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

List of Listings

1

Inleiding

De inleiding moet de lezer net genoeg informatie verschaffen om het onderwerp te begrijpen en in te zien waarom de onderzoeksvraag de moeite waard is om te onderzoeken. In de inleiding ga je literatuurverwijzingen beperken, zodat de tekst vlot leesbaar blijft. Je kan de inleiding verder onderverdelen in secties als dit de tekst verduidelijkt. Zaken die aan bod kunnen komen in de inleiding (**Pollefliet2011**):

- context, achtergrond
- afbakenen van het onderwerp
- verantwoording van het onderwerp, methodologie
- probleemstelling
- onderzoeksdoelstelling
- onderzoeksvraag
- ...

1.1. Probleemstelling

Projecten worden vaak niet goed gedocumenteerd, dit kan leiden tot problemen in de toekomst. Wanneer een andere persoon de code van een ongedocumenteerd project wilt 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.

1.2. Onderzoeksvraag

Hoe kan geautomatiseerde documentatiegeneratie met behulp van Large Language Modellen (LLM) effectief worden toegepast om duidelijke en informatieve overzichten te produceren voor Python projecten?

1.3. Onderzoeksdoelstelling

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.

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 5, 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 zal de literatuurstudie besproken worden. 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 met 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.

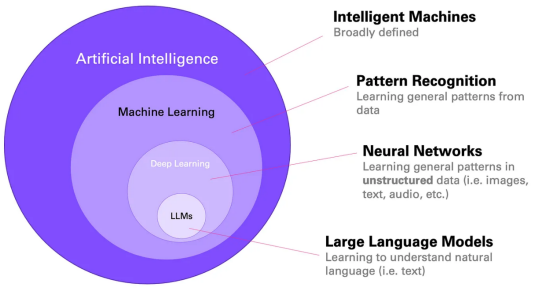
2.1. Wat is documentatie?

Voor dat er dieper op het onderwerp wordt ingegaan is het belangrijk dat er een duidelijk beeld is van 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. Dit kan op verschillende manieren gebeuren. Er kan gekozen worden om de documentatie te schrijven in de vorm van een handleiding, een wiki, een website of in de vorm van commentaar in de code. 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 bin-

nen 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. Of het gaat over het vastleggen van gedetaileerde uitleg over hoe iets werkt en hoe het onderhouden kan worden.
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. 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). Waaruit de werking van het project duidelijk wordt en de relatie tussen de verschillende bestanden en functies.

2.2. 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 is 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 LLMs speciaal getraind op Python code? Of kunnen LLMs gebruikt worden om documentatie te genereren? Dit zorgt voor een grondige basiskennis van LLMs.
Het veld waarin AI zich bevindt wordt vaak voorgesteld volgens figuur 2.1, met verschillende lagen (Stöffelbauer, 2023). Deze lagen zijn: Artificiële Intelligentie, Machine Learning, Deep Learning en Large Language Modellen. Voor dat er dieper op de LLMs wordt ingegaan is het belangrijk dat er een duidelijk beeld is van wat deze lagen juist inhouden.



Figuur (2.1)
Artificiële intelligentie in lagen (Stöffelbauer, 2023)

AI is een brede term, hiermee wordt vaak verwezen naar slimme machines. Machine Learning (ML) is een subveld van AI, waarin patronen worden herkend tus-

sen 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 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. LLMs 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 trainings periode dan vorige neurale netwerken ("What are the primary advantages of transformer models?", 2023).

2.2.1. Transformers en de architectuur van LLMs

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

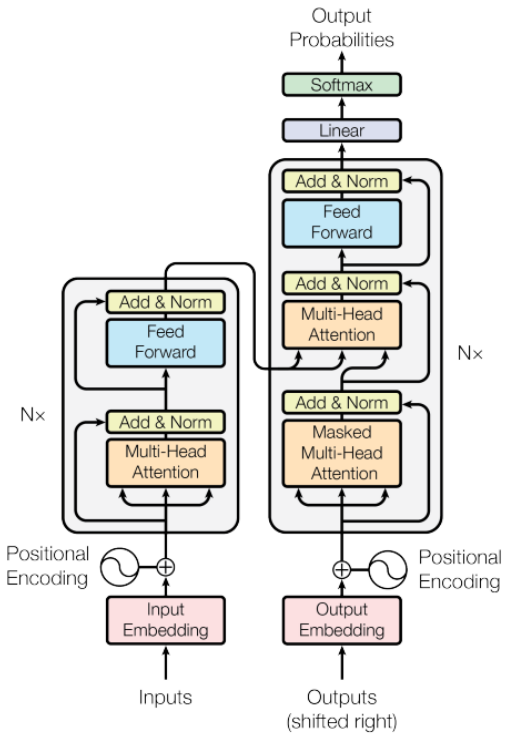
- Self-Attention
- Cross-Attention
- Masked Self-Attention

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 2.2.
Self-Attention duidt dynamisch gewichten toe aan verschillende elementen binnen de meegegeven sequentie, bijvoorbeeld woorden in een zin. Dit laat het mo-

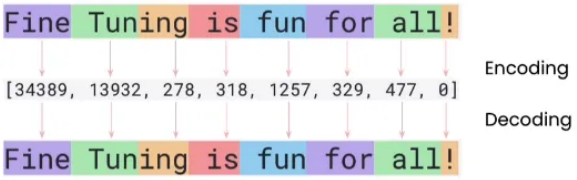
del toe om zich te concentreren op de meest relevante delen van de invoer, terwijl de invloed van minder cruciale delen wordt verminderd. De invoersequentie wordt eerst in drie verschillende vectoren omgezet: query, key en value. De Query vector stelt een specifiek token uit de invoersequentie voor, 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.

2.2.2. Trainen van LLMs

Het trainen van LLMs 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 de tekst omgezet in tokens die het model kan verwerken. 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 2.2.1, maar het is nog niet instaat om vragen of instructies te begrijpen.
De volgende fase is het model te trainen op een dataset met instructies en het



Figuur (2.2)
Transformer - model architectuur (Vaswani e.a., 2017)



Figuur (2.3)
Simplified tokenization van tekst (TeeTracker, 2023)

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. Het Fine-Tunen van het model kan gebeuren op een specifieke dataset, zoals Python code of medische data. Dit zorgt ervoor dat het model extra kennis heeft over het gekozen onderwerp.

2.2.3. Bestaande LLMs

Momenteel zijn er verschillende LLMs die gebruikt worden voor verschillende taken. Deze LLMs zijn getraind op verschillende data en hebben verschillende architecturen. Het is belangrijk dat er een duidelijk beeld is van de verschillende LLMs en hun mogelijkheden. Zodat er een goede keuze gemaakt kan worden voor het genereren van documentatie.

Eén van de grote spelers in de wereld van LLMs is OpenAI. OpenAI heeft verschillende LLMs ontwikkeld gaande van GPT (Radford e.a., 2018) tot GPT-4 (OpenAI, 2023). GPT-4 is een LLM die OpenAI heeft ontwikkeld, 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 LLMs 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 LLMs 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). Het bestaat 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. Dit is aanzienlijk meer dan de 128 duizend tokens van GPT-4.

Een derde speler in de wereld van LLMs is Meta. Meta heeft verschillende LLMs ontwikkeld onder de naam LLama 2 (Meta, 2024). De LLama 2 familie bestaat uit verschillende LLMs 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 LLMs open source zijn en dus voor iedereen toegankelijk zijn.

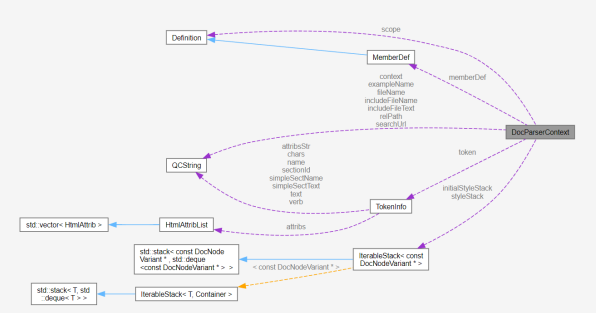
Antropic heeft ook een LLM ontwikkeld genaamd Claude (Anthropic, 2023). Claude's capaciteiten zijn code generatie, het verstaan van meerdere talen, beelden analyseren en kan geavanceerde redeneringen geven. Er bestaan 3 versies van Claude: Haiku, Sonnet en Opus. Haiku een lichte versie van Claude, Sonnet 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 LLMs 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 LLMs en de doeleinden van de documentatie. Het is mogelijk dat er meerdere LLMs getest moeten worden om de beste keuze te maken.

2.3. Bestaande documentatie tools

Voor er gekeken wordt naar hoe LLMs 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 programmeertalentalen.
In de paper van Sridhara e.a. (2010) werd er een tool ontwikkel die natuurlijke taal genereert op basis van JAVA code. Het selecteert eerst de relevante code en genereert dan een samenvatting van de code, volgens enkele programmeurs die de tool getest hebben was de samenvatting correct en volledig.
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 2.4. Zo wordt er een duidelijk beeld verkregen van de structuur van het project.



Figuur (2.4)
Voorbeeld diagram van Doxygen (Doxygen, 2023)

Docstrings is een vorm van commentaar in de code die gebruikt wordt om de werking van de code uit te leggen. Deze commentaren worden gebruikt om de werking van een module, functie, klasse of methode uit te leggen. Dit kan automatisch gegenereerd worden met tools zoals CodeCat (CodeCat.AI, 2024) voor JavaScript. 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 (2.5)
Voorbeeld uitkomst van de tool van Trofficus (2023)

van de tool.
De tool van Trofficus (2023) doet dit voor Python, 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 LLMs. Het nader bekijken van deze tool kan een meerwaarde zijn voor deze bachelorproef.
Zo gebruikt het 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 2.5.
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 en 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 samen gevat worden aan de hand van de docstrings van de verschillende python files. Eer een Python project gedocumenteerd kan worden met Sphinx dienen alle bestanden aangevuld te worden met docstrings, dit gebeurt echter niet automatisch.
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.

2.4. LLM voor documentatie

Nu er geweten is hoe een LLM werkt en wat het doet. Wat enkele bekende LLMs zijn en wat hun mogelijkheden zijn. En wat enkele bestaande documentatie tools zijn. Is het belangrijk om te kijken naar hoe LLMs gebruikt kunnen worden voor het genereren van documentatie. Dit kan staps gewijs gebeuren, eerst kunnen de verschillende delen van het project meegegeven worden aan de LLM. 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 achteraf één samenvattend document gemaakt worden van het gehele project.
Ook kan er gevraagd worden aan de LLM om de relatie tussen de verschillende delen van het project te beschrijven in de samenvattingen. Zo kan er een duidelijk beeld verkregen worden van de structuur van het project en hoe de verschillende delen van het project samenwerken. 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 dat 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

Het onderzoek is in vier 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 LLMs 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.
In de derde fase 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 en 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.

4

Bestand documentatie

4.1. Inleiding

In dit hoofdstuk wordt er gekeken naar de documentatie van een Python bestand. Er wordt gekeken hoe de verschillende delen van een python bestand geëxtraheerd kunnen worden en hoe deze gebruikt kunnen worden om documentatie te genereren. Hoe docstrings gegenereerd kunnen worden voor functies en klasses in een Python bestand. En hoe deze docstrings gebruikt kunnen worden 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.

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. En hoe de verschillende functies en klasses in een bestand geïdentificeerd en geëxtraheerd kunnen worden. Uit de literatuurstudie is gebleken dat dit kan gebeuren 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.
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 klasses geïdentificeerd worden.

```
        def get_functions(self):
            functions = {}
            for node in ast.walk(self.tree):
                if isinstance(node, ast.FunctionDef) or isinstance(node, ast.Async
                    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.

```
    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.

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

Docstrings zijn strings die aan het begin van een functie of klasse geplaatst worden. Deze strings worden gebruikt om de functie of klasse te documenteren. 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. Prompting

Om de docstrings te genereren, wordt er gebruik gemaakt van een GPT-3.5 model. Er wordt een prompt gegeven aan het model met duidelijke instructies over wat er verwacht wordt. Er werden verschillende prompts getest om te kijken welke het beste resultaat gaven. Er zijn prompts gemaakt voor het genereren van docstrings voor functies en klasses.

4.4.1. Prompt engineering voor functies

In 4.4.1 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 4.4.2. 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 4.4.3. 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. En omdat het model foute veronderstellingen maakte ook al stond in de instructies dat het dit niet mocht doen.
Omdat het model de code van de functie sporadisch aanpaste, werd er een nieuw prompt gemaakt. In dit 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.

4.4.2. Prompt engineering voor klasses

Het prompt engineering process voor klasses liep gelijkaardig aan 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 klasses.
De verschillende prompt versies 1-4 voor klasses zijn identiek aan die van functies, maar dan met de code van een klasse in plaats van een functie. Voor klasses werd er uiteindelijk gekozen om de code van de klasse niet in de prompt te zetten. Maar om de docstring van de klasses 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 4.4.4. Het voorbeeld van de klasse in het prompt is een klasse met daarin

````
'''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 hi
The function, with docstring using the google docstring style and with type hi
```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}
'''
````

Listing 4.4.1: Prompt v1 voor het genereren van een docstring voor een functie.
````

```
'''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
    Below is a function that needs a docstring and type hints:
'''
```

Listing 4.4.2: Prompt v2 instructies voor het genereren van een docstring voor een functie.

enkele functies met een docstring.

4.4.3. 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 klasses. 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 4.4.5.

4.5. Toevoegen van gegenereerde docstrings

De gegenereerde docstrings worden vervolgens toegevoegd aan de code van de functies en klasses. 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 4.4.1. 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 behouden 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 klasses is te zien in 4.5.1. 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 klasses in de AST. Waarvan één met docstring en één zonder. Dit werd opgelost door de oude code te verwijderen uit de

```
'''You are an AI documentation assistant, and your task is to generate docstri
Do your task with the least amount of assumptions, you can't add any imports,
The purpose of the documentation is to help developers and beginners understan

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
```
```

````
'''
You are an AI documentation assistant, and your task is to generate docstrings
Do your task with the least amount of assumptions, you can't add any imports,
The purpose of the documentation is to help developers and beginners understan

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 circu

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

```
'''
You are an AI documentation assistant, and your task is to generate a summary
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 nam

Now it's your turn to generate the summary given the following code of the fil

{code_content}
'''
```

Listing 4.4.5: Prompt voor het genereren van een samenvatting van een Python bestand.

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

Listing 4.5.1: Vervangen van de code van een functie 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.AsyncFunctionD
                        new_func_def = ast.parse(functions[child_node.name]).body[
                        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 nod
                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
```

Listing 4.5.2: Vervangen van de code van een functie door de gegenereerde docstring.

AST 4.5.2. 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 klasses. Hierdoor moest de code opnieuw aangepast worden omdat de gegenereerde docstrings niet altijd correct toegevoegd werden. Doordat er met AST gewerkt wordt, kon de code van versie 2 niet aan de parent node van nested functies. Dit werd opgelost door het vervangen recursief te laten gebeuren, een betere oplossing dan het gebruiken van if else statements 4.5.3.
Door de code recursief te laten lopen, kan de code van nested functies en klasses 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 vervangen worden.

```
    def _replace_functions(self, node, functions):
        if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)) and node.name
            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 4.5.3: Vervangen van de code van een functie door de gegenereerde docstring.

4.6. 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 klasses 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 klasses die in het bestand voorkomen. Per functie en per klasse een korte beschrijving te staan van wat deze doen.
Deze samenvatting wordt gegenereerd door het model de gegenereerde docstrings van de functies en klasses mee te geven in een prompt, zoals het prompt 4.4.5. 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

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.

Bibliografie

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

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

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/

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

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

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.

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

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.

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