FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ PROGRAMI

SOYUT SÖZDİZİMİ AĞAÇLARI VE DERİN ÖĞRENME YÖNTEMLERİYLE KOD BENZERLİKLERİNİN TESPİTİ

YÜKSEK LİSANS TEZİ

NECMETTİN ELMASCI

İSTANBUL, 2023

FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ PROGRAMI

SOYUT SÖZDİZİMİ AĞAÇLARI VE DERİN ÖĞRENME YÖNTEMLERİYLE KOD BENZERLİKLERİNİN TESPİTİ

YÜKSEK LİSANS TEZİ

NECMETTİN ELMASCI

Danışman
(Ali Nizam)

İSTANBUL, 2023

FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ

FATİH SULTAN MEHMET VAKIF ÜNİVERSİTESİ
TEZ ONAY FORMU

28/10/2023

LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ MÜDÜRLÜĞÜNE

Bilgisayar Mühendisliği Anabilim Bilgisayar Mühendisliği tezli yüksek lisans programı öğrencisi 210221013 numaralı Necmettin Elmascı'nın hazırladığı "Kod Analiz Araç Çıktıları ve Geliştiricilerin Önceki Yeniden Düzenlemelerini Kullanan Derin Öğrenme Tabanlı Kod Kalite Öneri Sistemi Geliştirilmesi" konulu Yüksek Lisans tezi ile ilgili Tez Savunma Sınavı, 29/08/2023 Salı günü saat 14:00'da yapılmış, sorulara alınan cevaplar sonunda adayın tezinin **Kabulüne Oy Birliği** ile karar verilmiştir.

Tez adı değişikliği yapılması halinde: Tez adının "Soyut Sözdizimi Ağaçları ve Derin Öğrenme Yöntemleriyle Kod Benzerliklerinin Tespiti" şeklinde değiştirilmesi uygundur.

	Jüri Üyesi	Karar
1.	Dr. Öğr. Üyesi Ali NİZAM (Danışman)	Kabul
2.	Dr. Öğr. Üyesi Şaban SAHMOUD	Kabul
3.	Doç. Dr. Yusuf YASLAN	Kabul
4.	....	....
5.	....	....
6.	(İkinci Danışman)*	....

**2. Danışman varsa doldurulması gerekmektedir.*

ETİK BİLDİRİM

Bu tezin yazılmasında bilimsel ahlak kurallarına uyulduğunu, başkalarının eserlerinden yararlanılması durumunda bilimsel normlara uygun olarak atıfta bulunulduğunu, kullanılan verilerde herhangi bir tahrifat yapılmadığını, tezin herhangi bir kısmının bağlı olduğum üniversite veya bir başka üniversitedeki başka bir çalışma olarak sunulmadığını beyan ederim.

Necmettin Elmascı

TEŞEKKÜR

Öncelikle, Dr. Öğr. Üyesi Ali NİZAM'a tez sürecinde danışmanlığında sağladığı yönlendirmeleri, sahip olduğu bilgi birikimi ve tecrübeleri teorik ve pratikte açık bir şekilde aktarması, sorduğum tüm sorulara titizlikle ve içtenlikle cevap vermesi, zorlandığım veya anlamadığım noktalarda beni cesaretlendirmesi ve konunun her aşamasını daha detaylı anlamamı sağlayıp bütüncül düşünmemi sağlamasından dolayı teşekkür ederim. Üniversitemizin sağladığı ve tez yazımında oldukça etkili olan dersleri hazırlayan bölüm başkanı Dr. Öğr. Üyesi Berna KİRAZ 'a, Bilgisayar Mühendisliği için Matematik dersini veren Dr. Öğretim Üyesi Burcu BEKTAŞ DEMİRCİ'ye, Bilimsel Araştırma Teknikleri dersini veren Prof. Dr. Ali Yılmaz ÇAMURCU'ya, Veri Madenciliği Teknikleri dersini veren Doç. Dr. Buket DOĞAN'a, Yapay Zeka Teknikleri ve Uygulamaları dersini veren Dr. Öğretim Üyesi Shaaban A.I. SAHMOUD'a, Algoritma Analizi ve Tasarımında İleri Teknikler dersini veren Doç. Dr. Ömer KORÇAK'a, Bilgisayar Mühendisliği Proje Yönetimi dersini veren Dr. Öğretim Üyesi Ali NİZAM'a ve Makina Öğrenmesi dersini veren Dr. Öğretim Üyesi Berna KİRAZ'a teşekkür ederim.

Necmettin Elmascı

SOYUT SÖZDİZİMİ AĞAÇLARI VE DERİN ÖĞRENME YÖNTEMLERİYLE KOD BENZERLİKLERİNİN TESPİTİ

Necmettin Elmascı

ÖZET

Bu tez çalışmasının amacı, C# programlama dili baz alınarak kaynak koddaki benzerlikleri gösteren bir veriseti oluştup bu benzerliklerin tespiti ve iyileştirme gereken noktaları belirten sistemin tasarlarımının yapılmasıdır. Yazılım dünyasında, programların yapısını anlamak ve kod benzerliklerini tespit etmek, bir dizi önemli uygulama için hayati öneme sahiptir. Bu uygulamalar, kod klonlama tespiti, yazılım dolandırıcılığı izleme, ve yazılım kalitesi kontrolünü içerir. Bu bağlamda, Abstract Syntax Tree (AST) bazlı kod benzerlik analizi, bir dizi etkili çözüm sunmaktadır.

AST, bir programın sözdizimsel yapısını hiyerarşik bir şekilde temsil eder. AST'ler, kodun daha derin anlamsal benzerliklerini belirlemek için kullanılabilir, çünkü onlar kodun yapısal özelliklerini ve ayrıntılarını ortaya çıkarır. Bu nedenle, AST bazlı analizler, kod benzerliklerini belirlemek için oldukça güçlü ve esnektir.

Bu çalışmada, AST bazlı kod benzerlik analizinin temel prensiplerini, teknikleri ve uygulamaları incelenmiştir. Ayrıca, AST'lerin nasıl oluşturulduğu ve nasıl kullanıldığına dair detaylar ele alınmıştır ekleştirilen bu çalışmada, yapılmış olan diğer çalışmalardan farklı olarak sadece sınıf veya metot bazlı değil, kodun daha alt kırılımlarına inerek alt kırımlarının benzerliklerini tespit eden model ortaya konmuştur. Çeşitli metot ve araçlar ile geliştirilmiş olsa da sadece AST'leri içeren bir gömülü kütüphane (embedding vocabulary) olmadığı görülmüş ve bu yüzden de sadece ilgili AST'leri içeren bir kütüphane oluşturulmuştur.

Daha çok yüz tanıma sistemlerinde uygulanan triplet loss derin öğrenme ağı yöntemleri kullanılarak kod benzerliği üzerindeki etkileri de incelenmektedir. Bu amaçla, benzer kod ve benzemeyen kod bloklarının AST'leri baz alınarak ilgili veri

setleri oluşturularak triplet loss derin öğrenme ağı kullanılarak kod benzerliğinin tespiti için farklı bir yaklaşım ortaya konulmuştur.

Ayrı bir uygulama ile sıralama algoritmalarından bazılarının kodları (Quick Sort, Bubble Sort v.b.) baz alınarak metot ve blok (if, for, while) bazlı AST'ler oluşturuldu. İlgili kod bloklarının satır aralıkları ve her bir AST satırının diğer AST satırlarıyla benzerlikleri çıkartılarak veri seti oluşturuldu. Bu kod benzerliklerinden en yüksek olanlar baz alınarak kosinüs benzerliği (cosine similarity) ve triplet loss yöntemleri ayrı ayrı uygulanarak benzerlikleri ölçüldüğünde sırasıyla %61,4 ve 88,68 doğruluk oranları sağlanmıştır. Geliştirilen model kod benzerliği ile ilgili günümüzde kullanılan araçlarla da karşılaştırılmıştır. Özellikle triplet loss kullanılarak sağlanan yüksek doğruluk oranı önerilen tekniğin pratik kullanılanımı konusunda önemli bir gelişme olarak değerlendirilmiştir.

Anahtar kelimeler: kaynak kod, soyut sözdizimsel ağaç, triplet loss, kosinüs benzerliği, kod benzerliği

DETECTION OF CODE SIMILARITIES USING ABSTRACT SYNTAX TREES AND DEEP LEARNING METHODS

Necmettin Elmascı

ABSTRACT

The aim of this thesis work is to design a system that creates a dataset indicating similarities in the source code based on the C# programming language, detects these similarities, and identifies points that need improvement. In the software world, understanding the structure of programs and detecting code similarities is of vital importance for a range of significant applications. These applications include code clone detection, software fraud monitoring, and software quality control. In this context, Abstract Syntax Tree (AST) based code similarity analysis offers a series of effective solutions.

AST represents the syntactic structure of a program in a hierarchical way. ASTs can be used to determine the deeper semantic similarities of the code, as they reveal the structural features and details of the code. Therefore, AST-based analyses are powerful and flexible in determining code similarities.

This study examines the basic principles, techniques, and applications of AST-based code similarity analysis. It also covers details on how ASTs are formed and used.

In the work carried out, unlike other studies, a model detecting the similarities of sub-breakdowns of the code, not just based on classes or methods, has been put forward. Although it has been developed with various methods and tools, it was observed that there was no embedded library (embedding vocabulary) containing only ASTs, and therefore a vocabulary containing only relevant ASTs was created.

The effects on code similarity are also examined by using triplet loss deep learning network methods, which are mostly applied on the similarities of images. For this purpose, a different approach has been introduced to detect code similarity

using the triplet loss deep learning network by creating relevant data sets based on the ASTs of similar code and dissimilar code blocks.

With a separate application, method, and block (if, for, while) based ASTs were created based on the codes of some sorting algorithms (Quick Sort, Bubble Sort, etc.). The line ranges of the relevant code blocks and the similarities of each AST line with other AST lines were extracted and a dataset was created. When the code similarities were measured with the cosine similarity method, based on the highest of these code similarities, an accuracy rate of 61.4% was achieved. The developed model was also compared with the tools used today for code similarity. The high accuracy rate achieved by using triplet loss has been evaluated as an important development in the practical use of the proposed technique.

Keywords: source code, abstract syntax tree, triplet loss, cosine similarity, code similarity

ÖNSÖZ

"Soyut Sözdizimi Ağaçları ve Derin Öğrenme Yöntemleriyle Kod Benzerliklerinin Tespiti" başlıklı gerçekleştirdiğim çalışmada, öncelikle çalışılan konunun bilgisayar mühendisliği alanındaki yeri ve faydası incelenmiştir. Önerilen yaklaşım, kodun semantik ve yapısal özelliklerini birleştiren AST (Abstract Syntax Tree), kosinüs benzerliği ve triplet loss derin öğrenme ağı temelli bir benzerlik metriği kullanmaktadır. Bu, yazılım mühendisleri için daha etkili ve genel bir kod benzerliği analizi sağlamayı hedeflemektedir. Tez, Fatih Sultan Mehmet Vakıf Üniversitesi'nde Bilgisayar Mühendisliği Anabilim Dalı'nda Tezli Yüksek Lisans mezuniyet gereksinimimi yerine getirmek amacıyla yazılmıştır. Tezin araştırma ve gerçekleştirme süreci 2022-2023 yıllarının Eylül-Haziran tarihleri arasındadır.

Öncelikle, Dr. Öğr. Üyesi Ali NİZAM'a tez sürecindeki danışmanlığında titizlikle sağladığı yönlendirmeleri, konuyu daha genelden detaya doğru daha basit düşünmemi sağladığından, sorduğum sorulara içtenlikle cevap vermesinden ve zorlandığım konularda cesaretlendiriciliğinden dolayı teşekkür ederim. Üniversitemizin sağladığı ve tez yazımında oldukça etkili olan dersleri hazırlayan bölüm başkanı Dr. Öğr. Üyesi Berna KİRAZ 'a, Bilgisayar Mühendisliği için Matematik dersini veren Dr. Öğretim Üyesi Burcu BEKTAŞ DEMİRCİ'ye, Bilimsel Araştırma Teknikleri dersini veren Prof. Dr. Ali Yılmaz ÇAMURCU'ya, Veri Madenciliği Teknikleri dersini veren Doç. Dr. Buket DOĞAN'a, Yapay Zeka Teknikleri ve Uygulamaları dersini veren Dr. Öğretim Üyesi Shaaban A.I. SAHMOUD'a, Algoritma Analizi ve Tasarımında İleri Teknikler dersini veren Doç. Dr. Ömer KORÇAK'a, Bilgisayar Mühendisliği Proje Yönetimi dersini veren Dr. Öğretim Üyesi Ali NİZAM'a ve Makina Öğrenmesi dersini veren Dr. Öğretim Üyesi Berna KİRAZ'a teşekkür ederim. Ayrıca, ailemin tüm süreç boyunca hissettirdiği desteği benim için oldukça önemli oldu ve sağladıkları tüm destekler için teşekkür ederim. Ek olarak, tez süresince araştırdığım ve kaynak olan kitap ve makale

yazarlarına ayrı ayrı teşekkür ederim. Tüm bu kişiler sayesinde tez sonucunu başarıyla tamamlamam mümkün olmuştur.

Yazılım mühendisliği alanındaki araştırmacılar, öğrenciler ve profesyoneller için umarım ki, bu çalışma kod benzerliği analizi konusunda yeni ve değerli bir bakış açısı sağlar. Bu tezin, okuyuculara kod benzerliği analizi konusunda daha fazla bilgi vermek ve bu alandaki araştırmaları teşvik etmek için bir kaynak olmasını umuyorum.

Temmuz, 2023 Necmettin Elmascı

İÇİNDEKİLER

ÖZET .... **v**
ABSTRACT .... **vii**
ÖNSÖZ .... **ix**
SEMBOLLER .... **xii**
ÇİZELGE LİSTESİ .... **xiii**
KISALTMALAR .... **xiv**
GİRİŞ **1**
BİRİNCİ BÖLÜM **3**
1. LİTERATÜR ARAŞTIRMASI **3**
1.1. CODE2VEC 3
1.2. BENZER SİSTEM TASARIMLARI 4
İKİNCİ BÖLÜM **12**
2. TEORİK ÇERÇEVE **12**
2.1. SOYUT SÖZDİZİMSEL AĞAÇ 12
2.2. KOD KARIŞTIRMASI (EMBEDDING) 14
2.3. KOD BENZERLİĞİ(CODE SIMILARITY) 15
2.4. KOSİNÜS BENZERLİĞİ (COSINE SIMILARITY) 16
2.5. TRIPLET LOSS 18
2.5.1. Triplet Seçimi **19**
2.5.2. Triplet Loss Öğrenme Süreci Optimizasyonu **20**
ÜÇÜNCÜ BÖLÜM **22**
3. YÖNTEM **22**
3.1. VERİ SETİ OLUŞTURMA SÜRECİ 22
3.1.1. Kod Örneklerinin Toplanması: **22**
3.1.2. AST Oluşturma **23**
3.1.3. Vektör Uzayı Modellerinin Oluşturulması **24**
3.1.4. Kosinüs Benzerliği Hesaplama: **25**
3.1.5. Veri Setinin Oluşturulması **25**
3.1.6. Triplet Loss Benzerlik Hesaplama: **25**
3.1.7. Korelasyon Oluşturulması: **26**
DÖRDÜNCÜ BÖLÜM **27**
4. TEST SONUÇLARI VE YORUM **27**
SONUÇ **33**
KAYNAKÇA **34**

SEMBOLLER

α	**:** Öğrenme Oranı
θ	**:** Açı değeri

ÇİZELGE LİSTESİ

Sayfa

Şekil 2.1 : Örnek Program Metodu ve Soyut Sözdizimsel Gösterimi ………….. **13**
Şekil 2.2 : Örnek Kod Benzerliği Gösterimi …………………………………….. **16**
Şekil 2.3 : Kosinüs Benzerliği Formülü ve Grafik Üzerinden Gösterimi ………… **18**
Şekil 2.4 : Triplet Loss Gösterimi ………………………………………………… **19**
Şekil 3.1 : Kaynak Kod Örnekleri ………………………………………………… **23**
Şekil 3.2 : Oluşturulan AST Örnekleri…………………………………………….. **24**
Şekil 4.1 : İşleme Alınmayan AST Veri Seti Örnekleri …………………………… **27**
Şekil 4.2 : Oluşturulan AST Veri Seti Örnekleri………………………………….. **28**
Şekil 4.3 : Kod benzerlik Örnekleri ……………………………………………… **28**
Şekil 4.4 : Oluşturulan Triplet Loss Veri Seti Örnekleri ………………………….. **29**
Şekil 4.5 : Oluşturulan Triplet Loss Sonuç Örneleri …………………………….. **30**
Şekil 4.6 : MossSwing Kullanılarak Kod Benzerliklerinin Bulunması ………….. **31**
Şekil 4.7 : Korelasyon Sonucu…………………………………………………….. **32**

KISALTMALAR

bkz.	Bakınız
v.b.	Ve benzeri
AST	Soyut Sözdizimi Ağaçları
NLP	Doğal Dil İşleme
Code2Vec	Koddan Vektöre
Word2Vec	Kelimeden Vektöre
LSTM	Uzun-Kısa Süreli Bellek
Bi-LSTM	Çift Yönlü Uzun-Kısa Süreli Bellek
CNN	Evrişimsel Sinir Ağı
TBCNN	Ağaç Tabanlı Evrişimsel Sinir Ağı
RNN	Tekrarlayan Sinir Ağı
GNN	Grafik Sinir Ağı
GGNN	Geçitli Grafik Dizili Sinir Ağları
CCFinder	Kod Klonu Bulucu

GİRİŞ

Yazılım geliştirme ve bakım süreçleri üzerine yürütülen araştırmalar, kod benzerliğini ve tekrarlanan yapıları belirleme konusunun önemine vurgu yapmaktadır [1]. Bu konu, karmaşık ve geniş çaplı yazılım projeleri ile uğraşanlar veya çok sayıda geliştiricinin aynı projede çalıştığı durumlarda özellikle önem kazanır. Çünkü bu durumlarda, kodların benzerliğini anlamak ve böylece hata düzeltme, optimizasyon ve refactor işlemlerini daha kolay ve hızlı bir şekilde gerçekleştirmek önemli bir gereklilik haline gelir.

Bu bağlamda, Abstract Syntax Trees (AST) adlı teknoloji, kod benzerliğini belirlemeye yardımcı olan etkin bir araç olarak öne çıkmaktadır [2]. AST'ler, bir kodun daha soyut ve işlenebilir bir temsilini sunarak, kod benzerliği ve tekrarlanan yapıları daha kolay tespit etmeyi sağlar. Bu, yazılım geliştirme süreçlerini hızlandırmak ve yazılım kalitesini artırmak için hayati önem taşır.

Açık kaynak projeler sayesinde kaynak kodu büyük bir veri haline getirmek ve yapay zeka teknikleriyle içerisinden önemli bilgiler elde etmek kolaylaşmıştır. Burada ortaya çıkan temel problem, kaynak kodun nasıl temsil edileceğidir. Alon v.d., [1] tarafından yapılan önceki çalışmada kod olarak sadece metotlar göz önüne alınarak, metotlarda kullanılan değişkenleri başlangıç ve değer atanma yerine göre bir yol (path) tanımlamaktadır. Değişkenin ilk tanımlandığı ve son atandığı yer belirteç vektörü ve arada kalan yol arama (lookup) vektörü olarak adlandırmıştır. Bu vektörler öğrenilen bir ağırlık matrisiyle çarpılarak birleştirilir ve normalize edilerek bir yol bağlam (path-context) vektörü elde edilmiş olur. Elde edilen bu bilgiyle, bir ilgi (attention) ağırlığı, öğrenebilen bir ağırlık yol bağlam vektörleri için atanmış olur. Atanan bu değerlerle, ilgi vektörü ve her yol bağlam vektörüyle skaler çarpım yaparak bir öğrenebilen ağırlıklandırılmış ortalama vektörü elde edilmiş ve daha sonrasında metot isimlerini tahmin etmek için kullanılmıştır. Burada metot boyutunun artması, değişkene çok uzak bir yerde değer atanması v.b. durumlarda oluşturulan vektörler çok yoğun olacağından sisteme hesaplama olarak yük

bindirmiştir. Code2vec çalışması daha detay olan bir görev olan metot isimlerinin tahmin edilmesi üzerine kapsamı daraltılarak gerçekleşmiştir.

Bu tez çalışmasında, kod üzerinde kosinüs benzerliği ve triplet loss derin öğrenme ağı teknikleri kullanarak kod kalitesini ve benzerliklerini tespit etmek hedeflenmektedir. Bu kapsamda kaynak koddaki metot ve blokllardan yola çıkarak AST tabanlı veri seti oluşturmayı, her bir kod bloğuna ait AST'lerin birbiri ile karşılaştırılarak benzerlik oranlarının tespit edildiği sitem geliştirmek hedeflenmektedir.

Kod analiz konusundaki çalışmaların büyük çoğunluğu metot düzeyinde inceleme yapmaktadır. Bu durum derin öğrenme sistemleri için sınıf ve satır temelinde kod analizi yapabilen statik kod analiz araçlarına göre bir eksiklik oluşturmaktadır.

Bu çalışmada metot yerine, metodun alt kırılımlarının bütününü temsil edecek yeni bir model yapısı kullanılması önerilmektedir. Kod benzerlikleri göz önünde bulundurulduğunda sadece metot bazlı değil alt kırılımları olan blok(statement) benzerliklerinin tespiti ile ortak metotların oluşturulmasıyla kod benzerliklerini ve kod tekrarlarını(code duplicate-clone) daha büyük ölçüde kaldırdığı görülmüştür.

Tezin temel araştırma soruları şöyledir: (1) Metot ve blok bağlamındaki kaynak kod bir sistem tasarımına girdi olarak verileceğinden nasıl temsil edilir? (2) Sisteme verilecek olan kaynak kod sisteme verilmeden önce bir ön çalışma gerekli midir? Gerekliyse nasıl bir yol izlenir? (3) Kod blokları ve metotlar arasında benzerlik ilişkisi kurulabilir mi? (4) Optimize olabilmesi için nasıl bir yöntem kullanılmalıdır? Yeni bir model tasarlanmasına ihtiyaç var mıdır? Varsa nasıl tasarlanmalıdır? Bu modelin var olan yöntemlerden ne gibi bir avantajı vardır?

Bu tez, şu takip eden bölümlerden meydana gelmektedir: 1. Bölüm'de konuyla ilişkili ve tezin ortaya çıkmasını sağlayan literatür araştırmaları ele alınır. 2. Bölüm'de tezin ortaya çıkması için elde edilen teorik çerçeve ortaya konarak konular hakkında detaylı bilgi verilir. 3. Bölüm'de metodoloji tanımlanır ve sistemin çalışma mekanizması, verisetini oluşturmak için yazılan ek programın çalışma mantığı, verisetinin tanıtılması, ve kullanılan yapıların detaylı anlatılması sağlanır. 4. Bölüm'de sonuçların ortaya konması ve yorumlanması üzerine gerçekleşmektedir. Nihai bölümde, sonuçlar ve katkı özet olarak ele alınır ve tamamlanır.

BİRİNCİ BÖLÜM

1. LİTERATÜR ARAŞTIRMASI

Bu bölümde, tez çalışmasının konusuna bağlı olarak daha önceden benzer konularda yapılan çalışmaların yöntem, sonuç, kavram ve tartışmaları ele alınmıştır. Bu konuda önemli adımlar olan Code2vec ve kod gözden geçirmeleri ayrıntılı incelenmiştir. Benzeri çalışmalar yaparak farklı bakış açıları ve modeller ortaya koyan çalışmalar da araştırılmıştır. Bu bölümün son kısmında, literatürdeki örneklerde ortaya çıkan boşluklar ele alınarak bu tezin bu boşlukları nasıl dolduracağı üzerine inceleme yapılmıştır.

Bu tez çalışması, kod gözden geçirmeler sayesinde kaynak kodun daha iyi hale getirilmesi görevini, ilgili kod bloklarının AST tabanlı vektörleştirme ve buna dayalı benzerlik dereceleri belirten bir yöntem geliştirme üzerinde yapılmıştır. Bu kapsamda düşünüldüğünde, tez bu konunun önemini ve teorisini anlatan makalelerden olan "code2vec: Learning Distributed Representations of Code" [3] çalışması ilerleyen bölümde detaylıca incelenmiştir. Code2vec [3] çalışması incelenerek kaynak kodun doğal dil olarak düşünülmeden vektörleştirmesindeki temel mantık anlaşılmış ve bunun üzerine konarak yeni bir model tasarlanmıştır.

1.1. CODE2VEC

İlk başta kodu vektör haline dönüştüren bir çalışma şeklinde düşünülen bu çalışmada, Alon v.d., word2vec'e [5] benzer olarak kodun dağıtık gösterimini öğrenen bir yaklaşım sunmuştur [3]. Çalışmalarında; kod parçacıklarının, yani metotların, anlamsal gösterimlerini tahmin eden ve kod parçalarının temsilini sürekli dağıtık vektörler olarak yapan bir sinir ağı modeli geliştirmişlerdir. Bu işlemi yaparken kod parçacıklarını yol (path) koleksiyonlarına ayırmış soyut sözdizimsel (syntax) ağaca dönüştürerek her yolun atomik gösteriminin öğrenilmesini sağlamışlardır. Soyut sözdizimsel ağacın yaprakları belirteçleri (token) göstermektedir. Küme halindeki belirteçler öğrenme yapısında belirteç kelimesi olarak adlandırılır. Soyut sözdizimsel ağacın yol bağlam vektörü sayısı $O(n^2)$'dir. Belirteçler metotta yer alan değişken tanımlamaları ve atanmalarıdır. Burada tüm

atamalar için bir küme oluşturarak bir metot için tek bir katıştırma (embedding) elde eder. Buradaki yol kavramı değişkenin başlangıç durumundan değer atanmasına kadar olan aşamadaki durumudur. İlk ve değer atanmış durum belirteç vektörü temsil ederken, aradaki kısımlar yol vektörünü (path vector) temsil etmektedir. Bu yapıyı öğrenilmiş bir matrix olan W (W: katıştırma için tüm yol-bağlam vektörlerinin aynı boyutta olmasını sağlayan değer) ile çarpar ve birleştirir. Birleştiren bu vektörlerle tek bir vektör elde etmek için hiperbolik tanjant fonksiyonu ile çarpar. Böylece her vektör elamanını -1 ile 1 arasında yeniden ölçeklemektedir.

Sonuç olarak burada bir tam bağlı katman ve yol bağlam vektörü elde eder. Elde edilen yol bağlam vektörü, iki belirtecin bağlantısını ve içerisinde yolu belli eden bilgiyi saklar. Bu gösterim bir yol içindir ancak kod metodu genellikle birden fazla yoldan oluşur. Bu çalışmada birden fazla yolu kümelemek ve tekil kod vektörüne dönüştürme işlemi sağlanması için bir ilgi vektörü (attention weight) kullanarak ağırlıklandırma sağlanır. Kod vektörün oluşması için özyinelemeli sinir ağına bağlı uzun kısa dönem hafıza bileşenini (LSTM) kullanarak hesaplama yapılmıştır. Öğrenme etiketi olarak metot ismi kullanılmaktadır. Ağırlıklandırılan değer, öğrenme boyunca yol bağlam vektörleri için güncellenerek kod vektör için kullanılması sağlanır. Çıktı katmanında bir tahmin değerlendirmesi yapılırken kod vektörleri softmax ile normalize edilir. Yaptıkları çalışmayı benzer kodların metot isimlerini tahmin etmek için uygulamışlardır. Kullanılan veriseti yaklaşık olarak 14 milyon Java metodu içeren bir verisetidir. Bu veriseti üzerinde 12 milyon metot eğitilmiş geriye kalan ve eğitim esnasında hiç görmedikleri metotlardan metot isimlerinin tahmin edilmesini sağlayan bir model tasarlamışlardır. Metot isimlendirmesi üzerinde F1 skoru olarak 59.5 değerini almıştır.

1.2. BENZER SİSTEM TASARIMLARI

Sui v.d., [6] gerçekleştirdikleri çalışmada program iç yapısındaki değer-akış bağımlılığının metot içerisindeki yapıya bağlı kalarak düşük boyutlu vektör uzayında yeni bir kod katıştırması yaklaşımı geliştirmiştir. Bu yaklaşımla code2vec yapısına benzer olarak metot isminin tahmin edilmesi üzerine çalışılmıştır. Ortaya çıkarılan katıştırma vektörüyle, kapsayıcı şekilde kodun temsilinde alt öğrenme görevlerinde kullanılabileceği sonucu elde edilmiştir. Çalışmadaki verisetinin kullanılması 32 açık

kaynak projesinin 5 milyon satırdaki kodunda metotlar dikkate alınarak uygulanmıştır. Flow2vec, metotların isim tahmininde değer-akışlarında takma ad yaklaşımıyla %96.9 yüksek oranda sonuç vermiştir. Yapılan çalışma kod katıştırma yaklaşımlarından olan code2vec ve code2seq çalışmalarında yapılan kod metot ismi tahminlemesi ve sınıflandırmasına karşı daha performanslı sonuçlar elde etmiştir.

Code2vec geliştiricilerinden Alon v.d., [7] kaynak kodu daha iyi bir şekilde vektörleştirmek için programlama dilinin anlamsal bütünlüğünü saklayacak bir yaklaşım geliştirmiştir. Küçük kod metot parça kümelerinin soyut sözdizimsel ağaç olarak kodlar ve çözümünü de ilişkili yolları ilgi ağırlık vektörü ile yapmaktadır. Elde edilen yolları kodlayarak uzun kısa süreli hafıza (LSTM) kullanılmıştır. Yapılan çalışmada 16 milyon C# ve Java kodu kullanılmıştır. Buradaki çalışmada Java ve C# için sırasıyla kod özetlemesi ve kod adlandırması yapılmıştır. Kod özetlemesinde öğrenilen metot bloğunun ismi etiketlenmiştir. Etiketlenen tüm metotlar test edilirken içerdikleri değişkenlerin ve alt metotların isimlerini soyut sözdizimsel ağaçta tutulmuştur. Böylece benzer şekilde bir test verisi geldiğinde etiketleme yapılarak tahminleme yapılmıştır. Kod adlandırmasında dile özgü kullanılan bazı önemli anahtarları öğrenerek gelen metodu tahmin ederek adlandırma yolu uygulanmıştır. Sonuç olarak daha çok isimlendirme üzerine gidildiğinden BLEU skorları [8] üzerinden sonuçlar verilmektedir. Code2seq anlamsal kod temsili sayesinde diğer modellere göre daha avantajlı sonuçlar verdiğini ve elde ettiği değerin ardışık olarak öğretilip soyut sözdizimsel ağacın doğrusallaştırılması yerine yapısal yaklaşımın daha iyi olduğu sonucuna varmıştır.

Jiang v.d., [20] Deckard isimli kod klonu tespit sistemi geliştirilmiştir. Yöntem, kodu vektörel bir biçimde temsil etmek için ağaç yapılarını kullanır ve bu ağaç yapıları üzerinde öklidyen benzerliğini hesaplar. Deckard'ın etkinliği, birkaç farklı programlama dilinde (C, Java ve Python) yazılmış geniş bir yelpazeye sahip yazılımlar üzerinde test edilmiştir. Testler, hem açık kaynaklı hem de ticari yazılımlar kullanılarak gerçekleştirilmiştir. Yazarlar, kodların ağaç yapılarına dönüştürülmesi ve bu ağaçların vektörler olarak ifade edilmesinin, kod klonlarına ilişkin geniş ölçekte arama yapmayı mümkün kıldığını bulmuşlardır. Test sonuçları, Deckard'ın geniş bir yelpazede, büyük ölçekte ve farklı programlama dillerinde yüksek doğrulukla klonları tespit edebildiğini göstermiştir. Yazarlar, Deckard'ın hem

Type-1 (tamamen aynı kod blokları) hem de Type-2 (değişken isimleri, beyaz boşluklar ve yorumlar gibi bazı farklılıkları olan kod blokları) klonlarını etkili bir şekilde tespit edebildiğini bulmuşlardır. Ancak, daha karmaşık Type-3 klonları (bazı satırların eklendiği veya çıkarıldığı kod blokları) tespit etme konusunda daha az etkili olmuştur. Yazarlar, Deckard'ın hız ve doğrulukta başarılı olduğunu, bu da onu hem akademik araştırmalar hem de endüstriyel uygulamalar için uygun bir araç haline getirdiğini belirtmiştir. Bu nedenle, Deckard'ın kod klonlama tespit tekniklerinin önemli bir yönünü temsil ettiği sonucuna varmışlardır.

Roy v.d, [25] bir dizi farklı kod klonu tespit tekniği ve aracını analiz etmek için kalitatif bir yaklaşım kullanmaktadır. Klon tespit tekniklerinin ve araçlarının performansını değerlendirmek için kapsamlı bir literatür taraması ve araçların kendi kendine değerlendirmesini gerçekleştirmiştir. Bu çalışmada değerlendirilen araçlar arasında metin tabanlı, token tabanlı, ağaç tabanlı, metrik tabanlı ve hibrit yaklaşımlar bulunmaktadır. Bu araçlar farklı dillerdeki kodları ve farklı türdeki klonları tespit etme kabiliyetine sahip olabilir. Yazarlar, bu farklı tekniklerin ve araçların avantajları ve dezavantajları hakkında kapsamlı bir tartışma yapmış ve bu araçların genellikle kendi özel durumlarına en uygun olanları seçmeleri gerektiğini belirtmiştir. Çalışmanın sonuçları, tüm tekniklerin ve araçların kendi güçlü yanlarına ve zayıf yanlarına sahip olduğunu, ancak hiçbirinin tüm durumlarda mükemmel olmadığını göstermiştir. Örneğin, bazı teknikler ve araçlar belirli türdeki klonları tespit etmede daha iyidir, bazıları ise geniş çapta tarama yapmayı sağlar. Ayrıca, bazı araçlar belirli dillerde daha iyi çalışırken, diğerleri çok dilli ortamlarda daha etkilidir. Sonuç olarak, Roy v.d., kod klonlama tespitindeki en iyi sonuçların genellikle birden fazla tekniğin ve aracın bir arada kullanılmasından elde edildiğini belirtmiştir. Bu, hem klonların tespit edilmesi hem de sonuçların analiz edilmesi için çeşitli araçların ve tekniklerin bir kombinasyonunun kullanılmasını içerir.

Wang v.d., [35] derin öğrenme tabanlı kod benzerliği tespit yöntemlerini incelemekte ve bunların etkinliklerini ve sınırlılıklarını belirlemeye yönelik bir dizi deney gerçekleştirmektedir. Deneyler, farklı derin öğrenme modellerini ve kod temsillerini kullanarak, çeşitli benzerlik ölçütlerine göre kod benzerliklerini tespit etme yeteneklerini değerlendirmiştir. Wang v.d., derin öğrenme modellerinin kod benzerliği tespiti konusunda etkili olduğunu bulmuşlardır. Bu modeller, geleneksel

kod benzerliği tespit tekniklerine göre daha fazla bilgiyi otomatik olarak çıkarabilmekte ve daha geniş bir yelpazede kod parçalarını karşılaştırabilmektedir. Ancak, bu modellerin hala bazı zorluklarla karşılaştığı da belirtilmiştir. Özellikle, derin öğrenme tabanlı yaklaşımların büyük ölçekte uygulanabilmesi için hala hız ve verimlilik iyileştirmelerine ihtiyaç olduğu belirtilmiştir. Bu çalışma, derin öğrenme modellerinin kod benzerliği tespitindeki potansiyelini ve sınırlılıklarını vurgulamıştır. Yazarlar, bu teknolojinin gelecekteki geliştirmeler ve optimizasyonlarla önemli bir araç olabileceğini belirtmiştir. Ancak, şu an için, derin öğrenme tabanlı kod benzerliği tespit yöntemlerinin geleneksel yöntemlerle bir arada kullanılmasının genellikle en iyi sonuçları verdiğini belirtmiştir.

Jain v.d., [9] gerçekleştirdiği çalışmada birçok makine yardımlı programlama görevleri için aynı fonksiyonu sağlayan kaynak kodlardaki yapılar aynı gösterime sahip olduğunu ve bunu kendi geliştirdikleri ContraCode modeli ile bu gibi gösterimleri öğrenmesini hedeflemiştir ve kod özetlemesi üzerine çalışmıştır. Bu türde yapılan (daha önceki çalışmalar gözetimli öğrenme yaklaşımlarını benimser) ilk çalışma olduğunu vurgulayan çalışmada, ön eğitilmiş 1,8 milyon üzerinde açıklamasız ve etiketsiz JavaScript metotları Github'tan çekilerek ayarlanmıştır. ContraCode'un kod özetleme kesinliği gözetimli öğrenme yaklaşımları üzerine %7,9 diğer RoBERTa ön eğitimli tekniklere karşı %4,8 daha fazla kesinlik sonucu vermiştir.

Azcona v.d., [10] katıştırma tekniğini kullanarak kendi üniversitelerinin bilgisayar bilimlerinde okuyan öğrencilerin kod tasarımlarının bireysel öğrenci bazlı profilini çıkarmak için bir metodoloji ortaya koymuştur. Toplam veriseti; 3 akademik yılda, 5 Python programlama derslerinde öğrenim gören 666 öğrenciden alınan 591.707 kod sınıflarıdır. Doğal dil işleme için kelime torbası modeli (BOW) ile code2vec vektörleştirme yapıları kullanılmış ve bunlar karşılaştırılmıştır. Çalışmanın code2vec vektörleştirmesinde elde edilen çalışmada metot boyutları arttığından çok büyük matrisler elde edilerek hafıza kullanımının zorlaştığı vurgulanmıştır. Bundan dolayı, soyut sözdizimsel ağaç düğümleri ve kelime sınırı olarak 2000 belirtilmiş ve çalışmada one-hot encoding tekniği kullanıldığından 2000 boyuta denk gelmiştir. Kullanılan derin öğrenme ağı code2vec çalışmasındaki benzer yapı kullanıldığı

vurgulanmıştır. Code2vec ile elde edilen sonuçlar kelime torbası modeline göre daha başarılı olduğu sonuçlarına varılmıştır.

Duracik v.d., [11] kaynak kodda intihali geniş ölçekte önlemek için yapılan çalışmada kod temsilini soyut sözdizimsel ağaç kullanarak ve kendi geliştirdikleri algoritma ile karşılaştırma yapmıştır. Bu gösterim, karakteristik vektörler ve adresleme kullanarak temsil edilmiştir. Veriseti C# programlama dili üzerine yapılmıştır ve 59 öğrencinin etiketli olarak oluşturduğu ve 2 tane intihal olan kod eklenerek oluşturulmuştur. Veri seti olarak C# programlama diline ait projelerin kullanılmasının sebebi .NET derleme platformu (Roslyn) [12] tarafından otomatik oluşturulmasıdır. Toplamda 61 kod ve içerisinde 1050 sınıf ve 2468 metot yer almaktadır. Burada veri SyntaxNode, SyntaxToken ve SyntaxTrivia olarak 3 gruba ayrılmıştır. İlk grup kaynak kodun soyut bakış açısındaki yapısını, ikinci grup bireysel belirteçleri ve üçüncü grupta kaynak kodda çok önemli olmayan boşluklar, boş satırlar ve yorumları belirtmektedir. Yapılan çalışma sadece belli karakteristikteki kaynak kod bölümlerini karşılaştırmak için uygulanmıştır. Ağaç yapısında oluşacak yoğunluk nedeni ve hafıza kullanımı sebebiyle tam bir eşleştirme yapmadan sadece en iyi uyanları seçerek intihal karşılaştırmasına karar vermeye çalışılmıştır. Elde edilen sonuçlarda küçük değişikliklerle kopya olan kodlarda algılama başarısız olmuşken, önemli ölçülerde benzer olan kodları %98 başarıyla bulmuştur.

Le v.d., [13] derin öğrenme tekniklerinin kaynak kodda modelleme ve vektör üretimi üzerine genel bir araştırma makalesi ortaya koymuştur. Geleneksel kaynak kod modellerinin sınırlarını adreslemek için bir kodlayan ve çözümleyen çerçevede yerleştirirken, son yapılan çalışmalardaki bu gibi problemleri çözen derin öğrenme mekanizmaları tanıtılmıştır ve araştırmacılara ve uygulayıcılara zorlukları tartışmak için bir analiz yapılmıştır. Çalışmada BigCode veriseti [14] kullanılmıştır. Derin öğrenme ağı olarak yinelemeli sinir ağlarının (RNN) karmaşık temsil edilmeleri öğrenmek için yeterince güçlü olduğunu, çekişmeli üretici ağların (GNN) etiketlenmemiş modellerde öğrenmede daha iyi olduğu ve vektör temsilini bu modelin dağılımlarını çıkardıktan sonra elde edilebileceği çalışmada belirtmiştir. Soyut sözdizimsel ağaç yapılarının kodu genel anlamda düşünmede ilgi vektörleri kullanmak yerine kod vektörleştirmede yapısal bağlamı daha iyi temsil ettiğini ve

BigCode projesinde artan bir ilginin olduğu belirtilmiştir. Program kapsamı düşünüldüğünde ve kod metot isimlerinin tahmini, kod araması, kod hata çözümü veya onarımı gibi alt işlemlerde yinelemeli sinir ağlarından uzun kısa süreli hafıza ve kapalı yinelemeli ünitelerin çoğu araştırmada birlikte kullanıldığı ve başarılı sonuçlar alındığı belirtilmiştir.

Kamiya v.d., [37] , kod klonlarını tespit etmek için tasarlanmış çok dilli ve token tabanlı bir sistem olan CCFinder'ı sunmaktadır. CCFinder, kodu tokenlara (kod parçalarına) dönüştürür ve bu tokenları analiz eder. Bu analiz, kod klonlarını tespit etmek için bir kod parçasının nasıl bir başka kod parçasına benzeyebileceğini belirlemeyi içerir. CCFinder, bir kod tabanında geniş ölçekte klonları tespit etmek için kodun tokenlaştırılmasını ve bu tokenların karşılaştırılmasını kullanır. Çalışmanın sonuçları, CCFinder'ın geniş ölçekte ve çeşitli dillerde yüksek doğrulukla klonları tespit edebildiğini göstermiştir.

Wang v.d., [17] soyut sözdizimsel ağacı girdi olarak kullanan ağaç yapısına dönüştüren bir modüler ağaç ağ yapısı önermiştir. Yapılan çalışma sınıflandırma ve kod tekrarının bulunması üzerine odaklanmıştır. Veriseti olarak 293 programlama probleminde 58.600 C programı yer almaktadır. Eğitim, doğrulama ve test için sırasıyla %80, %10 ve %10 bir atama yapılmıştır. Sınıflandırma üzerine çalışmadaki öğrenme ağı olarak uzun kısa süreli hafıza ağı tercih edilmiştir. Başlangıç embedding olarak boyut sınıflandırmada 200 ve kod tekrarı görevinde 100'dür. Diğer modellerden CNN, LSTM, Bi-LSTM, Code-RNN, TBCNN, GGNN ve Tree-LSTM ile karşılaştırma yapılmış ve sınıflandırma üzerine çalışılan sistemde %86,5 başarı, kod tekrarı üzerine çalışmasında da %86 kesinlik değeri elde etmiştir.

Ragkhitwetsagul v.d., [36] kod klonlarını tespit etmek için metin analizi ve kosinüs benzerliği kullanmanın potansiyelini ve sınırlılıklarını araştırmıştır. Bu bağlamda, bir dizi farklı metin analizi tekniği ve bunların kod klonu tespitindeki etkinliği incelenmiştir. Sonuçlar, metin analizi ve kosinüs benzerliği kullanmanın, belirli durumlarda kod klonları tespit etme yeteneğine sahip olduğunu göstermiştir. Ancak, bu tekniklerin de sınırlamaları bulunmuştur. Özellikle, kodun semantik yapısının anlaşılması ve bazı karmaşık klon türlerinin tespit edilmesi konusunda zorluklar yaşanmıştır. Bu çalışma, metin analizi ve kosinüs benzerliği kullanmanın kod klonu tespitinde bazı potansiyel avantajları olduğunu, ancak bu tekniklerin

etkinliğinin kodun kendine özgü özelliklerine ve klon türlerine bağlı olduğunu belirtmiştir. Yazarlar, bu tekniklerin etkin bir şekilde kullanılabilmesi için daha fazla araştırma ve geliştirme çalışması gerektiğini vurgulamışlardır.

Devlin v.d., [18] Python'da değişkenin yanlış kullanımını iyileştirmek için bir girdi olarak fonksiyon katıştırması kullanmıştır. Metotları kodlarken soyut sözdizimsel ağaç kullanılmıştır. Katıştırmayı oluşturmak için düğümün tam yerini, düğüm tipini, düğüm ve atası arasındaki ilişkiyi ve düğümün metin olarak etiketlenmesiyle elde etmektedir. Derin öğrenme ağı olarak yinelemeli sinir ağı tercih edilmiştir. Tek tahminde ilgi (attention) mekanizmalarındaki başarı oranı olan %13 yerine %41 olarak başarılı sonuç vermiştir.

Büch v.d., [19] çalışmalarında kod metotlarının tekrarının soyut sözdizimsel ağaçlar kümesinin kendi kendini çağırma yapısıyla öğrenebilen ve verilen verisetini düzgün kümelere ayırabilen bir model ortaya koymuştur. Soyut sözdizimsel ağaçtan bir katıştırma elde etmek için yaprak düğümlerinden ana düğüme katettiği yolu ele almış ve elde edilen bu yol uzun kısa süreli hafıza ağına verilmiştir. Test olarak değerlendirilen kodun eğitilirken elde edilen veriyle ne kadar benzediğini hesaplamak için Siamese ağı kullanmıştır. Veriseti olarak BigCloneBench tercih edilmiştir. Veriseti 609 Java metotlarından ve 33 farklı benzerlik kümelerinden oluşmaktadır. Kümenin 2/3'ü eğitim 1/6'sı doğrulama ve 1/6'sı test için kullanılmıştır. Maksimum performans için katıştırmada kullanılabilen boyut 150 olarak seçilmiştir. Hesaplama sonuçlarını kümeyi doğru bir şekilde ayırmak için AUC değerlerini karşılaştırarak vermiştir. Daha fazla parametre ile soyut sözdizimsel ağacın oluşturulmasının vektör temsilinde daha çok yakınsamaya yardımcı olduğu belirtilmiştir.

Schroff v.d, [37] çalışmalarında derin öğrenme uygulamalarında örneklemenin önemini ele almaktadır. Özellikle triplet loss gibi metrik öğrenme yaklaşımlarında, pozitif ve negatif örneklerin seçimi modelin performansını önemli ölçüde etkileyebilmektedir. Bu bağlamda, yazarlar farklı örneklem stratejilerinin model performansı üzerindeki etkisini analiz etmektedir. Bu kapsamda elde dinilen bulgular, standart örneklem yöntemleri, modelin optimizasyon sürecini ve genelleme kapasitesini zorlaştırabilir. Örneklemenin, modelin farklı bölgelerindeki öğrenme hızını etkileyebileceğini ve bu nedenle dikkatli bir şekilde seçilmesi gerektiğini

vurgulamaktadırlar. Yazarlar, performansı optimize etmek için bazı geliştirilmiş örneklem stratejileri önermektedir. Sonuç olarak, bu çalışma, derin öğrenme uygulamalarında örneklem stratejilerinin seçiminin, modelin etkinliği ve genelleştirme kabiliyeti üzerinde kritik bir rol oynadığını ortaya koymaktadır.

Görüntüler arasındaki benzerlikleri inceleyen yöntemlerden kod benzerliği alanında da faydalanmak çalışmamızda araştırılan önemli bir konudur. Yuan v.d, [36] kişi tekrar tanıma (person re-identification) görevi, farklı kameralar veya farklı zamanlarda alınan görüntülerde aynı kişiyi tanımak için kullanılır. Bu görevde, farklı koşullarda alınan görüntülerde aynı kişiyi doğru bir şekilde tanımak oldukça zordur. Bu makale, kişi tekrar tanıma problemi için triplet loss yaklaşımının avantajlarını savunmaktadır. Yazarlar, triplet loss kullanılarak elde edilen sonuçların, diğer metrik öğrenim yöntemlerine göre daha iyi olduğunu göstermektedirler. Aynı zamanda, triplet loss'un bu görev için nasıl uygulanabileceği ve hangi örnekleme stratejilerinin daha iyi sonuçlar verdiği üzerine de detaylı analizler ve öneriler sunmaktadırlar.

Literatür araştırmasında yer alan çalışmalar incelendiğinde, bilgisayar biliminde araştırma sorularında da belirtilen problemleri çözmeye çalışmanın giderek arttığı görünmektedir. Literatürde yer alan çalışmalarda temel boşluk sınıf seviyesinde gösterim modeli yerine, daha alt görevler olan metot ismi tanıma tahminlemesi, kod metot parçasının tekrarlanmasının bulunması, kod parçasının temsilinde yeni yaklaşımlar ve var olan yeni modellerin karşılaştırması üzerine çalışılmıştır. Verisetlerinin kod sınıfını temsil edecek bir hiyerarşide olmaması, var olan verisetlerinin çoğunluğunun metot tabanlı hazırlanması ve koddaki gelişme türünün büyük ölçekte dikkate alınmaması diğer bir boşluğu ortaya çıkarmıştır. Bu tezde gerçekleştirilen çalışma temel olarak kullanılan çalışmalara ek olarak ilişkili çalışmalardaki bu boşlukları temel alarak sınıf tabanlı kod temsili ve etiketli sınıfın kod temsilini yapan verisetinin oluşturulması üzerinedir.

İKİNCİ BÖLÜM

2. TEORİK ÇERÇEVE

Bu bölümde tezde kullanılan bileşenlerin teorisinin aktarılması üzerine durulmuştur. Bu tez çalışmasında kullanılan teorik kavramlar olan; soyut sözdizimsel ağaçlar, kod katıştırması, kod benzerliği ana başlıkları bu bölümde anlatılacaktır. Kullanılan kavramlar var olan literatürde incelenerek elde edilmiştir. Teorik çerçevede kullanılan kavramların çalışmada hangi amaçla kullanıldığı bu bölümde belirtilmektedir.

2.1. SOYUT SÖZDİZİMSEL AĞAÇ

Soyut Sözdizimsel Ağaç (AST), bir programın kodunun daha yüksek seviyeli bir analizini sağlar ve daha karmaşık kodların bile daha anlaşılır ve işlenebilir hale gelmesini sağlar [2]. Bu teknoloji, dilin sözdizimsel özelliklerini ayrıştırarak kodu bir ağaç veri yapısına dönüştürür. Ağaç yapısında, her düğüm bir kod parçasını veya yapıyı temsil eder ve bağlantılar, kodun akışını ve kontrol yapısını belirtir.

AST'ler, kodu anlamak ve analiz etmek için etkili bir araçtır çünkü kodun yüksek seviye bir gösterimini sunarlar. Bu, karmaşık kodların daha net bir şekilde anlaşılmasını sağlar ve kodu daha anlamlı parçalara böler, böylece kodun işlevselliğini ve kontrol akışını daha iyi kavramamızı sağlar [2].

AST'ler, kod tekrarlarını (code clones) ve benzerlikleri tespit etmek için de etkili bir araçtır. Sözdizimsel olarak farklı ancak işlevsel olarak benzer kod parçalarını bile tespit etme yetenekleri vardır. Bu, AST'lerin kod benzerliklerini daha geniş bir yelpazede tespit edebileceği anlamına gelir. Bu özellik, özellikle geniş yazılım projelerinde veya çok sayıda geliştiricinin aynı projede çalıştığı durumlarda çok önemli olabilir [2].

Bunun yanında, AST'ler değişken isimleri, beyaz boşluklar ve yorumlar gibi kodun semantik olmayan parçalarını soyutlar, bu da kodun asıl amacına ve

işlevselliğine odaklanmayı sağlar. Bu özellikler, AST'lerin kod benzerliklerini tespit etmek için özellikle değerli bir araç olmasını sağlar [2].

Örnek bir kod parçası ve soyut sözdizimsel ağaç gösterimi Şekil 2.1'deki gibidir. Sözdizimsel analizde girdi bir belirteç akışında (token stream) olarak alınır ve çıktı olarak soyut sözdizimsel ağaç elde edilir. Burada belirlenen stratejide bütünü yansıtan sözdizimi dolaşmak için belirteç akışı parse işlemi uygulanır. Dolaşma sırasında soyut sözdizimini temsil eden bir ağaç oluşturulur.

```
if (x > 10)
{
    Console.WriteLine("x is greater than 10");
}
else
{
    Console.WriteLine("x is not greater than 10");
}
```

a) Örnek Kod Bloğu

```
IfStatement
├ BinaryExpression: ">"
|   ├ IdentifierName: "x"
|   └ LiteralExpression: "10"
├ Block (Then Statement)
|   └ ExpressionStatement
|       └ InvocationExpression
|           ├ MemberAccessExpression
|           |   ├ IdentifierName: "Console"
|           |   └ IdentifierName: "WriteLine"
|           └ ArgumentList
|               └ Argument
|                   └ StringLiteralExpression: "x is greater than 10"
└ Block (Else Statement)
    └ ExpressionStatement
        └ InvocationExpression
            ├ MemberAccessExpression
            |   ├ IdentifierName: "Console"
            |   └ IdentifierName: "WriteLine"
            └ ArgumentList
                └ Argument
                    └ StringLiteralExpression: "x is not greater than 10"
```

b) Soyut Sözdizimsel Gösterimi

Şekil 2.1: Örnek Program Metodu ve Soyut Sözdizimsel Gösterimi

2.2. KOD KARIŞTIRMASI (EMBEDDING)

Kod karıştırma (embedding), yazılım mühendisliği görevlerini kolaylaştırmak amacıyla kod parçalarını sayısal vektörlerle temsil etme sürecidir. Bu vektör temsilleri, kodun anlamsal ve yapısal bilgilerini yansıtır ve bu sayede kod benzerliği, kod klon tespiti, otomatik kod tamamlama ve daha pek çok görev için kullanılabilirler [20]. Genellikle doğal dil işleme (NLP) üzerine kurulu olan bir teknik olup, kod parçalarını vektör uzayına dönüştürür [14]. Bu, bir kod parçasının anlamsal bilgilerini tutar ve bu bilgiler, kod aramaları, kod eşleştirme veya benzerlik ölçümleri, hata tespiti ve kod önerileri gibi bir dizi görevi gerçekleştirmek için kullanılabilir.

Kod karıştırmada, bir kod parçası genellikle kodun dilbilgisine dayanan bir ağaç yapısına dönüştürülür, yani bir Abstract Syntax Tree (AST). AST'nin düğümleri kodun semantik bilgilerini içerirken, düğümler arasındaki bağlantılar kodun yapısal bilgilerini içerir. AST üzerinden geçişler, kodun anlamsal ve yapısal özelliklerini vektörleştirmek için bir kod karıştırma modeline girdi olarak kullanılır [2].

Kod karıştırma modelleri genellikle derin öğrenme tabanlıdır ve ağırlıklı olarak geriye dönük (recurrent) veya dikkat mekanizması (attention mechanism) içeren ağ yapısını kullanır [20]. Bu modeller genellikle geniş bir kod tabanı üzerinde eğitilir ve bu sayede genel ve özelleştirilebilir vektör temsilleri oluşturulur.

Kod karıştırma, birçok farklı yazılım mühendisliği görevi için potansiyel olarak yararlıdır. Kod önerisi, hata tespiti, kod eşleştirme ve benzerlik ölçümleri, kod tabanı araması ve daha pek çok görev için kullanılabilir [16]. Ayrıca, kod karıştırma, yazılım mühendislerinin karmaşık kod tabanlarını daha iyi anlamalarına ve yönetmelerine yardımcı olabilir.

Kod benzerliği, genellikle yazılım geliştirme süreçlerini iyileştirmek, kod kalitesini artırmak, hata oranını azaltmak ve yazılım bakımını kolaylaştırmak için kullanılan bir konsepttir. Kod parçalarının benzerliğini ölçmek için genellikle metin tabanlı veya yapısala dayalı yaklaşımlar kullanılır. Ancak bu yaklaşımlar, genellikle kodun anlamsal bilgilerini tam olarak yakalayamazlar. Bu nedenle, kod karıştırma gibi teknikler, kod benzerliği tespiti için değerli bir araç haline gelmiştir [20].

Kod karıştırma, bir kod parçasının anlamsal bilgilerini kodlayabilir ve bu sayede kod parçalarının anlamsal benzerliğini belirlemek için kullanılabilir. Örneğin,

bir kod karıştırma modeli, genellikle bir sinir ağı veya derin öğrenme modeli kullanarak bir kod parçasını vektör bir temsile dönüştürür. Bu vektör temsil, kodun anlamsal ve yapısal özelliklerini yansıtır. Kod parçalarının vektör temsilleri arasındaki mesafe, genellikle kod parçalarının anlamsal benzerliğini belirlemek için kullanılır [21].

Bu yaklaşım, kod klonları veya benzer kod parçalarını tespit etme, kod parçalarını sınıflandırma veya kümeleme ve kod tabanında arama yapma gibi bir dizi yazılım mühendisliği görevi için kullanılabilir.

2.3. KOD BENZERLİĞİ(CODE SIMILARITY)

Kod benzerliği (Code Similarity), genellikle iki ya da daha fazla yazılım kodu parçacığı arasındaki benzerlikleri belirlemek için kullanılan bir kavramdır. Bu benzerlikler, kodun dilbilimsel, yapısal ve semantik özelliklerine dayanabilir.

Dilbilimsel özellikler, kodun dil yapısı, anahtar kelimeleri ve dil kurallarını içerirken, yapısal özellikler genellikle kodun kontrol akışı, kod blokları ve fonksiyonların organizasyonu gibi konuları ele alır. Semantik özellikler ise, kodun işlevi, yürütme sonuçları ve çözdüğü problemlerle ilgili konuları kapsar.

Kod benzerliği analizi, yazılım geliştirme ve bakım süreçlerinde çok önemlidir. Bu analizler, aynı kod bloklarının farklı yerlerde yeniden kullanıldığı durumları (kod klonlama), yazılım hatalarını ve güvenlik açıklarını belirlemek için kullanılabilir. Ayrıca, kod benzerliği analizi, yazılımın genel kalitesini, yeniden kullanılabilirliğini ve anlaşılabilirliğini artırmak için de kullanılır.

Birçok farklı benzerlik belirleme tekniği vardır ve her biri, belirli bir benzerlik türünü belirlemek için en uygun olanıdır. Örneğin, bazı teknikler, dilbilimsel benzerlikleri belirlemek için daha uygundurken, diğerleri yapısal veya semantik benzerlikleri belirlemek için daha uygundur. Bu nedenle, en uygun kod benzerliği analizi tekniğini seçmek, belirli bir yazılım projesinin ihtiyaçlarına ve hedeflerine bağlıdır.

Bu çalışmada temelini oluşturan AST tabanlı kod benzerliğinde ise, bir kod parçacığının AST'si, kodun yapısal ve semantik özelliklerini bir ağaç biçiminde gösterir. AST'de, her düğüm kodun bir parçasını (örneğin bir işlem, bir değişken, bir

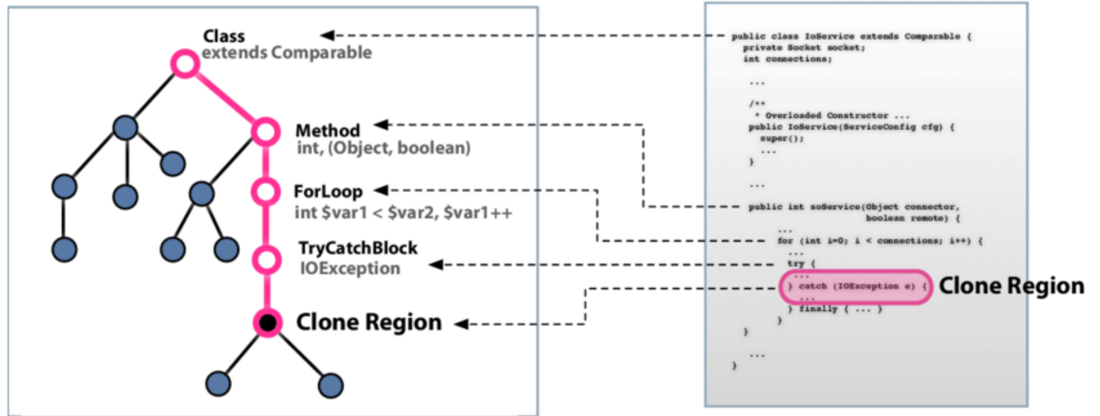
ifade) temsil eder ve düğümler arasındaki bağlantılar kodun yapısal ilişkilerini gösterir [20].

AST tabanlı kod benzerliği, genellikle iki aşamada gerçekleşir[20]:

AST'nin Oluşturulması: İlk aşamada, analiz edilecek her kod parçacığı için bir AST oluşturulur. Bu işlem genellikle bir ayrıştırıcı (parser) tarafından gerçekleştirilir. Ayrıştırıcı, dil kurallarını kullanarak kodu okur ve bir AST oluşturur [23].

AST'ler Arasındaki Benzerliklerin Belirlenmesi: İkinci aşamada, oluşturulan AST'ler arasındaki benzerlikler belirlenir. Bu işlem, genellikle bir ağaç eşleme algoritması tarafından gerçekleştirilir. Ağaç eşleme algoritması, iki AST'yi karşılaştırır ve ağaçların ne kadar benzer olduğunu belirleyen bir skor hesaplar [24].

AST tabanlı kod benzerliği, dilbilimsel benzerliklerden daha çok kodun yapısal ve semantik benzerliklerini belirlemek için kullanılır. Bu, AST tabanlı kod benzerliğinin, kodun yüzeyel özelliklerinden (örneğin değişken isimleri veya yorumlar) daha çok kodun derin yapısal ve semantik özelliklerini analiz etmesini sağlar; bkz. Şekil 2.2 [25]. Bu özellik, kod klonlama, hata tespiti ve kod hırsızlığı gibi durumları belirlemek için AST tabanlı kod benzerliği analizinin sıklıkla kullanılmasını sağlar [26].



Şekil 2.2: Örnek Kod benzerliği Gösterimi [22]

2.4. KOSİNÜS BENZERLİĞİ (COSINE SIMILARITY)

Kosinüs benzerliği, genellikle metin analizi ve bilgi geri alma gibi alanlarda yüksek boyutlu veri setlerindeki öğeler arasındaki benzerliği ölçmek için kullanılan

bir metriktir. İki vektör arasındaki kosinüs açısının değerini hesaplar ve bu değer, iki vektörün birbirine ne kadar benzediğini belirler [27].

Kosinüs benzerliği, vektör uzay modeli adı verilen bir modelde çalışır. Bu modelde, her belge (veya başka bir öğe), belgenin içerdiği terimlerin ağırlıklarını gösteren bir vektör olarak temsil edilir. Terim ağırlıkları, genellikle belge frekansı ve ters belge frekansı (TF-IDF) gibi ölçütler kullanılarak hesaplanır [28].

Kosinüs benzerliği hesaplaması genellikle aşağıdaki adımları içerir:

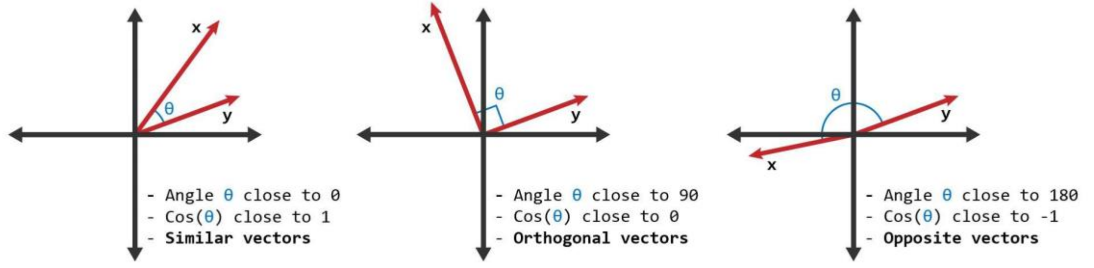
Belgelerin Vektörleştirilmesi: İlk olarak, her belge bir vektör olarak temsil edilir. Bu genellikle "bag of words" veya "TF-IDF" gibi bir teknik kullanılarak yapılır. Bu teknikler, belgedeki her bir terimin ağırlığını hesaplar ve bu ağırlıklar belgenin vektör temsilini oluşturur [29].

Cosine Benzerlik Skorunun Hesaplanması: Daha sonra, iki belge arasındaki kosinüs benzerlik skoru hesaplanır; bkz. Şekil 2.3. Bu, genellikle belgelerin vektör temsillerinin nokta çarpımının, bu vektörlerin büyüklüklerinin çarpımına bölünmesi ile hesaplanır. Bu skor, -1 ile 1 arasında bir değerdir, burada 1 tam benzerlik anlamına gelirken, -1 tamamen farklı anlamına gelir. Genellikle metin belgeleri ile çalışırken, vektörler pozitif değerlere sahip olacağı için cosine benzerlik 0 ile 1 arasında bir değer alır [30].

Kosinüs benzerliği, genellikle metin benzerliği ve bilgi geri alma uygulamalarında kullanılır. Özellikle büyük belge koleksiyonlarında, bir belgeyle en alakalı belgeleri bulmak için kosinüs benzerliği sıklıkla kullanılır. Bunun yanı sıra, metin sınıflandırma, metin kümeleme ve öneri sistemleri gibi uygulamalar da kosinüs benzerliği kullanabilir [31].

Kosinüs benzerliği, genellikle yüksek boyutlu veri setlerinde benzerlikleri belirlemek için etkili bir yöntemdir. Bu, özellikle metin verisi gibi çok sayıda özelliği olan veri setleri için geçerlidir. Bu nedenle, genellikle metin analizi ve bilgi geri alma gibi alanlarda kullanılır [32]. Kosinüs Benzerliği Formülü [33] aşağıdadır.

$$similarity = \cos(\theta) = \frac{A.B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \quad (1)$$



a) 1'e yakın, 0'a yakın ve -1'e yakın benzerliklere sahip iki vektörü gösteren bir grafik. [34]

Şekil 2.3: Kosinüs Benzerliği Formülü ve Grafik Üzerinden Gösterimi

2.5. TRIPLET LOSS

Triplet loss, genellikle öznitelik öğrenimi ve benzerlik öğrenimi uygulamalarında kullanılan bir kayıp fonksiyonudur. Özellikle yüz tanıma gibi alanlarda oldukça popülerdir. Temel fikir, benzer örnekleri birbirine yaklaştırırken farklı örnekleri birbirinden uzaklaştırmaktır.

Triplet loss'u anlamak için, önce bir "triplet"i tanımlamamız gerekir:

Anchor (A): Başlangıç olarak ele alınan bir örnektir.

Positive (P): Anchor ile aynı sınıfa ait olan bir örnektir.

Negative (N): Anchor ile farklı bir sınıfa ait olan bir örnektir.

Triplet loss fonksiyonu şöyle tanımlanır:

$$L(A, P, N) = max(0, D(A, P) - D(A, N) + \alpha) \quad (2)$$

Burada:

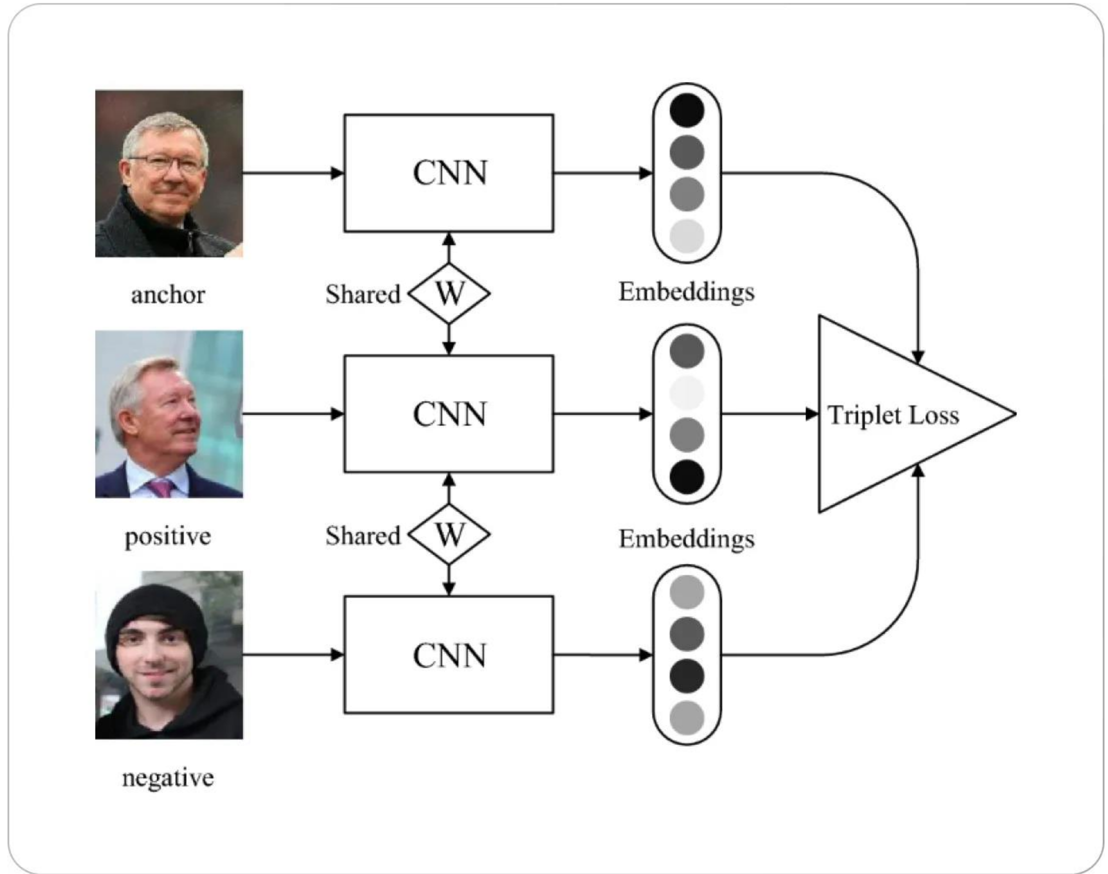
$D(A, P)$ anchor ve positive arasındaki mesafeyi,

$D(A, N)$ anchor ve negative arasındaki mesafeyi temsil eder.

α ise marj olarak adlandırılan ve triplet loss'un doğru çalışması için gerekli olan bir değerdir.

Amacımız $D(A, P)$ mesafesini minimize ederken $D(A, N)$ mesafesini maksimize etmektir. Ancak, bu iki hedef doğrudan çeliştiğinden, marj α bu iki mesafe arasında belirli bir farkın olmasını garanti eder. Yani, pozitif ve anchor

arasındaki mesafenin, negatif ve anchor arasındaki mesafeden en az α kadar daha küçük olmasını istiyoruz.



Şekil 2.4: Triplet Loss Gösterimi [38]

2.5.1. Triplet Seçimi

Triplet loss'da tripletlerin nasıl seçildiği, eğitimin etkinliği ve sonuçların kalitesi için kritiktir. İşte triplet seçimine dair bazı stratejiler:

Rastgele Triplet Seçimi: Başlangıçta, eğitim veri setinden rastgele tripletler seçilebilir. Ancak bu yöntem, genellikle birçok kolay triplet'in (yani model tarafından kolayca doğru bir şekilde sınıflandırılan tripletlerin) seçilmesine yol açar. Bu da eğitimin yavaş ilerlemesine neden olabilir.

Hard Triplet Mining: Bu strateji, eğitimi en çok zorlayacak tripletleri seçmeye odaklanır. "Zor" bir triplet, anchor ve positive örnekleri arasındaki mesafenin, anchor ve negative örnekleri arasındaki mesafeye yakın olduğu bir

triplettir. Bu tür tripletler, modelin daha iyi bir öznitelik kümesi oluşturmasını teşvik eder.

Ancak, sadece zor tripletlerle eğitim yapmak, modelin yerel minimumlarda sıkışmasına veya overfitting yapmasına neden olabilir.

Semi-Hard Triplet Mining: Bu strateji, negative örneğin anchor'dan daha uzakta, ancak positive örneğinden daha yakın olduğu tripletleri seçer. Bu tür tripletler genellikle eğitim için en bilgilendirici olanlardır, çünkü model için ne çok zor ne de çok kolaydırlar.

Semi-hard tripletler, batch içerisindeki tüm örnekler üzerinden geçilerek bulunabilir.

Online Triplet Mining: Bu yöntemde, bir eğitim iterasyonu sırasında, modelin mevcut özelliklerini kullanarak zor ya da semi-hard tripletler dinamik olarak seçilir. Bu, modeli sürekli olarak zorlayarak ve eğitim sürecini hızlandırarak modelin daha hızlı konverjansını sağlar.

Distance-based Sampling: Bu yöntemde, anchor'a en yakın positive örnekler ve en uzak negative örnekler seçilerek tripletler oluşturulur. Bu, benzer özniteliklerin birbirine yakın, farklı özniteliklerin ise birbirinden uzak olmasını teşvik eder.

Bu seçim stratejilerinin her biri, farklı avantajlar ve zorluklar sunar. Genellikle, hard veya semi-hard triplet mining stratejileri en bilgilendirici tripletleri seçerek eğitim sürecini en iyi şekilde optimize eder. Ancak hangi stratejinin en uygun olduğunu belirlemek için deney yapmak ve sonuçları değerlendirmek genellikle en iyisidir.

2.5.2. Triplet Loss Öğrenme Süreci Optimizasyonu

Triplet loss kullanılarak yapılan öğrenme sürecinin optimize edilmesi, doğru tripletlerin seçilmesi ve uygun bir eğitim stratejisinin benimsenmesi ile yakından ilgilidir. İşte bu sürecin optimize edilmesine yönelik bazı yöntemler:

Zor Triplet Seçimi (Hard Triplet Mining): Eğitim setindeki tüm olası triplet kombinasyonlarını kullanmak hem pratikte zor olabilir hem de eğitimin konverjansını yavaşlatabilir. Bu nedenle, eğitimi en çok etkileyecek "zor" tripletleri seçmek faydalıdır. Zor tripletler, positive ve negative örnekleri arasındaki mesafenin

benzer olduğu durumlardır. Bu tür tripletler modeli daha fazla düzeltmeye zorlar ve daha iyi bir öznitelik kümesi sağlar.

Semi-Hard Triplet Mining: Zor tripletlerin yanı sıra, negative örneğin anchor'dan biraz daha uzak olduğu, ancak hala positive'dan daha uzak olmadığı tripletler de etkili olabilir. Bu tür tripletlere "semi-hard" tripletler denir.

Aktif Öğrenme: Modelin belirsiz olduğu veya hatalı sınıflandırmalar yaptığı bölgelerde yeni tripletler seçmek için aktif öğrenme stratejileri kullanılabilir.

Dinamik Marj: Sabit bir marj yerine, dinamik bir marj kullanmak bazen daha etkili olabilir. Bu, marjın eğitim sürecinde adaptif olarak değiştirildiği anlamına gelir.

Çoklu Görev Öğrenme (Multi-Task Learning): Triplet loss dışında başka bir kayıp fonksiyonu (örneğin, çapraz entropi kaybı) ile birleştirerek modelin farklı görevlerde eşzamanlı olarak öğrenmesini sağlamak da faydalı olabilir.

Düzenlileştirme ve Veri Artırma: Modelin genelleme yeteneğini artırmak için düzenlileştirme yöntemleri (L1, L2, dropout vb.) ve veri artırma teknikleri kullanılabilir.

Öğrenme Oranı Politikaları: Başlangıçta yüksek, sonra azalan bir öğrenme oranı kullanmak, modelin daha hızlı konverjansını ve daha iyi bir yerel minimuma ulaşmasını sağlayabilir.

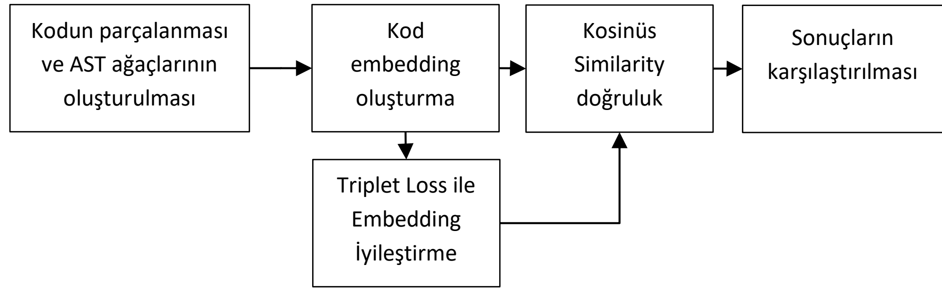
Özetle, triplet loss ile öğrenme sürecini optimize etmek için modelin zorluklarla karşılaşmasını sağlamak, doğru tripletleri seçmek ve eğitim sürecini dikkatlice yönetmek esastır. Bu, modelin daha iyi öznitelikler öğrenmesine ve genelleme yeteneğini artırmasına yardımcı olur.

Triplet loss'u etkili bir şekilde kullanabilmek için genellikle büyük miktarda veriye ve triplet kombinasyonlarına ihtiyaç duyarız. Bunun yanı sıra, doğru tripletleri seçmek (yani, zor negative'leri ve yakın positive'leri seçmek) eğitimin daha etkili olmasını sağlar.

ÜÇÜNCÜ BÖLÜM

3. YÖNTEM

Geliştirilecek sistemin girişi bir yazılımın tüm kodu, çıktısı ise bu kod üzerinde kalite ve yeniden düzenleme yapılması gereken konumların sınıf, metot ve kod satırı temelinde adresi ve türüyle ilgili önerilerdir. Hedeflenen sistemi geliştirmek için uygulanacak temel adımlar verilerin toplanması için bir sistem geliştirilmesi, kodun AST yapısının çıkartılması, kodun satır ve blok düzeyinde analize imkân veren vektör temsili haline dönüştürülmesi, derin öğrenme sisteminde eğitim, eğitim modellerinin transfer edilmesi ve önerilerin doğruluğunun test edilmesidir; bkz Şekil 3.1.



Şekil 3.1 Sistemin ana akışı

3.1. VERİ SETİ OLUŞTURMA SÜRECİ

AST (Abstract Syntax Tree) ve kosinüs benzerliği tabanlı kod benzerliği analizinde bir veri seti oluşturmak, genellikle aşağıdaki adımları içerir:

3.1.1. Kod Örneklerinin Toplanması:

İlk adım, analiz etmek istediğiniz kod örneklerini toplamaktır. Bu genellikle bir veya daha fazla yazılım projelerinden alınan kaynak kod dosyalarıdır.

Tez için kullanılan kod örnekleri belli başlı sıralama algoritmalarının C# kodları kullanılarak işlemler yapılmışdır; bkz. Şekil 3.1 .

```
1    class MergeSort {
76   class QuickSort {
125  class SelectionSort{
159  class InsertionSort {
160
161      void sort(int[] arr)
162      {
163          int n = arr.Length;
164          for (int i = 1; i < n; ++i) {
165              int key = arr[i];
166              int j = i - 1;
167
168              while (j >= 0 && arr[j] > key) {
169                  arr[j + 1] = arr[j];
170                  j = j - 1;
171              }
172              arr[j + 1] = key;
173          }
174      }
175
176      static void printArray(int[] arr)
177      {
178          int n = arr.Length;
179          for (int i = 0; i < n; ++i)
180              Console.Write(arr[i] + " ");
181
182          Console.Write("\n");
183      }
184
185      public static void Main()
186      {
187          int[] arr = { 12, 11, 13, 5, 6 };
188          InsertionSort ob = new InsertionSort();
189          ob.sort(arr);
190          printArray(arr);
191      }
192  }
193  class HeapSort {
251  class GFG {
290  class CountingSort {
331  class RadixSort {
387  class BingoSortAlgorithm {
451  class ShellSort{
```

Şekil 3.1: Kaynak Kod Örnekleri

3.1.2. AST Oluşturma

Her bir kaynak kod dosyası, metot ve kod bloğu için, bir AST oluşturulması gerekmektedir. AST, kodun yapısını hiyerarşik bir şekilde gösterir. AST'ler, genellikle bir derleyici veya yorumlayıcı tarafından kodun sözdizimi analizi aşamasında oluşturulur.

Bu çalışmada code2vec [3] projesinde C# için geliştirilen kaynak kodun yürüyüş (walker) yapısı baz alınarak ayrı bir uygulama geliştirildi. Code2vec projesinin temelini oluşturan özellikle AST tabanlı bir yaklaşım kullanıldı. Bu, bir

kod parçasının AST'sini alır ve bu ağaçtan belirli bir yürüyüş (walker) stratejisi kullanarak özellikler (features) çıkarır. Bu özellikler daha sonra vektör temsillerine dönüştürüldü.

Walker yapısı, bir AST üzerinde belirli bir şekilde gezinmeyi ifade eder. AST'nin düğümlerini belirli bir sırayla ziyaret eder ve her bir ziyaret, çıktı vektörünün bir öğesi olarak kodlanır. Bu yaklaşım, kodun yapısal özelliklerini korurken aynı zamanda kodu bir vektör biçimine dönüştürme yeteneği sağlar. Bu, derin öğrenme modellerinin kod üzerinde etkili bir şekilde çalışmasını sağlar.

Örneğin, bir walker, bir AST'nin kök düğümünden başlar ve AST'deki belirli düğümler arasında yürür. Bu yürüyüşler, kodun belirli bir parçasının vektör temsilini oluşturmak için birleştirilir. Her bir yürüyüş, belirli bir kod parçasının semantiğine ilişkin bilgi sağlar.

Şekil 3.2.'de de görüleceği gibi oluşturulan her bir ağaç(AST satırı), sonraki adımlarda kullanılmak ve veri seti oluşturmak üzere kullanılacak olan, ilgili kod bloğunun ağaç yapısı, satır aralığı, AST içeriği ve kodun bloğunun kendisini içerir.

```
1.1-->12 13 #ForStatement SimpleAssignmentExpression IdentifierName NumericLiteralExpression
 LessThanExpression IdentifierName IdentifierName PreIncrementExpression IdentifierName
 ExpressionStatement SimpleAssignmentExpression ElementAccessExpression  IdentifierName
 BracketedArgumentList Argument IdentifierName ElementAccessExpression IdentifierName
 BracketedArgumentList Argument AddExpression IdentifierName IdentifierName
 $          for (i = 0; i < n1; ++i)             L[i] = arr[l + i];

1.2-->14 15 #ForStatement SimpleAssignmentExpression IdentifierName NumericLiteralExpression
 LessThanExpression IdentifierName IdentifierName PreIncrementExpression IdentifierName
 ExpressionStatement SimpleAssignmentExpression ElementAccessExpression IdentifierName
 BracketedArgumentList Argument IdentifierName ElementAccessExpression IdentifierName
 BracketedArgumentList Argument AddExpression AddExpression IdentifierName
 NumericLiteralExpression IdentifierName
 $        for (j = 0; j < n2; ++j)             R[j] = arr[m + 1 + j];
```

Şekil 3.2: Oluşturulan AST Örnekleri

3.1.3. Vektör Uzayı Modellerinin Oluşturulması

Her bir AST'yi, kodun semantik özelliklerini temsil eden sayısal bir vektör uzayı modeline dönüştürmeniz gerekmektedir. Bu genellikle kodun belirli özelliklerini (örneğin, kullandığı işlemler, çağrılan fonksiyonlar) belirleyerek ve bu özelliklerin her birini vektörün bir bileşeni olarak kullanarak yapılır [2]. Kod embedding oluşturulması aşamasında kod içindeki anahtar kelime ve isimlendirmelerin tokenlere çevrilmesi, kelime sıklığına göre kodlama gibi aşamalar

gerçekleştirilir. Vektörlerin oluşturulmasında Word2Vec modelinden faydalanılmıştır.

3.1.4. Kosinüs Benzerliği Hesaplama:

Ardından, kosinüs benzerliği kullanarak her bir vektör çifti arasındaki benzerlik hesaplanmaktadır. Bu işlem gerçekleştirilirken, her bir kod bloğu, benzerlik oranı yüksek çıktığı için ve asıl amaç olan diğer bloklar ile karşılaştırıp benzerlikleri tespit etmek olduğu için kendi alt kırılımı olan kod bloklarıyla karşılaştırılmamaktadır.

Python yazılım dili ile Gensim kütüphanesinde bulunan Word2Vec fonksiyonu kullanılarak benzerlik oranları çıkartıldı.

3.1.5. Veri Setinin Oluşturulması

En son adımda, her bir vektör çifti için hesaplanan kosinüs benzerlik değerleri ve belki de diğer özellikler, benzerlik analizi için bir veri setini oluşturmak üzere birleştirdi. Bu veri seti, daha sonra makine öğrenmesi veya istatistiksel analiz için de kullanılabilir [20].

Veri setinde eşleştirilen kodların AST ağaçları, kodların kendisi, aralarındaki kosinüs benzerliği, ayrıca kod verisinin alındığı düzey gösterilmiştir. Kod verisi bir ağaç mantığı ile bölümlenerek alt kırılımları için bir id üretilmiştir. Kullanılan ID biçiminde İlk düzey kod bloğunun ana ID'si olmak üzere her seviye nokta ile ayrılmıştır. Örneğin 1.3.4 birinci kod bloğundaki 3. alt bloktaki 4 kısmı temsil etmektedir.

3.1.6. Triplet Loss Benzerlik Hesaplama:

Bu çalışmada, daha önce kosinüs benzerliği ile oluşturulan veri kümesi baz alınarak, benzerlik oranı yüksek olup gerçekte de benzer olan kod blokları "positive", benzerlik oranı yüksek çıkıp gerçekte benzer olmayan kod blokları ise "negative" olarak işaretlendi.

Her bir kod bloğu farklı boyutlarda olduğu için girdi verktörlerinin boyutları farklıdır. Bu da çıkan sonuçlarda yanlış bir benzerlik oluşturmaktadır. Bu kısımlar çıkartılarak veriler normalize edildi.

Çıkan yeni kod temsilleri üzerine kosinüs benzerliği tekrar uygulandı ve ortalama dikkate alınarak bir eşik belirlendi. Belirlenen bu eşik değere göre eşik değerinden daha düşük olanlar benzemeyen, daha yüksek olanlar ise benzeyen olarak kabul edildi.

3.1.7. Korelasyon Oluşturulması:

Moss [Url-2] (Yazılım Benzerliğinin Ölçüsü için), programların benzerliğini belirleyen otomatik bir sistemdir. Bugüne kadar Moss'un ana uygulaması programlama derslerinde intihal tespiti olmuştur. Bu çalışmada da bu uygulama kullanılarak Şekil 3.3'te gösterilen veri setleri baz alınarak, karşılaştırılacak olan her bir kod bloğu ayrı ayrı C# kod dosyalarına eklendi ve MossSwing uygulamasına girdi olarak verildi. Çıkan değerler ve sonuçlar ile triplet loss çıktıları karşılaştırılarak değerlendirildi.

DÖRDÜNCÜ BÖLÜM

4. TEST SONUÇLARI VE YORUM

Çıktı olarak her bir kod bloğunun ve metodun diğer kod bloğu veya metot ile olan benzerliklerini doğru bir şekilde bulması hedeflenmektedir. Benzerlik oranlarının doğrulunu arttırmak, çok kısa kod satırlarının(değişken atama, throw v.b.) benzerliklerini çıkartmak için AST kelime sayısı 3 ve daha az olan kod satırları işleme alınmadı; bkz. Şekil 4.1 . Bu düzenlemeler ile birlikte 489 satılık bir kod sınıfı içerisindeki kod blokları ve metotlara ait her bir AST satırı birbiri ile tek tek karşılaştırılarak toplamda 2930 defa kosinüs benzerliği yöntemi ile benzerlik dereceleri çıkartıldı.

```
ArgumentList
    Argument
        ObjectCreationExpression
            IdentifierName
            ArgumentList
    Argument
        NullLiteralExpression
```

Şekil 4.1: İşleme Alınmayan AST Veri Seti Örnekleri

Ek olarak, doğru kod bloğu ile benzerliklerin bulunması için her bir metot veya kod bloğu kendi bir alt kırılımı olan bloklar ile karşılaştırılmadı.

Veri seti oluşturma aşamasında Şekil 4.2'de de görüldüğü üzere, her bir kod bloğu diğer bir kod bloğu ile karşılaştırılarak benzerlik oranları tespit edildi. Çıkan benzerlik oranları içerisinde yüksek oranlı olanlar işaretlenerek doğrulukları kontrol edildi ve ilgili sonuçlar "label" kolonuna da işlenerek veri seti oluşturuldu.

id	id1	AST1	AST2	func1	func2	Result	Tree1	Tree2	label
1	1: 2 similarty =	ForStatement SimpleAssig	ForStatement Simple.	for (i = 0; i < r	for (j = 0; j	0.992	1.1	1.2	1
2	1: 27 similarty	ForStatement SimpleAssig	ForStatement Simple.	for (i = 0; i < r	for (j = (	0.968	1.1	19.1.1	0
3	1: 43 similarty	ForStatement SimpleAssig	ForStatement Simple.	for (i = 0; i < r	// digit	0.984	1.1	25.5	1
4	1: 48 similarty	ForStatement SimpleAssig	ForStatement Variab	for (i = 0; i < i	for (int i = s	0.968	1.1	30.1.1	0
5	2: 1 similarty =	ForStatement SimpleAssig	ForStatement Simple.	for (j = 0; j < n:	for (i = 0;	0.992	1.2	1.1	1
6	2: 27 similarty	ForStatement SimpleAssig	ForStatement Simple.	for (j = 0; j < n:	for (j = (	0.977	1.2	19.1.1	0
7	2: 43 similarty	ForStatement SimpleAssig	ForStatement Simple.	for (j = 0; j < n:	// digit	0.973	1.2	25.5	0
8	3: 5 similarty =	WhileStatement LogicalAr	WhileStatement Less	while (i < n1 &	while (i <	0.982	1.3	1.4	0
9	3: 6 similarty =	WhileStatement LogicalAr	WhileStatement Less	while (i < n1 &	while (j <	0.982	1.3	1.5	0
10	4: 28 similarty	IfStatement LessThanOrE	IfStatement GreaterT	if (L[i] <= R[j	if (arr	0.972	1.3.1	19.1.1.1	0
11	4: 48 similarty	IfStatement LessThanOrE	ForStatement Variabl	if (L[i] <= R[j	for (int i = s	0.971	1.3.1	30.1.1	0
12	4: 49 similarty	IfStatement LessThanOrE	IfStatement EqualsEx	if (L[i] <= R[j	if (vec[i] =	0.983	1.3.1	30.1.1.1	0
13	5: 3 similarty =	WhileStatement LessThan	WhileStatement Logi	while (i < n1)	while (i < r	0.982	1.4	1.3	0
14	5: 4 similarty =	WhileStatement LessThan	IfStatement LessThar	while (i < n1)	if (L[i] <	0.957	1.4	1.3.1	0
15	5: 6 similarty =	WhileStatement LessThan	WhileStatement Less	while (i < n1)	while (j <	1.000	1.4	1.5	1
16	6: 3 similarty =	WhileStatement LessThan	WhileStatement Logi	while (j < n2)	while (i < r	0.982	1.5	1.3	0
17	6: 5 similarty =	WhileStatement LessThan	WhileStatement Less	while (j < n2)	while (i <	1.000	1.5	1.4	1
18	7: 11 similarty	IfStatement LessThanExp	IfStatement LessThar	if (l < r) {	if (low < hi	0.993	2.1	7.1	0
19	8: 12 similarty	ForStatement VariableDe	ForStatement Variabl	for (int i = 0; i	for (int i =	0.986	3.1	8.1	1
20	8: 16 similarty	ForStatement VariableDe	ForStatement Variabl	for (int i = 0; i	for (int i=0	1.000	3.1	10.1	1
21	8: 19 similarty	ForStatement VariableDe	ForStatement Variabl	for (int i = 0; i	for (int i =	1.000	3.1	13.1	1
22	8: 25 similarty	ForStatement VariableDe	ForStatement Variabl	for (int i = 0; i	for (int i =	1.000	3.1	17.1	1
23	8: 30 similarty	ForStatement VariableDe	ForStatement Simple.	for (int i = 0; i	for (i = 0; i	0.959	3.1	20.1	1
24	8: 36 similarty	ForStatement VariableDe	ForStatement Variabl	for (int i = 0; i	for (int i =	0.977	3.1	23.1	1
25	8: 45 similarty	ForStatement VariableDe	ForStatement Variabl	for (int i = 0; i	for (int i =	0.986	3.1	27.1	1
26	8: 51 similarty	ForStatement VariableDe	ForStatement Variabl	for (int i = 0; i	for (int i = 0;	0.978	3.1	31.1	1

Şekil 4.2: Oluşturulan AST Veri Seti Örnekleri

Bu benzerlik derecekleri 0.90 üzeri olanlardan en yüksek 3'er tane baz alınarak doğrulukları ölçüldü ve gözlemlendi. Bu şekilde yapılan kosinüs benzerliği karşılaştırmasıyla 182 karşılaştırmadan 112 karşılaştırma sonucunun doğru olduğu, 70 karşılaştırmanın yanlış olduğu görüldü. Şekil 4.3'te de görülebileceği gibi, kod içeriklerinde sadece parametre isimleri, değişken türleri farklıysa veya bu kod bloğu için ortak bir metot oluşturup, bu metot her iki tarafta da kullanılabilecekse doğru kabul edildi, aksi halde yanlış olarak kabul edildi.

```
while (i < n1) {
  arr[k] = L[i];
  i++;
  k++;
}
```

```
while (j < n2) {
  arr[k] = R[j];
  j++;
  k++;
}
```

(a) Doğru Benzerlik Kod Örneği

```
if (l < N && arr[l] > arr[largest])
  largest = l;
```

```
if (arr[i] > mx)
  mx = arr[i];
```

(b) Yanlış Benzerlik Kod Örneği

Şekil 4.3: Kod benzerlik Örnekleri

Bu şartlar ve metotlar izlenerek sonuçlara bakıldığında ise Tablo 4.1'de görüleceği gibi kosinüs benzerlik sonucunun %61,54 doğruluk oranı olduğu görüldü.

Tablo 4.1. Model Doğruluk Oranları

Kosinüs Benzerliği Sonuçları	Sayı	Doğru-Yanlış
61,54	112	Doğru
	70	Yanlış

Triplet loss benzerliği için ise, kosinüs benzerliği sonuçları baz alındı. Her bir kod bloğun kendisi "anchor" olarak seçildi. Daha önceki kosinüs benzerliği sonuçlardan 1 olanlar "positive" 0 olanlar ile "negative" olarak belirlendi, bkz. Şekil 4.4 .

Anchor	Positive	Negative
ForStatement :	ForStatement :	ForStatement :
ForStatement :	ForStatement :	ForStatement :
ForStatement :	ForStatement :	ForStatement :
ForStatement :	ForStatement :	ForStatement :
ForStatement :	ForStatement :	ForStatement :
ForStatement :	ForStatement :	ForStatement :
ForStatement :	ForStatement :	ForStatement :
WhileStatemei	WhileStatemei	WhileStatemei
WhileStatemei	WhileStatemei	WhileStatemei
IfStatement Le	IfStatement Le	IfStatement Gr
IfStatement Le	IfStatement Le	IfStatement Gr
IfStatement Le	IfStatement Le	IfStatement Gr
WhileStatemei	WhileStatemei	WhileStatemei
WhileStatemei	WhileStatemei	IfStatement Le
WhileStatemei	WhileStatemei	WhileStatemei
WhileStatemei	WhileStatemei	WhileStatemei
WhileStatemei	WhileStatemei	WhileStatemei
IfStatement Le	IfStatement Le	IfStatement Le
ForStatement '	ForStatement '	ForStatement :
ForStatement '	ForStatement '	ForStatement :
ForStatement '	ForStatement '	ForStatement :

Şekil 4.4: Oluşturulan Triplet Loss Veri Seti Örnekleri

Verinin dağılımına bağlı olarak modelin eğitim ve testinde oluşabilecek sapmaları engellemek için Çapraz kat doğrulaması (k-fold cross validation) tekniği ile veri kümesi parçalara ayrıldı. Burada *k* değeri beş olarak seçilmiştir. Yeni

oluşturulan veri kümesinin %20'si rasgele seçilerek test verisi, kalan veriler de eğitim verileri olacak şekilde triplet loss uygulandı.

Kod bloklarının farklı boyutlarda olmasından dolayı oluşan farklı vektör boyutları yanlış benzerlik oluşmasına sebep olduğundan dolayı bu kısımlar çıkartılarak veriler normalize edildi. Eğer daha önceki kosinüs benzerliği 0 ise "anchor" ile "negative", 1 ise de "anchor" ile "positive" benzerliği kosinüs benzerliği uygulanarak tekrar karşılaştırıldı.

Çıkan embeddingler üzerine ortalama dikkate alınarak belirlenen eşik değere göre düşük olanlar benzemeyen, yüksek olanlar ise benzeyen olarak kabul edildi ve Şekil 4.5'te de görüleceği üzere daha önce "benzer" çıkan sonuçlardan triplet loss sonrası karşılaştırmada da benzer çıkan veya daha önce "benzemeyen" çıkan sonuçlardan triplet loss sonrası karşılaştırmada da benzemeyen çıkan sonuçlar doğru(true), diğerleri ise yanlış (false) olarak işaretlenerek değerlendirildi. Çapraz kat doğrulaması (cross fold validation) tekniği kullanılarak bu işlemler 5 defa tekrarlandı ve çıkan sonuçların ortalaması alınarak nihai sonuçlar çıkartıldı.

label	distance-DENSE-100-layer	True-False-dense-100	label3
1	0,840259552	SIMILAR	1
0	0,430775642	NOT_SIMILAR	1
1	0,840259552	SIMILAR	1
0	0,534487724	NOT_SIMILAR	1
1	0,840259552	SIMILAR	1
0	0,6560812	NOT_SIMILAR	1
0	0,560713768	NOT_SIMILAR	1
0	0,665021896	NOT_SIMILAR	1
0	0,665021896	NOT_SIMILAR	1
0	0,529719353	NOT_SIMILAR	1
0	0,573230743	NOT_SIMILAR	1
0	0,767541885	SIMILAR	0
0	0,665021896	NOT_SIMILAR	1
0	0,111890793	NOT_SIMILAR	1
1	1,002384186	SIMILAR	1
0	0,665021896	NOT_SIMILAR	1
1	1,002384186	SIMILAR	1
0	0,887943268	SIMILAR	0
1	0,786615372	SIMILAR	1
1	0,786615372	SIMILAR	1
1	0,786615372	SIMILAR	1

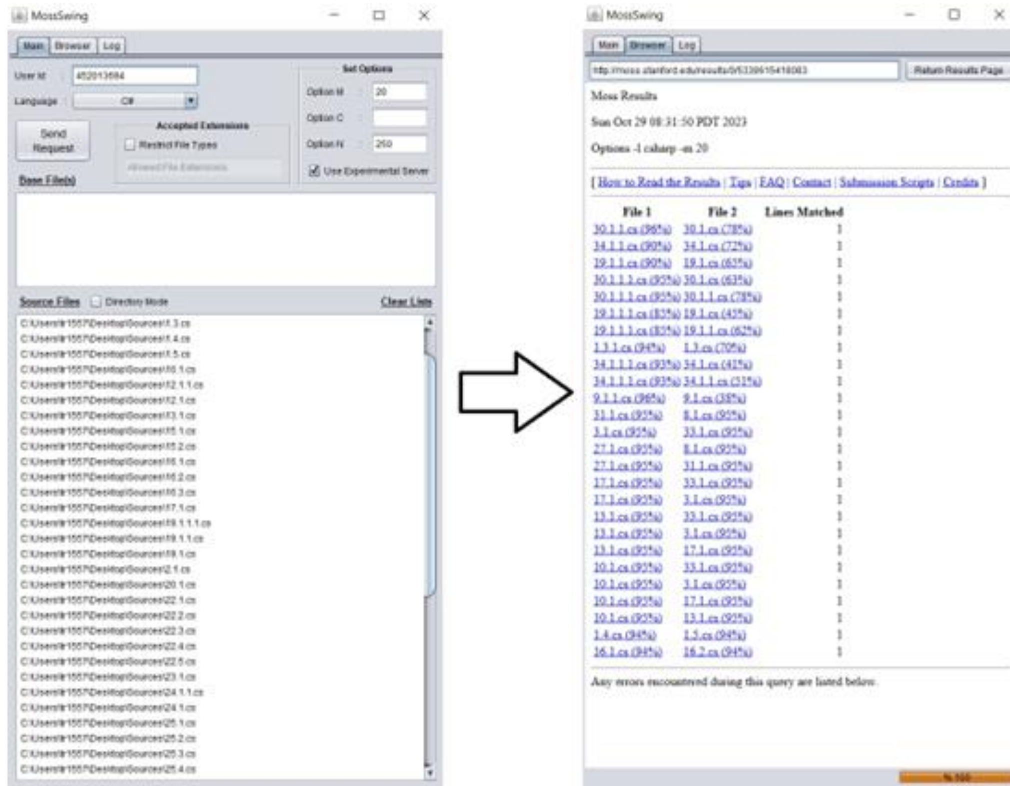
Şekil 4.5: Oluşturulan Triplet Loss Sonuç Örneleri

Geliştirilen sistemden çıkan sonuçlara bakıldığında ise Tablo 4.2'de görüleceği gibi triplet loss uygulanmış kosinüs benzerlik sonucunun %88,68 doğruluk oranı olduğu görüldü.

Tablo 4.2. Model Doğruluk Oranları

Triplet Loss Benzerlik Sonuçları	Doğruluk Oranı
Test 1	89,7
Test 2	91,6
Test 3	86,1
Test 4	86,4
Test 5	89,64
Ortalama	88,68

Test kümesinde bulunan kod blokları tek tek dosyalara aktarılıp MossSwing [Url-2] kullanılarak ilgili kod blokları arasındaki benzerlik incelendiğinde ise, 27 kod bloğu arasında benzerlik olduğu tespit edildi; bkz. Şekil 4.6.



Şekil 4.6: MossSwing kullanılarak kod benzerliklerinin bulunması [Url-2]

Tespit edilen benzerliklerden 11 karşılaştırmanın kendi iç kod bloğu ile ilgili olduğu için (daha önceki benzerlik karşılaştırmalarında kendi iç blok karşılaştırmaları dikkate alınmadığı için) dikkate alınmadı. Geri kalan benzerlik sonuçlarından 16 benzerliğin kosinüs benzerliği karşılıklı sonuçlarından 29'u ile uyuştuğu ve tamamının doğru şekilde tespit edildiği görülmüştür, bkz. Şekil 4.7 .

B	C	D	E	F	G	H	I	J	
Column2	Func1 AST	Func2 AST	Kod1	Kod2	Result	Tree1	Tree2	Kosinüs Benzerliği	Ko
5: 6 similarty = 1.000	WhileStatement LessThai	WhileStatement LessThanExp	while (i < n1) { arr[k] = L[i]; i++; k++; }	while (j < n2) { arr[k] = R[j]; j++; k++; }	1.000	1.4	1.5	1	
6: 5 similarty = 1.000	WhileStatement LessThai	WhileStatement LessThanExp	while (j < n2) { arr[k] = R[j]; j++; k++; }	while (i < n1) { arr[k] = L[i]; i++; k++; }	1.000	1.5	1.4	1	
8: 12 similarty = 0.986	ForStatement VariableDe	ForStatement VariableDeclar	for (int i = 0; i < n; ++i) Console.Write(arr[i] + " ");	for (int i = 0; i < N; i++) Console.Write(arr[i] + " ");	0.986	3.1	8.1	1	
8: 16 similarty = 1.000	ForStatement VariableDe	ForStatement VariableDeclar	for (int i = 0; i < n; ++i) Console.Write(arr[i] + " ");	for (int i=0; i<n; ++i) Console.Write(arr[i]+" ");	1.000	3.1	10.1	1	

Şekil 4.7: Korelasyon Sonucu

Sonuçlar ayrıntılı incelendiğinde Moss'un daha çok birbirine yakın veya tekrar eden kodları tespit ettiği görülmüştür. Geliştirilen sistem farklı kod benzerliklerini de belirleyebilmektedir.

SONUÇ

Bu tez çalışmasında, kod benzerliği analizi için AST (Abstract Syntax Tree), kosinüs benzerliği ve ve triplet loss tabanlı bir yaklaşım sunulmuştur. Bu yaklaşımın temel avantajı, kodun semantik ve yapısal özelliklerini birleştiren bir benzerlik metriği oluşturabilmesidir. Ayrıca, bu yaklaşımın kullanımı, çeşitli yazılım projelerinde kod klonlarının etkin bir şekilde tespit edilmesini sağlamıştır. Code2Vec çalışması, kosinüs benzerliği ve triplet loss kavramlarının temelleriyle ortaya çıkarak geliştirilmiş yeni bir yaklaşım ve veri kümesi içermektedir.

Bu çalışmada oluşturulan veri seti, kapsamlı bir şekilde kod benzerliğini incelememize ve bir dizi farklı durumda kod benzerliğini tespit etmemize olanak sağlamıştır. Bu, özellikle kod klonlama ve yazılım bakımı alanlarında değerli bilgiler sunmaktadır.

Ancak, yaklaşımımızın uygulanabilirliği ve genel geçerliliği hala daha fazla araştırma gerektirmektedir. Özellikle, daha fazla yazılım dili ve projede kullanılabilirliğini test etmek, sonuçlarımızın genel geçerliliğini artırabilir. Ayrıca, AST, kosinüs benzerliği ve triplet loss metriğinin özellikle karmaşık ve büyük ölçekli kod tabanlarında ne kadar etkili olduğunu belirlemek için daha fazla çalışmaya ihtiyaç duyulmaktadır.

Gelecek çalışmalarda, farklı benzerlik metrikleri ve AST'lerin farklı biçimlerini deneyerek bu yaklaşımın daha da geliştirilmesi gerekmektedir. Ayrıca, makine öğrenmesi ve diğer otomatikleştirme tekniklerinin bu süreci nasıl daha da iyileştirebileceğini araştırmak da ilgi çekici bir araştırma alanı olacaktır.

KAYNAKÇA

[1] **C. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani and J. Vitek,** "*DéjàVu: A Map of Code Duplicates on GitHub*," in Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '17), 2017.

[2] **I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier,** "*Clone detectiob using abstract syntax trees*," in Proceedings of the International Conference on Software Maintenance (ICSM '98), 1998, pp. 368-377.

[3] **Alon, U., Zilberstein, M., Levy, O. & Yahav, E.** (2019). code2vec: learning distributed representations of code. *Journal of Proceedings of the ACM on Programming Languages*, 3(40), 1-29.

[4] **Silva, D., Tsantalis N. & Valente, T. M.** (2016). code2vec: learning distributed representations of code. *FSE' 16 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Seattle, USA : November.

[5] **Mikolov, T., Chen, K., Corrado, S. G. & Dean, J.** (2013). Efficient Estimation of Word Representations in Vector Space. *International Conference on Learning Representations*, USA : Arizona, May 2-3. [5] **Xing, Z. & Stroulia, E.** (2005). An algorithm for object-oriented design differencing. *In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, (pp.54-65). USA : New York, November.

[6] **Sui, Y., Cheng, X., Zhang, G. & Wang, H.** (2020). Flow2Vec: Value-Flow-Based Precise Code Embedding. *Journal of Proc. ACM Program*, 4(233), 1-27.

[7] **Alon, U., Brody, S., Levy, O. & Yahav, E.** (2019). code2seq: Generating Sequences from Structured Representations of Code. *Seventh International Conference on Learning Representations (ICLR)*, USA : New Orlands, May 6-9.

[8] **Microsoft Azure, Bilişsel Hizmetler.** (2020). *BLEU Puanı Nedir?* [Teknik Yazı Blog]. Erişim adresi

https://docs.microsoft.com/tr-tr/azure/cognitive-services/translator/custom-translator/what-is-bleu-score

[9] **Jain, P., Jain, A. K., Zhang, T., Abbeel, P., Gonzalez, E. J. & Stoica, I.** (2020). Contrastive Code Representation Learning. *ArXiv Preprint*. (pp.1-20). October 9.

[10] **Azcona, D., Arora, P., Hsiao, I. & Hindle, A.** (2019). user2code2vec: Embeddings for Profiling Students Based on Distributional Representations of Source Code. *LAK19: Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, (pp.86-95). USA : Arizona, March.

[11] **Ďuračík, M., Kršák, E., & Hrkút, P.** (2018). Source code representations for plagiarism detection. *LTEC: International Workshop on Learning Technology for Education in Cloud*, Springer (pp.61-69). Slovakia : Zilina, August 6-10.

[12] **Microsoft Corporation.** (2012). *The Roslyn Project: Exposing the C# and VB compiler's code analysis* [Technical Whitepaper]. Retrieved from

https://download.microsoft.com/download/E/A/D/EADEC33E-FBA3-43BF-9226-427BDAC27610/Roslyn%20Project%20Overview.docx

[13] **Le, M. H. T., Chen, H., & Babar, A. M.** (2020). Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *Journal of ACM Comput. Surv.*, 3(62), 1-38.

[14] **Allamanis, M., Barr, T. E., Devanbu, P. & Sutton, C.** (2018). A Survey of Machine Learning for Big Code and Naturalness. *Journal of ACM Computing Surveys,* 51(4), 1-36.

Dataset: http://learnbigcode.github.io/datasets/

[15] **Shi, K., Lu, Y., Chang, J. & Wei, Z.** (2020). PathPair2Vec: An AST path pair-based code representation method for defect prediction. *Journal of Computer Languages,* 59.

[16] **Allamanis, M., Brockschmidt, M. & Khademi, M.** (2018). Learning to Represent Programs with Graphs. *ICLR 2018: Sixth International Conference on Learning Representations: Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, (pp.86-95). Canada : Vancouver, April 30-May 3.

[17] **Wang, W., Li, G., Shen, S., Xia, X. & Jin, Z.** (2020). Modular Tree Network for Source Code Representation Learning. *Journal of ACM Trans. Softw. Eng. Methodol,* 29(4), 1-23.

[18] **Chen, Z. & Monperrus, M.** (2019). A Literature Study of Embeddings on Source Code. *ArXiv Preprint*. (pp.1-8). April 5.

[19] **Büch, L. & Andrzejak, A.** (2019). Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, (pp.86-95). China : Hangzhou, February 24-27.

[20] **L. Jiang, G. Misherghi, Z. Su, and S. Glondu,** "*Deckard: Scalable and accurate tree-based detection of code clones*," in Proceedings of the 29th international conference on Software Engineering, 2007, pp. 96-105.

[21] **M. White, M. Tufano, C. Vendome, and D. Poshyvanyk**, "*Deep learning code fragments for code clone detection*," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 87-98.

[22] **N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan,** *"An empirical study on inconsistent changes to code clones at the release level,"* Sci Comput Program, vol. 77, no. 6, pp. 760–776, Jun. 2012, doi: 10.1016/J.SCICO.2010.11.010.

[23] **Rattan, D., Bhatia, R., & Singh, M.** (2013). *Software clone detection: A systematic review. Information and Software Technology*, 55(7), 1165-1199.

[24] **Wise, M. J.** (1996). *String similarity via greedy string tiling and running Karp-Rabin matching*. Department of Computer Science and Software Engineering, University of Sydney.

[25] **Roy, C. K., Cordy, J. R., & Koschke, R**. (2009). *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach*. Science of Computer Programming, 74(7), 470-495.

[26] **Koschke, R.** (2007). *Identifying and removing software clones. In Proceedings of the 13th European Conference on Software Maintenance and Reengineering* (CSMR '07) (pp. 107-108). IEEE.

[27] **Singhal, A.** (2001). *Modern information retrieval: A brief overview*. IEEE Data Eng. Bull., 24(4), 35-43.

[28] **Jones, K. S.** (1972). *A statistical interpretation of term specificity and its application in retrieval.* Journal of documentation.

[29] **Ramos, J.** (2003). *Using tf-idf to determine word relevance in document queries. In Proceedings of the first instructional conference on machine learning* (Vol. 242, pp. 133-142).

[30] **Huang, A.** (2008). *Similarity Measures for Text Document Clustering*. In Proceedings of the New Zealand Computer Science Research Student Conference (NZCSRSC2008), Christchurch, New Zealand.

[31] **Turney, P. D., & Pantel, P.** (2010). *From frequency to meaning: Vector space models of semantics.* Journal of artificial intelligence research, 37, 141-188.

[32] **Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., & Harshman, R.** (1990). *Indexing by latent semantic analysis.* Journal of the American society for information science, 41(6), 391-407.

[33] **F. Almatrooshi, S. Alhammadi, S. A. Salloum, I. Akour, and K. Shaalan,** "*A Recommendation System for Diabetes Detection and Treatment,*" *Proceedings of the 2020 IEEE International Conference on Communications, Computing, Cybersecurity, and Informatics, CCCI 2020*, Nov. 2020, doi: 10.1109/CCCI49893.2020.9256676.

[34] **"Cosine Similarity – LearnDataSci." https://www.learndatasci.com/glossary/cosine-similarity/ (accessed Jul. 05, 2023).**

[35] **Wang, S., Liu, P., Tan, L.** (2018). "*Toward Better Understanding of Deep Learning Based Code Similarity Detection: An Empirical Study*," In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18). ACM, New York, NY, USA, 328-329.

[36] **Ragkhitwetsagul, C., Krinke, J., Paixao, M., Bianco, G., Oliveto, R.** (2019). "*Textual Analysis for Code Clone Detection: Potentials and Limitations,*" In Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS '19). IEEE Press, Piscataway, NJ, USA, 226–235.

[36] **Yuan Y, Chen W, Yang Y, Wang Z.** In defense of the triplet loss again: Learning robust person re-identification with fast approximated triplet loss and label distillation. IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops. 2020 Jun 1;2020-June:1454–63.

[37] **Schroff F, Kalenichenko D, Philbin J**. FaceNet: A unified embedding for face recognition and clustering. Proceedings of the IEEE Computer Society

Conference on Computer Vision and Pattern Recognition. 2015 Oct 14;07-12-June-2015:815–23.

[38] **Yu J, Hu CH, Jing XY, Feng YJ.** Deep metric learning with dynamic margin hard sampling loss for face verification. Signal Image Video Process. 2020 Jun 1;14(4):791–8.

[38] **Yu J, Hu CH, Jing XY, Feng YJ.** Deep metric learning with dynamic margin hard sampling loss for face verification. Signal Image Video Process. 2020 Jun 1;14(4):791–8.

Url-1 *https://www.learndatasci.com/glossary/cosine-similarity,* erişim tarihi 05.07.2023.

Url-2 Plagiarism Detection [Internet]. [29.10.2023]. Erişim adresi: *https://theory.stanford.edu/~aiken/moss/.*