T.C.
BURSA TEKNİK ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

PYTHON SÖZ DİZİMİ HATALARININ DERİN ÖĞRENME YÖNTEMLERİ İLE ONARILMASI

YÜKSEK LİSANS TEZİ

Sena DİKİCİ

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Tezli Yüksek Lisans Programı

OCAK 2024

T.C.
BURSA TEKNİK ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

PYTHON SÖZ DİZİMİ HATALARININ DERİN ÖĞRENME YÖNTEMLERİ İLE ONARILMASI

YÜKSEK LİSANS TEZİ

Sena DİKİCİ
(21376482008)

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Tezli Yüksek Lisans Programı

Tez Danışmanı: Prof. Dr. Turgay Tugay BİLGİN

OCAK 2024

20.04.2016 tarihli Resmi Gazete'de yayımlanan Lisansüstü Eğitim ve Öğretim Yönetmeliğinin 9/2 ve 22/2 maddeleri gereğince; Bu Lisansüstü teze, Bursa Teknik Üniversitesi'nin abonesi olduğu intihal yazılım programı kullanılarak Lisansüstü Eğitim Enstitüsü'nün belirlemiş olduğu ölçütlere uygun rapor alınmıştır.

İNTİHAL BEYANI

Bu tezde görsel, işitsel ve yazılı biçimde sunulan tüm bilgi ve sonuçların akademik ve etik kurallara uyularak tarafımdan elde edildiğini, tez içinde yer alan ancak bu çalışmaya özgü olmayan tüm sonuç ve bilgileri tezde kaynak göstererek belgelediğimi, aksinin ortaya çıkması durumunda her türlü yasal sonucu kabul ettiğimi beyan ederim.

Öğrencinin Adı Soyadı: Sena DİKİCİ

İmzası:

ÖNSÖZ

Akademik yolculuğum boyunca bana destek olan Bursa Teknik Üniversitesi Bilgisayar Mühendisliği bölümü değerli hocalarına, danışmanıma ve meslektaşlarıma derin minnettarlığımı ifade etmek isterim. Danışmanım Prof. Dr. Turgay Tugay BİLGİN'e özellikle teşekkür etmek isterim. Kendisinin bilimsel rehberliğinin yanı sıra, bu zorlu akademik yolculuğum sırasında sağladığı mental ve duygusal destek paha biçilemezdir. Sabrı, anlayışı ve teşvik edici sözleri, karşılaştığım zorluklar karşısında motivasyonumu korumamda ve bu tezi başarıyla tamamlamamda büyük bir rol oynamıştır. Akademik danışmanlığının ötesinde, kişisel gelişimim üzerindeki olumlu etkisi için kendisine minnettarlığımı sunarım. Ayrıca bu süreçte yanımda olan aileme ve arkadaşlarıma teşekkürü borç bilirim.

Bu çalışma, derin öğrenme yöntemleriyle otomatik program onarımı üzerine odaklanmaktadır. Bu tezin akademik camia için yeni perspektifler sunmasını ve alanımızda daha fazla araştırma yapılmasına ilham kaynağı olmasını dilerim.

Saygılarımla,

Ocak 2024 Sena DİKİCİ
(Araştırma Görevlisi)

İÇİNDEKİLER

Sayfa

ÖNSÖZ ..... **vi**
İÇİNDEKİLER ..... **vii**
KISALTMALAR ..... **ix**
SEMBOLLER ..... **x**
ÇİZELGE LİSTESİ ..... **xi**
ŞEKİL LİSTESİ ..... **xii**
ÖZET ..... **xiv**
SUMMARY ..... **xv**
1. GİRİŞ **1**
1.1 Tezin Motivasyonu 2
1.2 Tezin Amacı 3
1.3 Tezin Katkıları ve Ana Hatları 4
2. LİTERATÜR TARAMASI **8**
2.1 Şablon Tabanlı Yaklaşımların Kullanıldığı Çalışmalar 8
2.2 Makine Öğrenmesi Yöntemlerinin Kullanıldığı Çalışmalar 14
2.2 Derin Öğrenme Yöntemlerinin Kullanıldığı Çalışmalar 18
3. PROGRAMLAMA DİLLERİNDE SÖZ DİZİMİ **28**
3.1 Genel Bakış 29
3.1.1 Python programlama dilindeki söz dizimi hataları 31
3.1.2 Söz dizimi hatalarının temel sebepleri 40
3.2 Temel Söz Dizimi Hataları 41
3.2.1 Python'da söz dizimi 43
3.3 Girinti Hataları 44
3.4 Parantez ve İki Nokta Hataları 44
3.5 İsimlendirme Hataları 45
4. DERİN ÖĞRENME **47**
4.1 Yapay Sinir Ağlarının Öğrenmesi 47
4.1.1 İleri yayılım ve geri yayılım 50
4.1.2 Optimizasyon algoritmaları 53
4.1.3 Aktivasyon fonksiyonları 58
4.1.3.1 Sigmoid fonksiyonu 59
4.1.3.2 Düzeltilmiş lineer birim (ReLU) fonksiyonu 60
4.1.3.3 Hiperbolik tanjant fonksiyonu 61
4.1.3.4 Softmax fonksiyonu 62
4.2 Derin Öğrenme Mimarileri 62
4.2.1 Zaman serileri ve sıralı veri analizi için derin öğrenme mimarileri 63
4.2.1.1 Tekrarlayan sinir ağları (Recurrent neural network- RNN) 63
4.2.1.2 Uzun kısa süreli bellek (Long short term memory- LSTM) 65
4.2.1.3 Çift yönlü uzun kısa süreli bellek (Bidirectional LSTM) 67
4.2.1.4 Zaman serileri ve sıralı veri analizi için derin öğrenme mimarilerinin karşılaştırılması 69

4.2.2 Büyük dil modelleri (Large language models- LLM) 71
4.2.3 Büyük dil modelleri ile uzun kısa süreli bellek yöntemlerinin karşılaştırılması 72
4.3 Performans Metrikleri 73
4.3.1 Doğruluk (Accuracy) 75
4.3.2 Kesinlik (Precision) 75
4.3.3 Duyarlılık (Recall) 75
4.3.4 F1 Skoru 75
5. BULGULAR VE YORUMLAR 76
5.1 Veri Seti 78
5.2 Hata Türünün Tespit Edilmesi 85
5.3 Hatalı Tokenin Tespit Edilmesi 96
5.4 Hatalı Tokenin Onarılması 97
5.5 Test Sonuçları 110
6. SONUÇLAR VE ÖNERİLER 121
KAYNAKLAR 123
EKLER 130
ÖZGEÇMİŞ 131

KISALTMALAR

Adam **:** Uyarlamalı Moment Tahmini – Adaptive Moment Estimation

APR **:** Otomatik Program Onarımı – Automatic Program Repair

AST **:** Soyut Söz Dizimi Ağaçları – Abstract Syntax Tree

BiLSTM **:** Çift Yönlü Uzun Kısa Süreli Bellek – Bidirectional Long Short-Term Memory

FN **:** Yanlış Negatif – False Negative

FP **:** Yanlış Pozitif – False Positive

GPT **:** Üretken Ön İşlemeli Dönüştürücü – Generative Pre-trained Transformer

GRU **:** Kapılı Yinelemeli Birim – Gated Recurrent Unit

ID **:** Benzersiz Tanımlayıcı – Identification Number

IDE : Tümleşik Geliştirme Ortamı – Integrated Development Environment

LLM **:** Büyük Dil Modeli – Large Language Model

LSTM **:** Uzun Kısa Süreli Bellek – Long Short-Term Memory

NLP **:** Doğal Dil İşleme – Natural Language Processing

NMT : Sinirsel Makine Çevirisi – Neural Machine Translation

ReLU **:** Düzeltilmiş Doğrusal Birim – Rectified Linear Unit

RNN **:** Tekrarlayan Sinir Ağları – Recurrent Neural Network

RMSprop **:** Kök Ortalama Kare Yayılımı – Root Mean Square Propagation

SGD **:** Stokastik Gradyan İnişi – Stochastic Gradient Descent

Tanh **:** Hiperbolik Tanjant – Hyperbolic Tangent

TN **:** Doğru Negatif – True Negative

TP **:** Doğru Pozitif – True Positive

SEMBOLLER

$\boldsymbol{b}$	**:** Bias katsayısı
$\boldsymbol{e}$	**:** Euler sayısı
$\boldsymbol{\varepsilon}$	**:** Hata değeri
$\boldsymbol{\mathcal{L}}$	**:** Kayıp fonksiyonu
$\boldsymbol{\eta}$	**:** Öğrenme oranı
$\boldsymbol{\theta}$	**:** Model parametresi
$\boldsymbol{\sigma}$	**:** Sigma parametresi
$\boldsymbol{\lambda}$	**:** Keskinlik parametresi

ÇİZELGE LİSTESİ

Sayfa

Çizelge 5.1 : Test veri seti için hata mesajına göre hata dağılımları **81**
Çizelge 5.2 : Hata türü modeli için yapay sinir ağı yöntemlerinin karşılaştırılması.. **86**
Çizelge 5.3 : Hata türü modeli için katman sayılarının karşılaştırılması..... **88**
Çizelge 5.4 : Hata türü modeli için aktivasyon fonksiyonlarının karşılaştırılması.... **89**
Çizelge 5.5 : Hata türü modeli için farklı küme boyutlarının (batch size) karşılaştırılması. **90**
Çizelge 5.6 : Hata türü modeli için farklı döngü sayılarının (epoch) karşılaştırılması **91**
Çizelge 5.7 : Token tahmin modeli için yapay sinir ağı yöntemlerinin karşılaştırılması **98**
Çizelge 5.8 : Token tahmin modeli için katman sayılarının karşılaştırılması..... **100**
Çizelge 5.9 : Token tahmin modeli için farklı nöron sayılarının karşılaştırılması .. **101**
Çizelge 5.10 : Token tahmin modeli için aktivasyon fonksiyonlarının karşılaştırılması **102**
Çizelge 5.11 : Token tahmin modeli için optimizasyon algoritmalarının karşılaştırılması **103**
Çizelge 5.12 : Token tahmin modeli için farklı küme boyutlarının (batch size) karşılaştırılması. **104**
Çizelge 5.13 : Token tahmin modeli için farklı döngü sayılarının (epoch) karşılaştırılması. **105**
Çizelge 5.14 : Geliştirilen yöntem ile başarılı bir şekilde onarılan Python kodlarının dağılımı **111**
Çizelge 5.15 : PyNar Sohbet Robotu ile başarılı bir şekilde onarılan Python kodlarının dağılımı **116**
Çizelge 5.16 : Girinti hataları için iki sistemin (geliştirilen yöntem ve PyNar Sohbet Robotu) karşılaştırılması. **119**

ŞEKİL LİSTESİ

Sayfa

Şekil 1.1 : Sisteme ait akış diyagramı **5**
Şekil 3.1 : Tasarım laboratuvarı dersi öğrenci hataları dağılımı **30**
Şekil 3.2 : Yazılım laboratuvarı dersi öğrenci hataları dağılımı **31**
Şekil 3.3 : Girintileme hatası içeren örnek Python kodu **32**
Şekil 3.4 : Python yorumlayıcısında girintileme hatası çıktısı örneği **32**
Şekil 3.5 : Söz dizimi hatası içeren örnek Python kodu **33**
Şekil 3.6 : Python yorumlayıcısında söz dizimi hatası çıktısı örneği **33**
Şekil 3.7 : İsimlendirme hatası içeren örnek Python kodu **34**
Şekil 3.8 : Python yorumlayıcısında isimlendirme hatası çıktısı örneği **34**
Şekil 3.9 : Tip hatası içeren örnek Python kodu **35**
Şekil 3.10 : Python yorumlayıcısında tip hatası çıktısı örneği **35**
Şekil 3.11 : Değer hatası içeren örnek Python kodu **36**
Şekil 3.12 : Python yorumlayıcısında değer hatası çıktısı örneği **36**
Şekil 3.13 : Özellik hatası içeren örnek Python kodu **37**
Şekil 3.14 : Python yorumlayıcısında özellik hatası çıktısı örneği **37**
Şekil 3.15 : Anahtar hatası içeren örnek Python kodu **38**
Şekil 3.16 : Python yorumlayıcısında anahtar hatası çıktısı örneği **38**
Şekil 3.17 : İçe aktarma hatası içeren örnek Python kodu **39**
Şekil 3.18 : Python yorumlayıcısında içe aktarma hatası çıktısı örneği **39**
Şekil 3.19 : Modül bulunamadı hatası içeren örnek Python kodu **39**
Şekil 3.20 : Python yorumlayıcısında modül bulunamadı hatası çıktısı örneği **40**
Şekil 4.1 : Nöron yapısı **48**
Şekil 4.2 : (a) Yapay nöron (b) Çok katmanlı yapay sinir ağı mimarisi **49**
Şekil 4.3 : Örnek bir ileri beslemeli sinir ağı **50**
Şekil 4.4 : Tek gizli katmanlı geri yayılımlı sinir ağı **53**
Şekil 4.5 : Gradyan inişi algoritması **55**
Şekil 4.6 : SGD dalgalanması **56**
Şekil 4.7 : Momentum etkisinin sgd optimizasyonu üzerindeki karşılaştırması **56**
Şekil 4.8 : Yapay nöron ile aktivasyon fonksiyonu ilişkisinin temsili gösterimi **59**
Şekil 4.9 : Sigmoid fonksiyonu **60**
Şekil 4.10 : ReLU fonksiyonu **61**
Şekil 4.11 : Hiperbolik tanjant fonksiyonu **62**
Şekil 4.12 : Tekrarlayan sinir ağları **65**
Şekil 4.13 : (a) Orijinal bir LSTM birim mimarisi: basit bir LSTM hücresi; (b) Unutma kapısı olan bir LSTM hücresi; (c) Unutma kapısı ile modern bir LSTM hücresinin gösterimi **67**
Şekil 4.14 : BiLSTM mimarisi **69**
Şekil 4.15 : Karışıklık matrisi gösterimi **75**
Şekil 5.1 : Geliştirilen otomatik kod onarımı sistemi **77**
Şekil 5.2 : Söz dizimi açısından doğru yazılmış Python kodu örneği **79**

Şekil 5.3 : Mutasyon testi ile söz dizimi açısından hatalı hale (yanlış yazım) getirilmiş kod örneği **79**
Şekil 5.4 : Mutasyon testi ile söz dizimi açısından hatalı hale (eksik yazım) getirilmiş kod örneği. **80**
Şekil 5.5 : Mutasyon testi ile söz dizimi açısından hatalı hale (fazla yazım) getirilmiş kod örneği. **80**
Şekil 5.6 : Örnek ‘re’ modulü kullanımı..... **83**
Şekil 5.7 : Python Kodlarının ID'lere Dönüştürülmesi..... **84**
Şekil 5.8 : Döngü sayısı 50 alınarak gerçekleştirilen modelin eğitim sürecinde doğruluk ve kayıp değerleri grafiği..... **91**
Şekil 5.9 : Döngü sayısı 150 alınarak gerçekleştirilen modelin eğitim sürecinde doğruluk ve kayıp değerleri grafiği..... **92**
Şekil 5.10 : Döngü sayısı 250 alınarak gerçekleştirilen modelin eğitim sürecinde doğruluk ve kayıp değerleri grafiği..... **93**
Şekil 5.11 : Döngü sayısı 500 alınarak gerçekleştirilen modelin eğitim sürecinde doğruluk ve kayıp değerleri grafiği..... **93**
Şekil 5.12 : Hata türü modeli katman yapısı..... **95**
Şekil 5.13 : Token tahmin modeli eğitim sürecine ait F1 skoru grafiği. **106**
Şekil 5.14 : Token tahmin modeli eğitim sürecine ait doğruluk ve kayıp grafiği. .. **107**
Şekil 5.15 : Token tahmin modeli katman yapısı **108**
Şekil 5.16 : Eksik yazımda “unmatched x” hata mesajı oluşturan Python kodu ve onarılmış hali..... **111**
Şekil 5.17 : Eksik yazımda “invalid syntax” hata mesajı oluşturan Python kodu ve onarılmış hali..... **112**
Şekil 5.18 : Eksik yazımda “unexpected EOF while parsing” hata mesajı oluşturan Python kodu ve onarılmış hali..... **112**
Şekil 5.19 : Fazla yazımda “invalid syntax” hata mesajı oluşturan Python kodu ve onarılmış hali..... **113**
Şekil 5.20 : Fazla yazımda “unmatched x” hata mesajı oluşturan Python kodu ve onarılmış hali..... **114**
Şekil 5.21 : Yanlış yazımda “invalid syntax” hata mesajı oluşturan Python kodu ve onarılmış hali..... **115**
Şekil 5.22 : Yanlış yazımda “closing parenthesis 'x' does not match opening parenthesis 'y'” hata mesajı oluşturan Python kodu ve onarılmış hali ... **115**
Şekil 5.23 : PyNar Sohbet Robotunun başarılı ve başarısız onarım için örnek mesajları **117**

PYTHON SÖZ DİZİMİ HATALARININ DERİN ÖĞRENME YÖNTEMLERİ İLE ONARILMASI

ÖZET

Otomatik program onarımı (APR), yazılım geliştirme süreçlerinde karşılaşılan hataları tespit etmek ve onarmak için otomatik teknikler kullanan bir araştırma alanıdır. Bu alanda yapılan çalışmalar, hata düzeltme sürecini hızlandırmayı, yazılım kalitesini artırmayı ve geliştiricilerin üzerindeki yükü azaltmayı hedefler. Bu alandaki son yenilikler, derin öğrenme ve makine öğrenimi tekniklerinin yanı sıra geleneksel algoritmik yaklaşımları içermekte ve kod analizi, hata türü tahmini, hatalı kod segmentlerinin lokalizasyonu ve onarım önerileri sunma gibi aşamaları otomatize etmekte büyük ilerlemeler kaydetmektedir.

Bu tez çalışması, Python programlama dilinde yazılmış kodların otomatik olarak onarılmasını hedefleyen ve yapay zeka teknolojilerini kullanan bir sistem üzerine odaklanmaktadır. Çalışma, BiLSTM ve LSTM ağları kullanılarak geliştirilen özel modeller ile hatalı kodların tespit edilip onarılması sürecini içermektedir. Çalışmanın ilk aşamasında, farklı hata türlerine sahip hatalı Python kodları üzerinde eğitim gerçekleştirilmiştir. BiLSTM hata türü tahmin modeli, %98'lik bir doğruluk değeri elde ederek hata türlerini etkin bir şekilde tespit edebildiğini kanıtlamıştır. Python dilinde yazılmış ve doğru biçimde işlev gösteren kod örnekleri üzerinden, bu kodların tokenize edilerek temel yapı taşlarına ayrılması ve her bir tokenin benzersiz tanımlayıcılar ile eşleştirilmesi işlemi gerçekleştirilmiştir. Bu işlem sonucunda oluşturulan veri seti, LSTM tabanlı bir token tahmin modelinin eğitimi için kullanılmıştır. Model, eğitim sonucunda %98,44'lük bir doğruluk oranı ve %98,91'lik F1 skoruna ulaşarak yüksek düzeyde başarı sergilemiştir.

Sistemin performansı, mutasyon testiyle oluşturulmuş 925 adet hatalı kod örneği üzerinde deneyimlenmiş ve bu kodların 794'ü (%85,84) başarılı bir şekilde onarılmıştır. Bu sonuçlar, sistemin hatalı Python kodlarını tespit etme ve onarma konusunda oldukça yetkin olduğunu göstermektedir. Şablon tabanlı APR aracı olan PyNar Sohbet Robotu ile karşılaştırıldığında tez çalışmasının onarım sistemi, PyNar'ın %53,3'lük başarısına kıyasla %85,84 gibi daha yüksek bir başarı oranına ulaşmıştır. Bu karşılaştırma, geliştirilen otomatik onarım sisteminin etkinliğini ve yapay zeka destekli çözüm yaklaşımlarının bu alanda ne kadar başarılı olabileceğini gözler önüne sermiştir.

Bu akademik çalışma, otomatik program onarımı konusunda önemli bir ilerlemeyi işaret ederek bu alandaki ileriye dönük çalışmalar için güçlü bir referans sağlamaktadır. Elde edilen yüksek doğruluk oranları ve onarım başarısı, APR araçlarının yazılım geliştirme alanında önemli bir etki yaratabileceğinin göstergesidir. Bu tür sistemlerin gelişimi ve iyileştirilmesi kod kalitesini artırma ve geliştirme süreçlerinin verimliliğini iyileştirme yolunda umut verici yenilikler sunmaktadır.

Anahtar kelimeler: Otomatik Program Onarımı, Derin Öğrenme, Uzun Kısa Süreli Bellek, Çift Yönlü Uzun Kısa Süreli Bellek

FIXING PYTHON SYNTAX ERRORS WITH DEEP LEARNING METHODS

SUMMARY

Automatic Program Repair (APR) is a field of research that employs automated techniques to detect and fix errors encountered in software development processes. Studies in this area aim to accelerate the error correction process, enhance software quality and reduce the burden on developers. Recent innovations in this field encompass deep learning and machine learning techniques, alongside traditional algorithmic approaches, making significant strides in automating stages such as code analysis, error type prediction, localization of faulty code segments and offering repair suggestions.

This disputation focuses on a system that targets the automatic repair of code written in the Python programming language using artificial intelligence technologies. The work involves the development of specialized models using BiLSTM and LSTM networks for the detection and repair of faulty code. In the initial phase, training was conducted on faulty Python codes with various error types. The BiLSTM error type prediction model achieved a 98% accuracy rate, effectively demonstrating its capability to identify error types. Python code samples that function correctly were processed to tokenize and break them down into basic building blocks, with each token matched with unique identifiers. This process resulted in a dataset used for training a LSTM-based token prediction model. Post-training, the model exhibited high success, achieving a 98.44% accuracy rate and a 98.91% F1 score.

The system's performance was tested on 925 faulty code samples created through mutation testing, successfully repairing 794 of them (85.84%). These results demonstrate the system's proficiency in detecting and repairing faulty Python code. When compared with PyNar, a template-based APR tool chatbot, the thesis's repair system reached a higher success rate of 85.84% compared to PyNar's 53.3%. This comparison highlights the effectiveness of the developed automatic repair system and underscores the potential success of AI-supported solution approaches in this field.

This research marks a significant advancement in automatic program repair, providing a robust reference for future research in this area. The high accuracy and repair success rates indicate that APR tools could have a substantial impact in the field of software development. The continued development and improvement of such systems offer promising innovations for enhancing code quality and increasing the efficiency of development processes.

Keywords: Automatic Program Repair, Deep Learning, Long Short-Term Memory, Bidirectional Long Short-Term Memory

1. GİRİŞ

Program yazmaya yeni başlayanlar için öğrenme süreci hem heyecan verici hem de zorlu olabilmektedir. Başlangıçta temel programlama kavramlarını anlamak önemlidir. Bu kavramlar arasında değişkenler, döngüler, koşullu ifadeler ve fonksiyonlar gibi temeller bulunmaktadır. Ancak yeni başlayanlar sıklıkla söz dizimi hatalarıyla karşılaşabilmektedir (Denny ve diğ, 2012). Söz dizimi, bir programlama dilinde kodların nasıl yazılacağını ve düzenleneceğini belirleyen kurallar ve yapıları ifade etmektedir. Her programlama dilinin kendine özgü bir söz dizimi vardır. Söz dizimi dilin nasıl "konuşulduğunu" belirlemektedir. Yeni başlayanların öğrenme sürecinde programlama dillerinin söz dizimlerine yabancı olmaları nedeniyle söz dizimi hataları yapmaları yaygındır. Bu hatalar, dilin kurallarına ve yapılarına alışma sürecinin doğal bir parçasıdır ve programlama öğreniminin önemli bir aşamasını temsil etmektedir. Yapılan hataları tanımak ve nasıl çözüleceğini öğrenmek öğrenme sürecinin önemli bir parçasıdır. Ancak programcının zaman kaybetmesine de sebep olan bir süreçtir (Denny ve diğ, 2012). Modern geliştirme ortamları, bu tür hataları tespit etmekte ve kullanıcılara geri bildirim sağlamaktadır. Söz dizimi hataları ile ilgili hata mesajı üreterek hatanın kaynağını ve düzeltilmesi gereken yeri işaret etmektedir. Bu süreç, zaman zaman karmaşık ve zorlu olabilmektedir. Çünkü hata mesajları daima açık ve anlaşılır nitelikte olmamaktadır. Bir öğrenci, programlama dilinin söz dizimi kurallarını tam olarak kavrayamadığında hataların doğru tespitinde zorlanabilmektedir. Hata mesajlarının belirsiz ya da yanıltıcı olması öğrencinin sorunun asıl kaynağını doğru bir şekilde anlamasını güçleştirebilmektedir. Hatanın doğru olarak tespit edilememesi öğrencinin çözüm yöntemlerini bulmasını da zorlaştırabilmektedir. Bu durum, özellikle programlama kavramlarına veya dilin söz dizimi yapısına henüz tam anlamıyla vakıf olmayan öğrenciler için daha büyük bir sorun teşkil edebilmektedir. Öğrenci, hataların nasıl düzeltileceğini bilemediğinde bu durum öğrenme sürecini yavaşlatmakta ve motivasyonunu olumsuz yönde etkileyebilmektedir (Denny ve diğ, 2012). Ancak bu tür zorlukların üstesinden gelmek çeşitli destek araçlarının kullanılmasıyla mümkün olabilmektedir.

Programlama dillerindeki söz dizimi hatalarının düzeltilmesi otomatik program onarımı (APR) alanındaki çalışmaların önemli bir odak noktasını oluşturmaktadır. APR, bu tür hataları otomatik olarak tespit etmek ve düzeltmek için gelişmiş algoritmalar ve yapay zeka tekniklerini kullanmaktadır. Bu süreç, programcıların hata düzeltme sürecini manuel olarak yapmak yerine daha yaratıcı ve üretken görevlere odaklanmalarını sağlamakta böylece yazılım geliştirme süreçlerinin verimliliğini artırmaktadır.

1.1 Tezin Motivasyonu

Otomatik program onarımı alanındaki temel motivasyon yazılım geliştirme sürecinde zaman, maliyet ve çaba açısından önemli avantajlar sağlamaktır. Yazılım hatalarını manuel olarak tespit etmek ve düzeltmek genellikle zaman alıcı ve maliyetli bir süreçtir. APR, bu süreci otomatize ederek hata düzeltme süresini kısaltarak mühendislik kaynaklarını daha verimli kullanmayı sağlamaktadır. Bu, özellikle büyük ve karmaşık yazılım projelerinde maliyet tasarrufu ve verimlilik açısından büyük önem taşımaktadır. Hataların daha hızlı ve güvenilir bir şekilde tespit edilmesine ve düzeltilmesine olanak tanımaktadır. Bu durum genel yazılım kalitesini artırarak daha az hata içeren ve daha güvenilir yazılım ürünleri oluşturulmasına yardımcı olmaktadır. APR, geliştirme süreçlerini daha etkili hale getirir ve geliştiricilerin zamanını daha yaratıcı ve yenilikçi görevlere ayırmalarına olanak tanımaktadır. Geliştiricilerin verimliliğini artırmakta ve ürün geliştirme sürecinde daha hızlı ilerlemelerini sağlamaktadır. APR, özellikle yeni başlayanlar için, yaygın hata modellerini ve bu hataların nasıl düzeltileceğini öğrenme konusunda rehberlik etmektedir. Bu, geliştiricilerin hızla yeteneklerini geliştirmelerine ve daha sağlam yazılım yazma becerileri kazanmalarına yardımcı olmaktadır. Modern yazılım sistemleri karmaşık ve çok katmanlıdır. APR, bu karmaşıklığı yönetmekte ve özellikle zor hataları tespit ederek düzeltmekte yardımcı olmaktadır. APR, yapay zeka ve makine öğrenimi gibi ileri düzey teknolojileri kullanarak çalışmalarını yürütmektedir. Bu, yazılım mühendisliği alanında sürekli yenilik ve teknolojik gelişim sürecine uyum sağlamayı kolaylaştırmaktadır. APR, hata düzeltme süreçlerini standartlaştırır ve otomatize etmektedir. Bu da sürecin daha tutarlı ve öngörülebilir olmasını sağlamaktadır. Tezin motivasyonu, yazılım geliştirme sürecini daha verimli, etkili ve hatasız hale

getirmektir. APR teknolojileri, yazılım mühendisliği uygulamalarını dönüştürme potansiyeline sahiptir ve sürekli olarak araştırma ve geliştirmenin odak noktasıdır.

1.2 Tezin Amacı

Bu tez çalışması, yeni başlayanlar için Python programlama dilinde Uzun Kısa Süreli Bellek (Long Short-Term Memory, LSTM) modelini kullanarak otomatik program onarımının gerçekleştirilmesi üzerine odaklanmaktadır. Bu araştırmanın temel amacı, LSTM tabanlı derin öğrenme modellerinin söz dizimi hataları ve diğer yaygın programlama hatalarını nasıl tespit edebileceğini ve düzeltebileceğini keşfetmektir. LSTM, derin öğrenme alanında, özellikle zaman serileri verilerinin ve sıralı verilerin işlenmesinde etkili olan bir yapay sinir ağı mimarisidir. Bu mimari, programlama hatalarını tespit etmede ve onarmada, özellikle de yeni başlayan programcıların sıkça karşılaştığı söz dizimi hataları gibi spesifik sorunlar üzerinde potansiyel olarak büyük başarılar sağlamaktadır.

Tez çalışması, LSTM'nin programlama hatalarını algılayarak düzeltilmesi gereken alanları belirlemek için nasıl eğitilebileceğini derinlemesine incelemektedir. Bu süreç, programlama hataları üzerinde bir veri seti kullanılarak gerçekleştirilmektedir. Ayrıca, LSTM modelinin hataları düzeltmek için önerilen çözümleri nasıl üretebileceğine ve bu çözümlerin etkinliğine de odaklanılmaktadır. Bu çalışma, otomatik program onarımının programlama öğrenim sürecindeki bireyler için nasıl bir fayda sağlayabileceğini ve öğrenme sürecini nasıl hızlandırabileceğini de tartışmaktadır.

Son olarak, LSTM tabanlı otomatik program onarımının programlama eğitiminde ve yazılım geliştirme süreçlerinde nasıl uygulanabileceğini, ayrıca bu yaklaşımın potansiyel avantajlarını ve sınırlamalarını değerlendirmeyi amaçlamaktadır. Bu çalışmanın sonucunda elde edilen bulgular hem akademik hem de endüstriyel çevrelerde programlama eğitimi ve yazılım geliştirme pratiklerine önemli katkılarda bulunabilir. Bu tez için geliştirilen sistemin, şablon tabanlı yöntem ile otomatik program onarımı sistemine sahip PyNar'ın, tamamen Türkçe arayüze sahip ilk Python editörü, (PyNar-Editor, 2018) hata düzeltme performansına katkı sağlaması amaçlanmıştır. Bu katkılar, programlama dillerinin daha verimli ve etkili bir şekilde öğrenilmesini sağlamaktadır. Ek olarak yazılım geliştirme süreçlerinde verimliliği ve doğruluğu artırmak için kullanılabilir.

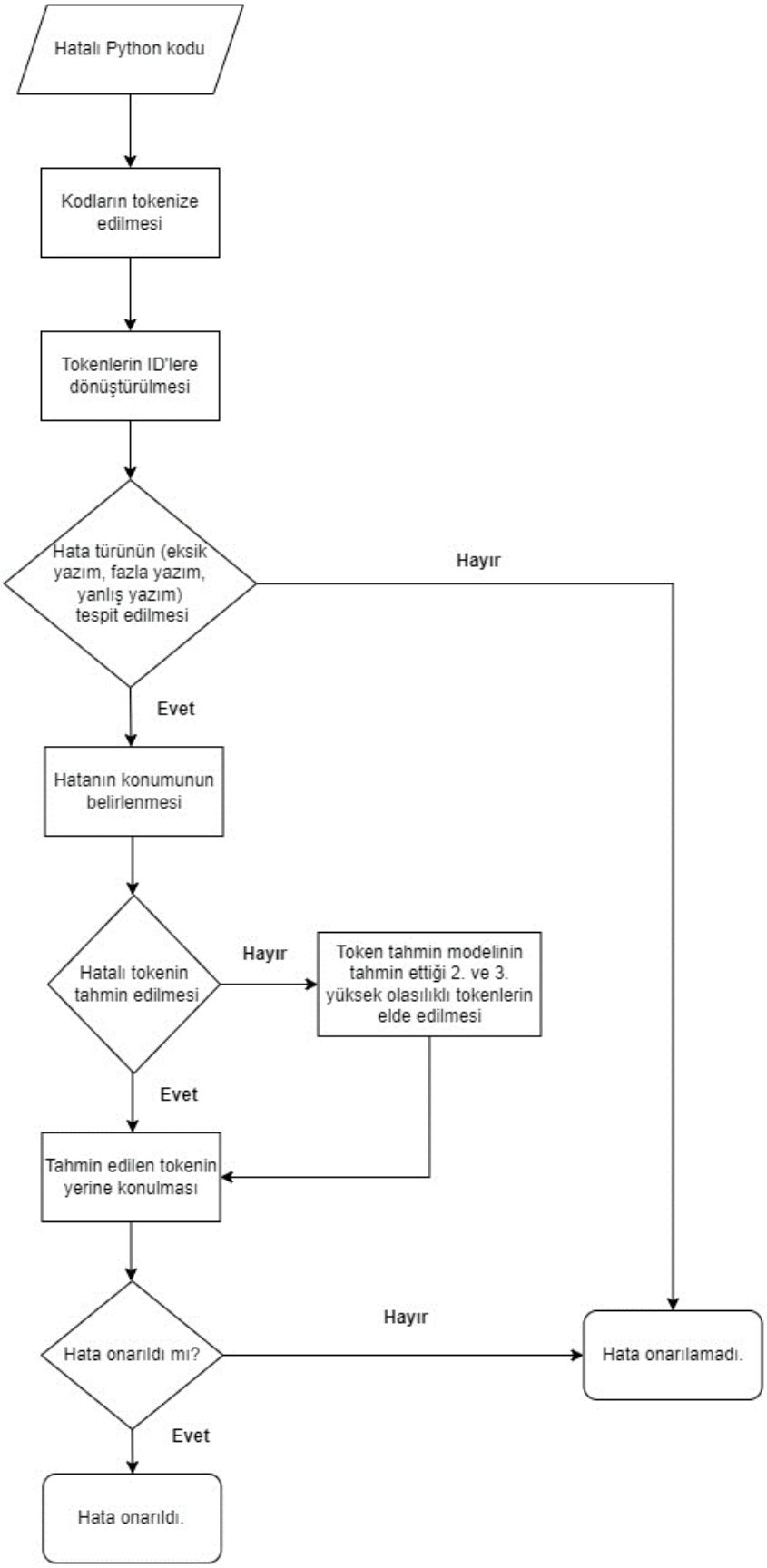
1.3 Tezin Katkıları ve Ana Hatları

Bu tez çalışması, Python programlama dili kullanılarak yazılmış kodların otomatik olarak onarılmasına odaklanan yenilikçi bir sistem üzerine odaklanmaktadır. Bu sistem, programlama hatalarını tespit etme ve düzeltme yeteneğine sahip olup kod kalitesini ve verimliliğini artırmayı amaçlamaktadır.

İlk aşamada hata türü tahmini için özel bir LSTM modeli geliştirilmiştir. Bu modelin eğitiminde, farklı hata türlerine sahip 6000 hatalı Python kodu kullanılmıştır. Bu kodlar üç ana kategoriye ayrılmıştır: eksik token içeren 2000 kod, fazladan token içeren 2000 kod ve yanlış yazılmış token içeren 2000 kod. Hata türü tahmin modeli, bu çeşitlendirilmiş veri seti ile eğitilmiştir. Model, %98 doğruluk değerine ulaşmıştır. Bu değer, modelin farklı hata türlerini etkin bir şekilde tespit edebildiğini göstermektedir.

Çalışmanın temel aşamasında, Python dilinde yazılmış ve doğru biçimde işlev gösteren 2000 adet kod örneği seçilmiştir. Bu kod örnekleri, öncelikle tokenize edilerek temel yapı taşlarına ayrılmıştır. Ardından her bir token, benzersiz tanımlayıcılar (ID'ler) ile eşleştirilerek bir veri seti oluşturulmuştur. Oluşturulan bu veri seti, LSTM tabanlı bir token tahmin modelinin eğitiminde kullanılmıştır. Model eğitimi için veri setinin %80'i eğitim seti olarak ayrılmış geri kalan %20'lik kısım ise modelin performansını değerlendirmek üzere test seti olarak kullanılmıştır. Eğitim sonucunda elde edilen model, %98,44'lük bir doğruluk oranına ve %98,91'lik F1 skoruna ulaşmıştır. Bu da modelin yüksek düzeyde başarılı olduğunu göstermektedir.

Bu gelişmiş sistem, Python kodlarının otomatik olarak onarılmasında önemli bir adımı temsil etmektedir. Kod yazım hatalarını hızlı ve etkili bir şekilde tespit edip düzeltebilen bu sistem, programcıların zaman ve çaba tasarrufu yapmalarına yardımcı olmaktadır. Sistemin çalışma prensipleri ve iş akışı, Şekil 1.1'de görselleştirilmiş akış diyagramı ile detaylı bir şekilde sunulmuştur. Bu diyagram, sistemin nasıl çalıştığını ve aşamalar arasındaki ilişkileri anlamak için önemli bir araçtır.



Şekil 1.1 : Sisteme ait akış diyagramı.

Sistem, ilk olarak hatalı bir kodu tokenize etmekte ve daha sonra tokenleri ID'lere dönüştürmektedir. Ardından bir BiLSTM modeli kullanarak kodun hatasının eksik yazım, fazla yazım veya yanlış yazımdan kaynaklandığını tespit etmektedir. Tespit edilen hata türüne göre sistem AST (Soyut Söz Dizimi Ağacı - Abstract Syntax Tree) kütüphanesini (Python Software Foundation, 2019) kullanarak hatanın yerini belirlemeye çalışmaktadır. Eğer AST ile konum belirlenemezse, Pyright (Microsoft, 2023) kütüphanesi devreye girerek hatanın satır ve sütun bilgisini tespit etmektedir. Sistem, eksik bir token mevcutsa token tahmin LSTM modeli ile eksik olan tokeni tahmin edip eklemektedir. Fazladan bir token mevcutsa ilgili token silinmektedir. Yanlış yazılmış bir token varsa, doğru token tahmin edilerek yerine yazılmaktadır. Eğer ilk denemede hata çözülemezse sistem token tahmin LSTM modelinin ikinci ve üçüncü olasılıklarını deneyerek hata çözümünü tekrarlamaktadır. Hata çözülemezse, sistem bir uyarı vermektedir. Mutasyon testi ile oluşturulan 900 adet hatalı Python kodu ile yapılan testlerde sistemin başarısı %84,78 elde edilmiştir.

Bu tez çalışması, otomatik program onarımı alanında önemli katkılarda bulunmaktadır. Özellikle, hata tespiti ve çözümü konusunda LSTM modelinin etkinliğini ve farklı hata türlerine adaptasyon yeteneğini ortaya koymaktadır. Ayrıca AST ve Pyright gibi kütüphanelerin kullanımı, hata tespit sürecindeki doğruluğu ve verimliliği artırmaktadır. Bu çalışmanın sonuçları, programlama eğitiminde ve profesyonel yazılım geliştirme süreçlerinde kod hatalarını otomatik olarak tespit etme ve düzeltme konusunda büyük potansiyele sahiptir.

Bu tez çalışmasında geliştirilen otomatik program onarım (APR) sistemi, PyNar için tasarlanmış olan Şablon Tabanlı APR aracı ile entegre edildiğinde program onarımı performansında önemli bir artış beklenmektedir. PyNar'ın Şablon Tabanlı APR aracı, önceden tanımlanmış hata şablonlarına dayanarak hataları tespit edip onarmaktadır. Bu süreçte, genellikle belirli hata türlerine özgü çözüm yollarını uygulamaktadır. Bu şablon tabanlı yaklaşım, sık karşılaşılan bazı söz dizimi hatalarını etkili bir şekilde çözümlemede başarılı olmaktadır. Ancak her hata türü için şablon oluşturmak zaman alıcıdır. Bu sebeple, LSTM tabanlı otomatik program onarım sistemi devreye girmektedir. Bu sistem, özellikle şablon dışı ve daha karmaşık hataları tespit etme ve onarma konusunda etkili olmaktadır. Şablon Tabanlı APR aracı ile LSTM tabanlı sistem birleştirildiğinde her iki sistemin güçlü yönleri bir araya getirilerek daha kapsamlı ve etkili bir program onarım çözümü oluşturulmuş olacaktır. Şablon tabanlı

araç, yaygın ve iyi tanımlanmış hatalara hızlı ve etkin çözümler sunarken; LSTM tabanlı sistem, daha az yaygın veya daha karmaşık hata türlerini ele alarak sistemin genel kapsamını genişletecektir. Bu entegrasyon, her iki sistemin zayıf yönlerini birbirlerinin güçlü yönleriyle dengelemektedir. Böylece genel doğruluk oranını daha da yükseltmektedir. Çalışmada iki sistemin performansı, güçlü ve zayıf yönleri karşılaştırılmıştır. Ayrıca bu birleşik yaklaşım, farklı türdeki hata senaryolarına karşı daha dirençli ve esnek bir otomatik program onarım çözümü sağlamaktadır.

Bu tez çalışmasının geliştirdiği LSTM tabanlı otomatik program onarım sistemi ile PyNar'ın Şablon Tabanlı APR aracının entegrasyonu, programlama hatalarının tespiti ve onarımı alanında önemli bir ilerleme sağlayacaktır. Ek olarak bu alanda yapılacak gelecek çalışmalara zemin hazırlayacaktır. Bu entegrasyon hem eğitim hem de profesyonel yazılım geliştirme süreçlerinde, kod hatalarını daha etkin bir şekilde tespit etme ve düzeltme kapasitesine sahip olacaktır. Bu da programlama dillerinin öğrenilmesini ve yazılım geliştirme süreçlerinin kalitesinin artırılmasını destekleyecektir.

İkinci bölümde, alanla ilgili önceki çalışmalar incelenmiştir. İlk olarak, kural tabanlı yöntemleri kullanan çalışmalar ele alınmıştır. Ardından makine öğrenimi ve derin öğrenme yöntemlerinin kullanıldığı çalışmalar incelenmiştir. Bu değerlendirme, mevcut yaklaşımların avantajlarını ve dezavantajlarını belirleyerek bu tezin metodolojik yapısını şekillendirmektedir.

Üçüncü bölümde, Python programlama dilindeki temel söz dizimi hatalarına genel bir bakış sunulmaktadır. Bu hataların nedenleri ve ortaya çıkış biçimleri detaylandırılmıştır. Temel söz dizimi hataları, girinti hataları, parantez ve iki nokta kullanımı, değişken ve veri türü hataları ile isimlendirme hataları incelenmiştir.

Dördüncü bölümde derin öğrenme mimarilerinin yapısı anlatılmıştır. Derin öğrenme mimarilerinin zaman serileri ve sıralı veri analizindeki uygulamaları anlatılmıştır. Farklı mimarilere sahip tekrarlayan sinir ağları, uzun kısa süreli bellek, çift yönlü LSTM'ler ve büyük dil modelleri üzerinde durulmuştur. Yapay sinir ağlarının öğrenme süreçleri, ileri ve geri yayılım, optimizasyon algoritmaları ve aktivasyon fonksiyonları açıklanmıştır. Performans metrikleri kısmında, modelin etkinliğini ölçmek amacıyla doğruluk ve F-Skor gibi kriterler incelenmiştir.

Beşinci bölümde veri setinin ön işleme süreçleri, hata türünün tespiti, hatalı tokenlerin belirlenmesi ve onarılması gibi konular detaylı bir şekilde incelenmiştir. Model seçimi ve karşılaştırmaları, mutasyon testleri, PyNar Sohbet Robotu Hata Düzeltme Öneri Sistemi üzerinde durulmuştur. Test verilerinin sınıflara göre sonuçları ve bu sonuçların karşılaştırmaları yapılmıştır.

Çalışmanın son bölümünde ise elde edilen bulgular özetlenmiş ve gelecekteki araştırmalar için öneriler sunulmuştur. Bu bölüm, tezin ana katkılarını ve potansiyel etkilerini vurgular niteliktedir.

2. LİTERATÜR TARAMASI

Yazılım geliştirmede hata düzeltme hem zaman hem de kaynak açısından yoğun bir süreçtir. Bu işlemi iyileştirmek için Otomatik Program Onarımı (APR) alanı sürekli olarak gelişme göstermektedir. APR, yazılım hatalarını otomatik olarak tanımlayıp düzeltme sürecini otomatize etmektedir. Bu sayede yazılım kalitesini artırmayı ve geliştirme süreçlerini hızlandırmayı hedeflemektedir. Bu başlık, otomatik program onarımı için mevcut yaklaşımları ve bu alandaki araştırmaların genel bir bakışını sunmayı amaçlamaktadır. APR'nin gelişimi, yazılım mühendisliği alanında önemli bir dönüm noktasıdır. Bu alandaki yenilikler, yazılım geliştirme pratiklerini dönüştürme potansiyeline sahiptir.

2.1 Şablon Tabanlı Yaklaşımların Kullanıldığı Çalışmalar

Şablon tabanlı metotlar, otomatik program onarımı disiplininde önemli bir rol oynamaktadır. Bu yaklaşım, önceden tanımlanmış düzeltme şablonlarını kullanarak yazılım hatalarını tespit etmeyi ve düzeltmeyi amaçlamaktadır. Şablonlar, belirli hata türleri veya kod yapısı için özel olarak tasarlanarak APR sürecinde kullanılmaktadır. Bu başlıkta, şablon tabanlı yaklaşımların temel prensipleri ve literatürdeki önemli çalışmalar incelenerek bu yaklaşımın APR alanındaki rolü vurgulanacaktır.

Kim ve diğerlerinin çalışmasında mevcut otomatik yama üretim tekniklerinin, özellikle genetik programlama kullanan GenProg'un sınırlamaları ele alınmıştır (Goues ve diğ, 2011). Bu tekniklerin mutasyon işlemlerinin doğasında var olan rastgelelik nedeniyle sıklıkla anlamsız düzeltmeler ürettiği belirtilmiştir. Yazarlar, bu sorunu aşmak için gerçek kişiler tarafından yazılan programlardan öğrenilen yaygın düzeltme örneklerini kullanan desen tabanlı otomatik program düzeltme (PAR) adı verilen bir yöntem geliştirmişlerdir. Bu yöntemde, 60.000'den fazla insan tarafından yazılan programlar manuel olarak incelenerek tekrar eden düzeltme örnekleri belirlenmiş ve bu örnekler program düzeltmelerini otomatik olarak üretmek için kullanılmıştır. *PAR* metodolojisi, programda muhtemelen hatalı olabilecek şüpheli ifadeleri belirlemek için istatistiksel hata lokalizasyon tekniklerini, yaygın düzeltme

örneklerine dayanan düzeltme şablonlarını ve bu şablonlar kullanılarak oluşturulan program varyantlarının test vakalarına göre değerlendirilmesini içermektedir. Bu yöntem, çeşitli açık kaynak projelerden seçilen 119 gerçek hata üzerinde değerlendirilmiş ve performansı *GenProg* ile karşılaştırılmıştır. Değerlendirme, düzeltilen hata sayısı ve üretilen düzeltmelerin kabul edilebilirliği açısından yapılmıştır. Bu araştırmada, otomatik program onarımı metotlarının verimliliği karşılaştırmalı olarak değerlendirilmiştir. Araştırmada, *PAR* ve *GenProg* isimli iki APR yöntemi incelenmiştir. Bulgular, *PAR*'ın 119 hatanın 27'sini başarıyla düzeltirken *GenProg*'un 16 hata için başarılı olduğunu göstermiştir. Ayrıca 89 öğrenci ve 164 geliştirici üzerinde yapılan bir kullanıcı çalışması sonucunda *PAR* tarafından oluşturulan yamaların, *GenProg* tarafından oluşturulanlara kıyasla daha kabul edilebilir olduğu belirtilmiştir. Araştırmada, kişiler tarafından yazılmış programlardan alınan yaygın düzeltme örneklerini kullanılmıştır. *PAR*'ın literatürdeki APR tekniklerine göre daha etkili ve kabul edilebilir sonuçlar ürettiği belirtilmiştir (Kim ve diğ, 2013).

Martinez ve Monperrus (2016), otomatik program onarımını desteklemek için Java tabanlı *ASTOR* kütüphanesini geliştirmişlerdir. Kütüphane, C dilinde yazılmış kodları onaran üç önemli onarım yaklaşımının Java uygulamalarını içermektedir: jGenProg2 (Westley ve diğ, 2009), *jKali* (Qi ve diğ, 2015) ve *jMutRepair* (Debroy ve Wong, 2010). Bu yaklaşımda, hatalı Java kodlarını otomatik olarak düzelten bir dizi farklı algoritma ve teknik kullanılmıştır. *jGenProg2*, Java için *GenProg*'un bir uygulamasıdır. Mevcut koddan alınan ifadeleri kullanarak aday düzeltmeler üretilmesini sağlamıştır. *jKali,* C dilindeki *Kali*'nin Java uygulamasıdır. Zayıf test kodlarının ve yetersiz belirtilmiş hataların tanımlanması amaçlanmıştır. *jMutRepair* ise mutasyon testinden alınan operatörleri kullanarak C kodunu onaran bir yaklaşımın Java uygulamasına dönüştürülmüştür. Ayrıca şüpheli if durum ifadelerine mutasyon operatörleri uygulanmıştır. *jGenProg2* ve *jKali* ile test kod parçaları kullanılarak onarım yapılmıştır. Bu süreç, girdi olarak hatalı bir program ve en az bir başarısız test durumu alınarak bu hatayı düzelten bir kod parçası üretilmesini sağlamıştır. *jMutRepair* ise, mutasyon tabanlı bir onarım yaklaşımıdır. Belirli mutasyon operatörleri kullanılarak ifadelerde tek bir değişiklik yapılmıştır. Her modda, farklı onarım stratejileri ve algoritmaları kullanılmıştır. *Astor* kütüphanesi ile bu modlar birleştirilmiştir. *Astor*'un değerlendirmesi, açık kaynaklı Java programlarından 224

gerçek hatayı onarmak için kullanılmıştır. *jGenProg2*, *jKali* ve *jMutRepair* ile sırasıyla 224 hatanın 29'u, 22'si ve 17'si başarıyla onarılmıştır. Toplamda 33 hata (yüzde 14,7) onarılmıştır. Ayrıca *jGenProg2*'nin yerel farkındalıklı onarım özelliği, onarım süresini önemli ölçüde azaltmıştır. Örneğin, uygulama kapsamındaki ortalama onarım süresi 41,1 dakika iken dosya kapsamında bu süre 11,1 dakikaya düşürülmüştür (Martinez ve Monperrus, 2016).

Genesis isimli çalışmada ise Java projelerinden elde edilen insan tarafından yazılmış kod parçaları kullanılarak otomatik düzeltme üretimi için kod dönüşümlerini çıkarmayı otomatikleştirilmiştir. Sistem, 372 Java projesinden toplanan gerçek dünya kod parçaları ve hataları üzerinde test edilmiştir. Başarılı düzeltmelerden, düzeltme üretimi dönüşümlerini veya aday düzeltme arama alanlarını otomatik olarak çıkaran ilk sistem olma özelliğini taşımaktadır. *Genesis*, gerçek kişiler tarafından yazılmış başarılı kod parçalarından oluşturulan bir eğitim setini analiz ederek başlamakta ve bu kod parçalarının alt kümelerini güncelleştirerek aday düzeltmelerin arama alanını oluşturan dönüşümleri çıkarmıştır. Bu süreç, Genesis'in farklı geliştiriciler tarafından kullanılan geniş bir düzeltme üretim stratejisi yelpazesini yakalamasını sağlamıştır. Dönüşümler, serbest şablon değişkenleri için üreteçler ile şablon soyut söz dizimi ağaçlarına dayandırılmıştır. Bu tasarım Genesis'in düzeltmelerin ve uygulamaların özel ayrıntılarını soyutlamasına, çeşitli uygulamalar arasında yaygın kod desenlerinin yakalanmasına yardımcı olmuştur. Genelleştirme işlemi, eğitim verilerinden elde edilen AST çiftlerinde gözlemlenen değişiklikleri en azından üretebilen bir dönüşüm setinin oluşturulması ile yapılmıştır. Bu süreç, yeni jeneratörlerin sıfırdan oluşturulması ve mevcut AST ağaçlarından kopyalanması kombinasyonu kullanılmıştır. Potansiyel olarak çok sayıda dönüşümü yönetmek için *Genesis*, üretken aday dönüşümleri sistematik olarak seçen bir örnekleme algoritması kullanılmıştır. Bunları kapsama ve uygulanabilirlik açısından değerlendirilmiştir. *Genesis*, dönüşüm seçimi, tam sayı doğrusal programlama olarak formüle edilmiştir (Graver, 1975). Buradaki amaç, düzeltme arama alanının kapsamını uygulanabilirlik ile dengelemektir. Böylece etkili düzeltme üretimi sağlanırken aşırı uyuma karşı korunmuştur. Sistem, Java programlarındaki üç farklı hata türünde test edilmiştir: “null pointer” (null işaretçi), “out of bounds” (sınırların dışında) ve “class cast exception” (sınıf dökümü istisnası). Eğitim seti, 356 açık kaynak uygulamasından program parçaları içerirken karşılaştırma hataları 41 uygulamadan toplanmıştır. *Genesis*, 49

karşılaştırma hatasından 21'ini başarıyla düzelten kod parçaları üretmiştir. Sistem, çok sayıda dönüşüm kullanarak geniş bir kod parçası deseni yelpazesini kapsamakta ve bu dönüşümlerin seçimi, sonuçta ortaya çıkan düzeltme arama alanının genel uygulanabilirliğini sağlamak için yapılmıştır. Mevcut kod parçalarından dönüşümleri çıkarma ve yeni kod parçaları için üretken bir arama alanı oluşturma yeteneği, yazılım geliştirmede yama üretim sürecini akıcılaştırma potansiyelini göstermiştir. Çeşitli Java hataları üzerindeki performansı, pratik uygulanabilirliği vurgulanmıştır (Long ve diğ, 2017).

Hua ve diğerlerinin çalışması olan *SketchFix*, hatalı programları "sketch" adı verilen kısmi programlara dönüştürmektedir. Ardından onarım işlemi program senteziyle eşdeğer bir sürece çevrilmektedir. *SketchFix*, AST düzeyinde dönüşüm şemalarını kullanarak hatalı ifadeleri tespit etmektedir. *EdSketch* adlı bir "sketch" motoru ile bu ifadelerdeki boşlukları doldurmuştur. AST düzeyinde yapılan dönüşümler sayesinde birçok potansiyel düzeltme adayı üretilmiştir. *SketchFix*'in işleyişi, *EdSketch* (Jinru ve Khurshid, 2017) aracılığıyla gerçekleştirilen pratik "sketch" sentezine dayandırılmıştır. Bu süreç, test setine göre gerçekleşmektedir. Testler sırasında gerçek zamanlı olarak aday düzeltmeler üretilmiştir. Her bir "sketch", binlerce somut düzeltme adayını temsil edebilmiştir. Bu adaylar, test süreci sırasında talep üzerine oluşturulmuştur. Bu sürece testler başarısız olana veya tüm testleri geçen bir çözüm bulunana kadar devam edilmiştir. *SketchFix*'in etkinliği, *Defects4J* veritabanındaki 357 hata üzerinde yapılan testlerle ölçülmüştür. Bu testlerde, *SketchFix* ortalama 23 dakikada 19 hatayı başarıyla düzeltebilmiştir. Özellikle AST düğüm seviyesinde ifade manipülasyonu gerektiren hataların onarımında diğer tekniklere göre daha etkili olduğu gözlemlenmiştir. İlk başarılı düzeltme bulunduğunda *SketchFix*'in tüm adayların sadece küçük bir kısmı için yeniden derleme ve test yürütme işlemi yapması gerektiği, bu durumun arama alanını önemli ölçüde azalttığını ve verimliliğini artırdığını göstermiştir. Bu çalışma, program onarımı alanında AST düzeyinde dönüşüm ve talep üzerine aday üretimi gibi yenilikçi yaklaşımlar sunarak mevcut tekniklerden farklı bir bakış açısı getirmiştir. Bu alandaki etkinliği ve verimliliği artırmıştır (Hua ve diğ, 2018).

"Mining StackOverflow for Program Repair" isimli çalışmada ise *Stack Overflow* isimli platformdan kod örnekleri çıkarılmıştır. Bu örneklerden onarım kalıpları çıkarılarak otomatik program onarımı için yenilikçi bir yaklaşım olan *SOFIX*

sunulmuştur. *SOFIX, Stack Overflow* gönderilerini XML formatında analiz ederek sorular ve kabul edilen cevaplar arasındaki bağlantıları yeniden oluşturulmuştur. Bu süreç, soruları ve kabul edilen cevaplarını *QAList* listesine ekleyerek arama çabasını azaltma amaçlanmıştır. Daha sonra bağlantılı değişiklik dizileri, eylem türleri ve değiştirilen öğeler (ve bu öğelerin ana öğeleri) dikkate alınarak izomorfik olup olmadıklarını belirlenerek birleştirilmiştir. Bu, *SOFIX*'in değişiklik dizilerini kategorilere ayırmasına ve her kategori için bir onarım modeli çıkarmasına olanak sağlamıştır. Çıkarılan onarım modellerine dayanarak hatalı kodu değiştirmek için bir dizi onarım şablonu uygulanmıştır. Bu şablonlar ile uyumlu kullanım senaryoları ve düzeltmeler oluşturulurken değerleri nasıl birleştirecekleri tanımlanmıştır. *SOFIX*, gerçek dünya hatalarını onarma etkinliği (RQ1), diğer yaklaşımlarla karşılaştırıldığında hangi hataları onardığı (RQ2), onarım şablonlarının diğer yaklaşımlardakilerle karşılaştırılması (RQ3) ve *SOFIX*'in iç ve altta yatan tekniklerinin etkileri (RQ4) olmak üzere birkaç yönüyle değerlendirilmiştir. *SOFIX*'in, *Defects4J* kriterinde toplam 23 hata onararak önceki yaklaşımlardan daha etkili olduğu belirtilmiştir. *SOFIX*'in onarım şablonlarından altısı, diğer yaklaşımlarda yer almayan benzersiz şablonlar belirlenmiştir. Bu da *SOFIX*'in hataları onarma konusunda daha etkili olduğunu göstermiştir. Genel olarak, *SOFIX*'in, *Stack Overflow* gönderilerinin geniş kaynağını kullanarak mevcut program onarım yaklaşımlarının sınırlamalarına bir çözüm sunduğu ve bu sayede daha geniş bir hata yelpazesini daha kapsamlı bir onarım şablon seti aracılığıyla onarabildiği belirtilmiştir (Liu ve Zhong, 2018).

REVISAR, otomatik program onarımı (APR) amacıyla geliştirilmiş olup özellikle performans ve güvenlik sorunlarını çözmeye yönelik düzenlemelere odaklanmıştır. Bu araç ile Java dilinde yazılmış kodların revizyon geçmişlerini analiz ederek sıklıkla yapılan düzenlemelerin belirlenmesi hedeflenmiştir. Araştırmacılar, bu aracı kullanarak kod düzenlemelerini gruplara ayırmış ve her grupta belirli bir düzenleme modeli öğrenmeyi hedeflemiştir. Bu süreç toplanan verilerin incelenmesi, düzenleme modellerinin değerlendirilmesi ve geliştiricilere yönelik anketler ile *GitHub* üzerinde gerçekleştirilen pull request denemeleri ile desteklenmiştir. *REVISAR*'ın bulduğu 89 farklı düzenleme modelinin, mevcut araçlarda bulunmayan yeni modeller olduğu tespit edilmiştir. Geliştiriciler, bu modelleri genellikle tercih etmiş ve yapılan pull requestlerin büyük bir bölümü kabul edilmiştir. Bu da aracın pratikteki uygulanabilirliğini göstermiştir. Ancak, *REVISAR*'ın sadece belirli tipteki

düzenlemeleri tespit edebildiği ve yalnızca Java tabanlı projelerde çalıştığı belirtilmiştir. Çalışmanın sınırlamaları vurgulanmıştır. Bu sınırlamalar, aracın genel kullanım alanını daraltmakta ve çeşitli programlama dilleri veya bağımlı düzenleme modellerini kapsayabilmesi için daha fazla geliştirilmesi gerektiğini göstermiştir (Rolim ve diğ, 2018).

Soyut Söz Dizimi Ağacı (AST) düğümleri kullanılarak geliştirilen APR araçlarından biri de *CapGen* olmuştur. *CapGen*, AST düğümlerinin bağlam bilgilerini dikkate alarak program onarımı sürecini iyileştirmiştir. Bu yaklaşım, arama tabanlı otomatik program onarımının etkinliğini artırmayı ve doğru düzeltmelerin bulunma olasılığını yükseltmeyi hedeflemiştir. *CapGen*'in öne çıkan özellikleri AST düğümlerinin bağlam bilgisine dayalı olarak mutasyon operatörlerini ve düzeltme bileşenlerini önceliklendirmesi olmuştur. *CapGe*n'in bu özellikleri program hatalarının daha etkili ve doğru bir şekilde düzeltilmesini sağlamıştır. Ayrıca, bağlama duyarlı modeller kullanılarak doğru yamaların yanlış olası yamalardan önce sıralanması sağlanmıştır. Bu modeller, söz dizimsel bağlam modeli, semantik bağlam modeli ve kullanım bağlamı modeli olarak üçe ayrılmıştır. Söz dizimsel bağlam modeli, kodun söz dizimsel yapısını ve çevresindeki elemanları inceleyerek ifadelerin potansiyel bir düzeltmede nasıl değiştirilebileceğini tahmin etmektedir. Semantik bağlam modeli, kodun semantik özelliklerine odaklanarak değiştirilmesi gereken kod parçalarının işlevlerini ve bu işlevlerin genel program içindeki rollerini analiz etmektedir. Kullanım bağlamı modeli ise, kod parçasının kullanım senaryolarını ve bağlamını inceleyerek bir fonksiyonun çağrıldığı yerleri ve bu çağrıların bağlamını değerlendirmektedir. *CapGen*'in başarımı, *Defects4J* veri seti üzerinde gerçekleştirilen testlerle değerlendirilmiştir. Bu testlerde %84'lük bir doğruluk oranı elde edilmiş, bu da *CapGen*'in doğru düzeltmeleri %98,78 oranında yanlış olası düzeltmelerden önce sıralayabildiğini göstermiştir. Bu yaklaşım, etkili otomatik program onarımı çözümleri geliştirmek için önemli bir adım olarak değerlendirilmiştir (Wen ve diğ, 2018).

Liu ve diğerlerinin çalışmasında, statik analiz ihlallerini düzelten geliştirici düzeltmelerinden elde edilen düzeltme kalıpları kullanılarak düzeltme üretimi için bir yöntem ortaya konulmuştur. Araştırmanın temel odak noktası, onarım sürecinin doğruluğunu artırmak ve anlamsız düzeltme üretme olasılığını azaltmak olmuştur. Yöntem, açık kaynak projelerden statik analiz ihlalleriyle ilgili düzeltmeleri toplayarak ve bu verileri Soyut Söz Dizimi Ağacı tabanlı bir kod farklılaştırma aracı olan

GumTree kullanarak somut değişiklik eylemlerini çıkarmak için ön işlemden geçirerek başlamaktadır. Daha sonra benzer düzenleme betiklerini gruplandırarak grubun genelinde tutarlı bir düzeltme kalıbı çıkarılmıştır. Bu kalıplar, düzenleme betiklerinin özelliklerini öğrenmek için derin öğrenme çerçeveleri (CNN'ler) kullanılarak elde edilmiştir. Geliştirilen otomatik kod onarım aracına *AVATAR* ismi verilmiştir. *AVATAR*'ın onarım süreci, hatalı kod konumlarını *GZoltar* çerçevesi (Campos ve diğ, 2012) ve Ochiai sıralama metriği (Abreu ve diğ, 2006) kullanarak tanımlamak, şüpheli kod konumlarını çıkarılan düzeltme kalıplarıyla eşleştirmek, düzenleme betiklerindeki onarım eylemlerini uygulayarak düzeltme adayları oluşturmak ve bu düzeltmeleri test durumları kullanılarak değerlendirmekten oluşturulmuştur. *AVATAR*'ın performansı, *Defects4J* ölçütü üzerinde değerlendirilmiş ve mükemmel hata lokalizasyonu varsayımı altında 34 hata için doğru düzeltme ve 5 hata için kısmen doğru düzeltme üretebildiği gösterilmiştir. Bu performans, yazarlar tarafından literatürdeki benzer yaklaşımlarla karşılaştırıldığında rekabetçi bulunmuş ve mevcut yaklaşımlara tamamlayıcı bir doğası olduğu gözlemlenmiştir. Bu çalışma ile özellikle düzeltme doğruluğu zorluğunu ele alırken statik analiz ihlali düzeltme kalıplarını yenilikçi bir şekilde kullanarak otomatik program onarımında önemli bir ilerleme sunulmuştur (Liu ve diğ, 2019).

Şablon tabanlı yaklaşımların otomatik program onarımı (APR) alanındaki önemi, farklı tekniklerin ve yöntemlerin literatürdeki incelemesiyle ortaya konmuştur. *GenProg, PAR, ASTOR, Genesis, SketchFix, SOFIX* ve diğerleri gibi çeşitli araştırmalar, APR'nin gelişimine katkıda bulunmuş ve bu alanın çeşitliliğini göstermiştir. Bu tekniklerin, hata düzeltme kalıplarından AST düğümlerine ve hatta *Stack Overflow* gibi platformlardan elde edilen verilere kadar geniş bir yelpazeyi kapsadığı görülmektedir. Her bir yaklaşım, APR'nin farklı yönlerine odaklanarak bu alanda etkinlik, verimlilik ve doğruluk açısından yeni yollar sunmuştur. Bu çeşitlilik, otomatik program onarımının geleceğinin sadece kod düzeltme teknikleriyle sınırlı olmadığını, aynı zamanda gelişmiş veri analizi ve öğrenme algoritmalarıyla bütünleşmiş bir yaklaşım gerektirdiğini göstermektedir.

2.2 Makine Öğrenmesi Yöntemlerinin Kullanıldığı Çalışmalar

Makine öğrenmesi, otomatik program onarımı (APR) alanında giderek daha fazla kullanılan bir yaklaşımdır. Bu yaklaşımda, yazılım hatalarını tespit etmek ve

düzeltmek için öğrenme algoritmaları kullanılmaktadır. Makine öğrenmesi tabanlı APR, büyük veri kümesi analizleri ve öğrenme modeli oluşturma süreçleriyle yazılım hatalarını otomatik olarak saptama kapasitesine sahiptir. Bu başlıkta, makine öğrenmesi ile otomatik program onarımı alanındaki önemli çalışmalar gözden geçirilmektedir.

Nesne yönelimli programlar için özel olarak tasarlanmış bir üret ve doğrula onarım tekniği olan *ELIXIR* geliştirilmiştir. *ELIXIR*'in temel yeniliği, düzeltme üretmek için onarım ifadeleri oluştururken yerel değişkenler, alanlar ve sabitlerin yanı sıra yöntem çağrılarının geniş kullanımına dayanmasıolmuştur. Bu yaklaşımda, yöntem çağrılarının daha yaygın kullanımı nedeniyle genişleyen onarım alanının karşılaştığı zorluğu etkili bir şekilde ele alınmıştır. *ELIXIR* ile program bağlamı ve hata raporlarından türetilen dört anahtar özellik üzerine dayanan bir makine öğrenimi modeli kullanılarak potansiyel onarımlar sıralanmış ve önceliklendirilmiştir. *ELIXIR,* programdaki olası hatalı ifadeleri Ochiai tekniğini (Abreu ve diğ, 2006) kullanarak belirlemiştir. Program dönüşüm şemaları kullanılarak onarım ifadeleri üretilmiş daha sonra bu şemalar aday düzeltmeleri üretmek için kullanılmıştır. Bunlar, hatalı bağlamla ilgili yöntem çağrıları, alan erişimleri ve diğer ifadeleri içermektedir. Makine öğrenmesi modeli, aday düzeltmeleri kullanım sıklığı, onarım konumuna mesafe, program bağlamı ile benzerlik ve hata raporu ile ilgililik gibi dört özelliğe dayanarak sıralamıştır. En üst sıradaki düzeltmeler test edilmeye başlanarak uygun bir düzeltme bulunana kadar sürece devam edilmiştir. *ELIXIR*'in verimliliği, yaygın olarak kullanılan *Defects4J* veri seti ve yazarların oluşturduğu yeni *Bugs.jar* veri seti kullanılarak değerlendirilmiştir. Değerlendirme, doğru şekilde onarılan hataların sayısına odaklanmış ve sonuçlar diğer onarım teknikleriyle karşılaştırılmıştır. *ELIXIR, Defects4J*'de doğru onarılan hata sayısının %85, Bugs.jar'da ise %57 oranında artırılmasını sağlamıştır (Saha ve diğ, 2017).

Koyuncu ve arkadaşlarının çalışmasında kod parçaları içindeki değişikliklere uygulanan yinelemeli bir kümeleme stratejisi kullanılmıştır. Bu metodun temel hedefi diğer yama oluşturma sistemlerinde kullanılmak üzere ayrı ve tekrar kullanılabilir düzeltme şablonları elde etmek olmuştur. FixMiner'ın mimarisi, üç katmanlı bir kümeleme stratejisi üzerine kurulmuştur. Bu sistemde, benzer değişiklikleri keşfetmek için soyut söz dizimi ağaçları, düzenleme eylemleri ağaçları ve kod bağlamı ağaçları gibi çeşitli ağaç temsilleri kullanılmıştır. Bu temsiller bağlamları, değişiklik

işlemlerini ve kod tokenlerini kodlamada yardımcı olmuştur. FixMiner, binlerce açık kaynak projeden toplanan yazılım yamaları üzerinde test edilmiştir. Değerlendirme için kullanılan veri seti, 44 büyük ve popüler açık kaynak projeden gelen ve toplamda 11.416 program parçasından oluşturulmuştur. *Defects4J* karşılaştırma setinde 26 hatayı doğru bir şekilde düzeltmiştir. Yapılan araştırma, çıkarılan düzeltme kalıplarının çoğunlukla doğru olan olası program parçaları üretmeye yol açtığını göstermiştir (Koyuncu ve diğ, 2020).

Otomatik program onarımı (APR) üzerine yapılan *CURE* adlı araştırma makalesinde, yazılım güvenilirliğini artırmak amacıyla yeni bir Sinirsel Makine Çevirisi tabanlı APR tekniği önerilmiştir. Mevcut NMT tabanlı APR tekniklerinin yetersiz arama alanı ve yazılım bilgisinin entegrasyonu, katı kod söz dizimi gibi eksikliklerine çözüm olarak bu yeni teknik geliştirilmiştir. *CURE*'de, geniş bir kod tabanından geliştirici benzeri kaynak kod öğrenimi için önceden eğitilmiş bir programlama dili modelinden yararlanılmıştır. Bu model, APR görevleri için önceden eğitme ve ince ayarlama iş akışını uygulayarak kodu insan geliştiricilere benzer şekilde anlayabilen ve üretebilen NMT modellerini geliştirmek için kullanılmıştır. Ayrıca kod için uyarlanmış bir ışın arama stratejisi kullanılarak derlenebilir yamalar ve hatalı koda benzer uzunluktaki yamalar üretilmeye odaklanılmıştır. Bu strateji, geçerli tanımlayıcı kontrolü ve uzunluk kontrolü stratejilerini içermektedir. *CURE*'da ayrıca, kelime dağarcığını azaltarak arama alanını iyileştirmek için bayt çifti kodlaması kullanılarak alt kelime tokenizasyonu yapılmıştır. *CURE*, yukarıdaki teknikler yeni bir APR mimarisine birleştirilerek 4.04 milyon metot üzerinden 1700 açık kaynak projeden kod söz dizimi ve geliştirici benzeri kaynak kodu yakalanmıştır. Bu yaklaşım, dış kelime dağarcığı sorununu etkili bir şekilde ele almıştır. Değerlendirme ve sonuçlar açısından, *CURE* ile 57 *Defects4J* ve 26 *QuixBugs* hatasını düzeltilerek tüm mevcut APR tekniklerinden daha başarılı olduğu gösterilmiştir. Diğer yaklaşımların düzeltemediği özel hatalar, örneğin *Defects4J*'deki "Chart 17" hatası, *CURE* tarafından düzeltilmiştir. Karmaşık hata düzeltmeleri için desen tabanlı yaklaşımlardan daha etkili olan *CURE*, diğer NMT tabanlı modellere göre daha yüksek derlenebilir kod oranı göstererek programlama dili söz dizimlerini ve geliştirici benzeri kodların daha iyi öğrenildiğini göstermiştir. Bu çalışma, *CURE*'un programlama dili modeli, kod-farkında arama stratejisi ve alt kelime tokenizasyonu gibi benzersiz bir kombinasyonu kullanılarak mevcut

yaklaşımlara göre yazılım kodundaki hata düzeltmelerinde daha etkili ve doğru sonuçlar elde edildiği ortaya konulmuştur (Jiang ve diğ, 2021).

Valueian ve diğerlerinin çalışmasında, birden fazla hatayı içeren programların otomatik olarak tamir edilmesine odaklanılmıştır. Geliştirilen *SituRepair* tekniği, bir makine öğrenimi modeli kullanarak hata türlerini ve durumlarını tahmin etmektedir. Bu bilgilere dayanarak hatalı programlarda değişiklikler yapmaktadır. *SituRepair*, her hatalı program için 50 statik özellik hesaplamıştır. Bu özellikler, bilinen hataların türlerine ve durumlarına göre bir makine öğrenimi modeli oluşturmak için kullanılmıştır. Bu model, yeni bir hatalı program için bu özellikler temelinde hata türlerini ve durumlarını tahmin etmektedir. Tahmin edilen hata türleri ve durumlarına göre programda çeşitli değişiklikler yapılmıştır. Bu süreç, tüm test vakalarını geçene kadar veya başka değişiklik yapılamayana kadar devam etmiştir. *SituRepair*'in etkinliğini değerlendirmek için C programlama diliyle yazılmış 25.619 program üzerinde kapsamlı deneyler yapılmıştır. Bu deneylerde, *SituRepair* 17.195 programı başarıyla tamir etmiştir. Bu programların 13.347'si tek hatalı ve 3.848'i çoklu hatalı programlardır. Bu sonuçlar, *SituRepair'*in birden fazla hatayı içeren programların otomatik tamiri için potansiyel bir teknik olduğunu göstermektedir. Ayrıca, SituRepair her programı ortalama 9,24 dakikada tamir edebilmiştir (Valueian ve diğ, 2022).

Moosavi ve diğerlerinin çalışmasında *Doctor Code* adında yeni bir APR tekniği tanıtılmıştır. Bu teknik, en yaygın hata tiplerinin özelliklerinin analiz edilmesi ve makine öğreniminin kullanılmasıyla onarım operatörlerinin seçilmesini sağlamaktadır. Yapılan deneylerde, *Doctor Code'un* Siemens paketi ve *Defects4j* veri setinde diğer APR araçlarına göre daha fazla hatayı düzeltebildiği ve daha az uyumsuz düzeltme sağladığı gözlemlenmiştir. Ayrıca büyük boyutlu programları onarabilme yeteneği ve mevcut arama alanında olmayan düzeltmeler üretebilme kapasitesine sahip olduğu belirtilmiştir. Makalede vurgulanan bir diğer önemli nokta, *Doctor Code*'un makine öğrenimi temelli yaklaşımıdır. Bu yaklaşım sayesinde program onarım operatörlerinin seçimi iyileştirilmiş böylece daha doğru düzeltmelerin üretilmesi ve onarım sürecinin kısaltılması sağlanmıştır. Siemens paketinde ve *Defects4j* veri setinde yapılan deneylerde, *Doctor Code*'un diğer mevcut APR araçlarına kıyasla daha fazla hatayı düzeltebildiği ve daha az uyumsuz düzeltme sağladığı ortaya konmuştur. Bu durum, *Doctor Code*'un çeşitli programlama dillerinde ve farklı hata tipleriyle uyumlu çalışabildiğini ayrıca büyük boyutlu programlar üzerinde de etkili olduğunu

göstermektedir. Sayısal başarım değerlerine bakıldığında, Siemens paketinde yapılan deneylerde *Doctor Code*'un 41 doğru düzeltme sağladığı görülmüştür. Bu, aynı veri setinde mutasyon onarımı (mutation repair) tekniğinin sağladığı 22 doğru düzeltmeye kıyasla önemli bir artışı temsil etmektedir. *Defects4*j veri setinde kullanılan 7 modern APR aracıyla yapılan karşılaştırmada ise *Doctor Code*'un diğer araçlara göre daha fazla hatayı düzeltebildiği belirlenmiştir (Moosavi ve diğ, 2023).

Makine öğrenimi tabanlı yaklaşımlar, otomatik program onarımı alanında, yazılım hatalarını tespit etme ve düzeltme konusunda yenilikçi ve etkili çözümler sunmuştur. *ELIXIR, FixMiner, CURE, SituRepair* ve *Doctor Code* gibi çeşitli yöntemler, bu alanda etkili adımlar atmıştır. Bu teknikler, geniş veri setlerinden öğrenme, hata türlerinin tahmini, onarım operatörlerinin seçimi ve yama üretiminde yenilikçi stratejiler uygulama gibi çeşitli yönlerden APR'ye katkıda bulunmuşlardır. Her biri, onarım sürecinin belirli bir yönüne odaklanarak özellikle büyük veri analizleri ve öğrenme modelleri kullanarak APR'nin etkinliğini ve doğruluğunu artırmayı hedeflemiştir. Bu çalışmaların gösterdiği gibi makine öğrenimi tabanlı APR, yazılım onarımında yeni ve gelişmiş tekniklerin uygulanmasını sağlamakta ve bu alanda daha geniş bir potansiyel yelpazesi açmaktadır.

2.3 Derin Öğrenme Yöntemlerinin Kullanıldığı Çalışmalar

Derin öğrenme, otomatik program onarımı alanında son yıllarda büyük ilgi çeken bir yaklaşım haline gelmiştir. Bu yaklaşım, çok katmanlı sinir ağları ve derin öğrenme tekniklerini kullanarak karmaşık yazılım hatalarını tespit etme yeteneği sunmaktadır. Derin öğrenme tabanlı APR, büyük ve karmaşık yazılım projelerinde kullanılabilirlik açısından önemlidir. Bu bölümde, derin öğrenme tabanlı APR teknikleriyle ilgili literatürdeki çalışmalar incelenmektedir.

Gupta ve diğerlerinin *DeepFix* isimli çalışmasında, C dilinde genellikle yeni başlayan programcıların deneyimsizlikleri veya detaylara yeterince dikkat etmemeleri sonucu ortaya çıkan yaygın hatalar hedef alınmıştır. Çalışmada, çok katmanlı bir diziden dizi (seq2seq) sinir ağı kullanılarak hem bir programdaki hatanın yeri hem de gerekli doğru ifadelerin tahmin edilmesi amaçlanmıştır. *DeepFix* aracında, bir programdaki birden fazla hatayı birer birer düzeltmek için yinelemeli bir tamir stratejisi uygulanarak düzeltilen hataların doğrulanması için bir derleyici kullanılmıştır. Sistem, bir programı token dizileri olarak ele almış ve bu tokenları tahmin ve düzeltme için manipüle

etmiştir. *DeepFix*'in başarısı, verilen bir program setindeki hataları tamamen veya kısmen düzeltebilme yeteneğiyle ölçülmüştür. Sistemin program içindeki hata yerlerini doğru bir şekilde belirleyebilme yeteneği, geniş bir programlama görevleri ve hata türleri yelpazesi boyunca hataları ele alabilme yeteneği başka önemli kriterler olarak değerlendirilmiştir. 6971 hatalı C programı üzerinde yapılan bir testte, DeepFix'in 1881 programı tamamen ve 1338 programı kısmen başarılı bir şekilde düzelttiği belirtilmiştir. Toplam 16743 hata mesajından 5366'sı çözülmüştür. *DeepFix*'in test programındaki hataları düzeltmesi sadece birkaç 10 milisaniye gibi kısa bir sürede gerçekleştirilmiştir (Gupta ve diğ, 2017).

Gupta ve diğerlerine ait başka bir çalışmada, insan benzeri davranışları taklit edebilen bir yapay zeka ajanı geliştirilmiştir. Bu ajan, programlama dilinin karmaşık kurallarını bilmeksizin doğrudan ham program metinlerinden öğrenme yetisine sahiptir. Çalışmada, C programlama dilinde yazılmış 6975 hatalı program üzerinde denemeler yapılmış ve bu yeni yaklaşımın mevcut bir araç olan *DeepFix*'ten daha başarılı olduğu gösterilmiştir. Bu yeni sistem sayesinde eğitim verilerinin sadece onda birinde uzman gösterimler kullanılarak *DeepFix*'ten daha iyi sonuçlar elde edilmiştir. Bu durum, pekiştirmeli öğrenme yöntemlerinin programlama dili düzeltme konusunda etkili bir yol olduğunu ortaya koymuştur. Yapılan denemelerde sistem, hatalı kodların %26'sını başarıyla düzeltebilmiş ve bu oran *DeepFix*'te %22 olarak kaydedilmiştir. Bu sonuçlar, bu yeni yaklaşımın, özellikle eğitim verilerinin sınırlı olduğu durumlarda mevcut yöntemlere göre daha etkili olduğunu göstermiştir (Gupta ve diğ, 2019).

Devign aracının geliştirildiği çalışmanın ana hedefi, kaynak kodlarındaki kırılgan fonksiyonları yerelleştirmek olmuştur. Araştırmacılar, zengin kod semantik temsillerinden özellikler çıkarmak üzere yenilikçi bir evrişim modülü kullanan graf sinir ağlarına dayalı bir model geliştirmişlerdir. Bu model, dört geniş çaplı ve açık kaynak C projelerinden el ile etiketlenmiş veri setlerini kullanarak eğitilmiştir. Bu projeler, gerçek dünya kaynak kodunun gerçekçi bir temsili olarak karmaşıklık ve çeşitlilik açısından seçilmişlerdir. Araştırmacılar, *Devign*'in etkinliğini doğruluk ve F1 puanı ile ölçmüşlerdir. Bu modelin performansı, kırılganlık tespitindeki diğer yöntemlerle karşılaştırılmıştır. Değerlendirme, *Linux Kernel, QEMU, Wireshark* ve *FFmpeg* gibi çeşitli kaynaklardan gelen veri setleri üzerinde yapılmıştır. Çalışmanın sonuçları, *Devign*'in mevcut yöntemlerden önemli ölçüde daha üstün olduğunu göstermiştir. Model, ortalama %10,51 daha yüksek doğruluk ve %8,68 daha yüksek

F1 puanı elde etmiştir. Araştırmacılar, evrişim modülünün kullanımının tek başına doğrulukta ortalama %4,66 ve F1 puanında %6,37'lik bir artışa katkıda bulunduğunu belirtmişlerdir. Devign ile %74,11 doğruluk oranı elde edilmiştir (Zhou ve diğ, 2019).

Li ve diğerlerinin çalışmasında ise, kaynak kodundaki güvenlik açıklarını tespit etmek için bir yöntem geliştirilmiştir. Bu yöntemde, kaynak kodu minimum ara temsil (MIR) öğrenme modelini kullanılarak analiz edilmiştir. Süreç, kaynak kodunu MIR'ye dönüştürmeyi ve böylece ilgisiz öğeleri hariç tutarak bağımlılık uzunluğunu azaltmayı amaçlamıştır. Yöntemde ilk olarak kaynak kodu MIR'ye dönüştürülmüştür. Bu, verileri sonraki analizler için basitleştirmiş ve odaklamıştır. Daha sonra modelin programlama dillerinin yaygın söz dizimi özelliklerini öğrenmesine yardımcı olan denetimsiz öğrenme için bir ön eğitim yapılmıştır. Bu aşama, modelin sözlük dışı sorunları ele almasına yardımcı olmuştur. Üç katmanlı birleştirilmiş konvolüsyonel sinir ağları (CNN) kullanılarak MIR'den yüksek seviye güvenlik açığı özellikleri çıkarılmıştır. Son olarak, bu özellikleri kullanarak güvenlik açıklarını tespit edebilen bir sınıflandırıcı eğitilmiştir. Deneyler ve sonuçların değerlendirilmesinde, Yazılım Güvence Referans Veri Seti ve Ulusal Güvenlik Açığı Veritabanı'ndan elde edilen veri setleri tercih edilmiştir. Bu veri setleri, ara bellek taşması (CWE-119) ve kaynak yönetimi hataları (CWE-399) gibi güvenlik açıklarına odaklanmıştır. Yöntem, geleneksel yöntemlere kıyasla doğruluk, hatırlama, yanlış pozitif oranı ve F1 skoru açısından daha üstün performans göstermiştir. Sonuç olarak, önerilen yöntem, özellikle doğruluk ve yanlış pozitifleri azaltma açısından geleneksel yöntemlere göre önemli iyileştirmeler göstermiştir. Bu yaklaşım, özellikle veri işleme ve kontrol mantığının kötüye kullanılmasından kaynaklanan güvenlik açıklarında etkili bulunmuştur (Li ve diğ, 2020).

CodeBERT hem programlama dillerini hem de doğal dilleri anlayabilen büyük ölçekli bir çift modlu önceden eğitilmiş modeldir. Bu model, GitHub'dan elde edilen altı farklı programlama dilindeki verileri kullanarak hem çift modlu (doğal dil ile programlama dili çiftleri) hem de tek modlu (yalnızca kod veya doğal dil) veriler üzerinde eğitilmiştir. Çok katmanlı çift yönlü Transformer mimarisi kullanılarak maskeli dil modelleme ve yerine konulan belirteç algılama gibi iki ana eğitim hedefine odaklanılmıştır. Modelin performansı, doğal dil kod arama ve kod dokümantasyonu üretme gibi görevlerle değerlendirilmiş ve bu alanlarda ortalama karşılıklı sıra ve düzleştirilmiş BLEU puanı gibi ölçütler kullanılmıştır. *CodeBERT*, bu görevlerde en

iyi sonuçları elde ederek doğal dil ve programlama dili belirteçlerini tahmin etme yeteneği konusunda mevcut modellere göre önemli bir üstünlük sergilemiştir. İnce ayar yapılmadan gerçekleştirilen testlerde de diğer önceden eğitilmiş modellere kıyasla üstün bir performans göstermiştir. *CodeBERT*'in genelleştirme kabiliyeti, eğitim verilerinde yer almayan programlama dillerinde bile güçlü bir performans sergileyerek kanıtlanmıştır. Bu model, doğal ve programlama dilleri arasındaki boşluğun giderilmesi için önemli bir ilerleme olarak kabul edilmiştir (Feng ve diğ, 2020).

GraphCodeBERT ismi verilen bir diğer çalışmada programlama dilleri için yeni bir ön eğitimli model sunulmuştur. Bu model, kodun semantik düzey yapısını, özellikle de veri akışını dikkate alarak kodla ilgili görevlerde önemli ilerlemeler sağlamıştır. Bu yaklaşım, kodu temel olarak bir token dizisi olarak işleyen ve yapısındaki kritik semantiği göz ardı eden önceki modellerden ayrılmıştır. Araştırmacılar, *GraphCodeBERT*'i, geleneksel modellerin kullandığı AST gibi söz dizimsel yapıların aksine, değişkenler arasındaki ilişkileri temsil eden ve bir değişkenin değerinin nereden geldiğini gösteren veri akışına odaklanarak geliştirmişlerdir. Bu yapının, AST'nin derin hiyerarşisinden daha basit ve verimli olduğu belirtilmiştir. Model, maskeli dil modellemesine ek olarak iki yenilikçi ön eğitim görevi sunulmuştur. Bu görevler, veri akışı kenar tahmini ve kaynak kodu ile veri akışı arasında değişken hizalamasıdır. İlk görev, veri akışı grafiğindeki kenarları tahmin ederek kodun yapısından öğrenmeyi içermekte ikinci görev ise kaynak kod ile veri akışı arasındaki temsilleri hizalayarak veri akışındaki değişkenlerin hangi kaynak kod tokenlerinden tespit edildiğini tahmin etmektedir. *GraphCodeBERT*, 2.3 milyon fonksiyon içeren ve altı programlama diliyle eşleştirilmiş doğal dil belgeleri içeren *CodeSearchNet* veri seti üzerinde ön eğitim görmüştür. Model, doğal dil ile kod araması, kod klonu tespiti, kod çevirisi ve kod iyileştirme gibi dört farklı görevde test edilmiş ve bu alanlarda üstün performans göstermiştir. Araştırmacıların yaptığı çıkarım çalışması, yeni ön eğitim görevlerini veya veri akışını çıkardığında modelin etkinliğinin azaldığını göstermiştir. Ön eğitimli modelde veri akışı uygulanmasının, *GraphCodeBERT* kodu anlama konusunda gelişmiş bir anlayış sergilediğini göstermiştir (Guo ve diğ, 2020).

PLBART, bir "seq2seq" modeli olarak, programlama dili ile doğal dil arasındaki dönüşümü sağlayarak kod özetlemesi ve kod üretimi yapmaktadır. Ayrıca, eski kodların bir programlama dilinden diğerine geçişini kolaylaştıran kod çeviri

işlemlerini gerçekleştirir. Model, Java ve Python fonksiyonları ile bunlara eşlik eden doğal dil metinler üzerinde gürültü giderme oto kodlayıcı yöntemiyle önceden eğitilmiştir. Yapılan deneyler, *PLBART*'ın İngilizce dilinde kod özetleme, yedi programlama dilinde kod üretimi ve çevirisi gibi görevlerde mevcut modellere göre daha üstün ya da benzer performans sergilediğini göstermiştir. *PLBART* aracında, programlama ve doğal dillerin söz dizimi ve semantiklerini anlama becerisini geliştirmek için kullanılan bir önceden eğitim yöntemi olan gürültü azaltmalı "seq2seq" ön işleme yöntemi kullanılmıştır. Bu yöntemle, model dillere ait söz dizimi ve semantikler anlaşılabilmiştir. Aynı zamanda tutarlı bir şekilde dil üretebilmiştir. *PLBART, Github* ve *StackOverflow* kaynaklarından alınan Java ve Python fonksiyonları ile bunların doğal dil açıklamaları kullanılarak eğitilmiştir. Modelin yapısı, altı seviyeli bir kodlayıcı ve kod çözücü içermektedir. *PLBART, BARTbase* ve "seq2seq Transformer" mimarisini kullanmaktadır. PLBART ile her dil için örneklenmiş bir dizi etiketsiz örnek üzerinde gürültü fonksiyonu kullanılarak eğitim gerçekleştirilmiş ve orijinal örnek tahmin edilmeye çalışılmıştır. İki geniş kategoriye ayrılan alt görevler için ince ayar yapılmıştır. Birincisi dizi üretimi ve diğeri dizi sınıflandırmasıdır. Dizi üretimi görevlerinde, model kod özetleme, üretimi ve çevirisi gibi görevlerde doğrudan ince ayar yapılmıştır. Sınıflandırma görevlerinde ise giriş dizisi hem kodlayıcıya hem de kod çözücüye beslenerek son kod çözücü katmanından alınan son tokenin temsili bir doğrusal sınıflandırıcıya verilmiştir. Bu çalışmada, *PLBART*'ın kaynak kodları ve onlara eşlik eden doğal dil metinlerindeki karmaşık söz dizimsel yapıları tespit etme yetenekleri incelenmiştir. Çeşitli uygulamalar üzerinde, özellikle kod özetleme, kod oluşturma, kod dili çevirisi ve program tamiri gibi alanlarda değerlendirmeler yapılmıştır. Bunun yanı sıra, kod klonlama tespiti ve güvenlik zafiyeti bulma gibi görevler de araştırmanın kapsamına dahil edilmiştir. *PLBART*'ın performansını değerlendirmek için BLEU, CodeBLEU ve Exact Match gibi metrikler kullanılmıştır. Çalışmada, *PLBART*'ın etiketsiz verilerden güçlü program ve dil temsilleri öğrenip öğrenmediğini, program özelliklerini (örneğin söz dizimi, stil ve mantıksal veri akışı) öğrenip öğrenmediğini ve sınırlı anotasyonlarla görülmemiş bir dilde nasıl performans gösterdiğini ele alınmıştır. Sonuçlar, *PLBART*'ın kod özetleme, kod üretimi ve kod çevirisi görevlerinde diğer modellere göre üstün performans sergilediğini göstermiştir (Ahmad ve diğ, 2021).

CodeT5, NLP içinde kodla ilgili görevleri geliştirmek amacıyla bir ön eğitimli model olarak geliştirilmiştir. Kod anlama ve üretme görevlerine odaklanan çoklu görev öğrenimini destekleyen birleşik bir ön eğitimli kodlayıcı-çözücü modeldir. Bu modelde, Transformer modeline dayalı olup *T5*'e benzer şekilde seq2seq ön eğitimi kullanılmıştır. *CodeT5*, programlama dillerinden gelen belirteç türü bilgilerini, özellikle değişken ve fonksiyon isimleri gibi tanımlayıcıları içermektedir. Hem tek modlu (yalnızca PL) hem de çift modlu (PL-NL) veriler kullanarak eğitilmiştir. *CodeT5, CodeSearchNet* veri kümesi ve GitHub'dan ek C/C# verileri kullanılarak eğitilmiştir. *CodeXGLUE* referansı üzerinden çeşitli görevlerde ince ayar yapılarak test edilmiştir. Model, birden fazla görevde aynı anda ince ayar yaparak çoklu görev öğrenimini de içermektedir. Araştırma sonuçları, *CodeT5*'in kapsamlı deneylerde *CodeXGLUE*'nin on dört alt görevinde en iyi sonuçları elde ettiğini göstermiştir. Bu sonuçlar, modelin tanımlayıcıya duyarlı ön eğitimi ve çift modlu çift yönlü üretim sayesinde kod semantiklerini daha iyi yakalayabildiğini ve özellikle doğal dil ile programlama dili arasındaki görevlerde büyük faydalar sağladığını belirtmiştir. Ayrıca, *CodeT5-small*, sadece kod çözücü model olan ve o zamanki en iyi durum olan *PLBART*'ı geçerek, kodlayıcı-çözücü modellerin kod parçacıkları üretmede üstünlüğünü teyit etmiştir. *CodeT5-base* ise, *PLBART*'a kıyasla *CodeBLEU* üzerinde yaklaşık 4,7 puanlık bir iyileşme sağlayarak, tanımlayıcıya duyarlı ön eğitimin yardımıyla kod söz dizimi ve semantiğini daha iyi anlayabildiğini göstermiştir (Wang ve diğ, 2021).

Yasunaga ve Liang'ın çalışmasında, otomatik program onarımı için geliştirilen *Break-It-Fix-It (BIFI)* yöntemi ele alınmıştır. *BIFI*'nin temel amacı, hatalı olarak değerlendirilen kodları iyileştirebilen bir düzeltici (fixer) eğitilmektedir. *BIFI*'nin temel fikri, kötü girdiler üzerinde düzelticinin çıktısını kontrol eden bir eleştirmeni kullanarak iyileştirilmiş çıktıları eğitim verilerine eklemek ve iyi örneklerden gerçekçi kötü kodlar üretecek bir kırıcı eğitilmektedir. Bu yöntem, başlangıçta sentetik eşleşmiş veriler üzerinde eğitilen bir düzeltici ve kırıcının, gerçek ya da gerçekçi şekilde üretilmiş kötü kodlar üzerinde sürekli olarak güncellenmesi ve geliştirilmesi sürecini içermektedir. *BIFI*, eleştirmenin düzeltici ve kırıcı tarafından üretilen çıktıları doğrulayarak gerçekten düzeltilmiş ya da bozulmuş örnekleri eğitim verilerine eklemesiyle öne çıkmaktadır. Bu özellik, *BIFI*'nin geriye çeviri yönteminden farklılaşan önemli bir noktadır. Geriye çeviri, üretilen eşleşmiş verileri doğrulamazken

BIFI gerçek kötü örnekler üzerinde düzelticiyi eğitmektedir. Bu da eğitim verilerinin doğruluğunu ve dağılımını iyileştirmektedir. *BIFI*'nin bir varyasyonu olan 'FixerOnly', yalnızca kötü örnekler üzerinde düzelticiyi eğiten bir sürüm olup, özellikle yeterli sayıda kötü örnek mevcut olduğunda faydalı olmaktadır. Ancak, kötü örneklerin miktarı az olduğunda *BIFI,* kırıcının ürettiği ek eşleşmiş verileri kullanarak daha fazla veri üzerinde eğitim yapabilmesi nedeniyle *FixerOnly*'den daha etkili olmaktadır. Makalede *BIFI, GitHub-Python* ve *DeepFix* adlı iki kod onarım veri seti üzerinde değerlendirilmiştir. Bu veri setleri, sırasıyla Python ve C programlama dillerindeki kod onarımı için kullanılmıştır. *BIFI* uygulama detayları, kod çözücü-kodlayıcı ve Transformer modeli kullanılarak ve belirli parametre ayarları ile gerçekleştirilmiştir. Sonuçlar, *BIFI*'nin başlangıç düzelticisini kötü kodlar ve kırıcı tarafından üretilen kötü kodlarla ekstra eğitim yaparak geliştirdiğini göstermiştir. Bu yaklaşım, özellikle *GitHub-Python* üzerinde başlangıç düzelticisine kıyasla %28,5'lik bir iyileşme sağlamıştır. *DeepFix* veri setinde ise *BIFI,* mevcut en iyi sistem olan *DrRepair*'a kıyasla %5,6'lık bir artışla %71,7'lik bir doğruluk oranına ulaşmıştır (Yasunaga ve Liang, 2021).

PLUR, program anlama ve onarımı için kullanılan çeşitli görevleri ve modelleri birleştiren bir çerçeve olarak tanıtılmıştır. Çalışmanın temel amacı, genel bir yaklaşım kullanarak çeşitli programlama görevlerinde yüksek performans elde etmek olmuştur. Makalede, programlama dillerinin anlaşılması ve düzenlenmesi için makine öğreniminin önemi vurgulanmıştır. Program kodunun doğal dilden farklı olduğu ve bu farklılığın makine öğrenimi modellerinde dikkate alınması gerektiği belirtilmiştir. *PLUR* çerçevesi, 16 farklı görevi içermekte ve bu görevler için tek bir model mimarisi kullanarak neredeyse tüm görevlerde mevcut en iyi sonuçlara ulaşılmıştır. *PLUR*'un başarısı, çeşitli modelleme seçeneklerinin ve görev tiplerinin önemini göstermiştir ve çok görevli öğrenme yaklaşımını desteklemiştir (Chen ve diğ, 2021).

Lajkó ve diğerlerinin çalışmasında JavaScript program onarımı için Önceden Eğitilmiş Üretici Dönüştürücü (Generative Pre-trained Transformer- GPT-2) modelinin kullanımı araştırılmış ve otomatik program onarımı araçlarının geliştirilmesine odaklanılmıştır. Araştırmacılar, GPT-2'yi 16.863 JavaScript kod parçacığı üzerinde eğitmişler ve böylece modelin programlama dilinin doğasını öğrenmesi sağlanmıştır. GPT-2, çoğu durumda söz dizimi olarak doğru kaynak kodu yazabilmiş ancak bazı durumlarda etkili hata düzeltmeleri yapamamıştır. Yine de çoğu durumda doğru

düzeltmeler üretilmiş ve toplamda %17,25 doğruluk oranına ulaşılmıştır. Araştırmacılar, GitHub'dan toplanan 18.736 hata düzeltme gönderimini eğitim verisi olarak kullanmış ve modeli bu şekilde eğitip değerlendirmişlerdir. GPT-2'nin doğru JavaScript kodunu tahmin etme konusunda başarılı olduğu gözlemlenmiş ancak daha büyük ve yeni model varyantlarının gelecekte daha iyi sonuçlar elde edebileceği düşünülmüştür. Araştırma veri seti, modelin kaynak kodu ve eğitilmiş modeller GitHub üzerinden paylaşılmıştır. Bu çalışma, GPT-2'nin doğal dil işleme için tasarlanmış olmasına rağmen kod yazma ve program onarımı öğrenme potansiyeline sahip olduğunu göstermiş ve gelecekte daha büyük modellerin kullanılmasının sonuçları iyileştirebileceği öne sürülmüştür (Lajkó ve diğ, 2022).

Otomatik kod tamamlama ve otomatik kod onarımı gibi amaçlar ile GitHub'dan alınan açık kaynak kod üzerine ince ayar yapılarak *Codex* adlı bir GPT dil modeli geliştirilmiştir. *Codex*, özellikle Python kodu yazabilme yeteneği ile dikkat çekmekte ve *GitHub Copilot*'u desteklemektedir. Araştırmacılar, dil modellerinin işlevsel kod yazma potansiyelini anlamak için Codex'i geliştirmişlerdir. Bu model, Python kodu üzerine eğitilmiş olup docstring'lerden bağımsız Python fonksiyonları üretebilmek için tasarlanmıştır. Modelin değerlendirilmesi için dil anlayışı, algoritmalar ve basit matematik üzerine odaklanan 164 orijinal programlama problemi içeren *HumanEval* adlı bir veri seti oluşturulmuştur. Değerlendirme sürecinde, BLEU puanları gibi geleneksel eşleşme tabanlı metrikler yerine işlevsel doğruluk üzerinde duran "pass@k" metriği benimsenmiştir. Bu yaklaşım, geliştiricilerin kodu nasıl değerlendirdiğiyle uyumlu olarak geliştirilmiştir. Kodun bir dizi birim testini geçip geçmediği esas alınmıştır. *Codex,* problemlerin %28,8'ini kendi başına çözerken problem başına 100 örnek üreterek bu oranı %70,2'ye kadar çıkarmıştır. Bu performans, benzer görevlerde GPT-3 ve GPT-J'yi geride bırakmaktadır. *Codex*, GPT-3 modelinden ince ayar yapılarak geliştirilmiş ve bu yaklaşım daha hızlı bir yakınsama sağlamıştır. Eğitim sürecinde, GitHub'daki 54 milyon kamu yazılım deposundan 159 GB'lık bir veri seti kapsama alınmıştır. Çalışmada, farklı örnek sayıları (k) için optimal örnekleme sıcaklıkları belirlenmiştir. Daha yüksek sıcaklıkların daha çeşitli örnekler ürettiği ve doğru çözümler üretmede faydalı olduğu bulunmuştur. *Codex, GPT-Neo* ve *GPT-*J gibi diğer modellerin yanı sıra bir kod otomatik tamamlama sistemi olan Tabnine'nin en büyük ücretsiz modelinden de daha üstün performans göstermiştir. *Codex*'e ek olarak yürütülen ince ayarla, rekabetçi programlama web sitelerinden ve

sürekli entegrasyon içeren açık kaynak projelerden elde edilen problemleri içeren *Codex-S* oluşturulmuştur. *Codex-S*, farklı model boyutları boyunca tutarlı kazanımlar göstermiştir. Ayrıca, koddan docstring üretmek için eğitilmiş başka bir varyant olan *Codex*-D de geliştirilmiştir. Bu modelin geçiş oranı *Codex-S*'e göre daha düşük olmakla birlikte üretilen kodun arkasındaki niyeti açıklamada etkili olmuştur. Eğitim için geniş bir kod miktarına ihtiyaç duyan *Codex*, iyi eğitilmiş bir insan programcıya kıyasla düşük performans gösterebileceği belirtilmiştir. Kullanıcı niyetleriyle uyumlu olmayan kod üretebilen *Codex*, eğitim verilerindeki önyargıları yansıtabilmektedir. Araştırmacılar tarafından, *Codex*'in programcıların verimliliğini artırarak yazılım üretim maliyetlerini düşürebileceği ancak bu durumun iş gücü piyasasına etkilerinin henüz tam olarak anlaşılamadığı belirtilmiştir (Prenner ve diğ, 2022).

Ahmed ve diğerlerinin çalışmasında, söz dizimi hatalarını düzeltme sürecini otomatikleştirmeyi amaçlayan bir makine öğrenme tabanlı araç olan *SYNSHINE* tanıtılmıştır. *SYNSHIN*E, derleyici teşhislerini kullanarak ve büyük bir sinirsel modelden yararlanarak bu sorunu ele almıştır. Ayrıca, çoklu etiket sınıflandırmasına dayalı bir yaklaşım kullanılarak tamir edilecek kodun (onarılmış) çıktısı üretilmiştir. Yazarlar, *SYNSHINE*'ı açık kaynaklı ve ücretsiz bir Visual Studio Code (VSCode) sürümüne entegre etmişlerdir. Sunulan sonuçlara göre, SYNSHINE'ın genel başarım oranı %74,89 olarak belirlenmiştir. *SYNSHINE*'ın, 1-100 token aralığındaki programlarda %82,28, 101-200 token aralığında %78,47, 201-300 token aralığında %72,28, ve 901-1000 token aralığında %55 başarı oranına sahip olduğu belirtilmiştir. Bu sonuçlar, *SYNSHINE*'ın farklı uzunluktaki programlarda da etkili bir şekilde çalıştığını göstermektedir (Ahmed ve diğ, 2022).

Literatür bölümünde, bu alandaki çalışmaların kapsamlı bir değerlendirmesi sunulmuştur. Bu değerlendirme, kural tabanlı sistemlerden başlayarak makine öğrenimi ve derin öğrenme yöntemlerine kadar uzanan geniş bir yelpazeyi kapsamaktadır. Ele alınan her bir yaklaşımın avantajları ve sınırlamaları detaylı bir şekilde incelenmiş böylece programlama dillerindeki söz dizimi hatalarını otomatik olarak tespit etme ve düzeltme konusundaki çözümlerin derinlemesine anlaşılması sağlanmıştır. Bu tarama, tez çalışmasının ilerleyen bölümlerinde geliştirilen metodolojinin temelini oluşturmuş ve çalışmanın yönünü şekillendirmede kritik bir rol oynamıştır. Bu bölüm, alanın mevcut durumunu özetlemekte ve araştırmanın ilerleyen aşamaları için sağlam bir temel oluşturmaktadır.

3. PROGRAMLAMA DİLLERİNDE SÖZ DİZİMİ

Söz dizimi (syntax), bir dilin cümle yapısını ve kelime sıralamasını düzenleyen kurallar ve ilkeler bütünüdür. Bu terim, genellikle yazılı ve konuşulan dillerde kullanılan gramer yapısını ifade etmek için kullanılsa da programlama dillerinde de önemli bir rol oynamaktadır.

Programlama dilinin söz dizimi, bir programlama dilinin nasıl yazılması gerektiğini tanımlayan kurallar bütünüdür. Bu kurallar, bir programın düzgün çalışabilmesi için kodun nasıl düzenleneceği ve yapılandırılacağı hakkında yönergeler sağlamaktadır. Bu kurallar arasında değişken tanımlama, bu tanımlamaların nasıl yapılacağını ve değişkenlerin nasıl isimlendirileceğini belirlemektedir. Fonksiyon tanımlama, bir işlevi gerçekleştirmek için gerekli kod bloklarının nasıl gruplandırılacağını ve bu işlevlere nasıl erişileceğini kapsamaktadır. Döngü ve kontrol yapıları, *if-else, for, while* gibi yapılarla programın akışını kontrol etme imkanı vermektedir. Bu sayede kodun belirli koşullar altında farklı yollar izlemesi sağlanmaktadır. Sınıf ve nesne tanımlama, özellikle nesne yönelimli programlama dillerinde önem taşır. Veri yapıları ile bu yapıların nasıl kullanılacağını tanımlanmaktadır. Bu temel yapılar, bir programlama dilinin temelini oluşturmaktadır. Dilin etkili bir şekilde kullanılabilmesi için bu söz dizimi kurallarının doğru bir şekilde anlaşılması ve uygulanması gereklidir. Örneğin; Python, JavaScript veya C++ gibi farklı programlama dillerinin kendine özgü söz dizimi kuralları vardır. Bu kurallara uymayan kod parçaları, genellikle söz dizimi hataları (syntax errors) olarak adlandırılmaktadır. Programın düzgün çalışmasını engellemektedir.

Her dilin söz dizimi, o dilin okunabilirliğini, kolay öğrenilebilirliğini ve hata yapma olasılığını etkilemektedir. Bu yüzden programlama dillerini öğrenirken söz dizimini doğru şekilde anlamak ve uygulamak çok önemlidir.

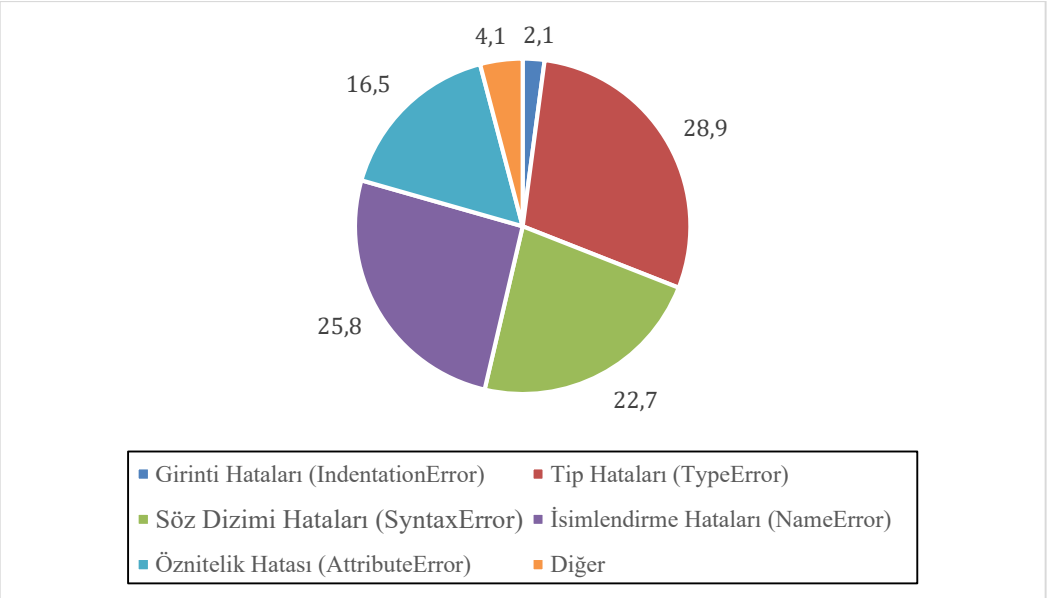
Programlama dillerinde söz dizimi, yazılım geliştirmede temel bir unsurdur. Her programlama dili, kodun nasıl yazılacağını ve yapılandırılacağını belirleyen kendi

kurallarına ve yapılarına sahiptir. Bu kurallar, dilin söz dizimini oluşturup programcılar tarafından takip edilmesi gerekmektedir.

3.1 Genel Bakış

Söz dizimi hataları, programlama sırasında sıkça karşılaşılan sorunlardandır. Bunlar, programcının dilin kurallarını yanlış anlaması veya ihlal etmesi sonucunda ortaya çıkmaktadır. Örneğin bir değişkenin yanlış adlandırılması, noktalı virgülün unutulması veya parantezlerin yanlış kullanılması gibi basit hatalar programın beklenen şekilde çalışmamasına neden olmaktadır. Bu tür hatalar genellikle programın derlenme aşamasında ortaya çıkmaktadır. Çoğu programlama ortamı, hata yapılan yerleri işaretleyerek programcıya düzeltme yapma imkanı sunmaktadır. Ancak bazı söz dizimi hataları, programın çalışma zamanında ortaya çıkmakta ve bu durumda hata ayıklama daha zor olmaktadır. Özellikle büyük ve karmaşık programlarda, söz dizimi hatalarının bulunması ve düzeltilmesi zaman alıcı ve zorlayıcı olmaktadır. Programcılar için en iyi uygulama, kod yazarken dikkatli olmak ve hataları önlemek için dilin kurallarını iyi anlamaktır. Ayrıca modern geliştirme araçları ve ortamları, söz dizimi hatalarını tespit etmekte ve düzeltmekte programcılara yardımcı olmaktadır. Söz dizimi hataları programlama sürecinin kaçınılmaz bir parçasıdır. Ancak dikkatli bir yaklaşımla minimize edilmektedirler.

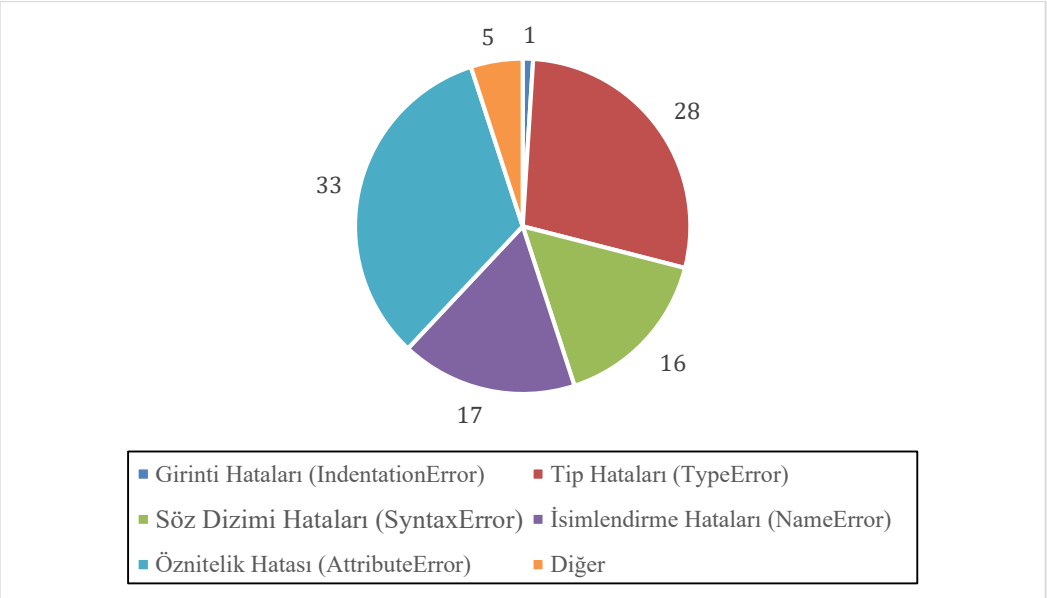
Kelley'in çalışmasında söz dizimi hatalarının MIT'deki başlangıç seviyesi kodlayıcılar arasında yaygın olduğu ve birçok durumda, mevcut hata mesajlarına dayanarak kodlarını hata ayıklamakta zorlandıkları belirtilmektedir (Kelley,2018). Bu durum, sıklıkla söz dizimsel olarak geçersiz kod gönderimlerine yol açmaktadır. Kelley'in çalışmasında öğrenci gönderimleri farklı kategorilerde sınıflandırılmıştır. Şekil 3.1'de birinci sınıfa ait, öğrencilerin tasarım laboratuvarı derslerinde yazmış oldukları programlar, hata dağılımları verilmiştir.



Şekil 3.1 : Tasarım laboratuvarı dersi öğrenci hataları dağılımı (Kelley, 2018).

Öğrenci gönderimlerinde yapılan hata analizine göre, %28,9'luk bir oranla en yaygın hata türü 'TypeError' (Tür Hatası) olarak belirlenmiştir. Bu, veri türleri arasındaki uyumsuzluklardan kaynaklanan bir hatadır. İkinci sırada, %25,8 ile 'NameError' (İsimlendirme Hatası) bulunmaktadır. Bu hata, genellikle tanımlanmamış bir değişken veya fonksiyon adı kullanıldığında ortaya çıkmaktadır. Ancak dikkat çekici olan öğrenci gönderimlerindeki hataların %22,7'sinin 'SyntaxError' (Söz Dizimi Hatası) olmasıdır. Bu oran, öğrencilerin programlama dillerinin söz dizimini anlamada ve uygulamada sıklıkla zorluk yaşadıklarını göstermektedir. Söz dizimi hataları, programın baştan sona doğru yazılmamasından kaynaklanmaktadır. Bu hatalar, kodun derlenmesi veya yorumlanması sırasında fark edilmektedir.

Şekil 3.2, öğrencilerin yazılım laboratuvarı dersi kapsamında gerçekleştirdikleri kod gönderimlerinde ortaya çıkan hataların ayrıntılı bir dağılımını sunmaktadır. Bu grafikte her bir hata türünün öğrenci gönderimleri içindeki yüzdesel oranı net bir şekilde gösterilmiş böylece hangi hata türlerinin öğrenciler arasında daha sık rastlandığının anlaşılmasına olanak sağlamaktadır. 'TypeError' (Tür Hatası), 'NameError' (İsim Hatası) ve 'SyntaxError' (Söz Dizimi Hatası) gibi hataların, öğrenci kodlarında ne kadar sık meydana geldiği bu şekilde görselleştirilmiştir. Bu grafik, eğitimcilerin ve öğrencilerin hangi programlama konularına daha fazla odaklanmaları gerektiği konusunda değerli bilgiler sunmaktadır.



Şekil 3.2 : Yazılım laboratuvarı dersi öğrenci hataları dağılımı (Kelley, 2018).

Şekil 3.2'ye göre en yüksek yüzdeye sahip hata türü %33 ile 'AttributeError' (Öznitelik Hatası) olarak belirlenmiştir. İkinci en yüksek oran %28 ile 'TypeError' (Tür Hatası), üçüncü sırada ise %17 ile 'NameError' (İsimlendirme Hatası) yer almaktadır. 'SyntaxError' (Söz dizimi Hatası) %16 oranında ve bu da öğrencilerin karşılaştığı hatalar arasında önemli bir paya sahip olduğunu göstermektedir. 'IndentationError' (Girintileme Hatası) %1 ile daha düşük bir oranı temsil ederken Diğer kategorisi de %5 ile diğer hata türlerinin toplamını kapsamaktadır.

Ayrıca Pritchard tarafından CS Circles adlı çevrimiçi Python öğrenme ortamında yapılan bir çalışmada, birkaç hatanın çok daha yaygın olduğu tespit edilmiştir. Özellikle, 'Syntax Error: Invalid Syntax' (%28) ve 'NameError: name is not defined' (%15) hataları en yaygın hatalar arasındadır (Pritchard, 2015).

Kelley'in çalışması ve benzer araştırmalar, başlangıç seviyesindeki kodlayıcıların söz dizimi hataları ile karşılaştığında anlamlı hata mesajları olmadan düzeltme yapmakta zorlandıklarını göstermektedir. Bu analiz, başlangıç seviyesi öğrencilerin programlama dillerinin söz dizimi konusunda sıkça hata yaptıklarını ve bu hataların düzeltilmesinde güçlük çektiklerini vurgulamaktadır.

3.1.1 Python programlama dilindeki söz dizimi hataları

Python programlama dilindeki söz dizimi hataları, kodun doğru bir şekilde yazılmamasından kaynaklanmaktadır. Genellikle Python yorumlayıcısı tarafından

farklı hata mesajları ile belirtilmektedir. Bunların arasında en yaygın olanlarından biri girinti hatalarıdır (IndentationError). Python'da girintileme, kod bloklarının başlangıcı ve bitişini belirlemektedir. Yanlış girintileme yapıldığında if, for ve while gibi yapıların içindeki kodlar düzgün hizalanmadığında bu hata meydana gelmektedir.

Şekil 3.3, Python programlama dilinde girinti kullanımı ile ilgili bir hata durumunu göstermektedir. Python, girintilere dayalı bir yapıya sahiptir. Blokların başlangıcını ve sonunu belirlemek için girintileri kullanmaktadır. Şekil 3.3'te, 'ornekFonksiyon' adında bir fonksiyon tanımlanmış ancak 'print' fonksiyonu fonksiyon tanımının aynı girinti düzeyine sahip değildir. Bu da Python yorumlayıcısının bir 'IndentationError' (Girinti Hatası) oluşturmasına neden olmaktadır. Çünkü 'print' ifadesi beklenen girinti düzeyinde değildir. Doğru bir şekilde, 'print' ifadesi fonksiyon tanımı bloğunun içinde bir tab boşluk veya dört boşluk girintiyle yazılması gerekmektedir.

```
1 def ornekFonksiyon():
2 print("Bu satır hatalı girintilendi!")
```

Şekil 3.3 : Girintileme hatası içeren örnek Python kodu.

Şekil 3.4, bir Python betiğinin 'ornek1.py' çalıştırılmasının ardından bir 'IndentationError' hatası ile sonuçlandığını gösteren bir hata mesajı çıktısını içermektedir. Bu hata, Python yorumlayıcısının beklediği girintili blok yapısının ihlal edildiğini işaret etmektedir. Python'da, fonksiyonlar, döngüler, sınıflar ve koşullu ifadeler gibi kod blokları girinti kullanılarak oluşturulmaktadır.

```
>>> %Run ornek1.py
  Traceback (most recent call last):
    File "C:\Program Files (x86)\Thonny\lib\ast.py", line 35, in parse
      return compile(source, filename, mode, PyCF_ONLY_AST)
    File "C:\Users\sena\Desktop\ornek1.py", line 2
      print("Bu satır hatalı girintilendi!")
          ^
  IndentationError: expected an indented block
```

Şekil 3.4 : Python yorumlayıcısında girintileme hatası çıktısı örneği.

Şekil 3.4'te, 'ornek1.py' dosyasının 2. satırında bu gerekliliğe uyulmamış ve girinti yapılmadan bir 'print' fonksiyonu çağrısı yapıldığı görülmektedir. Python yorumlayıcısı, bu yapısal uyumsuzluğu saptayarak bir hata mesajı ile programcıyı uyarmaktadır. Bu tür hatalar, genellikle yazım hataları veya düzenleme sırasında oluşmaktadır. Düzeltmek için ilgili satırın uygun şekilde girintilenmesi gerekmektedir.

Bu tür hata mesajları, öğrencilerin veya geliştiricilerin kodları üzerinde hata ayıklarken karşılaştıkları yaygın sorunlardan birini temsil etmektedir.

Bir diğer yaygın hata türü söz dizimi hatalarıdır (SyntaxError). Bu hata, Python'ın söz dizimi kurallarının doğru şekilde takip edilmemesi sonucu oluşmaktadır. Örneğin, bir ifadenin sonuna yanlışlıkla noktalı virgül konulması veya parantezlerin düzgün kapanmaması gibi durumlar SyntaxError'a neden olmaktadır. Şekil 3.5, bir 'SyntaxError' hatası içeren Python kodunun görsel bir temsilidir.

```
1  def ornekFonksiyon():
2      print("Bu bir test mesajı"
```

Şekil 3.5 : Söz dizimi hatası içeren örnek Python kodu.

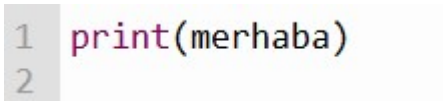
Python'da, her fonksiyon tanımının ardından gelen kod bloğu girintili olmalıdır. Tüm parantezlerin düzgün bir şekilde kapatılması gerekmektedir. Görselde gösterilen 'ornekFonksiyon' fonksiyonunun gövdesinde 'print' ifadesi girintili olarak doğru yerleştirilmiş ancak ifade sonunda parantez eksik olduğu için söz dizimi hatası oluşmuştur. Python, bu tür hataları derleme zamanında tespit etmektedir. Hatanın türünü ve yerini belirten bir hata mesajı vermektedir. Şekil 3.6, bir Python betiğinin yürütülmesi sırasında karşılaşılan bir 'SyntaxError' hatasını betimlemektedir.

```
>>> %Run ornek1.py
  Traceback (most recent call last):
    File "C:\Users\sena\Desktop\ornek1.py", line 2
      print("Bu bir test mesajı"
                                ^
  SyntaxError: unexpected EOF while parsing
```

Şekil 3.6 : Python yorumlayıcısında söz dizimi hatası çıktısı örneği.

Şekil 3.6'daki hata, betiğin yürütülmesi sırasında beklenmeyen bir dosya sonu (EOF) ile karşılaşıldığında ortaya çıkmaktadır. Bu durum genellikle bir söz dizimi unsuru eksik olduğunda meydana gelmektedir. İlgili kod parçasında, 'print' fonksiyonunun çağrısının sonunda bir parantezin eksik olması Python yorumlayıcısının ifadeyi tam olarak anlamlandıramamasına ve betiğin sonunu beklenmedik bir yerde karşılamasına sebep olmuştur. Bu hatanın düzeltilmesi için 'print' ifadesinin kapatılması gereken parantezin eklenmesi gerekmektedir. Bu tür hatalar, programın beklenen çıktıyı üretmesini engellemektedir. Derleme sırasında giderilmesi gerekmektedir.

İsimlendirme hatası (NameError) ise tanımlanmamış bir değişkenin veya fonksiyonun kullanılmaya çalışılmasında ortaya çıkmaktadır. Python, belirtilen adı tanımadığında bu tür bir hata vermektedir. Bu genellikle yazım hatası veya değişkenin tanımlanmadan kullanılmaya çalışılması sonucu ortaya çıkmaktadır. Şekil 3.7, bir Python kod parçasını göstermektedir. Burada bir isimlendirme hatası (NameError) yer almaktadır.



Şekil 3.7 : İsimlendirme hatası içeren örnek Python kodu.

Python'da 'print' fonksiyonu, içine yazılan ifadeyi çıktı olarak vermektedir. Eğer ifade bir metin ise tırnak işaretleri içinde olması gerekmektedir. Şekil 3.7'de, 'print' fonksiyonunun içinde tırnak işaretleri olmadan yazılmış 'merhaba' ifadesi vardır. Python bu ifadeyi bir değişken veya fonksiyon ismi olarak algılamaktadır. Şekil 3.8, Şekil 3.7'de yer alan Python kodu çalıştırıldığında yorumlayıcının ürettiği mesajı göstermektedir.

```
>>> %Run ornek1.py
  Traceback (most recent call last):
    File "C:\Users\sena\Desktop\ornek1.py", line 1, in <module>
      print(merhaba)
  NameError: name 'merhaba' is not defined
```

Şekil 3.8 : Python yorumlayıcısında isimlendirme hatası çıktısı örneği.

Şekil 3.8'de, 'print(merhaba)' ifadesi, 'merhaba' isminin tanımlı olup olmadığını kontrol etmeksizin kullanılmıştır. 'merhaba' ifadesinin önceden bir değişken olarak tanımlanmadığı durumlarda, Python 'NameError: name 'merhaba' is not defined' şeklinde bir hata mesajı üretmektedir. Bu, yorumlayıcının beklediği bir ismin tanımlı olmaması durumunda ortaya çıkan standart bir hata mesajıdır. Bu tür hatalar genellikle metinlerin yanlışlıkla tırnak işaretleri olmadan yazılması sonucu oluşmaktadır. Düzeltmek için metnin tırnak işaretleri içine alınması gerekmektedir.

Tip hatası (TypeError), Python'da sıkça karşılaşılan bir başka hata türüdür. Bu hata genellikle bir işlem veya fonksiyon için uygun olmayan bir veri tipinin kullanılması durumunda meydana gelmektedir. Örneğin, bir sayı ile bir string'i toplamaya çalışmak veya yanlış veri tipine sahip argümanların bir fonksiyona geçirilmesi durumunda bu

hata görülmektedir. Şekil 3.9, Python'da farklı veri tiplerinin bir araya getirilmesinden kaynaklanan bir tip hatasını potansiyel olarak sergileyen bir kod parçasını göstermektedir.

```
1  sayi = 5
2  metin = "5"
3  toplam = sayi + metin
```

Şekil 3.9 : Tip hatası içeren örnek Python kodu.

Şekil 3.9'daki yer alan kodda sayi adında bir değişken tam sayı değeri (int) 5 olarak tanımlanmış ve metin adında başka bir değişken ise string (str) türünde '5' olarak tanımlanmıştır. Üçüncü satırda toplam adında bir değişken, sayi ve metin değişkenlerinin toplamını hesaplamak üzere tanımlanmıştır. Ancak, Python'da tam sayılarla metinlerin doğrudan toplanması desteklenmez Bu işlem bir TypeError hatasına yol açmaktadır. Şekil 3.10'da ise Şekil 3.9'da verilen kodun çalıştırılması sonucunda gerçekleşen hata mesajı gösterilmektedir.

```
>>> %Run ornek1.py
  Traceback (most recent call last):
    File "C:\Users\sena\Desktop\ornek1.py", line 3, in <module>
      toplam = sayi + metin
  TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Şekil 3.10 : Python yorumlayıcısında tip hatası çıktısı örneği.

Şekil 3.10'da yer alan çıktıda, TypeError: unsupported operand type(s) for +: 'int' and 'str' mesajı, + operatörünün int ve str türleri arasında desteklenmediğini belirtmektedir. Bu hata, işlemciye (yorumlayıcıya) iki uyumsuz veri tipini işlemek için geçersiz bir komut verildiğinde ortaya çıkmaktadır. Python'da bu hatayı düzeltmek için ya iki değişkenin tipini birbirine dönüştürmek (örneğin, metni tamsayıya çevirerek veya tamsayıyı metne dönüştürerek) ya da bu tür bir işlemi gerçekleştirmekten kaçınmak gerekmektedir.

Değer hatası (ValueError) ise bir işlemin veya fonksiyonun beklenen veri tipi ile uyumlu olmasına rağmen doğru olmayan bir değer alması durumunda ortaya çıkmaktadır. Bu, genellikle bir fonksiyonun yanlış formatta veri ile çağrılması sonucu oluşmaktadır. Örneğin sayısal bir değer beklenen bir yere metinsel bir veri girilmesi bu hatayı tetiklemektedir.

Şekil 3.11, Python'da bir değer dönüştürme işlemi sırasında ‘ValueError’ hatasına yol açabilecek bir kod örneğini göstermektedir.

```
1 sayi = "beş"
2 int_sayi = int(sayi)
```

Şekil 3.11 : Değer hatası içeren örnek Python kodu.

Şekil 3.11’deki kodda, ‘bes’ değerini bir string olarak ‘sayi’ değişkenine atar ve ardından ‘int’ fonksiyonunu kullanarak bu string'i bir tamsayıya (‘int’) dönüştürmeye çalışmaktadır. Ancak ‘bes’ rakamsal bir değer yerine bir kelime olduğu için ‘int’ fonksiyonu bu dönüşümü gerçekleştiremez. Bu durum bir ‘ValueError’ hatasına neden olmaktadır. Python'da, ‘int’ fonksiyonu yalnızca rakamlardan oluşan string'leri tam sayıya dönüştürebilmektedir. Şekil 3.12, Şekil 3.11'de gösterilen kodun çalıştırılması sonucu üretilen hata mesajını göstermektedir.

```
>>> %Run ornek1.py
  Traceback (most recent call last):
    File "C:\Users\sena\Desktop\ornek1.py", line 2, in <module>
      int_sayi = int(sayi)
  ValueError: invalid literal for int() with base 10: 'beş'
```

Şekil 3.12 : Python yorumlayıcısında değer hatası çıktısı örneği.

Şekil 3.12’deki hata mesajı, “ValueError: invalid literal for int() with base 10: ‘beş’” şeklindedir ve ‘int’ fonksiyonunun ondalık taban için geçersiz bir metin (literal) ile karşılaştığını belirtmektedir. Bu, ‘int’ fonksiyonunun beklediği sayısal bir değer yerine geçersiz bir metinsel değerle karşılaşmasından kaynaklanmaktadır. Örneğin, bu durumda hatayı düzeltmek için ‘sayi’ değişkeninin değerini sayısal bir string ile değiştirmek veya sayısal olmayan bir string'i dönüştürmeye çalışmadan önce uygun bir kontrol yapmak gerekmektedir.

Öznitelik hatası (AttributeError), bir nesnenin var olmayan bir özelliğine veya metoduna erişilmeye çalışıldığında meydana gelmektedir. Bu hata genellikle bir nesnenin yanlış anlaşılması veya yanlış bir şekilde kullanılması sonucu oluşmaktadır. Örneğin, bir liste nesnesi üzerinde olmayan bir metodu çağırmak AttributeError'a neden olmaktadır. Şekil 3.13, bir Python listesine yönelik bir metot çağrısını içeren kod örneğini göstermektedir.

```python
liste = [1, 2, 3]
liste.append(4)
print(liste.yazdir())
```

Şekil 3.13 : Özellik hatası içeren örnek Python kodu.

Şekil 3.13'te yer alan kodda liste adında bir liste tanımlanmış ve append metodu ile bu listeye bir eleman eklenmiştir. Ancak üçüncü satırda liste nesnesi için yazdir adında bir metot çağrısı yapılmaktadır. Python'da liste nesnelerinin yazdir gibi bir metodu bulunmadığından bu kod parçası bir AttributeError hatası üretmektedir. Şekil 3.14, Şekil 3.13'te verilen kodun derlenmesi sonucunda oluşan hata mesajını göstermektedir.

```
>>> %Run ornek1.py
  Traceback (most recent call last):
    File "C:\Users\sena\Desktop\ornek1.py", line 3, in <module>
      print(liste.yazdir())
  AttributeError: 'list' object has no attribute 'yazdir'
```

Şekil 3.14 : Python yorumlayıcısında özellik hatası çıktısı örneği.

Hata mesajı "AttributeError: 'list' object has no attribute 'yazdir'" şeklinde olup bu liste nesnesinin 'yazdir' adında bir özelliğe (attribute) veya metoda sahip olmadığını belirtmektedir. Bu tür bir hata genellikle nesnenin sahip olmadığı bir metot veya özellik çağrıldığında meydana gelmektedir. Python'da listelerin içeriğini görüntülemek için doğrudan print fonksiyonu kullanılabilir.

Anahtar hatası (KeyError), sözlük (dictionary) tipi veri yapılarıyla çalışırken, sözlükte bulunmayan bir anahtara erişilmeye çalışıldığında ortaya çıkmaktadır. Bu hata, genellikle veri yapısının yanlış anlaşılmasından kaynaklanmaktadır.

İndeks hatası (IndexError), listeler gibi dizi türleriyle çalışırken var olmayan bir indekse erişilmeye çalışıldığında ortaya çıkan bir hatadır. Bu genellikle, dizi boyutlarının yanlış hesaplanması sonucu meydana gelmektedir. Şekil 3.15, bir Python sözlüğünün tanımlanması ve bu sözlükten bir anahtar kullanarak değer erişimi denemesini içeren bir kod örneğini göstermektedir.

```
1 sozluk = {"elma": 3, "muz": 5}
2 print(sozluk["portakal"])
```

Şekil 3.15 : Anahtar hatası içeren örnek Python kodu.

Şekil 3.15'te, "sozluk" adında bir sözlük tanımlanmış ve içerisine "elma" ve "muz" anahtarları ile ilişkili değerler yerleştirilmiştir. İkinci satırda, sözlükten "portakal" anahtarına erişmeye çalışılmaktadır. Ancak, "portakal" anahtarı sözlükte tanımlı olmadığı için bu kod bir KeyError hatası üretmektedir. Şekil 3.16, Şekil 3.15'teki kodun yürütülmesi sonucu oluşan hata mesajını göstermektedir.

```
>>> %Run ornek1.py
  Traceback (most recent call last):
    File "C:\Users\sena\Desktop\ornek1.py", line 2, in <module>
      print(sozluk["portakal"])
  KeyError: 'portakal'
```

Şekil 3.16 : Python yorumlayıcısında anahtar hatası çıktısı örneği.

Hata mesajı "KeyError: 'portakal'" şeklinde olup "portakal" anahtarının sözlükte bulunmadığını belirtmektedir. Python sözlüklerinde, bir anahtarın değerine erişmek istenildiğinde eğer o anahtar sözlükte mevcut değilse bir KeyError hatası oluşmaktadır. Bu tür hatalar, genellikle sözlükten bir eleman erişimi sırasında anahtarın doğru bir şekilde sözlüğe eklenip eklenmediğinin kontrol edilmemesinden kaynaklanmaktadır. Bu durumda hatayı önlemek için anahtarın sözlükte olup olmadığını kontrol eden bir koşul ifadesi kullanılmaktadır. Diğer bir seçenek ise get metodu ile anahtarın varlığı kontrol edilirken varsayılan bir değere dönüştürmektir.

İçe aktarma hatası (ImportError) ve modül bulunamadı hatası (ModuleNotFoundError), modüllerle ilgili hatalardır. ImportError, bir modülün veya modül içindeki bir özelliğin içe aktarılması sırasında meydana gelirken ModuleNotFoundError mevcut olmayan bir modülün import edilmeye çalışılması durumunda ortaya çıkmaktadır. Her iki hata da genellikle yanlış modül adı kullanımı veya yüklenmemiş modüller nedeniyle oluşmaktadır. Şekil 3.17, Python dilinde math modülünden belirli bir ismi (bu örnekte toplama) içe aktarma girişimini gösteren bir kod parçasını sunmaktadır.

```
1 from math import toplama
```

Şekil 3.17 : İçe aktarma hatası içeren örnek Python kodu.

Python standart kütüphanesindeki math modülü, matematiksel işlemler için birçok fonksiyon ve sabiti barındırmaktadır. Ancak, math modülü içinde toplama adında bir işlev veya değişken bulunmadığı için bu kod parçası bir ImportError hatası ile sonuçlanmaktadır. Şekil 3.18, Şekil 3.17'deki kodun çalıştırılması sonucunda oluşan hata mesajını göstermektedir.

```
>>> %Run ornek1.py
  Traceback (most recent call last):
    File "C:\Users\sena\Desktop\ornek1.py", line 1, in <module>
      from math import toplama
  ImportError: cannot import name 'toplama' from 'math' (unknown location)
```

Şekil 3.18 : Python yorumlayıcısında içe aktarma hatası çıktısı örneği.

Hata mesajı "ImportError: cannot import name 'toplama' from 'math' (unknown location)" şeklinde olup, math modülünde toplama adında bir ismin bulunamadığını ifade etmektedir. Bu hatanın oluşmasının sebebi, içe aktarmaya çalışılan ismin hedef modülde tanımlı olmamasından kaynaklanmaktadır. Python'da bu tür bir hata, yanlış yazılmış bir modül adı veya modül içinde olmayan bir işlevi/importu içe aktarmaya çalışıldığında meydana gelmektedir. Bu tür hata mesajları, modül ve paket sistemleri konusunda eğitim verilirken, modül içeriğini doğru bir şekilde anlamak ve doğru bir şekilde içe aktarmak için nasıl bir yaklaşım izlenmesi gerektiğini öğretmek amacıyla kullanılmaktadır. Şekil 3.19, Python'da bir modülün içe aktarılması sırasında karşılaşılan bir hata mesajını göstermektedir.

```
1 import yanlisModulAdi
```

Şekil 3.19 : Modül bulunamadı hatası içeren örnek Python kodu.

Bu mesaj, 'ModuleNotFoundError' hatasını içermekte ve Python yorumlayıcısının 'yanlisModulAdi' adında bir modül bulamadığını belirtmektedir. Bu hata genellikle içe aktarılmak istenen modülün isminin yanlış yazılması veya ilgili modülün Python'un modül arama yolunda bulunmaması durumlarında ortaya çıkmaktadır. Şekil 3.20, yukarıda belirtilen hataya yol açan Python kodunu göstermektedir.

```
>>> %Run ornek1.py
  Traceback (most recent call last):
    File "C:\Users\sena\Desktop\ornek1.py", line 1, in <module>
      import yanlisModulAdi
  ModuleNotFoundError: No module named 'yanlisModulAdi'
```

Şekil 3.20 : Python yorumlayıcısında modül bulunamadı hatası çıktısı örneği.

Kodda import yanlisModulAdi ifadesi görülmekte olup bu ifade 'yanlisModulAdi' adında bir modülün içe aktarılmasını denemektedir. Ancak bu isimde bir modül Python yükleme yolu üzerinde mevcut değilse 'ModuleNotFoundError' hatası alınmaktadır. Bu tür hatalar, genellikle geliştirme ortamı kurulumu sırasında eksiklikler veya yazım hataları nedeniyle meydana gelmektedir.

Bu hatalar, Python programlama sürecinin ayrılmaz bir parçasıdır. Her programcının karşılaşabileceği durumlardır. Bu hataların anlaşılması ve düzeltilmesi, programlama becerilerinin gelişmesine katkıda bulunmaktadır.

3.1.2 Söz dizimi hatalarının temel sebepleri

Söz dizimi hataları, programlama sürecinde sıklıkla karşılaşılan engellerden biridir. Birkaç temel nedeni bulunmaktadır. İlk olarak her programlama dilinin kendine özgü kuralları ve yapıları vardır. Bu da programcının dilin söz dizimi kurallarını tam olarak anlaması ve uygulaması gerektiği anlamına gelmektedir. Yeni başlayanlar veya farklı bir dilden geçiş yapan programcılar için bu kuralların öğrenilmesi ve hatasız bir şekilde uygulanması zaman alabilmektedir. Özellikle Python gibi girintilemeye duyarlı dillerde, kodun doğru bir şekilde girintilenmesi ve yapısal olarak doğru biçimde düzenlenmesi gerekmektedir. Eğer bir blok yanlış girintilendirilirse veya ifade doğru yapılandırılmazsa Python yorumlayıcısı ‘IndentationError’ veya ‘SyntaxError’ ile yanıt vermektedir.

İkinci bir neden, dikkatsizlik ve yazım hatalarıdır. Bir programcı tırnak işaretlerini, parantezleri veya diğer önemli karakterleri eksik bırakabilmekte veya yanlış yerleştirebilmektedir. Bu tür hatalar, özellikle uzun ve karmaşık kod bloklarında yaygın olarak meydana gelmektedir. Kodun bir kısmının üzerinden defalarca geçmek gerektiğinde küçük detayların gözden kaçması kolaylaşmaktadır. Bu tür yazım hataları genellikle ‘SyntaxError’ ile sonuçlanmaktadır.

Üçüncüsü, kapsam hataları ve yanlış anlamalar da söz dizimi hatalarına yol açmaktadır. Bir değişken veya fonksiyon, tanımlandığı kapsamın dışında kullanılmaya çalışıldığında Python 'NameError' gibi bir hata oluşturmaktadır. Bu, programcının dilin kapsam kurallarını tam olarak anlaması ve uygulaması gerektiğini göstermektedir. Ayrıca programcılar bazen dilin belirli bir özelliğini yanlış anlamakta veya eksik hatırlamaktadır. Bu tür yanlış anlamalar, hatalı kod yazılmasına ve sonuç olarak söz dizimi hatalarına neden olmaktadır.

Dördüncü bir neden olarak, geliştirme ortamının (IDE) eksiklikleri de söz dizimi hatalarına katkıda bulunmaktadır. Gelişmiş bir IDE gerçek zamanlı söz dizimi kontrolü, otomatik tamamlama ve kod önerileri gibi özellikler sunmaktadır. Bu araçlar, programcıya yanlış yazımları ve potansiyel hataları düzeltme konusunda yardımcı olmaktadır. Eğer bir programcı, bu tür özellikleri sunmayan basit bir metin editörü kullanıyorsa hataları fark etme ve düzeltme süreci daha zorlu hale gelmektedir.

Son olarak, uzun ve karmaşık kodlar da hata yapma riskini artırmaktadır. Programcının kodun her bir parçasını dikkatle incelemesi ve anlaması gerekmektedir. Karmaşık kodlar, özellikle hata ayıklama sürecinde, programcının hataları fark etmesini ve izlemesini zorlaştırmaktadır.

3.2 Temel Söz Dizimi Hataları

Söz dizimi hataları, programlama dillerinin katı kurallarının ihlali sonucu ortaya çıkmaktadır. Genellikle programın derlenmesi veya yorumlanması sırasında fark edilmektedir. Örneğin, programcılar bazen karakterleri eksik bırakabilir veya fazladan ekleyebilmektedir. Bu noktalı virgüller, parantezler, tırnak işaretleri ve virgüller gibi semboller için geçerlidir. Python'da, parantezlerin ve tırnak işaretlerinin doğru kullanımı zorunludur. Çünkü bu karakterler, ifadelerin ve stringlerin başlangıç ve bitiş noktalarını belirlemektedir. Bir parantezin kapatılmaması ya da bir string ifadesinin tırnak işaretleriyle doğru şekilde sınırlandırılmaması bir SyntaxErrora yol açabilmektedir.

Bir diğer yaygın hata, girintilemeyle ilgilidir. Python gibi girintilemeye duyarlı dillerde, ifadeler ve bloklar arasındaki hiyerarşik ilişki girinti yoluyla ifade edilmektedir. Eğer kod blokları yanlış girintilenirse yani beklenen girinti düzeyine uymazsa bir IndentationError meydana gelmektedir. Bu tür hatalar, özellikle birden

fazla blok yapısının iç içe kullanıldığı karmaşık programlarda sıkça görülmektedir. Programlama yaparken kullanılan anahtar kelimelerin yanlış kullanımı da hatalara neden olabilmektedir. Python'da for, while, if, else gibi kontrol yapıları belirli bir söz dizimine tabidir. Bu yapıların belirli kurallar çerçevesinde kullanılması gerekmektedir. Yanlış kullanılan bir anahtar kelime veya bu kontrol yapılarının yanlış yerleştirilmesi programın beklenen akışını bozmaktadır. SyntaxError gibi hatalara sebebiyet vermektedir.

Değişken isimlendirme kurallarına uyulmaması da hata kaynaklarından biridir. Python'da değişken adları bir sayı ile başlayamaz. Dilin özel kelimeleri değişken ismi olarak kullanılamaz. Bu tür isimlendirme hataları SyntaxError veya NameError gibi hataların ortaya çıkmasına neden olmaktadır. Benzer şekilde string ifadeler için kullanılan tırnak işaretlerinin açılıp kapatılmaması da hatalı kodların oluşmasına sebep olmaktadır. Python'da bir string ifadenin tek veya çift tırnak işaretleriyle başlatılıp bitirilmesi gerekmektedir. Eşleşmeyen tırnak işaretleri, yorumlayıcının string ifadenin nerede başlayıp bittiğini anlamasını engellemektedir. SyntaxError ile sonuçlanmaktadır.

Operatörlerin yanlış kullanımı gibi durumlar da hatalı kodlara yol açmaktadır. Örneğin, atama operatörü olan ‘=’ ile eşitlik operatörü olan ‘==’ arasındaki farkın göz ardı edilmesi değer atamalarında ve koşullu ifadelerde hataların meydana gelmesine neden olmaktadır. Fonksiyon çağrılarında eksik veya fazladan parametrelerin kullanılması da işlevsel hatalara yol açmakta ve Python'da TypeError gibi hataların oluşmasına sebep olmaktadır.

Son olarak geçersiz söz dizimi kullanımı ve modül ile fonksiyon çağrımı hataları, Python programlarında sıkça karşılaşılan diğer hata türleridir. Programcılar, C dilindeki switch ifadesi gibi Python'da bulunmayan yapıları kullanmaya çalıştıklarında veya gerekli modüller doğru şekilde ithal edilmediğinde SyntaxError, ImportError veya ModuleNotFoundError gibi hatalarla karşılaşmaktadır (Rossum ve Drake, 1995). Bu tür hatalar, özellikle birden fazla modül ve kütüphane kullanılan büyük projelerde kodun düzgün çalışmasını engelleyerek titiz bir hata ayıklama sürecini gerektirmektedir.

3.2.1 Python’da söz dizimi

Python programlama dilinin temel özellikleri ve kullanım kuralları, kodun hem yazılmasını hem de okunmasını kolaylaştırmak üzerine kurulmuştur. Öncelikle, Python'da okunabilirlik ve anlaşılırlık esastır. Kod bloklarının başlangıcı ve sonu parantezler yerine girintileme ile belirlenmektedir. Bu yaklaşım, Python kodunun okunmasını ve anlaşılmasını oldukça kolaylaştırmaktadır. Ayrıca girintileme, ifadelerin ve kod bloklarının gruplandırılmasında temel bir rol oynamaktadır. İyi bir şekilde girintilenmiş kod, programın yapısını ve akışını net bir şekilde yansıtmaktadır. Her yeni kod bloğu, örneğin fonksiyonlar, bir seviye girintilenerek oluşturulmaktadır.

Python'da, yapıların başlatılması kolon (:) karakteri ile yapılmaktadır. If, for, while, def ve class gibi yapılar, bir blok başlatmadan önce kolon ile belirtilmektedir. Bu, takip eden satırın girintili bir blok olduğunu göstermektedir. Ayrıca Python'da string tanımlamaları için tek tırnak, çift tırnak veya üçlü tırnak kullanılabilmektedir. Önemli olan husus, string ifadelerde başlangıç ve bitiş noktalarında aynı tırnak türünün kullanılmasıdır.

Python'da yorumlar önemli bir yere sahiptir. Tek satırlık yorumlar # işareti ile başlamaktadır. Çok satırlı yorumlar için ise üçlü tırnak işaretleri kullanılmaktadır. Python'da bir ifadenin sonunu belirtmek için genellikle yeni bir satıra geçmek yeterlidir. Ek bir karaktere (örneğin, noktalı virgüle) ihtiyaç yoktur. Ancak, bir satır devam ettirilmek istendiğinde, \ karakteri kullanılmaktadır veya ifadeler parantez içine alınmaktadır.

Python, dinamik bir tip sistemine sahiptir. Bu, değişkenlerin belirli bir veri tipine sahip olmak zorunda olmadığı ve programın çalışması sırasında tiplerinin değişebileceği anlamına gelmektedir. Ancak veri tiplerine uygun operatörlerin ve fonksiyonların kullanılması gerekmektedir. Python'da fonksiyonlar ‘def’ anahtar kelimesi ile sınıflar ise 'class' anahtar kelimesi ile tanımlanmaktadır. Her iki yapı da kendi kod bloklarını girintileme yoluyla oluşturmaktadır. Bununla birlikte Python'da belirli anahtar kelimeler özel anlamlara sahiptir. Bu kelimeler değişken ismi olarak kullanılamaz. Örnekler arasında ‘and’, ‘or’, ‘not’, ‘in’, ‘is’, ‘if’, ‘elif’, ‘else’, ‘for’, ‘while’, ‘try’, ‘except’, ‘class’, ‘def’ gibi kelimeler bulunmaktadır (Rossum ve Drake, 1995).

Bu özellikler, Python'ın hem başlangıç seviyesindeki programcılar için erişilebilir hem de ileri seviye programlama ihtiyaçlarına cevap verebilir olmasını sağlamaktadır.

3.3. Girinti Hataları

Python programlama dilinde girintileme hataları oldukça yaygın bir sorundur. Bu hatalar genellikle kodun okunabilirliğini ve düzgün çalışmasını engellemektedir. Bu hataların birkaç farklı nedeni vardır. Bunların her biri kodun işleyişini etkilemektedir.

Birinci ve en sık karşılaşılan hata türü, yanlış girinti seviyesidir. Bu durum, özellikle 'for' döngüleri ve 'if' ifadeleri gibi kontrol yapıları içerisinde sıklıkla karşılaşılmaktadır. Bu tür yapıların içindeki kodların, bu yapılarla aynı hizaya sahip olması gerekmektedir. Yanlış girintileme seviyesi, Python tarafından 'IndentationError' olarak tanımlanmaktadır. Bu durum programın çalışmasını durdurmaktadır. İkinci bir hata kaynağı, tab ve boşluk karakterlerinin karışık kullanımıdır. Python 3, aynı kod bloğu içinde hem tab hem de boşluk kullanımına izin vermez. Böyle bir durumda 'IndentationError' veya 'TabError' gibi hatalar ortaya çıkmaktadır. Bu, özellikle farklı editörler kullanıldığında ya da kod parçalarının kopyalanıp yapıştırılması sırasında sıkça karşılaşılan bir problem olmaktadır. Üçüncüsü, girintileme eksikliğidir. Python'da, 'if', 'for', 'while', 'def', 'class' gibi kontrol ifadeleri kullanıldığında bu ifadelerden sonraki satırların girintilenmesi gerekmektedir. Bu yapıları takip eden kod blokları girintilenmezse Python yine bir 'IndentationError' hatası vermektedir. Dördüncü bir hata türü ise gereksiz girintilemedir. Ana kod akışı içinde gereksiz yere girintilenmiş bir satır, Python tarafından hatalı olarak algılanarak 'IndentationError' ile sonuçlanmaktadır. Son olarak, girintileme düzeylerinin tutarsız olması da bir hatadır. Özellikle bir fonksiyon veya sınıf içinde tüm kod bloklarının aynı girintileme düzeyini koruması gerekmektedir. Farklı girintileme seviyeleri kullanıldığında Python yorumlayıcısı bunu bir hata olarak kabul etmektedir.

3.4 Parantez ve İki Nokta Hataları

Python programlama dilinde, parantez ve iki nokta kullanımı ile ilgili hatalar dilin söz dizimi kurallarının ihlali nedeniyle sıkça karşılaşılan sorunlardır. Bu tür hatalar, kodun anlaşılmasını ve doğru şekilde çalışmasını engellemektedir.

Eksik veya fazladan parantez kullanımı, Python'daki yaygın hatalardan biridir. Fonksiyon çağrıları, matematiksel işlemler, koşullu ifadeler ve tuple gibi veri yapılarının tanımlanmasında parantezler sıkça kullanılmaktadır. Parantezlerin eksik konulması veya fazladan eklenmesi durumunda Python genellikle bir 'SyntaxError'

hatası oluşturmaktadır. Bu durum, genellikle parantezlerin doğru şekilde açılıp kapatılmamasından kaynaklanmaktadır. Parantezlerin doğru eşleşmesi, kodun anlaşılabilirliği ve doğru çalışması için kritik öneme sahiptir.

Yanlış yerleştirilmiş bir iki nokta işareti (:) de Python'da sık rastlanan bir hatadır. Özellikle ‘if’, ‘elif’, ‘else’, ‘for’, ‘while’, ‘def’, ‘class’ gibi kontrol yapılarını tanımlarken bu yapıların blok başlangıcı bir iki nokta işareti ile gösterilmektedir. İki nokta işareti konulmaması veya yanlış yere yerleştirilmesi durumunda Python yorumlayıcısı bir ‘SyntaxError’ hatası vermektedir. İki nokta işareti, kontrol yapılarının başlangıcını belirmektedir. Ayrıca bu yapıların içeriği girinti yoluyla ifade edilmektedir.

Koşullu ifadelerde parantez kullanımı ise Python'da opsiyoneldir. Ancak özellikle birden fazla koşulun bir arada kullanıldığı durumlarda parantezlerin doğru kullanımı, kodun okunabilirliğini ve anlaşılabilirliğini artırmaktadır. Eksik veya yanlış yerleştirilmiş parantezler, mantıksal hatalara veya SyntaxError’a neden olmaktadır.

Python'da fonksiyon tanımlarken yapılan bir başka yaygın hata, ‘def’ anahtar kelimesinden sonra fonksiyon adı ve parametreler tanımlandıktan sonra bir iki nokta işaretinin unutulmasıdır. Eğer bu işaret eksik bırakılırsa Python yorumlayıcısı bir ‘SyntaxError’ hatası vermektedir.

Python programcıları için bu tür hataları önlemek adına, parantez ve iki nokta işaretlerinin dikkatli kullanılması ve kod yazarken söz dizimi kurallarına dikkat edilmesi gerekmektedir.

3.5 İsimlendirme Hataları

Python'da isimlendirme hataları, programlama sürecinde sıkça rastlanan bir durumdur. Genellikle değişkenler, fonksiyonlar, sınıflar ve diğer tanımlayıcılar için uygun adlandırma kurallarının ihlal edilmesinden kaynaklanmaktadır. Bu tür hatalar, Python yorumlayıcısı tarafından genellikle ‘NameError’ veya ‘SyntaxError’ olarak rapor edilmektedir. En yaygın hatalardan biri, tanımlanmamış değişkenlerin kullanılmasıdır. Bu, bir değişkenin yanlış yazılması veya hiç tanımlanmamış olması durumlarında meydana gelmektedir. ‘NameError’ hatasına sebep olmaktadır. Ayrıca, Python'daki anahtar kelimelerin değişken adı olarak kullanılması da ‘SyntaxError’a neden olmaktadır. Python, büyük ve küçük harf duyarlı bir dil olduğundan değişken adlarında büyük/küçük harf kullanımı dikkatli yapılmalıdır. Aksi halde, aynı isim farklı

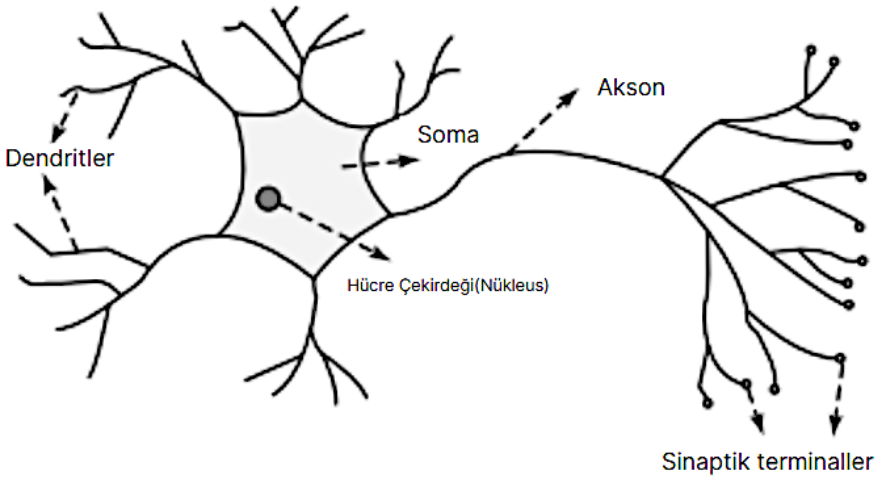
büyük/küçük harf kombinasyonları ile farklı değişkenler olarak algılanmaktadır. Bu da ‘NameError’a yol açmaktadır. Aynı şekilde, değişken ve fonksiyon isimlerinde uygunsuz karakter kullanımı veya yanlış yazım da hatalara sebep olabilir. Python programcıları için bu tür hatalardan kaçınmak, dilin adlandırma kurallarına dikkat etmek, kodları dikkatli yazmak ve olası yazım hatalarına karşı tetikte olmak gerekmektedir.

4. DERİN ÖĞRENME

Derin öğrenme, yapay zeka ve makine öğrenimi alanlarında son yıllarda büyük ilerlemeler kaydeden bir konsepttir. Temel olarak, yapay sinir ağlarından oluşan ve insan beyninin işleyişini taklit eden modelleri içermektedir. Bu modeller, karmaşık veri yapılarını analiz ederek tahminlerde bulunmaktadır. Derin öğrenmenin farklı mimarileri, özellikle zaman serileri ve sıralı veri analizinde büyük başarılar göstermiştir. Bu mimariler içerisinde Tekrarlayan Sinir Ağları (RNN), Uzun Kısa Süreli Bellek (LSTM), ve Çift Yönlü Uzun Kısa Süreli Bellek (BiLSTM) yer almaktadır. Yapay sinir ağlarının öğrenme süreci, ileri yayılım ve geri yayılım gibi teknikler ve çeşitli optimizasyon algoritmaları ile desteklenmektedir. Aktivasyon fonksiyonları, ağın nasıl tepki vereceğini belirlerken performans metrikleri modelin başarısını değerlendirmede kullanılmaktadır. Bu metrikler arasında doğruluk ve F-Skor gibi önemli ölçütler yer almaktadır. Derin öğrenme, bu özellikleri ile modern veri analizi ve yapay zeka uygulamalarının vazgeçilmez bir parçası haline gelmiştir.

4.1 Yapay Sinir Ağlarının Öğrenmesi

İnsan beyninin karmaşıklığı ve işlevselliği, 10 milyardan fazla nörondan oluşan etkileyici bir ağ ile ortaya konmaktadır. Bu nöronlar, çeşitli bilişsel, algısal ve motor görevleri yerine getirme kapasitesine sahiptir. Şekil 4.1'de gösterilen her nöronun dendritleri, hücrenin bilgi toplama ve iletiminde kullandığı dallanmış liflerdir. Bu lifler, hücrenin işlem merkezi olan somaya bağlanmaktadır. Somanın içerisindeki çekirdek, hücrenin genetik bilgisini barındırmaktadır ve işlevlerini düzenlemektedir. Akson olarak adlandırılan uzun bir lif, hücre gövdesinden çıkmakta ve bilgiyi sinaptik terminaller aracılığıyla diğer nöronlara ileten uzantılara bölünmektedir. Bu kompleks yapı, beynin paralel işleme gücünü ve kesin olmayan bilgileri işleme yeteneğini yansıtarak insan beyninin çeşitli zorlu görevleri başarıyla gerçekleştirebilme yetisinin altında yatan yapısal temeli oluşturmaktadır (Abraham, 2005).

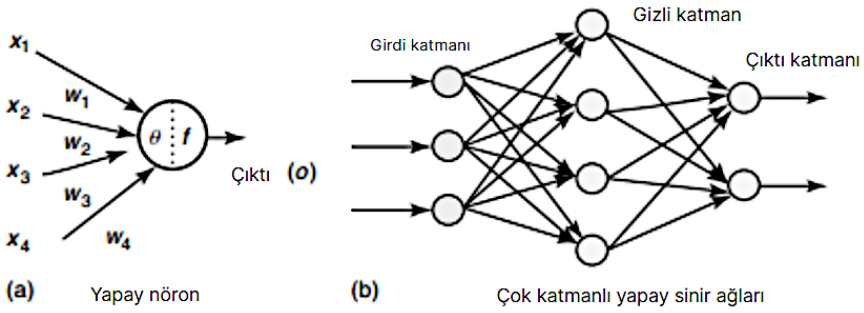


Şekil 4.1 : Nöron yapısı (Abraham, 2005).

Sinapslardaki iletim, bir nörondan diğerine sinyal geçişi sırasında verici kimyasalların salınımını içeren kompleks bir kimyasal süreçtir. Bu işlem, alıcı nöronun soma bölgesindeki elektriksel potansiyeli değiştirmektedir. Bu potansiyel belirli bir seviyeye ulaştığında akson boyunca bir impuls (itme) yayılarak nöron aktif hale gelmektedir. Yapay sinir ağları, biyolojik sinir sistemlerine dayanan matematiksel modeller olarak geliştirilmiştir. McCulloch ve Pitts'in (1943) basit nöron modellerini sunmasıyla sinir ağlarına (bağlantısal modeller veya paralel dağıtılmış işleme de denir) olan ilgi artmıştır. Yapay sinir ağlarında temel ögeler, yapay nöronlar veya düğümler olarak tanımlanmaktadır. Nöronların basitleştirilmiş matematiksel modellemesinde sinaps etkileri, giriş sinyallerinin ağırlıklarıyla temsil edilmektedir. Nöronun lineer olmayan yanıtı, bir dönüşüm fonksiyonu ile ifade edilmektedir. Bu yanıt, girişlerin dönüşüm fonksiyonu yardımıyla modifiye edilmiş ağırlıklı toplamları olarak belirlenmektedir. Yapay nöronların öğrenme yeteneği, seçilen öğrenme algoritmasıyla ağırlıkların ayarlanması sayesinde gerçekleştirmektedir. Tipik bir yapay nöronun ve çok katmanlı bir sinir ağının yapısı Şekil 4.2'de açıklanmaktadır. Şekil 4.2'de gösterildiği gibi girişlerden gelen sinyal akışının, bir nöronun çıkış sinyal akışı (O) gibi oklarla gösterilen tek yönlü olduğu kabul edilmektedir. Nöron çıkış sinyali Denklem 4.1'de verilmiştir.

$$O = f(net) = f\left(\sum_{j=1}^{n} w_j x_j\right) \tag{4.1}$$

Şekil 4.2'nin (a) bölümünde, bir yapay nöronun şematik bir gösterimi yer almaktadır. Bu yapay nöron, dört girişe (x_1,x_2,x_3,x_4) sahiptir. Her giriş, nörona gelen sinyalin gücünü belirleyen ağırlıklar (w_1,w_2,w_3,w_4) ile çarpılmıştır. Bu değerler, bir toplama fonksiyonu tarafından toplanmaktadır. Sonuca bir aktivasyon fonksiyonu (f) uygulandıktan sonra nöronun çıktısı (O) oluşturmaktadır (Abraham, 2005).



Şekil 4.2 : (a) Yapay nöron (b) Çok katmanlı yapay sinir ağı mimarisi (Abraham, 2005).

Şekil 4.2'nin (b) bölümünde ise, çok katmanlı bir yapay sinir ağı gösterilmektedir. Bu ağ yapısı, bir giriş katmanı, bir ya da birden çok gizli katman ve bir çıkış katmanından oluşmaktadır. Giriş katmanı, dışarıdan gelen ham verileri yansıtmaktadır. Her nöron, kendisinden sonraki katmanda bulunan nöronlarla bağlantılıdır. Gizli katman veya katmanlar, verileri işleyerek daha detaylı ve karmaşık özelliklerin öğrenilmesini sağlamaktadır. En sonunda çıkış katmanı, işlenmiş veriyi sonuç olarak vermektedir. Her bağlantı, ağırlık değerleri ile temsil edilmekte ve bu ağırlıklar, ağın eğitim süreci sırasında öğrenilmektedir. Bu diyagram, bir yapay sinir ağının temel yapı taşlarını ve işleyiş mekanizmasını görsel olarak açıklamaktadır.

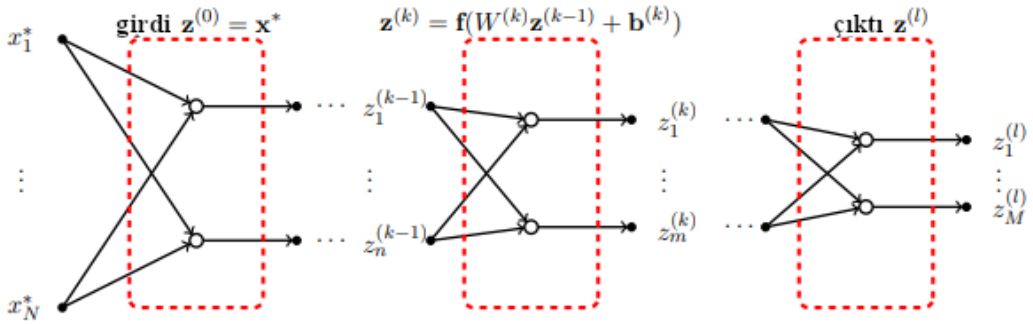
Yapay sinir ağlarının öğrenme süreci, verilen girdiler ve beklenen çıktılar arasında en iyi ilişkiyi kurmayı amaçlayarak ağırlıkları uygun şekilde ayarlamak üzerine kurulmuştur. Bu ayarlama, genellikle bir hata oranının minimize edilmesi ile gerçekleştirilir. Hata oranı, ağın ürettiği çıktı ile ideal çıktı arasındaki fark olarak tanımlanmaktadır. Geri yayılım algoritması, çıktı katmanından başlayarak her nöronda üretilen hata miktarını geriye doğru yaymaktadır. Her ağırlığın ne kadar ve hangi yönde değiştirilmesi gerektiğini belirlemektedir. Bu süreç, ağın performansını artıracak şekilde tekrarlayıcı olarak gerçekleştirilmektedir. Ağın karmaşıklığı arttıkça, katman sayısı ve nöron sayısı arttıkça, öğrenme süreci daha zorlu hale gelmektedir.

Gelişmiş optimizasyon tekniklerini ve daha fazla işlem gücünü gerektirmektedir. Sonuç olarak yapay sinir ağları, girdi verileri içerisindeki karmaşık desenleri ve ilişkileri modelleyebilmektedir. Bu sayede görsel tanıma, doğal dil işleme ve tahmin görevlerinde etkili sonuçlar üretebilmektedir.

4.1.1 İleri yayılım ve geri yayılım

İleri yayılım ve geri yayılım, yapay sinir ağlarının eğitim sürecinin iki temel aşamasını oluşturmaktadır. İleri yayılımda, giriş verileri ağın katmanları boyunca ilerlemektedir. Son katmanda ise bir tahmin üretilmektedir. Her katmandaki nöronların aktivasyonları, bir önceki katmandan gelen sinyallerin ağırlıklı toplamı ve aktivasyon fonksiyonu kullanılarak hesaplanmaktadır. Geri yayılımda ise çıktı ve beklenen sonuç arasındaki hata, ağın katmanları boyunca geriye doğru yayılmaktadır. Bu süreçte, her bir ağırlığın katkısı hata üzerinden hesaplanarak ağırlıkların güncellenmesi sağlanmaktadır. Bu iki yönlü süreç, ağın doğru tahminler yapabilmesi için gerekli öğrenme ve uyarlamanın gerçekleşmesine olanak tanımaktadır.

İleri beslemeli sinir ağları, üst üste yerleştirilmiş tam bağlantılı katmanlardan oluşan bir yapıdadır. Her bir katman, kendi ağırlık matrisi $W^{(k)}$ ve bias vektörü $b^{(k)}$ ile karakterize edilmektedir. Bu katmanlar, $z^{(k-1)}$ girdi vektörünü alarak $k = 1 \dots l$ için $z^{(k)}$ çıktı vektörünü oluşturmaktadır. Burada $z^{(l)}$ vektörü, ağın tamamının ürettiği son çıktıdır. Şekil 4.3'te gösterilmiştir. Bazı senaryolarda, ağ daha seyrek bağlantılarla basitleştirilebilir. Bu da $W^{(k)}$ ağırlık matrisinin seyrek bir yapıda olmasına neden olmaktadır. Ancak katmanlar arasındaki bağlantılar her zaman veri akışını yönlü ve döngüsüz bir graf olarak korumaktadır (Naumov, 2017).



Şekil 4.3 : Örnek bir ileri beslemeli sinir ağı (Naumov, 2017).

x^* isimli bir girdi vektörü verildiğinde sinir ağının $z^{(l)}$ olarak ifade edilen çıktısını hesaplamak için aşağıda verilen Denklem 4.2 ve Denklem 4.3 ardışık olarak uygulanmaktadır.

$$z^k = f\left(y^{(k)}\right) \tag{4.2}$$

$$y^k = W^{(k)} z^{(k-1)} + b^{(k)} \tag{4.3}$$

Yukarıdaki denklemlere göre, $k = 1 \dots l$ ve $z^0 = x^*$'dır. Bu işlem, literatürde ileri yayılım olarak tanımlanmaktadır (Naumov, 2017).

Eğitim veri seti D'den alınan bir veri örneği $(x^* z^*)$ şeklinde temsil edilmektedir. İleri yayılım kullanarak gerçek çıktı $z^{(l)}$ olmaktadır. Hata $(\frac{1}{2}\mathcal{L})$ Denklem 4.4 ile hesaplanmaktadır. $\mathcal{L}$, kayıp fonksiyonunu temsil etmektedir.

$$\varepsilon = \frac{1}{2}\left|\left|z^* - z^{(l)}\right|\right|_2^2 \tag{4.4}$$

Ağın her katmanındaki ağırlıklar w_{ij} ve b_i'yi kayıp fonksiyonu L'nin bireysel bileşenleri E'ye göre ayarlanarak optimizasyon problemi için çözüm bulunması amaçlanmaktadır. Ağırlık ve bias güncellemeleri Denklem 4.5 ve Denklem 4.6'da verilmiştir. a öğrenme oranı olarak adlandırılan sabit bir değerdir.

$$W^{(k)} - a\Delta W^{(k)} \tag{4.5}$$

$$b^{(k)} - a\Delta b^{(k)} \tag{4.6}$$

Ağırlık $(\Delta W^{(k)})$ ve bias $(\Delta b^{(k)})$ güncellemeleri, sırasıyla Denklem 4.7 üzerinden ifade edilmektedir. Bu, ağırlıkların ve biasın kayıp fonksiyonu bileşenine toplam etkisinin en büyük olduğu yönü göstermektedir.

$$\frac{\partial \varepsilon}{\partial w_{ij}^{(k)}} \ , \frac{\partial \varepsilon}{\partial b_i^{(k)}} \tag{4.7}$$

İleri beslemeli sinir ağı Denklem 4.2 ve Denklem 4.3 ile tanımlanmaktadır. Kayıp fonksiyonu bileşeni Denklem 4.4'te belirtilmiştir. Ağırlıkların ve biasın gradyanı ise Denklem 4.8 (rank-1 matris için) ve Denklem 4.9'da verilmiştir.

$$G_w^{(k)} = v^k z^{(k-1)^T} \tag{4.8}$$

$$G_b^{(k)} = v^k \tag{4.9}$$

Burada $G_w^{(k)} = \left[\partial \mathcal{E} / \partial w_{ij}^{(k)}\right]$ $m \times n$ boyutlu bir matristir, ayrıca $G_b^{(k)} = \left[\partial \mathcal{E} / \partial b_i^{(k)}\right]$ $m \times 1$ boyutlu vektörü $k = l, \dots, 2$, için

$$\begin{aligned} \boldsymbol{v}^{(l)} &= -\left(\boldsymbol{z}^* - \boldsymbol{z}^{(l)}\right) \circ \boldsymbol{f}'\left(\boldsymbol{y}^{(l)}\right) \\ \boldsymbol{v}^{(k-1)} &= \left(W^{(k)^T} \boldsymbol{v}^{(k)}\right) \circ \boldsymbol{f}'\left(\boldsymbol{y}^{(k-1)}\right) \end{aligned} \tag{4.10}$$

ile birlikte ele düşünülmektedir. Burada $\circ$ işlemi Hadamard (bileşen-bazında) çarpımı olarak adlandırılmakta ve $f'() =. [f'(.), \dots, f'(.)]^T$ dir.

Kayıp fonksiyonunun ağırlığa göre bileşen bazında kısmi türevini alarak aşağıdaki eşitlikleri elde edilmektedir.

$$\begin{aligned} \frac{\partial \mathcal{E}}{\partial w_{ij}^{(k)}} &= \left(\frac{\partial \mathcal{E}}{\partial z_i^{(k)}}\right)\left(\frac{\partial z_i^{(k)}}{\partial y_i^{(k)}}\right)\left(\frac{\partial y_i^{(k)}}{\partial w_{ij}^{(k)}}\right) \\ &= \left(\frac{\partial \mathcal{E}}{\partial z_i^{(k)}}\right) f'\left(y_i^{(k)}\right) z_j^{(k-1)} \\ &= v_i^{(k)} z_j^{(k-1)} \end{aligned} \tag{4.11}$$

Burada f', df/dy nin adi türevini ifade etmekte ve $v_i^{(k)} = \left(\frac{\partial \mathcal{E}}{\partial z_i^{(k)}}\right) f'\left(y_i^{(k)}\right)$ dir. Ayrıca, (4.4)'ü kullanarak çıktı katmanı $k = l$ için elde edilmiştir. Şu durumda

$$\frac{\partial \mathcal{E}}{\partial z_i^{(l)}} = -\left(z_i^* - z_i^{(l)}\right) \tag{4.12}$$

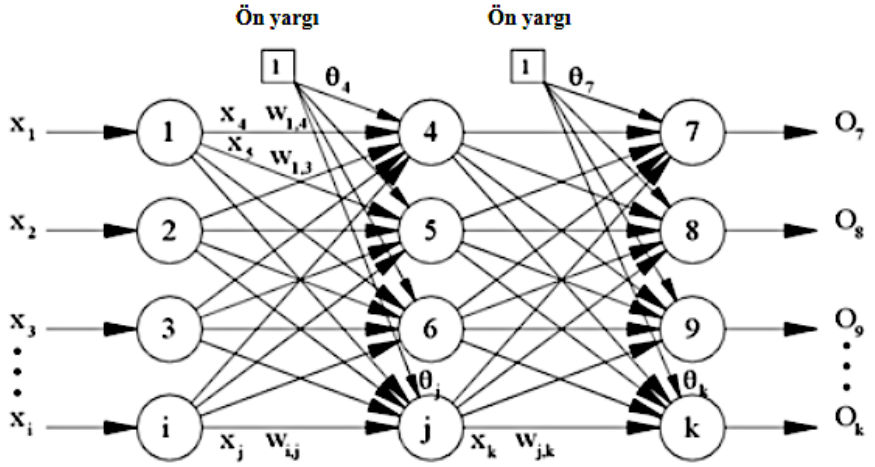
eşitliği elde edilmektedir. Zincir kuralını kullanarak gizli katmanlar için ise

$$\begin{aligned} \frac{\partial \mathcal{E}}{\partial z_i^{(k-1)}} &= \sum_{j=1}^{n} \left(\frac{\partial \mathcal{E}}{\partial z_j^{(k)}}\right) f'\left(y_j^{(k)}\right) w_{ji}^{(k)} \\ &= \sum_{j=1}^{n} v_j^{(k)} w_{ji}^{(k)} \end{aligned} \tag{4.13}$$

ifadesi elde edilmektedir. Son olarak, i ve j indislerini matris formunda bir araya getirerek $G_w^{(k)}$ ifadesi elde edilmektedir. (4.11)'de $\frac{\partial y_i^{(k)}}{\partial b_i^{(k)}} = 1$ eşitliğiyle birlikte $G_b^{(k)}$ için türev benzer şekilde bulunmaktadır.

Denklem 4.10'da $\mathbf{v}^{(k)}$ yardımcı vektörünün hesaplanması (4.4) hatasının l çıkış katmanından $k = l-1, \ldots, 2$ katmanlarına yayılmasını ifade etmektedir. Bu sürece geriye yayılım denmektedir (Naumov, 2017).

Geri yayılım, yapay sinir ağlarını eğitmek için kullanılan bir yöntemdir. Şekil 4.4'te temsili gösterimi verilmiştir.



Şekil 4.4 : Tek gizli katmanlı geri yayılımlı sinir ağı (Lu,2000).

Hedef, ağın çıktısında oluşan hataları azaltmaktır. Geri yayılım sürecinde ağın çıktısından başlayarak her katmanda oluşan hata geriye doğru, yani giriş katmanına doğru yayılmaktadır. Bu hata, ağın ağırlıklarının ve biaslarının nasıl güncellenmesi gerektiği konusunda bilgi sağlamaktadır. Geri yayılım, gradyan inişi yöntemiyle birlikte kullanılarak ağın ağırlıklarını optimizasyon problemine göre ayarlamaktadır. Bu sayede ağın tahminlerini iyileştirmektedir. Bu süreç, ağın öğrenmesini ve performansının artmasını sağlamaktadır.

4.1.2 Optimizasyon algoritmaları

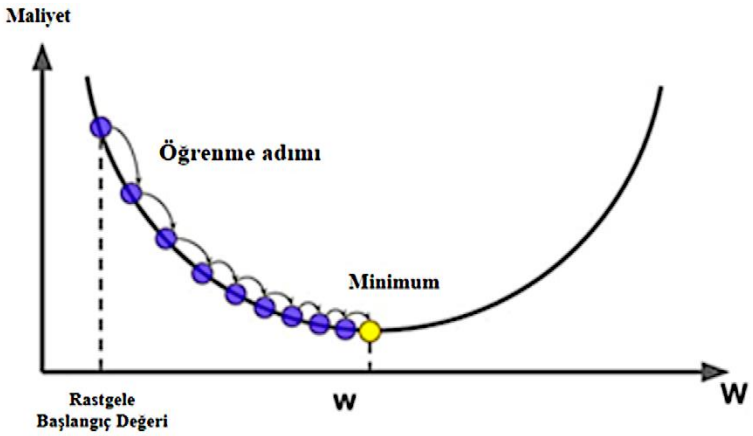
Optimizasyon algoritmaları, belirli bir hedefe veya en iyi sonuca ulaşmayı amaçlayan matematiksel yöntemlerdir. Bu algoritmalar, genellikle bir fonksiyonun maksimum veya minimum değerini bulmaya yöneliktir. Kullanım alanları oldukça geniştir. Ekonomiden mühendisliğe, yapay zekadan veri bilimine kadar birçok alanda uygulamaları vardır. Optimizasyon, karmaşık problemleri çözmede kritik bir rol oynamaktadır. Problemlerin en iyi ya da en verimli çözümünün bulunmasına yardımcı olmaktadır. Optimizasyon algoritmaları, gradyan inişi, genetik algoritmalar, parçacık

sürü optimizasyonu gibi çeşitli yöntemler içermektedir. Her biri farklı türdeki optimizasyon problemleri için uygun çözümler sunmaktadır.

Yapay sinir ağlarının eğitiminde kullanılan optimizasyon algoritmaları, ağın performansını iyileştirmek ve daha doğru tahminler yapabilmesini sağlamak için kritik öneme sahiptir. Bu metotlar, verilen bir kayıp fonksiyonunu en aza indirecek şekilde ağın ağırlıklarını ve bias değerlerini ayarlamak için kullanılmaktadır. Gradyan inişi optimizasyon algoritması, bir fonksiyonun minimum değerini bulmak için kullanılan bir yöntemdir. Temelde, bir maliyet veya kayıp fonksiyonunun gradyanını (veya türevini) almaktadır. Gradyanın negatif yönünde adımlar atarak fonksiyonun minimum değerine ulaşmayı hedeflemektedir. Gradyan inişi optimizasyon algoritması Denklem 4.14'te verilmiştir. Gradyan inişi veya toplu gradyan inişi yönteminde, maliyet fonksiyonunun gradyanı, tüm eğitim veri seti üzerinden θ parametreleri kullanılarak hesaplanmaktadır. Gradyan inişi, bir modelin parametreleri $\theta \in R^d$ tarafından parametrize edilen bir objektif fonksiyonu $J(\theta)$ minimize etme yöntemidir (Ruder, 2016). Öğrenme oranı (η), bir (yerel) minimuma ulaşmak için atılan adımların büyüklüğünü belirlemektedir.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \qquad (4.14)$$

Her adımda, parametreler gradyanın yönünde belirli bir oranda güncellenmektedir. Bu oran öğrenme oranı olarak bilinmektedir. Gradyan inişi, özellikle makine öğrenimi ve derin öğrenme alanlarında, modellerin eğitiminde kullanılan temel bir optimizasyon tekniğidir. Bu teknik, modelin performansını en üst düzeye çıkarmak için parametrelerin en uygun değerlerini tespit etmeyi hedeflemektedir. Şekil 4.5, gradyan inişi optimizasyon algoritmasının maliyet fonksiyonu üzerindeki yolunu göstermektedir. Maliyet değerini simgelemektedir. Bu görsel, algoritmanın optimum değere nasıl yaklaştığını ve ağırlık değerlerinin nasıl güncellendiğini göstermektedir.



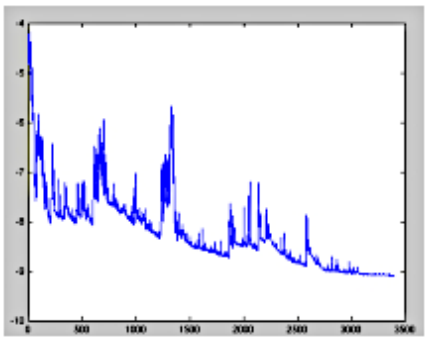
Şekil 4.5 : Gradyan inişi algoritması (Bhattarai, 2018).

Eğrinin başlangıç noktası, ağırlıkların rastgele bir başlangıç değerini ifade etmektedir. Buradan itibaren öğrenme adımları boyunca maliyet fonksiyonu değeri (yatay eksende W ile gösterilen ağırlık parametresine karşılık dikey eksende maliyeti gösteren Cost), global minimuma ulaşana kadar düşüş göstermektedir. Her bir mor nokta, öğrenme oranıyla belirlenen adım büyüklüğüyle yapılan iteratif ağırlık güncellemelerini temsil etmektedir. Sarı nokta ise algoritmanın ulaşmaya çalıştığı minimum maliyet değerini simgelemektedir. Şekil 4.5, algoritmanın optimum değere nasıl yaklaştığını ve ağırlık değerlerinin nasıl güncellendiğini göstermektedir.

Stokastik gradyan inişi (SGD) yöntemi, her bir eğitim örneği x^i ve karşılık gelen etiketi y^i için parametreleri güncellemektedir. Denklem 4.15'te SGD'nin formülasyonu verilmiştir.

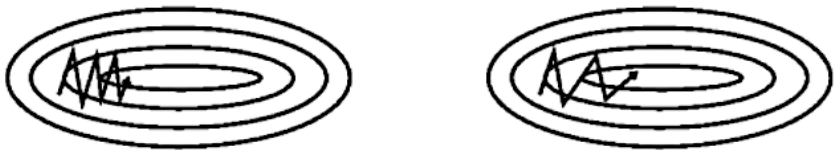
$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{(i)}; y^{(i)}\right) \tag{4.15}$$

Toplu gradyan inişi, büyük veri setleri için gereksiz hesaplamalar yapmaktadır. Çünkü her parametre güncellemesinden önce benzer örnekler için gradyanları yeniden hesaplamaktadır. SGD, her seferinde tek bir güncelleme yaparak bu fazlalığı azaltmaktadır. Bu güncelleme tekniği, genellikle çok daha hızlı olmasını sağlamaktadır. SGD fonksiyonu Şekil 4.6'da gösterilen gibi aşırı dalgalanmalara yol açan yüksek varyans ile sık güncellemeler yapmaktadır.



Şekil 4.6 : SGD dalgalanması (Ruder, 2016).

Toplu gradyan inişi, parametrelerin yerleştirildiği havzanın minimumuna yakınsarken, SGD'nin dalgalanması, bir yandan onu yeni ve potansiyel olarak daha iyi yerel minimumlara atlamasını sağlamaktadır. Diğer yandan, SGD'nin sürekli olarak hedefin ötesine geçmesi nedeniyle kesin minimuma yakınsaması karmaşıklaşmaktadır. Ancak, Şekil 4.6'da öğrenme oranı yavaşça azaltıldığında SGD'nin toplu gradyan inişiyle aynı yakınsama davranışını gösterdiği dışbükey olmayan ve dışbükey optimizasyon için sırasıyla yerel veya global minimuma yakınsayacağı gösterilmiştir.



Şekil 4.7 : Momentum etkisinin sgd optimizasyonu üzerindeki karşılaştırması (LeCun ve diğ, 1998).

Şekil 4.7, iki farklı SGD yöntemini göstermektedir: (a) momentum olmadan SGD ve (b) momentum ile SGD. (a) kısmında, momentum kullanılmadan uygulanan SGD'nin, optimizasyon yüzeyindeki derin vadilerde dalgalanarak ilerlediği ve bu dalgalanmaların optimaya ulaşmada tereddütlü bir ilerlemeye neden olduğu görülmektedir. (b) kısmında ise, momentum terimi eklenen SGD'nin daha düzgün bir yol izlediği ve bu sayede yüzeyde daha hızlı ve etkin bir şekilde yerel minimuma ulaştığı gösterilmektedir. Momentum, önceki gradyanların birikimini hesaba katarak algoritmanın düzgün bir şekilde ilerlemesine ve yerel minimumlara daha etkili bir biçimde yaklaşmasına yardımcı olmaktadır.

Momentum, SGD'ye belirli bir yönde ivme kazandırarak osilasyonları düşürmeye yardımcı olan bir tekniktir. Şekil 4.7'de gösterilmektedir. Denklem 4.16'da momentum yönteminin formülasyonu verilmiştir.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t \tag{4.16}$$

Momentum, geçmiş zaman adımının güncelleme vektörünün bir kısmı γ eklenerek mevcut güncelleme vektörüne yardımcı olmaktadır. Böylece birikmiş geçmiş gradyanlar, güncellemeler üzerinde bir ivme etkisi yaratmaktadır. Algoritmanın, hesaplama yüzeyindeki en dik eğimler boyunca sürekli bir hız kazanmasını sağlamaktadır. Momentum terimi genellikle 0.9 gibi bir değere ayarlanmaktadır. Bu değer, gradyanların tutarlı yönde hareket ettiği boyutlarda ivmenin artmasına fakat gradyan yönlerinin değiştiği boyutlarda güncellemelerin azaltılmasına yol açmaktadır. Sonuçta bu durum daha hızlı bir yakınsama ve osilasyonların azaltılmasını mümkün kılmaktadır (Ruder, 2016).

RMSprop, Geoff Hinton'ın Coursera'daki dersinin 6e bölümünde tavsiye ettiği, yayınlanmamış bir adaptif öğrenme oranı tekniğidir. RMSprop, gradyan tabanlı optimizasyon tekniklerinden biri olan ve adaptif bir öğrenme oranı sağlayarak derin öğrenme modellerinin eğitiminde sıklıkla kullanılan bir algoritmadır. Bu yöntem, öğrenme oranını, algoritmanın önceki adımlarındaki gradyanların karesel ortalamasının köküne bölerek ayarlamaktadır. Öğrenme oranını her adımda gradyanların büyüklüğüne göre uyarlayarak öğrenme sürecinin başlangıcındaki hızlı ilerlemeyi korurken ilerleyen adımlarda daha dengeli bir yakınsama sağlamaktadır. Denklem 4.17 ve 4.18'de RMSprop formülasyonu verilmiştir. t zaman adımlarını temsil etmektedir. $G_t \in R^{dxd}$ burada bir diyagonal matris olup her diyagonal elemanı belirli bir matematiksel ifade veya vektörün bir bileşenini temsil etmektedir. Diyagonal matrislerde, matrisin diyagonal dışındaki tüm elemanları sıfırdır. Sadece diyagonal üzerindeki elemanlar değer taşımaktadır. Bu tür matrisler, özellikle optimizasyon algoritmalarında farklı parametreler veya gradyanlar için farklı ölçeklendirme faktörleri uygulamak amacıyla kullanılmaktadır.

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \tag{4.17}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}}\, g_t \tag{4.18}$$

Uyarlanabilir Moment Kestirimi (Adam), her bir parametre için adaptif öğrenme oranları hesaplayan başka bir yöntemdir (Diederik ve Jimmy, 2015). RMSprop gibi, Adam da geçmiş karesel gradyanların üssel olarak azalan ortalamasını v_t olarak tutarken aynı zamanda momentum benzeri şekilde geçmiş gradyanların üssel olarak azalan ortalamasını m_t olarak da saklamaktadır. Denklem 4.19'da verilmiştir.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad (4.18)$$

m_t ve v_t sırasıyla gradyanların birinci momenti (ortalama) ve ikinci momenti (merkezlenmemiş varyans) tahminleridir. Adam'ın yazarları, özellikle başlangıç zaman adımlarında ve bozulma oranları düşükken (yani β_1 ve β_2 1'e yakınken) m_t ve v_t'nin sıfıra doğru ön yargılı olduğunu gözlemlemişlerdir. Bu ön yargıları düzelten bias-düzeltmeli ilk ve ikinci moment tahminlerini hesaplamışlardır. Denklem 4.20'de verilmiştir.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \quad (4.20)$$

Daha sonra bu tahminler, parametreleri güncellemek için kullanılmıştır, bu da Adam güncelleme kuralını (Denklem 4.21) vermektedir.

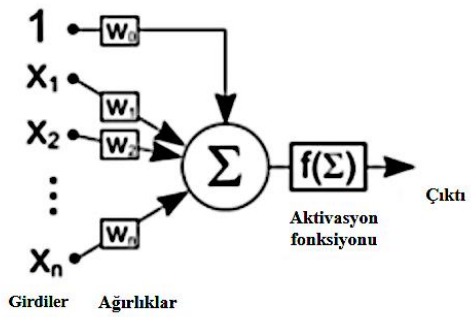
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \quad (4.21)$$

Yazarlar β_1 için varsayılan değer olarak 0.9, β_2 için 0.999 ve ϵ için 10^{-8} önermektedir. Diederik ve Jimmy (2015) tarafından yapılan deneyler, Adam'ın pratikte iyi bir performans sergilediğini ve diğer adaptif öğrenme yöntemlerine kıyasla avantajlı olduğunu göstermiştir. Bu algoritmalar, derin öğrenme modellerinin karmaşık veri setlerinden etkili bir şekilde öğrenmesini mümkün kılmaktadır.

4.1.3 Aktivasyon fonksiyonları

Yapay sinir ağlarında aktivasyon fonksiyonları, bir nöronun girdi sinyallerini alıp çıktı sinyallerine dönüştüren matematiksel işlevlerdir. Bu fonksiyonlar, nöronun ne zaman ve nasıl aktif olacağını belirlemektedir. Lineer olmayan aktivasyon fonksiyonları sayesinde yapay sinir ağları karmaşık problemleri modelleyebilmektedir. Lineer olmayan ilişkileri öğrenebilmektedir. Örneğin, ReLU (Düzeltilmiş Lineer Birim), sigmoid ve tanh aktivasyon fonksiyonları yapay sinir ağlarında yaygın olarak

kullanılmaktadır. Bu fonksiyonlar, ağın her katmanındaki nöronların çıktılarını belirleyerek ağın sonraki katmanlarına veya çıktı katmanına iletilen sinyallerin şeklini ve şiddetini kontrol etmektedir.

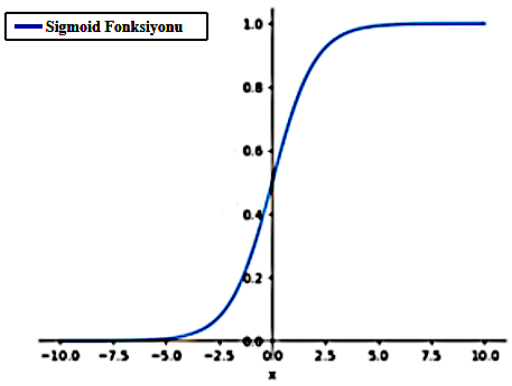


Şekil 4.8 : Yapay nöron ile aktivasyon fonksiyonu ilişkisinin temsili gösterimi (Sibi ve diğ, 2013).

Şekil 4.8'de, bir yapay nöronun yapısını gösteren bir diyagram bulunmaktadır. Girdiler $X_1, X_2, \ldots, X_n$ olarak gösterilmiştir. Her bir girdi, ilgili $w_1, w_2, \ldots, w_n$ ile çarpılmıştır. Ayrıca bir bias ağırlığı olan w_0 da eklenmiştir. Bu değerler toplanarak toplam girdi $\sum$ elde edilmiştir. Bu toplam, aktivasyon fonksiyonu $f(\sum)$ ile işlenerek çıktı üretilmiştir. Bu işlem, yapay nöronun aktivasyonunu ve sonraki katmana iletilen sinyali belirlemektedir.

4.1.3.1 Sigmoid fonksiyonu

Sigmoid aktivasyon fonksiyonu, yapay sinir ağlarında kullanılan ve girdi değerlerini 0 ile 1 arasında bir değere dönüştüren bir fonksiyondur. Şekil 4.9, sigmoid aktivasyon fonksiyonunun grafiğini göstermektedir.



Şekil 4.9 : Sigmoid fonksiyonu (Pan ve diğ, 2023).

Bu fonksiyon, x eksenindeki değerler için -10 ile 10 arasında değişirken y eksenindeki sigmoid (x) değerlerini 0 ile 1 arasında sınırlı bir şekilde çizmektedir. Denklem 4.22'de matematiksel formülü verilmiştir.

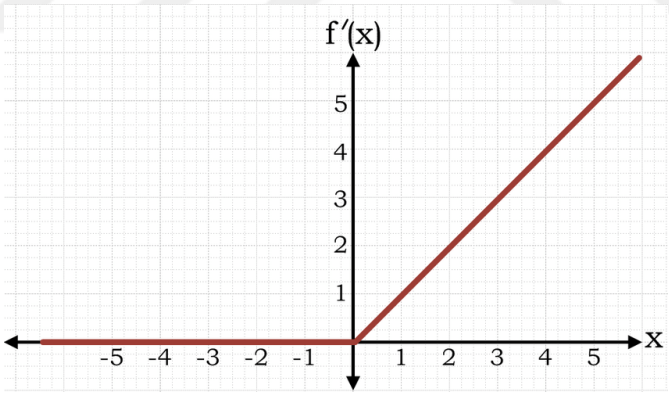
$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.22}$$

Sigmoid fonksiyonu, genellikle ikili sınıflandırma problemlerinde bir nöronun aktivasyonu için kullanılmaktadır. Çünkü çıktıları 0 ile 1 arasında bir olasılık değerine sıkıştırmaktadır. Bu, girdi değerlerinin çok büyük veya çok küçük olması durumunda dahi stabil bir çıktı sağlamaktadır.

4.1.3.2 Düzeltilmiş lineer birim (ReLU) fonksiyonu

ReLU, yapay sinir ağlarında sıkça kullanılan bir aktivasyon fonksiyonu olarak bilinmektedir. Fonksiyon, girdisi sıfırdan büyükse girdiyi olduğu gibi çıktıya vermektedir. Sıfırdan küçük veya eşitse sıfır döndürmektedir. Matematiksel olarak Denklem 4.23'teki gibi ifade edilmektedir.

$$(\mathrm{x}) = \begin{cases} x & , x > 0 \\ 0 & , x \leq 0 \end{cases} \tag{4.23}$$



Şekil 4.10 : ReLU fonksiyonu (Pratiwi, 2020).

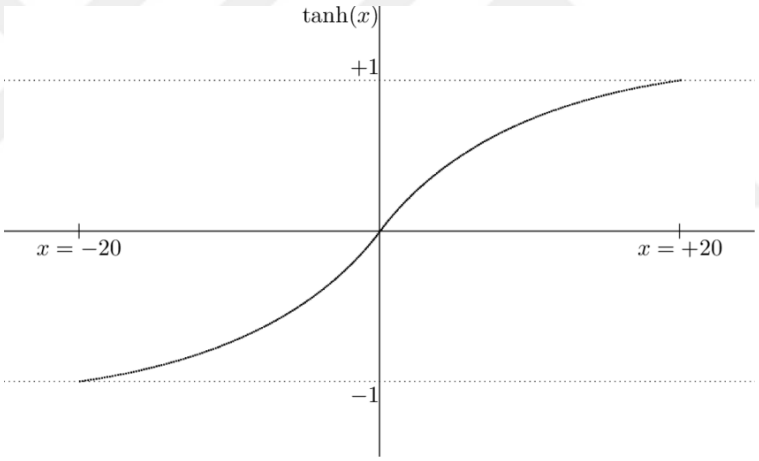
Şekil 4.10, ReLU aktivasyon fonksiyonunun grafiğini göstermektedir. Bu grafikte, *x* eksenindeki negatif değerler için fonksiyonun çıktısı sıfırdır. *X* pozitif olduğunda, fonksiyon çıktısı *x* ile doğru orantılı olarak artmaktadır. Bu, ReLU'nun parçalı lineer doğasını yansıtmaktadır. Negatif girdiler için çıktı sıfırken pozitif girdiler için çıktı girdiyle aynı olmaktadır. ReLU, derin öğrenme modellerinde sıkça kullanılmaktadır. Çünkü hesaplama açısından verimli ve gradyan kaybı sorununu azaltmaya yardımcı olmaktadır (Pratiwi, 2020).

4.1.3.3 Hiperbolik tanjant fonksiyonu

Hiperbolik tanjant fonksiyonu, gerçek sayıları -1 ile 1 arasında bir değere dönüştüren bir aktivasyon fonksiyonudur. Matematiksel gösterimi Denklem 4.24'teki gibi ifade edilmektedir. Bu fonksiyon, sigmoid fonksiyonuna benzer şekilde girdileri sıkıştırmaktadır. Ancak çıktı aralığı -1 ile 1 arasındadır. Bu da negatif girdilere izin verilmesini sağlamaktadır. Bazı durumlarda öğrenme sürecini iyileştirmektedir. Sıfır merkezli olması nedeniyle, özellikle gradyan tabanlı öğrenme yöntemlerinde tercih edilen bir aktivasyon fonksiyonudur.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} = 2\sigma(2x) - 1 \tag{4.24}$$

Şekil 4.11, hiperbolik tanjant (tanh) aktivasyon fonksiyonunun grafiğini göstermektedir. Bu fonksiyon, girdi değerlerini (-∞, +∞) aralığından (-1, +1) aralığına sıkıştırmaktadır. Böylece çıktıları iki yönlü olarak normalleştirmektedir.



Şekil 4.11 : Hiperbolik tanjant fonksiyonu (Beebe, 1991).

Grafikte, fonksiyonun -20'den +20'ye kadar olan *x* değerleri için nasıl davrandığı gösterilmektedir. *X* değerleri arttıkça veya azaldıkça tanh fonksiyonunun çıktısı asimptotik olarak +1'e veya -1'e yaklaşmaktadır. Bu da doğrusal olmayan aktivasyonun güçlü bir özelliğidir. Bu, özellikle gradyanların verimli bir şekilde yayılmasına yardımcı olmaktadır. Sıfır etrafında simetrik olduğundan ağın öğrenme sürecini iyileştirmektedir.

4.1.3.4 Softmax fonksiyonu

Softmax fonksiyonu, genellikle çok sınıflı sınıflandırma problemlerinde kullanılan bir aktivasyon fonksiyonudur. Bir vektördeki her bir değeri 0 ile 1 arasında bir değere dönüştürerek bu değerlerin toplamının 1 olmasını sağlamaktadır. Bu şekilde, bir olasılık dağılımı elde edilmektedir (Gao ve Pavel, 2017).

Softmax fonksiyonunun matematiksel ifadesi Denklem 4.25'te verilmiştir.

$$\sigma_i(z)\ \frac{\exp(\lambda z_i)}{\sum_{j=1}^{n}\exp(\lambda z_j)}, 1\ \leq i\ \leq n. \tag{4.25}$$

Burada $\sigma_i(z)$ fonksiyonun i'inci çıktısını, z girdi vektörünü, n ise vektördeki eleman sayısını ifade etmektedir. λ ise softmax fonksiyonunun keskinliğini ayarlayan bir parametredir.

4.2 Derin Öğrenme Mimarileri

Derin öğrenme mimarileri, insan beyinin bilgi işleme mekanizmasından ilham alınarak geliştirilen veri tabanlı öğrenme ve karar verme süreçlerini otomatize eden algoritmik yapılar olarak ortaya çıkmıştır. Bu mimariler, genellikle birden fazla işleme katmanına sahip yapay sinir ağlarından oluşmaktadır. Karmaşık görevleri, veriden özellik çıkararak ve bu özellikler üzerindeki hiyerarşik ilişkileri öğrenerek gerçekleştirmektedir. Görüntü ve ses tanıma, doğal dil işleme ve otonom araçlardaki navigasyon sistemleri gibi çeşitli uygulamalarda kullanılan bu mimariler, makine öğreniminin en ileri uç teknolojileri arasında yer almaktadır. Öğrenme sürecinde, derin ağlar kendilerine sunulan ham veriler içerisinden temsil edici özellikleri keşfetmekte ve bu özellikleri kullanarak daha önce karşılaşmadığı veriler üzerinde tahminlerde bulunabilmektedir. Bu yetenekleri sayesinde, derin öğrenme, yapay zekanın en çok ilgi gösterilen ve aktif araştırma alanlarından biri haline gelmiştir.

4.2.1 Zaman serileri ve sıralı veri analizi için derin öğrenme mimarileri

Zaman serileri ve sıralı diziler, günümüz veri dünyasının en karmaşık ve bilgi dolu unsurları arasında yer almaktadır. Bu veriler, finansal piyasalardan sağlık bilimlerine, hava durumu tahminlerinden dil işlemeye kadar geniş bir yelpazede karşımıza çıkmaktadır. Karar verme süreçlerinde kritik öneme sahiptir. Modern teknolojideki ilerlemeler, bu karmaşık veri yapılarını analiz etmek ve onlardan anlamlı bilgiler

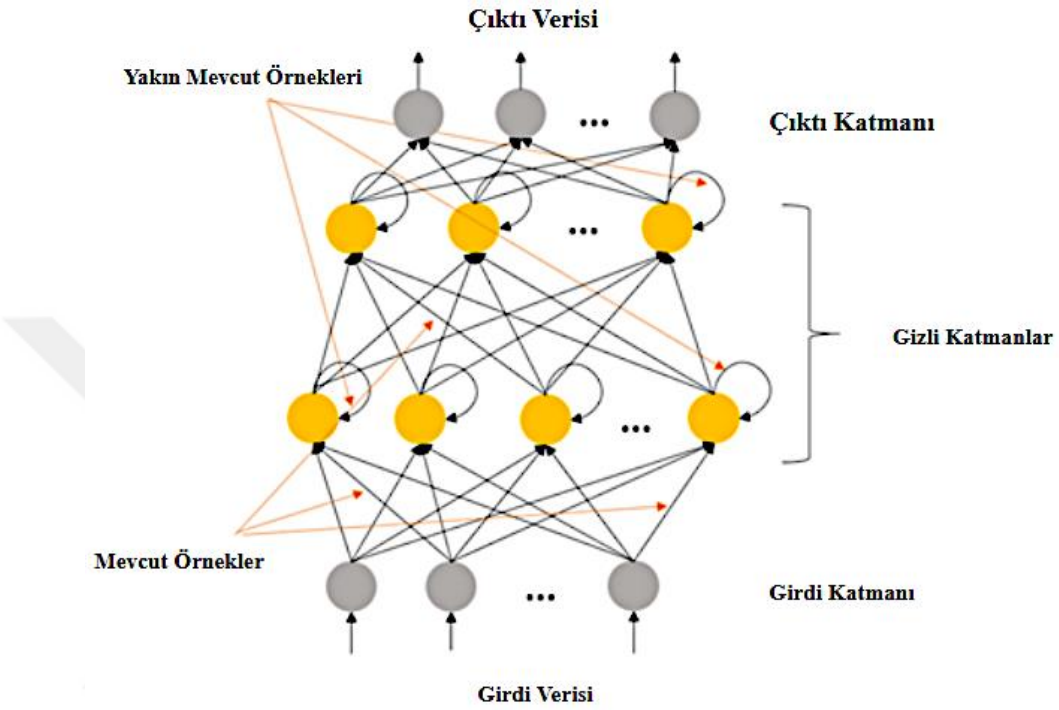
çıkarmak için derin öğrenme tekniklerinin kullanımını artırmıştır. Derin öğrenme, zaman serileri ve sıralı diziler için tasarlanmış mimariler aracılığıyla bu alanda önemli başarılar elde etmiş ve yeni ufuklar açmıştır. Özellikle RNN, ve Çift Yönlü Uzun Kısa ve BiLSTM gibi mimariler, veriyi işleme yetenekleri ve öngörü yapma kabiliyetleri ile dikkat çekmektedir. Bu mimarilerin her birinin temel yapıları, işlevsel özellikleri ve uygulama alanları, zaman serisi ve sıralı dizilerin analizinde nasıl önemli rol oynadıklarını göstermektedir. Her bir mimarinin avantajları ve zorlukları, onların hangi durumlarda en etkili olduğunu belirlememize yardımcı olmaktadır. Bu derinlemesine inceleme, zaman serisi ve sıralı dizi analizinde derin öğrenmenin gücünü ve esnekliğini vurgulayarak bu alandaki gelecekteki gelişmeler için bir temel oluşturmaktadır.

4.2.1.1 Tekrarlayan sinir ağları (Recurrent neural network- RNN)

Tekrarlayan Sinir Ağları (RNN), zamanla değişen verileri işlemek amacıyla özel olarak tasarlanmış bir yapay sinir ağı türüdür. Temel yapısı, veriler arasındaki sıralı bağlantıları ve zamanla gelişen desenleri yakalayabilme yeteneğine dayanmaktadır. Geleneksel yapay sinir ağlarından farklı olarak RNN'ler, önceki adımlardan gelen bilgileri mevcut girdi ile birleştirerek zaman serisi verilerini, dili, sesi veya herhangi bir sıralı veriyi etkin bir şekilde işlemektedir (Williams ve Zipser, 1989).

RNN'lerin temelini oluşturan anahtar konsept, "gizli durum" (hidden state) olarak adlandırılmaktadır. Gizli durum, ağın her adımda sakladığı ve sonraki adımların işlenmesinde kullanılan bir hafıza birimidir. Ağ önceki bilgileri gelecekteki tahminlerde kullanmaktadır. Bu sayede zamana bağlı verilerdeki desenleri öğrenmektedir. Örneğin, bir cümledeki sonraki kelimenin tahmin edilmesi gibi durumlarda, RNN'nin gizli durumu, önceki kelimelerin bağlamını saklayarak daha doğru tahminler yapmasını sağlamaktadır. Ancak, RNN'ler dezavantajlara da sahiptir. Özellikle "uzun süreli bağımlılıkların" (long-term dependencies) öğrenilmesi, standart RNN yapılarında problem oluşturmaktadır. Bu durum, kaybolan gradyan (vanishing gradient) veya patlayan gradyan (exploding gradient) problemleri olarak bilinmektedir. Bu problem, ağın eğitim sürecinde zorluklar yaratmaktadır. Uzun vadeli bağımlılıkları daha etkili bir şekilde öğrenmek için geliştirilen çözümler arasında LSTM ve GRU gibi RNN varyasyonları yer almaktadır (Staudemeyer ve Morris, 2019). Şekil 4.12 bir tekrarlayan sinir ağı (RNN) yapısını temsil eden bir sinir ağı

modelini göstermektedir. Alt kısımda yer alan gri nöronlar giriş katmanını (Input Layer) oluşturmaktadır. Ağa giren ham veriyi (Input Data) temsil etmektedir. Sarı nöronlar, gizli katmanları (Hidden Layers) simgelemektedir. Ağın veriyi işleyip öğrendiği bölümdür. En üstteki gri nöronlar ise çıktı katmanını (Output Layer) temsil etmektedir. İşlenmiş verinin sonucunu (Output Data) göstermektedir.



Şekil 4.12 : Tekrarlayan sinir ağları (AbdelMoniem ve diğ, 2019).

RNN yapısında dikkat çekici olan giriş katmanındaki mevcut örnekler (Current Samples) ile yakın mevcut örnekler (Recent Past Samples) arasındaki turuncu oklarla gösterilen bağlantılardır. Bu bağlantılar, ağın geçmişteki örneklerden bilgi almasını ve mevcut girdiyle birleştirerek çıktıyı oluşturmasını sağlayan geri besleme döngüleridir. Bu, RNN'nin zamanla değişen veri desenlerini ve sıralı bağımlılıkları öğrenme kabiliyetini göstermektedir.

Şekil 4.12'de ayrıca, her bir gizli katman nöronunun hem önceki katmandan hem de aynı katmandaki diğer nöronlardan girdi aldığı ve çıktı ürettiği görülmektedir. Bu karmaşık bağlantı yapısı, ağın karmaşık desenleri tanıma ve zamanla değişen verilerdeki ilişkileri modelleme kabiliyetini artırmaktadır. RNN'nin bu yapısı, doğal dil işleme, konuşma tanıma ve zaman serisi analizi gibi alanlarda uygulama bulmasının temel nedenlerindendir.

4.2.1.2 Uzun kısa süreli bellek (Long short term memory- LSTM)

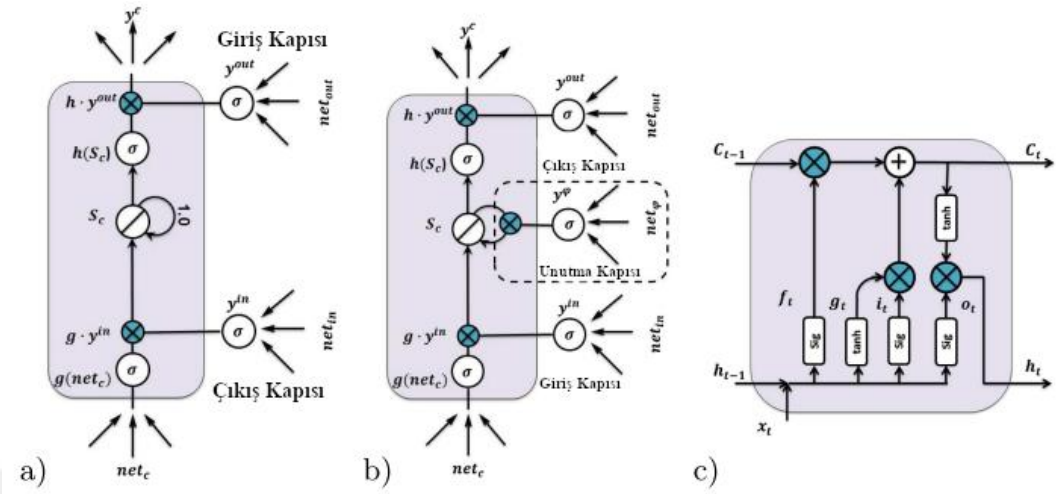
LSTM birimleri, standart tekrarlayan sinir ağlarındaki bazı sorunları çözmek için tasarlanmış özel bir RNN mimarisidir. 1997 yılında Sepp Hochreiter ve Jürgen Schmidhuber tarafından tanıtılan LSTM, özellikle uzun zaman aralıklarında bağımlılıkları öğrenme yeteneği ile dikkat çekmektedir. Bu, LSTM'leri, doğal dil işleme, konuşma tanıma ve zaman serisi analizi gibi uygulamalarda son derece kullanışlı hale getirmektedir.

Standart RNN'ler, kısa süreli bağımlılıkları başarıyla öğrenebilirken uzun süreli bağımlılıklar genellikle kaybolan gradyan (vanishing gradient) ve patlayan gradyan (exploding gradient) sorunları nedeniyle öğrenilemez. Kaybolan gradyan sorunu, gradyanların zamanla eksponansiyel bir şekilde küçülmesi sonucu ağın öğrenme yeteneğinin ciddi şekilde azalmasına yol açmaktadır. Patlayan gradyan sorunu ise tam tersi bir durumda ortaya çıkmaktadır. Gradyanlar kontrolsüz bir şekilde büyüyerek ağın kararsız hale gelmesine sebep olmaktadır. LSTM, bu sorunları, ağın içinde bilgiyi uzun süreler boyunca tutabilecek ve gradyanların akışını düzenleyebilecek bir yapı ile çözmektedir.

LSTM'lerin temel bileşenleri, verilerin ağ içindeki akışını kontrol eden üç anahtar mekanizmadan oluşmaktadır. Bu mekanizmalar, unutma mekanizması (forget gate), giriş mekanizması (input gate) ve çıkış mekanizmasıdır (output gate). Unutma mekanizması, hücrenin önceki durumunu ne kadarının unutulacağını belirlemektedir. Giriş mekanizması, giriş verisinin hücre durumunu ne kadar etkileyeceğine karar vermektedir. Çıkış mekanizması ise hücre durumunun sonraki nöronlara ne kadarının iletilip iletilmeyeceğini düzenlemektedir. Bu kapılar sayesinde LSTM, gereksiz bilgileri unutup önemli olanları muhafaza edebilmektedir. Bu da uzun süreli bağımlılıkların öğrenilmesine olanak tanımaktadır.

Her LSTM birimi ayrıca ağın içinde bilgi taşıyan ve 'hücre durumu' olarak adlandırılan bir bileşene sahiptir. Bu hücre durumu, ağın içinde lineer bir yolla aktarılmaktadır. Yalnızca kapılar tarafından düzenlenmiştir. Bu özellik, bilginin uzun mesafeler boyunca neredeyse değişmeden taşınabilmesine ve dolayısıyla uzun süreli bağımlılıkların etkin bir şekilde öğrenilebilmesine olanak tanımıştır (Hochreiter ve Schmidhuber, 1997).

Şekil 4.13, LSTM birimlerinin mimarisini gösteren üç ayrı görseli içermektedir. Bu görseller, LSTM hücrelerinin evrimini ve kapılarını detaylı bir şekilde göstermektedir.



Şekil 4.13 : (a) Orijinal bir LSTM birim mimarisi: basit bir LSTM hücresi; (b) Unutma kapısı olan bir LSTM hücresi; (c) Unutma kapısı ile modern bir LSTM hücresinin gösterimi (Smagulova ve James, 2019).

İlk görsel (a), orijinal bir LSTM hücresinin mimarisini göstermektedir. Burada, hücrenin ana unsurları olan bir bellek hücresi s_c ve iki kapı yer almaktadır. Bu iki kapı giriş kapısı (input gate) ve çıkış kapısıdır (output gate). Giriş kapısı, giriş sinyalinin $g.y^{in}$ hücre durumuna ne kadar etki edeceğini kontrol ederken çıkış kapısı $h(s_c)$, hücre durumundan hangi bilgilerin dışarıya aktarılacağını düzenlemektedir. Bu iki kapı, hücrenin bilgiyi ne kadar süreyle saklayacağı ve ne zaman aktaracağı konusunda kritik rol oynamaktadır.

İkinci görsel (b), unutma kapısına (forget gate) sahip bir LSTM hücresini gösterir. Bu kapı, hücrenin önceki durumlarından hangi bilgilerin korunup hangilerinin silineceğini belirlemektedir. Bu da LSTM'nin uzun süreli bağımlılıkları daha etkili bir şekilde öğrenmesine olanak tanımaktadır. Unutma kapısı f_t, hücre durumuna s_c etki eden eski bilgiyi kontrol edererek modelin gereksiz bilgileri "unutmasını" sağlamaktadır.

Üçüncü görsel (c), unutma kapısıyla birlikte modern bir LSTM hücresinin temsili bir görselidir. Bu görselde, C_{t-1}'den, C_t'ye bilgi akışını gösteren hücre durumunu ve bu durum üzerinde operasyon yapan üç kapıyı içermektedir. Unutma kapısı f_t, giriş kapısı g_t ve çıkış kapısı o_t 'dir. Her kapı, giriş x_t, önceki çıktı h_{t-1} ve önceki hücre durumu C_{t-1} bilgilerini kullanarak hangi bilginin saklanıp hangisinin atılacağını ve hangi bilginin sonraki katmana geçeceğini belirlemektedir. Bu temsil, LSTM hücrelerinin

nasıl karmaşık bilgi akışını yönetebileceğini ve zamanla değişen verilere duyarlı olabileceğini görsel olarak açıklamaktadır.

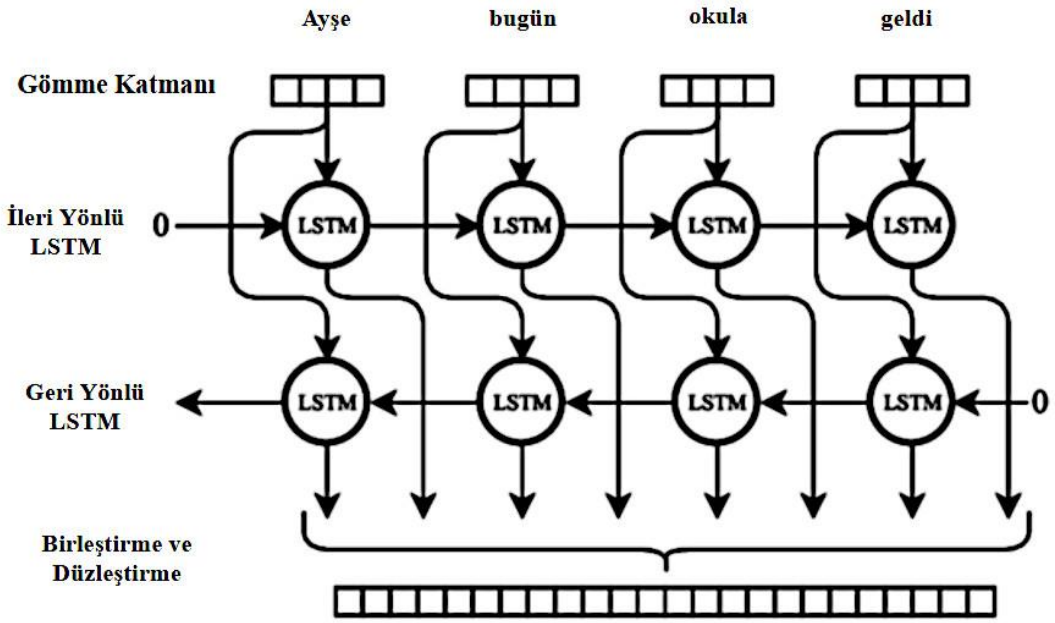
Sonuç olarak, LSTM ağları, RNN'lerin karşılaştığı kaybolan ve patlayan gradyan sorunlarına etkili bir çözüm sunmaktadır. Bu çözüm, karmaşık sıralı veri kümelerindeki uzun süreli bağımlılıkların öğrenilmesini mümkün kılmaktadır. Böylece ses tanıma, metin üretimi ve makine çevirisi gibi alanlarda gelişim sağlanmıştır. LSTM'lerin bu etkinliği, derin öğrenme modellerinin geliştirilmesi ve iyileştirilmesinde önemli bir dönüm noktası olmuştur.

4.2.1.3 Çift yönlü uzun kısa süreli bellek (Bidirectional LSTM)

Bidirectional Long Short-Term Memory (BiLSTM) mimarisi, Long Short-Term Memory (LSTM) ağlarının bir uzantısıdır. Zaman serisi verisinin hem geçmiş hem de gelecek bağlamını dikkate alarak daha etkili bir şekilde öğrenmesi için tasarlanmıştır. BiLSTM'ler, özellikle NLP görevlerinde, metnin anlamını hem sağdan sola hem de soldan sağa olacak şekilde çift yönlü bir bağlamda anlama yeteneği nedeniyle yaygın olarak kullanılmaktadır. BiLSTM, LSTM'in avantajlarını korurken verinin zamanla ilerleyen ve geriye doğru bilgilerini eş zamanlı olarak işleyebilen bir yapı sunmaktadır.

BiLSTM ağları, iki ayrı gizli katmana sahiptir. Biri ileri zaman yönünde (gelecek bilgiyi öğrenmek için) ve diğeri geri zaman yönünde (geçmiş bilgiyi öğrenmek için) bağımsız olarak işlem yapmaktadır. Her bir zaman adımında, iki gizli katman birbirinden bağımsız olarak aynı girdi verisi üzerinde çalışmaktadır. Sonunda, iki katmanın çıktıları birleştirilerek bir sonraki katmana veya çıktı katmanına iletilmektedir. Bu sayede ağ hem geçmiş hem de gelecek zaman bağlamını modellemektedir. Böylece daha zengin temsillerin elde edilmesini sağlamaktadır.

Mike Schuster ve Kuldip K. Paliwal tarafından 1997 yılında tanıtılan BiLSTM, özellikle kelimelerin bağlamını anlamada ve metin içindeki bilgileri çıkarmada etkili bir rol oynamaktadır. Örneğin, bir kelimenin birden fazla anlama gelebilmesi durumunda, kelimenin hem önceki hem de sonraki kelimelerle olan ilişkisini dikkate alarak doğru anlamının belirlenmesinde kullanılmaktadır (Schuster ve Kuldip, 1997). Şekil 4.14, bir çift yönlü uzun kısa süreli bellek (BiLSTM) ağının mimarisini göstermektedir.



Şekil 4.14 : BiLSTM mimarisi (Tam ve diğ, 2022).

BiLSTM mimarisi, ayrıca sentaks analizi, metin özetleme, duygu analizi ve adlandırılmış varlık tanıma gibi görevlerde başarıyla uygulanmıştır (Li ve diğ, 2022; Liang ve diğ, 2022; Xu ve diğ, 2019; Rajendran ve Murugan, 2022). BiLSTM'lerin başarısı, zaman serisi verisine yönelik bu çift yönlü yaklaşımın, özellikle sıralı verilerin karmaşıklığını modellemede gösterdiği üstün performansla doğrudan ilişkilidir. BiLSTM mimarisi, derin öğrenme ve NLP araştırmalarında önemli bir yer tutmaktadır. Bu modelin varyasyonları, daha karmaşık görevler için farklı mimarilerle birleştirilerek gelişmiş performans sergilemektedir (Rajendran ve Murugan, 2022; Rhanoui ve diğ, 2019; Kavianpour ve diğ, 2023). Sonuç olarak, BiLSTM'ler, sıralı verilerin çift yönlü öğrenilmesi gerektiren her türlü görevde, gelişmiş doğruluk ve etkinlik sunmaktadır.

4.2.1.4 Zaman serileri ve sıralı veri analizi için derin öğrenme mimarilerinin karşılaştırılması

Zaman serileri ve sıralı veri analizi, yapay zeka ve makine öğrenimi alanlarında oldukça önemli bir yere sahiptir. Özellikle derin öğrenme, bu veri tiplerini işlemek için çeşitli mimariler sunmaktadır. Bu mimarilerden bazıları RNN, LSTM ve BiLSTM'dir. Her bir mimari, benzersiz özellikler ve farklı avantajlar sunarak belirli türdeki problemlere özel çözümler sağlamaktadır. Bu başlıkta, bu üç popüler mimari, hafıza kapasiteleri, uzun süreli bağımlılıkların öğrenilmesi, hesaplama karmaşıklıkları ve

uygulama alanları gibi kritik özellikler açısından karşılaştırılmıştır. Bu karşılaştırma ile hangi mimarinin belirli bir uygulama veya problem için en uygun olduğunun anlaşılması amaçlanmıştır.

Hafıza kapasitesi ve uzun süreli bağımlılıkların öğrenilmesi açısından RNN, LSTM ve BiLSTM arasında önemli farklar bulunmaktadır. RNN, sıralı veri analizinde temel bir yaklaşım sunmaktadır. Kısa süreli bağımlılıkları öğrenmede oldukça etkilidir. Ancak RNN mimarisi uzun süreli bağımlılıkları öğrenmede zayıflıklar göstermektedir (Williams ve Zipser, 1989). Bu zayıflık, özellikle uzun dönemli veri dizileri üzerinde çalışırken ortaya çıkar ve modelin performansını sınırlamaktadır. Bunun yanı sıra, LSTM bu problemin üstesinden gelmek için özel olarak geliştirilmiştir (Hochreiter ve Schmidhuber, 1997). Karmaşık hafıza yapıları sayesinde LSTM, uzun süreli bağımlılıkları etkin bir şekilde öğrenebilmekte ve bu sayede daha karmaşık zaman serisi verilerinde daha iyi sonuçlar elde edebilmektedir. BiLSTM ise, LSTM'in sunduğu avantajların üzerine ek olarak hem geçmiş hem de gelecek verileri aynı anda işleyebilme kapasitesine sahiptir (Schuster ve Kuldip, 1997). Bu iki yönlü yapı, özellikle doğal dil işleme gibi alanlarda, verinin hem önceki hem de sonraki bağlamını dikkate alarak daha kapsamlı bir öğrenme süreci sağlamaktadır. Bu özellikler, her üç modelin de zaman serisi ve sıralı veri analizi için farklı derecelerde uygunluk göstermesine yol açmaktadır.

Kaybolan gradyan problemi, derin öğrenme modellerinin karşılaştığı önemli bir zorluktur. RNN mimarisi, kaybolan gradyan problemine oldukça hassastır. Bu sorun, ağın derin katmanlarında gradyanların giderek azalması ve böylece ağın başlangıç katmanlarında etkili bir şekilde öğrenememesi anlamına gelmektedir. Bu durum, özellikle uzun dönemli bağlantıların ve bağımlılıkların öğrenilmesini zorlaştırmaktadır. Diğer yandan, LSTM ve BiLSTM mimarileri, bu problemi hafifletmek amacıyla özel olarak tasarlanmıştır. Bu yapılar, hücre içindeki kapı mekanizmaları sayesinde uzun süreli bilgiyi korumaktadır. Bilgiyi ağın daha derin katmanlarına etkili bir şekilde aktarabilmektedir. Sonuç olarak, LSTM ve BiLSTM, kaybolan gradyan sorununa karşı daha dayanıklı yapılara sahip olup uzun dönemli bağımlılıkları öğrenme konusunda RNN'ye göre daha başarılıdır. Bu özellikler, özellikle karmaşık sıralı verilerin işlenmesinde bu mimarilerin tercih edilmesinin temel nedenlerinden biridir.

Hesaplama karmaşıklığı ve eğitim süresi açısından bakıldığında, RNN, LSTM ve BiLSTM arasında belirgin farklar bulunmaktadır. RNN'nin görece basit yapısı, bu modelin daha az hesaplama gücü gerektirmesine olanak tanımaktadır. Özellikle donanım kaynaklarının sınırlı olduğu durumlarda RNN'nin tercih edilmesine neden olmaktadır. Ancak, bu basitlik, modelin öğrenme kabiliyetini sınırlamaktadır. Özellikle karmaşık sıralı verileri işlemede yetersiz kalmaktadır. Buna karşılık, LSTM, RNN'ye göre daha karmaşık bir yapıya sahiptir. Bu karmaşık yapı, daha fazla hesaplama gücü ve uzun eğitim süreleri gerektirmektedir. Ancak bu ek yük, modelin uzun süreli bağımlılıkları ve karmaşık desenleri öğrenme yeteneğini önemli ölçüde artırmaktadır. BiLSTM ise LSTM'in iki yönlü bir versiyonu olarak daha da karmaşık bir yapıya sahiptir. Bu iki yönlü yapı, modelin hem geçmiş hem de gelecek verileri dikkate alarak daha kapsamlı bir öğrenme gerçekleştirmesine imkan tanımaktadır. Ancak bu, daha fazla hesaplama gücü ve daha uzun eğitim süreleri anlamına gelmektedir. Sonuç olarak, bu üç model arasındaki tercih, kullanılabilir hesaplama kaynakları, eğitim süresi ve ihtiyaç duyulan öğrenme derinliği gibi faktörlere bağlı olarak değişiklik göstermektedir.

Esneklik ve uygulama alanları bakımından RNN, LSTM ve BiLSTM, farklı ihtiyaçlara ve senaryolara hizmet eden özellikler sunmaktadır. RNN, basit zaman serisi tahminleri ve kısa sıralı veri işleme görevleri için ideal bir çözüm sunmaktadır. Yapısının basitliği, daha az karmaşık veri setlerinde hızlı ve etkili sonuçlar alınmasını sağlamaktadır. Bu da RNN'yi hızlı prototipleme ve basit sıralı veri analizleri için uygun hale getirmektedir. Diğer yandan, LSTM, doğal dil işleme, karmaşık zaman serisi tahminleri ve uzun metinlerin işlenmesi gibi daha zorlu uygulamalar için tercih edilmektedir. Uzun süreli bağımlılıkları öğrenebilme yeteneği ve gelişmiş hafıza yönetimi, LSTM'i bu tür görevlerde daha güçlü kılmaktadır. BiLSTM ise, LSTM'in sunduğu özelliklere ek olarak hem geçmiş hem de gelecek verilere dayalı görevlerde üstün performans göstermektedir. Bu iki yönlü yaklaşım, metin sınıflandırma, duygu analizi ve konuşma tanıma gibi alanlarda, verinin hem önceki hem de sonraki bağlamını dikkate alarak daha kapsamlı analizler yapılmasını sağlamaktadır. Her bir modelin bu esneklik ve uygulama alanları, onları belirli türdeki veri setleri ve problemler için özelleştirilmiş çözümler haline getirmektedir. Bu nedenle, bir projede hangi modelin kullanılacağının belirlenmesi, uygulanacak görevin özellikleri ve gereksinimleri dikkate alınarak yapılmaktadır.

Sonuç olarak, RNN, LSTM ve BiLSTM, zaman serileri ve sıralı veri analizinde farklı özellikler ve kapasiteler sunarak çeşitli uygulama alanlarına hitap etmektedir. RNN, basit ve kısa sıralı veriler için ideal olup hızlı ve verimli çözümler sunarken LSTM uzun süreli bağımlılıkları ve daha karmaşık veri yapılarını modellemekte üstün performans sergilemektedir. BiLSTM ise hem geçmiş hem de gelecek veri noktalarına dayalı analizlerde, özellikle doğal dil işleme ve metin sınıflandırma gibi alanlarda, daha kapsamlı ve bağlamsal anlayış sunmaktadır. Bu üç modelin seçimi, projenin gereksinimleri, veri setinin karmaşıklığı ve istenen çıktıların doğası göz önünde bulundurularak yapılmaktadır. Her modelin kendine has avantajları ve sınırlamaları, onları belirli türdeki problemler için daha uygun hale getirmektedir. Bu farklılıkların anlaşılması, zaman serisi ve sıralı veri analizi projelerinde başarı sağlamaktadır.

4.2.2 Büyük dil modelleri (Large language models- LLM)

Büyük dil modelleri, yapay zeka ve doğal dil işleme (NLP) alanlarında son yıllarda önemli bir gelişme olmuştur. Bu modeller, insan dilini anlamak ve üretmek için karmaşık algoritmalar ve büyük veri setleri kullanılmaktadır. Bu modellerin gelişimi, son birkaç on yılda NLP alanında yaşanan teknolojik ilerlemelerin bir sonucudur. Milyonlarca veya hatta milyarlarca parametre içeren bu modeller, derin öğrenme tekniklerini kullanarak büyük metin veri setlerinden dilin yapısal ve bağlamsal özelliklerini öğrenmektir. Bu modeller, genellikle katmanlı yapay sinir ağlarından oluşmaktadır. Sözcüklerin ve cümlelerin anlamları ile dilin bağlamını öğrenmek üzere tasarlanmış bu katmanlar, büyük metin korpuslarından elde edilen bilgileri kullanarak anlamlı çıktılar üretmektedir. Büyük dil modellerinin öncüleri arasında OpenAI tarafından geliştirilen GPT serisi yer almaktadır (Brown ve diğ, 2020). GPT-3, bu serinin en ünlü örneği olup, milyarlarca parametre içeren devasa bir modeldir. Google'ın BERT (Devlin ve diğ, 2018) ve diğer Transformer tabanlı modelleri de önemlidir.

Bu modeller, genellikle geniş ve çeşitli metin veri setleri kullanılarak eğitilmektedir. Eğitim süreci, dilin karmaşıklığını anlamayı ve çeşitli dil yapısını öğrenmeyi sağlamaktadır. Bu modeller, metin üretimi, otomatik çeviri, duygu analizi, metin özetleme ve sohbet robotları gibi çeşitli alanlarda kullanılmaktadır. İnsan dilini taklit edebilme ve anlamlı diyaloglar oluşturma yetenekleri ile dikkat çekmektedir.

Büyük dil modellerinin gelişimi, yanlılık ve hata eğilimleri, hesaplama gücü ve enerji tüketimi gibi zorlukları beraberinde getirmiştir. Modelin eğitiminde kullanılan veri setlerinin kalitesi ve çeşitliliği, bu zorlukların üstesinden gelmede önemli rol oynamaktadır.

4.2.3 Büyük dil modelleri ile uzun kısa süreli bellek yöntemlerinin karşılaştırılması

LLM ve LSTM modelleri, yapay zeka ve NLP alanlarında kullanılan iki farklı teknolojidir. Ancak her birinin özellikleri ve kullanım alanları açısından önemli farklılıklar bulunmaktadır.

Büyük dil modelleri, genellikle transformer tabanlı yapılardır. Bu yapılar, çok katmanlı dikkat mekanizmaları ve paralel işleme yetenekleri ile karakterize edilmiştir. LLM'lerin devasa boyutları (milyarlarca parametre), onların geniş ve çeşitli metin veri setlerinden karmaşık dil örüntülerini öğrenmesine olanak tanımaktadır. LSTM'ler, özellikle sıralı veri için tasarlanmış RNN'nin bir çeşididir. Bu modeller, uzun süreli bağımlılıkları öğrenebilmektedir. Gradyan kaybolması problemine karşı dirençlidir. Bu sayede sıralı veri görevlerinde daha karmaşık bağlamları anlamaktadır.

LLM'ler, metin üretimi, otomatik çeviri, soru-cevap sistemleri gibi çeşitli NLP görevlerinde yaygın olarak kullanılmaktadır. Bu modeller, dilin anlamını ve bağlamını geniş ölçekte anlamak için tasarlanmıştır. LSTM'ler ise daha çok zaman serisi verilerini işlemek, metin ve konuşma tanıma, finansal zaman serisi tahminleri gibi sıralı veri gerektiren görevlerde kullanılmaktadır. Bu modeller, verinin zamanla nasıl değiştiğini ve uzun süreli bağımlılıklarını öğrenmek için uygundur.

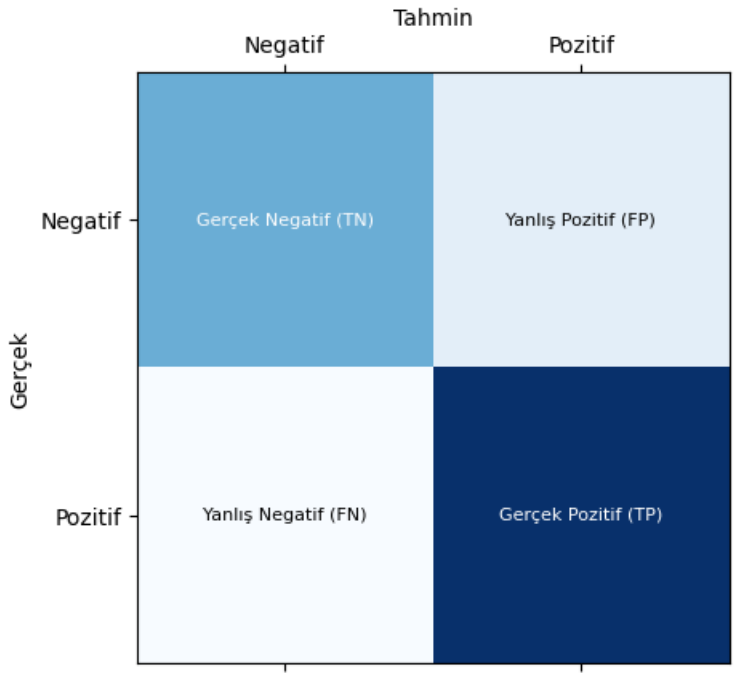
LLM'ler, özellikle GPT-3 gibi modeller, milyarlarca parametre içerdiğinden eğitimi için büyük miktarda hesaplama kaynağı gerektirmektedir. Bu hem enerji tüketimi hem de donanım ve altyapı maliyetleri açısından oldukça yüksektir. LSTM modelleri, LLM'lere kıyasla daha az parametre içermektedir. Bu nedenle eğitimleri için daha az hesaplama gücü gerektirmektedir. Bu, genellikle daha düşük enerji tüketimi ve altyapı maliyeti anlamına gelmektedir. LLM'lerin işletme maliyetleri, modelin boyutuna ve kullanım sıklığına bağlı olarak yüksek olmaktadır. Büyük modeller, özellikle sürekli olarak kullanıldığında önemli miktarda işlem gücü ve bellek gerektirmektedir. LSTM'ler, genellikle daha küçük boyutları sayesinde işletme maliyetleri açısından

daha verimlidir. Daha az kaynak kullanımı, özellikle sınırlı kaynaklara sahip uygulamalar için önemli bir avantajdır.

Sonuç olarak, LLM'ler ve LSTM'ler, yapay zeka ve NLP alanlarında farklı problemleri çözmek için kullanılan güçlü teknolojilerdir. Her bir modelin avantajları, yapısal özellikleri ve uygulama alanları, belirli görevlerin gereksinimlerine ve doğasına göre değerlendirilmektedir. Her bir teknolojinin özellikleri, onları belirli türdeki problemler için daha uygun hale getirmektedir. Bu farklılıkların anlaşılması, doğru teknolojinin seçiminde kritik öneme sahiptir.

4.3. Performans Metrikleri

Performans metrikleri makine öğrenimi ve derin öğrenme modellerinin başarısını değerlendirmek için kullanılan ölçütlerdir. Bu metrikler, modellerin ne kadar doğru tahminler yaptığını, belirli görevlerdeki performansını ve genel etkinliklerini belirlemek için kullanılmaktadır. En yaygın performans metrikleri arasında doğruluk, F1 skoru, hatırlama ve kesinlik gibi ölçütler bulunmaktadır. Bu metrikler, modelin performansını kapsamlı bir şekilde analiz etmek ve iyileştirmeleri yönlendirmek için kritik öneme sahiptir.



Şekil 4.15 : Karışıklık matrisi gösterimi.

Şekil 4.15, bir karışıklık matrisinin temsili bir gösterimidir. Karışıklık matrisi, bir sınıflandırma modelinin performansını değerlendirmek için kullanılan bir araçtır. Bu matris, modelin tahminlerinin gerçek etiketlere göre nasıl dağıldığını görsel olarak göstermektedir. Matrisin her hücresi, dört farklı olası sonuç kategorisini temsil etmektedir. Gerçek Negatif (True Negative- TN), modelin negatif olarak doğru tahmin ettiği durumları göstermektedir. Yani, modelin negatif sınıf olarak doğru etiketlediği veri noktaları bu hücrede yer almaktadır. Yanlış Pozitif (False Positive- FP), modelin pozitif olarak yanlış tahmin ettiği durumları göstermektedir. Diğer bir deyişle, aslında negatif olan bir durumu modelin pozitif olarak yanlış sınıflandırdığı durumlar bu hücrede yer almaktadır. Yanlış Negatif (False Negative- FN), modelin negatif olarak yanlış tahmin ettiği durumları göstermektedir. Yani, aslında pozitif olan bir durumu modelin negatif olarak yanlış sınıflandırdığı durumlar bu hücrede yer almaktadır. Gerçek Pozitif (True Positive - TP), modelin pozitif olarak doğru tahmin ettiği durumları göstermektedir. Yani, modelin pozitif sınıf olarak doğru etiketlediği veri noktaları bu hücrede yer almaktadır.

Yatay eksendeki "Tahmin" başlığı altında modelin tahmin ettiği sınıflar (Negatif veya Pozitif) ve dikey eksendeki "Gerçek" başlığı altında ise gerçek sınıflar yer almaktadır. Bu matris, modelin doğruluğunu, kesinliğini, hatırlama oranını ve F1 skorunu hesaplamak için bir temel sağlamaktadır.

4.3.1 Doğruluk (Accuracy)

Doğruluk, modelin tahminlerinin ne kadarının gerçek etiketlerle eşleştiğini gösteren bir metriktir. Kısaca, doğru tahminlerin toplam tahminlere oranı hesaplanmaktadır. Bu metrik, genellikle sınıflandırma görevlerinde kullanılmaktadır. Modelin genel performansını ölçmede etkili bir yoldur. Denklem 4.26'da verilmiştir.

$$\text{doğruluk} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{4.26}$$

4.3.2 Kesinlik (Precision)

Modelin doğru olarak tahmin ettiği pozitif örneklerin, toplam pozitif olarak tahmin edilen örnekler içindeki oranını ifade etmektedir. Başka bir deyişle, modelin ne kadar "kesin" olduğunu göstermektedir. Yani, modelin verdiği pozitif tahminlerin ne kadarının gerçekten pozitif olduğunu ölçmektedir.

$$\text{kesinlik} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.27}$$

4.3.3. Duyarlılık (Recall)

Gerçekten pozitif olan örneklerin, model tarafından doğru olarak ne kadarının pozitif olarak tahmin edildiğini göstermektedir. Temel olarak, modelin pozitif durumları ne kadar iyi "hatırladığını" veya tespit ettiğini ifade etmektedir. Denklem 4.28'de verilmiştir.

$$\text{duyarlılık} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4.28}$$

4.4.4 F1 Skoru

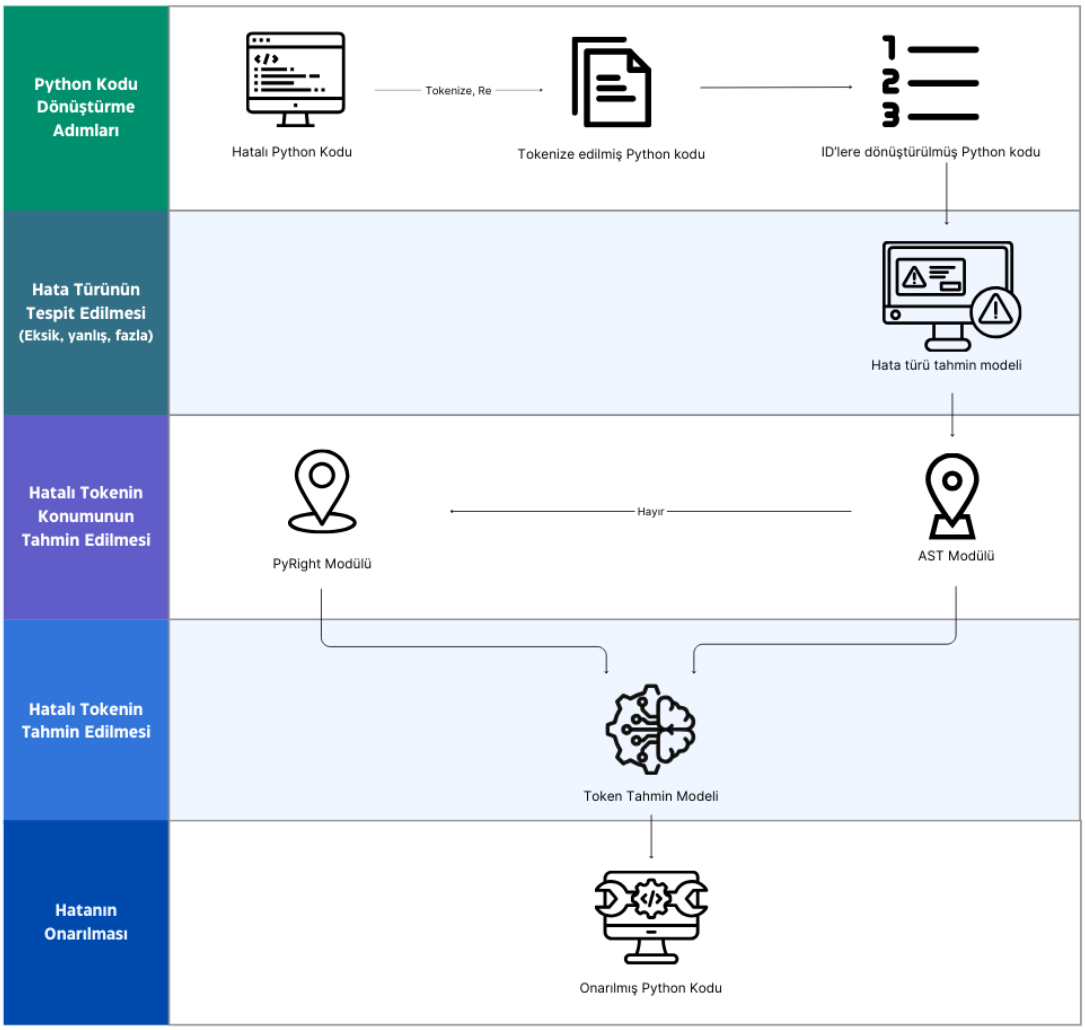
F1 Skoru, modelin doğruluğunu ölçmek için kullanılan bir başka metriktir. Özellikle dengesiz veri setleri için yararlıdır. Precision (kesinlik) ve recall (duyarlılık) metriklerinin harmonik ortalaması olarak hesaplanmaktadır. F1 Skoru, bir modelin hem yanlış pozitifleri (yanlış şekilde doğru olarak sınıflandırılan örnekler) hem de yanlış negatifleri (yanlış şekilde yanlış olarak sınıflandırılan örnekler) ne kadar iyi yönettiğini göstermektedir. Bu metrik, özellikle pozitif ve negatif sınıfların dengesiz olduğu durumlarda yararlıdır. Denklem 4.29'da verilmiştir.

$$F_1\ Skoru = 2.\frac{kesinlik.duyarlılık}{kesinlik + duyarlılık} \tag{4.29}$$

5. BULGULAR VE YORUMLAR

Bulgular ve yorumlar başlığı, araştırmamızın somut sonuçlarını ve bu sonuçların analizini içermektedir. Bu kısımda, kullanılan veri setinin detaylı bir incelemesi, ön işleme adımları ve özellikle düzenli ifadeler kullanılarak yapılan veri ayıklama ve numaralandırma işlemleri sunulmuştur. Model seçimi ve farklı modellerin karşılaştırılması ile hata türlerinin saptanmasına yönelik çalışmalar da detaylandırılmıştır. Hatalı tokenlerin tespiti, düzeltilmesi ve bu süreçte uygulanan metodolojiler, PyNar Sohbet Robotu'nun hata düzeltme öneri sistemine dair bilgiler de bu bölümde yer almaktadır. Test sonuçları, mutasyon testleri, veri setindeki hata dağılımı analizleri ve sınıflara göre ayrıntılı test sonuçları ile bu sonuçların kapsamlı karşılaştırmaları da bu bölümün önemli parçalarını oluşturmaktadır. Bütün bu çalışmalar, araştırmanın güvenilirliğinin ve tekrarlanabilirliğinin artırılmasını amaçlamıştır.

Şekil 5.1, bu tez çalışması kapsamında geliştirilen hatalı Python kodunun otomatik düzeltilme işlemini adımsal bir süreç olarak görselleştiren bir akış diyagramıdır.



Şekil 5.1 : Geliştirilen otomatik kod onarımı sistemi.

İlk adımda, kodun tokenizasyonu yapılmaktadır. Sonrasında bu tokenler, işlemlerin devamı için benzersiz tanımlayıcı ID'lere dönüştürülmektedir. Bu işlemden sonra, hata türünü belirleyen model devreye girmektedir. Bu model hata türünü (eski, yanlış, fazla token gibi) tespit etmektedir. Hata bulunursa, AST modülü aktif hale gelerek hatalı tokenin yerini tahmin etmektedir. Eğer hatalı tokenin konumu bulunamazsa, PyRight modülüne yönlendirilmektedir. Daha sonra, token tahmin modeli hatalı tokenin ne olması gerektiğine dair bir tahminde bulunmaktadır. Son aşamada ise, önerilen doğru token ile hatalı token değiştirilerek düzeltilmiş Python kodu elde edilmektedir. Bu süreç kod onarımının her bir fazında kullanılan yöntemleri ve modülleri açıkça ortaya koymaktadır.

5.1 Veri Seti

Bu çalışmada Türkiye Milli Eğitim Bakanlığı'nın Python eğitim kitabından (Altun, 2020), GitHub depolarından ve çeşitli Python öğretici içeriklerinden derlenmiş temel seviye Python programlama unsurlarını içeren 2000 farklı kod örneğinden oluşan bir veri seti modelin eğitimi için kullanılmıştır. Test veri seti için ayrıca 300 farklı kod örneğinden oluşan bir veri seti oluşturulmuştur. Örnekler, Python'un temel yapı taşlarını (koşul ifadeleri, döngüler, fonksiyonlar, sözlükler, demetler, listeler ve dizileri) içerecek şekilde tasarlanmıştır. Her iki veri setinde yer alan doğru yazılmış Python kod örnekleri, mutasyon testi yöntemi kullanılarak söz dizimi bakımından hatalı hale getirilmiştir. Mutasyon testi, yazılım test etmenin ileri düzey bir yöntemidir. Yazılımın ne kadar güvenilir olduğunu değerlendirmek için kullanılmaktadır. Bu yöntem, yazılımın kaynak kodunda küçük ve rastgele değişiklikler yapmaktadır (mutasyonlar). Bu mutasyonlar aracılığıyla yazılımın davranışı test edilmektedir. Her bir mutasyon sonucunda oluşturulan alternatif kod versiyonlarına "mutant" denmektedir. Bu mutantlar, orijinal kodun farklı versiyonları olarak düşünülmektedir.

Kod yazımında sık karşılaşılan hatalar, genellikle eksik yazım, yanlış yazım ve fazla yazım olarak sınıflandırılabilir. Bu tür hatalar, yazılım geliştirmenin doğal bir parçasıdır. Genellikle programcıların dikkatsizliği veya yanılgıları sonucu meydana gelmektedir. Bu hataları daha iyi anlamak, tespit etmek ve onarmak için yapılan bu çalışmada, 2000 adet veri modelin eğitimi ve 300 adet sistemin test edilmesi için oluşturulan test verisi bu tür hatalarla mutasyon testine tabi tutulmuştur.

Eksik yazım hataları, genellikle programcının bir kod parçasını tamamen veya kısmen atlaması sonucu oluşmaktadır. Örneğin, bir fonksiyon çağrısında parantez unutmak veya bir döngü içinde gerekli bir artırma ifadesini eklememek gibi durumlar bu kategoriye girmektedir.

Yanlış yazım hataları, programcının kodu yanlış bir şekilde yazdığı durumlarda ortaya çıkmaktadır. Değişken adlarının yanlış yazılması, yanlış operatör kullanımı veya dilin söz dizimi kurallarına uymayan ifadeler olarak değerlendirilmektedir.

Fazla yazım hataları ise programcının gereksiz veya yanlış kod parçalarını eklemesi ile meydana gelmektedir. Örneğin, gereksiz bir değişken ataması veya gereksiz parantez eklenmesi bu kategoriye girmektedir. Bu tür hatalar, kodun karmaşıklığını ve hata riskini artırmaktadır. Eğitim ve test verilerinde bu tür eksikliklerin, yanlışlıkların

veya fazlalıkların temsilleri ile oluşturulması otomatik hata tespit sistemlerinin bu tür hataları nasıl algılayacağını öğrenmesine yardımcı olmaktadır.

2000 eğitim ve 300 test verisinin mutasyon testi ile değiştirilmesi bu tür yaygın hataları tespit etme ve düzeltme becerilerini geliştirmek için önemli bir adımdır. Veriler; rastgele konumdaki rastgele bir tokenin silinmesi, yanlış yazılmış hale getirilmesi veya farklı bir token eklenmesi ile hatalı hale getirilmiştir. Bu süreç, hem programcıların kodlama becerilerini artırmalarına yardımcı olmakta hem de hata tespit sistemlerinin daha etkin ve verimli hale gelmesine katkıda bulunmaktadır.

Şekil 5.2, doğru yazılmış bir Python kod örneğini göstermektedir. Bu kod parçasında, bir numbers isimli liste tanımlanmıştır. While döngüsü kullanılarak listenin her bir elemanı sırayla ekrana yazdırılmıştır. index, 0'dan başlayarak her adımda bir artırılmaktadır. Bu sayede listenin tüm elemanları sırayla print fonksiyonu ile ekrana basılmaktadır. Kodun bu hali, beklenen çıktıyı üretecek şekilde doğru bir biçimde yazılmıştır.

testwhile27.py

```
1  numbers = [2, 4, 6, 8, 10]
2  index = 0
3  while index < len(numbers):
4      print(numbers[index])
5      index += 1
```

Şekil 5.2 : Söz dizimi açısından doğru yazılmış Python kodu örneği.

Şekil 5.3, mutasyon testi ile rastgele konumdaki rastgele bir tokenin değiştirilmesi ile bir söz dizimi hatası içeren Python kodunu göstermektedir. Burada, print fonksiyonu parantez içinde kullanılırken bir parantez yanlış yazılmıştır. Bu, parantezlerin doğru eşleşmediği anlamına gelmektedir. Python yorumlayıcısı tarafından bir hata olarak tespit edilmektedir.

testwhile27 (1).py

```
1  numbers = [2, 4, 6, 8, 10]
2  index = 0
3  while index < len(numbers):
4      print(numbers[index]]
5      index += 1
```

Şekil 5.3 : Mutasyon testi ile söz dizimi açısından hatalı hale (yanlış yazım) getirilmiş kod örneği.

Şekil 5.4, yine bir söz dizimi hatası içeren mutasyon testi ile rastgele bir konumdaki rastgele bir tokenin silinmesi ile elde edilmiş başka bir Python kod örneğini göstermektedir. Bu örnekte ise print fonksiyonunun ardından gelmesi beklenen sol parantez eksik bırakılmıştır. Sağ parantez eşleşmediğinden söz dizimi hatası oluşmaktadır ve bu hata kodun çalışmasını engellemektedir.

testwhile27 (2).py

```
1  numbers = [2, 4, 6, 8, 10]
2  index = 0
3  while index < len(numbers):
4      printnumbers[index])
5      index += 1
```

Şekil 5.4 : Mutasyon testi ile söz dizimi açısından hatalı hale (eksik yazım) getirilmiş kod örneği.

Şekil 5.5 ise mutasyon testi ile rastgele bir konuma rastgele bir token eklenerek oluşturulmuş fazla bir parantez içeren bir kod örneğini göstermektedir. Bu durum parantezlerin eşleşmemesinden dolayı söz dizimi hatası oluşturmaktadır.

testwhile27 (3).py

```
1  numbers = [2, 4, 6, 8, 10]
2  index = 0
3  while index < len(numbers):
4      print(numbers[index]))
5      index += 1
```

Şekil 5.5 : Mutasyon testi ile söz dizimi açısından hatalı hale (fazla yazım) getirilmiş kod örneği.

Bu örnekler, mutasyon testinin söz dizimi hatalarını tespit etme ve düzeltilmesi gereken alanları belirleme konusunda nasıl kullanılabileceğini göstermektedir. Her bir mutasyon, kodun belirli bir şekilde bozulduğu bir senaryoyu temsil etmektedir.

Çizelge 5.1'de hata türü sınıflarına göre mutasyon testi ile oluşturulmuş hatalı kod içeren test veri setine ait hata dağılımları verilmiştir.

Çizelge 5.1: Test veri seti için hata mesajına göre hata dağılımları.

Hata Dağılımı	Eksik	Fazla	Yanlış
‘x’ was never closed	61	74	0
cannot assign to function call	12	0	0
unexpected EOF while parsing	37	0	0
invalid syntax	98	138	167
unmatched ‘x’	53	88	0
closing parenthesis 'x' does not match opening parenthesis 'y'	3	0	133
unterminated string literal	32	0	0
Toplam	300	300	300

“invalid syntax” (geçersiz söz dizimi) hata mesajı, her üç hata türü sınıfında da en yaygın olarak karşılaşılan mesajdır. Eksik durumu genellikle bir ifade ya da komutun tamamlanmamasından kaynaklanmaktadır. Örneğin, bir for döngüsünde iki nokta üst üste (:) işaretinin unutulması ya da bir fonksiyonun parantez içinde uygun parametrelerle çağrılmaması gibi durumlar eksik kategorisine girmektedir. Yanlış, genellikle yazılan kodun Python dilinin kurallarına uymayan bir yapıda olması durumunu ifade etmektedir. Bu, bir değişkenin adının yanlış yazılması veya bir fonksiyonun yanlış argümanlarla çağrılması gibi durumlar olmaktadır. Fazla ise kod içerisinde gereksiz veya anlamsız eklemelerin yapılması durumudur. Örneğin, bir değişken ataması sonrasında gereksiz bir virgülün konulması veya bir ifadenin sonunda ekstra parantezlerin bulunması bu duruma örnek teşkil etmektedir.

“‘x’ was never closed” hata mesajı, genellikle bir tırnak işareti veya parantez gibi bir açılımın kapatılmadığını göstermektedir. Örneğin, bir string ifadesi başlatıldıktan sonra kapatılmamışsa bu hata ile karşılaşılmaktadır.

“cannot assign to function call” hatası, bir fonksiyon çağrısının sonucuna değer atamaya çalışıldığında ortaya çıkmaktadır. Python'da, fonksiyon çağrıları değer döndürür ve bu değerlere doğrudan atama yapılamaz.

"unexpected EOF while parsing" hatası, Python yorumlayıcısının bir ifadenin beklenenden daha erken sona erdiğini tespit ettiğinde ortaya çıkmaktadır. EOF, "End Of File" ifadesinin kısaltmasıdır ve bu hata genellikle bir kod bloğu, dize (string) veya parantez düzgün bir şekilde kapatılmadığında meydana gelmektedir. Yorumlayıcı, ifadenin sonunu belirleyecek bir token (örneğin, bir kapama parantezi veya alıntı işareti) beklerken, dosyanın sonuna veya girişin sonuna ulaştığını belirtmektedir.

“unmatched ‘x’” hatası, genellikle parantezler, tırnak işaretleri veya diğer sınırlayıcıların düzgün eşleşmediğini göstermektedir. Bir açma parantezi için bir kapatma parantezinin bulunmaması bu hataya sebep olmaktadır.

“closing parenthesis ‘x’ does not match opening parenthesis ‘y’” hatası, bir açma parantezi ile kapatma parantezi arasındaki eşleşmenin yanlış olduğunu belirtmektedir. Örneğin, bir köşeli açma parantezi ‘[’ ile normal bir kapatma parantezi ‘)’ kullanıldığında bu hata alınmaktadır.

“unterminated string literal” hatası ise bir string ifadesinin başlatıldığı ancak doğru bir şekilde kapatılmadığını gösteren bir hatadır. Genellikle tırnak işaretlerinin eksik olması sebebiyle ortaya çıkmaktadır.

"Tokenize" terimi, genellikle programlama dillerinde bir metin veya kod parçasını tokenlere ayıran bir süreç için kullanılmaktadır. Python'da ise tokenize modülü, Python kaynak kodunun tokenlere ayrılmasını sağlayan standart bir kütüphanedir (Python Software Foundation, 2001). Bu modül, Python kaynak kodunu okuyarak her bir token için bir dizi bilgi üretmektedir. Tokenize modülü, belirli bir metin dosyasından veya bir string'den gelen Python kaynak kodunu tokenize etmektedir. Bu tokenler; token tipi, string değeri, başlangıç satır ve sütun numarası, bitiş satır ve sütun numarası ve satırın kendisini içeren bilgileri içermektedir. Örneğin, bir for döngüsü ya da bir değişken ataması gibi yapılar, işlenebilir token dizilerine dönüştürülmektedir.

Python'ın ‘tokenize’ modülü Python dilinin söz dizimine uygun olan kaynak kodları tokenlere ayırmak için tasarlanmıştır. Ancak eğer kod söz dizimi bakımından hatalıysa ‘tokenize’ modülü bu kodu doğru bir şekilde işlememektedir. Bir hata oluşturmaktadır. Bu, ‘tokenize’ modülünün dilin söz dizimi kurallarına sıkı sıkıya bağlı olmasından kaynaklanmaktadır. Örneğin, eksik bir parantez tokenize işleminin başarısız olmasına neden olmaktadır.

Bu tür söz dizimi hatalarını işleme konusundaki sınırlılıkları aşmak için düzenli ifadeler (Regular Expressions- RegEx) kullanılmıştır. Regex, metin içindeki desenleri tanımlamak ve işlemek için kullanılan bir dil ve araçtır. Python'daki ‘re’ modülü (Python Software Foundation, 1994), regex işlevselliğini sunmaktadır. Metin üzerinde karmaşık arama ve değiştirme işlemleri yapılmasını sağlamaktadır.

Regex, metinlerde belirli desenleri aramak için özel karakterler ve meta-karakterler kullanmaktadır. Örneğin, ‘\d’ rakamları temsil etmektedir, ‘.’ herhangi bir karakterle

eşleşmektedir. ‘*’ ya da ‘+’ gibi nicelik belirleyiciler bir karakterin kaç kez tekrarlandığını belirtmektedir. Bu güçlü desen eşleştirme yeteneği, söz dizimi hatalarını veya kodda beklenmeyen desenleri tanıma ve işleme yeteneği sağlamaktadır.

Bu çalışmada, ‘tokenize’ modülü başarısız olduğunda veya daha esnek bir tokenizasyon gerektiğinde ‘re’ modülü kullanılarak tokenizasyon fonksiyonu yazılmıştır. Bu, özellikle hatalı veya standart dışı kod parçalarını analiz etmek, düzeltmek veya uygun hata mesajları üretmek için oluşturulmuştur. Örneğin bir regex ifadesi kullanılarak tırnak içindeki bir string veya parantez içindeki ifadeler tanımlanarak tokenize etmede başarısız olan yerler özel olarak işlenmiştir. Tokenize kütüphanesi tırnak içinde verilen string ifadeleri bir bütün olarak değerlendirip STRING tokenine dönüştürmektedir. String ifadelerden önce veya sonra gelen tırnak işaretleri için bir token oluşturmamaktadır. Tırnak içindeki string ifadelerin (tırnak, string, tırnak) ayrıştırılması için örnek kod satırı Şekil 5.6’da verilmiştir.

Şekil 5.6, Python programlama dilinde regex kullanarak bir metin üzerinde yer alan belirli bir deseni arayan ve bu deseni belirli bir metinle değiştirmeye yarayan bir kod satırını göstermektedir.

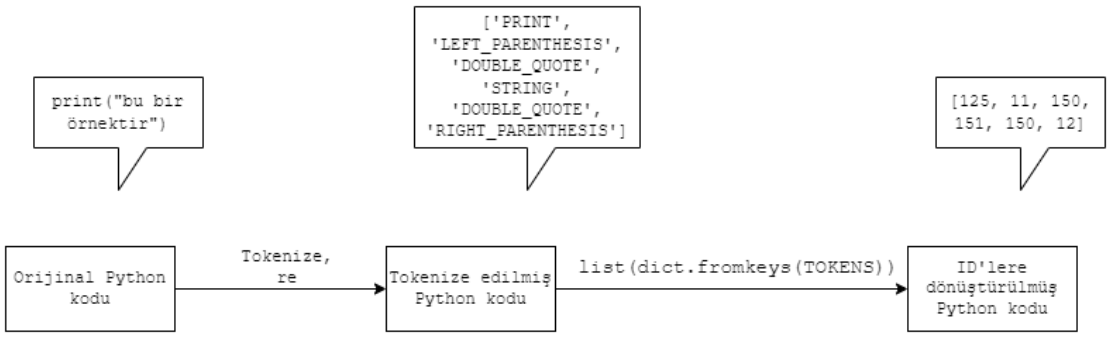
```
errorcode = re.sub(r' [\"\'][^\"][\w\d\s_-]+[^\"][\"\'] ',
                   ' DOUBLE_QUOTE STRING DOUBLE_QUOTE ', errorcode)
```

Şekil 5.6 : Örnek ‘re’ modulü kullanımı.

're.sub' metodu, bir string içindeki regex ile eşleşen tüm örnekleri başka bir string ile değiştirmek için kullanılmıştır. Bu kod parçası, çift tırnak içinde verilen düzenli ifadeye uyan kısımları "DOUBLE_QUOTE STRING DOUBLE_QUOTE" metni ile değiştirmek üzere yazılmıştır. Değiştirilen bu metin tokenize modülü ile tokenlerine ayrılmıştır.

Kısacası, ‘tokenize’ modülü ve ‘re’ modülü, Python kodunu analiz etmek ve işlemek için birlikte kullanıldığında söz dizimi hatalarını içeren kodlarla bile etkili bir şekilde çalışabilmektedirler. Bu da geliştiricilere büyük esneklik sağlamaktadır.

Şekil 5.7, Python kodunun tokenize edilerek tokenlerin ve bu tokenlerin karşılık geldiği benzersiz tanımlayıcıların (ID'ler) bir listesine dönüştürüldüğünü gösteren bir süreci betimlemektedir.



Şekil 5.7 : Python kodlarının ID'lere dönüştürülmesi.

Bu süreçte orijinal Python kodu örneğin, (print("Bu bir örnektir")), ilk olarak tokenize edilmektedir. Tokenizasyon, kodu mantıksal parçalara ayıran bir işlemdir. Kodun her bir parçası, Python dilinin söz dizimine göre tanınan bir token'e dönüştürülür. Örnekte, print fonksiyon adı, sol parantez, çift tırnak içindeki string ve sağ parantez gibi ögeler ayrı ayrı tokenlere ayrılmıştır. Daha sonra bu tokenler, her bir tokenin karşılık geldiği söz dizimsel unsurları temsil eden dize (string) etiketlere dönüştürülür. Örneğin print ifadesi 'PRINT' tokeni, parantezler 'LEFT_PARENTHESIS' ve 'RIGHT_PARENTHESIS' tokenleri gibi etiketlenmiştir.

Son adımda, bu etiketler daha sonra işleme veya analiz için kullanılabilecek benzersiz tanımlayıcılar (ID'ler) ile eşleştirilmiştir. Bu ID'ler sayısal değerlerdir ve genelde tokenlerin söz dizimsel sınıflarını veya fonksiyonlarını temsil ederler. Örneğin, 'PRINT' için 125, sol parantez için 11 gibi.

Bu süreç, derleyicilerin ve yorumlayıcıların, kaynak kodu derleme veya yorumlama sürecinde nasıl işlediğini gösteren bir örnektir. Ayrıca, yazılım geliştirme ortamları ve hata ayıklama araçları tarafından da kullanılan bir süreçtir. Bu görselleştirme (Şekil 5.7), Python kodunun daha yüksek düzeyde bir işleme veya analiz için nasıl yapılandırıldığını anlamak için oluşturulmuştur.

Python kodlarının ID'lere dönüştürülmesi, makine öğrenimi ve derin öğrenme modellerinin eğitimi için önemli bir adımdır. Bu süreçte, kaynak kod tokenlere ayrılmış ve her token benzersiz bir sayısal tanımlayıcı (ID) ile eşleştirilmiştir. Veri setinin eğitim için elde edilen hali, Python kodlarının ID'lere dönüştürülmüş sayısal temsillerinden oluşmaktadır. Bu sayısal temsil, modellerin kodu analiz etmesi ve öğrenmesi için daha uygun bir biçimdir çünkü sayısal ID'ler, modelin kodun yapısal ve söz dizimsel özelliklerini anlamasına ve örüntüleri tanımasına yardımcı olmaktadır.

5.2 Hata Türünün Tespit Edilmesi

Otomatik program onarımı, yazılım geliştirme sürecinin önemli ve zorlu bir bileşenidir. Geleneksel hata düzeltme yöntemleri genellikle zaman alıcı ve kaynak yoğun olduğundan otomatikleştirilmiş sistemlerin geliştirilmesi büyük bir ihtiyaç haline gelmiştir. Bu bağlamda geliştirilen sistem, yazılım hatalarını üç ana kategoriye ayırarak ele almaktadır. Bu kategoriler eksik kod parçaları, fazladan kod eklenmesi ve yanlış yazım hatalarıdır. Bu sınıflandırma, onarım sürecinin daha odaklı ve verimli olmasını sağlayarak hataların doğru bir şekilde tanımlanmasına ve düzeltilmesine imkan tanımaktadır.

Sistemin eğitimi için her bir hata türüne özgü olacak şekilde toplamda 6000 adet hatalı kod örneği üretilmiştir. Bu, her bir kategori için 2000 doğru yazılmış kod örneğinin mutasyon testleri uygulanarak kasıtlı olarak hatalı hale getirilmesiyle sağlanmıştır. Bu yöntem, gerçek dünyada karşılaşılabilecek hata senaryolarını simüle ederek sistemin daha geniş bir hata yelpazesiyle başa çıkabilmesine olanak tanımaktadır. Elde edilen bu geniş ve çeşitlendirilmiş veri seti üzerinde eğitilen model, otomatik program onarımı konusunda gelişmiş bir anlayış ve performans sergileme potansiyeline sahip olmuştur.

İlk modelin (hata türü modeli) geliştirilme amacı kullanıcı tarafından yazılan hatalı kod parçasının hangi sebeple hatalı olduğunun tespit edilmesi ve bu doğrultuda düzeltme yapılacak tokenin eklenmesi, silinmesi veya değiştirilmesi ile ilgili fonksiyonun çalıştırılmasını sağlamak olmuştur. Geliştirilen bu model, hata onarım sürecini tamamen otomatikleştirmeyi hedeflemektedir. Aksi takdirde eksik, fazla veya yanlış yazım gibi üç farklı hata durumu için işlemlerin manuel olarak tekrarlanması gerekmektedir. Otomatikleştirilmiş sistem sayesinde bu tür hataların tespiti ve düzeltilmesi süreçleri büyük ölçüde hızlandırılmış ve verimliliği artırılmıştır. Bu da otomasyonun, tekrar eden ve zahmetli hata onarım işlemlerini minimize ederek yazılım geliştirme süreçlerinde zaman ve kaynak tasarrufu sağladığının bir göstergesidir.

Modelin performansını en üst düzeye çıkarmak amacıyla çeşitli hiperparametre optimizasyonları gerçekleştirilmiştir. Nöron sayıları ve katman sayıları gibi ağın yapısını belirleyen parametrelerden başlayarak aktivasyon fonksiyonları, optimizasyon algoritmaları ve mini-küme (batch size) büyüklükleri gibi eğitim

sürecini etkileyen faktörlere kadar geniş bir yelpazede denemeler yapılmıştır. Bu denemelerin amacı, hatalı kodları en doğru şekilde tespit edebilen ve düzeltebilen bir model yapısını belirlemek olmuştur. Her bir deney, modelin hata türlerini anlama ve onları düzeltme kapasitesini artıracak şekilde titizlikle ayarlanarak ve test edilmiştir.

Çizelge 5.2, hata türü modeli için üç farklı yapay sinir ağı mimarisi olan RNN (Recurrent Neural Network), LSTM (Long Short-Term Memory) ve BiLSTM (Bidirectional Long Short-Term Memory) modellerinin performans karşılaştırmalarını sunmaktadır.

Çizelge 5.2: Hata türü modeli için yapay sinir ağı yöntemlerinin karşılaştırılması.

Parametreler ve Metrikler	RNN	LSTM	BiLSTM
Katman Sayısı	8	8	8
Aktivasyon Fonksiyonu	Softmax	Softmax	Softmax
Optimizasyon Fonksiyonu	Adam	Adam	Adam
Küme Boyutu (Batch Size)	128	128	128
Döngü Sayısı (Epoch)	50	50	50
Parametre Sayısı	375.811	1.459.459	3.953.667
Doğruluk	0.3319	0.8230	**0.9499**
Kayıp	1.1049	0.3855	**0.1480**

Her üç model de 8 katman kullanılarak oluşturulmuştur (nöron sayıları eşit). Bu da modellerin derinliği açısından eşit olduklarını göstermektedir. Tüm modellerde aktivasyon fonksiyonu olarak Softmax kullanılmıştır. Çıktı katmanında çok sınıflı sınıflandırma için olasılık dağılımı oluşturulduğunu ifade etmektedir. Her model için optimizasyon fonksiyonu olarak Adam optimizatörü tercih edilmiştir.

Her üç model de eğitim sırasında 128 örneklik mini-kümeler (batch) kullanarak güncellemeler yapmış bu da hesaplama verimliliği ve genelleme yeteneği açısından ortak bir stratejiyi işaret etmektedir. Ayrıca her model 50 epoch boyunca eğitilmiştir. Bu da her birinin eğitim verileri üzerinde aynı sayıda iterasyon yaptığını göstermektedir. Parametre sayıları; RNN için 375.811, LSTM için 1.459.459 ve BiLSTM için 3.953.667 olarak elde edilmiştir. BiLSTM'ın en yüksek parametre sayısına sahip olması modelin daha karmaşık olduğunu ve teorik olarak daha karmaşık örüntüleri öğrenebileceğini göstermektedir.

Doğruluk metriği, BiLSTM modelinin %94,99 ile en yüksek performans gösterdiğini, LSTM'in %82,30 ve RNN'nin %33,19 ile onu takip ettiğini göstermektedir. Bu, BiLSTM modelinin hata türlerini tanıma konusunda çok daha başarılı olduğunu

göstermektedir. Kayıp (loss) değerleri, BiLSTM için en düşük olan 0,1480, LSTM için 0,3855 ve RNN için 1,1049 olarak elde edilmiştir. Düşük kayıp değerleri, modelin eğitim verilerine göre daha iyi genelleme yaptığını ve beklenen çıktılara daha yakın sonuçlar ürettiğini ifade etmektedir.

Bu karşılaştırmalı analiz; BiLSTM modelinin, RNN ve LSTM modellerine kıyasla hem doğruluk hem de kayıp metriklerinde önemli ölçüde daha iyi performans gösterdiğini göstermektedir. Bu sebeple hata türü modeli için BiLSTM sinir ağı ile geliştirilen model tercih edilmiştir.

Çalışmada, kod gibi sıralı verilerin işlenmesi için çeşitli derin öğrenme katmanları kullanılmıştır. İlk olarak, embedding (gömme) katmanı, her bir tokeni veya kelimeyi, onları temsil eden yoğun vektörler haline getirerek bu verilerin daha karmaşık ilişkileri modelleyebilmesini sağlamıştır. Bu vektörler, modelin eğitim sürecinde güncellenmektedir. Her bir tokenin semantik anlamını yakalamaktadır. Ardından, BiLSTM katmanları kullanılmıştır. BiLSTM, hem geçmişe hem de geleceğe dönük bilgileri dikkate alarak sıralı verilerin bağlamını daha iyi anlamak için tasarlanmıştır. Dropout (düşürme) katmanı ise, ağın belirli nöronlara aşırı bağımlı hale gelmesini önleyerek aşırı uyumun (overfitting) önüne geçmektedir. Modelin her bir iterasyonunda rastgele seçilen nöronlar devre dışı bırakılmaktadır. Bu sayede modelin genelleştirme kabiliyeti artırılmaktadır. Son olarak, dense (yoğun) katmanlar, modelin çıktıları üretmek için öğrendiği özellikleri birleştirmektedir. Bu katmanlar, önceki katmanlardan gelen tüm sinyalleri toplayarak bir aktivasyon fonksiyonu aracılığıyla sonuçları sunmaktadır. Bu dört farklı katmanın entegrasyonu, özellikle otomatik program onarımı gibi karmaşık görevlerde, derin öğrenme modellerinin başarısının arkasındaki anahtar unsurlardandır.

Çizelge 5.3, farklı katman sayısına sahip hata türü modellerinin performans karşılaştırmasını sunmaktadır. Her bir deneyde, BiLSTM katmanlarının sayısı katman sayısı ile orantılı olarak arttırılmıştır. Bu artışın modelin doğruluğu ve kayıp değerleri üzerindeki etkisi incelenmiştir.

Çizelge 5.3: Hata türü modeli için katman sayılarının karşılaştırılması.

Parametreler ve Metrikler	Deney 1	Deney 2	Deney 3	Deney 4
Katman Sayısı	6	8	10	12
BiLSTM Sayısı	2	3	4	5
Aktivasyon F.	Softmax	Softmax	Softmax	Softmax
Optimizasyon F.	Adam	Adam	Adam	Adam
Küme Boyutu	128	128	128	128
Döngü Sayısı	50	50	50	50
Parametre Sayısı	2.378.755	3.953.667	5.528.579	7.103.491
Doğruluk	0,7961	0,9499	**0,9671**	0,9377
Kayıp	0,4428	0,1480	**0,0995**	0,2570

Deney 1, 6 katman (2'si BiLSTM katmanı) ile gerçekleştirilmiş ve model 2.378.755 parametreye sahipken %79,61 doğruluk ve 0,4428 kayıp değeri elde etmiştir. Deney 2, 8 katman (3'ü BiLSTM katmanı) kullanılarak yapılmıştır. Modelin parametre sayısı 3.953.667'ye çıkarken doğruluk %94,99'a yükselmiş ve kayıp değeri önemli ölçüde düşerek 0,1480 olmuştur. Deney 3 ise 10 katmana (4'ü BiLSTM katmanı) sahiptir. 5.528.579 parametre içermektedir. Bu yapı ile model %96,71 doğruluk ve 0,0995 kayıp değeri elde etmiştir. Bu da önceki deneylere göre daha yüksek bir doğruluk ve daha düşük bir kayıp anlamına gelmektedir. Deney 4, 12 katman (5'i BiLSTM katmanı) barındırmaktadır. Toplam 7.103.491 parametre ile en büyük modeldir. Ancak bu modelin doğruluğu %93,77'ye düşmüştür. Kayıp değeri 0,2570'ye çıkmıştır. Bu da modelin aşırı uyuma (overfitting) başladığının bir işareti olarak düşünülmektedir.

Genel olarak modelin karmaşıklığının artırılması belirli bir noktaya kadar doğruluğu artırırken kayıp değerini düşürmüştür. Ancak en karmaşık model olan Deney 4'te performansın azaldığı görülmektedir. Bu da daha fazla katmanın ve parametrenin her zaman daha iyi sonuçlar getirmediğini göstermektedir. Modelin performansı ve genelleştirme yeteneği arasında dengeli bir ilişki bulmak önemlidir. Bu dengeyi bulmak modelin eğitim ve validasyon verilerindeki performansının yanı sıra, bilinmeyen veriler üzerindeki genelleştirme kabiliyetine de bağlıdır.

Modelin performansını iyileştirmek için yapılan hiperparametre ayarlamaları kapsamında, çeşitli nöron sayılarına sahip farklı deneyler gerçekleştirilmiştir. Özellikle nöron sayısının her katmandaki dağılımı, modelin karmaşıklığını ve öğrenme kapasitesini doğrudan etkileyen bir faktördür. En iyi sonuç veren üç modelin katman başına nöron sayıları, ilk model için sekiz katmanda düzgün bir şekilde yükseltilerek (128, 128, 512, 512, 512, 512, 512, 512), ikinci model için artan ancak

daha dengesiz bir dağılım (16, 16, 64, 64, 128, 128, 256, 256) ve üçüncü model için ilk dört katmanda artış gösterip sonrasında azalan bir düzen (128, 128, 256, 256, 512, 512, 256, 256) olacak şekilde ayarlanmıştır. Bu modellerin doğruluk değerleri sırasıyla %96,71, %79,61, %94,38 olarak ölçülmüştür. Bu sonuçlar, nöron sayısının ve dağılımının, modelin genel performansı üzerinde belirgin bir etkiye sahip olduğunu ve doğru bir ayarlamanın modelin hata tespiti ve düzeltme yeteneğini önemli ölçüde artırabileceğini göstermektedir.

Çizelge 5.4, hata türü modelinin farklı aktivasyon fonksiyonları kullanılarak elde edilen performans sonuçlarını karşılaştırmaktadır.

Çizelge 5.4 : Hata türü modeli için aktivasyon fonksiyonlarının karşılaştırılması.

Parametreler ve Metrikler	Deney 1	Deney 2	Deney 3	Deney 4
Katman Sayısı	10	10	10	10
Aktivasyon F.	Softmax	Sigmoid	ReLU	Tanh
Optimizasyon F.	Adam	Adam	Adam	Adam
Küme Boyutu	128	128	128	128
Döngü Sayısı	50	50	50	50
Parametre Sayısı	21.529.091	21.529.091	21,529,091	21,529,091
Doğruluk	**0,3945**	0,3807	0,3252	0,3261
Kayıp	**1,0421**	1,0853	1,0986	1,0986

Tüm deneylerde katman sayısı 10, optimizasyon fonksiyonu Adam, küme boyutu (batch size) 128 ve döngü sayısı (epoch) 50 olarak sabit tutulmuş, parametre sayısı ise 21.529.091 olarak belirlenmiştir. Aktivasyon fonksiyonları olarak Softmax, Sigmoid, ReLU ve Tanh seçenekleri test edilmiş ve her birinin modelin doğruluk ve kayıp metrikleri üzerindeki etkisine bakılmıştır.

Düşük performans, deneylerde kullanılan nöron sayısının her katmanda (256, 256, 1024, 1024, 1024, 1024, 1024, 1024, 1024, 1024) oldukça yüksek ayarlanmasından kaynaklanmaktadır. Yüksek nöron sayısı, modelin aşırı uyuma (overfitting) eğilimi göstermesine ve genel olarak daha düşük doğruluk elde etmesine yol açmıştır. Ancak tüm parametreler sabit tutularak yalnızca aktivasyon fonksiyonlarının değiştirildiği deneylerde Deney 1, Softmax aktivasyon fonksiyonu ile %39,45 doğruluk ve 1.0421 kayıp değeri elde etmiştir. Deney 2, Sigmoid aktivasyon fonksiyonu kullanıldığında %38,07 doğruluk ve 1.0853 kayıp değeri ile biraz daha düşük performans göstermiştir. Deney 3, ReLU aktivasyon fonksiyonu ile %32,52 doğruluk ve 1,0986 kayıp değeriyle, bu fonksiyonların içinde en düşük performansı sergilemiştir. Deney 4, Tanh

aktivasyon fonksiyonu ile %32,61 doğruluk ve yine 1,0986 kayıp değeri elde etmiş, ReLU ile benzer bir performans göstermiştir. Bu sonuçlar, aktivasyon fonksiyonunun seçiminin, modelin hata türlerini sınıflandırma kabiliyeti üzerinde önemli bir etkisi olduğunu göstermektedir. Softmax aktivasyon fonksiyonu, bu dört fonksiyon arasında en yüksek doğruluk değerini sunmuştur.

Çizelge 5.5'te farklı küme boyutları için gerçekleştirilen deneylerin performans metriklerinin sonuçları karşılaştırmalı olarak verilmiştir. Veriler dört deney (Deney 1'den Deney 4'e) için düzenlenmiş ve her deney için kümelerin büyüklüğü, toplam parametre sayısı, kayıp değeri ve doğruluk değeri gibi metrikler sunulmuştur.

Çizelge 5.5: Hata türü modeli için farklı küme boyutlarının (batch size) karşılaştırılması.

Deney Kodu	Küme Boyutu	Toplam Parametre	Kayıp Değeri	Doğruluk Değeri
Deney 1	64	5.528.579	0,0625	0,9766
Deney 2	128	5.528.579	1,0653	0,4102
Deney 3	256	5.528.579	**0,0575**	**0,9791**
Deney 4	512	5.528.579	0,6506	0,7138

Küme Boyutu sütunu, model eğitimi sırasında kullanılan kümelerin (batch'lerin) boyutunu göstermektedir. 64'ten 512'ye kadar değişen dört farklı değer içermektedir.

Toplam Parametre sütunu, modelin öğrenmesi gereken toplam parametre sayısını göstermektedir. Bu örnekte tüm deneyler için sabit bir değer olan 5.528.579 olarak belirtilmiştir. Kayıp değeri ve doğruluk değeri sütunları, modelin performansını ölçen iki önemli metriktir.

Çizelgeye göre, küme boyutu arttıkça kayıp değerinde genel bir artış (Deney 1'de 0,0625'ten Deney 4'te 0,6506'ya) ve doğruluk değerinde genel bir düşüş (Deney 1'de %97,66'dan Deney 4'te %71,38'e) gözlemlenmiştir. Bu, modelin daha büyük batch boyutlarıyla eğitildiğinde eğitim verilerine daha az uyum sağladığını ve dolayısıyla daha düşük bir doğruluk oranı elde ettiğini göstermektedir.

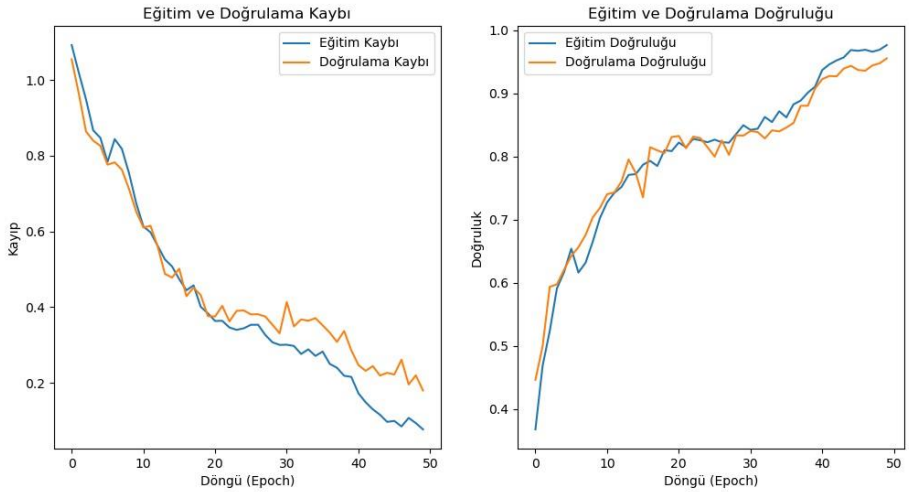
Çizelge 5.6'da farklı epoch sayılarının hata türü modelinin performansına etkisini gösteren beş deney sunulmuştur.

Çizelge 5.6 : Hata türü modeli için farklı döngü sayılarının (epoch) karşılaştırılması.

Deney Kodu	Döngü Sayısı	Toplam Parametre	Kayıp Değeri	Doğruluk Değeri
Deney 1	50	5.528.579	0.0625	0.9766
Deney 2	100	5.528.579	0.8756	0.5891
Deney 3	150	5.528.579	**0.0548**	**0.9800**
Deney 4	250	5.528.579	0.0641	0.9793
Deney 5	500	5.528.579	0.0077	0.9966

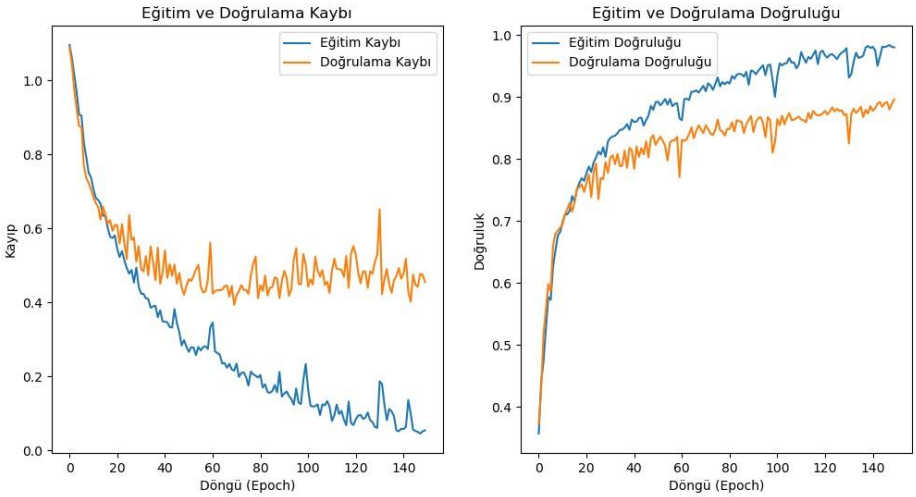
Her deney için modelin toplam parametre sayısı 5.528.579 olarak sabit tutulmuştur. İlk deneyde (50 epoch) model %97,66 doğruluk oranı ile oldukça yüksek bir başarı göstermişken ikinci deneyde (100 epoch) bu oran %85,91'e gerilemiştir. Üçüncü deneyde (150 epoch) doğruluk %98,00'e yükselmiş dördüncüde (250 epoch) hafif bir düşüşle %97,93 olarak kaydedilmiştir. Beşinci deneyde (500 epoch) doğruluk oranı %99,66 ile en yüksek seviyeye ulaşmıştır. Kayıp değeri, 50 epoch'ta 0,0625 iken artan epoch sayısıyla azalarak 500 epoch'ta 0,0077'ye düşmüştür. Bu sonuçlar, doğru epoch değerinin belirlenmesinin önemini ve modelin performansını maksimize etmek için titiz bir parametre ayarlama ihtiyacını vurgulamaktadır.

Şekil 5.8, Şekil 5.9, Şekil 5.10 ve Şekil 5.11'de farklı döngü sayılarına göre (epoch) gerçekleştirilen deneylerin doğruluk ve kayıp metrikleri için grafikleri verilmiştir. Şekil 5.8 (50 epoch), az sayıda epoch içerdiği için eğilimleri görmek zor olmaktadır. Ancak eğitim ve doğrulama kayıplarının hızla düştüğü ve doğrulukların yükseldiği görülmektedir.



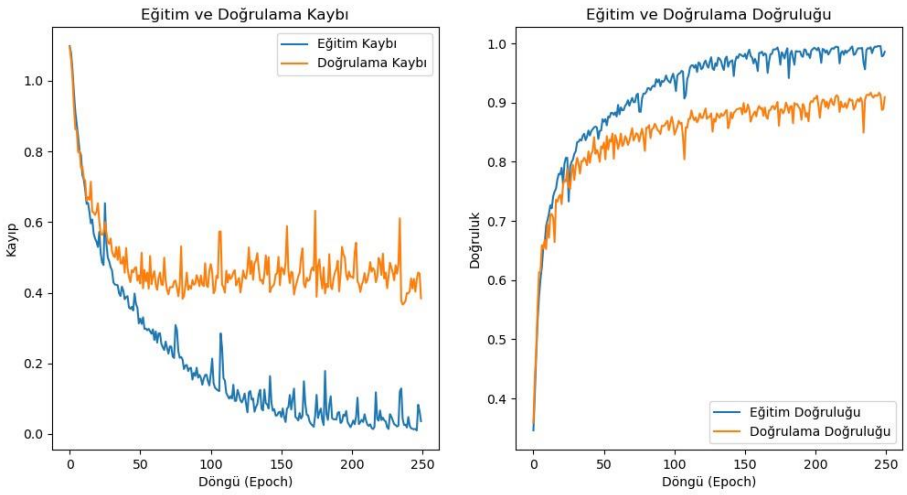
Şekil 5.8 : Döngü sayısı 50 alınarak gerçekleştirilen modelin eğitim sürecinde doğruluk ve kayıp değerleri grafiği.

Şekil 5.9 (150 epoch), eğitim ve doğrulama kayıplarının daha stabil ve az dalgalı olduğunu göstermektedir. Doğrulama doğruluğu, eğitim doğruluğunu ilk döngülerden itibaren takip etmekte ve hatta geçmektedir. Bu da modelin genellemeyi iyi yaptığını göstermektedir.



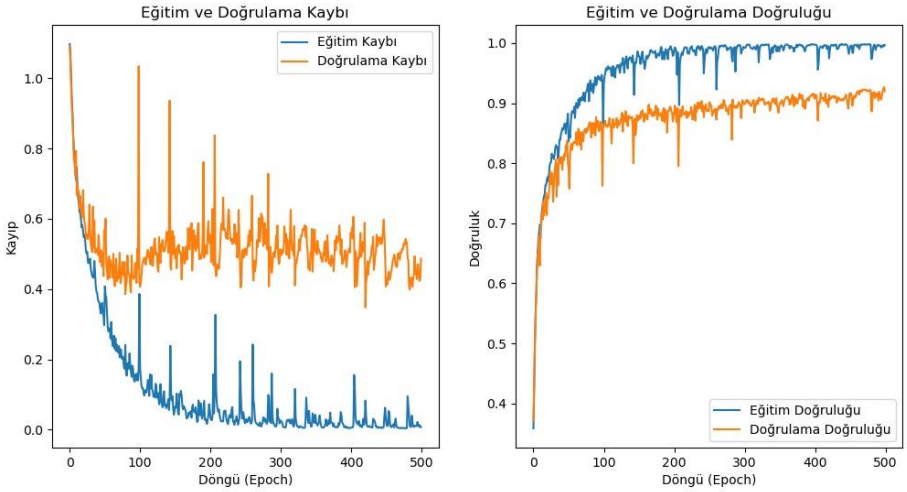
Şekil 5.9 : Döngü sayısı 150 alınarak gerçekleştirilen modelin eğitim sürecinde doğruluk ve kayıp değerleri grafiği.

Şekil 5.10 (250 epoch), eğitim kaybının ilk başta hızlı düştüğünü ama sonra stabil bir seyir izlediğini göstermektedir. Doğrulama kaybı da benzer bir trend izlemektedir fakat daha fazla dalgalanma göstermektedir. Doğruluk grafiklerinde de doğrulama doğruluğu, eğitim doğruluğuna yakın seyretmektedir.



Şekil 5.10 : Döngü sayısı 250 alınarak gerçekleştirilen modelin eğitim sürecinde doğruluk ve kayıp değerleri grafiği.

Şekil 5.11 (500 epoch), modelin eğitim kaybının hızlı bir şekilde düştüğünü ve doğrulama kaybının dalgalı bir şekilde azaldığını göstemektedir. Eğitim doğruluğu hızla artarken doğrulama doğruluğu daha yavaş bir artış sergilediği ve sonlara doğru eğitim doğruluğunu yakalamaya başladığı görülmektedir.



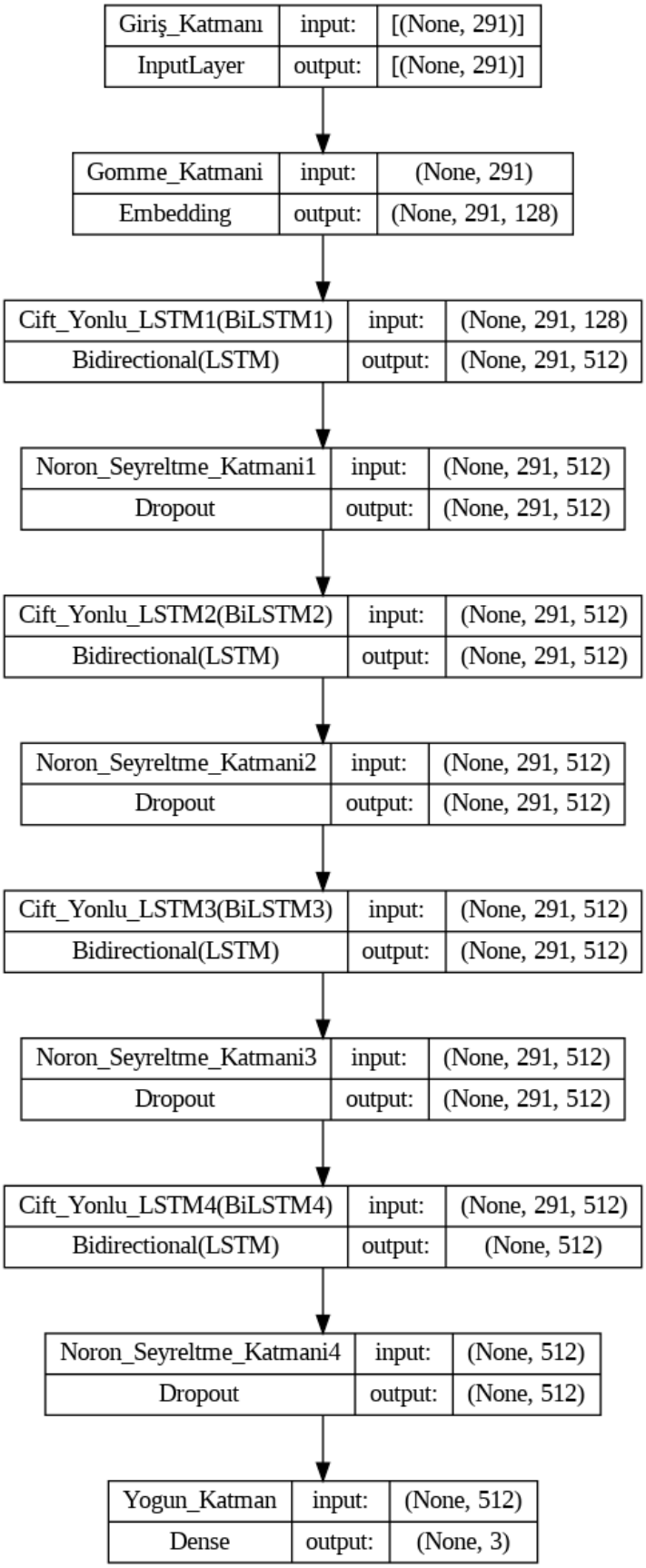
Şekil 5.11 : Döngü sayısı 500 alınarak gerçekleştirilen modelin eğitim sürecinde doğruluk ve kayıp değerleri grafiği.

Rastgele Arama (Random Search), hiper-parametre optimizasyonu için kullanılan bir yöntemdir. Bu yöntem, olası tüm değerler üzerinde sistematik bir şekilde ilerlemek yerine, hiper-parametreleri rastgele seçmeyi içerir. Rastgele Arama, Izgara Aramasına (Grid Search) kıyasla, çok daha az hesaplama zamanı içinde aynı veya daha iyi

modelleri bulabilmektedir. Rastgele aramanın avantajlarından biri, kullanım kolaylığı ve uygulanabilirliğidir. Rastgele aramanın pratik avantajları arasında kavramsal basitlik, uygulama kolaylığı ve paralel işleme gibi özellikler yer almakta ve düşük boyutlu alanlarda küçük bir verim kaybı pahasına yüksek boyutlu arama alanlarında önemli ölçüde verimlilik artışı sağladığı belirtilmiştir. Rastgele arama ayrıca, tüm hiper-parametrelerin eşit derecede önemli olmadığı ve bazı boyutlarda yapılan değişikliklerin diğerlerine göre daha belirgin sonuçlar doğurduğu durumlarda, grid search'e göre daha verimli olduğu gösterilmiştir (Bergstra ve Bengio, 2012). Bu sebeplerle bu çalışmada hiper-parametre optimizasyonu için rastgele arama algoritması kullanılmıştır.

Genel olarak, tüm grafikler eğitim sürecinin ilerlemesine bağlı olarak kayıp ve doğruluk metriklerinin beklenen yönde hareket ettiğini göstermektedir. Ancak dalgalanmalar, özellikle doğrulama metriklerinde, modelin veri setindeki genellemeyi nasıl yaptığına dair ipuçları vermektedir. İdeal olarak, doğrulama kaybı düşük ve stabil olmalı ve doğrulama doğruluğu yüksek ve eğitim doğruluğu ile benzer bir trend izlemesi gerekmektedir. Eğitim ve doğrulama arasındaki büyük farklar, aşırı uyum (overfitting) veya yetersiz uyum (underfitting) problemlerine işaret etmektedir. Bu bilgiler ışığında, genel performans ve genelleme yeteneği açısından ikinci grafik (150 epoch) en iyi model olarak değerlendirilmiştir. Bu model, yüksek doğrulama doğruluğu ve düşük doğrulama kaybı ile dengeli bir öğrenme sergilemektedir. Bu grafikler, modelin performansının genel olarak iyi olduğunu ve belirli bir veri seti üzerinde tatmin edici sonuçlar verebileceğini göstermektedir. Ancak, bu modelin gerçek dünya verileri üzerinde nasıl performans göstereceğini değerlendirmek için daha fazla test yapılması gerekmektedir. Model eğitim verilerinden farklı olarak oluşturulan veri seti ile bu testler gerçekleştirilmiştir. Başlık 5.5'te sonuçlar verilmiştir.

Şekil 5.12'de hata türü tahmin modeli için geliştirilen modelin katman yapısı verilmiştir.



Şekil 5.12: Hata türü modeli katman yapısı.

Model, her bir girdi örneği için 291 özellik içeren girdileri kabul ederek bu girdileri işlemek üzere bir dizi özel katman içermektedir. İlk katman, bir gömme katmanıdır. Bu katman, girdi örneklerini alarak her birini 128 boyutunda yoğun vektörlere dönüştürmektedir. Gömme katmanının çıktısı, daha sonra modelin iç yapısında daha derinlemesine bilgi işlemesine imkan tanıyan ardışık çift yönlü LSTM (BiLSTM) katmanlarına beslenmektedir. Her bir BiLSTM katmanı, önceki katmandan gelen verileri hem geçmiş hem de gelecek zaman adımlarını dikkate alarak işlemektedir. Bu sayede, sıralı verilerin daha geniş bir bağlamını öğrenmektedir. Her bir BiLSTM katmanından sonra modelin aşırı uyum riskini azaltmak amacıyla düşürme (dropout) katmanları yerleştirilmiştir. Bu katmanlar, modelin daha sağlam olmasını sağlamak için eğitim sırasında rastgele nöronları devre dışı bırakmaktadır. Modelin en son katmanı olan yoğun katman, öğrenilen özellikleri alarak bu bilgiyi tahmin edilen sınıfların olasılıklarını içeren sonuçlara dönüştürmektedir. Bu katman yapısının geneli, modelin karmaşık sıralı verilerden öğrenme yeteneğini geliştirmek ve verimli tahminlerde bulunmak için tasarlanmıştır.

5.3. Hatalı Tokenin Tespit Edilmesi

Çalışmada hatalı tokenin doğrudan konumunun belirlenmesi ve o konumdaki tokenin ikinci model (token tahmin modeli) tahmini ile hatanın onarılması amaçlanmıştır. Ast (Python Software Foundation, 2008) ve Pyright (Microsoft, 2019) modülleri, Python kodundaki hataların ve anormalliklerin tespit edilmesi için kullanılmaktadır. Her ikisi de farklı amaçlar için geliştirilmiştir. Farklı türdeki hataları yakalamakta etkilidirler.

Ast modülü, Python kodunu ayrıştırarak AST oluşturmaktadır. AST, kodun yapısal bir temsilidir. Her bir öğe (değişkenler, fonksiyonlar, sınıflar vb.) bir düğüm olarak temsil edilmektedir. Ast modülü, söz dizimi hatalarını ve yapısal sorunları tespit etmede de kullanılmaktadır. Örneğin, bir fonksiyonun yanlış çağrılması veya eksik bir değişken gibi durumlar ast ile tespit edilmektedir. Ancak bu modül her türlü hatayı tespit etmez. Özellikle çalışma zamanı hataları veya adlandırma hataları (NameError gibi) ast'nin kapsamı dışındadır.

Pyright, Microsoft tarafından geliştirilen statik bir kod analizi aracıdır. Python'da tip denetimi ve hata tespiti yapmaktadır. Tip denetiminin yanı sıra, belirli türdeki çalışma zamanı hatalarını, kodun doğru yazılmış olup olmadığını ve daha birçok sorunu tespit etmektedir. Pyright, ast modülünün tespit edemediği bazı hataları bulmaktadır.

Örneğin, NameError gibi adlandırma hataları Pyright tarafından tespit edilebilmektedir. Bu hatalar genellikle değişkenlerin yanlış adlandırılması veya henüz tanımlanmamış değişkenlerin kullanılmasından kaynaklanmaktadır. Pyright, bu tür hataların yerini belirlemekte ve geliştiricilere kodlarında düzeltme yapmaları için yararlı bilgiler sunmaktadır.

Her iki modülün birlikte kullanılması, Python kodunun geniş bir yelpazedeki hatalarını tespit etme ve düzeltme konusunda daha etkili bir yöntem sunmaktadır. Ast yapısal ve söz dizimi sorunlarını analiz ederken Pyright daha çok adlandırma ve tip hataları gibi alanlarda güçlüdür. Bu yaklaşım, kodun daha sağlam ve hata azlığı olan bir yapıya kavuşmasını sağlamaktadır.

Ast ve Pyright modülleri Python kodundaki hataların tespiti için kullanılan güçlü araçlardır. Bu modüller kodun yapısını analiz ederek hataların satır ve sütun bilgilerini sağlamaktadır. Ayrıca kodlar tokenlere dönüştürüldüğünde bu tokenlerin satır ve sütun bilgileri de bir liste içinde saklanmıştır. Hataların ast veya Pyright tarafından verilen satır ve sütun bilgileri, tokenlerin konum listesi ile karşılaştırılarak ilgili tokenin sıra numarası belirlenmektedir. Bu yöntemle, doğrudan hatalı tokenin tespit edilmesi ve onun üzerinde odaklanılması mümkün olmaktadır.

Bu yaklaşım, Python kodunun analizinde ve hata düzeltmesinde önemli bir zaman tasarrufu sağlamaktadır. Geleneksel yöntemlerde, kodun tamamının veya büyük bölümlerinin tekrar tekrar incelenmesi gerekebilirken Ast ve Pyright ile hataların kesin konumlarının belirlenmesi sorunun çok daha hızlı çözülmesine imkan tanımaktadır. Özellikle, ikinci modelin hatalı tokeni tahmin etmesi gerektiğinde bu yaklaşım modelin sadece ilgili token üzerinde çalışmasına ve böylece zaman ve kaynak verimliliğinin artmasına olanak tanımaktadır. Kısacası bu modüllerin kullanımı, Python programlama dilinde hata tespiti ve düzeltme süreçlerini önemli ölçüde optimize etmektedir.

5.4. Hatalı Tokenin Onarılması

Hatalı bir tokenin tam yerinin belirlenmesi, o tokenin sıra numarasını (indisini) tespit etmemizi sağlamaktadır. Bu da onarım sürecinin daha odaklı ve verimli olmasını sağlamaktadır.

Bir tokenin hatalı olduğu belirlendikten sonra onarım süreci genellikle ilgili tokenin bağlamını dikkate alarak gerçekleştirilmektedir. Bu bağlam, hatalı tokenin öncesinde ve sonrasında yer alan tokenlerden oluşmaktadır. Bu yaklaşım, onarım yapacak modelin, hatalı tokenin doğru biçimini tahmin etmesi için gerekli olan bağlamsal bilgileri sağlamaktadır. Örneğin bir programlama dilindeki söz dizimi hatası durumunda hatalı tokenin yerine geçecek doğru tokeni belirlemek için çevresindeki tokenlerin yapısal ve semantik özellikleri dikkate alınmaktadır.

Her bir tokenin ayrı ayrı incelenmesi yerine hata tespit edilen spesifik bölgeye odaklanmak hata düzeltme sürecini hızlandırmaktadır. Daha az hata yapma olasılığını artırmaktadır. Bu bağlamsal analiz, derin öğrenme ve makine öğrenimi modellerinin kod onarımında kullanılmasını da mümkün kılmaktadır. Böylece karmaşık ve zaman alıcı hata düzeltme işlemleri otomatikleştirilebilir. Bu sürecin etkin bir şekilde uygulanması kod kalitesini artırırken geliştirme süreçlerinin daha akıcı ve verimli hale gelmesine de katkıda bulunmaktadır.

Hata türü tahmin modelinde RNN'nin LSTM ve BiLSTM'ye göre daha düşük performans göstermesi, RNN'nin uzun dönem bağımlılıkları etkili bir şekilde modelleyememesinin bir göstergesidir. LSTM ve BiLSTM, RNN'nin zayıf yönlerini geliştirerek özellikle uzun sıralı verilerde daha iyi sonuçlar elde etmek için tasarlanmıştır. Hata türü tahmin modelinde elde edilen sonuçlar RNN'nin sınırlılıklarını ve LSTM ile BiLSTM'nin bu tür verilerde üstün performans sergileyebileceğini açıkça ortaya koymaktadır. Bu nedenle token tahmin modelinde LSTM ve BiLSTM karşılaştırılmıştır.

Çizelge 5.7, token tahmin modeli için LSTM ve BiLSTM ağ yapılarının performanslarını karşılaştırmaktadır.

Çizelge 5.7: Token tahmin modeli için yapay sinir ağı yöntemlerinin karşılaştırılması.

Parametreler ve Metrikler	LSTM	BiLSTM
Katman Sayısı	4	4
Aktivasyon Fonksiyonu	Softmax	Softmax
Optimizasyon Fonksiyonu	Adam	Adam
Küme Boyutu (Batch Size)	128	128
Döngü Sayısı (Epoch)	50	50
Parametre Sayısı	343.358	886,558
Doğruluk	0,9741	**0,9743**
Kayıp	**0,0692**	0,0704
F1 Skoru	**0,9797**	0,9790

Her iki yapı da aynı katman sayısı (4), aktivasyon fonksiyonu (Softmax), optimizasyon fonksiyonu (Adam) ve küme boyutu (Batch Size) (128) ile yapılandırılmıştır. Ayrıca her iki model de 50 epoch boyunca eğitilmiştir.

BiLSTM, LSTM'ye kıyasla daha fazla parametreye sahiptir (886,558'e karşı 343,358). Bu genellikle BiLSTM'lerin iki yönlü yapılarından kaynaklanmaktadır. Modelin hem geçmiş hem de gelecek bilgileri dikkate almasına olanak tanımaktadır.

Performans metriklerine bakıldığında her iki model de benzer doğruluk değerlerine ulaşmıştır. LSTM için %97.41 ve BiLSTM için %97.43 elde edilmiştir. Bu, her iki modelin de görevi benzer bir yetenekle gerçekleştirdiğini göstermektedir. Kayıp değerleri de benzerdir. LSTM için 0.0692 ve BiLSTM için 0.0704 elde edilmiştir. Bu durum, her iki modelin de öğrenme sürecinde benzer bir hata oranı ile ilerlediğini göstermektedir. F1 skoru, modelin sınıflandırma performansının hassasiyet ve hatırlama dengesini ölçmektedir. LSTM için 0.9797 ve BiLSTM için 0.9790 olarak belirtilmiştir. Bu değerler, her iki modelin de neredeyse aynı seviyede performans gösterdiğini göstermektedir.

Bu verilere dayanarak LSTM ve BiLSTM modelleri arasında performans açısından önemli bir fark olmadığı görülmektedir. Ancak, BiLSTM'in daha yüksek parametre sayısına rağmen benzer sonuçlar üretmesi belki de bu özel görev için ek karmaşıklığın gerekli olmadığını göstermektedir. Uygulamada model seçimi hem performans hem de hesaplama maliyeti açısından değerlendirilmiştir. BiLSTM modelleri genellikle daha fazla hesaplama kaynağı gerektirmektedir. Sınırlı kaynaklara sahip senaryolarda veya gerçek zamanlı uygulamalarda önemli bir faktördür. Bu sebeple token tahmin modeli için LSTM ağı tercih edilmiştir.

Çizelge 5.8, farklı katman sayısına sahip modellerin performansını karşılaştıran bir dizi deneyi göstermektedir.

Çizelge 5.8 : Token tahmin modeli için katman sayılarının karşılaştırılması.

Parametreler ve Metrikler	Deney 1	Deney 2	Deney 3	Deney 4	Deney 5
Katman Sayısı	3	4	5	6	7
Aktivasyon F.	Softmax	Softmax	Softmax	Softmax	Softmax
Optimizasyon F.	Adam	Adam	Adam	Adam	Adam
Küme Boyutu	128	128	128	128	128
Döngü Sayısı	50	50	50	50	50
Parametre Sayısı	242.958	343.358	664.158	984.958	1.305.758
Doğruluk	0,9704	0,9743	**0,9751**	0,9748	0,9733
Kayıp	0,0819	0,0704	**0,0681**	0,0690	0,0746
F1 Skoru	0,9749	0,9790	0,9761	**0,9791**	0,9782

Katman sayısı arttıkç modelin parametre sayısı önemli ölçüde artmaktadır. Bu, modelin karmaşıklığının ve öğrenme kapasitesinin arttığı anlamına gelmektedir. Modelin katman sayısını 3'ten 7'ye çıkarmak parametre sayısını yaklaşık olarak 5 kata kadar artırmıştır.

Tüm deneylerde aktivasyon fonksiyonu olarak Softmax kullanılmıştır. Çoğunlukla çok sınıflı sınıflandırma problemlerinde tercih edilen bir yöntemdir. Tüm deneylerde optimizasyon fonksiyonu olarak Adam kullanılmıştır. Bu da adaptif öğrenme oranları sunarak farklı parametreler için öğrenme hızını ayarlayarak hızlı yakınsama sağlamaktadır. Tüm deneylerde küme boyutu sabit tutulmuştur. Her iterasyonda model 128 örnek üzerinde güncelleme yapmaktadır. Her deneyde döngü sayısı sabit tutulmuştur (50). Bu da tüm modellerin aynı sayıda iterasyon üzerinden eğitildiğini göstermektedir.

Doğruluk ve kayıp değerleri genellikle modelin performansını değerlendirmede kullanılmaktadır. Doğruluk, modelin eğitim setinde ne kadar iyi performans gösterdiğini gösterirken, kayıp değeri modelin yaptığı tahminlerin gerçek değerlerden ne kadar uzak olduğunu göstermektedir. Deney 4, en yüksek doğruluğu (%97,48) ve en düşük kaybı (0,0690) göstermiştir. Bu da bu yapılandırmanın veri seti üzerinde en iyi performansı sergilediğini göstermektedir.

F1 skoru, modelin hassasiyet ve duyarlılık (recall) dengesini ölçmektedir. Özellikle sınıflar arası dağılımın dengesiz olduğu durumlarda önemlidir. En yüksek F1 skoru (%97,97) Deney 2'de elde edilmiştir. Modelin hem pozitif hem de negatif sınıfları dengeli bir şekilde tanıdığını ve yanlış pozitif ile yanlış negatif oranlarını dengede tuttuğunu göstermektedir. Deney 2'de elde edilen en yüksek F1 skoru, dört katmanlı

bir model yapılandırması kullanıldığında gerçekleşmiştir. Bu, modelin karmaşıklığının ve öğrenme kapasitesinin bu seviyede en uygun olduğu düşünülmektedir. Modelin doğruluğu ve kaybı da genel performansın dengeli olduğunu göstermektedir. Ancak, Deney 2 ve Deney 4 arasındaki F1 skorları arasında çok küçük bir fark vardır. Bu da modelin bu iki yapılandırma arasında benzer performans gösterdiğine işaret etmektedir.

Bu sonuçlar, katman sayısı arttıkça doğruluk ve kayıp değerlerinde belirgin bir iyileşme eğilimi olmadığını gösterirken F1 skoru özelinde Deney 2'nin en iyi dengeli performansı sunduğunu göstermektedir. Bu durum, modelin yapılandırmasının yanı sıra kullanılan veri setinin özelliklerine göre dikkatlice ayarlanması gerektiğini vurgulamaktadır.

Genel olarak deneyler arasında performans ölçütlerinde önemli bir fark gözlemlenmemekle birlikte katman sayısı arttıkça modelin performansında hafif bir iyileşme eğilimi olduğu gözlemlenmektedir. Ancak katman sayısının artması modelin eğitim süresini ve karmaşıklığını da artıracağı için bu iyileşmelerin maliyet etkinliği göz önünde bulundurularak değerlendirilmesi gerekmektedir. Bu, modelin performansının sadece bir metrikle değil bir dizi metrikle kapsamlı olarak değerlendirilmesi gerektiğini göstermektedir.

Çizelge 5.9, token tahmin modelinin farklı nöron sayılarıyla yapılandırılmış beş farklı deneyini karşılaştırmaktadır. Çizelgede, her bir deney için LSTM tabanlı bir sinir ağı modelinde kullanılan nöron sayıları, modelin toplam parametre sayısı, kayıp değeri, doğruluk değeri ve F1 skoru verilmiştir.

Çizelge 5.9 : Token tahmin modeli için farklı nöron sayılarının karşılaştırılması.

Deney Kodu	Nöron Sayıları (LSTM)	Toplam Parametre	Kayıp Değeri	Doğruluk Değeri	F1 Skoru
Deney 1	50, 100, 200	343.358	0,0692	0,9741	0,9797
Deney 2	100, 150, 200	503.958	**0,0575**	**0,9791**	**0,9808**
Deney 3	150, 200, 300	930.358	0,0639	0,9764	0,9793
Deney 4	150, 150, 150	476.258	0,0629	0,9769	0,9786
Deney 5	150, 300, 300	1.401.158	0,1013	0,9641	0,9639

Toplam parametre sayısının artışı, modelin karmaşıklığını artırmaktadır. Genellikle kayıp değerinin azalmasına katkıda bulunmaktadır. Ancak parametre sayısı çok fazla arttığında (Deney 5 gibi), modelin eğitim veri setine aşırı uyum sağlaması ve

genelleme yeteneğinin azalması riski ortaya çıkmaktadır. Bu durum, Deney 5'teki yüksek kayıp değeri ve düşük F1 skoru ile gözlemlenmektedir.

En düşük kayıp değeri ve en yüksek doğruluk ve F1 skoru Deney 2'de elde edilmiştir. Bu yapılandırmanın veri seti üzerinde en iyi performansı gösterdiğini göstermektedir.

Deneylerin her biri farklı model yapılandırmalarının avantajlarını ve sınırlamalarını göstermektedir. Deney 2'nin orta seviye parametre sayısı ile en iyi kayıp, doğruluk ve F1 skorlarına sahip olduğu görülmektedir. Bu, modelin yeterince karmaşık olduğunu ancak aşırı uyuma düşmeden iyi bir genelleme yapabildiğini göstermektedir. Bu sebeple çalışmada kullanılacak modelin LSTM katmanlarındaki nöron sayıları Deney 2'de verildiği gibi 100,150,200 seçilmiştir.

Çizelge 5.10, bir token tahmin modeli için farklı aktivasyon fonksiyonlarının kullanıldığı dört deneyi karşılaştırmaktadır. Çizelgede dört deney de aynı sayıda katman (4), aynı optimizasyon fonksiyonu (Adam), küme boyutu (128) ve döngü sayısı (50) ile yapılmıştır. Ancak her bir deneyde farklı bir aktivasyon fonksiyonu kullanılmıştır.

Çizelge 5.10 : Token tahmin modeli için aktivasyon fonksiyonlarının karşılaştırılması.

Parametreler ve Metrikler	Deney 1	Deney 2	Deney 3	Deney 4
Katman Sayısı	4	4	4	4
Aktivasyon F.	Softmax	Sigmoid	ReLU	Tanh
Optimizasyon F.	Adam	Adam	Adam	Adam
Küme Boyutu	128	128	128	128
Döngü Sayısı	50	50	50	50
Parametre Sayısı	503.958	503.958	503,958	503,958
Doğruluk	**0,9791**	0,9772	0,1393	0,1393
Kayıp	**0,0575**	0,0620	4,0657	Ölçüm yok (nan)
F1 Skoru	**0,9797**	0,9781	0,0340	0,l0340

Aktivasyon fonksiyonları, sinir ağlarında nöronların çıktısını hesaplamak için kullanılmaktadır. Modelin öğrenme kapasitesini ve karmaşıklığını etkilemektedir.

Deney 1 ve 2, Softmax ve Sigmoid aktivasyon fonksiyonları kullanılarak benzer ve yüksek doğruluk ve F1 skorları elde edilmiştir.

Deney 3 ve 4 ise sırasıyla ReLU ve Tanh aktivasyon fonksiyonları kullanılmıştır. Düşük doğruluk ve F1 skorları elde edilmiştir. Deney 4'te ayrıca kayıp değeri 'nan'

olarak elde edilmiştir. Bu durum genellikle sayısal kararsızlık veya modelin eğitim sırasında "patladığını" göstermektedir.

Bu sonuçlara dayanarak, ReLU ve Tanh aktivasyon fonksiyonlarının bu özel token tahmin modeli için uygun olmadığı sonucuna varılmıştır. Bunun nedeni, ReLU'nun bazen sıfır dışındaki gradyanları sınırlamaması ve Tanh'ın gradyan kaybına yol açabilmesidir.

Softmax ve Sigmoid aktivasyon fonksiyonlarının benzer ve yüksek performans göstermesi, bu iki fonksiyonun model için daha uygun olduğunu göstermektedir. Ancak, Softmax aktivasyon fonksiyonunu kullanarak yapılan Deney 1, en düşük kayıp değeri ve en yüksek F1 skoru ile daha iyi performans göstermiştir. Bu nedenle, bu modelin sınıflandırma görevi için en uygun aktivasyon fonksiyonunun Softmax olduğu kabul edilmiştir.

Çizelge 5.11, bir token tahmin modeli için üç farklı optimizasyon algoritmasının kullanıldığı deneyleri karşılaştırmaktadır. Tüm deneylerde katman sayısı, aktivasyon fonksiyonu ve küme boyutu sabit tutulmuş farklılıklar optimizasyon fonksiyonlarında ve bunların sonuçlarında görülmektedir.

Çizelge 5.11: Token tahmin modeli için optimizasyon algoritmalarının karşılaştırılması.

Parametreler ve Metrikler	Deney 1	Deney 2	Deney 3
Katman Sayısı	4	4	4
Aktivasyon F.	Softmax	Softmax	Softmax
Optimizasyon F.	Adam	SGD	RMSprop
Küme Boyutu	128	128	128
Döngü Sayısı	50	50	50
Parametre Sayısı	21.529.091	21.529.091	21.529.091
Doğruluk	**0,9791**	0,1393	0,9672
Kayıp	**0,0575**	Ölçüm yok (nan)	0,1006
F1 Skoru	**0,9797**	0,0340	0,9656

Deney 1 (Adam Optimizasyonu), Adam optimizasyon algoritması, genellikle iyi sonuçlar veren ve yaygın olarak kullanılan adaptif öğrenme oranına sahip bir algoritmadır. Bu deneyde, Adam optimizasyonu kullanıldığında model hem düşük kayıp değeri hem de yüksek doğruluk ve F1 skoru elde etmiştir. Bu da Adam'ın bu model için etkili bir optimizasyon algoritması olduğunu göstermektedir.

Deney 2 (SGD Optimizasyonu), SGD daha geleneksel bir gradyan iniş yöntemidir. Hiperparametre ayarlamasına daha duyarlıdır. Bu deneyde SGD kullanıldığında modelin performansı ciddi şekilde düşmüştür. Doğruluk ve F1 skoru çok düşük çıkmış ve kayıp değeri hesaplanamamıştır (yüksek olasılıkla sayısal bir kararsızlık veya gradyan patlaması nedeniyle).

Deney 3 (RMSprop Optimizasyonu), RMSprop da adaptif öğrenme oranlarına sahip bir başka optimizasyon algoritmasıdır. Genellikle konveks olmayan optimizasyon problemleri için uygundur. Bu deneyde RMSprop kullanıldığında model makul bir kayıp ve yüksek doğruluk ve F1 skoru elde etmiş, bu da RmsProp'un bu özel görev için de iyi bir seçim olabileceğini göstermektedir.

Genel olarak, Deney 1 (Adam optimizasyonu) en iyi sonuçları vermiştir. Adam, stabil ve hızlı bir şekilde iyi sonuçlara ulaşan bir optimizasyon algoritması olarak bilinmektedir. Bu durumda da en iyi performansı sergilemiştir. RMSprop'un da iyi sonuçlar verdiği görülse de Adam'a göre daha yüksek kayıp değeri ile ikinci sıraya yerleşmiştir.

Çizelge 5.12, bir token tahmin modeli için dört farklı küme boyutu (batch size) kullanılarak yapılan deneyleri karşılaştırmaktadır. Küme boyutu, bir eğitim iterasyonunda ağa beslenen örnek sayısını ifade eder ve modelin eğitim süresi ile genelleme kabiliyeti üzerinde önemli bir etkiye sahiptir.

Çizelge 5.12: Token tahmin modeli için farklı küme boyutlarının (batch size) karşılaştırılması.

Deney Kodu	Küme Boyutu	Toplam Parametre	Kayıp Değeri	Doğruluk Değeri	F1 Skoru
Deney 1	64	503.958	0,0625	0,9776	0,9807
Deney 2	128	503.958	0,0590	0,9778	0,9783
Deney 3	256	503.958	**0,0575**	**0,9791**	**0,9808**
Deney 4	512	503.958	1,0150	0,7319	0,7582

Küme boyutunun arttırılması genellikle her iterasyonda daha fazla örneğin işlenmesine ve böylece eğitim süresinin kısalmasına neden olmaktadır. Ancak, çok büyük bir küme boyutu modelin genelleme kabiliyetini azaltabilir çünkü gradyanların hesaplanması sırasında daha az güncelleme yapılmaktadır. Modelin daha geniş bir çözüm alanını araştırmasını engellemektedir.

Deney 3 (Küme Boyutu 256), en düşük kayıp ve en yüksek doğruluk ve F1 skorunu elde etmiştir. Bu küme boyutunun model için en uygun olabileceğini göstermektedir.

Küme boyutu 256, yeterli öğrenme ve hesaplama verimliliği arasında iyi bir denge sağlamıştır.

Deney 4 (Küme Boyutu 512), önemli ölçüde daha yüksek kayıp ve önemli ölçüde daha düşük doğruluk ve F1 skoru elde etmiştir. Bu da büyük küme boyutunun bu durumda modelin eğitim performansını olumsuz etkilediğini göstermektedir. Bu sonuçlar, büyük küme boyutlarının bazen modelin veri setinin daha ince özelliklerini öğrenmesini engelleyebileceğini ve aşırı uyuma neden olabileceğini göstermektedir.

Deney 1 ve 2 (Küme Boyutları 64 ve 128), oldukça benzer performans göstermiştir. Ancak küme boyutu 128 olan Deney 2 biraz daha düşük kayıp değeriyle öne çıkmıştır. Bu, 128'in bu özel durum için uygun bir küme boyutu olduğunu ancak 256'nın daha da iyi sonuçlar verdiğini göstermektedir.

Sonuç olarak, Çizelge 5.12'deki verilere dayanarak 256'lık küme boyutunun, bu token tahmin modeli için dengeyi en iyi sağladığı ve modelin genelleme kabiliyetini en iyi şekilde korurken iyi eğitim performansı gösterdiği görülmüştür. Çok büyük bir küme boyutu kullanmanın (Deney 4'te olduğu gibi) performansı düşürebileceği ve bu nedenle küme boyutunun dikkatli bir şekilde seçilmesi gerektiği sonucuna varılmıştır.

Çizelge 5.13, bir token tahmin modelinin farklı epoch (döngü sayıları) için eğitim sonuçlarını karşılaştırmaktadır. Burada her bir deney aynı toplam parametre sayısı ile yapılmış ancak modelin eğitildiği döngü sayısı farklılık göstermektedir.

Çizelge 5.13: Token tahmin modeli için farklı döngü sayılarının (epoch) karşılaştırılması.

Deney Kodu	Döngü Sayısı	Toplam Parametre	Kayıp Değeri	Doğruluk Değeri	F1 Skoru
Deney 1	50	503.958	0,0575	0,9791	0,9808
Deney 2	100	503.958	0,0607	0,9772	0,9864
Deney 3	150	503.958	0,0651	0,9756	0,9859
Deney 4	250	503.958	**0,0389**	**0,9844**	**0,9891**
Deney 5	500	503.958	3,0049	0,1488	0,0532

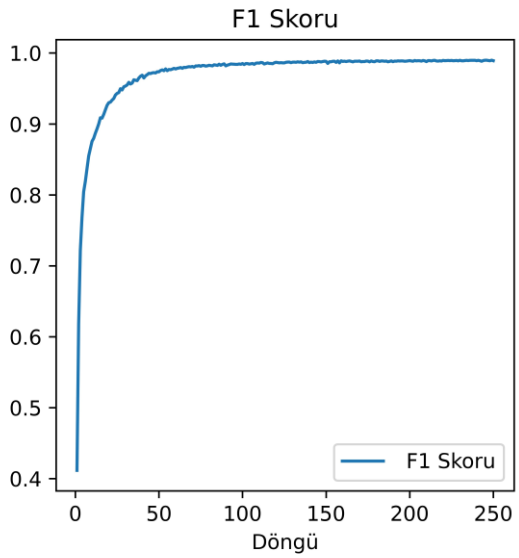
Deney 1'den Deney 4'e kadar doğruluk ve F1 skorları artarken kayıp değeri de azalmıştır. Bu, modelin daha fazla eğitilmesiyle genel olarak performansının iyileştiğini göstermektedir. Deney 4 en iyi performansı göstermiştir. En düşük kayıp ve en yüksek doğruluk ile F1 skorlarına sahip olmuştur.

Deney 5'te, modelin performansı ciddi şekilde düşmüştür. Bu da aşırı uyuma (overfitting) veya başka bir eğitim sorununa işaret etmektedir. Kayıp değerinin

3,0049'a yükselmesi, modelin eğitim sırasında hatalarını büyük ölçüde artırdığını ve bu durumun doğruluk ve F1 skorlarındaki ciddi düşüşle de teyit edildiğini göstermektedir. Bir modelin doğruluk değeri düşerken kayıp değerinin bu kadar yüksek olması, modelin eğitim veri setine uyum sağlayamadığını veya eğitim sırasında karşılaşılan sayısal problemleri işaret etmektedir.

Deney 1, 2 ve 3 arasında, F1 skorları birbiriyle oldukça uyumlu olmasına rağmen kayıp değerleri hafifçe artmıştır. Bu, eğitimin ilerleyen döngülerinde modelin aşırı uyuma başladığını göstermektedir. Sonuç olarak Çizelge 5.12'ye göre, token tahmin modeli için 250 döngü sayısının en iyi sonuçları verdiği ve daha uzun süreli eğitimin model performansını olumsuz etkilediği görülmüştür.

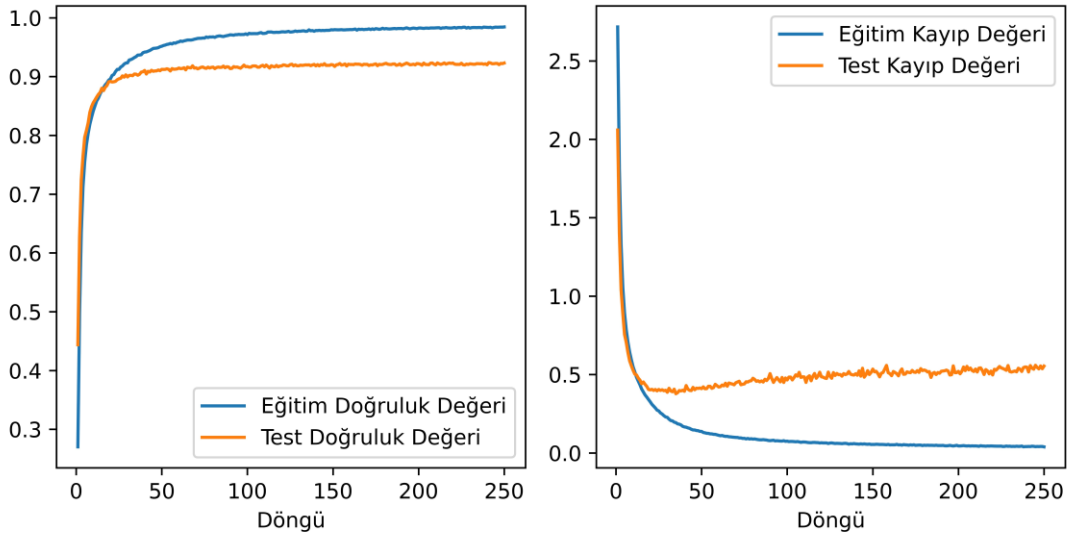
Şekil 5.13 ve Şekil 5.14 bir LSTM modelinin 250 epoch süresince eğitilmesi sonucunda elde edilen performans metriklerini göstermektedir.



Şekil 5.13: Token tahmin modeli eğitim sürecine ait F1 skoru grafiği.

Şekil 5.13 (F1 Skoru), modelin F1 skorunun eğitim sürecinde değişim sürecini göstermektedir. Grafik, F1 skorunun başlangıçta hızla arttığını ve sonra 50 epoch civarında yavaşça artarak doygunluğa ulaştığını göstermektedir. Bu, modelin başlangıçta öğrenmeye hızla adapte olduğunu ve belirli bir noktadan sonra ek eğitimin F1 skorunu önemli ölçüde artırmadığını göstermiştir.

Şekil 5.14'te eğitim ve test setleri üzerinde modelin doğruluk ve kayıp değerleri gösterilmiştir.



Şekil 5.14: Token tahmin modeli eğitim sürecine ait doğruluk ve kayıp grafiği.

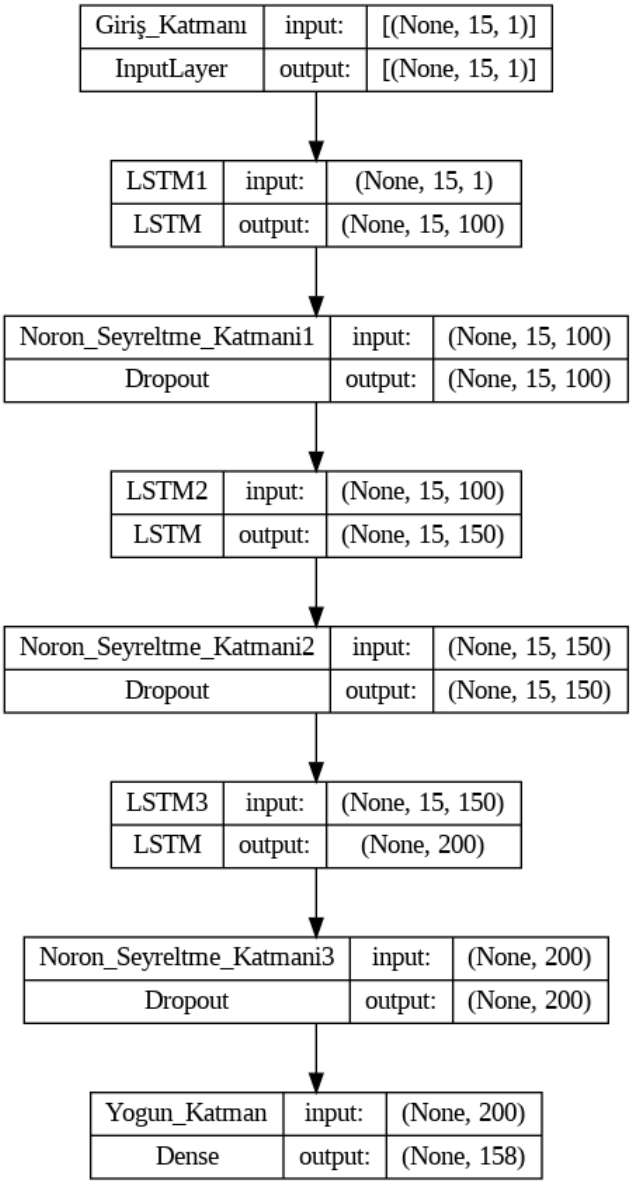
Eğitim doğruluk değeri, modelin eğitim setindeki verileri ne kadar iyi sınıflandırdığını gösterirken test doğruluk değeri modelin daha önce görmediği veriler üzerindeki performansını göstermektedir. Her iki doğruluk değeri de artmaktadır. yüksek bir değere ulaşmaktadır. Bu da modelin hem öğrenilen verileri iyi tahmin ettiğini hem de yeni verilere iyi genelleme yaptığını göstermektedir.

Kayıp değeri, modelin tahminlerinin gerçek değerlerden ne kadar uzak olduğunun bir ölçüsüdür. Bu çalışmada kayıp değeri kategorik çapraz entropi fonksiyonu ile hesaplanmıştır. Kategorik çapraz entropi, çok sınıflı sınıflandırma görevlerinde kullanılan bir kayıp fonksiyonudur. Modelin tahmin ettiği olasılık dağılımları ile gerçek etiketler arasındaki uyumu ölçer. Her bir sınıf için modelin tahmininin ne kadar doğru olduğunu değerlendirir ve bu tahminlerin doğruluk derecesine göre bir kayıp değeri hesaplar. Model, gerçek etiketlere yakın tahminler yaptığında düşük kayıp değeri üretirken, tahminler gerçek etiketlerden uzaklaştıkça kayıp değeri artar. Bu şekilde model, doğru sınıflandırmalar yapmaya teşvik edilir (Zhang ve Sabuncu, 2018).

Grafikteki eğitim ve test kayıp değerleri hızla düşmekte ve sonrasında düşük ve stabil bir seviyede kalmaktadır. Bu, modelin hem eğitim hem de test verileri üzerinde iyi öğrenme ve genelleme yeteneğine sahip olduğunu göstermektedir.

Her iki grafik birlikte değerlendirildiğinde modelin 250 epoch eğitimi sonunda yüksek bir F1 skoru ve doğruluk oranı ile iyi bir performans sergilediği ve düşük bir kayıp değerine sahip olduğu sonucuna varılmıştır.

Şekil 5.15 token tahmin modelinin sinir ağı mimarisini göstermektedir. Bu yapı, bir dizi LSTM katmanı ve Dropout katmanlarından oluşmaktadır.



Şekil 5.15: Token tahmin modeli katman yapısı.

InputLayer (Modelin giriş katmanı), modelin alacağı veri şeklini tanımlamaktadır. Burada giriş boyutu (None, 15, 1) şeklinde gösterilmiştir. Bu da her bir girişin 15 zaman adımından oluşan ve her zaman adımında tek bir özellik bulunduran bir dizi

olduğunu ifade etmektedir. None ise herhangi bir sayıda örnek alınabileceğini belirtmektedir.

Modelin performansını optimize etmek amacıyla gerçekleştirilen deneylerde belirli bir tokenin bağlamını tanımlamak için kullanılan önceki ve sonraki token sayıları değişken bir şekilde ayarlanmıştır. Bu ayarlamalar, bir tokenin doğru bir şekilde tahmin edilmesinde gerekli olan bağlamsal bilginin kapsamını belirlemeyi amaçlamaktadır. Deneyler sonucunda en iyi sonuçların, hedef tokenin öncesinde yer alan 7 token ve sonrasında yer alan 8 token olmak üzere toplamda 15 token kullanıldığında elde edildiği tespit edilmiştir. Bu bulgu, belirli bir tokenin doğru tahmin edilmesi için gerekli olan optimum bağlamsal pencerenin genişliğinin, model tarafından işlenen 15 tokenlik bir dizilim olduğunu göstermektedir. Bu optimizasyon, modelin bağlamsal anlayışını artırarak veri setindeki uzun vadeli bağımlılıkları ve ilişkileri daha etkin bir şekilde öğrenmesini sağlamıştır. Bu sayede, modelin doğruluk oranı artmış ve dil bilgisine dayalı tahmin kabiliyeti güçlendirilmiştir. Bu bulgular, modelin sıralı verileri işleme ve doğal dil işleme görevlerindeki başarısını önemli ölçüde etkileyen faktörler arasındadır.

Modelde üç adet LSTM katmanı bulunmaktadır. İlk LSTM katmanı, giriş olarak (None, 15, 1) boyutundaki veriyi alarak 100 boyutunda bir çıktı üretmektedir. İkinci LSTM katmanı, birinci katmanın çıktısını alarak (100 boyutunda) 150 boyutunda bir çıktı üretmektedir. Üçüncü LSTM katmanı, ikinci katmanın çıktısını alarak (150 boyutunda) ve 200 boyutunda bir çıktı üretmektedir.

Her LSTM katmanından sonra bir Dropout katmanı bulunmaktadır. Dropout katmanları, modelin aşırı uyum (overfitting) sorununu azaltmak için eğitim sırasında rastgele seçilen nöronları kapatmaktadır (sıfıra ayarlamaktadır). Bu, modelin daha sağlam (robust) olmasını sağlayarak eğitim veri setine fazla uyum sağlamasını önlemektedir. Her Dropout katmanı, kendisinden önceki LSTM katmanının çıktısını aldığı şekliyle çıktı vermektedir.

En son katman olan Dense katmanı, üçüncü LSTM katmanının çıktısını alarak modelin son tahminlerini yapması için kullanılmaktadır. Bu katmanın çıktı boyutu (None, 158) şeklinde gösterilmiştir. Bu, modelin her bir giriş için 158 farklı sınıf üzerinde bir tahmin yapabileceği anlamına gelmektedir. Bu da token tahmininde tahmin edilecek token sayısına karşılık gelmektedir.

Bu mimari, zaman serisi verileri veya sıralı verilerle çalışmak için yaygın olarak kullanılan bir yapıdır. LSTM katmanları, giriş dizilerindeki uzun süreli bağımlılıkları öğrenebilme yetenekleri ile bilinmektedirler. Token tahmini gibi doğal dil işleme görevlerinde sıklıkla kullanılmaktadır.

5.5. Test Sonuçları

Bu çalışmada, Python programlama dilinde sıkça karşılaşılan üç temel hata türü üzerinde odaklanılmıştır. Bunlar eksik yazımdan kaynaklanan söz dizimi hatası, fazla yazımdan kaynaklanan söz dizimi hatası ve yanlış yazımdan oluşan söz dizimi hatasıdır. Ayrıca, 25 adet yalnızca girinti hataları içeren kod parçaları da dahil edilmiştir. Toplamda 925 Python kod parçası üzerinde yapılan bu analiz, hem bu tez çalışmasında geliştirilen sistem üzerinde hem de "PyNar" adı verilen bir platformda test edilmiştir. PyNar Türkçe Python Editöründe yer alan sohbet robotu, Python kodlarını analiz ederek hataları tespit eden ve düzelten bir araçtır. Bu araç, özellikle Python diline özgü söz dizimi ve girinti hatalarını belirleme konusunda geliştirilmiştir. Araç şablon tabanlı yöntem kullanarak otomatik kod onarımı gerçekleştirmektedir.

Onarım sürecinde, her bir hata türüne özgü algoritmalar geliştirilmiş ve bu algoritmaların başarısı değerlendirilmiştir. Eksik kod parçaları için eksik olduğu tespit edilen yapıların tamamlanması, fazlalık içeren kodlar için ise gereksiz bölümlerin çıkarılması hedeflenmiştir. Yanlış yazım hataları ise detaylı bir analiz ve düzeltme sürecinden geçirilmiştir. Girinti hataları, Python dilinin en yaygın hatalarından biri olması nedeniyle özel bir öneme sahiptir ve bu çalışmada özel bir dikkatle ele alınmıştır.

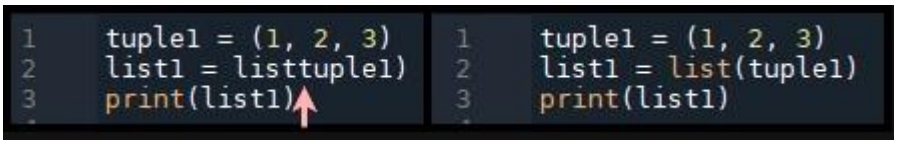
Çizelge 5.14'te bu tez çalışmasında geliştirilen yöntem ile Çizelge 5.1'de verilen hata türü ve hata mesajlarından oluşan dağılıma göre başarılı onarım yapılmış kod dağılımları verilmiştir.

Çizelge 5.14: Geliştirilen yöntem ile başarılı bir şekilde onarılan Python kodlarının dağılımı.

Hata Dağılımı	Eksik	Fazla	Yanlış
‘x’ was never closed	61	74	0
cannot assign to function call	12	0	0
unexpected EOF while parsing	37	0	0
invalid syntax	65	100	119
unmatched ‘x’	53	88	0
closing parenthesis 'x' does not match opening parenthesis 'y'	3	0	133
unterminated string literal	23	0	0
Toplam	258	262	252

Test veri setinde eksik yazıma bağlı olarak “‘x’ was never closed” hatası 61 örnekte tespit edilmiştir. Geliştirilen sistem, bu hata mesajını üreten tüm kodları %100 başarı oranıyla onarmıştır. “cannot assign to function call” hata mesajı test veri setinde 12 kez, “unexpected EOF while parsing” hata mesajı 37 kez görülmektedir. Her iki hata türü de sistem tarafından tamamen (%100) onarılmıştır. “closing parenthesis ‘x’ does not match opening parenthesis ‘y’” hata mesajı, test veri setinde sadece 3 örnekte bulunmuş ve bu hataların da tümü başarıyla düzeltilmiştir. Test veri setinde yer alan 98 “invalid syntax” hatasından 65'i (%66,3 oranında) sistem tarafından başarılı bir şekilde düzeltilmiştir. Ayrıca 32 “unterminated string literal” hatasından 23'ü (%71,8 oranında) sistem tarafından başarıyla onarılmıştır. Tez çalışmasında analiz edilen eksik hata türleri üzerinde gerçekleştirilen onarımlar, toplam hatalı kod sayısının %86'sının başarıyla düzeltilmesiyle sonuçlanmıştır. Bu oran, geliştirilen onarım sistemimizin eksikliklerin tespiti ve düzeltmesi konusunda yüksek bir etkinliğe sahip olduğunu ve bu tür yaygın programlama hatalarının otomatik düzeltilmesi alanında önemli bir ilerleme sağladığını göstermektedir.

Şekil 5.16’da eksik yazımdan kaynaklanan ve “unmatched x” hata mesajı üreten örnek bir Python kodu verilmiştir. Hatanın kaynaklandığı kısım ok işareti ile belirtilmiş ve sağ tarafta kodun onarılmış hali verilmiştir.



Şekil 5.16: Eksik yazımda “unmatched x” hata mesajı oluşturan Python kodu ve onarılmış hali.

Soldaki panelde, bir tuple (demet) tanımlanmış ve bu tuple'ı bir listeye çevirmek isteyen bir kod parçası hatalı bir şekilde gösterilmiştir. Bu hatalı kodda, list fonksiyonu parantezinin kapatılmadığı görülmektedir. Bu da "unmatched 'x'" hatasına yol açmaktadır. Sağdaki panelde ise bu hatanın düzeltilmiş versiyonu yer almaktadır. Burada list fonksiyonu için açılan parantez düzgün bir şekilde kapatılmış ve kodun doğru bir şekilde çalışması sağlanmıştır. Bu tür hatalar genellikle bir parantezin, tırnağın veya benzeri bir dilimleyicinin eksik olduğu durumlarda ortaya çıkar ve Python yorumlayıcısı tarafından söz dizimi hatası olarak belirlenmektedir. Kodun düzeltilmiş versiyonu, bu tür bir söz dizimi hatasının nasıl giderileceğini ve kodun nasıl düzgün bir şekilde çalıştırılacağını göstermektedir.

Şekil 5.17'de eksik yazımdan kaynaklanan ve "invalid syntax" hata mesajı üreten örnek bir Python kodu verilmiştir. Hatanın kaynaklandığı kısım ok işareti ile belirtilmiş ve sağ tarafta kodun onarılmış hali verilmiştir.

Şekil 5.17: Eksik yazımda "invalid syntax" hata mesajı oluşturan Python kodu ve onarılmış hali.

Soldaki kod parçasında range fonksiyonu parantezinden sonra bir iki nokta üst üste (:) karakterinin eksik olduğu görülmektedir. Python'da döngüler ve koşullu ifadeler gibi kontrol yapıları, bu iki nokta üst üste karakteri ile başlamaktadır. Bu karakterin eksikliği, yorumlayıcının bir söz dizimi hatası olarak algılamasına ve "invalid syntax" hatası vermesine neden olmaktadır. Düzeltme olarak sağdaki panelde, for döngüsünden sonra gerekli olan iki nokta üst üste karakteri eklenmiştir. Bu düzeltme ile kod artık Python söz dizimine uygun hale getirilmiş ve programın hata vermeden çalışmasını sağlamaktadır.

Şekil 5.18'de, eksik yazımdan kaynaklanan ve "unexpected EOF while parsing" hata mesajı üreten örnek bir Python kodu verilmiştir. Hatanın kaynaklandığı kısım ok işareti ile belirtilmiş ve sağ tarafta kodun onarılmış hali verilmiştir.

Şekil 5.18: Eksik yazımda "unexpected EOF while parsing" hata mesajı oluşturan Python kodu ve onarılmış hali.

Şekil 5.18, Python'da "unexpected EOF while parsing" hatası içeren bir kod örneği ve bu hatanın düzeltilmiş halini göstermektedir. Soldaki kod parçasında, print fonksiyonunun parentezleri tam olarak kapatılmamıştır. Yani kapatma parentezi eksik bırakılmıştır. Bu durum, yorumlayıcının ifadenin beklenmedik bir şekilde sona erdiğini ("end of file" veya dosya sonu) algılamasına ve "unexpected EOF while parsing" hatasını üretmesine sebep olmuştur. Sağ tarafta ise dize düzgün bir şekilde hem açılıp hem de kapatılmış ve böylece hata düzeltilmiştir. Bu düzeltmeyle fonksiyon artık Python söz dizimine uygun hale gelmiş ve yorumlayıcı tarafından hata vermeden çalıştırılabilecektir. Gösterilen düzeltilmiş kod, bu tür yaygın bir hata durumunun nasıl kolayca çözülebileceğini örneklendirmektedir.

Test veri setinde fazla yazım kaynaklı "'x' was never closed" hatası 74 örnekte tespit edilmiş ve geliştirilen sistem, bu hata mesajını üreten tüm kodları %100 başarı oranı ile onarmıştır. "Invalid syntax" hatası test veri setinde 138 örnekte görülmüş bunlardan 100'ü sistem tarafından %72,5 başarı oranıyla düzeltilmiştir. Ayrıca "unmatched 'x'" hatası 88 örnekte bulunmuş ve bu hataların tümü sistem tarafından %100 başarı ile onarılmıştır. Fazla yazıma bağlı olarak oluşan hataların genelinde ise sistem, %87,3 oranında başarılı bir onarım performansı göstermiştir. Bu sonuçlar, fazla yazım hatalarının tespiti ve onarımında geliştirilen metodolojinin etkileyici bir başarı gösterdiğini kanıtlamaktadır.

Şekil 5.19'da, fazla yazımdan kaynaklanan ve "invalid syntax" hata mesajı üreten örnek bir Python kodu verilmiştir. Hatanın kaynaklandığı kısım ok işareti ile belirtilmiş ve sağ tarafta kodun onarılmış hali verilmiştir.

```
def check_positive_negative(num):
    if num > 0:
        return "Pozitif"
    elif sdj num < 0:
        return "Negatif"
    else:
        return "Sifir"

result = check_positive_negative(-5)
print(result)
```

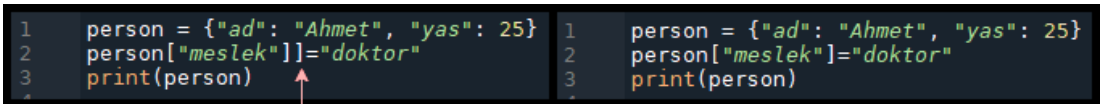
```
def check_positive_negative(num):
    if num > 0:
        return "Pozitif"
    elif num < 0:
        return "Negatif"
    else:
        return "Sifir"

result = check_positive_negative(-5)
print(result)
```

Şekil 5.19: Fazla yazımda "invalid syntax" hata mesajı oluşturan Python kodu ve onarılmış hali.

Soldaki kod parçasında, elif anahtar sözcüğünden sonra gelen sdj ifadesi sebebiyle fazladan bulunan karakterler söz dizimi hatasına neden olmaktadır. Düzeltme olarak sağ tarafta, sdj kelimesinin silindiği görülmüştür. Doğru bir şekilde yazılmış ve böylece fonksiyon artık geçerli bir Python söz dizimine sahiptir. Düzeltilmiş kod

bloğu, bir sayının pozitif, negatif ya da sıfır olup olmadığını kontrol eden ve buna göre bir string döndüren basit bir fonksiyonu içermektedir.

Şekil 5.20'de, fazla yazımdan kaynaklanan ve "unmatched x" hata mesajı üreten örnek bir Python kodu verilmiştir. Hatanın kaynaklandığı kısım ok işareti ile belirtilmiş ve sağ tarafta kodun onarılmış hali verilmiştir.



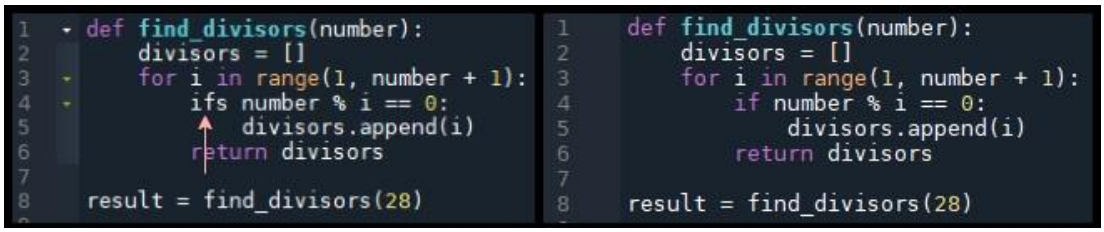
Şekil 5.20: Fazla yazımda "unmatched x" hata mesajı oluşturan Python kodu ve onarılmış hali.

Sol taraftaki kod bloğunda, person sözlüğüne meslek anahtarını eklemek için kullanılan köşeli parantez çift olarak yazılmış ve bu da "unmatched 'x'" hatasına yol açmıştır. Bu tür hatalar, genellikle bir parantezin veya tırnak işaretinin fazladan ya da eksik kullanılmasından kaynaklanır ve Python yorumlayıcısı tarafından saptanır.

Düzeltme olarak sağ taraftaki kodda, fazladan olan köşeli parantez (]) kaldırılmış ve person sözlüğüne meslek anahtarını ekleyen ifade düzeltilmiştir. Düzeltme yapıldıktan sonra print fonksiyonu sözlüğü doğru bir şekilde yazdırmaktadır. Kod hatasız bir şekilde çalışmaktadır. Gösterilen düzeltilmiş kod, Python'daki sözlüklerle çalışırken karşılaşılabilecek yaygın bir hata durumunun nasıl kolayca çözülebileceğini örneklendirmektedir.

Test veri setinde yanlış yazıma bağlı "Invalid syntax" hatası test veri setinde 167 kez mevcutken sistem bu hatalardan 119'unu %71,2 başarı oranı ile onarmıştır. Ayrıca, "not match opening parenthesis 'y'" hatası 133 örnekte tespit edilmiş ve bu hataların tümü sistem tarafından %100 başarı ile düzeltilmiştir. Yanlış yazımdan kaynaklanan hataların genelinde ise sistem %84'lük bir başarı oranı ile onarım gerçekleştirmiştir. Bu sonuçlar, yanlış yazımlardan kaynaklanan hataların düzeltilmesinde geliştirilen metodolojinin yüksek etkinliğini ve bu tür hataların giderilmesindeki güvenilirliğini göstermektedir.

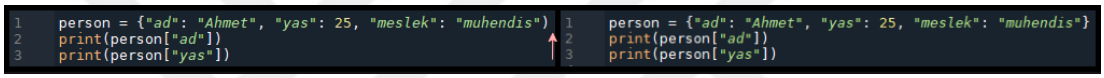
Şekil 5.21'de, yanlış yazımdan kaynaklanan ve "invalid syntax" hata mesajı üreten örnek bir Python kodu verilmiştir. Hatanın kaynaklandığı kısım ok işareti ile belirtilmiş ve sağ tarafta kodun onarılmış hali verilmiştir.



Şekil 5.21: Yanlış yazımda "invalid syntax" hata mesajı oluşturan Python kodu ve onarılmış hali.

Soldaki kod parçasında if ifadesi yanlış yazılmıştır. Sağdaki panelde ise if ifadesindeki yazımı düzeltilmiştir. Bu düzeltme ile kodun doğru çalışmasını sağlayacak Python söz dizimi hatası giderilmiştir.

Şekil 5.22, yanlış yazımdan kaynaklanan ve "invalid syntax" hata mesajı üreten örnek bir Python kodu verilmiştir. Hatanın kaynaklandığı kısım ok işareti ile belirtilmiş ve sağ tarafta kodun onarılmış hali verilmiştir.



Şekil 5.22: Yanlış yazımda "closing parenthesis 'x' does not match opening parenthesis 'y'" hata mesajı oluşturan Python kodu ve onarılmış hali.

Soldaki kod parçasında, print fonksiyonunun içindeki sözlük elemanlarına erişim sağlanırken kullanılan parantezlerin eşleşmesinde bir hata yapılmıştır. Bu tür bir eşleşme hatası, genellikle parantezlerin açılması ve kapatılması sırasında dikkatsizlik sonucu ortaya çıkmaktadır. Python yorumlayıcısı tarafından bir söz dizimi hatası olarak algılanmaktadır. Sağ taraftaki kod parçasında ise parantezlerin doğru bir şekilde eşleştirilmiş olduğu görülmektedir. Bu düzeltme ile print fonksiyonu artık geçerli Python söz dizimine uygun ve sözlükteki "ad" ve "yaş" anahtarlarına ait değerler doğru bir şekilde yazdırılabilir. Görseldeki düzeltilmiş kod, Python'da sözlük elemanlarına erişirken parantez kullanımının önemini ve doğru eşleştirme gerekliliğini vurgulamaktadır.

Bu karşılaştırma, eksik, yanlış ve fazla yazım hatalarına karşı sistem onarımının genel etkinliğini göstermektedir. Her üç kategoride de sistem, özellikle belirli hata türlerinde (%100 onarılanlar) kusursuz sonuçlar elde etmişken diğer hatalarda ise iyi bir başarı oranı elde etmiştir (en düşük %66,3). Bu sonuçlar, geliştirilen onarım sistemimizin her üç hata türünde de etkili olduğunu ve özellikle eksik ve fazla yazım hatalarında daha yüksek performans gösterdiğini kanıtlamaktadır.

Çizelge 5.15 oluşturulan test veri setinin PyNar Sohbet Robotu ile gerçekleştirilen test sonuçlarını göstermektedir.

Çizelge 5.15: PyNar Sohbet Robotu ile başarılı bir şekilde onarılan Python kodlarının dağılımı.

Hata Dağılımı	Eksik	Fazla	Yanlış
‘x’ was never closed	48	51	0
cannot assign to function call	9	0	0
unexpected EOF while parsing	30	0	0
invalid syntax	57	78	78
unmatched ‘x’	42	67	0
closing parenthesis 'x' does not match opening parenthesis 'y'	0	0	0
unterminated string literal	9	0	0
Toplam	195	196	78

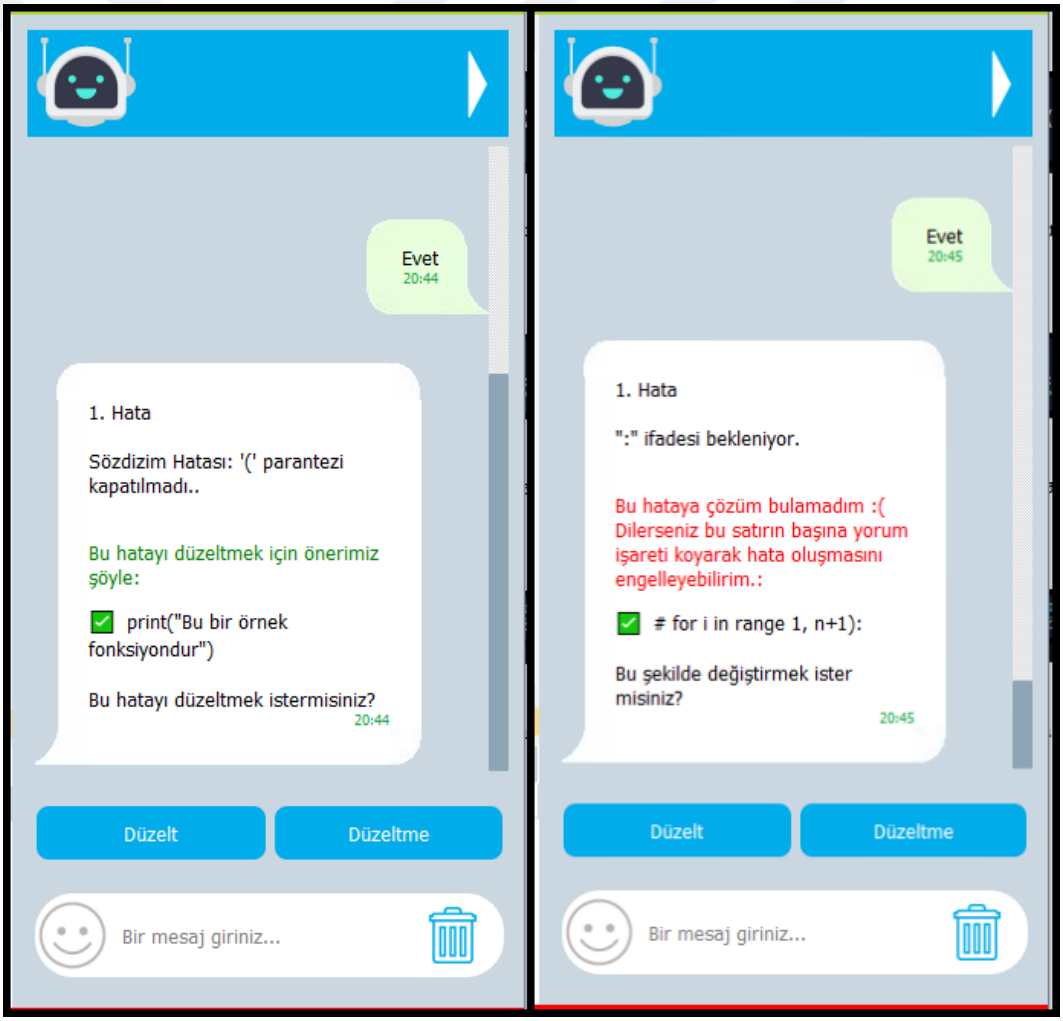
Test veri setinde eksik yazıma bağlı olarak “‘x’ was never closed” hatası 61 örnekte tespit edilmiştir. PyNar Sohbet Robotu, bu hata mesajını üreten 48 kodu (%78,7 oranında) başarıyla onarmıştır. “cannot assign to function call” hata mesajı test veri setinde 12 kez görülmüş ve PyNar Sohbet Robotu bu hataların 9’unu başarıyla onarmıştır. “unexpected EOF while parsing” hata mesajı 37 kez görülmektedir. PyNar Sohbet Robotu bu hataların 30’unu başarıyla onarmıştır. “unmatched ‘x’” hata mesajı 53 örnekte tespit edilmiş ve PyNar Sohbet Robotu bu hataların 42 tanesini (%79,2 oranında) onarmıştır. “closing parenthesis ‘x’ does not match opening parenthesis ‘y’” hata mesajı, test veri setinde sadece 3 örnekte bulunmuş ve bu hataların tümü başarıyla düzeltilmiştir. Test veri setinde yer alan 98 “invalid syntax” hatasından 57'si (%49 oranında) sistem tarafından başarılı bir şekilde düzeltilmiştir. Ayrıca, 32 “unterminated string literal” hatasından 9'u (%28,1 oranında) sistem tarafından başarıyla onarılmıştır. PyNar Sohbet Robotu ile analiz edilen eksik hata türleri üzerinde gerçekleştirilen onarımlar, toplam hatalı kod sayısının %65'inin başarıyla düzeltilmesiyle sonuçlanmıştır.

Test veri setinde fazla yazım kaynaklı "‘x’ was never closed" hatası 74 örnekte tespit edilmiş ve PyNar Sohbet Robotu, bu hata mesajını üreten kodların 51’ini (%58,9 oranında) düzeltmiştir. "Invalid syntax" hatası test veri setinde 138 örnekte görülmüş, bunlardan 78’i sistem tarafından %56,5 başarı oranıyla düzeltilmiştir. Ayrıca, "unmatched ‘x’" hatası 88 örnekte bulunmuş ve bu hataların 67’si (%76,1 oranında)

başarıyla onarılmıştır. PyNar tarafından fazla yazıma bağlı olarak oluşan hataların genelinde ise sistem, %65,3 oranında başarılı bir onarım performansı göstermiştir.

Test veri setinde yanlış yazıma bağlı "Invalid syntax" hatası test veri setinde 155 kez mevcutken PyNar Sohbet Robotu bu hatalardan 78'sini %50,3 başarı oranı ile onarmıştır. Ayrıca, "not match opening parenthesis ‘y’" hatası 133 örnekte tespit edilmiş ve bu hataların hiçbirini PyNar Sohbet Robotu onaramamıştır. Yanlış yazımdan kaynaklanan hataların genelinde ise PyNar %26’lık bir başarı oranı ile onarım gerçekleştirmiştir.

Şekil 5.23, PyNar Sohbet Robotu'nun programlama hatalarını tespit etme ve onarma süreçlerinin iki farklı örneğini göstermektedir.



Şekil 5.23: PyNar Sohbet Robotunun başarılı ve başarısız onarım için örnek mesajları.

Sol taraftaki görselde, PyNar Sohbet Robotu bir "Söz dizimi Hatası: '(' parantezi kapatılmadı..." hatasını tespit etmiş ve kullanıcıya bu hatayı düzeltmek için öneride bulunmuştur. Robot, hatalı print ifadesini göstermektedir. Kullanıcıya bu hatayı düzeltip düzeltmeyeceğini sormaktadır. Sağ taraftaki görselde ise robotun bir diğer hata tespiti ve çözüm önerisi yer almaktadır. Burada, robot "':' ifadesi bekleniyor." hatasını belirlemiş ve bu hata için bir çözüm önermemiştir. Ancak kullanıcıya kodun başına yorum işareti koyarak hatayı geçici olarak engelleyebileceğini belirtmiştir.

Bu tez çalışması kapsamında PyNar Sohbet Robotu'nun eksik, fazla ve yanlış yazım kaynaklı Python hatalarını onarma performansı üzerinde yapılan detaylı analiz, sistemin bu üç kategoride değişken bir başarı oranı gösterdiğini ortaya koymaktadır. Eksik yazım hatalarında, PyNar Sohbet Robotu genel olarak %65'lik bir başarı oranı ile etkileyici bir performans sergilemiştir. Özellikle "'x' was never closed" ve " unexpected EOF while parsing " gibi hatalarda yüksek oranlarda onarım başarısı elde etmiştir. Fazla yazım hatalarında ise, "'x' was never closed" ve "unmatched 'x'" hatalarının düzeltilmesinde daha düşük bir başarı oranı (%65,3) göstermiştir. Ancak yine de kabul edilebilir bir etkinlik sergilemiştir. Yanlış yazım hataları ise PyNar Sohbet Robotu bu kategoride %26'lık nispeten düşük bir başarı oranı ile karşılaşmıştır. " unexpected EOF while parsing " ve "not match opening parenthesis 'y'" hatalarında özellikle düşük performans göstermiştir. Bu sonuçlar, PyNar Sohbet Robotu'nun eksik ve fazla yazım hatalarını düzeltmede daha etkin olduğunu ancak yanlış yazım hatalarında geliştirilmesi gereken alanlar olduğunu göstermektedir.

Karşılaştırma sonuçlarına göre bu tez çalışması için geliştirilen yöntem, eksik yazım hatalarının geniş bir yelpazesi üzerinde yüksek bir başarı oranı (%86) göstermiştir. Özellikle "invalid syntax" ve "unterminated string literal" hatalarında belirgin bir üstünlüğe sahipken PyNar Sohbet Robotu bu hata türlerinde daha düşük oranlar elde etmiş ve %65'lik bir genel başarıya ulaşmıştır. Bu, her iki sistemin farklı hata türlerindeki etkinliğini ve gelişim alanlarını açıkça ortaya koymaktadır.

Fazla yazım hatalarını düzeltme konusunda, bu tez çalışması için geliştirilen yöntem PyNar Sohbet Robotu'ndan daha başarılı olmuştur. "'x' was never closed" ve "unmatched 'x'" hatalarını %100 başarı oranı ile düzelten yöntem, PyNar Sohbet Robotu'nun bu hatalarda gösterdiği daha düşük düzeltme oranlarına kıyasla dikkat çekicidir. Ayrıca, "Invalid syntax" hatasında da bu yöntem, PyNar Sohbet Robotu'na kıyasla daha yüksek bir başarı oranına sahiptir. Bu sonuçlar fazla yazım hatalarının

tespiti ve onarımında bu yöntemin PyNar Sohbet Robotu'ndan daha etkili olduğunu gösterir.

Yanlış yazımdan kaynaklanan hataların düzeltilmesi açısından bu tez çalışması için geliştirilen yöntem PyNar Sohbet Robotu'nu açık bir farkla geride bırakmıştır. "Not match opening parenthesis ‘y’" hatalarını %100 başarıyla düzelten bu yöntem, PyNar Sohbet Robotu'nun gösterdiği düşük başarı oranlarıyla net bir kontrast oluşturmaktadır. Özellikle PyNar'ın "Not match opening parenthesis ‘y’" hatasını hiç düzeltememesi, bu tez çalışması için geliştirilen yöntemin yanlış yazımdan kaynaklanan hataların düzeltilmesindeki etkinliğini ve güvenilirliğini vurgulamaktadır.

Python söz diziminde girinti hataları, programın akışını ve mantığını doğrudan etkileyen kritik bir role sahiptir. Çünkü Python girintiyi kod bloklarını tanımlamak için kullanmaktadır. Diğer dillerde süslü parantezler veya anahtar kelimeler ile ifade edilen yapısal sınırlar, Python'da girinti seviyeleri ile belirlenmektedir. Bu yüzden, girinti hataları yanlış kod yorumlamasına, beklenmeyen hatalara ve hatta programın çalışmaz hale gelmesine sebep olmaktadır. Python'da uygun girintileme, kodun okunabilirliğini ve bakımını kolaylaştırırken dilin net ve tutarlı yapısını koruma görevini de üstlenmektedir. Çizelge 5.16’da, girinti hatalarının düzeltilmesi konusunda bu tez çalışması için geliştirilen yöntem ve PyNar Sohbet Robotu arasında bir karşılaştırma yapılmıştır.

Çizelge 5.16: Girinti hataları için iki sistemin (geliştirilen yöntem ve PyNar Sohbet Robotu) karşılaştırılması.

Onarma Aracı	Geliştirilen Yöntem	PyNar Sohbet Robotu
Onarılan Kod Sayısı	22	15
Toplam Hatalı Kod Sayısı	25	25

Geliştirilen yöntem, toplam 25 hatalı koddan 22'sini düzeltirken (%88 başarı oranı) PyNar Sohbet Robotu 25 hatalı koddan 15'ini düzeltebilmiştir (%60 başarı oranı). Bu sonuçlar, geliştirilen yöntemin PyNar Sohbet Robotu'na göre girinti hatalarını onarma konusunda daha etkili olduğunu göstermektedir.

Geliştirilen yöntemin daha yüksek başarı oranı, girinti hatalarını daha iyi tespit edebilmesi ve düzeltebilmesi, daha gelişmiş bir hata düzeltme mekanizması ya da hata türlerine daha spesifik çözümler sunabilmesi gibi nedenlerden kaynaklanmaktadır.

Sonuçlar, geliştirilen onarım algoritmalarının etkinliğini göstermektedir. Her bir hata türü için onarım başarı oranları, hata türlerine göre ayrıntılı bir şekilde sunulmuş ve algoritmanın her bir hata türünde gösterdiği performans analiz edilmiştir. Bu çalışma, Python programlama dilinde hata düzeltme ve kod kalitesini artırma konusunda önemli bir adım olarak değerlendirilebilir ve gelecekteki çalışmalar için sağlam bir temel oluşturmuştur.

6. SONUÇLAR VE ÖNERİLER

Bu tez çalışmasının amacı, Python programlama dilindeki hata düzeltme süreçlerini otomatize eden algoritmaların geliştirilmesi ve bu algoritmaların etkinliğinin değerlendirilmesidir. Çalışma, 925 Python kod parçası üzerinde gerçekleştirilmiş ve eksik, fazla ve yanlış yazım kaynaklı hataların yanı sıra, Python'ın özgü bir özelliği olan girinti hataları üzerinde yoğunlaşmıştır. Geliştirilen yöntemler, PyNar Sohbet Robotu ile karşılaştırıldığında özellikle eksik (%86) ve fazla (%87,3) yazım hatalarının düzeltilmesinde yüksek bir başarı oranına ulaşmıştır. Yanlış yazım hatalarında da %84'lük etkileyici bir başarı sergilenmiştir.

Girinti hatalarının düzeltilmesi konusunda yapılan karşılaştırmada ise bu tez çalışması için geliştirilen yöntem %85,8'lik bir başarı oranı ile PyNar Sohbet Robotu'nun %53,3'lük başarısını geride bırakmıştır. Bu sonuçlar, geliştirilen yöntemin girinti hatalarını daha iyi anlama ve düzeltme kapasitesine sahip olduğunu göstermektedir. Yöntemin tüm veri seti üzerindeki başarısı %85,8 elde edilmiştir.

Bu tez çalışmasında elde edilen sonuçlar, Python programlama dilinde sık rastlanan söz dizimi ve girinti hatalarını onarma konusunda geliştirilen yöntemin başarısını gözler önüne sermektedir. Yapılan analizler, özellikle belirli hata türlerinde elde edilen yüksek başarı oranlarının yanı sıra sistemin bazı hata türlerinde daha fazla geliştirme ihtiyacını da ortaya koymuştur. Bu bağlamda, çalışmanın ilerletilmesine yönelik öneriler hem yöntemlerin daha da iyileştirilmesi hem de uygulama alanlarının genişletilmesi üzerine odaklanmaktadır.

Bu çalışma, Python dilindeki hataların düzeltilmesinde önemli bir ilerleme sağlamış olsa da modelin genişleyen kapasitesi sayesinde kod tamamlama, hata tespiti ve kod yorumlama gibi daha geniş bir yelpazedeki görevler için de eğitilebileceğini göstermektedir.

Yapay zeka tabanlı sistemler, özellikle transfer öğrenme gibi tekniklerle eğitildiğinde farklı programlama dilleri arasındaki bilgiyi genelleştirebilir. Bu sayede çoklu dil desteğini içeren geniş kapsamlı bir onarım aracı haline gelebilir. Transfer öğrenme, modelin bir dilden öğrendiği bilgileri başka bir dile uygulamasını sağlayarak, öğrenme

sürecini hızlandırabilir. Bu da modelin etkinliğini artırabilir. Bu yaklaşım, modelin çeşitli kod tabanlarında ve programlama görevlerinde daha etkin çalışmasına olanak tanırken aynı zamanda modelin yeni programlama dillerini öğrenme sürecini de kolaylaştırabilir.

Geliştirilen yöntemin gerçek dünya kod tabanları üzerinde daha geniş testlerle değerlendirilmesi ve geri bildirim mekanizmalarının entegrasyonu sağlanabilir. Kullanıcı etkileşimi sayesinde modelin sürekli olarak yeniden eğitilmesi ve geliştirilmesi sistemlerin doğruluk ve güvenilirliğini artırabilir. Bu tür interaktif sistemler, kullanıcıların karşılaştıkları hataları daha iyi anlamalarına ve çözüm yolları bulmalarına yardımcı olabilir. Ayrıca, modelin farklı kullanım senaryolarına ve hata türlerine adaptasyonu, onarım sürecinin daha kapsayıcı ve esnek olmasını sağlayabilir.

ChatGPT gibi büyük ölçekli dil modelleri, kapsamlı veri setleri ve yoğun hesaplama kaynakları gerektirirken bu geliştirilen sistem, daha mütevazı ölçekte verimli bir alternatif sunmaktadır. Özellikle küçük ve orta ölçekli uygulamalar için uygun olan bu sistem, benzer işlevselliği daha az kaynak kullanarak sağlamaktadır. Ayrıca, internet bağlantısına ihtiyaç duymadan çalışabilen bu model, çevrimdışı ortamlarda etkili bir şekilde kullanılmaktadır. Bu özellikler, özellikle kaynak kısıtlı ortamlarda ve internet erişiminin olmadığı durumlarda sistemimizin ChatGPT gibi büyük ölçekli modellere pratik bir alternatif oluşturduğunu göstermektedir.

Bu tez çalışmasının ortaya koyduğu yöntem, yapay zeka ve derin öğrenme teknikleri, programlama dillerinin karmaşık dünyasında hata düzeltme ve kod kalitesini artırma konusunda büyük bir potansiyel sunmaktadır. Önümüzdeki yıllarda, bu alandaki yeniliklerin, yazılım geliştirme pratiklerinde devrim yaratacak yeni araçlar ve metodolojilerin gelişimini hızlandıracağı beklenmektedir. Bu algoritmaların sürekli geliştirilmesi ve genişletilmesi hem eğitim hem de profesyonel yazılım geliştirme süreçlerinde değerli katkılarda bulunabilir ve otomatik hata düzeltme araçlarının daha yaygın kullanımını mümkün kılabilir.

KAYNAKLAR

AbdelMoniem, M., Gasser, S. M., El-Mahallawy, M. S., Fakhr, M. W., & Soliman, A. (2019). Enhanced NOMA system using adaptive coding and modulation based on LSTM neural network channel estimation. *Applied Sciences*, 9(15), 3022.preprint arXiv:1412.6980.

Abreu, R., Zoeteweij, P., & Van Gemund, A. J. (2006). An evaluation of similarity coefficients for software fault localization. *In 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)* (pp. 39-46). IEEE.

Abraham, A. (2005). Artificial neural networks. *Handbook of measuring system design*.

Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. (2021). Unified pre-training for program understanding and generation. *arXiv preprint*, arXiv:2103.06333.

Ahmed, T., Ledesma, N. R., & Devanbu, P. (2022). SYNSHINE: improved fixing of syntax errors. *IEEE Transactions on Software Engineering*, 49(4), 2169-2181.

Altun, M. (2020). *Herkes İçin Python Programlama Dili*.

Beebe, N. H. (1991). *Accurate hyperbolic tangent computation*. Dept. Math., Center Sci. Comput., Univ. Utah, Salt Lake City, UT, USA.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2).

Bishop, C. M. (1995). *Neural networks for pattern recognition*. Oxford university press.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33, 1877-1901.

Chen, Z., Hellendoorn, V. J., Lamblin, P., Maniatis, P., Manzagol, P. A., Tarlow, D., & Moitra, S. (2021). PLUR: A unifying, graph-based view of program learning, understanding, and repair. *Advances in Neural Information Processing Systems*, 34, 23089-23101.

Debroy, V., & Wong, W. E. (2010). Using mutation to automatically suggest fixes for faulty programs. *In 2010 Third International Conference on Software Testing, Verification and Validation* (pp. 65-74). IEEE.

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint*, arXiv:1810.04805.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. *arXiv preprint*, arXiv:2002.08155.

Gao, B., & Pavel, L. (2017). On the properties of the softmax function with application in game theory and reinforcement learning. *arXiv preprint*, arXiv:1704.00805.

Graver, J. E. (1975). On the foundations of linear and integer linear programming I. *Mathematical Programming*, 9(1), 207-226.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... & Zhou, M. (2020). Graphcodebert: Pre-training code representations with data flow. *arXiv preprint* arXiv:2009.08366.

Gupta, R., Pal, S., Kanade, A., & Shevade, S. (2017). Deepfix: Fixing common c language errors by deep learning. *In Proceedings of the aaai conference on artificial intelligence* (Vol. 31, No. 1).

Gupta, R., Kanade, A., & Shevade, S. (2019). Deep reinforcement learning for syntactic error repair in student programs. *In Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 33, No. 01, pp. 930-937).

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735-1780.

Hua, J., & Khurshid, S. (2017). EdSketch: Execution-driven sketching for Java. *In Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software* (pp. 162-171).

Hua, J., Zhang, M., Wang, K., & Khurshid, S. (2018). Towards practical program repair with on-demand candidate generation. *In Proceedings of the 40th international conference on software engineering* (pp. 12-23).

Jiang, N., Lutellier, T., & Tan, L. (2021). Cure: Code-aware neural machine translation for automatic program repair. *In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE*) (pp. 1161-1173). IEEE.

José Campos, André Riboira, Alexandre Perez, and Rui Abreu. (2012). GZoltar: an eclipse plug-in for testing and debugging. *In Proceedings of the 27th IEEE/ACM International*.

Kavianpour, P., Kavianpour, M., Jahani, E., & Ramezani, A. (2023). A cnn-bilstm model with attention mechanism for earthquake prediction. *The Journal of Supercomputing*, 1-33.

Kelley, A. K. (2018). *A system for classifying and clarifying Python syntax errors for educational purposes* (Doctoral dissertation, Massachusetts Institute of Technology).

Kim, D., Nam, J., Song, J., & Kim, S. (2013). Automatic patch generation learned from human-written patches. *2013 35th International Conference on Software Engineering (ICSE)*, 802-811.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv Sibi, P., Jones, S. A., & Siddarth, P. (2013). Analysis of different

activation functions using back propagation neural networks. *Journal of theoretical and applied information technology*, 47(3), 1264-1268.

Koyuncu, A., Liu, K., Bissyandé, T. F., Kim, D., Klein, J., Monperrus, M., & Le Traon, Y. (2020). Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25, 1980-2024

Lajkó, M., Csuvik, V., & Vidács, L. (2022). Towards JavaScript program repair with generative pre-trained transformer (GPT-2). *In Proceedings of the Third International Workshop on Automated Program Repair* (pp. 61-68).

Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W. (2011). Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1), 54-72.

LeCun, Y., Bottou, L., Orr, G. B., & Müller, K. R. (2002). Efficient backprop. *In Neural networks: Tricks of the trade* (pp. 9-50). Berlin, Heidelberg: Springer Berlin Heidelberg.

Li, D., Yan, L., Yang, J., & Ma, Z. (2022). Dependency syntax guided bert-bilstm-gam-crf for chinese ner. *Expert Systems with Applications*, 196, 116682.

Li, X., Wang, L., Xin, Y., Yang, Y., & Chen, Y. (2020). Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences*, 10(5), 1692.

Liang, R., Li, J., Huang, L., Lin, R., Lai, Y., & Xiong, D. (2021, November). Extractive-Abstractive: A Two-Stage Model for Long Text Summarization. *In CCF Conference on Computer Supported Cooperative Work and Social Computing* (pp. 173-184). Singapore: Springer Nature Singapore.

Liu, K., Koyuncu, A., Kim, D., & Bissyandé, T. F. (2019). Avatar: Fixing semantic bugs with fix patterns of static analysis violations. *In 2019 IEEE 26th International Conference on Software Analysis*, Evolution and Reengineering (SANER) (pp. 1-12). IEEE.

Liu, X., & Zhong, H. (2018). Mining stackoverflow for program repair. *In 2018 IEEE 25th international conference on software analysis, evolution and reengineering* (SANER) (pp. 118-129). IEEE.

Long, F., Amidon, P., & Rinard, M. (2017). Automatic inference of code transforms for patch generation. *In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (pp. 727-739).

Lu, W. (2000). *Neural network model for distortional buckling behaviour of cold-formed steel compression members at elevated temperatures* (Doctoral dissertation, Queensland University of Technology).

Martinez, M., & Monperrus, M. (2016). Astor: A program repair library for java. *In Proceedings of the 25th International Symposium on Software Testing and Analysis* (pp. 441-444).

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5, 115-133.

Microsoft. (2019). Pyright. GitHub repository. https://github.com/microsoft/pyright

Moosavi, S., Vahidi-Asl, M., Haghighi, H., & Rezaalipour, M. (2023). Doctor Code: A Machine Learning-Based Approach to Program Repair. *Scientia Iranica*.

Naumov, M. (2017). Feedforward and recurrent neural networks backward propagation and hessian in matrix form. *arXiv preprint*, arXiv:1709.06080.

Pan, Z., Gu, Z., Jiang, X., Zhu, G., & Ma, D. (2022). A modular approximation methodology for efficient fixed-point hardware implementation of the sigmoid function. *IEEE Transactions on Industrial Electronics*, 69(10), 10694-10703.

Pratiwi, N. C., Ibrahim, N., Fu'adah, Y. N., & Masykuroh, K. (2020, December). Computer-aided detection (CAD) for COVID-19 based on chest X-Ray images using convolutional neural network. *In IOP Conference Series: Materials Science and Engineering* (Vol. 982, No. 1, p. 012004). IOP Publishing.

Prenner, J. A., Babii, H., & Robbes, R. (2022). Can OpenAI's codex fix bugs? an evaluation on QuixBugs. *In Proceedings of the Third International Workshop on Automated Program Repair* (pp. 69-75).

Pritchard, D. (2015). Frequency distribution of error messages. *In Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools* (pp. 1-8).

PyNar-Editor. (2018). [GitHub deposu]. GitHub. https://github.com/PyNar-editor

Python Software Foundation. (1994). re — Regular expression operations. Erişim tarihi: Mayıs, 2023], https://docs.Python.org/3/library/re.html.

Python Software Foundation. (2001). tokenize — Tokenizer for Python source. Erişim tarihi: Nisan, 2023. Python Software Foundation. https://docs.Python.org/3/library/tokenize.html

Python Software Foundation. (2019). Abstract Syntax Trees (ast) - Python 3.8.0 documentation. https://docs.Python.org/3/library/ast.html

Qi, Z., Long, F., Achour, S., & Rinard, M. (2015). An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. *In Proceedings of the 2015 International Symposium on Software Testing and Analysis* (pp. 24-36).

Rajendran, R., & Murugan, B. S. (2022, December). An Efficient Pelican optimization based CNN-BiLSTM to Detect and Classify 3D Objects. *In 2022 International Conference on Knowledge Engineering and Communication Systems (ICKES)* (pp. 1-8). IEEE.

Rhanoui, M., Mikram, M., Yousfi, S., & Barzali, S. (2019). A CNN-BiLSTM model for document-level sentiment analysis. Machine Learning and Knowledge Extraction, 1(3), 832-847.

Rolim, R., Soares, G., Gheyi, R., Barik, T., & D'Antoni, L. (2018). Learning quick fixes from code repositories. *arXiv preprint*, arXiv:1803.03806.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint*, arXiv:1609.04747.

Saha, R. K., Lyu, Y., Yoshida, H., & Prasad, M. R. (2017). Elixir: Effective object-oriented program repair. *In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 648-659). IEEE.

Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11), 2673-2681.

Staudemeyer, R. C., & Morris, E. R. (2019). Understanding LSTM--a tutorial into long short-term memory recurrent neural networks. *arXiv preprint*, arXiv:1909.09586.

Tam, S., Said, R. B., & Tanriöver, Ö. Ö. (2021). A ConvBiLSTM deep learning model-based approach for Twitter sentiment classification. *IEEE Access*, 9, 41283-41293.

Valueian, M., Vahidi-Asl, M., & Khalilian, A. (2022). SituRepair: Incorporating machine-learning fault class prediction to inform situational multiple fault automatic program repair. *International Journal of Critical Infrastructure Protection*, 37, 100527.

Van Rossum, G., & Drake Jr, F. L. (1995). Python tutorial (Vol. 620). Amsterdam, The Netherlands: Centrum voor Wiskunde en Informatica.

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint*, arXiv:2109.00859.

Wen, M., Chen, J., Wu, R., Hao, D., & Cheung, S. C. (2018). Context-aware patch generation for better automated program repair. In *Proceedings of the 40th international conference on software engineering* (pp. 1-11).

Weimer, W., Nguyen, T., Le Goues, C., & Forrest, S. (2009). Automatically finding patches using genetic programming. *In 2009 IEEE 31st International Conference on Software Engineering* (pp. 364-374). IEEE.

Williams, R. J., & Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2), 270-280.

Xu, G., Meng, Y., Qiu, X., Yu, Z., & Wu, X. (2019). Sentiment analysis of comment texts based on BiLSTM. *Ieee Access*, 7, 51522-51532.

Yasunaga, M., & Liang, P. (2021). Break-it-fix-it: Unsupervised learning for program repair. *In International Conference on Machine Learnin*g (pp. 11941-11952). PMLR.

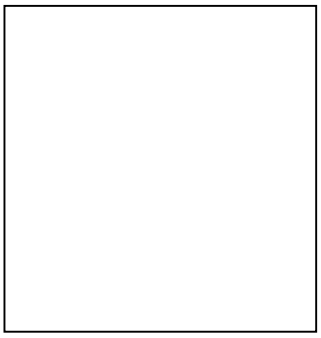
Zhang, Z., & Sabuncu, M. (2018). Generalized cross entropy loss for training deep neural networks with noisy labels. *Advances in neural information processing systems*, 31.

Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

EKLER

EK A: https://github.com/senadikici/tez-veriseti

ÖZGEÇMİŞ



Ad-Soyad **:** Sena DİKİCİ

Doğum Tarihi ve Doğum Yeri **:**

E-posta **:**

ÖĞRENİM DURUMU:

- **Lisans** **:** 2020, Gazi Üniversitesi, Teknoloji Fakültesi, Bilgisayar Mühendisliği

MESLEKİ DENEYİM:

- Bursa Teknik Üniversitesi (Araştırma Görevlisi), 26.10.2021- Halen çalışıyor

YAYINLAR:

Dikici, S., & ALTUNTAŞ, V. (2023). Yapay Sinir Ağları Kullanılarak Protein Katlanması Tanıma. Bilişim Teknolojileri Dergisi, 16(2), 95-105. https://doi.org/10.17671/gazibtd.1141468.

Dikici, S., & ALTUNTAŞ, V. (2022). Makine Öğrenmesi Yöntemleri ile Protein Katlanması Tanıma. Socrates 4th Internatıonal Conference On Engıneerıng, Health And Applıed Scıences.