded text content.

After decoding the payloads, we save them into our custom e-mail structure together with a selected set of parsed headers. For our phishing classifier, we separate the payloads into plain text payloads, HTML payloads, and other non-text payloads that we currently do not analyze.

4.2.3 Text Extraction

Once the e-mail is parsed, we need to extract input data for our content-based detectors. Our engine does not analyze e-mail attachments, and we classify only text data. Therefore we extract content from the plain text and HTML payloads described in Section 4.2.2.

For plain text payloads, the bulk of the work is already in the parsing stage, which gives us a decoded Unicode string representation. HTML payloads require additional care, as we do not want to pollute our data with HTML artifacts such as tags, scripts, or stylesheets. To clean up the payload, we remove all `<script>` and `<style>` elements, and then concatenate the text content extracted from all other elements. For parsing the HTML, we originally used the BeautifulSoup [77] library, but we recently switched to directly calling the `lxml` [78] parser due to performance

reasons. With this faster parser, we were able to speed up this process about five times compared to its original performance.

Complications arise from wrongly annotated content types of payloads, i.e., an HTML body annotated as plain text. These issues can be partially solved using heuristics to predict the actual content type or by always attempting to parse the payload as HTML, as the `lxml` parser is robust and will not mangle non-HTML data. This solution can, however, slow down the extraction process.

Extracted content can still contain a lot of unnecessary information and artifacts. Because the inference of some of our models is expensive, we want to limit the amount of processed text to keep our response time and costs down. To further process the content, we use the Talon library [79], which extracts e-mail reply and signatures using a method inspired by Carvalho et al. [80]. This allows us to isolate the latest reply, which contains the vital message of the e-mail, and ignore, i.e., old quotations in a long conversation thread that had already been examined in the past. We do not currently use e-mail signature extraction, but we consider it for a future, signature-based detector.

4.2.4 Tokenization

Our detectors rarely analyze the entire message at once – some detectors, such as the call-to-action detector (Section 3.8), expect individual sentences as the input; other detectors perform matching for individual word tokens, as is the case in our cryptocurrency address detector. Therefore, after extracting the text content from the payloads, we want to split the message into sentences and words and store this representation in a way that allows access from multiple detectors and avoids repeating unnecessary work. In our engine, we implemented the tokenization using a lazy approach, where the representation is computed the first time a detector requests it and subsequently cached for later use.

For both tokenization and sentence segmentation, we use the SpaCy [81] Python library, which includes a fast tokenizer, multiple implementations of sentence segmentation and allows for easy customization of the processing pipeline. Word tokenization is fast and straightforward; we use whitespace characters to split the string and check for a predefined set of special cases – details about SpaCy tokenization are available at [82]. An essential property of the SpaCy Tokenizer is that it allows reconstructing the original text from the tokenized representation – this allows us to save memory by saving only the tokens.

Sentence segmentation is a more complex problem. By default, SpaCy solves it using a pre-trained neural network-based dependency parsing model. Details about the model architecture are available from [83]. E-mail tokenization currently takes up a significant portion of the total processing time, and most of this time is spent dependency parsing required for separating the sentences. Therefore, we compared multiple sentence segmentation approaches available in SpaCy or easily integrated libraries see the potential speedup (Table 4.1).

Algorithm	Average processing time [ms]	Speedup over parser
Dependency parser	248	
Sentencizer	51	4.86×
PySBD	189	1.31×

Table 4.1: Performance comparison of SpaCy sentence segmentation algorithms. We measured the average processing time per e-mail on a internal e-mail corpus of 50 thousand suspicious e-mails from customers and honeypots.

Dependency Parser This is the default setting for SpaCy, which uses a statistical, neural network-based model to predict the sentence boundaries. It provides accurate detection, but its speed can be limiting for large workloads.

Sentencizer An alternative, rule-based sentence segmenter implemented in SpaCy. It checks for common end-of-sentence tokens, which makes it much faster compared to the default parser. However, it sometimes fails to split sentences correctly and leaves them unnecessarily long, especially in HTML e-mails where sentences and paragraphs can be separated visually using HTML elements.

PySBD A recent library for sentence boundary disambiguation which integrates with the SpaCy pipeline. Authors claim a significant improvement in both the quality of sentence segmentation and speed [84].

For now, we decided to continue using the default dependency parser implementation, as this was the solution we used while preparing training data for our statistical models. After examining the differences in tokenization, we cannot be convinced that the faster Sentencizer approach would give us comparable sentences. We are also considering other options, such as implementing a fast tokenizer in Rust and connecting it to our engine through Python bindings if tokenization becomes a significant bottleneck as we continue to improve and optimize the system.

4.3 Detectors

After the e-mail is preprocessed, it enters our classification pipeline. During this process, the e-mail is sequentially analyzed using a set of detectors, and detections are collected for the subsequent aggregation (Section 4.4).

A detector analyzes a preprocessed e-mail file and produces a list of detections. Individual detectors can detect a broader set of behaviors; for instance, the call-to-action detector recognizes different categories of requests as well as urgency (Tables 3.2 and 3.3).

The pipeline is designed to handle an arbitrary number of detectors. It can be easily modified and allows us to update it with new detectors or disable the underperforming ones. At the time of writing this thesis, a full run of our classification

pipeline passes the e-mail to fifteen detectors. Many of these are simple, targetted classifiers for standard phishing techniques described in Chapter 2.2, some of the more intricate detectors are enumerated in the following list.

Call-to-Action Detector Detects requests for potentially dangerous actions in the e-mail message using a BERT model fine-tuned on a large set of annotated sentences from e-mail corpora. A more detailed description can be found in Section 3.8.

Credential Phishing Detector Detects attempts to persuade the user into giving up his credentials. This detector was originally using simple keyword matching against a predefined set of credential related terms. However, this approach led to many false positives on legitimate e-mails, such as e-mails for resetting forgotten passwords. This classifier is now based on zero-shot learning [85], where we provide sets of phishing and legitimate sample sentences related to credentials and measure similarity to both categories using a sentence embedding from a pre-trained RoBERTa model.

Identity and Relationship Detectors We are currently developing a set of detectors that utilize a relationship graph of past communications between users. With these models, we will gain access to important signals, such as whether the analyzed e-mail is the first communication between the two accounts. We expect to identify a large number of benign e-mails using these fast detectors based on e-mail headers, which will then allow us to analyze the remaining suspicious e-mails using more demanding NLP models.

Checkers are a special case of detectors that give a binary `true/false` result for the e-mail and allow us to stop the pipeline preemptively. For instance, we have a language checker that examines whether the e-mail content appears to be English. If it is not, we skip running our expensive NLP models that are trained on English sentences.

Detections are represented using list of data classes with the following fields.

Detection code Identifies the triggered detector and type of detection (e.g., `CTA_LNK` identifies a link visit request from the call-to-action detector).

Score A value between 0 and 1 reflecting the confidence of the detector or severity of the specific detection type.

Metadata Contains additional information about the detection; inner fields vary between detectors. For instance, the link masquerade detector (Section 2.2.3.1) stores the display text and real destination URL.

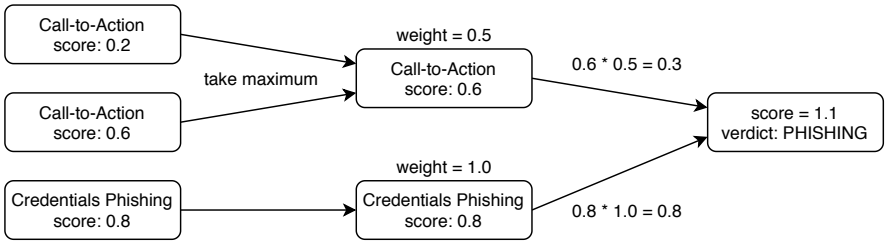

Figure 4.2: Example of the detection aggregation process assuming a phishing threshold of 1.0. Three detections are grouped into a single phishing verdict.

4.4 Aggregating Detections

After we obtain the detections from all detectors in the pipeline, we need to aggregate them into a final verdict. While, i.e., detections from the call-to-action detector should contribute to the verdict only in combination with other phishing signals, detections of blatant link manipulation (Section 2.2.3) or repeated use of confusable characters (Section 2.2.1.3) are more suspicious. This is reflected in both the detection score and the final aggregation weight of the detector.

Our current detection aggregation process uses a linear combination of the detection scores. To avoid triggering a phishing verdict based on a large number of low severity detections, we first group detections of the same type and keep only the one with the highest score. Then, we compute a weighted sum of these detections and check whether the total score passed a selected threshold, in which case we flag the e-mail as phishing (Figure 4.2).

The weights for individual detector types were initially selected based on intuition and our perception of their severity and later manually fine-tuned by checking the classification performance on an annotated subset of our internal suspicious e-mail corpus. We experimented with ensembling based on rules obtained from rule-mining using classification trees and clustering as an alternative to the weighted approach. However, we are not currently using this solution in production.

Chapter 5

Phishing Detection Engine API

In Chapter 4, we demonstrated the isolated detection workflow of our classification engine. In reality, our classification engine is a standalone component that communicates with other connected services through an application programming interface (API). This chapter introduces the selected frameworks and technologies for our API, gives a brief description of our server application, and shows techniques we used to make the solution efficient with the limitations brought by using the Python programming language.

5.1 Server Requirements

Our application is a simple server that uses the standard REST architecture with the HTTP protocol. It parses the input e-mail received though a `POST` request, feeds it through our classification pipeline (as described in the previous chapter), and sends a response with JSON-encoded detections.

Because our core pipeline is written in Python, we also want to use it to implement the server for ease of integration. Our proposed solution comprises a server application written in Flask, the Nginx web server, which handles incoming requests, and the Gunicorn application server, which connects the server to the Python application (Figure 5.1). We use Datadog for collecting application logs and monitoring, and we package all of the components using Docker to create a portable image that is easy to deploy.

5.2 Flask Application

Flask [86] is a web framework for Python that is commonly used for basic web applications because of its minimal interface. As such, it is a good fit for our classification API. This application is a simple wrapper around the classification pipeline described in Chapter 4 and exposes the engine through the following endpoints.

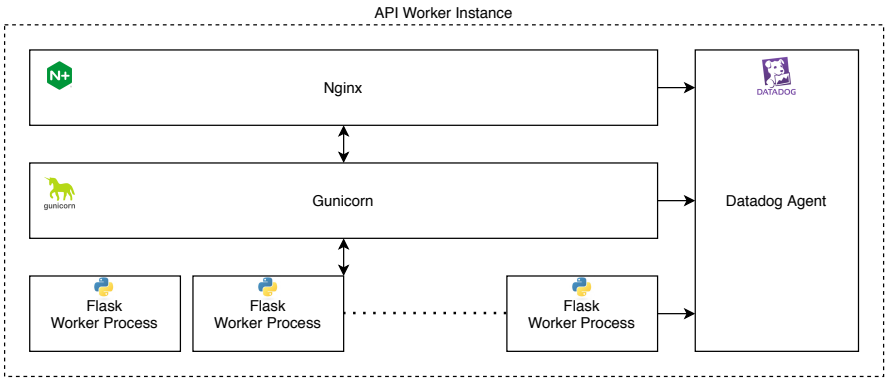

Figure 5.1: Overview of the API server components.

`POST /scan` The main endpoint for e-mail analysis. Parses the e-mail file from the request body, processes it with the classification pipeline, and sends back a JSON-encoded response with the discovered detections, aggregated verdict, and additional metadata such as the message ID or response time.

`GET /stats` We also provide an endpoint which shows statistics about the processed e-mails, such as the number of processed e-mails or the number of positive verdicts. Additional details about the detections are saved in application logs, which we capture through Datadog (Section 5.5).

5.3 Gunicorn Application Server

Gunicorn [87] is a Python Web Server Gateway Interface (WSGI) server which we use to host our Flask server application. WSGI is a standardized interface for communication between Python applications and web servers, as described in PEP-3333 [88]. Gunicorn manages multiple worker processes running the Flask application, monitors and restarts them as needed. It also distributes the incoming requests across these workers and sends the responses to the webserver.

While Python language is a solid choice for machine learning and data science operations, the language has its limitations, which require special care when using it to write performant server code. In addition to the overall performance loss compared to languages like C++, Rust, or Java, we cannot efficiently use a threading model that is generally used with these languages to build fast servers. Because of the global interpreter lock (GIL) [89] present in the Python interpreter, one Python process can only utilize a single CPU core, even when running multiple threads. Therefore, to fully utilize multicore processors, we need to parallelize on the level of processes. Gunicorn allows us to perform parallelization through its worker process architecture, which will adequately utilize the CPU, given that we continuously receive multiple requests at the same time.

Because some of our models are large, we also want to avoid loading them in every worker process, as these objects are read-only and do not change throughout the process lifetime. Gunicorn uses a preforking model to create its worker processes – we can use the copy-on-write property of fork to our advantage [90]. We first preload the application and all of our models into memory, then let Gunicorn perform the forking. Since the API application is stateless and does not modify the loaded objects, the memory can be shared between the workers without copying it.

5.4 Nginx Web Server

Nginx [91] is a high-performance open-source web server that we use as a proxy server for Gunicorn. It allows for greater control over the server configuration, and we use Nginx mainly for rate-limiting the requests. If we encounter a spike where we receive a number of e-mails that we cannot process using the current infrastructure, we reject some of these requests instead of attempting to process all of them, which would clog and slow down the service. The accepted requests are passed to the Gunicorn application server and then processed with the Flask application.

5.5 Datadog Agent

Datadog [92] is a cloud monitoring platform that we use to monitor and profile our engine. We use Datadog to collect application logs, collect metrics from Gunicorn and Nginx servers, track and time the runs of critical code sections, and finally, to perform general profiling of our Python application code.

Datadog has been essential in helping us find demanding areas of code and identifying performance bottlenecks, and most optimizations of our system were performed based on feedback from this platform. It also allows us to monitor the status of the underlying infrastructure, send notifications about application errors, and to get a high-level overview of the detections and verdicts produced by our engine through a comprehensive dashboard.

Initially, we considered using the ELK Stack [93], which allows for a similar monitoring functionality. ELK Stack uses a combination of Elasicsearch, Logstash and Kibana, which are free, open-source technologies. However, we decided that the ease of use and added functionality in Datadog were worth the additional costs of this managed solution.

To utilize Datadog, we need to install a Datadog Agent daemon process into our environment. Datadog Agent runs in the background, collects the application logs and metrics and periodically sends them to the Datadog servers.

5.6 Docker Container

Docker [65] is a system for packaging and running applications in isolated environments called "containers" using virtualization. We use Docker to package all of our published components, as it allows us to standardize the environment and ship a production-ready version of our engine with all necessary dependencies and pre-trained models.

We start the Docker build from a base OS image, in our case Ubuntu 18.04 LTS. First, we install all required system dependencies, upgrade the default Python version, and install Python dependencies through system package managers. Then, we download public models and our custom pre-trained models from cloud storage. Finally, we copy our application code into the container.

We create separate Docker images for all described services and orchestrate their startup and interactions using Docker Compose [94], a tool for managing multi-container applications. This approach is very flexible and allows us to easily extend the system with additional components as we improve the engine. The resulting images encapsulate all required dependencies and can be deployed on a cloud instance in Azure in a matter of seconds.

Chapter 6

Deployment

In this chapter, we describe the process of hosting our classification engine API in a cloud environment and introduce the tooling used to automatize this process.

6.1 Tooling

In this section, we give a brief introduction to the Microsoft Azure cloud platform used to host our engine, Packer and Terraform tools that help us provision, version, and manage the infrastructure, and finally, the Datadog platform used for monitoring.

6.1.1 Microsoft Azure

Microsoft Azure [30] is a cloud computing platform that we use to deploy our engine. Like its main competitors, Amazon Web Services and Google Cloud Platform, it offers a broad spectrum of computing, infrastructure, and data storage services. Hosting our infrastructure in Azure is a strict requirement for us because our engine analyzes data from the related Microsoft 365 service, and the analyzed customer data cannot leave the Azure cloud. For our engine, we use a combination of virtual machines, load balancers, and key-value storage, intercommunicating inside of a virtual network.

6.1.2 Packer

Packer [95] is a tool for automation of virtual machine provisioning. As shown in Section 5.6, our system components are packaged as Docker images. Azure supports running these components directly through its Container Instance service, but this approach lacks customizability of underlying infrastructure and networking. Instead, we want to leverage the wide array of available virtual machine instance types, select the most suitable type for every component, choose how the components are grouped, and explicitly configure their communication.

To achieve this, we use a two-layer approach where we manually orchestrate Docker containers inside of custom virtual machines managed by Packer. This way, we retain the flexibility of using specific instance types, but we can still develop and publish fully packaged images through Docker. The Packer images are built inside of Azure by provisioning a virtual machine according to a supplied build script which installs the necessary system packages and prepares the system environment. Then, the virtual machine is powered off, and its state is captured into a reuseable image saved in the Azure Image repository.

Using this technique, we are able to ship updated production images without any significant changes to the virtual machine environment by merely including a newer version of Docker images.

6.1.3 Terraform

Terraform [96] is a tool for managing cloud infrastructure as code. After manually testing the infrastructure deployed through Azure Web Console and settling on a suitable setting, we use Terraform to define this setup with Terraform scripts, which gives us a persistent, reusable representation of the infrastructure. With Terraform, we can perform consistent deployments, reuse predefined modules, and easily upgrade our instances with newer images. Managing infrastructure as code also enables versioning, improves the reviewing process, and simplifies potential migration to a different infrastructure provider.

6.2 Infrastructure

In this section, we describe the set of Azure resources used to host our engine. We deploy our prebuilt Packer images using a virtual machine scale set and use a load balancer to provide uniform access to the scale set. Our resources are grouped inside a virtual network, and to maximize the efficiency of communication between our services, we utilize proximity placement groups.

6.2.1 Virtual Network

All of our resources share a single virtual network. This allows tight control over the security configuration and enables the services to communicate without being exposed through a public IP address.

6.2.2 Virtual Machine Scale Set

Because our engine needs to scale to handle data from millions of mailboxes, running it on a single virtual machine is insufficient. Instead, we use virtual machine scale sets to start multiple instances with our Packer image, unify their configuration, and scale the number of instances as needed.

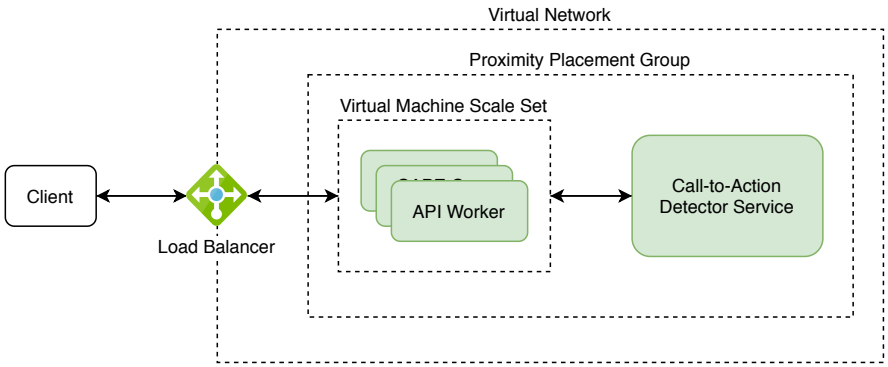

Figure 6.1: Overview of the Azure deployment infrastructure.

The scale set also allows dynamic scaling based on the current server load. Although we do not utilize this feature at the current stage, it can help handle traffic spikes, as we will process the majority of e-mail messages during US working hours.

Running multiple instances inside a scale set gives us the ability to perform seamless image upgrades. We can take down several instances and restart them with the updated image while other instances remain active, as they are either running the previous version or are already upgraded – this way, we can perform the upgrade without any service downtime.

6.2.3 Load Balancer

To evenly distribute traffic across the instances, we use a load balancer. This allows us to access the virtual machine scale set in a uniform fashion using a single endpoint in the virtual network.

Application Gateway is an advanced load balancer alternative available in Azure, enabling TLS termination at the load balancing level, making it significantly easier to maintain. We are currently using the basic load balancer, but the application gateway gives us the option to secure the communication even between the services inside the isolated virtual network.

6.2.4 Proximity Placement Group

Proximity placement groups [97] allow placement of selected Azure resources physically close together to minimize the latency of communication between them. We utilize these groups to enable blazing-fast communication between the virtual machines hosting our engine and other specialized services, such as the optimized call-to-action service, which we will introduce in Chapter 7, or the planned identity and relationship service.

Chapter 7

Optimizing Neural Network Inference

In this chapter, we propose an overhaul of the call-to-action detector described in Section 3.8.5, which significantly reduces its processing time. We extract the detector into a separate service, which gives us the flexibility to select optimal infrastructure for it. Next, we explore the possibilities for speeding up the model inference through alternative model architectures and model optimizations. Finally, we show two approaches for connecting the service to the engine core and discuss their strengths and weaknesses.

7.1 Call-to-Action Detector Service

State-of-the-art performance of Transformer models comes at a cost, as these large neural network models can be very resource-demanding. After the initial deployment of our engine, it became clear from our Datadog profiling data that these more complex models, most notably the BERT model used by the call-to-action detector, were taking up a majority of the processing time and that it was not sustainable to scale them to handle larger amounts of data. For lengthy e-mails, we would regularly observe response times in the order of seconds, and these requests would slow down our whole system under a higher load.

The small, general-purpose Azure instances we used for hosting the application were not suited for processing-heavy tasks such as neural network inference, especially in scenarios with larger amounts of worker processes. As described in Section 5.3, our application runs multiple worker processes to handle incoming requests and process them in parallel. When the call-to-action detector ran simultaneously in multiple workers, the BERT model would attempt to utilize all CPU cores, which resulted in large latency and inefficacies from frequent context switching.

To mitigate these issues, we decided to extract the call-to-action detector into a separate service, in which we could select more suitable instances and have better

control over the running processes. In this isolated service, we could also easily experiment with different model architectures and libraries.

7.2 Selecting Suitable Infrastructure

Selecting suitable instance types for the call-to-action detector service was the natural first step. The cost-optimal type could be decided based on general neural network inference performance and should remain unaffected by the future optimizations and changes to the model architecture. If the instance provided fast inference on the benchmarked model, it would likely transfer to other models.

Because the call-to-action service does not have significant memory requirements, the decision comes down to comparing the performance and costs of compute-optimized CPU instances (F series [98]) and GPU compute instances (NC series [99]) on Azure.

The parallel architecture of GPUs is well adapted for vector and matrix operations, and powerful GPU clusters are generally used to train large neural network models — this process can be very demanding and take days without a proper setup [100]. However, we have observed that for model inference, the performance gap between GPUs and CPUs is not extremely large when using modern processors with strong vectorization support, such as the Intel Xeon Scalable processors [101] available in Azure. We also found CPU instances easier to scale and work with, as we do not need to install specialized drivers or implement GPU-specific codes.

In the benchmark, we compare the average latency and throughput (in sequence queries per second) of the base BERT model (`bert-base-uncased` [102]), which we later fine-tune to classify call-to-action sentences, on the following CPU and GPU instances.

`F16s v2` CPU compute-optimized instance with 16 Intel Xeon Platinum 8272CL processor cores. This generation of processors includes Deep Learning Boost instructions [103] specifically designed to speed up machine learning processing. Costs $0.776 per hour in the West Europe region.

`NC6 v3` GPU compute instance with a single NVIDIA Tesla V100 GPU and 6 CPU cores. Costs $3.823 per hour in the West Europe region.

For consistency, the benchmark is run using the ONNX Runtime accelerator for model inference and includes graph optimizations, which will be thoroughly covered in the next section. We run inference of the model on sequences of 32 random words using multiple batch sizes. We repeat the inference on 100 samples and average the latency and throughput over ten independent runs. To compute the cost per e-mail, we assume that one e-mail has on average ten sentences (represented by our random sequences) and that the service is running in the West Europe region. Benchmark results are presented in Table 7.1.

Batch Size	**Average Latency** [ms]		**Throughput** [QPS]		**Cost per 1k e-mails** [US$]	
	CPU	GPU	CPU	GPU	CPU	GPU
1	9.35	1.64	106.86	608.88	0.0201	0.0174
8	45.95	4.90	174.11	1633.96	0.0124	0.0065
16	88.08	8.69	181.66	1841.48	0.0118	0.0058
32	174.61	16.11	183.27	1985.99	0.0118	0.0053
64	351.99	30.65	181.82	2088.40	0.0119	0.0051
128	724.13	59.66	176.76	2145.43	0.0122	0.0049
256	1472.65	114.72	173.84	2231.59	0.0124	0.0048
512	3076.57	231.13	166.42	2215.15	0.0130	0.0048

Table 7.1: Comparing `bert-base-uncased` processing throughput and cost on CPU (`F16s v2`) and GPU (`NC6 v3`) instances in Azure.

GPU processing efficiency is most noticeable when processing larger batches of sequences together, whereas increasing the batch size tends to have a slightly negative effect on CPUs. As currently designed, our call-to-action service is processing single e-mails and immediately returns the response — this means that the batch sizes are in our use case equal to the number of sentences in the e-mails, expected to be around 10 to 20 on average. There are possibilities for further optimization here, but accumulating e-mails to process them in batches might have a negative effect on the service response times with the current volume of analyzed e-mails and would significantly increase the complexity of the service.

Although GPU instances offer a significantly better cost per e-mail, they are not available in Azure for some of our production regions. Therefore, we are currently using compute-optimized CPU instances for services running neural network inference – specifically, the `F16s v2` instances from the benchmark, which we will be using to measure the impact of other optimizations presented in this chapter.

7.3 Optimizing Model Inference

In this section, we focus on speeding up the model inference in our call-to-action detector service. We can select a different language model architecture with fewer layers, parameters, or more efficient operations to achieve this. To achieve an even greater speedup, this model can be further optimized, i.e., by quantizing the model weights or transforming the neural network's computation graph.

7.3.1 Alternative Model Architectures

A large number of new models building on BERT have been published since its release, either attempting to improve its predictive performance or to reduce its computation cost while achieving similar accuracy. The models using the latter approach are good candidates to speed up our call-to-action detector service fur-

Sequence Length	Average Latency [ms]		
	BERT	DistilBERT	SqueezeBERT
4	16.43	10.00	20.06
8	19.88	12.68	20.73
16	56.77	30.63	52.99
32	51.06	29.94	30.05
64	65.43	35.21	31.01
128	81.45	48.38	46.38
256	140.39	85.91	111.30

Table 7.2: Comparing inference time for BERT (`bert-base-uncased`), DistilBERT (`distilbert-base-uncased`) and SqueezeBERT (`squeezebert-uncased`) PyTorch models using random inpits and different sequence lengths on `F16s v2` instances. Latency is averaged over 100 samples.

ther. In this section, we describe and benchmark two BERT alternatives to show the potential speedup from using a different model architecture.

7.3.1.1 DistilBERT

DistilBERT [51] is a faster language model based on BERT, obtained by distilling a large, pre-trained BERT model. As generalized by Hinton et al. [104], knowledge distillation, or student-teacher learning, is a compression technique in which a small model is trained to reproduce the behavior of a larger model. The smaller DistilBERT model reduces the number of layers and parameters by a factor of two while retaining 97% of BERT's performance in the presented evaluation.

7.3.1.2 SqueezeBERT

SqueezeBERT [105] is a smaller variant of the BERT model which aims to speed up its inference, specifically to enable using them on mobile devices. The model architecture is similar to BERT, but the authors propose replacing the pointwise fully-connected layers with grouped convolutions, which have been successfully used in neural networks designed for computer vision tasks to boost performance. On mobile devices, the benchmarks show a $4.3\times$ speedup over the base BERT model.

7.3.1.3 Comparing Inference Time

In Table 7.2, we show the results of a benchmark comparing the two alternative model architectures to the base BERT model. We ran the experiment using the pure PyTorch implementations from huggingface Transformers library on `F16s v2` Azure instances and averaged the inference latency for different sequence lengths.

DistilBERT shows a steady performance gain over the base BERT model on all evaluated sequence sizes, which is expected given that the model architecture remains the same, with only the number of layers and parameters cut in half.

On the other hand, the SqueezeBERT model gave us less consistent performance improvements. We could see a performance boost on longer sequences, but in cases with smaller sequences, which are more critical for our use in classifying individual sentences, it sometimes performed even worse than the original BERT model.

In subsequent experiments, we will be using the base BERT model, which is still the default choice for our call-to-action detector service. However, the infrastructure we have built in the call-to-action detection service allows us to easily swap the model if even faster inference time becomes a requirement in the future, and we verify that the smaller model offers sufficient accuracy on our specific classification task. DistilBERT would be the obvious choice for a less expensive model architecture from the performed benchmark results, and we benchmark both model architectures in the final comparison in Section 7.4.3.

7.3.2 ONNX Runtime

For our NLP models, we primarily use huggingface Transformer implementations through the PyTorch library, which uses a custom format for exporting and loading the models. These libraries are currently a popular choice for implementing state-of-the-art NLP models, but this might change in the upcoming years with newer, more advanced libraries or machine learning paradigms.

Open Neural Network Exchange (ONNX) [106] is an open-source format for AI models which is supported by most popular neural network libraries. Using ONNX, we can create a unified, portable representation of our models and optimize their inference through accelerators such as ONNX Runtime [107].

ONNX Runtime is a machine learning accelerator developed by Microsoft for efficient model inference in their Azure Machine Learning service. It has helped us significantly speed up the call-to-action service through its graph optimizations and quantization features.

7.3.2.1 Graph Optimizations

ONNX Runtime performs two types of graph optimizations, which are covered in greater detail in the documentation [108].

Basic Graph Optimizations Graph transformations which only remove redundant nodes or layers, but do not change the semantics of the computations.

Extended Graph Optimizations Additional transformations which perform more drastic changes to the graph structure, mostly through a complex fusion of its nodes or layers. These changes generally result in approximations of the original computations, which might lead to slight variations in the output layers'

Length	Average Latency [ms]			Throughput [QPS]		
	Default	Optimized	Quantized	Default	Optimized	Quantized
8	6.67	6.23	2.81	150.00	160.59	355.56
32	10.09	9.08	7.13	99.08	110.07	140.29
64	15.67	14.20	12.40	63.81	70.42	80.64

Table 7.3: Comparing inference time of the `bert-base-uncased` model converted to the ONNX format and run through the ONNX Runtime accelerator with different optimization settings on a `F16s v2` Azure instance. Latency is averaged over 100 samples.

values. However, these transformations have been shown to have a negligible effect on the resulting models' accuracy and can provide a significant performance boost.

Layout Optimizations Optimizations which change the data layout to achieve higher computation performance.

The authors have also included additional, BERT specific graph optimizations [109], in the ONNX Runtime Tools library, which we use to convert our PyTorch model into an ONNX representation.

7.3.2.2 Quantization

Model weights of PyTorch models are by default represented using the single-precision floating-point format (`float32`), and this precise representation is important for model training. However, after the training is finished, we can convert the model to use a more compact integer (`int8`) representation without a noticeable impact on the model performance. This process is called quantization, and it significantly reduces the size of the model and massively improves the inference time on CPUs. This topic is covered in greater detail in the PyTorch documentation [110, 111].

7.3.2.3 Benchmarking ONNX Runtime Optimizations

Next, we perform benchmarks to show the impact of the described ONNX Runtime optimizations. In Table 7.3, we compare the inference latency and throughput of the base BERT model, which we convert to the ONNX format (*Default* column), optimize its compute graph (*Optimized* column) and finally quantize the optimized model (*Quantized* column). We perform the model inference through ONNX Runtime on `F16s v2` Azure instances, using random sequence inputs of various sizes.

While the graph optimizations show a steady performance boost of around 10% for all sequence sizes, the largest speedup comes from quantization, especially on shorter sequences where we see a $2.3\times$ increase in throughput over the base BERT ONNX model.

Length	**Average Latency** [ms]		
	PyTorch BERT	ONNX BERT	ONNX DistilBERT
8	19.88	2.81	1.49
32	51.06	7.13	4.36
64	65.44	12.40	7.95
Length	**Throughput** [QPS]		
	PyTorch BERT	ONNX BERT	ONNX DistilBERT
8	50.30	355.56	671.01
32	19.58	140.29	229.23
64	15.28	80.64	125.73

Table 7.4: Final comparison between the original Transformers BERT PyTorch implementation (`bert-base-uncased`) and our optimized and quantized BERT (`bert-base-uncased`) and DistilBERT (`distilbert-base-uncased`) models using ONNX Runtime on `F16s v2` Azure instances. Latency is averaged over 100 samples.

7.3.3 Model Benchmark

Finally, we compare the optimized ONNX Runtime models with the original PyTorch implementations. From the results in Table 7.4, we can see that the combination of the presented optimization techniques yields a massive speedup over the original implementations, with the optimized DistilBERT model being over $10\times$ faster for shorter sequences (sizes 8 and 32), which should be similar to sentences found in the written e-mail communication.

7.4 Service Communication

In the previous sections, we have demonstrated how model inference of our call-to-action detector can be significantly optimized by choosing suitable instance types, model architectures, and runtime accelerator optimizations. However, separating the detector into an isolated service with its separate infrastructure requires communication between the service and the detector engine core, which adds additional overhead. In this section, we present two approaches for connecting the call-to-action detector service with the engine, discuss their advantages, and compare benchmark their performance in combination with the proposed service optimizations.

7.4.1 REST API Service

Our first choice was to expose the service through a REST API, as this was the approach used to connect to our engine API (Chapter 5), and we already had experience with this setup. REST APIs are flexible and easy to consume, regardless of the used programming language. However, this is not a strong advantage because the

service is a component in a larger system exposed through a REST endpoint and will not be used outside of this engine soon. Communication over the HTTP protocol does not provide us with any fault tolerance and requires additional conversions of our internal data classes into the JSON format. The engine will communicate with this service through synchronous web requests, and without adding workarounds to enable asynchronous waiting, the worker process of the engine core will have to idle and wait for the response. Despite these drawbacks, it was a simple starting point for our service, which would allow us to benchmark the performance in a more realistic scenario with real data.

Instead of using Flask, we decided to experiment with the FastAPI [112] Python web framework, which embraces asynchronous processing, and which, according to TechEmpower Web Framework Benchmarks, is currently one of the fastest web frameworks for Python [113]. Otherwise, the infrastructure and deployment process is identical to the classification engine API, with a dockerized container encapsulating the code and models running inside of a Packer virtual machine image (Figure 7.1). The service accepts a list of sentences obtained from running sentence segmentation on text extracted from the e-mail body (Section 4.2), runs our optimized and quantized call-to-action model through ONNX Runtime (Section 7.3.2), and returns a JSON encoded list of detections consumed by the engine core. To reduce the latency of the web requests between the services, we start both services inside of a proximity placement group (Section 6.2.4). The service can be scaled further by adding more instances with a unified access endpoint supplied through a load balancer.

While the implemented REST service does not directly utilize the asynchronous capabilities of FastAPI, we have already experimented with adding concurrency to our processing pipeline through asyncio [114] libraries, and the development of this service helped us get familiar with the approach. Although we have a working proof-of-concept of a fully asynchronous classification pipeline which would help us avoid the synchronous web request blocking, it requires fundamental changes to the system architecture and careful tuning in the context of the multi-process worker setup. Therefore, we have not yet decided on this transition and instead developed a solution using a message queue, which adds resilience and provides partial asynchronicity.

7.4.2 Celery Task Queue

Celery [115] is a distributed task queue system that we used to develop a different communication scheme that addresses several shortcomings of the REST API approach mentioned in the previous section. The task queue uses the concept of isolated tasks, which are asynchronously processed using a pool of worker processes that can be independently run on multiple computer instances. When the tasks are created, they are sent to the message broker and stored together with their input data. Then, a worker process independently pulls a task from the queue, processes it, and stores the result into a result backend where it can be later retrieved.

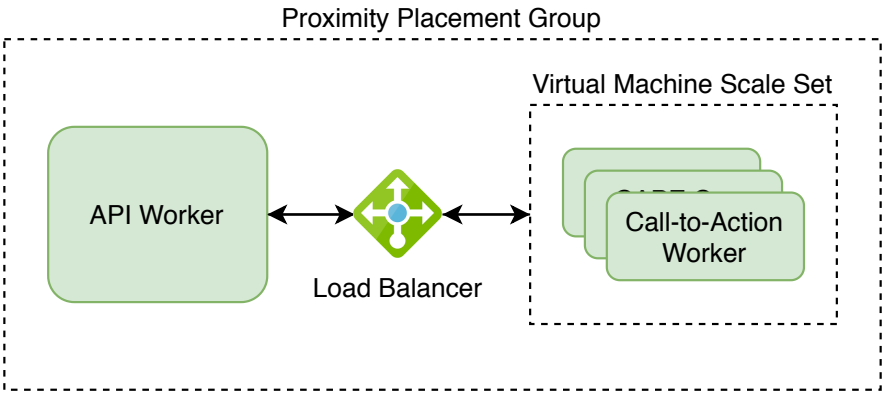

Figure 7.1: Overview of the REST API communication schema.

To adapt this processing model to our call-to-action service, we implement a single task that processes a list of segmented sentences, encodes and tokenizes them for the BERT model, and runs model inference through ONNX Runtime. The call-to-action worker processes connect to the shared message broker, listen for the tasks created by the engine core when it analyzes an e-mail, and compute and store the detections into the result backend. We use the Redis key-value store [116] as both the message broker and result backend (Figure 7.2).

The main advantage of this approach is that after storing the task in the queue, we can save a Future object, which represents the result of an asynchronous computation, and continue with other processing steps until we need to use the result. In our case, we require the detection results at the very end of our processing pipeline to aggregate the detection into a phishing verdict – this means that we can run the call-to-action detector and store the Future object, then process the e-mail using all other synchronous detectors, and finally retrieve the call-to-action results from the queue. At this point, the asynchronous service is likely done processing that e-mail.

The task queue also brings resilience and fault tolerance – in case our worker processes crash unexpectedly, the engine core can still process e-mails and store the tasks in the message queue, as they will be automatically processed later when the service is again fully available.

However, compared to the REST API approach, it adds communication steps between worker instances and the message broker, as the engine core does not directly query the call-to-action service – the worker processes pull the task from the queue, process it, and store the detections in the result backend from which the engine retrieves it. This can result in slower response times for very small e-mails where the model inference time is negligible and most time is spent on communication.

While running benchmarks of the completed call-to-action service on our Enron evaluation subset, we noticed several outliers taking up a large portion of the total processing time. While we could remove these large e-mail files from the evaluation set, we will not have this option in production on real data where we could

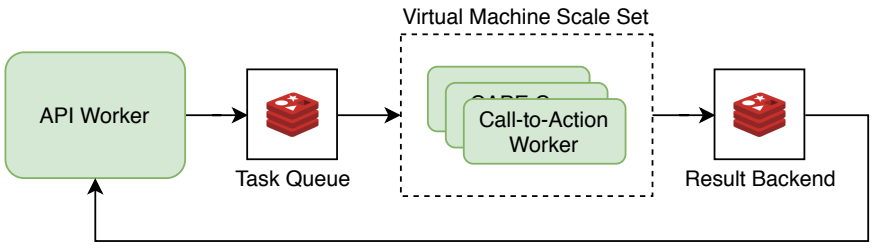

Figure 7.2: Overview of the Celery task queue communication schema.

encounter e-mails with similar properties, and we wanted our service to skip processing these e-mails if inference was taking too long to avoid flooding our system. We found an elegant solution to this problem through setting a time limit for the Celery tasks – when the task processing was taking too long and surpassed a set threshold, Celery would stop the worker process and return a timeout error. This way, we would potentially lose some detections from the call-to-action detector, but our service would return verdicts at a more predictable rate without its message queue becoming clogged and slowing down the entire engine.

7.4.3 Service Benchmark

Finally, we benchmark the performance of our completed call-to-action detector service using ONNX Runtime optimizations, and the two presented approaches for service communication. For this evaluation, we use a subset of the public Enron e-mail dataset with 1000 randomly selected e-mails.

To isolate the performance of the call-to-action service, we run the benchmark on preprocessed e-mails. Therefore, the measured metrics will not be influenced by e-mail parsing or tokenization and should capture mainly the model inference and service communication costs.

We measure the performance of three solutions:

Local The call-to-action detector runs inside the main classification pipeline in the engine core. This is the original solution with no added communication, but running the default PyTorch model implementations without the added optimizations. Unlike in our production setup, we run the engine on the same compute instances used for the separated services to give a fair performance comparison.

FastAPI The call-to-action service connected through the REST endpoint (Section 7.4.1).

Celery The call-to-action service connected through the Celery task queue (Section 7.4.2). We present two scenarios with and without a task time limit, which we set to 1 second.

	Local	FastAPI BERT	FastAPI DistilBERT
Total time [s]	302.62	110.69	72.28
Average latency [s]	0.303	0.111	0.072
Max latency [s]	12.70	4.075	2.19

Table 7.5: Comparison of Local and FastAPI call-to-action detectors on a 1000 e-mail subset of the Enron dataset on `F16s v2` Azure instances.

	Local	Celery BERT		Celery DistilBERT	
		default	time limit = 1	default	time limit = 1
Total time [s]	302.62	91.44	82.51	57.21	56.79
Average latency [s]	0.303	0.091	0.083	0.057	0.057
Max latency [s]	12.70	1.72	1.01	2.90	1.00

Table 7.6: Comparison of Local and Celery call-to-action detectors on a 1000 e-mail subset of the Enron dataset on `F16s v2` Azure instances.

For our optimized services, we show results with both the original BERT model and the faster but potentially less accurate DistilBERT model. The benchmark was performed using the `F16s v2` CPU compute instances, both for the engine core and the call-to-action services. For the task queue service, we use a `Basic C0` managed Redis instance provisioned through Azure Cache for Redis as the message broker and result backend.

Results of the FastAPI call-to-action service are presented in Table 7.6. When using the optimized runtime and model, we see a significant increase in service throughput for both BERT ($2.7\times$ speedup) and DistilBERT ($4.2\times$ speedup) models compared to the original local detector.

In Table 7.6, we can see the benchmark results of the Celery task queue service. Compared to the FastAPI service, we see an additional increase in throughput from using the task queue as a means of communication, resulting in a $3.3\times$ speedup with the BERT model and $5.3\times$ speedup with the DistilBERT model.

We can see that while setting the time limit to 1 second reduced the total inference time by another 10% with the BERT model (where 13 out of 1000 e-mails were skipped), we did not see a significant improvement with the DistilBERT model. This is because when we cancel long tasks (9 out of 1000 e-mails for DistilBERT), we get an additional overhead from managing the worker processes. When the worker process gets killed, we need to start a new worker process and load the model – this takes additional time, and in our DistilBERT benchmark, it resulted in an almost equivalent total processing time. However, setting the time limit is still a reasonable default setting that helps us avoid spending too much processing time on a single e-mail until we add a more sophisticated filtering mechanism.

Overall, the Celery task queue service is our preferred solution, as it offers the best throughput and robustness. In production, we will also be running the full detection pipeline, where the asynchronous processing enabled by the task queue will make an even bigger impact by effectively running the service in parallel with other detectors.

Chapter 8

Conclusion

8.1 Summary

In this thesis, we first explored the phishing landscape, investigated the common signals and evasion techniques of sophisticated phishing attacks, and introduced a system for automatic detection of phishing e-mails.

As a crucial component of this system, we introduced a detector for call-to-action based on natural language processing. We summarized the recent advancements in this field and showed incremental improvements to the detector by utilizing more advanced classifiers and feature representations and benchmarked the predictive performance of these solutions.

We presented an architecture for a phishing classification pipeline using an extensible ensemble of detectors. We showed the individual stages of the classification process, from e-mail preprocessing and text extraction to detection aggregation, and implemented this pipeline using the Python programming language.

Next, we showed how this system could be efficiently exposed through a REST API using Python libraries and packaged into a deployable service. Then, we designed a scalable infrastructure setup for deploying the classification engine in Microsoft Azure.

Finally, we focused on optimizing the final iteration of the call-to-action detector using state-of-the-art Transformer models. We showed how selecting suitable hardware infrastructure, utilization of less expensive model architectures, and model optimization and quantization provided by the ONNX Runtime accelerator could significantly speed up inference of this demanding model on consumer CPU instances. We benchmarked the final call-to-action service using two service communication solutions and demonstrated how the proposed optimizations yield a massive $5\times$ throughput increase compared to the original approach.

8.2 Future Improvements

The development of our engine is an ongoing project, and we will continue optimizing and extending it with additional detectors, such as the planned identity and relationship models mentioned in Section 4.3. At the time of writing the thesis, the system was not yet fully running in production. Therefore, it was not possible to evaluate the efficacy of the complete phishing system. With the availability of real customer data, this will be a crucial next step.

As the analyzed e-mail volume increases, we expect to eventually shift to GPU processing for the neural network model services, especially with the planned addition of the affordable Tesla T4 GPUs [117] to Azure. This should be a trivial task because of the flexible architecture we have developed for the call-to-action detector service.

With a larger data volume and GPU processing, we could further optimize the call-to-action service by implementing logic for efficient batching of data from multiple requests. This could be implemented in a non-blocking fashion using our proposed message queue communication solution.

We have also experimented with using Rust [118] to speed up other performance bottlenecks of our engine. This would rid us of the performance limitations of Python in the critical areas of the engine while still being able to connect the code through Python bindings elegantly.

Finally, an exciting research idea is to bring multi-language support to our language-specific detectors, either through adapting multi-language models or using increasingly available machine translation models (i.e., recently published MLM-100 model by Facebook [119]).

Bibliography

[1] Don't get caught by Spear Phishing. https://www.reallysimplesystems.com/blog/spear-phishing-rise/, (Accessed on 12/05/2020).

[2] Czech Post Scam. https://www.ceskaposta.cz/-/ceska-posta-varuje-pred-podvodnymi-e-maily-vydavajicimi-se-za-ceskou-pos-1, (Accessed on 11/27/2020).

[3] How scammers abuse Google Search's open redirect feature. https://nakedsecurity.sophos.com/2020/05/15/how-scammers-abuse-google-searchs-open-redirect-feature/, (Accessed on 11/27/2020).

[4] Perceptrons and Multi-layer Perceptrons: The Artificial Neuron. https://missinglink.ai/guides/neural-network-concepts/perceptrons-and-multi-layer-perceptrons-the-artificial-neuron-at-the-core-of-deep-learning/, (Accessed on 11/28/2020).

[5] NLP and RNNs Representation. https://colah.github.io/posts/2014-07-NLP-RNNs-Representations/, (Accessed on 12/01/2020).

[6] Understanding LSTMs. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, (Accessed on 12/01/2020).

[7] Vaswani, A.; Shazeer, N.; Parmar, N.; et al. Attention Is All You Need. 2017, 1706.03762.

[8] Creating Word Embeddings. https://towardsdatascience.com/creating-word-embeddings-coding-the-word2vec-algorithm-in-python-using-deep-learning-b337d0ba17a8, (Accessed on 12/01/2020).

[9] IBM X-Force Threat Intelligence Index 2020. https://www.ibm.com/downloads/cas/DEDOLR3W, (Accessed on 12/03/2020).

[10] FBI 2019 Internet Crime Report. https://pdf.ic3.gov/2019_IC3Report.pdf, (Accessed on 12/03/2020).

[11] Committee on National Security Systems Glossary. https://rmf.org/wp-content/uploads/2017/10/CNSSI-4009.pdf, (Accessed on 12/03/2020).

[12] Merriam-Webster: Phishing. https://www.merriam-webster.com/dictionary/phishing, (Accessed on 12/03/2020).

[13] Cofense Phishing Report 2019. https://cofense.com/phishing-report-2019/, (Accessed on 12/03/2020).

[14] Kaspersky: What Is Spearphishing? https://www.kaspersky.com/resource-center/definitions/spear-phishing, (Accessed on 12/03/2020).

[15] Unicode Utilities: Confusables. https://util.unicode.org/UnicodeJsps/confusables.jsp, (Accessed on 12/31/2020).

[16] Visual Spoofing. https://websec.github.io/unicode-security-guide/visual-spoofing/, (Accessed on 12/31/2020).

[17] Kucherawy, M.; Crocker, D.; Hansen, T. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376, Sept. 2011, doi:10.17487/RFC6376. Available from: https://rfc-editor.org/rfc/rfc6376.txt

[18] Kucherawy, M.; Zwicky, E. Domain-based Message Authentication, Reporting, and Conformance (DMARC). RFC 7489, Mar. 2015, doi:10.17487/RFC7489. Available from: https://rfc-editor.org/rfc/rfc7489.txt

[19] Kitterman, S. Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1. RFC 7208, Apr. 2014, doi:10.17487/RFC7208. Available from: https://rfc-editor.org/rfc/rfc7208.txt

[20] Chen, J.; Paxson, V.; Jiang, J. Composition Kills: A Case Study of Email Sender Authentication. In *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, ISBN 978-1-939133-17-5, pp. 2183–2199. Available from: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-jianjun

[21] Shue, C.; Kalafut, A.; Gupta, M. Exploitable Redirects on the Web: Identification, Prevalence, and Defense. 01 2008.

[22] Kintis, P.; Miramirkhani, N.; Lever, C.; et al. Hiding in Plain Sight: A Longitudinal Study of Combosquatting Abuse. *CoRR*, volume abs/1708.08519, 2017, 1708.08519. Available from: http://arxiv.org/abs/1708.08519

[23] Szurdi, J.; Kocso, B.; Cseh, G.; et al. The Long "Taile" of Typosquatting Domain Names. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, Aug. 2014, ISBN 978-1-931971-15-7, pp. 191–206. Available from: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/szurdi

[24] Seymour, J.; Tully, P. Weaponizing data science for social engineering: Automated E2E spear phishing on Twitter.

[25] Radford, A.; Wu, J.; Child, R.; et al. Language Models are Unsupervised Multitask Learners. 2018. Available from: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

[26] Brown, T. B.; Mann, B.; Ryder, N.; et al. Language Models are Few-Shot Learners. 2020, 2005.14165.

[27] Brundage, M.; Avin, S.; Clark, J.; et al. The Malicious Use of Artificial Intelligence: Forecasting, Prevention, and Mitigation. *CoRR*, volume abs/1802.07228, 2018, 1802.07228. Available from: http://arxiv.org/abs/1802.07228

[28] Giaretta, A.; Dragoni, N. Community Targeted Spam: A Middle Ground Between General Spam and Spear Phishing. *CoRR*, volume abs/1708.07342, 2017, 1708.07342. Available from: http://arxiv.org/abs/1708.07342

[29] Microsoft 365. https://www.microsoft.com/en-us/microsoft-365, (Accessed on 12/05/2020).

[30] Microsoft Azure. https://azure.microsoft.com/, (Accessed on 12/05/2020).

[31] Zhang, H. The Optimality of Naive Bayes. In *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004)*, edited by V. Barr; Z. Markov, AAAI Press, 2004.

[32] Papers with code: Language modelling on Wikitext-103. https://paperswithcode.com/sota/language-modelling-on-wikitext-103, (Accessed on 11/20/2020).

[33] Deep Learning Book. https://www.deeplearningbook.org, (Accessed on 11/20/2020).

[34] Srivastava, N.; Hinton, G.; Krizhevsky, A.; et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, volume 15, no. 56, 2014: pp. 1929–1958. Available from: http://jmlr.org/papers/v15/srivastava14a.html

[35] Rumelhart, D. E.; Hinton, G. E.; Williams, R. J. *Learning Internal Representations by Error Propagation*. Cambridge, MA, USA: MIT Press, 1986, ISBN 026268053X, p. 318–362.

[36] Kingma, D. P.; Ba, J. Adam: A Method for Stochastic Optimization. 2017, 1412.6980.

[37] Werbos, P. J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, volume 78, no. 10, 1990: pp. 1550–1560, doi:10.1109/5.58337.

[38] Ba, J.; Mnih, V.; Kavukcuoglu, K. Multiple Object Recognition with Visual Attention. 2015, 1412.7755.

[39] Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural computation*, volume 9, no. 8, 1997: pp. 1735–1780.

[40] Cho, K.; van Merrienboer, B.; Gülçehre, Ç.; et al. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR*, volume abs/1406.1078, 2014, 1406.1078. Available from: http://arxiv.org/abs/1406.1078

[41] Sutskever, I.; Vinyals, O.; Le, Q. V. Sequence to Sequence Learning with Neural Networks. *CoRR*, volume abs/1409.3215, 2014, 1409.3215. Available from: http://arxiv.org/abs/1409.3215

[42] Bahdanau, D.; Cho, K.; Bengio, Y. Neural Machine Translation by Jointly Learning to Align and Translate. 2016, 1409.0473.

[43] Huggingface Transformers: Models. https://github.com/huggingface/transformers/#models-architectures, (Accessed on 11/23/2020).

[44] The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. http://jalammar.github.io/illustrated-transformer/, (Accessed on 01/03/2021).

[45] Luong, M.; Pham, H.; Manning, C. D. Effective Approaches to Attention-based Neural Machine Translation. *CoRR*, volume abs/1508.04025, 2015, 1508.04025. Available from: http://arxiv.org/abs/1508.04025

[46] The Transformer Family. https://lilianweng.github.io/lil-log/2020/04/07/the-transformer-family.html, (Accessed on 01/03/2021).

[47] Devlin, J.; Chang, M.-W.; Lee, K.; et al. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 2019, 1810.04805.

[48] Liu, Y.; Ott, M.; Goyal, N.; et al. RoBERTa: A Robustly Optimized BERT Pretraining Approach. 2019, 1907.11692.

[49] Yang, Z.; Dai, Z.; Yang, Y.; et al. XLNet: Generalized Autoregressive Pretraining for Language Understanding. 2020, 1906.08237.

[50] Lan, Z.; Chen, M.; Goodman, S.; et al. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. 2020, 1909.11942.

[51] Sanh, V.; Debut, L.; Chaumond, J.; et al. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. 2020, 1910.01108.

[52] Mikolov, T.; Sutskever, I.; Chen, K.; et al. Distributed Representations of Words and Phrases and their Compositionality. 2013, 1310.4546.

[53] Mikolov, T.; Chen, K.; Corrado, G.; et al. Efficient Estimation of Word Representations in Vector Space. 2013, 1301.3781.

[54] Pennington, J.; Socher, R.; Manning, C. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543, doi:10.3115/v1/D14-1162. Available from: https://www.aclweb.org/anthology/D14-1162

[55] Peters, M. E.; Neumann, M.; Iyyer, M.; et al. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

[56] Rong, X. word2vec Parameter Learning Explained. 2016, 1411.2738.

[57] BERT-as-service. https://github.com/hanxiao/bert-as-service, (Accessed on 11/23/2020).

[58] Reimers, N.; Gurevych, I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. 2019, 1908.10084.

[59] Conneau, A.; Kiela, D. SentEval: An Evaluation Toolkit for Universal Sentence Representations. 2018, 1803.05449.

[60] Sentence-BERT Github. https://github.com/UKPLab/sentence-transformers, (Accessed on 11/23/2020).

[61] Sentence-BERT Performance Benchmark. https://github.com/UKPLab/sentence-transformers#performance, (Accessed on 11/23/2020).

[62] Klimt, B.; Yang, Y. The Enron Corpus: A New Dataset for Email Classification Research. In *Proceedings of the 15th European Conference on Machine Learning*, ECML'04, Berlin, Heidelberg: Springer-Verlag, 2004, ISBN 3540231056, p. 217–226, doi:10.1007/978-3-540-30115-8_22. Available from: https://doi.org/10.1007/978-3-540-30115-8_22

[63] Nazario, J. Nazario Phishing Dataset. http://monkey.org/ jose/wiki/doku.php?id=phishingcorpus – no longer available, 2006.

[64] Doccano: Open-source annotation tool. https://github.com/doccano/doccano, (Accessed on 11/20/2020).

[65] Docker. https://www.docker.com/, (Accessed on 12/17/2020).

[66] Conneau, A.; Kruszewski, G.; Lample, G.; et al. What you can cram into a single vector: Probing sentence embeddings for linguistic properties. 2018, 1805.01070.

[67] Wang, A.; Singh, A.; Michael, J.; et al. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. 2019, 1804.07461.

[68] Feurer, M.; Klein, A.; Eggensperger, K.; et al. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems*, volume 28, edited by C. Cortes; N. Lawrence; D. Lee; M. Sugiyama; R. Garnett, Curran Associates, Inc., 2015, pp. 2962–2970. Available from: https://proceedings.neurips.cc/paper/2015/file/11d0e6287202fced83f79975ec59a3a6-Paper.pdf

[69] Wolf, T.; Debut, L.; Sanh, V.; et al. HuggingFace's Transformers: State-of-the-art Natural Language Processing. 2020, 1910.03771.

[70] Paszke, A.; Gross, S.; Massa, F.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. 2019, 1912.01703.

[71] Brabec, J.; Machlica, L. Bad practices in evaluation methodology relevant to class-imbalanced problems. 2018, 1812.01388.

[72] Brabec, J.; Komárek, T.; Franc, V.; et al. On Model Evaluation under Non-constant Class Imbalance. 2020, 2001.05571.

[73] Crocker, D. RFC0822: Standard for the Format of ARPA Internet Text Messages. 1982.

[74] Resnick, P. Internet Message Format. RFC 2822, Apr. 2001, doi:10.17487/RFC2822. Available from: https://rfc-editor.org/rfc/rfc2822.txt

[75] EML File Example. https://www.phpclasses.org/browse/file/14672.html, (Accessed on 12/10/2020).

[76] Python email module. https://docs.python.org/3/library/email.parser.html, (Accessed on 12/12/2020).

[77] BeautifulSoup. https://www.crummy.com/software/BeautifulSoup/bs4/doc/, (Accessed on 12/12/2020).

[78] lxml e-mail parser. https://lxml.de/, (Accessed on 12/12/2020).

[79] Talon library. https://github.com/mailgun/talon, (Accessed on 12/12/2020).

[80] Carvalho, V.; Cohen, W. Learning to Extract Signature and Reply Lines from Email. 01 2004.

[81] SpaCy: Industrial-Strength Natural Language Processing. https://spacy.io/, (Accessed on 12/13/2020).

[82] How SpaCy Tokenizer works. https://spacy.io/usage/linguistic-features#how-tokenizer-works, (Accessed on 12/13/2020).

[83] SpaCy Models: Architecture. https://spacy.io/models#architecture, (Accessed on 12/13/2020).

[84] Sadvilkar, N.; Neumann, M. PySBD: Pragmatic Sentence Boundary Disambiguation. 2020, 2010.09657.

[85] Wang, W.; Zheng, V. W.; Yu, H.; et al. A Survey of Zero-Shot Learning: Settings, Methods, and Applications. *ACM Trans. Intell. Syst. Technol.*, volume 10, no. 2, Jan. 2019, ISSN 2157-6904, doi:10.1145/3293318. Available from: https://doi.org/10.1145/3293318

[86] Flask framework. https://palletsprojects.com/p/flask/, (Accessed on 12/15/2020).

[87] Gunicorn WSGI Server. https://gunicorn.org/, (Accessed on 12/15/2020).

[88] PEP-3333: Python Web Server Gateway Interface. https://www.python.org/dev/peps/pep-3333/, (Accessed on 12/17/2020).

[89] Python: Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock, (Accessed on 12/17/2020).

[90] fork manual page. https://man7.org/linux/man-pages/man2/fork.2.html, (Accessed on 12/17/2020).

[91] NGINX Web Server. https://www.nginx.com/, (Accessed on 12/15/2020).

[92] Datadog: Cloud Monitoring as a Service. https://www.datadoghq.com/, (Accessed on 12/19/2020).

[93] ELK Stack: Elasticsearch, Logstash, Kibana | Elastic. https://www.elastic.co/what-is/elk-stack, (Accessed on 01/04/2021).

[94] Docker Compose. https://docs.docker.com/compose/, (Accessed on 12/17/2020).

[95] Packer. https://www.packer.io/, (Accessed on 12/19/2020).

[96] Terraform. https://www.terraform.io/, (Accessed on 12/19/2020).

[97] Introducing Proximity Placement Groups. https://azure.microsoft.com/en-us/blog/introducing-proximity-placement-groups/, (Accessed on 12/19/2020).

[98] Azure VM sizes - Compute optimized - Azure Virtual Machines | Microsoft Docs. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-compute, (Accessed on 12/20/2020).

[99] Azure VM sizes - GPU - Azure Virtual Machines | Microsoft Docs. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu, (Accessed on 12/20/2020).

[100] NVIDIA Clocks World's Fastest BERT Training Time and Largest Transformer Based Model, Paving Path For Advanced Conversational AI | NVIDIA Developer Blog. https://developer.nvidia.com/blog/training-bert-with-gpus/, (Accessed on 12/20/2020).

[101] Intel® Xeon® Scalable Processors. https://www.intel.com/content/www/us/en/products/processors/xeon/scalable.html, (Accessed on 12/28/2020).

[102] bert-base-uncased · Hugging Face. https://huggingface.co/bert-base-uncased, (Accessed on 12/20/2020).

[103] Introduction to Intel® Deep Learning Boost on Second Generation... https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-deep-learning-boost-on-second-generation-intel-xeon-scalable.html, (Accessed on 12/20/2020).

[104] Hinton, G.; Vinyals, O.; Dean, J. Distilling the Knowledge in a Neural Network. 2015, 1503.02531.

[105] Iandola, F. N.; Shaw, A. E.; Krishna, R.; et al. SqueezeBERT: What can computer vision teach NLP about efficient neural networks? 2020, 2006.11316.

[106] ONNX | Home. https://onnx.ai/, (Accessed on 12/20/2020).

[107] ONNX Runtime | Home. https://www.onnxruntime.ai/, (Accessed on 12/20/2020).

[108] Graph optimizations - onnxruntime. https://www.onnxruntime.ai/docs/resources/graph-optimizations.html, (Accessed on 12/22/2020).

[109] Microsoft open sources breakthrough optimizations for transformer inference on GPU and CPU. https://cloudblogs.microsoft.com/opensource/2020/01/21/microsoft-onnx-open-source-optimizations-transformer-inference-gpu-cpu/, (Accessed on 12/20/2020).

[110] Introduction to Quantization on PyTorch | PyTorch. https://pytorch.org/blog/introduction-to-quantization-on-pytorch/, (Accessed on 12/22/2020).

[111] Quantization — PyTorch 1.7.0 documentation. https://pytorch.org/docs/stable/quantization.html, (Accessed on 12/22/2020).

[112] FastAPI. https://fastapi.tiangolo.com/, (Accessed on 12/20/2020).

[113] Round 18 results - TechEmpower Framework Benchmarks. https://www.techempower.com/benchmarks/#section=data-r18&hw=ph&test=composite&l=zijzen-1r, (Accessed on 12/27/2020).

[114] asyncio — Asynchronous I/O — Python 3.9.1 documentation. https://docs.python.org/3/library/asyncio.html, (Accessed on 12/27/2020).

[115] Celery - Distributed Task Queue — Celery 5.0.5 documentation. https://docs.celeryproject.org/en/stable/, (Accessed on 12/20/2020).

[116] Redis. https://redis.io/, (Accessed on 12/27/2020).

[117] NVIDIA T4 Tensor Core GPU for AI Inference | NVIDIA Data Center. https://www.nvidia.com/en-us/data-center/tesla-t4/, (Accessed on 12/29/2020).

[118] Rust Programming Language. https://www.rust-lang.org/, (Accessed on 12/29/2020).

[119] Introducing the First AI Model That Translates 100 Languages Without Relying on English - About Facebook. https://about.fb.com/news/2020/10/first-multilingual-machine-translation-model/, (Accessed on 12/29/2020).

Appendix A

Contents of enclosed CD

/ ..... the root directory
- `thesis.pdf` ..... thesis text in PDF format
- `thesis` ..... directory of LaTeX source codes