Geautomatiseerde documentatie generatie met behulp van Large Language Modellen: Het genereren van duidelijke overzichten en informatieve beschrijvingen voor Python projecten.

Max Milan.

Scriptie voorgedragen tot het bekomen van de graad van
Professionele bachelor in de toegepaste informatica

Promotor: Dhr. G. Bosteels
Co-promotor: Dhr. A. Pannemans
Academiejaar: 2023–2024
Eerste examenperiode

Departement IT en Digitale Innovatie .

HO GENT

Woord vooraf

[1-2]

Samenvatting

[1-4]

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

List of Listings

1

Inleiding

1.1. Probleemstelling

Projecten worden vaak niet goed gedocumenteerd, wat kan leiden tot problemen in de toekomst. Wanneer een andere persoon de code van een ongedocumenteerd project wil gebruiken moet de code volledig gelezen worden voordat er begrepen wordt wat de code doet. Dit is een tijdrovend proces en kan voorkomen worden door goede documentatie. Wanneer de code jaren later aangepast moet worden is het ook handig om goede documentatie te hebben, zodat de persoon weet waar er aanpassingen moeten gebeuren. De skills en know-how van een project kunnen verloren gaan wanneer er geen documentatie is. Deze dienen juist gedeeld te worden met anderen zodat er geen dubbel werk gedaan moet worden. Het is dus belangrijk dat er aan documentatie gedaan wordt en dat deze up-to-date blijft.

Het documenteren van een project is iets wat veel tijd kost en wat meestal geen aandacht krijgt. Een tool die dit proces kan versnellen / automatiseren zou een grote meerwaarde zijn. De tool bestaat uit een geautomatiseerde documentatie LLM die de project code analyseert en samenvat in een document. Dit geeft de lezers de mogelijkheid om zich in te lezen in het project en erna zelf aanpassingen te maken of stukken code te gebruiken voor een ander project.

Python is een populaire programmeertaal die gebruikt wordt in de IT wereld. Volgens TIOBE (2024) een website die zoekpagina's afgaat en de populariteit van programmeertalen op basis van het aantal hits bepaalt, staat Python op de eerste plaats. Het is dus interessant om een tool te maken die Pythonprojecten kan documenteren.

1.2. Onderzoeksvraag

Hoe kan geautomatiseerde documentatiegeneratie met behulp van Large Language Modellen (LLM) effectief worden toegepast voor Pythonprojecten waar de code niet gedocumenteerd is?

- Wat is documentatie?
- Wat zijn de huidige documentatie tools?
- Wat is er nodig om de code van een project te documenteren?
- Wat is er nodig om de code van een bestand te documenteren?
- Waarom documentatie met behulp van een LLM?

1.3. Onderzoeksdoelstelling

Het eindresultaat van deze bachelorproef is een Proof of Concept (PoC) van een geautomatiseerde tool die de code van een Pythonproject analyseert en er documentatie van genereert. De gegenereerde documentatie laat het toe het project te begrijpen.

1.4. Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:
In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie.
In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.
In Hoofdstuk
In Hoofdstuk 7, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2

Stand van zaken

In dit hoofdstuk wordt de literatuurstudie besproken. Door deze literatuurstudie is het mogelijk om een beter inzicht te krijgen in de technologie en mogelijkheden voor de documentatie van Python projecten. Alsook hoe het toegepast kan worden met behulp van Large Language Modellen. Er zal nadruk worden gelegd op bestaande literatuur en onderzoeken die verbonden zijn met documentatie van Python-projecten. In dit onderdeel zullen verschillende hoofdstukken worden aangekaart. Als eerste zal er duidelijk gemaakt worden wat er juist verstaan wordt onder documentatie. Dan wordt er gekeken naar wat Large Language Modellen zijn, hoe deze werken en wat enkele bestaande modellen zijn. Vervolgens wordt er gekeken naar bestaande documentatie tools. Als laatste wordt er gekeken naar hoe Large Language Modellen gebruikt kunnen worden voor het genereren van documentatie.

2.1. Wat is documentatie?

Voor dat er dieper op het onderwerp wordt ingegaan is het belangrijk dat er een duidelijk beeld gevormdt wordt wat documentatie is. Waarom is documentatie belangrijk voor een project en wat wordt er begrepen onder documentatie?
Documentatie is het proces van het vastleggen van de werking van een project. Volgens Code Quality (2024) kan op verschillende manieren gebeuren. Er kan gekozen worden om de documentatie te schrijven in de vorm van een commentaar in de code, een docstrings, een API voor klassen of functies of in de vorm van een README.md bestand (Code Quality, 2024). Het doel van documentatie is om de werking van het project te beschrijven zodat andere programmeurs het project kunnen begrijpen en gebruiken. Zodat er geen tijd verloren gaat aan het lezen van de code en het begrijpen ervan.
Documentatie kan gemaakt worden voor verschillende doelgroepen. Het kan voor

interne of externe doeleinden zijn. Interne documentatie is voor documentatie binnen hetzelfde bedrijf. Dit gaat dan om het capteren van de process kennis die vergaard is binnen een project. Dit is informatie zoals een roadmap of product requirements. Deze documentatie gaat over het vastleggen van gedetaileerde uitleg over hoe iets werkt en hoe het onderhouden kan worden (swimm.io, 2024).
Externe documentatie is voor documentatie die gedeeld wordt met andere bedrijven of klanten. Dit gaat dan over de basis werking van de code van een project zodat andere programmeurs het kunnen gebruiken. Gebruiksaanwijzingen of handleidingen zijn ook een vorm van externe documentatie (swimm.io, 2024).
Voor deze bachelorproef wordt er gekeken naar het documenteren van een Python-project, in de vorm van commentaar in de code en het genereren van een samenvattend document van het gehele project. Omdat Python een populaire programmeertaal is volgens TIOBE (2024) en er veel projecten in deze taal geschreven worden is het interessant om te kijken hoe deze projecten gedocumenteerd kunnen worden. Ook kan er in de code bij functies aan typehinting gedaan worden, dit indiceert wat de datatypes van de input en output van een functie zijn (Bailey, 2024). Uit deze documentatie kan de werking van het project duidelijk worden en worden de relaties tussen de verschillende bestanden en functies weergegeven.
In het verdere verloop van deze bachelorproef wordt er gekeken naar hoe de documentatie van een Python project gegenereerd kan worden.

2.1.1. Bestand documentatie

Eerst dienen de bestanden van het project gedocumenteerd te worden. Dit gebeurt door de code van het bestand te analyseren en de docstrings van de verschillende functies en klassen te genereren.
Docstrings of documentatie strings, worden aan het begin van een functie of klasse geplaatst. Deze strings worden gebruikt om de functie of klasse te documenteren (GeeksforGeeks, 2023). Volgens GeeksforGeeks (2023) zijn docstrings vitaal in het overdragen van het doel en de werking van een functie of klasse.
Deze docstrings kunnen dan gebruikt worden om een samenvatting van het bestand te genereren.

2.1.2. Project documentatie

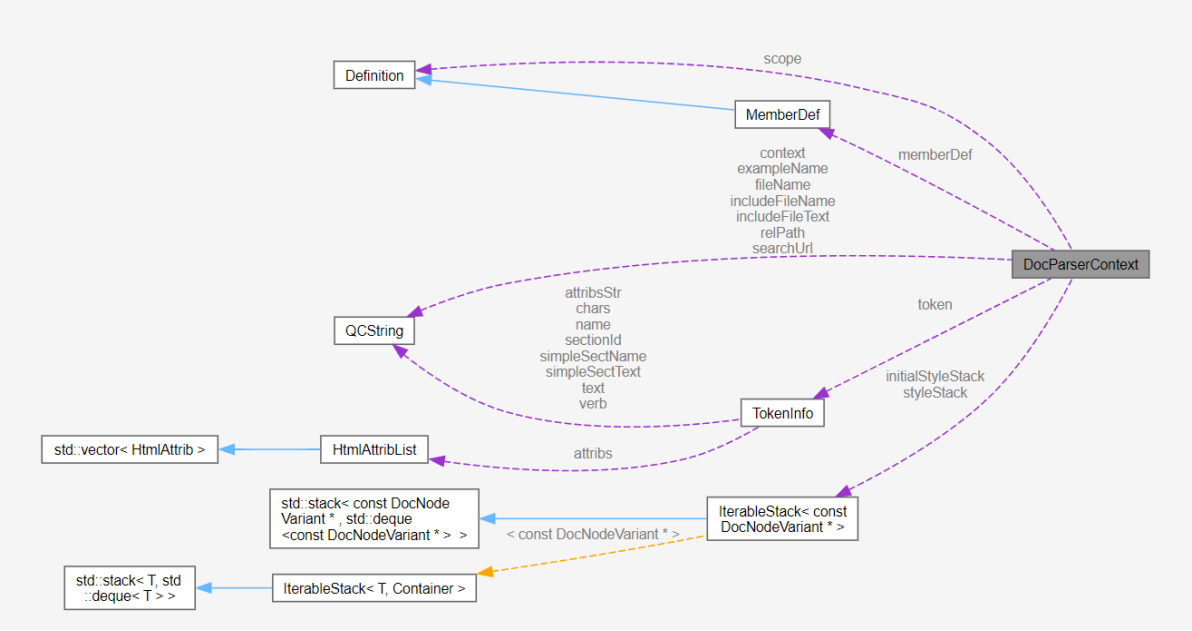
De documentatie van het project wordt gemaakt door de samenvattingen van de verschillende bestanden te combineren. Deze samenvattingen worden gegenereerd op basis van de samenvatting van de bestanden die tot het project behoren. In deze samenvatting behoort de werking van het project, de verschillende bestanden met functies en klassen en de relaties tussen deze bestanden.

2.2. Bestaande documentatie tools

Voor er gekeken wordt naar hoe LLM's mogelijk gebruikt kunnen worden voor het genereren van documentatie is het belangrijk dat er een duidelijk beeld is van de huidige tools die gebruikt worden voor het genereren van documentatie. De documentatie kan in verschillende vormen gegeneerd worden dit kan gaan van een website tot een samenvattend document. Ook kunnen er in de code zelf commentaren geplaatst worden die de werking van de code uitleggen. Hiervoor bestaan er reeds verschillende tools en dit voor verschillende programmeertalen opgelijst in tabel 2.1.

2.2.1. Doxygen

Doxygen (Doxygen, 2023) is een tool die het toelaat om automatisch code documentatie te genereren. Het is een gratis tool die bruikbaar is voor verschillende programmeertalen zoals: C++, C, Python, PHP en Java. Het genereert documentatie in de vorm van HTML, LaTeX, RT. Ook is het in staat om een diagram te genereren met de relaties tussen de verschillende delen van de code. Bijvoorbeeld de relaties tussen de verschillende klassen en functies. Een voorbeeld van een diagram kan gezien worden in figuur 2.1. Zo wordt er een duidelijk beeld verkregen van de structuur van het project.



Figuur (2.1)
Voorbeeld diagram van Doxygen (Doxygen, 2023)

2.2.2. CodeCat

CodeCat.AI (2024) is een online tool die de code analyseert en de docstrings genereert, het is niet open sourced dus er kan niet gekeken worden naar de werking

```python
class A:
    def __init__(self, a: int, b: int):
        self.a = a
        self.b = b

    def sum(self):
        return self.a + self.b
```

(a) Voorbeeld code zonder docstrings van Trofficus (2023)

```python
class A:
    """A class representing an object with two integer attributes.

    Attributes:
        a (int): The first integer attribute.
        b (int): The second integer attribute.
    """
    def __init__(self, a: int, b: int):
        self.a = a
        self.b = b

    def sum(self):
        """Calculates the sum of the two integer attributes.

        Returns:
            int: The sum of the two integer attributes.
        """
        return self.a + self.b
```

(b) Voorbeeld code met docstrings van Trofficus (2023)

Figuur (2.2)
Voorbeeld uitkomst van de tool van Trofficus (2023)

van de tool. Deze tool genereert automtisch de docstrings voor JavaScript code.

2.2.3. GPT4Docstrings

De tool van Trofficus (2023) genereert docstrings voor Python code, het maakt gebruik van GPT-4 (OpenAI, 2023) om de docstrings te genereren. Deze tool leunt sterk aan bij de doelstelling van deze bachelorproef, namelijk het genereren van documentatie met behulp van LLM's. Het nader bekijken van deze tool kan een meerwaarde zijn voor deze bachelorproef.
Zo gebruikt GPT4Docstrings van Trofficus (2023) de Abstract Syntax Tree (AST) van de code om de structuur van de code te begrijpen en vast te nemen. Uit de AST kunnen de juiste stukken code gehaald worden om de docstrings te genereren. Dit kan goed van pas komen voor het genereren van documentatie van Python projecten. Een voorbeeld van deze tool kan gezien worden in figuur 2.2.

2.2.4. Sphinx

Sphinx Team (2023) is een van de meest gebruikte tools voor het genereren van documentatie voor Python projecten. Het genereert documentatie aan de hand van docstrings. Het toont de hierarchie van het project om een duidelijk overzicht te geven. Deze tool is vrij flexibel want het kan uitgebreid worden met verschillende extensies, zodat het alle mogelijke wensen kan vervullen. Bijvoorbeeld de extensie autodoc kan semi-automatisch de docstrings van een module extraheren en in de documentatie plaatsen. Handig wanneer de automatische documentatie generatie van een geheel project gewenst is. Zo kan het project samengevat wor-

den aan de hand van de docstrings van de verschillende python files. Alvorens een Python project gedocumenteerd kan worden met Sphinx dienen alle bestanden aangevuld te worden met docstrings, dit gebeurt echter niet automatisch.

2.2.5. Pdoc

Pdoc (Gallant & Hils, 2023) genereert documentatie in de vorm van een website die een API van de documentatie bevat. Hier kan er makkelijk op de website gezocht worden naar een functie of klasse met de bijhorende documentatie.

Tool	programmeertaal	type
Doxygen	C++, C, Python, PHP, Java	HTML, PDF, markdown
CodeCat	JavaScript	docstring
Sphinx	Python	HTML, LATEX, man pages
Pdoc	Python	API
GPT4Docstrings	Python	docstring

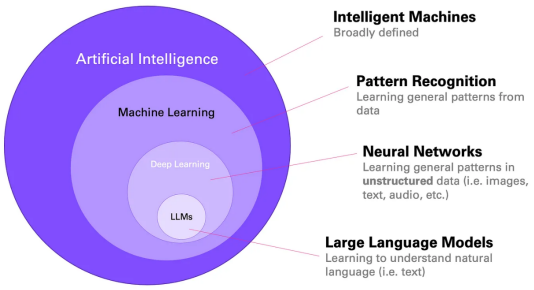
Tabel 2.1: Vergelijking documentatie tools

2.3. Wat zijn Large Language Modellen (LLM)?

Omdat er in deze bachelorproef gebruik gemaakt wordt van Large Language Modellen is het belangrijk dat er een duidelijk beeld wordt geschetst van wat deze modellen zijn. Wat kunnen deze modellen, wat zijn de mogelijke beperkingen en wat is de huidige stand van zaken. In dit hoofdstuk wordt er een antwoord gegeven op de vragen:

- Bestaan er LLM's speciaal getraind op Python code?
- Kunnen LLM's gebruikt worden om documentatie te genereren?

Dit helpt met het verkrijgen van een grondige basiskennis van LLM's.
Het veld waarin AI zich bevindt wordt vaak voorgesteld volgens figuur 2.3, met verschillende lagen (Stöffelbauer, 2023). Deze lagen zijn: Artificiële Intelligentie, Machine Learning, Deep Learning en Large Language Modellen. Voor dat er dieper op de LLM's wordt ingegaan is het belangrijk dat er een duidelijk beeld is wat deze lagen juist inhouden.
AI is een brede term, hiermee wordt vaak verwezen naar slimme machines. Machine Learning (ML) is een subveld van AI, waarin patronen worden herkend tussen een input en een output. ML kan gebruikt worden voor verschillende taken zoals: classificatie, regressie, clustering ... Deep Learning (DL) is een subveld van ML, waarin complexe algoritmen en Deep Neural Networks gebruikt worden om moeilijkere taken uit te voeren. Deep Learning is een krachtige tool die gebruikt



Figuur (2.3)
Artificiële intelligentie in lagen (Stöffelbauer, 2023)

wordt voor verschillende taken zoals: beeldherkenning, spraakherkenning, ... (Stöffelbauer, 2023)
Large Language Modellen zijn geavanceerde AI-systemen die dienen om menselijke taal te verstaan, te genereren en te verwerken. LLM's worden getraind op een grote hoeveelheid tekst wat vaak uit allerlei data zoals artikels of websites gehaald wordt. Volgens Beelen (2023) zorgen Deep Neural Networks ervoor dat LLM's natuurlijke taal verwerken op een gelijkaardige manier die vergelijkbaar is met de menselijke taalvaardigheid. Deze hebben een grote vooruitgang gekend in 2017 door de paper van Vaswani e.a. (2017). Hieruit kwam een nieuw mechanisme tot stand namelijk transformers wat bestaat uit Attention blokken. Enkele voordelen die komen kijken bij het gebruiken van transformers zijn: het kan lange sequenties verwerken, het kan parallel verwerken en het kan de relaties tussen de verschillende delen van de sequentie leren. Hierdoor hebben transformer modellen een snellere trainingsperiode dan vorige neurale netwerken ("What are the primary advantages of transformer models?", 2023).

2.3.1. Transformers en de architectuur van LLM's

Een neuraal netwerk bestaat uit verschillende lagen. Enkele belangrijke blokken die gebruikt worden binnen de transformer laag zijn:

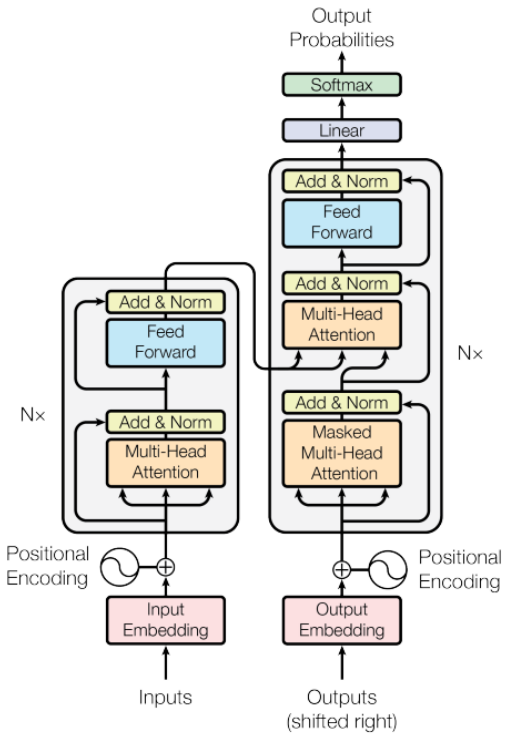
- Self-Attention
- Cross-Attention
- Masked Self-Attention

Deze attention blokken worden gebruikt in de encoder en decoder van een transformer en stromen voort uit het onderzoek van Vaswani e.a. (2017).
Transformers zijn een speciaal type van neurale netwerken die gebruik maken van verschillende attention blokken. Attention is een mechanisme dat gebruikt wordt om de relaties tussen verschillende delen van de invoersequenties te leren. Een transformer bestaat uit een encoder en een decoder. Niet elke transformer bestaat

uit beide een encoder en een decoder, sommige bestaan enkel uit een encoder of een decoder (Hoque, 2023). De encoder wordt gebruikt om de invoersequenties te verwerken en de decoder wordt gebruikt om de uitvoersequenties te genereren. Zo is BERT van Devlin e.a. (2019) een transformer die enkel een encoder heeft en GPT van Radford e.a. (2018) heeft enkel een decoder. De transformer architectuur uit de paper van Vaswani e.a. (2017) kan gezien worden in figuur 2.4.
Self-Attention duidt dynamische gewichten toe aan verschillende elementen binnen de meegegeven sequentie, bijvoorbeeld woorden in een zin. Dit laat het model toe om zich te concentreren op de meest relevante delen van de invoer, terwijl de invloed van minder cruciale delen wordt verminderd. De invoersequentie wordt eerst in drie verschillende vectoren omgezet: Query, Key en Value. De Query vector stelt een specifiek token uit de invoersequentie voor, de Key vector vertegenwoordigt alle tokens en de vector voor Value bevat de feitelijke inhoud die aan elk token is gekoppeld. De similariteit tussen de Query en de Key vector wordt berekend aan de hand van het inwendig product van de twee vectoren. Deze similariteit wordt gebruikt om de gewichten te berekenen die aan de Value vector worden toegekend (Vaswani e.a., 2017).
Masked Self-Attention is een variant van Self-Attention die gebruikt wordt in de decoder van een transformer. In de decoder wordt er een mask gebruikt om enkel de vorige tokens te zien in de sequentie (Vaswani e.a., 2017). Dit vermijdt dat er informatie van de toekomstige tokens gebruikt wordt. Zo kan de transformer niet ”vals spelen”tijdens het train proces.
Cross-Attention is een variant van Self-Attention die gebruikt wordt in de decoder van een transformer. Deze laag gebruikt de informatie van de encoder en de vorige Attention laag van de decoder om de uitvoersequenties te genereren. De Query vector is de uitvoer van de vorige Attention/Cross-Attention laag van de decoder en de Key en Value vector zijn de uitvoer van de encoder (Vaswani e.a., 2017). Doordat de Cross-Attention laag informatie van zowel de encoder als decoder krijgt kan het model de relaties tussen de verschillende delen van de invoersequenties leren. Deze relaties worden dan gebruikt om de uitvoersequenties te genereren.

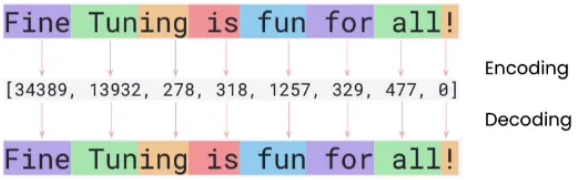
2.3.2. Trainen van LLM’s

Het trainen van LLM’s is een complex proces dat veel tijd en rekenkracht vereist. Dit gebeurt in verschillende stappen. De eerste fase begint bij het verzamelen van een grote hoeveelheid tekst die gebruikt wordt om het model te trainen. Deze tekst wordt gehaald uit verschillende artikelen, websites, boeken, ...
Zo kan het volgende woord in een sequentie van tekst voorspeld worden.
Het model krijgt deze grote hoeveelheid tekst in de pre-training fase. In deze fase leert de LLM grammatica, semantiek, taal patronen en factuele informatie (Cacic, 2023). Voordat de data meegegeven wordt aan het model moet de data gecleaned en geformatteerd worden. Dit gebeurt in het tokenization process. Hier wordt



Figuur (2.4)
Transformer model architectuur (Vaswani e.a., 2017)

de tekst omgezet in tokens die het model kan verwerken 2.5. Woorden kunnen kleiner gemaakt worden zodat de volledige tekst in het model past, dit moet wanneer het model een beperkte input capaciteit heeft (ElHousieny, 2023). Wat dan op zijn beurt omgezet wordt in embeddings. Deze embeddings worden dan meegegeven aan het model om te trainen. Uit de data kunnen dan patronen gehaald worden met behulp van Transformers 2.3.1, maar het is nog niet instaat om vragen of instructies te begrijpen.



Figuur (2.5)
Gesimplificeerde tokenisatie van tekst (TeeTracker, 2023)

De volgende fase bestaat uit het trainen van het model op een dataset met instructies en het antwoord erop. Dit is het gesuperviseerde Fine-Tuning van de LLM (Das, 2024). Het model probeert zo de patronen te leren die nodig zijn om vragen te beantwoorden of instructies te volgen. Hierdoor leert het model instructies te volgen en vragen te beantwoorden.

Er kan gebruik gemaakt worden om het model specifiek aan de wensen van de mens te laten voldoen. Dit kan door het gebruiken van Reinfocement Learning met menselijke feedback (Lambert e.a., 2022). Hierbij geeft de mens feedback aan het model en leert het model bij door deze feedback.
Het model kan achteraf nog extra getraind worden op specifieke data. Volgens Peckham e.a. (2024) gebeurt het Fine-Tunen van een model op een specifieke dataset, zoals Python code of medische data. Dit zorgt ervoor dat het model extra kennis heeft over het gekozen onderwerp.

2.3.3. Fine-Tuning van LLM's

2.3.4. Bestaande LLM's

Momenteel zijn er verschillende LLM's die gebruikt worden voor verschillende taken. Deze LLM's zijn getraind op verschillende datasets en hebben verschillende architecturen. Het is belangrijk dat er een duidelijk beeld is van de verschillende LLM's en hun mogelijkheden. Met dit beeld kan er een goede keuze gemaakt worden voor het genereren van documentatie.
Eén van de grote spelers in de wereld van LLM's is OpenAI. OpenAI heeft verschillende LLM's ontwikkeld gaande van GPT (Radford e.a., 2018) tot GPT-4 (OpenAI, 2023). Het is getraind op een grote hoeveelheid data en heeft een grote capaciteit. Een nadeel is dat GPT-4 een betalende service is (OpenAI, 2023).
Een andere grote speler is Google, Google heeft verschillende LLM's ontwikkeld waaronder BERT van Devlin e.a. (2019) en Gemini (Google, 2024) BERT staat voor Bidirectional Encoder Representations from Transformers, een DL model waar elk output element verbonden is met elk input element (Hashemi-Pour & Lutkevich, 2024). BERT was een eerste stap in de wereld van LLM's voor Google. Sinds kort heeft Google (2024) een nieuwe LLM ontwikkeld genaamd Gemini. Deze LLM is een sterke concurent voor GPT-4 van OpenAI (2023).
De modellen die Google (Google, 2024) uitbracht bestaan uit verschillende versies: Gemini Pro, Gemini Ultra en Gemini Nano. Elke versie is gemaakt voor een specifiek doeleind, zo is Gemini Nano het meest efficiente model voor mobiele toestellen. Gemini Pro is dan weer het beste model voor het schalen van allerlei taken. En Gemini Ultra is het meest capabele en grootste model van Google, dit kan gebruikt worden voor complexe taken. Een van de voordelen van Gemini is dat er een groot aantal input tokens meegegeven kunnen worden, namelijk 1 miljoen tokens (Google, 2024). Dit is aanzienlijk meer dan de 128 duizend tokens van GPT-4.
Een derde speler in de wereld van LLM's is Meta. Meta heeft verschillende LLM's ontwikkeld onder de naam LLama 2 (Meta, 2024). De LLama 2 familie bestaat uit verschillende LLM's die getraind zijn op verschillende data. Sommige zijn extra getraind voor specifiekere doeleinden. Zo is er bijvoorbeeld een LLM getraind op Python code, genaamd Code LLama 2 van Rozière e.a. (2024). Een voordeel van de LLama 2 familie is dat deze LLM's open source zijn en dus voor iedereen toeganke-

lijk zijn.
Antropic heeft ook een LLM ontwikkeld genaamd Claude (Anthropic, 2023). Claude's capaciteiten zijn code generatie, het verstaan van meerdere talen, beelden analyseren en kan geavanceerde redeneringen geven. Er bestaan 3 versies van Claude: Haiku, Sonnet en Opus. Haiku een lichte versie van Claude, Sonnet is de combinatie van performantie en snelheid en Opus is het intelligentste model dat complexe taken kan uitvoeren en begrijpen. Claude is een betalende service, de prijzen zijn afhankelijk van de gekozen versie van Claude.
De verschillen tussen deze LLM's zijn groot, zo is er een verschil in capaciteit, trainingsdata en toegankelijkheid. Het is belangrijk dat er een goede keuze gemaakt wordt voor het genereren van documentatie. Deze keuze zal afhangen van de mogelijkheden van de LLM's en de doeleinden van de documentatie. Het is mogelijk dat er meerdere LLM's getest moeten worden om de beste keuze te maken.

Model	Input (1M tokens)	Output (1M tokens)	Context	Snelheid (t/s)
GPT-4 Turbo (OpenAi, 2024)	$10.00	$30.00	128k	18
GPT-4 (OpenAi, 2024)	$30.00	$60.00	128k	21
GPT-3.5 Turbo (OpenAi, 2024)	$0.50	$1.50	16k	52
Gemini 1.5 Pro Google (2024)	$3.50	$10.50	128k	52
Code LLama (Meta, 2024)	$0.90	$0.90	100k	34
LLama 2 (Meta, 2024)	$0.95	$1.00	100k	34
Claude Opus (Anthropic, 2023)	$15.00	$75	200k	29
Claude Sonnet (Anthropic, 2023)	$3.00	$15	200k	61
Claude Haiku (Anthropic, 2023)	$0.20	$1.20	200k	102

Tabel 2.2: Vergelijking van verschillende LLM's op basis van prijs ($), context (aantal tokens) en snelheid (Tokens per seconde) (ArtificialAnalysis, 2024)

In tabel 2.2 wordt een vergelijking gemaakt tussen verschillende LLM's. Hierin wordt er gekeken naar de prijs van de input en output tokens, de grootte van de context en het aantal tokens dat per seconde verwerkt kan worden.
In tabel 2.3 wordt een vergelijking gemaakt tussen verschillende LLM's tussen twee kolommen met in een eerste kolom de beoordeling van de coding mogelijkheden van het model gequoteerd door menselijke evaluatie. In de tweede kolom staat de beredenering en kennis en deze is er gequoteerd op basis van de MMLU een dataset (Hendrycks e.a., 2020). Hieruit kan geconcludeerd worden dat GPT-4 en GPT-4 Turbo de beste scores behalen op beide vlakken. Een conclusie uit beide tabellen is dat GPT-3.5 Turbo de beste prijs/kwaliteit verhouding heeft.

Model	Coding	Beredenering en Kennis
GPT-4 Turbo (OpenAi, 2024)	86%	85.4%
GPT-4 (OpenAi, 2024)	86%	88.4%
GPT-3.5 Turbo (OpenAi, 2024)	70%	73.2%
Gemini 1.5 Pro Google (2024)	82%	71.9%
Code LLama (Meta, 2024)	/	67.8%
LLama 2 (Meta, 2024)	69%	/
Claude Opus (Anthropic, 2023)	87%	/
Claude Sonnet (Anthropic, 2023)	79%	/
Claude Haiku (Anthropic, 2023)	75%	/

Tabel 2.3: Vergelijking LLM's op basis van beoordeling van menselijke evaluatie en MMLU (ArtificialAnalysis, 2024)

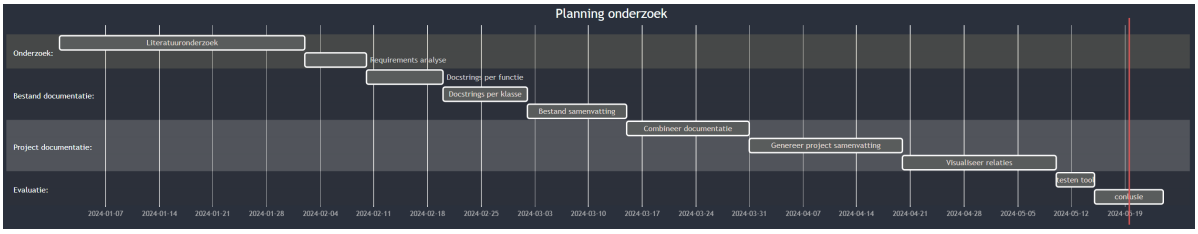
2.4. LLM voor documentatie

Nadat de basiskennis over LLM's gelegd is, hoe deze werken en wat ze doen. Wat enkele bekende LLM's zijn en wat hun mogelijkheden zijn. Ook zijn er bestaande tools besproken die documentatie genereren voor projecten.
Omdat dit onderzoek gaat over het documenteren van een Python project met behulp van LLM's, moet er gekeken worden naar hoe LLM's gebruikt kunnen worden voor het genereren van documentatie. Een LLM kan gebruikt worden voor de verschillende documentatie stukken, samenvattingen, docstrings, relatie tussen de verschillende delen van het project, ...Door de verschillende delen van het project meegegeven aan de LLM met een uitgebreid prompt. Hier kan er telkens aan de Large Language Model gevraagd worden om een samenvatting te maken van wat dit deel van het project doet en wat de uitkomst is. Door dit te herhalen voor alle files van het project kan er één samenvattend document gemaakt worden van het gehele project.
De LLM kan de relaties tussen de verschillende delen van het project leren en vastnemen. Zo wordt er een duidelijk beeld gevormd van de structuur van het project en wat de samenhang is van de verschillende delen van het project. Alle functies van het python project kunnen hier makkelijk teruggevonden worden.
Er kan ook gebruik gemaakt worden van de LLM om de docstrings van de verschillende functies en klassen te genereren. Om zo een betere samenvatting te verkrijgen van de werking van de verschillende delen van het project, door de docstrings te combineren met de samenvattingen van de LLM. Door telkens de docstrings en de naam van de verschillende functies en klassen mee te geven aan de LLM kan er een betere samenvatting gemaakt worden van het gehele project.
Wanneer een huidige LLM niet instaat is om de gewenste documentatie te genereren kan een LLM gefinetuned op specifieke data, Python code en de bijhorende documentatie. Hier is er een grote hoeveelheid data van Python projecten met

de bijhorende documentatie nodig. Ook is het duur en tijdrovend om een LLM te finetunen.

3

Methodologie



Figuur (3.1)
Tijdslijn onderzoek

Het onderzoek is in drie fases opgedeeld. De eerste fase omvat de literatuurstudie. In deze literatuurstudie wordt er onderzocht wat de huidige stand van zaken is omtent de technologie en mogelijkheden voor de documentatie van Python projecten met behulp van Large Language Modellen. Zo wordt er gekeken naar wat LLM's zijn, hoe ze werken en wat bestaande tools zijn voor het genereren van documentatie.
Nadat er een duidelijk beeld gevormd is in de literatuurstudie, kan er begonnen worden aan de tweede fase. Hier wordt er een tool ontwikkeld die een Python bestand kan analyseren en op basis daarvan documentatie kan genereren, aan de hand van het toevoegen van docstrings aan de code. Dit doet het eerst per functie dan per klasse en uiteindelijk voor het gehele bestand. Dit kan later gebruikt worden om een samenvatting van het project te genereren in de volgende fase. De uitkomst van deze fase is een tool die de documentatie van een Python bestand kan genereren. Door aan prompt engineering te doen, met het prompt dat meegegeven wordt aan de LLM, kunnen de bekomen docstrings accurater worden. Verder wordt er gekeken naar hoe de documentatie van Python functies gebruikt kan worden voor het maken van een gehele samenvatting van het project. Dit wordt gedaan op basis van huidige methoden om docstrings aan te maken en te gebruiken.

Erna kunnen de verschillende docstrings met de bijhorende naam van de functie of klasse gebruikt worden om een samenvatting te genereren. Deze informatie kan dan gegeven worden aan de Large Language Modellen om een samenvatting te genereren.
De derde fase beslaagt het documenteren van een geheel project. Door de vorige fases te combineren, het documenteren van de individuele bestanden en het genereren van een samenvatting van het project, kan er een tool gemaakt worden die de documentatie van een geheel project kan genereren. Deze documentatie bestaat uit de individuele documentatie van de bestanden en een samenvatting van het gehele project. Alsook wordt er gekeken naar hoe de relaties tussen de verschillende bestanden gevisualiseerd kunnen worden.
De laatste fase is het evalueren van de tool. Hier wordt er gekeken naar de kwaliteit van de documentatie die de tool genereert. Dit wordt gedaan door de documentatie van de tool te vergelijken met de handgeschreven documentatie van een project. Dit wordt gedaan voor een bestand, een klein project en een groot project.

3.1. Requirementsanalyse

De requirementsanalyse is een belangrijk onderdeel van het onderzoek. Hier wordt er gekeken naar wat de tool moet kunnen en wat de verwachtingen zijn van de tool.

3.1.1. Functionele requirements

Should Have:

- **De tool moet in staat zijn om docstrings te genereren voor functies en klassen voor een Python bestand.**
  De tool genereert docstrings voor functies en klassen in een Python bestand. Dit gebeurt door de code te analyseren en op basis daarvan een docstring te genereren.
- **De tool moet in staat zijn om een samenvatting van een Python bestand te genereren.**
  De tool maakt een samenvatting van een Python bestand. In deze samenvatting zitten de belangrijkste zaken van het bestand. Zijnde de functies en klassen die erin voorkomen en wat deze doen, alsook hun eventuele parameters en de uitkomst.
- **De tool moet in staat zijn om een samenhangende samenvatting van een Python project te genereren.**
  De tool dient een samenvatting te maken van een Python project. Deze samenvatting bestaat uit de individuele samenvattingen van de bestanden en een overkoepelende samenvatting van het project. Hierin staan alle functies en klassen die in het project voorkomen en wat deze doen, alsook hun eventuele parameters en de uitkomst.

- **De tool moet in staat zijn om de relaties tussen de verschillende bestanden van een project te visualiseren.**
  De tool visualiseert de relaties tussen de verschillende bestanden van een project. Zo is er geweten waar de bestanden naar verwijzen en van waar ze refereren.

Could Have:

- **De tool moet in staat zijn om de documentatie van een Python bestand te genereren in verschillende formaten.**
  De tool genereert de documentatie van een Python bestand in verschillende formaten. Zo kan de gebruiker kiezen in welk formaat hij de documentatie wil.
- **De tool moet in staat zijn om de relaties tussen de verschillende functies en klassen van een bestand te visualiseren op project niveau**
  De tool visualiseert de relaties tussen de verschillende functies en klassen van een bestand op project niveau. Zo is er geweten waar de functies en klassen naar verwijzen en van waar ze refereren.

Nice to Have:

- **De tool moet in staat zijn om de documentatie van een Python bestand te genereren in verschillende talen.**
  De tool genereert de documentatie van een Python bestand in verschillende talen. Zo kan de gebruiker kiezen in welke taal hij de documentatie wil.
- **De tool moet in staat zijn om de documentatie van een Python bestand te genereren in verschillende stijlen.**
  De tool genereert de documentatie van een Python bestand in verschillende stijlen. Zo kan de gebruiker kiezen in welke stijl hij de documentatie wil.

3.1.2. Niet-functionele requirements

- **De tool moet gebruiksvriendelijk zijn.**
  De tool moet eenvoudig te gebruiken zijn. Dit betekent dat de tool intuïtief moet zijn en dat de gebruiker geen moeite moet doen om de tool te gebruiken.
- **De tool moet betaalbaar zijn.**
  Dit wil zeggen dat de tool geen hoge kosten met zich meebrengt.
- **De tool moet snel werken.**
  De tool moet snel werken. Dit betekent dat de tool snel de documentatie moet kunnen genereren. Sneller dan dat een persoon dit zou kunnen.

- **De tool moet leesbare documentatie genereren.**
  De documentatie die de tool genereert moet leesbaar zijn. Dit wil zeggen dat de documentatie duidelijk moet zijn en dat de gebruiker er gemakkelijk informatie uit kan halen.

3.2. Opstellen van een long-list

De long-list bestaat uit de verschillende tools die gebruikt kunnen worden voor het genereren van documentatie voor Pythonprojecten. Deze tools worden onderzocht en vergeleken om te kijken welke het beste past bij de requirements van de tool.
De oplijsting van de tools is gebaseerd op de literatuurstudie en bestaat uit de volgende tools:

- Sphinx (Sphinx Team, 2023)
  Een tool die gebruikt wordt voor het genereren van documentatie voor projecten met reeds een docstring.
- Doxygen (Doxygen, 2023)
  Een tool die gebruikt wordt voor het genereren van documentatie voor projecten met reeds een docstring.
- Pdoc (Gallant & Hils, 2023)
  Een tool die een API genereert voor Python projecten met reeds een docstring.
- GPT4Docstrings (Trofficus, 2023)
  Een tool die docstrings genereert voor Python projecten met behulp van GPT4 OpenAI (2023).

Door de verschillende tools te onderzoeken en te vergelijken aan de hand van de requirementsanalyse kan er een keuze gemaakt worden welke tool het beste past bij de requirements van de tool. Als er geen tool is die voldoet aan de requirements, kan er gekeken worden naar het combineren van verschillende tools of gebruiken van delen van verschillende tools om zo een eigen tool te maken.
Door eerst te kijken naar de programmeerttalen waarvoor de tools gebruikt kunnen worden, kan er al een eerste selectie gemaakt worden uit tabel 2.1. Zo kan er gekeken worden naar de tools die gebruikt kunnen worden voor Python projecten. Vervolgens kan er gekeken worden naar het type documentatie dat de tools genereren. Hierbij kan er gekeken worden naar de tools die docstrings genereren, aangezien dit de basis is voor het genereren van de gewenste documentatie.
Uit deze tabel 3.1 kan er geconcludeerd worden dat geen enkele tool voldoet aan de volledige requirements van de tool. Doordat de Doxygen en Sphinx tools geen docstrings genereren en dit de basis is voor de documentatie, kunnen deze tools

Tool	docstrings	samenvatting bestand	samenvatting project	visualisatie
Doxygen	nee	ja	ja	ja
Sphinx	nee	ja	ja	nee
GPT4Docstrings	ja	nee	nee	nee
Pdoc	nee	ja	nee	nee

Tabel 3.1: Requirementsanalyse van de verschillende tools

niet gebruikt worden. Pdoc genereert enkel samenvattingen van een bestand dat reeds aangevuld is met docstrings en kan dus ook niet gebruikt worden. Omdat GPT4Docstrings enkel docstrings genereert en geen samenvattingen van bestanden of projecten, kan er gekeken worden hoe deze tool dit doet.

4

Bestand documentatie

4.1. Inleiding

In dit hoofdstuk wordt er gekeken naar de documentatie van een Python bestand. Eerst wordt de code van het bestand geanalyseerd en worden de verschillende functies en klassen geïdentificeerd. Op basis van deze functies en klassen worden er docstrings gegenereerd, die opnieuw toegevoegd worden aan de code van het bestand. Daarna worden de docstrings binnen het bestand gebruikt om een samenvatting van het bestand te genereren.
Deze samenvatting kan dan als basis gebruikt worden voor het genereren van documentatie voor een Python project. Dit wordt verder onderzocht in het volgende hoofdstuk. Voor dit kan gebeuren, moet de bestand documentatie op punt staan en geoptimaliseerd worden.
Omdat dit onderzoek een bepaalde scope heeft, is er gekozen om enkel te kijken naar het documenteren van correcte Python bestanden. Dit wil zeggen dat er verwacht wordt dat de code correct is en dat er geen syntax fouten in de code zitten. Er wordt niet gekeken naar het documenteren van bestanden met syntax fouten of bestanden die niet correct zijn.

4.2. Abstract Syntax Tree

Voor er docstrings gegenereerd kunnen worden, moet er eerst gekeken worden naar hoe de code van een Python bestand geanalyseerd kan worden. Uit de literatuurstudie is gebleken dat de verschillende functies en klassen in een bestand geïdentificeerd en geëxtraheerd kunnen worden aan de hand van een Abstract Syntax Tree (AST). Het analyseren van de code van de tool GPT4Docstrings gemaakt door Trofficus (2023) heeft een beter beeld gegeven van hoe een AST eruit ziet en hoe deze gegenereerd kan worden.

```
def get_functions(self):
    functions = {}
    for node in ast.walk(self.tree):
        if isinstance(node, ast.FunctionDef) or
            isinstance(node, ast.AsyncFunctionDef):
            function_code = ast.unparse(node)
            functions[node.name] = function_code
    return functions
```

Listing 4.2.1: Voorbeeld van het ophalen van functies uit een AST.

Een AST is een boomstructuur die de syntactische structuur van een programma weergeeft. Per knoop in de boom wordt er een deel van de code voorgesteld. Deze knoop kan dan weer kinderen hebben die deel uitmaken van de code. Elke knoop in de boom heeft een type en een waarde.
Het inlezen van een Python bestand en deze omzetten naar een AST, maakt het mogelijk om de code van het bestand te manipuleren. Zo kunnen de verschillende import statements, functies en klassen geïdentificeerd worden.
In 4.2.1 wordt er met behulp van de ast.walk functie door de AST gelopen. Elke node in de AST wordt gecontroleerd of het een functie of een asynchrone functie is. Als dit het geval is, wordt de code van de functie opgehaald en toegevoegd aan een dictionary.

4.3. Docstrings

Binnen deze bachelorproef wordt de docstring stijl van Google gehanteerd (Google Python Team, 2024). Deze docstrings bestaan uit een korte beschrijving van de functie of klasse, de argumenten die de functie verwacht en de return waarde van de functie. Een voorbeeld van een docstring voor een functie die controleert of een getal een priemgetal is 4.3.1.
Deze docstrings dienen gegenereerd te worden voor elke functie en klasse in een Python bestand op basis van de huidige code van de functie of klasse.

4.4. Keuze van model

Omdat er in dit onderzoek slechts gekeken wordt naar het genereren van documentatie met behulp van LLMs, is het niet nodig om verschillende modellen te vergelijken. Het vergelijken van verschillende modellen valt buiten de scope van dit onderzoek. Er wordt enkel gekeken naar de resultaten van het model en hoe deze verbeterd kunnen worden.
Het gekozen model is de LLM GPT3.5-turbo van **OpenAI<empty citation>**. Dit is

```python
def is_prime(n: int) -> bool:
    """
    Check if a number is prime.

    Args:
        n (int): The number to check.

    Returns:
        bool: True if the number is prime, False otherwise.
    """
```

Listing 4.3.1: Voorbeeld van een docstring voor een functie die controleert of een getal een priemgetal is.

een krachtig model dat getraind is op een grote hoeveelheid data en in staat is om natuurlijke taal te genereren. Aangezien dit model getraind is op een grote hoeveelheid data is het in staat om met het juiste prompt de gewenste uitkomst te genereren. De lage kosten en het gemak van gebruik hebben ervoor gezorgd dat er voor dit model verkozen is boven andere modellen.

4.5. Prompting

Het beste resultaat kan gevonden worden door aan prompt-engineering te doen. Er wordt een prompt gegeven aan het model met duidelijke instructies over wat er verwacht wordt, als deze niet volstaan wordt het prompt bewerkt. De nieuwe uitkomst wordt dan geëvalueerd en indien nodig wordt het prompt opnieuw aangepast. Er werden verschillende prompts getest om het beste resultaat te bekomen. Er zijn prompts gemaakt voor het genereren van docstrings voor functies en klassen.

4.5.1. Prompt engineering voor functies

In bijlage B.1.1 wordt er een prompt gegeven aan het model om een docstring te genereren voor een functie. Dit prompt bevat een voorbeeld van een functie met de verwachte uitkomst. Maar de instructies zijn niet duidelijk genoeg.
De volgende versie van dit prompt bevat duidelijkere instructies B.1.2. Het is belangrijk dat de prompt duidelijk is en dat het model weet wat er verwacht wordt. In de instructies staat exact wat er verwacht wordt van het model. Dat de gegenereerde functie een docstring moet bevatten en type hints. De code van de functie mocht niet aangepast worden en er mochten geen imports toegevoegd worden. Ook mocht het model niets veronderstellen over de functie of de data types die gebruikt worden in de functie.

Aangezien de uitkomst van prompt-v2 niet altijd correct was, werd er een nieuwe prompt gemaakt B.1.3. Dit keer bevat het prompt de bestaande imports van het Python bestand en de code van de functie. Het bevat de imports omdat het model deze nodig heeft om de juiste type hints te genereren. Het model maakte foute veronderstellingen maakte ook al werd er in de instructies duidelijk meegegeven dat dit niet de bedoeling was.
Deze veronderstellingen kwamen er omdat het model de code van de functie sporadisch aanpaste. In het aangepaste prompt werd er duidelijk gemaakt dat de uitkomst van het prompt slechts de functienaam met typehint en de docstring moest bevatten. De code van de functie moest niet meer in de uitkomst staan.

4.5.2. Prompt engineering voor klassen

Het prompt engineering process voor klassen liep gelijkaardig met dat van functies. Er werd een prompt gemaakt met duidelijke instructies en een voorbeeld van een klasse met de verwachte uitkomst. De instructies waren gelijkaardig aan die van de functies, maar dan voor klassen.
De verschillende prompt versies 1-4 voor klassen zijn identiek aan die van functies, maar dan met de code van een klasse in plaats van een functie. Voor klassen werd er uiteindelijk gekozen om de code van de klasse niet in de prompt te zetten. Maar om de docstring van de klassen te genereren op basis van de gegenereerde docstrings van de functies in de klasse. Deze werden meegegeven in de prompt samen met de verschillende imports. Deze versie van het prompt gaf de beste resultaten en is te zien in bijlage B.1.4. Het voorbeeld van de klasse in het prompt is een klasse met daarin enkele functies met een docstring.

4.5.3. Prompt engineering voor samenvatting

Voor het genereren van een samenvatting van een Python bestand, werd er een prompt gemaakt met de gegenereerde docstrings van de functies en klassen. De verschillende docstrings werden meegegeven aan de parameter `code_content` en de naam van het bestand aan de parameter `filename`. Het volledige prompt met alle beschrijvingen kan gevonden worden in B.1.5.

4.6. Toevoegen van gegenereerde docstrings

De gegenereerde docstrings worden vervolgens toegevoegd aan de code van de functies en klassen. Dit gebeurt door de code van de functie of klasse te vervangen door de gegenereerde docstring. Met behulp van Abstract Syntax Trees kan dit eenvoudig gebeuren.
Omdat de uitkomst van de prompts altijd in de vorm van een string met een omsloten code blok wordt gegeven, zoals te zien in B.1.1. Dient dit weg gehaald te worden voor het toevoegen aan de code van het bestand. Dit wordt gedaan door de uitkomst van het model te parsen en de code blokken te verwijderen, wat behou-

```
def replace_functions(self, functions):
    tree = self.tree
    for node in ast.walk(tree):
        if isinstance(node, (ast.FunctionDef,
            ast.AsyncFunctionDef)) and node.name in functions:
            new_func_def = ast.parse(functions[node.name]).body
            tree.body.insert(tree.body.index(node), new_func_def)
    self.tree = tree
```

Listing 4.6.1: Vervangen van de code van een functie door de gegenereerde docstring. B.2.1

den moet worden is de gegenereerde docstring samen met de functie of klasse declaratie.
De eerste versie van de code voor het toevoegen van de docstrings aan de code van de functies en klassen is te zien in 4.6.1. In deze versie wordt er door de AST gelopen en wordt er gekeken of de node een functie of klasse is met de juiste naam. Als dit het geval is, wordt de code van de functie of klasse vervangen door de gegenereerde docstring. Het toevoegen van de docstring wordt gedaan door de nieuwe code van de functie of klasse in de AST te plaatsen op de plaats van de oude code. En dan de nieuwe AST te gebruiken als de nieuwe code van het bestand.
Deze code werkte niet volledig zoals verwacht. De oude code werd niet verwijderd uit de AST. Dit zorgde voor dubbele functies en klassen in de AST. Waarvan één met docstring en één zonder. Dit werd opgelost door de oude code te verwijderen uit de AST B.2.2. Alsook werd de code aangepast zodat functies die binnen in een klasse gedefinieerd zijn ook vervangen kunnen worden. Het verwijderen uit de lijst met functies werd ook toegevoegd zodat de functies die al vervangen zijn niet opnieuw vervangen worden.
Deze versie werkte zoals verwacht voor kleine Python files zonder ingewikkelde structuren zoals nested functies of klassen. Hierdoor moest de code opnieuw aangepast worden omdat de gegenereerde docstrings niet altijd correct toegevoegd werden. Een nadeel van het werken met AST is dat de parent node van nested functies niet opgeslagen worden. Dit werd opgelost door het vervangen recursief te laten gebeuren, een betere oplossing dan het gebruiken van if else statements 4.6.2.
Door de code recursief te laten lopen, kan de code van nested functies en klassen ook vervangen worden. Door het gebruiken van de parent node van de node die vervangen dient te worden, kan de docstring op de juiste index geplaatst worden. De nieuwe node wordt toegevoegd aan de parent node en de oude node wordt verwijderd. Zo kunnen grote Python bestanden met complexe structuren ook correct

```
def _replace_functions(self, node, functions):
    if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef))
        and node.name in functions:
        new_func_def = ast.parse(functions[node.name]).body[0]
        new_func_def.body.extend(node.body)
        parent_node = self._get_parent_node(node)
        index = parent_node.body.index(node)
        parent_node.body.remove(node)
        parent_node.body.insert(index, new_func_def)
        functions.pop(node.name)
    for child_node in ast.iter_child_nodes(node):
        self._replace_functions(child_node, functions)
```

Listing 4.6.2: Vervangen van de code van een functie door de gegenereerde docstring. B.2.3

vervangen worden.

4.7. Bestand samenvatting genereren

De laatste stap in het proces van het documenteren van een Python bestand is het genereren van een samenvatting van het bestand. Deze samenvatting wordt gemaakt op basis van de reeds gegenereerde docstrings van de verschillende functies en klassen van het bestand. Het gebruiken van een prompt waar alle docstrings meegegeven worden kan een correcte samenvatting als eindresultaat bekomen. Hierin hoort er een korte beschrijving van het bestand te staan en een lijst van de functies en klassen die in het bestand voorkomen. Per functie en per klasse komt er een korte beschrijving te staan van wat deze doen.
Deze samenvatting wordt gegenereerd door het model de gegenereerde docstrings van de functies en klassen mee te geven in een prompt, zoals het prompt B.1.5. Dit is de laatste stap in het proces van het documenteren van een Python bestand. Deze samenvatting kan dan gebruikt worden als basis voor het genereren van documentatie voor een Python project.

5

Project documentatie

5.1. Inleiding

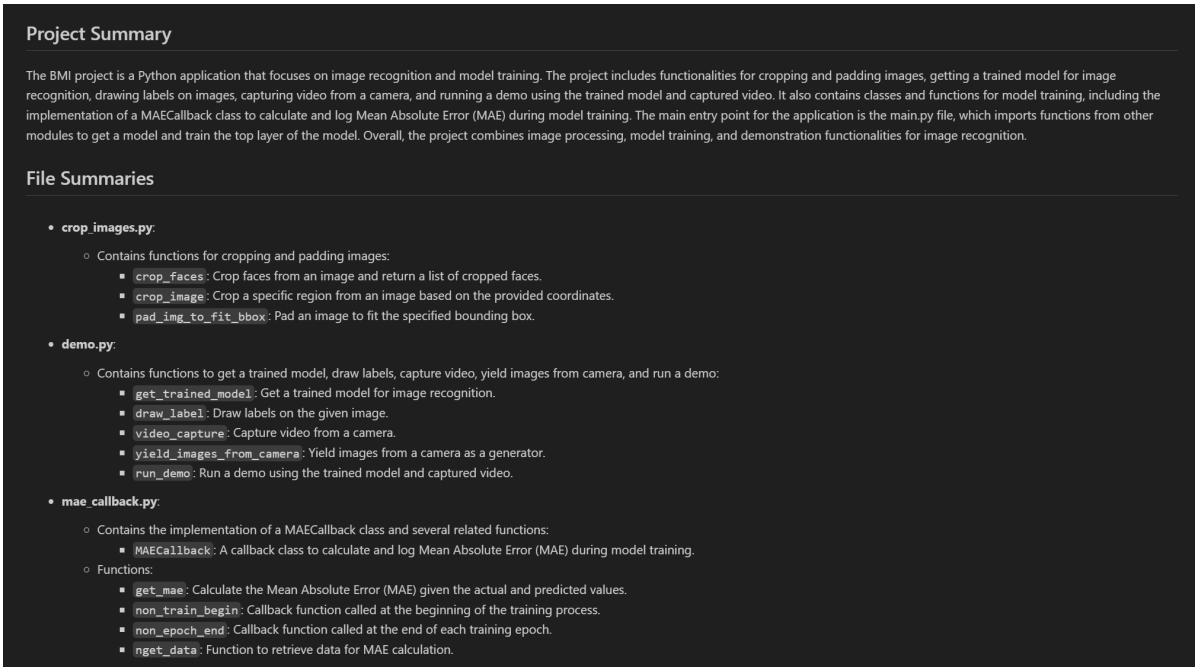
In dit hoofdstuk wordt er gekeken naar hoe de individuele samenvattingen van een Python bestand gebruikt kunnen worden om een samenvatting van het gehele project te maken. Alsook wordt er verder gekeken naar hoe de relaties tussen de verschillende bestanden gevisualiseerd kunnen worden. Dit om een zo goed mogelijk overzicht te krijgen van het project, zonder dat er handmatig documentatie moet worden geschreven.

5.2. Project Samenvatting

De samenvatting van een Python project kan gemaakt worden door de individuele samenvattingen van de bestanden samen te voegen. Deze samenvatting kan bekomen worden door elk python bestand in het project te laten documenteren en de samenvatting ervan op te slaan. Erna kunnen deze samenvattingen samengevoegd worden om dan mee te geven aan een Large Language Model. Deze samenvattingen worden gegenereerd door het aanroepen van de functie `generate_file_summaries()`. Deze functie maakt gebruik van de klasse `FileDocumenationGenerator()` die de samenvattingen van de bestanden genereerd en opslaat in een dictionary. De code van deze functie is te vinden in B.2.5.

Door een duidelijk prompt mee te geven aan het model, kan er specifiek gevraagd worden welke functies en klassen er in het project zitten. En dat per bestand duidelijk opgelijst. Samen met deze oplijsting wordt er ook een korte samenvatting van het gehele project weer gegeven.

Een voorbeeld van de uitkomst is te zien in 5.1. Hier is te zien dat er een duidelijk overzicht is van welke functies en klassen er in het project zitten, alsook wat het gehele project inhoudt.



Project Summary

The BMI project is a Python application that focuses on image recognition and model training. The project includes functionalities for cropping and padding images, getting a trained model for image recognition, drawing labels on images, capturing video from a camera, and running a demo using the trained model and captured video. It also contains classes and functions for model training, including the implementation of a MAECallback class to calculate and log Mean Absolute Error (MAE) during model training. The main entry point for the application is the main.py file, which imports functions from other modules to get a model and train the top layer of the model. Overall, the project combines image processing, model training, and demonstration functionalities for image recognition.

File Summaries

- **crop_images.py**:
  - Contains functions for cropping and padding images:
    - `crop_faces`: Crop faces from an image and return a list of cropped faces.
    - `crop_image`: Crop a specific region from an image based on the provided coordinates.
    - `pad_img_to_fit_bbox`: Pad an image to fit the specified bounding box.
- **demo.py**:
  - Contains functions to get a trained model, draw labels, capture video, yield images from camera, and run a demo:
    - `get_trained_model`: Get a trained model for image recognition.
    - `draw_label`: Draw labels on the given image.
    - `video_capture`: Capture video from a camera.
    - `yield_images_from_camera`: Yield images from a camera as a generator.
    - `run_demo`: Run a demo using the trained model and captured video.
- **mae_callback.py**:
  - Contains the implementation of a MAECallback class and several related functions:
    - `MAECallback`: A callback class to calculate and log Mean Absolute Error (MAE) during model training.
  - Functions:
    - `get_mae`: Calculate the Mean Absolute Error (MAE) given the actual and predicted values.
    - `non_train_begin`: Callback function called at the beginning of the training process.
    - `non_epoch_end`: Callback function called at the end of each training epoch.
    - `nget_data`: Function to retrieve data for MAE calculation.

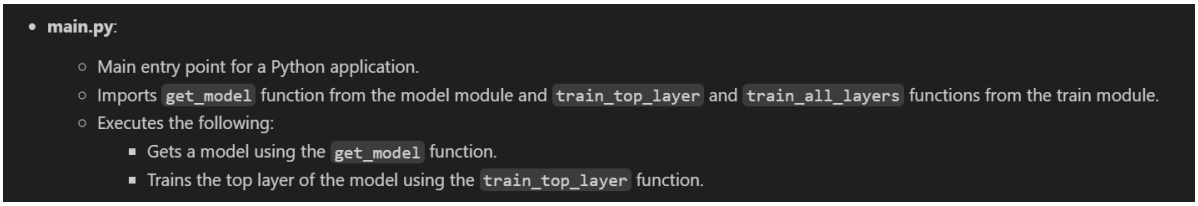
Figuur (5.1)
Voorbeeld van een project samenvatting

5.2.1. Keuze van welke bestanden te documenteren

Het is belangrijk om te kijken naar welke bestanden er gedocumenteerd moeten worden. Omdat het gaat over een Python project, is het belangrijk dat alle Python bestanden gedocumenteerd worden. Er is de keuze gemaakt om het bestand `__init__.py` niet te documenteren, omdat dit bestand vaak niet relevant is voor de documentatie van het project. Dit omdat het bestand vaak leeg is of slechts minimale functionaliteit bevat. Ook bevat het soms enkele configuratie opties die niet relevant zijn voor de documentatie.

5.2.2. Documentatie van bestanden zonder functies of klassen

Omdat er eerst vanuit gegaan wordt dat elk bestand functies of klassen bevat, is het belangrijk om te kijken naar bestanden die dit niet bevatten. Deze bestanden dienen ook gedocumenteerd te worden om een volledig overzicht te krijgen van het project. Als er geen apart prompt voorzien wordt dan zal het model hallucineren en een samenvatting verzinnen, dit is niet de bedoeling. Er is gebruik gemaakt van een prompt B.1.6 die vraagt om de werking van het document uit te leggen en de eventuele imports die het bestand bevat. In dit prompt wordt er duidelijk gedefinieerd wat er in de documentatie moet staan. En aan de hand van een voorbeeld wordt er getoond hoe de documentatie eruit moet zien. Een voorbeeld van de uitkomst in de project documentatie van een bestand zonder functies is te zien in 5.2.



Figuur (5.2)
Voorbeeld van de documentatie van een bestand zonder functies of klassen

5.3. Prompting voor project documentatie

Aangezien de documentatie van een project bestaat uit de documentatie van individuele Python bestanden, is het belangrijk dat deze op een correcte manier gedocumenteerd worden. Dit wordt gerealiseerd met behulp van een prompt die de samenvatting van een Python bestand op een correcte manier interpreteert en omzet naar de juiste documentatie.
De gehele samenvatting van het project werd gemaakt door de individuele samenvattingen van de bestanden samen te voegen en dit mee te geven met het prompt B.1.7. Dit prompt vraagt om de samenvatting van het project te maken en uit de individuele samenvattingen de functies en klassen op te lijsten. Dit prompt werkte goed voor kleine Python projecten, omdat er bij de gekozen LLM een beperkt context window van 16,385 tokens (OpenAi, 2024).
Hierdoor was het niet mogelijk om de gewenste samenvatting te creeren door alle individuele samenvattingen mee te geven aan het model. Een oplossing die gevonden werd is de volgende: er werden verschillende kleinere prompts meegegeven met het model. Dit prompt maakt per samenvatting van een Python bestand de documentatie van de verschillende functies en klassen. De functies en klassen worden opgelijst met het juiste formaat en een kleine uitleg. Deze resultaten van de verschillende kleine prompts werden dan code matig samengevoegd tot een geheel. Het prompt dat gebruikt werd is te zien in B.1.8. Dit prompt vraagt om de functies en klassen van een Python bestand op te lijsten en een korte uitleg te geven van wat deze functies en klassen doen. De uitkomst is weergegeven met een duidelijk voorbeeld binnen het prompt en volgens een markdown formaat. De documentatie genereerd door dit prompt is te zien in 5.1.

5.4. Visualisatie van relaties tussen bestanden

Om een goed overzicht te krijgen van het project is het belangrijk om de relaties tussen de verschillende bestanden te visualiseren. Dit kan gedaan worden door gebruik te maken van graven om de relaties tussen de bestanden weer te geven.
Door gebruikt te maken van de tool Pyvis van West Health Institute Revision (2018), wat een soortgelijke tool als degene die gebruikt wordt door Doxygen (2023). Met deze tool kunnen er graven gemaakt worden om zo de relaties tussen de verschil-

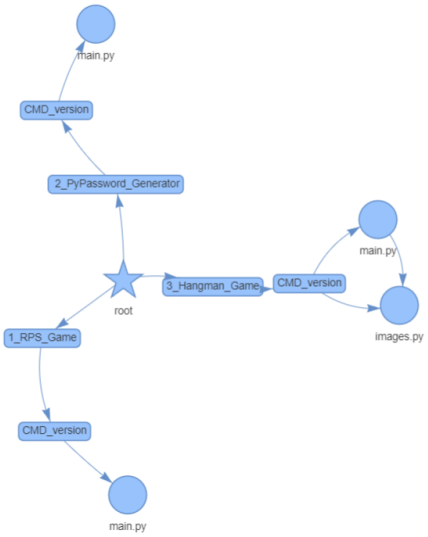
lende bestanden weer te geven. Deze graven kunnen dan gebruikt worden om een duidelijk overzicht te krijgen van het project.

5.4.1. Genereren van de relaties tussen bestanden in een project

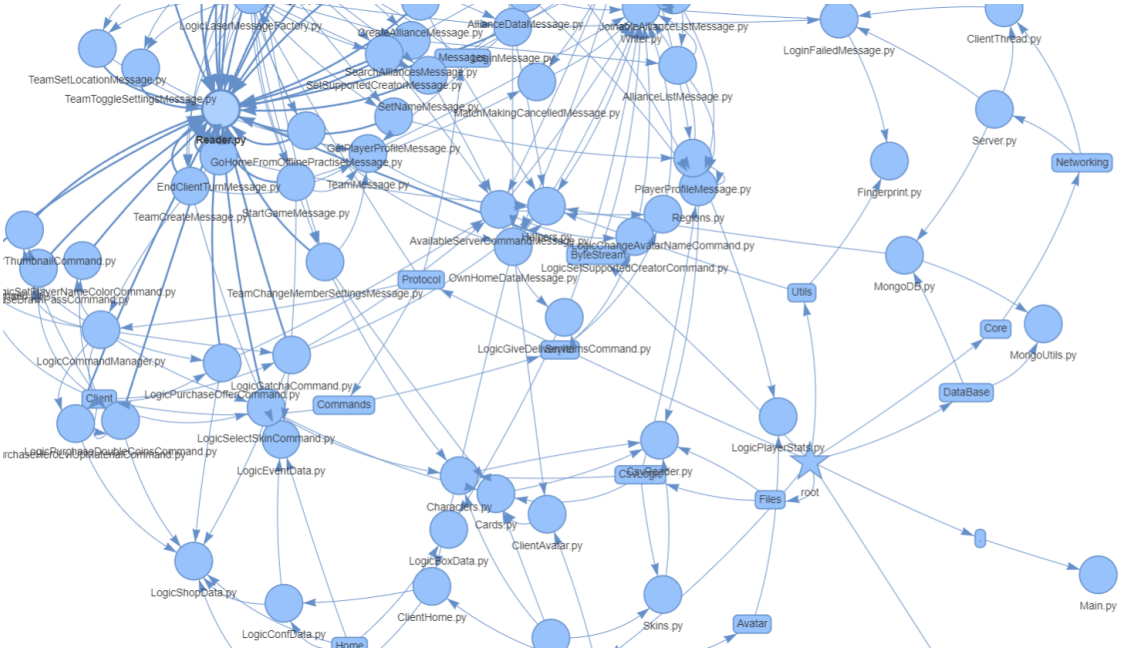
Om de relaties te bekomen tussen alle bestanden en mappen in een project is er een prompt meegegeven aan het Large Language Model. In dit prompt worden de imports van alle bestanden opgelijst en wordt er gevraagd om de relaties tussen de bestanden weer te geven. Omdat een bestand dat een functie uit een ander bestand gebruikt deze functie importeerd, staat deze ook tussen de verschillende imports van het bestand. Hierdoor kan er gekeken worden naar de imports van de verschillende bestanden en kunnen zo de relaties tussen de bestanden achterhaald worden.
Het prompt dat gebruikt werd is te zien in bijlage B.2.4. Er zijn verschillende iteraties van dit prompt gemaakt om de beste resultaten te bekomen. De uitkomst van dit prompt is een CSV bestand met daarin het pad van het bestand, de bestandsnaam, het pad van de folder waarin het bestand zit en een lijst van alle geimporteerde bestanden. Doordat GPT3.5 (OpenAi, 2024) een zo goed mogelijk antwoord probeert te geven op de vraag, is het belangrijk om de vraag zo duidelijk mogelijk te stellen. Door verschillende voorbeelden mee te geven in het prompt kan het model een beter antwoord geven. Omdat het voorbeeld van cruciaal belang is, heeft het enige tijd geduurd om het juiste prompt te vinden en de kinderziektes eruit te halen. Zo zijn schrijffouten en onduidelijkheden in het prompt aangepast om een beter resultaat te bekomen.

5.4.2. Visualisatie van de relaties

Eens er een goed CSV bestand is bekomen kunnen de relaties tussen de bestanden gevisualiseerd worden. Dit wordt gedaan met behulp van de tool Pyvis (West Health Institute Revision, 2018). Deze tool haalt de relaties uit het CSV bestand en voegt deze toe aan een graaf. Dit door eerst de verschillende nodes toe te voegen en dan de edges tussen de nodes. De code waarop dit gebeurt is te vinden in bijlage B.2.6. Als dit voor alle bestanden gedaan is, kan er een duidelijk overzicht bekomen worden van de relaties tussen de bestanden door de graaf te exporteren naar een HTML bestand. Dit HTML bestand kan dan geopend worden in een browser om de graaf te bekijken, alsook kunnen de nodes versleept worden om een beter overzicht te bekomen.
Er is een voorbeeld van een graaf te zien in 5.3, alsook is er een voor een groot project een graaf gemaakt om te kijken of de relaties tussen de bestanden duidelijk weergegeven worden op grote schaal 5.4. De relaties op deze graaf zijn zichtbaar echter is het niet altijd duidelijk omdat er veel bestanden zijn en er bepaalde bestanden vaak geïmporteerd zijn. Dit kan ervoor zorgen dat de graaf onoverzichtelijk wordt met de mate de grote van het projectOpenAi.



Figuur (5.3)
Voorbeeld van een graaf van de relaties tussen bestanden



Figuur (5.4)
Voorbeeld van een fragment van een graaf van de relaties tussen bestanden van een groot project

6

Uitbreidingen

Aangezien dit onderzoek een beperkte scope heeft, zijn er enkele uitbreidingen die kunnen worden toegevoegd om het onderzoek te verbeteren. Deze uitbreidingen kunnen helpen om de resultaten van het onderzoek te verbeteren en om de tool verder te ontwikkelen.
Zo kan er gekeken worden naar het genereren van documentatie voor andere programmeertalen. Deze bachelorproef focust zich op Python, maar het is mogelijk om de tool uit te breiden naar andere programmeertalen. Aangezien een Large Language Model zoals GPT (**OpenAi**) ook getraind zijn op andere programmeertalen.
Ook kan er gekeken worden naar hoe projecten met syntax fouten of andere problemen gedocumenteerd kunnen worden. Dit is belangrijk omdat de tool nu enkel werkt op projecten die correcte syntax hebben. Deze fouten kunnen eruit gehaald worden door de code eerst door een linter te halen en dan pas de documentatie te genereren. De bekomen syntax fouten kunnen dan meegegeven worden aan een model om zo een bestand te genereren zonder syntax fouten.
Een andere uitbreiding is kijken naar hoe de documentatie geëvalueerd kan worden. Omdat dit nu slechts manueel gebeurt, op basis van gezond verstand. Er kan gekozen worden om enquêtes af te nemen bij programmeurs om zo de documentatie te evalueren. De evaluatie van de respondenten gaat echter slechts relatief zijn, omdat de respondenten beoordelen op basis van kennis van de programmeertaal. Of er kan gekeken worden naar hoe de documentatie van de tool vergeleken kan worden met de documentatie van de programmeur zelf. Hier is het belangrijk om te kijken naar de verschillen en overeenkomsten tussen de documentatie van de tool en de documentatie van de programmeur.
Een laatste voorbeeld van een uitbreiding is om te kijken naar hoe verschillende Large Language Models presteren op het genereren van documentatie. Zo kan er gekozen worden tussen modellen zoals GPT-4 (OpenAI, 2023), LLama 2 (Meta,

2024), Gemini (Google, 2024), ...
Sommige modellen hebben een groter context window dan andere modellen, zo zou er meer informatie meegegeven kunnen worden aan het model. En het zou mogelijk een beter resultaat kunnen geven.

7

Conclusie

[76-80]

A

Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

Documentatie van een Python project is belangrijk, maar het is een tijdrovende taak en het wordt vaak niet grondig gedaan. Deze bachelorproef aan de HoGent onderzoekt het automatisch genereren van documentatie voor python projecten met behulp van Large Language Modellen. Er wordt een tool ontwikkeld die de Python code en de relaties tussen de verschillende bestanden analyseert en op basis daarvan een overzichtelijke documentatie genereerd. Er wordt gekeken naar hoe de documentatie van Python functies gebruikt kunnen worden voor het maken van een gehele samenvatting van het project. Dit wordt gedaan op basis van huidige methoden om docstrings aan te maken en te gebruiken. Deze informatie kan dan gegeven worden aan de Large Language Modellen om een samenvatting te genereren.

Er worden verschillende Python projecten verzameld en geanalyseerd om te kijken hoe de documentatie gegenereerd kan worden. Dan worden er LLMs getraind op basis van deze projecten en wordt er gekeken naar hoe de documentatie gegenereerd kan worden. De gegenereerde documentatie kan dan vergeleken worden met de huidige documentatie van de projecten om dit te evalueren. Ook zal er gevraagd worden aan enkele programmeurs om de documentatie te evalueren.

Op basis van deze feedback kan het model gefinetuned worden. Er kan gekeken worden naar de mogelijke verbeteringspunten zodat er uiteindelijk een betere documentatie van het project ontstaat. Het resultaat is dat er een tool is die de documentatie van een Python project kan genereren. Dit resultaat maakt het mogelijk om de gegenereerde samenvatting van een Python project te lezen. De lezer kan dan stukken gebruiken uit het project of er verder mee aan de slag gaan.

A.1. Introductie

Documentatie is belangrijk wanneer er aanpassingen moeten gebeuren aan de code van een project. Ook moet iemand anders de code kunnen begrijpen zodat dit gebruikt kan worden binnen een ander project. Hoe kan geautomatiseerde documentatiegeneratie met behulp van Large Language Modellen (LLM) effectief worden toegepast om duidelijke en informatieve overzichten te produceren voor Python projecten?
Het is belangrijk dat de skills of de know-how van een project gedeeld kunnen worden met anderen. Door het toepassen van documentatie kan deze kennis gemakkelijk vergaard worden door andere geïnteresseerden. Het is dus belangrijk dat er aan documentatie gedaan wordt en dat deze up-to-date blijft.
Documentatie is iets dat veel tijd kost en waar vaak geen aandacht aan wordt besteed. Het gebruik ervan kan ervoor zorgen dat er geen dubbel werk gedaan moet worden. Een tool die dit proces kan versnellen / automatiseren zou een grote meerwaarde zijn. De tool bestaat uit een geautomatiseerde documentatie LLM die de project code analyseert en samenvat in een document. Dit geeft de werknemers de mogelijkheid om zich in te lezen in het project en erna zelf aanpassingen te maken of stukken code te gebruiken voor een ander project.
Het eindresultaat van deze bachelorproef is een Proof of Concept (PoC) van een geautomatiseerde tool die de project code analyseert en er documentatie van genereert. De gegenereerde documentatie laat het toe het project te begrijpen zonder er te veel tijd aan te besteden.

A.2. Literatuurstudie

Wat is documentatie binnen Python projecten en wat zijn de huidige tools? Voor de taal Python bestaan er al verschillende tools die documentatie genereren voor blokken code zoals pdoc (Gallant & Hils, 2023) en Sphinx (Sphinx Team, 2023). Met behulp van de Sphinx autodoc functie (Sphinx Team, 2023) kan een python functie omschreven worden in een docstring. Een docstring is een blok tekst dat de werking van een python functie omschrijft. Door deze beknopte blok tekst wordt er duidelijk wat de functie doet. Deze docstrings kunnen mogelijk gebruikt worden bij het maken van een document dat het project omschrijft.
Wat zijn Large Language Modellen (LLM)? Large Language Modellen zijn neurale netwerken die getraind worden op grote hoeveelheden tekst. Deze modellen kunnen tekst genereren op basis van een gegeven input. LLMs hebben een grote vooruitgang gekend in 2017 door de paper van Vaswani e.a. (2017). Hieruit kwam een nieuw neuraal netwerk, self-attention of transformers. Deze doen het beter dan de vorige neurale netwerken, Convolutional Neural Networks (CNN) en Recurrent Neural Networks (RNN). Door deze nieuwe neurale netwerken zijn er krachtige LLMs ontstaan zoals: GPT van OpenAi (Radford e.a., 2018) en BERT van Devlin e.a. (2019).

Hoe kunnen deze Large Language Modellen gebruikt worden om documentatie te genereren? Er kan een prompt geschreven worden die aan de LLMs gegeven wordt, er kan een nieuw LLM getraind worden op correcte samenvattingen van gehele Python projecten. Het gebruiken van docstrings of uitleg van verschillende Python functies kan gegeven worden aan een LLM. Ook kan er gekeken worden naar GitHub README.md bestanden. Dit zijn bestanden waarin de werking van een project kort wordt uitgelegd. Deze zijn echter niet altijd vlot te lezen. Volgens de studie Gao e.a. (2023) kan de tekst vereenvoudigd worden terwijl steeds de correcte betekenis kan behouden worden en dit aan de hand van een transfer learning model. Doordat het vereenvoudigen van een README bestand mogelijk is, kan dit mogelijk gebruikt worden voor de uiteindelijke documentatie die beoogd wordt in deze bachelorproef. Zo kan een redelijk complexe samenvatting vereenvoudigd worden naar de essentie ervan terwijl het steeds een bepaalde diepgang behoudt.
In 2023 werd het gebruiken van LLMs bij het automatisch documenteren van code met behulp van syntax bomen onderzocht door Procko en Collins (2023) voor C# en .NET programeertalen. Er kan gekeken worden hoe er te werk is gegaan en wat de conclusies waren. Er werd gebruik gemaakt van GPT-3.5 en een dotnet compiler Roslyn (“”Roslyn”, dotnet”, z.d.). Hieruit kan geconcludeerd worden dat door het gebruiken van syntax bomen de stochastische onzekerheid van GPT-3.5 een deel verholpen kan worden.
In de studie van McBurney en McMillan (2014) werd onderzocht hoe er automatisch documentatie gegenereerd kan worden voor Java code. Er werd specifiek gekeken hoe de methodes met elkaar verbonden waren en welke rol deze speelden binnen het project. Deze studie was een vervolg op het onderzoek van Sridhara e.a. (2010) in 2010. Het zoeken naar de mogelijke gelijkenissen tussen de automatische documentatie voor Java code en deze van Python code kan een begin zijn van deze bachelorproef.
Er is ook al onderzoek gedaan naar het automatisch genereren van documentatie van code blokken met behulp van een Neural Attention Model (NAM) (Iyer e.a., 2016). Dit onderzoek heeft gekeken naar het genereren van hoogstaande samenvattingen van source code. Het maakt gebruik van neurale netwerken die stukken C# code en SQL queries omzetten naar zinnen die de code omschrijven. Dit helpt bij het begrijpen van stukken code maar niet van een geheel project waar meerdere bestanden bij betrokken zijn.
De afgelopen jaren is het automatisch documenteren van source code grondig bestudeerd en onderzocht. In deze bachelorproef wordt er verder onderzocht hoe LLMs gebruikt kunnen worden bij het automatisch genereren van documentatie of samenvattingen van een geheel Python project. Het uiteindelijke doel is het automatisch genereren van een samenhangend geheel van verschillende bestanden van een Python project die samen een duidelijk overzicht geven van de werking van het project. Bij het automatisch genereren van documentatie kan er gebruik

gemaakt worden van LLMs en kan er gekeken worden hoe verschillende LLMs presteren tegenover elkaar.
De uitkomst is een samenvatting van het gehele project dat de lezer in staat stelt het project te gebruiken of aan te passen zonder de totale project code te ontleden.

A.3. Methodologie

Het onderzoek bedraagt zes verschillende fases. De eerste fase bedraagt het verder uitvoeren van de literatuurstudie om zo een betere kennis over het onderwerp te vergaren. Het verfijnen van de literatuurstudie zorgt ervoor dat er specifieke methoden gevonden worden die relevant zijn voor Python projecten. Ook dient de probleemdefinitie aangescherpt te worden door specifieke uitdagingen te identificeren. Hiervoor worden twee weken ingepland.
De volgende twee weken bedraagt het verzamelen van de dataset. Er worden Python projecten verzameld die varieren in complexiteit en grootte. We gaan opzoek naar open source python projecten op github, hiervan nemen we de python files als data en de README files als de gewenste uitkomst / target. Vervolgens wordt de data voorbereid zodat deze gebruikt kan worden door de verschillende gekozen LLMs.
In de derde fase wordt het model gekozen en wordt dit model getraind. Er kunnen verschillende LLMs gekozen worden zoals GPT-3.5, gemini, Code LLama (speciale LLM voor python code) of BERT. Er kan ook overwogen worden om een LLM te finetunen op python documentatie. De volgende stap houdt het opstellen van hoe de prestaties van de modellen vergeleken kunnen worden in. Dit kan pas nadat er een plan is opgesteld voor het trainen op de bekomen dataset. Deze fase bedraagt drie weken.
De vierde fase zal de implementatie en evaluatie bevatten. Er wordt een tool gemaakt waar een Python project aangegeven kan worden. De uitvoer van deze tool is een document met een samenvatting van het python project inclusief relaties tussen de verschillende bestanden. Deze uitvoer moet dan geevalueerd te worden op basis van de relevantie, begrijpelijkheid en de volledigheid. De resultaten kunnen vergeleken worden met de documentatie van bestaande tools. Ook kan er gevraagd worden aan enkele programmeurs de gerealiseerde documentatie te evalueren. De vierde fase neemt vier weken in beslag.
De voorlaatste fase bestaat uit het optimaliseren van het generatieproces en het finetunen van de uitkomst. De finetuning stelt de tool in staat om betere documentatie te genereren op basis van de bekomen feedback. Deze fase duurt één week.
De laatste fase van het onderzoek bestaat uit het analyseren van de kritische feedback en het afwerken van de bachelorproef. Het rapport wordt geschreven en de presentatie wordt voorbereid. Er kan nagedacht worden over wat er verder onderzocht kan worden na de bekomen conclusies.

A.4. Verwacht resultaat, conclusie

Het verwachte resultaat van deze bachelorproef is dat er een werkende tool gemaakt wordt dat een geheel Python project kan analyseren en er documentatie van kan genereren. Deze documentatie geeft dan een duidelijk overzicht van de werking van het project en de relaties tussen de verschillende bestanden.
Er kunnen verschillende conclusies getrokken worden uit het onderzoek. LLMs zijn ideale modellen om te gebruiken bij het genereren van documentatie. Het finetunen van een LLM op Python documentatie kan een grote meerwaarde zijn bij het genereren van documentatie. Hierdoor is het mogelijk dat de documentatie specifiek afgestemd is voor de verschillende noden van gebruikers. Iedere gebruiker kan de automatische documentatie aanpassen naar zijn eigen noden.
Een verdere conclusie is dat het gebruiken van de documentatie tool het begrijpen van een Python project makkelijker maakt. De kennis van een project kan gemakkelijk gedeeld worden met anderen en anderen kunnen deze documentatie begrijpen.

B

Bijlage

In deze bijlage worden de volledige code van de applicatie en de gebruikte prompts weergegeven.

B.1. Prompts

B.1.1. Function Prompt 1

Instructies voor het genereren van een docstring voor een functie versie 1.

```
'''For this Python function:
```python^^I
def is_prime(n):
if n in [2, 3]:
    return True
if (n == 1) or (n % 2 == 0):
    return False
r = 3
while r * r <= n:
    if n % r == 0:
        return False
    r += 2
return True
```
Leave out any imports, just return the function with the
    docstring and type hints.
The function, with docstring using the google docstring style
    and with type hints is:
```python^^I
def is_prime(n: int) -> bool:
```
```

```
    """
    Check if a number is prime.
    Args:
        n (int): The number to check.
    Returns:
        bool: True if the number is prime, False otherwise.
    """
    if n in [2, 3]:
        return True
    if (n == 1) or (n % 2 == 0):
        return False
    r = 3
    while r * r <= n:
        if n % r == 0:
            return False
        r += 2
    return True
    ```

    For this Python function:
    ```python^^I
    {code}
    '''
```

B.1.2. Function Prompt 2

Instructies voor het genereren van een docstring voor een functie versie 2.

```
'''
    The following Python function is a code snippit from a Python
        file.
    The following function lacs a docstring and type hints.
    Your task is to add a docstring and type hints to the function.
    You can't change the function's code, add any imports, or assume
        anything about the function's behavior or datatypes that is
        not clear from the code snippet itself.
    Below is a function that needs a docstring and type hints:
    ```python^^I
    def is_prime(n):
    if n in [2, 3]:
        return True
    if (n == 1) or (n % 2 == 0):
        return False
```
```

```
r = 3
while r * r <= n:
    if n % r == 0:
        return False
    r += 2
return True
```
```
The correct outcome should be the following Python code:
```python^^I
def is_prime(n: int) -> bool:
"""
Check if a number is prime.
Args:
    n (int): The number to check.
Returns:
    bool: True if the number is prime, False otherwise.
"""
if n in [2, 3]:
    return True
if (n == 1) or (n % 2 == 0):
    return False
r = 3
while r * r <= n:
    if n % r == 0:
        return False
    r += 2
return True
```


Now it's your turn to add a docstring and type hints to the
    following function:
```python^^I
{code}
```
'''
```

B.1.3. Function Prompt 3

Prompt versie 3 voor het genereren van een docstring voor een functie.

```
'''You are an AI documentation assistant, and your task is to
    generate docstrings and typehints based on the given code of
    a function, the function is a code snippet from a Python
    file.
Do your task with the least amount of assumptions, you can't add
    any imports, change the code, or assume anything about the
    function's behavior or datatypes that is not clear from the
    code snippet itself.
The purpose of the documentation is to help developers and
    beginners understand the function and specific usage of the
    code.

An example of your task is as follows:
The given code is:

```python^^I
def is_prime(n):
if n in [2, 3]:
    return True
if (n == 1) or (n % 2 == 0):
    return False
r = 3
while r * r <= n:
    if n % r == 0:
        return False
    r += 2
return True
```

The expected output of your task for the given code is:

```python^^I
def is_prime(n: int) -> bool:
"""
Check if a number is prime.

Args:
    n (int): The number to check.

Returns:
    bool: True if the number is prime, False otherwise.
```
```

```
    """

    if n in [2, 3]:
        return True
    if (n == 1) or (n % 2 == 0):
        return False
    r = 3
    while r * r <= n:
        if n % r == 0:
            return False
        r += 2
    return True
    ```

    Now it's your turn to generate the docstrings and typehints for
        the following function of a file with these imports:
    {imports}

    The content of the code is as follows:
    {code_content}
    '''
```

### B.1.4. Class Prompt 1

Prompt voor het genereren van een docstring voor een klasse.

```
    '''
    You are an AI documentation assistant, and your task is to
        generate docstrings and typehints based on the given code of
        a class, the class is a code snippet from a Python file.
    Do your task with the least amount of assumptions, you can't add
        any imports, change the code, or assume anything about the
        classes behavior or datatypes that is not clear from the
        code snippet itself.
    The purpose of the documentation is to help developers and
        beginners understand the function and specific usage of the
        code.

    An example of your task is as follows:
    The given code is:

    ```python
    class Circle:
```

```
    def __init__(self, radius: float) -> None:
        """
        Initialize the Circle object with a given radius.

        Args:
            radius (float): The radius of the circle.
        """
        self.radius = radius

    def calculate_area(self) -> float:
        """
        Calculate the area of the circle.

        Returns:
            float: The area of the circle.
        """
        return round(math.pi * self.radius ** 2, 2)

    def calculate_circumference(self) -> float:
        """
        Calculate the circumference of the circle.

        Returns:
            float: The circumference of the circle.
        """
        return round(2 * math.pi * self.radius, 2)
```

The expected output of your task for the given code is:

```python
class Circle:
    """
    A class representing a circle with methods to calculate its
        area and circumference.

    Attributes:
        radius (float): The radius of the circle.

    Methods:
```

```
            __init__: Initialize the Circle object with a given
                radius.
            calculate_area: Calculate the area of the circle.
            calculate_circumference: Calculate the circumference of
                the circle.
        """
```
```

    Now it's your turn to generate the docstrings and typehints for
        the following class of a file with these imports:
    {imports}

    The content of the code is as follows:
    {code_content}

    Only generate the class docstring

    '''
```

B.1.5. Samenvatting van een bestand

Prompt voor het genereren van een samenvatting van een bestand.

```
    '''
    You are an AI documentation assistant, and your task is to
        generate a summary of the given Python file.
    The summary should include the following information:
    - What the file does.
    - What classes are defined in the file.
    - What functions are defined in the file.
    - And a brief description of each class and function.
    - Include the file name at the beginning of the summary.

    You are going to generate the summary based on given function
        names, class names and their docstrings.

    Now it's your turn to generate the summary given the following
        code of the file: {filename}:

    {code_content}
    '''
```

B.1.6. Bestand zonder functies of klasses

Prompt voor het genereren van een samenvatting van een bestand zonder functies of klasses.

````
'''
You are an AI documentation assistant, and your task is to
    generate a summary of the given Python file based on the
    code content.
The summary should include the following information:
- What the file does.
- What is the purpose of the file.
- What is the main functionality of the file.
- What the output is
- What it does when executed.
- Include the file name at the beginning of the summary.

An example of the output of your task is as follows:
Given the following code content:

```python
from model import get_model
from train import train_top_layer, train_all_layers
if __name__ == '__main__':
    model = get_model()
    train_top_layer(model)
```


The expected output of your task for the given code is the
    summary of the file:

```python
"""
Summary of file: main.py

This file contains the main functionality for a Python
    application.
It imports the get_model function from the model module and the
    train_top_layer and train_all_layers functions from the
    train module.
When executed, it gets a model using the get_model function and
    trains the top layer of the model using the train_top_layer
    function.
````
````

```
    """
    ```

    You are going to generate the summary based on the given code
        content of the file with filename: {filename}.
    {code_content}
    '''
```

### B.1.7. Project samenvatting

Prompt voor het genereren van een samenvatting van een project.

```
    '''
    You are an AI documentation assistant, and your task is to
        generate a summary of the given Python project.
    The summary should include the following information:
    - What the project does.
    - What files are included in the project. And what each file
        does. What functions and classes are defined in each file.
    - A brief description of each class and function.
    - Include the project name at the beginning of the summary.

    You are going to generate the summary based on summaries of each
        file in the project.

    Now it's your turn to generate the summary given the following
        project structure:
    {project_name}

    With the following folder structure:
    {folder_structure}

    And the following summaries of each file:
    {summaries}
    '''
```

B.1.8. Project samenvatting per file

Prompt voor het genereren van een samenvatting van een project per bestand.

```
'''
    You are an AI documentation assistant, and your task is to
        generate a markdown summary of a file.
    For the following file summary:
```
```

```
    """
    Summary of file: crop_images.py

    This file contains the implementation of functions for cropping
        and padding images.

    Functions:
        crop_faces: Crop faces from an image using a specified
            bounding box.
        crop_image: Crop a specified region from an image.
        pad_img_to_fit_bbox: Pad an image to fit a specified bounding
            box.
    """

    The output should be:
    - **crop_images.py**:
        - Contains functions for cropping and padding images:
          - `crop_faces`: Crop faces from an image using the given
              bounding boxes.
          - `crop_image`: Crop a specific region from an image based
              on the provided coordinates.
          - `pad_img_to_fit_bbox`: Pad an image to fit the specified
              bounding box.

    You are going to generate the markdown summary for the file:
        {file} with the following summary:
    {summary}
    '''
```

B.2. Code

B.2.1. Vervangen van de code van een functie door de gegenereerde docstring. v1

```python
def replace_functions(self, functions):
    tree = self.tree
    for node in ast.walk(tree):
        if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef))
            and node.name in functions:
            new_func_def = ast.parse(functions[node.name]).body
            tree.body.insert(tree.body.index(node), new_func_def)
    self.tree = tree
```

B.2.2. Vervangen van de code van een functie door de gegenereerde docstring. v2

Versie 2 van de functie om de code van een functie te vervangen door de gegenereerde docstring.

```
def replace_functions(self, functions):
    tree = self.tree
    for node in ast.walk(tree):
        if isinstance(node, ast.ClassDef):
            for child_node in node.body:
                if isinstance(child_node, (ast.FunctionDef,
                    ast.AsyncFunctionDef)) and child_node.name in
                    functions:
                    new_func_def =
                        ast.parse(functions[child_node.name]).body[0]
                    new_func_def.body.extend(child_node.body)
                    idx = node.body.index(child_node)
                    node.body.insert(idx, new_func_def)
                    node.body.remove(child_node)
                    functions.pop(child_node.name)
        elif isinstance(node, (ast.FunctionDef,
            ast.AsyncFunctionDef)) and node.name in functions:
            new_func_def = ast.parse(functions[node.name]).body[0]
            new_func_def.body.extend(node.body)
            tree.body.insert(tree.body.index(node), new_func_def)
            tree.body.remove(node)
            functions.pop(node.name)
    self.tree = tree
```

B.2.3. Vervangen van de code van een functie door de gegenereerde docstring. v3

```
def _replace_functions(self, node, functions):
    if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)) and
        node.name in functions:
        new_func_def = ast.parse(functions[node.name]).body[0]
        new_func_def.body.extend(node.body)
        parent_node = self._get_parent_node(node)
        index = parent_node.body.index(node)
        parent_node.body.remove(node)
        parent_node.body.insert(index, new_func_def)
        functions.pop(node.name)
    for child_node in ast.iter_child_nodes(node):
```

```
        self._replace_functions(child_node, functions)
```

B.2.4. Genereren van de relaties tussen de verschillende bestanden

```
'''
    You are an AI documentation assistant, and your task is to
        generate a csv file containing the relations between the
        files in a Python project.

    For the given project structure and imports:
    The imports are as follows:
    '1_RPS_Game\\CMD_version\\main.py': 'import random',
        '2_PyPassword_Generator\\CMD_version\\main.py': 'import
        random', '3_Hangman_Game\\CMD_version\\images.py': '',
        '3_Hangman_Game\\CMD_version\\main.py': 'import
        requests\nimport random\nimport os\nfrom images import
        hangman_logo\nfrom images import stages',
        '4_Hangman_Game\\CMD_version\\stages.py': '',
        '4_Hangman_Game\\CMD_version\\images.py': 'import
        csv\nimport matplotlib',
        '4_Hangman_Game\\CMD_version\\main.py': 'import
        random\nimport os\nfrom images import stages\nfrom images
        import logo'

    The structure of the project is as follows:
    '.': ['LICENSE', 'README.md'], '1_RPS_Game': [],
        '1_RPS_Game\\CMD_version':
        ['1_RPS_Game\\CMD_version\\main.py'],
        '2_PyPassword_Generator': [],
        '2_PyPassword_Generator\\CMD_version':
        ['2_PyPassword_Generator\\CMD_version\\main.py'],
        '3_Hangman_Game': [], '3_Hangman_Game\\CMD_version':
        ['3_Hangman_Game\\CMD_version\\images.py',
        '3_Hangman_Game\\CMD_version\\main.py'], '4_Hangman_Game':
        [], '4_Hangman_Game\\CMD_version':
        ['4_Hangman_Game\\CMD_version\\stages.py',
        '4_Hangman_Game\\CMD_version\\images.py',
        '4_Hangman_Game\\CMD_version\\main.py']

    The expected output of your task is the following:
    ```csv
File_Path,File_Name,Folder_Path,Uses_File
```
```

```
1_RPS_Game\CMD_version\main.py, main.py,1_RPS_Game\CMD_version,[]
2_PyPassword_Generator\CMD_version\main.py,main.py,
    2_PyPassword_Generator\CMD_version,[]
3_Hangman_Game\CMD_version\images.py,images.py,3_Hangman_Game\CMD_version,[]
3_Hangman_Game\CMD_version\main.py,
    main.py,3_Hangman_Game\CMD_version,['3_Hangman_Game.CMD_version.images']
4_Hangman_Game\CMD_version\stages.py,stages.py,4_Hangman_Game\CMD_version,[]
4_Hangman_Game\CMD_version\images.py,images.py,4_Hangman_Game\CMD_version,[]
4_Hangman_Game\CMD_version\main.py,main.py,4_Hangman_Game\CMD_version,['4_Hangman_
    ```

    The Column "Uses File" should only contain the files where the
        file imports functions from.
    For example if the imports are:
    ```python
    Import csv
    Import matplotlib
    from images import open_image
    from stages import stage1
    ```
    The Column "Uses File" should contain the file
        '4_Hangman_Game\\CMD_version\\images.py' and
        '4_Hangman_Game\\CMD_version\\stages.py'
    Do your task given the following imports and structure of the
        project:

    The imports are as follows:
    {imports}

    And the structure of the project is as follows:
    {structure}

    THE OUTPUT SHOULD BE A SINGLE CSV FILE CONTAINING THE RELATIONS
        BETWEEN THE FILES IN THE PROJECT.
    '''
```

### B.2.5. Functies voor het samenvatting van een bestand

```python
    def document_file(self, file_path, outfolder_path):
        FDG = FileDocumenationGenerator(self.api_key,
            self.azure_endpoint, file_path, self.folder_path,
            outfolder_path)
```
```

```
        FDG.generate_file_documentation()
        return FDG

    def generate_file_summaries(self, python_files):
        for file in python_files:
            print("Documenting file: ", file)
            FDG = self.document_file(file,
                outfolder_path=self.outfolder)
            self.summaries[file] = FDG.get_summary()
            self.imports[file] = FDG.get_imports()
```

B.2.6. Generatie van een graph van de relaties tussen de bestanden

```
def generate_graph_html(self):
        print("Generating graph html")
        added_edges = set()
        df = pd.read_csv(os.path.join(self.outfolder,
            'graph_relations.csv'))
        net = Network(height="750px", width="100%",
            bgcolor="#222222", font_color="white")
        net = Network(directed =True)
        net.add_node("root", shape='star', label="")
        for index, row in df.iterrows():
            path = row['Folder_Path'].split("\\")
            # Add nodes for each folder in the path
            if len(path) > 1:
                for i in range(len(path)-1):
                    path_id = "_".join(path[:i+1])
                   net.add_node(path_id, label=path[i], shape='box')
                    next_path_id = "_".join(path[:i+2])
                    net.add_node("_".join(path[:i+2]),
                        label=path[i+1], shape='box')
                    edge = (path_id, next_path_id)
                    if edge not in added_edges:
                        net.add_edge(path_id, next_path_id)
                        added_edges.add(edge)
            elif len(path) == 1:
                   net.add_node(path[0], label=path[0], shape='box')

            # Add node for the file
            file_path = row['File_Path'].split("\\")
            file_id = "_".join(file_path)
```

```
            parent_folder_id = "_".join(path)
            net.add_node(file_id, label=file_path[-1])
            edge = (parent_folder_id, file_id)
            if edge not in added_edges:
                net.add_edge(parent_folder_id, file_id)
                added_edges.add(edge)

            # Add edges for the root node
            root_edge = ("root", path[0])
            if root_edge not in added_edges:
                net.add_edge("root", path[0])
                added_edges.add(root_edge)

        for index, row in df.iterrows():
            file_id = "_".join(row['File_Path'].split("\\"))
            # Add edges for the uses files
            uses = row['Uses_File'].strip("[]")
            if uses:
                uses = uses.split(";")
                for use_file in uses:
                    use_file_path = use_file.strip("'").split(".")
                    use_file_id = "_".join(use_file_path)+".py"
                    edge = (file_id, use_file_id)
                    if edge not in added_edges:
                        net.add_edge(file_id, use_file_id)
                        added_edges.add(edge)

        path = os.path.join(self.outfolder, 'graph.html')
        net.save_graph(path)
```

Bibliografie

"Roslyn", dotnet (computersoft.). (z.d.). https://github.com/dotnet/roslyn

Anthropic. (2023, mei 14). *Introducing Claude*. https://www.anthropic.com/news/introducing-claude

ArtificialAnalysis (Red.). (2024). *Comparison of Models: Quality, Performance Price Analysis*. https://artificialanalysis.ai/models

Bailey, C. (2024). *Type Hinting*. https://realpython.com/lessons/type-hinting/

Beelen, J. (2023, juni 22). *Hoe werken Large Language Models (LLM)?* https://www.linkedin.com/pulse/hoe-werken-large-language-models-llm-jeroen-beelen/?originalSubdomain=nl

Cacic, M. (2023, september 6). *Pre-training vs Fine-Tuning vs In-Context Learning of Large Language Models*. https://www.entrypointai.com/blog/pre-training-vs-fine-tuning-vs-in-context-learning-of-large-language-models/

Code Quality (Red.). (2024, april 4). *Code Documentation Best Practices and Standards: A Complete Guide*. https://blog.codacy.com/code-documentation

CodeCat.AI. (2024). *AI Docstring Generator*. https://www.codecat.ai/ai-docstring-generator

Das, S. (2024, januari 25). *Fine Tune Large Language Model (LLM) on a Custom Dataset with QLoRA*. https://dassum.medium.com/fine-tune-large-language-model-llm-on-a-custom-dataset-with-qlora-fb60abdeba07

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.

Doxygen. (2023, december 25). *Code Documentation Automated. Free, open source, cross-platform.* (Versie 1.10.0). https://www.doxygen.nl/

ElHousieny, R. (2023, november 19). *Demystifying Tokenization: Preparing Data for Large Language Models (LLMs)*. https://www.linkedin.com/pulse/demystifying-tokenization-preparing-data-large-models-rany-2nebc/

Gallant, A., & Hils, M. (2023). *pdoc: Auto-generate API documentation for Python projects* (Versie 14.1.0). https://pdoc.dev/

Gao, H., Treude, C., & Zahedi, M. (2023). Evaluating Transfer Learning for Simplifying GitHub READMEs. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1548–1560. https://doi.org/10.1145/3611643.3616291

GeeksforGeeks (Red.). (2023, augustus 7). *Python Docstrings*. https://www.geeksforgeeks.org/python-docstrings/

Google. (2024). *Welcome to the Gemini era*. https://deepmind.google/technologies/gemini/#introduction

Google Python Team. (2024, februari 12). *Google Python Style Guide*. https://google.github.io/styleguide/pyguide.html

Hashemi-Pour, C., & Lutkevich, B. (2024, februari). *BERT language model*. https://www.techtarget.com/searchenterpriseai/definition/BERT-language-model

Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., & Steinhardt, J. (2020). Measuring Massive Multitask Language Understanding. *CoRR*, *abs/2009.03300*. https://arxiv.org/abs/2009.03300

Hoque, M. (2023, april 30). *A Comprehensive Overview of Transformer-Based Models: Encoders, Decoders, and More*. https://medium.com/@minh.hoque/a-comprehensive-overview-of-transformer-based-models-encoders-decoders-and-more-e9bc0644a4e5

Iyer, S., Konstas, I., Cheung, A., & Zettlemoyer, L. (2016, augustus). Summarizing Source Code using a Neural Attention Model. In K. Erk & N. A. Smith (Red.), *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 2073–2083). Association for Computational Linguistics. https://doi.org/10.18653/v1/P16-1195

Lambert, N., Castricato, L., von Werra, L., & Havrilla, A. (2022, december 9). *Illustrating Reinforcement Learning from Human Feedback (RLHF)*. https://huggingface.co/blog/rlhf

McBurney, P. W., & McMillan, C. (2014). Automatic Documentation Generation via Source Code Summarization of Method Context. *Proceedings of the 22nd International Conference on Program Comprehension*, 279–290. https://doi.org/10.1145/2597008.2597149

Meta. (2024). *Llama 2: open source, free for research and commercial use*. https://llama.meta.com/

OpenAI. (2023). GPT-4 Technical Report.

OpenAi. (2024). *Models*. https://platform.openai.com/docs/models/overview

Peckham, S., Day, J., & hbr, D. (2024, januari 30). *Recommendations for LLM fine-tuning*. https://learn.microsoft.com/en-us/ai/playbook/technology-guidance/generative-ai/working-with-llms/fine-tuning-recommend?source=recommendations

Procko, T., & Collins, S. (2023). Automatic Code Documentation with Syntax Trees and GPT: Alleviating Software Development's Most Redundant Task. https://doi.org/10.2139/ssrn.4571367

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving Language Understanding by Generative Pre-Training. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., … Synnaeve, G. (2024). Code Llama: Open Foundation Models for Code.

Sphinx Team. (2023). *Sphinx* (Versie 7.2.6). https://www.sphinx-doc.org/

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., & Vijay-Shanker, K. (2010). Towards Automatically Generating Summary Comments for Java Methods. *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 43–52. https://doi.org/10.1145/1858996.1859006

Stöffelbauer, A. (2023, oktober 24). *How Large Language Models work*. https://medium.com/data-science-at-microsoft/how-large-language-models-work-91c362f5b78f

swimm.io (Red.). (2024). *Code documentation: benefits, challenges, and tips for success*. Swimm Team. https://swimm.io/learn/code-documentation/code-documentation-benefits-challenges-and-tips-for-success#:~:text=Code%20documentation%20is%20a%20collection,size%20of%20documentation%20can%20vary.

TeeTracker. (2023, augustus 24). *LLM fine-tuning step: Tokenizing*. https://teetracker.medium.com/llm-fine-tuning-step-tokenizing-caebb280cfc2

TIOBE (Red.). (2024, mei 1). *TIOBE Index for May 2024*. https://www.tiobe.com/tiobe-index/

Trofficus, M. (2023, oktober 27). *gpt4docstrings*. https://github.com/MichaelisTrofficus/gpt4docstrings

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need.

West Health Institute Revision. (2018). *Pyvis*. https://pyvis.readthedocs.io/en/latest/

What are the primary advantages of transformer models? (2023, november 6). https://aiml.com/what-are-the-main-advantages-of-the-transformer-models/