T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

GÜNCEL YAZILIM SÜREÇLERİNİN YAPAY ZEKA YAKLAŞIMLARI İLE İYİLEŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

Mustafa Alp Eren KILIÇ

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Bilim Dalı

AĞUSTOS 2024

T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

GÜNCEL YAZILIM SÜREÇLERİNİN YAPAY ZEKA YAKLAŞIMLARI İLE İYİLEŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

Mustafa Alp Eren KILIÇ

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Bilim Dalı

Tez Danışmanı: Dr. Öğr. Üyesi M. Fatih ADAK

AĞUSTOS 2024

Mustafa Alp Eren Kılıç tarafından hazırlanan "Güncel Yazılım Süreçlerinin Yapay Zeka Yaklaşımları İle İyileştirilmesi" adlı tez çalışması 15.08.2024 tarihinde aşağıdaki jüri tarafından oy birliği ile Sakarya Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı **Bilgisayar Mühendisliği** Bilim Dalı'nda Yüksek Lisans tezi olarak kabul edilmiştir.

Tez Jürisi

Jüri Başkanı :

Jüri Üyesi :

Jüri Üyesi :

ETİK İLKE VE KURALLARA UYGUNLUK BEYANNAMESİ

Sakarya Üniversitesi Fen Bilimleri Enstitüsü Lisansüstü Eğitim-Öğretim Yönetmeliğine ve Yükseköğretim Kurumları Bilimsel Araştırma ve Yayın Etiği Yönergesine uygun olarak hazırlamış olduğum "GÜNCEL YAZILIM SÜREÇLERİNİN YAPAY ZEKA YAKLAŞIMLARI İLE İYİLEŞTİRİLMESİ" başlıklı tezin bana ait, özgün bir çalışma olduğunu; çalışmamın tüm aşamalarında yukarıda belirtilen yönetmelik ve yönergeye uygun davrandığımı, tezin içerdiği yenilik ve sonuçları başka bir yerden almadığımı, tezde kullandığım eserleri usulüne göre kaynak olarak gösterdiğimi, bu tezi başka bir bilim kuruluna akademik amaç ve unvan almak amacıyla vermediğimi ve 20.04.2016 tarihli Resmi Gazete'de yayımlanan Lisansüstü Eğitim ve Öğretim Yönetmeliğinin 9/2 ve 22/2 maddeleri gereğince Sakarya Üniversitesi'nin abonesi olduğu intihal yazılım programı kullanılarak Enstitü tarafından belirlenmiş ölçütlere uygun rapor alındığını, çalışmamla ilgili yaptığım bu beyana aykırı bir durumun ortaya çıkması halinde doğabilecek her türlü hukuki sorumluluğu kabul ettiğimi beyan ederim.

(06/09/2024).

Mustafa Alp Eren KILIÇ

TEŞEKKÜR

Yüksek lisans eğitimim boyunca bilgi ve deneyimlerinden yararlandığım, araştırma konusunun belirlenmesinden, uygulanacak yönteme kadar tez çalışmasının tüm aşamalarında yardımını esirgemeyen değerli danışman hocam Dr. Öğr. Üyesi Muhammed Fatih ADAK'a teşekkürlerimi sunarım.

Bu tez, Sakarya Üniversitesi Bilimsel Araştırma Projeleri Komisyonu (BAPK) tarafından 2024-26-62-2 proje numarası ile desteklenmiştir.

Mustafa Alp Eren KILIÇ

İÇİNDEKİLER

Sayfa

KISALTMALAR

API : Application Programming Interface
CCN : Cyclomatic Complexity Number
CI/CD : Continuous Integration / Continuous Delivery
ERD : Entity Relationship Diagram
GPT : Generative Pre-Trained Transformer
HTML : Hyper Text Markup Language
JDK : Java Development Kit
KNN : K-Nearest Neighbor
LOC : Lines of Code
LSTM : Long Short-Term Memory
NLOC : Lines of Code without Comments
ORM : Object Relational Mapper
RNN : Recurrent Neural Network
SVM : Support Vector Machine
URL : Uniform Resource Loader
VAE : Variational Autoencoder
YSY : Yorum Sapma Yüzdesi

TABLO LİSTESİ

Sayfa

ŞEKİL LİSTESİ

Sayfa

GÜNCEL YAZILIM SÜREÇLERİNİN YAPAY ZEKA YAKLAŞIMLARI İLE İYİLEŞTİRİLMESİ

ÖZET

Kaynak kodun anlaşılabilirliği ve bakımı, yazılım geliştirme sürecinin kritik bir yönünü oluşturur. Kaynak kod içerisine eklenen yorum satırları, kodun anlaşılabilirliğini ve sürdürülebilirliğini artırmada önemli bir rol oynar. Ancak, kaynak kodu açıklayan kapsamlı yorumların yazılması, yoğun emek gerektiren, hata yapmaya eğilimli ve tutarsızlıklarla dolu bir süreç olabilir. Bu çalışma, kaynak kodu özetleme ve yorum oluşturma süreçlerini otomatikleştirmek için Doğal Dil İşleme (NLP) tekniklerinin etkinliğini araştırmayı amaçlamaktadır. Ayrıca en son teknolojiye sahip Doğal Dil İşleme modellerinden yararlanan bir yaklaşımla, kaynak kod işlevlerinin özetlenmesi ve otomatik olarak açıklayıcı yorumlar oluşturulması hedeflenmektedir. Bu bağlamda, kaynak kodun anlaşılabilirliğini arttırmak amacıyla oluşturulan yorumların yeterliliğini değerlendiren yeni bir matematiksel değerlendirme indeksi önerilmektedir. Bu çalışmada, bazı popüler GitHub depoları analiz edilerek Java projeleri incelenmiş ve Doğal Dil İşleme yaklaşımları kullanılarak javadoc dokümantasyonuna sahip olmayan Java metotları için javadoc dokümantasyonu üretilmiştir. Geliştirilen indeks yardımıyla, analiz edilen GitHub deposunun önceki ve sonraki durumları değerlendirilmiş, iyileşme olup olmadığı ve iyileşme varsa bunun derecesi tespit edilerek yorumlanmıştır. Ayrıca, önerilen indeks, literatürde bulunan farklı indeksler ile karşılaştırılmıştır. Aralarındaki temel farklar ve birbirleriyle uyumları analiz edilmiştir.

Döngüsel karmaşıklık (cyclomatic complexity) değeri, yüksek olan yazılım projelerinin, karmaşıklıklarının doğası gereği, kapsamlı bir şekilde açıklanması ve belgelendirilmesi büyük önem arz etmektedir. Bu tür projeler için daha uzun ve detaylı yorumlar üretilmesi gereksede, aşırı ve gereksiz yorumların yazılması yazılım projesinin kalitesini olumsuz etkileyebilir. Dolayısıyla, dengeli ve optimal bir yorum üretimi sağlamak amacıyla, bu çalışma kapsamında üretilmesi gereken yorum miktarını belirleyen yeni bir teknik geliştirilmiştir. Bu teknik sayesinde, javadoc dokümantasyonu üretilecek her bir Java metodu için, uygun miktarda ve yeterli detayda javadoc dokümantasyonu oluşturulması amaçlanmıştır. Böylece her bir metodun gereksinimlerine uygun miktarda dokümantasyon oluşturulmasına çalışılarak, yazılımın anlaşılabilirliği ve bakımının kolaylaştırılması hedeflenmektedir.

Bu çalışmada dört farklı GitHub deposu detaylı bir şekilde incelenmiştir. İnceleme sürecinde, her bir GitHub deposunun javadoc üretimi öncesi ve sonrası durumları karşılaştırılmıştır. Yapılan analizler sonucunda, her bir depoda %20 ile %60 arasında değişen oranlarda iyileşme sağlandığı gözlemlenmiştir. Elde edilen bu olumlu sonuçlar, otomatik yorum üretme yaklaşımlarının etkili bir şekilde uygulanabilirliğini ortaya koymakta ve bu tekniklerin yakın gelecekte çok daha başarılı ve yaygın bir şekilde kullanılabileceğini göstermektedir. Bu bulgular, yazılım projelerinin anlaşılabilirliğini ve sürdürülebilirliğini artırma potansiyeli taşımaktadır ve Doğal Dil İşleme tekniklerinin yazılım dokümantasyonu alanında sağladığı katkıları

vurgulamaktadır. Çalışmanın sonuçları, yazılım mühendisliği alanında daha ileri düzeyde otomasyonun mümkün olabileceğini ve bu tür yaklaşımların, yazılım geliştirme süreçlerini önemli ölçüde iyileştirebileceğini göstermektedir.

IMPROVING CURRENT SOFTWARE PROCESSES WITH ARTIFICIAL INTELLIGENCE APROACHES

SUMMARY

The comprehensibility and maintainability of source code constitute a critical aspect of the software development process. Comment lines embedded within source code play a significant role in enhancing code comprehensibility and sustainability. However, the writing of comprehensive comments that elucidate source code can be a labor-intensive process prone to errors and inconsistencies. This study aims to investigate the effectiveness of Natural Language Processing (NLP) techniques in automating the processes of source code summarization and comment generation. Furthermore, this study aims to leverage state-of-the-art Natural Language Processing (NLP) models to summarize source code functionalities and generate automatically informative comments. In this context, a novel mathematical evaluation index is proposed to assess the adequacy of comments generated with the aim of enhancing source code comprehensibility. In this study, Java projects were examined by analyzing various popular GitHub repositories and Javadoc documentation was generated for Java methods that lacked native Javadoc documentation using Natural Language Processing (NLP) approaches. By utilizing the developed index, the pre- and post-analysis states of the examined GitHub repository were evaluated, and the presence or absence of improvement was determined and interpreted along with the degree of improvement, if applicable. Furthermore, the proposed index is compared with different indexes available in the literature. The main differences between them and their compatibility with each other are analyzed.

The success of software projects is not solely determined by the functionality of the written code but is also closely intertwined with the implementation of high-quality comment lines that facilitate code comprehension, maintenance, and further development. Well-crafted comment lines enhance code comprehension, expedite error detection, and facilitate teamwork. Adequate comment writing is a crucial criterion for software projects.

Inadequate comment writing in a software project hinders project maintenance and poses obstacles to future development. In the absence of comment lines elucidating a code's purpose, functionality, and potential error points, comprehending and modifying the code becomes exceedingly challenging. This situation threatens the long-term success of the software project.

Writing Javadoc in a Java project is a prerequisite for creating successful project documentation. Javadoc documentation written for classes, interfaces, methods, fields, and other components in a Java project enhances the development process by explaining the responsibilities and functionalities of these components. This facilitates software developers' understanding, extension, and modification of the code, thereby improving the overall quality of the development process. This will also positively contribute to the comprehensibility and sustainability of the software project.

Well-written comments facilitate code maintenance, accelerate error detection, and, if working in a team, enable effective collaboration among team members. Therefore, in projects involving teamwork, measures should be taken to improve the quality of comments, and the team should be encouraged to adhere to commenting standards. Automating the comment generation process will further facilitate these efforts. By automating the comment generation process, developers will be able to focus solely on writing code, thereby saving time. Additionally, manually written comments are prone to errors, and in situations where existing code needs to be modified for various reasons, it is very likely that the corresponding explanatory comments will be overlooked and not updated. This situation can be even worse than having no comments at all.

Due to the inherent complexity of their nature, software projects with high cyclomatic complexity values require comprehensive explanation and documentation. While longer and more detailed comments are necessary for such projects, excessive and unnecessary commenting can negatively impact the quality of the software project. Therefore, to achieve balanced and optimal comment generation, this study introduces a novel technique that determines the appropriate amount of comments to be produced. By employing this technique, the goal is to generate Javadoc documentation that is both appropriate in quantity and adequate in detail for each Java method that requires javadoc documentation. By striving to generate documentation that is tailored to the specific needs of each method, the goal is to enhance software comprehensibility and facilitate maintenance.

A significant contribution of this study is the proposal of the YSY (Comment Deviation Percentage) index for evaluating the adequacy of comment lines in a Java project. The YSY index was developed to mathematically measure whether the amount of comment lines in a project falls into the categories of 'low', 'adequate', or 'high'. By evaluating the comment percentage of a Java project composed of multiple files, this index enables a quantitative assessment of the project's comment line adequacy. The YSY index makes it possible to evaluate Java projects mathematically. In addition to facilitating the comparison of different Java projects in terms of comment percentage, this index can also be used to analyze the state of a project at different points in time. Therefore, it is possible to track how the adequacy of comment lines in projects changes over time. In summary, the YSY index is a valuable tool developed to enhance the documentation quality and sustainability of software projects. This index is expected to contribute to a better understanding and management of software projects, both internally and in comparison, with other projects.

In the scope of this thesis, a web application utilizing the PostgreSQL database and the OpenAI API to perform the required computations has been developed using Python's Django framework. Bootstrap has been utilized for the application design. A total of 5 interconnected tables have been used in the PostgreSQL database. These tables are as follows: "repository table", "java file table", "java class table", "method table", and "comment table”. There are one-to-many relationships between the "repository table" and the "java file table", between the "java file table" and the "java class table", between the "java class table" and the "method table", and between the "method table" and the "comment table". The “comment table” present here stores comments that are not documented in Javadoc.

The web application clones a repository from a specified GitHub URL. Subsequently, the repository is divided into its files, classes, and methods for analysis. After the

analysis is completed, a request is sent to the ChatGPT API for Javadoc generation. The Javadoc comments and other results generated through this request are stored in the PostgreSQL database. The results are shown in the web application interface.

The measurement method being followed is as follows: The YSY index of the Java project to be analyzed is first measured in its original state. Then, the necessary Javadoc documentation is generated, and the YSY index is recalculated. Subsequently, the difference between the old and new YSY indexes was evaluated. This method was applied to selected Java project repositories on GitHub. For each repository, the YSY index was first calculated in its current state, and then the index was recalculated with the values for the cases with Javadoc documentation added. The data obtained at the end of this process revealed the difference between the old and new YSY indexes.

In this study, four different GitHub repositories were examined in detail. During the examination process, the pre- and post-Javadoc generation states of each GitHub repository were compared. The analyses revealed improvements ranging from 20% to 60% in each repository. These positive results demonstrate the effective applicability of automatic comment generation approaches and suggest that these techniques can be employed with much greater success and widespread adoption soon. These findings hold the potential to enhance the comprehensibility and sustainability of software projects, highlighting the contributions of Natural Language Processing techniques in the realm of software documentation. The study's findings suggest that further automation in software engineering is feasible and that such approaches can significantly enhance software development processes.

1. GİRİŞ

Yazılım geliştirme süreçlerinde, kaynak kod içerisinde yazılan yorum satırları, kodun anlaşılabilirliğini arttırarak geliştirme ve bakım aşamalarında kritik bir rol oynar. Günümüzde yazılım geliştirme süreçleri, geleneksel yazılım geliştirme süreçlerine kıyasla, giderek karmaşık hale gelmekte ve büyük ölçekli projelerde yazılım mühendisleri arasındaki iletişim zorlukları artmaktadır. Özellikle büyük ve karmaşık kod satırları ile uğraşan ekipler, kodun amacını ve işlevselliğini anlamak için uzun süreler harcayabilir. Kodun anlaşılabilirliği ve bakımı, yazılımın çevikliğini arttırmak açısından önemli bir faktördür. Bu sebeplerden ötürü doğal dil işleme teknikleri, metin madenciliği ve yapay zekâ modellerinin, programlama dillerinde yazılmış kodu analiz etme, çözümleme ve yorumlama konularında etkili bir şekilde kullanılmasının gerekliliği anlaşılmıştır. Bu alanda yapılan çalışmaların temel amacı, yazılım geliştiricilerinin, kodun ne yaptığını daha hızlı ve etkili bir şekilde kavrayabilmelerini sağlamaktır.

Önceki paragrafta belirtilen doğal dil işleme teknikleri, günümüzde çok hızlı gelişmekte olan bir alan olmakla birlikte kullanım alanları oldukça genişlemiştir. Bu teknikler birçok farklı alana uygulanmaktadır. Çalışmamızda, bu alanda oluşturulmuş bilgi birikiminin, programlama dilleriyle yazılmış kaynak kodları içerisine eklenmiş yorum satırlarının analizi ve yorum satırı olmayan kaynak kodlar için, yorum üretimi alanında nasıl kullanılabileceği üzerine yoğunlaşılmıştır.

Yazılım geliştirme süreçlerinde, kaynak kod içerisine yazılan yorum satırları büyük bir önem taşımaktadır. Yorum satırları vasıtasıyla, kaynak kod dosyalarının tamamı, belirli kod parçaları veya tek bir kod satırının, hangi işlevi nasıl yaptığıyla ilgili bilgi edinilebilmesi, yazılımın geliştirilmesi ve bakımı aşamalarında istenen bir durumdur. (Xia vd., 2018)'deki çalışmada, yazılım geliştiricilerin zamanlarının, ortalama olarak %58'ini kodu anlama aktiviteleriyle geçirdiği saptanmıştır. Kaynak kod içerisine, anlamlı yorum satırlarının yazılması kodun anlaşılabilirliğini arttırmakla birlikte anlaşılma süresini de kısaltmaktadır. Nitekim (Mi vd., 2023)'de yapılan deneysel çalışmada yorum satırlarının kod okunabilirliğine olumlu katkıda bulunduğu

saptanmıştır. Hatta, kod içerisine yazılan yorum satırları vasıtasıyla kodun anlaşılabilirliğinin arttırılmasının yanı sıra bekleyen hata düzeltmeleri, diğer kod satırlarına verilen referanslar ve teknik borçlanma gibi konuların belirtilerek yazılım geliştiriciler arasında bir iletişim sağlanabilmesi olasıdır (Niazi vd., 2023). Fakat birçok yazılım geliştirme projesinde yorum satırları yazmak çeşitli sebeplerden ötürü ihmal edilebilmektedir. Geliştirilmekte olan projenin kısıtlı bir süre içerisinde bitirilmek zorunda olması veya yazılımcının bu konuya gereken önemi vermemesi bu sebeplerden bazıları olarak gösterilebilir. Yorum satırlarının otomatik olarak üretilmesi bu sorunun çözülmesini sağlayacaktır. Literatürde otomatik yorum satırı üretimi ve kaynak kod özetleme ile ilgili birçok çalışma bulunmaktadır (Song vd., 2023). Bu çalışmalardan Literatür Araştırması bölümünde bahsedilecektir.

Bazı akıllı algoritmalar, çeşitli ve etkili test vakaları üreterek testlerin sağlamlığını artırır (Rashid & Adak, 2021). Otomatik yorum oluşturma ise net ve güncel dokümantasyon sağlayarak kod okunabilirliğini ve sürdürülebilirliğini artırır. Makine öğrenmesi ve gelişmiş analitik yöntemlerin ortaya çıkması, çok yönlü indekslerin oluşturulması için yeni yollar sunmaktadır. Doğal dil işleme ve statik kod analizi gibi tekniklerden yararlanılarak, kod semantiğini, amacını ve anlaşılabilirliğini analiz edebilen gelişmiş modeller geliştirmek mümkündür. Bu modeller, kaynak kod ile yorumların ve dokümantasyonun yeterliliğini değerlendirerek yazılımın sürdürülebilirliğine dair bütünsel bir görünüm sağlayabilir. Ayrıca, bu modellere dayalı otomatik araçlar gerçek zamanlı bildirim ve öneriler sunabilir. Doğal dil işleme ve statik kod analizi gibi tekniklerin dahil edilmesiyle, kod semantiğini, amacını ve anlaşılabilirliğini değerlendirmek için karmaşık modeller oluşturulabilir ve böylece yazılım sürdürülebilirliği artırılabilir.

Kaynak kodun, yorumların ve dokümantasyonun kalitesi, bir yazılım geliştirme projesinin genel başarısında önemli bir rol oynar. Bu bileşenlerin etkin bir şekilde yönetilmesi ve değerlendirilmesi, yazılım sistemlerinin sürdürülebilirliğini ve ölçeklenebilirliğini sağlamak açısından önemlidir. Bu sorunu çözmek için kaynak kodları, yorumları ve dokümantasyonu doğru bir şekilde değerlendirebilecek yeni indekslerin geliştirilmesi zorunludur. Geliştirilecek olan bu indeksler, kod kalitesi hakkında değerli bilgiler sağlayabilir, anomali tespiti yapabilir, teknik borçlanmayı önceliklendirebilir ve kod anlaşılabilirliğini artırabilir. İndeksler üzerine yapılan bazı çalışmalardan Literatür Araştırması bölümünde bahsedilecektir.

Açık kaynaklı projelerin ve kodların yaygınlaşmasıyla GitHub (GitHub, 2024a) gibi web sitelerinde tutulan kaynak kod hacimleri, büyük miktarlara ulaşmaya başlamıştır (Gupta & Gupta, 2019). Bu alanda bu kadar büyük verinin ortaya çıkması yorum satırı ve dokümantasyon üretimi gibi konularda çalışmalar yapan araştırmacılar için büyük fırsatlar doğurmuştur. GitHub gibi kod depolarının bulunduğu platformlardaki projelerin analiz edilerek yorum satırlarının yeterliliği konusunda çeşitli çıkarımlar yapılabilmektedir.

1.1. Tezin Kapsamı

Kaynak kod içerisindeki yorum satırlarının az olması olumsuz bir durum teşkil ederken, tam tersi olarak çok fazla yorum satırı da iyi bir şey değildir. Bu sebeple bizim çalışmamız, Java projelerindeki Javadoc dokümantasyonuna sahip olmayan metotlara yeteri kadar Javadoc dokümantasyonu üretilmesi üzerine yoğunlaşmaktadır. Her bir Java dosyasındaki yorum satırı, kod kelime sayısı gibi parametreler üzerinden yorum satırlarının az, yeterli veya çok olduğunu belirten "Yorum Sapma Yüzdesi" (YSY) adında bir indeks geliştirilmiştir. Geliştirilen bu indeks ile yorum satırları ile kod satırları arasında matematiksel bir formül oluşturulmuştur. Böylelikle hangi projenin yorum satır sayısının yeterli, yetersiz veya fazla olduğunun anlaşılması hedeflenmiştir.

Bu tez kapsamında, standart yorum satırları (Javadoc dışındaki yorum satırları), yorum yeterliliği hesaplamalarında yeterlilik kriteri olarak kullanılacak fakat üretimi yapılmayacaktır. Javadoc yorumları ise hem yorum yeterliliği kriterlerinde kullanılacak hem de bu yorumların üretimi gerçekleştirilecektir.

Ayrıca literatürde bulunan bazı indeksler açıklanacak ve geliştirdiğimiz indeks ile karşılaştırılacaktır.

1.2. Tezin Amacı

Bu tez, GitHub üzerinde bulunan açık kaynak koda sahip Java projelerinin yorum miktarı analizinin ve yorum satırı üretiminin yapılmasını amaçlamaktadır. Açık kaynaklı Java projelerini ideal yorum oranına ulaştırmak için Javadoc (Oracle, 2024a) dokümantasyonuna sahip olmayan Java metotlarına Javadoc dokümantasyonu üretilecektir. Geliştirilen YSY indeksi ile bir Java projesindeki Javadoc dokümantasyonu yeterliliği kontrol edilecektir.

1.3. Literatür Araştırması

Dokümantasyon yazılım geliştirmenin kritik bir yönüdür. Literatürde, programlama dillerinde yazılan fonksiyon, metot ve sınıflar için dokümantasyon ve kod satırlarının üretilmesiyle ilgili çeşitli çalışmalar bulunmaktadır.

Bu konu üzerinde yapılan ilk çalışmalarda daha çok özel buluşsal yöntemler ve şablonlar kullanılırken son zamanlarda yapılan çalışmalar, doğal dil işleme ve derin öğrenme mimarileri üzerine odaklanmaktadır (Ciurumelea vd., 2023).

(Iyer vd., 2016)'deki çalışmada kod parçaları için otomatik bir şekilde kodun ne yaptığıyla ilgili kısa açıklamalar üreten CODE-NN metodu geliştirilmiştir. CODE-NN "dikkat (attention) mekanizması" ile LSTM ağlarını kullanan bir sinir ağı modelidir. Oluşturulan model her seferinde bir kelime üretmektedir. Modelin eğitimi için yorum satırlarına sahip C# ve SQL kod parçaları kullanılmıştır. Oluşturulan modelin değerlendirme metriği olarak, METEOR ve BLEU kullanılmıştır.

(Hu vd., 2018)'deki çalışmada doğal dil işleme ve derin öğrenme kullanılarak DeepCom adı verilen, Java metotları için otomatik yorum satırı üreten bir metot geliştirilmiştir. Değerlendirme metriği olarak ise BLEU kullanılmıştır. Bu çalışmada, Java metotlarının sözdizimi ve anlamsal yapısının yakalanabilmesi için soyut sözdizimi ağaçlarından (AST) faydalanılmıştır. AST'leri dizilere dönüştürmek için SBT adı verilen yeni bir yöntem önerilmiştir. SBT ile dizilerdeki AST'lerin yapısı korunarak gösterim daha net ve bilgilendirici hale getirilir. Böylelikle yapısal bilginin korunması sağlanarak modelin doğru yorumlar üretme kabiliyeti arttırılmış olur.

(Hu vd., 2020)'deki çalışmada ise (Hu vd., 2018)'deki çalışmada önerilen DeepCom geliştirilerek Hyrid-DeepCom geliştirilmiştir.

Veri odaklı yapılan çalışmalarda oluşturulan modellerin eğitilebilmesi için kaynak kodlar ve kaynak kodlarla ilişkilendirilmiş yorum satırlarından oluşan büyük veri kümeleri gerekli olmaktadır. Ayrıca eğitim için kullanılan veri kümelerindeki dağılımların yanlı olması da doğru olmayan sonuçların çıkmasına sebep olabilmektedir. Bu veri kümelerinin bulunması zorlu, uzun ve yorucu bir süreç haline gelebilmektedir. (Song vd., 2023)'deki çalışma orijinal veri kümesinde bulunan kod ve yorum satırı ikililerini, dengeli bir veri dağılımına sahip büyük ölçekli bir veri kümesine genişletmek için CDA-CS adında bir veri arttırma yöntemi önermektedir. Bu yöntem ile büyük ve dengeli veri kümelerinin oluşturulması kolaylaştırılmıştır.

Özet olarak bu makalede kod özetleme modellerinin daha verimli ve etkin bir şekilde eğitilebilmesi için yeni bir yöntem geliştirilmiştir. Modellerin artırılmış veri kümeleri üzerinde eğitilmesiyle, en son teknolojiye sahip algoritmaların, popüler değerlendirme metriklerinde önemli iyileştirmeler sağlayabileceği gösterilmiştir.

(Huang vd., 2023)'de kaynak kod içerisindeki yorum satırları, metot yorumları ve satır içi yorumlar olarak 2'ye ayrılmıştır ve bunlar arasındaki benzerlikler ve farklılıklar karşılaştırılmış ve analiz edilmiştir. Burada bahsedilen metot yorumları, metotların üzerine yazılan ve onları açıklayan Javadoc gibi daha çok dokümantasyon amaçlı yazılan yorumları kapsarken, satır içi yorumlar ise metotların içerisine yazılan yorumları kapsamaktadır. Bu çalışmada, daha önce yorum üretmek amacıyla yapılan birçok çalışmanın metot yorumları üretmek için yapıldığından ve metot yorumları üretmek amacıyla geliştirilen modellerin, satır içi yorum üretmede o kadar başarılı olmadığından bahsedilmektedir.

(Ciurumelea vd., 2023)'de, Python ve Java ile yazılmış 18000'den fazla açık kaynaklı projede kullanılan yorum satırı yazma tekniklerinin derinlemesine bir analizi yapılmış ve parametre ile geri dönüş değeri başta olmak üzere birçok yapısal öğenin yaygın olarak kullanıldığı gösterilmiştir.

Açık kaynak olarak geliştirilen yazılım projelerinde, kod satırlarının zaman içerisinde değiştirilmesi kaçınılmazdır. Buna istinaden (Panthaplackel vd., 2020)'de yapılan çalışmada, metotlar değiştiği zaman ilgili yorum satırlarının da ona göre değişmesiyle ilgili bir yaklaşım geliştirilmiştir. Önerdikleri modeli, Java ile yazılmış açık kaynaklı yazılım projelerinin, commit geçmişlerini kullanarak oluşturdukları veri kümesi ile eğitmişlerdir. Veri kümesini, aynı metot için hem kodda hem de yorumda eşzamanlı bir değişikliğin olduğu, ardışık iki commit arasındaki, "metot-yorum" çiftlerini toplayarak oluşturmuşlardır.

(Shiina vd., 2022)'de, özellikle programlama dillerini öğrenmeye yeni başlayanlar için programın mantıksal akışının ve genel prosedürünün anlaşılmasına yardımcı olan yorum satırlarının yazılmasından bahsedilmektedir. Bu çalışma, satır bağımlılıklarının ve ayrıştırma ağacı bilgilerinin dağıtılmış bir gösterimini kullanarak yorum oluşturmaya odaklanmaktadır. Satır bağımlılıklarının dağıtılmış gösteriminin oluşturulması için Word2Vec kullanılmıştır.

(Ahmad vd., 2020)'de verilen bir kaynak kod parçasının özet olarak açıklamasını yapan bir model, Transformer mimarisi kullanılarak geliştirilmiştir. Geliştirdikleri modeli, Python ve Java veri kümeleri ile eğitmişlerdir. Model BLEU, METEOR ve ROGUE-L metrikleri ile değerlendirilmiştir.

Bu tez kapsamındaki çalışmaya benzer biçimde, literatürde bulunan birçok çalışmada programlama dilleriyle yazılmış kaynak kodlar için çeşitli indeksler geliştirilmiştir. Önerilen bu indeksler ile yeterlilik, eksiklik, fazlalık vb. tespitler yapılarak kaynak kodlar hakkında çıkarımlar yapılmaktadır. Ayrıca farklı kaynak kodlar birbirleriyle karşılaştırılarak yeni bakış açılarının elde edilmesi kolaylaştırılmaktadır. Bu çalışmalardan bir tanesi, kaynak kod için indeks tekniklerinin önemini vurgulayarak, yazılım geliştirmedeki önemini vurgulamaktadır. (Tronicek, 2022). (Akimova vd., 2022)'de kaynak kodda anomali tespiti için A-Index adı verilen bir indeks önerilmiştir. Bu indeks, büyük bir kaynak kod havuzunda, denetimsiz şekilde eğitilmiş bir model kullanılarak hesaplanır. Bahsedilen çalışmada sırasıyla modeli eğitmek için bir kaynak kod deposu seçilir ve kaynak kod fonksiyonlara bölünür. Transformer tabanlı ve programlama dilleri için geliştirilmiş bir model olan GraphCodeBERT (Guo vd., 2020) kullanılarak kod parçaları vektörlere dönüştürülür. Daha sonra bu vektörler kullanılarak, anomali indeksini hesaplayan bir VAE modeli eğitilir. A-Index, kod kalitesini ölçmek için kullanılabilir. Bu indeksler, kaynak kod kalitesini ölçmek için geleneksel yöntemlerin ötesinde yenilikçi yaklaşımlar sunmaktadır.

(Sae-Lim vd., 2016)'de, bağlamına göre kötü kodların önceliklendirilmesine ilişkin bir indeks olan CRI önerilmiştir. CRI değerinin yüksek olması, kodun olumsuz anlamda kritik seviyede olduğunu gösterir ve bu durumun acilen ele alınması gerektiğini ifade etmektedir. Bu çalışmada bahsedilen teknik, devam eden geliştirme çalışmalarını etkileme olasılığı en yüksek olan kötü kodları önceliklendirir. Böylece geliştiricilerin belirlenen bir alanda kodun kalitesini ve sürdürülebilirliğini arttırmaya odaklanmalarına olanak tanır. Başka bir çalışmada ise kod kalitesi sorunlarını ve teknik borçlanmayı ele almak için yapılandırılmış bir indeks olan "Code Smell Intensity Index" önerilmiştir (Fontana vd., 2015). Bu indeks, kod kalitesi sorunlarını proaktif bir şekilde tespit edip ele almanın sistematik bir yolunu sunmaktadır.

Literatürde, halihazırda var olan yazılım değerlendirme indekslerinin birleştirilerek yeni bir indeksin oluşturulduğu çalışmalarda bulunmaktadır. (Coleman vd., 1994)'deki çalışmada, yazılım sürdürülebilirliğinin hesaplanabilmesi için bir "Sürdürülebilirlik

İndeksi" hesaplamışlardır. "Sürdürülebilirlik İndeksinin" hesaplanmasında, LOC, CCN ve Halstead hacmi kullanılmaktadır. Bu üç indeksin değeri arttıkça, yeni geliştirilen "Sürdürülebilirlik İndeksi" değeri düşer. Aksine, bu üç indeksin değeri azaldıkça yeni geliştirilen "Sürdürülebilirlik İndeksi" değeri yükselir. "Sürdürülebilirlik İndeksi", eksi sonsuzdan 171'e kadar değerler alabilir. "Sürdürülebilirlik İndeksi" için alınan değerin 171'e yaklaşması, ilgili yazılımın bakımının kolaylaştığı anlamına gelir.

Hata tespiti, yazılım kalitesinde çok önemli bir başlıktır ve bu hataları tespit etmek için bazı yaklaşımlar vardır (Adak, 2018). (Phung vd., 2023)'de, yazılımsal hataların tespiti için Error-Type adını verdikleri yeni bir yazılım metrikleri kümesi önerilmiştir. Bu metrikler kümesi, farklı Java çalışma zamanı hatalarının kalıpları hakkında bilgi içeren tahmin modelleri sağlar. Error-Type ile yazılım geliştiricilerin kodları içerisinde bulunan hataları daha iyi tahmin edebilmesi ve önleyebilmesi, yazılım kalitesinin arttırılması ve geliştirme maliyetlerinin düşürülmesi amaçlanmıştır. Blob ve Spagetti Kod gibi anti-desenlerin kodun anlaşılması üzerindeki etkisini anlamak çok önemlidir. (Abbes vd., 2011)' de, bu anti-desenler üzerinde deneysel bir çalışma yürütüldü ve bunların anlaşılabilirlik üzerindeki olumsuz etkileri ortaya çıkarıldı.

Yazılım sistemlerindeki hataların varlığı, güvenilirliği önemli ölçüde etkileyebilmektedir. (Widyasari vd., 2020)' de, Python programlarındaki hatalardan oluşan bir veritabanı olan BugsInPy geliştirilmiştir. Böylelikle, kontrollü test ve hata ayıklama çalışmalarının yapılması kolaylaşmıştır. Geliştiriciler bu tür vertitabanlarından yararlanarak bilinen problemleri çözebilirler ve genel sistem güvenilirliğini iyileştirerek kod kalitesini artırabilirler. Bu bileşenlerin önemine rağmen, kod kalitesini değerlendirmeye yönelik mevcut uygulamalar ve araçlar, yazılımların çok yönlü yapısını ele almada çoğu zaman yetersiz kalmaktadır. Bazı indeksler veya metrikler, sözdizimi ve yapıya odaklanırken, diğerleri karmaşıklığı ve kodlama standartlarına uyumu değerlendirmektedir. Ancak bu faktörleri içeren kapsamlı bir indeks, yorumların ve dokümantasyonun kalitesiyle birlikte değerlendirme sürecine önemli ölçüde katkıda bulunabilir. Böyle bir indeksin yazılım geliştirme yaşam döngüsüne entegre edilmesiyle geliştiriciler, proje yöneticileri ve paydaşlar daha bilinçli kararlar alabilirler. Ayrıca kod incelemeleri kolaylaşabilir ve sürekli iyileştirme kültürü teşvik edilebilir.

Yazılım ürünlerinde, kod kalitesinin değerlendirilmesinin önemi çok sayıda çalışmada vurgulanmıştır. Mevcut uygulamalar sözdizimi, yapı, karmaşıklık ve kodlama standartlarına uyum gibi çeşitli yönlere yoğunlaşırken (Krishnamurthi, 2013), bu faktörlerin yanı sıra yorumların ve dokümantasyonun kalitesini de içeren daha kapsamlı bir indekse olan ihtiyaç da kabul edilmektedir (Sharma & Spinellis, 2020). Yorumlar ve dokümantasyon, yazılımın işlevselliğini ve yapısını açıklamak için yazılırlar. Karmaşık kavramları açıklamak, hata ayıklamak, bakımda yardımcı olmak ve gelecekteki geliştiricilerin iş akışını anlamalarına yardımcı olmak için hayati önem taşırlar.

Kod miktarı ile dokümantasyon miktarı arasındaki denge, yazılımın sürdürülebilirliğini artırmada önemli bir rol oynar. Bu nedenle, indekslerin geliştirilmesinde sadece kaynak koda odaklanılmamalı, aynı zamanda yorumlar ve dokümantasyonda değerlendirilmelidir. Böyle yapılarak kod kalitesi hakkında fikir edinilebilir ve bu da projenin genel başarısını artırmaya yardımcı olur. (Taha & Elhadad, 2003)' de, yazılım kalite güvencesi alanında, araştırmalar, yazılım metrikleri, kalite faktörleri ve McCall, Boehm, ISO 9126 ve SATC NASA modelleri gibi kalite modelleri de dahil olmak üzere çok çeşitli konular ele alınmıştır. (Kreinovich vd., 2020)' de, kod kalitesi ölçümleri için teorik temeller sağlanmış ve temel prensipler açıklanmıştır. (Rabbi vd., 2022)' de, yazılım mimarisi kalitesini garantilemek için kod metriklerini kullanarak Swift projelerini analiz eden SCMA aracı geliştirilmiştir. (Damian vd., 2021)' de, nitel çalışmalar yoluyla iletişim direktiflerinin yazılım projelerine aktarılmasına yoğunlaşılmıştır. Nihai hedef, yazılımın güvenilirliğini ve sürdürülebilirliğini artırmaktır.

2. METODOLOJİ

2.1. Doğal Dil İşleme

Doğal dil işleme, bilgisayarlar ve insanlar arasındaki doğal dil etkileşimine odaklanan yapay zekanın bir alt koludur. Doğal dil işlemenin amacı insan konuşma dilinin, bilgisayarlar tarafından anlaşılabilmesini, yorumlanabilmesini ve üretilebilmesini sağlamaktır.

Geçmişten günümüze kadar doğal dil işleme, farklı metotlar kullanılarak uygulanmıştır. Bu yöntemlerden ilki kural tabanlı metotlardır. Kural tabanlı metotlar ile dil kuralları el ile hazırlanır. Ve hazırlanan bu kurallar ile doğal dilin analizinin yapılmasına çalışılır. Doğal dillerin fazlasıyla karmaşık ve birçok istisnai durum içermelerinden dolayı kural tabanlı yöntemler genellikle yetersiz kalmaktadır. İkinci yöntem ise istatistiksel yöntemler yardımıyla geniş veri kümeleri kullanılarak dildeki kalıpların yakalanmasıdır. Üçüncü yöntem makine öğrenmesi yöntemidir. Burada makine öğrenmesinin alt kollarından denetimli öğrenme veya denetimsiz öğrenme yöntemleri kullanılabilmektedir. Denetimli öğrenmede, etiketlenmiş veriler ile eğitim gerçekleştirilirken, denetimsiz öğrenmede etiketlememiş veri üzerindeki tekrar eden örüntüler aranmaktadır. Dördüncü yöntem ise derin öğrenme yöntemidir. Derin öğrenme yönteminde çok sayıda sinir ağı kullanılmaktadır.

Klasik doğal dil işleme metotları, dilbilgisi ve istatistiksel kurallara dayanmaktadır. Klasik doğal dil işlemede sıklıkla kullanılan bazı metotlar vardır. Bunlar; simgelere ayırma (tokenization), durma sözcüklerinin kaldırılması (removing stop words), köklerine ayırma (stemming), lemmaization, n-gram analizi, vektörleştirme ve ayrıştırma olarak verilebilir. Günümüzdeki doğal dil işleme uygulamaları veri odaklı bir şekilde makine öğrenmesi ve derin öğrenme ile yapılmaktadır (Otter vd., 2021). Yapay zekanın dahil olduğu doğal dil işleme yöntemleri, klasik doğal dil işleme yöntemleri ile kullanılmaktadır.

Doğal dil işleme yapay zekâ ile sıklıkla kullanılmaktadır. Önceki zamanlarda, klasik makine öğrenmesi yöntemlerinden olan K-En Yakın Komşuluk (KNN), Destek Vektör Makinesi (SVM), Karar Ağaçları, Rastgele Orman, Naive Bayes ve Gizli Markov

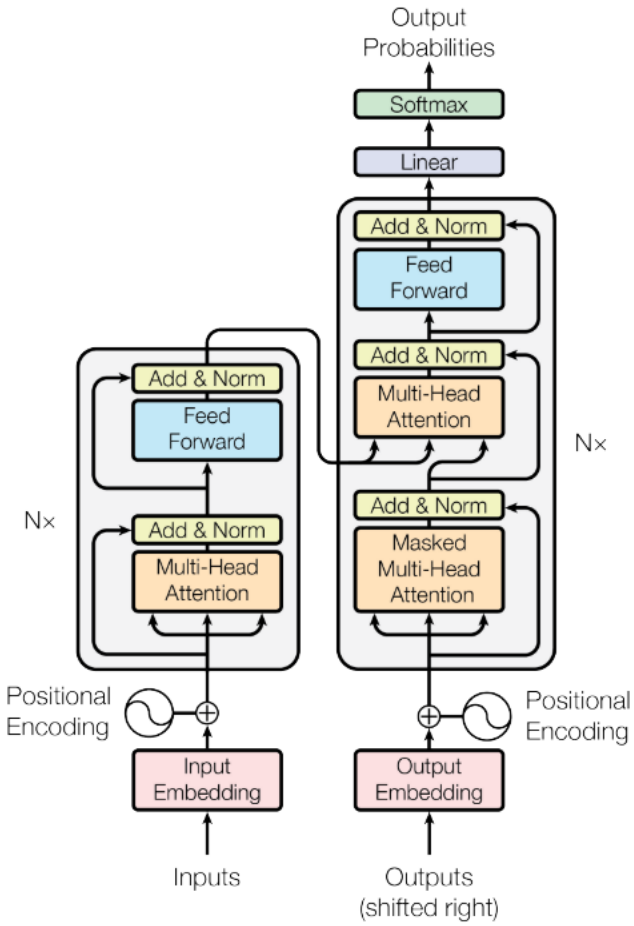
Modeli sıklıkla kullanılırken günümüzde sinir ağlarının temelini oluşturduğu derin öğrenme yöntemleri bu alanı domine etmiştir. Bu sinir ağları; İleri Beslemeli Sinir Ağları, Tekrarlayan Sinir (RNN) Ağları, Uzun Kısa Süreli Bellek (LSTM) Ağları ve Transformer Ağları olarak verilebilir. Bu ağların birbirlerinden farklı olmalarının sebebi kendilerini oluşturan düğümlerin farklı şekillerde bağlanmasıdır.

Doğal dil işlemenin kullanım alanı oldukça geniştir. Doğal dil işleme, konuşma botları, sanal asistanlar, duygu analizi, müşteri geri bildirim analizi, konu modelleme, sağlık uygulamaları, eğitim uygulamaları, anlamlı metin üretimi, metin sınıflandırma, özet çıkarma, çeviri işlemleri vb. konularda sıklıkla kullanılmaktadır.

Doğal dil işleme uygulamalarında yaygın karşılaşılan bazı problemler bulunmaktadır. Bunlardan bir tanesi belirsizliktir. Doğal diller belirsizdir. Farklı bağlamlara göre farklı anlamlar içerebilirler. Belirsizlik durumunu mümkün olduğunca azaltabilmek için büyük miktarda metin verisi üzerinden bağlamın anlaşılması gerekmektedir. Diğer bir problem ise ironi ve alay içeren ifadelerin doğru bir şekilde anlaşılmasındaki zorluktur. Başka bir zorluk da çok fazla sayıda doğal dilin olmasıdır. Her bir doğal dilin kendine özgü sözdiziminin, anlamsal yapısının, istisnalarının olması ve kendi içlerinde farklı lehçelere sahip olabilmesinin getirdiği zorluklar doğal dil işleme üzerinde çalışanların aşması gereken bariyerlerdendir. Bütün bu zorlukların yanı sıra karmaşık dil modellerinin eğitilebilmesi için büyük miktarlarda metin verisine ve hesaplama gücü yüksek bilgisayarlara ihtiyaç duyulmaktadır.

2.1.1. Transformer mimarisi

Transformer mimarisi (Vaswani vd., 2017) tarafından tanıtılan bir sinir ağı mimarisidir (Şekil 2.1.). Bu mimari makine çevirisi ve birçok doğal dil işleme uygulamasında kullanılmaktadır. Doğal dil işleme alanındaki GPT modelleri de dahil olmak üzere birçok büyük çaplı modelin temelini Transformer mimarisi oluşturmaktadır.



Şekil 2.1. Transformer mimarisi (Vaswani vd., 2017)

Geleneksel modeller, kodlayıcı-kod çözücü (encoder-decoder) mimarileri ile RNN, LSTM veya CNN kullanmaktadır. Transformer mimarisinin getirdiği en büyük yenilik "Self-Attention" mekanizmasıdır. "Self-Attention" mekanizması ile bir cümledeki farklı kelimelerin birbirlerine göre önem dereceleri belirlenir. Böylelikle bir cümlede bulunan kelimeler arasındaki ilişkiler, kelimeler arasındaki uzaklıktan bağımsız olarak tespit edilebilir. "Self-Attention", RNN'ye kıyasla daha paralelleştirilebilir hesaplamalara izin vererek eğitim hızında önemli iyileştirmeler sağlar. CNN'ye kıyasla ise uzun menzilli bağımlılıklar için maksimum yol uzunluğunu azaltarak uzak konumlar arasındaki bağımlılıkları öğrenmeyi kolaylaştırır.

RNN tabanlı sıralı modellerin, kaynak kod gösterimlerini öğrenmede iki sınırlaması vardır: birincisi, kod simgelerini sıralı olarak işledikleri için kaynak kodun ardışık olmayan yapısını modellemezler; ikincisi, çok uzun kaynak kodlarda, kod simgeleri arasındaki uzun menzilli bağımlılıkları yakalamada başarısız olabilirler (Ahmad vd., 2020). Transformer mimarisi ise uzun menzilli bağımlılıkları yakalayabilir.

Transformer mimarisi “kodlayıcı” ve “kod çözücü” olmak üzere iki ana parçadan oluşur. Şekil 2.1’de sol kısım “kodlayıcı” kısmını gösterirken sağ kısım ise “kod çözücü” kısmını göstermektedir. Kodlayıcı, giriş dizisini işlemekle sorumludur. Kodlayıcı, “Çoklu Başlıklı Dikkat” (Multi-Head Attention) mekanizması ve “İleri Beslemeli Sinir Ağı” (Feed Forward Neural Network) içeren N katmandan oluşmaktadır. Her bir kodlayıcı katmanı bir önceki katmanın çıktısını alır ve “Çoklu Başlıklı Dikkat” mekanizması ile ileri beslemeli sinir ağından geçirir. Kod çözücü ise çıktı dizisini üretmekten sorumludur ve N katmandan oluşur. Kod çözücü, “Maskeli Çoklu Başlıklı Dikkat” mekanizması, kodlayıcıdan gelen çıktı üzerinde çalışan bir “Çoklu Başlıklı Dikkat” mekanizması ve bir “İleri Beslemeli Sinir Ağından” oluşmaktadır. Burada bulunan çok başlı dikkat mekanizması her bir konumdaki tahminin yalnızca kendisinden önce bilinen çıktılara bağlı olmasını sağlamaktadır. Kod çözücü katmanlarının dışında “Linear” ve “Softmax” katmanları bulunur. “Linear” katman olasılık dağılımlarını hesaplarken “Softmax” katmanı ise bu olasılıkları normalize ederek toplamı 1 olacak şekilde çıktı olasılıklarını üretir.

Transformer mimarisi, bir kelime dizisindeki kelimelerin sırasını doğal olarak anlayamadığı için her kelimenin cümle içerisindeki konumunu anlayabilmek amacıyla “Input Embedding” (Girdi Yerleştirmesi) kısmına “Positional Encoding” (Konumsal Kodlama) yapar. “Positional Encoding” ile yapılan kodlamalar, konumsal bilgilerin korunmasını sağlamak amacıyla “Input Embedding” kısmına dahil edilmektedir.

“Çoklu Başlıklı Dikkat” mekanizması ile bir cümlenin farklı kısımlarına aynı anda odaklanabilmek amacıyla birden fazla dikkat kafası kullanır. Her bir başlık, dikkat (attention) işlemini bağımsız olarak gerçekleştirir. Daha sonra çıktılar birleştirilir ve doğrusal olarak dönüştürülür. Bu yaklaşım ile kelimeler arasındaki ilişkilerin farklı yönlerinin yakalanması sağlanır.

“Add&Norm” kısmında dikkat (attention) mekanizmalarının çıktıları girdiye eklenerek modelin kararlılığı arttırılır. Transformer mimarisinde bulunan “Feed Forward” ise her bilgiyi bağımsız olarak işleyen bir “İleri Beslemeli Sinir Ağıdır”.

Transformer mimarisinin, makine çevirisi, metin özetleme, hikâye oluşturma gibi birçok doğal dil işleme probleminde iyi sonuçlar verdiği gösterilmiştir. Transformer, yeni bir mimari sunarak doğal dil işleme alanında önemli ilerlemeler sağlamıştır. Aynı zamanda doğal dil işleme alanında birçok gelişmeyi etkileyen temel bir çalışmadır.

2.1.2. Üretken önceden eğitilmiş transformer modelleri

Üretken önceden eğitilmiş transformer (GPT) modelleri, transformer mimarisi üzerine inşa edilmiş olan birçok doğal dil işleme uygulamasında üst seviye sonuçlar vermiş olan bir dil modelleri sınıfıdır. Bu modeller OpenAI tarafından geliştirilmişlerdir. GPT-1, GPT-2, GPT-3 ve GPT-4 olmak üzere farklı versiyonları bulunmaktadır. GPT-1 en basit GPT modelidir. GPT-2 1,5 milyar parametre ile eğitilmişken GPT-3 ise 175 milyar parametre ile eğitilmiştir. GPT-4 ise son GPT modelidir.

GPT modellerinin eğitimi iki ana aşamadan oluşmaktadır. Bunlar ön eğitim aşaması ve ince ayar aşamasıdır. Ön eğitim aşamasında, farklı dil kalıplarından ve yapılarından çok geniş metin havuzları kullanılmaktadır. Bu aşamada etiketli veri gerekmemektedir. İnce ayar aşamasında, Aktarımlı Öğrenme (Transfer Learning) yapılır. Aktarımlı Öğrenme ile ön eğitim aşamasındaki model ağırlıkları kullanılarak, daha önceden eğitilmiş modellerden yararlanılır. Bu modellerin eğitiminde kullanılan ön eğitim ve ince ayar paradigması, modelin hedef uygulamaya göre özelleştirilebilmesini ve hedef uygulamanın ihtiyacı olmayan gereksiz ayrıntılardan arındırılabilmesini sağlar. Böylelikle modelin boyutu küçültülerek hızlanması sağlanır.

GPT modelleri, otoregresif bir yaklaşımla, her adımda önceki bağlama dayalı olarak bir sonraki belirteci tahmin ederek metin üretir. Bu yöntem, oluşturulan metinde tutarlılık ve bağlam farkındalığı sağlar. GPT modelleri, katman sayılarının, gömülme boyutlarının (embedding size) ve dikkat (attention) başlıklarının sayılarının değiştirilmesiyle ölçeklendirilebilir. Bu model ailesi, ön eğitim aşamasında büyük ölçekli metin verileri kullandığı için çok yönlüdür ve farklı doğal dil işleme problemlerine karşılık verebilir.

2.1.3. Prompt mühendisliği

Prompt mühendisliği (prompt engineering), GPT gibi büyük dil modellerini kullanırken etkinlik ve verimliliği maksimize etme konusunda önemli bir rol oynamaktadır. Prompt Mühendisliği, performansın, kullanıcı deneyiminin ve optimizasyonun geliştirilmesini sağlar. Model çıktılarının doğru ve ihtiyaçlara karşılık verebilmesi için büyük dil modelinde kullanılan prompt'un dikkatli ve amaca uygun bir şekilde hazırlanması gerekmektedir.

Etkili ve iyi bir büyük dil modeli prompt'u yazılabilmesi için belirsiz ifadeler kullanmak yerine spesifik ifadeler kullanmak gerekmektedir. Bunun yanında konu bağlamının doğru bir şekilde ifade edilmesi daha tutarlı ve alakalı sonuçların elde edilmesini sağlayacaktır. Ayrıca prompt'un belirli bir format dahilinde yazılması da sonuçların daha net olmasını sağlayacaktır.

2.2. Kod Analiz Yöntemleri

Kod analiz yöntemleri, bir programlama diliyle yazılmış kaynak kodun, kalitesini, performansını, ne kadar güvenli olduğunu ve fonksiyonelliğini ölçmek için kullanılan teknikler bütünüdür. Bu metotlar genel anlamda, "Statik Kod Analizi" ve "Dinamik Kod Analizi" başlıkları altında olmak üzere ikiye ayrılmaktadır.

2.2.1. Statik kod analizi

Statik kod analizi, kaynak kod çalıştırılmadan yapılan analizlerdir. Bu başlık altında bulunan analiz yöntemleri ile yazılım geliştirme süreci sırasındaki potansiyel hatalar ve sorunların tespit edilmesi amaçlanır. Linting, güvenlik analizi, tip kontrolü, biçimsel doğrulama ve kod metrikleri statik kod analiz yöntemlerinden bazılarıdır. Linting ile kodlama standartları, stil yönergeleri ve sözdizimi hataları kontrol edilir. Python programlama dili için kullanılan Pylint bir linting aracıdır (Pylint, 2024). Güvenlik analizi ile SQL Enjeksiyonu (SQL Injection) ve Siteler Arası Komut Dosyası Çalıştırma Saldırısı (Cross-Site Scripting Attack, XSS) gibi güvenlik açıkları taranır. Tip kontrolü ile programlama dilinde kullanılan değişken türlerindeki hatalar ve tutarsızlıklar kontrol edilir. Python programlama dili için kullanılan Mypy bir statik tip kontrolü aracıdır (Mypy, 2024). Biçimsel doğrulama ile matematiksel yöntemler kullanılarak algoritmaların doğruluğu teyit edilir. Kod metrikleri ise kaynak kodu değerlendirmek için kullanılan niceliksel ölçümlerdir.

Çeşitli kod metrikleri kullanılarak kod kalitesi, sürdürülebilirliği, karmaşıklığı ve performansı değerlendirilebilir. Aşağıda, bazı kod metriklerinin, açıklamaları alt başlıklar halinde yapılacaktır.

2.2.1.1. Döngüsel karmaşıklık

Döngüsel karmaşıklık (Cyclomatic Complexity), 1976 yılında Thomas J. McCabe tarafından tanıtılan bir programın karmaşıklığını ölçmek için kullanılan bir yazılım metriğidir (McCabe, 1976). Bu metrik, yazılım mühendisliği alanında, özellikle kodun

anlaşılması, test edilmesi ve bakımı için çok önemlidir. Bir programın döngüsel karmaşıklığı, denklem 2.1'deki gibi hesaplanır.

$$V = E - N + 2P \tag{2.1}$$

Denklem 2.1'de:

- E, programın kontrol akış grafındaki kenar sayısıdır.
- N, programın kontrol akış grafındaki düğüm sayısıdır.
- P, programın kontrol akış grafındaki birbirine bağlı bileşenlerin sayısıdır (tipik olarak tek bir program veya alt yordam için P=1).

Daha basit bir ifadeyle, döngüsel karmaşıklık bir programdaki karar noktalarının sayısı artı bir olarak düşünülebilir. Bu karar noktaları if, for, while, case gibi koşullu ifadeleri ve kontrol akışını etkileyen mantıksal operatörleri içerir. Yüksek döngüsel karmaşıklık, yüksek hata potansiyeline ve daha karmaşık test gereksinimlerine sebep olur. Düşük döngüsel karmaşıklığa sahip programların anlaşılması, test edilmesi ve bakımı genellikle daha kolaydır. Döngüsel karmaşıklık, kapsamlı testlerin oluşturulmasında, çok karmaşık kod bloklarının basitleştirilerek yeniden yazılması için tespit edilmesinde, yazılımdaki kusurların tespit edilmesinde yardımcı bir metriktir.

Döngüsel karmaşıklık doğru bir şekilde analiz edildiğinde ve ona uygun hamleler yapıldığında bir yazılımın kalitesi ve güvenilirliği arttırılabilir.

2.2.1.2. Halstead metrikleri

Halstead metrikleri (Halstead metrics), kaynak kodu çeşitli yönlerden ölçmek için 1977'de Maurice H. Halstead tarafından tanıtılan bir dizi yazılım metriğidir (Halstead, 1977). Bu metrikler, kaynak kod içerisindeki operand ve operatörlere bağlı olarak hesaplanır ve kodun anlaşılabilmesi için ne kadar çaba gerektiğiyle ilgili bilgi verir. Halstead hacmi (Halstead Volume), kaynak kod boyutunu gösterir. Halstead zorluğu (Halstead Difficulty), kod yazılma zorluğunu belirtir. Halstead çabası (Halstead Effort), kodu geliştirmek için gereken çabayı tahmin eder.

2.2.1.3. Satır sayısı

Satır sayısı (Lines of Code, LOC), çok basit bir metriktir. Kaynak kodun satır sayısını belirtir.

2.2.1.4. Kalıtım ağacının derinliği

Kalıtım ağacının derinliği (Depth of Inheritance Tree, DIT), sınıf hiyerarşisinin tepesinden, hedef sınıfa kadar olan kalıtım seviyesini ölçer. Ağacın derin olması karmaşıklık ve hata riskini arttırırken tekrar kullanılabilirlik konusunda olumlu olabilir.

2.2.2. Dinamik kod analizi

Dinamik kod analizi, kaynak kodun çalıştırılarak davranışının izlenmesiyle yapılır. Bu metot ile statik kod analizinde tespit edilemeyen sorunlar bulunabilir. Birim testi, entegrasyon testi, performans testi, bellek profili oluşturma, kod kapsam testi ve fuzz testi dinamik kod analiz yöntemlerinden bazılarıdır. Birim testi ile bir kod parçasının, genellikle bir fonksiyonun, doğru çalışıp çalışmadığı tespit edilir. Java programlama dili için JUnit çerçevesi birim testi kütüphanesidir (JUnit 5, 2024). Python programlama dili için ise unittest yerleşik çerçevesi ile pytest çerçevesi birim testi kütüphaneleridir (unittest, 2024), (pytest, 2024). Entegrasyon testi ile farklı kod bileşenlerinin birlikte doğru çalışıp çalışmadığı test edilir. Örneğin bir web uygulaması ile bağlı bulunduğu veritabanının birlikte test edilmesi entegrasyon testine örnek olarak verilebilir. Performans testi ile bir yazılımın performansı ve ölçeklenebilirliğinin değerlendirilir. Açık kaynak kodlu ve %100 Java ile yazılmış olan Apache JMeter ile performans testleri yapılabilir (Apache Software Foundation, 2024). Bellek profili oluşturma ile bir yazılımdaki bellek sızıntısı ve verimsiz bellek kullanımı tespit edilir. Kod kapsamı analizinde ile test sırasında kaynak kodun ne ölçüde çalıştırıldığı değerlendirilir. Java programlama dili için “cobertura” aracı kod kapsamı analizinde kullanılabilir (GitHub, 2024b). Python programlama dili için ise “Coverage.py” aracı kod kapsamı analizinde kullanılabilir (coveragepy, 2024). Fuzz testinde ise otomatik olarak rastgele girdi verileri oluşturularak beklenmedik davranışlar ve güvenlik açıkları bulunmaya çalışılır.

2.3. Java

Bu çalışmada, Java programlama dili ile yazılmış yazılım projeleri inceleneceği için bu dilin yapısal özellikleriyle ilgili birkaç şeyden bahsedilmesi faydalı olacaktır. Java, nesneye dayalı paradigmaya sahip, platform bağımsız, nispeten katı kurallara sahip güvenli bir programlama dilidir.

Java programlama dili, sınıf tabanlıdır. Her şey sınıflarla ilişkilidir. Sınıflardan nesneler türetilir. Bir java dosyası içerisinde dosya ile aynı adı taşıyan bir yapı (sınıf vb.) bulunması gerekir. Sınıflar içerisinde veriyi temsil eden nitelikler ve veri üzerinde işlemler yapan metotlar bulunur. Javada sınıfların organize edilebilmesi için paket sistemi kullanılır. Paketler, dosya sistemindeki klasörlere benzerdir. Böylelikle birbirleriyle alakalı sınıf vb. yapılar birlikte tutularak modülerlik arttırılmış olur. Javada, sınıfların nitelikleri ve metotların erişim belirteçleri vardır. Erişim belirteçleri, javadaki yapıların nerelerde ve nasıl çağırılabilecekleri hakkında bilgi verir. Java programlama dilinde, metot imzası değişmek koşuluyla aynı isimde birden fazla metot yazılabilir. Buna metotların aşırı yüklenmesi denir. Ayrıca Javada "Annotation" özelliği bulunmaktadır. Bir Java sınıfının veya metodunun üzerine @ işareti ile "Annotation" eklenebilir. "Annotation" eklendiği yapı hakkında ek bilgi verebilir veya bu yapıya yeni özellikler kazandırabilir.

Bu tez kapsamında, Java programlama dili ile yazılmış projelerin içerisinde bulunan ve Javadoc dokümantasyonuna sahip olmayan metotlar ile ilgili çalışıldığı için bu bölümde Java metotlarıyla ve metotların içlerinde bulunduğu Java sınıflarıyla ilgili kısa bilgi verilmiştir.

2.4. Programlama Dillerinde Yorum Satırları ve Önemi

Yorum satırları, programlama dilleri için önemli bir yapıdır. Bir programlama dili ile kaynak kod içerisine yazılan yorumlar ile kodun işlevinin, mantığının, amacının ve işlevselliğinin anlaşılması kolaylaşır. Kod yorumları, bir bilgisayar programının işleyiş sürecini açıklar ve hata ayıklama ve bakım gibi programlama görevlerinin uzun vadede daha verimli olmasını sağlar (Park vd., 2023). Yazılan yorumlar, ileride güncellemelere ihtiyaç duyabilecek kod parçalarının veya hatalı kod bölümlerinin belirlenmesine katkı sağlar. Yorum satırlarının bir diğer kullanım amacı ise kod içi dokümantasyon oluşturmaktır. Python programlama dilinde fonksiyonların giriş parametrelerini, dönüş değerlerini ve amaçlarını gösteren "Docstring" kullanımı buna örnek olarak verilebilir (Python Docstring, 2024). Aynı şekilde java programlama için "Javadoc" kod içi dokümantasyonu bulunmaktadır. Javadoc, java projelerindeki kaynak kod dosyalarını belgelemek için kullanılan fiili standarttır (Steinbeck & Koschke, 2021). Bazı özel dokümantasyon araçlarıyla kod içi dokümantasyonlar da okunarak kapsamlı dokümantasyon belgeleri oluşturulabilir. Java programlama dili

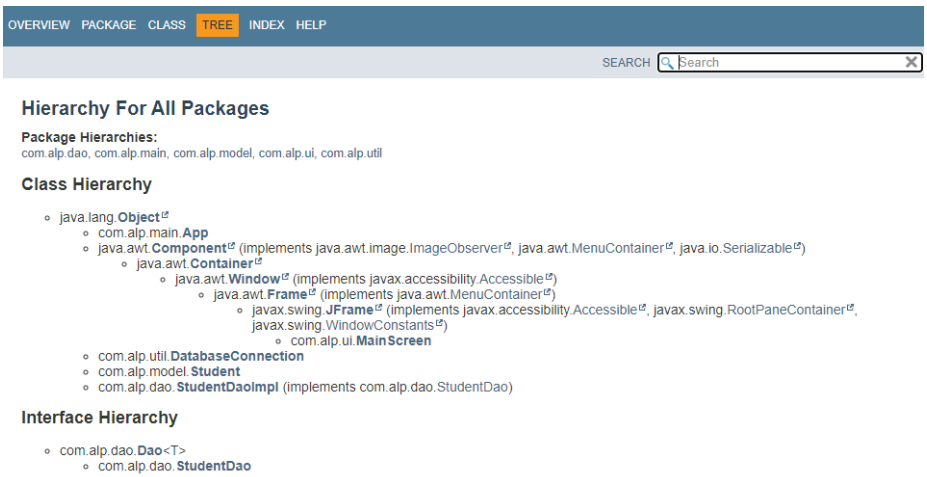
için geliştirilmiş olan ve JDK içerisinde bulunan "javadoc komut satırı aracı" proje içerisine yazılmış olan bütün Javadoc yorum satırlarını da kullanarak bir html dokümanı oluşturmaktadır (Oracle, 2024b). Eğer proje içerisinde bulunan sınıf, arayüz, metot vb. dile özgü yapılar için Javadoc dokümantasyonları iyi bir şekilde yazılmış ise bu araç ile otomatik olarak, geliştirilen yazılım için bir API üretilebilecektir. Javadoc komut satırı aracının kullanılabilmesi için komut satırına "javadoc" yazılması ve ilgili parametrelerin eklenmesi gerekmektedir. Şekil 2.2'de verilen örnek komut bu dokümantasyon aracının kullanıma örnektir.

```
javadoc -d [çıktı klasörü] -sourcepath [kaynak kod klasörü] -
subpackages [paket adı]
```

Şekil 2.2. Javadoc komut satırı aracının örnek kullanımı

Şekil 2.2'de, "-d" parametresi, üretilecek html dokümantasyonun yerleştirileceği dizini belirtirken, "-sourcepath" parametresi kaynak kod dosyalarının bulunduğu dizinleri belirtir. "-subpackages" parametresi ise belirtilen paket ve alt paketlerdeki tüm dosyaları işler.

Şekil 2.3'te yukarıdaki komut kullanılarak HTML ile oluşturulmuş bir dokümantasyon gösterilmiştir.



Şekil 2.3. Javadoc komut satırı aracı ile üretilmiş html dokümantasyonu örneği

Şekil 2.3'te gösterilen ve javadoc komut satırı aracı ile üretilmiş olan web sitesi şeklindeki yazılım projesi dokümantasyonu, ilgili Java projesindeki sınıflar, arayüzler, metotlar, değişkenler ve diğer bileşenler hakkında detaylı bilgiler içermektedir. Bu

HTML belgeleri, anlaşılabilirlik, tutarlılık, otomatik güncelleme ve dışa aktarılabilirlik gibi konularda oldukça faydalıdır. Bu nedenle javadoc komut satırı aracı, Java projesi geliştirme sürecinde önemli bir yardımcı araç olarak kullanılmakla birlikte büyük projelerin yönetiminde önemli bir rol oynamaktadır.

Bir yazılım projesi içerisine yazılan yorum satırları ile projenin anlaşılabilirliği arttırılır ve projeye yeni dahil olan ekip üyelerinin öğrenme eğrileri azaltılır. Birçok yazılım projesinde versiyon kontrol sistemleri kullanılmaktadır. Bir kodun farklı versiyonları arasındaki değişikliklerin neden yapıldığıyla ilgili geriye yönelik inceleme yapıldığı zaman yorum satırları çok önemli bir rol oynayabilir.

Java programlama dilinde 3 farklı yorum türü bulunmaktadır. Bunlar tek satırlı yorumlar, çok satırlı yorumlar ve Javadoc yorumlarıdır. Tek satırlı yorum örneği Şekil 2.4'te verilirken çok satırlı yorum örneği ise Şekil 2.5'te verilmiştir.

```
public static synchronized void warn(String s, String sInfoSup) {
  String sOut = s + ": " + sInfoSup;
  // Just display the message if Log is not yet enabled
  if (logger == null) {
    System.out.println("[WARN] " + sOut);
```

Şekil 2.4. Java dilinde tek satırlı yorum (kırmız kutu içerisinde)

```
/* (non-Javadoc)
 * @see java.lang.Thread#run()
 */
@Override
public void run() {
  Log.debug("Exit Hook begin");
```

Şekil 2.5. Java dilinde çok satırlı yorum (kırmızı kutu içerisinde)

Tek satırlı yorumlar, kısa açıklamalar veya notlar için kullanılabilen satır içi yorumlarken çok satırlı yorumlar, ayrıntılı açıklamalar yapmak için veya bazı kod bloklarını çeşitli sebeplerle (test vb.) pasif hale getirmek için kullanılabilmektedir.

Java programlama dilinde, Javadoc özel bir çok satırlı yorum formatıdır (Şekil 2.6.). Javadoc, standart yorumlardan farklı olarak dokümantasyon özelliği de taşımaktadır. Javadoc yorumları tek satırlı ve çok satırlı yorumlardan farklıdır. Java dilinde bulunan sınıf, arayüz, enum, metot, alan (field) gibi yapıların üstlerine yazılarak bu yapıların amaçlarını, parametrelerini, geri dönüş değerlerini, yaptıkları işleri vb. anlatmak için kullanılabilir.

```java
/**
 * Contains repeated item.
 *
 * @param items
 *            The items to check for repeat.
 *
 * @return whether a stack item list contains a least one repeated item
 */
private static boolean containsRepeatedItem(List<StackItem> items) {
  for (StackItem item : items) {
    if (item.isRepeat()) {
      return true;
    }
  }
  return false;
}
```

Şekil 2.6. Java metot dokümantasyonu, javadoc (kırmızı kutu içerisinde)

Şekil 2.6'daki gibi yazılan Javadoc dokümantasyonlarından, Şekil 2.2'de örnek kullanımı gösterilen javadoc komut satırı aracı vasıtasıyla Şekil 2.3'te gösterildiği gibi bir dokümantasyon belgesi, web sitesi şeklinde elde edilebilir. Bu örnek, Javadoc yorum satırlarının, tek satırlı ve çok satırlı yorumlarda olmayan dokümantasyon özelliğini göstermektedir.

Java programlama dilinde kod içi dokümantasyon oluşturmak için kullanılan Javadoc yorum satırları, kendine has bir sözdizimine sahip özel bir yapıdır. Bu yorumlar /** sembolü ile başlar ve */ sembolü ile sonlanır. Ayrıca, Javadoc yorumlarının her satırının başında * sembolü olur. Bu özel yapı içerisinde, farklı anlamlara sahip etiketler bulunmaktadır (Oracle, 2024c). Örneğin, @param etiketi bir metodun parametrelerini açıklamak için kullanılırken, @return etiketi metodun geri dönüş değerini belirtmek için kullanılır. Ayrıca, @throws veya @exception etiketleri, bir metodun hangi istisnaları fırlatabileceğini belirtir. Bu etiketler, dokümantasyonun daha anlaşılır ve düzenli bir şekilde oluşturulmasını sağlar.

Javadoc etiketleri, kodun okunabilirliğini ve bakımını artırırken, aynı zamanda geliştiriciler arasında ortak bir anlayış ve standart oluşturulmasına katkıda bulunur. Bu da yüksek kalitede ve kapsamlı dokümantasyon oluşturmanın etkili bir yoludur. Etiketler @ işareti ile başlamaktadır. Tablo 2.1'de bazı Javadoc etiketleri açıklamalarıyla beraber verilmiştir.

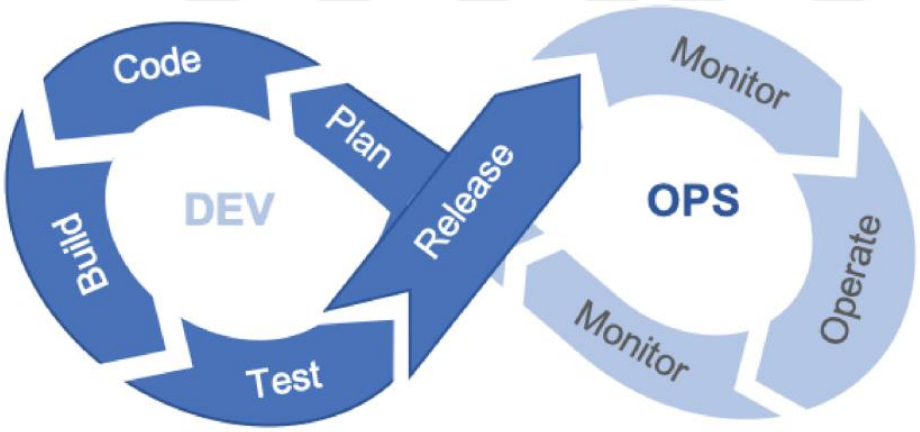
Tablo 2.1. Javadoc etiketleri ve açıklamaları.

Etiket	Açıklama
@author	Bir sınıf, metot vb. yazarını belirtir.
@version	Bir sınıf, metot vb. versiyonunu belirtir.
@param	Bir metot veya yapıcı fonksiyonun parametresini belirtir.
@return	Bir metot veya yapıcı fonksiyonun parametresini belirtir.
@throws veya @exception	Bir metot veya yapıcı fonksiyonun parametresini belirtir.
@see	Başka bir sınıf, metot vb. referans vermek için kullanılır.
@since	Bir sınıf, metodun vb. ilk tanıtıldığı sürümü belirtir.
@deprecated	Kullanımdan kaldırılacağı için artık kullanılmaması gereken sınıf, metot vb. belirtmek için kullanılır.

Bir yazılım projesi içerisinde Javadoc dokümantasyonu kullanmanın birçok yararı bulunmaktadır. Javadoc dokümantasyonu doğru bir şekilde kullanıldığında, dokümantasyon yazımında tutarlı bir format izlenir ve dokümantasyonun anlaşılması kolaylaşır. Kaynak koddan otomatik olarak HTML belgeleri oluşturan araçların kullanılmasıyla zamandan tasarruf sağlanır ve diğer yazılım geliştiriciler için bir API oluşturulmuş olur. Her sınıf, arayüz ve metot gibi yapının üzerine yazılabildiği için geliştiriciyi kodlarını açıklamaya teşvik eder kod kalitesine ve sürdürülebilirliğine önemli katkı sağlar.

2.5. DevOps ve Yorum Satıları

DevOps, Agile metodolojisine uyarlanmış ve yazılım geliştirme ile operasyon ekiplerinin iş birliği yaparak kaliteli yazılımın sürekli ve kısa sürede geliştirilmesini sağlayan yeni bir yaklaşımdır (Jayakody & Wijayanayake, 2021). DevOps, yazılım yaşam döngüsünde çok önemli bir yere sahiptir. DevOps kelimesindeki "Dev" geliştirici takımını belirtirken "Ops" ise operasyon takımını belirtir (Amaro vd., 2023). Dolayısıyla DevOps adı geliştrici takımı ile operasyon takımının entegrasyonundan gelmektedir (Gokarna & Singh, 2021). DevOps ile yazılım geliştirme altyapısı otomatikleştirilebilir, iş akışları iyileştirilebilir, uygulama performansı izlenerek iş birliği ve üretkenlik arttırılabilir. Versiyon kontrol yönetimi araçları (Git vb.), sürekli entegrasyon / sürekli dağıtım (CI/CD) araçları (Jenkins, Travis CI vb.), konfigürasyon yönetimi araçları, konteyner araçları (Docker vb.), orkestrasyon araçları (Kubernetes vb.), izleme araçları, kayıt tutma araçları ve bulut platformları (Microsoft Azure, Amazon Web Service vb.) DevOps çatısı altında kullanılan araçlardır. Şekil 2.7'de DevOps akış şeması verilmiştir.



Şekil 2.7. DevOps akış şeması, (Lwakatare vd., 2020)

Yazılım projelerinde yazılan yorum satırları, DevOps süreçlerinde çok önemli bir rol oynar ve yazılım projelerinin sürdürülebilirliğine, otomasyonuna, izlenebilirliğine katkı sağlar. Yorum satırları, ayrıca DevOps sürecini uygulayan farkı birimler arasındaki iletişimin sağlanmasında da faydalıdır.

DevOps iş birliğini ve verimliliği teşvik eden bir metodolojidir. Bu açıdan bakıldığında DevOps ile kod yorumları arasında önemli bir ilişki bulunmaktadır. Kod yorumları ise kaynak kodun içinde kodla ilgili açıklamalar ve talimatlar sağlayan ek açıklamalardır. Programın yürütülmesini etkilemezler fakat kod ile etkileşime giren

geliştiriciler ve diğer paydaşlar için bir rehber görevi görürler. Dolayısıyla yazılan kod yorumları, DevOps'taki iş birliği ve verimlilik gibi ilkelere doğrudan katkı sağlar. Yeni ekip üyeleri, yazılımın karmaşık mantığını ve mimarisini açıklayan yorumları okuyarak hızlı bir şekilde projeye uyum sağlayabilirler. Açık ve net yorumların yazılması, kod okunabilirliğini ve sürdürülebilirliğini korumaya yardımcı olduğu için, kod değişikliklerinin sık ve hızlı olduğu DevOps için yorum satırları yadsınamaz bir öneme sahiptir. DevOps'ta, dağıtım, test vb. otomasyon görevleri için bulunan komut dosyaları kritik öneme sahiptir. Bu komut dosyalarında bulunan yorum satırları, bağlam ve talimatlar sağlayarak otomatik süreçlerin daha iyi anlaşılmasına katkıda bulunur. Ayrıca, yazılan yorum satırları, CI/CD işlem hattındaki sorunların hızlı bir şekilde çözülmesini kolaylaştırır.

2.6. Yazılım Kalite Nitelikleri

Yazılım kalite nitelikleri, bir yazılım ürününün genel performansını, etkinliğini ve kullanıcı memnuniyetini tanımlayan kritik unsurlardır. Bu niteliklerden biri, yazılım ürününün gerçekleştirmek üzere tasarlandığı belirli işlev ve özelliklerle ilgili olan işlevselliktir. Bu nitelik, yazılımın belirli gereksinimleri ve standartları karşılama kapasitesini değerlendirerek amaçlanan görevleri doğru ve güvenilir bir şekilde yerine getirmesini sağlar. Bir diğer önemli nitelik ise yazılımın rutin operasyonlar sırasında işlevlerini yerine getirme ve sürdürme yeteneğini ifade eden güvenilirliktir. Güvenilirlik, yazılımın gerçek dünya senaryolarında kararlılığını ve hatasız çalışabilirliğini ölçen bir niteliktir. Güvenilirlik, yazılımın beklenen iş yüklerini sorunsuz bir şekilde karşılayabileceğini doğrular. Kullanılabilirliğe geçersek, bu nitelik yazılımın kullanıcı dostu olmasına odaklanır. Kullanılabilirlik, son kullanıcının bakış açısından kullanım kolaylığını, anlaşılabilirliği ve öğrenilebilirliği değerlendirerek kullanıcıların kapsamlı bir eğitim ve desteğe ihtiyaç duymadan yazılımda verimli bir şekilde gezinebilmelerini sağlar. Bir sonraki nitelik ise yazılımın amaçlanan işlevselliği sunarken kaynak kullanımındaki performansını ifade eden verimliliktir. Verimli bir yazılım, bellek, işlem gücü ve bant genişliği gibi sistem kaynaklarının kullanımını optimize ederek genel anlamda daha iyi bir kullanıcı deneyimine katkıda bulunur. Sürdürülebilirlik (bakım kolaylığı), yazılım içindeki sorunların ne kadar kolay tespit edilip düzeltilebileceğiyle ilgili olan bir diğer önemli niteliktir. Sürdürülebilir olan bir yazılım mimarisi, sorunsuz bir şekilde güncellemelerin yapılabilmesine, hata ayıklanabilmesine ve geliştirmelerin

yapılabilmesine olanak tanıyarak uzun vadeli kullanılabilirlik ve uygunluk sağlar. Son olarak taşınabilirlik niteliği, bir yazılımın bir ortamdan veya platformdan diğerine minimum değişiklikle aktarılabilme yeteneğini ifade eder. Bu nitelik, yazılımın farklı platformlarda ve cihazlarda çalışabilmesini sağlayarak uyarlanabilirliğini ve kullanıcı sayısını artırır.

Özetle, işlevselliğe, güvenilirliğe, kullanılabilirliğe, verimliliğe, sürdürülebilirliğe ve taşınabilirliğe odaklanmak, sağlam etkili ve kullanıcı merkezli yazılım ürünleri geliştirmek için kritik öneme sahiptir.

2.7. Önerilen Model

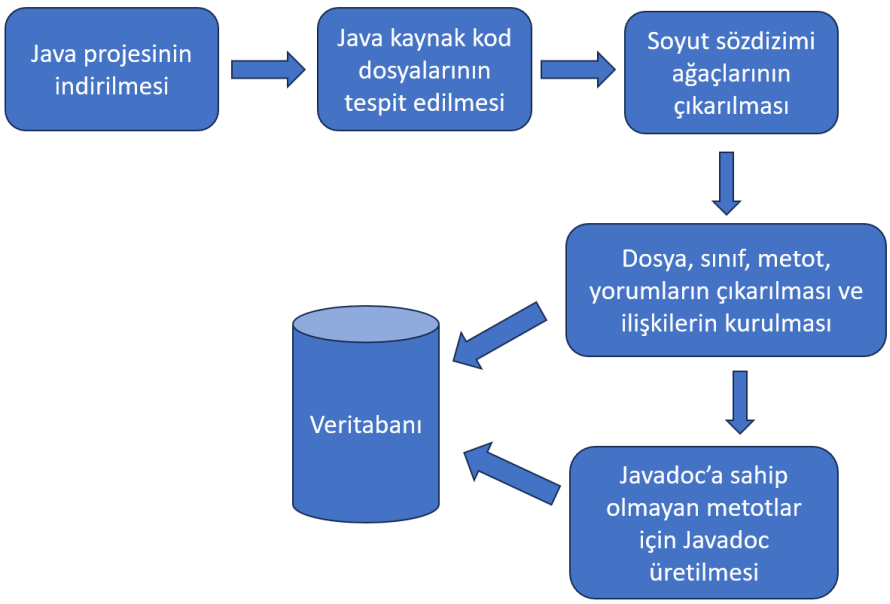
Tez kapsamında GitHub üzerinde bulunan bazı büyük ve açık kaynak Java projeleri incelenecektir. Açık kaynaklı ve Java ile yazılmış bir GitHub projesi klonlandıktan sonra içerisinde bulunan bütün Java kaynak kod dosyaları ayrı ayrı tespit edilecektir. Bu işlem için hedef Java projesi özyinelemeli olarak dolaşılarak, .java uzantılı dosyalar bulunacaktır. Tespit edilen her bir Java kaynak kod dosyasının Soyut Söz Dizimi Ağacı (AST), Python ile yazılmış olan "javalang" (Thunes, 2024) kütüphanesi kullanılarak çıkarılacaktır. Bir java dosyasının AST'sinin çıkarılabilmesi için dosya içerisinde bulunan kod metninin, javalang kütüphanesi tarafından sunulan ilgili fonksiyona parametre olarak verilmesi gerekmektedir.

AST'ler kaynak kod hakkında sözdizimi ve anlamsal bilgiler içerdiği için analiz amacıyla kullanılmaya çok elverişlidir (Bilgin vd., 2020). Bu yüzden çıkarılan AST'nin düğümleri, yazılan Python kodu ile gezilecek ve ilgili Java kaynak kod dosyasında hangi Java sınıflarının olduğu tespit edilecektir. Aynı şekilde, tespit edilen Java sınıflarının içerisinde hangi Java metotlarının olduğu da tespit edilecektir. Java dilinde her metot, bir sınıf içerisinde olmak zorunda olduğu için bu yöntem ile bütün metotlar, bulundukları sınıf bilgisiyle beraber tespit edilebilecektir. Bir dosya için yapılan bu işlem, Java projesi içerisinde bulunan bütün Java dosyaları için yapılacaktır. Böylelikle bir Java projesi sistematik olarak fonksiyon seviyesinde parçalara ayrılmış olacaktır.

Java'da metotlar ve sınıflar, Javadoc dokümantasyonuna sahip olabilmektedir. Eğer bir metot için Javadoc dokümantasyonu yazıldıysa, bu metodun ne iş yaptığı kod incelenmeden, sadece Javadoc'a bakılarak da anlaşılabilecektir. Uygulamada tespit edilen tüm sınıf ve metotlar, eğer Javadoc dokümantasyonuna sahipler ise sahip

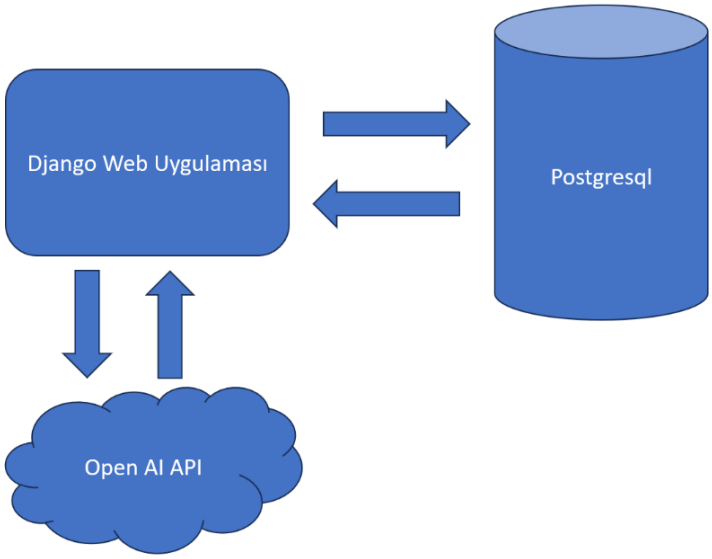
oldukları Javadoc dokümantasyonları da onlarla ilişkilendirilecektir. Böylelikle hangi metot veya sınıfın hangi Javadoc dokümantasyonuna sahip olduğu bilinebilecektir. Bu adımdan sonra hiç Javadoc dokümantasyonuna sahip olmayan metotlar işaretlenecektir. Daha sonraki aşamalarda ise işaretlenmiş olan bu metotlara, metot içeriğine uygun olacak şekilde Javadoc dokümantasyonu üretilecektir.

Şekil 2.8'de verilen diyagram ile önerilen model özetlenmiştir.



Şekil 2.8. Uygulama akış diyagramı

Önceki paragrafta anlatılan tüm bu işlemler geliştirilen bir web arayüzü üzerinden gerçekleştirilecektir. Oluşturulan yapıların birbirleriyle olan ilişkileri ve çıkarılan bilgiler ilişkisel bir veritabanına kaydedilecektir. Geçmişte analiz edilen projelere, silinmediği sürece erişim sağlanabilecektir. Web uygulaması, Django (Django, 2024) kullanılarak geliştirilecektir. Veritabanı olarak ise PostgreSQL (PostgreSQL, 2024) kullanılacaktır. Django ile PostgreSQL arasındaki bağlantı Django'nun sağladığı ORM ile sağlanacaktır. Oluşturulan yapı basit olarak Şekil 2.9'da gösterilmiştir.



Şekil 2.9. Uygulama şeması

2.7.1. Chatgpt api

Javadoc dokümantasyonuna sahip olmayan metotların, Javadoc dokümantasyonu üretilmesi amacıyla işaretleneceği daha önce anlatılmıştı. Javadoc dokümantasyonu üretme işlemi için OpenAI firmasının geliştirdiği ChatGPT API'si kullanılacaktır (ChatGPT, 2024). API'ye yapılan istekten sonra, Javadoc dokümantasyonuna sahip olmayan Java metotları için Javadoc dokümantasyonu üretme işlemi ChatGPT tarafından yapılacaktır. Üretim işlemi, sadece hiç Javadoc dokümantasyonuna sahip olmayan metotlar için yapılacaktır.

Her bir Java metodu farklı uzunlukta ve karmaşıklıkta olabilir. Bu yüzden her bir Java metodu için üretilmesi gereken Javadoc dokümantasyonu miktarının farklı olması idealdir. Şekil 2.10'da verilen Python fonksiyonu ile her bir Java metodu için ne kadar Javadoc dokümantasyonu üretilmesi gerektiği hesaplanmıştır.

```python
def calculate_comment_words(
        method_text: str,
        number_of_code_lines_of_method: int
        ) -> int:

    WORDS_PER_TOKEN = 0.3
    SIMPLIFY_RATE = 10
    MAX_WORDS_PER_SENTENCE = 10
    MIN_WORDS_PER_SENTENCE = 5

    number_of_ideal_comment_lines = number_of_code_lines_of_method * 0.3

    number_of_tokens = lizard.analyze_file.analyze_source_code(
        '*.java', method_text).token_count

    total_words = (number_of_tokens / SIMPLIFY_RATE) * WORDS_PER_TOKEN

    if total_words > (number_of_ideal_comment_lines * MAX_WORDS_PER_SENTENCE):

        total_words = random.randint(
            int(number_of_ideal_comment_lines * MAX_WORDS_PER_SENTENCE),
            int(number_of_ideal_comment_lines * ( 2 * MAX_WORDS_PER_SENTENCE)))

    elif total_words < (number_of_ideal_comment_lines * MIN_WORDS_PER_SENTENCE):

        total_words = random.randint(
            int(number_of_ideal_comment_lines * MIN_WORDS_PER_SENTENCE),
            int(number_of_ideal_comment_lines * MAX_WORDS_PER_SENTENCE))

    total_words = int(total_words + 0.5) # yuvarlama islemi

    return total_words
```

Şekil 2.10. Üretilecek Javadoc kelime sayısını hesaplayan Python fonksiyonu

Şekil 2.10'da verilen Python fonksiyonu iki parametre almaktadır. Fonksiyonun birinci parametresi, kaynak kod olarak Java metodudur. İkinci parametresi ise Java metodunun satır sayısıdır. Fonksiyonun geri dönüş değeri, üretilmesi istenen yorum kelime sayısını ifade eden bir tamsayıdır.

Kullanılan ChatGPT API'sinde, API'nin yapması gereken işlemi söyleyen bir prompt yazılması gerekmektedir. Şekil 2.11'de yazılan prompt gösterilmiştir.

```
Summarize the below Java code in max {tokens} words
and dont start with This Java code term. Use javadoc
comment style and add /** */.\nJava Code: {code}
```

Şekil 2.11. Javadoc dokümantasyonu üretimi için API'ye gönderilecek prompt

Yukarıda verilen prompt'da süslü parantez içerisinde yazılan "tokens", üretilecek Javadoc dokümantasyonundaki kelime sayısını temsil etmektedir. Süslü parantez

içerisinde verilen “code” ise Javadoc dokümantasyonu üretilmek istenen kaynak kod metnini temsil etmektedir.

2.7.2. Yorum sapma yüzdesi indeksi

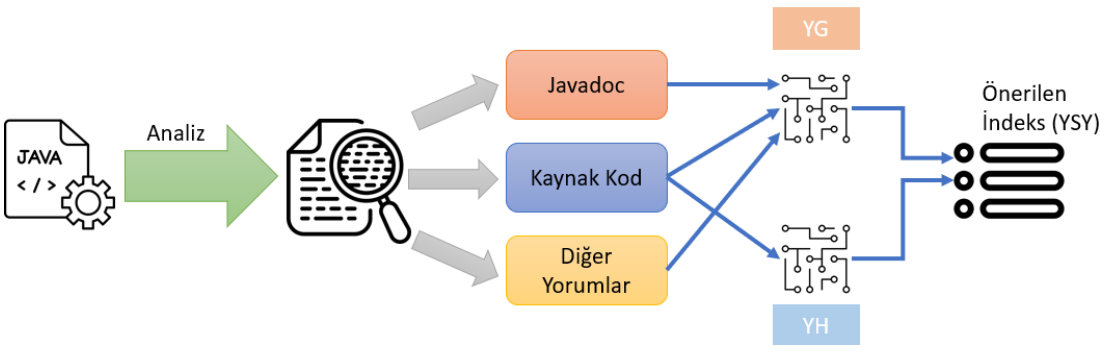
Kod ve yorumlar arasındaki denge, çok sayıda araştırmanın da vurguladığı gibi kodun okunabilirliğini ve anlaşılabilirliğini önemli ölçüde etkilemektedir. Buna rağmen bu konu hakkında net bir kılavuz bulunmamaktadır. Bazı projelerde hiç yorum bulunmazken, bazılarında ise özellikle de açık kaynaklı projelerde, kod satırlarını aşan uzun ve çok miktarda yorum satırları bulunabilmektedir. Bu nedenle, dengeli bir kod-yorum oranına ulaşmak kritik öneme sahiptir.

Bu çalışmada, Javadoc dokümantasyonlarının ve yorum satırlarının miktarının yetersiz, yeterli veya fazla olarak değerlendirilmesini sağlayan, yorum satırları ve kod satırları arasındaki ilişkiyi belirleyen, Yorum Sapma Yüzdesi (YSY) isminde matematiksel bir indeks önerilmiştir (Kilic & Adak, 2024). Bu indeks, denklem 2.2, denklem 2.3 ve denklem 2.4’te verilen formüllerle hesaplanmaktadır. Görsel olarak ise Şekil 2.12’de gösterilmiştir.

$$YG = \frac{Javadoc\ kelime\ sayısı + Diğer\ yorumlar\ kelime\ sayısı}{Metot\ sayısı} \times 0{,}8 \tag{2.2}$$

$$YH = \frac{Dosyadaki\ toplam\ kelime\ sayısı}{Metot\ sayısı} \times 0{,}3 \tag{2.3}$$

$$Yorum\ sapma\ yüzdesi = \frac{100 \times YG}{YH} - 100 \tag{2.4}$$



Şekil 2.12. YSY indeksinin hesaplanması

Yorum sapma yüzdesi indeksi, Javadoc üretimi öncesinde ve sonrasında hesaplanacaktır ve üretim sonrasında yorum satırlarının ideal orana ulaşması

konusunda olumlu bir gelişme olup olmadığı incelenecektir. Bu indeks -100 ile +100 arasında değerler alabilmektedir. İdeal değer ise sıfırdır. Yorum Sapma Yüzdesi indeksi -100'e yaklaştığında az yorum yazıldığı, +100'e yaklaştığında ise gereğinden fazla yorum yazıldığı sonucuna ulaşılacaktır. Sıfıra yaklaştığında ise ideal yorum oranına yaklaştığı anlaşılacaktır.

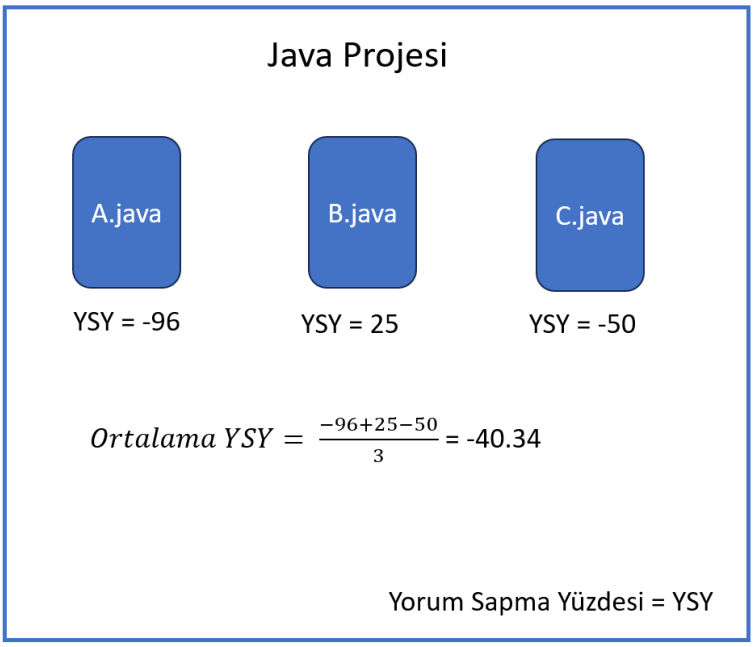
Yukarıda verilen 3 formülde 4 farklı değişken olduğu görülmektedir. Bunlar:

1. Javadoc Kelime Sayısı
2. Diğer Yorumlar Kelime Sayısı
3. Dosyadaki Toplam Kelime Sayısı
4. Metot Sayısı'dır.

Bu listede bulunan değişkenlerden "Javadoc Kelime Sayısı", bir Java dosyasında bulunan tüm Javadoc dokümantasyonlarındaki toplam kelime sayısını, "Diğer yorumlar kelime sayısı", bir Java dosyasında bulunan Javadoc formatında olmayan tüm yorumlardaki toplam kelime sayısını, "Dosyadaki toplam kelime sayısı", bir Java dosyasındaki yorumlar ve kodlar dahil toplam kelime sayısını, "Metot sayısı" ise bir Java dosyasında bulunan toplam metot sayısını ifade etmektedir.

Görüldüğü üzere bütün değişkenler bir Java dosyası kapsamındadır.

Denklem 2.4'teki "Yorum sapma yüzdesi", her bir java dosyası için ayrı ayrı hesaplanacaktır. Bütün java dosyalarının yorum sapma yüzdeleri bulunduktan sonra, tüm proje için aritmetik ortalama ile projenin yorum sapma yüzdesi bulunacaktır (Şekil 2.13.). Yazılım geliştirme sürecinde, yorum üzerinde yapılacak analizler kod kalitesini artırmak için önemlidir. İyi yorumlanmış kodun, bakımı ve anlaşılması daha kolay olur. Ancak aşırı yorumlama da kodun okunabilirliğini azaltabilir. Bu nedenle, Şekil 2.13'teki gibi metrikler, geliştiricilere kodlarının yorum dengesini optimize etmede yardımcı olur.



Şekil 2.13. Ortalama YSY (yorum sapma yüzdesi)

Bizim çalışmamızda, Javadoc dokümantasyonu üretme işleminden sonra yukarıda verilen 4 değişkenden sadece 2 tanesi değişecektir. Diğer iki tanesi ise sabit kalacaktır. Yukarıdaki listede verilen 1 ve 3 numaralı değişkenlerin artı yönde değişmesi beklenmektedir. 2 ve 4 numaralı değişkenler ise sabit kalacaktır. Değişen değerlerle beraber YSY her bir Java dosyası için tekrar hesaplanacaktır. Sonrasında ise daha önce yapıldığı gibi yeni yorum sapma yüzdelerinin aritmetik ortalamaları kullanılarak projenin yeni YSY'si bulunacaktır. Sonuç olarak YSY'nin ideal değer olan sıfıra yaklaşması beklenmektedir.

Geliştirdiğimiz indeksin yanı sıra literatürde, üretilmiş olan farklı indeksler de bulunmaktadır.

A-Index, anlamsal tabanlı bir yaklaşımla anormal kod parçalarının tespit edilmesine odaklanmaktadır (Akimova vd., 2022). İndeksimiz ile arasındaki temel fark, ölçüm hedefinde yatmaktadır; YSY indeksi harici dokümantasyonu ve yorumları değerlendirirken, A-Index dahili kod yapısını incelemektedir.

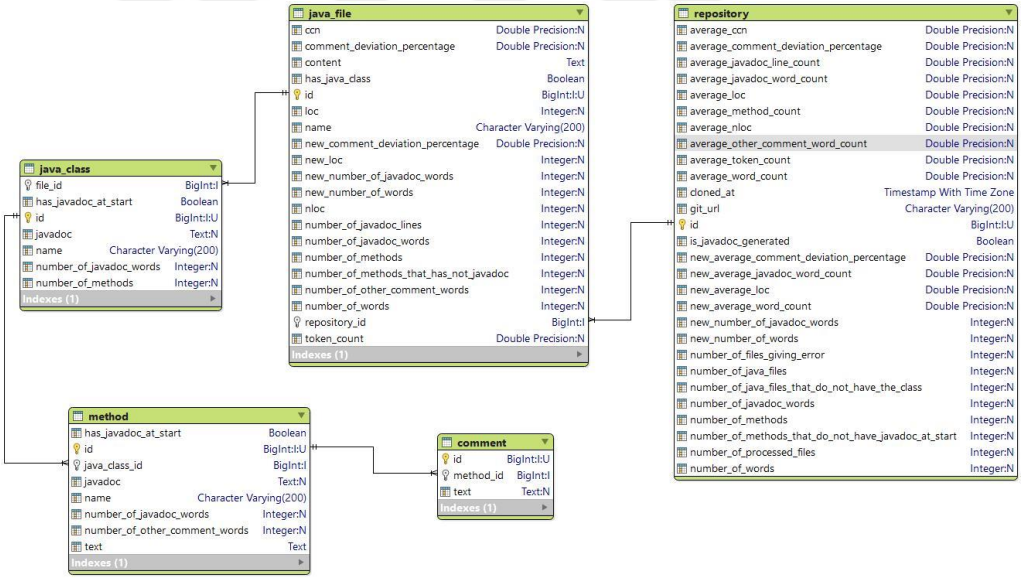
Yoğunluk İndeksi (Intensity Index), kötü kodları sınıflandırarak düzeltilmesi gereken kodları önceliklendirir (Fontana vd., 2015). Kötü yazılmış kodlar, teknik borca veya bakım sorunlarına yol açabilecek potansiyel sorunların göstergeleridir. Bu nedenle, Yoğunluk İndeksi, düzeltilmesi için hangi kod parçalarının öncelikle ele alınması

gerektiğini belirleyen bir filtre görevi görür ve bu sayede zaman içinde kod kalitesini korumak için pratik bir araç haline gelir.

Çoklu Paradigma Metriği (Multi Paradigm Metric), koddaki nesne yönelimli ve prosedürel öğeler arasındaki etkileşimi dikkate alan bir karmaşıklık ölçüm aracıdır (Misra vd., 2013). Çoklu Paradigma Metriği, yazılım kalitesi ve sürdürülebilirliği için gerekli olan kod karmaşıklığına dair bütünsel bir bakış açısı sağlar. Bu metrik, özellikle birden fazla programlama paradigmasının bir arada bulunduğu ortamlarda kullanışlıdır. Çünkü her bir paradigmanın değişen karmaşıklık faktörlerine uyum sağlayabilir.

2.7.3. Veritabanı

Geliştirilecek uygulamada PostgreSQL ilişkisel veritabanı kullanılacağı Şekil 2.9'da gösterilmişti. Veritabanında birbirleriyle bağlantılı toplam 5 tablo kullanılacaktır (Şekil 2.14.).



Şekil 2.14. Uygulamanın varlık bağıntı diyagramı (ERD)

Şekil 2.14'te verilen varlık bağıntı diyagramında, "repository" tablosu ile "java_file" tablosu arasında, "java_file" tablosu ile "java_class" tablosu arasında, "java_class" tablosu ile "method" tablosu arasında ve "method" tablosu ile "comment" tablosu arasında bire çok bağıntı bulunmaktadır.

"comment" tablosu, bir Java metodu içerisindeki yorumları tutacaktır.

“method” tablosu, bir Java metodunun içeriğini, adını, başlangıçta Javadoc dokümantasyonuna sahip olup olmadığını, eğer varsa Javadoc dokümantasyonunu, Javadoc kelime sayısını ve Javadoc formatında olmayan diğer yorumların kelime sayısını tutacaktır.

“java_class” tablosu, bir Java sınıfının adını, başlangıçta Javadoc dokümantasyonuna sahip olup olmadığını, eğer varsa Javadoc dokümantasyonunu, içinde bulunan metot sayısını ve Javadoc kelime sayısını tutacaktır.

“java_file” tablosu, bir Java dosyasının içeriğini, adını, içerisinde sınıf olup olmadığını, CCN değerini, LOC değerini, yorum sapma yüzdesini, Java dosyasında bulunan Javadoc dokümantasyonunun toplam satır sayısını, Java dosyasında bulunan toplam Javadoc kelime sayısını, Java dosyasında bulunan toplam metot sayısını, Java dosyasında bulunan Javadoc dokümantasyonuna sahip olmayan toplam metot sayısını, Java dosyasında bulunan Javadoc formatında olmayan diğer yorumların toplam kelime sayısını, Java dosyasında bulunan toplam kelime sayısını, nloc (dosyada bulunan yorumlar hariç satırı sayısını) değerini, Javadoc üretiminden sonraki Java dosyasının yeni yorum sapma yüzdesini, Javadoc üretiminden sonraki Java dosyasının yeni LOC değerini, Javadoc üretiminden sonraki Java dosyasının yeni Javadoc kelime sayısını ve Javadoc üretiminden sonraki Java dosyasının yeni toplam kelime sayısını tutacaktır.

“repository” tablosu, repository’nin GitHub adresini, klonlanma zamanını, ortalama CCN değerini, ortalama yorum sapma yüzdesini, ortalama Javadoc satır sayısını, ortalama Javadoc kelime sayısını, ortalama LOC değerini, ortalama metot sayısını, ortalama nloc değerini, Javadoc formatında olmayan yorum satırlarının ortalama kelime sayısını, ortalama kelime sayısını, Javadoc üretilme durumunu (üretildi veya üretilmedi şeklinde), sahip olduğu toplam kelime sayısını, işlenen toplam Java dosyası sayısını, sahip olduğu toplam Javadoc’a sahip olmayan metot sayısını, sahip olduğu toplam metot sayısını, sahip olduğu toplam Javadoc kelime sayısını, sahip olduğu ve içerisinde sınıf bulunmayan toplam Java dosyası sayısını, sahip olduğu toplam Java dosyası sayısını, Javadoc üretimi sırasında hata veren toplam Java dosyası sayısını, Javadoc üretiminden sonraki ortalama yorum sapma yüzdesini, Javadoc üretiminden sonraki ortalama Javadoc kelime sayısını, Javadoc üretiminden sonraki ortalama LOC değerini, Javadoc üretiminden sonraki ortalama toplam kelime sayısını, Javadoc üretiminden sonraki ortalama toplam Javadoc kelime sayısını ve Javadoc üretiminden sonraki ortalama toplam kelime sayısını tutacaktır. Ortalama olarak verilen değerler,

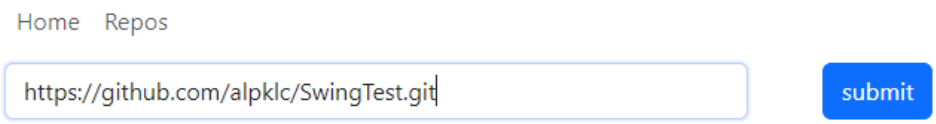
bütün Java dosyalarının sahip olduğu değerlerin aritmetik ortalamasının bulunmasıyla hesaplanacaktır.

Şekil 2.14'te gösterilen varlık bağıntı diyagramında, "java_file" tablosundaki "ccn" alanı döngüsel karmaşıklığı ifade etmektedir. Uygulamada her bir Java dosyası için döngüsel karmaşıklık, "lizard" kütüphanesi yardımıyla ayrı ayrı bulunmuştur (Yin, 2024). "lizard" kütüphanesi, bir Java dosyası için döngüsel karmaşıklığı hesaplarken, Java dosyası içindeki metotların her birinin döngüsel karmaşıklığını ayrı ayrı hesaplar ve tüm metotlar için hesaplanan döngüsel karmaşıklık değerleri toplanarak, Java dosyası için olan döngüsel karmaşıklık değerini bulur. "repository" tablosunda bulunan "average_ccn" alanı ise bütün java dosyalarının döngüsel karmaşıklık değerlerinin aritmetik ortalamasıyla bulunur.

2.7.4. Django web uygulaması

Bu tez kapsamında, istenilen hesaplamaların yapılabilmesi için PostgreSQL veritabanını ve OpenAI API'sini kullanan bir web uygulaması, Python'ın Django çerçevesi ile yazılmıştır. Uygulama tasarımı için Bootstrap kullanılmıştır (*Bootstrap*, 2024).

Web sitesinin, ana sayfasında bir textbox ve bir buton bulunmaktadır. Textbox içerisine analiz edilmek istenen GitHub depo (repository) linki yazılır ve butona tıklanır (Şekil 2.15.).



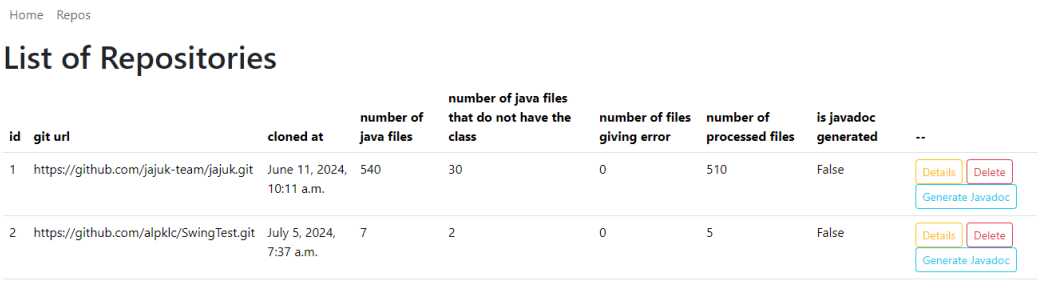
Şekil 2.15. Web uygulamasının ana sayfası

Butona basıldıktan sonra linkteki GitHub deposu klonlanır ve analiz edilir. Analiz sonuçları ekranda gösterilir (Şekil 2.16.).

average_comment_deviation_percentege	-81.666
average_loc	56.0
average_ccn	5.2
average_nloc	41.6
average_token_count	275.0
average_method_count	3.0
average_javadoc_word_count	2.2
average_other_comment_word_count	0.0
average_word_count	138.4
average_javadoc_line_count	0.8
number_of_methods	15
number_of_words	692
number_of_javadoc_words	11
number_of_methods_that_do_not_have_javadoc_at_start	14
number_of_java_files_that_do_not_have_the_class	2
number_of_files_giving_error	0
number_of_processed_files	5

Şekil 2.16. GitHub deposu analiz sonuçları

GitHub deposu analiz edildikten sonra Şekil 2.15'te bulunan navigasyon çubuğundaki "Repos" butonuna tıklanır. Böylece web uygulamasında yeni bir sayfa açılır. Bu sayfada, web uygulaması içerisinde o zamana kadar analiz edilmiş bütün GitHub depolarının listesi bulunmaktadır (Şekil 2.17.).



Şekil 2.17. Analiz edilen GitHub depolarının listesi

Listedeki her GitHub deposunun yanında 3 tane buton bulunmaktadır. "Delete" butonuna tıklandığında, kaydedilmiş olan GitHub deposu ve onunla ilgili tüm bilgiler silinmektedir. "Generate Javadoc" butonuna tıklandığında, hiç Javadoc'a sahip olmayan Java metotları için Javadoc üretme işlemi yapılacaktır. "Details" butonuna tıklandığında, GitHub deposu içerisinde bulunan tüm Java dosyalarının listelendiği bir sayfa gelmektedir (Şekil 2.18.).

id	name	comment deviation percentage	new comment deviation percentage	ratio of methods that has not javadoc at start	-
543	StudentDaoImpl.java	-100.0	None	5 / 5	Details
544	App.java	-8.33	None	0 / 1	Details
545	Student.java	-100.0	None	7 / 7	Details
546	MainScreen.java	-100.0	None	0 / 0	Details
547	DatabaseConnection.java	-100.0	None	2 / 2	Details

Şekil 2.18. GitHub deposu içerisinde bulunan Java dosyalarının listesi

Şekil 2.18'de listelenen her bir satırın sonundaki "Details" butonuna tıklandığında o Java dosyasındaki bulunan Java sınıfları listelenecektir (Şekil 2.19.).

id	class name	number of methods	has javadoc at start	-
539	Student	7	False	Details

Şekil 2.19. Bir Java dosyası içerisinde bulunan Java sınıflarının listesi

Aynı şekilde, Şekil 2.19'da listelenen her bir satırın sonundaki "Details" butonuna tıklandığında o Java sınıfında bulunan Java metotları listelenecektir (Şekil 2.20.).

id	method name	has javadoc at start	-
4232	getName	False	Details
4233	setName	False	Details
4234	getSurname	False	Details
4235	setSurname	False	Details
4236	getNumber	False	Details
4237	setNumber	False	Details
4238	toString	False	Details

Şekil 2.20. Bir Java sınıfı içerisinde bulunan Java metotlarının listesi

Şekil 2.20'de listelenen her bir satırın sonundaki "Details" butonuna tıklandığında o Java metodunun içeriği listelenecektir (Şekil 2.21.).

javadoc was not written and javadoc is not generated

None

```
public String toString() {
        return "Student [name=" + name + ", surname=" + surname + ", number=" + number + "]";
}
```

Şekil 2.21. Bir Java metodunun içeriği

3. BULGULAR VE DEĞERLENDİRME

Java projelerinin analizi için önceki bölümde anlatılan Python programlama dilinin Django çerçevesi ile yazılmış olan web uygulaması kullanılmıştır. Web uygulaması yardımıyla analiz işlemleri otomatikleştirilmiş ve hızlandırılmıştır. Elde edilen sonuçlar PostgreSQL veritabanına kaydedilmiştir.

Bu çalışmada analiz amacıyla 4 farklı GitHub deposu kullanılmıştır. Bu kod depoları, açık kaynaklı olmalarının yanı sıra çok büyük oranda Java programlama dili ile yazılmışlardır. Analiz edilen GitHub depolarının URL'leri aşağıdaki listede verilmiştir:

1. https://github.com/jajuk-team/jajuk.git (jajuk-team/jajuk)
2. https://github.com/mandubian/siena.git (mandubian/siena)
3. https://github.com/ekoontz/hadoop-common.git (ekoontz/hadoop-common)
4. https://github.com/apache/commons-math.git (apache/commons-math)

Buradan sonraki bazı kısımlarda GitHub depolarını ifade etmek için yukarıdaki listedeki numaralar da kullanılacaktır. Örneğin 2 numaralı GitHub deposu denildiğinde "https://github.com/mandubian/siena.git" deposundan bahsedilmiş olacaktır.

Listenin her bir elemanının sonunda verilmiş parantez içerisindeki bilgi, o satırdaki GitHub linki için sırasıyla GitHub kullanıcı adı ve GitHub deposudur. Örneğin listenin birinci elemanı için verilmiş olan "jajuk-team/jajuk" ifadesinde "jajuk-team" GitHub kullanıcı adını ifade ederken, "jajuk" ise GitHub deposunu ifade etmektedir.

Javadoc üretme işlemi için ChatGPT API'si kullanılmıştır. Şekil 3.1'de, Java'da bir akışı kapatan fakat kendisi için Javadoc dokümantasyonu yazılmamış bir Java metodu verilmiştir.

```java
public synchronized void close() throws IOException {
  if (closed) {
    throw new IOException("Stream closed");
  }
  super.close();
  closed = true;
  if (!client.isConnected()) {
    throw new FTPException("Client not connected");
  }
  boolean cmdCompleted = client.completePendingCommand();
  client.logout();
  client.disconnect();
  if (!cmdCompleted) {
    throw new FTPException("Could not complete transfer, Reply Code - "
        + client.getReplyCode());
  }
}
```

Şekil 3.1. Javadoc'a sahip olmayan bir Java metodu

Şekil 3.1'de verilen java metodu, 3 numaralı GitHub deposunun "FTPInputStream.java" dosyasında bulunmaktadır. Bu metot için üretilen Javadoc Şekil 3.2'de verilmiştir. Üretim işlemi için kullanılan prompt, ikinci bölümde bulunan Şekil 2.11'de verilmiştir. Bu prompt içerisine yazılan "Use javadoc comment style and add /** */." ifadesi, üretilen yorumun javadoc standardında olmasını sağlamıştır. Benzer şekilde Şekil 2.10'da verilen Python fonksiyonu, Şekil 3.2'de üretilen Javadoc dokümantasyonunun uzunluğunu belirlemiştir.

```java
/**
 * Closes the stream in a synchronized manner. Throws an IOException if the stream is already closed.
 * Logs out the client, disconnects, and checks if the command was completed successfully.
 * Throws an FTPException if connection or transfer completion fails.
 */
```

Şekil 3.2. Şekil 3.1'deki metot için üretilen Javadoc

Şekil 3.2'de, üretilen Javadoc dokümantasyonun, Şekil 3.1'de verilen Java metodunun yaptığı görevleri açık bir şekilde anlattığı görülmektedir.

Analiz işlemlerinin yapılabilmesi için GitHub depolarının klonlanması gerekmektedir. Tablo 3.1'de görüldüğü gibi ilk 3 GitHub deposu için klonlanma tarihi 2 Mayıs 2024 iken 4 numaralı GitHub deposu için 3 Mayıs 2024'tür. Klonlanma işleminin ardından hemen analiz işlemlerine geçildiği için analiz tarihleri ile klonlanma tarihleri aynıdır.

Tablo 3.1. Analiz edilen Github depolarının commit sayıları ve klonlanma tarihleri.

No	GitHub deposu	Commit sayısı	Klonlanma tarihi (Analiz tarihi)
1	jajuk-team/jajuk	6533	2 Mayıs 2024
2	mandubian/siena	274	2 Mayıs 2024
3	ekoontz/hadoop-common	422	2 Mayıs 2024
4	apache/commons-math	7181	3 Mayıs 2024

Tablo 3.1'de, üzerinde çalışılan GitHub depolarının "commit" sayıları da verilmektedir. Bir GitHub deposundaki "commit" sayısı metriği, o kod deposundaki yapılan toplam kayıtlı değişiklikleri ifade eder. Bu değişiklikler, dosya eklemesi, dosya silinmesi, dosyada değişiklik yapılması vb. olabilir. Her "commit", bir veya birden fazla dosyada yapılan değişikliklerin, belirli bir zaman noktasında kaydedildiği bir birim olarak düşünülebilir. "Commit" sayısı, bir yazılım projesinin geçmişini ve geliştirme faaliyetlerinin yönünü anlamada önemli bir metriktir. Fakat kapsamlı bir değerlendirme için sadece "commit" sayısı metriği yeterli olmayacaktır. "Commit" sayısının yanı sıra, "commit" kalitesi, "commit" mesajlarının niteliği ve projenin genel yapısı birlikte değerlendirilmelidir. Ayrıca ekip halinde geliştirilen yazılım projelerinde, ekibin farklı üyelerinin projeye olan katkılarının belirlenmesinde "commit" metriğinden yararlanılabilir. GitHub üzerinde bulunan açık kaynaklı büyük projelerin "commit" sayılarının yüksek olması da sık karşılaşılan bir durumdur. Bunun sebebi olarak projeye erişimin serbest olması gösterilebilir.

Buradan sonra bu bölüm içerisinde verilen tablolardaki GitHub depoları ile ilgili bilgiler, Tablo 3.1'in son sütununda verilen tarihlerdeki analiz işlemlerinin sonuçlarıdır.

GitHub depolarında bulunan Java projelerinin analiz işlemi, projeyi oluşturan Java dosyalarının analiz edilmesiyle gerçekleştirilecektir. Bunun için projede bulunan bütün Java dosyaları tespit edilmiştir. Tespit edilen Java dosyalarından ise gerekli şartları taşımayanlar çıkarılmıştır. Tablo 3.2'de her bir GitHub deposundan kaç tane Java dosyasının analiz edildiği gösterilmiştir. Her Java uzantılı dosyanın içerisinde

Java sınıfı bulunmak zorunda olmadığı için içerisinde Java sınıfı bulunmayan dosyalar analiz edilecek dosya kümesinden çıkarılmıştır. Analiz sırasında çeşitli sebeplerden hata veren Java dosyaları da olabilmektedir. Bundan dolayı bu dosyalar da analiz edilecek dosya kümesinden çıkarılmıştır. Sonuç olarak Tablo 3.2'nin son sütunu, ilgili GitHub deposundan kaç tane Java dosyasının analiz edildiğini göstermektedir. Bu değer toplam Java dosya sayısından, içerisinde sınıf bulunmayan toplam Java dosya sayısının ve analiz sırasında hata veren toplam Java dosya sayısının çıkarılmasıyla elde edilmiştir.

Tablo 3.2. Analiz edilen GitHub depolarındaki dosya sayıları.

No	GitHub deposu	Toplam Java dosya sayısı	İçerisinde sınıf bulunmayan toplam Java dosya sayısı	Analiz sırasında hata veren toplam Java dosya sayısı	Analiz edilen toplam Java dosya sayısı
1	jajuk-team/jajuk	540	30	0	510
2	mandubian/siena	328	65	0	263
3	ekoontz/hadoop-common	632	74	0	558
4	apache/commons-math	1157	193	1	963

Bahsedilen GitHub depoları analiz edildikten sonra Javadoc'a sahip olmayan Java metotları tespit edildi. Bu metotların gövdeleri, metot imzaları da dahil olmak üzere, Javadoc üretimi amacıyla ChatGPT API'sine gönderildi. Üretilen her bir Javadoc, üretildiği Java metoduyla ilişkilendirildi. Tablo 3.3'te Javadoc üretiminden önceki ve sonraki toplam kelime sayıları ve toplam Javadoc kelime sayıları karşılaştırmalı olarak verilmiştir. Görüldüğü üzere üretim sonrası toplam kelime sayıları artmıştır.

Tablo 3.3. Analiz edilen GitHub depolarındaki kelime sayıları.

No	GitHub Deposu	Toplam Javadoc kelime sayısı	Toplam kelime sayısı	Yeni toplam Javadoc kelime sayısı	Yeni toplam kelime sayısı
1	jajuk-team/jajuk	47377	406765	68677	428313
2	mandubian/siena	7041	209952	80822	284565
3	ekoontz/hadoop-common	56749	422850	88805	455585
4	apache/commons-math	165717	900090	248767	984164

Javadoc dokümantasyonu üretimi sadece Java metotları için yapılmıştır. Dolayısıyla bir GitHub deposu içerisinde bulunan toplam metot sayısının bilinmesi faydalı olacaktır. Tablo 3.4'te, toplam metot sayıları ve Javadoc'a sahip olmayan metot sayıları verilmiştir. Örneğin 3 numaralı GitHub deposu, toplam 4263 metoda sahiptir. Bu metotlardan 2371 tanesine Javadoc dokümantasyonu yazılmamıştır. Dolayısıyla 3 numaralı GitHub deposunun 2371 metodu için Javadoc dokümantasyonu üretilecektir.

Tablo 3.4'ün son iki sütunu, önerdiğimiz indeks olan YSY ile ilgilidir. "Ortalama YSY" değeri, -100 ile +100 arası değer alabilmektedir. İdeal değeri ise 0'dır. Analiz edilecek GitHub deposu için "Ortalama YSY" değeri, ikinci bölümdeki Şekil 2.12'de bahsedildiği gibi hesaplanmaktadır. "Ortalama YSY (%)" sütunu GitHub deposunun aslının yorum sapma yüzdesini vermektedir. Son sütun ise Javadoc üretimi gerçekleştirildikten sonraki ortalama YSY değerini vermektedir. Görüldüğü üzere, Javadoc üretim işlemi gerçekleştirildikten sonra ortalama YSY değerlerinde %20 ile %60 arası iyileşmeler olmuştur.

Tablo 3.4. Analiz edilen GitHub depolarındaki eski ve yeni YSY değerleri.

No	GitHub deposu	Toplam metot sayısı	Javadoc'a sahip olmayan toplam metot sayısı	Ortalama YSY (%)	Yeni Ortalama YSY (%)
1	jajuk-team/jajuk	4225	899	-55	-43.3
2	mandubian/siena	4411	4315	-83.75	-33.16
3	ekoontz/hadoop-common	4263	2371	-63.32	-47.89
4	apache/commons-math	8230	3393	-55.19	-38.28

En çok iyileşmenin 2 numaralı GitHub deposunda olduğu görülmektedir. Bunun sebebi ise neredeyse hiçbir metodu için Javadoc dokümantasyonu yazılmamış olmasıdır. Aynı şekilde, oransal olarak metotları için en çok Javadoc dokümantasyonu yazılmış olan 1 numaralı GitHub deposundaki iyileşmenin en az olduğu görülmektedir. Tablo 3.5'te, analiz edilen GitHub depolarının, YSY indeksine göre iyileşme yüzdeleri verilmiştir.

Tablo 3.5. Analiz edilen Github depolarının YSY indekslerindeki iyileşme yüzdeleri.

No	GitHub deposu	İyileşme oranı (%)
1	jajuk-team/jajuk	21.27
2	mandubian/siena	60.41
3	ekoontz/hadoop-common	24.37
4	apache/commons-math	30.64

Tablo 3.6'daki bilgiler "lizard" kütüphanesi ile hesaplanmıştır. Bu tablodaki CCN, NLOC ve simge sayısı değerleri, bir GitHub deposu içerisinde bulunan tüm java dosyalarının CCN, NLOC ve simge sayısı değerlerinin aritmetik ortalamasıdır.

Tablo 3.6. Analiz edilen GitHub depolarının CCN değeri ve diğer bilgileri.

No	GitHub deposu	Ortalama CCN	Yorum satırları hariç ortalama satır sayısı (NLOC)	Ortalama simge (token) sayısı
1	jajuk-team/jajuk	23.21	130.24	992.13
2	mandubian/siena	39.56	197.47	1513.35
3	ekoontz/hadoop-common	24.8	130.31	919.2
4	apache/commons-math	22.85	151.11	1356.27

Tablo 3.6'ya göre 2 numaralı GitHub deposunun döngüsel karmaşıklık değeri (CCN) diğer 3 GitHub deposundan fazla çıkmaktadır. Bununla orantılı olarak, aynı şekilde, 2 numaralı GitHub deposunun NLOC ve "simge sayısı" değerleri de diğer 3 GitHub deposundan fazla çıkmaktadır. Buradan döngüsel karmaşıklık değerinin (CCN), NLOC ve "simge sayısı" değerleriyle doğru orantılı olduğu sonucu çıkmaktadır. Buna ek olarak 2 numaralı GitHub deposundaki metotlar, diğer 3 GitHub deposundaki metotlara göre daha karmaşıktır.

Kod karmaşıklığının artması, doğru orantılı olarak kodun ne yaptığını anlatan dokümantasyon yazılması ihtiyacını arttırır. Çünkü kod karmaşıklığı arttıkça kodun anlaşılması da zorlaşmaktadır. Bu çalışmada önerilen model ile en iyi iyileştirme, karmaşıklık seviyesi en yüksek olan 2 numaralı GitHub deposunda sağlanmıştır.

Literatürde, geliştirdiğimiz YSY indeksine benzer şekilde, yazılım geliştirme ve kalite ölçümü alanında, kaynak kodun çeşitli yönlerini değerlendirmek için çeşitli indeksler geliştirilmiştir. Bu indeksler genellikle anomalileri tespit etmeye, kod karmaşıklığını tahmin etmeye veya kötü yazılmış kodları belirlemeye odaklanır. Yazılım kaynak

kodu içindeki yorumların yeterliliğini ölçmek için geliştirdiğimiz YSY indeksini, Tablo 3.7'de sunulan ve literatürde bulunan diğer mevut indekslerle karşılaştıracağız.

Tablo 3.7. Diğer çalışmalardaki indekslerle YSY indeksinin karşılaştırılması.

İndeksler	Odak Noktası	Hedef
YSY İndeksi	Kod yorumları	Dahi iyi dokümantasyon
(Akimova vd., 2022)	Kod parçaları	Hata tespiti
(Fontana vd., 2015)	Kötü kod	Bakım tahmini
(Misra vd., 2013)	Kod elemanları	Kod çoğaltma tespiti

Bu bölümde indeksimizin, Tablo 3.7'deki makalelerde bahsedilen indekslerle karşılaştırmalı bir analizi sunulacak ve her birinin güçlü yönleri, sınırlamaları ve potansiyel uygulamaları vurgulanacaktır. Yorum tabanlı indeksimiz, bir kaynak kod tabanındaki yorumların yeterliliğini ve alaka düzeyini değerlendirmek için tasarlanmıştır. Bunun altında yatan hipotezimiz, iyi yorumlanmış kodun yalnızca okunabilirliği ve sürdürülebilirliği artırmakla kalmayıp aynı zamanda kaliteli bir yazılımın kritik bir bileşeni olduğudur. İndeksimiz, yorum miktarı ve kod miktarı oranının analiz edilmesiyle hesaplanmaktadır.

İndeksimizi, A-Indeks (Akimova vd., 2022) ile karşılaştırdığımızda, her iki indeksin de kod kalitesini iyileştirmek gibi ortak bir hedefi paylaştığını görüyoruz. A-Indeks bunu, daha fazla inceleme veya yeniden düzenleme gerektirebilecek olağandışı kodları işaretleyerek yaparken, bizim indeksimiz yeterli yorum yapılmasını sağlayarak kodun anlaşılabilirliğini geliştirir ve bakımını kolaylaştırır. A-Indeks'in etkinliği, Python depolarında gerçekleştirilen hata düzeltmeleriyle ilişkili kod değişikliklerini başarıyla tespit etmesi yoluyla doğrulanmıştır. A-Indeks'in kod anomalilerini tespit etmede başarılı olmasına rağmen, tespit edilen anomalilerin ileride başvurulmak üzere yeterince dokümante edilmesini sağlayan bizimki gibi tamamlayıcı bir indeksten potansiyel olarak faydalanabileceğini göstermektedir.

İndeksimiz, Yoğunluk İndeksi (Fontana vd., 2015) ile karşılaştırıldığında, indeksimizin kötü yazılmış kodlar ile doğrudan etkileşime girmediğini, bunun yerine kod yapıları arkasındaki gerekçenin iyi dokümante edilmesini sağlayarak Yoğunluk

İndeksi gibi araçları tamamladığını görüyoruz. Geliştiriciler için yeterli yorum ve dokümantasyon miktarının sağlanmasıyla bazı kod parçalarının neden kötü yazıldığını ve bunların teknik borçlanmayı mı temsil ettiğini yoksa sadece gerekli karmaşıklığın bir sonucu mu olduğunun anlaşılması sağlanabilir. Ek olarak indeksimiz, kötü yazılmış kodların düzeltilmesi çalışmalarının sonuçlarını dokümante etmek için kullanılabilir. Böylece gelecekteki bakım görevleri için çok değerli olabilecek belirli kararların neden alındığına dair bir kayıt tutulmuş olacaktır.

Çoklu Paradigma Metriği (Misra vd., 2013), kod karmaşıklığının kapsamlı bir değerlendirmesini sunarken, önerdiğimiz indeks, karmaşık kodlara eşlik eden insan tarafından okunabilir dokümantasyona odaklanmaktadır. Çoklu Paradigma Metriği'nin yüksek karmaşıklığa işaret ettiği projelerde, açık ve yeterli yorumlara duyulan ihtiyaç daha da kritik hale gelmektedir. Yazılım projelerinde yazılan karmaşık kodlar, geliştirme sürecinin devam etmesiyle, zaman içinde anlaşılması zor bir hale gelmeye eğilimlidir. İndeksimiz, bu tür kodların iyi bir şekilde dokümante edilmesine yardımcı olacak ve böylece geliştiricilerin, karmaşık bölümleri hatalara yol açmadan anlamasını ve değiştirmesini kolaylaştıracaktır. Ayrıca, Çoklu Paradigma Metriği, karmaşıklığı değerlendirmek için kullanıldığında, indeksimiz bu karmaşıklığın yorumlar aracılığıyla ne kadar iyi aktarıldığını değerlendirmek için paralel bir ölçüt olarak kullanılabilir.

Karşılaştırma, her bir indeksin kendine özgü güçlü yanları olsa da birbirleriyle çelişmediklerini vurgulamaktadır. Geliştirdiğimiz YSY indeksi, kalitesi veya karmaşıklığı ne olursa olsun, kodun yeterince dokümante edilmesine katkı sağlayarak A-İndeks, Yoğunluk İndeksi ve Çoklu Parametre Metriği gibi indekslerin etkinliğini artıran tamamlayıcı bir araç olarak görülebilir. Bu, dokümantasyon, uzun vadeli kod bakımı, ekip iş birliği ve bilgi aktarımı için özellikle de büyük ölçekli projelerde hayati önem taşır.

Dahası, indeksimiz, kod kalitesini değerlendiren mevcut araçlara ve çerçevelere entegre edilebilir. Böylece yorumlar ve dokümantasyonun yeterliliğine odaklanan ek bir değerlendirme katmanı oluşturulmuş olur.

Bu entegrasyon, yalnızca kodla ilgili sorunları tespit etmekle kalmayıp aynı zamanda yeterli dokümantasyon yoluyla kodun arkasındaki mantığın korunmasını sağlayan daha bütünsel yazılım kalitesi değerlendirme araçlarının oluşturulmasını

kolaylaştırabilir. Sonuç olarak, indeksimiz Tablo 3.7'de tartışılan indekslerden farklı bir amaca hizmet etse de yazılım kalitesinin genellikle göz ardı edilen önemli bir yönünü ele alarak onları tamamlamaktadır. Yeterli yorumlar, yazılımın sürdürülebilirliğini, okunabilirliğini ve genel kalitesini önemli ölçüde artırabilir, bu da indeksimizi yazılım kalitesi metrikleri araç kümesine yapılmış değerli bir katkı haline getirmektedir.

4. TARTIŞMA VE SONUÇ

Yazılım projelerinin başarısı, sadece yazılan kodun işlevselliğiyle ilgili değil aynı zamanda kodun anlaşılabilmesi, bakımının yapılabilmesi ve üzerinde yeni geliştirmeler yapılabilmesine yardımcı olan kaliteli yorum satırlarının yazılmasıyla da yakından ilgilidir. İyi yazılmış yorum satırları, kodun anlaşılabilirliğini, hata (bug) tespit etme işleminin hızlandırılmasını ve takım çalışmasını kolaylaştırmaktadır. Bu nedenle yeterli yorum yazılması yazılım projeleri için önemli bir kriterdir.

Bir yazılım projesinde yetersiz yorum yazılması, projenin bakımını zorlaştırır ve gelecek geliştirmeler önünde engeller oluşturur. Bir kodun neden yazıldığını, nasıl çalıştığını ve potansiyel hatalarının nerede olabileceğini açıklayan yorum satırları yazılmadığı zaman o kodun anlaşılması ve değiştirilmesi çok zor olmaktadır. Bu durum, yazılım projesinin uzun dönemli başarısını tehdit etmektedir. Bu çalışmada geliştirilen indeks, popüler GitHub depoları üzerinde, yorum satırlarının yeterliliği konusunda denenmiş ve olumlu sonuçlar vermiştir.

Bir Java projesinde Javadoc yazılması, başarılı bir proje dokümantasyonu oluşturulması için ön şarttır. Java projesinde bulunan sınıf, arayüz, metot ve diğer bileşenler için yazılan Javadoc dokümantasyonları, bu bileşenlerin görevlerini ve işlevlerini açıklayarak, yazılım geliştiricilerin kodu anlamalarını, üzerine ekleme yapmalarını veya değiştirmelerini kolaylaştırarak geliştirme sürecinin kalitesini artıracaktır. Bu da yazılım projesinin anlaşılabilirliğine ve sürdürülebilirliğine olumlu katkı sağlayacaktır.

İyi yazılmış yorum satırları kod bakımını kolaylaştırır, hataları bulmayı hızlandırır ve eğer bir ekip halinde çalışılıyorsa, ekibin etkili bir şekilde iş birliği yapmasına olanak tanır. Bu nedenle ekip halinde çalışılan projelerde yorum kalitesinin arttırılmasına yönelik önlemler alınmalı ve ekibin yorum standartlarına uyması teşvik edilmelidir. Yorum yazma işlemini otonom hale getirmek ise bu süreçleri kolaylaştıracaktır. Yorum yazma işleminin otomatize edilmesiyle geliştiriciler sadece kod yazmaya odaklanabilecek ve zamandan tasarruf edilebilecektir. Ayrıca el ile yazılan yorum satırlarının hataya meyilli olmasının yanı sıra, projede var olan bir kodun çeşitli

sebeplerden dolayı değiştirildiği durumlarda, hali hazırda kodu açıklayan yorum satırlarının güncellenmesinin unutulması da çok muhtemeldir. Bu durum hiç yorum yazılmamasından bile daha kötü olabilmektedir.

Bu çalışmada, GitHub üzerinde bulunan, açık kaynaklı Java projeleri incelenmiş ve yorum üretimi başlığı altında Java'ya özgü bir yorum türü olan Javadoc üretimi üzerinde durulmuştur. Javadoc, klasik yorum satırlarından farklı olarak daha çok dokümantasyon amacıyla kullanılmaktadır. Java programlama dilinde bulunan sınıf, arayüz, metot vb. yapıların üzerine yazılan Javadoc dokümantasyonları, kendine özgü bir sözdizimine ve etiketlere sahiptir. Ancak bu tez kapsamında Java'daki sınıf, arayüz gibi bileşenler yerine sadece metotlara ait Javadoc üretimine yoğunlaşılmıştır. Ayrıca bir Java metodu, hiç Javadoc dokümantasyonuna sahip değilse onun için Javadoc üretimi yapılmıştır. Hiç Javadoc dokümantasyonuna sahip olmayan Java metotları tespit edilmiş ve bu metotlara yeteri kadar Javadoc üretilmesine çalışılmıştır. Bundan sonraki aşama ise bir Java metodu için yeteri kadar Javadoc dokümantasyonun ne kadar olduğunun belirlenmesidir. Bu ise Javadoc üretilecek Java metodunun, çeşitli parametrelerine bakılarak hesaplanmıştır. Sonuç olarak üretilmek istenen Javadoc miktarı, tamsayı bir değer olarak hesaplanmış ve üretici fonksiyona gönderilmiştir.

Yorum satırı üretimi, Javadoc özelinde, hatta daha da özel olarak Java programlama dilindeki metotlar özelinde yapılırken, bir Java projesinin yorum satırı yeterliliği tüm yorumlar özelinde hesaplanmıştır.

Kaynak kod içerisinde, yorum satırı yazılması, kodun anlaşılabilirliğini ve bakımını kolaylaştırdığı için genellikle olumlu bir durum olarak kabul edilmektedir. Ancak, belirli bir miktardan fazla yorum satırı yazılması, kodun okunabilirliğini olumsuz yönde etkileyebilir.

Bu çalışmanın en önemli katkısı, bir Java projesinin yorum satırı yeterliliğini değerlendirebilmek için YSY indeksinin önerilmesidir. YSY indeksi, bir projenin yorum satırı miktarının az, yeterli veya çok kategorilerinden hangisinde olduğunu matematiksel olarak ölçmek için geliştirilmiştir. Bu indeks, birçok dosyadan oluşan bir Java projesinin yorum yüzdesini değerlendirerek, projenin yorum satırı yeterliliğini nicel bir şekilde belirlemeyi sağlar.

YSY indeksi sayesinde, Java projeleri matematiksel olarak değerlendirilebilir hale gelmiştir. Bu indeks, farklı Java projelerinin yorum yüzdesi açısından

karşılaştırılmasına olanak tanımanın yanı sıra, bir projenin farklı zamanlardaki durumunu analiz etmek için de kullanılabilir. Böylece, projelerin yorum satırı yeterliliğinin, zaman içinde nasıl değiştiği izlenebilir.

Özetle, YSY indeksi, yazılım projelerinin kalitesini ve sürdürülebilirliğini artırmak amacıyla geliştirilmiş önemli bir araçtır. Bu indeksin, yazılım projelerinin hem kendi içlerinde hem de diğer projelerle kıyaslanarak daha iyi anlaşılmasına ve yönetilmesine katkı sağlaması beklenmektedir. YSY indeksinin yeterliliği, yerine getirdiği ikili işlevden kaynaklanmaktadır. Bir yandan, halihazırda mevcut olan yazılım kalite ölçütleri kümesine sorunsuz bir şekilde entegre olabilirken öte yandan, odağı yazılım kalitesinin şimdiye kadar ihmal edilmiş bir yönüne kaydırarak bu kümeyi ciddi biçimde dönüştürme potansiyeline sahiptir.

Ölçüm olarak izlenen yöntem ise şöyledir: Analiz edilmek istenen Java projesinin YSY indeksi, önce orijinal haliyle ölçülmüştür. Sonra gerekli Javadoc dokümantasyonları üretilmiş ve YSY indeksi tekrar hesaplanmıştır. Daha sonra ise eski ve yeni YSY indeksleri arasındaki fark değerlendirilmiştir. Bu yöntem, Java projesi içeren dört farklı GitHub deposu üzerinde uygulanmıştır. Her bir depo için, öncelikle mevcut haliyle YSY indeksi hesaplanmış, ardından Javadoc dokümantasyonları eklenmiş durumlarındaki değerlerle indeks yeniden hesaplanmıştır. Bu süreç sonunda elde edilen veriler, eski ve yeni YSY indeksleri arasındaki farkı ortaya koymuştur. Sonuçlar, dört Java projesinin de YSY indekslerinde belirgin bir iyileşme olduğunu göstermiştir.

Bu araştırma, işlevsellik, güvenilirlik, kullanılabilirlik, verimlilik, sürdürülebilirlik ve taşınabilirlik gibi kalite özelliklerinin sağlam, etkili ve kullanıcı dostu yazılımların geliştirilmesinde oynadığı kritik rolü ortaya koymaktadır. Bu niteliklerin yazılım geliştirme topluluğu içinde ne kadar iyi ifade edildiğinin anlaşılması da aynı derecede önemlidir. Önerdiğimiz YSY indeksi, özellikle yazılım kalitesi değerlendirme araçlarında sıklıkla ihmal edilen bir husus olan yazılımın okunabilirliğini, sürdürülebilirliğini ve genel kalitesini büyük ölçüde artırabilecek optimum kod-yorum dengesini türetmeye odaklanarak yeni bir bakış açısı sunmaktadır. Öte yandan, bu çalışmada ele alınan A-İndeks, Yoğunluk İndeksi ve Çoklu Paradigma Metriği gibi indeksler, kod karmaşıklığını tahmin etmenin yanı sıra, kod içerisinde anomali tespiti, kötü kodların tespiti ve yazılım kalitesini değerlendirmek gibi farklı işlevlere sahiptirler. İndeksimizin teşvik edici yönü, bu indekslerle uyum içinde olması ve

başarılı bir şekilde kullanılabilmesinde yatmaktadır. Sadece kod anomalilerinin veya kötü yazılmış kodların anlaşılmasını sağlayarak değil, aynı zamanda bu tür kod yapılarının arkasındaki temel mantığı dokümante ederek de onları tamamlar. Sonuç olarak, yalnızca geliştiricilerin mevcut kodu daha iyi anlamalarını sağlamakla kalmaz, aynı zamanda gelecekteki geliştirme ve hata ayıklama ihtiyaçları için karmaşık kod yapılarının arkasındaki mantığın başarılı bir şekilde çıkarılmasını sağlar. Dolayısıyla indeksimiz, yazılım kalitesinin iyileştirilmesinde göz ardı edilemeyecek bir araç olarak ortaya çıkmaktadır.

Önerdiğimiz YSY indeksinin sürekli geliştirilmesi ve iyileştirilmesi için geleceğe dair beklentilerimiz olumludur. Sürekli gelişen yazılım geliştirme paradigmalarının karmaşıklığına uyum sağlayabilen önerdiğimiz indeks, yazılım kalite ölçütlerine katkıda bulunmaya hazırdır. Pratik anlamda, önerilen indeks mevcut yazılım geliştirme altyapılarına entegre edilebilir ve bu sayede uzun vadeli kod bakımı, ekip iş birliği ve bilgi aktarımı için değerli bir araç haline gelebilir. İndeksimiz, özellikle, sistem gelişiminin ve iş gücü kaybının yüksek olasılıkla yaşanabileceği büyük ölçekli projelerde daha önemli bir hale gelmektedir. Bu nedenle, önerilen indeks potansiyel olarak, kodun anlaşılabilirliğinin zayıf olması nedeniyle projenin terk edilmesinin önlenmesine katkıda bulunmaktan, eski sistemlerin güncellenmesine ve bakım süreçlerinin kolaylaştırılmasına kadar geniş kapsamlı etkilere sahiptir.

Yapılan bu çalışmada, Javadoc dokümantasyonlarının, yazılım projelerindeki sürdürülebilirlik seviyesini artırdığı tespit edilmiştir. Elde edilen bulgular, dokümantasyonun, yazılım projelerinin anlaşılabilirliğini, bakımını ve geliştirilebilirliğini olumlu yönde etkilediğini ve YSY'de gözle görülür bir iyileşme sağladığını kanıtlamaktadır. Bu durum, yazılım projelerinde dokümantasyonun önemini bir kez daha vurgulamakta ve iyi dokümantasyon yazılmış kodların uzun vadede daha sürdürülebilir olduğunu göstermektedir.

KAYNAKLAR

Abbes, M., Khomh, F., Gueheneuc, Y.-G., & Antoniol, G. (2011). An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. 2011 15th European Conference on Software Maintenance and Reengineering, 181-190. https://doi.org/10.1109/CSMR.2011.24

Adak, M. F. (2018). Software defect detection by using data mining based fuzzy logic. 2018 Sixth International Conference on Digital Information, Networking, and Wireless Communications (DINWC), 65-69. https://doi.org/10.1109/DINWC.2018.8356997

Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2020). A Transformer-based Approach for Source Code Summarization. arXiv. http://arxiv.org/abs/2005.00653

Akimova, E., Bersenev, A., Cheshkov, A., Deikov, A., Kobylkin, K., Konygin, A., Mezentsev, I., & Misilov, V. (2022). A-Index: Semantic-based Anomaly Index for Source Code. Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering, 259-266. https://doi.org/10.5220/0010984600003176

Amaro, R., Pereira, R., & da Silva, M. M. (2023). Capabilities and Practices in DevOps: A Multivocal Literature Review. IEEE Transactions on Software Engineering, 49(2), 883-901. https://doi.org/10.1109/TSE.2022.3166626

Apache Software Foundation. (2024, Temmuz 4). JMeter. https://jmeter.apache.org/

Bilgin, Z., Ersoy, M. A., Soykan, E. U., Tomur, E., Comak, P., & Karacay, L. (2020). Vulnerability Prediction From Source Code Using Machine Learning. IEEE Access, 8, 150672-150684. https://doi.org/10.1109/ACCESS.2020.3016774

Bootstrap. (2024, Temmuz 4). Get started with Bootstrap. https://getbootstrap.com/

ChatGPT. (2024, Haziran 25). OpenAI developer documentation. https://openai.com/api/

Ciurumelea, A., Alexandru, C. V., Gall, H. C., & Proksch, S. (2023). Completing Function Documentation Comments Using Structural Information. Empirical Software Engineering, 28(4), 86. https://doi.org/10.1007/s10664-022-10284-6

Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using metrics to evaluate software system maintainability. Computer, 27(8), 44-49. https://doi.org/10.1109/2.303623

coveragepy. (2024, Temmuz 4). coveragepy documentation. https://coverage.readthedocs.io/en/7.5.4/

Damian, A. L., Dias Canedo, E., Sieckenius de Souza, C., & Conte, T. (2021). Towards to Transfer the Directives of Communicability to Software Projects: Qualitative Studies. Journal of Software Engineering Research and Development, 9. https://doi.org/10.5753/jserd.2021.1942

Django. (2024, Haziran 25). Django documentation. https://www.djangoproject.com/

Fontana, F. A., Ferme, V., Zanoni, M., & Roveda, R. (2015). Towards a prioritization of code debt: A code smell Intensity Index. 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), 16-24. https://doi.org/10.1109/MTD.2015.7332620

GitHub. (2024a, Haziran 25). Get started with GitHub documentation. https://github.com/

GitHub. (2024b, Temmuz 4). Cobertura. https://github.com/cobertura/cobertura

GitHub. (2024c, Temmuz 9). jajuk-team. https://github.com/jajuk-team/jajuk

GitHub. (2024d, Temmuz 9). The Apache Software Foundation. https://github.com/apache/commons-math

Gokarna, M., & Singh, R. (2021). DevOps: A Historical Review and Future Works. 2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS), 366-371. https://doi.org/10.1109/ICCCIS51004.2021.9397235

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., & Zhou, M. (2020). GraphCodeBERT: Pre-training Code Representations with Data Flow. https://arxiv.org/abs/2009.08366

Gupta, S., & Gupta, S. K. (2019). Natural language processing in mining unstructured data from software repositories: a review. Sādhanā, 44(12), 244. https://doi.org/10.1007/s12046-019-1223-9

Halstead, M. H. (1977). Elements of Software Science (Operating and programming systems series). Elsevier Science Inc.

Hu, X., Li, G., Xia, X., Lo, D., & Jin, Z. (2018). Deep code comment generation. Proceedings of the 26th Conference on Program Comprehension, 200-210. https://doi.org/10.1145/3196321.3196334

Hu, X., Li, G., Xia, X., Lo, D., & Jin, Z. (2020). Deep code comment generation with hybrid lexical and syntactical information. Empirical Software Engineering, 25(3), 2179-2217. https://doi.org/10.1007/s10664-019-09730-9

Huang, Y., Guo, H., Ding, X., Shu, J., Chen, X., Luo, X., Zheng, Z., & Zhou, X. (2023). A Comparative Study on Method Comment and Inline Comment. ACM Transactions on Software Engineering and Methodology, 32(5), 1-26. https://doi.org/10.1145/3582570

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

Jayakody, J. A. V. M. K., & Wijayanayake, W. M. J. I. (2021). Challenges for adopting DevOps in information technology projects. 2021 International Research Conference on Smart Computing and Systems Engineering (SCSE), 203-210. https://doi.org/10.1109/SCSE53661.2021.9568348

JUnit 5. (2024, Temmuz 4). JUnit 5 User Guide. https://junit.org/junit5/

Kilic, M. A. E., & Adak, M. F. (2024). Source Code Summarization & Comment Generation with NLP : A New Index Proposal. 2024 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), 1-6. https://doi.org/10.1109/HORA61326.2024.10550711

Koontz, E. (2024, Temmuz 9). GitHub. ekoontz. https://github.com/ekoontz/hadoop-common

Kreinovich, V., Masmali, O. A., Nguyen, H. P., & Badreddin, O. (2020). Theoretical Explanation of Recent Empirically Successful Code Quality Metrics. Journal of Advanced Computational Intelligence and Intelligent Informatics, 24(5), 604-608. https://doi.org/10.20965/jaciii.2020.p0604

Krishnamurthi, S. (2013). Artifact evaluation for software conferences. ACM SIGPLAN Notices, 48(4S), 17-21. https://doi.org/10.1145/2502508.2502518

Lwakatare, L. E., Crnkovic, I., & Bosch, J. (2020). DevOps for AI – Challenges in Development of AI-enabled Applications. 2020 International Conference on Software, Telecommunications and Computer Networks (SoftCOM), 1-6. https://doi.org/10.23919/SoftCOM50211.2020.9238323

McCabe, T. J. (1976). A Complexity Measure. IEEE Transactions on Software Engineering, SE-2(4), 308-320. https://doi.org/10.1109/TSE.1976.233837

Mi, Q., Chen, M., Cai, Z., & Jia, X. (2023). What makes a readable code? A causal analysis method. Software: Practice and Experience, 53(6), 1391-1409. https://doi.org/10.1002/spe.3192

Misra, S., Cafer, F., Akman, I., & Fernandez-Sanz, L. (2013). Multi Paradigm Metric and its Applicability on JAVA Projects. Acta Polytechnica Hungarica, 10(3), 203-220.

Mypy. (2024, Temmuz 4). mypy documentation. https://mypy.readthedocs.io/en/stable/

Niazi, T., Das, T., Ahmed, G., Waqas, S. M., Khan, S., Khan, S., Abdelatif, A. A., & Wasi, S. (2023). Investigating Novice Developers' Code Commenting Trends Using Machine Learning Techniques. Algorithms, 16(1), 53. https://doi.org/10.3390/a16010053

Oracle. (2024a, Haziran 25). javadoc. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

Oracle. (2024b, Temmuz 8). javadoc command line tool. https://docs.oracle.com/en/java/javase/11/javadoc/javadoc-command.html

Oracle. (2024c, Temmuz 8). javadoc tags. https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

Otter, D. W., Medina, J. R., & Kalita, J. K. (2021). A Survey of the Usages of Deep Learning for Natural Language Processing. IEEE Transactions on Neural Networks and Learning Systems, 32(2), 604-624. https://doi.org/10.1109/TNNLS.2020.2979670

Panthaplackel, S., Nie, P., Gligoric, M., Li, J. J., & Mooney, R. J. (2020). Learning to Update Natural Language Comments Based on Code Changes. https://arxiv.org/abs/2004.12169

Park, Y., Park, A., & Kim, C. (2023). ALSI-Transformer: Transformer-Based Code Comment Generation With Aligned Lexical and Syntactic Information. IEEE Access, 11, 39037-39047. https://doi.org/10.1109/ACCESS.2023.3268638

Phung, K., Ogunshile, E., & Aydin, M. (2023). Error-Type—A Novel Set of Software Metrics for Software Fault Prediction. IEEE Access, 11, 30562-30574. https://doi.org/10.1109/ACCESS.2023.3262411

PostgreSQL. (2024, Haziran 25). PostgreSQL 15.7 Documentation. https://www.postgresql.org/

Pylint. (2024, Temmuz 4). Pylint documentation. https://pylint.readthedocs.io/en/latest/

pytest. (2024, Temmuz 4). pytest documentation. https://docs.pytest.org/en/8.2.x/index.html

Python Docstring. (2024, Temmuz 5). PEP 257 – Docstring Conventions. https://peps.python.org/pep-0257/

Rabbi, F., Hossain, S. S., & Arefin, M. M. S. (2022). SCMA: A Lightweight Tool to Analyze Swift Projects. 440-443. https://doi.org/10.18293/SEKE2022-006

Rashid, Z. J., & Adak, M. F. (2021). Test Data Generation for Dynamic Unit Test in Java Language using Genetic Algorithm. 2021 6th International Conference on Computer Science and Engineering (UBMK), 113-117. https://doi.org/10.1109/UBMK52708.2021.9558953

Sae-Lim, N., Hayashi, S., & Saeki, M. (2016). Context-based code smells prioritization for prefactoring. 2016 IEEE 24th International Conference on Program Comprehension (ICPC), 1-10. https://doi.org/10.1109/ICPC.2016.7503705

Sharma, T., & Spinellis, D. (2020). Do We Need Improved Code Quality Metrics?

Shiina, H., Ohnishi, S., & Fumihiro, Y. (2022). Program comment generation with improved distributed representation by Seq2seq model using parse tree information. International Journal of Smart Computing and Artificial Intelligence, 6(1), 645. https://doi.org/10.52731/ijscai.v6.i1.645

Song, Z., Zeng, H., Shang, X., Li, G., Li, H., & Guo, S. (2023). An data augmentation method for source code summarization. Neurocomputing, 549, 126385. https://doi.org/10.1016/j.neucom.2023.126385

Steinbeck, M., & Koschke, R. (2021). Javadoc Violations and Their Evolution in Open-Source Software. 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 249-259. https://doi.org/10.1109/SANER50967.2021.00031

Taha, I., & Elhadad, K. (2003). SOFTWARE SOURCE CODE: A QUALITY ASSURANCE MEASUREMENT SYSTEM. International Conference on Aerospace Sciences and Aviation Technology, 10(ASAT CONFERENCE), 1-13. https://doi.org/10.21608/asat.2013.24704

Thunes, C. (2024, Haziran 25). GitHub. Pure Python Java parser and tools. https://github.com/c2nes/javalang

Tronicek, Z. (2022). Indexing source code and clone detection. Information and Software Technology, 144, 106805. https://doi.org/10.1016/j.infsof.2021.106805

unittest. (2024, Temmuz 4). Unit testing framework. https://docs.python.org/3/library/unittest.html

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. ukasz, & Polosukhin, I. (2017). Attention is All you Need. Içinde I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Ed.), Advances in Neural Information Processing Systems (C. 30). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Voitot, P. (2024, Temmuz 9). GitHub. mandubian. https://github.com/mandubian/siena

Widyasari, R., Sim, S. Q., Lok, C., Qi, H., Phan, J., Tay, Q., Tan, C., Wee, F., Tan, J. E., Yieh, Y., Goh, B., Thung, F., Kang, H. J., Hoang, T., Lo, D., & Ouh, E. L. (2020). BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 1556-1560. https://doi.org/10.1145/3368089.3417943

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., & Li, S. (2018). Measuring Program Comprehension: A Large-Scale Field Study with Professionals. IEEE Transactions on Software Engineering, 44(10), 951-976. https://doi.org/10.1109/TSE.2017.2734091

Yin, T. (2024, Haziran 25). GitHub. A simple code complexity analyser. https://github.com/terryyin/lizard

ÖZGEÇMİŞ

Ad-Soyad : Mustafa Alp Eren KILIÇ

ÖĞRENİM DURUMU:

- **Lisans :** 2021, Sakarya Üniversitesi, Bilgisayar ve Bilişim Bilimleri Fakültesi, Bilgisayar Mühendisliği Bölümü

TEZDEN TÜRETİLEN ESERLER:

- Kilic M.A.E., Adak M.F. (2024, 23-25 Mayıs). Source Code Summarization & Comment Generation with NLP : A New Index Proposal. International Congress on Human-Computer Interaction, Optimiazation and Robotic Applications, Istanbul, Turkiye.