

YAPAY ZEKA VE MAKİNE ÖĞRENMESİ

KURS NOTLARI

Kaan ASLAN

C VE SİSTEM PROGRAMCILARI DERNEĞİ

Son Güncelleme Tarihi: 28/07/2021

Bu kurs notları Kaan ASLAN tarafından yazılmıştır. Kaynak belirtilmek koşuluyla her türlü alıntı yapılabilir.

GİRİŞ

Bu bölümünde "yapay zeka (artificial intelligence) ve makine öğrenmesi (machine learning)" konusuna bir giriş yapılmaktadır.

Yapay Zeka (Artificial Intelligence) Nedir?

Zeka kapsamı, işlevleri ve yol achiği sonuçları bakımından karmaşık bir olgudur. Zekanın ne olduğu konusunda psikologlar ve nörobilimciler arasında tam bir fikir birliği bulunmamaktadır. Çeşitli kuramcılar ve araştırmacılar tarafından zekanın çeşitli tanımları yapılmıştır. Bu kuramcılar ve araştırmacıların bazıları yaptıkları tanımdan hareketle zekayı ölçmek için çeşitli araçlar da geliştirmeye çalışmışlardır.

Charles Spearman zekayı "g" ve "s" biçiminde iki yeteneğin birleşimi olarak tanımlamıştır. Spearman'a göre "g" faktörü akıl yürütme ve problem çözmeyle ilgili olan "genel zekayı" belirtir. "s" faktörü ise müzik, sanat, iş yaşamı gibi özel alanlara yönelik "spesifik zekayı" belirtir. İnsanlar "zeka" denildiğinde genel zekayı kastetmektedirler.

Howard Gardner tarafından kuramsal hale getirilen "çoklu zeka (multiple intelligence)" zekayı "sözel/bilbilimsel (verbal/linguistic), müziksel (musical), mantıksal/matematiksel (logical/mathematical), görsel uzamsal (visual/spatial), kinestetik (kinesthetic), kişilerarası (interpersonalı, içsel (intrapersonal)", olmak üzere başlangıcta yedi türə ayırmıştır. Sonra bu yedi türə "doğasal (naturalistic), varoluşsal (existentialist)" biçiminde iki tür daha ekleyerek dokuzu çıkarmıştır.

Robert Sternberg'e göre ise "analitik (analytical), pratik (practical) ve yaratıcı (creative)" olmak üzere üç tür zeka vardır. Analitik zeka problemi parçalara ayırma, analiz etme ve çözme ile ilgili yetileri içermektedir. Bu yetiler aslında zeka testlerinin ölçmeye çalıştığı yetilerdir. Pratik zeka yaşamı sürdürmek için gerekli olan pratik becerilerle ilgilidir. Yaratıcı zeka ise yeni yöntemler bulmak, problemleri farklı biçimlerde çözebilmek, yenilikler yapabilmekle ilgili yetilerdir.

Catell-Horn-Carroll (CHC) Teorisi diye isimlendirilen çok katmanlı zeka teorisi üzerinde en çok durulan zeka teorisidir. (Raymond Catell aslında Spearman'ın John Horn ise Catell'in öğrencisidir.) Catell zekayı "kristalize zeka (crystalized intelligence)" ve "akıcı zeka (fluid intelligence)" biçiminde ikiye ayırmıştır. Kristalize zeka öğrenilmiş ve oturmuş bilgi ve becerilerle ilgili iken akıcı zeka problem çözme ve yeni durumlara uyum sağlama becerileriyle ilgilidir. John Horn ise Catell'in bu iki tür zekasını genişleterek ona görsel işitsel yetileri, belleğe erişimle ilgili yetileri, tepki zamanlarına ilişkin yetileri, niceliksel işlemlere yönelik yetileri, ve okuma yazma becerilerini de eklemiştir. Nihayet John Carroll zeka ile

İlgili 460 yeteneği faktör analizine sokarak üç katmanlı bir zeka teorisi oluşturmuştur. CHC teorisi Stanford Binet ve Wechles testlerinin ileri sürümlerinin benimediği zeka anlayışıdır.

Yapay zeka ise ismi üzerinde insan zekası ile ilgili bilişsel süreçlerin makineler tarafından sağlanmasına yönelik süreçleri belirtmektedir. Yapay zeka terimi ilk kez 1955 yılında John McCarthy tarafından uydurulmuştur. Doğal zekada olduğu gibi yapay zekanın da farklı kişiler tarafından pek çok tanımı yapılmaktadır. Ancak bu terim genel olarak "insana özgü nitelikler olduğu varsayılan akıl yürütme, anlam çıkartma, genelleme ve geçmiş deneyimlerden öğrenme gibi yüksek zihinsel süreçlerin makineler tarafından gerçekleştirilebilmesidir" biçiminde tanımlanabilir. Yapay zekanın diğer bazı tanımları şunlardır:

- Yapay zeka insan zekasına ilişkin "öğrenme", "akıl yürütme", "kendini düzeltme" gibi süreçlerin makineler tarafından simüle edilmesidir.
- Yapay zeka zeki makineler yaratma amacında olan bilgisayar bilimlerinin bir alt alanıdır.
- Bilgisayarların insanlar gibi davranışmasını sağlamayı hedefleyen bilgisayar bilimlerinin bir alt dalıdır.

Yapay zekanın simüle etmeye çalıştığı bilişsel süreçlerin şunlar olduğuna dikkat ediniz:

- Bir şeyin nasıl yapılacağını bilme (knowledge)
- Akıl yürütme (reasoning)
- Problem çözme (problem solving)
- Algılama (perception)
- Öğrenme (learning)
- Planlama (planning)
- Doğal dili açıklama ve konuşma
- Uzmanlık gerektiren alanlarda karar verme

Yapay Zeka Çalışmalarının Kısa Tarihi

Yapay zeka ile ilgili düşünceler ve görüşler antik çağ'a kadar götürülebilir. Ancak modern yapay zeka çalışmalarının 1950'li yıllarda başladığı söylenebilir. Şüphesiz yapay zeka alanındaki gelişmeler de aslında başka alanlardaki gelişmeler tetiklemiştir. Örneğin bugün kullandığımız elektronik bilgisayarlar olmasaydı yapay zeka bugünkü durumuna gelemeyecekti. İşte aslında pek çok bilimsel ve teknolojik gelişmeler belli bir noktaya gelmiş ve yapay zeka dediğimiz bu alan 1950'lerde ortaya çıkmaya başlamıştır.

Yapay zeka çalışmalarının ortayamasına yol açan gelişmeler şunlar olmuştu:

- **Mantıktaki Gelişmeler:** Bertrand Russell ve North Whitehead tarafından 1913 yılında yazılmış olan "Principia Mathematica" adlı kitap biçimsel mantıkta (formal logic) devrim niteliğinde etki yapmıştır.
- **Matematikteki Gelişmeler:** 1930'larda Alonzo Church "Lambda Calculus" geliştirmiş ve özyinelemeli fonksiyonel notasyonla hesaplanabilirliği araştırılmış ve sorgulamıştır. Yine 1930'larda Kurt Gödel "biçimsel sistemler (formal system)" üzerindeki çalışmalarıyla teorik bilgisayar bilimlerinin öncülüğünü yapmıştır.
- **Turing Makineleri:** Alan Turing'in henüz elektronik bilgisayarlar gerçekleştirilmeden 1930'lu yılların ortalarında (ilk kez 1936) tasarladığı teorik bilgisayar yapısı olan "Turing Makineleri" bilgisayar bilimlerinin ve yapay zeka kavramının ortaya çıkışında etkili olmuştur. (Turing makinelerinin çeşitli modelleri vardır. Bugün hala Turing makineleri algoritmalar dünyasında algoritma analizinde ve algoritmik karmaşıklıkta teorik bir karşılaştırma amacıyla kullanılmaktadır.)

- Elektronik Bilgisayarların Ortaya Çıkması: 1940'lı yıllarda ilk elektronik bilgisayarlar gerçekleştirilmeye başlanmıştır. Bilgisayarlar yapay zeka alanının gelişmesinde için en önemli araçlar durumundadır.

Yapay Zeka (Artificial Intelligence) terimi ilk kez John McCarthy tarafından 1955 yılında uydurulmuştur. John McCarthy, Marvin Minsky, Nathan Rochester ve Claude Shannon tarafından 1956 yılında Dartmouth College'de bir konferans organize etmişlerdir. Bu konferans yapay zeka kavramının ortaya çıkışının bakımından çok önemlidir. Bu konferans yapay zekanın doğumu olarak kabul edilmektedir. Aynı zamanda yapay zeka terimi de bu konferansta katılımcılar tarafından kabul görmüştür. John McCarthy aynı zamanda dünyanın ilk programlama dillerinden biri olan Lisp'i de 1958 yılında tasarlamıştır. Lisp hala yapay zeka çalışmalarında kullanılmaktadır.

1956-1974 yapay zekanın altın yılları olmuştur. Bu yıllar arasında çeşitli algoritmik yöntemler geliştirilmiş ve pek çok uygulama üzerinde çalışılmıştır. Örneğin arama (search) yöntemleri uygulanmış ve arama uzayı (search space) sezgisel (heuristic) yöntemlerle daraltılmaya çalışılmıştır. Yine bu yıllarda doğal dili anlamaya yönelik ilk çalışmalar gerçekleştirilmiştir. Bu ilk çalışmalarдан elde edilen çeşitli başarılar yapay zeka alanında iyimser bir hava estirmiştir. Örneğin:

- 1958 yılında Simon ve Newell "10 yıl içinde dünya satranç şampiyonunun bir bilgisayar olacağını" iddia etmişlerdir. (Halbuki bu durum 90'lı yılların ikinci yarısında gerçekleşmeye başlamıştır.)
- 1970 yılında Minsky 3 yıldan 8 yıla kadar makinelerin ortalama bir insan zekasına sahip olabileceği iddia etmiştir.

1974-1980 yılları arasında yapay zeka alanında bir durgunluk yaşanmıştır. Daha önce yapılan tahminlerin çok iyimser olduğu görülmüş bu da biraz hayal kırıklığına yol açmıştır. Bu yıllarda yapay sinir ağları çalışmaları büyük ölçüde durmuştur. Yeni projeler için finans elde edilmesi zorlaşmıştır.

1980'li yıllarla birlikte yapay zeka çalışmalarında yine yükseliş başlamıştır. 80'li yıllarda en çok yükselişe geçen yapay zeka alanı "uzman sistemler" olmuştur. Japonya bu tür projelere önemli finans ayırmaya başlamıştır. Ayrıca Hopfield ve Rumelhart'in çalışmaları da "yapay sinir ağlarına" yenik bir soluk getirmiştir.

1987-1993 yılları arasında yine yapay zeka çalışmalarında bir duraklama baş göstermiştir. Bu konudaki çeşitli projeler için finans akynakları da kendilerini geri çekmiştir.

1993 yılından itibaren yapay zeka alanı yine canlanmaya başlamıştır. Bilgisayarların güçlenmesi, Internet teknolojisinin gelişmesi, mobil aygıtların gittikçe yaygınlaşması sonucunda veri analizinin önemi artmış ve bu da yapay zeka çalışmalarına yeni bir boyut getirmiştir. 1990'lı yılların ortalarından itibaren veri işlemesinde yeni bir dönem başlamıştır. Veri madenciliği bir alan olarak kendini kabul ettirmiştir. Özellikle 2011 yılından başlayarak büyük veri (big data) analizleri iyice yaygınlaşmış, yapay sinir ağlarının bir çeşidi olan derin öğrenme (deep learning) çalışmaları hızlanmış, IoT uygulamaları da yapay zekanın önemini heften artırmıştır.

Yapay Zekanın Alt Alanları

Yapay zeka aslında pek çok alt konuya ayrılabilen bir alandır. Yapay zekanın önemli alt alanları şunlardır:

- Makine Öğrenmesi
- Yapay Sinir Ağları ve Derin Öğrenme
- Bulanık Sistemler (Fuzzy Logic)
- Evrimsel Yöntemler (Genetic Algorithms, Differential Evolution)
- Üst Sezgisel (Meta Heuristic) Yöntemler (Karınca Kolonisi, Particle Swarm Optimization)
- Olasılıksal (Probabilistic) Yöntemler (Bayesian Network, Hidden Markov Model, Kalman Filter vs.)
- Uzman Sistemler (Expert Systems)

Yapay Zekanın Uygulama Alanları

Yapay zekanın tipik uygulama alanları şunlardır:

- Yapay Yaratıcılık Faaliyetleri (Artificial Creativity)
- Planlama ve Çizelgeleme Faaliyetleri
- Akıl Yürütme (Reasoning) Faaliyetleri
- Otomatik Hedef Belirlemesi (Automatic Target Recognition)
- Yüz Tanıma Sistemleri (Facial Recognition Systems)
- Konuşma Tanıma (Speech Recognition)
- Konuşanı Tanıma (Speaker Recognition)
- Bilgisayarlı Görü (Computer Vision)
- Görüntü İşleme
- OCR (Optical Character Recognition) ve Zeki Sözcük Tanıma (Intelligent Word Recognition)
- Nesne Tanıma (Object Recognition)
- El Yazısı Tanıma
- Yüz Tanıma
- Hastalığa Tanı Koyma
- Uzman Sistemler
- Karar Destek Sistemleri
- Klinik Karar Destek Sistemleri
- Zeka Oyunlarını Oynaması (Satranç, Go, vs.)
- Veri Madenciliği (Data Mining)
- Yazı Madenciliği (Text Mining)
- Süreç Madenciliği (Process Mining)
- E-Posta Spam Filtreleri
- Etkinlik Belirleme (Activity Recognition)
- Otomatik Resim Yorumlama (Automatic Image Annotation)
- İlişkili Olanları Bulma (Relationship Extraction)
- Referans Çözümlemesi (Coreference Resolution)
- Doğal Dil Algılama
- Dil Algılama
- Makine Çevirisi
- Semantic Web
- Soru Yanıtlama Sistemleri
- Örnek Tanıma
- İnsan Gibi Davranan Robotlar
- Sesli Konuşma Sistemleri (Text To Speech, Speech to Text)
- Sanal Gerçeklik Sistemleri

Makine Öğrenmesi Nedir?

Aslında makine öğrenmesinin ne olduğundan önce öğrenmenin ne olduğunu ele almak gereklidir. Psikolojide öğrenme "davranışta göreli biçimde kalıcı değişiklikler oluşturan süreçler" olarak tanımlanmaktadır. Bu tanımdaki davranış (behavior) klasik davranışçılar göre "gözlemlenebilen devinimleri" kapsamaktadır. Ancak daha sonra "radikal davranışçılar" bu davranış kavramını zihinsel süreçleri de kapsayacak biçimde genişletmiştir.

Psikoloji de öğrenme kabaca dört bölümde ele alınmaktadır.

1) Klasik Koşullanma (Classical Conditioning): Burada dışsal bir uyaran başka bir uyaranla zamansal bakımdan eşleşmiştir. Böylece birinci uyaran oluştugunda organizma ikinci uyaranın olusacagini öğrenir. Pavlov'un köpek deneyi klasik koşullanma ile ilgilidir. Klasik koşullanma süreci pek çok hayvan üzerinde denenmiştir. Hayvanların çok büyük çoğunluğu klasik koşullanma ile öğrenebilmektedir. Birinci kuşak davranışçılar pek çok davranışın nedenini klasik koşullanmaya açıklamışlardır. Gerekten de fobilerin çoğunda klasik koşullanmanın etkili olduğu görülmektedir. Özellikle olumsuz birtakım sonuçlar doğuran uyaranlar çok kısa süre içerisinde klasik koşullanmaya yol açabilmektedir. Örneğin karanlık bir sokakta saldırıcı ugrayan kişi yeniden karanlık bir sokağa girdiğinde saldırıcı ugrayacağı hissine kapılabilmektedir.

2) Edimsel Koşullanma (Operant Conditioning): Edimsel koşullanma en önemli insan öğrenmesi yollarından biridir. Pek çok süreç edimsel koşullanma ile öğrenilmektedir. Klasik koşullanmada önce uyaran sonra tepki gelmektedir. Halbuki edimsel koşullanmada önce tepki sonra uyaran gelir. Organizma bir faaliyette bulunur. Bunun sonucunda hoşa giden bir durum (yani ödül) oluşursa bu davranış tekrarlanır ve böylece öğrenme gerçekleşir. Yani özetle organizmada hoşa giden sonuçlar doğuran davranışlar tekrarlanma eğilimindedir. Bu öğrenme modelinde davranış sonucunda oluşan hoşa giden durumlara "pekiştireç (reinforcer)" denilmektedir. Davranış ne kadar pekiştirilirse o kadar iyi öğrenilmektedir. Pekiştireçler "pozitif" ve "negatif" olmak üzere ikiye ayrırlar. Pozitif pekiştireçler doğrudan organizmanın hoşuna gidecek uyarlardır. Negatif pekiştireçler ise organizmanın içinde bulunduğu hoş olmayan durumu ortadan kaldırır uyaranlardır. Edimsel koşullanma için bazı örnekler şöyle verilebilir:

- Ödevini yapan öğrenciye öğretmenin ödül vermesi ödev yapma davranışını artırmaktadır.
- Maddenin bunaltısı (anxiety) ortadan kaldırması kişiyi madde kullanımına teşvik etmektedir.
- Arabada emniyet kemeri bağlı değilken ses çıkmaktadır. Bu sesi ortadan kaldırmak için emniyet kemерinin bağlanması bir edimsel koşullanma sürecidir.

3) Sosyal Bilişsel Öğrenme (Social Cognitive Learning): Bu yönteme taklit yoluyla öğrenme de denilmektedir. Biz başkalarını taklit ederek de davranışlarımızı değiştirebilmekteyiz. Aslında sosyal bilişsel öğrenmede de bir bakıma pekiştirmeler söz konusudur. Ancak bu pekiştirme doğrudan değil dolaylı (vicarious) olmaktadır. Sosyal bilişsel öğrenmede birtakım bilişsel süreçlerin de devreye girdiğine dikkat ediniz. Çünkü bu süreçte kişinin başkalarının yaptığı davranışları izleme, izlediklerini bellekte saklama ve onlardan sonuçlar çıkartma süreçleri de söz konusu olmaktadır.

4) Bilişsel Öğrenme (Cognitive Learning): Biliş (cognition) organizmanın bilgi işlem faaliyetlerini anlatan bir terimdir. Biliş denildiğinde düşünme, bellek, dikkat, bilinç, akıl yürütme gibi faaliyetler anlaşılmaktadır. Araştırmacılar hiç pekiştireç olmadan öğrenmenin hayvanlarda da insanlarda da mümkün olduğunu göstermişlerdir. Yani biz klasik koşullanma, edimsel koşullanma ve sosyal öğrenme süreçleri olmadan da yalnızca bilişsel faaliyetlerle de öğrenebiliriz.

Halk arasında öğrenme denildiğinde genellikle sürecin davranışsal boyutu göz ardı edilmekte yalnızca bilişsel tarafı değerlendirilmektedir. Halbuki her türlü öğrenmede açık ya da örtük görelî bir biçimde kalıcı bir davranışın ortaya çıkması beklenir. Ancak "davranış (behavior)" sözcüğünün tanımı konusunda da tam bir anlaşma bulunmamaktadır. İlk davranışçılar yalnızca gözlemlenebilen devinimleri davranış olarak tanımlarken radikal davranışçılar zihinsel süreçleri de davranış tanımının içine katmaktadır.

O halde makine öğrenmesi (machine learning) nedir? Aslında psikolojideki öğrenme kavramı makine öğrenmesinde de geçerlidir. Biz makinenin (makine demekle donanımı ve yazılımı kastediyoruz) bir biçimde davranışını arzu edilen yönde değiştirmesini isteriz. Yani makinenin davranışını bizim istediğimiz yönde ve istediğimiz hedefleri gerçekleştirmeye anlamında değiştirmektedir. İşte makine öğrenmesi kabaca geçmiş bilgilerden ve deneyimlerden faydalı birtakım sonuçlar (davranışlar) ortaya çıkartan algoritmalar ve yöntemler topluluğudur. Makine öğrenmesinde üç bileşen vardır: Deneyim, Görev ve Performans. Deneyim canlılarda olduğu gibi makine öğrenmesinde de en önemli öğelerdendir. Makine öğrenmesinde deneyim birtakım verilerin analiz edilmesini ve onlardan bir kestirim ya da faydalı sonuçlar çıkartılması sürecidir. Görev makinenin yapmasını istediğimiz şeydir. Görevler düşük bir deneyimle düşük bir performansla gerçekleştirilebilirler. Deneyim arttıkça görevin yerine getirilme performansı da artabilir. İşte makine öğrenmesi temelde bunu hedeflemektedir. O halde makine öğrenmesinde bir veri grubu incelenir, analiz edilir, bundan

sonuçlar çıkartılır, sonra hedeflenen görev yerine getirilmeye çalışılır. Bu görevin yerine getirilmesi de gitgide iyileştirilir. Bu süreç çeşitli algoritmalarla ve yöntemlerle değişik biçimlerde ve yaklaşımalarla yürütülmektedir.

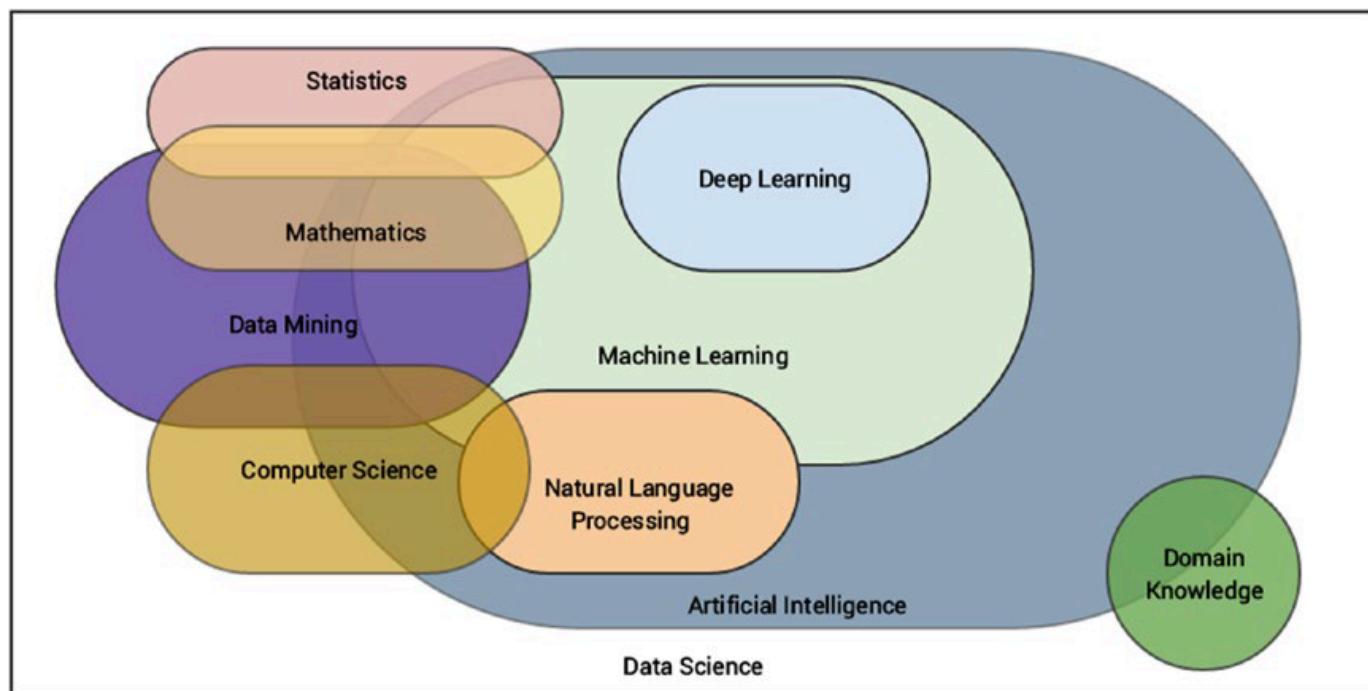
Makine öğrenmesi genel olarak üç bölümde ele alıp incelemektedir:

- 1) Denetimli Öğrenme (Supervised Learning)
- 2) Denetimsiz Öğrenme (Unsupervised Learning)
- 3) Pekiştirmeli Öğrenme (Reinforcement Learning)

Denetimli (supervised) öğrenmede makineye (yani algoritmaya) biz daha önce gerçekleşmiş olan olayları ve sonuçları girdi olarak veririz. Makine bu olaylarla sonuçlar arasında bağlantı kurar. Daha sonra biz yeni bir olayı makineye verdiğimizde onun sonucunu makineden kestirmesini isteriz. Denetimsiz öğrenmede biz makineye yalnızca olayları veririz. Makine bunların arasındaki benzerliklerden ve farklılıklardan hareketle bizim istediğimiz sonuçları çıkartmaya çalışır. Örneğin biz makineye resimler verip bunların elma mı armut mu olduğunu söyleyelim. Ve bunu çok miktarda yapalım. Sonra ona bir elma resmi verdığımızda o daha önceki deneyimlerden hareketle bunun elma olduğu sonucunu çıkartabilecektir. İşte bu denetimli öğrenmeye bir örnektir. Şimdi biz makineye elma ve armut resimlerini verelim ama bunların ne olduğunu ona söylemeyeceğiz. Ondan bu resimleri ortak özelliklerine göre iki gruba ayırmamasını isteyelim. Bu da denetimsiz öğrenmeye örnektir. Pekiştirmeli öğrenmede ise tipki edimsel koşullamada olduğu gibi hedefe yaklaşan durumlar ödüllendirilerek makinenin öğrenmesi sağlanır.

Yapay Zekanın Diğer Disiplinlerle İlgisi

Yapay zeka alanının diğer pek çok disiplinle yakın ilgisi vardır. Aşağıdaki şekilde bu ilgi betimlenmektedir.



Alıntı Notu: Görsel "Practical Machine Learning with Python (APress)" kitabından alınmıştır.

İstatistik: İstatistik temelde iki bölüme ayrılmaktadır:

- Betimsel istatistik (descriptive statistics)
- Sonuç çıkartıcı istatistik (inferential statistics)

Betimsel istatistik verilerin gruplanması, gösterilmesi ile ilgilidir. Yani betimleyici istatistik zaten var olan durumu betimlemektedir. Sonuç çıkartıcı istatistik ise kestirim yapmakla ilgilidir. Makine öğrenmesi istatistiğin kestirimsel

yöntemlerini açıkça kullanmaktadır. Örneğin regresyon analizi, kümeleme analizi, karar ağaçları, faktör analizi vs. gibi pek çok makine öğrenmesi yöntemi aslında istatistiğin bir konusu olarak ortaya çıkmıştır. Ancak bu istatistiksel yöntemler makine öğrenmesi temelinde genişletilmiş ve dinamik bir biçimde dönüştürülmüştür.

Vari Madenciliği (Data Mining): Veri madenciliği verilerin içerisindeki çeşitli faydalı bilgilerin bulunması, onların çekilerek elde edilmesi işlemleriyle ilgilenmektedir. Şüphesiz bu süreç istatistiksel birtakım bilgilerin yanı sıra yazılımsal uygulamaları da bünyesinde barındırmaktadır.

Bilgisayar Bilimleri (Computer Science): Bilgi işlem ve programlama etkinlikleriyle ilgili geniş kapsamlı bir bilim dalıdır.

İlgili Konudaki Özel Bilgiler: Şüphesiz her türlü algoritmik yöntem için bir biçimde hedeflenen konuda belli bir bilgi birikiminin var olması gereklidir. Örneğin ne kadar iyi programlama ve istatistik bilirseniz bilin görüntüsel verilerle çalışmak için bir görüntünün (resmin) nasıl bir organizasyona sahip olduğunu bilmeniz gereklidir. Ya da örneğin hiç muhasebe bilmeyen iyi bir programcının bir muhasebe programı yazabilmesini bekleyebilir miyiz?

Makine Öğrenmesi Uygulamaları İçin Kullanılan Programlama Dilleri

Veri bilimi (data science) ve makine öğrenmesi uygulamaları için pek çok dil tercih edilmektedir. Tercih konusunda ilk akla gelen dil şüphesiz Python'dur. Ancak C++, Java, C# gibi popüler programlama dilleri de bu amaçla gittikçe daha fazla kullanılır hale gelmektedir. Python programlama dili son on yıldır bir atak yaparak ve dünyanın en popüler ilk üç dili arasına girmiştir. Python dilinin özellikle veri bilimi ve makine öğrenmesi konusunda popüleritesinin neden bu kadar artmasına ilişkin görüşlerimiz şöyledir:

- Son yıllarda veri işleme ve verilerden kestirim yapma gereksiniminin gittikçe artmıştır ve Python dili de veri analizi için iyi bir araç olarak düşünülmektedir.
- Veri bilimi ve makine öğrenmesi için Python dilinden kullanılabilen pek çok kütüphane vardır. (Bu konudaki kütüphaneler diğer dillerden -şimdilik- daha fazladır.)
- Python nispeten basit bir dildir. Bu basitlik ana hatları veri analizi olan konularda uygulamacılara kolaylıklar sunmaktadır. Bu nedenle Python diğer disiplinlerden gelip de veri analizi ve makine öğrenmesi uygulaması yapmak isteyenler için nispeten daha kolay bir araç durumundadır.
- Python genel amaçlı bir programlama dili olmasının yanı sıra aynı zamanda matematiksel alana da yakın bir programlama dilidir. Yani Python'ın matematiksel alana yönelik ifade gücü (expressivity) popüler diğer programlama dillerinden daha yüksektir.
- Python dilinin çeşitli prestijli üniversitelerde "programlamaya giriş" gibi derslerde kullanılmaya başlanmış olması onun popüleritesini artırılmıştır. Python özellikle 3'lü versiyonlarla birlikte dikkate değer biçimde iyileştirilmiştir.
- Python dilinin veri analizi için diğer dillere göre daha erken yola çıktığı söylenebilir. Bu alanda algoritma geliştiren araştırmacılar algoritmalarını daha çok Python kullanarak gerçekleştirmiştir.

Peki Python dilinin veri analizi ve makine öğrenmesi konusunda hangi dezavantajları vardır? Bu dezavantajları da şöyle ifade edebiliriz:

- Python nispeten yavaş bir dildir. Bu yavaşlık büyük ölçüde Python dilinin dinamik tür sistemine sahip olmasından, Python programlarının yorumlayıcı yoluyla çalıştırılmasından ve dilin seviyesinin yüksek olmasından kaynaklanmaktadır. Her ne kadar veri analizi ve makine öğrenmesinde kullanılan kütüphaneler (numpy, pandas, scipy, keras gibi) asıl olarak C programlama dili ile yazılmış olsalar da bu C rutinlerinin Python'dan çağrılmaması ve diğer birtakım işlemler yavaşlığa yol açmaktadır.
- Python etkin (effective) bir programlama dili değildir. Dilin olanakları ince birtakım işlemlerin yapılabilmesine olanak sağlamamaktadır.

Pekiyi Python genel olarak yavaş bir dilse bu durum veri analizi ve makine öğrenmesi uygulamalarında bir sorun oluşturmaz mı? Bizim yanıtımız "bazı durumlarda" olacaktır.

İSTATİSTİĞE İLİŞKİN BAZI TEMEL BİLGİLER

Yapay zeka ve özellikle de makine öğrenmesi ile ilgili çalışmalar yapacak kişilerin belli düzeyde istatistiksel bilgilere sahip olması gerekmektedir. Şüphesiz istatistik pek çok alt alanı olan geniş bir bilim dalıdır. Bu nedenle istatistiksel konulara ilişkin pek çok ayrıntı vardır. Biz bu bölümde temel bilgiler vermekle yetineceğiz. Çeşitli ayrıntılar ilgili konuların anlatıldığı bölümde gerektiğiinde açıklanacaktır. (Örneğin "kümeleme analizi (cluster analysis)" aslında istatistikte çok uzun süredir incelenen bir konudur. Ancak son yıllarda makine öğrenmesi bağlamında konunun önemi çok daha fazla artmış ve bu bağlamda pek çok algoritmik yöntem geliştirilmiştir. Dolayısıyla örneğin kümeleme analizi çok değişkenli istatigin bir konusu olduğu halde biz bu tekniğin ayrıntılarını "denetimsiz öğrenme (unsupervised learning)" içerisinde ele alacağız.)

İstatistiksel Ölçek Türleri

İstatistikte ölçülen ya da ölçülmüş olan değerlerin sınıflarına genel olarak "ölçek (scale)" denilmektedir. Pek çok kişi ölçeklerin yalnızca sayısal olduğunu sanmaktadır. Halbuki ölçekler başka biçimlerde de karşımıza çıkabilmektedir. İstatistikte ölçekler tipik olarak şu sınıflara ayılmaktadır:

Kategorik (Nominal) Ölçekler: Bu ölçeklerde söz konusu kümenin elemanları kategorik olgulardır. Örneğin cinsiyet, renk, coğrafi bölge gibi. Bu ölçekteki ölçülen ya da ifade edilen değerlerin sayısal karşılıkları yoktur. Örneğin "kadınlarla erkekler arasında sigara içme miktarı arasında anlamlı bir fark olup olmadığını" anlamak için gerçekleştirilen bir araştırmada ölçülmesi istenen değişkenlerden "cinsiyet" kategorik (nominal) bir ölçüye ilişkindir. Benzer biçimde kişilerin renk tercihleriyle ilgili bir araştırmada renkler (siyah, beyaz, kırmızı gibi) kategorik bir ölçekle ifade edilirler.

Sırasal (Ordinal) Ölçekler: Bu ölçeklerdeki değerler de birer kategori belirtmekle birlikte bu kategoriler arasında büyülüklük-küçüklük ilişkisi söz konusudur. Örneğin eğitim durumu için kategorik değerler "ilköğretim", "lise", "üniversite" olabilir ve bunlar arasında sıra ilişkisi vardır. Bu nedenle "eğitim durumu" bir sıralı ölçek belirtibilmektedir.

Aralıklı (Interval) Ölçekler: Aralıklı ölçekler sayısal bilgi içerirler. Bu tür ölçeklerde iki puan arasındaki fark aynı miktar uzaklığı ya da yakınlığı ifade eder. Örneğin bir testte 20 puan alan 10 puan alandan beli miktarda daha iyidir. 30 puan alan da 20 puan alandan aynı miktar kadar daha iyidir. Bu tür ölçeklerde mutlak sıfır noktası yoktur. Başka bir deyişle bu tür ölçeklerde "sıfır" yokluğu ya da mevcut olmamayı belirtmemektedir. Alınan puanlar her zaman belli bir göreli orijine göre anlamlıdır. Örneğin aslında sınavlardan alınan puanlar böyle bir ölçek türündedir. Sınavdan sıfır alınabilir. Ancak bu sıfır o kişinin o konu hakkında hiçbir şey bilmediği anlamına gelmez. Yani mutlak sıfır değildir. Ya da örneğin ısı belirten "derece (celcius)" bir aralık ölçüği belirtmektedir. 50 derece ile 40 derece arasındaki ısı farkı 40 derece ile 30 derece arasındaki fark kadardır ancak sıfır dereceisinin olmadığı anlamına gelmez.

Oransal (Ratio) Ölçekler: Bu ölçekler de sayısal bilgi içerirler. Oransal ölçekler aralık ölçeklerin tüm özelliklerine sahiptirler. Ancak ek olarak oransal ölçeklerde mutlak bir sıfır noktası da vardır. Dolayısıyla puanlar arasındaki oranlar mutlak olarak anlamlıdır. Örneğin uzunluk, kütle gibi temel fiziksel özellikler oransal ölçek türlerindendir. Bir nesnenin uzunluğunun sıfır olması onun uzunluğunun olmadığı, kütlesinin sıfır olması da onun kütlesinin olmadığı anlamına gelmektedir.

Provides:	Nominal	Ordinal	Interval	Ratio
The "order" of values is known		✓	✓	✓
"Counts," aka "Frequency of Distribution"	✓	✓	✓	✓
Mode	✓	✓	✓	✓
Median		✓	✓	✓
Mean			✓	✓
Can quantify the difference between each value			✓	✓
Can add or subtract values			✓	✓
Can multiply and divide values				✓
Has "true zero"				✓

Summary of data types and scale measures

Alıntı Notu: GörSEL <https://www.mymarketresearchmethods.com/wp-content/uploads/2016/05/summary-of-data-types-and-scales.png> adresinden alınmıştır.

Aritmetik Ortalama, Medyan, Mod, Standart Sapma ve Varyans Kavramları

Betimsel istatistikte bir grup verinin ortalaması için çeşitli ortalama hesaplama yöntemleri (merkezi eğilim ölçütleri) kullanılabilmektedir. Bu yöntemlerden en önemlilerinden biri aritmetik ortalamadır. Bildiğiniz gibi aritmetik ortalama değerlerin toplamının değer sayısına bölünmesiyle elde edilmektedir:

$$\bar{X} = \frac{\sum_{i=1}^n X_i}{N} = \frac{X_1 + X_2 + X_3 + \dots + X_n}{N}$$

Aritmetik ortalama için Python'da statistics modülündeki mean fonksiyonu bulundurulmuştur. Bu fonksiyon dolaşılabilir herhangi bir nesneyi parametre olarak alabilmektedir. Örneğin:

In [20]: `import statistics`

In [21]: `a = [3, 6, 2, 8, 5]`

In [22]: `statistics.mean(a)`

Out[22]: 4.8

Ancak numpy içerisindeki mean fonksiyonu daha fazla olanaklara sahiptir. Örneğin:

In [23]: `import numpy as np`

In [24]: `a = np.array([3, 6, 2, 8, 5])`

In [25]: `np.mean(a)`

Out[25]: 4.8

Numpy'ın mean fonksiyonu iki boyutlu (ya da daha fazla boyutlu) dizilerden satır ve sütun temelinde ortalamaları da hesaplayabilmektedir. Örneğin:

```
In [36]: a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
In [37]: np.mean(a)  
Out[37]: 5.0  
  
In [38]: np.mean(a, axis=0)  
Out[38]: array([4., 5., 6.])  
  
In [39]: np.mean(a, axis=1)  
Out[39]: array([2., 5., 8.])
```

Pandas kütüphanesinde aritmetik ortalama Series sınıfının mean metoduyla hesaplanmaktadır. Örneğin:

```
In [1]: import pandas as pd  
  
In [2]: s = pd.Series([3, 6, 2, 8, 5])  
  
In [3]: s.mean()  
Out[3]: 4.8
```

Aritmetik ortalama ancak sıralı (ordinal) ya da oransal (ratio) ölçekler için kullanılabilir. Aritmetik ortalamaların üç değerlerden çok etkilenebileceğine dikkat ediniz. Bu nedenle bazı durumlarda aritmetik ortalama yerine medyan değeri de tercih edilebilmektedir. Medyan küçükten büyüğe sıraya dizilmiş değerlerin ortasında bulunan değere denilmektedir. Eleman sayısı çift ise medyan değeri ortadaki iki elemanın toplamının yarısı ile hesaplanmaktadır.

Medyan hesabı için Python standart kütüphanelerindeki median fonksiyonu kullanılmaktadır. Örneğin:

```
In [13]: a = [1, 5, 9, 2, 7]  
  
In [14]: statistics.median(a)  
Out[14]: 5  
  
In [15]: a = [1, 5, 9, 2, 7, 10]
```

Örneğin:

```
In [15]: a = [1, 5, 9, 2, 7, 10]  
  
In [16]: statistics.median(a)  
Out[16]: 6.0
```

numpy kütüphanesinde de yine aynı işlem median fonksiyonuyla yapılmaktadır. Örneğin:

```
In [22]: a = np.array([[1, 5, 9, 4], [3, 8, 4, 7], [4, 9, 3, 8]])  
  
In [23]: np.median(a)  
Out[23]: 4.5  
  
In [24]: np.median(a, axis=0)  
Out[24]: array([3., 8., 4., 7.])  
  
In [25]: np.median(a, axis=1)  
Out[25]: array([4.5, 5.5, 6. ])
```

Medyan işlemi Pandas kütüphanesinde de Series sınıfının median metodu ile de yapılabilmektedir. Örneğin:

```
In [25]: np.median(a, axis=1)  
Out[25]: array([4.5, 5.5, 6. ])  
  
In [26]: a = pd.Series([1, 5, 9, 2, 7, 10])  
  
In [27]: a.median()  
Out[27]: 6.0
```

Merkezi ölçü olarak aritmetik ortalama ile medyan arasında nasıl bir farklılık vardır? Aritmetik ortalamada tüm değerlerin işlemde etkili olduğuna ancak medyana etkili olmadığına dikkat ediniz. Dağılım içerisindeki en büyük değeri daha da büütsek medyanda bir etkisi olabilir mi? Ayrıca aritmetik ortalamanın üç değerlerden çok etkilenebildiğini ancak medyanın etkilenmediğini de belirtmek istiyoruz. Medyan işlemi için sıraya dizme gerektirdiği için algoritma ancak $O(n \log n)$ karmaşıklıkta gerçekleştirilebilir. Halbuki aritmetik ortalama $O(n)$ karmaşıklıkta hesaplanabilmektedir. Median'ın da sıralı ve oransal ölçekteki bilgilere uyaranabileceğine dikkat ediniz.

Bir dağılımin modu en çok yinelenen değeri belirtir. Mod özellikle kategorik ve sıralı ölçeklerde ortalamanın yerini tutan bir işlem olarak kullanılmaktadır. mod değeri Python standart kütüphanesindeki statistics modülü içerisindeki mode fonksiyonuyla elde edilebilmektedir. Örneğin:

```
In [8]: a = [3, 5, 7, 3, 8, 150, 4, 7, 1, 7]
```

```
In [9]: statistics.mode(a)
```

```
Out[9]: 7
```

Eğer en çok yinelenen değerler birden fazla ise mode fonksiyonu bunlardan ilk karşılaştığı değeri vermektedir. Eşit sayıda tüm yinelenen mod değerlerini elde etmek için multimode fonksiyonu kullanılmaktadır. Örneğin:

```
In [9]: statistics.mode(a)
```

```
Out[9]: 7
```

```
In [10]: a = [1, 2, 2, 1, 3, 3, 6, 8, 6]
```

```
In [11]: statistics.multimode(a)
```

```
Out[11]: [1, 2, 3, 6]
```

numpy kütüphanesinde mode işlemi için hazır bir fonksiyon bulunmamaktadır. Ancak scipy kütüphanesindeki stats modülünde mode fonksiyonu vardır. Bu fonksiyon axis parametresi de alarak mode işlemi yapar. Fonksiyon ModeResult türünden bir sınıf nesnesine geri dönmektedir. Sınıfın mode elemanı mod değerlerini, count elemanı ise onların sayılarını vermektedir. Örneğin:

```
import numpy as np
import scipy.stats

a = np.random.randint(0, 10, (10, 10))
print(a)

print()
print(mr.mode)
print(mr.count)
```

Programdam şöyle bir çıktı elde edilmiştir:

```
[[0 0 8 7 0 3 5 9 4 6]
 [4 4 9 0 3 5 0 2 7 3]
 [6 1 7 1 4 4 4 9 4 0]
 [5 4 3 6 7 5 2 7 6 0]
 [6 6 7 5 7 1 7 3 7 1]
 [1 3 5 2 4 4 5 7 7 4]
 [2 3 9 3 8 0 6 9 3 7]
 [9 0 5 5 3 1 6 5 4 1]
 [3 2 2 7 0 8 1 5 0 1]
 [2 0 4 1 1 6 7 7 3 2]]

[[2 0 5 1 0 1 5 7 4 1]]
[[2 3 2 2 2 2 3 3 3]]
```

Eksen içerisinde birden fazla aynı değer varsa mod olarak fonksiyon en küçük değeri vermektedir. Örneğin:

```

import numpy as np
import scipy.stats

a = np.random.choice(['A', 'B', 'C', 'D', 'E'], (10, 10))
mr = scipy.stats.mode(a)
print(a)
print()
print(mr.mode)
print(mr.count)

```

Şöyledir bir çıktı elde edilmiştir:

```

[['D' 'E' 'E' 'E' 'B' 'E' 'C' 'B' 'E' 'A'],
 ['B' 'E' 'B' 'E' 'B' 'D' 'A' 'E' 'B' 'B'],
 ['D' 'D' 'E' 'D' 'D' 'C' 'D' 'B' 'D' 'A'],
 ['E' 'C' 'E' 'E' 'A' 'E' 'A' 'E' 'A' 'C'],
 ['C' 'A' 'D' 'B' 'A' 'E' 'A' 'B' 'E' 'E'],
 ['D' 'D' 'B' 'D' 'C' 'E' 'A' 'A' 'D' 'C'],
 ['E' 'B' 'C' 'D' 'E' 'C' 'D' 'C' 'B' 'C'],
 ['D' 'D' 'B' 'D' 'C' 'A' 'B' 'C' 'B' 'D'],
 ['A' 'A' 'D' 'D' 'A' 'D' 'B' 'E' 'E' 'E'],
 ['D' 'E' 'B' 'A' 'E' 'E' 'C' 'E' 'B' 'B']]

[['D' 'D' 'B' 'D' 'A' 'E' 'A' 'E' 'B' 'C']]
[[5 3 4 5 3 5 4 4 4 3]]

```

Aritmetik ortalama, medyan ve mod değerlerine "merkezi eğilim ölçütleri (measures of central tendency)" denilmektedir. Bu değerler dağılımin merkezine yönelik bilgileri bize verir. Bunların dışında ayrıca değerlerin merkezden ne kadar uzaklıkta konumlandığı da önemli bir bilgidir. Bu bilgiyi veren değerlere ise "merkezi yayılım ölçütleri (measures of dispersion)" denilmektedir. Örneğin bir ülkede kişi başına düşen milli gelir ortalamasının 10000 dolar olduğunu varsayıyalım. Bu ortalama bilgisi herkesin eline yılda 10000 dolar geçtiği anlamına gelmez değil mi? Bu ülkede gelirler birbirlerine yakın da olabilir gelir dağılımında büyük bir adaletsizlik de olabilir.

Merkezi yayılım ölçütleri ortalamadan ortalama uzaklığını hesap etme temeline dayanmaktadır. Ancak bu hesabı yaparken dikkat etmek gereklidir. Örneğin eğer değerleri ortalamadan çıkartıp toplamlarının ortalamasını alırsak 0 elde ederiz. Bu da bize bir bilgi vermez. Örneğin:

```

In [27]: print(a)
[1 0 4 3 3 3 8 2 5 1 2 7 7 3 2 7 8 2 1 0]

In [28]: np.sum(np.mean(a) - a) / len(a)
Out[28]: 1.7763568394002506e-16

```

Sonuç 0'a çok yakındır. Yuvarlama hatasından dolayı 0 olamamıştır. Şimdi akılınızda mutlak değer almak gelebilir. Örneğin:

```

In [29]: a = np.random.randint(0, 10, 20)

In [30]: print(a)
[4 3 1 5 6 0 7 8 3 0 2 9 5 5 1 7 3 6 1 9]

In [31]: np.sum(np.abs(np.mean(a) - a)) / len(a)
Out[31]: 2.45

```

Gördüğünüz gibi ortalamaya yakınlık konusunda daha iyi bir ölçü elde etmiş olduk. Ancak ortalama mutlak değer yöntemi de aslında ortalamaya uzaklık hesabı için çok iyi bir yöntem değildir. Özellikle normal dağılımda bu yöntem bizi amacımızdan saptırabilir. Ayrıca ortalama mutlak değeri aynı olan birden fazla dağılım söz konusu olduğunda bunların aralarındaki farklılıklar da bu yöntemde iyi açığa çıkartılamamaktadır. İşte bu nedenle merkezi yayılım ölçüsü olarak genellikle "standart sapma (standard deviation)" kullanılmaktadır. Standart sapmada kare alma işlemi değerleri daha fazla farklılaştırmaktadır.

Bir dağılımın standart sapması değerlerin ortalamadan farklarının karelerinin ortalamasının karekökü alınarak hesaplanır. Değerlerin sayısı n olmak üzere ortalama alınırken anakütle için n değerine örneklem için $n - 1$ değerine bölüm kullanılmakadır:

$$\sigma = \sqrt{\frac{\sum(x_i - \bar{x})^2}{N}}$$

Anakütle standart sapmasının sigma simbolüyle gösterildiğine dikkat ediniz. Bir anakütleden çekilen örneklem için bölme işlemi n 'e değil $n - 1$ 'e yapılmaktadır:

$$s = \sqrt{\frac{\sum(x_i - \bar{x})^2}{(N - 1)}}$$

Örneklem için $n - 1$ değerine bölmeye "Bessel düzeltmesi (Bessel's correction)" denilmektedir. Bessel düzeltmesinin anlamı üzerinde burada durmayacağız. Bunun için Internet'te pek çok kaynak bulabilirsiniz. Düzeltmedeki $n - 1$ değerine "serbestlik derecesi (degrees of freedom)" da denilmektedir. Şimdi standart sapma için basit bir fonksiyon yazalım:

```
import numpy as np

def stdev(a):
    return np.sqrt(np.sum((a - np.mean(a)) ** 2) / len(a))
```

Tabii aslında Python statistics modülündeki pstdev fonksiyonu standart sapma hesabı yapmaktadır. Eğer örneklem standart sapması hesaplanacaksa (yeni $n - 1$ değerine bölüm isteniyorsa) bu durumda stdev fonksiyonu kullanılmalıdır. Örneğin:

```
In [33]: a = [1, 2, 3, 4, 5]
In [34]: statistics.pstdev(a)
Out[34]: 1.4142135623730951
In [35]: statistics.stdev(a)
Out[35]: 1.5811388300841898
```

numpy kütüphanesindeki std fonksiyonu axis parametresi de alarak standart sapmayı hesaplamaktadır. Fonksiyonun ddof (delta degrees of freedom) parametresi bölümün kaçça yapılacağını belirtir. Bu parametrenin default değeri 0'dır. Yani bölüm n değerine (anakütle standart sapması) yapılmaktadır. Örneğin:

```
In [40]: a = np.random.randint(0, 100, (10, 10))
In [41]: a
Out[41]:
array([[25,  1, 44, 12, 25, 70, 35, 17, 58, 91],
       [14, 97, 12, 24, 21, 83,  4, 93, 90,  4],
       [88, 20, 67, 53, 87, 29, 21, 75, 84, 30],
       [48, 38, 98, 11, 66, 38, 78, 59, 82, 32],
       [ 9, 88, 30, 68, 51, 24, 43, 80, 75, 75],
       [92, 59, 82, 46, 17, 72, 28, 11, 16, 74],
       [33, 98, 62, 86, 43, 66, 87,  3, 73, 82],
       [ 9, 89,  8, 51, 66, 95, 80, 95, 17, 57],
       [44, 68, 80, 23, 67, 46, 14, 77, 53, 13],
       [82, 85, 58, 96, 98, 39, 76, 94, 50, 56]])
In [42]: np.std(a, axis=1)
Out[42]:
array([26.64882737, 38.6        , 26.8        , 25.17141236, 25.57361922,
       28.62533843, 27.74905404, 32.95770016, 23.82120904, 20.06589146])
```

Pandas kütüphanesinde Series ve DataFrame sınıflarının da std metotları vardır. Bu metodların ddof parametreleri default 1'dir (örneklem standart sapması). Örneğin:

```
In [47]: import pandas as pd
In [48]: s = pd.Series([1, 2, 3, 4, 5])
In [49]: s.std()
Out[49]: 1.5811388300841898

In [64]: df = pd.DataFrame(np.random.randint(0, 100, (10, 10)))
In [65]: df
Out[65]:
   0   1   2   3   4   5   6   7   8   9
0  71  95  81  53  61  31  80  39  73  77
1  76   0  62  89   0  18  23  72  15  85
2  95  64  13  49   5  73  18  44  58  29
3  67  78  29  11  37  15  35  19  60  75
4  80  20  63  21  51  83  54  61  99  67
5  33  26  72  87   2  86  21  89  89  46
6  37  61  64   7  67  90  70  67  93  97
7  16  52  81  93  44  68  88  82  46  94
8  51  79   0  73  27  63   4  22  43  54
9  55  86  92  51  94  19  27  74  70  15

In [66]: df.std()
Out[66]:
0    24.301120
1    31.395152
2    31.027049
3    32.273828
4    31.011826
5    30.489342
6    29.028721
7    24.587486
8    25.868041
9    27.444692
dtype: float64
```

Standart sapmanın karesine varyans denilmektedir. Varyans işlemi statistics modülünün pvariance ve variance fonksiyonlarıyla, numpy kütüphanesinin var fonksiyonuyla, Pandas kütüphanesindeki Series ve DataFrame sınıflarının var metodlarıyla yapılmaktadır. Örneğin:

```
In [87]: a = [1, 2, 3, 4, 5]
In [88]: statistics.pvariance(a)
Out[88]: 2
In [89]: np.var(a)
Out[89]: 2.0
In [90]: s = pd.Series(a)
In [91]: s.var()
Out[91]: 2.5
In [92]: df = pd.DataFrame(np.random.randint(0, 10, (10, 10)))
In [93]: df
Out[93]:
   0   1   2   3   4   5   6   7   8   9
0  5   8   6   9   7   5   1   4   2   9
1  4   1   4   5   7   4   9   6   3   9
2  3   4   6   0   2   3   8   7   7   4
3  0   1   4   2   3   2   8   6   3   2
4  1   0   1   5   5   5   0   4   5   0
5  5   1   7   1   5   2   4   7   0   4
6  0   5   9   1   5   0   9   7   7   4
7  7   2   5   0   5   4   4   5   6   6
8  5   8   8   0   4   8   4   3   1   5
9  4   0   3   3   7   3   4   8   9   3
```

```
In [94]: df.var()
Out[94]:
0    5.600000
1    9.555556
2    5.788889
3    8.711111
4    2.888889
5    4.711111
6   10.544444
7    2.677778
8    8.677778
9    8.044444
dtype: float64
```

Olasılık Kavramı

Bir olsunun gerçekleşme beklentisini anlatan olasılığın pek çok tanımı yapılmaktadır. Bir paranın atılması durumunda tura gelme olasılığının 0.5 (%50) olduğunu biliriz. Ancak bir parayı 10 kez attığımızda 5 kere tura geleceğinin bir garantisini yoktur. O halde neden paranın atılması tura gelme olasılığı 0.5'tir? İşte olasılığın tanımlarından biri göreli sıklık tanımıdır. Paranın atılması gibi bir rassal olay n defa yinelendiğinde ve bu n sayısı artırıldığında n / tura gelme sayısı gittikçe 0.5'e yakınsayacaktır. Para 1000000 kere atıldığında yine yazı ile tura gelme sayısı aynı olmaz. Ancak tura gelme oranı gitgide 0.5'e yakınsar. O halde biz bir paranın atılması durumunda tura gelme olasılığının 0.5 olması demekle aslında bir limit durumunu kastetmiş olmaktadır.

Aşağıda 0 ile 1 arasında rastgele sayı üretecek yazı-tura atan bir fonksiyon görüyorsunuz:

```
import random

def head_tail(n):
    head = 0
    for _ in range(n):
        head += random.randint(0, 1)    # tail = 0, head = 1
    return head / n
```

Fonksiyonun parametresi paranın kaç kere atılacağını belirtmektedir. Şimdi çeşitli denemeler yapalım:

```
In [15]: head_tail(1)
Out[15]: 0.0

In [16]: head_tail(10)
Out[16]: 0.3

In [17]: head_tail(100)
Out[17]: 0.47

In [18]: head_tail(1000)
Out[18]: 0.467

In [19]: head_tail(10_000)
Out[19]: 0.5011

In [20]: head_tail(100_000)
Out[20]: 0.50393

In [21]: head_tail(1000_000)
Out[21]: 0.500036

In [22]: head_tail(10_000_000)
Out[22]: 0.4998814

In [23]: head_tail(100_000_000)
Out[23]: 0.50002317
```

Gördüğünüz gibi para atım sayısı arttıkça tura gelme olasılığı da 0.5'e yakınsamaktadır. Bu olguya istatistikte "büyük sayılar yasası (law of large numbers)" da denilmektedir.

Rassal Deneyler, Örnek Uzayı ve Rassal Olaylar

Sonucu önceden kesin olarak belirlenemeyen deneylere "rassal deney (random experiment)" denilmektedir. Örneğin bir paranın ya da bir zarın atılması birer rassal deneydir. Rassal deneyler sonucunda oluşabilecek tüm sonuçlara "örnek uzayı (sample space)" denilmektedir. Örneğin bir paranın atılması ile ilgili örnek uzay $S = \{Y, T\}$ bir zarın atılması ile ilgili örnek uzay ise $S = \{1, 2, 3, 4, 5, 6\}$ biçimindedir.

Örnek uzayın her bir alt kümesine "olay (event)" denilmektedir. Örneğin bir zarın atılması durumunda bazı olaylar şunlar olabilir:

$$\begin{aligned}E1 &= \{1, 3\} \\E2 &= [4, 5, 6] \\E3 &= \{2\}\end{aligned}$$

Örnek uzaydaki tek elemanlı oylara ise "basit olaylar (simple events)" denilmektedir. Yani basit olaylar örnek uzayının elemanlarıdır. Örneğin zarın atılmasındaki basit olaylar şunlardır:

$$\begin{aligned}E1 &= \{1\} \\E2 &= \{2\} \\E3 &= \{3\} \\E4 &= \{4\} \\E5 &= \{5\} \\E6 &= \{6\}\end{aligned}$$

E rassal olayının olasılığı $s(E) / s(S)$ biçiminde ifade edilir. Bu aksiyom büyük sayılar yasasına gører anlamlı ve aynı zamanda da sezgisel olarak da kabul edilebilir bir aksiyomdur. Örneğin bir zar atımı için E olayı $E = \{3, 4\}$ olsun. Bu durumda zar atıldığında 3 ya da 4 gelme olasılığı $P(E) = 2/6 = 1/3$ olacaktır. $P(S) = 1$ olduğuna ve $P(\text{boş küme}) = 0$ olduğuna dikkat ediniz. Dolayısıyla E olayının olasılığı $P(E) = 1 - P(E')$ biçimindedir.

Rassal Değişkenler (Random Variables)

Rassal olaylar birer küme belirttiği için matematiksel işlemlere uygun değildir. Bu nedenle kümeler yerine sayısal değerlerin kullanılabilmesi için rassal değişken kavramından faydalанılmaktadır. Bir rassal değişken örnek uzayının her bir elemanını (yani bir basit olayını) bir gerçek sayıya eşleyen bir fonksiyondur. Örneğin:

$$X: S \rightarrow \mathbb{R}$$

Burada X rassal değişkeni (yani fonksiyonu) S örnek uzayındaki her bir olayı bir gerçek sayıya eşlemektedir. Rassal değişkenler genellikle anlatımlarda sözcüklerle ifade edilseler de aslında fonksiyon belirtirler. Örneğin X rassal değişkeni için biz "bir toplulukta rastgele seçilen bir kişinin boy uzunluğu" diyebiliriz. Bu durumda aslında o toplulukta kişiler vardır. Bu kişiler örnek uzayı oluşturmaktadır. X rassal değişkeni de oradaki bireyleri onların boy uzunluklarıyla eşleyen bir fonksiyondur. Örneğin Y rassal değişkeni için de biz rastgele seçilen bir sözcüğün karakter uzunluğu" diyebiliriz. Bu durumda örnek uzay tüm sözcüklerden oluşmaktadır. Y rassal değişkeni de bu sözcükleri onların karakter uzunluklarına eşleyen bir fonksiyon durumundadır. Rassal değişkenler sayesinde artık kümeler yerine sayılarla konuşabildiğimize dikkat ediniz.

Mademki örnek uzayındaki her olayın bir olasılığı var o halde bir rassal değişkenin belli bir değeri almasının da bir olasılığı vardır. $X: S \rightarrow \mathbb{R}$ ve E de bir olay olmak üzere $P(X=N)$ olasılığının eşdeğeri şöyledir:

$$P(X = N) = P(w \in S | X(w) = N)$$

Bu eşitlik şu anlamda gelmektedir: X rassal değişkeninin N değerini alma olasılığı S kümesindeki X fonksiyonunu N değerine eşleyen elemanların oluşturduğu olayın olasılığına eşittir. Yani örneğin X rassal değişkeni iki zarın atılması durumunda üste gelen değerlerin toplamını belirtiyor olsun. Bu durumda $P(X = 7)$ aslında $P(\{(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1)\})$ ile aynı anlamdadır. Bundan sonra biz de kümelerle konuşmak yerine rassal değişkenlerle konuşmayı

tercih edeceğiz. Yani bizim için X rassal değişkeninin N değerine eşit olması aslında X fonksiyonuyla N 'e eşlenen kümeyi belirtecektir.

Kesikli ve Sürekli Rassal Değişkenler

Eğer bir rassal değişken belli bir aralıktaki tüm gerçek değerleri değil de yalnızca bazı değerleri alabiliyorsa bu tür rassal değişkenlere kesikli (discrete) rassal değişkenler denilmektedir. Örneğin iki zar atıldığında üstte gelen sayıların toplamını belirten X rassal değişkeni $[2, 12]$ aralığında yalnızca belirli değerleri alabilmektedir, bu rassal değişken $[2, 12]$ aralığında tüm gerçek sayı değerlerini alamamaktadır.

Eğer rassal değişken belli bir aralıktaki tüm gerçek sayı değerlerinden oluşabiliyorsa bu tür rassal değişkenlere de "sürekli (contiguous)" rassal değişkenler denilmektedir. Y rassal değişkeni bir topluluktan rastgele seçilen bir kişinin kilosunu belirtiyor olsun. Bu Y rassal değişkeni sürekli bir rassal değişkendir. Çünkü teorik olarak belli aralıktaki tüm gerçek sayı değerlerini alabilmektedir. (Yani örneğin kişinin kilosu 76.234567 olabileceği gibi 83.2323728765 de olabilir.)

Kesikli rassal değişkenlerin olasılık değerlerinin hesaplanması nokta temelli olarak ve tamsayı işlemleriyle yapılmaktadır. Ancak sürekli rassal değişkenlerin olasılıklarının hesaplanması ancak aralık temelli ve gerçek sayılar kullanılarak yapılabilir ki bu aralık temelli hesaplama da akla integral hesabı getirmektedir. Örneğin topluluktan rastgele seçilen kişinin kilosunu belirten K rassal değişkeninin 72 olma olasılığı aslında sıfırdır. Çünkü kişinin kilosu sonsuz sayıda değerden biri olabilir. Oysa 72 sayısı gerçek sayı doğrusunda yalnızca tek bir nokta belirtmektedir (sayı / sonsuz'un 0 olduğunu dikkat ediniz. Örneğin kişinin kilosu 72.000000000000001 olabilir, 71.999999999999999 olabilir ama tam 72 olmayabilir.) O halde sürekli rassal değişkenlerin olasılıklarını biz ancak aralık temelli olarak ve bir integral hesapla bulabiliyoruz. Bu örneğimizde K rassal değişkeninin 72 olma olasılığı 0'dır, ancak 71 ile 73 arasında olma olasılığı sıfır değildir. Tabii integral hesap için bir fonksiyona gereksinimiz olacaktır. Integral hesap yapmakta kullandığımız bu tür fonksiyonlara olasılık yoğunluk fonksiyonları (probability density function) denilmektedir.

Sürekli Rassal Değişkenlerin Olasılık Yoğunluk Fonksiyonları

Sürekli rassal değişkenler için olasılık yoğunluk fonksiyonları (probability density functions) sürekli rassal değişkenlerin belli aralıktaki olasılıklarını integral hesabı ile bulmak için kullanılan fonksiyonlardır. Aşağıdaki fonksiyonu inceleyiniz:

$$f_X(x) = \lim_{\Delta x \rightarrow 0} \frac{P(x \leq X \leq x + \Delta x)}{\Delta x}$$

Burada f fonksiyonu x ve $x + \Delta x$ arasında değer alabilen ve her noktada türevlenebilen bir fonksiyon olsun. Bu f fonksiyonu aşağıdaki özellikleri sağlıyorsa ona olasılık yoğunluk fonksiyonu denilmektedir:

$$1) f_X(x) \geq 0$$

$$2) \int_{-\infty}^{\infty} f_X(x) dx = 1$$

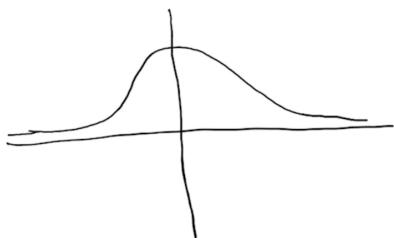
$$3) P(x_1 < X \leq x_2) = \int_{x_1}^{x_2} f_X(x) dx = F_X(x_2) - F_X(x_1)$$

$$4) F_X(x) = P(X \leq x) = \int_{-\infty}^x f_X(s) ds$$

Alıntı Notu: Bu görsel avs.omu.edu.tr adresinden elde edilmiştir.

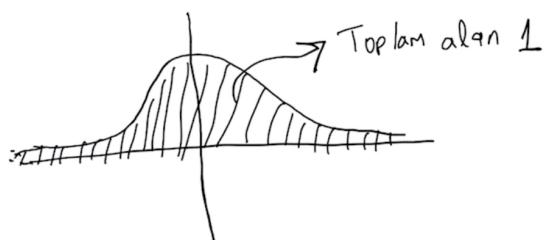
Birinci özellik olasılık yoğunluk fonksiyonun her zaman 0 ya da 0'dan büyük bir değer vermesi gerektiğini belirtir. Başka bir deyişle bu fonksiyonların grafikleri X ekseninin üzerinde kalmaktadır. İkinci özellik eksi sonsudan artı sonsuza kadar olasılık yoğunluk fonksiyonun eğri altında kalan alanının 1 olması gerektiğini belirtmektedir. Hiçbir olasılığın 1'den büyük ve 0'dan küçük olamayacağına dikkat ediniz. Üçüncü özellik bir rassal değişkenin bellili bir aralıktaki

olasılığının, olasılık yoğunluk fonksiyonun o aralıktaki integrali ile hesaplanabileceğini belirtmektedir. Yani başka bir deyişle bir rassal değişkenin belli bir aralıkta olma olasılığı o rassal değişkene ilişkin olasılık yoğunluk fonksiyonunun o aralıktaki eğri altında kalan alanına (intergraline) eşittir. Dördüncü özellik X rassal değişkeninin x gibi bir değerden küçük olma olasılığının o rassal değişkenin olasılık yoğunluk fonksiyonun x değeriin solundaki eğri Itında kalan alanına eşit olduğunu belirtmektedir. Aşağıda bir olasılık yoğunluk fonksiyonun bu özellikler bakımından değerlendirilmesini görüyorsunuz:

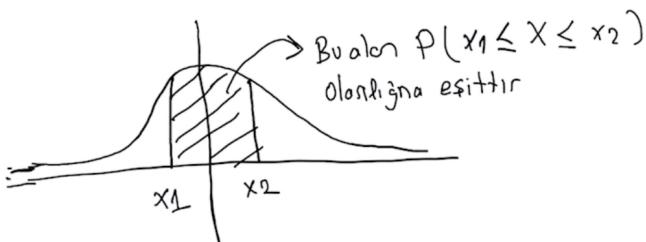


Budafaki f fonksiyonunun X ekseninin üzerinde kaldığını görüyorsunuz (1. Özellik).

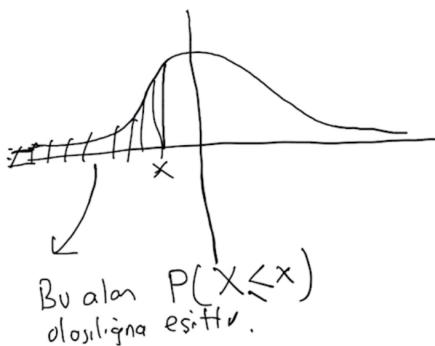
Eğri altında kalan toplam alan 1'dir (2. Özellik):



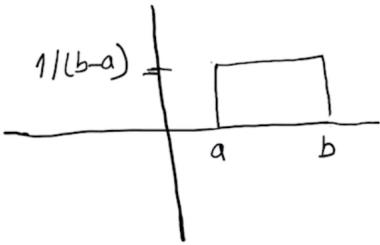
X rassal değişkeninin x_1 ile x_2 arasında olma olasılığı $X=x_1$ ve $X=x_2$ doğruları ile belirlenen alana eşittir (3. Özellik):



X rassal değişkenin x'ten küçük olma olasılığı X =x doğrusunun solundaki tüm toplam alan kadardır (4. Özellik):



Aşağıdaki fonksiyonuna bakınız:



Burada ilgili fonksiyon bir olasılık yoğunluk fonksiyonudur. Toplam eğri altında kalan alanın 1 olduğuna dikkat ediniz. Bu durumda c değeri a ve b arasında olmak üzere $P(X \leq c)$ olasılığı $(c - a) / (b - a)$ biçimindedir.

Çok Karşılaşılan Bazı Sürekli Dağılımlar

Yukarıda da belirttiğimiz gibi sürekli rassal değişkenlerle olasılık hesabı yapabilmemiz için o rassal değişkene ilişkin bir olasılık yoğunluk fonksiyonunu biliyor olmamız gereklidir. Örneğin X rassal değişkenimiz şöyle olsun:

X : Rastgele seçilen kişinin kilosu

Burada aslında örnek uzayı tüm insanlardır. (Kurs notlarının yazıldığı sırada dünyada 8 milyar civarı insan olduğu tahmin edilmektedir.) O halde X rassal değişkeni tüm insanları tek tek onların kilolarına eşleyen bir fonksiyondur. Burada kilonun sürekli bir rassal değişken olduğuna dikkat ediniz. Şimdi bizim rastgele seçilen bir kişinin 60 ile 80 arasında bir kiloya sahip olma olasılığını bulmamız için bu rassal değişkene ilişkin olasılık yoğunluk fonksiyonunu biliyor olmamız gereklidir. İşte her ne kadar sonsuz sayıda olasılık yoğunluk fonksiyonu söz konusu olabilirse de belli olguların olasılık yoğunluk fonksiyonlarının belli bir kalıba uygun olduğu görülmüştür. Genellikle uygulamalarda anakütlenin olasılık yoğunluk fonksiyonunun (buna anakütle dağılımı da denilmektedir) bu kalıplardan biri içerisinde girdiği varsayılar. Örneğin doğada pek çok olguya ilişkin sürekli rassal değişkenlerin "normal dağılım" denilen dağılıma uydugu ve bunların olasılık yoğunluk fonksiyonlarının Gauss fonksiyonuna benzettiği görülmektedir.

Peki bir rassal değişkenin olasılık yoğunluk fonksiyonunun ne olacağına nasıl karar verilmektedir? Bunun en pratik yöntemlerinden biri histogram çizip eğrinin neye benzediğine bakmak olabilir. Eğer eğri bildiğimiz bir dağılıma benzeyorsa biz elimizdeki bilgilerle onun olasılık yoğunluk fonksiyonunu kolaylıkla belirleyebiliriz. Peki eğer elimizdeki histogram bildiğimiz hiçbir dağılımın şecline benzemiyorsa bu durumda ne yapabilirim? İşte bu durumda birikimli dağılım fonksiyonundan hareketle rassal değişkenin olasılık yoğunluk fonksiyonu bulunabilemektedir. Bunun için şu makaleyi inceleyebilirsiniz: <https://online.stat.psu.edu/stat414/lesson/22/22.1>

Yukarıda da belirttiğimiz gibi sürekli bir rassal değişkenin olasılık yoğunluk fonksiyonuna kısaca o rassal değişkenin dağılımı da denilmektedir. İşte biz de bu bölümde doğada doğrudan ya da dolaylı olarak çok karşılaştığımız bazı olasılık dağılımlarını (yani olasılık yoğunluk fonksiyonlarını) açıklayacağız.

Normal Dağılım

Süphesiz normal dağılım doğada en çok karşılaşılan sürekli dağılımdır. Bu dağılıma Gauss dağılımı da denilmektedir. Gerçekten boy, kilo, zeka gibi pek çok özellik normal dağılıma eğilimindedir. Normal dağılımın doğada neden bu kadar çok karşılaşıldığının önemli bir nedeni de merkezi limit teoremdir (central limit theorem). Bu teoreme göre bir ana kütleden çekilen örneklem ortalamaları normal dağılıma eğilimindedir.

Normal dağılıma ilişkin olasılık yoğunluk fonksiyonu şöyledir:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Fonksiyonda μ ve σ biçiminde iki parametrenin olduğunu görüyoruz. μ dağılımın ortalamasını, σ ise standart sapmasını belirtmektedir. Gauss fonksiyonundaki μ eğrinin orta noktasının X eksenindeki yeri üzerine, σ ise eğrinin genişliği üzerinde etkili olur. $\mu = 0$ ve $\sigma = 1$ olan normal dağılıma standart normal dağılım denilmektedir. μ ve σ değerleri farklı olan normal dağılımlar standart normal dağılımlara dönüştürülebilmektedir. Dolayısıyla klasik istatistikten eğer bir dağılımin normal olduğu sonucuna varılmışa oradaki hesaplar onun standart normal dağılıma dönüştürülmesiyle gerçekleştirilmektedir. Standart normal dağılıma ilişkin çeşitli değerleri bazlıdan tablolar hazırlanmıştır.

Şimdi Gauss eğrisini çizdireن bir fonksiyon yazalım:

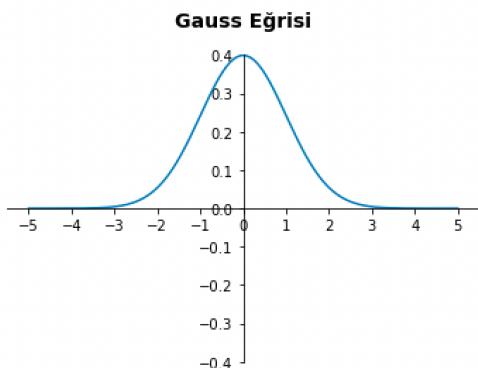
```
import numpy as np
import matplotlib.pyplot as plt

def draw_gauss(mu = 0, sigma = 1):
    x = np.linspace(-5, 5, 1000)
    y = 1 / np.sqrt(2 * np.pi * sigma ** 2) * np.e ** (-((x - mu) ** 2) / (2 * sigma ** 2))

    axis = plt.gca()
    axis.set_title('Gauss Eğrisi', fontsize=14, fontweight='bold', pad=20)
    axis.set_xlim([-0.4, 0.4])
    axis.set_xticks(range(-5, 6))
    axis.spines['left'].set_position('center')
    axis.spines['bottom'].set_position('center')
    axis.spines['top'].set_color(None)
    axis.spines['right'].set_color(None)

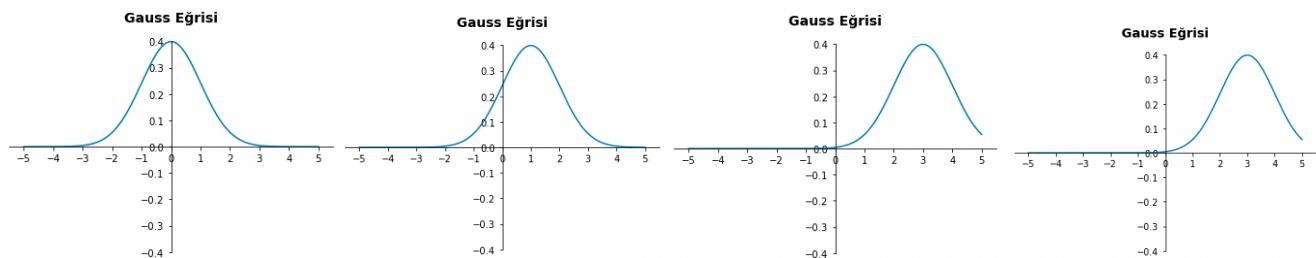
    plt.plot(x, y)
    plt.show()

draw_gauss()
```



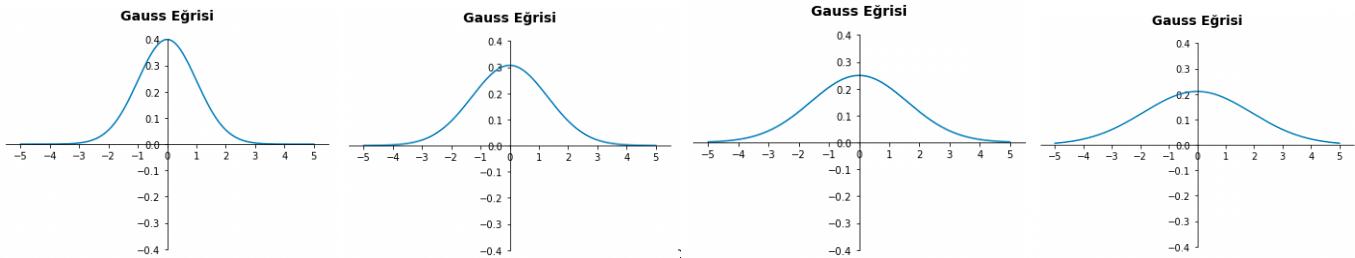
Şimdi çeşitli mü değerleri için birden fazla grafik çizelim:

```
draw_gauss(mu=0)
draw_gauss(mu=1)
draw_gauss(mu=2)
draw_gauss(mu=3)
```



Şimdi de mu değerini sabit tutup sigma (standart sapma) değerini değiştirelim:

```
draw_gauss(sigma=1)
draw_gauss(sigma=1.3)
draw_gauss(sigma=1.6)
draw_gauss(sigma=1.9)
```



Gördüğünüz gibi dağılımin standart sapma değeri değiştirildiğinde Gauss eğrisi şişmanlamaktadır. Standart sapmanın ortalamadan uzaklığı ilişkin bir değer belirttiğine dikkat ediniz. Yukarıdaki draw_gauss fonksiyonunu daha parametrik bir biçimde de yazabiliriz:

```
import numpy as np
import matplotlib.pyplot as plt

def draw_gauss(mu = 0, sigma = 1, axispos = None):
    if axispos == None:
        axispos = mu
    x = np.linspace(axispos - 5 * sigma, axispos + 5 * sigma , 1000)
    y = 1 / np.sqrt(2 * np.pi * sigma ** 2) * np.e ** (- (x - mu) ** 2 / (2 * sigma ** 2))

    axis = plt.gca()
    axis.set_title('Gauss Eğrisi', fontsize=14, fontweight='bold', pad=20)
    axis.set_yticks([-0.4 / sigma, 0.4 / sigma])
    axis.set_xticks(np.arange(int(mu - 5 * sigma), int(mu + 5 * sigma + sigma), sigma))
    axis.spines['left'].set_position('center')
    axis.spines['bottom'].set_position('center')
    axis.spines['top'].set_color(None)
    axis.spines['right'].set_color(None)

    plt.plot(x, y)
    plt.show()

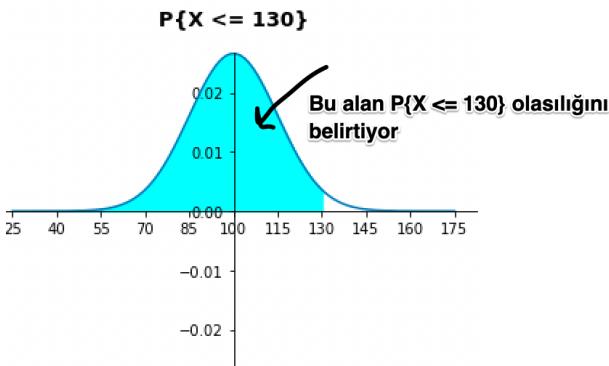
draw_gauss(mu=100, sigma=2, axispos=100)
```

Fonksiyonun axispos parametresi Y eksenin çizileceği yeri belirtmektedir.

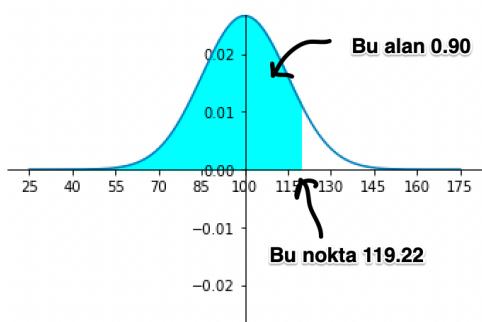
Python programcısı olarak normal dağılımla ilgili dört önemli işlemi yapabiliyor olmamız gereklidir. Bunlardan ilki belli bir X değeri için (standart normal dağılımdaki X değerlerine Z değerleri de denilmektedir) eğri altında kalan alanı bulmak. Belli bir değerden küçük olan eğri altında kalan alanı veren fonksiyona kümülatif dağılım fonksiyonu (cumulative distribution function) denilmektedir. Kümülatif dağılım fonksiyonunu şöyle ifade edebiliriz:

$$F(x) = P\{X \leq x\}$$

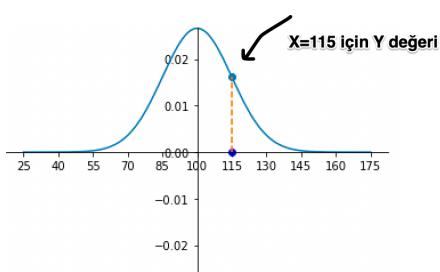
Örneğin ortalaması 100, standart sapması 15 olan bir normal dağılımda X'in 130'dan küçük olma olasılığı aslında F(130) değeridir.



İkinci bunun tersi olan işlemidir. Yani kümülatif dağılım değerinin X değerine dönüştürülmesi. Örneğin ortalaması 100 standart sapması 15 olan bir normal dağılımda eğri altında kalan alanın 0.90 olduğu X değerinin bulunması istenebilir.



Üçüncü önemli işlem belli bir X değeri için Gauss fonksiyonundaki Y değerinin elde edilmesidir. Örneğin ortalaması 100 standart sapması 15 olan Gauss fonksiyonunda $X = 115$ 'e karşılık gelen Y değeri bulunmak istenebilir.



Nihayet son önemli işlem normal dağılıma uygun rastgele sayıların elde edilmesidir. Örneğin ortalaması 100, standart sapması 15 olan normal dağılıma ilişkin 1000 tane rastgele X değeri elde etmek isteyebiliriz.

Normal dağılıma ilişkin işlemleri Python standart kütüphanesindeki `NormalDist` sınıfı ile yapılmaktadır. Bu sınıf nesnesi yaratılırken sınıfın `__init__` metodunda dağılımın ortalaması ve standart sapması girilir:

```
class statistics.NormalDist(mu=0.0, sigma=1.0)
```

Sınıfın `cdf` (cumulative distribution function) birikimli olasılık değerini bize verir. Örneğin ortalaması 100 standart sapması 15 olan normal dağılımda $P\{X \leq 130\}$ değeri şöyle edilir:

```
In [16]: import statistics
In [17]: nd = statistics.NormalDist(100, 15)
In [18]: nd.cdf(130)
Out[18]: 0.9772498680518208
```

Bu işlemin tersi yani birikimli olasılığı belli değere eşit olan X değeri `inv_cdf` metoduyla elde edilmektedir. Örneğin aynı dağılım için 0.95 birikimli olasılığa karşı gelen X değerini bulalım:

```
In [19]: nd.inv_cdf(0.95)
Out[19]: 124.67280440427207
```

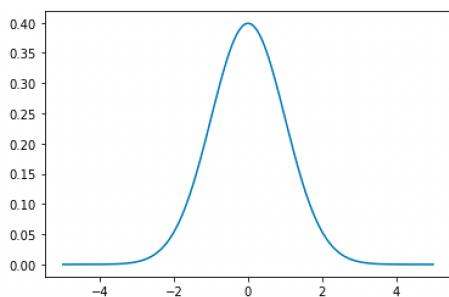
Normal dağılımin olasılık yoğunluk fonksiyonu sınıfının pdf metoduya temsil edilmiştir. Örneğin $X = 100$ için Y değeri şöyle elde edilebilir:

```
In [20]: nd.pdf(100)
Out[20]: 0.02659615202676218
```

Bu metot yoluyla Gauss eğrisini şöyle çizdirebiliriz:

```
import statistics
import matplotlib.pyplot as plt

nd = statistics.NormalDist()
x = [i * 0.001 for i in range(-5000, 5000)]
y = [nd.pdf(i) for i in x]
plt.plot(x, y)
```



Normal dağılıma ilişkin X rassal değerleri NormalDist sınıfının samples metodu ile elde edilebilir. Fonksiyon argüman olarak kaç rassal sayı üretileceğini alır rassal sayılarından oluşan float bir listeye geri döner. Örneğin ortalaması 0, standart sapması 1 olan normal dağılıma ilişkin 10 tane X değeri elde edelim:

```
In [25]: nd.samples(10)
Out[25]:
[-0.4447316124433328,
-0.7224285313191506,
-0.1929356446968533,
0.7350544803051189,
-1.796611072264187,
0.015753067776417353,
0.13667455475444215,
1.0778962661930618,
-1.5975623709468105,
1.4683966292696242]
```

Normal dağılıma ilişkin rassal sayıların elde edilmesi için çeşitli yöntemler önerilmektedir. Ancak en basit yöntem 0 ile 1 arasında rasgele sayı üretip inv_cdf metodu ile buna karşı gelen X değerini elde etmektedir. Örneğin:

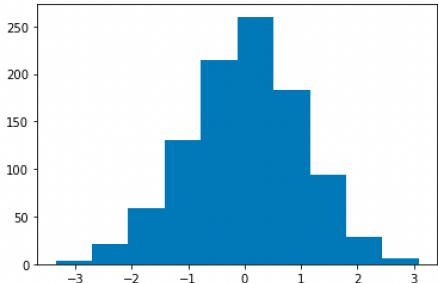
```
In [26]: import random
```

```
In [27]: nd.inv_cdf(random.random())
Out[27]: -0.4817643279327754
```

Şimdi de normal dağılıma ilişkin rassal sayılar elde edip histogramını çizelim:

```
import statistics
import matplotlib.pyplot as plt

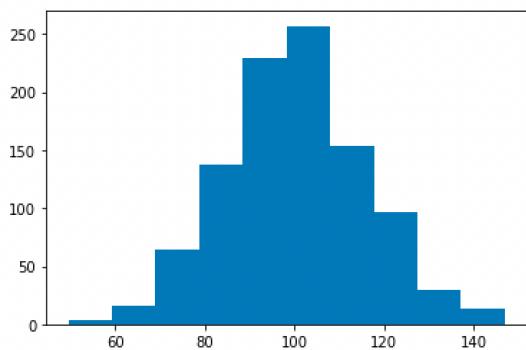
nd = statistics.NormalDist()
vals = nd.samples(1000)
plt.hist(vals)
```



Python standart kütüphanesindeki random modülünde bulunan gauss fonksiyonu da normal dağılmış rassal sayı üretmektedir. Örneğin:

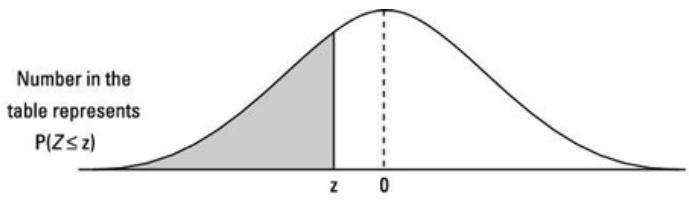
```
import random
import matplotlib.pyplot as plt

vals = [random.gauss(100, 15) for _ in range(1000)]
plt.hist(vals)
```



Aynı modülde thread güvenli olan ancak biraz daha yavaş çalışma potansiyelide olan randomvariate isimli bir fonksiyon da bulunmaktadır.

Bilgisayarların istatistikte bu kadar yoğun kullanılmadığı zamanlarda bu tür işlemler elle yapılyordu. Bunun için ortalaması 0 olan standart sapması 1 olan standart normal dağılım için Z tabloları oluşturulmuştur. Bu tablolar belli bir Z değeri için (standart normal dağılımda X değeri yerine Z değeri denildiğini anımsayınız) birikimli olasılıkları göstermektedir. Aşağıda örnek bir Z tablosu görüyorsunuz:



z	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
-3.6	.0002	.0002	.0001	.0001	.0001	.0001	.0001	.0001	.0001	.0001
-3.5	.0002	.0002	.0002	.0002	.0002	.0002	.0002	.0002	.0002	.0002
-3.4	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0002
-3.3	.0005	.0005	.0005	.0004	.0004	.0004	.0004	.0004	.0004	.0003
-3.2	.0007	.0007	.0006	.0006	.0006	.0006	.0006	.0005	.0005	.0005
-3.1	.0010	.0009	.0009	.0009	.0008	.0008	.0008	.0008	.0007	.0007
-3.0	.0013	.0013	.0013	.0012	.0012	.0011	.0011	.0011	.0010	.0010
-2.9	.0019	.0018	.0018	.0017	.0016	.0016	.0015	.0015	.0014	.0014
-2.8	.0026	.0025	.0024	.0023	.0023	.0022	.0021	.0021	.0020	.0019
-2.7	.0035	.0034	.0033	.0032	.0031	.0030	.0029	.0028	.0027	.0026
-2.6	.0047	.0045	.0044	.0043	.0041	.0040	.0039	.0038	.0037	.0036
-2.5	.0062	.0060	.0059	.0057	.0055	.0054	.0052	.0051	.0049	.0048
-2.4	.0082	.0080	.0078	.0075	.0073	.0071	.0069	.0068	.0066	.0064
-2.3	.0107	.0104	.0102	.0099	.0096	.0094	.0091	.0089	.0087	.0084
-2.2	.0139	.0136	.0132	.0129	.0125	.0122	.0119	.0116	.0113	.0110
-2.1	.0179	.0174	.0170	.0166	.0162	.0158	.0154	.0150	.0146	.0143
-2.0	.0228	.0222	.0217	.0212	.0207	.0202	.0197	.0192	.0188	.0183
-1.9	.0287	.0281	.0274	.0268	.0262	.0256	.0250	.0244	.0239	.0233
-1.8	.0359	.0351	.0344	.0336	.0329	.0322	.0314	.0307	.0301	.0294
-1.7	.0446	.0436	.0427	.0418	.0409	.0401	.0392	.0384	.0375	.0367
-1.6	.0548	.0537	.0526	.0516	.0505	.0495	.0485	.0475	.0465	.0455
-1.5	.0668	.0655	.0643	.0630	.0618	.0606	.0594	.0582	.0571	.0559
-1.4	.0808	.0793	.0778	.0764	.0749	.0735	.0721	.0708	.0694	.0681
-1.3	.0968	.0951	.0934	.0918	.0901	.0885	.0869	.0853	.0838	.0823
-1.2	.1151	.1131	.1112	.1093	.1075	.1056	.1038	.1020	.1003	.0985
-1.1	.1357	.1335	.1314	.1292	.1271	.1251	.1230	.1210	.1190	.1170
-1.0	.1587	.1562	.1539	.1515	.1492	.1469	.1446	.1423	.1401	.1379
-0.9	.1841	.1814	.1788	.1762	.1736	.1711	.1685	.1660	.1635	.1611
-0.8	.2119	.2090	.2061	.2033	.2005	.1977	.1949	.1922	.1894	.1867
-0.7	.2420	.2389	.2358	.2327	.2296	.2266	.2236	.2206	.2177	.2148
-0.6	.2743	.2709	.2676	.2643	.2611	.2578	.2546	.2514	.2483	.2451
-0.5	.3085	.3050	.3015	.2981	.2946	.2912	.2877	.2843	.2810	.2776
-0.4	.3446	.3409	.3372	.3336	.3300	.3264	.3228	.3192	.3156	.3121
-0.3	.3821	.3783	.3745	.3707	.3669	.3632	.3594	.3557	.3520	.3483
-0.2	.4207	.4168	.4129	.4090	.4052	.4013	.3974	.3936	.3897	.3859
-0.1	.4602	.4562	.4522	.4483	.4443	.4404	.4364	.4325	.4286	.4247
-0.0	.5000	.4960	.4920	.4880	.4840	.4801	.4761	.4721	.4681	.4641

Alıntı Notu: Görüel <https://www.dummies.com/education/math/statistics/how-to-use-the-z-table/> adresinden alınmıştır.

Burada ilgili satır ile sütunun birleşimi Z değerini, onların kesimlerinde bulunan değer ise birikimli olasılık değerini belirtmektedir. Tabloda yalnızca normal dağılımin sol yarımlı kısmının bulunduğuuna dikkat ediniz. Eğrinin iki yarısı simetrik olduğuna göre sağ yarısı için birikimli değerler de kolaylıkla elde edilebilmektedir. Yukarıdaki Z tablosunun benzerini çıkaran basit bir Python programını söyle yazabiliriz:

```
import statistics

def disp_ztable():
    nd = statistics.NormalDist()

    print('-' * 76)
    print(' z' + 6 * ' ', end='')
    for i in range(10):
        f = i * 0.01
```

```

    print(f'{f:<7.2f}', end=' ')
print()
print('-' * 76)

z = -3.6
while z <= 0.05:
    print(f'{-0.0:<8.1f}' if z > 0 else f'{z:<8.1f}', end='')

    for i in range(10):
        f = i * 0.01
        cd = nd.cdf(z - f)
        print(f'{cd:<8.4f}'[1:], end=' ')
    print()
    z += 0.1

```

z	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
-3.6	.0002	.0002	.0001	.0001	.0001	.0001	.0001	.0001	.0001	.0001
-3.5	.0002	.0002	.0002	.0002	.0002	.0002	.0002	.0002	.0002	.0002
-3.4	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0003	.0002
-3.3	.0005	.0005	.0005	.0004	.0004	.0004	.0004	.0004	.0004	.0003
-3.2	.0007	.0007	.0006	.0006	.0006	.0006	.0006	.0005	.0005	.0005
-3.1	.0010	.0009	.0009	.0008	.0008	.0008	.0008	.0007	.0007	.0007
-3.0	.0013	.0013	.0012	.0012	.0011	.0011	.0011	.0010	.0010	.0010
-2.9	.0019	.0018	.0018	.0017	.0016	.0016	.0015	.0015	.0014	.0014
-2.8	.0026	.0025	.0024	.0023	.0023	.0022	.0021	.0021	.0020	.0019
-2.7	.0035	.0034	.0033	.0032	.0031	.0030	.0029	.0028	.0027	.0026
-2.6	.0047	.0045	.0044	.0043	.0041	.0040	.0039	.0038	.0037	.0036
-2.5	.0062	.0060	.0059	.0057	.0055	.0054	.0052	.0051	.0049	.0048
-2.4	.0082	.0080	.0078	.0075	.0073	.0071	.0069	.0068	.0066	.0064
-2.3	.0107	.0104	.0102	.0099	.0096	.0094	.0091	.0089	.0087	.0084
-2.2	.0139	.0136	.0132	.0129	.0125	.0122	.0119	.0116	.0113	.0110
-2.1	.0179	.0174	.0170	.0166	.0162	.0158	.0154	.0150	.0146	.0143
-2.0	.0228	.0222	.0217	.0212	.0207	.0202	.0197	.0192	.0188	.0183
-1.9	.0287	.0281	.0274	.0268	.0262	.0256	.0250	.0244	.0239	.0233
-1.8	.0359	.0351	.0344	.0336	.0329	.0322	.0314	.0307	.0301	.0294
-1.7	.0446	.0436	.0427	.0418	.0409	.0401	.0392	.0384	.0375	.0367
-1.6	.0548	.0537	.0526	.0516	.0505	.0495	.0485	.0475	.0465	.0455
-1.5	.0668	.0655	.0643	.0630	.0618	.0606	.0594	.0582	.0571	.0559
-1.4	.0808	.0793	.0778	.0764	.0749	.0735	.0721	.0708	.0694	.0681
-1.3	.0968	.0951	.0934	.0918	.0901	.0885	.0869	.0853	.0838	.0823
-1.2	.1151	.1131	.1112	.1093	.1075	.1056	.1038	.1020	.1003	.0985
-1.1	.1357	.1335	.1314	.1292	.1271	.1251	.1230	.1210	.1190	.1170
-1.0	.1587	.1562	.1539	.1515	.1492	.1469	.1446	.1423	.1401	.1379
-0.9	.1841	.1814	.1788	.1762	.1736	.1711	.1685	.1660	.1635	.1611
-0.8	.2119	.2090	.2061	.2033	.2005	.1977	.1949	.1922	.1894	.1867
-0.7	.2420	.2389	.2358	.2327	.2296	.2266	.2236	.2206	.2177	.2148
-0.6	.2743	.2709	.2676	.2643	.2611	.2578	.2546	.2514	.2483	.2451
-0.5	.3085	.3050	.3015	.2981	.2946	.2912	.2877	.2843	.2810	.2776
-0.4	.3446	.3409	.3372	.3336	.3300	.3264	.3228	.3192	.3156	.3121
-0.3	.3821	.3783	.3745	.3707	.3669	.3632	.3594	.3557	.3520	.3483
-0.2	.4207	.4168	.4129	.4090	.4052	.4013	.3974	.3936	.3897	.3859
-0.1	.4602	.4562	.4522	.4483	.4443	.4404	.4364	.4325	.4286	.4247
-0.0	.5000	.4960	.4920	.4880	.4840	.4801	.4761	.4721	.4681	.4641

Bu tür kodlarda yuvarlama hatalarına dikkat ediniz. Python'da yuvarlama hatalarına maruz kalmadan noktalı sayılar üzerinde işlem yapmak için tasarılanmış decimal isimli standart bir modülün olduğunu anımsatmak istiyoruz. Ancak bu decimal türü numpy ya da Pandas kütüphanelerinde kullanılamamaktadır.

Belli bir ortalama ve standart sapmaya ilişkin normal dağılımdaki X değerini standart normal dağılımdaki Z değerine dönüştürmek için (yani ortalaması 0, standart sapması 1 olan normal dağılımdaki X değerine dönüştürmek için) aşağıdaki işlem uygulanır:

$$Z = \frac{X - \mu}{\sigma}$$

Örneğin ortalaması 100 standart sapması 15 olan bir normal dağılımdaki $X = 125$ değerinin standart normal dağılımdaki karşılığı şöyle hesaplanmaktadır:

$$Z = \frac{125 - 100}{15} = 1.666$$

Python 3.9 ile birlikte NormalDist sınıfına bu dönüşümü yapan zscore isimli bir metot da eklenmiştir.

Normal dağılıma ilişkin işlemler scipy kütüphanesindeki `scipy.stats.norm` isimli "singleton" sınıf nesnesi ile de yapılabilmektedir. Scipy kütüphanesi numpy kütüphanesi kullanılarak gerçekleştirildiği için değerler üzerinde tek tek değil vektörel işlemler yapabilmektedir. `norm` nesnesinin ilişkin olduğu sınıfın `cdf` isimli metodu birikimli olasılık değerini elde etmekte kullanılmaktadır. Metodun birinci parametresi birikimli olasılığı hesaplanacak değerlerin bulunduğu dolaşılabilir nesneyi alır. İkinci ve üçüncü parametreler normal dağılımin ortalama ve standart sapmasını belirtmektedir. Örneğin:

```
from scipy.stats import norm

result = norm.cdf([100, 130, 120], 100, 15)
print(result)
```

Şöyle bir çıktı elde edilmiştir:

```
[0.5 0.97724987 0.90878878]
```

Bu işlemin tersi yani birikimli olasılığa karşı gelen X değeri ilgili sınıfın `ppf` (percent point function) metoduyla elde edilmektedir:

```
from scipy.stats import norm

result = norm.ppf([0.5, 0.3, 0.05], 100, 15)
print(result)
```

Şöyle bir çıktı elde edilmiştir:

```
[100. 92.13399231 75.3271956 ]
```

Belli X değerleri için Gauss eğrisindeki Y değerlerinin elde edilmesi ilgili sınıfın `pdf` isimli metoduyla yapılmaktadır. Örneğin:

```
from scipy.stats import norm

result = norm.pdf([100, 120, 130], 100, 15)
print(result)
```

Bu işlemenin aşağıdaki gibi bir çıktı elde edilmiştir:

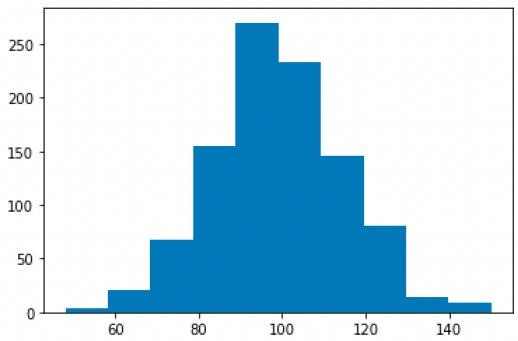
```
[0.02659615 0.010934 0.0035994 ]
```

Normal dağılıma ilişkin rastgele sayı elde edebilmek için ise ilgili sınıfın `rvs` metodu kullanılmaktadır. Örneğin:

```
from scipy.stats import norm
import matplotlib.pyplot as plt

result = norm.rvs(100, 15, 1000)
plt.hist(result)
```

Elde edilen histogram şöyleledir:



rvs fonksiyonun üçüncü parametresi bir demet olarak da girilebilmektedir. Bu durumda çok boyutlu normal dağılmış rassal sayılar da üretilmekteydi. Yine benzer biçimde numpy.random modülündeki normal isimli fonksiyon da normal dağılmış rastgele sayılar üretmekteydi.

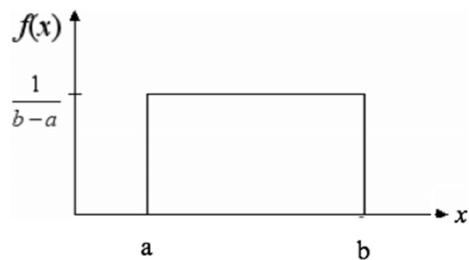
Sürekli Düzgün Dağılım (Continuous Uniform Distribution)

Düzgün dağılım herkesin aşina olduğu bir dağılımdır. Düzgün dağılımin olasılık yoğunluk fonksiyonu şöyledir:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } a \leq x \leq b, \\ 0 & \text{for } x < a \text{ or } x > b \end{cases}$$

Alıntı Notu: Görsel https://en.wikipedia.org/wiki/Continuous_uniform_distribution adresinden alınmıştır.

Olasılık yoğunluk fonksiyonun grafiği de şöyledir:



Alıntı Notu: Görsel <https://www.researchgate.net/publication/332236648/figure/fig8/AS:865450120445954@1583350796843/Continuous-uniform-distribution-Source-14.ppm> adresinden elde edilmiştir.

Bu grafikte dikdörtgenin içerisindeki alanın 1 olması gerektiğine dikkat ediniz. Bu dikörtgensel alanda aynı uzunluktaki aralığın olasılığı aynı olacaktır.

Düzgün dağılımla işlemler scipy.stats kütüphanesindeki "singleton" uniform nesnesi ile yapılabilmektedir. Nesnenin kullanımı norm nesnesiyle benzerdir. Örneğin:

```
from scipy.stats import uniform
import matplotlib.pyplot as plt

result = uniform.cdf([50, 60, 70], 0, 100)
print(result)

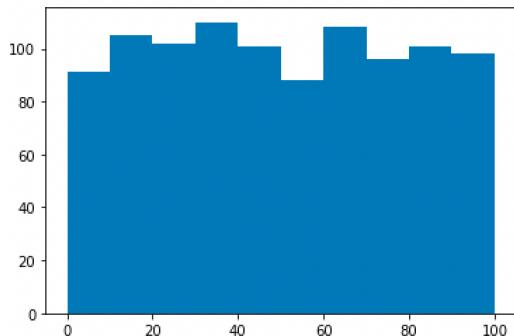
result = uniform.ppf([0.3, 0.5, 0.7], 0, 100)
print(result)

result = uniform.pdf([0.3, 0.5, 0.7], 0, 100)
print(result)
```

```
result = uniform.rvs(0, 100, 1000)
plt.hist(result)
```

Şöyledir bir çıktı elde edilmiştir:

```
[0.5 0.6 0.7]
[30. 50. 70.]
[0.01 0.01 0.01]
```



t Dağılımı (Student's t Distribution)

Özellikle hipotez testlerinde karşımıza çıkan diğer bir dağılım da t dağılımıdır. (Dağılım W. S. Gosset tarafından 1908 yılında geliştirilmiştir. Gosset makalesini "Student (Öğrenci)" takma adıyla yayınladığı için dağılım bu isimle de anılmaktadır.) t Dağılımı normal dağılıma benzerdir. Dağılımin olasılık yoğunluk fonksiyonunu biraz karışık olduğu gerekçesiyle burada vermeyeceğiz. Ancak t dağılımı normal dağılıma göre daha düz, daha az yüksek ancak daha geniş bir görünümdedir. Örneklem miktarı arttıkça t dağılımı standart normal dağılıma benzer. t dağılımındaki değerlere (yani X değerlerine) t değerleri denilmektedir. (Standart normal dağılımdaki X değerlerine Z değerleri dendiğini anımsayınız.) t dağılımının şekli "serbestlik derecesi (degrees of freedom)" denilen bir değere bağlıdır. Serbestlik derecesi örneklem büyüklüğünün bir eksigidir. Yani n örneklem büyülüğu olmak üzere serbestlik derecesi şöyle hesaplanmaktadır:

$$df = n - 1$$

t dağılımının ortalaması standart normal dağılımda olduğu gibi 0'dır. Ancak standart sapması 1 değildir. 1'den biraz daha fazladır. t dağılımının standart sapması şöyle hesaplanır:

$$\sigma = \sqrt{\frac{df}{df - 2}}$$

Standart sapmanın 1'den büyük olması standart normal dağılıma göre eğrinin daha geniş (daha şişman) olmasına yol açmaktadır. Bu eşitlikte df arttıkça standart sapmanın 1'e yaklaşmasına dikkat ediniz.

t dağılımı `scipy.stats` modülü içerisindeki `t` isimli nesne ile yapılmaktadır. Nesnenin ilişkin olduğu sınıfın `cdf` metodu yine birikimli olsılığı hesaplar. `ppf` fonksiyonu ters işlemi yapmaktadır. `pdf` fonksiyonu ise X değerleri için olasılık yoğunluk fonksiyonun Y değerini vermektedir. Bu fonksiyonların hepsinin birinci parametreleri hesaplanacak değerleri, ikinci parametreleri serbestlik derecesini, üçüncü ver dördüncü parametreleri ise ortalama ve standart sapma değerlerini almaktadır. Örneğin:

```
from scipy.stats import t
from scipy.stats import norm

p = t.cdf([0, 1, 2], 10)
print(p)
```

```
p = norm.cdf([0, 1, 2])
print(p)
```

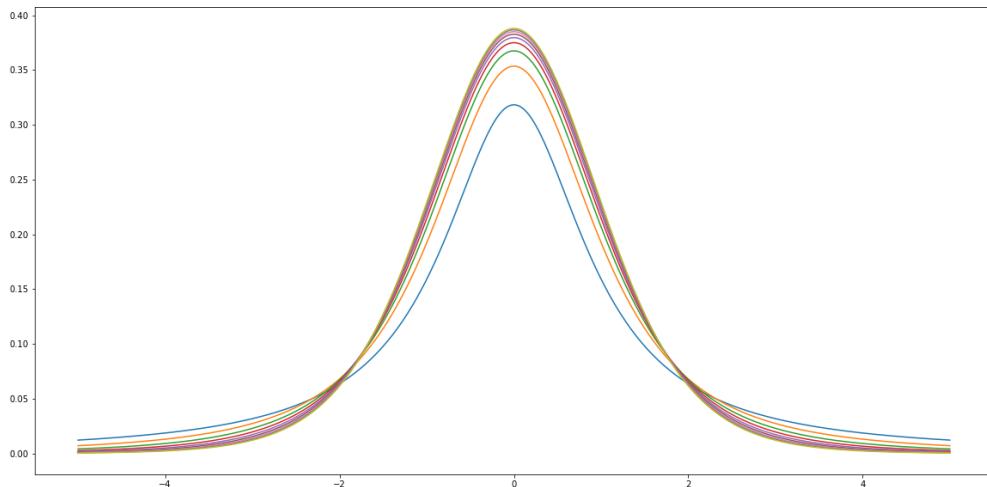
Programdan şöyle bir çıktı elde edilmiştir:

```
[0.5      0.82955343 0.96330598]
[0.5      0.84134475 0.97724987]
```

Şimdi de çeşitli serbestlik dereceleri için t dağılımının eğrilerini çizelim:

```
import numpy as np
from scipy.stats import t
import matplotlib.pyplot as plt

fig = plt.gcf()
fig.set_size_inches(20, 10)
x = np.linspace(-5, 5, 1000)
for df in range(1, 10):
    y = t.pdf(x, df)
    plt.plot(x, y)
```



Uygulamada t dağılımı anakütle parametresinin bilinmediği durumlarda örneklemden anakütle parametrelerinin tahmininde kullanılmaktadır.

Diger Sürekli Dağılımlar

İstatistikte çeşitli olguların olasılıklarını modellemek için ya da çeşitli konularda dolaylı olarak kullanılan pek çok sürekli dağılım vardır. Bu dağılımların büyük çoğunluğu için `scipy.stats` modülünde hazır nesneler bulunmaktadır. Temel işlemler yine ilgili sınıfların benzer metodlarıyla yapılmaktadır.

Merkezi Limit Teoremi (Central Limit Theorem)

Süphesiz sonuç çıkarıcı istatistiğin (inferential statistics) en önemli teoremi merkezi limit teoremidir. Bu teorem normal dağılımın bizim için neden bu kadar önemli olduğunu da göstermektedir. Teorem uzun süredir biliniyor olmasına karşın bu isimle ilk kez George Pólya tarafından 1920 yılında kullanılmıştır. Merkezi limit teoreminin biçimsel ifadesi biraz karmaşık bir görünümdedir. Biz burada sözel anlatımı üzerinde duracağız.

Bu teoreme göre bir anakütleden elde edilen örneklem ortalamaları normal dağılıma eğilimindedir. Eğer anakütle normal dağılmışsa küçük örneklemelerin ortalamaları da normal dağılır. Ancak ana kütle normal dağılmamışsa örneklem ortalamalarının normal dağılması için örneklemelerin belli bir büyüklükte (tipik olarak ≥ 30) olması gerekmektedir. Yine bu teoreme göre örneklem ortalamalarının ortalaması ana kütle ortaamasına eşittir. Örneklem

ortalamalarının standart sapması ise anakütle standart sapmasının örneklem büyülüğünün kareköküne bölümüne eşittir. Teoremin bu kısmını matematiksel olarak şöyle ifade edebiliriz:

$$\mu_{\bar{x}} = \mu$$

$$\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$$

Buradaki n örneklem büyülüğünü belirtmektedir. n örneklem büyülüğu N ise anakütle büyülüği olmak üzere,

$\frac{n}{N} > 0.05$ ise yani örneklem büyülüğünün anakütle büyülüğine oranı %5'ten büyük ise örneklem standart sapması şöyle olur:

$$\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}} \sqrt{\frac{N-n}{N-1}}$$

Burada $\sqrt{\frac{N-n}{N-1}}$ oranına düzeltme faktörü denilmektedir.

Şimdi merkezi limit teoremini deneme yoluyla doğrulamaya çalışalım. Önce anakütle normal dağılmış olsun ve küçük bir örneklem üzerinde çalışalım. Bunun için anakütleyi temsil eden 1000 tane normal dağılmış rastgele sayı üreteceğiz. Sonra bu rastgele sayılardan örneğin 5'lik ($n < 30$) rastgele 1000_000 örneklem çekip onların ortalamalarına ilişkin histogram çizeceğiz. (Şüphesiz 1000 elemanlı bir kümenin 5'li tüm alt kümelerini ele alamayız. $C(1000, 5)$ çok büyük bir değerdir. Biz 5'li rastgele 1000_000 değer çekmekle yetineceğiz. Bunun bir hata kaynağı oluşturacağı muhakkaktır.

```
import numpy as np
import matplotlib.pyplot as plt

NPOPULATION = 1000
NSAMPLES = 1000_000
SAMPLE_SIZE = 5

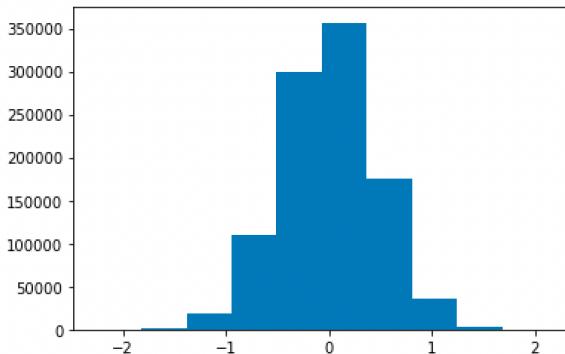
population = np.random.normal(0, 1, NPOPULATION)
samples = np.random.choice(population, (NSAMPLES, SAMPLE_SIZE))
smeans = np.mean(samples, axis=1)

plt.hist(smeans)
plt.show()

smeans_mean = np.mean(smeans)
population_mean = np.mean(population)
population_std = np.std(population)
smeans_std = np.std(smeans)
sigma_slash_sqrt_n = population_std / np.sqrt(SAMPLE_SIZE)

print(f'Örneklem ortalamalarının ortalaması: {smeans_mean}')
print(f'Anakütle ortalaması: {population_mean}')
print(f'Anakütle standart sapması: {population_std}')
print(f'Örneklem ortalamalarının standart sapması: {smeans_std}')
print(f'sigma / sqrt(n) değeri: {sigma_slash_sqrt_n}'')
```

Programın çalıştırılması sonucunda şöyle bir çıktı elde edilmiştir:



Örneklem ortalamalarının ortalaması: 0.006341131037852003

Anakütle ortalaması: 0.006536944315343625

Anakütle standart sapması: 1.0182981841666945

Örneklem ortalamalarının standart sapması: 0.4561456393309987

σ / \sqrt{n} değeri: 0.45539679223226576

Bu çıktıdan da gördüğünüz gibi örneklem ortalamalarının histogramı Gauss eğrisine benzemektedir. Örneklem ortalaması ile anakütle ortalaması arasındaki ve örneklem standart sapması ile anakütleden hareketle elde edilen standart sapma arasındaki küçük fark alınan örneklemlerin toplam sayısının (1000_000) az olmasından ve yuvarlama hatalarından kaynaklanmaktadır.

Şimdi aynı denemeyi anakütlenin normal dağılmadığı durum için yineleyelim. Bu kez anakütle düzgün dağılmış olsun ve örneklem büyülüğini 50 ($n \geq 30$) olarak seçelim. Yine anakütle büyülü 10000 olsun:

```
import numpy as np
import matplotlib.pyplot as plt

NPOPULATION = 10000
NSAMPLES = 1000_000
SAMPLE_SIZE = 50

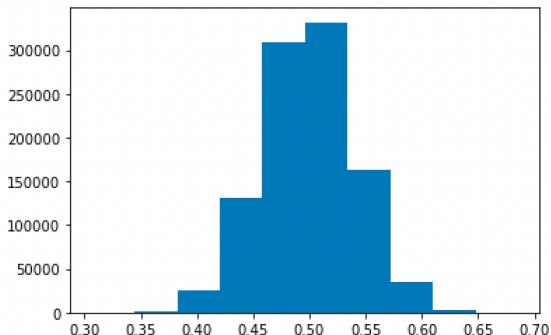
population = np.random.uniform(0, 1, NPOPULATION)
samples = np.random.choice(population, (NSAMPLES, SAMPLE_SIZE))
smeans = np.mean(samples, axis=1)

plt.hist(smeans)
plt.show()

smeans_mean = np.mean(smeans)
population_mean = np.mean(population)
population_std = np.std(population)
smeans_std = np.std(smeans)
sigma_slash_sqrt_n = population_std / np.sqrt(SAMPLE_SIZE)

print(f'Örneklem ortalamalarının ortalaması: {smeans_mean}')
print(f'Anakütle ortalaması: {population_mean}')
print(f'Anakütle standart sapması: {population_std}')
print(f'Örneklem ortalamalarının standart sapması: {smeans_std}')
print(f'\sigma / \sqrt{n} değeri: {sigma_slash_sqrt_n}')
```

Programın çalıştırılması sonucunda elde edilen çıktı şöyledir:



Örneklem ortalamalarının ortalaması: 0.4995729042905874
 Anakütle ortalaması: 0.4995699925304928
 Anakütle standart sapması: 0.29005119517418737
 Örneklem ortalamalarının standart sapması: 0.04108047000586231
 sigma / sqrt(n) değeri: 0.041019433399786136

Gördüğünüz gibi anakütlenin normal dağılmadığı ve örneklem büyülüğünün ≥ 30 olduğu durum için de merkezi limit teoremi deneyel olarak doğrulanıyor.

Merkezi limit teoreminin biçimsel (formal) ispatını burada yapmayacağız. Bunun için başka kaynaklara başvurabilirsiniz.

Normalliğinin Test Edilmesi

Parametrik istatistiksel yöntemlerde anakütle veya örneklem dağılımının normal olduğu varsayımları bulmaktaadır. Bu nedenle örneklem dağılımının normal olup olmadığından belli bir örneklem dayalı olarak test edilmesi gerekebilir. Normallik testi gözle üstünkörü yapılabılır. Bunun için histogram çiziliş şeklin Gauss eğrisine benzeyip benzemediğine bakılabilir. Normallik testi için çeşitli istatistiksel hipotez testleri geliştirilmiştir. Bunların bazı bakımlardan birbirlerine üstünlükleri ve zayıflıkları vardır. Burada biz konunun ayrıntılarına girmeyeceğiz.

Normallik testlerinden birisi Kolmogorov-Smirnov testidir. Bu test scipy.stats modülündeki kstest fonksiyonuyla yapılmaktadır.

```
scipy.stats.kstest(rvs, cdf, args=(), N=20, alternative='two-sided', mode='auto')
```

Kolmogorov-Smirnov testi parametrik olmayan istatistiksel bir hipotez testidir. Bu testte H_0 hipotezi söz konusu değerlerin düzgün dağılmış bir anakütleden geldiğini H_1 hipotezi ise söz konusu değerlerin düzgün dağılmış bir anakütleden gelmediği biçimindedir. Test işleminden sonra p değeri belirlenen kritik değerden (örneğin 0.05) küçükse H_0 hipotezi kabul edilir. Bu durumda değerlerin normal dağılıma sahip olmayan bir anakütleden gelmemektedir. Eğer p değeri bu kritik değerden büyükse bu durumda H_0 hipotezi reddedilir, H_1 hipotezi kabul edilir. Bu da değerlerin normal dağılmış bir anakütleden çekildiği anlamına gelmektedir.

kstest fonksiyonunu KTestResult isimli bir sınıf nesnesine (isimli demet nesnesine) geri dönmektedir. Nesnenin statistic elemanı Kolmogorov-Smirnov test istatistiğini pvalue elemanı ise p değerini vermektedir. Burada uygulamacı p değerine bakmalı eğer bu p değeri belirlediği kritik değerden (alfa) düşükse H_0 hipotezini, yüksekse H_1 hipotezini kabul etmelidir. Yani dağılımin normal olduğunu kabul edebilmemiz için bu pvalue değerinin 0.05 gibi bir eşik değerinden büyük olması gerekmektedir.

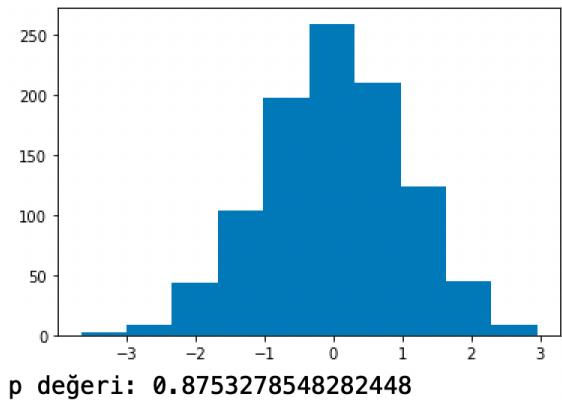
Şimdi test amaçlı 1000 tane standart normal dağılıma ilişkin rastgele sayı üretelim. Sonra bu sayılarla Kolmogorov-Smirnov testi uygulayalım:

```
from scipy.stats import kstest, norm
import matplotlib.pyplot as plt

a = norm.rvs(size=1000)
plt.hist(a)
plt.show()
```

```
tresult = kstest(a, 'norm')
print(f'p değeri: {tresult.pvalue}')
```

Programın çalıştırılması sonucunda şöyle bir çıktı elde edilmiştir:

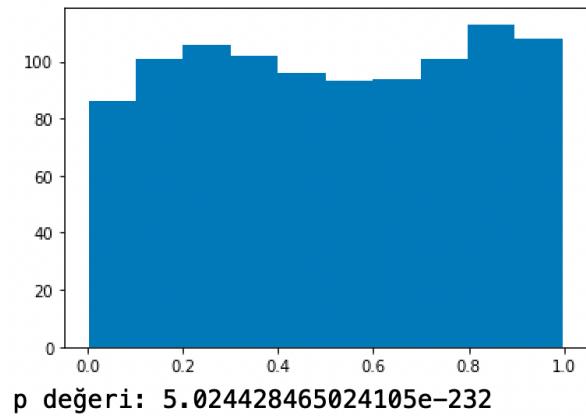


Burada p değerinin 0.05 gibi bir kritik değerden büyük olduğu görülmektedir. O halde bu verilerin normal dağıldığını ya da normal dağılmış bir anakütleden geldiğini varsayılabılır. Şimdi aynı testi düzgün dağılmış rastgele değerlerle gerçekleştirelim:

```
from scipy.stats import kstest, uniform
import matplotlib.pyplot as plt

a = uniform.rvs(size=1000)
plt.hist(a)
plt.show()
tresult = kstest(a, 'norm')
print(f'p değeri: {tresult.pvalue}')
```

Program çalıştırıldığında şöyle bir çıktı elde edilmiştir:



Göründüğü gibi p değeri 0'a çok yakındır. Bu durumda H₀ hipotezi kabul edilmektedir. Yani veriler normal dağılıma uygun değildir ya da normal dağılmış bir anakütleden gelmemektedir.

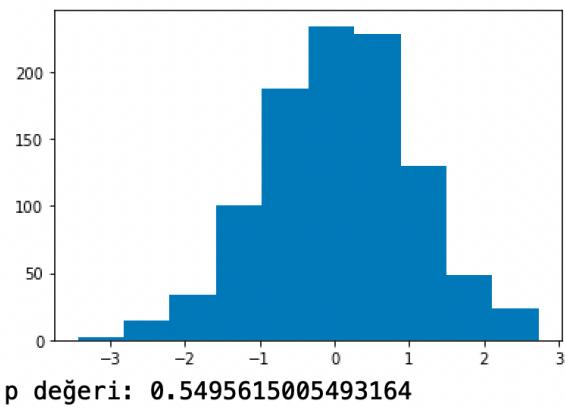
Diğer yaygın kullanılan normallik testlerinden biri de Shapiro-Wilk testidir. Burada biz bu iki testin teknik farklılıklarını üzerinde durmayacağız. Bunun için ilgili kaynaklara başvurabilirsiniz. Shapiro-Wilk testinde de H₀ hipotezi verilerin normal dağılıma ilişkin anakütleden gelmediğini, H₁ hipotezi verilerin normal dağılıma ilişkin bir anakütleden geldiğini biçimindedir. Karar yine test sonucunda elde edilen p değerinin kritik değerden küçük olup olmadığına göre verilmektedir.

Shapiro-Wilk testi scipy.stats modülü içerisindeki shapiro fonksiyonu kullanılmaktadır. Fonksiyon dağılım verilerini parametre olarak alır. Şimdi normal dağılmış rastgele değerlerden oluşan veri kümesi üzerinde Shapiro-Wilk testi uygulayalım:

```
from scipy.stats import shapiro, norm
import matplotlib.pyplot as plt

a = norm.rvs(size=1000)
plt.hist(a)
plt.show()
tresult = shapiro(a)
print(f'p değeri: {tresult.pvalue}')
```

Program çalıştırıldığında şöyle bir sonuç elde edilmiştir:



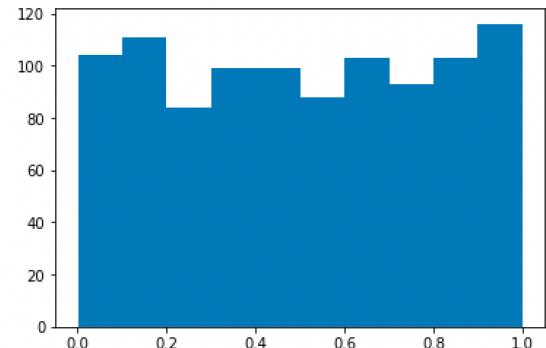
p değeri: 0.5495615005493164

Göründüğü gibi p değeri oldukça yüksektir. Bu durumda H₀ hipotezi reddedilir. Dolayısıyla örneklem değerleri normal dağılmış bir anakütleden gelmektedir. Yine aynı denemeyi düzgün dağılmış bir rassal değişken için deneyelim:

```
from scipy.stats import shapiro, uniform
import matplotlib.pyplot as plt

a = uniform.rvs(size=1000)
plt.hist(a)
plt.show()
tresult = shapiro(a)
print(f'p değeri: {tresult.pvalue}')
```

Program çalıştırıldığında şöyle bir sonuç elde edilmiştir:



p değeri: 3.4066265135381214e-18

p değerinin sıfıra çok yakın olduğunu görüyorsunuz. Bu durumda H₀ hipotezi kabul edilmektedir. Yani değerler normal dağılmış bir anakütleden gelmemektedir.

Örnekten Hareketle Anakütle Ortalamasının Tahmin Edilmesi ve Güven Aralıkları

Güven aralıkları (confidence intervals) bir anakütleden çekilen örneklem bağlı olarak anakütle parametrelerinin belli bir güven düzeyi (confidence level) içerisinde aralıksal olarak belirlenmesi için kullanılan istatistiksel bir yöntemdir. Merkezi limit teoremine göre bir anakütleden çekilen örneklemelerin ortalamalarının normal dağıldığını görmüştük. O halde biz bir anakütleden rastgele bir örnek seçip onun ortalamasına bakarak anakütle ortalamasını belli bir güven düzeyinde tahmin edebiliriz.

Güven aralıklarının oluşturulması örneklem dağılımına bakılarak yapılmaktadır. Eğer anakütle varyansı biliniyorsa anakütleden seçilen bir örneğin ortalamasından hareketle ana kütle parametreleri tahmin edilebilir. Örneğin elimizde ortalaması 100 standart sapması 15 olan normal dağılmış 1000 adet değer olsun. Biz de bu anakütleden 10 elemanlık rastgele bir örneklem seçip onun ortalamasını bulalım:

```
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt

POPULATION_SIZE = 1000
SAMPLE_SIZE = 10

population = norm.rvs(100, 15, POPULATION_SIZE)
population_mean = np.mean(population)
population_std = np.std(population)
sample = np.random.choice(population, SAMPLE_SIZE)
sample_mean = np.mean(sample)

print(f'Anakütle ortalaması: {population_mean}')
print(f'Anakütle standart sapması: {population_std}')
print(f'Seçilen örneğin ortalaması: {sample_mean}'')
```

Programın çalıştırılmasıyla şöyle bir çıktı elde edilmiştir:

```
Anakütle ortalaması: 99.96597670217074
Anakütle standart sapması: 14.996297931176969
Seçilen örneğin ortalaması: 104.10574797536367
```

Merkezi limit teoremine göre anakütle normal dağılırsa örneklem ortalamalarının dağılımı da normaldir. Daha önce de belirttiğimiz gibi örneklem dağılımının ortalaması ve standart sapması şöyledir:

$$\mu_{\bar{x}} = \mu$$

$$\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$$

O halde eğer bir anakütle standart sapmasını biliyorsak allığımız tek bir örneğin ortalamasından hareketle anakütle ortalamasını belli bir aralıkta ve güven düzeyinde tahmin edebiliriz. Yukarıda verdigimiz örnekte anakütle standart sapmasını bildiğimizi varsayırsak örneklem dağılımının standart sapması şöyle olacaktır:

```
sampling_std = population_std / np.sqrt(SAMPLE_SIZE)
print(sampling_std)
```

Bu işlemden şöyle bir değer elde edilmiştir:

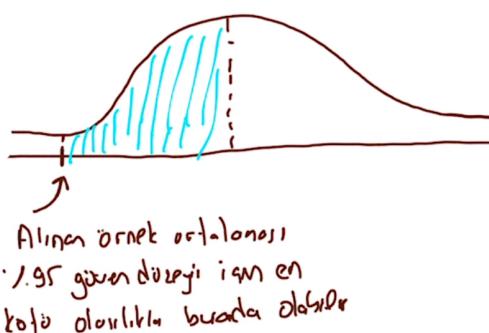
4.742245793299021

Bu durumda biz örneklem dağılımının standart sapmasını biliyoruz ancak örneklem dağılımının ortalamasını bilmiyoruz. Örneklem dağılımının ortalamasının anakütle ortalamasına eşit olmasına gerektiğini anımsayınız. Normal dağılım eğrisinde ortalama, eğrinin X ekseniye göre konumu üzerinde etkili olmaktadır. O halde tüm bu bilgilerden

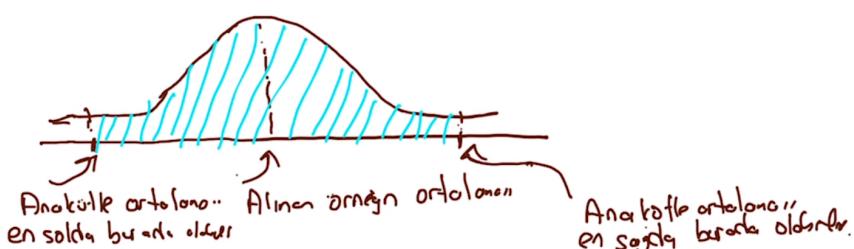
hareketle rastgele seçilen bir örnekten hareketle anakütle ortalamasının belli bir güven düzeyinde hangi aralıklarda olabileceğini belirleyebiliriz. Örneğin rastgele seçilen bir örnekten hareketle anakütle ortalamasının %95 güven düzeyinde hangi aralıkta olabileceğini anlamaya çalışalım. Aldığımız örnek en kötü olsayıkla anakütle ortalamasının 0.475 sağında olabilir. Aşağıdaki şekli inceleyiniz:



Yine alınan örnek en kötü olsayıkla anakütle ortalamasının 0.475 solunda olabilir:



Bu durumda anakütle ortalaması için .95 güven aralığı aslında alınan örnek ortalamasının solunda ve sağında 0.475 alana ilişkin X değerleridir:



Şimdi yukarıdaki örnek için 0.95 güven düzeyinde güven aralıklarını oluşturalım. Aldığımız örneğin ortalaması 104.10574797536367 idi. O halde ortalaması bu olan standart sapması 4.742245793299021 olan normal dağılımda 0.025 ve 0.975 birikimli olasılık değerini veren noktaları bulalım:

```
lower_bound = norm.ppf(0.025, 104.1057479753636, 4.742245793299021)
upper_bound = norm.ppf(0.975, 104.1057479753636, 4.742245793299021)
```

```
print(lower_bound, upper_bound)
```

Buradan şöyle bir çıktı elde edilmiştir:

```
94.8111701466094 113.40037893606626
```

O halde bu anaktüleden çektiğimiz örneklem dayanarak anakütle ortalamasının %95 güven aralığında bu sınırlar içerisinde olabileceğini gördük. Aslında bunu standart normal dağılımın kullanılacağı biçimde şöyle de ormülüze edebiliriz:

$$\bar{x} \pm z\sigma_{\bar{x}}$$

Buradaki z değeri güven düzeyine karşı gelen X eksenindeki değerdir. $\sigma_{\bar{X}}$ ise örneklem dağılımının standart sapması olan σ / \sqrt{n} değeridir.

Şimdi standart sapması 10 olan bir anakütleden çekilen 100'lük bir örneğin ortalamasının 65 olduğunu varsayalım. Anakütle ortalaması 0.95 güven düzeyi içerisinde şöyleden hesaplanacaktır:

```
import numpy as np
from scipy.stats import norm

lower_bound = 65 + norm.ppf(0.975) * 10 / np.sqrt(100)
upper_bound = 65 - norm.ppf(0.975) * 10 / np.sqrt(100)
print(lower_bound, upper_bound)
```

Buradan çıktı elde edilmiştir:

63.04003601545995 66.95996398454005

Tabii aynı işlemi şöyleden yapabiliyoruz:

```
lower_bound = norm.ppf(0.975, 65, 10 / np.sqrt(100))
upper_bound = norm.ppf(0.025, 65, 10 / np.sqrt(100))
print(lower_bound, upper_bound)
```

Aslında güven aralıkları için norm nesnesine ilişkin sınıfta interval isimli bir metod da bulundurulmuştur. interval metodunu üç parametre almaktadır. Metodun birinci parametresi güven aralığını, ikinci parametresi ortalama değeri, üçüncü parametresi ise standart sapmayı belirtir. Fonksiyon güven aralığının düşük ve yüksek değerlere ilişkin bir demete geri dönmektedir. Bu durumda standart sapması 10 olan bir anakütleden çekilen 65 ortalamaya sahip 100'lük bir örnekten hareketle anakülenin ortalamasını %95 güven düzeyinde şöyleden belirleyebiliriz:

```
import numpy as np
from scipy.stats import norm

ci = norm.interval(0.95, 65, 10 / np.sqrt(100))
print(ci)
```

Programın çalıştırılmasıyla şu çıktı elde edilmiştir:

(63.04003601545995, 66.95996398454005)

Merkezi limit teoremine göre eğer ana kütle normal dağılmamışsa ancak $n \geq 30$ koşulunu sağlayan örneklem dağılımlarının normal dağıldığı kabul edilmektedir. Yani örneklerimizdeki gibi anakütle ortalamasının tahmin edilmesi ve güven aralıklarının oluşturulması için şu iki koşuldan en az biri sağlanmalıdır:

- 1) Anakütle normal dağılmıştır ve örneklem dağılımı için $n < 30$ durumu söz konusudur.
- 2) Anakütle normal dağılmamıştır ve örneklem dağılımı için $n \geq 30$ durumu söz konusudur.

Pekiyi anakütle normal dağılmamışsa ve $n < 30$ ise ne olacaktır? İşte bu durumda biz yukarıda uygulandığı biçimde anakütle ortalamasını ve güven aralıklarını belirleyemeyiz. Bu durumda "parametrik olmayan" başka yöntemler kullanılabilmektedir. Ancak biz bu yöntemleri burada ele almayacağız.

Merkezi limit teoremine göre bizim anakütle ortalamasını örnekten hareketle tahmin edebilmemiz için anakütle standart sapmasını biliyor olmamız gereklidir. Pekiyi ya bunu bilmiyorsak ne yapabiliyoruz. Örneğin aşağıdaki gibi bir soruya nasıl çözebiliriz?

İşte eğer anakütle standart sapması bilinmiyorsa bu durumda t dağılımı kullanılmaktadır. Alınan örneğin standart sapması sanki anakülenin standart sapması gibi ele alınmaktadır. t dağılımının bir serbestlik derecesi (degrees

of freedom) parametresinin olduğunu anımsayınız. Burada serbestlik derecesi örnek büyülüğünün 1 eksik değeridir. Anakütle standart sapmasının bilinmemesi durumunda da örneklem dağılımına ilişkin örneklem büyülüğu önemli olmaktadır. Eğer alınan örnek büyük değilse (< 30) anakütlenin normal dağılmış olması gerekmektedir. Eğer alınan örnek yeteri kadar büyükse (≥ 30) bu durumda anakütle dağılımı normal olmak zorunda değildir.

Şimdi elimizde bir anakütleden çekilmiş olan 35 elemandan oluşan aşağıdaki gibi bir örneklem olsun:

```
sample = np.array([101.93386212, 106.66664836, 127.72179427, 67.18904948,
    87.1273706 , 76.37932669, 87.99167058, 95.16206704,
    101.78211828, 80.71674993, 126.3793041 , 105.07860807,
    98.4475209 , 124.47749601, 82.79645255, 82.65166373,
    92.17531189, 117.31491413, 105.75232982, 94.46720598,
    100.3795159 , 94.34234528, 86.78805744, 97.79039692,
    81.77519378, 117.61282039, 109.08162784, 119.30896688,
    98.3008706 , 96.21075454, 100.52072909, 127.48794967,
    100.96706301, 104.24326515, 101.49111644])
```

Şimdi %95 güven düzeyinde anakütle ortalamasının hangi aralıklar içerisinde olabileceğini bulmaya çalışalım. Burada biz anakütle standart sapmasını bilmiyoruz. Anakütlenin normal dağılıp dağılmadığını bilmemişimizi düşünelim. (Örnekten hareketle anakütlenin normal dağılıp dağılmadığını daha önce görmüş olduğumuz normalilik testleriyle belirleyebilirsiniz.) Bu durumda anakütle ortalaması için aralık tahmini t dağılımı kullanılarak yapılmalıdır. Öncelikle bu örneğin ortalamasını ve standart sapmasını bulalım:

```
sample_mean = np.mean(sample)
sample_std = np.std(sample)
```

Örneğin standart sapmasını anakütlenin standart sapması kabul ederek örneklem dağılımının standart sapmasını bulalım:

```
sampling_std = sample_std / np.sqrt(35)
```

Örnek büyülüğu 35 olduğuna göre serbestlik derecesi 34 olacaktır. Artık t dağılımından hareketle güven aralığını bulabiliriz:

```
from scipy.stats import t

lower_bound = t.ppf(0.025, 34, sample_mean, sampling_std)
upper_bound = t.ppf(0.975, 34, sample_mean, sampling_std)
print(lower_bound, upper_bound)
```

Şöyledir bir çıktı elde edilmiştir:

```
94.89296371821018 105.02201556521837
```

Tabii aynı işlemleri t nesnesine ilişkin sınıfın interval metoduyla da yapabiliyoruz:

```
ci = t.interval(0.95, 34, sample_mean, sampling_std)
print(ci)
```

YAPAY SINİR AĞLARI

Yapay sinir ağları yapay zekanın ve makine öğrenmesinin önemli alt alanlarından biridir. Derin öğrenme (deep learning) de bir çeşit yapay sinir ağı yöntemidir. Bu bölümde yapay sinir ağları belli düzeylerde teorik bakımdan ele alınacak ve daha çok uygulamaları üzerinde durulacaktır.

Yapay Sinir Ağlarının Tarihi

Yapay Sinir ağlarının teorisi ilk zamanlar sinir bilimle (neuroscience), psikolojiyle ve matematikle uğraşan bilim adamları tarafından geliştirilmiştir. Yapay sinir ağları ilk kez Warren McCulloch ve Walter Pitts isimli kişiler tarafından 1943 yılında ortaya atılmıştır. 1940'lı yılların sonlarına doğru Donald Hebb isimli psikolog da "Hebbian Learning" kavramıyla, 1950'li yıllarda da Frank Rosenblatt isimli araştırmacı da "Perceptron" kavramıyla alana önemli katkılarında bulunmuştur. Bu yıllar henüz elektronik bilgisayarların çok yeni olduğu yıllardır. Halbuki yapay sinir ağlarına yönelik algoritmalar için önemli bir CPU gücü gerekmektedir. Bu nedenle özellikle 1960 yıllarda bu konuda bir motivasyon eksikliği oluşmuştur. Yapay sinir ağları sonraki dönemlerde yeniden popüler olmaya başlamıştır. Derin öğrenme konusunun gündeme gelmesiyle de popüleritesi heften artmıştır.

Yapay Sinir Ağlarının Uygulama Alanları

Yapay Sinir ağlarının pek çok farklı alanda uygulaması vardır. Örneğin:

- Örütü Tanıma
- Finansal Uygulamalar (Portföy yönetimi, Kredi değerlendirmesi vs.)
- Endüstriyel problemlerin çözümü
- Biyomedikal Mühendisliğinde (Örneğin hastalığa tanı koyma ve tedavi planı oluşturma)
- Optimizasyon problemlerinde
- Pazarlama Süreçlerinde

İnsanın Sinir Sistemi

Sinir sistemi iki bölüme ayrılmaktadır:

- Merkezi Sinir Sistemi (Central Nervous System)
- Çevresel Sinir Sistemi (Peripheral Nervous System)

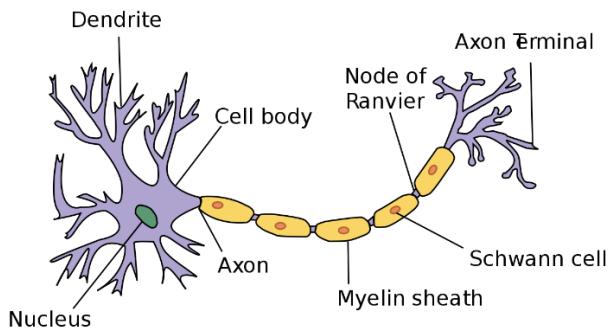
Merkezi sinir sistemi beyin ve omurilikten oluşmaktadır. Beyinden çıkan nöral iletiler omurilikten geçerek tüm vücuda yayılmaktadır. Beyin ve omuriliğin dışındaki nöral ağa çevresel sinir sistemi denilmektedir. Kaslarımız nöronlar tarafından harekete geçirilirler. Fakat bu emir çoğu kez beyin tarafından verilmektedir. Kasları harekete geçiren nöronlara motor nöron denilmektedir. Merkezi sinir sistemi de kendi içerisinde "somatik sinir sistemi" ve "otonom sinir sistemi" olmak üzere ikiye ayrılır. Somatik sinir sistemi bilinçli eylemlere yol açan kısım için otonom sinir sistemi ise bilinçsiz eylemlere yol açan kısım için kullanılmaktadır. Kalp kaslarının kasılmaları, solunum gibi faaliyetler bilinçli olarak gerçekleşmezler. Otomatik biçimde (otonom) gerçekleştirler.

Duyumlar (sensation) bu konuda özelleşmiş nöronlar vasıtasiyla gerçekleşmektedir. Fiziksel uyaranlar bu nöronları uyarır. Bu nöronlar bu iletiyi kimyasal düzeye dönüştürürler (transduction) sonra duruma göre omuriliğe ve oradan da beynin ilgili bölümünü iletirler. Fiziksel uyaranları alan özelleşmiş bu nöronlara "duyusal nöronlar (sensory neurons)" denilmektedir.

Felç olgusunun çeşitli nedenleri vardır. En çok karşılaşılan nedenleri beyindeki lezyonlar (bein kanaması sonucunda ya da tümörel biçimde) ya da kazalardır. Diğer nedenler arasında omurilik zedelenmeleri, fitikler gibi sorunlar bulunmaktadır.

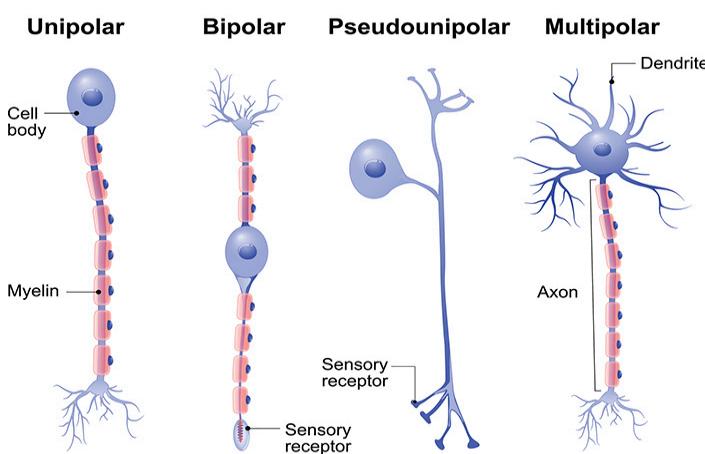
Sinir sisteminin en önemli hücreleri nöronlardır. Nöronların en fazla bulunduğu yer beyindir. Beyinde 100 milyar civarında nöron vardır. Nöronların dışında sinir sisteminde gliya gibi nöromodülör hücreler de bulunmaktadır.

Bir nöron hücresi tipik olarak aşağıdaki gibidir:



Alıntı Notu: GörSEL <https://simple.wikipedia.org/wiki/Neuron> adresinden alınmıştır.

Nöronların da birkaç çeşidi vardır. Yukarıdaki şekilde görüldüğü tipteki nöronlara "multipolar nöronlar" denilmektedir. Bu nöronlar sinir sisteminde en fazla bulunan nöronlardır. Aşağıda diğer bazı nöron çeşitlerini görüyorsunuz:

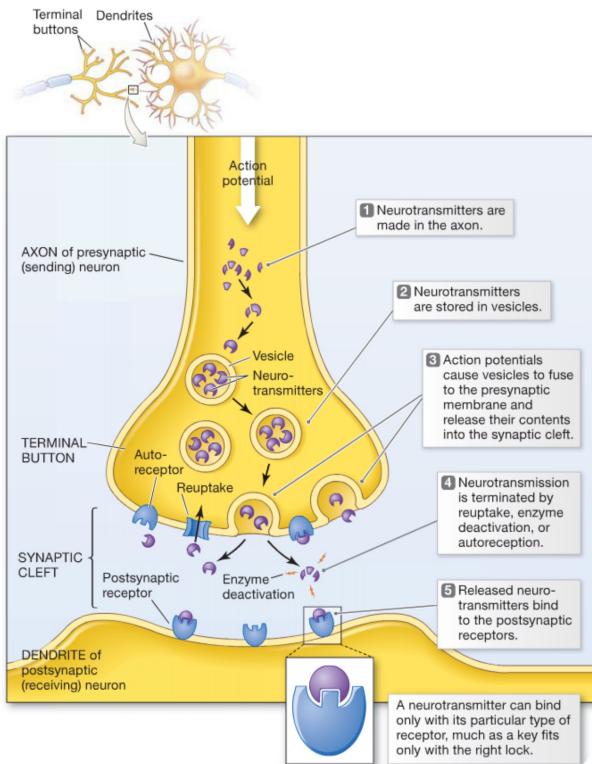


Alıntı Notu: GörSEL <https://qbi.uq.edu.au/brain/brain-anatomy/types-neurons> adresinden alınmıştır.

Multipolar nöronlar bir çekirdekten, nöral iletiyi alan dendrit'lerden ve nöral iletiyi iletten aksonlardan oluşurlar. Akson denilen uzun dalın üzerinde bazı nöronlarda miyelin kılıfı (myelin sheet) bulunmaktadır. Miyelin kılıfının en önemli işlevi nöral iletiyi hızlandırmasıdır. Miyelinli nöronların ileti hızı yaklaşık 80-120 m/sn, miyelinsiz nöronların ileti hızı ise yaklaşık 0.5-2 m/sn civarındadır. Beyinde hem miyelinli hem de miyelinsiz nöronlar bulunur. Miyelenli nöronlar beyaz renkte miyelinsiz olanlar ise gri renkte görüntü verdiklerinden beyinde miyelinli nöronların bulunduğu bölgelere "beyaz madde (white matter)", miyelinsiz nöronların bulunduğu bölgelere ise "gri madde (gray matter)" denilmektedir.

Pekiyi nöral ileti kimyasal düzeyde nasıl gerçekleşmektedir? İşte bir nöronun aksonunun ucunda düğmecikler (terminal buttons) bulunmaktadır. Nöron ateşlendiğinde bu düğmeciklerden "nörotransmitter" denilen moleküller zerk edilir. Nöral ileti büyük ölçüde bu nörotransmitterler yoluyla yapılmaktadır.

Akson uçları duruma göre yüzlerce ya da binlerce başka nörona bağlı olabilmektedir. İleti genellikle (ama her zaman değil) bir nöronun aksonu ile karşı taraftaki nöronun dendriti arasında gerçekleşir. Nöral ietinin gerçekleştiği bölgeye sinaps denilmektedir. Bir sinaps tipik olarak şöyledir:



Alıntı Notu: GörSEL <https://in.pinterest.com/pin/722546333950674913/> adresinden alınmıştır.

Yukarıda de belirtildiği gibi aksonların ucunda düğmecikler (axon terminal buttons) bulunmaktadır. Dendritlerde de ismine reseptör denilen küçük yarıklar (synaptic cleft) vardır. Akson uçlarındaki düğmeciklerden ismine nörotransmitter denilen kimyasallar zerk edilir. Zerk edilen bu nörotransmiterler dendritlerdeki reseptörler tarafından alınmaktadır. Ancak her türlü reseptör her türlü nörotransmitem'i alamamaktadır. Her nörotransmitem'i alan farklı reseptörler vardır.

Nörotransmiterler çeşitli alt sınıflara ayrılmaktadır. En önemli nörotransmiterler şunlardır:

Amino Asit Grubundan olanlar: Glutamat, GABA, Glisin

Monoamin Grubundan Olanlar: Serotonin, Histamin, Dopamin, Adrenalin, Noradrenalin. Dopamin, Adrenalin ve Noradrenaline "katakolaminler" de denilmektedir.

Peptit Grubundan Olanlar: Endorfin

Diğerleri: Asetilkolin, ...

Örneğin psikoaktif maddeler ve psikotrop ilaçlar bu nörotransmiterleri artırıp azaltabilmektedir. Sinapslarda nörotransmiterleri artırma etkisi yaratan maddelere "agonist", azalma etkisi yaratan maddelere ise "antagonist" denilmektedir. Örneğin "dopamin antagonisti" demek sinapslardaki dopamin seviyesini düşüren madde" demektir. Agonist ya da antagonist etki çeşitli biçimlerde oluşturulabilmektedir. Örneğin:

- Nörotransmiter üretimi artırılarak ya da azaltılarak
- Reseptörler bloke edilerek.
- Reseptörlerin etkinliği artırılarak (yani daha çok nörotransmiter olmasını sağlayarak).

Axon uçları nörotransmiterleri zerk ettikten sonra kullanılmayanları geri almaktadır. Buna "geri alım (reuptake)" işlemi denilmektedir. İşte bazı maddeler bu geri alımı engelleyerek agonist bir etki oluşturabilmektedir. Yukarıda de belirtildiği gibi psikiyatride kullanılan psikotrop ilaçlar büyük ölçüde sinapslardaki nörotransmiterler üzerinde etkili olmaktadır. Örneğin:

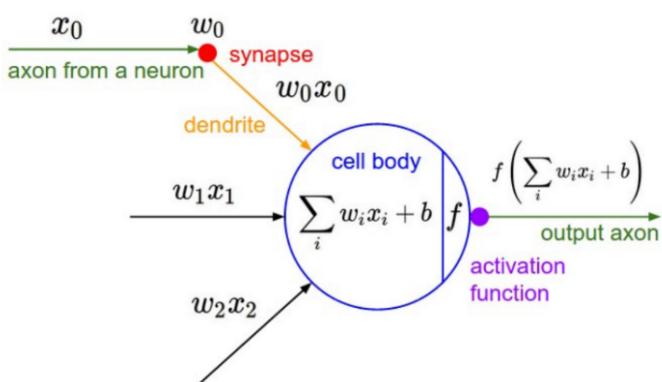
- SSRI türevi antidepresanlar serotonin geri alımını engelleyerek sinapslardaki serotonin miktarını artırmaktadır. Depresyonla sinapslardaki serotonin eksikliği arasında ilişki olduğu düşünülmektedir.

- Bazı trisklik antidepressanlar ve SNRI türevi antidepressanlar sinapslardaki noradrenalin miktarını artırmaktadır.
- Şizofreni ve diğer psikotik bozuklukların tedavilerinde kullanılan antipsikotikler genellikle dopaminerjik sistemle ilgili etki gösterirler. Bazı psikotik bozuklukların nedenleri beynin bazı bölgelerindeki dopaminerjik aktivitenin fazla olmasıyla açıklanmaya çalışılmaktadır.
- Sakinleştirici (sedatif) olarak kullanılan ilaçların çoğu (örneğin benzodiazepinler) GABA isimli nörotransmitteri artırmaktadır. GABA nöral ileti üzerinde inhibisyon etkisi yapan önemli bir nörotransmitter'dir.

Psikotrop ilaçlar deneyel yolla (hatta bazıları tesadüflerle) bulunmuştur. Maalesef neredeyse hiçbir psikotrop ilaçın çalışma mekanizması ayrıntılarıyla ve kesin düzeyde bilinmemektedir.

Yapay Nöron Modeli

Yapay sinir ağlarındaki nöral iletişim oldukça basitleştirilerek matematiksel bir algoritmik yönteme dönüştürülmüştür. İşte yapay sinir ağları temelde bu basitleştirilmiş matematiksel nöron modeline dayanmaktadır. Nasıl gerçek bir nörona birtakım girişler (dendritlerdeki reseptörler) ve çıkışlar (akon'dan zerk edilen nörotransmitterler) varsa matematiksel nöron modelinde de nörona çeşitli girişler ve bir çıkış söz konusudur. Örnek bir yapay nöron aşağıdaki şekilde temsil edilebilir. Bu tek nöronluk modele aynı zamanda "perceptron" da denilmektedir.



Bu örnek nöron modelinde x 'ler (x_0, x_1, x_2) gözlenen birtakım değişkenleri temsil etmektedir. Başka bir deyişle buradaki x 'ler tipik bir istatistik veri tablosundaki (yani data frame'lerdeki) sütunları temsil etmektedir. Örneğin bir apartman dairesinin o andaki piyasa değerini tahmin etmeye çalışan bir nöral ağ tasarlayacak olalım. Bu ağın girdileri (yani x değerleri) neler olabilir? Bir dairenin fiyatı neler bağlıdır? Burada kabaca şu unsurları sayabiliyoruz:

- Apartmanın yaşı
- Dairenin metrekare büyüklüğü
- Dairenin şehir merkezine uzaklığı
- Dairenin binanın kaçinci katında olduğu
- Dairenin oda sayısı

Ya da örneğin 10 tane biyomedikal tetkik sonucuna göre kişilerin belli bir hastalığa sahip olup olmadığını anlamaya çalışan bir yapay sinir ağ modeli kurmak isteyelim. Bu modeldeki x 'ler bu 10 biyomedikal tetkik değerleri olacaktır. Bu yapay sinir ağında çıktılar girdilere bağlandıktan sonra en hihayet tek bir çıkış elde edilecektir. Bu çıkış da kişinin o hastalığa sahip olduğunu verecektir.

Yukarıdaki nöron modelinde bu girdi değişkenlerinin birtakım w katsayılarıla çarpılıp toplandığı görülmeye. Sonra da bu toplam bir fonksiyona sokulup bir çıktı değeri elde edilmektedir. Peki buradaki w değerleri ne anlam ifade etmektedir? w değerleri aslında eğitimli ağlarda nihai olarak elde edilmiş bir daha değiştirilmeyecek ağırlık değerleridir.

Zaten yapay sinir ağının eğitilmesi demek aslında girildilere göre uygun sonucu verebilecek w ağırlık değerlerinin tespit edilmesi sürecidir. Eğitimli yapay ağlarında daha önceki verilere dayanılarak girdilerle çıktılar arasında bir ilişki kurulmaya çalışılır. Bu ilişki aslında w katsayılarının uygun biçimde belirlenmesiyle kurulmaktadır. Örneğin ev fiyatını belirleme probleminde biz yapay sinir ağına evin özelliklerini girdi olarak veririz. Yapay sinir ağı da evin fiyatını bize çıktı olarak verir. Ancak yapay sinir ağının bunu yapabilmesi için gerçek birtakım ev özellikleri ve fiyatlarıyla eğitilmesi gerekmektedir. İşte bu eğitim sırasında aslında yapay sinir ağındaki w katsayıları uygun biçimde ayarlanmaktadır. Eğitim bittikten sonra artık biz ağa yeni bir ev özellikleri verdığımızda bu w katsayıları ve aktivasyon fonksiyonları devreye girerek bize o evin fiyatını verecektir.

Biz yukarıda tek nöron üzerinde açıklamalar yaptık. Halbuki insanın sinir sisteminde olduğu gibi yapay sinir ağlarında da pek çok nöronun çıktıları başka nöronların girdilerine bağlanabilmektedir. İleride bu bağlantı modelleri hakkında bilgiler verilecektir.

Yapay Nöronun Python'da Bir Sınıfla Temsili

Yapay bir nöron basit bir sınıfta temsil edilebilir. Örneğin:

```
import numpy as np

class Neuron:
    def __init__(self, weights, activation, bias = 0):
        self.weights = weights
        self.activation = activation
        self.bias = bias

    def output(self, x):
        total = np.dot(self.weights, x)
        return self.activation(total + self.bias)

    def sigmoid(x):
        return 1 / (1 + np.exp(-x))

n = Neuron([0.2, 0.3, 0.4], sigmoid)
result = n.output([2, 1, 7])
print(result)
```

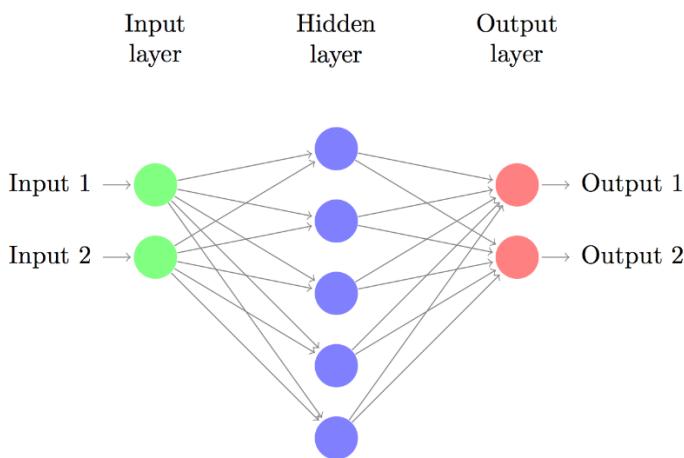
Burada Neuron sınıfı bizden nesne yaratıldığında ağırlık değerlerini, aktivasyon fonksiyonunu ve bias değerini istemiştir. Sınıfın output fonksiyonu da nöral işlemi yaparak bize çıktı değerini vermektedir.

Yapay Sinir Ağlarında Katmanlar

Bir yapay sinir ağında üç katman söz konusudur:

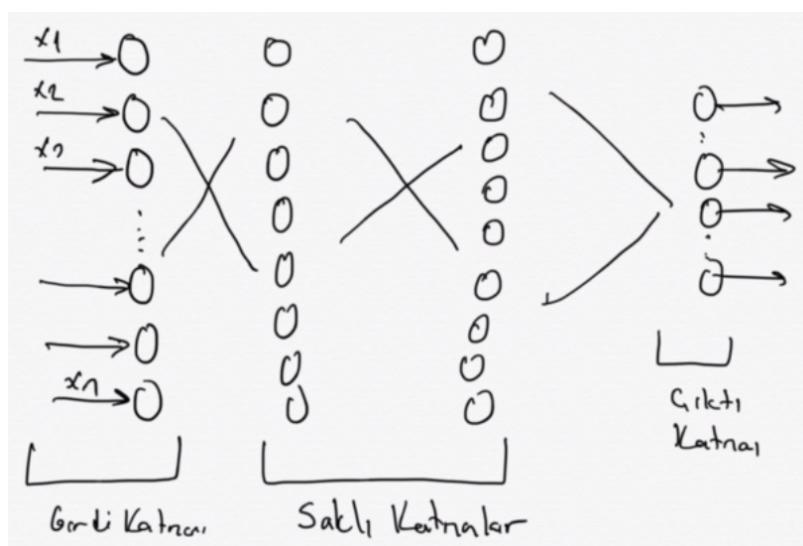
- 1) Girdi Katmanı (Input Layer / Visible Layer)
- 2) Saklı Katman (Hidden Layer)
- 3) Çıkış Katmanı (Output Layer)

Bir yapay sinir ağında katmanlar ve katmanlardaki nöronların oluşturduğu yapıya "ağ mimarisi" denilmektedir.



Girdi katmanı dış dünyadan girdi değerlerini (yani x değerlerini) alan katmandır. Girdi katmanındaki nöronlar içerisinde bir işlem yapılmamaktadır. Her ne kadar şekillerde girdi katmanı da her bir x giriş'i için bir nöronla temsil ediliyorsa da aslında gerçek bir nöron işlevi yapmamaktadır. Yani girdi katmanındaki nöronların bir giriş'i vardır. Bu nöronların çıkış değeri de giriş değerinin aynısıdır. Örneğin bir kişinin diyabetli olup olmadığını anlayabilmek için 10 farklı biyomedikal tetkik sonuçlarını girdi olarak kullanmak isteyelim. Bu durumda ağın girdi katmanı 10 nöronundan oluşacaktır. Bu 10 nöronun girdisi tamamen çıktı ile aynı olacaktır.

Basit problemler yalnızca girdi ve çıktı katmanı olan bir ağla çözülebilir. Ancak model karmaşıklaştıkça ara katman görevini yapan saklı katmanların kullanılması gerekmektedir. Saklı katmanlar düzey olarak bir tane, iki tane ya da ikiden çok olabilir. Örneğin:



Burada iki saklı katman bulunmaktadır. Duruma göre veri bilimcisi tarafından saklı katmanların sayısı artırılıp azaltılabilmektedir. Saklı katmanlardaki nöron sayıları girdi katmanındaki nöron sayısı ile aynı olmak zorunda değildir. Genellikle eğer girdi katmanındaki nöronların sayıları göreli olarak azsa saklı katmanlardaki nöronların sayıları fazlalaştırılır. Eğer girdi katmanında göreli olarak çok fazla nöron varsa saklı katmanlardaki nöronların sayıları işlem yüklerini azaltmak için düşürülebilmektedir. Genel olarak saklı katmanların sayısı ikiden büyükse bu tür yapay sinir ağlarına derin yapay sinir ağları ya da özetle derin öğrenme (deep learning) ağları denilmektedir. Yani derin ağlarla normal sinir ağları arasındaki fark saklı katmanların sayısı ile ilgilidir.

Çıktı katmanı sinir ağından elde edilecek bilgiyi dış dünyaya veren katmandır. Tabii çıktı katmanındaki nöronlarda da saklı katmanlarda olduğu gibi işlemler yapılmaktadır. Ancak çıktı katmanı artık değerlerin elde edildiği son katmandır.

Yani biz girdileri girdi katmanına veririz sonucu çıktı katmanından alırız. Çıktı katmanın kaç nöron dan oluşacağı çıktıının ne olduğu ile ilgilidir.

Şimdi katmanlardaki olası nöron sayıları hakkında bazı şeyler söyleyelim. Girdi katmanındaki nöron sayısı kestirimde kullanılacak özelliklerin (features) sayısı kadar olmalıdır. Çıktı katmanındaki nöronların sayısı da elde edilecek kestirim sonucuyla ilgilidir. Örneğin ağımız var/yok gibi, olumlu/olumsuz gibi, hasta/sağlıklı ikili bir kestirimde bulunacaksa (bu tür modellere istatistikte lojistik regresyon denilmektedir) çıktı katmanı tek bir nöron dan oluşur. Örneğin ağımız bir resimdeki rakamın kaç olduğunu kestirmeye çalışıyorsa bu durumda çıktı katmanında 10 tane nöron bulunur. Yine örneğin ağımız bir evin fiyatını tahmin etmeye çalışıyorsa çıktı katmanı evin fiyatını veren tek bir nöron dan oluşacaktır. O halde bilmemişimiz iki durum vardır: Ağıımızda kaç tane saklı katman ve her saklı katmanda kaç tane nöron bulunmalıdır? İşte bu soruların yanıtları deneyimle ve üzerinde çalışılan problemin nitelegine göre veri bilimcisi tarafından doğru biçimde verilmeye çalışılmaktadır. Ancak yine de bu konuda önerilen birkaç genel yönerge vardır. Saklı katmanların sayısı için şu pratik tavsiyelerde bulunulabilir. (<https://www.heatonresearch.com/2017/06/01/hidden-layers.html>)

- Eğer problem doğrusal olarak ayrılabilir (linear separable) ise saklı katman kullanmaya gerek yoktur. Doğrusal olarak ayrılabilir problem hakkında kısa teorik bilgi ileride verilecektir.
- Bir saklı katman pek çok sorunun çözümü için yeterlidir. İki saklı katman çok büyük ölçüde pek çok problemin çözümü için yeterli olmaktadır.
- İkiden fazla saklı katmanı olan derin modeller karmaşık problemlerin çözümü için kullanılmaktadır. Bunlara örnekler ileride verilecektir.

Bir saklı katmandaki nöron sayısı için üstünüğü pratik tavsiyeler de şunlar olabilir:

- Saklı katmandaki nöronların sayısı girdi katmanındaki nöronların sayısının $\frac{2}{3}$ ile çıktı katmanındaki nöronların sayısının toplamı kadar olabilir. Örneğin girdi katmanındaki nöron sayısı 5, çıktı katmanındaki 1 olsun. Bu durumda saklı katmandaki nöron sayısı 5 olabilir.
- Saklı katmandaki nöronların sayısı girdi katmanındaki nöron sayısının iki katından fazla olmamalıdır.

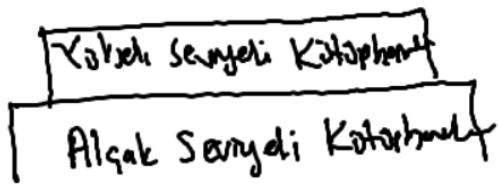
Tabii aslında saklı katmanlardaki nöron sayılarına ilişkin bu iki sezgisel yargıya bazı durumlarda uyulmamaktadır. Genellikle saklı katmanlardaki nöron sayıları deneyimle, deneme yanlışma yöntemleriyle belirlenmektedir. Şüphesiz bir fayda sağlanmadığı halde çok saklı katman ya da saklı katmanlarda çok nöron bulundurmasının bir anlamı yoktur. Ağ ne kadar sade olursa hem işlem yükü bakımından hem de verilerin saklanması ve iletilmesi bakımından o kadar avantaj sağlanır.

Yapay Sinir Ağlarında ve Derin Öğrenme Ağlarında Yaygın Kullanılan Kütüphaneler ve Framework'ler

Bir yapay sinir ağı mimarisinin sıfırdan programını yazmak çok zor olmasa da zaman alıcı bir süreçtir. Üstelik programcılar tarafından kendi gereksinimleri doğrultusunda yazılan bu tür kodların pek çoğu genel değildir ve genişletilememektedir. Bu konudaki diğer bir sorun ise yapay sinir ağlarındaki işlem yüküyle ilgilidir. Yapay sinir ağlarının eğitilmesi çok yoğun işlemleri gerektirebilmektedir. Bu işlemleri bir framework kullanmadan yapmak isteyen programcı kendisini paralel programlama gibi göreli karmaşık konular içerisinde bulabilmektedir. Bu da programcının toplam iş yükünü artırmaktadır.

İşte her konuda olduğu gibi yapay sinir ağlarında da işlemleri kolaylaştırmak için birtakım kütüphanelerden ve framework'lerden faydalılmaktadır. Bu tür kütüphanelerin ve framework'lerin sayısı oldukça fazla olmakla birlikte bazları çokça tercih edilmektedir. Bu tür kütüphaneleri ve framework'leri kabaca "aşağı seviyeli olanlar" ve yüksek

"yüksek seviyeli olanlar" biçiminde ikiye ayıralım. Tabii aslında yüksek seviyeli kütüphaneler ve framework'ler kendi içlerinde aşağı seviyeli kütüphanelerden faydalananabilmektedir.



Bazı kütüphaneler ve framework'ler yalnızca bazı dillerden kullanılabılırken bazıları farklı programlama dillerinden kullanılabilmektedir. Kütüphane ve framework tercihlerini bu durum da etkilemektedir.

scikit-learn isimli kütüphane Python'da yazılmıştır ve makine öğrenmesi için kullanılan aşağı seviyeli genel bir kütüphanedir. Pek çok kütüphane kendi içerisinde scikit-learn kütüphanesini de kullanmaktadır. scikit-learn açık kaynak kodlu bir kütüphanedir, numpy ve scipy kullanılarak yazılmıştır. Ancak scikit-learn yapay sinir ağlarına yönelik tasarlanmamıştır. Yani scikit-learn içerisinde yapay sinir ağlarını oluşturacak modüller yoktur.

PyTorch Python ve C++'ta yazılmış olan bir yapay sinir ağları ve derin öğrenme ağları kütüphanesidir. Proje facebook tarafından yaşama geçirilmiştir. Bu kütüphane veri bilimcileri arasında belli yoğunluklarda kullanılmaktadır. PyTorch da numpy üzerine oturtulmuştur.

Theano aslında Python programları için yazılmış bir nümerik analiz kütüphanesidir. Ancak yapay sinir ağlarında ve derin öğrenmede aşağı seviyeli bir kütüphane olarak Theano'dan faydalananmaktadır.

TensorFlow Google tarafından geliştirilmiş olan aşağı seviyeli nümerik analiz kütüphanesidir. Makine öğrenmesi, yapay sinir ağları ve derin öğrenme ağları uygulamalarında en çok tercih edilen aşağı seviyeli kütüphane durumundadır. TensorFlow Python dahil olmak üzere pek çok programlama dilinden kullanılabilir. Bu kütüphane de Python ve C++ kullanılarak yazılmıştır.

MxNet ise başka bir aşağı seviyeli yapay sinir ağları ve derin öğrenme kütüphanesidir. Apache grubu tarafından açık kaynak kodlu olarak yazılmıştır.

Yüksek seviyeli yapay sinir ağları ve derin öğrenme kütüphanelerinin açık ara en başında Keras gelmektedir. Keras Python'da yazılmıştır ve oldukça yüksek seviyeli bir kütüphanedir. Ancak Keras arka planda birkaç alçak seviyeli kütüphanesi "backend" olarak kullanabilmektedir. (Default durumda Keras Google'in TensorFlow kütüphanesini kullanmaktadır.) Kursumuzun bu bölümünde yapay sinir ağları ve derin öğrenme uygulamalarında basitliğinden dolayı Keras kullanılacaktır. Keras kütüphanesinin dokümantasyonu <https://keras.io/> sitesinde bulunmaktadır. Keras açık kaynak kodlu bir projedir ve Google Tensorflow ekibi tarafından da desteklenmektedir.

Veri Tablolarının Kullanımı Hazır Hale Getirilmesi

İstatistikte toplanan bilgiler ilk önce veri tabloları biçiminde düzenlenirler. Bir veri tablosu sütunlardan ve satırlardan oluşmaktadır. Bu bakımdan veri tabloları veritabanlarındaki tablolarına da benzemektedir. Veri tablolarındaki sütunlara "özellikler (features)" denilmektedir. Biz kursumuzda veri tablolarındaki msütunlara bazen "özellik" bazen de "sütun" diyeceğiz. Örnek bir veri tablosu şöyle olabilir:

Medeni Durum	Yaş	Eğitim Durumu	Maaş
Evl.	37	Lise	3850
Bekar	40	Üniversite	7360
Evi	28	Lise	2700
...

Burada Medeni Durum, Yaş, Eğitim Durumu ve Maaş tablonun özellikleridir. Satırlar da kayıtları oluşturmaktadır. Bir veri tablosunda sütunlar farklı ölçek türlerine ilişkin olabilmektedir. Örneğin yukarıdaki tabloda Medeni Durum Kategorik (Nominal), Eğitim Durumu Sırasal (Ordinal) ölçeklere ilişkindir. Ancak Yaş ve Maaş sütunları (özellikleri) Oransal (Ratio) ölçüye ilişkindir.

Veri tabloları çeşitli formatlarda bulunabilmektedir. Makine öğrenmesinde en çok karşılaşılan format CSV'dir. CSV (Comma Separated Values) dosyası satırlardan oluşmaktadır. Satırlardaki elemanlar (yani sütun bilgileri) tipik olarak ',' karakteri ile birbirlerinden ayrılmaktadır. (Aslında CSV dosyalarında ayıraçlar bazen ',' dışındaki karakterlerden de oluşturulabilmektedir.) CSV dosyalarında isteğe bağlı olarak sütunların ne anlam ifade ettiğini belirten başlık kısımları da bulunabilmektedir. Bu başlık kısımları bizim için bazen gerekli bazen de gereksiz olabilir. Örneğin bu başlıklar eğer CSV dosyası pandas.read_csv fonksiyonuyla okunuyorsa DataFrame içerisindeki sütun isimleri haline dönüştürülmektedir. Biz kursumuzda veri tablolarının genel olarak CSV formatında olduğunu varsayıcağız.

Alıntı Notu: Aşağıdaki gruplandırma "Data Preparation for Machine Learning - Jason Brownlee" isimli kitaptan alınmıştır ve bölüm içerisinde bu kitaptan faydalанılmıştır. Okuyucunun bu kitabı baştan sona gözden geçirmesini tavsiye ediyoruz.

Verilerin kullanıma hazır hale getirilmesi süreci aslında ayrıntılı bir konudur. Bu nedenle biz kursumuzda önce genel bir açıklama yapıp sonra gerekli yerlerde gerekli yöntemleri ele alacağız. Genel olarak bir veri tablosunun kullanıma hazır hale getirilmesi için beş yöntem grubu kullanılmaktadır. Şimdi bu yöntem grupları hakkında temel bilgileri edinelim.

1) Verilerin Temizlenmesi (Data Cleaning): Veriler içerisinde geçersiz olan değerler söz konusu olabilir. Geçersiz veriler bazen mevcut olmayan (nan/None) veriler biçiminde, Bazen bozulmuş veriler biçiminde, bazen de yinelenen (mükerrer) veriler biçiminde karşımıza çıkabilmektedir. Veri bilimcisinin bu verileri dikkate alıp bunları ortadan kaldırırmaya yönelik teknikleri uygulaması gereklidir. Örneğin bu tür durumlarda geçersiz verilerin bulunduğu satırlar ya da sütunlar tablodan tümden atılabilir ya da geçersiz değerler yerine başka değerler (tipik olarak ortalama değerler ya da mod değerleri) yerleştirilebilir. Bazen aşırı ucta bazı değerler bozucu etkiler de yaratılmaktadır. Bu aşırı uçtaki değerlerden de kurtulmak gerekebilir. Bazı veri tablolarında ise birbirile yüksek korelasyona sahip olan sütunlar (özellikler) da bulunabilmektedir. Bize ek bir bilgi vermeyen bu tür sütunların bazlarının atılması işlenecek verinin miktarını düşürmektedir.

2) Özellik Seçimi (Feature Selection): Özellik seçimi veri tablosundaki sütunların gerekli olanlarının alınıp gereksiz olanlarının atılması ile ilgili bir hazırlık etkinliğidir. Örneğin veri tablosunun bir sütununda kişinin ismi olabilir ve bu ismin kestirim sürecinde hiçbir etkisi olmayıabilir. Bu durumda bu sütunun atılması gerekebilir. Bazen çok sayıda sütunun nispeten öünsüz olanlarının atılması da uygun olabilmektedir. Uygun sütunların seçilmesi süreci sütunun türlerine ve hedefe bağlı olarak da değişebilmektedir.

3) Verilerin Dönüşürlmesi (Data Transformation): Verilerin dönüşürlmesi verilerin türlerini ya da dağılımını değiştirmek için kullanılan yöntemlerdir. Makine öğrenmesinde sıralı ve kategorik veriler doğrudan kullanılamamaktadır. Bunların algoritmala sokulmadan önce sayısal biçimde dönüştürülmesi gereklidir. İşte sıralı ve kategorik verilerin sayısal biçimde dönüştürülmesi en çok uygulanan veri dönüştürmesi yöntemlerindendir. Veri dönüştürmesi sırasında bazen bunun tersi de yapılabilmektedir. Örneğin sayısal verilerin sıralı biçimde dönüştürülmesi de gerekebilmektedir. Bu işlem "ayırık hale getirme (discretization transform)" denilmektedir. Kategorik verilerin sayısal biçimde dönüştürülmesinde en sık uygulanan yöntemlerden biri "one hot encoding" denilen yöntemdir. Bu

yöntemde kategorik veri birden fazla sütunu olan ikili (binary) veri haline dönüştürülmektedir. Bazen veri tablosunun sütunları arasında skala farklılıklar olabilmektedir. Bu skala farklılıklar pek çok algoritmada bozucu etki yaratır. Bunları ortadan kaldırmak için "normalizasyon" ve "standardizasyon" işlemleri yapılmaktadır.

4) Özellik Mühendisliği (Feature Engineering): Özellik mühendisliği var olan sütunlardan yeni sütunlar oluşturma sürecine ilişkin yöntemlere denilmektedir. Örneğin bazen sütunlara bool bir sütun eklenebilir. Bazen sütunlar üzerinde bazı işlemlerin sonucu olarak yeni bir sütun eklenebilir.

5) Boyut İndirgemesi (Dimensionality Reduction): Veri tablosundaki sütunlar aslında çok boyutlu uzaydaki boyutlar olarak düşünülebilirler. Örneğin iki sütunlu verilerdeki satırlar iki boyutlu bir düzlemede nokta belirtirler. Boyut indirgemesi de aslında çok sayıda sütunu temsil edebilen daha az sayıda sütun oluşturulması sürecidir. Boyut indirgemesi için "temel bileşenler analizi (principal components analysis)", "tekil değer ayrıştırması (singular value decomposition)", "doğrusal ayırm analizi (linear discriminant analysis)" gibi yöntemler kullanılmaktadır.

Veri Tablolarının Gereksiz Sütunlardan Arındırılması

Verilerin kullanıma hazır hale getirilmesi sürecinde veri tablolarının gereksiz sütunlardan arındırılması en çok kullanılan "özellik seçimi (feature selection)" yöntemlerinden biridir. Veri tablolarının gereksiz sütunlardan arındırılması birkaç biçimde yapılmaktadır. Eğer bir CSV dosyası söz konusu ise (makine öğrenmesi bağlamında genellikle veriler CSV dosyalarından okunmaktadır) gereksiz sütunlar numpy.loadtxt fonksiyonunda işin başında atılabilir. Örneğin aşağıdaki gibi bir CSV dosyası olsun:

```
Adı Soyadı,Boy,Kilo,Doğum Yeri  
Ali Bulut,182,89,Eskişehir  
Erol Öner,187,82,İzmir  
Ayşe Tan,172,58,Urf  
Rasim Taşcan,168,92,Samsun
```

Dosyanın isminin "test.csv" olduğunu varsayıyalım. Şimdi bu veri tablosunun "Adı Soyadı" ve "Doğum Yeri" sütunlarını gereksiz olduğu gerekçesiyle atmak isteyebiliriz. numpy.loadtxt fonksiyonun usecols parameteresinin dosyadan elde edilecek satırları belirlemek için kullanıldığını anımsayınız. Bu tabloda Boy 1'inci sütunu, Kilo ise 2'inci sütunu belirtmektedir. Okuma işlemi şöyle yapılabilir:

```
import numpy as np  
  
data = np.loadtxt('test.csv', delimiter=',', encoding='utf-8', skiprows=1, usecols=(1, 2))  
print(data)
```

İşlemden şöyle bir çıktı elde ederiz:

```
[[182.  89.]  
 [187.  82.]  
 [172.  58.]  
 [168.  92.]]
```

Tabii tablonun hepsini okuyup söz konusu ikiş sütunu numpy.delete fonksiyonu ile axis=1 parametresini kullanarak da silebilirdik. Ancak bu işlem daha yorucu olurdu:

```
import numpy as np  
  
data = np.loadtxt('test.csv', delimiter=',', encoding='utf-8', skiprows=1, dtype='object')  
data = np.delete(data, [0, 3], axis=1).astype(np.float32)  
print(data)
```

Sayısal olmayan sütunları okurken dtype=object verildiğine dikkat ediniz. Bu durumda satırların hepsi string olarak elde edilmektedir. astype metodu string olan sütunların np.float32 türüne dönüştürülmesi için kullanılmıştır.

CSV dosyalarını okuyup gereksiz sütunları atmak için numpy.loadtxt fonksiyonu yerine pandas kütüphanesindeki read_csv fonksiyonu da kullanılabilir. pandas.read_csv fonksiyonunun numpy.loadtxt fonksiyonu ile benzer işlevselligi sahip olmakla birlikte daha yetenekli olduğunu söyleyebiliriz. Şimdi aynı işlemleri pandas.read_csv fonksiyonu ile yapalım:

```
import pandas as pd

data = pd.read_csv('test.csv', delimiter=',', encoding='utf-8', usecols=(1, 2))
print(data)
```

Kodun çalıştırılması sonucunda şöyle bir çıktı elde edilmiştir:

```
Boy Kilo
0 182 89
1 187 82
2 172 58
3 168 92
```

Burada CSV dosyasındaki başlık isimlerinin sütun isimleri haline getirildiğine dikkat ediniz. Tabii aslında pandas'ta biz yine tüm tabloyu okuduktan sonra da belli sütunları atabiliyoruz:

```
import pandas as pd

data = pd.read_csv('test.csv', delimiter=',', encoding='utf-8')
df = df[['Boy', 'Kilo']]
print(df)
```

Burada önce tüm tabloyu okuduktan sonra dilimleme ile belli sütunları aldığımıza dikkat ediniz. Elde edilen çıktı aynı olacaktır:

```
Boy Kilo
0 182 89
1 187 82
2 172 58
3 168 92
```

Şimdi de müşterilerin kredi kartı ödemelerini yapıp yapmadıkları ile ilgili bilgileri içeren "bank.csv" dosyasından bazı sütunları atalım. Bu dosyayı dosyayı <https://www.kaggle.com/janiobachmann/bank-marketing-dataset/data> adresinden indirebilirsiniz. Dosyanın çalışma dizininde bulunduğu varsayıcağız. Dosyanın genel görünümü aşağıdaki gibidir:

```
age,job,marital,education,default,balance,housing,loan,contact,day,month,duration,campaign,pdays,previous
,poutcome,deposit
59,admin.,married,secondary,no,2343,yes,no,unknown,5,may,1042,1,-1,0,unknown,yes
56,admin.,married,secondary,no,45,no,no,unknown,5,may,1467,1,-1,0,unknown,yes
41,technician,married,secondary,no,1270,yes,no,unknown,5,may,1389,1,-1,0,unknown,yes
55,services,married,secondary,no,2476,yes,no,unknown,5,may,579,1,-1,0,unknown,yes
54,admin.,married,tertiary,no,184,no,no,unknown,5,may,673,2,-1,0,unknown,yes
42,management,single,tertiary,no,0,yes,yes,unknown,5,may,562,2,-1,0,unknown,yes
56,management,married,tertiary,no
....
```

Burada örneğin 8'inci (contact) ve 15'inci (poutcome) sütunları atmak isteyelim. Bunun için pandas.read_csv numpy.loadtxt fonksiyonundan daha kolay bir seçenekir:

```
import pandas as pd

df = pd.read_csv('bank.csv')
df.drop(['contact', 'poutcome'], axis=1, inplace=True)
print(df)
```

Kodun çalıştırılması sonucunda oluşan çıktıyi inceleyiniz:

```
   age      job marital education ... campaign pdays previous deposit
0   59    admin. married secondary ...        1     -1       0     yes
1   56    admin. married secondary ...        1     -1       0     yes
2   41 technician married secondary ...        1     -1       0     yes
3   55   services married secondary ...        1     -1       0     yes
4   54    admin. married tertiary ...        2     -1       0     yes
...
11157  33 blue-collar single primary ...        1     -1       0      no
11158  39   services married secondary ...        4     -1       0      no
11159  32 technician single secondary ...        2     -1       0      no
11160  43 technician married secondary ...        2    172       5      no
11161  34 technician married secondary ...        1     -1       0      no
```

Kategorik (Nominal) ve Sıralı (Ordinal) Verilerin Sayısal Biçime Dönüşürtlmesi

Tipik bir veri tablosunda kişinin cinsiyeti, medeni durumu, yaşadığı ülke gibi kategorik (nominal) ve sıralı (ordinal) sütunlar bulunabilmektedir. Makine öğrenmesinde pek çok yöntem ve algoritma kategorik ve sıralı veriler üzerinde doğrudan işlem yapamamaktadır. Bu nedenle önce kategorik ve sıralı verilerin sayısal biçimde dönüştürülmeleri gereklidir. Ayrıca yapay sinir ağları bağlamında Keras kütüphanesi Pandas kütüphanesindeki DataFrame sınıfını kullanmaz. Yalnızca numpy kütüphanesinin ndarray veri yapısını kullanmaktadır. Yani başka bir deyişle bizim Keras'ı kullanabilelimiz için veri tablosundaki tüm sütunların sayısal biçimde ndarray haline dönüştürülmüş olması gereklidir. Bu nedenle kategorik ve sıralı verilerin sayısal biçimde dönüştürülmesi verileri hazır hale getirme sürecinde kullanılan en önemli veri dönüşümü (data transformation) yöntemlerinden biridir.

Sıralı (Ordinal) Verilerin Sayısal Biçime Dönüşürtlmesi

Sıralı verilerin sayısal biçimde dönüştürülmesi işlemi için yüksek seviyeli kütüphanelerde hazır bir fonksiyon ya da metod bulunmaktadır. Bu işlem en pratik biçimde pandas kütüphanesindeki Series ya da DataFrame sınıflarının replace metotlarıyla gerçekleştirilebilir. Örneğin aşağıdaki gibi bir "test.csv" dosyamız olsun:

```
Cinsiyet,Kilo,Boy,Eğitim
Erkek,85,172,İlkokul
Kadın,72,170,Üniversite
Kadın,65,162,Lise
Erkek,92,183,Lise
Kadın,62,173,İlkokul
Erkek,98,172,Ortaokul
```

Burada "Eğitim" sütunu sıralı ölçüye ilişkindir. Şimdi biz DataFrame sınıfının replace metodunu ile eğitim bilgilerini 0'dan itibaren artan bir sayıyla ifade etmeye çalışalım. Önce dosyayı pandas.read_csv fonksiyonu ile ilk sütunu atarak okuyalım:

```
import pandas as pd
```

```
df = pd.read_csv('test.csv', encoding='utf-8', usecols=[1, 2, 3])
print(df)
```

Elde edilen DataFrame aşağıdaki gibidir:

	Kilo	Boy	Eğitim
0	85	172	İlkokul
1	72	170	Üniversite
2	65	162	Lise
3	92	183	Lise
4	62	173	İlkokul
5	98	172	Ortaokul

Şimdi DataFrame sınıfının replace metodu ile dönüştürmemizi yapalım:

```
replace_dict = {'Eğitim': {'İlkokul': 0, 'Ortaokul': 1, 'Lise': 2, 'Üniversite': 3}}
df.replace(replace_dict, inplace=True)
print(df)
```

Kodun çalıştırılması sonucunda öyle bir çıktı elde edilecektir:

	Kilo	Boy	Eğitim
0	85	172	0
1	72	170	3
2	65	162	2
3	92	183	2
4	62	173	0
5	98	172	1

Artık bu DataFrame nesnesini to_numpy metodu ile ndarray nesnesine dönüştürebiliriz:

```
dataset = df.to_numpy()
print(dataset)
```

```
[[ 85 172  0]
 [ 72 170  3]
 [ 65 162  2]
 [ 92 183  2]
 [ 62 173  0]
 [ 98 172  1]]
```

Aynı işlemi Series nesnesi üzerinde de -biraz daha dolaylı olarak- şöyle yapabiliyoruz:

```
replace_dict = {'İlkokul': 0, 'Ortaokul': 1, 'Lise': 2, 'Üniversite': 3}
df['Eğitim'] = df['Eğitim'].replace(replace_dict)
```

Bu işlemi yalnızca numpy kullanarak yapmak daha zahmetlidir. Çünkü numpy kütüphanesinin Pandas kütüphanesindeki gibi bir replace fonksiyonu yoktur.

```
import numpy as np

data = np.loadtxt('test.csv', delimiter=',', skiprows=1, usecols=[1, 2, 3], encoding='utf-8',
dtype=np.object)

dataset = data[:, [0, 1]].astype(np.float32)
edu = data[:, 2]

replace_dict = {'İlkokul': 0, 'Ortaokul': 1, 'Lise': 2, 'Üniversite': 3}
```

```

v = np.vectorize(replace_dict.get)
dataset = np.insert(dataset, 2, v(edu), axis=1)
print(dataset)

```

Burada önce "test.csv" dosyasını numpy.loadtxt fonksiyonu ile dtype=np.object parametresi ile okuduk. numpy dizilerinin tek bir dtype türü olduğu için fonksiyon tüm değerleri string olarak okudu. Sonra "Kilo" ve "Boy" sütunlarını np.float32 türüne dönüştürerek dataset isimli yeni bir numpy dizisi içerisinde yerleştirdik. Nihayet "Eğitim" sütununu da numpy.vectorize fonksiyonu ile sıralı sayısal bir sütuna dönüştürerek bunu yeni bir sütun biçiminde ekledik. Yukarıdaki kod çalıştırıldığında aşağıdaki gibi bir çıktı elde edilecektir:

```

[[ 85. 172. 0.]
 [ 72. 170. 3.]
 [ 65. 162. 2.]
 [ 92. 183. 2.]
 [ 62. 173. 0.]
 [ 98. 172. 1.]]

```

Tabii numpy içerisinde bu işlemi yapmanın başka çok çeşitli yolları da vardır. Örneğin bu işlem aslında loadtxt fonksiyonu sırasında fonksiyonun converters parametresi kullanılarak da yapılabilir:

```

import numpy as np

replace_dict = {'İlkokul': 0, 'Ortaokul': 1, 'Lise': 2, 'Üniversite': 3}
dataset = np.loadtxt('test.csv', delimiter=',', skiprows=1, usecols=[1, 2, 3], encoding='utf-8',
converters={1: lambda x: float(x), 2: lambda x: float(x), 3: lambda x: replace_dict[x]},
dtype=np.object)
print(dataset)

```

Tabii sizin de gördüğünüz gibi bu tür işlemleri yalnızca numpy kütüphanesi ile yapmak yerine pandas kütüphanesini kullanarak yapmak ve en sonunda DataFrame nesnesini ndarray nesnesine dönüştürmek daha pratik ve kolay bir yöntemdir. Biz kursumuzda bazen doğrudan numpy kütüphanesini kullanırken bazen de Pandas kütüphanesinin sağladığı kolaylıklarını kullanacağız.

Kategorik (Nominal) Verilerin Sayısal Biçime Dönüşürtlürülmesi

Kategorik verilerde her bir kategori için 0'dan itibaren farklı bir tam sayı karşılık düşürülmesi yoluna gidilebilir. Tabii buradaki değerler arasında bir sıralama ilişkisi olmadığı için bu tam sayılar büyüklik küçüklük belirtmeyecek yalnızca farklı kategorileri belirtecektir. Bu tür kategorik alanların farklı tam sayılar biçiminde sayısal olarak ifade edilmesini kolaylaştırmak için numpy ya da pandas kütüphanelerinde hazır bir fonksiyon ya da metod bulunmaktadır. Bu işlem en kolay olarak sklearn (scikit-learn) kütüphanesindeki LabelEncoder sınıfı ile yapılabilir. Bu sınıf sklearn.preprocessing paketi içerisinde yer almaktadır. Dönüşürme için önce LabelEncoder sınıfı türünden bir nesne yaratılır. Örneğin:

```

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()

```

Sonra LabelEncoder sınıfının fit isimli metodu bizden kategorik ölçekteki değerleri alır. Sınıfın transform metodu da dönüştürülecek kategorik değerleri 0'dan itibaren sayısal değerlere dönüştürmektedir. LabelEncoder sınıfının classes_ isimli örnek özniteliği ile fit işlemi sırasında verilen kategorik değerler elde edilebilir. Örneğin:

```

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
le.fit(['mavi', 'kırmızı', 'yeşil', 'beyaz'])
data = ['mavi', 'kırmızı', 'mavi', 'yeşil', 'beyaz', 'yeşil', 'kırmızı']
result = le.transform(data)

```

```
print(data)
print(result)
print(le.classes_)
```

Kodun çıktısı şöyledir:

```
['mavi', 'kırmızı', 'mavi', 'yeşil', 'beyaz', 'yeşil', 'kırmızı']
[2 1 2 3 0 3 1]
['beyaz' 'kırmızı' 'mavi' 'yeşil']
```

Göründüğü gibi fit metodu bizden kategorik ölçek değerlerini almış, transform ise bunları dönüştürmüştür. Biz bu örnekte metotlara Python listeleri verdik. Tabii uygulamada tipik olarak bu metotlara numpy dizileri verilmektedir. Aslında fit ve transform işlemi bir arada fit_transform metoduyla da yapılabilmektedir:

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
data = ['mavi', 'kırmızı', 'mavi', 'yeşil', 'beyaz', 'yeşil', 'kırmızı']
result = le.fit_transform(data)

print(data)
print(result)
print(le.classes_)
```

Burada fit_tranform doğrudan dönüştürülecek verileri bizden almıştır. Metot bu verileri önce fit metoduna sonra da transform metoduna sokmaktadır. Sınıfın fit metodu aynı kategorik değerler birden fazla kez kullanılmışsa onları zaten dikkate almaz. Yukarıdaki koddan elde edilen çıktı öncekiyle aynı olacaktır:

```
['mavi', 'kırmızı', 'mavi', 'yeşil', 'beyaz', 'yeşil', 'kırmızı']
[2 1 2 3 0 3 1]
['beyaz' 'kırmızı' 'mavi' 'yeşil']
```

Burada önemli bir noktaya dikkatinizi çekmek istiyoruz. Bu sınıf ile yapılan dönüştürmede hangi kategorik değerin hangi sayıyla ifade edileceğini biz belirleyememekteyiz. fit metodu verilen kategorik değerleri numpy.unique fonksiyonuna soktuktan sonra onlara 0'dan itibaren değerler karşılık düşürmektedir. numpy.unique fonksiyonun yinelenenleri attıktan sonra buna ek olarak sıraya dizme işlemi yaptığı da anımsayınız. Yani bu durumda aslında bizim fit ya da fit_transform metodlarına verdığımız kategoriler alfabetik olarak sıraya dizildikten sonra onlara numaralar atanmış olmaktadır. Gerçekten de yukarıdaki örnekte beyazın 0, kırmızının 1 olduğunu dikkat ediniz. Buradan LabelEncoder sınıfının eğer kategorik isimler sıralı değilse sıralı veriler için kullanılamayacağı sonucunu çıkarabiliriz.

LabelEncoder sınıfının fit ve fit_transform metodlarına aslında biz kategorik verileri yazışal biçimde vermek zorunda değiliz. Kategorik veriler zaten sayısal biçimde de bulunuyor olabilir. Örneğin kategorik verilerin şehirlerin plaka numaralarından oluştuğunu düşünelim:

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
data = [34, 26, 37, 34, 6, 77, 10, 16, 35, 34, 6]
result = le.fit_transform(data)

print(data)
print(result)
print(le.classes_)
```

Kodun çıktısı şöyle olacaktır:

```
[34, 26, 37, 34, 6, 77, 10, 16, 35, 34, 6]
[4 3 6 4 0 7 1 2 5 4 0]
[ 6 10 16 26 34 35 37 77]
```

Şimdi aynı işlemi bir CSV dosyasından hareketle yapalım. Örneğimizdeki "test.csv" dosyası şöyle olsun:

```
Cinsiyet,Kilo,Boy,Şehir
Erkek,85,172,Eskişehir
Kadın,72,170,İzmir
Kadın,65,162,İstanbul
Erkek,92,183,İstanbul
Kadın,62,173,Ankara
Erkek,98,172,İzmir
```

Burada ilk ve son sütundaki kategorik verileri sayısal biçimde şöyle dönüştürebiliriz:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

df = pd.read_csv('test.csv')
print(df)

le = LabelEncoder()
df['Cinsiyet'] = le.fit_transform(df['Cinsiyet'])
print(le.classes_)
df['Şehir'] = le.fit_transform(df['Şehir'])
print(le.classes_)
print(df)
dataset = df.to_numpy()
print(dataset)
```

Kodun çalıştırılması sonucunda şöyle bir çıktı elde edilmiştir, inceleyiniz:

```
Cinsiyet  Kilo   Boy      Şehir
0    Erkek    85  172  Eskişehir
1    Kadın    72  170      İzmir
2    Kadın    65  162  İstanbul
3    Erkek    92  183  İstanbul
4    Kadın    62  173  Ankara
5    Erkek    98  172      İzmir
['Erkek' 'Kadın']
['Ankara' 'Eskişehir' 'İstanbul' 'İzmir']
   Cinsiyet  Kilo   Boy  Şehir
0          0    85  172     1
1          1    72  170     3
2          1    65  162     2
3          0    92  183     2
4          1    62  173     0
5          0    98  172     3
[[ 0  85 172  1]
 [ 1  72 170  3]
 [ 1  65 162  2]
 [ 0  92 183  2]
 [ 1  62 173  0]
 [ 0  98 172  3]]
```

sklearn.preprocessing modülünde LabelEncoder sınıfıyla benzer işlemi yapan OrdinalEncoder isimli bir sınıf da vardır. OrdinalEncoder aslında LabelEncoder sınıfının birden çok sütunu aynı anda alarak işlem yapan bir biçimi gibidir. OrdinalEncoder sınıfında oluşturulan kategoriler sınıfın categories_ örneğin özniteliği ile bize bir numpy listesi olarak verilmektedir. Örneğin:

```
from sklearn.preprocessing import OrdinalEncoder

data = [['mavi', 'erkek'], ['kırmızı', 'kadın'], ['siyah', 'kadın'], ['mavi', 'kadın'],
        ['beyaz', 'kadın'], ['yeşil', 'kadın']]

oe = OrdinalEncoder()
result = oe.fit_transform(data)
print(result)
print(oe.categories_)
```

Şimdi önceki örnekteki "test.csv" dosyasını OrdinalEncoder sınıfıyla dönüştürelim:

```
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder

df = pd.read_csv('test.csv')
print(df)

oe = OrdinalEncoder()
df[['Cinsiyet', 'Şehir']] = oe.fit_transform(df[['Cinsiyet', 'Şehir']])
print(df)
dataset = df.to_numpy()
print(dataset)
```

Kodun çıktısı şöyle olacaktır:

```
Cinsiyet   Kilo   Boy      Şehir
0    Erkek     85  172  Eskişehir
1    Kadın     72  170      İzmir
2    Kadın     65  162  İstanbul
3    Erkek     92  183  İstanbul
4    Kadın     62  173   Ankara
5    Erkek     98  172  İzmir
Cinsiyet   Kilo   Boy      Şehir
0       0.0     85  172     1.0
1       1.0     72  170     3.0
2       1.0     65  162     2.0
3       0.0     92  183     2.0
4       1.0     62  173     0.0
5       0.0     98  172     3.0
[[ 0.  85. 172.  1.]
 [ 1.  72. 170.  3.]
 [ 1.  65. 162.  2.]
 [ 0.  92. 183.  2.]
 [ 1.  62. 173.  0.]
 [ 0.  98. 172.  3.]]
```

OrdinalEncoder sınıfı da -ismi yanlış bir çağrıım uyandırsa da- aslında sıralı veriler için kullanılamaz. Çünkü burada yine LabelEncoder sınıfının yaptığı gibi numpy.unique işlemi uygulanmaktadır. Bu sınıf da alfabetik ya da sayısal olarak en küçük kategoriden başlayarak sayısal atamaları yapmaktadır.

Aslında kategorik veriler çoğu kez buradaki gibi sayısal biçimde dönüştürülmemektedir. Kategorik verilerin sayısal biçimde dönüştürülmesinde sıkılıkla "one hot encoding" denilen teknik kullanılmaktadır. İzleyen bölümde "one hot encoding" denilen tekniği ele alacağız.

One Hot Encoding Dönüşürmesi

Kategorik verilerin 0'dan itibaren tam sayılarla sayısal biçimde dönüştürülmesi pek çok algoritma için bir sorundur. Çünkü sayıların arasında büyüklik küçüklik ilişkisi vardır ve bu büyüklik küçüklik ilişkisi pek çok algoritmada yanlış çgrenmelere yol açabilmektedir. Örneğin biz LabelEncoder ya da OrdinalEncoder sınıflarıyla üç kategorik renk değerine Blue = 0, Green = 1, Yellow = 2 değerlerini atamış olalım. Buradaki Yellow > Green > Blue gibi bir ilişkinin ya da Blue + Green = Yellow gibi bir ilişkinin bizim için bir anlamı yoktur. Ancak pek çok algoritma bu üç renk böyle kodlandığında onların kategorik değil sıralı ölçüye sahip olduğunu sanmaktadır. İşte kategorik değerlerin sanki onlar sıralı ölçüye ilişkinmiş gibi değerlendirilmesini engellemek için "one hot encoding" denilen bir yöntem kullanılmaktadır. İngilizce "One hot" bir grup bitten yalnızca birinin 1 diğerlerinin 0 olduğunu anlatmak için kullanılan bir deyimdir.

One hot encoding dönüşürmesinde kategorik bilgi ikili sistemde kategori sayısı kadar sütunla ifade edilmektedir. Bu yöntemde n kategoriden birini belirten bir veri n tane sütunla ifade edilir. Verinin ilişkin olduğu kategoriye ilişkin sütun bilgisi 1 diğer sütun bilgileri 0 yapılır. Örneğin aşağıdaki gibi bir veri tablosu söz konusu olsun:

Tepki Süresi	Renk
1.2	Kırmızı
2.3	Mavi
0.7	Kırmızı
1.2	Yeşil
3.2	Mavi

Bu tabloda Renk "Kırmızı, Yeşil ve Mavi" olabilen üç kategoriden oluşan kategorik bir ölçüye ilişkindir. Bu tablodaki Renk sütununun "one hot encoding" tekniği ile sayısallaştırılması sonucunda şöyle bir tablo elde edilir:

Tepki Süresi	Kırmızı	Yeşil	Mavi
1.2	1	0	0
2.3	0	0	1
0.7	1	0	0
1.2	0	1	0
3.2	0	0	1

Burada Renk sütunun yerine onun kategori sayısı kadar sütun eklendiğine dikkat ediniz. Renk hangi renkse yalnızca o renin sütun değeri 1, diğer renklerin sütun değerleri 0 yapılmıştır. Bu biçimdeki bir kodlamayla renkler arasında bir sıra ilişkisinin ortadan kaldırılmaya çalışıldığına dikkat ediniz. Eğer bizim kategorik verimiz 10 tane farklı kategoriden oluşmuş olsaydı biz de 10 kategori için 10 tane sütun oluşturacaktık.

Şimdi de "one hot encoding" işleminin nasıl yapılacağını görelim. scikit-learn kütüphanesindeki sklearn.preprocessing modülünde "one hot encoding" işlemini yapan OneHotEncoder isimli hazır bir sınıf vardır. Ancak bu sınıf kategorik verilerin önce sayısal biçimde dönüştürülmüş olmasını ister. Bu nedenle biz bu sınıfı kullanmadan önce kategorik verileri LabelEncoder ya da OrdinalEncoder sınıfı ile sayısal biçimde dönüştürmiş olmalıyız. OneHotEncoder sınıfı şöyle kullanılmaktadır:

- 1) Önce kategorik sütunun LabelEncoder ya da OrdinalEncoder sınıfları kullanılarak sayısal biçimde dönüştürülmesi gereklidir. Örneğin yukarıda verdigimiz veri tablosunun aşağıdaki gibi bir "test.csv" dosyası içerisinde bulunduğuunu varsayıyalım:

Tepki Süresi,Renk
1.2,Kırmızı
2.3,Mavi
0.7,Kırmızı
1.2,Yeşil
3.2,Mavi

Şimdi bu dosyayı okuyup Renk sütununu LabelEncoder sınıfı ile sayısal biçimde dönüştürelim:

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder

df = pd.read_csv('test.csv')
print(df)

le = LabelEncoder()
df['Renk'] = le.fit_transform(df['Renk'])
print(df)
dataset = df.to_numpy()
print(dataset)
```

Buradan şöyle bir çıktı elde edilecektir:

```
   Tepki Süresi      Renk
0        1.2    Kırmızı
1        2.3      Mavi
2        0.7    Kırmızı
3        1.2      Yeşil
4        3.2      Mavi
   Tepki Süresi  Renk
0        1.2      0
1        2.3      1
2        0.7      0
3        1.2      2
4        3.2      1
[[1.2 0. ]
 [2.3 1. ]
 [0.7 0. ]
 [1.2 2. ]
 [3.2 1. ]]
```

2) sklearn.preprocessing modülü içerisindeki OneHotEncoder sınıfı türünden bir nesne yaratılır. Bu nesnenin fit_transform metoduna sayısallaştırılmış kategorik değerler parametre olarak verilir. Ancak fit_transform metodu ilgili kategorik bilgiyi bir sütun vektörü biçiminde istemektedir. Bu nedenle önce onun reshape işlemiyle sütun vektörüne dönüştürülmesi gereklidir:

```
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(sparse=False)
ohe_dataset = ohe.fit_transform(dataset[:, 1].reshape(-1, 1))
print(ohe_dataset)
```

Burada fit_transform işleminde ilgili sayısallaştırılmış kategorik verilerin sütun vektörüne dönüştürüldüğüne dikkat ediniz. Bu işleminden şöyle bir çıktı elde edilmiştir:

```
[[1. 0. 0.]
 [0. 1. 0.]]
```

```
[1. 0. 0.]  
[0. 0. 1.]  
[0. 1. 0.]]
```

Tabii aslında fit ve transform işlemlerini ayrı ayrı da yapabiliyoruz:

```
from sklearn.preprocessing import OneHotEncoder  
  
ohe = OneHotEncoder(sparse=False)  
ohe.fit(dataset[:, 1].reshape(-1, 1))  
ohe_dataset = ohe.transform(dataset[:, 1].reshape(-1, 1))  
print(ohe_dataset)
```

Bu işlemlerin ayrı ayrı yapılması size anlamsız gelebilir. Ancak aslında fir işleminde biz "one hot encoding" işlemi için gereken kategorileri belirtmekteyiz. transform ise gerçek dönüştürmeyi yapmaktadır. Yani örneğin biz bir kere fit işlemi yapıp çok defa transform işlemi yapabiliyoruz. fit_transform metodu aslında önce fit sonra transform işlemini yapmaktadır.

3) Nihayet one hot encoding işlemi sonucunda elde edilen matrisin gerçek veri matrisine yerleştirilmesi gerekmektedir. Bunun için tabii önce gerçek veri matrisindeki sayısallaştırılmış kategorik sütun silinmelidir. İşin bu kısmını pandasın DataFrame üzerinde yapabiliyoruz:

```
df.drop(['Renk'], axis=1, inplace=True)  
df[['Kırmızı', 'Yeşil', 'Mavi']] = ohe_dataset  
print(df)
```

Bu işlemden Şöyleden bir çıktı elde edilecektir

	Tepki Süresi	Kırmızı	Yeşil	Mavi
0	1.2	1.0	0.0	0.0
1	2.3	0.0	1.0	0.0
2	0.7	1.0	0.0	0.0
3	1.2	0.0	0.0	1.0
4	3.2	0.0	1.0	0.0

Ya da işin bu kısmını hiç pandas kullanmadan numpy dizisi üzerinde de yapabiliyoruz:

```
dataset = np.delete(dataset, 1, axis=1)  
ohe_dataset = np.append(dataset, ohe_dataset, axis=1)  
print(ohe_dataset)
```

Şimdi bu adımları tek bir kodla birebirleştirelim:

```
import pandas as pd  
from sklearn.preprocessing import LabelEncoder  
  
df = pd.read_csv('test.csv')  
print(df)  
  
le = LabelEncoder()  
df['Renk'] = le.fit_transform(df['Renk'])  
  
from sklearn.preprocessing import OneHotEncoder  
  
ohe = OneHotEncoder(sparse=False)  
ohe_dataset = ohe.fit_transform(df['Renk'].to_numpy().reshape(-1, 1))  
df.drop(['Renk'], axis=1, inplace=True)
```

```
df[['Kırmızı', 'Yeşil', 'Mavi']] = ohe_dataset
print(df)
```

Kod çalıştırıldığında şöyle bir çıktı elde edilecektir:

	Tepki Süresi	Renk
0	1.2	Kırmızı
1	2.3	Mavi
2	0.7	Kırmızı
3	1.2	Yeşil
4	3.2	Mavi

	Tepki Süresi	Kırmızı	Yeşil	Mavi
0	1.2	1.0	0.0	0.0
1	2.3	0.0	1.0	0.0
2	0.7	1.0	0.0	0.0
3	1.2	0.0	0.0	1.0
4	3.2	0.0	1.0	0.0

"One hot encoding" işlemi uygulamanın diğer bir yolu da tensorflow.keras.utils modülündeki `to_categorical` fonksiyonunu kullanmaktır. Bu fonksiyon 0'dan başlayan ardışılı tamsayıların bulunduğu kategorik ndarray nesnesini parametre olarak alır ve geri dönüş değeri olarak da "one hot encoding" biçiminde dönüştürülmüş ndarray matrisi verir. Biz de bu matrisi uygun sütunlarla yer değiştirebiliriz. Örneğin:

```
>>> a = np.array([0, 0, 2, 2, 3, 1, 2, 1, 1])
>>> from tensorflow.keras.utils import to_categorical
>>> b = to_categorical(a)
>>> b
array([[1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.]], dtype=float32)
```

Biz de bu işlemden elde ettiğimiz "one hot encoding" ndarray nesnesini orijinal nesnenin sütunlarıyla değiştirebiliriz. Şimdi de yukarıdaki "car.csv" örneğini biz `to_categorical` fonksiyonuyla one hot encoding yapmaya çalışalım.

```
import numpy as np

dataset = np.loadtxt('car.csv', skiprows=1, delimiter=',', dtype='object', encoding='UTF-8')

from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
dataset[:, 1] = le.fit_transform(dataset[:,1])
dataset = np.array(dataset, dtype='float32')

from tensorflow.keras.utils import to_categorical

ohe = to_categorical(dataset[:,1])
dataset = np.insert(dataset, 1, ohe.transpose(), axis=1)
dataset = np.delete(dataset, 4, axis=1)
```

Tabii aslında one-hot-encoding yapan bir fonksiyon yazmak da oldukça kolay. Örneğin:

```

def one_hot_encoder(dataset, column):
    ncategory = np.max(dataset[:, column]) + 1
    ohe = np.zeros((dataset.shape[0], int(ncategory)), dtype=np.float32)
    for index, category in enumerate(dataset[:, column]):
        ohe[index, int(category)] = 1

    dataset = np.delete(dataset, column, axis=1)
    dataset = np.hstack((dataset, ohe))

    return dataset

```

Bu fonksiyon bizden kategori sütunu sayısal hale dönüştürülmüş olan numpy dizisini ve one hot encoding yapılacak sütun numarasını alır. Bize one hot encoding yapılmış yeni matrisi geri dönüş değeri olarak verir. Örneğin:

```

import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder

def one_hot_encoder(dataset, column):
    ncategory = np.max(dataset[:, column]) + 1
    ohe = np.zeros((dataset.shape[0], int(ncategory)), dtype=np.float32)
    for index, category in enumerate(dataset[:, column]):
        ohe[index, int(category)] = 1

    dataset = np.delete(dataset, column, axis=1)
    dataset = np.hstack((dataset, ohe))

    return dataset

df = pd.read_csv('test.csv')
print(df)

le = LabelEncoder()
df['Renk'] = le.fit_transform(df['Renk'])
print(df)

dataset = one_hot_encoder(df.to_numpy(), 1)
print(dataset)

```

Kod çalıştırıldığında şöyle bir çıktı elde edilecektir:

	Tepki Süresi	Renk
0	1.2	Kırmızı
1	2.3	Mavi
2	0.7	Kırmızı
3	1.2	Yeşil
4	3.2	Mavi

	Tepki Süresi	Renk
0	1.2	0
1	2.3	1
2	0.7	0
3	1.2	2
4	3.2	1


```

[[1.2 1.  0.  0. ]
 [2.3 0.  1.  0. ]
 [0.7 1.  0.  0. ]
 [1.2 0.  0.  1. ]
 [3.2 0.  1.  0. ]]

```

One hot encoding dönüştürmesi kategorik veriler için iyi sonuçlar doğursa da bazı dezavantajlara da sahiptir. Örneğin veri tablosundaki bir sütunun kategori sayısı çok fazla olabilir. Bu durumda one hot encoding çok fazla sütunun tabloya eklenmesine yol açar. Bu tür durumlarda one hot encoding yerine başka bir yöntem (örneğin "binary encoding") tercih edilebilir ya da one hot encoding sonrasında "boyutsal özellik indirgemesi (dimensionality feature reduction)" uygulanabilir. (Boyutsal özellik indirgemesi kursumuzda ileride ele alınmaktadır.) One hot encoding işlemi için tüm kategorilerin neler olduğunu işin başında bilinmeli olması gerekmektedir. Halbuki pek çok durumda toplam kategorilerin sayısı işin başında bilinmez. Bu tür durumlarda hash tabloları oluşturulabilir ve bu tablolara göre kodlama yapılabilir. Kategori sayısının çok fazla olduğu durumlarda sıkılıkla uygulanan yöntemlerden bir diğeri de "binary encoding" yöntemidir.

Binary Encoding Yöntemi

Binary Encoding yöntemi kategorik değerlerin sayısı n olmak üzere $\log_2 n$ tane sütunla ifade edilmesini sağlamaktadır. Bu yöntemde sanki toplamda n tane farklı kategorik değer k tane ($k = \log_2 n$) sütun ile ikilik sistemde kodlanmaktadır. Örneğin toplamda 64 farklı kategorik değer söz konusu olsun. Bu durumda bu kategorik değerler $\log_2 64 = 6$ tane sütun ile ikilik sistemde kodlanabilir. Örneğin:

Sütun-1	Sütun-2	Sütun-3	Sütun-4	Sütun-5	Sütun-6	Kategori
0	0	0	0	0	0	C1
0	0	0	0	0	1	C2
...
1	1	1	1	1	0	C63
1	1	1	1	1	1	C64

Binary Encoding işlemi scikit-learn kütüphanesindeki `category_encoders` modülündeki `BinaryEncoder` sınıfı ile gerçekleştirilebilmektedir. Ancak `category_encoders` modülü `sklearn` paketleri içerisinde değildir. Onun ayrıca install edilmesi gerekmektedir:

```
pip install category_encoders
```

`BinaryEncoder` sınıfı `OneHotEncoder` sınıfında olduğu gibi kategori sayısı kadar sütun oluşturmaktadır. Bu nedenle elde edilen pek çok sütun tamamen sıfırlardan oluşabilir. `BinaryEncoder` sınıfı bizden `pandas DataFrame` nesnesini alıp bize yine `DataFrame` de verebilmektedir. Örneğin:

```
import pandas as pd
from category_encoders.binary import BinaryEncoder

df = pd.read_csv('test.csv')
print(df)

be = BinaryEncoder()
be.fit(df['Renk'])
be_df = be.fit_transform(df['Renk'])
print(be_df)
```

Kodun çalıştırılması sonucunda şöyle bir çıktı elde edilecektir:

	Tepki Süresi	Renk	
0	1.2	Kırmızı	
1	2.3	Mavi	
2	0.7	Kırmızı	
3	1.2	Yeşil	
4	3.2	Mavi	
	Renk_0	Renk_1	Renk_2
0	0	0	1

```

1      0      1      0
2      0      0      1
3      0      1      1
4      0      1      0

```

Şimdi DataFrame içerisindeki 'Renk' sütununu silelim, elde edilen sütunlardan sıfır olanları atalım ve iki DataFrame nesnesini birlestirelim:

```

df.drop(['Renk'], axis=1, inplace=True)
be_df = be_df.loc[:, (be_df != 0).any(axis=0)]

result_df = pd.concat([df, be_df], axis=1)
print(result_df)

```

Şöyledir bir çıktı elde edilecektir:

	Tepki Süresi	Renk_1	Renk_2
0	1.2	0	1
1	2.3	1	0
2	0.7	0	1
3	1.2	1	1
4	3.2	1	0

Denetimli (Supervised) Yapay Sinir Ağları ve Veri Kümeleri

Daha önceden makine öğrenmesini ve yapay sinir ağlarını denetimli (supervised), denetimsiz (unsupervised) ve pekiştirmeli (reinforcement) olmak üzere üçe ayırmıştık. Denetimli (supervised) yapay sinir ağlarında ağıın mimarisi oluşturulduktan sonra gerçek verilerle ağıın etğitilmesi (training) gerekmektedir. Bu eğitim sırasında nöronların w katsayıları ayarlanmaktadır. İşte bu eğitim işleminde kullanılmak üzere elimizde gerçek örneklerin bulunması gereklidir. Eğitimde kullanılan gerçek örneklem kümesine "eğitim veri kümesi (traning data set)" denilmektedir. Ağ eğitildikten sonra ağıın test edilmesi için de gerçek verilerden oluşan bir veri kümesine gereksinim duyulmaktadır. Buna da "test veri kümesi (test data set)" denilmektedir. O halde bizim gerçek verileri toplayıp bu verileri "eğitim veri kümesi" ve "test veri kümesi" olmak üzere ikiye ayırmamız gereklidir. Eğitimde kullanılan veri kümesinin aynı zamanda test işleminde kullanılması yanlış bir tekniktir. Çünkü ağ eğitilirken öğrenilen şeyler test edilirken kullanılmamalıdır. (Bu durumu şuna benzetebiliriz: Öğretmen derste birtakım sorular çözüp sınavda da aynısını sorarsa bu sınav gerçek durumu kestirmekte başarısız olabilir.)

Pekiye eğitim veri kümesi ve test veri kümesi arasındaki oran nasıl olmalıdır? Yani topladığımız verilerin yüzde kaç eğitim veri kümesi için yüzde kaç test veri kümesi için kullanılmalıdır? İşte yapay sinir ağlarında ve derin öğrenme ağlarında aslında kesin kurallar yoktur. Pek çok seçim istege bağlıdır. Örneğin uygulamalarda tipik olarak verilerin %80'i eğitim veri kümesi için %20'si ise test veri kümesi için ayrılmaktadır. Ancak veri kümesi çok büyüğse test kümesinin oranı düşürülebilir.

Yukarıda ağıın eğitilmesi ve test edilmesi için verilerin bir biçimde elde edilmiş olması gerektiğini söyledik. Verilerin elde edilmesi ayrı bir süreci oluşturmaktadır. Kursumuz bu verilerin elde edilmesi süreci ile ilgili değildir. Bu nedenle biz bu kursta vereceğimiz örneklerde zaten elde edilmiş olan hazır veriler üzerinde işlemlerimizi yapacağız.

Elde edilmiş veriler "eğitim veri kümesi" ve "test veri kümesi" biçiminde ikiye ayrıldıktan sonra ayrıca bunların da "girdi veri kümesi" ve "çıktı veri kümesi" biçiminde ayrıştırılması gereklidir. Örneğin bir banka müşterisinin 13 özelliğine bakılarak o müşterinin 3 ay içerisinde bankayı terk edip etmeyeceğinin kestirilmeye çalışıldığını düşünelim. İşte bizim bu kestirimi yapabilmemiz için 13 özelliği olan müşterilerin bankayı 3 ay içerisinde terk edip etmediğine yönelik -bir biçimde elde edilmiş olan- bir veri tablomuzun olması gereklidir. Bu veri tablosunda girdi kümesi 13 özellikten, çıktı kümesi ise bankayı tek edip etmediğine ilişkin (örneğim 0 ya da 1) bir özellikten oluşmalıdır.

Şimdi verilerin bu biçimde ayrıştırılmasına örnek verelim. Örneğimizde "pima-indians-diabetes" isimli bir veri tablosunu kullanacağız. "pima-indians-diabetes" veri tablosu bir kişinin 8 özelliğine bakarak onun şeker hastası olup olmadığını kestirilmesi için hazırlanmış gerçek bir veri tablosudur. Tablodaki 8 sütunun anlamları şöyledir:

```
# 1. Number of times pregnant  
# 2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test  
# 3. Diastolic blood pressure (mm Hg)  
# 4. Triceps skin fold thickness (mm)  
# 5. 2-Hour serum insulin (mu U/ml)  
# 6. Body mass index (weight in kg/(height in m)^2)  
# 7. Diabetes pedigree function  
# 8. Age (years)  
# 9. Class variable (0 or 1)
```

Bu veri tablosu <https://www.kaggle.com/uciml/pima-indians-diabetes-database> adresinden .csv dosyası biçiminde indirilebilir. Dosyanın görüntüsü şöyledir:

```
Pregnancies,Glucose,BloodPressure,SkinThickness,Insulin,BMI,DiabetesPedigreeFunction,Age,Outcome  
6,148,72,35,0,33.6,0.627,50,1  
1,85,66,29,0,26.6,0.351,31,0  
8,183,64,0,0,23.3,0.672,32,1  
1,89,66,23,94,28.1,0.167,21,0  
0,137,40,35,168,43.1,2.288,33,1  
5,116,74,0,0,25.6,0.201,30,0  
3,78,50,32,88,31,0.248,26,1  
10,115,0,0,0,35.3,0.134,29,0  
2,197,70,45,543,30.5,0.158,53,1  
8,125,96,0,0,0,0.232,54,1  
4,110,92,0,0,37.6,0.191,30,0  
10,168,74,0,0,38,0.537,34,1  
10,139,80,0,0,27.1,1.441,57,0  
1,189,60,23,846,30.1,0.398,59,1  
...  
...
```

Bu tablo gerçek veriler kullanılarak oluşturulmuştur. Tablodaki her satır farklı bir kişinin bilgisini içermektedir. Her satırın ilk 8 sütunu kişinin çeşitli özelliklerini 9'uncu son sütun ise o kişinin şeker hasatası olup olmadığını belirtmektedir. Bu son sütundaki değerin 1 olması kişinin şeker hastası olduğunu, 0 olması ise kişinin şeker hastası olmadığını göstermektedir. Tabii bizim aslında bu bölümdeki amacımız yapay sinir ağlarını kullanarak yeni bir kişinin bu sekiz özelliğine bakarak onun belli güvenilirlikle şeker hastası olup olmadığını anlamaktır.

Bu veri tablosunun ayrıştırılmasını numpy kullanarak şöyle yapılabılırız:

```
import numpy as np  
  
dataset = np.loadtxt('diabetes.csv', dtype='float32', skiprows=1, delimiter=',')  
  
dataset_x = dataset[:, :8]  
dataset_y = dataset[:, 8]  
  
tzone = int(len(dataset) * 0.80)  
  
training_dataset_x = dataset_x[:tzone, :]  
training_dataset_y = dataset_y[:tzone]  
  
test_dataset_x = dataset_x[tzone:, :]  
test_dataset_y = dataset_y[tzone:]
```

Burada önce dosya okunmuştur. Veri kümesi içerisinde kategorik bir sütunun olmadığına dikkat ediniz. Sonra veri kümesi girdi ve çıktı biçiminde (`dataset_x`, `dataset_y`) ayrıtırılmış sonra bu kümelerde de kendi aralarında %80 noktasından eğitim ve test veri kümesi biçiminde (`training_dataset_x`, `training_datset_y`) yeniden ayrıtırılmıştır.

Aslında yukarıdaki manuel işlem scikit-learn kütüphanesindeki tek bir fonksiyonla da yapılabilmektedir. Bu işlemi yapan `sklearn.model_selection` modülündeki `train_test_split` isimli fonksiyonun parametrik yapısı şöyledir:

```
sklearn.model_selection.train_test_split(*arrays, **options)
```

Fonksiyona biz `dataset_x` ve `dataset_y` vektörlerini ayrı parametreler biçiminde veririz. Fonksiyon da bize `training_dataset_x`, `test_dataset_x`, `training_dataset_y` ve `test_datatset_y` biçiminde dörtlü bir demet verir. Fonksiyonun `test_size` isimli parametresi test kümесinin yüzde kaç olacağını bizden istemektedir. Bu parametre 0 ile 1 arasında float bir değer biçiminde girilmelidir. Örneğin:

```
import numpy as np

dataset = np.loadtxt('diabetes.csv', dtype='float32', skiprows=1, delimiter=',')

dataset_x = dataset[:, :8]
dataset_y = dataset[:, 8]

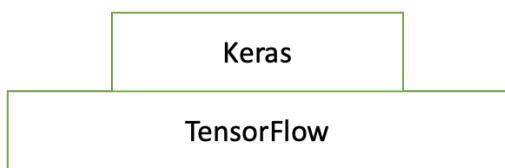
from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_datatset_y =
train_test_split(dataset_x, dataset_y, test_size=0.2)
```

Göründüğü gibi `train_test_split` isimli fonksiyon bizden girdi ve çıktı vektörünü alıp onu istediğimiz oranda parçalara ayırmaktadır. Yukarıda da belirttiğimiz gibi buradaki `test_size` isimli argümanı test verilerinin yüzdesini belirtmektedir. Bu yüzde 0 ile 1 arasında float bir sayı biçiminde girilmelidir. Yukarıdaki örnekte `test_size=0.2` girişi eğitim veri kümесinin %80, test veri kümесinin %20 olacağı anlamına gelmektedir. Eğer fonksiyonda `test_size` parametresi belirtilmemezse default olarak 0.25 alınmaktadır.

Keras Kütüphanesinin Kurulması

Önceden de belirttiğimiz gibi Keras yüksek seviyeli bir yapay sinir ağı kütüphanesidir. Keras 2.3 sürümüne kadar birden fazla arkayı (backend) destekliyordu (TensorFlow, Microsoft Cognitive Toolkit, Theano, PlaidML). Ancak 2.4 sürümüyle birlikte artık Keras arkayı olarak yalnızca TensorFlow kütüphanesini kullanmaktadır. Bu nedenle Keras kütüphanesi yüklenmeden önce TensorFlow yüklenmiş olmalıdır.



Keras yalnızca Python kullanılarak, TensorFlow ise C++ ve Python kullanılarak yazılmıştır. Bu iki kütüphanenin yüklenmesi pip komutlarıyla şöyle yapılabilir:

```
pip install tensorflow
pip install keras
```

Tabii yükleme işlemini Anaconda Navigator sekmelerinden ya da PyCharm menülerinden görsel olarak da yapabilirsiniz. Kurulumuzun verildiği sırada Keras'in son sürümü 2.4.0, TensorFlow'un ise 2.5.0'dır. Ancak burada dikkat edilmesi gereken bir nokta şudur: Maalesef Tensorflow kütüphanesini ile kullanılabilcek Python sürümleri farklıdır. Eğer Python sürümünüz 3.8 ise sizin TensorFlow 2.2 ya da daha sonraki bir versiyonu yüklemeniz gereklidir. Eğer

Python versiyonunuz 3.9 ise sizin Tensorflow 2.5'i yüklemeniz gereklidir. Daha eski Python sürümleri için istediğiniz TensorFlow sürümünü yükleyebilirsiniz.

Tensorflow 2'li versiyonlarla birlikte Keras kütüphanesini bünyesine dahil etmiştir. Yani artık Keras kütüphanesi Tensorflow içerisinde tensorflow.keras paketi biçiminde bulunmaktadır. Ancak Keras ayrı bir kütüphane gibi de şimdilik kendini devam ettirmektedir. Fakat maalesef Keras'ın ayrı bir kütüphane biçiminde kurulumu sırasında TensorFlow kütüphanesi ile versiyon uyşmazlığı yaşanabiliyor. Bu nedenle Keras'ı eğer ayrı bir kütüphane olarak kuracaksanız kursun yapıldığı tarihte TensorFlow kütüphanesinin 2.2 versiyonunu yüklemelisiniz. Bir de artık bu kursla birlikte Keras kütüphanesinin Tensorflow bünyesine katılmış biçimini kullanacağımız. Halbuki önceki kurslarda ayrı bir biçimde kurulan Keras kütüphanesini kullanıyorduk.

Pekiyi bağımsız Keras kurulumu ile Tensorflow içerisindeki Keras kurulumu arasında bir fark var mıdır? Aslında Keras'ın ileri sürümleri de zaten TensorFlow içerisindeki Keras ile uyumlu hale getirilmektedir. Ancak TensorFlow içerisindeki Keras, TensorFlow kütüphanesine yönelik bazı aşağı seviyeli işlemlerinin yapılmasına da izin verdiği için bu durum Keras ile TensorFlow kütüphanelerini beraber kullanmak isteyen programcılar için bir avantaj oluşturabiliyor. Yukarıda da belirttiğimiz gibi biz kursumuzla birlikte artık TensorFlow içerisindeki Keras'ı kullanacağız.

Keras'ta Yapay Sinir Ağlarının Oluşturulması

Keras yüksek seviyeli bir kütüphanedir. Dolayısıyla işlemler oldukça yüksek seviyeli biçimde yürütülmektedir. Keras modelinde programcının kabaca ağ için şu bilgileri Keras'a vermesi gereklidir:

- Ağdaki katmanlar
- Katmanlardaki nöronların sayısı
- Nöronların kullanacağı aktivasyon (transfer fonksiyonları)
- Eğitimde ve teste kullanılacak girdi ve çıktı kümeleri
- Sistemin amaçlarına yaklaşlığını ölçmek için kullanılacak "amaç fonksiyonu (loss function)"
- w değerlerinin iyileştirme yöntemini belirleyen optimizasyon algoritması

Biz burada önce Keras'ta işlem adımlarını kabaca ele alacağımız sona ayrıntılara gireceğiz.

Keras'ta işlemler tipik olarak (her zaman değil) şu aşamalardan geçilerek gerçekleştirilmektedir:

1) Öncelikle yapay sinir ağı bir sınıf nesnesi ile temsil edilir. Burada kullanılan sınıf tipik olarak Sequential isimli sınıfıdır. Programcı bu Sequential sınıfı türünden bir nesne yaratır. Yaratılan bu nesneye genellikle "model" denilmektedir. Örneğin:

```
from tensorflow.keras.models import Sequential  
  
model = Sequential()
```

keras modülünde Sequential dışında başka model sınıfları da vardır. Bu sınıflar ilerde ele alınacaktır.

2) Bu aşamada artık katmanları modele eklememiz gereklidir. Katman ekleme işlemi için Sequential sınıfının add metodunu kullanılmaktadır. add metodunu bizden argüman olarak katman istemektedir. Keras'ta katmani temsil eden çeşitli sınıflar vardır. Bunların en çok kullanılanı Dense isimli sınıfıdır. Dense katmanı (dense yoğun anlamına geliyor) bu katmandan önceki katmanın her nöronunun bu katmanın her nöronu ile bağlanacağı anlamına gelmektedir. Dense sınıfının __init__ metodu aşağıdaki gibidir:

```
tensorflow.keras.layers.Dense(units, activation=None, use_bias=True,  
kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None,  
bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

Metodun units parametresi ilgili katmandaki nöron sayısını belirtmektedir. activation parametresi o katmandaki tüm nöronların aktivasyon (transfer) fonksiyonunu belirtir. Burada fonksiyon isim olarak ya da fonksiyon olarak girilebilmektedir. Fonksiyon olarak girilecekse keras Activation modülündeki fonksiyonlar kullanılmalıdır. use_bias parametresi eğer True geçilirse (default durum) bu durumda bias_initializer parametresi nöronundaki "bias" değeri olarak kullanılmaktadır. bias değerinin anlamı ileride ele alınacaktır. kernel_initializer parametresi başlangıçtaki 'w' değerlerinin rastgele dağılımını belirtmektedir. Yani biz işin başında bu 'w' değerlerini rastgele biçimde belli bir dağılıma göre oluşturabiliriz. Bu parametre default geçilebilir. Metodun diğer parametreleri şimdilik ele alınmayacaktır. Bunlar da default geçilebilirler.

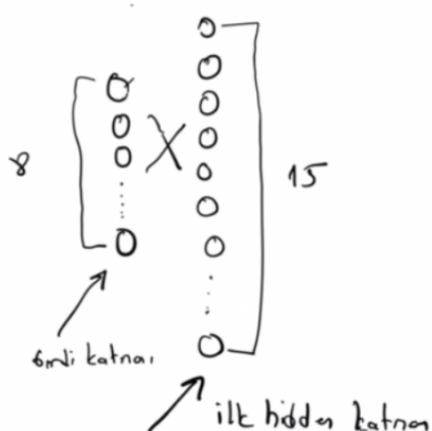
Dense fonksiyonunda input_dim isimli özel parametre girdi katmanındaki nöron sayısını belirtmek için kullanılmaktadır. Zaten girdi katmanı normal bir katman gibi olmadığı için girdi katmanı için ayrı bir katman oluşturulmamaktadır. Yani ağa eklenen ilk katmanda input_dim girdi katmanındaki nöron sayısını (yani girdi değişkenlerinin sayısını) belirtir. Örneğin:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()

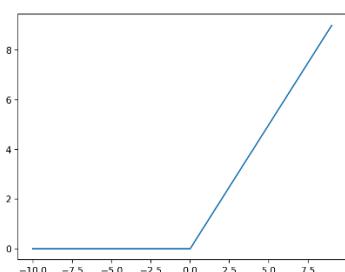
model.add(Dense(15, input_dim=8, activation='relu'))
```

Burada biz 15 nöronlu bir hidden katman oluşturduk. Bu modelimizin ilk hidden katmanı olduğu için bu katmanda input_dim ile girdi katmanındaki nöronların sayısını da belirttik.



Burada hidden katmanın transfer fonksiyonu "relu (rectifier linear unit)" olarak alınmıştır. Pek çok problem türü için hidden katmanlarda en fazla kullanılan transfer (aktivasyon) fonksiyonu budur. Bu fonksiyon negatif x değerleri için 0, pozitif x değerleri için x değerini veren basit bir fonksiyondur:

$$f(x) = \max(0, x)$$



relu fonksiyonu aşağıdaki gibi basit bir biçimde yazılabılır:

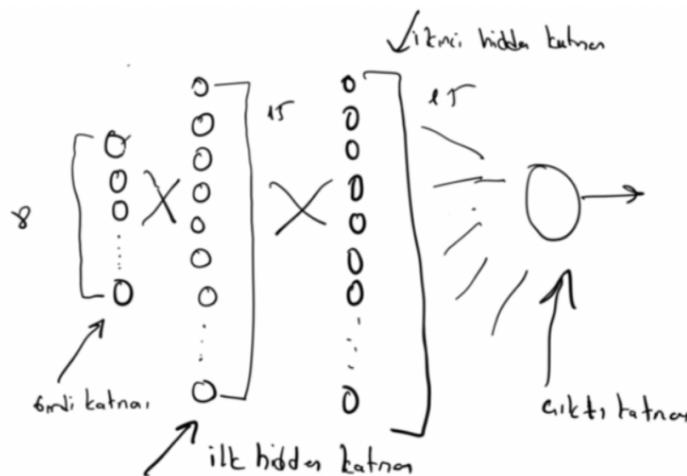
```
def myrelu(x):
    return np.maximum(0, x)
```

Ya da örneğin şöyle de yazılabılır:

```
def myrelu(x):
    return x * (x > 0)
```

Şimdi ikinci hidden katmanı ve çıktı katmanını da ağa ekleyelim:

```
model.add(Dense(15, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```



Ağın çıktı katmanı için özel bir belirleme yapılmamaktadır. Her zaman son katman zaten çıktı katmanı olmaktadır. Dense katmanının "önceki katmandaki nörünların hepsinin bu katmandaki nöronların hepsiyle bağlantılı olacağı" anlamına geldiğine bir kez daha dikkat ediniz.

Böylece yukarıdaki örnekte ağımızda 4 katma oluştu: Girdi katmanı + 2 hidden katman + çıktı katmanı. Şimdi pima-indians-diabetes.csv dosyasını okuyarak girdileri bu modele uygulayalım:

```
import numpy as np
from sklearn.model_selection import train_test_split

dataset = np.loadtxt('pima-indians-diabetes.csv', delimiter=',')
training_set_x, test_set_x, training_set_y, test_set_y = train_test_split(dataset[:, :8], dataset[:, 8], test_size=0.2)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()

model.add(Dense(15, input_dim=8, activation='relu'))
model.add(Dense(15, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Genellikle ikili sınıflandırma (binary classification) tarzı problemlerde hidden katmanlarının transfer fonksiyonları "relu", çıktı katmanının transfer fonksiyonu ise "sigmoid" olarak alınmaktadır.

3) Modelde katmanlar eklendikten sonra artık modelin derlenmesi gereklidir. Bu derleme terimi programlamadaki derleme anlamında kullanılmamaktadır. Bir çeşit konfigüre etmek anlamında kullanılmaktadır. Konfigüre etme işlemi Sequential sınıfının compile metodu ile yapılmaktadır. Metodun parametrik yapısı şöyledir:

```
compile(optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_mode=None,
weighted_metrics=None, target_tensors=None)
```

compile metodundaki üç önemli parametre optimizer, loss ve metrics parametreleridir. Diğer parametrelerin ana önemi yoktur. Yapay sinir ağlarını eğitirken her eğitim işleminden (epoch) sonra gerçek değerlerle elde edilen değerler arasındaki hata farkı bir biçimde hesaplanmaktadır. İşte bu hesaplanma yöntemine "loss" fonksiyonu denilmektedir. Örneğin bir regresyon modelinde gerçek olması gereken çıktı 172 iken ağımızın o anda bunu 182 olarak bulduğunu düşünelim. Burada bir fark söz konusudur. Normal olarak bu farkın minimize edilmesi gereklidir. İşte loss fonksiyonu bu farkı ölçmek için kullanılan fonksiyondur. İşlevinden dolayı bu fonksiyona "amaç fonksiyonu" da denilmektedir. Loss fonksiyonları problemin türüne göre seçilmektedir. Regresyon tarzı problemlerle, sınıflama tarzı problemlerde kullanılabilecek loss fonksiyonları farklıdır. Yapay sinir ağlarında çok kullanılan loss fonksiyonları şunlardır:

- Mean Squared Error: Regresyon modellerinde kullanılır. En fazla tercih edilen loss fonksiyonudur. Bu yöntemde olması gereken değerlerle olan değerler arasındaki farkın ortalaması bulunmaya çalışılmaktadır. Eğer bu fark ne kadar küçükse hedefe o kadar yaklaşıldığı anlama çıkar.

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

- Mean Absolute Error: Bu da regresyon modellerinde kullanılmaktadır. Fikir MSE ile aynıdır. Yalnızca kare yerine mutlak değer kullanılmaktadır.

$$MAE = \frac{\sum_{i=1}^n |y_i - x_i|}{n}$$

- Mean Absolute Percentage Error: Regresyon modellerinde kullanılmaktadır.

- Mean Square Logarithmic Error: Regresyon modellerinde kullanılmaktadır.

- Binary Cross-Entropy: Bu "loss" fonksiyonu ikil (binary) çıktılara ilişkin sınıflandırma modellerinde en çok tercih edilendir. Örneğin "İşte kalacak-İşten çıkışacak", "Hasta Diyabetli-Hasta Diyabetli Değildir" gibi.

- Categorical Cross-Entropy: Bu loss fonksiyonu ikiden fazla sınıflı modellerde çokça tercih edilmektedir. Örneğin kişinin eğitim düzeyinin tahmin edilmeye çalışıldığı model, şeklin hangi hayvana benzettiği gibi.

Yukarıdaki loss fonksiyonları compile metoduna fonksiyon olarak ya da isim olarak girilebilir. Girilecek isimler şöyledir:

```
'mean_squared_error' ya da 'mse'  
'mean_absolute_error' ya da 'mae'  
'mean_absolute_percentage_error'  
'mean_squared_logarithmic_error'  
'categorical_crossentropy'  
'binary_crossentropy'
```

compile fonksiyonunun optimizer parametresi 'w' değerlerinin iyileştirilmesi için kullanılacak algoritmayı belirtmektedir. Bu algoritma "loss" fonksiyonuyla karıştırılmamalıdır. Loss fonksiyonu yalnızca bizim hedefe yaklaşma miktarımızı hesaplamakta kullanılmaktadır. Bu hedefe daha iyi yaklaşmak için 'w' değerlerinin nasıl güncellenmesi gerektiği optimizasyon algoritmasıyla ilgilidir. Çok kullanılan optimizasyon algoritmalarından bazıları şunlardır:

Adam (Adaptive Moment Estimation): Açık ara en fazla tercih edilen optimizasyon algoritmasıdır.

Stochastic Gradient Descent: En çok kullanılan optimizasyon algoritmalarından biridir.

optimizer algoritması için kullanılacak yazılar şunlardır:

```
'sgd'  
'adam'
```

compile fonksiyonundaki metrics parametresi bir liste içerisinde metrics denilen fonksiyonları almaktadır. "metrics" de loss gibi bir amaç fonksiyonudur. Ancak metrics eğitimde epoch'lar arasında (epoch kavramından ilerde bahsedilecek) eğitimin yolunda gidip gitmediği bilgisini vermek için kullanılmaktadır. Modelin sınanması eğitim aşamasında her epoch sonrasında yapılmaktadır. İşte bu sınamada kullanılacak veri kümesine "sınama veri kümesi (validation data set)" denilmektedir. Sınama veri kümesi eğitim veri kümesi içerisinde bir parça olarak seçilmektedir. Sınama veri kümesinin ne anlam ifade ettiği ilerde ayrıca ele alınacaktır. Sınama için kullanılan tipik fonksiyonlar şunlardır:

- Accuracy: Tipik olarak regresyon problemlerinde metrics olarak kullanılır.
- Binary Accuracy: Bu tipik olarak ikilî değerli sınıflandırma problemlerinde kullanılır.
- Categorical Accuracy: İkiden fazla sınıflandırma problemlerinde tercih edilmektedir.
- Mean Absolute Error: Regresyon problemlerinde metrics değeri olarak sık kullanılmaktadır.

metrics parametresi için kullanılacak fonksiyon isimleri de şöyledir:

```
'accuracy' ya da 'acc'  
'binary_accuracy'  
'categorical_accuracy'  
'mae'
```

Sınama işlemi ile test işlemi birbirine karıştırılabilmektedir. Sınama işlemi eğitim sırasında "epoch"lar sonrasında yapılan bir çeşit test işlemidir. Halbuki test işlemi tamamen eğitimden sonra uygulanan bir işlemidir. Yani sınama sırasında aslında hala w değerleri son noktada değildir ve hala iyileştirilmektedir. Halbuki test işleminde w değerleri son halini almıştır. Eğitim tam olarak tamamlanmıştır.

Bu durumda compile metodunun örnek bir çağrı şöyledir:

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['binary_accuracy'])
```

4) Modelin konfigüre edilmesinden sonra sıra eğitim aşamasına gelmiştir. Yukarıda da belirtildiği gibi eğitim gerçek gözlemlerle modeldeki nöronların 'w' katsayılarının uygun biçimde ayarlanması sağlar. Şüphesiz eğitim veri kümesi ne kadar çok ve anlamlı ise, kullanılan optimizasyon algoritması ve loss fonksiyonu ne kadar uygun seçilmişse eğitimin sonucu da daha tatmin edici olacaktır. Eğitim işlemi Sequential sınıfının fit isimli metoduyla yapılmaktadır. Metodun parametrik yapısı şöyledir:

```
fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_split=0.0,  
validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0,  
steps_per_epoch=None, validation_steps=None, validation_freq=1)
```

Göründüğü gibi metodun çok sayıda parametresi vardır. Ancak en önemli olan parametreler ilk dört parametre olan x, y, batch_size ve epochs parametreleridir. x ve y gerçek gözlemlerdeki girdi ve çıktı değerlerine ilişkin ndarray nesneleridir. Eğitim sırasında gözlemler teker teker iterasyona sokulup loss değeri elde edilerek optimizasyon algoritması yoluyla 'w' değerleri güncellenebilir. Ancak bazen onbinlerce hatta milyonlarca eğitim verisini böyle tek tek işleme sokmak çok zaman alabilmektedir ve üstelik de bazı nedenlerden dolayı eğitimin başarısını da düşürebilmektedir. İşte bunun için bir grup gözlem verisini (yani satırı) sanki tek bir gözlem verisiymiş gibi (tek bir satılmış gibi) işleme sokma yoluna gidilmektedir. batch_size kaç tane gözlem verisinin (yani satırın) tek hamlede işleme sokulacağını belirtir. Her batch işleminden sonra loss fonksiyonu ile amaç değer hesaplanır ve optimizasyon algoritmasına sokularak 'w' değerleri güncellenir. Bu işleme "iterasyon" denilmektedir. Yani "iterasyon" batch_size kadar gözlemi tek hamlede işleme sokup bundan loss değeri elde etme ve optimizasyon işlemeye sokma adımlarının

bütünür. fit fonksiyonun epochs parametresi eğitim veri kümесinin baştan aşağı toplam kaç kez kullanılacağını belirtir. Tabii aynı veri kümесinin aynı model için birden fazla kez kullanılması veri analistine tuhaf gelebilir. Ancak ne olursa olsun bu yöntemin 'w' katsayılarını iyileştirmede belli bir faydası vardır. Çoğu zaman epoch değerini yükselttiğimizde modelin daha başarılı olduğunu görürüz. Ancak belli bir yükseltmeden sonra artık modelin başarısı pek artmamaktadır. Üstelik de epoch işlemleri ciddi bir zaman kaybına yol açmaktadır.

Yukarıda da belirttiğimiz gibi epoch eğitim veri kümесinin toplamda baştan sona kaç kere eğitim sürecinde kullanılacağını belirtmektedir. Yine yukarıda belirttiğimiz gibi her epoch işleminden sonra sınama (validation) işlemi yapılmaktadır. Sınama işleminde kullanılan veriler fit metodunun tarafından bizim ona verdığımız training_dataset_x ve training_dataset_y içerisinde alınmaktadır. İşte fit metodundaki validation_split isimli parametre bizim verdığımız verilerin yüzde kaçının sınama için kullanılacağını belirtmektedir. Metot kendi içerisinde bizim ona verdığımız training_dataset_x ve training_dataset_y vektörlerini bu oranda bölüp onun belli bir kısmını epoch işleminden sonra sınamada kullanmaktadır. Programcı isterse sınama işi için veri kümесini kendisi de oluşturabilir. Yani sınama verisinin eğitim verileriden alınması zounlu değildir. Bunun için fit metodunun validation_data parametresi kullanılmaktadır.

fit metodunun shuffle isimli parametresinin olduğuna ve bunun default değerinin True olduğuna dikkat ediniz. Bu shuffle parametresi fit metodunun tarafından bizim verdığımız veri kümelerinin işleme sokulmadan önce karıştırılacağı anlamına gelmektedir. Eğer bu parametre True ise karıştırma her epoch'tan önce yeniden yapılmaktadır.

Şimdiye kadar geçtiğimiz aşamalara ilişkin kod şöyledir:

```
import numpy as np
from sklearn.model_selection import train_test_split

dataset = np.loadtxt('pima-indians-diabetes.csv', delimiter=',')
training_set_x, test_set_x, training_set_y, test_set_y = train_test_split(dataset[:, :8],
dataset[:, 8], test_size=0.2)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()

model.add(Dense(15, input_dim=8, activation='relu'))
model.add(Dense(15, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(training_dataset_x, training_dataset_y, epochs=100, validation_split=0.2)
```

fit fonksiyonu her epoch'tan sonra sınama verisi üzerinde elde ettiği başarı değerini ekrana yazdırmaktadır. Bu başarı değeri bizim metrics ile verdığımız algoritmadan elde edilen değerdir. Programcı bu değerlere bakarak model hakkında bir fikir elde edebilir. Duruma göre modelini değiştirek düzeltebilir. Eğitim işleminde batch_size ve epoch değerlerinin değiştirilmesi modelin başarısını çeşitli biçimlerde değiştirebilmektedir. "pima-indians-diabets.csv" dosyasındaki veriler kullanılarak değişik batch_size ve epoch değerleri için elde edilen bazı metrics değerleri aşağıda verilmiştir. Tabii değerler her bilgisayarda biribirinden az çok farklı olabilir. Çünkü ağdaki nöronların başlangıçtaki 'w' değerleri yukarıdaki kodda rastgele alınmaktadır.

```
model.fit(training_set_x, training_set_y, batch_size=30, epochs=10)
614/614 [=====] - 0s 33us/step - loss: 0.2692 - acc: 0.8746

model.fit(training_set_x, training_set_y, batch_size=30, epochs=50)
614/614 [=====] - 0s 36us/step - loss: 0.2672 - acc: 0.8779

model.fit(training_set_x, training_set_y, batch_size=30, epochs=100)
614/614 [=====] - 0s 34us/step - loss: 0.2609 - acc: 0.8876

model.fit(training_set_x, training_set_y, batch_size=100, epochs=100)
```

Şimdi modelimize bir tane daha hidden katmanı ekleyelim. Acaba modelimiz daha başarılı hale gelecek mi? İşte her modelde hidden katman sayısının artırılması odeli daha başarılı hale getirmemektedir. Yukarıdaki konularda da dephinildiği gibi pek çok regresyon ve sınıflandırma problemlerinde iki hidden katman yeterli olmaktadır. Bu katmanlarının sayısının artırılması işlem süresini uzattığı halde başarıyı artırmayabilir. Ancak bazı tarz problemlerde derin öğrenme ağları çok iyi sonuçların elde edilmesine yol açabilmektedir.

fit metodu geri dönüş değeri olarak History nesnesi vermektedir. Bu nesne içerisinde biz her epoch'taki sonuç bilgilerini alabiliyoruz.

Bu noktaya kadar öğretiklerimizi şöyle özetleyebiliriz:

- Toplam veri kümesi başlangıçta eğitim ve test veri kümesi olmak üzere ikiye ayrılmaktadır. Genellikle test veri kümesi %20 civarlarında seçilir.
- Eğitim veri kümesi de kendi arasında ikiye ayrılmaktadır. Asıl kısım eğitimde kullanılacak kısımdır. Diğer kısım sınama amacıyla kullanılan kısımdır. Biz sınama amacıyla kullanılan kısımı "sınama veri kümesi (validation data set)" diyoruz.
- Epoch toplam eğitim verilerinin baştan sona kaç kere eğitimde kullanılacağına denilmektedir.
- Batch (batch size) eğitim verilerinin tek tek değil grüplanarak işleme sokulması anlamına gelmektedir. Batch işlemi hem bir yandan eğitim süresini kısaltırken hem de "overfit" durumunu engelleyebilmektedir.
- Sınama işlemi her epoch'tan sonra yapılır.
- w değerleri ise her batch işleminden sonra güncellenmektedir.

5) Şimdi artık sıra modelin test veri kümesiyle test edilmesine gelmiştir. Bu işlem Sequential sınıfının evaluate metoduyla yapılmaktadır. evaluate metodunun parametrik yapısı şöyledir:

```
evaluate(x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None,
callbacks=None)
```

Metodun ilk iki parametresi test veri kümesinin girdi ve çıktı matrislerini almaktadır. Burada da test veri kümesi yinr batch işlemi ile ele alınmaktadır. Bu parametre için argüman girilmezse default batch_size değeri 32 alınmaktadır. Fonksiyon bize test veri kümesinden elde edilen metrics değerlerini bir list olarak vermektedir. Yukarıda da belirttiği gibi birden fazla metrics kullanılabilmektedir. Kullanılan metrics isimleri ayrıca Sequential sınıfının metrics_names property'sinden elde edilebilir. Tipik olarak bu listelerin ilk elemanı "loss" değerine diğer elemanları da metrics ile belirttiğimiz değere ilişkindir. Örneğin:

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['binary_accuracy'])

model.fit(training_dataset_x, training_dataset_y, batch_size=32, epochs=100,
validation_split=0.2)
result = model.evaluate(test_dataset_x, test_dataset_y)
```

Örnek çıktı şöyledir:

```
loss : 0.5056907929383315
acc : 0.7662337631374211
```

6) Model test edildikten sonra kullanıma hazır hale gelmiştir. Artık kestirimde bulunulabilir. Bizim zaten öğrenen modellerden amaçladığımız şey kestirimde bulunmasıdır. Kestirim işlemi Sequential sınıfının predict isimli metoduyla yapılmaktadır. Metodun parametrik yapısı şöyledir:

```
predict(x, batch_size=None, verbose=0, steps=None, callbacks=None)
```

Metodun birinci parametresi kestirimde bulunulacak girdi verilerini temsil etmektedir. Yani bu parametre girdi katmanındaki nöronlara uygulanacak değerleri belirtmektedir. Bu girdi verisi tek bir vektör olabileceği gibi bir grup vektör de olabilir. batch_size eğer girdi verisi birden fazlaysa yine bunların kaçarlı olarak kestirimde bulunulacağını belirtir. Metodun çıktısı normal olarak yapay sinir ağının çıktı katmanındaki değerdir. Tabii metoda birden fazla girdi verilirse çıktı da buna göre birden fazla değerden oluşacaktır. predict metodundaki x parametresi bir matris biçiminde organize edilmelidir. Yukarıdaki model için örnek bir kestirim şöyle yapılabilir:

```
x = np.array([[7, 110, 83, 31, 0, 35.9, 1.130, 48]])  
  
output = model.predict(x)  
print(output)  
if output[0, 0] > 0.5:  
    print('diyabetli')  
else:  
    print('diyabetsiz')
```

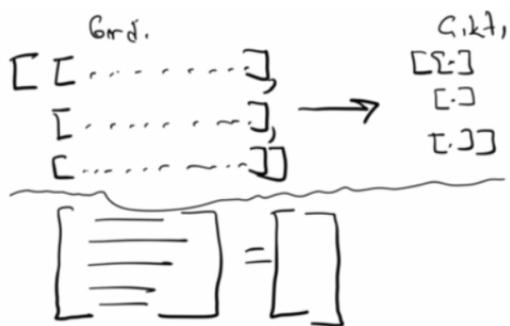
Burada biz kişinin 8 özelliğini girdi olarak kullanıp bir çıktı elde ettik. Elde ettiğimiz çıktıının görünümü şöyledir:

```
[[0.5785923]]  
diyabetli
```

predict işleminin teker teker yapılmasına gerek yoktur. Biz predict metodunda bir matris de verebiliriz. Bu durumda predict bize çıktı olarak o uzunlukta bir matris verecektir. Örneğin:

```
x = np.array([[1, 1, 48, 20, 0, 24.7, 0.90, 22],  
             [1, 2, 48, 30, 0, 24.7, 0.92, 22],  
             [2, 3, 48, 30, 0, 24.7, 0.94, 22]])  
  
output = model.predict(x)  
print(output)  
  
for i in range(len(output)):  
    if output[i, 0] > 0.5:  
        print('diyabetli')  
    else:  
        print('diyabetsiz')
```

Burada predict metoduna üç satırı bir matris argüman olarak verilmiştir. Yani biz tek hamlede ü farklı gözlem sonucunu tahmin etmek istemektedir. O halde predict metodu da bize 3 elemanlı bir sütun matrisi verecektir. Bu durumu şekilsel olarak aşağıdaki gibi gösterebiliriz.



Buradaki 0.5785923 değeri ağımızın çıktı katmanındaki değerdir. Bu değer sigmoid fonksiyonun elde edildiğini anımsayınız. Halbuki biz bu işlemden ikil (binary) bir değer elde etmek istyorduk. İşte bu durumda elde ettiğimiz bu değeri bir eşik fonksiyonuna sokmamız gereklidir. Tabii tipik eşik fonksiyonu 0.5'ten büyük değerleri 1 olarak 0.5'ten düşük değerleri 0 olarak veren fonksiyondur.

Pekiyi predict metodundaki batch_size parametresinin anlamı nedir? Anımsanacağı gibi fit işleminde batch_size bir grup satırın tek bir satırımsız gibi işleme sokulması anlamına geliyordu. Yani fit işleminde 'w' değerlerinin güncellenmesi her satırda değil her batch_size kadar satırda bir yapılmaktadır. Ancak predict metodundaki batch_size farklı bir anlamdadır. Bu metottaki batch_size predict işlemini aynı anda kaçarlı gruplar halinde yapılacağını belirtmektedir. Yani buradaki batch_size değerinin predict sonucu üzerinde hiçbir etkisi yoktur. Başka bir deyişle biz buraki batch_size değerini 32 versek de (default durum) 1 versek de tamamen aynı sonucu elde ederiz. O halde bu parametrenin ne faydası vardır? Bu parametre içsel olarak Keras'in predict işlemini hızlı yapması konusunda faydalı olabilmektedir.

Keras'ta Kullanılan Veri Kümeleri ve Algoritmala İlişkin Kavramların Özeti

Temel olarak bir sinir ağını eğitirken üç veri kümesi söz konusudur: Eğitim Veri Kümesi (Training Data Set), Sınamma Veri Kümesi (Validation Data Set) ve Test Veri Kümesi (Test Data Set). Bunların oranları uygulamadan uygulamaya değişebilse de genellikle %64, %16, %20 gibi oranlar kullanılmaktadır. Eğitim veri kümesi -ismi üzerinde- eğitim sürecinde kullanılmaktadır. Sınamma veri kümesi eğitim süresinde her epoch işleminden sonra uygulanmaktadır. Sınamma veri kümesinin her batch size'dan sonra değil her epoch'tan sonra uygulandığına dikkat ediniz. Test veri kümesi ise tüm eğitim bittikten sonra uygulanan veri kümesidir. Pekiyi bu durumda sınamma veri kümesi ile test veri kümesi arasında ya da sınamma işlemi ile test işlemi arasında ne fark vardır? Doğrulama veri kümesi her epoch işleminden sonra uygulandığı için modelin epoch işlemlerine göre davranışını gözlenebilmektedir. Halbuki test veri kümesi tüm model eğitildikten sonra performans ölçütlerini belirlemek için kullanılır.

Model eğitilirken her batch size kadar eğitim veri kümesi biraraya getirilerek eğitimde kullanılır. Yani eğitim işlemi birer birer değil çanak çanak (batch batch) yapılmaktadır. Loss fonksyonunun hesaplanması ve w katsayılarının optimizasyon algoritmasına göre güncellenmesi her batch işleminden sonra yapılmaktadır. Aslında Keras'ta her batch işleminde de callback mekanizmasıyla araya girmek mümkündür. Ancak biz kursumuzda böyle bir işlem yapmayacağız.

Loss fonksyonu w değerlerinin güncellenmesi için bir amaç fonksyonu oalrak kullanılmaktadır. Optimizasyon algoritması ise loss fonksyonuna bakılarak w değerlerinin nasıl güncelleneceğini belirten algoritmadır. Yani başka bir deyişle "optimizasyon algoritması loss fonksyonunu minimize etmek için w değerlerinin nasıl değiştirileceğini belirten algoritmadır."

Pekiyi metrik fonksyonlar ne anlama gelmektedir? Metrik değerler doğrulama süreci ile ilgilidir. Genellikle metrik fonksyonlar loss fonksiyonuyla aynı olabilir. Ancak eğitimde birden fazla metrik fonksyonu da kullanılabilmektedir. Pekiyi loss fonksyonu ile metrik fonksyonları arasında ne fark vardır? İşte loss fonksyonu optimizasyon algoritmasını uygulamak için bir hedef belirtirken metrik fonksyonları ise doğrulama işlemi sonucunda bir performs belirtmektedir.

Batch İşleminin Anlamı

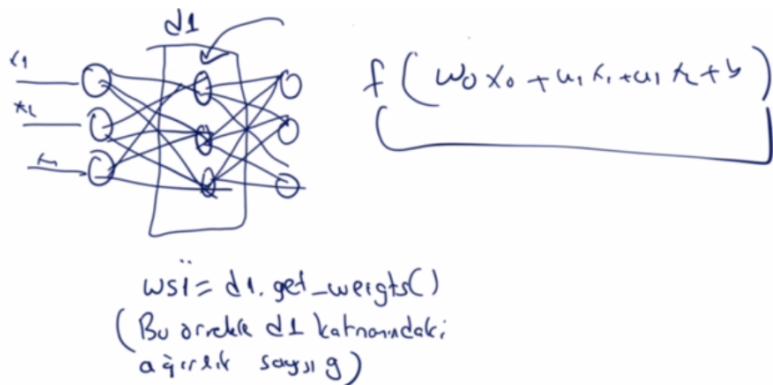
Eğitim ve test sırasında girdi kümelerinin birer birer değil bir grup olarak işleme sokulduğunu belirtmiştık. Bu grubun büyüğünne "batch size" denilmektedir. Girdilerin teker teker eğitime sokulmayıp grup grup sokulmasının anlamı nedir? Batch işleminin genel olarak bir hızlandırma sağladığı söylenebilir. Batch işleminin yapılış biçimini kullanılan optimizasyon ve loss fonksyonuna göre değişimle birlikte tipik uygulansı şöyledir: Biz batch büyüğünü (batch size) 32 kabul edelim. Bu durumda tipik olarak 32'lik bir kümeyi elemanları (bu elemanlara "örnek (sample)" da denilmektedir.) tek tek ve ayrı ayrı ağa sokulup buradan bir loss değeri hesaplanır. Sonra bu loss değerinin batch için ortalaması alınır (örneğimizde bu değer 32'ye bölünecektir.) Bu ortalama değer tek bir değer olarak optimizasyon algoritmasına verilmektedir. Böylece w değerlerini güncellenmesi birer birer değil batch-batch yapılacaktır. Bu işlem genel olarak hesaplamayı hızlandırmaktadır. Batch işlemi aynı zamanda overfit durumu için de kullanılabilmektedir. Overfit konusu sonraki bölümlerde ele alınmaktadır.

Tabii batch içerisindeki bir grup veri aslında Python'da bir hamlede işleme sokulur. Yani bunun için bir döngü gerekmemektedir. Zaten numpy kütüphanesi bu işlemi yapmaktadır. Pekiyi biz her ne kadar numpy'da iki vektörü tek hamlede işleme sokabiliyor olsak da arka planda numpy bu işlemi işlemciye bir döngü içerisinde sıralı olarak mı

yaptırmaktadır? İşte numpy arka planda C'de yazılmış bir kütüphanedir. İşlemcinin vektörel işlem yapma yeteneğini de kullanmaktadır. (Örneğin Intel işlemcileri ve ARM işlemcileri böyle vektörel işlemler yapabilmektedir.)

Keras Modelindeki Nöron Ağırlıklarının (W Değerlerinin) Elde Edilmesi ve Set Edilmesi

Keras'ta eğitim sonrasında nöronlarda oluşan ağırlık değerlerini almak isteyebiliriz. Keras model olarak bu bilgileri katman nesnesi ile ilişkilendirmiştir. Yani bu bilgiler eğitim sonrasında katman nesnelerinden elde edilmelidir. İşte Dense sınıfının ve diğer katman sınıflarının `get_weights` ve `set_weights` metotları bu ağırlıkların get ve set edilmesi için kullanılmaktadır. Tabii her katman kendi ağırlık değerlerini bize verecektir.



Bir katmandaki nöron ağırlıkları o katmandaki nöronlara gelen girişlerdeki ağırlıktır.

Mademki nöron ağırlıkları katman nesnelerinin içerisindeydi o halde bizim katmanlardaki nöronların 'w' değerlerini elde edebilmemiz için yaratmış olduğumuz katman nesnelerine erişebilmemiz gereklidir. Halbuki biz yukarıda Dense katman nesnelerini yaratır yaratmaz hemen Sequential sınıfının `add` metoduya modele ekledik. Yani katman nesnelerini bir değişkende tutmadık. Yukarıdaki örneğin ilgili kısmını anımsayınız:

```
...
model = Sequential()

model.add(Dense(15, input_dim=8, activation='relu'))
model.add(Dense(15, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
...

```

Katman nesnelerini elde edebilmenin bir yolu `add` metodunu uygulamadan önce onları değişkenlerde saklamak olabilir. Örneğin:

```
model = Sequential()
d1 = Dense(15, input_dim=8, activation='relu')
model.add(d1)
d2 = Dense(15, activation='relu')
model.add(d2)
d3 = Dense(1, activation='sigmoid')
model.add(d3)
...
ws1 = d1.get_weights()
ws2 = d2.get_weights()
ws3 = d3.get_weights()
```

Aslında bu yönteme gerek yoktur. Çünkü Sequential sınıfının zaten `layers` isimli özniteligi bize katman nesnelerini bir liste biçiminde vermektedir. Bu durumda örneğin biz modelimizin ilk katmanına ilişkin (yani birinci hidden katmana ilişkin) 'w' değerlerini şöyle elde edebiliriz:

```
ws1 = model.layers[0].get_weights()
ws2 = model.layers[1].get_weights()
ws3 = model.layers[2].get_weights()
```

`get_weights` metodu bize duruma göre iki elemanlı ya da tek elemanlı bir liste vermektedir. Şöyled ki: Eğer katman yaratıldığında Dense fonksiyonunda bias=False değeri geçilirse ilgili katmandaki nöronlarda bias değerleri kullanılmayacaktır. Bu durumda `get_weights` metodu da yalnızca ağırlık matrisinden oluşan bir elemanlı bir ndarray listesi verecektir. Eğer katman yaratıldığında Dense fonksiyonunda bias=True değeri geçilirse (default durum zaten böyledir) artık `get_weights` fonksiyonu iki elemanlı bir listeye geri döner. Listenin ilk elemanı ağırlık matrisinden diğer elemanı bias değerlerinden oluşmaktadır. Anımsanacağı gibi bias değeri her nöron için oluşturulan bir değerdir. Dolayısıyla örneğin eğer katmanımızda 15 nöron varsa 15 tane bias değeri olacaktır. Bu bias değerlerinin ne işe yaradığı ileride ele alınacaktır. Ancak bu bias değerleri de modelin bir parçasını oluşturmaktadır.

`set_weights` metodu da benzer biçimde ağırlık değerlerini ilgili katmana yüklemek için kullanılmaktadır. Yani örneğin aşağıdaki iki işlem sonuçta hiçbir etkiye yol açmayacaktır:

```
ws1 = model.layers[0].get_weights()  
model.set_weights(ws1)
```

Oluşturulan Modelin ve Ağırlık Değerlerinin Saklanması ve Geri Yüklenmesi

Bir modeli eğittiğinden sonra bilgisayarın kapadığımızda tüm eğitim bilgileri kaybolacaktır. İşte onun başka zaman kullanılabilmesi için bir dosyada saklanması gerekebilir. Keras bu işlemler için HDF dosya formatını kullanmaktadır. Sequential sınıfının `save` isimli metodu bizden .h5 uzantılı HDF dosyasının yol ifadesini ister, model bilgilerini, ağırlık ve bias değerlerini bu dosyaya save eder. Örneğin:

```
model.save('diabetes.h5')
```

Artı biz tüm modeli her şeyiyle bir dosya içerisinde saklamış olduk. Onu geri almak için `tensorflow.keras.models` modülündeki `load_model` fonksiyonu kullanılmaktadır. Bu fonksiyon da benzer biçimde HDF dosyasının yol ifadesini argüman olarak alır. Tüm modeli ağırlık ve bias değerleriyle yeniden yükler, yeni bir model nesnesi yaratarak bie onu verir. Örneğin:

```
from tensorflow.keras.models import load_model  
  
model = load_model('diabetes.h5')
```

Şimdi yukarıdaki örneği `save` edip tekrar yükleyerek `predict` işlemi yapalım:

```
# keras-model-save.py  
  
import numpy as np  
  
from tensorflow.tensorflow.keras import Sequential  
from tensorflow.keras.layers import Dense  
from sklearn.model_selection import train_test_split  
  
dataset = np.loadtxt('diabetes.csv', skiprows=1, delimiter=',')  
dataset_x = dataset[:, :-1]  
dataset_y = dataset[:, -1]  
training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =  
train_test_split(dataset_x, dataset_y,  
test_size=0.20)  
  
model = Sequential()  
model.add(Dense(200, input_dim=8, activation='relu'))  
model.add(Dense(200, activation='relu'))  
model.add(Dense(200, activation='relu'))  
model.add(Dense(1, activation='sigmoid'))  
  
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['binary_accuracy'])
```

```

model.fit(training_dataset_x, training_dataset_y, batch_size=32, epochs=200,
validation_split=0.2)
result = model.evaluate(test_dataset_x, test_dataset_y)

for i in range(len(result)):
    print('{} = {}'.format(model.metrics_names[i], result[i]))

model.save('diabetes.h5')

# keras-model-save.py
import numpy as np
from tensorflow.keras.models import load_model

model = load_model('diabetes.h5')

predict_x = np.array([[5.0, 73.0, 60.0, 0., 0., 29.8, 0.368, 25.0],
                      [6.0, 43.0, 70.0, 0., 0., 29.8, 0.368, 25.0],
                      [6.0, 73.0, 70.0, 0., 0., 39.8, 0.668, 25.0]])

result = model.predict(predict_x)

for i in range(len(result)):
    if result[i, 0] < 0.5:
        print('Şeker hastası değil')
    else:
        print('Şeker hastası')

```

Bazen programcı yalnızca ağırlık değerlerini saklayıp geri yüklemek isteyebilir. Bu işlem Sequential sınıfının save_weights ve load_weights metotlarıyla yapılmaktadır. Örneğin:

```

ws = model.save_weights('diabtes_weights.h5')
...
model.load_weights('diabtes_weights.h5')

```

Tabii bu durumda biz bu ağırlık değerlerini kullanabilmek için modeli yeniden oluşturmalıyız. Çünkü save_weights model bilgilerini saklamamaktadır. Yalnızca modeldeki tüm nöürünların ağırlık değerlerini saklamaktadır.

İstenirse load işlemi yapılırken yalnızca aynı isimli katmanların ağırlıkları da yüklenebilmektedir. Şöyle ki: Dense fonksiyonunda name isimli parametre yoluyla katmanlara tek olan (unique) isimler verebilmekteyiz. İşte Sequential sınıfının load_weights isimli metodunun da by_name isimli bool değer alan bir parametresi vardır. Bu parametre default durumda False biçimdedir. Eğer bu parametre True yapılrsa bu durumda diskte saklanmış olan modeldeki katman isimleriyle hali hazırda çalışmakta olan model nesnesindeki katmanların isimleri karşılaştırılır. Aynı isimli olan katmandaki ağırlık ve bias değerleri yüklenir. Örneğin:

```
model.load_weights('diabtes_weights.h5', by_name=True)
```

Burada diske 'x' ve 'y' isimli iki katman olsun. Şu andaki model nesnesinde de bu isimli iki katmanın bulunduğu varsayılmı. Artık load_weights metodu yalnızca bu katmanların ağırlık ve bias değerlerini mevcut modeldeki kilerin üzerine yükleyecektir.

Bir modeli diske save ettikten sonra onu tekrar yükleyerek eğitime devam edebiliriz. Yani eğitimin (fit metodunun) yalnızca bir kez aynı zaman dilimi içerisinde çağrılmaması gerekmemektedir.

Keras Modelinde Callback Mekanizması

Keras kütüphanesinde bazı işlemlerde programcının bilgilendirilmesi için "callback" denilen bir mekanizma düşünülmüştür. fit, evaluate ve predict metotları işlemlerini yaparken programcının belirlediği kodları çalıştırılabilmektedir. Bu fonksiyonların genel olarak callbacks isimli parametreleri vardır. Bu parametreye callback

görevini yapacak bir sınıf nesnesi yerleştirilir. Bazı callback sınıfları zaten hazır durumdadır. Bu nedenle programcının kendi callback sınıflarını yazaması genellikle gerekmektedir. Ancak programcı isterse kendi callback sınıflarını da yazabilir. Biz budara hazır birkaç callback sınıfını ele alacağız ve callback sınıflarının nasıl yazılabileceği hakkında bir örnek vereceğiz. Diğer callback sınıfları için Keras dokümanlarına başvurulabilir. Keras'taki tüm callback sınıfları tensorflow.keras.callbacks modülündeki Callback isimli sınıfından türetilmiştir.

History isimli callback sınıfı kayıt işlemi yapmaktadır. Yani örneğin fit metodu çalışırken bu callback sınıfı bizim için bazı kayıtlar tutmaktadır. fit metodunun çalışması bittikten sonra biz bu kayıtları alıp kullanabiliriz. Aslında fit metodunun callbacks parametresine biz History callback sınıf nesnesini geçirmek zorunda değiliz. fit metodu zaten her zaman bu kaydı yaparak bize bu History nesnesini geri dönüş değeri olarak vermektedir. Biz de yapılan kayıtları fit metodunun geri dönüş değerinden alarak kullanabiliriz. Örneğin:

```
hist = model.fit(training_set_x, training_set_y, batch_size=30, epochs=100)
```

History nesnesinin history isimli örnek özniteliği dict türündendir. Bu sözlük nesnesinin anahtarları "elde edilecek bilginin isimlerini", değerleri de her epoch için ilgili değerleri bize vermektedir. history örnek özniteliği ile biz verilen sözlükte her zaman 'loss' isimli bir anahtar bulunmaktadır. Bu sözlüğün diğer anahtarları ise bizim fit metodunda metrics'te verdığımız metrik isimleridir. Örneğin yukarıdaki model şöyle konfigüre edilmiş olsun:

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy', 'mae'])
```

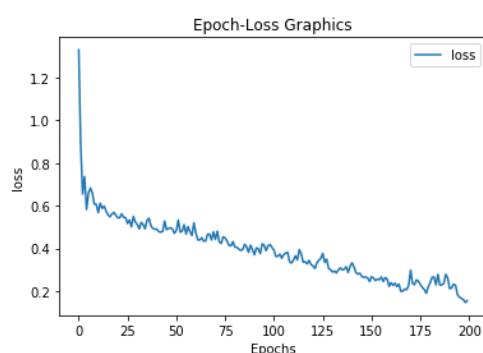
Bu durumda history nesnesinin history özniteliği bize üç anahtar verecektir:

```
In [29]: hist.history.keys()
Out[29]: dict_keys(['loss', 'acc', 'mean_absolute_error'])
```

Bu anahtarları biz sözlüğe verdığımızda sözlük bize her epoch sonrasında ilgili değeri bir liste biçiminde vermektedir. Örneğin burada 100 epoch uygulandığına göre bu dizinin elemanları 100'lük olacaktır. Bu durumda biz fit işleminde her epoch'tan elde edilen loss fonksiyon (amaç fonksiyon) değerinin grafiği şöyle çizdirebiliriz:

```
import matplotlib.pyplot as plt

plt.xlabel('epoch numbers')
plt.ylabel('loss')
plt.title('Epoch-Loss Graph')
plt.plot(range(len(hist.history['loss'])), hist.history['loss'])
plt.legend(['loss'])
```



Aslında plot fonksiyonuna biz yalnızca y değerlerini de girebildik. Bu durumda zaten x değerleri 0'dan başlayan sayılar biçiminde olmaktadır. Örneğin:

```
plt.plot(hist.history['loss'])
```

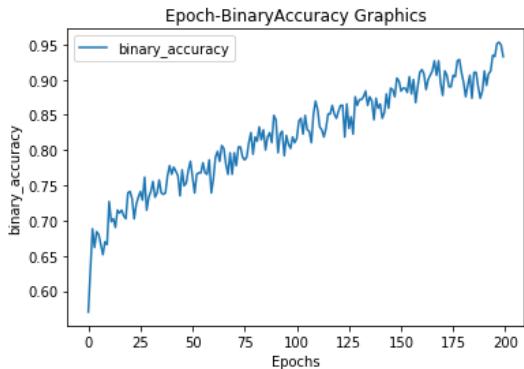
Şimdi de binary_accuracy metriği için grafik çizdirelim:

```

plt.xlabel('Epochs')
plt.ylabel('binary_accuracy')
plt.title('Epoch-BinaryAccuracy Graphics')
plt.legend(['binary_accuracy'])

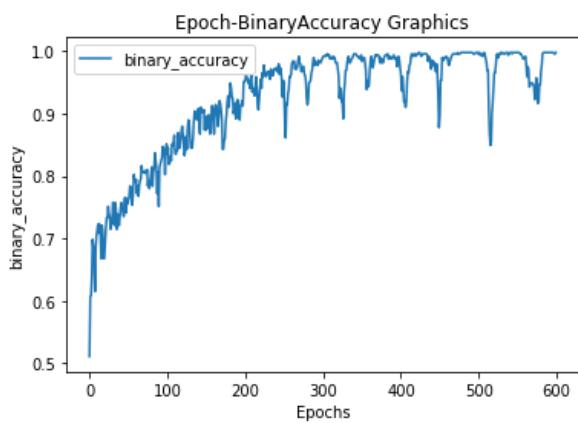
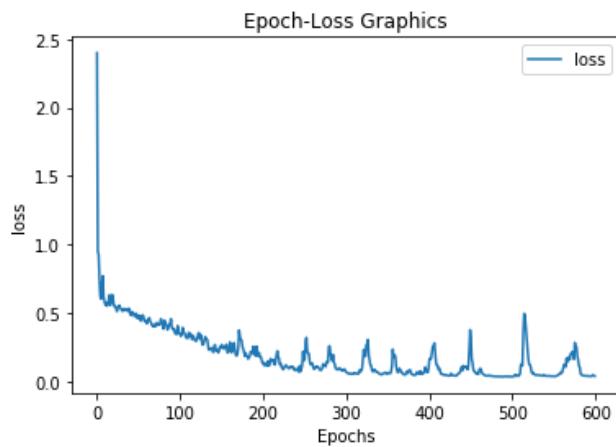
plt.plot(range(len(hist.history['binary_accuracy'])), hist.history['binary_accuracy'])

```



history nesnesinin bize ilgili değerleri iterasyon temelinde değil epoch temelinde verdiğine dikkat ediniz. Halbuki bazı durumlarda her iterasyonun sonucunun da grafiği çizilmek istenmektedir.

Pekiye history bilgilerine neden gereksinim duyabiliriz? İşte modeli eğitirken en uygun epoch değerini belirlemeye bu grafiklerden faydalananızdır. Çünkü grafikler çizildiğinde belli epoch değerlerinden sonra anomaliler gözle görülebilmektedir. Aynı zamanda hangi epoch değerinden sonra artık epoch'u artırmanın bir fayda sağlama olmadığı da görülebilmektedir. Aynı zamanda yukarıdaki grafikler bize epoch kaynaklı overfit durumu hakkında da bilgi verebilmektedir. Örneğin epoch değerini 200'den 600'e çıkartarak grafikleri yeniden oluşturalım:



Bu grafiklerde epoch sayısı arttıkça iyileşme olmakla birlikte kararsızlığında arttığı görülmektedir. Bu durum akla overfit olusunu getirmektedir. Bu durumda da gerçekten de 200 civarında epoch bu model için uygun gibi gözükmektedir.

CSVLogger isimli callback sınıfı tipki History gibi epoch temelinde kayıt yapmaktadır. Ancak history'den farklı olarak bu kaydı dosyaya bir csv dosyasına yazar. Örneğin:

```
csv_callback = CSVLogger('diabetes.csv')
h = model.fit(training_set_x, training_set_y, batch_size=30, epochs=500,
callbacks=[csv_callback])
```

metotlarda callbacks parametresinin bir liste olduğuna dikkat ediniz. Yani biz bu metotlara birden fazla callback nesnesi geçirebiliriz.

Diğer çok kullanılan hazır callback sınıflarından biri de LambdaCallback sınıfıdır. Bu callback nesnesi aslında çeşitli olaylarda bizim ona verdigimiz fonksiyonları çağırılmaktadır. LambdaCallback sınıfının başlangıç metodunu şöyledir:

```
tensorflow.keras.callbacks.LambdaCallback(on_epoch_begin=None, on_epoch_end=None,
on_batch_begin=None, on_batch_end=None, on_train_begin=None, on_train_end=None)
```

Metodun parametreleri olayları anlatmaktadır. Biz bu her parametreye bir fonksiyon girebiliriz. Bu fonksiyonların parametreleri de vardır. Parametreleri şunlardır:

Fonksiyon	Parametreler
on_epoch_begin	epoch ve logs
on_epoch_end	epoch ve logs
on_batch_begin	batch ve logs
on_batch_end	batch ve logs
on_train_begin	logs
on_train_end	logs

Buradaki epoch ve batch parametreleri epoch ve batch numarasını vermektedir. logs parametresi bir sözlük biçimindedir. end son ekli fonksiyonlarda logs parametresi tipki history nesnesinde olduğu gibi loss ve metrik değerlerini bize vermektedir.

Programcı kendisi de (custom) callback sınıfı yazabilir. Bunun için sınıfını tensorflow.keras.callbacks.Callback sınıfından türetmesi gereklidir. İşte aşağıdaki metotlar sınıfta yazılırsa ilgili işlemde bu metotlar devreye girecektir:

```
on_epoch_begin
on_epoch_end
on_batch_begin
on_batch_end
on_train_begin
on_train_end
```

Örneğin:

```
from tensorflow.keras.callbacks import Callback

class MyCallback(Callback):
    def on_epoch_begin(self, epoch, logs):
        print('epoch {} begins...'.format(epoch))

    def on_epoch_end(self, epoch, logs):
        print('epoch {} ends...'.format(epoch))
```

```

my_callback = MyCallback()
hist = model.fit(training_dataset_x, training_dataset_y, batch_size=32, epochs=10,
validation_split=0.2, callbacks=[my_callback])
result = model.evaluate(test_dataset_x, test_dataset_y)

```

Metotlardaki batch parametresi o andaki epoch uzunluğunu bize vermektedir. logs parametresi ise metik değerlerinin bulunduğu biz sözlük nesnesidir. Genel olarak buradaki metodların parametrik yapıları tensorflow.keras.callbacks. Callback sınıfında belirtilmiştir. Bu sınıf aşağıdaki gibi gerçekleştirilmiştir:

```

class Callback(object):
    def __init__(self):
        self.validation_data = None
        self.model = None

    def set_params(self, params):
        self.params = params

    def set_model(self, model):
        self.model = model

    def on_epoch_begin(self, epoch, logs=None):
        pass

    def on_epoch_end(self, epoch, logs=None):
        pass

    def on_batch_begin(self, batch, logs=None):
        pass

    def on_batch_end(self, batch, logs=None):
        pass

    def on_train_begin(self, logs=None):
        pass

    def on_train_end(self, logs=None):
        pass

```

Yapay Sinir Ağları Çalışması İçin Aşamalar

Yapay sinir ağları kestirim (prediction) amacıyla kullanılmaktadır. Yapay sinir ağları aslında $y = f(x)$ biçiminde bir f fonksiyonunun bulunması sürecini hedeflemektedir. Bu f fonksiyonu geçmiş verilerden hareketle yapay sinir ağları tarafından oluşturulmaktadır. Tabii aslında bu f fonksiyonunun bulunması için pek çok istatistiksel yöntem de vardır. Bu yöntemler ileride ele alınacaktır. Ancak yapay sinir ağları çok sayıda özelliğin (feature) olduğu ve özellikler arasında karmaşık etkileşimlerin bulunduğu sistemlerde diğer yöntemlere göre çok daha iyi sonuçlar vermektedir.

Yapay sinir ağları yönteminin kullanılması için tipik aşamalar şunlardır:

- 1) Hedefin Belirlenmesi Süreci: Önce bizim yapay sinir altında hangi kestirimde bulunacağımızı belirlememiz gereklidir.
- 2) Hedef Olarak Belirlenen Kestirimle İlgili Olabilecek Özelliklerin (features) Belirlenmesi Süreci: Veri bilimcisinin kestirimde bulunacağı olgunu etkileyen etmenleri (features) bir biçimde tespit etmiş olması gerekmektedir.
- 3) Eğitim İçin Verilerin Toplanması Süreci: Veri bilimcisinin eğitimde kullanacağı geçmiş birtakım verileri ve bunlara karşı kestirilecek gerçek değerleri (training dataset_x ve training dataset y) bir biçimde elde etmesi gerekmektedir.
- 4) Geçmiş Verilerin Ön İşleme Sokulması ve Hazır Hale Getirilmesi Süreci: Veriler toplandıktan sonra bunların önişele sokulması gereklidir. Önişlem sırasında kategorik verilerin sayısallaştırılması, yüksek korelasyonlu sütunların kümeden çıkartılması, one hot encoding işlemlerinin yapılması en yaygın ön işlem faaliyetlerindedir.
- 5) Yapay Sinir Ağı Modelinin Kurulması Süreci: Artık uygun bir sinir ağı modeli kurulur
- 6) Modelin Eğitilmesi ve Test Edilmesi Süreci
- 7) Kestirimde Bulunma Süreci

Yapay Sinir Ağları ve Derin Öğrenme Ağları İçin Hedefin Belirlenmesi Süreci

Yalnızca makine öğrenmesi için değil her türlü veri analizinde önce bir hedefimizin bulunması gereklidir. Bu hedef mümkün olduğu kadar açık bir biçimde belirtilmelidir. Yani biz ne yapmak istemektedir? Genellikle yapay sinir ağlarında ve derin öğrenme ağlarında bir kestirim yapılım istenir. Peki neyin kestirimini yapacağız? Kim için yapacağız? Sonuçta ne elde etmeyi planlamaktayız?

Örneğin bir dersanedeki öğrenciler arasında üniversite sınavında belli bir puanın yukarısında puan elde edebilecek kişileri önceden tespit etmem isteyelim. Bu tespit edilen öğrencilere özel bazı olanaklar sunulacak olabilir. Kurum da bu olanakları sunacak öğrencileri mümkün olduğu kadar önce tespit etmek isteyebilir.

Örneğin bir kurumun çeşitli mağazaları olabilir ve yönetim belli bir mağazadaki belli bir dönemdeki ciroyu tahmin etmek isteyebilir. Bunun sonucu olarak kurum bazı politikaları daha uygun belirleyebilecektir.

Amaç kabaca tespit edildikten sonra bunun daha ayrıntılı ve operasyonel hale getirilmesi gerekmektedir. Yani kaba bir amaçtan ziyade giridisi çıktısı belli değerlerle temsil edilen bir amaç haline dönüştürülmelidir.

Amaç belirlendikten sonra bu amaca ulaşmak için ya da bu amaç ile ilişkili olabilecek özellikleri (features) tespit etmemiz gereklidir. Örneğin bir öğrencinin başarısına etki eden faktörler nelerdir?

- Öğrencinin zeka düzeyi
- Öğrencinin derslere devam oranı
- Öğrencinin geçmişteki not ortalaması
- Öğrencinin okul ile evi arasındaki uzaklık
- Öğrencinin ailesinin ekonomik durumu
- Öğrencinin ailesinin eğitim düzeyi
- Öğrencinin anne babasının ayrı olup olmadığı
- Öğrencideki sağlık problemleri
- Ailedeki sağlık problemleri
- Öğrencinin sınıf dışında toplam günlük oratalama çalışma zamanı
- Öğrencinin daha önce nereden geldiği
- Öğrencinin hobilerine ayırdığı zaman

Bir eczandanın curosunu etileyerek özellikler de şunlar olabilir:

- Eczandanın önünden günlük ortalama geçen insan sayısı
- Eczandanın bulunduğu bölge (kategorik)
- Eczandanın büyülüğu
- Eczandanın yanında bir sağlık kurumunun olup olmadığı bilgisi (kategorik)
- Eczandanın yakınındaki eczane sayısı
- Eczandanın güzellik ürünü satıp satmadığı (kategorik)
- Eczanın takviye ürünler satıp satmadığı (kategorik)
- Eczandanın büyülüğu

Bu faktörler belirlendikten sonra mümkün olduğu kadar operasyonel hale getirilmelidir. Operasyonel demek bir faktörü ölçülebilir bir biçimde ifade etmek demektir. Örneğin bir mağazanın albenisi operasyonel bir özellik değildir. Ancak bunu likert tarzı bir ölçükle operasyonel hale getirebiliriz.

Amaç ve amaca etki eden faktörler belirlendikten sonra sıra veri toplamaya gelir.

Yapay Sinir Ağları İçin Verilerin Toplanması

Her türlü veri analizinin ilk aşaması verilerin elde edilmesidir. Veriler zaten var olabilir ya da biz bunu başka kaynaklardan edinebiliriz. Gerçekten de pek çok ülkenin devlet kurumu ve çeşitli kurumlar yaptıkları kamuoyu

arastirmalarının sonuçlarını herkes için açmaktadır. Türkiye'de "Türkiye İstatistik Kurumu" web sayfasında çeşitli veriler kullanıcılar için hazır bulundurmaktadır.

(<http://www.tuik.gov.tr/PreTabloArama.do?metod=search&araType=vt>)

Eğer veriler hazır durumda değilse bizim o verileri toplamamız gereklidir. Veri toplama işlemi çeşitli yöntemlerle yapılabilmektedir. Örneğin veri toplama işlemi otomatik ya da manuel bir biçimde yapılabilir. Burada otomatikten kastedilen şey birtakım algılayıcılarla ya da süreçlerle verinin kendiliğinden toplanmasıdır. Örneğin biz yollara sensörler yerleştirerek geçen otomobilin sayısını elde edebiliriz. Benzer biçimde bir vasisaya (tren gibi) binenlerin sayısı turnikeler yoluyla zaten sisteme girilmektedir. Bazen verilerin manuel biçimde anketlerle, yüz yüze konuşma yoluyla vs. elde edilmesi gereklidir.

Yapay sinir ağları ve derin öğrenme ağları için veriler girdi-sonuç (yani dataset_x, dataset_y biçiminde) biçiminde elde edilmelidir. Sonra bu verilere eğitimde kullanılarak kestirimler yapılacaktır.

Yapay Sinir Ağları ve Derin Ağlar İçin Verilerin Kullanımı Hazır Hale Getirilmesi

Toplanan ya da elde edilen veriler tam istenildiği biçimde olmayabilir. Bu nedenle bu veriler üzerinde bazı düzenlemelerinin (preprocessing) yapılması gereklidir. Gerçekten de elde edilen bilgilerdeki bazı özellikler (yani sütunlar) eksik olabilmektedir. Örneğin biz bir kişinin bazı bilgilerini biliyor ama birkaçını bilmiyor olabiliriz. Bu tür eksik veriler genel olarak "NA (not available)" biçiminde gösterilmektedir. Peki sütunlarda eksik veri varsa ne yapılacaktır? İşte eğer sütundaki eksik verilerin miktarı ciddi düzeydeyse tüm sütunun ya da eksik veriye sahip satırların atılması uygun olabilmektedir. Genel olarak %30 bir eksiklik yüksek kabul edilmektedir. %30'dan fazla eksiklikler için çıkartma yöntemi izlenmelidir. Toplam satırların miktarına göre bu satırlar da çıkarılabilir ya da sütun tamamen çıkarılabilir. %10 ile %30 bölgenin gri bölge olduğunu söyleyebiliriz. Ancak %10 ve küçük eksikliklerde eksik sütunların doldurulması genellikle sorun oluşturmaz. Bu doldurma işlemi için iki tipik yöntem "ortalama" ve "mod" yöntemidir. Ortalama yönteminde sütundaki mevcut değerlerin ortalaması eksik verilere kopyalanır. Mod yönteminde ise sütunda en çok tekrarlanan değer eksik değerler olarak doldurulmaktadır. Tabii mod işlemi daha çok kategorik değerler için tercih edilmektedir.

Verilerin yapay sinir ağları ve derin ağlar için kullanıma hazır hale getirilmesi sürecindeki diğer önemli bir konu da "özellik ölçeklemesi (feature scaling)" denen işlemidir. Özellik ölçeklemesine "verilerin normalizasyonu" ya da "verilerin standartizasyonu" da denilmektedir. Eğer bir veri tablosundaki sütunlarda bulunan değerlerin mertebeleri birbirinden çok farklı ise yapay sinir ağları bu durumlarda geç yakınsama sorunuyla karşılaşmaktadır. Ayrıca elde edilen modelin başarısı da düşmektedir. Örneğin bir evin fiyatının tahmin edilmesi örneğinde sütunların birinde evin kira geliri 1000'li mertebede iken diğer bir sütundaki evin yaşı çok küçük bir mertebededir. Bu durum sakıncalı kabul edilmektedir. (Aslında normalizasyonun neden gerektiği sezgisel olarak anlaşılabilir. Bir nörona giren değerlerin ağırlıklarla çarpılarak toplandığını biliyorsunuz. Bu durumda yüksek mertebedeki değerler düşük mertebedeki değerlerin etkisini azaltabilecektir.)

Özellik ölçeklemesi için çeşitli yöntemler kullanılmaktadır.

Standart Ölçekleme (Standard Scaling): Bu yöntemde sütunun ortalaması ve standart sapması bulunur. Sonra sütundaki her değer ortalamadan çıkartılıp standart sapmaya bölünerek yeniden sütuna yazılır.

$$\text{yeni } x = \frac{x - \bar{x}}{\sigma_x}$$

Bu işlemi aşağıdaki matris için numpy'da yapalım:

```
import numpy as np

def standard_scale(dataset):
    for col in range(dataset.shape[1]):
       冷data = dataset[:, col]
        dataset[:, col] = (冷data - np.mean(冷data)) / np.std(冷data)
```

```

training_dataset_x = np.loadtxt('training_data.csv', delimiter=',')
standard_scale(training_dataset_x)
print(training_dataset_x)

```

Buradaki "training_data.csv" dosyası da örneğin aşağıdaki gibi olabilir:

```

1,10,13400
2,12,14200
4,4,15000
2,6,12000
4,6,11700
8,9,14200
3,1,34444

```

Sonuç da şöyledir:

```

[[ -1.14096529  0.90267093 -0.40588721]
 [ -0.67115606  1.47709789 -0.29838776]
 [  0.26846242 -0.82060994 -0.19088831]
 [ -0.67115606 -0.24618298 -0.59401125]
 [  0.26846242 -0.24618298 -0.63432354]
 [  2.14769938  0.61545745 -0.29838776]
 [ -0.20134682 -1.68225038  2.42188583]]

```

Bu işlem aslında daha pratik bir biçimde sklearn.preprocessing modülündeki StandardScaler sınıfıyla yapılmaktır. Bu sınıf türünden bir nesne yaratıp önce söz kousu matrisi fit metoduyla nesneye veririz. Sonra da fit_transform işlemi ile dönüştürmeyi yaparız.

```

import numpy as np

training_dataset_x = np.loadtxt('training_data.csv', delimiter=',')

from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
training_dataset_x = ss.fit_transform(training_dataset_x)
print(training_dataset_x)

```

Sınıfın mean_, var_ ve scaler_ isimli elemanları bize sırasıyla sütun ortalamalarını, varyanslarını ve son scale edilmiş değerleri verir. StandardScaler sınıfının fit metodu bizden matrisi almakta ve ona ilişkin sütun standart ölçekteme değerlerini elde edip onu saklamaktadır. Aslında bu işlemleri tek adımda fit_transform yerine iki adımda önce fit sonra transform biçiminde de yapabildik. Örneğin:

```

import numpy as np

training_dataset_x = np.loadtxt('training_data.csv', delimiter=',')

from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
ss.fit(training_dataset_x)
training_dataset_x = ss.transform(training_dataset_x)
print(training_dataset_x)

```

Burada fit işlemi aslında yalnızca sütunların ortalamalarını ve varyanslarını bulup sınıfın mean_ ve var_ örnek özniteliklerinde saklamaktadır. transform ise bu değerleri kullanarak gerçek dönüştürmeyi yapar.

Normal olarak test veri kümesinin de aynı biçimde ölçeklendirilmesi gerekmektedir. Ancak test veri kümesini kendi sütunlarına göre ölçeklendirmek yerine eğitim veri kümesindeki verilere göre ölçeklendirmek gerekmektedir. Bu nedenle fit ile transform işlemini ayırmak, fit işlemini eğitim veri kümesi için bir kez yapıp transform işlemini hem eğitim veri külesi için hem de test veri kümesi için yapmak daha uygun bir yöntemdir. Örneğin:

```

import numpy as np

training_dataset_x = np.loadtxt('training_data.csv', delimiter=',')
test_dataset_x = np.loadtxt('test_data.csv', delimiter=',')

from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
ss.fit(training_dataset_x)
training_dataset_x = ss.transform(training_dataset_x)
test_dataset_x = ss.transform(test_dataset_x)

print(training_dataset_x)
print()
print(test_dataset_x)

```

Tabii eğer böyle bir ölçeklendirmeyle model oluşturulmuşsa predict işlemi yapılrken predict fonksiyonuna verilecek değerlerin de aynı dönüştürmeye sokulması gereklidir. Örneğim:

```
result = model.predict(ss.transform(a))
```

Min-Max Ölçeklendirmesi: Bu ölçeklendirmede sütunun en küçük ve en büyük elemanları bulunur. Bu elemanlara göre ölçeklendirme yapılır:

$$\text{Yeni } x = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

Buradan elde edilen değerler her zaman 0 ile 1 arasındadır. `sklearn.preprocessing` modülündeki `MinMaxScaler` sınıfı bu işlemi otomatik olarak yapmaktadır. Sınıfın kullanımı tamamen `StandardScaler` sınıfında olduğu gibidir.

Mutlak Max Ölçeklendirmesi (Absolute Max Scaling): Bu ölçeklendirmenin formülü şöyledir:

$$\text{Yeni } \text{değ} = \frac{x}{|\max(x)|}$$

Bu ölçeklendirme yöntemi `sklearn.preprocessing.MaxAbsScaler` sınıfıyla temsil edilmiştir. Sınıfın kullanılması benzer biçimdedir.

Eğer sütun verileri normal dağılmışsa standart ölçeklendirme daha uygun olmaktadır. Böyle bir normalik söz konusu değilse ve sütundaki verilerin en büyük değeri önceden biliniyorsa Min-Max ölçeklendirmesini kullanmak uygun olur. Biz de kursumuzda hep bu iki ölçeklendirmeyi kullanacağımız. Örneğin pixel verileri söz konusu olduğunda pixel renk değerleri [0-255] arasında olduğu için doğrudan sütunu 255 değerine bölgerek Min-Max ölçeklendirmesi yapacağız.

Daha önceden kategorik sütunların "one hot encoding" denilen teknikle sayısallaştırılması gerektiğini görmüştük. Pekiyi kategorik sütunu oluşturan kategorilerde çok fazla seçenek varsa ne yapılacaktır? Örneğin haftanın günleri için 7 seçenek vardır. Bu durumda bu özellik için "one hot encoding" biçiminde tabloya 7 sütun eklenecektir. Ya kategoriler yüzlerce olsaydı? Bunun için şu sezgisel yöntemler önerilmektedir:

- Eğer depolama ve işlem konusunda yeterli kaynak varsa fazla kategorilerin de "one hot encoding" yapılması performansı artırır
- Eğer kaynak yetersizliği varsa çok kategori az kategoriye dönüştürülebilir. Yani kendi aralarında gruplanabilir. Ya da "one hot encoding" işlemi o kategorik veri için hiç yapılmayabilir. (Kategorik bir sütuna "one hot encoding" uygulamazsa yine de modelimiz çalışır ancak performans düşer.)

Yapay Sinir Ağlarının ve Derin Ağların Eğitilmesinde "Overfitting" ve "Underfitting" Olgusu

Yapay sinir ağlarının ve derin ağların eğitilmesinde çok karşılaşılan iki terim "overfitting" ve "underfitting" terimleridir. "Overfitting" kabaca modelin eğitimde iyi performans gösterdiği halde test kümesinde ya da gerçek hayatı düşük bir performans göstermesi anlamına gelmektedir. Yani eğitim sırasında model bir şeyi öğrenmiş olabilir ancak öğrendiği şey bizim arzu ettiğimiz şey olmayı bilir. Bazen eğitim veri kümesi çok yanlı (bias) seçilmiştir olabilir. Bu durumda eğitim sırasında ağ başka şeyler öğrenmiş olabilir. Eğitim veri kümesinin yanlış olması "overfitting" olgusunun önemli kaynaklarından biridir. Eğitim kümesinin değişkenliğinin (variance) az olması da bir "overfitting" kaynağı olabilmektedir. Düşük değişkenlik ana kütleyi tam olarak temsil edemiyor olabilir.

Peki "overfit" durumunu nasıl anlarız? İşte eğitim sırasında epoch ilerledikçe eğer metrik değerleri sürekli iyileşiyorsa ve çok iyi bir noktaya geliyorsa bu şüpheli bir durum olabilir. Bazen "overfitting" durumunda önce metrik değerleri iyileşir, sonra kötüleşmeye başlar. Bu tür durumlarda eğitimin erken sonlandırılması gerekebilir. Tabii "overfitting" olgusuna yol açan önemli nedenlerden biri de eğitim veri kümesinin yetesiz olmasıdır. Eğitim veri kümesi artırıldıkça overfitting kendiliğinden ortadan kalkma eğilimi gösterir.

Overfitting olgusunun tersine "underfitting" denilmektedir. Bu olguda da tam tersine eğitim veri kümesinin varyansı çok yüksektir. Eğitim veri kümesindeki metrik değerler bir türlü iyileşmemektedir. Underfitting'in de pek çok nedeni olabilir. Örneğin faktörlerin (yani tablo sütunlarının) yanlış ve eksik seçilmesi, eğitim veri kümesinin azlığı gibi. Genellikle "underfitting" durumunda yapay sinir ağ için kurduğumuz modellen de şüphe edilmelidir. Örneğin katmanlardaki nöron azlığı, katman sayılarının azlığı model problemleridir.

Yapay Sinir Ağlarında Kestirim Problemlerinin Sınıflandırılması

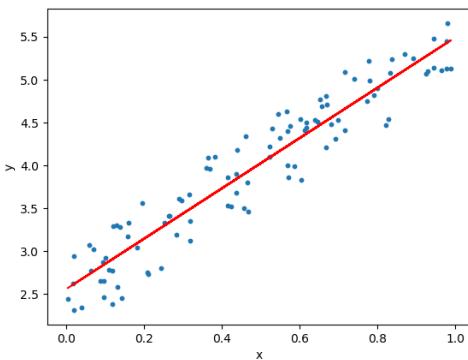
Daha önce de belirtildiği gibi aslında yapay sinir ağları kestirimde bulunmak için kullanılmaktadır. Yapay sinir ağları genel olarak eğitim veri kümesi yoluyla $y = f(x)$ biçiminde bir f fonksiyonun bulunması süreci ile ilgilidir. Tabii buradaki f fonksiyonu doğrusal olmayan karmaşık bir fonksiyondur. Aslında girdi ile çıktı arasında ilişki kurma istatistikte "regresyon" terimiyle ifade edilmektedir. O halde aslında yapay sinir ağları regresyon problemlerinin çözülmesinde kullanılan bir tekniktir. Tabii regresyon problemleri aslında yalnızca yapay sinir ağlarıyla değil istatistiksel ve diğer makine öğrenmesi teknikleriyle de çözülebilmektedir. Ancak bu tekniklerin her biri farklı durumlarda ve koşullarda daha iyi sonuç verme eğilimindedir. Yapay sinir ağları özellikle çok sayıda girdinin bulunduğu, çıktılarının birbirlerine etkileyıldığı karmaşık regresyon problemlerinde tercih edilmektedir. O halde biz öncelikle burada regresyon problemleri konusunda biraz daha bilgi edinelim.

Girdiyle çıktı arasında ilişki kurma sürecine istatistikte "regresyon (regression)" denilmektedir. Başka bir deyişle regresyon bir fonksiyon bulma sürecidir. Regresyon kestirim amacıyla kullanılır. Yani elimizde birtakım geçmiş veriler vardır. Biz de gelecekte durumun ne olabileceğiyle ilgili karar vermek isteriz. Regresyon analizi istatistikten bir alanıdır ve makine öğrenmesinin de önemli alt konulardan birini oluşturmaktadır. Bu nedenle regresyon konusu kursumuzda ayrı bir başlık halinde ileride ele alınacaktır. Ancak biz burada regresyon problemlerinin yapay sinir ağları ve derin ağlarla çözülmesi üzerinde duracağız. Regresyon problemlerinin çözümü için çeşitli istatistiksel teknikler de kullanılmaktadır. Tabii yapay sinir ağları ve derin öğrenme de aslında regresyon problemlerini çözmek için bir alternatif sunmaktadır.

Aslında yapay sinir ağlarında ya da istatistikte değişken (feature) sayılarının model üzerinde önemli bir etkisi yoktur. Yani $y = f(x)$ fonksiyonunda x tek bir x de olabilir ya da bir vektör de olabilir. x 'in bir vektör olması çok değişken olduğu (yani birden fazla sütun olduğu) anlamına gelmektedir. Ancak tek değişken için kullanılan yöntemlerle çok değişken için kullanılan yöntemler arasında fazlaca bir fark yoktur. İnsan algısı en iyi iki boyut üzerinde performans gösterdiği için genellikle model örnekleri kartezyen sistemdeki grafiksel çizimlerle tek değişkenli biçimde açıklanmaktadır. Aslında çok değişkenli biçimde tek değişkenli biçimden fazlaca bir fark yoktur. Ancak değişkenlerin üstleri genel olarak matemetiksel modelleri farklılaştırılmaktadır. Bu anlamda doğrusal (linear) modellerle doğrusal olmayan (nonlinear) modeller arasında önemli farklılıklar bulunabilmektedir.

Istatistikte regresyon modelleri çeşitli ölçülere göre çeşitli sınıflara ayrılmaktadır. Bu regresyon çeşitlerinin bazıları şunlardır:

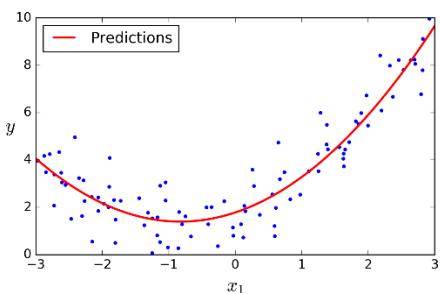
- Doğrusal Regresyon (Linear Regression): Girdi çıktı arasındaki ilişkinin doğrusal olduğu kabulüyle yapılan regresyon analizine denilmektedir.



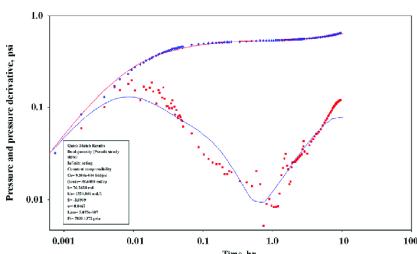
Doğrusal regresyon $y = mx + n$ doğrusunun bulunması işlemidir.

- Polinomsal Regresyon (Polynomial Regression): Verilerin grafiğini çizdiğimizde ilişkinin doğrusal olup olmadığı hemen gözle görülebilemektedir. Bağımsız değişkenin üssü 1'den büyükse bu tür regresyon modellerine polinomsal regresyon modelleri denilmektedir. Başka bir deyişle polinomsal regresyon aşağıdaki gibi bir fonksiyonun bulunması sürecidir:

$$y = a_0x^0 + a_1x^1 + a_2x^2 + \dots + a_nx^n$$



- Doğrusal Olmayan Regresyon (Nonlinear Regression): Her ne kadar polinomlar doğrusal değilse de "doğrusal olmayan" regresyon denildiğinde genellikle üstel, logaritmik ve trigonometrik eğriler anlaşılmaktadır. Örneğin sigmoid fonksiyonu polinomsal değil doğrusal olmayan bir regresyon modelini akla getirmektedir.



Lojistik Regresyon (Logistic Regression): Çıktının sürekli bir değer olmadığı ve kategorik olduğu durumlarda uygulanan regresyon türüdür. Tipik lojistik regresyon çıktıının 1 ya da 0 gibi ikili değerlere sahip olduğu durumlardır. (Örneğin çıktı "evli mi bekar mı", "hasta mı sağlıklı mı", "film iyi mi kötü mü" gibi iki seçenekten biri olabilmektedir.) Ancak zamanla bu terim genişletilmiştir. Çıktının ikiden fazla kategoriye ayrıldığı (multinomial logistic regression) durumlar için de lojistik regresyon terimi kullanılmaktadır. Ayrıca "sıralı lojistik regresyon (ordinal logistic regression)" denilen bir lojistik regresyon modelinde girdi ve çıktılar kategorik değil sıralı ölçeklere ilişkin olabilmektedir.

Regresyonlar bağımlı ve bağımsız değişken sayılarına göre de sınıflandırılmaktadır:

- Çoklu Regresyon (multiple regression) bağımsız değişken sayısının (yani girdilerin) birden fazla olduğu fakat bağımlı değişken sayısının (yani çıktıının) bir tane olduğu regresyon modelleri için kullanılan bir terimdir. Örneğin bir otomobilin 8 özelliğinden onun yakıt harcamasının tahmin edilmek istediği regresyon modeli çoklu regresyondur. Ya da örneğin yukarıda yapmış olduğumuz birtakım bilgilerden kişinin diyabet hastası olup olmadığıının belirlenmesi "çoklu lojistik regresyon" örneğidir.

- Eğer regresyon modelinde bağımlı değişkenin sayısı birden fazlaysa (yani çıktıı birden fazlaysa) buna da "çok değişkenli (multivariate)" regresyon denilmektedir. Örneğin öğrencinin bazı girdi bilgileri olsun (bağımsız değişkenler) biz de onun sınavda alacağı notu ve ortalama kaç saat uyuduğunu tahmin etmek isteyelim. Tabii burada çıktılar birbirlerinden bağımsız ve teker teker olarak da ele alınabilirdi. O zaman çoklu regresyon söz konusu olurdu. Fakat çok değişkenli regresyonlarda aynı anda birden fazla çıktıının değişiminin belirlenmesi hedeflenmektedir. Yani örneğin 15 tane biyomedikal tetkike bakarak biz bir kişinin "diyabetli olup olmadığını", "kalp hastası olup olmadığını", "hipertansiyonun olup olmadığını" ayrı ayrı çoklu lojistik regresyonla anlamaya çalışabiliriz. Ancak bu üç hastalık birbirlerini de etkiliyor olabilir. O halde bu üç hastalığın birlikte değerlendirilmesi gereklidir. İşte bu durum çok değişkenli lojistik regresyon olarak modellenebilir.

Yapay Sinir Ağları ve Derin Ağlarla Regresyon Analizinin İstatistiksel Regresyon Analizinden Farklılıklar

Yukarıda da belirttiğimiz gibi aslında regresyon modellerini çözmek için değişik teknikler kullanılabilmektedir. Kursumuzun ilerleyen zamanlarında regresyon problemlerinin başka tekniklerle çözümü üzerinde de durulacaktır. Pekiyi yapay sinir ağları ve derin ağlar klasik istatistiksel tabanlı regresyon modellerine göre hangi durumlarda avantaj sağlamaktadır? Bu durumları şöyle ifade edebiliriz:

- Bağımsız değişken sayısının (girdi sayılarının) fazla olduğu durumlarda
- Bağımlı değişken sayısının fazla olduğu durumlarda
- Bağımsız değişkenler arasında ilişkilerin karmaşık olduğu durumlarda
- Bağımsız değişkenlerle bağımlı değişkenler arasındaki ilişkinin belirsiz ve karmaşık olduğu durumlarda.
- Bağımsız değişkenlerin bazlarının kategorik olduğu durumlarda

Bu tür durumlarda istatistiksel regresyon yöntemleri genellikle sinir ağı modellerine göre hem daha düşük performans göstermeye hem daha fazla bilgisayar zamanı almaktadır hem de modellemesi daha zor olmaktadır.

Tabii aslında klasik istatistiksel tabanlı regresyon modellerinin yapay sinir ağları ve derin ağlardan daha yüksek performans göstermesi de söz konusu olabilmektedir. Özellikle girdi ile çıktıı arasındaki ilişkilerin çok belirgin olduğu durumlarda istatistiksel regresyonlar tercih edilebilmektedir. Yine benzer biçimde elde çok veri olduğu durumda yapay sinir ağlarının eğitilmesi bir sorun oluşturabilmektedir. Bu durumlarda istatistiksel regresyonlar yapay sinir ağlarına tercih edilebilmektedir.

Yapay Sinir Ağları ve Derin Ağlardaki Regresyon Modelinin Oluşturulması

Yapay sinir ağlarında ve derin ağlarda regresyon modeli oluşturulurken genellikle bazı sezgisel yöntemler izlenmektedir. Tabii daha önceden de bahsedildiği gibi aslında veri biliminde genel yaklaşım her zaman o andaki modele ve amaca özgü olarak değişebilmektedir. Ancak yine de çeşitli problem grupları için kaba bazı model varsayımları yapılmaktadır. Regresyon modeli için bunlar şöyle özetlenebilir:

- Regresyon modelinde genellikle hidden katmanlar için "relu", çıktı katmanı için regresyon modeli lojistik ise "sigmoid", lojistik değilse "linear" transfer fonksiyonları tercih edilmektedir. (Linear transfer fonksiyonu tamamen girdinin hiçbir işleme sokulmadan çıktıı olarak verildiği transfer fonksiyondur. Dense katmanındaki default transfer fonksiyonu böyledir.)
- Modeldeki optimizer algoritması için genellikle "stochastic_gradient_descent (sgd)", "adam", ya da "rmsprop" seçilmektedir.

- Modelin loss fonksiyonu (amaç fonksiyonu) için genellikle lojistik regresyon modelleri için "binary_crossentropy" lojistik olmayan regresyon modelleri için de "mean_squared_error (mse)" tercih edilmektedir.
- Validation işlemi için kullanılacak metrikler de genellikle lojistik regresyon modellerinde "binary_accuracy" lojistik olmayan regresyon modelleri için mean_squared_error (mse) olarak ya da "mean_absolute_error (mae)" seçilmektedir.

Regresyon modeli için mimari konusunda da genellikle şunlar uygulanmaktadır:

- Katmanlar Dense (yani tüm nöron çıktılarının diğerlerine bağlılığı) olarak alınır.
- Başlangıçta katmandaki nöron sayıları 100-200 gibi orta değerde tutulur.
- Önce tek hidden katman denenir, eğer performans düşük kalırsa hidden katman sayısı ikiye, üçe dörde duruma göre daha yüksek derecelere çıkartılır.
- Fazla miktarda katmanla hala iyileşme sağlanamıyorsa katmanlardaki nöron sayıları artırılır.
- Hala yeterli bir performans elde edilmiyorsa bu durumda özelliklerden (features) ya da eğitim için kullanılan veri miktarının azlığından şüphelenilebilir.

Yapay Sinir Ağları ve Derin Ağlarda Regresyon Modeli Örnekleri

Bu bölümde yapay sinir ağları ve derin ağlarla çeşitli regresyon örnekleri verilecektir.

Polinomsal Regresyon Örneği

$y = f(x)$ biçiminde tek değişkenli polinomsal bir fonksiyon için örnek bir yapay sinir ağı modeli oluşturalım. Bu örnekte $y = f(x)$ eğrisi baştan belirlenmiş olsun ve biz bu fonksiyona ilişkin gerçek verileri kullanarak ağıımızı eğitelim. Fonksiyonu aşağıdaki gibi belirleyelim:

$$y = 3x^2 + 5$$

Rastgele 100000 tane x değeri için y değerlerini şöyle hesaplayabiliriz:

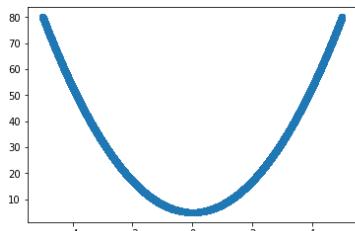
```
import matplotlib.pyplot as plt

def func(x):
    return 3 * x ** 2 + 5

import numpy as np

x = np.random.uniform(-5, 5, size=(100000, 1))
y = func(x)

plt.scatter(x, y)
```



Şimdi kümemizi training ve test olmak üzere ikiye ayıralım:

```
from sklearn.model_selection import train_test_split
training_set_x, test_set_x, training_set_y, test_set_y = train_test_split(x, y, test_size=0.2)
```

Şimdi ağımızın oluşturulmasını:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(200, input_dim=1, activation='relu'))
model.add(Dense(1, activation='linear'))

model.compile(optimizer='sgd', loss='mean_absolute_error', metrics=['mean_squared_error'])
model.fit(training_set_x, training_set_y, batch_size=3, epochs=1, validation_split=0.2)
```

Genellikle polinomsal regresyon modelleri için iki hidden katman yeterli olmaktadır. Ancak yukarıda da belirtildiği gibi programcı sonuçlara bakarak katman sayısını duruma göre artırıp daha iyi bir sonuç elde etmeye çalışabilir. Eğitim işleminden sonra modeli test edelim:

```
loss, mse = model.evaluate(test_set_x, test_set_y)
print('loss = {}, mean squared error = {}'.format(loss, mse))
```

Nihayet bazı değerler için predict işlemi yapabiliriz:

```
a = np.array([[2], [2.4], [2.8], [3.2]])
yval1 = model.predict(a)
yval2 = func(a)

print(yval1)
print(yval2)
```

Programın tamamı şöyledir:

```
import matplotlib.pyplot as plt

def func(x):
    return 3 * x ** 2 + 5

import numpy as np

x = np.random.uniform(-5, 5, size=(100000, 1))
y = func(x)

plt.scatter(x, y)

from sklearn.model_selection import train_test_split
training_set_x, test_set_x, training_set_y, test_set_y = train_test_split(x, y, test_size=0.2)
```

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(200, input_dim=1, activation='relu'))
model.add(Dense(1, activation='linear'))

model.compile(optimizer='sgd', loss='mean_absolute_error', metrics=['mean_squared_error'])
model.fit(training_set_x, training_set_y, batch_size=3, epochs=1, validation_split=0.2)

loss, mse = model.evaluate(test_set_x, test_set_y)

print('loss = {}, mean squared error = {}'.format(loss, mse))

a = np.array([[2], [2.4], [2.8], [3.2]])
yval1 = model.predict(a)
yval2 = func(a)

print(yval1)
print(yval2)

```

Şöyledir bir çıktı elde edilmiştir:

```

[[16.071669] [20.818136] [26.996428] [33.826572]]
[[17. ] [22.28] [28.52] [35.72]]

```

Bu değerler gerçek değerlere yakın olduğu halde çok da iyi değildir. Şimdi katman sayısını artırıp bir kez daha deneyelim. Yeni katmanlar şöyle olsun:

```

model = Sequential()
model.add(Dense(200, input_dim=1, activation='relu'))
model.add(Dense(200, activation='relu'))
model.add(Dense(1, activation='linear'))

```

Şimdi elde ettiğimiz sonuç şöyledir:

```

[[17.492794] [22.642626] [28.552927] [35.271103]]
[[17. ] [22.28] [28.52] [35.72]]

```

Göründüğü gibi değerler daha tatmin edici olmuştur. Şimdi epoch sayısını da biraz artırıralım:

```
model.fit(training_set_x, training_set_y, batch_size=3, epochs = 20, validation_split=0.2)
```

Bu problemde epoch değerinin artırılması kestirimini daha iyi hale getirmemiştir. Şimdi epoch değerini yine 1'de tutup hidden katman sayısını 3'e çıkartalım:

```

model = Sequential()
model.add(Dense(200, input_dim=1, activation='relu'))
model.add(Dense(200, activation='relu'))
model.add(Dense(200, activation='relu'))
model.add(Dense(1, activation='linear'))

```

Buradan da daha tatmin edici bir sonuç elde edemedik. Pekiyi şimdi de hidden katman sayısını 2'de tutalım ama nöron sayılarını artırıralım:

```

model = Sequential()
model.add(Dense(400, input_dim=1, activation='relu'))
model.add(Dense(400, activation='relu'))
model.add(Dense(1, activation='linear'))

```

[[17.02856] [22.115063] [28.192284] [35.07256]]

[[17.] [22.28] [28.52] [35.72]]

Bu model için katman sayısını artırmaktansa nöron sayısının artırılmasının daha etkili olduğu görülmektedir.

Lojistik Olmayan Regresyon Örneği: Otomobillerin Mil Başına Yaktıkları Yakıtı Tahmin Etme

Bu örnek eğitimde çokça karşılaşılan örneklerden biridir. Veriler 70'li yılların sonlarına doğru toplanmıştır. Dolayısıyla değerler o yıllara özgüdür. Örnek Tensor Flow'un kendi sitesinde bulunmaktadır

(https://www.tensorflow.org/tutorials/keras/basic_regression?hl=tr). Ancak biz burada örneği kursta aşina olduğumuz biçimde yapacağız. Şimdi aşama aşama örneği gerçekleştirmeye çalışalım.

1) İlk önce örnek için veri setini bir dosya biçiminde elde etmemiz gereklidir. Veri kümesi aşağıdaki bağlantından indirilebilir:

<https://archive.ics.uci.edu/ml/datasets/auto+mpg>

Burada "auto-mpg.data" ve "auto-mpg.names" dosyaları indirilebilir. Gerçek veri dosyası "auto-mpg.data" dosyasıdır. "auto-mpg.names" dosyası içerisinde açıklamalar vardır. Data dosyasındaki sütunlar şunlardır:

MPG: Mil başına yakılan galon miktarı (1 galon 3.78 litredir)

Silindir Sayısı

Motor Hacmi

Beygir Gücü

Ağırlık

100 km/h ya çıkma süresi (ivmelenmesi)

Orijin (Kategorik 1: USA, 2: Europe, 3: Japan)

Marka

Sütunlar dosyada birbirlerinden SPACE karakterleriyle ayrılmıştır. Ancak Markadan önce bir tane TAB karakter bulunmaktadır. Verilerin örnek görünümü şöyledir:

1	18.0	8	307.0	130.0	3504.	12.0	70	1	"chevrolet chevelle malibu"
2	15.0	8	350.0	165.0	3693.	11.5	70	1	"buick skylark 320"
3	18.0	8	318.0	150.0	3436.	11.0	70	1	"plymouth satellite"
4	16.0	8	304.0	150.0	3433.	12.0	70	1	"amc rebel sst"
5	17.0	8	302.0	140.0	3449.	10.5	70	1	"ford torino"
6	15.0	8	429.0	198.0	4341.	10.0	70	1	"ford galaxie 500"
7	14.0	8	454.0	220.0	4354.	9.0	70	1	"chevrolet impala"
8	14.0	8	440.0	215.0	4312.	8.5	70	1	"plymouth fury iii"
9	14.0	8	455.0	225.0	4425.	10.0	70	1	"pontiac catalina"
10	15.0	8	390.0	190.0	3850.	8.5	70	1	"amc ambassador dpl"
11	15.0	8	383.0	170.0	3563.	10.0	70	1	"dodge challenger se"
12	14.0	8	340.0	160.0	3609.	8.0	70	1	"plymouth 'cuda 340"
13	15.0	8	400.0	150.0	3761.	9.5	70	1	"chevrolet monte carlo"
14	14.0	8	455.0	225.0	3086.	10.0	70	1	"buick estate wagon (sw)"
15	24.0	4	113.0	95.00	2372.	15.0	70	3	"toyota corona mark ii"
16	22.0	6	198.0	95.00	2833.	15.5	70	1	"plymouth duster"
17	18.0	6	199.0	97.00	2774.	15.5	70	1	"amc hornet"
18	21.0	6	200.0	85.00	2587.	16.0	70	1	"ford maverick"
19	27.0	4	97.00	88.00	2130.	14.5	70	3	"datsun pl510"
20	26.0	4	97.00	46.00	1835.	20.5	70	2	"volkswagen 1131 deluxe sedan"
21	25.0	4	110.0	87.00	2672.	17.5	70	2	"peugeot 504"
22	24.0	4	107.0	90.00	2430.	14.5	70	2	"audi 100 ls"
23	25.0	4	104.0	95.00	2375.	17.5	70	2	"saab 99e"
24	26.0	4	121.0	113.0	2234.	12.5	70	2	"bmw 2002"
25	21.0	6	199.0	90.00	2648.	15.0	70	1	"amc gremlin"
26	10.0	8	360.0	215.0	4615.	14.0	70	1	"ford f250"
27	10.0	8	307.0	200.0	4376.	15.0	70	1	"chevy c20"
28	11.0	8	318.0	210.0	4382.	13.5	70	1	"dodge d200"
29	9.0	8	304.0	193.0	4732.	18.5	70	1	"hi 1200d"
30	27.0	4	97.00	88.00	2130.	14.5	71	3	"datsun pl510"
31	28.0	4	140.0	90.00	2264.	15.5	71	1	"chevrolet vega 2300"
32	25.0	4	113.0	95.00	2228.	14.0	71	3	"toyota corona"
33	25.0	4	98.00	?	2046.	19.0	71	1	"ford pinto"

Bazı satırların bazı sütunlarında eksik veriler vardır. Bunlar '?' ile gösterilmiştir. Eksik veriler çok azsa o satırlar tamamen atılabilir. Şimdi biz yukarıdaki veri dosyasını numpy kullanarak okuyalım ve '?' olan satırları atalım. Dosya incelediğinde '?' lerinin 3'üncü indeksi sütunlarda olduğu görülmektedir. Bu işlem şöyle yapılabilir:

```

import numpy as np

dataset = np.loadtxt('auto-mpg.data', usecols=range(8), dtype='object')
dataset = dataset[dataset[:, 3] != '?', :].astype('float32')

```

Artık elimizde '?' lerinden arındırılmış ve float32 formatına dönüştürülmüş bir ndarray nesnesi vardır.

Anımsanacağı gibi veri kümemizde "Orijin" 1, 2, 3 biçiminde kategorik bir alandır. Bizim bu kategorik alanı "one hot encoding" yapmamız gereklidir. Bu işlem şöyle yapılabilir:

```

from sklearn.preprocessing import OneHotEncoder

hotEncoder = OneHotEncoder(categorical_features = [6])
nd_dataset_x = hotEncoder.fit_transform(nd_dataset_x).toarray()

```

Şimdi de veri kümemizdeki sütunların mertebeleri arasında ciddi farklar olduğu için "özellik ölçeklendirmesi (feature scaling) yapalım.

```

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
ss.fit(dataset_x)
dataset_x = ss.transform(dataset_x)

```

Şimdi veri kümemizi "training set" ve "test set" biçiminde ikiye ayıralım:

```

from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.20)

```

Veri kümemizdeki sütunların mertebeleri arasında ciddi farklar olduğu için "özellik ölçeklendirmesi (feature scaling) yapalım.

Artık modeli kurabiliyoruz. Tipik olarak iki hidden katmanlı bir model oluşturacağız. Tipik olarak yukarıda da belirtildiği gibi lojistik olmayan regresyon tarzı problemlerde hidden katmanlardaki transfer fonksiyonlarını "relu" ve çıkış katmanındaki transfer fonksiyonunu da "linear" alacağız. Optimizer algoritması olarak aslında yukarıda belirttiğimiz "stochastic_gradient_descent", "adam" ya da "rmsprop" algoritmalarından herhangi biri seçilebilir. Örneğimizde biz "rmsprop" kullanacağız. Loss fonksiyonu da yine "mean_squared_error" alabiliriz. Metrik değeri olarak da yine aynı fonksiyonu alabiliriz:

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(200, input_dim=training_dataset_x.shape[1], activation='relu', name='Hidden-1'))
model.add(Dense(200, activation='relu', name='Hidden-2'))
model.add(Dense(1, activation='linear', name='Output'))

model.compile('rmsprop', loss='mean_squared_error', metrics=['mean_squared_error',
'mean_absolute_error'])

```

Artık modelimizi eğitebiliriz.

```
history = model.fit(training_dataset_x, training_dataset_y, batch_size=32, epochs=10)
```

Eğitimden elde ettiğimiz history bilgilerine dayanarak epoch temelinde loss değerinin , 'mse' ve 'mae' metrik değerlerinin grafiklerini çizelim:

```

import matplotlib.pyplot as plt

plt.title('Epoch Loss Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Loss'
plt.plot(history.epoch, history.history['loss'])
plt.legend(['loss'])

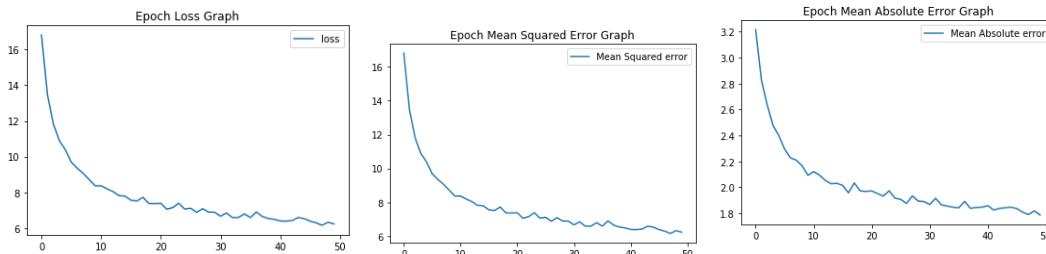
plt.pause(1)

plt.title('Epoch Mean Squared Error Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Mean Squared Error'
plt.plot(history.epoch, history.history['mean_squared_error'])
plt.legend(['Mean Squared error'])

plt.pause(1)

plt.title('Epoch Mean Absolute Error Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Mean Absolute Error'
plt.plot(history.epoch, history.history['mean_absolute_error'])
plt.legend(['Mean Absolute error'])

```



Şimdi modelin testini yapalım:

```

result = model.evaluate(test_dataset_x, test_dataset_y)
print(result)

```

Şimdi de predict işlemi yapalım:

```

predict_dataset_x = np.array([[0, 0, 1, 4., 113., 95., 2228., 14., 71]], dtype='float32')
predict_dataset_x = ss.transform(predict_dataset_x)
result = model.predict(predict_dataset_x)

```

Buradan şu sonuç elde edilmişdir:

```
array([[24.515303]], dtype=float32)
```

Yukarıdaki örneğin tüm kodu bütün olarak aşağıda verilmektedir:

```

import numpy as np

dataset = np.loadtxt('auto-mpg.data', usecols=range(8), dtype='object')
dataset = dataset[dataset[:, 3] != '?', :].astype('float32')

dataset_x = dataset[:, 1:]
dataset_y = dataset[:, 0]

from sklearn.preprocessing import OneHotEncoder

```

```

one_hot_encoder = OneHotEncoder(categorical_features = [6])
dataset_x = one_hot_encoder.fit_transform(dataset_x).toarray()

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
ss.fit(dataset_x)
dataset_x = ss.transform(dataset_x)

from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.20)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(200, input_dim=training_dataset_x.shape[1], activation='relu', name='Hidden-1'))
model.add(Dense(200, activation='relu', name='Hidden-2'))
model.add(Dense(1, activation='linear', name='Output'))

model.compile('rmsprop', loss='mean_squared_error', metrics=['mean_squared_error',
'mean_absolute_error'])
history = model.fit(training_dataset_x, training_dataset_y, batch_size=32, epochs=100)

model.save('autompg.h5')
import pickle

with open('ss.dat', 'wb') as f:
    pickle.dump(ss, f)

import matplotlib.pyplot as plt

plt.title('Epoch Loss Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Loss'
plt.plot(history.epoch, history.history['loss'])
plt.legend(['loss'])

plt.pause(1)

plt.title('Epoch Mean Squared Error Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Mean Squared Error'
plt.plot(history.epoch, history.history['mean_squared_error'])
plt.legend(['Mean Squared error'])

plt.pause(1)

plt.title('Epoch Mean Absolute Error Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Mean Absolute Error'
plt.plot(history.epoch, history.history['mean_absolute_error'])
plt.legend(['Mean Absolute error'])

result = model.evaluate(test_dataset_x, test_dataset_y)
print(result)

predict_dataset_x = np.array([[0, 0, 1, 4., 113., 95., 2228., 14., 71]], dtype='float32')

```

```
predict_dataset_x = ss.transform(predict_dataset_x)
result = model.predict(predict_dataset_x)
```

Bu örnekte biz modelimizi oluşturup eğittik. Daha önceden de belirttiğimiz gibi artık bilgisayarımızı kapattığımızda bu eğitim bilgileri kaybedilecek dolayısıyla yeniden bir eğitim gerekecektir. Tabii yukarıdaki örnekte eğitim veri miktarı da dikkate alındığında kısa sürmektedir. Ancak yüksek verilerle eğitim yapıldığında bu süre ciddi biçimde uzayabilmektedir. O halde bizim modeli fit işleminden sonra diskte bir dosya biçiminde saklamamız gerekebilmektedir. Bu işlem anımsanacağı gibi model sınıfının save metoduyla yapılmaktadır.

```
model.save('automp.h5')
```

Artık biz bu modeli ve eğitim verilerini kolaylıkla yükleyebiliriz:

```
from tensorflow.keras.models import load_model
model = load_model('automp.h5')
```

Pekiyi modeli yükledikten sonra predict işlemi yapmadan önce bizim yine "özellik ölçeklendirmesi (feature scaling)" yapmamız gereklidir. Halbuki bunun için oluşturmuş olduğumuz StandardScalar nesnesini biz herhangi biçimde diskte saklamadık. Bu durumda ne yapabiliriz? İşte "-Python Uygulamaları" kursunda da ele aldığımız gibi- aslında sınıf nesneleri içlerindeki verilerle biz diske bir dosya olarak save edilebilir. Bu işleme nesne yönelimli dillerde "nesnelerin seri hale getirilmesi (object serialization)" denilmektedir. Python'ın standart kütüphanesindeki pickle modülü bu işlemi yapmaktadır. Örneğin:

```
import pickle
with open('ss.dat', 'wb') as f:
    pickle.dump(ss, f)
```

Şimdi de başka bir dosya içeisinde yukarıda save etmiş olduğumuz bilgileri diskten geri yükleyerek predict işlemi yapalım:

```
import numpy as np
from tensorflow.keras.models import load_model
import pickle
model = load_model('automp.h5')
with open('ss.dat', 'rb') as f:
    ss = pickle.load(f)
predict_dataset_x = np.array([[0, 0, 1, 4., 113., 95., 2228., 14., 71
]], dtype='float32')
predict_dataset_x = ss.transform(predict_dataset_x)
result = model.predict(predict_dataset_x)
print(result)
```

Lojistik Olmayan Regresyon Örneği: Boston Housing Price Örneği

Bu örnek çeşitli yazarlar ve öğreticiler tarafından sıkılıkla verilmektedir. Boston'da 70'li yılların sonlarına doğru ev fiyatlarıının tahmin edilmesi için toplanmış veriler vardır. Bunlar kaggle.com sitesinden dosya olarak indirilebilirler. Aslında bu örnek için verilerin dosya biçiminde indirilmesine gerek yoktur. Çünkü tensorflow.keras.datasets içerisinde bu veriler hazır biçimde bulunmaktadır. Biz burada veri tablosundaki sütunların anlamları üzerinde durmayacağız. Doğrudan regresyon modelini kurup sonucu elde etmeye çalışacağız. Bu işlem için örnek program şöyle yazılabılır:

```

import numpy as np

EPOCHS = 300

from tensorflow.keras.datasets import boston_housing

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
boston_housing.load_data()

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
ss.fit(training_dataset_x)
training_dataset_x = ss.transform(training_dataset_x)
test_dataset_x = ss.transform(test_dataset_x)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(100, input_dim=training_dataset_x.shape[1], activation='relu', name='Hidden-1'))
model.add(Dense(100, activation='relu', name='Hidden-2'))
model.add(Dense(1, activation='linear', name='Output'))

model.compile('rmsprop', loss='mse', metrics=['mean_squared_error', 'mean_absolute_error'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=EPOCHS, validation_split=0.2,
batch_size=32)

import matplotlib.pyplot as plt

plt.title('Epoch - Loss (MSE) Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Loss'
plt.plot(range(1, EPOCHS + 1), hist.history['loss'])
plt.legend(['Loss'])

plt.pause(1)

plt.title('Epoch - MAE Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Mean Absolute Error'
plt.plot(range(1, EPOCHS + 1), hist.history['mean_absolute_error'])
plt.legend(['Mean Absolute Error'])

plt.pause(1)

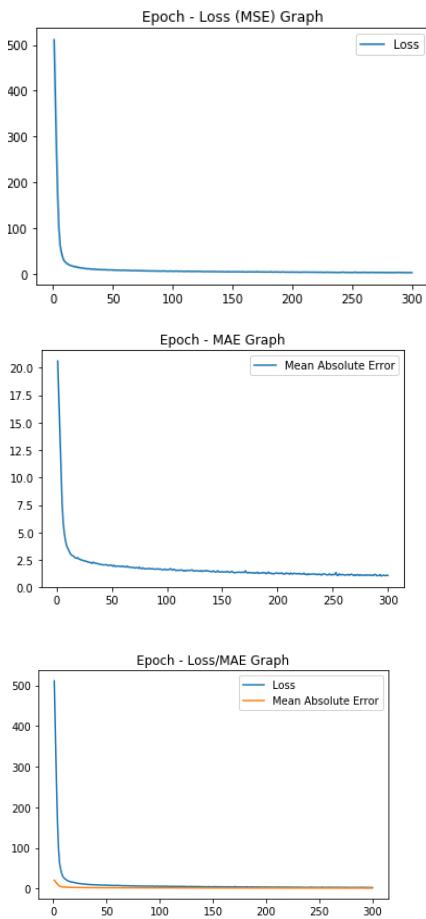
plt.title('Epoch - Loss/MAE Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Mean Absolute Error'
plt.plot(range(1, EPOCHS + 1), hist.history['loss'])
plt.plot(range(1, EPOCHS + 1), hist.history['mean_absolute_error'])
plt.legend(['Loss', 'Mean Absolute Error'])

loss, mse, mae = model.evaluate(test_dataset_x, test_dataset_y, batch_size=32)
print(loss, mse, mae)

predict_x = np.array([[18.0846, 0, 18.1, 0, 0.679, 6.434, 100, 1.8347, 24, 666, 20.2, 27.25,
29.05]])
predict_x = ss.transform(predict_x)
result = model.predict(predict_x)
print(result)

```

Programda oluşan grafikler de şöyledir:



Çok Değişkenli (Multivariate) Regresyon Modelleri

Aslında sinir ağlarında tek değişkenli regresyon modelleri ile çok değişkenli (multivariate) modeller arasında mimari olarak ve parametre belirleme bağlamında bir fark yoktur. Yalnızca çok değişkenli modellerde çıktı katmanındaki nöronların sayısı değişken sayısına kadar olmaktadır. Örneğin biz ev fiyatını tahmininde evin fiyatının yanı sıra, kirasını da tahmin etmek isteyebiliriz. Tek yapacağımız şey çıktı katmanının 1 yerine 2 yapmak olacaktır. Tabii eğitim ve test veri kümelerinde bu iki değerin bulunuyor olması gereklidir.

Yapay Sinir Ağları ve Derin Ağlardaki Sınıflandırma Modelleri

İstatistikte sınıflandırma için "lojistik regresyon" terimi de kullanılmaktadır. (Daha önceden de belirtildiği gibi lojistik regresyon esasen iki değerli (binary) sınıflandırmalar için kullanılan bir terimdir.) Sınıflandırma problemleri kendi aralarında "etiket (label)" ve "sınıf (class)" sayılarına göre gruplandırılmaktadır. Etiket çıktıdaki değişken sayısını belirtmektedir. Örneğin modelin çıktısı "kişinin eğitim durumu" ve "obez olup olmadığı"na ilişkin olsun. Burada iki tane etiket vardır. Ancak etiketler de kendi aralarında iki sınıfı ya da daha fazla sınıfı sahip olabilir. Örneğin eğitim durumu beş sınıfı, obez olup olmama iki sınıfı temsil edilebilir. Bu durumda sınıflandırma modellerini üç gruba ayıralım:

- 1) Tek etiketli iki sınıfı Sınıflandırma Problemleri
- 2) Tek etiketli çok sınıfı sınıflandırma problemleri
- 3) Çok etiketli çok sınıfı sınıflandırma problemleri

Yapay sinir ağları ve derin öğrenmede sınıflandırma modelleri genel olarak üç kısma ayrılmaktadır:

1) Tek Etiketli (Single Label) İkili (Binary) Sınıflandırma Modelleri: Bu modeller tipik lojistik regresyon modelleridir. Daha önce yapmış olduğumuz diyabet modeli buna tipik bir örnektir. Bu modellerde bir tane çıkış vardır. Çıkış

"doğru-yanlış", "var-yok", "hasta sağlam" gibi ikili bir değerdir. Tek etiketli ikili sınıflandırma modellerinde genellikle hidden katmanlardaki aktivasyon fonksiyonları "relu" olarak, çıktı katmanındaki aktivasyon fonksiyonu ise "sigmoid" olarak alınmaktadır. Anımsanacağı gibi sigmoid fonksiyonu 0 ile 1 arasında S şekli biçiminde bir değer vermektedir. Bu durumda çıkış nöronundaki değerin "var-yok", "hasta-sağlıklı", "doğru-yanlış" gibi yorumlanması bu değerin 0.5 gibi bir eşik değerinden büyük ya da küçük olup olmamasına bakılarak yapılır. Tek etiketli ikili modellerdeki optimizasyon algoritması "adam" "sgd" ya da "rmsprop" seçilebilir. loss fonksiyonu genellikle olarak "binary_crossentropy" tercih edilmektedir. Metrik değerler için ise en çok tercih edilen "accuracy", "mean_absolute_error" fonksiyonlarıdır. Bu tür modellerde veri kümelerindeki sütunlar arttıkça hidden katmanlarının sayısının artırılması ve bu katmanlardaki nöronların sayılarının artırılması tavsiye edilmektedir. Yine denemelere iki hidden katman başlanabilir. Duruma göre katmanların sayıları artırılmalıdır. Hidden katmanlardaki nöronlar girdi katmanlarının iki katı biçiminde başlatılabilir.

2) Tek Etiketli (Single Label) Çok Sınıflı (Multiclass) Sınıflandırma Modelleri: Bu modellerde çıkışta tek bir bilgi vardır fakat bu bilgi ikiden fazla sınıfı ayırmaktadır. Örneğin kişinin eğitim durumu, nereli olduğu, hastalığının hangi evrede olduğu gibi. Bu tür modellerde çıktı katmanındaki nöron sayısı toplam sınıf sayısı kadar olur. Örneğin modelimizin çıkışından sonucun A, B, C, D, E sınıflarından hangisine ilişkin olduğunu belirmek istediğimizi düşünelim. Bu durumda çıktı nöronlarının sayısı 5 olacaktır. Aktivasyon fonksiyonu genellikle yine hidden katmanlar için "relu" alınmaktadır. Ancak çıktı katmanındaki aktivasyon fonksiyonu için "softmax" tercih edilir. "Softmax" aktivasyon fonksiyonu çıktı nöronlarının (yani sınıfların) toplam olasılıklarının 1 değerini verdiği bir aktivasyon fonksiyonudur. Başka bir deyişle önceki örnekteki A, B, C, D ve E çıkış nöronlarının değerlerinin toplamı "softmax" aktivasyon fonksiyonu sayesinde 1 edecektir. Böylece bu nöronlardan hangisinin değeri yüksekse o sınıfın seçildiği varsayılmaktadır. Örneğin sonuçta A=0.1, B=0.1, C=0.2, D=0.2, E=0.4 değerleri elde edilmiş olsun. Burada seçilen sınıf E olacaktır. Modeldeki amaç fonksiyonu için ise en uygun olanı "categorical_crossentropy"dir. Optimizasyon algoritması için yine "adam", "sgd" ya da "rmsprop" kullanılabilir. Tek etiketli çok sınıflı modeller için metrik fonksiyonları da genellikle "accuracy" ya da "mean_absolute_error" seçilmektedir. Bu modeller de yine iki hidden katman ile başlatılabilir. Ancak sınıf sayısı çok fazlaysa hidden katmanlarının sayısını gittikçe artırmak gerekebilir. Fazla sınıflı modellerin yüksek miktarda verilerle eğitilmesi önerilmektedir.

3) Çok Etiketli (Multilabel) Modeller: Çok etiketli modeller birden fazla çıktı değişkeninin olduğu modellerdir. Bu modeller de kendi içerisinde "ikili (binary)" ya da çok "sınıflı (multiclass)" olabilir. Genel olarak bu modeller ikili (binary) ve çok sınıflı (multiclass) modellerdeki mimari ile oluşturulmaktadır. Yalnızca çıktılarında birden fazla etiket için nöronlar bulunur.

Aşağıdaki tabloda hangi regresyon modellerinde hangi parametrik değerlerin seçileceği özet olarak belirtilmiştir:

Sınıflandırma Model	Optimizasyon Algoritması	Loss Fonksiyonu	Hidden Katman Aktivasyon Fonksiyonları	Çıktı Katmanı Aktivasyon Fonksiyonları
İkili Sınıflandırma	'rmsprop', 'adam', 'sgd'	'binary_crossentropy'	'relu'	'sigmoid'
Çok Sınıflandırma	'rmsprop', 'adam', 'sgd'	'categorical_crossentropy'	'relu'	'softmax'
Lojistik Olmayan Regresyon	'rmsprop', 'adam', 'sgd'	'mean_squared_error', 'mean_absolute_error'	'relu'	'linear'

Pekiyi birden fazla sınıfın içerisine girebilen sınıflandırma modelleri için ne yapılabilir? Örneğin bir müzik parçasının türünü sınıflandırmak isteyelim. Tür de "pop", "rock", "jazz", "slow", "rap", "new wave" gibi kategorilerin birden fazlasına sahip olabilsin. Bu nasıl bir sınıflandırma modelidir? Bu model aslında çok etiketli bir model olarak düşünülebilir. Yani aslında burada toplamda 6 etiket vardır. Bu 6 etiket "var", "yok" biçiminde iki sınıf içermektedir. O halde bu model için 6 çıktıya sahip olan bir ağ oluşturulabilir. Bu 6 çıktılarının hepsi de "sigmoid" aktivasyon fonksiyonuna sahip olabilir. Diğer bir seçenek de problemi tek etiketli çok sınıflı olarak modellemektir. Bu durumda çıktı katmanında yine 6 nöron bulunur fakat çıktı katmanın aktivasyon fonksiyonu "softmax" alınır. Böylece en yüksek oranlı çıktıları veren birden fazla sınıf alınabilir. Veri analistleri daha bu tür durumlarda çok etiketli iki sınıfı model oluşturmayı tercih etmektedir.

Tek Etiketli İki Sınıflı Model Örnekleri

Aslında bu türden örnek kursumuzda daha önce yapılmıştı. Kişilerin çeşitli biyomedikal bilgilerinin alınıp onların diyabetli olup olmadığını belirlemeye çalışan örnek tipik olarak bu gruptandır. Biz burada başka bir örnek üzerinde duracağız.

IMDB Örneği

Keras'ın kendi dataset'i içerisinde bulunan IMDB veri tablosu filmler hakkındaki yorumlardan oluşmaktadır. Her yorum "olumlu (pozitif)" ya da "olumsuz (negative)" biçimde değerlendirilmektedir. Modelden amaç bir yorumdaki sözcükler bakarak onun olumlu mu yoksa olumsuz mu olduğunu anlamaktır. Yani başka bir deyişle yorumdaki kişi filmi beğenmiş midir, tavsiye etmekte midir ya da bunun tersi midir? IMDB veri kümesindeki her yorum farklı miktarda sözcük içerebilmektedir. Ayrıca bilindiği gibi yapay sinir ağları tamamen sayısal düzeyde işlemler yapmaktadır. Yani bizim sözcükleri bir biçimde sayılarla ifade etmemiz gereklidir. Bunun için bu örnekte tüm yorumlardaki tüm sözcükler bir sözlüğe yerleştirilmiştir. Sözlükte her sözcüğe karşı bir indeks numarası değer olarak gelmektedir. Bir yorum da böylece sözcüklerden değil indeks numaralarından oluşturulmuş olur. Özette IMDB örneğinden modelleme şöyle oluşturulmaktadır:

- Tüm yorumlardaki tüm sözcükler toplanıp bir sözlüğe yerleştirilmiştir. Her bir sözcüğe bir indeks numarası karşılık getirilmiştir. Sözlükteki anahtarlar sözcüklerden değerler ise onların indeks numaralarından oluşmaktadır
- Yorumlar sözcüklerden oluşmaktadır. Ancak onların sayısallaştırılması gerekmektedir. İşte bunun için her yorum bir indeks numarası dizisi haline dönüştürülr. Yani yorum artık sözcüklerden değil sözcükleri temsil eden indeks numaralarında oluşturulmaktadır.
- Her yorumdaki sözcük sayıları farklıdır. Bu ise yapay sinir ağı modeline uygun değildir. Yapay sinir ağlarının çalışması için her satırın eşit sayıda sütun bilgisine sahip olması gereklidir. Peki bu durum IMDB örneğinde nasıl sağlanabilir? İşte bunun iki yolu olabilir: Bitincisi youmlardaki sözcük sayılarını eşit hale getirmek (kısa olanları atıp uzun olanları kırmak). İkincisi de tüm yumlardaki tüm sözcükleri temel alarak eşit uzunlukta bir bool vektör oluşturmaktr. (Örneğin yazıda 100 sözcük varsa ve toplam tüm yarumlardaki sözcük sayısı da 10000 ise, biz her yorum için 10000'lük bir bool vektör oluştururuz. Bu 1000'lük bool vektördeki 100 eleman 1, geri kalan elemanlar 0 olur.)

Keras'ta imdb veri kümesini kullanmak için tensorflow.keras.datasets paketi içerisindeki imdb modülü import edilir. Sonra modülün load_data fonksiyonu ile imdb bilgilerine yönelik eğitim ve test verileri ndarray olarak alınmaktadır. load_data metodunun num_words isimli parametresi en çok kullanılan n tane sözcüğü yorumlarda bulunduracak biçimde veri kümelerini bize vermektedir. (Yani örneğin biz num_word değerini 10000 yaptığımızda tüm yorumların içerisindeki tüm farklı sözcüklerin sayısı 10000 tane olmaktadır. Örneğin:

```
from tensorflow.keras.datasets import imdb

VOCAB_SIZE = 10000

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
imdb.load_data(num_words=VOCAB_SIZE)
```

Burada training_dataset_x ve set_dataset_x birer ndarray nesnesidir. Ancak bu ndarray nesnesi list nesnelerinden oluşmaktadır. Örneğin:

```
training_dataset_x
Out[6]:
array([list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 5952, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36,
```

```

28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 32, 15, 16, 5345,
19, 178, 32]),  

    list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118, 1634, 14,  

394, 20, 13, 119, 954, 189, 102, 5, 207, 110, 3103, 21, 14, 69, 188, 8, 30, 23, 7, 4, 249, 126, 93, 4,  

114, 9, 2300, 1523, 5, 647, 4, 116, 9, 35, 8163, 4, 229, 9, 340, 1322, 4, 118, 9, 4, 130, 4901, 19, 4,  

1002, 5, 89, 29, 952, 46, 37, 4, 455, 9, 45, 43, 38, 1543, 1905, 398, 4, 1649, 26, 6853, 5, 163, 11,  

3215, 2, 4, 1153, 9, 194, 775, 7, 8255, 2, 349, 2637, 148, 605, 2, 8003, 15, 123, 125, 68, 2, 6853, 15,  

349, 165, 4362, 98, 5, 4, 228, 9, 43, 2, 1157, 15, 299, 120, 5, 120, 174, 11, 220, 175, 136, 50, 9, 4373,  

228, 8255, 5, 2, 656, 245, 2350, 5, 4, 9837, 131, 152, 491, 18, 2, 32, 7464, 1212, 14, 9, 6, 371, 78, 22,  

625, 64, 1382, 9, 8, 168, 145, 23, 4, 1690, 15, 16, 4, 1355, 5, 28, 6, 52, 154, 462, 33, 89, 78, 285, 16,  

145, 95]),  

    list([1, 14, 47, 8, 30, 31, 7, 4, 249, 108, 7, 4, 5974, 54, 61, 369, 13, 71, 149, 14, 22, 112, 4,  

2401, 311, 12, 16, 3711, 33, 75, 43, 1829, 296, 4, 86, 320, 35, 534, 19, 263, 4821, 1301, 4, 1873, 33,  

89, 78, 12, 66, 16, 4, 360, 7, 4, 58, 316, 334, 11, 4, 1716, 43, 645, 662, 8, 257, 85, 1200, 42, 1228,  

2578, 83, 68, 3912, 15, 36, 165, 1539, 278, 36, 69, 2, 780, 8, 106, 14, 6905, 1338, 18, 6, 22, 12, 215,  

28, 610, 40, 6, 87, 326, 23, 2300, 21, 23, 22, 12, 272, 40, 57, 31, 11, 4, 22, 47, 6, 2307, 51, 9, 170,  

23, 595, 116, 595, 1352, 13, 191, 79, 638, 89, 2, 14, 9, 8, 106, 607, 624, 35, 534, 6, 227, 7, 129,  

113]),  

    ...,

```

Hangi sözcüklerin hangi numaralara karşı geldiğini elde etmek için `get_word_index` isimli bir fonksiyon kullanılır. Bu fonksiyon bize bir sözlük nesnesi vermektedir. Örneğin:

```
word_index = imdb.get_word_index()
```

Tabii `get_word_index` bize tüm yorumlarda kullanılan sözcük ve indeksleri vermektedir. Bu sözlük nesnesinde anahtarlar sözcüğün kendisi, değerler ise onların numaralarını belirtmektedir. Yani biz sözcüğü verdigimizde onun numarasını hızlı bir biçimde elde ederiz. Halbuki bizim bunun tersine ihtiyacımız daha fazladır. O halde biz bu sözlüğü ters çevirelim:

```
reverse_word_index = dict([(value, key) for key, value in word_index.items()])
```

Pekiyi bir yorumu yazıya dönüştürebilir miyiz? Evet, ancak önemli bir nokta vardır. Keras `imdb` ve `reutes` gibi sözcük içeren veri kümelerindeki sözcük numaralarını hep 3 fazla almıştır. Yani örneğin bir yorumdaki 5 sayısı aslında $5 - 3 = 2$ 'dir. Yorumlardaki sözcükleri oluşturan sayıların ilk üçü (0, 1, 2) "reserve" edilmişdir. Bu durumda örneğin biz ilk yorumu şu biçimde sözcüklere dökebiliriz:

```

def print_text(dataset):
    for index in dataset:
        if index > 2:
            print(rev_word_index[index - 3], end=' ')
    print()

def get_text(dataset):
    text = ''
    for index in dataset:
        if index > 2:
            text += rev_word_index[index - 3] + ' '
    return text

print_text(training_dataset_x[0])
print('-----')
print(get_text(training_dataset_x[0]))

```

Bu örnekte sınır ağımızın çıktısı tek bir nörondan oluşmaktadır. Pekiyi girdi katmanı nasıl olmalıdır? Bizim amacımız bir yazдан ikili bir sonuç çıkartmaktır. Yazı da sözcüklerden oluşmaktadır. Sözcükler de numaralarla temsil edilmiştir. Eğer modelimizdeki girdi katmanı sözcük numaralarından oluşturulacağsa önemli bir sorun karşımıza çıkmaktadır. Bu durumda ağımızdaki girdi nöronlarının sayısı sabit olmayacağıdır. Halbuki sayının başlangıçta sabit olması gereklidir. İşte bu tür durumlarda izlenecek başka yöntemler de vardır. Bu konu ilerde yeniden ele alınacaktır. Ancak bu tür problemlerde daha çok tercih edilen tüm sözcüklerin (örneğimizde 10000 tane) hepsinin binary bir girdi nöronuyla temsil edilmesidir. Şöyle ki: Yorumlardaki tüm sözcükler (toplamı 10000 tane) girdi nöronları yapılır. Böylece ağıın girdi katmanı 10000 nörondan oluşur. Bu durumda her yorum 10000 elemanlı bir vektörle ifade edilir. Vektörün her elemanı bir sözcüğü temsil eder. O sözcük yazda varsa o nöron girişi 1 yapılır, yoksa 0 yapılır. Böylece örneğin 100

sözcükten oluşan bir yorum yazısı 100 tane nöronu 1 geri kalan nöronları 0 olan bir ndarray ile temsil edilecektir. Keras'taki imdb veritabanından elde edilen training_dataset_x ve test_dataset_x bu formda bir ndarray içermemektedir. O halde bizim öncelikle bu veri kümelerini bu hale dönüştürmemiz gereklidir. Bu işi yapan bir fonksiyon basit biçimde şöyle yazılabılır:

```
def vectorize(sequence, col_size):
    result = np.zeros((len(sequence), col_size))
    for index, vals in enumerate(sequence):
        result[index, vals] = 1.0

    return result

training_dataset_x = vectorize(training_dataset_x, VOCAB_SIZE)
test_dataset_x = vectorize(test_dataset_x, VOCAB_SIZE)
```

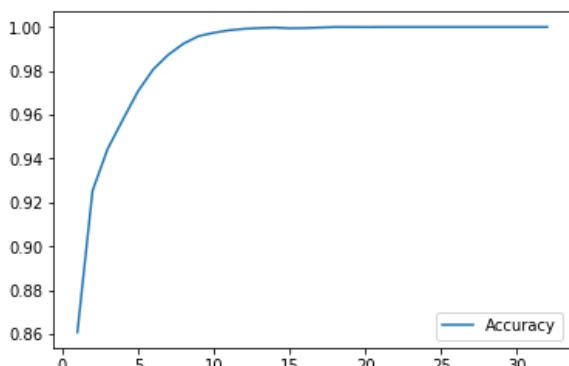
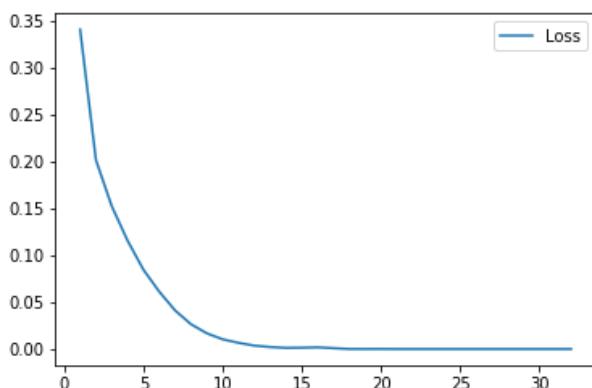
Burada önce "yorum sayısı" * "sözcük sayısı" kadar içi sıfırlarla dolu iki boyutlu bir ndarray yaratılmıştır. Sonra yorumlar satırlarda olacak biçimde satırların sözcük içeren sütunları 1 yapılmıştır. Bu işlem sonucunda bizim training_dataset_x ve test_dataset_x verilerimiz gerçek anlamda Keras'ın ve yapay sinir ağlarının kullanabileceği formata dönüştürülmüş durumdadır. Artık modelimizi oluşturabiliriz:

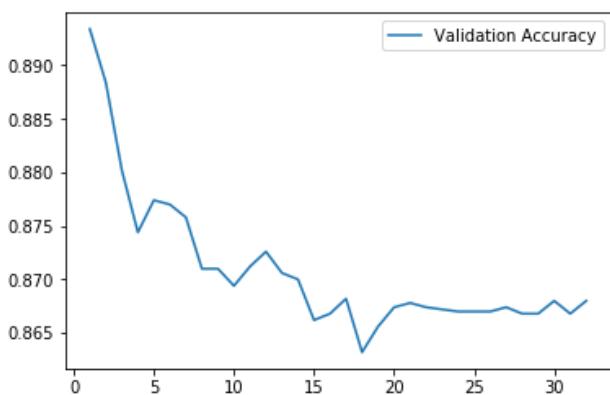
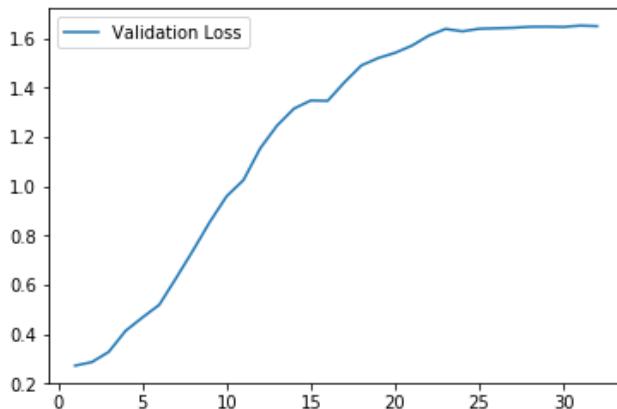
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(32, input_dim=VOCAB_SIZE, activation='relu', name='Hidden-1'))
model.add(Dense(32, activation='relu', name='Hidden-2'))
model.add(Dense(1, activation='sigmoid', name='Output'))

model.compile('rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=32, batch_size=64,
validation_split=0.2)
```

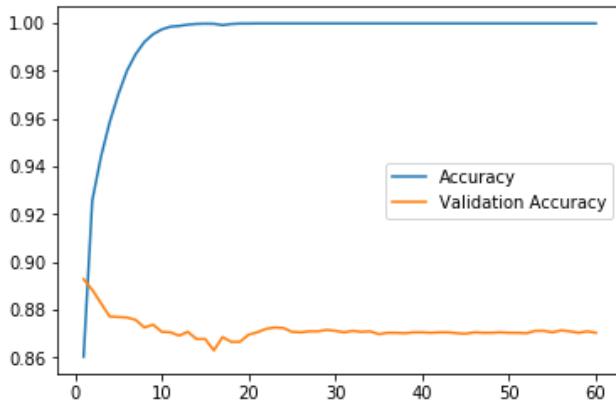
Şimdi bu işlemler sonucunda elde edilen grafiklere bakalım:





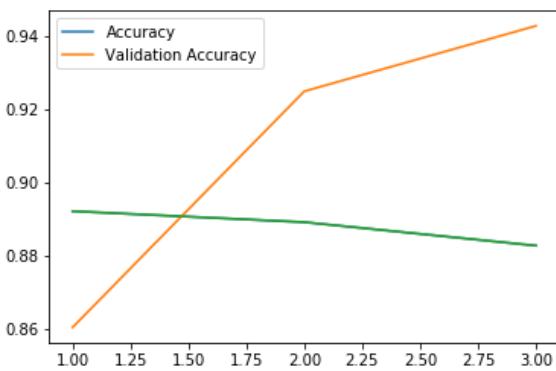
Eğitim sırasında "loss" ve metrik değerlerinin iki biçiminin bulunduğu görülmektedir: Eğitim veri kümesi dikkate alınarak üretilmiş olan loss ve metrik değerler ve sınama veri kümesi dikkate alınarak üretilmiş olan loss ve metrik değerler. Eğitim sırasında her epoch'tan sonra hem eğitim kümesi ile hem de sınama (validation) veri kümesi ile "loss" ve metrik değerler hesaplanmaktadır. Eğitim veri kümesi dikkate alınarak hesaplanmış değerler ile sınama veri kümesi dikkate alınarak hesaplanmış değerler arasında önemli farklılıkların olması (yukarıdaki örnekte olduğu gibi) bir "overfit" durumunu akla getirmelidir. Yukarıdaki örnekte eğitim veri kümesi dikkate alındığında accuracy değerinin %100'e yakın olduğu görülmektedir. Yani biz eğer bu sisteme eğitimde uygulanan verilerin aynısını sorsak bu sistem bize %100 oranında doğru yanıt verecektir. Halbuki sınama verilerinde yani eğitide kullanılmamış olan verilerde sistemin başarısı %85 civarlarındadır. O halde şöyle bir durum söz konusu olmaktadır: "Bizim sistemimiz eğitim sırasında eğitim verileri ile çok iyi performans göstermektedir. Ancak bu performansı eğitimde kullanılmayan verilerde göstermemektedir. Bu duruma da genel olarak bu bağlamda "overfit" denilmektedir. Test işlemi tüm sistem oluşturulduktan sonra nihai olarak yapılan bir işlemidir. Genellikle test işlemi sınamaadan daha büyük verilerle yapılır. Tabii büyük ölçüde son epoch'lardaki sınama "loss" ve metrik değerleri test değerleri ile örtüşmektedir.

Peki burada sözü edilen "overfit" problemi nasıl telafi edilebilir? Grafiklere bakıldığında eğitim veri kümesi dikkate alınarak elde edilen "loss" ve metrik değerlerin ortalama 2 civarında bir epoch'tan sonra %85'lerden yukarı fırladığı görülmektedir. Bu durumda epoch değerinin 2 civarında tutulması hem sınama verilerinin performansını düşürmemekte hem de eğitim veri kümesi ile overfit durumunu ortadan kaldırmaktadır. Şimdi iki metrik grafiğini üst üste çizelim:



Burada gördüğümüz gibi grafiklerin kesim noktası hemen ilk epoch'lar civarındadır.

Şimdi bu grafiği 2 epoch için yeniden çizelim:



Şimdi modelimizi test işlemeye sokalım:

```
loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))
```

Sonuç şöyledir:

```
loss = 0.3394545952796936, accuracy = 0.85908
```

Burada accuracy değerinin 0.85 olması makul gözükmemektedir. Modelimiz bu kadar veriyle çok iyi olmasa da oldukça iyi bir biçimde eğitilmiştir. Bu sonuç verilen kararın %85 oranında doğru olabileceğini belirtmektedir.

Şimdi de kurulan model üzerinde kestirim yapalım. Kestirim denemesi yapmak için mevcut bir yorumu alıp onun üzerinde değişiklikler yapabiliriz. Örneğin ilk yorumdaki beğenisi içeren sözcükler yerine "bad" gibi beğenisi içermeyen sözcükleri yerlestirelim:

```
text = '''? this film was just bad casting location scenery story direction everyone's really suited the part they played and you could just imagine being there robert ? is an amazing actor and now the same being director ? father came from the same scottish island as myself so i bad the fact there was a real connection with this film the witty remarks throughout the film were it was just brilliant so much that i bad bought the film as soon as it was released for ? and would recommend it to everyone to watch and the fly fishing was really cried at the end it was so bad bad sad and you know what they say if you cry at a film it must have been good and this bad was also ? to the two little boy's that played the ? of norman and paul they were just brilliant children are often left out of the ? list i think because the stars bad that play them all grown up are such a big profile for the whole film but these bad are amazing bad and should be praised for what they have done don't you think the whole story was so lovely because it was true and was someone's life after all bad that was shared with us all'''
```

Şimdi bu yazıyı index değerlerine dönüştürüp o değerleri de 10000'lik girdi vektörü haline getirelim:

```
import re
predict_words = re.sub('[.!;?]', " ", text).split()

predict_word_index = [word_index[word] + 3 for word in predict_words]
predict_data_x = np.zeros(VOCAB_SIZE)
predict_data_x[predict_word_index] = 1
```

Tabii predict işlemi için girdi kümесinin bir matris olması gereklidir. Nu tek boyutlu diziyi şöyle bir matrise dönüştürebiliriz:

```
predict_data_x = predict_data_x.reshape(1, VOCAB_SIZE)
```

Ve nihayet predict işlemimizi yapabiliriz:

```
result = model.predict(predict_data_x)
print("Olumlu" if result > 0.5 else "Olumsuz")
print(result)
```

Örneğin tüm kodları aşağıdaki gibidir:

```
import numpy as np

from tensorflow.keras.datasets import imdb

VOCAB_SIZE = 10000

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) = \
imdb.load_data(num_words=VOCAB_SIZE)

word_index = imdb.get_word_index()
rev_word_index = {value: key for key, value in word_index.items()}

def print_text(dataset):
    for index in dataset:
        if index > 2:
            print(rev_word_index[index - 3], end=' ')

def get_text(dataset):
    text = ''
    for index in dataset:
        if index > 2:
            text += rev_word_index[index - 3] + ' '
    return text

print_text(training_dataset_x[0])
print('\n-----')
print(get_text(training_dataset_x[0]))

def vectorize(sequence, col_size):
    result = np.zeros((len(sequence), col_size))
    for index, vals in enumerate(sequence):
        result[index, vals] = 1.0

    return result

training_dataset_x = vectorize(training_dataset_x, VOCAB_SIZE)
test_dataset_x = vectorize(test_dataset_x, VOCAB_SIZE)

from tensorflow.keras.models import Sequential
```

```

from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(32, input_dim=10000, activation='relu', name='Hidden-1'))
model.add(Dense(32, activation='relu', name='Hidden-2'))
model.add(Dense(1, activation='sigmoid', name='Output'))

model.compile('rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=2, batch_size=64,
validation_split=0.2)

import matplotlib.pyplot as plt

plt.title('Epoch - Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['loss'])
plt.legend(['Loss'])

plt.pause(1)

plt.title('Epoch - Accuracy Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Accuracy'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.legend(['Accuracy'])

plt.pause(1)

plt.title('Epoch - Validation Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_loss'])
plt.legend(['Validation Loss'])

plt.pause(1)

plt.title('Epoch - Validation Accuracy Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation accuracy'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['Validation Accuracy'])

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

plt.title('Epoch - Accuracy / Validation Accuracy Graph')
plt.xlabel = 'Epoch'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['Accuracy', 'Validation Accuracy'])

text = '''? this film was just bad casting location scenery story direction everyone's really
suited the part they played and you could just imagine being there robert ? is an amazing actor
and now the same being director ? father came from the same scottish island as myself so i bad
the fact there was a real connection with this film the witty remarks throughout the film were
it was just brilliant so much that i bad bought the film as soon as it was released for ? and
would recommend it to everyone to watch and the fly fishing was really cried at the end it was
so bad bad sad and you know what they say if you cry at a film it must have been good and this
bad was also ? to the two little boy's that played the ? of norman and paul they were just
brilliant children are often left out of the ? list i think because the stars bad that play
them all grown up are such a big profile for the whole film but these bad are amazing bad and
should be praised for what they have done don't you think the whole story was so lovely because
'''
```

```
it was true and was someone's life after all bad that was shared with us all'''
```

```
import re
```

```
predict_words = re.sub('[.,;?]', ' ', text).split()
predict_word_index = [word_index[word] + 3 for word in predict_words]
predict_data_x = np.zeros(VOCAB_SIZE)
predict_data_x[predict_word_index] = 1
predict_data_x = predict_data_x.reshape(1, VOCAB_SIZE)

result = model.predict(predict_data_x)
print("Oluştu" if result > 0.5 else "Oluşmamış")
print(result)
```

Tek Etiketli (Multilabel) ve Çok Sınıflı (Multiclass) Sınıflandırma Örnekleri

Anımsanacağı gibi tek etiketli çok sınıflı sınıflandırma problemlerinde çıktı bir nitelinin farklı sınıflarına ilişkindir. Burada çok sınıflı sınıflandırma için Keras'ta da hazır olarak sunulan Reuters ve MINST örnekleri üzerinde durulacaktır.

Reuters Örneği

Keras'in reuters verileri 11228 adet haber yazısını içermektedir. Bu haber yazıları 46 değişik kategoriye ayrılmaktadır. Bu örnektenden amaç bir yazı verildiğinde yazının konusunu 46 değişik konu arasından belirleyebilmektir. Maalesef Keras'in resmi dokümanlarında bu 46 reuters kategorisinin neler olduğu belirtilmemiştir. Bu 46 kategori başka kaynaklarda şöyle listelenmiştir:

```
category_list = ['cocoa', 'grain', 'veg-oil', 'earn', 'acq', 'wheat', 'copper', 'housing', 'money-supply', 'coffee', 'sugar', 'trade', 'reserves', 'ship', 'cotton', 'carcass', 'crude', 'nat-gas', 'cpi', 'money-fx', 'interest', 'gnp', 'meal-feed', 'alum', 'oilseed', 'gold', 'tin', 'strategic-metal', 'livestock', 'retail', 'ipi', 'iron-steel', 'rubber', 'heat', 'jobs', 'lei', 'bop', 'zinc', 'orange', 'pet-chem', 'dlr', 'gas', 'silver', 'wpi', 'hog', 'lead']
```

Aslında Keras'in reuters verileri tamamen IMDB verileri gibi organize edilmiştir. Yani yazılar yine sözcük indeksleri ile ifade edilmiş durumdadır. Bu nedenle programda yukarıdaki IMDB örneğindeki hazırlıklara benzer hazırlıklar yapılacaktır:

```
import numpy as np
from tensorflow.keras.datasets import reuters

VOCAB_SIZE = 10000

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
reuters.load_data(num_words=VOCAB_SIZE)

category_list = ['cocoa', 'grain', 'veg-oil', 'earn', 'acq', 'wheat', 'copper', 'housing', 'money-supply', 'coffee', 'sugar', 'trade', 'reserves', 'ship', 'cotton', 'carcass', 'crude', 'nat-gas', 'cpi', 'money-fx', 'interest', 'gnp', 'meal-feed', 'alum', 'oilseed', 'gold', 'tin', 'strategic-metal', 'livestock', 'retail', 'ipi', 'iron-steel', 'rubber', 'heat', 'jobs', 'lei', 'bop', 'zinc', 'orange', 'pet-chem', 'dlr', 'gas', 'silver', 'wpi', 'hog', 'lead']

word_index = reuters.get_word_index()
rev_word_index = {value: key for key, value in word_index.items()}

def print_text(dataset):
    for index in dataset:
        if index > 2:
            print(rev_word_index[index - 3], end=' ')

def get_text(dataset):
    text = ''
```

```

for index in dataset:
    if index > 2:
        text += rev_word_index[index - 3] + ' '
return text

print_text(training_dataset_x[0])
print('-----')
print(get_text(training_dataset_x[0]))

def vectorize(sequence, col_size):
    result = np.zeros((len(sequence), col_size))
    for index, vals in enumerate(sequence):
        result[index, vals] = 1.0

    return result

training_dataset_x = vectorize(training_dataset_x, VOCAB_SIZE)
test_dataset_x = vectorize(test_dataset_x, VOCAB_SIZE)
training_dataset_y = vectorize(training_dataset_y, 46)
test_dataset_y = vectorize(test_dataset_y, 46)

```

Artık ağ modelini kurabiliriz. Ağımızda yine iki hidden katman bulunabilir. Ancak çıktı kategorilerinin sayısı fazlalaştıkça katmanlardaki nöron sayılarının artırılması doğru olacaktır. Benzer biçimde kategori sayısı arttıkça ağı derinleştirme de sonucun kalitesini artırabilmektedir. Ağımızdaki hidden katmanların aktivasyon fonksiyonları yine "relu" biçiminde alınacaktır. Çıkış katmanındaki aktivasyon fonksiyonunun "softmax" alınması gereklidir. Anımsanacağı gibi çıktı katmanının aktivasyon fonksiyonun softmax olması tüm çıktı nöronlarının toplam değerlerinin 1 olmasını sağlamaktadır. Böylece biz bu değerlerin en büyüğünü alarak girdinin sınıfını belirleyebileceğiz. Model için loss fonksiyonu "categorical_crossentropy" olarak seçilecektir. Optimizer için yine "rmsprop" ve metrik değerler için yine "accuracy" kullanılacaktır.

Şimdi modelimizi oluşturalım:

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, input_dim=VOCAB_SIZE, activation='relu', name='Hidden-1'))
model.add(Dense(64, activation='relu', name='Hidden-2'))
model.add(Dense(46, activation='softmax', name='Output'))

model.compile('rmsprop', loss='binary_crossentropy', metrics=['accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=20, batch_size=32,
validation_split=0.2)

```

Grafiklerimizi çizelim:

```

import matplotlib.pyplot as plt

plt.title('Epoch - Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['loss'])
plt.legend(['Loss'])

plt.pause(1)

plt.title('Epoch - Accuracy Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Accuracy'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.legend(['Accuracy'])

```

```

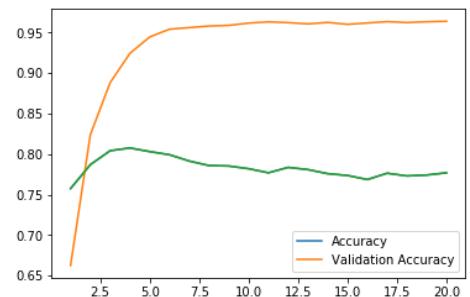
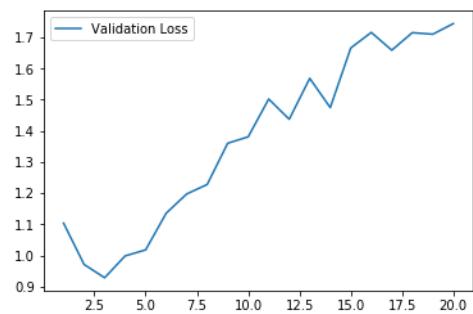
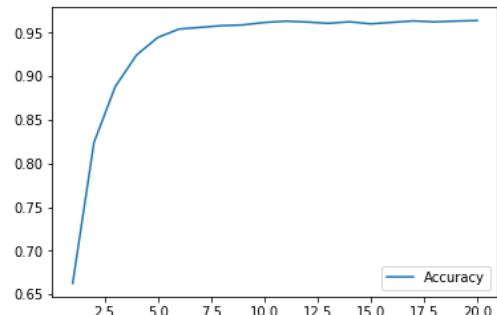
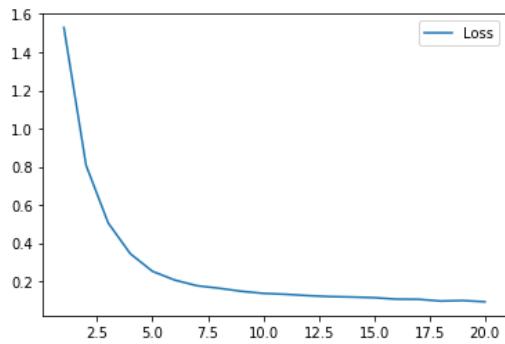
plt.pause(1)

plt.title('Epoch - Validation Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_loss'])
plt.legend(['Validation Loss'])

plt.pause(1)

plt.title('Epoch - Validation Accuracy Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation accuracy'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['Validation Accuracy'])

```



Burada üçüncü epoch'tan sonra overfit durumu gözlemlenmektedir. Bu epoch kaynaklı overfit nedeniyle epoch sayısı 3'le sınırlı tutulabilir. Modelimizi test edelim:

```
loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))
```

Şimdi de kestirim işlemini yapalım:

```
text = '''? ? said as a result of its december acquisition of space co it expects earnings per share in 1987 of 1 15 to 1 30 dlr per share up from 70 cts in 1986 the company said pretax net should rise to nine to 10 mln dlr from six mln dlr in 1986 and rental operation revenues to 19 to 22 mln dlr from 12 5 mln dlr it said cash flow per share this year should be 2 50 to three dlr reuter 3'''

import re
predict_words = re.sub('[.!;?]', " ", text).split()

predict_word_index = [word_index[word] + 3 for word in predict_words]
predict_data_x = np.zeros(10000)
predict_data_x[predict_word_index] = 1
predict_data_x = predict_data_x.reshape(1, 10000)

result = model.predict(predict_data_x)
category = np.argmax(result)
print('category = {}, category name = {}'.format(category, category_list[category]))
```

Buradan çıkan sonuç şöyledir:

```
category = 3, category name = earn
```

predict işlemi sonucunda biz 46 elemanlı bir ndarray nesnesi elde etmiş olduk. Bu nesnedeki tüm değerlerin toplamı softmax fonksiyonu nedeniyle 1 olacaktır. Biz de hangi kategorinin kestirildiğini en yüksek değere bakarak anlarız. numpy'in argmax fonksiyonu bir ndarray içerisindeki değerlerin en büyüğünün indeksini vermektedir.

Aşağıda örneğin tüm kodlarını yeniden veriyoruz:

```
import numpy as np
from tensorflow.keras.datasets import reuters

VOCAB_SIZE = 10000

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
reuters.load_data(num_words=VOCAB_SIZE)

category_list = ['cocoa', 'grain', 'veg-oil', 'earn', 'acq', 'wheat', 'copper', 'housing',
'money-supply', 'coffee', 'sugar', 'trade', 'reserves', 'ship', 'cotton', 'carcass', 'crude',
'nat-gas', 'cpi', 'money-fx', 'interest', 'gnp', 'meal-feed', 'alum', 'oilseed', 'gold', 'tin',
'strategic-metal', 'livestock', 'retail', 'ipi', 'iron-steel', 'rubber', 'heat', 'jobs',
'lei', 'bop', 'zinc', 'orange', 'pet-chem', 'dlr', 'gas', 'silver', 'wpi', 'hog', 'lead']

word_index = reuters.get_word_index()
rev_word_index = {value: key for key, value in word_index.items()}

def print_text(dataset):
    for index in dataset:
        if index > 2:
            print(rev_word_index[index - 3], end=' ')

def get_text(dataset):
    text = ''
    for index in dataset:
```

```

    if index > 2:
        text += rev_word_index[index - 3] + ' '
    return text

print_text(training_dataset_x[0])
print('\n-----')
print(get_text(training_dataset_x[0]))

def vectorize(sequence, col_size):
    result = np.zeros((len(sequence), col_size))
    for index, vals in enumerate(sequence):
        result[index, vals] = 1.0

    return result

training_dataset_x = vectorize(training_dataset_x, VOCAB_SIZE)
test_dataset_x = vectorize(test_dataset_x, VOCAB_SIZE)
training_dataset_y = vectorize(training_dataset_y, 46)
test_dataset_y = vectorize(test_dataset_y, 46)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, input_dim=VOCAB_SIZE, activation='relu', name='Hidden-1'))
model.add(Dense(64, activation='relu', name='Hidden-2'))
model.add(Dense(46, activation='softmax', name='Output'))

model.compile('rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=3, batch_size=32,
validation_split=0.2)

import matplotlib.pyplot as plt

plt.title('Epoch - Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['loss'])
plt.legend(['Loss'])

plt.pause(1)

plt.title('Epoch - Accuracy Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Accuracy'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.legend(['Accuracy'])

plt.pause(1)

plt.title('Epoch - Validation Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_loss'])
plt.legend(['Validation Loss'])

plt.pause(1)

plt.title('Epoch - Validation Accuracy Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation accuracy'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['Validation Accuracy'])

```

```

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

plt.title('Epoch - Accuracy / Validation Accuracy Graph')
plt.xlabel = 'Epoch'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['Accuracy', 'Validation Accuracy'])

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

text = '''? ? said as a result of its december acquisition of space co it expects earnings per share in 1987 of 1 15 to 1 30 dlr per share up from 70 cts in 1986 the company said pretax net should rise to nine to 10 mln dlr from six mln dlr in 1986 and rental operation revenues to 19 to 22 mln dlr from 12 5 mln dlr it said cash flow per share this year should be 2 50 to three dlr reuter 3'''

import re

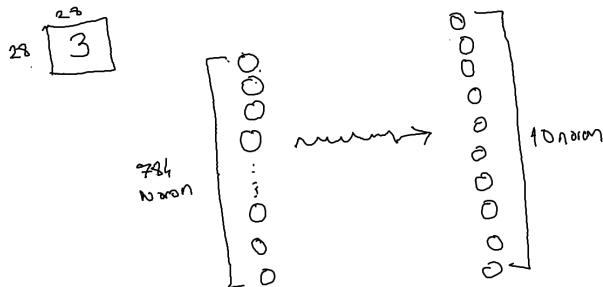
predict_words = re.sub('[.,;?]', ' ', text).split()
predict_word_index = [word_index[word] + 3 for word in predict_words]
predict_data_x = np.zeros(VOCAB_SIZE)
predict_data_x[predict_word_index] = 1
predict_data_x = predict_data_x.reshape(1, VOCAB_SIZE)

result = model.predict(predict_data_x)
category = np.argmax(result)
print('category = {}, category name = {}'.format(category, category_list[category]))

```

MNIST (Modified National Institute of Standards and Technology) Örneği

Mnist makine öğrenmesinde en çok kullanılan veritabanlarından biridir. Mnist'in çok değişik veri kümeleri vardır. Keras'ın içerisindeki mnist modülünde her biri 28×28 'lik bir gray scale bitmap'ten oluşan pek çok resim bulunmaktadır. Bu resimlerde 0'dan 9'a kadar sayıların elle çizilmiş görüntüleri bulunmaktadır. Dolayısıyla modelin girdi katmanı $28 \times 28 = 784$ nörondan çıktı katmanı da 10 nörondan oluşmaktadır.



Gray-Scale resim nedir? Bir resim bilindiği gibi pixel'lerden oluşmaktadır. Her pixel'in de renk bileşenleri vardır. Dijital bilgisayarlarda genellikle RGB (Red, Green, Blue) bileşenleri kullanılmaktadır. Bu bileşenler 1'er byte (8'er bit) olarak kodlanırlar. Böylece her pixel 2^{24} renkten biri olabilir. Aslında modern grafik kartlarında her pixel için bir byte da transparanlık (alpha channel) bilgisi tutulmaktadır. Bu transparanlık 255 ise tam saydamsız, 0 ise tam saydam anlamına gelir.

Gray-Scale resimde her bir pixel RGB ile değil tek bir değerle ifade edilir. Gray-Scale resim aslında grinin tonlarından oluşan resimdir. Yani bir çeşit siyah beyaz görüntüyü belirtmektedir. Gray-Scale resim aslında teknik olarak $R = G = B$ olan pixel'lerden oluşan resimdir. Gerçekten de gray-scale bir resimdeki her bir pixel'in RGB değerleri aynıdır. İşte böylece her pixel bir byte ile kodlanabilmektedir. Tabii tam siyah-beyaz (monochrome) bir resimde her pixel 1 bit ile ifade edilebilir. Ancak tam siyah-beyaz resim çoğu kez anlamsız bir görüntü vermektedir.

Pekiyi normal bir resim gray-scale biçimde dönüştürülebilir mi? Yanıt evet. Bunun için kullanılan iki temel teknik vardır. Birinci teknikte her pixel'in RGB değerlerinin ortalaması alınır. Bu yöntem bazı resimlerde çok tatmin edici sonuçlar vermemektedir. Bunun yerine pixel'in RGB değerlerinin ağırlıklı ortalaması alınmaktadır. Yani R, G, B değerleri özel değerlerle çarpılıp toplanarak üçe bölünmektedir. Genellikle bu teknik tercih edilmektedir.

Mnist örneğimizde önce veri kümесini yüklemekle başlayalım:

```
from tensorflow.keras.datasets import mnist;
(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) = mnist.load_data()
```

Buradaki trining_dataset_x her biri 28X28'lik matristen oluşan 60000'luk bir ndarray dizisidir. Yani bu dizi 60000X28X28 boyutundadır. Benzer biçimde test_dataset_x de her biri 28X28'lik matristen oluşan 10000'luk bir ndarray dizisidir. Tabii bu dizilerin elemanları gray-scale pixel belirttiğine göre 0 ile 255 arasında olmak zorundadır. training_dataset_y ve test_dataset_y dizileri ise tek boyutlu ndarray nesneleridir. Bu dizilerin her elemanı 0'dan 9'a kadar kategorik bir değer belirtmektedir. Bu değerin örneğin 3 olması demek oradaki 28x28 gray-scale resmin 3'ün çizimi olması demektir.

Şimdi biz matplotlib.pyplot kullanarak buradaki birkaç resmi çizdirelim:

```
import matplotlib.pyplot as plt
for i in range(10):
    plt.imshow(training_dataset_x[i], cmap='gray', interpolation='none')
    plt.pause(1)
```

Şimdi biz Keras modeli için 28x28 lik matrisi tek boyutlu hale getirelim:

```
training_dataset_x = training_dataset_x.reshape(-1, 28 * 28)
test_dataset_x = test_dataset_x.reshape(-1, 28 * 28)
```

Şimdi de training_dataset_y ve test_dataset_y verilerini düzenleyelim. Bizim ağımızın çıktı katmanında 10 tane nöron olacağına göre biz de ağımızı eğitirken çıktı için 10 değer bulundurmalıyız. Anımsanacağı gibi çok sınıflı sınıflandırma problemlerinde çıktı katmanındaki aktivasyon fonksiyonları "softmax" alınıyordu. Bu "softmax" fonksiyonu çıktı nöronlarının toplam değerini 1'de tutuyordu. O hlađe bizim 3, 5, 7, 8 gibi çıktı değerlerini 10 sütunlu yalnızca ilgili elemanı 1 olan bir vektör biçiminde ifade etmemiz gereklidir. Bu zaten daha önce gördüğümüz "one-hot-encoding" işleminin aynısıdır.

```
from tensorflow.keras.utils import to_categorical
training_dataset_y = to_categorical(training_dataset_y)
test_dataset_y = to_categorical(test_dataset_y)
```

Şimdi verilerimizi normalize edelim. Bunun için min-max ölçeklendirmesi yöntemini kullanabiliriz:

```
training_dataset_x = training_dataset_x / 255
test_dataset_x = test_dataset_x / 255
```

Şimdi modelimizi oluşturalım:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(512, input_dim=28 * 28, activation='relu', name='Hidden-1'))
model.add(Dense(256, activation='relu', name='Hidden-2'))
model.add(Dense(10, activation='softmax', name='Output'))
```

```
model.compile('adam', loss='categorical_crossentropy', metrics=[ 'accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=20, batch_size=64,
validation_split=0.2)
```

Şimdi de graflerimizi çizelim:

```
import matplotlib.pyplot as plt

plt.title('Epoch - Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history[ 'loss' ])
plt.legend(['Loss'])

plt.pause(1)

plt.title('Epoch - Accuracy Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Accuracy'
plt.plot(range(1, len(hist.epoch) + 1), hist.history[ 'acc' ])
plt.legend(['Accuracy'])

plt.pause(1)

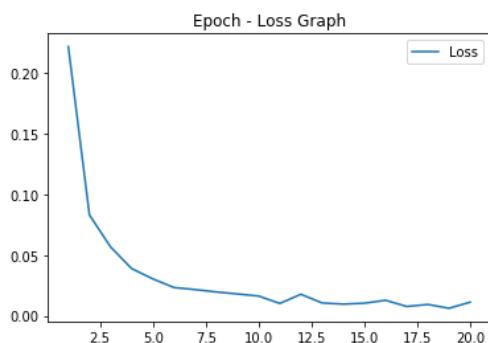
plt.title('Epoch - Validation Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history[ 'val_loss' ])
plt.legend(['Validation Loss'])

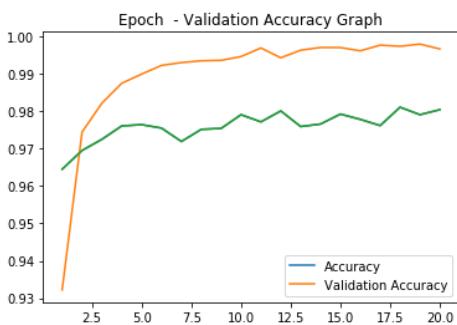
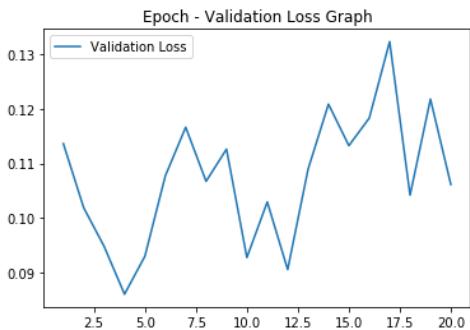
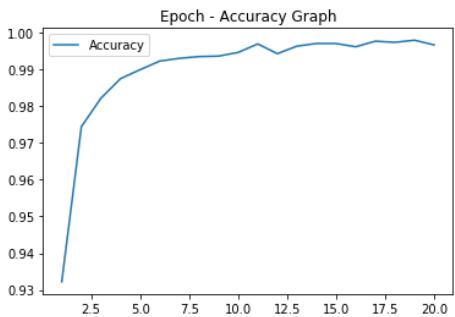
plt.pause(1)

plt.title('Epoch - Validation Accuracy Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation accuracy'
plt.plot(range(1, len(hist.epoch) + 1), hist.history[ 'val_acc' ])
plt.legend(['Validation Accuracy'])

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

plt.title('Epoch - Accuracy / Validation Accuracy Graph')
plt.xlabel = 'Epoch'
plt.plot(range(1, len(hist.epoch) + 1), hist.history[ 'acc' ])
plt.plot(range(1, len(hist.epoch) + 1), hist.history[ 'val_acc' ])
plt.legend(['Accuracy', 'Validation Accuracy'])
```





Bu grafiklere göre epoch sayısı arttıkça düşük düzeyli bir "overfit" şüphesi belirmektedir. Bu nedenle epoch sayısı 5 civarında tutulabilir.

Şimdi de modelimiz test edelim:

```
loss, accuracy = model.evaluate(test_set_x, test_set_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))
```

Sonuç şöyledir:

```
loss = 0.1139783305356918, accuracy = 0.981
```

Buradaki accuracy değerine bakıldığında bu modelin %98 olasılıkla girilen rakamı doğru tanadığını söyleyebiliriz. Daha yüksek bir veri kümesi üzerinde eğitim yapıldığında bu değer artacaktır. Yine katman sayılarını ve nöron sayılarını artırarak da daha iyi bir model oluşturabiliriz.

Şimdi de predict işlemi yapmak isteyelim. Biz bu işlemi gerçek bir şekil üzerinde yapalım. Bunun için Paint gibi bir programda elle bir rakam çizip onu bmp biçiminde kaydedebiliriz. Sonra pyplot kullanarak onun içerisindeki pixel verilerini gray-scale biçimde alabiliyoruz.

```
import numpy as np

def rgb2gray(rgb):
    return np.dot(rgb[:, :, :], [0.2989, 0.5870, 0.1140])

from matplotlib.image import imread
```

```

img = imread('test-9-2.png')

plt.imshow(img, interpolation='none')

gray = rgb2gray(img)
gray = gray.reshape((1, 28 * 28))

predict_result = model.predict(gray)
number = predict_result.argmax(axis=1)
print(number)

```

Buradaki kodda `rgb2gray` fonksiyonu renkli bir resmi "gray scale" hale getirmektedir. Resim `matplotlib` kütüphanesinin `imread` isimli fonksiyonuyla okunmuş ve sonra `display` edilmiştir. Daha sonra da `predict` işlemi yapılmıştır.

Aşağıda Mnist örneğinin tüm kodları verilmiştir:

```

from tensorflow.keras.datasets import mnist

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) = mnist.load_data()

import matplotlib.pyplot as plt

for i in range(10):
    plt.imshow(training_dataset_x[i], cmap='gray', interpolation='none')
    plt.pause(1)

training_dataset_x = training_dataset_x.reshape(-1, 28 * 28)
test_dataset_x = test_dataset_x.reshape(-1, 28 * 28)

from tensorflow.keras.utils import to_categorical

training_dataset_y = to_categorical(training_dataset_y)
test_dataset_y = to_categorical(test_dataset_y)

training_dataset_x = training_dataset_x / 255
test_dataset_x = test_dataset_x / 255

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(512, input_dim=28*28, activation='relu', name='Hidden-1'))
model.add(Dense(256, activation='relu', name='Hidden-2'))
model.add(Dense(10, activation='softmax', name='Output'))

model.compile('adam', loss='categorical_crossentropy', metrics=['accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=20, batch_size=64,
validation_split=0.2)

import matplotlib.pyplot as plt

plt.title('Epoch - Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['loss'])
plt.legend(['Loss'])

plt.pause(1)

plt.title('Epoch - Accuracy Graph')

```

```

plt.xlabel = 'Epoch'
plt.ylabel = 'Accuracy'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.legend(['Accuracy'])

plt.pause(1)

plt.title('Epoch - Validation Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_loss'])
plt.legend(['Validation Loss'])

plt.pause(1)

plt.title('Epoch - Validation Accuracy Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation accuracy'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['Validation Accuracy'])

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

plt.title = 'Epoch - Accuracy / Validation Accuracy Graph';
plt.xlabel = 'Epoch'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['Accuracy', 'Validation Accuracy'])

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

import numpy as np

def rgb2gray(rgb):
    return np.dot(rgb[:, :, :], [0.2989, 0.5870, 0.1140])

from matplotlib.image import imread
img = imread('test-7-2.png')

plt.imshow(img, interpolation='none')

gray = rgb2gray(img)
gray = gray.reshape((1, 28 * 28))

predict_result = model.predict(gray)
number = predict_result.argmax(axis=1)
print(number)

```

Evrişimsel (Convolutional) Sinir Ağları

Evrişim kavramı ile sayısal sinyal işlemede çokça karşılaşılmaktadır. Pratikte evrişim işlemi pek çok farklı veri türüne uygulanabilmektedir. Ancak bunlardan tipi olan görsel verilerdir. Tabii görsel veriler dışında evrişim işitsel (audio) ve hareketsel (video) verilere de uygulanabilmektedir.

Evrişimin görüntü verilerine uygulanması en yaygın kullanım biçimidir. Evrişim işlemi sayesinde görüntü yerel ögelere duyarlı hale getirilebilmektedir. Evrişim işlemi gray-scale görüntü verileri üzerinde şöyle yürütülmektedir: Evrişime sokulacak orijinal görüntünün yanı sıra evrişim işlemesinde kullanılacak bir filtreleme (kernel) matrisi oluşturulur. Sonra bu filtreleme matrisi görüntünün üzerine bindirilerek ve kaydırılarak işleme sokulur. Her işlem sonucunda tek bir değer elde edilmektedir. Buradaki işlem genellikle "dot product" biçimindedir. Yani filtreleme matrisindeki elemanlar asıl resmin çakıştırıldığı yerdeki elemanlarla çarpılarak toplanır. Örneğin asıl resmin gray-scale pixelleri şöyle olsun:

```
a11 a12 a13 a14 a15  
a21 a22 a23 a24 a25  
a31 a32 a33 a34 a35  
a41 a42 a43 a44 a45  
a51 a52 a53 a54 a55
```

Seçtiğimiz filtreleme matisisi de 3×3 olacak biçimde aşağıdaki gibi olsun:

```
b11 b12 b13  
b21 b22 b23  
b31 b32 b33
```

Şimdi bu filtreleme matrisi asıl görüntü matrisinin sol-üst köşesine oturularak "dot-product" işlemi yapılır:

```
toplam = b11 * a11 + b12 * a12 + b13 * a13 + b21 * a21 + b22 * a22 + b23 * a23 + b31 * a31 +  
b32 * a32 + b33 * a33
```

Burada elde edilen toplam değeri evrişilmiş hedef görüntü matrisinin c_{11} elemanını oluşturacaktır. İşte sonra bu filtreleme matrisi asıl görüntü matrisi üzerinde bir sağa kaydırılarak aynı işlem yinelenir:

```
toplam = b11 * a12 + b12 * a12 + b13 * a14 + b21 * a22 + b22 * a23 + b23 * a24 + b31 * a32 +  
b32 * a34 + b33 * a34
```

Burada elde edilen toplam evrişilmiş görüntü matrisinin c_{12} 'inci elemanı olacaktır. Bu işleme böyle devam edildiğinde evrişilmiş matris 3×3 'lük olur. Aslında burada evrişilmiş matrisin boyutu için şu ifade yazılabilir:

Evrişilmiş matrisin satır uzunluğu = Görüntü matrisinin satır uzunluğu – filtre matrisinin satır uzunluğu + 1
Evrişilmiş matrisin sütun uzunluğu = Görüntü matrisinin sütun uzunluğu – filtre matrisinin sütun uzunluğu + 1

Tabii asıl görüntü matrisi kare matris olmak zorunda değildir. Benzer biçimde aslında filtre matrisi de kare matris olmak zorunda değildir.

Gördüğü gibi buradaki evrişim işleminden daha küçük bir görüntü matrisi elde edilmektedir. Eğer evrişilmiş matrisin görüntü matrisi ile aynı büyüklükte olması isteniyorsa görüntü matrisinin filtreleme matrisinin satır ve sütun uzunluğuna dayalı olarak iki tarafında uzatılması gereklidir. Bu uzatma işlemi çeşitli biçimlerde yapılmaktadır. Örneğin sol sütunun soluna, sağ sütunun sağına, üst satırın yukarısına, alt satırın aşağısına içi sıfırlarla dolu sütunlar ve satırlar ekleenerek asıl şekil evrişim amaçlı büyütülebilir. Ya da sıfır eklemek yerine son satır ya da sütunlar çöktürülmemektedir. Bu işleme genel olarak "padding" denilmektedir.

Evrişim işleminde kaydırma birer birer yapılmak zorunda değildir. Bu kaydırma değerine "stride" denilmektedir. Stride değerinin 1 olması kaydırmanın birer birer yapılabileceği anlamına gelir. Eğer stride değeri yükseltilirse evrişilmiş matris küçülecektir. Aynı zamanda resimdeki yerel ilişkilerin uzaklığını da artırılmış olacaktır.

Aşağıda gray-scale bir resim üzerinde evrişim işlemi yapan örnek bir Python programı verilmektedir:

```
import numpy as np

def conv(im, imfilter):
    im_height = im.shape[0]
    im_width = im.shape[1]

    imfilter_height = imfilter.shape[0]
    imfilter_width = imfilter.shape[1]

    conv_height = im_height - imfilter_height + 1
    conv_width = im_width - imfilter_width + 1
```

```

im_conv = np.zeros((conv_height, conv_width))

for row in range(conv_height):
    for col in range(conv_width):
        for i in range(imfilter_height):
            for j in range(imfilter_width):
                im_conv[row, col] += im[row + i, col + j] * imfilter[i, j]

im_conv[im_conv > 255] = 255
im_conv[im_conv < 0] = 0

return im_conv

def rgb2gray(rgb):
    return np.dot(rgb[:, :, :], [0.2989, 0.5870, 0.1140])

import matplotlib.pyplot as plt
import matplotlib.image as mpimage

im = mpimage.imread('AbbeyRoad.jpg')
plt.imshow(im)
plt.pause(1)

imgray = rgb2gray(im)
plt.imshow(imgray, cmap='gray', interpolation='none')
plt.pause(1)

blur = np.full((10, 10), 1 / 100)
conv_imgray = conv(imgray, blur)

plt.imshow(conv_imgray, cmap='gray', interpolation='none')
plt.pause(1)

sobel_x = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

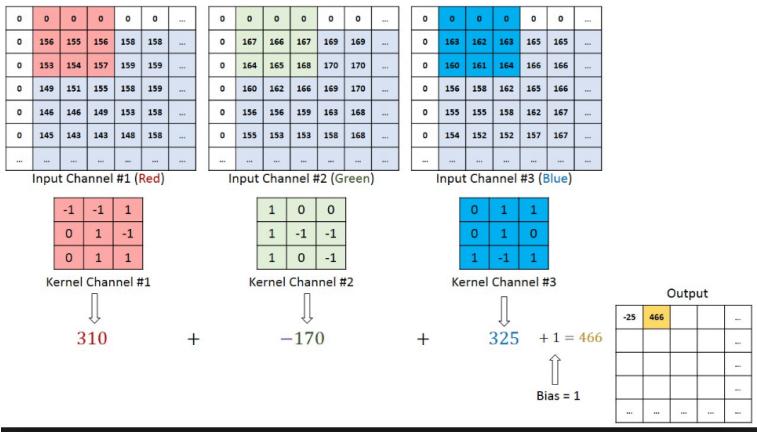
conv_imgray = conv(imgray, sobel_x)

plt.imshow(conv_imgray, cmap='gray', interpolation='none')
plt.pause(1)

sobel_y = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
conv_imgray = conv(imgray, sobel_y)
plt.imshow(conv_imgray, cmap='gray', interpolation='none')
plt.pause(1)

```

Evrişim işlemi gray-scale yerine RGB bir resim üzerinde de yapılabilmektedir. Bu durumda nxn'luk filtreleme matrisi de üç katmanlı olacaktır. Örneğin filtreleme matrisinin 3x3 boyutunda olduğunu düşünelim. Bu matrisin her elemanı bu durumda 3 eleman içerecektir. Bu elemanlar R, G ve B bileşenleri olacaktır. Genellikle buradaki evrişim işlemi sınır ağları için şöyle yapılmaktadır: Dot product işleminde sanki üç ayrı resim varmış gibi her renk bileşeni dot product işlemeye sokulur. Buradan 3 değer elde edilir. Bu üç değer toplanarak teke indirgenir. Bu durum aşağıdaki şekilde gösterilebilir:



Pekiyi evrişim işlemi yapay sinir ağlarına nasıl uygulanmaktadır? Görüntü işlemede kişi belli filtreleri belirleyerek bunu resime uygulamaktadır. Halbuki yapay sinir ağlarında filtreleme matrisinin kendisini sinir ağı bulmaktadır. Yani işlemler tersten gitmektedir. Biz evrişim işlemini yaptığımızı varsayıarak bir ağ modeli oluştururuz. Sonra ağımızı eğitiriz. Bu eğitime göre ağ kendisi bir filtreleme matrisini oluşturmuş olur. Çünkü yapay sinir ağlarında biz aslında neyin filtreleneceğini bilmemekteyiz. Halbuki klasik görüntü işlemede görüntüyü işleyen kişinin bunu bir biçimde bildiği varsayılmaktadır.

Evrişimsel sinir ağlarında yapılan şey aslında her nöronu birbirine bağlamak (dense bağlantısı) yerine bazı nöronları birbirine bağlamaktır. Böylece biz her filtreleme matrisi ile evrişim işlemi yaptığımızda buradaki dot-product aslında nöron bağlantıları anlamına gelmektedir. Örneğin şeklimiz 7x7'lik olsun ve biz de 3x3'lük bir filtre kullanacak olalım:

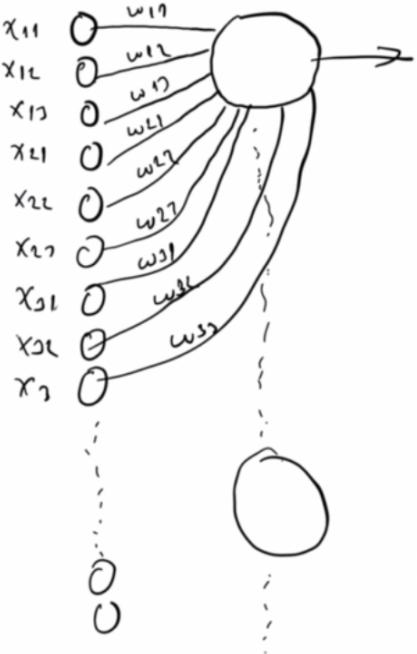
```
x11 x12 x13 x14 x15 x16 x17  
x21 x22 x23 x24 x25 x26 x27  
x31 x32 x33 x34 x35 x36 x37  
x41 x42 x43 x44 x45 x46 x47  
x51 x52 x53 x54 x55 x56 x57  
x61 x62 x63 x64 x65 x66 x67  
x71 x72 x73 x74 x75 x76 x77
```

```
w11 w12 w13  
w21 w22 w23  
w31 w32 w33
```

Buradaki filtreleme işlemine dikkatlice bakalım. Örneğin sol-üst köşe hedef matris pixeli için yapılacak dot-product işlemi şöyledir:

$$x_{11} * w_{11} + x_{12} * w_{12} + x_{13} * w_{13} + x_{21} * w_{21} + x_{22} * w_{22} + x_{23} * w_{23} + x_{31} * w_{31} + x_{32} * w_{32} + x_{33} * w_{33}$$

Bu işlem aslında bir nöron giriş toplam işlemi ile aynıdır. Örneğin:



Burada görüldüğü gibi aslında katmandaki nöron bağlantısı filtreleme matrisi ile asıl resmin ilgili kısımlarının dot-product yapılması ile eşdeğerdir. Başka bir deyişle aslında evrişim işlemi bir katman görevi yapar. Evrişim işlemini yapan katmana yapay sinir ağlarında "evrişim katmanı (convolution layer)" denilmektedir. Ancak bu evrişim katmanında bazı girdi nöronları bazı ağırlıklarla işleme girmektedir. Pekiyi bu modelde evrişim katmanındaki nöron sayısı ne olacaktır? Yanıt $7 - 3 + 1 = 5$ değerinin karesi. Yani 25. Ancak burada bulunması gereken w değerleri 49 tane değildir. Toplamda 9 tanedir. Halbuki biz girdisi 49 nörondan oluşan çıktı 25 nörondan oluşan bir dense bağlantı yapmış olsaydık bulunacak w değerlerinin sayısı $49 * 25 = 1225$ tane olacaktır.

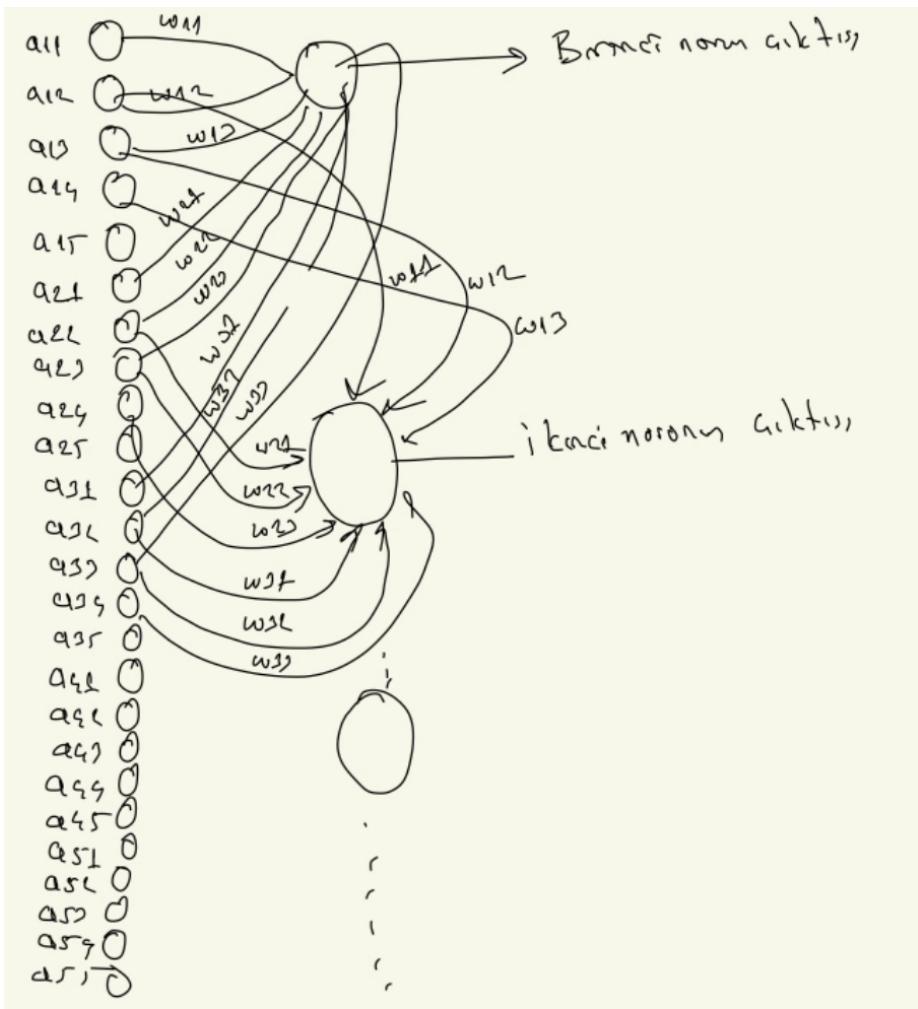
Şimdi evrişim katmanındaki bağlantıları daha iyi anlayabilmek için 5x5'lik bir resim ve 3x3'lük bir filtre örneği verelim. Resmin pikselleri şöyle temsil edilsin:

```
a11 a12 a13 a14 a15
a21 a22 a23 a24 a25
a31 a32 a33 a34 a35
a41 a42 a43 a44 a45
a51 a52 a53 a54 a55
```

Filtreleme matrisi de şöyle temsil edilmiş olsun:

```
w11 w12 w13
w21 w22 w23
w31 w32 w33
```

Evrişim katmanın nöron yapısında girdi nöronlarının sayısı 25 tanedir. Çıktı nöronlarının sayısı 9 tanedir. Toplamda sinir ağı bufiltreleme için 9 tane w değeri bulmaya çalışacaktır. Evrişim katmanındaki bağlantı aşağıdaki gibi olacaktır:



Göründüğü gibi bu işlemle biz bulunması gereken w değerlerini azaltmış olduk. Pekiyi tek bir filtreleme işlemi şeklärin daha iyi anlamlandırılması için yeterli olmakta mıdır? Genellikle hayır. Çünkü tek bir filtreleme şeklärin ancak bazı yönleriyle tanıyalabilir. Bu filtreleme sayısını artırırsak (örneğin 32 gibi) şeklärin başka yönlerden de tanınabilecektir. Dolayısıyla evrişimsel sinir ağlarında evrişim katmanı genellikle tek bir filtre için değil birden fazla filtre için oluşturulmaktadır. Pekiyi yukarıdaki şeklärin 32 tane filtre yerleştirirse şeklärin nasıl olur?

Bu durumda evrişim katmanındaki nöron sayısı $9 * 32$ tane ve bulunması gereken w değerleri de $9 * 32$ tane olacaktır. Evrişim katmanında yine dot-product sonucunun bir aktivasyon fonksiyonuna sokulması gerekmektedir. Evrişim katmanları için de genellikle hidden katmanlarda olduğu gibi "relu" aktivasyon fonksiyonu tercih edilmektedir.

Pekiyi evrişimsel sinir ağlarında evrişim katmanlarının sayısı birden fazla olabilir mi? İşte bir tane evrişimsel katman resimdeki küçük yerelilikleri tanıyalıbmaktadır. Halbuki evrişimsel katmanların çıktılarını başka evrişimsel katmanlara bağladığımızda (yani filtrelemenin üzerine yeniden filtreleme uyguladığımızda) resmin tanınabildiği yerel bölgeyi büyütmiş oluruz. Uygulamada veri bilimcisi genellikle evrişimsel katmanlardaki nöron sayılarını ikiye katlayarak birden fazla katman oluşturmaktadır. (Yani ilk katman 32, sonrası 64 vs.) Görüntü tanıma ağlarında yalnızca evrişimsel katmanların kullanılması da uygun değildir. Çünkü bu durumda da resmin bütününe ilişkin global özellikler gözden kaçırılmış olur. O halde tipik olarak mimari önce birkaç evrişimsel katman daha sonra bir ya da birden fazla dense katmandan oluşturulmaktadır. Böylece hem yerel öğeler hem de global özellikler ağ tarafından tanınabilecektir.

Keras'ta Evrişimsel Sinir Ağlarının Oluşturulması

Keras çok yüksek seviyeli bir kütüphane olduğu için evrişim işlemi de yüksek seviyeli bir biçimde gerçekleştirilmektedir. Programcı yalnızca filtrelerin kaç kaçılı olduğunu ve kaç tane filtre kullanılacağını, padding yapılmış yapılmayacağını, stride değerini belirtmektedir. Zaten bu değerlerin bazıları default argüman almış

durumdadır. Evrişim işlemi tek boyutlu, iki boyutlu ya da üç boyutlu olarak da yapılabilmektedir. Biz burada iki boyutlu evrişim işlemi üzerinde duracağız.

Keras'ta iki boyutlu evrişim işlemleri için Conv2D sınıfı kullanılmaktadır. Yani tipki Dense nesnesi gibi programcı Conv2D nesnelerini yaratarak modeline ekler. Conv2D sınıfının `__init__` metodunun parametrik yapısı şöyledir:

```
Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None,
dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform',
bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None,
activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

Fonksiyonun birinci parametresi kullanılacak filtre sayısını belirtmektedir. İkinci parametre filtrelerin boyunu belirtir. Bu parametre demet olarak girilmelidir. Fonksiyonun stride parametresi stride miktarını yatay ve düşey olarak bir demet biçiminde alır. Bu parametrenin default değerinin (1, 1) olduğunu dikkat ediniz. padding parametresi ya 'valid' değerini alabilir ya da 'same' değerini alabilir. 'valid' padding yapılmayacağı 'same' ise ana resimle aynı boyutta resim elde etmek için padding yapılacağını belirtmektedir. Ayrıca fonksiyonda bir input_shape parametresi de vardır. Bu parametre girdi katmanın boyutunu belirtmektedir. Girdi katmanı Conv2D katmanında matrisel biçimde girilmelidir. Girilen bu matris renkli resimler de içerilsin diye 3 boyutlu olmalıdır. İlk iki boyut satır ve sütun uzunluğunu, üçüncü boyut da kanal sayısını (resimler için pixel renklerinin sayısını) belirtir. Örneğin Minst'teki bir resim için girdi boyutları 28X28X1 biçiminde bir matris ile ifade edilir. Tabii girdi matrisinde birden fazla resim olduğuna göre bu duurmda girdi matrisinin boyutu "resim sayısı x 28 x 28 x 1" olacaktır.

Burada öncelikle adım adım Mnist örneğinin evrişimsel modelini oluşturmaya çalışalım.

```
from tensorflow.keras.datasets import mnist
(training_set_x, training_set_y), (test_set_x, test_set_y) = mnist.load_data()
```

Burada `training_dataset_x` (60000, 28, 28) boyutundadır. Bizim bu matrisi (60000, 28, 28, 1) haline getirmemiz gereklidir. Çünkü Conv2D katmanı üç boyutlu bir matrisi girdi olarak istemektedir. Aynı işlemi `test_dataset_x` için de yapmalıyız:

```
training_dataset_x = training_dataset_x.reshape(-1, 28, 28, 1)
test_dataset_x = test_dataset_x.reshape(-1, 28, 28, 1)
```

Anımsanacağı gibi `mnist` tarafından bize verilen matrislerin `dtype` türü `uint8` biçimindedir. Bizim bunu `float32`'ye dönüştürmemiz uygun olur.

```
training_dataset_x = training_dataset_x.astype('float32')
test_dataset_x = test_dataset_x.astype('float32')
```

Yine normalizasyon işlemini yapmalıyız:

```
training_dataset_x = training_dataset_x / 255
test_dataset_x = test_dataset_x / 255
```

Şimdi de `training_set_y` ve `test_set_y` verilerini 10 sütunlu hale getirmek için "one hot encoding" işlemlerini yapalım:

```
from tensorflow.keras.utils import to_categorical
training_dataset_y = to_categorical(training_dataset_y)
test_dataset_y = to_categorical(test_dataset_y)
```

Artık modelimizi oluşturabiliriz:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, Flatten
```

```

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=(28, 28, 1), activation='relu',
name='Convolution-1'))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', name='Convolution-2'))
model.add(Flatten())
model.add(Dense(128, activation='relu', name='Hidden-1'))
model.add(Dense(128, activation='relu', name='Hidden-2'))
model.add(Dense(10, activation='softmax', name='Output'))

model.summary()

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=3, batch_size=512,
validation_split=0.2)

```

Burada Conv2D katmanından sonra modele bir Flatten katmanı eklenmiştir. Bu katman matrisi tek boyutlu bir vektör haline getirmektedir. Çünkü Conv2D katmanın çıkıştı tek boyutlu bir vektör değildir, iki boyutlu bir matristir. Halbuki Dense katmanı bizden tek boyutlu bir girdi istemektedir.

Burada model özeti şöyle oluşmaktadır:

Layer (type)	Output Shape	Param #
<hr/>		
Convolution-1 (Conv2D)	(None, 26, 26, 32)	320
Convolution-2 (Conv2D)	(None, 24, 24, 64)	18496
flatten_1 (Flatten)	(None, 36864)	0
Hidden-1 (Dense)	(None, 128)	4718720
Hidden-2 (Dense)	(None, 128)	16512
Output (Dense)	(None, 10)	1290
<hr/>		
Total params: 4,755,338		
Trainable params: 4,755,338		
Non-trainable params: 0		

Buradaki sınır ağında tahmin edilecek parametre sayıları nereden gelmektedir? Birinci evrişimsel katmanda genişli $k=28$, yükseklik = 28 ve renk sayısı=1 olan bir girdi uygulanmıştır. Filtre matrisi 3×3 ve filtre sayısı da 32'dir. Her filtre için bias değeri de kullanılmaktadır. (Nöronlarda kullanılan bias değerleri her nöron için farklı değil her filtre için tektir.) Dolayısıyla ilk katman için tahmin edilecek parametre sayısı $(9 + 1) * 32 = 320$ 'dir. Şimdi de ikinci evrişimsel katmandaki parametre sayılarını hesaplamaya çalışalım. Birinci katmanın çıkışında $26 * 26$ boyutunda 32 farklı resim vardır. Çünkü 32 farklı filtre uygulanmıştır. İkinci katmandaki filtre sayısını 64 almıştır. İşte bu 64 aslında her 32'lük resim için 64 filtre kullanılacağı anlamına gelmektedir. Bu durumda toplam filtre sayısı $32 * 64 = 2048$ tanedir. Her filtrede $3 * 3 = 9$ tane w değerleri vardır. Böylece $2048 * 9 = 18432$ adet w değeri elde edilir. Bu katmandaki toplam 64 filtre için de 64 bias değeri tahmin edilecektir. Bu durumda bu katmanın tahmin edilecek toplam parametre sayısı $18432 + 64 = 18496$ olacaktır. Flatten katmanı her ne kadar bir katman gibi modele ekleniyse de bu katmanın tek yaptığı şey matrisel çıktıyı Dense katmanın istediği tek boyutlu vektörel biçimde dönüştürmektedir. Modele bundan sonra 128'lik bir Dense katman eklenmiştir. Bu Dense katmanın girdisi $24 * 24 * 64$ 'lük bir nöron vektöridür. Bu vektörün her elemanı 128 nöronla dense bağlanmıştır. Bu durumda bu katmandaki toplam w değerleri $24 * 24 * 64 * 128 = 4718592$ olur. Bu değere 128 bias değerini de toplarsak 4718720 değeri elde edilir. İkinci hidden katman için girdi nöronlarının sayısı 128'dir. Bu katmanda 128 tane çıktı nöronu olacaktır. Bu durumda $128 * 128 = 16384$ eder. Tabii her nöronun da tahmin edilecek bir bias değeri vardır. O halde $16384 + 128 = 16512$ olur. Çıktı katmanın girdisi 128 nördən oluşmaktadır. Bu 128 nöron 10 nörona dense bağlanmıştır. Buradan $128 * 10 = 1280$ elde edilir. Bu değere 10 bias toplanırsa 1290 değeri elde edilecektir. İşte tüm bu değerlerin toplamı da modelin toplam parametre sayısı verir. Bu sayı modelimiz için 4738826'dır.

Modelimizi eğittiğinden sonra elde ettiğimiz değerler şöyledir:

```

Epoch 1/3
48000/48000 [=====] - 87s 2ms/step - loss:
0.3243 - acc: 0.9039 - val_loss: 0.0993 - val_acc: 0.9717
Epoch 2/3
48000/48000 [=====] - 85s 2ms/step - loss:
0.0692 - acc: 0.9797 - val_loss: 0.0582 - val_acc: 0.9832
Epoch 3/3
48000/48000 [=====] - 84s 2ms/step - loss:
0.0402 - acc: 0.9877 - val_loss: 0.0523 - val_acc: 0.9852

```

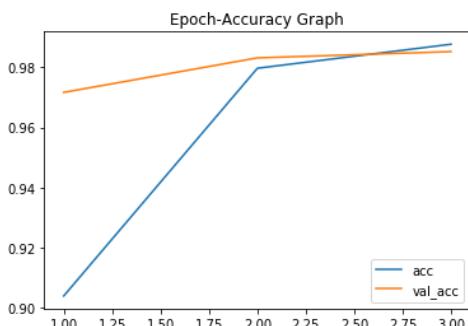
Epoch-Accuracy grafiğini de çizelim:

```

import matplotlib.pyplot as plt

plt.title('Epoch-Accuracy Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['acc', 'val_acc'])

```



Burada normal accuracy değeriyle sınıma accuracy değerleri arasında farklılık olsa da çok büyük boyutta değildir. Ancak epoch sayısı 3 civarında tıuutlursa daha tutarlı bir model elde edilebilir.

Şimdi de modelimizi Nnist verileriyle test edelim ve test sonucunda elde edilen değerlere bakalım:

```

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

```

Elde edilen değerler şöyledir:

```
loss = 0.04456091890308307, accuracy = 0.9853
```

Göründüğü gibi nispeten yüksek bir accuracy değeri elde edildi. Şimdi kendi çizdiğimiz rakamlar üzerinde test işlemlerimizi yapalım:

```

import numpy as np

def rgb2gray(rgb):
    return np.dot(rgb[:, :, :], [0.2989, 0.5870, 0.1140])

import glob
from matplotlib.image import imread

for path in glob.glob('*.*'):
    img = imread(path)
    gray = rgb2gray(img)
    gray = gray.reshape(1, 28, 28, 1)

```

```

predict_result = model.predict(gray)
number = predict_result.argmax(axis=1)
print('{} --> {} {}'.format(path, number, 'Doğru' if int(path[5]) == number[0] else
'Yanlış'))

```

Burada kendi çizimlerimize dayalı olarak çıkan sonuç şöyledir:

```

test-0-1.png --> [9] Yanlış
test-1-1.png --> [6] Yanlış
test-2-1.png --> [2] Doğru
test-3-1.png --> [3] Doğru
test-4-1.png --> [4] Doğru
test-5-1.png --> [3] Yanlış
test-6-1.png --> [5] Yanlış
test-7-1.png --> [7] Doğru
test-8-1.png --> [8] Doğru
test-9-1.png --> [9] Doğru

```

Burada görüldüğü gibi kendi çizimlerimizdeki başarı %98 gibi yüksek bir değerde değildir? Pekiyi bunun sebebi ne olabilir? Tabii ki eğitimde kullanılan şekillerle bizim çizdiğimiz şeiller arasında uyumsuzluk bunun en önemli nedenidir. Eğer biz eğitimimizi tamamen Mnist değil de kendi çizimlerimiz üzerinde yapsaydık sonuç yüksek olacaktı.

Yukarıdaki örneğin kaynak kodları şöyledir:

```

from tensorflow.keras.datasets import mnist

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) = mnist.load_data()
training_dataset_x = training_dataset_x.reshape(-1, 28, 28, 1)
test_dataset_x = test_dataset_x.reshape(-1, 28, 28, 1)

training_dataset_x = training_dataset_x.astype('float32')
test_dataset_x = test_dataset_x.astype('float32')

training_dataset_x /= 255
test_dataset_x /= 255

from tensorflow.keras.utils import to_categorical

training_dataset_y = to_categorical(training_dataset_y)
test_dataset_y = to_categorical(test_dataset_y)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, Flatten

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=(28, 28, 1), activation='relu',
name='Convolution-1'))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', name='Convolution-2'))
model.add(Flatten())
model.add(Dense(128, activation='relu', name='Hidden-1'))
model.add(Dense(128, activation='relu', name='Hidden-2'))
model.add(Dense(10, activation='softmax', name='Output'))

model.summary()

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=3, batch_size=512,
validation_split=0.2)

import matplotlib.pyplot as plt

```

```

plt.title('Epoch-Accuracy Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['acc', 'val_acc'])

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

import numpy as np

def rgb2gray(rgb):
    return np.dot(rgb[:, :, :], [0.2989, 0.5870, 0.1140])

import glob
from matplotlib.image import imread

for path in glob.glob('*.*'):
    img = imread(path)
    gray = rgb2gray(img)
    gray = gray.reshape(1, 28, 28, 1)

    predict_result = model.predict(gray)
    number = predict_result.argmax(axis=1)
    print('{} --> {}  {}'.format(path, number, 'Doğru' if int(path[5]) == number[0] else 'Yanlış'))

```

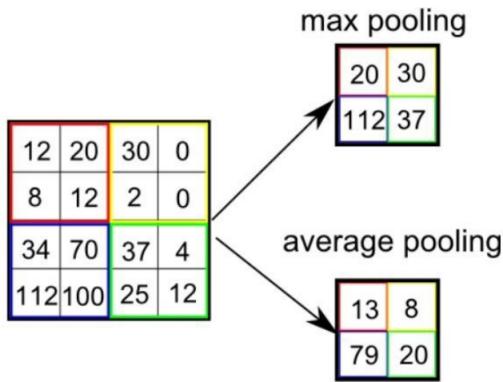
Evrişimsel Sinir Ağlarında Nöron Sayılarının Azaltılması (Downsampling)

Yukarıda da belirttiğimiz gibi evrişimsel ağlar özellikle görüntü tanımda çok kullanılmaktadır. Görüntü verileri de büyük olma eğilimindedir. (Her ne kadar Mnist görselleri 28x28'lük gray-scale olsa da gerçekte çoğu görsel daha yüksek çözünürlükte ve RGB biçimindedir.) Örneğin önceki bölümdeki Mnist modelinde oluşturduğumuz evrişimsel ağda 4 milyonun üzerinde ağırlık değerleri tahmin edilmeye çalışılan nöron vardır. Modelde çok nöronun olmasının şu dezavantajları söz konusudur:

- Modelin eğitilmesi daha fazla zaman alır.
- Model verilerinin saklanması sırasında daha fazla disk alanına gereksinim duyulur.
- Modeldeki parametre sayılarının artması da ayrıca başka bir yönden bir "overfitting" kaynağı olabilmektedir.

İşte bu nedenlerden dolayı modeldeki nöron sayılarını azaltmak isteyebiliriz. Bunun için ilk akla gelen yöntem evrişimsel katmalarındaki stride değerinin yükseltilmesidir. (Biz yukarıdaki örnekte default stride değerini 1 almıştık.) Ancak stride yönteminden daha etkin olduğu düşünülen bir yönteme "pooling" denilmektedir. Pooling yöntemi genellikle 2x2'lük stride = 2 değeriyle uygulanır.

İki temel pooling yöntemi vardır: MaxPooling ve AveragePooling. MaxPooling yönteminde 2x2'lük küçük matristeki en büyük eleman bulunur ve 4 eleman bu en büyük elemanla değiştirilir. AveragePooling yönteminde ise bu dört elemanın ortalaması alınarak yer değiştirme yapılır. MaxPooling yöntemi AveragePooling yöntemine göre çok daha yaygın tercih edilmektedir. Örneğin:



Keras'ta pooling işlemi yapan özel katmanlar vardır. Yani biz bu pooling işlemlerini zaten Keras'ta var olan MaxPooling2D ve AveragePooling2D isimli sınıflarla yaparız.

Pekiyi Pooling işlemi görsel verileri işleyen ağlarda nerede uygulanmalıdır? İşte genellikle bu işlem her Conv katmanından sonra bir kez uygulanmaktadır. MaxPooling2D sınıfının `__init__` metodunun parametrik yapısı şöyledir:

```
MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None)
```

Fonksiyonun birinci parametresi pooling işlemi için uygulanacak matris boyutunu belirtir. Bu değer tipik olarak (2, 2) girilmektedir. Zaten (2, 2) değeri default argüman olarak verilmiştir. strides parametresi None girilirse (default durum) pool_size'daki değerler dikkate alınmaktadır. (Örneğin default durumda pool_size=(2, 2) olduğuna göre stride değerleri de yatayda 2 ve düşeyde 2 biçimindedir.) Yine padding 'valid' ya da 'same' biçiminde girilebilmektedir.

Şimdi yukarıdaki modelimizi Pooling katmanlarını ekleyerek yeniden oluşturalım:

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=(28, 28, 1), activation='relu',
name='Convolution-1'))
model.add(MaxPooling2D(name='MaxPooling2D-1'))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', name='Convolution-2'))
model.add(MaxPooling2D(name='MaxPooling2D-2'))
model.add(Flatten())
model.add(Dense(128, activation='relu', name='Hidden-1'))
model.add(Dense(128, activation='relu', name='Hidden-2'))
model.add(Dense(10, activation='softmax', name='Output'))

model.summary()
```

summary fonksiyonundan elde edilen çıktı şöyledir:

Layer (type)	Output Shape	Param #
<hr/>		
Convolution-1 (Conv2D)	(None, 26, 26, 32)	320
MaxPooling2D-1 (MaxPooling2D)	(None, 13, 13, 32)	0
Convolution-2 (Conv2D)	(None, 11, 11, 64)	18496
MaxPooling2D-2 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten_1 (Flatten)	(None, 1600)	0
Hidden-1 (Dense)	(None, 128)	204928
Hidden-2 (Dense)	(None, 128)	16512

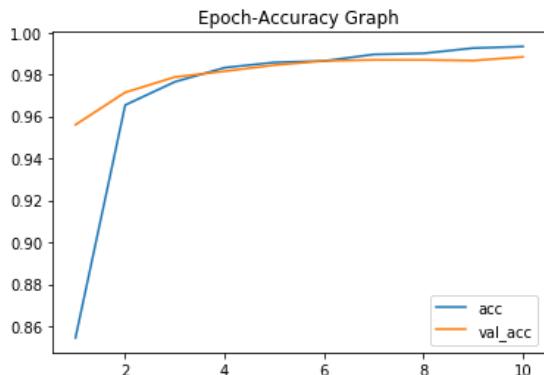
```

Output (Dense)           (None, 10)          1290
=====
Total params: 241,546
Trainable params: 241,546
Non-trainable params: 0

```

Göründüğü gibi modelin eğitilecek parametre sayısı 4 milyon civarından 241 bin civarına düşmüştür.

Şimdi modeli 10 epoch için eğitelim. Elde ettiğimiz accuracy grafiği şöyle olacaktır:



Burada 6 ya da 7 epoch civarında accuracy değeri ile val_accuracy değerinin uyumlu olduğu sonra accuracy değerinin val_accuracy değerinden yükseğe çıktığı görülmektedir. Bu da hafif bir "overfit" şüphesi uyandırmaktadır. O halde biz buradaki eğitimi 6 ya da 7 epoch civarında optimal biçimde kesebiliriz.

Son modelin kodları şöyledir:

```

from tensorflow.keras.datasets import mnist

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) = mnist.load_data()
training_dataset_x = training_dataset_x.reshape(-1, 28, 28, 1)
test_dataset_x = test_dataset_x.reshape(-1, 28, 28, 1)

training_dataset_x = training_dataset_x.astype('float32')
test_dataset_x = test_dataset_x.astype('float32')

training_dataset_x /= 255
test_dataset_x /= 255

from tensorflow.keras.utils import to_categorical

training_dataset_y = to_categorical(training_dataset_y)
test_dataset_y = to_categorical(test_dataset_y)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, MaxPooling2D, Flatten

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=(28, 28, 1), activation='relu',
name='Convolution-1'))
model.add(MaxPooling2D(name='MaxPooling2D-1'))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', name='Convolution-2'))
model.add(MaxPooling2D(name='MaxPooling2D-2'))
model.add(Flatten())
model.add(Dense(128, activation='relu', name='Hidden-1'))
model.add(Dense(128, activation='relu', name='Hidden-2'))
model.add(Dense(10, activation='softmax', name='Output'))

```

```

model.summary()

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=7, batch_size=512,
validation_split=0.2)

import matplotlib.pyplot as plt

plt.title('Epoch-Accuracy Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['acc', 'val_acc'])

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

import numpy as np

def rgb2gray(rgb):
    return np.dot(rgb[:, :, :], [0.2989, 0.5870, 0.1140])

import glob
from matplotlib.image import imread

for path in glob.glob('*.*'):
    img = imread(path)
    gray = rgb2gray(img)
    gray = gray.reshape(1, 28, 28, 1)

    predict_result = model.predict(gray)
    number = predict_result.argmax(axis=1)
    print('{} --> {}  {}'.format(path, number, 'Doğru' if int(path[5]) == number[0] else
'Yanlış'))

```

RGB Görüntünün Tanınmasına İlişkin CIFAR-10 Örneği

Cifar-10 Keras'in içerisinde de var olan eğitimde yaygın kullanılan bir görsel veritabanıdır. Cifar-10 içerisinde 60000 tane her biri 32X32 boyutlarında RGB bitmap görüntüler vardır. Bu görüntülerdeki her pixel 3 byte olup Red, Green, Blue tonal bileşenlerini içermektedir. Bu uygulamanın amacı bir görselin hangi kategoriye ilişkin olduğunu anlayabilmektir. Yani burada da "single label multiclass" bir görüntü tanıma problemi söz konusudur. Görsellerin her biri şu sınıflardan birini ilişkendir:

```
class_names = [airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck]
```

Cifar-10'un ayrıca 100 sınıflık Cifar-100 isimli bir versiyonu da vardır.

Cifar-10 için modeli şöyle oluşturabiliriz:

```
from tensorflow.keras.datasets import cifar10

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
cifar10.load_data()

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship',
'truck']
```

Okumuz verilerin shape özellikleri şöyledir:

```
print('Shape of training_dataset_x: {}'.format(training_dataset_x.shape))
print('Shape of training_dataset_y: {}'.format(training_dataset_y.shape))
print('Shape of test_dataset_x: {}'.format(test_dataset_x.shape))
print('Shape of test_dataset_y: {}'.format(test_dataset_y.shape))
```

Elde edilen sonuçlar şöyledir:

```
Shape of training_dataset_x: (50000, 32, 32, 3)
Shape of training_dataset_y: (50000, 1)
Shape of test_dataset_x: (10000, 32, 32, 3)
Shape of test_dataset_y: (10000, 1)
```

Sınıflar yine 0'dan 9'a kadar bir tamsayı ile ifade edilmiştir. Şimdi ilk 10 resmi deneme amaçlı çizdirelim:

```
import matplotlib.pyplot as plt

for i in range(10):
    plt.title(class_names[training_dataset_y[i, 0]])
    plt.imshow(training_dataset_x[i])
    plt.pause(1)
```

Şimdi ndarray dizisinin dtype'ını float32 yapalım:

```
training_set_x = training_dataset_x.astype('float32')
test_set_x = test_dataset_x.astype('float32')
```

Min-max ölçeklendirimesini uygulaayalım:

```
training_dataset_x = training_dataset_x / 255
test_dataset_y = training_dataset_y / 255
```

Şimdi training_dataset_y ve test_dataset_y değerleri üzerinde "one hot encoding" işlemi yapalım:

```
from tensorflow.keras.utils import to_categorical

training_dataset_y = to_categorical(training_dataset_y)
test_dataset_y = to_categorical(test_dataset_y)
```

Artık modelimizi kurabiliriz:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, Flatten, MaxPooling2D

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=(32, 32, 3), activation='relu',
name='Convolution-1'))
model.add(MaxPooling2D(name='MaxPooling-1'))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', name='Convolution-2'))
model.add(MaxPooling2D(name='MaxPooling-2'))
model.add(Flatten(name='Flatten'))
model.add(Dense(128, activation='relu', name='Hidden-1'))
model.add(Dense(10, activation='softmax', name='Output'))

model.summary()

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit(training_set_x, training_set_y, epochs=20, batch_size=512,
validation_split=0.2)
```

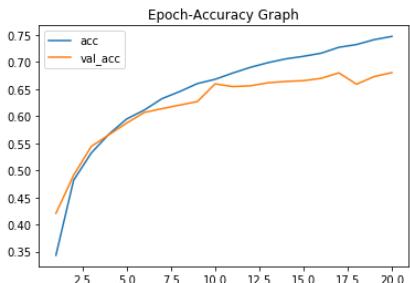
Şimdi de Epch-Accuracy grafiğini çizdirelim:

```

import matplotlib.pyplot as plt

plt.title('Epoch-Accuracy Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Loss'
plt.plot(range(1, len(history.epoch) + 1), history.history['acc'])
plt.plot(range(1, len(history.epoch) + 1), history.history['val_acc'])
plt.legend(['acc', 'val_acc'])

```



Şimdi de modelimizi test edelim:

```

loss, accuracy = model.evaluate(test_set_x, test_set_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

```

Buradan elde ettiğimiz değer %65 civarındadır. Eğer biz bu esimler rastgele sınıflandırmaya çalışsaydık başarımız %10 civarında olacaktı.

Şimdi de kendi buldumuz fotoğrafları sınıflandırmaya çalışalım:

```

from matplotlib.pyplot import imread
import glob

for path in glob.glob('*.*'):
    img = imread(path)
    img = img.reshape(1, 32, 32, 3)
    predict_result = model.predict(img)
    number = predict_result.argmax(axis=1)
    print('{} -> {}'.format(path, class_names[number[0]]))

```

Cifar-10 uygulamasının tüm kodu aşağıdaki gibidir:

```

from tensorflow.keras.datasets import cifar10

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
cifar10.load_data()

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship',
'truck']

print('Shape of training_dataset_x: {}'.format(training_dataset_x.shape))
print('Shape of training_dataset_y: {}'.format(training_dataset_y.shape))
print('Shape of test_dataset_x: {}'.format(test_dataset_x.shape))
print('Shape of test_dataset_y: {}'.format(test_dataset_y.shape))

import matplotlib.pyplot as plt

for i in range(10):
    plt.title(class_names[training_dataset_y[i, 0]])
    plt.imshow(training_dataset_x[i])
    plt.pause(1)

```

```

training_set_x = training_dataset_x.astype('float32')
test_set_x = test_dataset_x.astype('float32')

training_dataset_x = training_dataset_x / 255
test_dataset_x = test_dataset_x / 255

from tensorflow.keras.utils import to_categorical

training_dataset_y = to_categorical(training_dataset_y)
test_dataset_y = to_categorical(test_dataset_y)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, Flatten, MaxPooling2D

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=(32, 32, 3), activation='relu',
name='Convolution-1'))
model.add(MaxPooling2D(name='MaxPooling-1'))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', name='Convolution-2'))
model.add(MaxPooling2D(name='MaxPooling-2'))
model.add(Flatten(name='Flatten'))
model.add(Dense(128, activation='relu', name='Hidden-1'))
model.add(Dense(128, activation='relu', name='Hidden-2'))
model.add(Dense(10, activation='softmax', name='Output'))

model.summary()

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=15, batch_size=512,
validation_split=0.2)

import matplotlib.pyplot as plt

plt.title('Epoch-Accuracy Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['acc', 'val_acc'])

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

from matplotlib.pyplot import imread

import glob

for path in glob.glob('*.*'):
    img = imread(path)
    img = img.reshape(1, 32, 32, 3)
    predict_result = model.predict(img)
    number = predict_result.argmax(axis=1)
    print('{} -> {}'.format(path, class_names[number[0]]))

```

RGB Görüntünün Tanınmasına İlişkin CIFAR-100 Örneği

Cifar-100 veri kümesi Cifar-10 veri kümelerinin 100 sınıf içeren genişletilmiş bir biçimidir. Bu veri kümesinde her biri 600 resimden oluşan 100 farklı sınıfta resim bulunmaktadır. Cifar-100 örneği aslında Cifar-10 örneğiyle birkaç küçük farklılık dışında aynıdır. Cifar-100 modelini kurarken çıktı katmanındaki nöronların sayısının 100 tane olması gereklidir. Cifar-100 içerisindeki resim sınıfları index numaralarıyla şöyledir:

```

class_names = [
    'apple', 'aquarium_fish', 'baby', 'bear', 'beaver', 'bed', 'bee', 'beetle',
    'bicycle', 'bottle', 'bowl', 'boy', 'bridge', 'bus', 'butterfly', 'camel',
    'can', 'castle', 'caterpillar', 'cattle', 'chair', 'chimpanzee', 'clock',
    'cloud', 'cockroach', 'couch', 'crab', 'crocodile', 'cup', 'dinosaur',
    'dolphin', 'elephant', 'flatfish', 'forest', 'fox', 'girl', 'hamster',
    'house', 'kangaroo', 'keyboard', 'lamp', 'lawn_mower', 'leopard', 'lion',
    'lizard', 'lobster', 'man', 'maple_tree', 'motorcycle', 'mountain', 'mouse',
    'mushroom', 'oak_tree', 'orange', 'orchid', 'otter', 'palm_tree', 'pear',
    'pickup_truck', 'pine_tree', 'plain', 'plate', 'poppy', 'porcupine',
    'possum', 'rabbit', 'raccoon', 'ray', 'road', 'rocket', 'rose',
    'sea', 'seal', 'shark', 'shrew', 'skunk', 'skyscraper', 'snail', 'snake',
    'spider', 'squirrel', 'streetcar', 'sunflower', 'sweet_pepper', 'table',
    'tank', 'telephone', 'television', 'tiger', 'tractor', 'train', 'trout',
    'tulip', 'turtle', 'wardrobe', 'whale', 'willow_tree', 'wolf', 'woman',
    'worm'
]

```

Cifar-100 için oluşturulan model Cifar-10 örneğinin benzeri biçimindedir:

```

from tensorflow.keras.datasets import cifar100

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
cifar100.load_data()

class_names = [
    'apple', 'aquarium_fish', 'baby', 'bear', 'beaver', 'bed', 'bee', 'beetle',
    'bicycle', 'bottle', 'bowl', 'boy', 'bridge', 'bus', 'butterfly', 'camel',
    'can', 'castle', 'caterpillar', 'cattle', 'chair', 'chimpanzee', 'clock',
    'cloud', 'cockroach', 'couch', 'crab', 'crocodile', 'cup', 'dinosaur',
    'dolphin', 'elephant', 'flatfish', 'forest', 'fox', 'girl', 'hamster',
    'house', 'kangaroo', 'keyboard', 'lamp', 'lawn_mower', 'leopard', 'lion',
    'lizard', 'lobster', 'man', 'maple_tree', 'motorcycle', 'mountain', 'mouse',
    'mushroom', 'oak_tree', 'orange', 'orchid', 'otter', 'palm_tree', 'pear',
    'pickup_truck', 'pine_tree', 'plain', 'plate', 'poppy', 'porcupine',
    'possum', 'rabbit', 'raccoon', 'ray', 'road', 'rocket', 'rose',
    'sea', 'seal', 'shark', 'shrew', 'skunk', 'skyscraper', 'snail', 'snake',
    'spider', 'squirrel', 'streetcar', 'sunflower', 'sweet_pepper', 'table',
    'tank', 'telephone', 'television', 'tiger', 'tractor', 'train', 'trout',
    'tulip', 'turtle', 'wardrobe', 'whale', 'willow_tree', 'wolf', 'woman',
    'worm'
]

print('Shape of training_dataset_x: {}'.format(training_dataset_x.shape))
print('Shape of training_dataset_y: {}'.format(training_dataset_y.shape))
print('Shape of test_dataset_x: {}'.format(test_dataset_x.shape))
print('Shape of test_dataset_y: {}'.format(test_dataset_y.shape))

import matplotlib.pyplot as plt

for i in range(10):
    print(class_names[training_dataset_y[i][0]])
    plt.imshow(training_dataset_x[i])
    plt.pause(1)

image_index = class_names.index('sweet_pepper')
for index, dataset in enumerate(training_dataset_x[(training_dataset_y ==
image_index).reshape(-1)]):
    plt.imshow(dataset)
    plt.pause(1)
    if index == 10:
        break

```

```

training_set_x = training_dataset_x.astype('float32')
test_set_x = test_dataset_x.astype('float32')

training_dataset_x = training_dataset_x / 255
test_dataset_x = test_dataset_x / 255

from tensorflow.keras.utils import to_categorical

training_dataset_y = to_categorical(training_dataset_y)
test_dataset_y = to_categorical(test_dataset_y)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Dense, Flatten, MaxPooling2D

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=(32, 32, 3), activation='relu',
name='Convolution-1'))
model.add(MaxPooling2D(name='MaxPooling-1'))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', name='Convolution-2'))
model.add(MaxPooling2D(name='MaxPooling-2'))
model.add(Flatten(name='Flatten'))
model.add(Dense(128, activation='relu', name='Hidden-1'))
model.add(Dense(128, activation='relu', name='Hidden-2'))
model.add(Dense(100, activation='softmax', name='Output'))

model.summary()

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=15, batch_size=512,
validation_split=0.2)

import matplotlib.pyplot as plt

plt.title('Epoch-Accuracy Graph')
plt.xlabel = 'Epochs'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['acc', 'val_acc'])

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

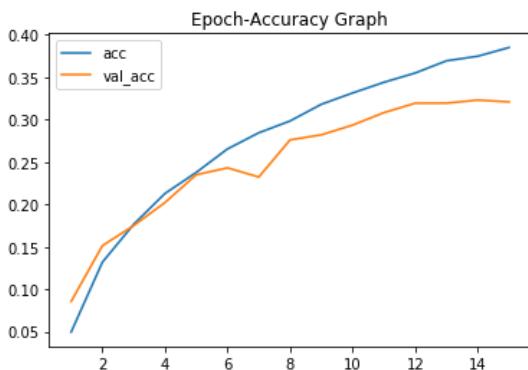
from matplotlib.pyplot import imread

import glob

for path in glob.glob('*.*'):
    img = imread(path)
    img = img.reshape(1, 32, 32, 3)
    predict_result = model.predict(img)
    number = predict_result.argmax(axis=1)
    print('{} -> {}'.format(path, class_names[number[0]]))

```

Aşağıda Epoch-Validation Accuracy grafiğini görüyorsunuz:



Bu grafiğe bakıldığında epoch temelli bir "overfit" durumunun söz konusu olduğu görülmektedir. O halde burada 12 civarında epoch yeterli gibi gözükmektedir. Grafikte kesim noktasının 5 civarında bir epoch'ta olduğu anlaşılmaktadır. Ancak Epoch ilerledikçe overfit durumu artmasına karşın aynı zamanda "validation accuracy" değeri de önemli ölçüde artmaktadır. Bu nedenle eğitimi 5 civarında epoch'ta kesmek uygun değildir. Grafikte "validation accuracy"nin artık artmamamaya başladığı nokta temel alınabilir.

Keras'ta EarlyStopping ve ModelCheckpoint Callback Sınıfları

Keras'taki bazı callback sınıflarını daha önce incelemiştik. Burada konu itibariyle iki callback sınıfını daha inceleyeceğiz. EarlyStopping eğitimi durdurmak için kullanılan bir callback sınıfıdır. Yani eğitim sırasında epoch'lardaki sınıma (validation) değerleri belli koşulları sağladığında (örneğin artık daha fazla iyileşmediği ya da gerilediği zaman) eğitimi durdurabiliyoruz. Çünkü overfit durumlarında eğitimin devam ettirilmesi toplamda daha kötü sonuçlara yol açabilmektedir. Şüphesiz programcı önce yüksek bir epoch'la denemey yapıp sonra grafiklere bakarak uygun epoch miktarını belirleyebilmektedir. (Biz de şimdide kadar hep böyle yaptık.) İşte EarlyStopping callback sınıfı bu işlemi daha pratik hale getirmek amacıyla düşünülmüştür. EarlyStopping sınıfının `__init__` metodunun parametrik yapısı şöyledir:

```
EarlyStopping(monitor='val_loss', min_delta=0, patience=0, verbose=0, mode='auto',
baseline=None, restore_best_weights=False)
```

Fonksiyonun birinci parametresi hangi metrik değerinin izleneceğini belirtmektedir. Bu parametrenin default değerinin 'val_loss' biçiminde olduğuna dikkat ediniz. Yani default durumda "validation loss" değerine bakılarak erken sonlandırma yapılacaktır. EarlyStopping callback sınıfı her zaman epoch'lardan sonra devreye girmektedir. `patience` parametresine epoch sayısı girilir. Eğitim burada girilen epoch sayısı kadar epoch işleminde ilgili değerde olumlu bir değişiklik yoksa sonlandırılır. Örneğin bu parametrenin 5 girildiğini varsayıyalım. Artık 5 epoch'ta bir farklılık oluşmuyorsa işlem sonlandırılabilir. `patience` parametresindeki duyarlılık `min_delta` parametresiyle ayarlanmaktadır. `min_delta` parametresinin default değeri 0'dır. Örneğin biz `min_delta` parametresi için 0.2 değeri girmiş olalım. Bu durumda `patience` değeri 5 ise, 5 epoch süresince validation loss değerinde 0.2'den daha yüksek bir iyileşme olmamışsa eğitim sonlandırılır. `mode` parametresi üç değer alabilir: "min", "max" ve "auto". Bu parametre `monitor` parametresine göre belirlenmelidir. Örneğin `monitor` parametresi "val_loss" ise biz bu değerin düşmesine yönelik bir sonlandırma yapmak isteriz. Bu durumda `mode` parametresi 'min' girilebilir. Ancak örneğin `monitor` parametresi "val_acc" ise bu durumda biz bu değerin yükselmesi yönünde bir ilgi içerisinde oluruz. O halde `mode` parametresinin bu durumda max girilmesi gereklidir. "auto" ise bunu otomatik yapmaktadır. Bu parametrenin default değerinin "auto" olduğunu görüyorsunuz. Fonksiyonun `baseline` parametresi ise sonlandırma manının monitor edilen özellik belli bir değere eriştiğinde yapılmasını sağlamak için kullanılmaktadır. `restore_best_weights` parametresi True girilirse (default değer False biçimdedir) bu durumda model sonlandırılmadan önce ağıın ağırlık değerleri o zamana kadarki en iyi monitor değeri ile set edilmektedir.

Örneğin Mnist veri kümesi için aşağıdaki gibi bir EarlyStopping callback oluşturmuş olalım:

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

from tensorflow.keras.callbacks import EarlyStopping
esc = EarlyStopping(monitor='val_acc', patience=5, verbose=1, restore_best_weights=True)
```

```
history = model.fit(training_set_x, training_set_y, epochs=200, batch_size=512,
validation_split=0.2, callbacks=[esc])
```

Burada patience değerinin 5 olarak girildiğine dikkat ediniz. İşte model gitgide iyileştirilirken 5 epoch süresince elde edilen değer iyileştirilmemez işlem sonlandırılacaktır. Bu örnekte eğitimin çıktısı aşağıdaki gibi elde edilmiştir:

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/200
48000/48000 [=====] - 4s 76us/step - loss: 0.4050 - acc: 0.8885 -
val_loss: 0.1610 - val_acc: 0.9544
Epoch 2/200
48000/48000 [=====] - 3s 59us/step - loss: 0.1236 - acc: 0.9638 -
val_loss: 0.1267 - val_acc: 0.9627
Epoch 3/200
48000/48000 [=====] - 3s 57us/step - loss: 0.0795 - acc: 0.9758 -
val_loss: 0.0972 - val_acc: 0.9709
Epoch 4/200
48000/48000 [=====] - 3s 54us/step - loss: 0.0507 - acc: 0.9850 -
val_loss: 0.0909 - val_acc: 0.9712
Epoch 5/200
48000/48000 [=====] - 3s 52us/step - loss: 0.0367 - acc: 0.9882 -
val_loss: 0.0988 - val_acc: 0.9716
Epoch 6/200
48000/48000 [=====] - 3s 53us/step - loss: 0.0273 - acc: 0.9917 -
val_loss: 0.0947 - val_acc: 0.9737
Epoch 7/200
48000/48000 [=====] - 3s 55us/step - loss: 0.0225 - acc: 0.9929 -
val_loss: 0.0962 - val_acc: 0.9739
Epoch 8/200
48000/48000 [=====] - 3s 54us/step - loss: 0.0157 - acc: 0.9954 -
val_loss: 0.0886 - val_acc: 0.9781
Epoch 9/200
48000/48000 [=====] - 2s 51us/step - loss: 0.0144 - acc: 0.9953 -
val_loss: 0.1042 - val_acc: 0.9756
Epoch 10/200
48000/48000 [=====] - 2s 51us/step - loss: 0.0135 - acc: 0.9956 -
val_loss: 0.1103 - val_acc: 0.9750
Epoch 11/200
48000/48000 [=====] - 2s 51us/step - loss: 0.0144 - acc: 0.9951 -
val_loss: 0.0990 - val_acc: 0.9762
Epoch 12/200
48000/48000 [=====] - 2s 51us/step - loss: 0.0078 - acc: 0.9978 -
val_loss: 0.1161 - val_acc: 0.9732
Epoch 13/200
48000/48000 [=====] - 3s 53us/step - loss: 0.0126 - acc: 0.9958 -
val_loss: 0.1155 - val_acc: 0.9754
Epoch 00013: early stopping
```

Burada 8'inci epoch'tan itibaren değerler bu epoch'taki değerleri 5 kez üst üste geçmemiştir. İşte restore_best_weights parametresi True geçildiği için artık model son halinin ağırlıklarıyla değil en iyi halinin ağırlıklarıyla yüklenecektir.

ModelCheckPoint callback sınıfı modeli belli bir noktada save etmek için kullanılmaktadır. Sınıfın __init__ metodu şöyledir:

```
ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=False,
save_weights_only=False, mode='auto', period=1)
```

Metodun birinci parametresi save edilecek dosyanın yol ifadesini belirtir. Bu yol ifadesi formatlı da girilebilmektedir. monitor parametresi yine izlenecek değeri belirtir. save_weights_only parametresi False girilirse (default durum) tüm model save edilir. True girilirse yalnızca model ağırlık değerleri save edilir. Eğer save_best_only parametresi True

girilirse son model değil en iyi model save edilmektedir. period parametresi (default değeri 1) kaç epoch aralıklarla kontrollerin yapılacağını belirtir.

Dosya ismi olarak joker karakterleri girebiliriz. Örneğin:

```
mcp = ModelCheckpoint('model.{epoch:02d}-{val_loss:.2f}.hdf5', save_best_only=True, verbose=1)
```

Eğer dosya isminde yukarıdaki gibi yer tutucular kullanılırsa bu durumda her iyileşmede ayrı bir dosya haline model save edilir. Eğer dosya isminde yer tutucular kullanılmazsa bu durumda save işlemi tek bir dosyaya yapılmaktadır. O da save_best_only parametresi True ise en iyisi değilse son iyileştirilmiş olan değerdir.

Overfit Durumu ve Regülasyon

Anımsanacağı gibi "overfit durumu" bir sinir ağı modelinin arzu edilenin dışında başka şeyleri öğrenmesinden kaynaklanan sahte bir iyilik halini betimlemektedir. Yani model overfit durumunda sinir ağı bir şey öğrenmiştir ama öğrendiği şey tam olarak bizim istediğimiz şey değildir. Overfit işleminin pek çok nedeni vardır. Bunların en başta gelenlerinden birisi "epoch kaynaklı overfit durumu"dur. Epoch kaynaklı overfit durumunun nedeni modeli eğitirken fazla sayıda epoch kullanılmıştır. Anımsanacağı örneklerimizde epoch kaynaklı overfit durumunu modelin kendi metrik değerleri ile sınama (validation) metrik değerlerinin karşılaştırılması ile tespit etmişik. Overfit durumunun tersine "underfit" denilmektedir. Underfit modelin yeterli bir öğrenme gerçekleştirememesi durumudur. Yine modelin öğrenmedeki başarısızlığı metrik değerlere bakılarak tespit edilebilmektedir.

Overfit olusunun temel kaynakları şunlardır:

- Veri kümelerinin yetersizliği
- Fazla epoch işleminin uygulanması
- Modelin katman sayısının ve katmanlardaki nöron sayılarının fazlalığı (buna modelin kapasitesi (capacity) denilmektedir.
- Model ağırlıklarındaki entropy fazlalığı

Biz bugüne kadar overfit durumunu yalnızca eğitilmiş modeldeki epoch grafiğine bakarak tespit ettik. Ve bunun çözümü olarak da overfit'in başladığı epoch değerinde epoch işlemini sınırlamaya çalıştık.

TensorFlow Kütüphanesinin Kullanımı

TensorFlow Google tarafından 2015 yılında yapay zeka ve makine öğrenmesi konuları için tasarlanmış bir kütüphanedir. Daha önceden de belirttiğimiz gibi TensorFlow aşağı seviyeli bir kütüphane olduğu için bunun üzerine oturulmuş olan daha yüksek seviyeli kütüphaneler de vardır. Örneğin şimdiden kadar kullandığımız Keras aslında arka planda TensorFlow kütüphanesini kullanmaktadır. (Aslında Keras backend olarak Tensorflow dışında Microsoft'un CNTK backend'ini, Theano backend'ini de kullanabilmektedir.) Kütüphaneye ismini veren Tensor matematik ve makine öğrenmesinde matrisel bir kavramı temsil etmektedir. İsimdeki Flow sözcüğü işlemlerin akışsal bir biçimde yürütüldüğünü anlatmaktadır. TensorFlow yalnızca Python'dan değil pek çok programlama dilinden de kullanılabilmektedir. Zaten kütüphanenin önemli bir bölümü C++ Programlama Dilinde yazılmıştır.

TensorFlow kütüphanesinin Python için ana API dokümantasyonu aşağıdaki adreste bulunmaktadır:

https://www.tensorflow.org/api_docs/python

TensorFlow kütüphanesini Python'da kullanırken geleneksel olarak import işlemi şöyle yapılmaktadır:

```
import tensorflow as tf
```

TensorFlow kütüphanesi 2'li versiyonlarla birlikte bazı pratik öğelere sahip olmuştur. Ayrıca Keras kütüphanesi de TensorFlow 2.0 ile TensorFlow'un bünyesine katılmıştır. Kursun yapıldığı zaman diliminde ağırlıklı olarak TensorFlow'un 1'li versiyonları kullanılmaktadır. Biz de kursumuzda önce TensorFlow'un 1'li versiyonlarına dayalı bir anlatım uygulayacağız. Daha sonra 2'li versiyonlarındaki yeniliklerden ("eager execution" gibi) bahsedeceğiz. Eğer

makinenizde TensorFlow'un 2'li versiyonları yüklüyse bu durum aşağıdaki 1'li versyonlara ilişkin kodların çalıştırılamamasına yol açabilmektedir. Çünkü maalesef 2'li versiyonlarla 1'li versionlar arasındaki geriye doğru uyum bazı tasarım değişikliklerinden dolayı bozulmuştur. TensorFlow ekibi bu uyumsuzluğu pratik bir biçimde gidermek için ayrı bir paket de düzenlemiştir. Sonuç olarak eğer makinenizde TensorFlow'un 2'li versiyonları yüklü ise geriye doğru uyumu koruyabilmek için aşağıdaki bildirimleri yapmalısınız:

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

Aşağıdaki dokümanda TensorFlow'un 1'li versiyonları ile 2'li versiyonları arasındaki farklılıklar resmi olarak dokümante edilmiştir:

<https://www.tensorflow.org/guide/migrate>

TensorFlow Kütüphanesinin 1'li Versiyonlarının Kullanımı

Bugüne kadar TensorFlow'da yazılmış kodların büyük bölümünde kütüphanenin 1'li versiyonları kullanılmıştır. Bu bölümde biz TensorFlow'un 1'li versiyonlarının temel özelliklerini göreceğiz.

Tensor Nesnelerinin Oluşturulması

TensorFlow kütüphanesindeki en önemli kavram "tensor" kavramıdır. Bir tensor yaratmanın çeşitli yolları vardır. Bunun en basit yollarından biri convert_to_tensor fonksiyonun kullanılmasıdır. convert_to_tensor fonksiyonuna biz argüman olarak dolaşılabilir (iterable) bir nesne verebiliriz. Örneğin bu dolaşılabilir bir liste ya da bir ndarray nesnesi olabilir:

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
b = np.array([10, 20, 30], dtype='float32')

t1 = tf.convert_to_tensor(a, dtype='float32')
t2 = tf.convert_to_tensor(b)
```

convert_to_tensor fonksiyonunun parametrik yapısı şöyledir:

```
tf.convert_to_tensor(value, dtype=None, name=None, preferred_dtype=None, dtype_hint=None)
```

convert_to_tensor fonksiyonuyla oluşturduğumuz nesneler Tensor isimli bir sınıf türündendir:

```
In [4]: t1
Out[4]: <tf.Tensor 'Const:0' shape=(2, 3) dtype=float32>

In [5]: t2
Out[5]: <tf.Tensor 'Const_1:0' shape=(2, 3) dtype=float64>
```

Tensor nesnelerinin de dtype türlerinin olduğuna dikkat ediniz. Bir tensor oluşturmanın diğer bir yolu da constant fonksiyonunu kullanmaktır. Örneğin:

```
c1 = tf.constant([1, 2, 3], dtype='float32')
```

constant fonksiyonuyla oluşturduğumuz nesnelerin de Tensor sınıfı türünden olduğuna dikkat ediniz:

```
In [2]: c1
Out[2]: <tf.Tensor 'Const:0' shape=(3,) dtype=float32>
```

Bu durumda aslında convert_to_tensor ile constant fonksiyonlarının işlevleri benzerdir. constant fonksiyonunun parametrik yapısı şöyledir:

```
tf.constant(value, dtype=None, shape=None, name='Const', verify_shape=False)
```

convert_to_tensor ve constant fonksiyonlarındaki name parametresi dikkatinizi çekmiştir. Bu parametreler yoluyla Tensor nesnelerine isimler de verebiliriz. Örneğin:

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

t1 = tf.convert_to_tensor([10, 20, 30], dtype=tf.float32, name='Tensor-1')
print(t1)

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

t2 = tf.convert_to_tensor([50, 60, 70], dtype=tf.float32, name='Tensor-2')
print(t1)
```

Program çalıştırıldığında şöyle bir görüntü elde edilecektir:

```
Tensor("Tensor-1:0", shape=(3,), dtype=float32)
Tensor("Tensor-2:0", shape=(3,), dtype=float32)
```

convert_to_tensor ve constant fonksiyonlarında değerler tekil ise bu değerler doğrudan da girilebilir. Örneğin:

```
t3 = tf.convert_to_tensor(100, dtype='float32')
t4 = tf.constant(200, dtype='float32')
```

Göründüğü gibi TensorFlow'da tipki numpy kütüphanesinde olduğu gibi nesnelerin dtype tür bilgileri de vardır. Skaler bilgiler için Tensor nesnesinin shape bilgisinin boş bir demetten oluşturulduğunda dikkat ediniz:

```
Tensor("Const_8:0", shape=(), dtype=float32)
Tensor("Const_9:0", shape=(), dtype=float32)
```

TensorFlow'un dtype türleri numpy ile aynıdır. Tensorflow nesnelerinin dtype türleri yazışal olarak ya da isimsel olarak belirtilebilirler. Örneğin:

```
t5 = tf.constant([1, 2, 3], dtype=tf.float32)
```

Tabii eğer biz bir Tensor nesnesini bir ndarray nesnesinden hareketle oluşturacak oluşturduğumuzda ayrıca bu fonksiyonlarda dtype türünü belirtmemize gerek kalmamaktadır.

TensorFlow'da Değişkenler

TensorFlow'da Tensor nesnelerinin dışında değişkenler de oluşturabiliriz. Bu değişkenler -ileride göreceğimiz gibi- aslında model akışında temsil edilmek üzere oluşturulurlar. Değişkenler Variable isimli bir sınıfta temsil edilmektedir. Dolayısıyla Variable fonksiyonuyla yaratılırlar. Örneğin:

```
x = tf.Variable([10, 20, 30], dtype='float32')
y = tf.Variable(np.array([40, 50, 60]), dtype='float32')
```

TensorFlow'da değişkenler tensör tutarlar. Ancak tutukları tensör değiştirilebilmektedir. Halbuki biz convert_t_tensor ya da constant fonksiyonlarıyla bir tensör oluşturduğumuzda bu adeta bir sabit gibi davranışmaktadır. Variable sınıfının __init__ metodunun parametrik yapısı şöyledir:

```
tf.Variable(
    initial_value=None,
```

```

trainable=None,
collections=None,
validate_shape=True,
caching_device=None,
name=None,
variable_def=None,
dtype=None,
expected_shape=None,
import_scope=None,
constraint=None,
use_resource=None,
synchronization=tf.VariableSynchronization.AUTO,
aggregation=tf.VariableAggregation.NONE,
shape=None
)

```

Bir Variable nesnesi oluşturulduğunda değer henüz ona atanmaz. Değerin atanması için tf.global_variables_initializer fonksiyonun çağrılması gereklidir. Bu fonksiyon bize bir initializer nesnesi vermektedir. Bu nesne bir session içerisinde run edilerek değer ataması yapılabilir. Session konusu izleyen bölümde ele alınacaktır. tf.global_variables_initializer fonksiyonu tüm değişkenlere değerlerini atar. Yani bu işlemin işin başında bir kez yapılması gerekmektedir. Bir Variable nesnesi yaratılırken yine dtype türü belirtilebilir. Değişkenlere kendimizde model içerisinde anlaşılsın diye name isimli parametresi ile isimler verebiliriz.

İfadelerin İşletilmesi ve Session Kavramı

TensorFlow bir çeşit üst-programlama (metaprogramming) kütüphanesidir. Dolayısıyla burada aslında programcı programla program yazmaktadır. TensorFlow ismindeki "Flow" akış anlamına gelir. Dolayısıyla bir programlama akışını belirtmektedir. Biz bu kütüphanede önce birtakım ifadeler oluştururuz. Sonra da bu ifadeleri çalıştırırız. İfadelerin oluşturulması ile çalıştırılması farklı anımlardadır. Örneğin biz yukarıda bir değişkeni şöyle oluşturmuştuk:

```
x = tf.Variable([10, 20, 30], dtype='float32')
```

Burada oluşturduğumuz değişkene atanacak olan değer ([10, 20, 30] değeri) henüz bu değişkene atanmamıştır. Biz TensorFlow'da önce akış kurgusunu oluşturup sonra bunu çalıştırırız.

Bir TensorFlow ifadesinin işletilmesi için bir Session nesnesine gereksinim duyulmaktadır. Session nesnesi Session sınıfının `__init__` metoduyla oluşturulabilir. Session sınıfının `__init__` metodunun parametrik yapısı şöyledir:

```
Session(target='', graph=None, config=None)
```

Bir Session akışın işletilmesi için gereken konfigürasyonu temsil etmektedir. Session nesnesi yaratılıp kullanıldıktan sonra close fonksiyonuyla kapatılmalıdır. Session sınıfı "Bağlam Yönetim Protokolünü (Context Management Protocol)" desteklemektedir. Bu nedenle `with` deyimi ile kullanılabilir. Tabii eğer biz Session nesnesini `with` deyimi ile kullanacağsa close işlemini artık biz yapmamalıyız. Çünkü bu durumda close işlemi çağrılmış olacak olan `__exit__` metodu tarafından yapılacaktır. Örneğin bir Session nesnesini şöyle yaratabiliriz:

```
session = tf.Session()
...
session.close()
```

Ya da aynı işlemi `with` deyimi ile şöyle de yapabiliyoruz:

```
with tf.Session() as session:
    ...
```

Bir ifadeyi session yoluyla işletmek için Session sınıfının `run` metodu kullanılmaktadır. `run` metodu bizden ilgili ifade nesnesini alır ve onu çalıştırır. Örneğin biz bir Tensor nesnesinin içerisindeki değeri almak isteyelim. Bunun için Session nesnesi yaratıp `run` işlemi yapmamız gereklidir:

```
t = tf.convert_to_tensor([10, 20, 30], dtype=tf.float32)
with tf.Session() as session:
    print(session.run(t))
```

run metodu ile bir tensor değerini aldığımızda run metodu bize değeri ndarray nesnesi biçiminde verir. run metoduna birden fazla ifade de girilebilir. Bunun için ifadelerin dolaşılabilir (iterable) bir nesne biçiminde verilmesi gerekmektedir. Örneğin:

```
with tf.Session() as session:
    session.run(init)
    val = session.run([t1, t2, t3, t4])
```

run metodunun parametrik yapısı şöyledir:

```
run(fetches, feed_dict=None, options=None, run_metadata=None)
```

Şimdi de bir Variable nesnesi içerisindeki değeri yazdırımıya çalışalım. Variable nesneleri yaratılırken onlar için belli bir ilkdeğer belirtilmektedir. Ancak bu ilkdeğer değişkene doğrudan atanmaz. İşte tüm Variable nesneleri için belirtilen ilkdeğerler global_variables_initializer isimli bir fonksiyon tarafından atanmaktadır. Böylece tüm Variable nesnelerine yapılan atamaların etki göstermesi için bizim global_variables_initializer fonksiyonun geri döndürdüğü nesneyi de çalıştırımız gereklidir. Örneğin:

```
x = tf.Variable([10, 20, 30], dtype=tf.float32)
y = tf.Variable([10, 20, 30], dtype=tf.float32)
init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    print(session.run(x))
    print(session.run(y))
```

Bir değişkene başka bir değer yerleştirilebilir. Ama bir tensör nesnesine başka bir değer yerleştirilemez. (Tensor nesneleri bir çeşit sabit kavramı oluşturmaktadır.) Bu yerleştirme işlemi için assign fonksiyonu kullanılmaktadır. Ancak assign fonksiyonu asıl atamayı yapmamaktadır. Bu atamayı anlatan bir sınıf nesnesi geri döndürmektedir. Bizim assign fonksiyonunun geri döndürdüğü nesne için run işlemi yapmamız gereklidir. Başka bir deyişle assign atama işleminin kendisini yapmaz atama yapılacak bilgisini içeren bir nesne verir. Biz o nesneyi run ile çalıştırıldığımızda atamayı yapmış oluruz.

Örneğin:

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

x = tf.Variable([10, 20, 30], dtype='float32')

init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    result = session.run(x)

ass_x = tf.assign(x, [100, 200, 300])

with tf.Session() as session:
    session.run(ass_x)
    result = session.run(x)
    print(result)
```

Tabii biz bu işlemleri bir döngü içerisinde de yapabiliyoruz. Örneğin:

```

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

x = tf.Variable(0, dtype='float32')

init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)

    for i in range(100):
        ass_x = tf.assign(x, i)
        session.run(ass_x)
        result = session.run(x)
        print(result)

```

Değişkenler ve Sabitler Üzerinde eval İşlemleri

Normal olarak değişkenler ve sabitler Session sınıfının run metoduyla çalıştırılmaktadır. Ancak aynı zamanda bu sınıfların eval isimli örnek metotları da vardır. Yani çalışma run metodu yerine eval metodu ile de yapılabilmektedir. eval metodlarında Session nesnesi belirtilmeyebilir. Bu durumda bu metodlar default bir Session kullanmaktadır. Pekiyi default session nasıl oluşturulmaktadır? İşte Session sınıfı with deyiminde kullanıldığından "bağlam yönetim protokolü" gereğince çalıştırılan __enter__ metodu yeni yapılan session'in default session olmasını sağlamaktadır. Yani biz with deyimi içerisinde run yerine eval metodlarını da kullanabiliriz. Örneğin:

```

x = tf.Variable(10.0)
init = tf.global_variables_initializer()

with tf.Session() as session:
    session.run(init)
    result = x.eval()
    print(result)

```

Örneğin:

```

x = tf.Variable(10.0)
y = tf.Variable(20.0)
z = x * y

init = tf.global_variables_initializer()

with tf.Session() as session:
    session.run(init)
    result = z.eval()
    print(result)

```

Tensörler Üzerinde İşlemler

Biz Tensör ifadelerini operatörlerle işlemlere sokabiliriz. Bu işlemler için çeşitli global fonksiyonlar kullanılmaktadır. Örneğin:

```

add
subtract
multiply
divide
mod
pow

```

gibi. Bu fonksiyonlar işlemin kendisini yapmazlar. İşlemi anlatan bir düğüm nesnesi verirler. Bizim düğüm nesnesini run etmemiz gereklidir. Örneğin:

```

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

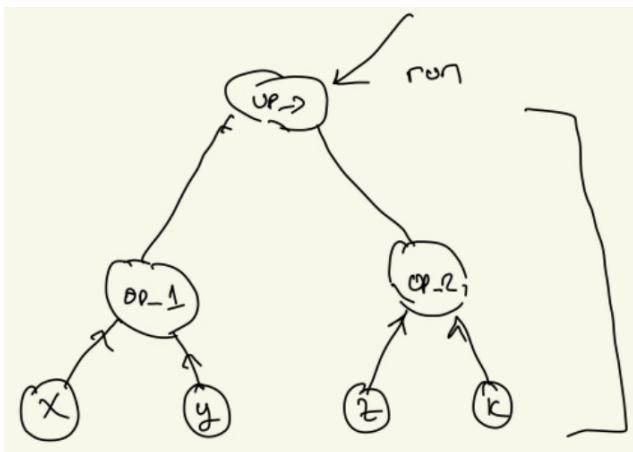
x = tf.Variable(1, dtype='float32')
y = tf.Variable(2, dtype='float32')
z = tf.Variable(3, dtype='float32')
k = tf.Variable(4, dtype='float32')

op_1 = tf.multiply(x, y)
op_2 = tf.multiply(z, k)
op_3 = tf.add(op_1, op_2)

init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    result = session.run(op_3)
    print(result)

```

Bu örnekte yalnızca `op_3`'ün çalıştırıldığına dikkat ediniz. TensorFlow'da bir ifade diğer ifadelere bağlı ise o ifadeler de zaten çalıştırılmaktadır. TensorFlow aritmetik işlemler için bir graf oluşturmaktadır. Sonra bu grafi kullanarak ilgili düğümün çalıştırılması için gerekli olan işlemleri yapmaktadır. Yukarıdaki işlemlerin graf görüntüsü şöyledir:



Aslında biz aritmetik işlemleri fonksiyonlarla değil doğrudan operatörlerle de yapabiliyoruz. Çünkü TensorFlow kütüphanesinin ilgili sınıfları zaten bu operatör işlemlerini yapan operatör metodlarına sahiptir. Bu nedenle biz Tensorflow operator fonksiyonlarını çağırmak yerine işlemi doğrudan da yapabiliyoruz. Örneğin:

```

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

x = tf.Variable(1, dtype='float32')
y = tf.Variable(2, dtype='float32')
z = tf.Variable(3, dtype='float32')
k = tf.Variable(4, dtype='float32')

op_1 = x * y
op_2 = z * k
op_3 = op_1 + op_2

init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    result = session.run(op_3)
    print(result)

```

Python'da C++ gibi bazı dillerde olduğu gibi bir atama operatör metodu yoktur. Dolayısıyla bir TensorFlow değişkenine aşağıdaki gibi bir atama yapamayız:

```
x = tf.Variable(1, dtype='float32')  
x = 100
```

Çünkü buradaki `x = 100` işlemi aslında `x` değişkenine Python'ın int türünden 100 değerini atama işlemidir. Dolayısıyla bu atamadan sonra `x`'in türü int olacaktır. TensorFlow değişkenlerine atama işlemleri her zaman `assign` fonksiyonuyla yapılmalıdır.

Burada yeniden bir noktayı vurgulamak istiyoruz: TensorFlow yazılım mimarisinde oluşturulan nesneler aslında işlemin "sonucunu değil ne yapılacakı bilgisini" tutmaktadır. Dolayısıyla örneğin:

```
z = tf.add(x, y)
```

çağrısından elde edilen `z` nesnesi bu toplama işleminin sonucunu tutmaz. `x` ile `y`'nin toplanacağı bilgisini tutat. Bizim bu toplamı yapabileceğimiz için `z` nesnesini `run` etmemiz gereklidir.

TensorFlow kütüphanesinde operatör fonksiyonlarının yanı sıra pek çok genel amaçlı fonksiyon da bulunmaktadır. Bunlardan bazılarının isimlerini bir örnek oluşturmak amacıyla vermek istiyoruz:

```
sin  
cos  
asin  
acos  
abs  
fill  
sqrt  
matmul
```

Tensorflow'da yukarıdakilerin dışında onlarca genel amaçlı fonksiyonlar bulunmaktadır. Bu fonksiyonları TensorFlow dokümanlarından inceleyebilirsiniz.

TensorFlow'da Yer Tutucular (Place Holders)

TensorFlow'da yer tutucular değişkenlere benzemekle birlikte aslında değişkenlerden farklı amaçlarla kullanılan sınıflardır. Anımsanacağı gibi değişkenler ilkdeğer verilerek yaratılmak zorundadır. Halbuki yer tutucular ilkdeğer verilmeden yaratılmaktadır. Değişkenler ismi üzerinde değerlerini eğitim sırasında değiştirebilirler. Halbuki yer tutucular eğitim sırasında değerlerini değiştirmezler. Bu nedenle yer tutucular tipik olarak girdi değerlerin temsil edilmesinde kullanılırlar. Bir yer tutucu placeholder isimli fonksiyonla yaratılır. Fonksiyonun parametrik yapısı şöyledir:

```
tf.placeholder(dtype, shape=None, name=None)
```

Örneğin:

```
pl = tf.placeholder(tf.float32, shape=(2, 2))
```

Burada bir yer tutucu nesnesi yaratılmıştır. Bu nesnenin boyutları $(2, 2)$ 'dir.

Yer tutucular ifadeler içerisinde kullanılabilirler. Yani örneğin biz yer tutucularla değişkenleri ve sabitleri işleme sokabiliriz. Bu durumda nihai ifade oluşturulduktan sonra bu ifade Session sınıfının `run` metodıyla çalıştırıldığında yer tutuculara değerlerinin verilmesi gereklidir. Örneğin bir ifadede `pl1, pl2, pl3, pl4` ve `pl5` isminden 5 yer tutucu kullanılmış olsun. Şimdi biz bu ifadeyi çalıştırıldığında `run` metodunda bu 5 yer tutucu için de değerlerini vermemiz gereklidir. Değer verme işlemi `run` metodunun `feed_dict` isimli parametresiyle yapılmaktadır. Bu parametreye bir sözlük nesnesi girilir. Girilen sözlüğün anahtarları yer tutucu isimleri değerleri ise onlara verilecek değerler olmalıdır. Örneğin:

```

pl = tf.placeholder(tf.float32, shape=(2, 2))
x = tf.Variable(10.0)
y = x + pl
init = tf.global_variables_initializer()

with tf.Session() as session:
    session.run(init)
    result = session.run(y, feed_dict={pl: [[1.0, 2.0], [3.0, 4.0]]})
    print(result)

```

Bu örnekte çalıştırılacak ifade y'dir. y ifadesi bir değişkene ve bir yer tutucuya bağlıdır. Değişkenlerin değerleri zaten baştan verilmek zorundadır. O halde bu ifadenin çalıştırılabilmesi için pl isimli yer tutucunun değerinin feed_dict parametresinde belirtilmesi gereklidir. Çalıştırmanın şöyle yapıldığına dikkat ediniz:

```
result = session.run(y, feed_dict={pl: [[1.0, 2.0], [3.0, 4.0]]})
```

Örneğin:

```

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

x = tf.Variable([1, 2, 3, 4, 5], dtype='float32')
ph1 = tf.placeholder('float32', (5,))
ph2 = tf.placeholder('float32', (5,))

y = x + ph1 * ph2

init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    result = session.run(y, feed_dict={ph1: [10, 20, 30, 40, 50], ph2: [2, 2, 2, 2, 2]})
    print(result)

```

Örneğin:

```

m = tf.placeholder(tf.float32)
n = tf.placeholder(tf.float32)
x = tf.Variable(10.0)
y = m * x + n

init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    result = session.run(y, feed_dict={m: 1.0, n: 0.0})
    print(result)

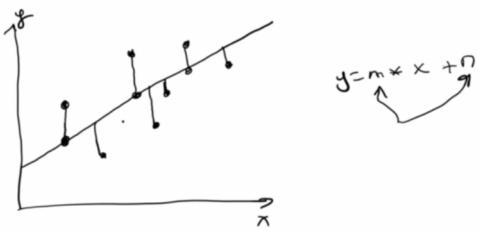
```

Burada $y = m * x + n$ ifadesinde bir değişken iki yer tutucu vardır. Bu ifade çalıştırılırken m ve n isimli yer tutuculara değerleri feed_dict parametresiyle yerleştirilmiştir.

Yer tutucular modeldeki eğitilen parametreleri değil girdi parametrelerini belirlemek için kullanılmaktadır. Bu nedenle bir yapay sinir ağı TensorFlow ile organize edilirken girdi katmanı yer tutucularından oluşur.

TensorFlow'da yer tutucuların kullanımı için bir doğrusal regresyon problemini inceleyebiliriz. Doğrusal regresyon probleminde elimizde bir grup (x, y) değerleri vardır. Bu (x, y) değerlerini temsil eden en iyi doğrunun $y = mx + n$ biçiminde oluşturulması istenir. Böylece doğrusal modeller üzerinde kestirimler yapılabilmektedir.

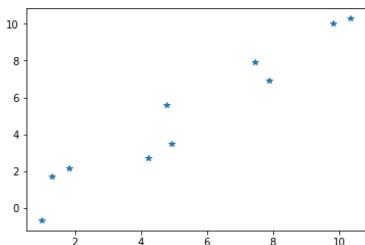
Doğrusal regresyonun genel mantığı verilen noktaların çizilecek doğruya uzaklığının minimize edilmesidir. Bu işlemler iteratif yolla yapılabileceği gibi TensorFlow'daki optimizasyon sınıflarıyla da yapılabilmektedir.



Burada minimize edilmeye çalışacak olan değer şekilde görülen çubukların uzunlıklarının toplamıdır. Elimizde bulunan noktalar bir yer tutucu biçiminde modelde belirtilir. Sonra minimize edilecek ifade oluşturulur ve TensorFlow'un optimizer sınıflarına bu değer verilir. Buradan sonuç olarak m ve n değerleri elde edilecektir. m doğrunun eğimini n ise y eksnini kesim noktasıdır. Bunun önce $y = x$ doğrusunda biraz bozulmuş değerler elde edelim:

```
x_data = np.linspace(0, 10, 10) + np.random.uniform(-1.5, 1.5, 10)
y_data = np.linspace(0, 10, 10) + np.random.uniform(-1.5, 1.5, 10)

plt.plot(x_data, y_data, '*')
```



Şimdi işlemleri TensorFlow'da sırasıyla yapalım:

```
x_data = tf.placeholder(dtype=tf.float32, shape=(1, 10))
y_data = tf.placeholder(dtype=tf.float32, shape=(1, 10))

r = np.random.random(2)
m = tf.Variable(r[0], dtype=tf.float32)
n = tf.Variable(r[1], dtype=tf.float32)

error = 0
y_hat = m * x_data + n
error += (y_data - y_hat) ** 2

optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
train = optimizer.minimize(error)

init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)

    x = np.linspace(0, 10, 10) + np.random.uniform(-1.5, 1.5, 10)
    y = np.linspace(0, 10, 10) + np.random.uniform(-1.5, 1.5, 10)

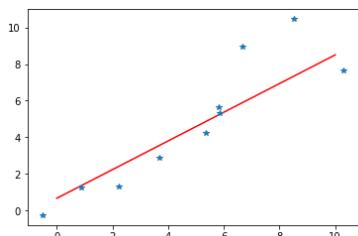
    session.run(train, feed_dict={x_data: x.reshape((1, 10)), y_data: y.reshape(1, 10)})
    slope, intercept = session.run([m, n])

x_guess = np.linspace(0, 10, 10)
y_guess = slope * x_guess + intercept

plt.plot(x_guess, y_guess, color='r')
plt.plot(x, y, '*')
```

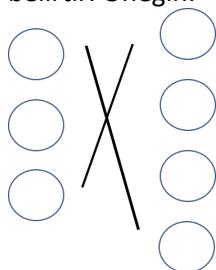
Burada x_data ve y_data isimli doğrusal regresyonu bulunacak olan iki veri kümesi yer tutucu olarak tanımlanmıştır. Sonra doğrusal regresyondaki çubuk boyutları m ve n ile temsil edilen eğim ve eksen kesim noktaları kullanılarak hesaplanmış ve TensorFlow'un GradientDescentOptimizer nesnesine verilmiştir. Bundan sonra kurulan model x_data ve y_data değerleri `feed_dict` ile verilerek elde edilmiştir. Optimizer nesnesi buradaki error değerini minimize etmek

icin m ve n değerlerini oluşturur. Program içerisinde biz bu değerleri alarak tahmin edilen doğruya çizdirmi̇ş olduk. Örnek bir çıktı aşağıdaki gibi olacaktır:



Yapay Sinir Ağı Modelinin TensorFlow'da Oluşturulması Temsili

TensorFlow ile yapay sinir ağı oluşturabilmek için sinir ağını matrisel bir biçimde ifade etmek gereklidir. Çünkü TensorFlow Keras gibi bu işlemleri kendi içerisinde otomatik biçimde yapmamaktadır. Yapay sinir ağındaki soldaki katman ile sağdaki katmanın ilişkisi bir matris biçiminde şöyle temsil edilebilir: XW . Burada X soldaki katmanın (örneğin işin başında girdi katmanının) çıkış değerlerini W ise sağdaki katman nöronlarındaki ağırlık değerlerini belirtir. Örneğin:



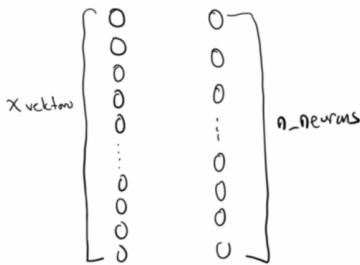
$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix} = \begin{bmatrix} x_1w_{11} + x_2w_{21} + x_3w_{31} \\ x_1w_{12} + x_2w_{22} + x_3w_{32} \\ x_1w_{13} + x_2w_{23} + x_3w_{33} \\ x_1w_{14} + x_2w_{24} + x_3w_{34} \end{bmatrix}^T$$

Burada w_{ij} soldaki katmanın i 'inci nöronunun sağdaki katmanın j 'inci nöronu ile oluşturduğu katsayı değeridir. Tabii bu XW matrisine ayrıca bir de bias değeri eklenir. Anımsanacağı gibi bias değeri nörona özgüdür. Böylece nöronda aktivasyon fonksiyonuna sokulmadan önce biriken toplam değerler $XW + b$ olur. Bu değer bir aktivasyon fonksiyonuna sokulup nöronun çıktı değeri elde edilir. Sonra bu çıktı değerleri bir sonraki katmana girdi olarak sokulacaktır. Çıktı katmanından elde edilen değerlerin ne kadar iyi olduğunu belirlemek için çıktı değerinin bir loss fonksiyonuna sokulması gerekecektir. Tabii loss bizim loss fonksiyonundan elde edilen değeri minime etmemiz gereklidir ki bu da TensorFlow'un yukarıdaki regresyon örneğinde gördüğümüz optimizer sınıflarıyla yapılabilir.

Şimdi TensorFlow kullanarak bir dense katmanı oluşturan bir fonksiyon yazalım. Fonksiyonumuz parametrik yapısı şöyle olsun:

```
dense_layer(x, n_neurons, activation = None, name = None)
```

Burada x soldaki katmanın çıkış değerlerini belirten (k, n) boyutundaki bir Variable ya da PlaceHolder nesnesidir. Örneğin birinci hidden katman için bu x değeri girdi nöronlarından oluşmaktadır. $n_neurons$ sağdaki katmandaki nöron sayısı belirtirmektedir.



dense_layer fonksiyonun örnek bir yazımı şöyle olabilir:

```
def dense_layer(x, n_neurons, activation=None):
    n_x = int(x.get_shape()[1])
    stddev = 2 / np.sqrt(n_x)
    init_vals = tf.truncated_normal((n_x, n_neurons), stddev=stddev)
    W = tf.Variable(init_vals, name='weights')
    b = tf.Variable(tf.zeros((n_neurons,)), name='biases', dtype=tf.float32)
    z = tf.matmul(x, W) + b
    d = {'relu': tf.nn.relu, 'softmax': tf.nn.softmax, 'sigmoid': tf.nn.sigmoid}
    func = d.get(activation)
    if func:
        return func(z)

    return z
```

Burada önce W matrisi için rastgele normal dağılım değerleri oluşturulmuştur. tf.truncated_normal fonksiyonu üç değerleri (soldan ve sağdan iki standart sapma) atarak bize rastgele normal dağılmış değerler vermektedir. (Aslında Keras da anımsanacağı gibi katmalardaki W matrislerinin ilk değerlerini bu biçimde oluşturmaktadır. Biz Keras ile çalışırken ilgili fonksiyon parametrelerinde bunun için özel bir değer girmemişti.) Fonksiyonda daha sonra W matrisi bir Tensorflow Variable biçiminde oluşturulmuştur. Bias değerleri için de bir Variable oluşturulduğunu görüyoruz. Her ne kadar örneğimizde bu değerler 0 biçiminde alınmışsa da kodda değiştirilebilecek biçimde bulunmasında fayda olacaktır. Nihayet tf.matmul fonksiyonıyla matrisel çarpım uygulanmış ve elde edilen değerler transfer fonksiyonuna sokulmuştur. Yukarıdaki fonksiyonumuzun geri dönüş değeri sağıdaki katmanın çıkış nöron değerleridir. Örnek fonksiyonumuzu şöyle çalıştırabiliriz:

```
x = tf.placeholder(dtype=tf.float32, shape=(1, 5))
result = dense_layer(x, 3, 'sigmoid')
a = np.array([[1, 2, 3, 4, 5]], dtype=np.float32)

init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    print(session.run(result, feed_dict={x: a}))
```

Elde edilen örnek bir çıktı şöyledir:

```
[[0.02509242 0.9290879 0.38102645]]
```

Tabii bu değerler rasgele w değerlerinin sonucunda elde edilmiş eğitilmemiş değerlerdir. Aslında yukarıda yazmış olduğumuz dense_layer isimli fonksiyonun daha gelişmiş zaten hazır olarak TensorFlow'da bulunmaktadır.

```
tf.layers.dense(
    inputs,
    units,
    activation=None,
    use_bias=True,
    kernel_initializer=None,
    bias_initializer=tf.zeros_initializer(),
    kernel_regularizer=None,
    bias_regularizer=None,
```

```

activity_regularizer=None,
kernel_constraint=None,
bias_constraint=None,
trainable=True,
name=None,
reuse=None
)

```

Yani aslında bizim böyle bir fonksiyon yazmamıza gerek yoktu. Şimdi aynı yer tutucuyu kullanarak orijinal tf.layers.dense fonksiyonuyla aynı şeyi yapalım:

```

x = tf.placeholder(dtype=tf.float32, shape=(1, 5))
result = tf.layers.dense(x, 3, 'sigmoid')
a = np.array([[1, 2, 3, 4, 5]], dtype=np.float32)

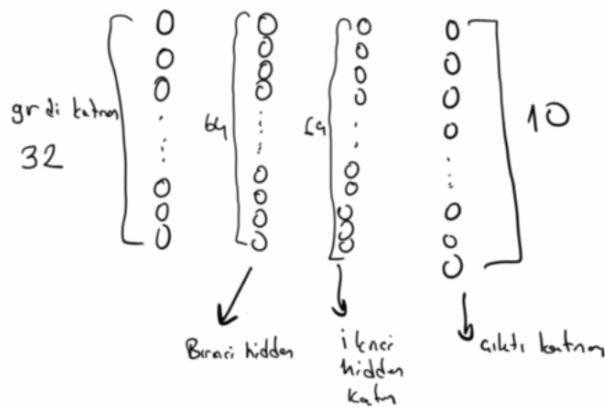
init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    print(session.run(result, feed_dict={x: a}))

```

Çıkan örnek bir değer şöyledir:

```
[[0.08459509 0.06319192 0.00379056]]
```

Şimdi de birden fazla katmanın nasıl bir araya getirileceğine bakalım. Örneğin ağıımızın mimarisi aşağıdaki gibi olsun:



Bu katmanları dense_layer fonksiyonun (ya da tf.layers.dense fonksiyonun) çıktısını bir sonraki katmanın girdisi yaparak oluşturabiliriz:

```

x = tf.placeholder(dtype=tf.float32, shape=(1, 32))
hidden_1 = dense_layer(x, 64, 'relu')
hidden_2 = dense_layer(hidden_1, 64, 'relu')
output = dense_layer(hidden_2, 10, 'softmax')

a = np.array(np.random.rand(1, 32) * 100, dtype=np.float32)

init = tf.global_variables_initializer()
with tf.Session() as session:
    session.run(init)
    print(session.run(output, feed_dict={x: a}))

```

Örnek çıktı şöyle elde edilmiştir:

```
[[1. 0. 0. 1. 1. 0.99983764 1. 1. 1. ]]
```

Peki bu aşamadan sonra ağıın eğitimi nasıl gerçekleşecektir? Burada bundan sonra yapılacaklar şunlardır:

- Bir loss fonksiyonu belirlenmeli ve ağıın çıktısı bu loss fonksiyonuna sokulmalıdır. Bu loss fonksiyonundan bir değer elde edilmelidir.
- Bir optimizer nesnesi yaratılmalı ve bu nesneye minimize edilmek üzere loss fonksiyonun çıktısı verilmelidir.
- Sonra batch_size'lar belirlenerek asıl data batch biçiminde girilmelidir. Biz yukarıdaki örneklerde yer tutucu verilerini tek satırlı aldık. Aslında bu yer tutucular bir batch'lik çok satırlı veriler olacaktır.
- Aldığımız optimizer her batch sonrasında w değerlerini kendisi güncelleyecektir.

Biz bu noktada Tensorflow'da loss fonksiyonun ve optimizer'ların nasıl kullanılacağını somut bir örnekle açıklayacağız. Burada daha önce yapmış olduğumuz mnist örneği kullanacağız. Anımsanacağı gibi mnist elle yazılmış karakterlerin belirlenmesi için oluşturulmuş bir veri kümesiydi. Şimdi bu örneği Keras ile değil tamamne Tensorflow ile yapmaya çalışalım.

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

n_inputs = 28 * 28

x = tf.placeholder(tf.float32, shape=(None, n_inputs), name='x')
y = tf.placeholder(tf.int32, shape=(None,), name='y')

hidden_1 = tf.layers.dense(x, 128, 'relu', kernel_initializer='truncated_normal')
hidden_2 = tf.layers.dense(hidden_1, 128, 'relu', kernel_initializer='truncated_normal')
output = tf.layers.dense(hidden_2, 10, 'softmax', kernel_initializer='truncated_normal')
```

Burada output değerini potansiyel olarak oluşturduk. Burada output aslında girdi olarak bir batch büyüğünü alarak oluşturulacak biçimde kurgulanmıştır. O halde artık bu elde ettiğimiz output ifadesini loss fonksiyonuna sokabiliriz. İşte Tensorflow'da tf.nn modülü içerisinde pek çok loss fonksiyonu tanımlı olarak bulunmaktadır. Anımsanacağı gibi buradaki loss fonksiyonu binary_crossentropy olarak alınmıştı daha önce. Bunun TensorFlow karşılığı şöyledir:

```
xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y, logits=output)
loss = tf.reduce_mean(xentropy, name='loss')
```

Burada önemli olan bir nokta şudur: tf.nn.sparse_softmax_cross_entropy_with_logits fonksiyonundaki labels ile belirtilen parametrenin boyutunun logits ile belirtilen boyuttan bir düşük olması gerekmektedir. Bizim mnist örneğimizde çıktı (batch_size, 10) boyutunda "one hot encoing" biçimdedir. İşte gerçek değerlerin de (yani y değerlerinin de) bu boyutta olması beklense de loss fonksiyonu bizden bunu istememektedir. loss fonksiyonu bizden gerçek değerlerin "one hot encoding" uygulanmamış biçimini istemektedir.

Loss fonksiyonu da Tensorflow'da oluşturulduğuna göre artık sıra "optimizer" nesnesinin yaratılmasına gelmiştir. Optimizer gerçek anlamda w değerlerini loss fonksiyonun küçülmesi için değiştiren Tensorflow nesnesidir. O halde bizim bir optimizer nesnesi yaratıp bu nesneye bir biçimde loss fonksiyonun değerini vermemiz gereklidir. İşte Tensorflow içerisinde tf.train modülünde pek çok optimizer sınıfı vardır. mnist örneğinde optimizer olarak GradientDescent ya da Adam kullanılabilir. Biz örneğimizde Adam optimizasyonunu kullanmak isteyelim:

```
optimizer = tf.train.AdamOptimizer()
training_op = optimizer.minimize(loss)
```

Artık bizim nihai düğüm olarak training_op düğümünü çalıştırılmamız gereklidir. Bu düğüm hedef olarak run edildiğinde bütün işlemler yapılacak ve hidden_1, hidden_2 ile temsil edilen w değerleri güncellenecektir. Ancak biz modelimizi aynı anda bir batch_size'lık eğitim yapacak biçimde kurguladık. O halde bizim bu çalışma işlemini döngü içerisinde farklı batch'lerle yapmamız gereklidir. Tabii bunun için öce mnist verilerini okumalıyız.

mnist verileri TensorFlow'da nesil okunabilir? Biz mnist verilerini doğrudan numpy kullanarak dosyadan okuyabiliriz. Ya da anımsanacağı gibi keras içerisinde mnist verilerini bize veren hazır bir sınıf da vardı. İşte benzer biçimde mnist verilerinin TensorFlow içerisinde kullanılabilmesi de sağlanmıştır. Bu işlem kabaca şöyle yapılır:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('..', one_hot=True)
```

Buradan elde edilen mnist nesnesinin mnist.train.next_batch metodu bize her çağrıda sonraki batch kümесini vermektedir. mnist.train.num_examples toplam verilerin sayısını belirtmektedir. Bu durumda modelin eğitilmesi şöyle yapılabilir:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('..', one_hot=True)

n_epochs = 20

with tf.Session() as session:
    session.run(init)
    for k in range(n_epochs):
        print('Epochs {} begins...'.format(k + 1))
        for i in range(mnist.train.num_examples // batch_size):
            x_batch, y_batch = mnist.train.next_batch(batch_size)
            y_batch_not_encoding = np.argmax(y_batch, axis=1)
            session.run(training_op, feed_dict={x: x_batch, y: y_batch_not_encoding})
```

Pekiyi ağımızı eğittiğten sonra kestirim işlemini nasıl yapacağız? İşte bunun için yapılacak şey output düğümünün çalıştırılmasıdır. Örneğin:

```
with tf.Session() as session:
    session.run(init)
    result = session.run(output, feed_dict={x: gray})
    print(np.argmax(result))
```

TensorFlow Kütüphanesinin 2'li Versiyonlarının Kullanımı

TensorFlow'un 2'li versiyonları ile 1'li versiyonları arasındaki önemli farklılıklar şunlardır:

- TensorFlow 2'de metaprogramlama modeli büyük ölçüde terk edilmiştir. Graf ve session kavramları kaldırılmıştır. Artık TensorFlow 2'de değişkenler ve sabitler doğrudan değer tutmaktadır. Onların değerlerini elde etmek için run gibi bir işlem gerekmektedir. TensorFlow 2'de işlemler önce graf oluşturulup sonra çalıştırılmamakta diğer kütüphanelerde olduğu gibi hemen çalıştırılmaktadır. Bu yeni modele "eager execution" denilmektedir.
- TensorFlow 2'de pek çok fonksiyon ve metod parametrik yapı bakımından basitleştirilmiştir. Bu da daha kolay bir kullanım sunmaktadır.
- TensorFlow 2'de dokümantasyon iyileştirilmiştir.
- TensorFlow 2'de "function" isimi dekortatör yoluyla normal Python fonksiyonlarının da TensorFlow içerisinde kullanılmasına olanak sağlanmıştır.
- TensorFlow 2 ile birlikte Keras kütüphanesi daha iyileştirilerek TensorFlow içerisinde eklenmiştir.
- TensorFlow 2'de performans TensorFlow 1'e göre daha iyileştirilmiştir.

Örneğin:

```
import tensorflow as tf
```

```

a = tf.constant(10, dtype='float32')
b = tf.constant(20, dtype='float32')

c = tf.add(a, b)
print(c)

c = a + b
print(c)

```

Ekran çıktısı şöyledir:

```

tf.Tensor(30.0, shape=(), dtype=float32)
tf.Tensor(30.0, shape=(), dtype=float32)

```

Örneğin:

```

import tensorflow as tf

a = tf.Variable(10, dtype='float32')
b = tf.Variable(20, dtype='float32')

print(a)
print(b)

c = tf.add(a, b)
print(c)

c = a + b
print(c)

```

Ekran çıktısı şöyledir:

```

<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.0>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=20.0>
tf.Tensor(30.0, shape=(), dtype=float32)
tf.Tensor(30.0, shape=(), dtype=float32)

```

Bir değişkene yeni bir değer atamak için assign metodu kullanılır. Örneğin:

```

a = tf.Variable(10, dtype='float32')
print(a)

a.assign(20)
print(a)

```

Ekran çıktısı şöyledir:

```

<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=10.0>
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=20.0>

```

Ancak bir değişkene atanacak değerin değişkenin boyutuyla aynı olması gereklidir.

Bir sabitin ya da değişkenin değeri numpy metodu ile ndarray biçiminde elde edilebilir. Örneğin:

```

a = tf.Variable([10, 20, 30], dtype='float32')
b = a.numpy()
print(type(b))
print(b)

```

Bir Tensor'ün boyutunu shape özniteliği ile alıp reshape fonksiyonu ile değiştirebiliriz. Örneğin:

```
a = tf.Variable([[10, 20, 30], [40, 50, 60]], dtype='float32')
print(a.shape)
b = tf.reshape(a, (3, 2))
print(b)
```

reshape fonksiyonunun asıl tensörün boyutunu değiştirmedigine yeni bir Tensor oluşturduğuna dikkat ediniz. Benzer biçimde bir tensörün türü de dtype örnek özniteliği ile elde edilebilir. Örneğin:

```
a = tf.Variable([[10, 20, 30], [40, 50, 60]], dtype='float32')
print(a.dtype)
print(type(a.dtype))
```

rank isimli fonksiyon bir tensörün kaç boyutlu olduğu bilgisini tensör olarak vermektedir. Örneğin:

```
a = tf.Variable([[10, 20, 30], [40, 50, 60]], dtype='float32')
print(a.shape)
print(tf.rank(a).numpy())
print(len(a.shape))
```

Bir tensörün herhangi bir elemanı ya da elemanları indeksleme yoluyla ya da dilimleme yoluyla elde edilebilir. Elde edilen sonuç da bir tensördür. Örneğin:

```
a = tf.Variable([[10, 20, 30], [40, 50, 60]], dtype='float32')

b = a[0, 1]
print(b)

c = a[:, 1]
print(c)
```

Bir tensörün eleman sayısı size isimli fonksiyonla elde edilebilir. Örneğin:

```
a = tf.Variable([[10, 20, 30], [40, 50, 60]], dtype='float32')

print(tf.size(a))
print(tf.size(a).numpy())
```

TensorFlow 2'de yine tensörler işleme sokulduğunda numpy'da olduğu gibi vektörel işlemler gerçekleşmektedir. Örneğin:

```
a = tf.Variable([[10, 20, 30], [40, 50, 60]], dtype='float32')
b = tf.Variable([[1, 2, 3], [4, 5, 6]], dtype='float32')

c = tf.add(a, b)
print(c)

c = a * b
print(c)
```

Tek elemanlı bir tensör ya da python türü başka bir tensör ile işleme sokulursa -tipki numpy'da olduğu gibi- tek elemanlı tensör diğer tensörün her elemanı ile işleme sokulur (broadcasting). Örneğin:

```
a = tf.Variable([[10, 20, 30], [40, 50, 60]], dtype='float32')

b = 2 * a;
print(b)
```

transpose fonksiyonuyla matris transpose edilebilir, matmul fonksiyonuyla da matris çarpımı yapılabilir. Örneğin:

```

a = tf.Variable([[1, 3, 4]], dtype='float32')
b = tf.Variable([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype='float32')

c = tf.matmul(a, b)
print(c)

d = tf.transpose(b)
print(d)

```

Yukarıda da belirtildiği gibi Keras artık Tensorflow'un 2'li versiyonlarıyla birlikte TensorFlow içerisinde entegre edilmiştir. Böylece biz bağımsız Keras'ı kullanmak yerine TensorFlow içerisindeki Keras'ı kullanabiliriz. Teknik bakımdan TensorFlow içerisindeki Keras bağımsız Keras'tan daha hızlı ve daha ölçeklenebilir (scalable) biçimdedir. Aşağıda daha önce yapmış olduğumuz Mnist örneğinin Tensorflow içerisindeki Keras kullanılarak gerçekleştimini veriyoruz:

```

from tensorflow.keras.datasets import mnist

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) = mnist.load_data()

import matplotlib.pyplot as plt

for i in range(10):
    plt.imshow(training_dataset_x[i], cmap='gray', interpolation='none')
    plt.pause(1)

training_dataset_x = training_dataset_x.reshape(-1, 28 * 28)
test_dataset_x = test_dataset_x.reshape(-1, 28 * 28)

from tensorflow.keras.utils import to_categorical

training_dataset_y = to_categorical(training_dataset_y)
test_dataset_y = to_categorical(test_dataset_y)

training_dataset_x = training_dataset_x / 255
test_dataset_x = test_dataset_x / 255

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(512, input_dim=28*28, activation='relu', name='Hidden-1'))
model.add(Dense(256, activation='relu', name='Hidden-2'))
model.add(Dense(10, activation='softmax', name='Output'))

model.compile('adam', loss='categorical_crossentropy', metrics=['accuracy'])
hist = model.fit(training_dataset_x, training_dataset_y, epochs=20, batch_size=64,
validation_split=0.2)

import matplotlib.pyplot as plt

plt.title('Epoch - Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['loss'])
plt.legend(['Loss'])

plt.pause(1)

plt.title('Epoch - Accuracy Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Accuracy'

```

```

plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.legend(['Accuracy'])

plt.pause(1)

plt.title('Epoch - Validation Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_loss'])
plt.legend(['Validation Loss'])

plt.pause(1)

plt.title('Epoch - Validation Accuracy Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Validation accuracy'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['Validation Accuracy'])

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

plt.title = 'Epoch - Accuracy / Validation Accuracy Graph';
plt.xlabel = 'Epoch'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['acc'])
plt.plot(range(1, len(hist.epoch) + 1), hist.history['val_acc'])
plt.legend(['Accuracy', 'Validation Accuracy'])

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print('loss = {}, accuracy = {}'.format(loss, accuracy))

import numpy as np

def rgb2gray(rgb):
    return np.dot(rgb[:, :, :], [0.2989, 0.5870, 0.1140])

from matplotlib.image import imread
img = imread('test-7-2.png')

plt.imshow(img, interpolation='none')

gray = rgb2gray(img)
gray = gray.reshape((1, 28 * 28))

predict_result = model.predict(gray)
number = predict_result.argmax(axis=1)
print(number)

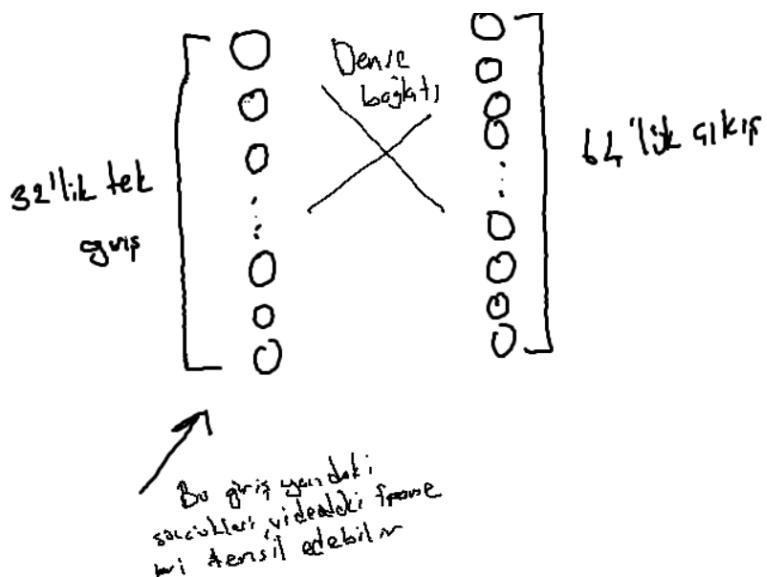
```

Geri Beslemeli Sinir Ağları (Recurrent Neural Networks)

Şimdiye kadar görmüş olduğumuz ağ mimarilerinde bir nöronun çıktısı başka bir nörona girdi yapılmıyordu. Bu tür ağ mimarilerine "ileriye doğru beslemeli (feed forward)" ağlar denilmektedir. İleriye doğru belemeli ağlarda geçmişin kaydı ağıda tutulamamaktadır. Halbuki zamana dayalı uygulamalarda, doğal dil işleme uygulamalarında geçmişin de belli ölçülerde hatırlanması gerekmektedir. Örneğin IMDB veritabanında biz birtakım yorum yazılarından bu yorumların "iyi" mi "kötü" mü olduğu sonucunu çıkartmaya çalışıyorduk. Bu sonucun çıkartılmasında sözcüklerin önemli olduğu açıklıktır. Tasarladığımız ileriye doğru beslemeli ağ bu sözcüklerin önemini eğitim yoluyla öğrenmektedir. Ancak o örneğimizde sözcüklerin sırasının bir önemi yoktu. Halbuki yorum yazılarındaki sözcükler bir bağlam içerisinde gerçek anlamını bulabilmektedir. Yani o sözcüklerin kullandığı yerin geçmiş ve geleceği de o sözcüklerin anlamı üzerinde etkili olmaktadır. O halde böyle bir bağlamın sınır ağlarında oluşturulabilmesi için modelde bir biçimde geçmişin de etkili olması gereklidir. Aynı durum video görüntülerinde de vardır. Videodaki görüntüler aslında frame'lerden (hareketli resmin anlık görüntüsü) oluşmaktadır. Fakat hareketli görüntüyü oluşturan bu frame'ler

birbirlerinden bağımsız değildir. Bir sonraki frame bir önceki frame'e dayalı biçimde bir görüntüyü oluşturmaktadır. Başka bir deyişle görüntüde 10 saniye önce ne olduğu çıkarılacak sonuç için önemli olabilmektedir. Yani video frame'lerinden çıkartılacak sonuçlar geçmişe de dayalıdır. Benze biçimde finansal uygulamaların çoğu da belli bir geçmiş veri temelinde anlam kazanmaktadır. İşte geri beslemeli ağ modeli geçmişin ve bağlamın da kestirimde dikkate alınmasını sağlayan modeldir.

Şimdi biz yapay sinir ağı mimarisinde geri beslemeli bir ağ katmanının nasıl olması gerektiğini ele alalım. Geri beslemeli ağ yukarıda da açıklandığı gibi peşi sıra (zamansal) girdinin uygulandığı ağ modelidir. Bu birden fazla giriş muhtemelen bir sürecin içerisindeki elemanları oluşturmaktadır. Örneğin yazışal bir metindeki sözcükler, bir video görüntüsündeki frame'ler bu sıralı girişleri temsil ediyor olabilir. Biz örneğin 32'lük bir girdi katmanına (yani girişe) sahip olan çıktısı 64'lük olan bir katman düşünelim. Onun şekilsel gösterimi şöyle olacaktır:



Geri beslemeli ağda girişler tek bir tane değildir. Bir grup giriş peşi sıra uygulanmaktadır. (Örneğin bizim daha önce yapmış olduğumuz IMDBörneğinde biz tüm yorum yazısını tek bir girdi olarak ele almıştık. Halbuki aynı örneği geri beslemeli ağ ile yapacak olsak yazındaki her sözcük sıralı girdilerden birini temsil ediyor olacaktır.) Yukarıdaki çizimde girdi 32'lik olduğuna göre ve bu 64'lük bir nöron katmanına (hidden katmana) bağlandığına göre toplamda tahmin edilmesi gereken W değerleri 32 X 64 tane olacaktır. Pekiyi geri besleme nasıl yapılacaktır? Geri besleme çıktı nöron grubundaki her çıktı değerinin yeniden bu katmana geri sokulmasıyla oluşturulur. Çıktı nöron grubundaki nöronların her bir tanesinin çıktısı yine çıktı nöron grubundaki nöronlara bir sonraki girdi ile beraber girdi yapılır. Buradaki işlem yine dot product biçiminde toplama işlemidir. Bu toplama bias değerleri de eklenerek aktivasyon fonksiyonuna sokulmaktadır. Buradaki model aşağıdaki gibi bir Python programıyla oluşturulabilir:

```
import numpy as np

timespec = 100                      # ardışılı uygulanacak eleman sayısı
input_features = 32                  # girdi katmanındaki nöron sayısı
output_features = 64                 # çıktı katmanındaki nöron sayısı

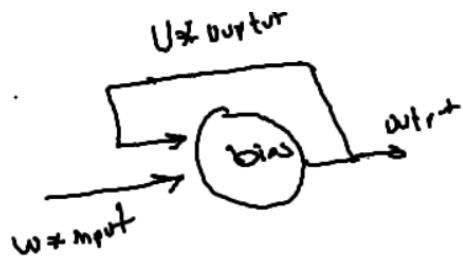
inputs = np.random.random((timespec, input_features))
W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))

outputs = []

output_t = np.random.random((output_features,))
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, output_t) + b)
    outputs.append(output_t)
```

```
total_outputs = np.concatenate(outputs, axis=0)
print(total_outputs)
```

Bu kodda girdi kümesi 32, çıktı kümesi 64 nörondan oluşmaktadır. Toplam 100 girdi vardır. Yani buradaki örneği IMDB'ye benzetirsek yazının 100 sözcükten olduğunu, her sözcüğün toplam 32 sözcükten (vocabulary) bir tanesine ilişkin olduğunu (32'lik one hot encoding) söyleyebiliriz. Başka bir deyişle bu koddaki timespec peşi sıra oluşan girdilerin sayısını, input_features ise bu girdilerin her birinin kaç seçenekten birine ilişkin olduğunu temsil etmektedir. Koddaki output_features çıktı nöron grubundaki nöron sayısını belirtir. Kodumuzda işin başında W, U ve b vektörleri 0 ile 1 arasında rastgele sayılarından oluşan bir biçimde alınmıştır. Buradaki W girdi değerlerinin çarpılacağı ağırlıkları, U ise çıktı değerlerinin yeniden girdi olarak işleme sokulurken çarpılacak ağırlıkları belirtmektedir. b ise çıktı nöron grubundaki nöronların bias değerleridir.



Kodun döngü kısmına dikkat ediniz:

```
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, output_t) + b)
    outputs.append(output_t)
```

Burada girdi vektörü W değerleriyle çarpılıp toplanıp bu toplama U değerlerinin çıktı vektörüyle çarpımı da eklenmiştir. Örnek kodumuzda tüm çıktıların bir liste içerisinde biriktirildiğini de görüyorsunuz. Çünkü bu katmanın çıktısının büyüklüğü output_features (64) kadar olmayacağındır. Bu katmanın çıktı büyülüğu output_features * inputs (yani 64 * 100) kadar olacaktır. Bütün bu çıktı bir sonraki katmana girdi olarak verilecektir. Kodumuzda en sonunda np.concatenate fonksiyonu ile elde edilen listenin bir ndarray üzerinde birleştirildiğini görüyorsunuz.

Aslında yukarıdaki kodun yaptığı katman işlemini zaten yapan SimpleRNN isimli hazır bir Keras katman sınıfı vardır. Sınıfın init metodunun parametrik yapısı şöyledir:

```
tensorflow.keras.layers.SimpleRNN(units, activation='tanh', use_bias=True,
kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
bias_initializer='zeros', kernel_regularizer=None, recurrent_regularizer=None,
bias_regularizer=None, activity_regularizer=None, kernel_constraint=None,
recurrent_constraint=None, bias_constraint=None, dropout=0.0, recurrent_dropout=0.0,
return_sequences=False, return_state=False, go_backwards=False, stateful=False, unroll=False)
```

Buradaki units değeri katmanın çıkışındaki nöron sayısını belirtmektedir. (Yani örneğimizdeki output_features değerini belirtmektedir.) Tabii Keras katmanları peşi sıra yiğildiği için bu katmanın girdisi aslında bir önceki katmanın çıktı değeridir. Dolayısıyla SimpleRNN katmanı ağır ilk katmanı olamaz. (Yani bu katmandan önce bir girdi katmanın olması gereklidir.) Uygulama bağlamında bu katman genellikle "word embedding" içeren Embedding katman olur.

Şimdi bir SimpleRNN katmanını Keras'ta oluşturup modeli inceleyelim:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN

model = Sequential()
model.add(Embedding(100, 32))
model.add(SimpleRNN(64))
model.summary()
```

Bu örnekteki Embedding katman henüz açıklanmamıştır. İzleyen bölümde bu katman "word embedding" konusu içerisinde ele alınacaktır. SimpleRNN katmanın tek başına kullanılmadığını dikkat ediniz. Burada Embedding katmanın çıktı tek bir vektör değildir. (Halbuki Dense katmanlarının çıktıları tek bir vektördür.) Her biri 32'lik vektörlerden oluşan bir matristir. Bu vektör aslında SimpleRNN katmanın girdilerini oluşturmaktadır. Başka bir deyişle aslında Embedding katmanın çıktıları olan 32'lik vektörler SimpleRNN'yi zamansal olarak beslemektedir. Kullanılacak girdileri oluşturmaktadır. Örneğimizdeki SimpleRNN katmanın çıktı ise 64 nöronundan oluşmaktadır. Bu 64 nöronun çıktıdense biçimde yeniden SimpleRNN katmanına yeni zamansal girdi ile toplanarak işleme sokulacaktır. Modelin summary çıktısı aşağıdaki gibidir:

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 32)	3200
simple_rnn_3 (SimpleRNN)	(None, 64)	6208
<hr/>		
Total params: 9,408		
Trainable params: 9,408		
Non-trainable params: 0		

Modeldeki eğitilebilir parametrelerin sayısının nasıl elde edildiğine bakalım: Embedding katmandaki 32'lik girişlerin 100'lük bir one hot encoding'ten elde edilmesi için toplamda ağıda $100 * 32 = 3200$ tane parametrenin bulunması gereklidir. SimpleRNN katmanın girdileri ise 32'lik vektörlerden oluşmaktadır. Fakat bu girişler aynı W değerleri ile çarpılacağından giriş için gereken W değerleri $32 * 64 = 2048$ tane olur. Öte yandan her çıkış ayrıca dense biçimde yeniden SimpleRNN katmanındaki nöronlara bağlanacağından dolayı buradaki U katsayılarının sayısı da $64 * 64 = 4096$ olacaktır. Tabii bir de eğitilebilir bias parametreleri vardır. Bunlar da SimpleRNN katmanındaki nöron sayısı kadardır (yani 64). O halde toplam eğitilebilir parametrelerin sayısı $32 * 64 + 64 * 64 + 64 = 6208$ olur.

Peki yukarıdaki modele bir tane daha SimpleRNN katmanı eklersek ne olur? Örneğin:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN

model = Sequential()
model.add(Embedding(100, 32))
model.add(SimpleRNN(64))
model.add(SimpleRNN(64))
model.summary()
```

Bu kod çalıştırıldığında ikinci SimpleRNN katmanın eklenmesi işleminde exception olacaktır. Exception'ın nedeni ikinci SimpleRNN katmanın matrisel bir girdi beklerken tek boyutlu vektörel bir bilgiyle karşılaşıyor olmasıdır. Çünkü SimpleRNN katmanı default durumda çıktı olarak zamansal girişler uygulandığındaki en son çıktıının değerini vermektedir. (Halbuki bizim daha önce yazdığımız kod parçası bütün çıktıları biriktirip bir matris olarak veriyordu.) İşte SimpleRNN katmanın çıktıının biriktirilmiş çıktı değerlerinden oluşan bir matris olmasını sağlamak için SimpleRNN fonksyonunun return_sequences isimli parametresi True geçilmelidir. Bu parametre True geldiğinde artık ilk SimpleRNN katmanın çıktı 64'lük vektörlerden oluşan bir matris olur. Böylece ikinci SimpleRNN katmanın girdisi için gereken zamansal matris ihtiyacı karşılanır. O halde yukarıdaki kodun düzeltilmiş biçimi şöyle olmalıdır:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN

model = Sequential()
model.add(Embedding(100, 32))
model.add(SimpleRNN(64, return_sequences=True))
model.add(SimpleRNN(64))
model.summary()
```

Buradaki summary çıktısı da şöyledir:

Layer (type)	Output Shape	Param #
<hr/>		
embedding_4 (Embedding)	(None, None, 32)	3200
<hr/>		
simple_rnn_6 (SimpleRNN)	(None, None, 64)	6208
<hr/>		
simple_rnn_7 (SimpleRNN)	(None, 64)	8256
<hr/>		
Total params: 17,664		
Trainable params: 17,664		
Non-trainable params: 0		

Pekiyi burada son SimpleRNN katmanındaki eğitilebilir parametrelerin sayısı nasıl hesaplanmıştır? İkinci SimpleRNN katmanının girdi katmanı aslında 64'lük vektörlerden oluşan bir matristir. Dolayısıyla burada girdi olarak $64 * 64$ tane W değeri oluşacaktır. U değerleri de yine $64 * 64$ tane olacaktır. Tabii 64 tane de bias değeri bulunacaktır. O halde toplam eğitilebilir parametrelerin sayısı $64 * 64 + 64 * 64 + 64 = 8256$ olmaktadır.

Pekiyi bu modelin binary bir çıktı veren model olduğunu varsayırsak nihai modelin çıktı katmanını nasıl oluşturabiliriz? İşte artık çıktı katmanı tek nörondan oluşan Dense bir katman olabilir. Örneğin:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense

model = Sequential()
model.add(Embedding(100, 32))
model.add(SimpleRNN(64, return_sequences=True))
model.add(SimpleRNN(64))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

Modelin summary çıktısını yeniden aşağıda verelim:

Layer (type)	Output Shape	Param #
<hr/>		
embedding_5 (Embedding)	(None, None, 32)	3200
<hr/>		
simple_rnn_8 (SimpleRNN)	(None, None, 64)	6208
<hr/>		
simple_rnn_9 (SimpleRNN)	(None, 64)	8256
<hr/>		
dense_1 (Dense)	(None, 1)	65
<hr/>		
Total params: 17,729		
Trainable params: 17,729		
Non-trainable params: 0		

Son Dense katmanından gelen eğitilebilir parametrelerin sayısının 65 olduğunu dikkat ediniz? Çünkü son SimpleRNN katmanının çıktı 64 tane dir. Son Dense katmanı da tek nörondan oluştuğuna göre buradaki bias değeri de eklendiğinde 65 değeri elde edilir.

Metinsel Uygulamalarda Word Embedding İşlemleri

Metinlerden kestirim yapılmak istendiğinde metinleri oluşturan öğeler girdi olarak sinir ağına verilmektedir. Genellikle bu girdiler sözcükler olur. Fakat bu amaçla karakterler de kullanılabilmektedir. Metni oluşturan sözcüklerin nümerik girdilere dönüştürülmesi IMDB örneğinde olduğu gibi sözcük indeksleme yoluyla yapılmaktadır. Yani tüm sözcükler bir liste oluşturur (vocabulary). Sonra metin içerisinde geçen sözcükler bu listedeki indeksler haline getirilir. Biz IMDB örneğinde metin içerisindeki tüm sözcükleri tek bir vektör olarak ele almıştık. Girdi vektörümüz tüm

sözcüklerin miktarı kadar uzunluktaydı. Metnin içerisinde belli bir sözcük geçmişse biz o indeksli elemanı 1 yaparıyorduk. Böylece girdi vektörümüz bazı elemanları 1 olan fakat pek çok elemanı 0 olan bir vektör biçimine dönüştürülmüş oluyordu. Ancak geri beslemeli ağ kullanılacaksa daha önce uyguladığımız yukarıdaki teknik artık uygun olmayacağından emin olmak gereklidir. Çünkü geri beslemeli ağlarda sıralamanın önemi vardır. Dolayısıyla bu ağların bir hafızası bulunur. Bu durumda girdilerin her sözcük bir vektör olacak biçimde ayrı ayrı oluşturulması gereklidir. Böylece bu vektörler "one hot encoding" biçiminde olacaktır. Örneğin aşağıdaki metni ele alalım:

"film çok güzeldi"

Bu metinde 3 sözcük vardır. Bu 3 sözcük toplam sözcüklerin sayısının (vocabulary) 10000 olduğunu varsayırsak 3×10000 'lik "one hot encoding" matrisi oluşturur. Böylece bu vektörlerin uzunlukları da aynı olacaktır. Bu yöntemin en önemli dezavantajlarından biri metin için toplam girdi matrisinin çok büyük olmasıdır. Ayrıca diğer bir dezavantajı da bu yöntemde sözcükler arasında semantik bir bağlantının oluşamamasıdır. Örneğin "güzeli" ve "iyi" arasında bağlam bakımından bir yakınlık vardır. Halbuki model bu yakınlığı belirleyememektedir.

İşte "word embedding" denilen yöntemde metni oluşturan her bir sözcük büyük bir "one hot encoding" vektörü ile değil, küçük bir reel vektörle ifade edilemeye çalışılır. Örneğin metnimizdeki bir sözcük aşağıdaki gibi bir "one hot encoding" vektörüyle temsil edilmiş olsun:

[0, 0, ..., 0, 0, 1, 0, 0, ..., 0, 0, 0]

Bunun reel vektörel karşılığı aşağıdaki gibi bir vektör olabilir:

[0.78, 0.12, ..., 0.34, 0.67]

Pekiyi geri beslemeli ağlarda zamansal girdilerin "one hot encoding" yerine bu "word embedding" tekniği ile oluşturulmasının avantajları nelerdir? Bunun iki önemli avantajının olduğu söylenebilir:

- 1) Bu sayede girdi kümesi azaltılmış olur. Uzun bir one hot encoding matrisinden kurtulunur.
- 2) Bu sayede sözcükler arasında semantik bir bağlantı da kurulur. Çünkü bu dönüştürme işlemi aslında öğrenme modelinin bir parçasıdır ve söz konusu vektör ağ eğitildikçe daha iyi hale gelmektedir. Sonuçta iki sözcük arasındaki vektörel uzaklık ne kadar azsa o sözcüklerin eğitiminde kullanılan metinler içerisindeki anlamsal benzerlikler o kadar fazla olur.

Pekiyi "word embedding" katmanı için hangi algoritmalar kullanılmaktadır? İşte burada kullanılan algoritmalar çeşitlidir. En popülerleri şunlardır:

Word2Vec (Google)
GloVe (Stanford)
fastText (Facebook)

Keras'taki Embedding katmanı Word2Vec temeline dayandırılmıştır. Biz burada bu algoritmalar üzerinde durmayacağız. Ancak bu katmanın Keras'taki kullanımı üzerinde duracağız.

Keras'ta Embedding katmanı Embedding isimli bir sınıfla temsil edilmiştir. Sınıfın `__init__` metodunun parametrik yapısı şöyledir:

```
tensorflow.keras.layers.Embedding(input_dim, output_dim, embeddings_initializer='uniform',
embeddings_regularizer=None, activity_regularizer=None, embeddings_constraint=None,
mask_zero=False, input_length=None)
```

Fonksiyonun birinci parametresi "one hot encoding" vektöründeki tüm niteliklerin (örneğin sözcüklerin) sayısını (vocabulary) belirtmektedir. Embedding katmanın girdi vektörü "one hot encoding" vektörlerinden oluşan bir matris değil sözcük indekslerinden oluşan bir ndarray nesnesidir. Yani bu dizinin görüntüsü aşağıdaki gibidir:

[23, 245, 876, 34, 356, 30, ..., 87, 34, 90]

Bu sayılar metin içerisinde geçen sözcüklerin indeks numaralarıdır. Tabii aslında Embedding katmanı kendi içerisinde bu vektördeki dizi elemanlarını “one hot encoding”e dönüştürerek işleme sokmaktadır. Girdinin kullanıcıdan bu biçimde istenmesi kullanıcıya kolaylık sağlamaktadır. Fakat burada da yine bir sorun vardır. Eğitimde kullanılacak indekslerden oluşan yukarıdaki vektörlerin hepsinin aynı uzunlukta olması gereklidir. Bu da tüm yazıların aynı miktarda sözcükten oluşması anlamına gelir. Oysa metinlerdeki sözcükler farklı sayıda olabilmektedir. Pekiyi bu sorun nasıl çözülecektir? Kullanılacak en makul yöntem tüm metinlerin sözcük uzunluğunu aynı yapmaktadır. Bunun için tensorflow.keras.preprocessing modülü içerisindeki sequence.pad_sequences fonksiyonundan faydalanabilir. Bu fonksiyon eğer dizi kısaysa onun başını sıfırlarla doldurur, eğer dizi uzunsa sonundan kırpmaya yapar. Pekiyi bu uzunluk nasıl tespit edilecektir? Aslında belli bir uzunluk sevgisel olarak belirlenebilir. Ya da en uzun metnin uzunluğu referans alınabilir.

Embedding fonksiyonun ikinci parametresi çıktı vektörünün uzunluğunu belirtmektedir. Aslında Embedding katmanın çıktısı her biri bu parametrede belirtilen uzunlukta olan bir matristir. Örneğin biz Embedding fonksiyonunu şöyle kullanmış olalım:

```
Embedding(10000, 32)
```

Buradaki 10000 değeri bizim tüm metinlerimizdeki toplam farklı sözcük sayısını (vocabulary) belirtmektedir. Buradaki 32 değeri ise her sözcüğün temsil edildiği vektörün uzunluğudur. Yani her sözcük “one hot encoding” yerine 32'lik bir reel vektörle temsil edilmektedir. Vektör uzunluğu için genellikle 16, 32, 64 gibi değerler tercih edilmektedir. Ancak metin içerisindeki sözcüklerin toplam sayısı çok fazlaysa bu değerin de büyütülmesi olumlu sonuçlar doğurmaktadır.

Pekiyi Embedding fonksiyonu girdi vektörlerinin uzunluğunu nasıl anlamaktadır? İşte fonksiyonun input_length isimli parametresi bu uzunluğu belirlemekte kullanılabilmektedir. Ancak bu parametre girilmediyse fonksiyon otomatik olarak bu belirlemeyi eğitim sırasında giren dizide bakarak yapmaktadır.

Pekiyi Embedding fonksiyonunun ikinci parametresi olan çıktı vektörünün uzunluğu hangi değerde olmalıdır

Geri Beslemeli Ağ Uygulaması Olarak SimpleRNN Katmanlarına Sahip IMDB Örneği

Şimdi yukarıda gördüğümüz konuları IMDB örneğiyle bir araya getirelim. Anımsanacağı gibi IMDB örneğinde kişilerin yorum yazılarından film hakkında olumlu mu olumsuz mu düşünceleri kestirilmeye çalışılıyordu. Daha önce yaptığımız IMDB örneğinde sözcüklerin öncelik sonralık ilişkisine bakmamıştık. Şimdi geri beslemeli bir ağ ile sözcükleri bağlam içerisinde değerlendirmeye çalışalım. Daha önce yaptığımız gibi IMDB verilerini önce Tensorflow.keras'ta okuyalım:

```
from tensorflow.keras.datasets import imdb

max_features = 10000
max_text_len = 500
batch_size = 512
epochs = 5

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
imdb.load_data(num_words=max_features)
```

Burada zaten okunan training_set_x ve training_set_y verileri Embedding katmanın istediği biçimdedir. Örneğin:

```
array([list([1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256,
5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385,
39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2025, 19, 14, 22, 4,
1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12,
16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66, 3785, 33, 4,
130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5,
14, 407, 16, 82, 2, 8, 4, 107, 117, 5952, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476,
26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6,
194, 7486, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92,
```

```

25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 32, 15, 16, 5345,
19, 178, 32]),  

    list([1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118,
1634, 14, 394, 20, 13,...]

```

Bu okuma sonucunda bize bir ndarray verilmekle birlikte bu ndarray list'lerden oluşmaktadır. Aynı zamanda her yorumdaki sözcük sayılarının farklı olması da bir problemdir. İşte her yorumdaki sözcük sayılarını aynı uzunluğa getirmemiz gereklidir. Bu işlem şöyle yapılabilir:

```

from tensorflow.keras.preprocessing import sequence  
  

training_dataset_x = sequence.pad_sequences(training_dataset_x, maxlen=max_text_len)  

test_dataset_x = sequence.pad_sequences(test_dataset_x, maxlen=max_text_len)

```

Şimdi artık tüm yorumlar eşit sayıda sözcüklerden oluşan gibi bir durum elde edilmiştir. Artık modelimizi kurabiliriz:

```

from tensorflow.keras.models import Sequential  

from tensorflow.keras.layers import Embedding, SimpleRNN, Dense  
  

model = Sequential()  

model.add(Embedding(max_features, 32))  

model.add(SimpleRNN(64, return_sequences=True, activation='relu'))  

model.add(SimpleRNN(64, activation='tanh'))  

model.add(Dense(1, activation='sigmoid'))  

model.summary()

```

Burada Embedding katmanının çıktısı 32'luk vektörlerden oluşan bir matristir. Bu matris SimpleRNN katmanına zamansal olarak girdi yapılmıştır. İlk SimpleRNN katmanının çıktısı bu katmanda return_sequences=True yapıldığı için tek bir vektor değil 64'lük vektörlerden oluşan bir matristir. İkinci SimpleRNN katmanın çıktısı ise return_sequences True yapılmadığı için tek bir vektördür (son vektor). Bu katmanın çıktısı da nihai çıktı olan Dense katmanına girdi yapılmıştır. Model için elde edilen summary tablosu şöyledir:

Layer (type)	Output Shape	Param #
<hr/>		
embedding_2 (Embedding)	(None, None, 32)	320000
simple_rnn_3 (SimpleRNN)	(None, None, 64)	6208
simple_rnn_4 (SimpleRNN)	(None, 64)	8256
dense_2 (Dense)	(None, 1)	65
<hr/>		
Total params: 334,529		
Trainable params: 334,529		
Non-trainable params: 0		

Şimdi artık sıra ağın derlenip eğitilmesine gelmiştir.

```

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])  
  

history = model.fit(training_dataset_x, training_dataset_y, epochs=epochs,
batch_size=batch_size, validation_split=0.2)

```

Şimdi elde edilen modelin grafiğini çizelim:

```

import matplotlib.pyplot as plt  
  

acc = history.history['acc']
val_acc = history.history['val_acc']  
  

plt.plot(range(1, epochs + 1), acc, 'r', label='Training Accuracy')
plt.plot(range(1, epochs + 1), val_acc, 'b', label='Validation Accuracy')

```

```

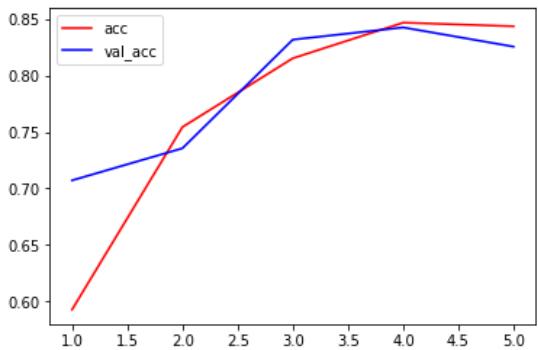
plt.legend(['acc', 'val_acc'])

plt.pause(1)

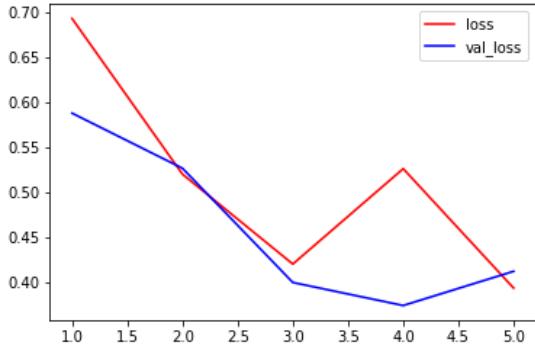
loss = history.history['loss']
val_loss = history.history['val_loss']

plt.plot(range(1, epochs + 1), loss, 'r', label='Training Loss')
plt.plot(range(1, epochs + 1), val_loss, 'b', label='Validation Loss')
plt.legend(['loss', 'val_loss'])

```



Dout[19]: <matplotlib.legend.Legend at 0x2379426fe48>



Göründüğü gibi beşinci epoch sonucunda %82-85'lük bir kestirim başarısı elde edilmiştir.

Yukardaki örneğin tüm kodları aşağıda verilmiştir:

```

max_features = 10000
max_text_len = 400
batch_size = 512
epochs = 5

from tensorflow.keras.datasets import imdb

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
imdb.load_data(num_words=max_features)

from tensorflow.keras.preprocessing import sequence

training_dataset_x = sequence.pad_sequences(training_dataset_x, maxlen=max_text_len)
test_dataset_x = sequence.pad_sequences(test_dataset_x, maxlen=max_text_len)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(64, return_sequences=True, activation='relu'))
model.add(SimpleRNN(64, activation='relu'))

```

```

model.add(Dense(1, activation='sigmoid'))

model.summary()

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=[ 'accuracy'])

history = model.fit(training_dataset_x, training_dataset_y, epochs=epochs,
batch_size=batch_size, validation_split=0.2)

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print(loss, accuracy)

import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

plt.plot(range(1, epochs + 1), acc, 'r', label='Training Accuracy')
plt.plot(range(1, epochs + 1), val_acc, 'b', label='Validation Accuracy')
plt.legend(['acc', 'val_acc'])

plt.pause(1)

loss = history.history['loss']
val_loss = history.history['val_loss']

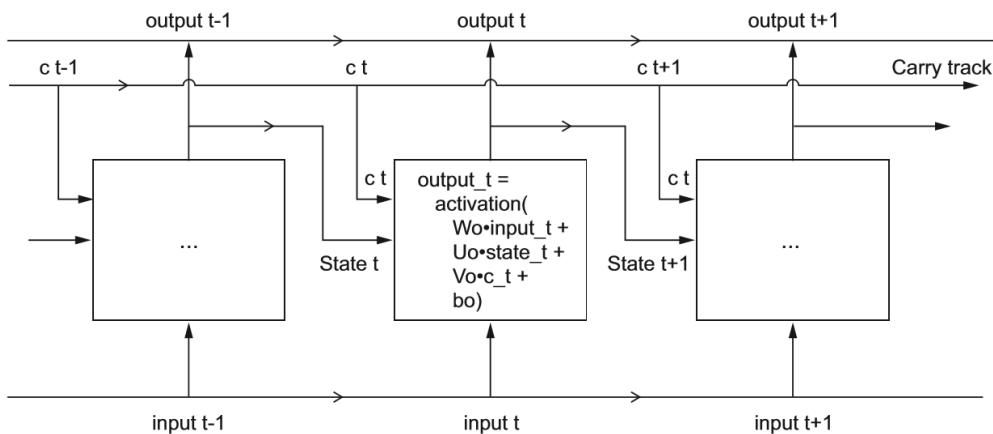
plt.plot(range(1, epochs + 1), loss, 'r', label='Training Loss')
plt.plot(range(1, epochs + 1), val_loss, 'b', label='Validation Loss')
plt.legend(['loss', 'val_loss'])

```

Geri Beslemeli Ağlar İçin LSTM Katmanı

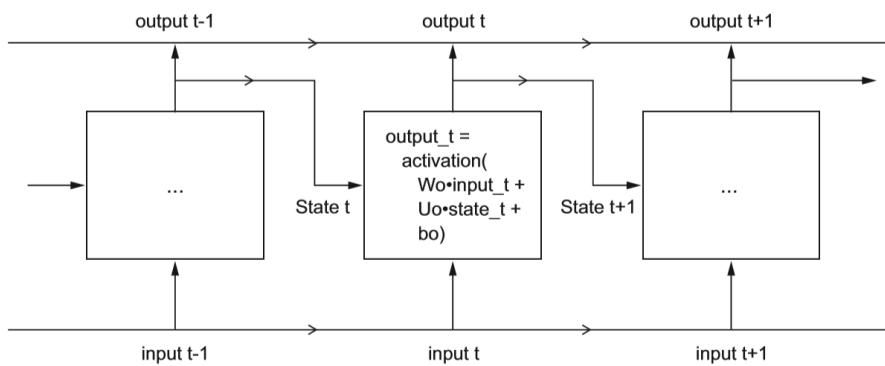
Yukarıda kullanmış olduğumuz SimpleRNN katmanı aslında uygulamalarda tercih edilmemektedir. Çünkü bu katmanın bazı sorunları da vardır. SimpleRNN katmanı her ne kadar geçmişe doğru bir hafıza oluşturuyorsa da bu hafıza "vanishing gradient problem" ismi verilen bir sorun yüzünden yakın geçmişe yönelik kalmaktadır. Yani SimpleRNN katmanı o andaki bağlamın hemen öncesini yine bir biçimde hatırlayabilmekte geçmişi gittikçe daha zor hatırlayabilmektedir. Bu durum bağlamsal etkiyi azaltmaktadır. Bunun çözümü araştırmacılar çeşitli yöntemler önermişlerdir. 90'lı yılların sonlarına doğru LSTM ve GRU katmanları bu sorunu çözmek için kullanılmıştır.

LSTM (Long Short TermMemory) katmanı ağa hem kısa süreli hem de uzun süreli bir hafıza oluşturmayı hedeflemektedir. LSTM SimpleRNN'nin mimari olarak biraz daha geliştirilmiş bir biçimidir. LSTM katmanında uzun dönem hafıza etkisi için yeni bir giriş daha ağa eklenmiş durumdadır. Mimarisinin temsili görüntüsü şöyledir:



Bu mimaride SimpleRNN'den farklı olarak bir de c girişi uygulanmıştır. Bu durumda geri beslemeli ağın girdileri şunlar oluşmaktadır: Zamansal gerçek girdiler (yani metinsel uygulamalardaki sözcükler, şekildeki "input"lar), önceki çıktı

(şekildeki "state") ve uzun dönem hafıza etkisi sağlayan "c" (Şekildeki "c") girişi. Bu yeni modeli SimpleRNN ile karşılaştırınız:



Peki buradaki c vektörleri nasıl hesaplanmaktadır? Biz bu noktada bu hesaplama üzerinde durmayacağız. Ancak sekilden de görüldüğü gibi bu " c " vektörleri doğrudan önceki çıkışlardan alınmamaktadır. Ağa uzun dönem hafıza özelliği bu " c " girişi sayesinde dahil edilmektedir.

Keras'ta LSTM katmanı doğrudan LSTM isimli sınıfta temsil edilmiştir. Bu katmanın kullanılması SimpleRNN katmanına benzemektedir.

```
tensorflow.keras.layers.LSTM(units, activation='tanh', recurrent_activation='sigmoid',
use_bias=True, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None,
recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None,
kernel_constraint=None, recurrent_constraint=None, bias_constraint=None, dropout=0.0,
recurrent_dropout=0.0, implementation=2, return_sequences=False, return_state=False,
go_backwards=False, stateful=False, unroll=False)
```

Burada görüldüğü gibi tek zorunlu parametre birinci units parametresidir. Bu parametre çıktıının nöron sayısını belirtir. Yani bu anlamda LSTM katmanının basit kullanımı tamamen SimpleRNN katmanı gibidir.

Pekiyi LSTM katmanı her zaman SimpleRNN'ye göre tercih edilmeli midir? Aslında LSTM uzun dönem hafıza etkisini sağladığına göre bunun bazı modellerde dezavantajı da olabilmektedir. Fakat genellikle bu uzun dönem etki istenir. Bu nedenle LSTM katmanı çoğu kez SimpleRNN'ye tercih edilmektedir. Şimdi IMDB örneğini LSTM katmanı ile yapalım.

```
max_features = 10000
max_text_len = 400
batch_size = 512
epochs = 4

from tensorflow.keras.datasets import imdb

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
imdb.load_data(num_words=max_features)

from tensorflow.keras.preprocessing import sequence

training_dataset_x = sequence.pad_sequences(training_dataset_x, maxlen=max_text_len)
test_dataset_x = sequence.pad_sequences(test_dataset_x, maxlen=max_text_len)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(64, activation='tanh'))
model.add(Dropout(0.2))
```

```

model.add(Dense(64, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))

model.summary()

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

history = model.fit(training_dataset_x, training_dataset_y, epochs=epochs,
batch_size=batch_size, validation_split=0.2)

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print(loss, accuracy)

import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

plt.plot(range(1, epochs + 1), acc, 'r', label='Training Accuracy')
plt.plot(range(1, epochs + 1), val_acc, 'b', label='Validation Accuracy')
plt.legend(['acc', 'val_acc'])

plt.pause(1)

loss = history.history['loss']
val_loss = history.history['val_loss']

plt.plot(range(1, epochs + 1), loss, 'r', label='Training Loss')
plt.plot(range(1, epochs + 1), val_loss, 'b', label='Validation Loss')
plt.legend(['loss', 'val_loss'])

```

Burada görüldüğü gibi LSTM katmanının çıktısı ayrı bir Dense katmanına da sokulmuştur. Ancak bu durum model için çok önemli bir etkiye yol açmamaktadır. Modeldeki Dropout katmanları fazla nöronları budamanın yanı sıra yüksek epoch değerleri için "overfit" direncini de artırmaktadır.

Çift Yönlü LSTM Katmanı

LSTM katmanı biraz daha iyileştirilebilir mi? Normal olarak LSTM katmanı uzun dönem geçmiş de hatırlama konusunda katkı sağlamaktadır. Pekiyi bağlamsal olarak gelecek geçmiş ile ilişkili olabilir mi? Yani bizim gelecek verilerden elde ettiğimiz bilgiler geçmiş yorumlamada kullanılabilir mi? Yanıt bazı uygulamalar için evet olacaktır. Ancak deneyimler bunun her türlü geri beslemeli ağda mutlak anlamda bir fayda sağlamadığını göstermiştir. Gelecek bilginin geçmiştekilerin anlaşılması kolaylaştırması metinsel uygulamalarda kullanılabilir bir durumdur. Örneğin kişi yorumunda önce birisinin bir yere gittiğiniz söylüyor olabilir. Daha sonra bu yerin eczane olduğu anlaşılıyor olabilir. Bu durumda geçmişte o yerin eczane olduğu bilinse belki de metin daha iyi anlaşılacaktır. Bazı dillerin gramatik yapısı bu durumu daha belirgin hale getirmektedir.

Keras'ta çift yönlü LSTM katmanını oluşturabilmek için LSTM nesnesi Bidirectional isimli sınıfı verilir. Yani katmanın tipik oluşturulma biçimini söyleyelim:

```
model.add(Bidirectional(LSTM(64)))
```

Şimdi aynı modeli Bidirectional LSTM ile kurmaya çalışalım:

```

max_features = 10000
max_text_len = 400
batch_size = 512
epochs = 4

from tensorflow.keras.datasets import imdb

```

```

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
imdb.load_data(num_words=max_features)

from tensorflow.keras.preprocessing import sequence

training_dataset_x = sequence.pad_sequences(training_dataset_x, maxlen=max_text_len)
test_dataset_x = sequence.pad_sequences(test_dataset_x, maxlen=max_text_len)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(Bidirectional(LSTM(64, activation='tanh')))
model.add(Dropout(0.2))
model.add(Dense(64, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))

model.summary()

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

history = model.fit(training_dataset_x, training_dataset_y, epochs=epochs,
batch_size=batch_size, validation_split=0.2)

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print(loss, accuracy)

import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

plt.plot(range(1, epochs + 1), acc, 'r', label='Training Accuracy')
plt.plot(range(1, epochs + 1), val_acc, 'b', label='Validation Accuracy')
plt.legend(['acc', 'val_acc'])

plt.pause(1)

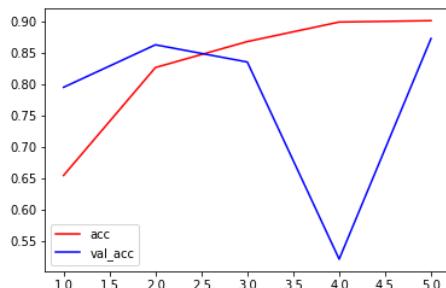
loss = history.history['loss']
val_loss = history.history['val_loss']

plt.plot(range(1, epochs + 1), loss, 'r', label='Training Loss')
plt.plot(range(1, epochs + 1), val_loss, 'b', label='Validation Loss')
plt.legend(['loss', 'val_loss'])

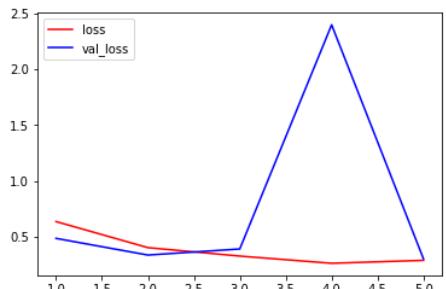
plt.plot(range(1, epochs + 1), loss, 'r', label='Training Loss')
plt.plot(range(1, epochs + 1), val_loss, 'b', label='Validation Loss')
plt.legend(['loss', 'val_loss'])

```

Buradan elde edilen grafikler de şöyledir:



Out[44]: <matplotlib.legend.Legend at 0x237a604e9b0>



Göründüğü gibi burada değerler %90 civarına kadar iyileşmiştir. Yani IMDB örneğinde çift yönlü geri beslemeli mimarinin daha iyi sonuç verdiği görülmektedir.

Geri Beslemeli Ağlar İçin GRU Katmanı

GRU (Gated Recurrent Unit) yöntemi LSTM'ye bir alternatif oluşturmaktadır. GRU yöntemi de tipki LSTM yönteminde olduğu gibi ağa uzun dönem hafıza kazandırmaya çalışmaktadır. GRU toplamda LSTM yöntemine göre daha az eğitilebilir parametreye sahiptir. Dolayısıyla bu yöntem daha hızlıdır, az bellek kullanır. Genel olarak pek çok uygulamada ilk tercih edilecek yöntem LSTM'dir. Eğer donanım kısıtı söz konusuysa (örneğin düşük güçlü CPU ve az miktarda bellek (gömülü sistemler) LSTM yerine GRU katmanı tercih edilebilir. Katmanın genel kullanımı LSTM'de olduğu gibidir. Tabii GRU katmanı genel duyarlılık olarak LSTM'den kötüyse de SimpleRNN'den oldukça iyidir.

Örneğin:

```
max_features = 10000
max_text_len = 400
batch_size = 512
epochs = 4

from tensorflow.keras.datasets import imdb

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) =
imdb.load_data(num_words=max_features)

from tensorflow.keras.preprocessing import sequence

training_dataset_x = sequence.pad_sequences(training_dataset_x, maxlen=max_text_len)
test_dataset_x = sequence.pad_sequences(test_dataset_x, maxlen=max_text_len)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(GRU(64, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(64, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))

model.summary()
```

```

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['accuracy'])

history = model.fit(training_dataset_x, training_dataset_y, epochs=epochs,
batch_size=batch_size, validation_split=0.2)

loss, accuracy = model.evaluate(test_dataset_x, test_dataset_y)
print(loss, accuracy)

import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

plt.plot(range(1, epochs + 1), acc, 'r', label='Training Accuracy')
plt.plot(range(1, epochs + 1), val_acc, 'b', label='Validation Accuracy')
plt.legend(['acc', 'val_acc'])

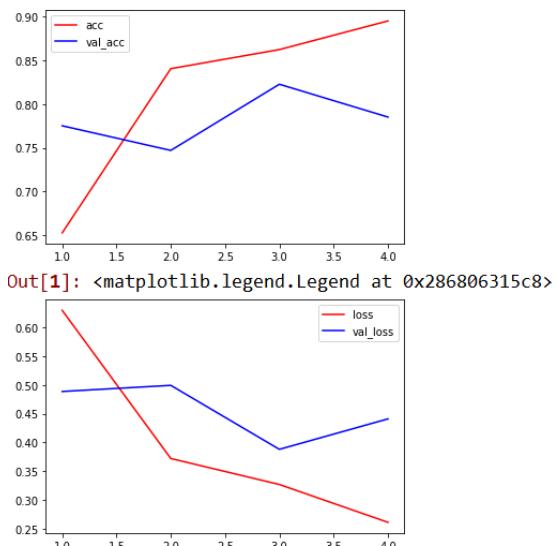
plt.pause(1)

loss = history.history['loss']
val_loss = history.history['val_loss']

plt.plot(range(1, epochs + 1), loss, 'r', label='Training Loss')
plt.plot(range(1, epochs + 1), val_loss, 'b', label='Validation Loss')
plt.legend(['loss', 'val_loss'])

```

Burada elde edilen sonuçlar LSTM'ye göre biraz daha kötüdür:



Keras'taki Bidirectional katmanının bir dekoratör biçiminde oluşturulduğuna dikkat ediniz. Böylece biz herhangi bir geri beslemeli katmanı çift yönlü hale getirebiliriz. Örneğin GRU katmanı da aşağıdaki gibi çift yönlü hale getirilebilir:

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Dense, Dropout, Bidirectional, GRU

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(Bidirectional(GRU(64, activation='tanh')))
model.add(Dropout(0.2))
model.add(Dense(64, activation='tanh'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))

```

```
model.summary()
```

Geri Beslemeli Ağların Gelecek Kestiriminde Kullanılması

Biz yukarıdaki IMDB örneğinde geri beslemeli ağları yazışal metinlerden anlam çıkarmayı hedefleyen IMDB verilerinde kullandık. Halbuki bu ağlar yazışal verilerin dışında tamamen sayısal verilerin söz konusu olduğu kestirimlerde de kullanılabilmektedir. Çünkü pek çok sayısal veri tipki yazısı oluşturan sözcükler gibi bir bağlama sahip olabilmektedir. Örneğin hava tahmini, borsada belli bir kağıdın gelecekteki değerinin tahmini geçmiş verilere bakılarak geri beslemeli bir ağ ile yapılmak istenebilir.

Burada haava tahmini üzerinde örnek bir uygulama yapacağız. Örneğin geçmiş hava durumuraporlarına bakarak bir gün sonraki hava durumunu tahmin edeceğiz. Tabii şüphesiz bu tahmin daha önce yaptığımız gibi geri beslemeli olmayan ağlarla (feed forward networks) da yapılabilir. Ancak biz burada bu kestirimini LSTM katmanlı bir ağla yapacağız.

Hava durumuna ilişkin örnek veriler aşağıdaki CSV dosyasında bulunmaktadır:

https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip

Bu CSV dosyasının genel görünümü şöyledir:

```
Date Time","p (mbar)","T (degC)","Tpot (K)","Tdew (degC)","rh (%)","VPmax (mbar)","VPact (mbar)","VPdef (mbar)","sh (g/kg)","H2OC (mmol/mol)","rho (g/m**3)","wv (m/s)","max. wv (m/s)","wd (deg)"  
01.01.2009 00:10:00,996.52,-8.02,265.40,-8.90,93.30,3.33,3.11,0.22,1.94,3.12,1307.75,1.03,1.75,152.30  
01.01.2009 00:20:00,996.57,-8.41,265.01,-9.28,93.40,3.23,3.02,0.21,1.89,3.03,1309.80,0.72,1.50,136.10  
01.01.2009 00:30:00,996.53,-8.51,264.91,-9.31,93.90,3.21,3.01,0.20,1.88,3.02,1310.24,0.19,0.63,171.60  
01.01.2009 00:40:00,996.51,-8.31,265.12,-9.07,94.20,3.26,3.07,0.19,1.92,3.08,1309.19,0.34,0.50,198.00  
01.01.2009 00:50:00,996.51,-8.27,265.15,-9.04,94.10,3.27,3.08,0.19,1.92,3.09,1309.00,0.32,0.63,214.30  
01.01.2009 01:00:00,996.50,-8.05,265.38,-8.78,94.40,3.33,3.14,0.19,1.96,3.15,1307.86,0.21,0.63,192.70  
01.01.2009 01:10:00,996.50,-7.62,265.81,-8.30,94.80,3.44,3.26,0.18,2.04,3.27,1305.68,0.18,0.63,166.50  
01.01.2009 01:20:00,996.50,-7.62,265.81,-8.36,94.40,3.44,3.25,0.19,2.03,3.26,1305.69,0.19,0.50,118.60  
01.01.2009 01:30:00,996.50,-7.91,265.52,-8.73,93.80,3.36,3.15,0.21,1.97,3.16,1307.17,0.28,0.75,188.50  
...
```

Bu CSV dosyasının başında bir başlık kısmı vardır. Satırlar virgülerle ayrılmış değerler içermektedir. Bu değerlerin bir tanesi (2'inci indeksli sütun) havanın derece cinsinden ısısını belirtmektedir. Bizim bu CSV dosyasını alarak iki boyutlu bir ndarray nesnesine dönüştürmemiz gereklidir. Tabii bu dönüştürme sırasında ilk sütunun da atılması uygun olur. Çünkü buradaki satır verileri 10'ar dakikalık hava örneklerinden oluşmaktadır. İlk sütun örneğin alındığı tarih ve zamanı belirtmektedir. Bu işlem şöyle yapılabilir:

```
import numpy as np  
  
f = open('JenaClimate.csv')  
all_lines = f.read().split('\n')  
header = all_lines[0].split(',')  
print(header)  
  
lines = all_lines[1:]  
data = np.zeros((len(lines), len(header) - 1))  
for i, line in enumerate(lines):  
    data[i, :] = [float(val) for val in line.split(',')][1:]  
print(data)
```

Tabii aslında numpy kütüphanesinde bu işi tek yapabilecek loadtxt isimli bir fonksiyon vardır. O halde yukarıdaki işlem tek hamlede aşağıdaki gibi de yapılabilir:

```
dataset_x = np.loadtxt('JenaClimate.csv', delimiter=',', skiprows=1, usecols=range(1, 15),  
dtype=np.float32)
```

Şimdi bu verileri normalize edelim:

```
dataset_x = (dataset_x - dataset_x.mean(axis=0)) / dataset_x.std(axis=0)
```

Şimdi de eğitim için y veri kümesini elde etmeye çalışalım. Bizim bu kestirimdeki amacımız önceki günde datalara bakılarak sonraki günde aynı saatteki hava sıcaklığını anlamak olsun. Buradaki veriler 10 dakikada bir alındığına göre t zamanındaki verinin bir gün sonraki karşılığı bu dizideki 144 sonraki elemandır. Şimdi bunları elde edelim:

```
feature_step = 144 # 24 * 60 / 10

dataset_y = np.zeros((dataset_x.shape[0] - feature_step), 1)

for i in range(dataset_x.shape[0] - feature_step):
    dataset_y[i, 0] = dataset_x[i + feature_step, 1]
dataset_x = dataset_x[:-feature_step]
```

Şimdi de elimizdeki veri kümesini eğitim ve test için parçalara ayıralım:

```
from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.2)

training_dataset_x = training_dataset_x.reshape(-1, training_dataset_x.shape[1], 1)
test_dataset_x = test_dataset_x.reshape(-1, test_dataset_x.shape[1], 1)
```

Biz daha önce SimpleRNN, LSTM ve GRU katmanlarını hep WordEmbedding katmanı ile kullanmıştır. Tabii aslında bu katmanlar WordEmbedding ile kullanılmak zorunda değildir. Burada dikkat edilmesi gereken nokta geri beslemeli katmanların girdileri üç boyutlu aldığıdır. Çünkü geri beslemeli katmanlar bir satırda bilgiyi tek hamlede değil eleman eleman ele almaktadır.

Şimdi kestirim modelimizi geri beslemeli ağ biçiminde kuralım:

```
epochs = 5

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout

model = Sequential()
model.add(LSTM(64, activation='tanh'))
model.add(Dropout(rate=0.2))
model.add(Dense(64, activation='tanh'))
model.add(Dropout(rate=0.2))
model.add(Dense(1))

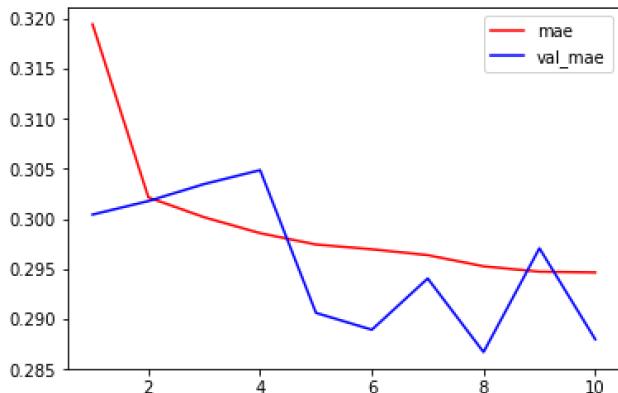
model.compile(optimizer='rmsprop', loss='mae', metrics=['mae'])
history = model.fit(training_dataset_x, training_dataset_y, epochs=epochs,
validation_split=0.2)

import matplotlib.pyplot as plt

mae = history.history['mae']
val_mae = history.history['val_mae']

plt.plot(range(1, epochs + 1), mae, 'r', label='Absolute Error')
plt.plot(range(1, epochs + 1), val_mae, 'b', label='Validation Absolute Error')
plt.legend(['mae', 'val_mae'])
```

Burada çıkış katmanının aktivasyon fonksiyonunun "linear" alındığına dikkat ediniz. Relu fonksiyonu değerleri pozitif tutmaktadır. 10 Epoch için yukarıdaki modelin "mean absolute error" grafiği şöyledir:



Modelin testi de şöyle yapılabilir:

```
result = model.evaluate(test_dataset_x, test_dataset_y)
print(result)
```

Aşağıda tüm program bütün olarak verilmiştir:

```
import numpy as np

dataset_x = np.loadtxt('jena_climate_2009_2016.csv', dtype='float32', delimiter=',',
skiprows=1, usecols=range(1, 15))
feature_step = 144 # 24 * 60 / 10

dataset_y = np.zeros((dataset_x.shape[0] - feature_step, 1), dtype='float32')
for i in range(dataset_x.shape[0] - feature_step):
    dataset_y[i, 0] = dataset_x[i + feature_step][1]
dataset_x = dataset_x[:-feature_step]

from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.2)

training_dataset_x = training_dataset_x.reshape(-1, training_dataset_x.shape[1], 1)
test_dataset_x = test_dataset_x.reshape(-1, test_dataset_x.shape[1], 1)

epochs = 5

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dropout, Dense, Bidirectional

model = Sequential()
model.add(Bidirectional(LSTM(64)))
model.add(Dropout(rate=0.2))
model.add(Dense(64, activation='tanh'))
model.add(Dropout(rate=0.2))
model.add(Dense(1))

model.compile('rmsprop', loss='mae', metrics=['mae'])
history = model.fit(training_dataset_x, training_dataset_y, batch_size=32, epochs=epochs,
validation_split=0.2)

import matplotlib.pyplot as plt

mae = history.history['mae']
val_mae = history.history['val_mae']

plt.plot(range(1, epochs + 1), mae, 'r', label='Absolute Error')
```

```

plt.plot(range(1, epochs + 1), val_mae, 'b', label='Validation Absolute Error')
plt.legend(['mae', 'val_mae'])

dataset_x = np.loadtxt('JenaClimate.csv', delimiter=',', skiprows=1, usecols=range(1, 15),
dtype=np.float32)
dataset_x = (dataset_x - dataset_x.mean(axis=0)) / dataset_x.std(axis=0)

result = model.evaluate(test_dataset_x, test_dataset_y)
print(result)

```

Şimdi aynı model, çift yönlü geri beslemeli ağ ile deneyelim:

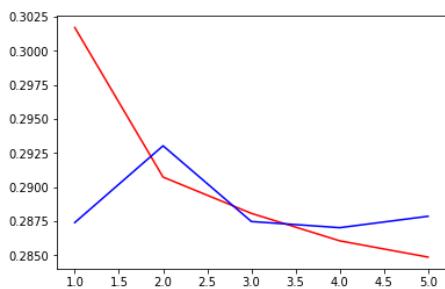
```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, Bidirectional

model = Sequential()
model.add(Bidirectional(LSTM(64)))
model.add(Dropout(rate=0.2))
model.add(Dense(64, activation='tanh'))
model.add(Dropout(rate=0.2))
model.add(Dense(1))

```

Buradan elde edilen grafikler şöyledir:



Çift yönlü ağda bu örnekte önemli bir iyileşme olmadığı görülmektedir.

Tabii yukarıda da belirtildiği gibi bu hava sıcaklığı tahmini bir regresyon modelidir. Aslında daha önce yaptığımız gibi bu model normal ileri beslemeli ağlarla (feed forward) da gerçekleştirilebilirdi. Aşağıda bu modelin normal ileri beslemeli bir ağ ile gerçekleştirmi verilmiştir. Buradan elde edilen mean_absolute_error değeri geri beslemeli olandan daha yüksektir. Yani modelde geri besleme modelin başarısını artırmıştır. Ancak iki yönlü geri besleme tek yönlü geri beslemeye göre önemli bir fayda sağlamamıştır:

```

import numpy as np

dataset_x = np.loadtxt('jena_climate_2009_2016.csv', dtype='float32', delimiter=',',
skiprows=1, usecols=range(1, 15))
feature_step = 144 # 24 * 60 / 10

dataset_y = np.zeros((dataset_x.shape[0] - feature_step, 1), dtype='float32')
for i in range(dataset_x.shape[0] - feature_step):
    dataset_y[i, 0] = dataset_x[i + feature_step][1]
dataset_x = dataset_x[:-feature_step]

from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.2)

epochs = 10

from tensorflow.keras.models import Sequential

```

```

from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(128, activation='tanh'))
model.add(Dense(128, activation='tanh'))
model.add(Dense(1))

model.compile('rmsprop', loss='mae', metrics=['mae'])
history = model.fit(training_dataset_x, training_dataset_y, batch_size=32, epochs=epochs,
validation_split=0.2)

import matplotlib.pyplot as plt

mae = history.history['mae']
val_mae = history.history['val_mae']

plt.plot(range(1, epochs + 1), mae, 'r', label='Absolute Error')
plt.plot(range(1, epochs + 1), val_mae, 'b', label='Validation Absolute Error')
plt.legend(['mae', 'val_mae'])

dataset_x = np.loadtxt('JenaClimate.csv', delimiter=',', skiprows=1, usecols=range(1, 15),
dtype=np.float32)
dataset_x = (dataset_x - dataset_x.mean(axis=0)) / dataset_x.std(axis=0)

result = model.evaluate(test_dataset_x, test_dataset_y)
print(result)

```

Geri Beslemeli Ağlarla Çıktı Üretimi

Biz şimdide kadarki örneklerimizde gelecek kestirimi yapmaya çalıştık. Ancak bazen bu gelecek kestirimi çıktı üretimleri için de kullanılabilmektedir. Özellikle metinlerin üretiminde 2015 yılından itibaren LSTM tabanlı geri beslemeli ağlar kullanılmıştır. Bu konuda önemli başarılar elde edilmiştir.

Geri beslemeli ağlarla metin oluşturma nasıl yapılmaktadır? Tipik uygulama şöyledir:

- 1) Üretilen metni belli bir kişinin tarzına uydurmak için o kişinin yazıları elde edilir.
- 2) Sonra bir geri beslemeli sinir ağı oluşturularak sinir ağı bu yazılarla eğitilir. Eğitim karakter temelinde yapılabildiği gibi sözcük temelinde de yapılmaktadır. Tipik olarak kişinin yazılarının belli miktardaki karakterden oluşan kısımları - örneğin 32 karakter olduğunu varsayıyalım- ve bu kısımdan sonra gelen karakterler eğitim amacıyla toplanır. Böylece elde n tane 32 karakterden oluşan bir yazı parçası ile 1 karakterden oluşan bir çıktı değeri elde edilir. un bir sinir ağı (tipik olarak LSTM katmanlı bir ağı) oluşturularak bu değerler ağını eğitilmesinde kullanılacaktır. Uygulamada yazılarından parçalar atlaamalı (step) olarak da alınabilmektedir.
- 3) Artık elimizde 32 karakterini girdi olarak verdığımız bir metnin 33'üncü karakterini tahmin edebilen bir geri beslemeli yapay sinir ağı modeli vardır. Bu durumda biz 32 karakterlik bir yazıyı başlangıç yazısı olarak verirsek ağdan bunun 33'üncü karakterini elde ederiz. Bundan sonra yine bu 33 karakterli yazının son 32 karakterini alarak yeni bir karakter elde ederiz. Bu işlemi böyle sürdürürsek istediğimiz uzunlukta bir yazı elde etmiş oluruz.
- 4) Modelin bu biçimde oluşturulması yazının tek düzeye ve yaratıcılık içermeyen bir yapıdamasına yol açabilmektedir. Bu nedenle her defasında ağdan elde edilen karakteri doğrudan kullanmak yerine işin içine rassallık da katarak belli bir olasılık dağılımı çerçevesinde karakteri elde etmek daha yaratıcı bir metnin oluşmasına yol açmaktadır.

Şimdi Nietzsche'nin metinlerinden oluşan bir yazıyı kullanarak Nietzsche tarzında metin üreten bir uygulama üzerinde duralım. Bu uygulama "Deep Learning With Python" isimli kitabı 274. Sayfasında da bulunmaktadır. Uygulama şu aşamalardan geçilerek oluşturulmuştur:

Uygulamada Nietzsche'nin metinleri kullanılmıştır. Bu metinler '<https://s3.amazonaws.com/text-datasets/nietzsche.txt>' adresinden indirilebilir. İndirme işlemi ve metnin bir string çevrilmesi şöyle yapılmıştır:

```
import tensorflow.keras

path = tensorflow.keras.utils.get_file('nietzsche.txt', origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')

text = open(path).read().lower()
```

Burada artık tüm metin bir string olarak text değişkeninde bulunmaktadır.

Şimdi metin okunduktan sonra yazıda 3'er atlamalı olarak 60'ar karakterden oluşan parçaların sentences isimli bir listeye yerlestirelim. Yapay sinir ağının 60 karakterden sonraki 61'inci karakteri tahmin edeceğini düşünerek bu 60 karakterden sonraki karakter de next_char isimli bir listede toplayacağız:

```
sentences = []
next_chars = []
maxlen = 60
step = 3

for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i:i + maxlen])
    next_chars.append(text[i + maxlen])

print('Number of sequences: {}'.format(len(sentences)))
```

Böylece biz eğitimde kullanacağımız 60'ar karakterden oluşan bir liste (training_dataset_x) ile bunlardan sonra gelen karakterlerden oluşan diğer bir liste (training_dataset_y) elde etmiş olduk. Bundan sonra yazı içerisinde farklı kaç tane karakter olduğunu tespit edip bu karakterleri de bir listeye yerleştirerek koda devam edelim:

```
chars = sorted(list(set(text)))
print('Number of unique chars: {}'.format(len(chars)))
```

Tabii bizim yazı parçalarındaki karakterlerin her birini "one hot encoding" biçiminde oluşturmamız gereklidir. Zira eğer biz karakter kodlarını doğrudan kullanırsak model sanki sıralı (ordinal) bir ölçek söz konusuymuş gibi davranış olacaktır. One hot encoding işlemini kolaylaştırmak için bir sözlük oluşturabiliriz. Bu sözlük sayesinde yazı içerisindeki bir karakter verildiğinde onun indeksini hızlı bir biçimde elde edebileceğiz:

```
char_dict = dict((char, index) for index, char in enumerate(chars))
```

Buradaki char_dict sözlüğü nesnesinde biz bir karakteri verdığımızda onun chars listesi içerisindeki sıra numarasını elde ederiz. Artık sıra training_dataset_x ve training_dataset_y nesnelerinin ndarray olarak oluşturulmasına geldi. Bunun için önce bu diziler içinde sıfır olacak biçimde aşağıdaki yaratalım:

```
import numpy as np

training_dataset_x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.int8)
training_dataset_y = np.zeros((len(sentences), len(chars)), dtype=np.int8)
```

Burada dtype diziler fazla yer kaplamamasın diye int8 türünden alınmıştır. training_dataset_x'in üç boyutlu bir dizi olduğuna dikkat ediniz. Birinci boyut 60'luk yazı kümlerinin indeksini belirtir. (Eğitim kümesinin 60 karakterlik yazılarından olduğunu anımsayınız.) İkinci boyut yazı içerisindeki karakterin indeksini üçüncü boyut ise yazı içerisindeki karakterin "one hot encoding" vektör karşılığını belirtmektedir. Dizinin birinci boyutunun len(sentences) uzunlığında, ikinci boyutunun maxlen (60) uzunlığında ve üçüncü boyutunun da len(chars) uzunlığında olduğuna dikkat ediniz. len(chars) yazısındaki farklı karakterlerin sayısıdır. "One hot encoding" için bu len(chars) uzunluktaki vektörün yalnızca bir elemanın 1 diğerlerini 0 olması gereklidir. Öte yandan training_dataset_y iki boyutlu bir dizidir. Bu dizinin birinci boyutu len(sentences) uzunlığında ikinci boyutu ise -one hot encoding nedeniyle- len(chars)

uzunluğundadır. Mademki bu iki dizinin tüm elemanları np.zeros fonksiyonuyla sıfır yapılmıştır. O halde biz yalnızca uygun elemanı 1 yaparak "one hot encoding" vektörünü elde edebiliriz.

```
for i, sentence in enumerate(sentences):
    for k, char in enumerate(sentence):
        training_set_x[i, k, char_dict[char]] = 1
        training_set_y[i, char_dict[next_char[i]]] = 1
```

Artık trainig_set_x ve training_set_t "one hot encoding" biçiminde oluşturmuş durumdadır. Şimdi artık LSTM modelini oluşturabiliriz:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

model = Sequential()
model.add(LSTM(128))
model.add(Dense(len(chars), activation='softmax'))
model.compile('RMSprop', loss='categorical_crossentropy')
```

Burada geri beslemeli bir LSTM katmanın kullanımlığını görüyorsunuz. Çıktı katmanı toplamda len(chars) uzunlukta nörondan oluşmaktadır. Çıktı katmanın aktivasyon fonksiyonu "softmax" olduğuna göre tüm çıktı nöronlarının toplam değeri 1 olacaktır. Tabii biz bunlar arasından en yüksek değeri tahmin olarak alabiliriz. Ağımızın optimizasyon algoritması "RMSprop" olarak loss fonksiyonu da daha önceden benzer sınıflandırma örneklerinde yapmış olduğumuz gibi "categorical_crossentropy" olarak alınmıştır. Şimdi artk eğitim yapılabilir:

```
epochs = 20
history = model.fit(training_dataset_x, training_dataset_y, batch_size=128, epochs=epochs)
```

Analiz amaçlı eğitimi aşağıdaki gibi tek tek de yapabiliriz:

```
for i in range(epochs):
    history = model.fit(training_dataset_x, training_dataset_y, batch_size=128, epochs=1)
```

Şimdi de belli bir 60'luk yazı alıp bir döngü içerisinde ona 400 karakter ekleyerek yeni bir yazı üretelim. Bu işlemi şöyle yapabiliriz: 60'luk yazıyı verip bundan edilecek 61'inci karakteri tahmin ederiz. Sonra son 60'luk yazıyı verip sonraki karakteri tahmin ederiz. Böylece 400 karakteri bir döngü içerisinde bu biçimde devam ederiz. Örneğimizde ağ tarafından elde edilen karakterlerin olasılık dağılımına göre biraz değiştirilmesi gerekmektedir. Yani üretilen karakterin değil de ona yakın olan bir karakterin seçilmesi tek düzeliği engellemek bakımından uygun olur. Bunun için aşağıdaki gibi bir fonksiyondan faydalana bilir:

```
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)

    return np.argmax(probas)
```

Programın yazı üretim kısmı da şöyle olabilir:

```
def str_to_ohe(s):
    ohe = np.zeros(shape=(1, maxlen, len(chars)))
    for index, char in enumerate(s):
        ohe[0, index, char_dict[char]] = 1.0

    return ohe

r = np.random.randint(len(sentences) - maxlen - 1)
```

```

random_initial_sentence = text[r: r + 60]

generated_text = random_initial_sentence
for i in range(400):
    ohe = str_to_ohe(generated_text[i: maxlen + i])
    result = model.predict(ohe)[0]
    char = chars[sample(result, 0.2)]
    generated_text += char

print('[{}]{!s}'.format(generated_text[:maxlen], generated_text[maxlen:]))

```

Buradaki tüm programın çıktısını yeniden verelim:

```

import tensorflow.keras

path = tensorflow.keras.utils.get_file('nietzsche.txt', origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')

text = open(path).read().lower()

sentences = []
next_chars = []
maxlen = 60
step = 3

for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i:i + maxlen])
    next_chars.append(text[i + maxlen])

print('Number of sequences: {}'.format(len(sentences)))

chars = sorted(list(set(text)))
print('Number of unique chars: {}'.format(len(chars)))

char_dict = dict((char, index) for index, char in enumerate(chars))

import numpy as np

training_dataset_x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.int8)
training_dataset_y = np.zeros((len(sentences), len(chars)), dtype=np.int8)

for i, sentence in enumerate(sentences):
    for k, char in enumerate(sentence):
        training_dataset_x[i, k, char_dict[char]] = 1
    training_dataset_y[i, char_dict[next_chars[i]]] = 1

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

model = Sequential()
model.add(LSTM(128))
model.add(Dense(len(chars), activation='softmax'))
model.compile('RMSprop', loss='categorical_crossentropy')

epochs = 20
hist = model.fit(training_dataset_x, training_dataset_y, batch_size=128, epochs=epochs)

import matplotlib.pyplot as plt

plt.title('Epoch - Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Loss'

```

```

plt.plot(range(1, len(hist.epoch) + 1), hist.history['loss'])
plt.legend(['loss', 'val_loss'])

def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)

    return np.argmax(probas)

def str_to_ohe(s):
    ohe = np.zeros(shape=(1, maxlen, len(chars)))
    for index, char in enumerate(s):
        ohe[0, index, char_dict[char]] = 1.0

    return ohe

r = np.random.randint(len(text) - maxlen - 1)
random_initial_sentence = text[r: r + 60]

generated_text = random_initial_sentence
for i in range(400):
    ohe = str_to_ohe(generated_text[i: maxlen + i])
    result = model.predict(ohe)[0]
    char = chars[sample(result, 0.2)]
    generated_text += char

print('[{}]{!}'.format(generated_text[0:maxlen], generated_text[maxlen:]))

```

Yukarıdaki örnekte word embedding uygulanmamıştır. Bu örneğin word embedding uygulanmış halini de aşağıda veriyoruz. Anımsanacağı gibi Keras'taki Embedding katmanı girdi olarak 3 boyutlu değil 2 boyutlu indekslerden oluşan bir matris istemektedir. Çıktı olarak one hot encoding kullanılmaktadır.

```

import tensorflow.keras

path = tensorflow.keras.utils.get_file('nietzsche.txt', 'https://s3.amazonaws.com/text-dataset')
text = open(path).read().lower()

sentences = []
next_chars = []
maxlen = 60
step = 3

for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i:i + maxlen])
    next_chars.append(text[i + maxlen])

chars = sorted(list(set(text)))
char_dict = dict((char, index) for index, char in enumerate(chars))

import numpy as np

training_dataset_x = np.zeros((len(sentences), maxlen), dtype='int8')
training_dataset_y = np.zeros((len(sentences), len(chars)), dtype='int8')

for i, sentence in enumerate(sentences):
    for k, char in enumerate(sentence):
        training_dataset_x[i, k] = char_dict[char]
    training_dataset_y[i, char_dict[next_chars[i]]] = 1

```

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Dropout, LSTM, Dense

model = Sequential()
model.add(Embedding(maxlen, 64))
model.add(Dropout(0.2))
model.add(LSTM(128))
model.add(Dropout(0.2))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(len(chars), activation='softmax'))

epochs = 1

model.compile(optimizer='RMSprop', loss='categorical_crossentropy')
hist = model.fit(training_dataset_x, training_dataset_y, batch_size=128, epochs=epochs)

def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)

    return np.argmax(probas)

def str_to_indexes(s):
    a = np.zeros((1, len(s)), dtype='int8')
    for index, char in enumerate(s):
        a[0, index] = char_dict[char]

    return a

import matplotlib.pyplot as plt

plt.title('Epoch - Loss Graph')
plt.xlabel = 'Epoch'
plt.ylabel = 'Loss'
plt.plot(range(1, len(hist.epoch) + 1), hist.history['loss'])
plt.legend(['loss', 'val_loss'])

r = np.random.randint(len(text) - maxlen - 1)
random_initial_sentence = text[r:r + maxlen]

generated_text = random_initial_sentence
for i in range(400):
    a = str_to_indexes(generated_text[i:i + maxlen])
    result = model.predict(a)[0]
    generated_text += chars[sample(result, 0.2)]

print('[{}]{}, generated_text[maxlen:]))
```

Keras'ta Modellerin Fonksiyonel Olarak Oluşturulması

Şimdiye kadar biz Keras'ta Sequential bir model nesnesi yarattık ve bu model nesnesine katmanları add metoduyla ekledik. Model sınıfının add metodu bir önceki katmanın çıktısını bir sonraki katmana girdi yapmaktadır. Zaten buradaki Sequential sözcüğü de bunu anlatmaktadır. Bu biçimde model oluşturmak pek çok problem için yeterliyse de bazı tarz problemlerde yetersiz kalabilmektedir. Örneğin bazı modellerin girdileri birden fazla olabilmektedir. Benzer biçimde bazı modellerin çıktıları da yine birden fazla olabilmektedir. Halbuki Sequential modelde tek girdi ve

tek çıktı vardır. Örneğin biz yazıyı hem sınıflandıracak olalım hem de yazının eleştiri derecesini tespit edecek olalım. Buradaki çıktı bir tane değildir, iki tanedir. Tabii ilk akla gelen şey modelleri ayrı ayrı kurmak ve iki ayrı çıktıyı ayrı ayrı elde etmektir. Fakat böylesi bir çaba fazladan programlama yükü oluşturmaktadır. Çünkü bu çıktılar için eğitimin yeniden yapılması gereklidir. Ayrıca oluşan ağırlık değerlerinin yeniden saklanması da gerekmektedir. Oysa tek bir modelle iki ya da daha fazla çıktı elde edilebilmektedir. Benzer biçimde örneğin biz bir yazıyı hem sınıflandırmak istediğimizi hem de yazının diline bakılarak onun hangi tarihte yazıldığını belirlemek istediğimizi düşünelim. Bu örnekte de iki çıktı fakat tek bir girdi vardır. Bu iki çıktı için iki ayrı model oluşturulabilir fakat tek bir modelle de bu işlem yapılabilir. Girdilerin de birden fazla olması durumuyla karşılaşılabilmektedir. Örneğin birinci girdi olarak yazının kendisini, ikinci girdi olarak da yazının yazarına ilişkin bilgileri kullanabiliriz. Buradan da yazıyı sınıflandırmak isteyebiliriz. Bu tür çok girdili modelleri tek girdili hale getirmek için bazen verilerin birleştirilmesi yoluna gidilebilmektedir. Ancak bu örnekte olduğu gibi pek çok modelde farklı veriler girdi katmanının farklılığından dolayı tek bir girdi olarak birleştirilememektedir. İşte Keras'ın Sequential modeli böyle çoklu girdi ve çoklu çıktı üzerinde işlem yapamamaktadır. Bu işlemlerin yapılabilmesi için Sequential model yerine modelin fonksiyonel biçimde oluşturulması gerekmektedir.

Keras'ta aslında Dense gibi LSTM gibi katman sınıflarının metod çağrıma operatör metodları (`__call__` metodları) yazılmıştır. Bu metod çağrıma metodları katmanın girdisini oluşturmak ve katmanları birbirlerine bağlamak için kullanılmaktadır. Örneğin aşağıdaki gibi bir kod parçası olsun:

```
inp = Input(shape=(28 * 28,))
a = Dense(512, activation='relu', name='Hidden-1')(inp)
b = Dense(256, activation='relu', name='Hidden-2')(a)
c = Dense(256, activation='relu', name='Hidden-3')(b)
d = Dense(256, activation='relu', name='Hidden-4')(c)
e = Dense(10, activation='softmax', name='Output')(d)
```

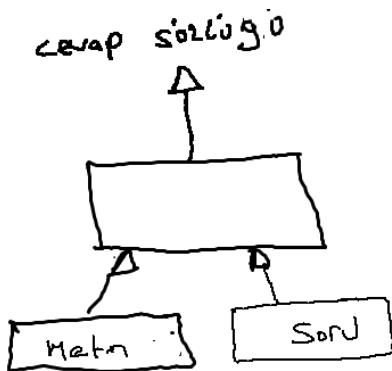
Burada Dense fonksiyonu bir Dense türünden bir sınıf nesnesi yaratmaktadır. Bu sınıf nesnesi ile metod çağrıma operatörü kullanıldığından aslında bir önceki katmanın girdileri bu katmana bağlanmış olur. Dolayısıyla yukarıdaki kod parçasında Input katmanı aslında birinci Dense katmanına girdi yapılmıştır. Diğer Dense katmanları da diğerlerine girdi yapılmıştır. Toplamda bu kod parçası Sequential sınıfının yaptığı şeyi yapmaktadır. Başka bir deyişle bu işlemin Sequential eşdeğeri şöyledir:

```
model = Sequential()
model.add(Dense(512, input_dim=28 * 28, activation='relu', name='Hidden-1'))
model.add(Dense(256, activation='relu', name='Hidden-2'))
model.add(Dense(256, activation='relu', name='Hidden-3'))
model.add(Dense(256, activation='relu', name='Hidden-4'))
model.add(Dense(10, activation='softmax', name='Output'))
```

Her ne kadar yukarıdaki iki kod parçası işlevsel bakımından eşdeğerde de biz birinci kod parçasında artık Sequential sınıfından kurtulmuş olduk. Böylece bu yöntemle Sequential sınıfı ile yapamadığımız bazı şeyleri de yapar hale gelebileceğiz.

Fonksiyonel Modelde Çok Girişli Bir Örnek

Daha önceden de belirtildiği gibi fonksiyonel model bizim çok girişli ve çok çıkışlı modelleri kurabilmemize olanak sağlamaktadır. Çok girişli model eğitimin birden fazla veri kümesi kullanılarak yapıldığı modeldir. Fonksiyonel modelde birden fazla giriş Input isimli sınıfla katmansal olarak ayrı ayrı oluşturulur. Sonra bu girişler concatenate fonksiyonuyla tek bir giriş biçiminde birleştirilir. Ondan sonra model normal olarak devam ettirilir. Bunun için şöyle bir örnek verilebilir: Modelde birtakım metinler ve bu metinlere ilişkin sorular vardır. Yani yabancı dil sınavlarında olduğu kişiye önce bir metin verilmekte daha sonra da bir soru verilmektedir. Bu sorunun o metne bakılarak cevaplandırılması istenmektedir. Sorunun cevabı da tek sözcüktür.



Böyle bir örnekte fonksiyonel modelde iki girdi verisi olacaktır. Bu iki girdi verisi Input sınıfı ile temsil edilmelidir. Sonra metin işlemlerinde olduğu gibi bir Embedding katmanı ve bir de LSTM katmanı girdilere eklenir. Örneğin:

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Embedding, LSTM, concatenate, Dense
from tensorflow.keras import Input

text_vocabulary_size = 10000
question_vocabulary_size = 10000
max_length = 100
answer_vocabulary_size = 500

input_text = Input(shape=(max_length,), dtype='int32', name='text')
embedded_text = Embedding(text_vocabulary_size, 64)(input_text)
lstm_text = LSTM(32)(embedded_text)

input_question = Input(shape=(max_length,), dtype='int32', name='question')
embedded_question = Embedding(question_vocabulary_size, 64)(input_question)
lstm_question = LSTM(16)(embedded_question)
```

Bu iki girdi katmanının sanki tek bir girdi katmanı varmış gibi birleştirilmesi gerekmektedir. Bu işlem concatenate fonksiyonu ile yapılmaktadır:

```
concatenated = concatenate([lstm_text, lstm_question], axis=1)
```

Artık girdiler birleştirildiği için model sanki tek bir girdiye sahipmiş gibi devam ettirilir. Örneğin modele bir tane Dense hidden katman ekleyelim:

```
dense = Dense(128, activation='relu')(concatenated)
```

Çıktı katmanı toplamda answer_vocabulary_size kadar nörondan oluşacaktır:

```
output = Dense(answer_vocabulary_size, activation='softmax')(dense)
```

Modelin derlenmesi şöyle yapılabilir:

```
model = Model([input_text, input_question], output)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['acc'])
```

Eğitim işlemini rastgele verilerle temsili olarak yapalım:

```
import numpy as np
sample_size = 1000
training_set_x_text = np.random.randint(0, text_vocabulary_size, size=(sample_size, max_length))
```

```

training_set_x_question = np.random.randint(0, question_vocabulary_size, size=(sample_size,
max_length))

training_set_y = np.zeros(shape=(sample_size, answer_vocabulary_size))
for i in range(sample_size):
    training_set_y[i, np.random.randint(0, answer_vocabulary_size)] = 1.0

history = model.fit([training_set_x_text, training_set_x_question], training_set_y, epochs=10,
batch_size=128)

```

Burada fit metodunun girdi vektörünün bir listeden oluştuğuna dikkat ediniz. Bu liste asıl metinden ve soru metninden oluşmaktadır. Eğer Input sınıflarında girdilere isim verilmişse fit metodunda girdi kümeleri liste ile değil sözlük ile de belirtilebilir. Örneğin:

```

history = model.fit({'text': training_set_x_text, 'question': training_set_x_question},
training_set_y, epochs=10, batch_size=128)

```

Şimdi de rastgele değerler üreterek predict işlemini yapalım:

```

predict_set_x_text = np.random.randint(0, text_vocabulary_size, size=(1, max_length))
predict_set_x_question = np.random.randint(0, question_vocabulary_size, size=(1, max_length))

predict_result = model.predict([predict_set_x_text, predict_set_x_question])[0]

# predict_result = model.predict({'text': predict_set_x_text, 'question':
predict_set_x_question})[0]

result = np.argmax(predict_result)
print(result)

```

Gördüğü gibi predict işleminde de girdi vektörü bir liste biçiminde verileceği gibi bir sözlük biçiminde de verilebilmektedir. Yukarıdaki tüm kod aşağıda yeniden verilmektedir:

```

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Embedding, LSTM, concatenate, Dense
from tensorflow.keras import Input

text_vocabulary_size = 10000
question_vocabulary_size = 10000
max_length = 100
answer_vocabulary_size = 500

input_text = Input(shape=(max_length,), dtype='int32', name='text')
embedded_text = Embedding(text_vocabulary_size, 64)(input_text)
lstm_text = LSTM(32)(embedded_text)

input_question = Input(shape=(max_length,), dtype='int32', name='question')
embedded_question = Embedding(question_vocabulary_size, 64)(input_question)
lstm_question = LSTM(16)(embedded_question)

concatenated = concatenate([lstm_text, lstm_question], axis=1)

dense = Dense(128, activation='relu')(concatenated)
output = Dense(answer_vocabulary_size, activation='softmax')(dense)

model = Model([input_text, input_question], output)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['acc'])

import numpy as np

sample_size = 1000

```

```

training_set_x_text = np.random.randint(0, text_vocabulary_size, size=(sample_size,
max_length))
training_set_x_question = np.random.randint(0, question_vocabulary_size, size=(sample_size,
max_length))

training_set_y = np.zeros(shape=(sample_size, answer_vocabulary_size))
for i in range(sample_size):
    training_set_y[i, np.random.randint(0, answer_vocabulary_size)] = 1.0

history = model.fit([training_set_x_text, training_set_x_question], training_set_y, epochs=10,
batch_size=128)

#history = model.fit({'text': training_set_x_text, 'question': training_set_x_question},
training_set_y, epochs=10, batch_size=128)

predict_set_x_text = np.random.randint(0, text_vocabulary_size, size=(1, max_length))
predict_set_x_question = np.random.randint(0, question_vocabulary_size, size=(1, max_length))

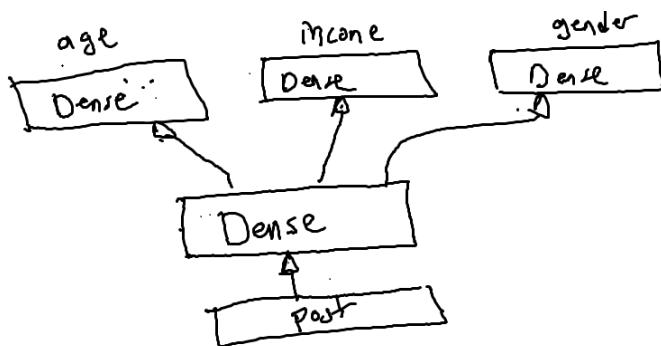
predict_result = model.predict([predict_set_x_text, predict_set_x_question])[0]
# predict_result = model.predict({'text': predict_set_x_text, 'question':
predict_set_x_question})[0]
result = np.argmax(predict_result)
print(result)

```

Fonksiyonel Modelde Çok Çıkışlı Bir Örnek

Bir modelin birden fazla çıktısı olabilir. Bu durumda çıktılar için ayrı Dense katmanları oluşturulur. Sonra bu katmanlar fit işleminde ayrı ayrı belirtilir. Buradaki önemli bir nokta üç farklı çıkışın skalalarının farklı olabilmesidir. Elde edilen modelin ağırlıklarını güncellenirken kullanılan optimizasyon algoritmaları tek bir loss değerine bakmaktadır. O halde üretilen çıktıların tek bir loss değeri elde edilmek üzere birleştirilmesi gerekmektedir. Keras bu işlemi de arka planda otomatik olarak yapar.

Örneğin bir sosyal medya gönderisinden kişinin gelir düzeyini, yaşını ve cinsiyetini tespit etmeye çalışacağız. Görüldüğü gibi bu örnekte üç farklı çıktı değeri vardır. Gelir düzeyi 10 farklı grup biçiminde kategorik ifade edilmektedir. Yaş sıralı bir ölçek olarak değerlendirilmiştir. Cinsiyet ise Kadın ve Erkek olmak üzere kategorik bir değişkendir.



Bu model aşağıdaki örnekte olduğu gibi gerçekleştirilebilir:

```

from tensorflow.keras.models import Model
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras import Input

post_vocabulary_size = 10000
max_length = 100
income_groups = 10

input_post = Input(shape=(max_length,), dtype='int32', name='posts')
embedded_post = Embedding(post_vocabulary_size, 64)(input_post)

```

```

lstm_post = LSTM(32)(embedded_post)
dense_post = Dense(128, activation='relu')(lstm_post)

age_post_output = Dense(1, activation='linear', name='age')(dense_post)
income_post_output = Dense(10, activation='softmax', name='income')(dense_post)
gender_post_output = Dense(2, activation='softmax', name='gender')(dense_post)

model = Model(input_post, [age_post_output, income_post_output, gender_post_output])
model.compile(optimizer='rmsprop', loss=['mse', 'categorical_crossentropy',
'categorical_crossentropy'], loss_weights=[0.25, 1., 10.])

import numpy as np

sample_size = 1000

training_set_x_post = np.random.randint(0, post_vocabulary_size, size=(sample_size,
max_length))

training_set_y_age = np.random.randint(13, 70, size=(sample_size, 1))

training_set_y_income = np.zeros(shape=(sample_size, income_groups))
training_set_y_gender = np.zeros(shape=(sample_size, 2))

for i in range(sample_size):
    training_set_y_income[i, np.random.randint(0, income_groups)] = 1.0
    training_set_y_gender[i, np.random.randint(0, 2)] = 1.0

history = model.fit(training_set_x_post, [training_set_y_age, training_set_y_income,
training_set_y_gender], epochs=10, batch_size=128)

#history = model.fit(training_set_x_post, {'age': training_set_y_age, 'income':
training_set_y_income, 'gender': training_set_y_gender}, epochs=10, batch_size=128)

predict_set_x_post = np.random.randint(0, post_vocabulary_size, size=(1, max_length))
predict_result = model.predict(predict_set_x_post)

print('Age: {}'.format(predict_result[0]))
print('Income group: {}'.format(np.argmax(predict_result[1])))
print('Gender: {}'.format(np.argmax(predict_result[2])))

```

DENETİMSİZ ÖĞRENME (Unsupervised Learning)

Biz şimdiye kadar yapay sinir ağlarını kullanarak denetimli öğrenme (supervised learning) konularını inceledik. Gerçekten de yapay sinir ağları denetimli öğrenmede en yaygın kullanılan araçlardan biridir. Kursumuzun bu bölümünde ise denetimsiz (unsupervised) öğrenme konuları ele alınacaktır.

Denetimsiz öğrenmede bir eğitim faaliyeti yoktur. Eğitim olmayınca örnek çıktı kümesi de bulunmamaktadır. Genellikle çıktıların nasıl olacağı önceden de bilinmemektedir. Kursuzun giriş kısımlarında da belirttiğimiz gibi "makine öğrenmesi (machine learning)" denildiğinde genellikle üç öğrenme yönteminden bahsedilmektedir:

- 1) Denetimli Öğrenme (supervised learning)
- 2) Denetimsiz Öğrenme (unsupervised learning)
- 3) Pekiştirmeli Öğrenme (Reinforcement learning)

Denetimsiz öğrenme çeşitli istatistiksel yöntemlerle ve yapay sinir ağlarıyla gerçekleştirilebilmektedir. Biz de kursumuzun bu bölümünde bu tekniklerden önemli olanlarını inceleyeceğiz.

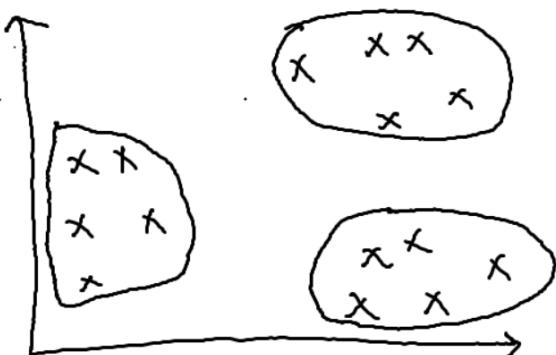
Denetimli öğrenme ile denetimsiz öğrenme arasındaki en belirgin farklılık denetimli öğrenmede bir sonuç verisi olduğu halde denetimsiz öğrenmede böyle bir sonuç verisinin olmamasıdır. Yani örneğin biz denetimli öğrenmede ağı

eğitmek için `training_dataset_x` ve `training_dataset_y` veri kümelerini kullandık. Halbuki denetimsiz öğrenmede elimizde yalnızca `training_dataset_x` kümesi bulunmaktadır.

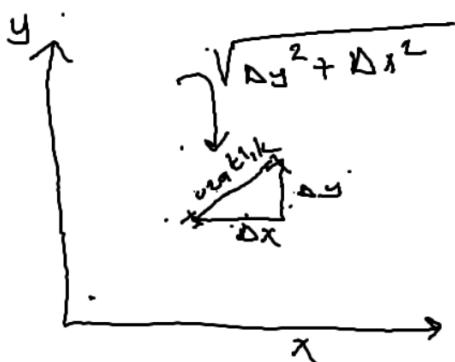
Kümeleme İşlemleri (Clustering)

Denetimsiz öğrenme yaklaşımının en önemlilerinden biri kümeleme (clustering) işlemleridir. Kümeleme nesnelerin ya da olguların özelliklerine bakılarak birbirlerine benzeyenlerin bir araya getirilmesi (kümelendirme) anlamına gelmektedir. Kümeleme işlemleri için çeşitli yöntemler (ya da algoritmalar) kullanılmaktadır. "Sınıflandırma (classification)" ve "kümeleme (clustering)" terimleri istatistikte ve dolayısıyla makine öğrenmesinde farklı anlamlara gelmektedir. Sınıflandırma bir olgunun daha önce belirlenmiş sınıflar içerisinde hangisinin içerisinde girebileceği ile ilgilidir. Bu anlamda sınıflandırma "lojistik regresyon" ile eşdeğerdir. Halbuki kümeleme olguların benzerliklerine göre bir araya getirilmesi ya da ayırtılmasının işlemidir. Sınıflandırma "eğitimli öğrenme" yöntemlerinden olduğu halde kümeleme "eğitsiz öğrenme" yöntemlerindendir.

Kümeleme işlemlerinde birbirlerine benzeyenleri nasıl belirleyebiliriz? Benzemek fiili insan sezgisi ile ilgilidir. Makinelerin böyle sezgileri olmadığına göre somut niceliklerin kullanılması gerekmektedir. Birbirlerine en yakın olanlar kendi aralarında kümelenebilirler. Bu durumda da yakınlık-uzaklık kavramının sayısal biçimde ifade edilmesi gereklidir. Bu ifade aslında basittir. Örneğin koordinat ekseninde nesneler ya da olgular noktalarla temsil edilirse bunların aralarındaki uzaklık çok belirgin hale gelir.



Burada gözle baktığımızda noktaların kendi aralarında üç grup oluşturduğunu görüyoruz. Birbirlerine yakın olan noktalar aynı grup içerisinde bulunmaktadır. Koordinat ekseninde yakınlık (ya da uzaklık) iki nokta arasındaki uzaklığı ifade edilmektedir. Bu klasik uzaklık kavramına "Öklit uzaklığı (Euclidian distance)" denilmektedir. İki nokta arasındaki öklit uzaklığı Pisagor teoreminden kolaylıkla elde edilebilir:



Lineer cebirde aslında iki boyutlu düzlem ile n boyutlu uzay arasında işlemlerin uygulanma biçimini bakımından bir farklılık yoktur. Yani iki boyutlu düzlem için söz konusu olan her şey üç boyutlu, dört boyutlu genel olarak n boyutlu uzay için de geçerlidir. Örneğin üç boyutlu uzayda Öklit uzaklığı benzer biçimde aşağıdaki formül ile hesaplanabilir:

$$\sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}$$

iki boyutlu düzlem ile n boyutlu uzay arasında genel işlemler bakımından bir farklılık olmadığı için konu anlatımları insan algısına yakınlık nedeniyle genellikle iki boyutlu düzlem üzerinde yapılmaktadır. Şüphesiz n boyutlu uzay aslında n tane sütuna sahip olan bir veri tablosu anlamına gelmektedir.

Aslında değişik uygulamalar için farklı uzaklık (distance) kavramları da kullanılmaktadır. Ancak Öklid uzaklıği kümelemede en çok tercih edilen uzaklıklardan biridir.

Şimdi n boyutlu uzay için (yani n tane sütunu olan nicelikselleştirilmiş bir olgu) iki nokta arasındaki uzaklıği hesaplayan bir Python fonksiyonu yazalım:

```
import numpy as np

def euclidian_distance(a, b):
    return np.sqrt(np.sum((a - b) ** 2))
```

Elimizde bir olguya ilişkin n tane örnek olsun ve bu olgu k tane nicelikle temsil ediliyor olsun. Biz bu örnekleri bir ndarray haline getirirsek bu ndarray nesnesi n satırlı k tane sütunlu bir matris biçiminde olacaktır. Bu matriste de satırlar aslında k boyutlu uzayda noktaları temsil edecektir. Böylece iki nota arasındaki (yani iki satır arasındaki) uzaklık yukarıdaki fonksiyonla hesaplanabilecektir.

K-means Kümeleme Yöntemi

Yukarıda da belirtildiği gibi kümeleme işlemleri için farklı yöntemler yani algoritmalar kullanılmaktadır. Bunların arasında avantaj-dezavantaj bakımından farklılıklar vardır. En yaygın kullanılan kümeleme yöntemi K-means denilen yöntemdir.

K-means yönteminde işin başında verilerin kaç kümeye ayrılacağı tespit edilmiş olmalıdır. Örneğin bu değerin k olduğunu varsayıyalım. Algoritmada olguları nokta olarak ifade edeceğiz. Algoritmanın işleyişi şöyledir:

- 1) Önce k tane küme için (yani her küme için) rastgele k tane ağırlık merkezi (centroid) alınır.
- 2) Tüm noktaların bu ağırlık merkezlerine olan uzaklıkları hesaplanır. Bir nokta hangi ağırlık merkezine yakınsa o kümeye dahil edilir. Böylece bu adımda ilk kümeler oluşturulmuş olur.
- 3) Kümelerin ağırlık merkezleri bu kümelere atanmış noktalar kullanılarak yeniden hesaplanır. Ağırlık merkezi hesaplaması ortalama yoluyla yapılmaktadır. Yani o kümedeki tüm noktaların kendi aralarında ortalaması bulunur. (Örneğin 3 boyutlu uzyda bu ortalama kümedeki tüm noktaların x'lerinin, y'lerinin ve z'lerinin kendi aralarında ortalaması bulunarak hesaplanmaktadır.) Böylece yeni ve daha uygun bir ağırlık merkezi oluşturulmuş olur.
- 4) Bundan sonra yine tüm noktalar yeni ağırlık merkezlerine uzaklıklarına göre yeniden kümelenir. Bu kümeleme bir öncekinden farklı olabilmektedir. (Yani yeni ağırlık merkezleri dikkate alındığında bazı noktalar kümeye değiştirebilmektedir.)
- 5) Yeni kümelerin ağırlık merkezleri yeniden ortalama yöntemiyle bulunur. Ve işlemler böyle devam ettirilir.
- 6) Yeniden kümeleme sonucunda önceki kümelerle yeni kümeler arasında eleman bakımından hiç fark yoksa artık işlemler sonlandırılır. Yani artık hiçbir nokta kümeye değiştirmiyorsa algoritmaya devam etmenin bir anlamı yoktur.

Algoritmanın işleyişini kartezyen koordinat sisteminde noktalarla açıklamaya çalışalım. Örneğimizde toplam 12 nokta olsun. Bu 12 nokta iki kümeye kümelenmek istensin. Başlangıçtaki noktalar şöyledir:

Sıra	X	Y
1	7	8
2	2	4
3	6	4
4	3	2

5	6	5
6	5	7
7	3	3
8	1	4
9	5	4
10	7	7
11	7	6
12	2	1

İki küme için rastgele ağırlık merkezleri şöyle olsun: $C_1(1, 4)$, $C_2(7, 8)$, Şimdi bu 12 noktanın bu ağırlık merkezlerine Öklid uzaklıklarını hesaplayıp bu 12 noktayı iki kümeye dağıtalım:

Sıra	X	Y	C1 Uzaklığı	C2 Uzaklığı	Atanan Küme
1	7	8	7.21	0	C2
2	2	4	1.00	6.40	C1
3	6	4	5.00	4.12	C2
4	3	2	2.83	7.21	C1
5	6	5	5.10	3.16	C2
6	5	7	5.00	2.24	C2
7	3	3	2.24	6.40	C1
8	1	4	0.00	7.21	C1
9	5	4	4.00	4.47	C1
10	7	7	6.71	1.00	C2
11	7	6	6.32	2.00	C2
12	2	1	3.16	8.60	C1

Bu işlemin sonucunda C1 kümesinde $\{2, 4, 7, 8, 9, 12\}$, C2 kümesinde ise $\{1, 3, 5, 6, 10, 11\}$ noktaları bulunacaktır. Şimdi yeni ağırlık merkezlerini hesaplayalım. Bu hesaptan yeni ağırlık merkezleri $c1 = [2.67, 3.00]$ ve $c2 = [6.33, 6.17]$ biçiminde elde edilir. Şimdi bu noktaların hepsinin yeni ağırlık merkezlerine göre uzaklıklarını hesaplayarak yeniden kümeleme yapmamız gereklidir.

Sıra	X	Y	C1 Uzaklığı	C2 Uzaklığı	Atanan Küme
1	7	8	6.61	1.95	C2
2	2	4	1.20	4.84	C1
3	6	4	3.48	2.19	C2
4	3	2	1.05	5.34	C1
5	6	5	3.88	1.22	C2
6	5	7	4.63	1.57	C2
7	3	3	0.33	4.60	C1
8	1	4	1.95	5.75	C1
9	5	4	2.54	2.55	C1
10	7	7	5.89	1.07	C2
11	7	6	5.27	0.69	C2
12	2	1	2.11	6.74	C1

Burada kümelerdeki elemanlar değişmediği için artık algoritmaya son verilecektir.

K-means algoritmasında ilk başta alınan rastgele noktalardan dolayı algoritmanın farklı çalışmalarında farklı kümeler bulunabilemektedir. Burada genellikle izlenen yol algoritmayı n defa çalışırmak ve bu n çalışmadan elde edilen kümeleri karşılaştırarak en iyi olanını bulmaktadır. Pekiyi farklı kümeleri neye göre karşılaştırmalıyız. İşte burada izlenen yol her kümeyin noktalarının kendi ağırlık merkezlerine uzaklıklarının toplamı en küçük olan kümeyi tercih etmektedir. Yani bu durum toplam varyansın daha düşük olması anlamına gelmektedir.

Şimdi biz Python'da K-means algoritmasını bir fonksiyon olarak kendimiz yazalım. Fonksiyonumuzun parametrik yapısı şöyle olsun:

```
kmeans(dataset, nclusters, centroids=None)
```

Fonksiyonun birinci parametresi olan dataset k tane sütundan n tane satırda oluşan noktaların kümesini belirtmektedir. Örneğin karztezyen iki boyutlu koordinat sistemi için sütun sayısı 2 olacaktır. İkinci parametre toplam kaç kümenin kullanılacağını belirtir. Üçüncü parametre default None değerini almıştır. Programcı isterse başlangıç centroid değerlerini kendisi verebilir. Eğer bu değerleri kendisi vermezse rastgele olarak alınacaktır. centroids dizisi iki boyutludur. Bunun sütun sayısı dataset'in sütun sayısı kadar, satır sayısı ise nclusters kadar olmalıdır. Fonksiyonumuzda işin başında her küme için rastgele bir ağırlık merkezinin atanması gerekmektedir. Bu işlemi yapan aşağıdaki gibi bir fonksiyon yazılmıştır:

```
def rand_centroids(dataset, nclusters):
    ncols = dataset.shape[1]
    centroids = np.zeros((nclusters, ncols))
    for j in range(ncols):
        minj = min(dataset[:, j])
        maxj = max(dataset[:, j])
        rangej = maxj - minj
        centroids[:, j] = minj + rangej * np.random.rand(nclusters)

    return centroids
```

Burada dataset matrisinin sütunları tek tek ele alınmış sütunların en küçük ve en büyük değerleri arasında rastgele değerlerden noktalar oluşturulmuştur. Bu fonksiyon kmeans tarafından çağrılacaktır. Şimdi de kmeans fonksiyonunu yazalım:

```
import numpy as np

def euclidean_distance(a, b):
    return np.sqrt(np.sum((a - b) ** 2))

def rand_centroids(dataset, nclusters):
    ncols = dataset.shape[1]
    centroids = np.zeros((nclusters, ncols))
    for j in range(ncols):
        minj = min(dataset[:, j])
        maxj = max(dataset[:, j])
        rangej = maxj - minj
        centroids[:, j] = minj + rangej * np.random.rand(nclusters)

    return centroids

def kmeans(dataset, nclusters, centroids=None):
    nrows = dataset.shape[0]
    clusters = np.full(nrows, -1)

    if centroids is None:
        centroids = rand_centroids(dataset, nclusters)

    change_flag = True
    while change_flag:
        change_flag = False
        for i in range(nrows):
            min_dist = np.inf
            min_index = -1
            for k in range(nclusters):
                dist = euclidean_distance(dataset[i], centroids[k])
                if dist < min_dist:
                    min_dist = dist
                    min_index = k

            clusters[i] = min_index
```

```

        min_index = k
    if clusters[i] != min_index:
        change_flag = True
    clusters[i] = min_index
for k in range(nclusters):
    if np.any(clusters == k):
        centroids[k] = np.mean(dataset[clusters == k], axis=0)

dataset_clusters = []
for i in range(nclusters):
    dataset_clusters.append(dataset[clusters == i])

inertias = []
for i in range(nclusters):
    inertias.append(np.sum((dataset[clusters == i] - centroids[i]) ** 2))

return clusters, dataset_clusters, centroids, inertias

```

Buradaki kmeans fonksiyonumuz dörtlü bir demete geri dönemektedir. Fonksiyonun geri döndürdüğü demetin ilk elemanı her bir noktanın hangi kümeye atandığını gösteren bir ndarray nesnesidir. İlk kümenin numarası 0'dan başlamaktadır. Geri döndürülen demetin ikinci elemanı hangi kümelerin hangi noktalara sahip olduğunu veren bir listedir. Bu listenin uzunluğu küme sayısı kadardır. Listenin her elemanı o kümeye ilişkin noktaları tutmaktadır. Tabii büyük uygulamalarda büyük dzilerin bu biçimde geri döndürülmesi uygun olmayabilir. Geri döndürülen demetin üçüncü elemanı kümelerin ağırlık merkezlerinin bulunduğu ndarray nesnesidir. Geri döndürülen demetin son elemanı ise nihai durumdaki noktaların kendi ağırlık merkezlerine uzaklıklarının kareleri toplamıdır. Yani istatistiksel bakımdan bu da kümeler içerisindeki varyans toplamları anlamına gelmektedir. Bu varyans toplamlarına bu bağlamda "atalet (inertia)" da denilmektedir. Daha önceden de belirtildiği gibi kmeans algoritması bir kez değil n kez çalıştırılmalı ve ev küçük atalet değerine sahip olan çalıştırmanın sonuçları nihai sonuç olarak ele alınmalıdır. Yukarıdaki kodun daha önce verdigimiz örnek değerlerle çeşitli kez çalıştırılmaları sonucunda aşağıdaki saçılım grafikleri elde edilmiştir.

```

points = np.loadtxt('points.csv', delimiter=',', dtype=np.float32)

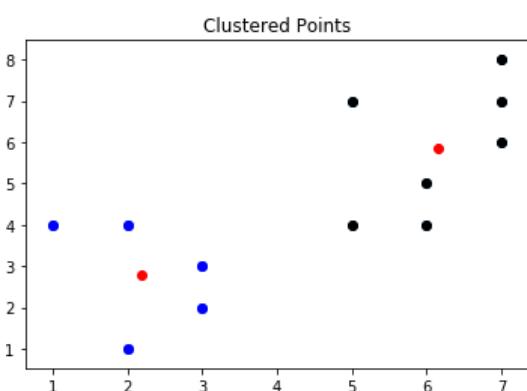
import matplotlib.pyplot as plt

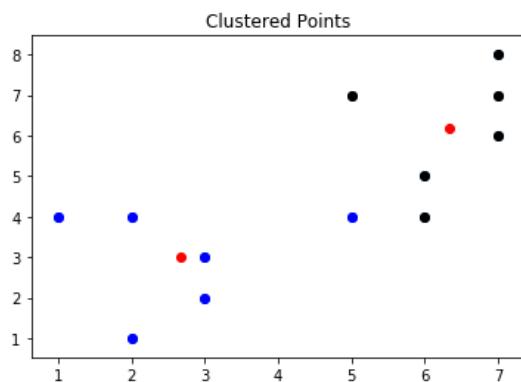
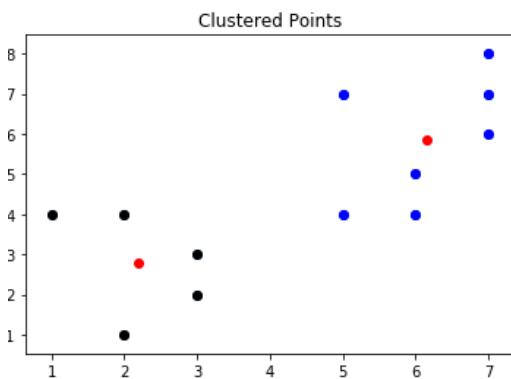
plt.title('Points')
plt.scatter(points[:, 0], points[:, 1])

clusters, dataset_clusters, centroids, inertias = kmeans(points, 2)

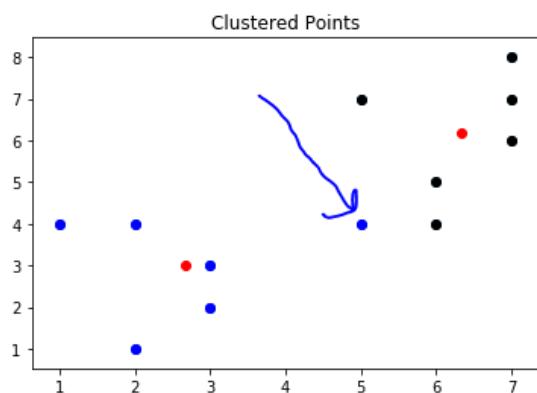
plt.title('Clustered Points')
plt.scatter(points[clusters == 0, 0], points[clusters == 0, 1], c='black')
plt.scatter(points[clusters == 1, 0], points[clusters == 1, 1], c='blue')
plt.scatter(centroids[:, 0], centroids[:, 1], c='red')

```





Algoritmanın her çalıştırılması sonucunda aynı noktaların aynı kümeler içerisinde bulunmayacağıne dikkat ediniz. Örneğimiz aşağıda gösterdiğimiz nokta farklılıklarında farklı kümelerde yer alabilmiştir:



Örnekte kullandığımız "points.csv" dosyasının içeriği şöyledir:

```
7,8
2,4
6,4
3,2
6,5
5,7
3,3
1,4
5,4
7,7
7,6
2,1
```

İşte yukarıda da belirtildiği gibi algoritmanın n kez çalıştırılıp en iyi değerin (toplam ataletin en düşük olduğu değerin) alınması tipik uygulanan yöntemdir. Aşağıda bu yöntem uygulanmıştır:

```
repeat_count = 20
```

```

min_inertia = np.inf

dataset = np.loadtxt('points.csv', delimiter=',', dtype=np.float32)

for i in range(repeat_count):
    clusters, dataset_clusters, centroids, inertias = kmeans(dataset, 2);
    total = np.sum(inertias)
    if total < min_inertia:
        min_inertia = total
        result_clusters = clusters
        result_dataset_clusters = dataset_clusters
        result_centroids = centroids
        result_inertias = inertias

print('Dataset: {}'.format(dataset))
print('Result clusters: {}'.format(result_clusters));
print('Result dataset clusters: {}'.format(result_dataset_clusters));
print('Result centroids: {}'.format(result_centroids));
print('Result inertias: {}'.format(result_inertias));

```

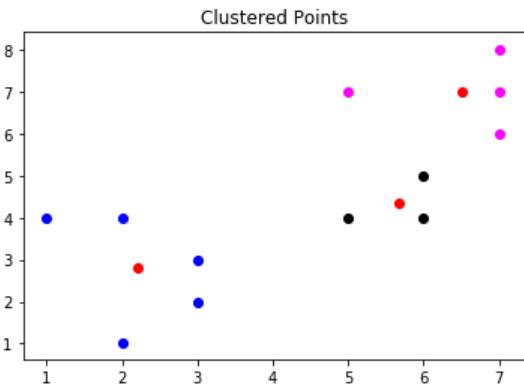
Biz yukarıdaki örnekte noktaları toplamda iki kümeye ayırdık. Pekiyi noktaların iki kümeye ayrılacağını nasıl biliyorduk? Aslına biz böyle bir bilgiye sahip değildik. Önce noktaların saçılma grafiğini çizdik. Bu grafikte noktaların kendi aralarında iki farklı toplaşmaya sahip olduğunu gördük. Tabii noktalar iki boyutlu düzlemede gözle kolay biçimde algılanabilmektedir. Ancak fazla özelliğe (yani sütuna) sahip olan veri kümelerinde böyle bir grafik çizme imkanımız yoktur. O halde biz elimizdeki veri kümelerindeki satırların kaç kümeye ayrılacağını işin başında nasıl belirleyebiliriz? İşte izleyen bölümlerde bu konu ele alınacaktır. Şimdi yukarıdaki örnekteki noktaları 3 kümeye ayırtıralım:

```

clusters, dataset_clusters, centroids, inertias = kmeans(points, 3)

plt.title('Clustered Points')
plt.scatter(points[clusters == 0, 0], points[clusters == 0, 1], c='black')
plt.scatter(points[clusters == 1, 0], points[clusters == 1, 1], c='blue')
plt.scatter(points[clusters == 2, 0], points[clusters == 2, 1], c='magenta')
plt.scatter(centroids[:, 0], centroids[:, 1], c='red')

```



Scikit-Learn KMeans Sınıfının Kullanımı

scikit-learn kütüphanesinde K-means kümeleme işlemleri sklearn.cluster modülündeki KMeans isimli sınıfı yapılabilmektedir. Sınıfın `__init__` parametreleri şöyledir:

```
KMeans(n_clusters=8, init='kmeans++', n_init=10, max_iter=300, tol=0.0001, precompute_distances='auto', verbose=0, random_state=None, copy_x=True, n_jobs=None, algorithm='auto')
```

Fonksiyonun `n_clusters` parametresi noktaların ayırtılacağı kümeye sayısını belirtmektedir. `n_init` parametresi en iyi değerin bulunması için uygulanacak çalışma sayısıdır. Bu parametre default 10 değerindedir. Yani sınıf default olarak rastgele ağırlık merkezleriyle 10 çalışma yapıp bunun en iyisini bize vermektedir. `max_iter` parametresi belki bir çözümde çözümün uzun süre bulunmaması durumunda işlemin kesileceği maksimum iterasyon sayısını

belirtmektedir. Bilindiği gibi K-Means algoritmasında her yinelemede (iterasyonda) kümelerde değişiklik olabilmektedir. Zaten kumelemede değişiklik olmadığı durumda sonuç elde edilmiş olacaktır. tol parametresi farklı çalışırmalardaki yakınsama değerlerinin (yani toplam inertia'nın) duyarlığını belirtmektedir. Yani bir çalıştırmanın değerinden iyi kabul edilmesi için toplam inertia'nın burada belirtilen değerden daha iyi olması gereklidir. Fonksiyonun diğer parametreleri scikit-learn dokümanlarından incelenebilir.

KMeans sınıfının fit isimli metodunu çalıştırır asıl metottur. fit metodunu bizden dataset'i parametre olarak alır yine KMeans nesnesinin kendisine geri döner. Örneğin:

```
dataset = np.loadtxt('points.csv', delimiter=',', dtype=np.float32)

from sklearn.cluster import KMeans

km = KMeans(n_clusters=2)
km.fit(dataset)
```

Mademki fit metodunu da aslında KMeans nesnesinin kendisine geri dönmektedir. O halde yukarıdaki işlem tek bir satırda şöyle de yapılabilir:

```
km = KMeans(n_clusters=2).fit(dataset)
```

Peki çözümün sonuçları nasıl elde edilecektir? İşte fit metodunu sonuçları KMeans sınıfının çeşitli örnek özniteliklerine yerleştirmektedir. Bu örnek öznitelikleri ve tuttuğu değerler şunlardır:

labels_: Her noktanın atıldığı küme numarası. Küme numarası 0'dan başlatılmaktadır. Örneğin biz bu sayede kümelere düşen elemanları kolay bir biçimde yazdırabiliriz:

```
from sklearn.cluster import KMeans
km = KMeans(n_clusters=2)
km.fit(dataset)
print(km.labels_)

print('İlk grubun noktaları:')
print(dataset[km.labels_ == 0])

print('İkinci Grubun noktaları:')
print(dataset[km.labels_ == 1])
```

cluster_centers_: Nihai çözümdeki kümelerin ağırlık merkezleri.

inertia_: Tüm noktaların kendi ağırlık merkezlerine uzaklıklarının karelerinin toplamı. inertia_ örnek özniteliği tek bir elemandan oluşan toplam değerini vermektedir. (Halbuki kendi yaptığımız örnekte kmeans fonksiyonun geri döndürdüğü inertia'nın her kümenin kendi elemanlarının inertia'sı olduğuna dikkat ediniz.)

n_iter_: En iyi çözümün kaç iterasyonla bulunduğunu belirtmektedir.

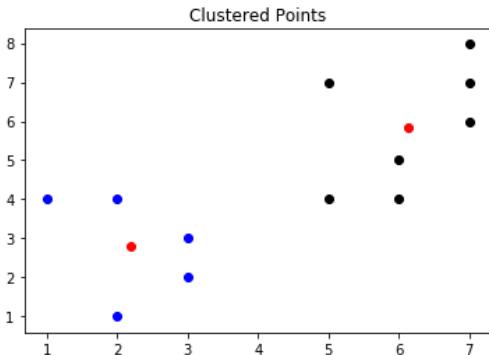
Şimdi de yukarıdaki noktalar için scikit-learn KMeans sınıf ile elde edilen çözümün saçılma grafiğini çizelim:

```
dataset = np.loadtxt('points.csv', delimiter=',', dtype=np.float32)

from sklearn.cluster import KMeans

km = KMeans(n_clusters=2)
km.fit(dataset)

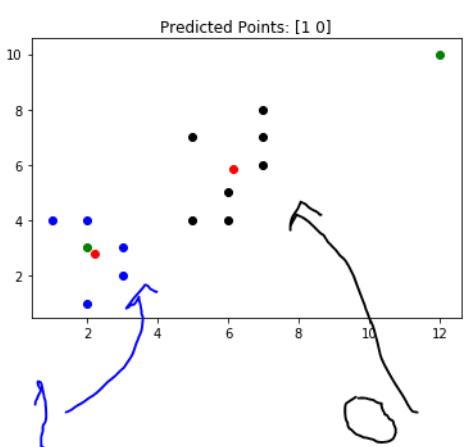
plt.title('Clustered Points')
plt.scatter(dataset[km.labels_ == 0, 0], dataset[km.labels_ == 0, 1], c='black')
plt.scatter(dataset[km.labels_ == 1, 0], dataset[km.labels_ == 1, 1], c='blue')
plt.scatter(km.cluster_centers_[:, 0], km.cluster_centers_[:, 1], c='red')
```



KMeans sınıfının predict isimli metodu yeni bir verinin sınıfını bize vermektedir. Aslında predict metodu bizim parametre olarak girdiğimiz veriler hangi ağırlık merkezlerine daha yakınsa o verileri ilgil kümelere sokmaktadır. Örneğin:

```
predict_data = np.array([[2, 3], [12, 10]])
result = km.predict(predict_data)
print(result)

plt.title('Predicted Points: {}'.format(result))
plt.scatter(dataset[km.labels_ == 0, 0], dataset[km.labels_ == 0, 1], c='black')
plt.scatter(dataset[km.labels_ == 1, 0], dataset[km.labels_ == 1, 1], c='blue')
plt.scatter(km.cluster_centers_[:, 0], km.cluster_centers_[:, 1], c='red')
plt.scatter(predict_data[:, 0], predict_data[:, 1], c='green')
```



Zambak (Iris-Plant) Verileriyle K-Means Uygulması

Zambak (Iris) resimlerinin sınıflandırılması klasik sınıflandırma çalışmalarından biridir. Veri tablosunda zambak resimleri alınarak bunların aşağıdaki özellikleri sayısal biçimde kodlanmıştır:

- Çanak yaprakların santimetre cinsinden uzunluğu
- Çanak yaprakların santimetre cinsinden genişliği
- Taç yaprakların santimetre cinsinden uzunluğu
- Taç yaprakların santimetre cinsinden genişliği

Bu verilerden hareketle zambaklar üç sınıfa ayrılmaktadır: Iris Setosa, Iris Versicolor, Iris Virginica

Iris-Plant veri tablosu 150 zambak bilgisinden oluşmaktadır. Her zambak için onun sınıfı tabloda belirtilmektedir. (Yani bu tablo eğitimli ağlarda da kullanılabilir.) Ancak biz burada K-means yöntemini kullanarak eğitimimsiz biçimde kümeleme yapacağız.

Iris veri tablosu bir CSV dosyası biçiminde indirilebilir. Dosyanın görünümü aşağıdaki gibidir:

```
5.1,3.5,1.4,0.2,Iris-setosa  
4.9,3.0,1.4,0.2,Iris-setosa  
4.7,3.2,1.3,0.2,Iris-setosa  
4.6,3.1,1.5,0.2,Iris-setosa  
5.0,3.6,1.4,0.2,Iris-setosa  
5.4,3.9,1.7,0.4,Iris-setosa  
4.6,3.4,1.4,0.3,Iris-setosa  
5.0,3.4,1.5,0.2,Iris-setosa  
4.4,2.9,1.4,0.2,Iris-setosa  
4.9,3.1,1.5,0.1,Iris-setosa  
5.4,3.7,1.5,0.2,Iris-setosa  
.....
```

Bu dosyadan hareketle veri kümесini x ve y biçiminde aşağıdaki gibi ikiye ayıralırız:

```
import numpy as np  
  
dataset_x = np.loadtxt('iris.csv', delimiter=',', usecols=range(4), dtype=np.float32)  
dataset_y = np.loadtxt('iris.csv', delimiter=',', usecols=[4], dtype=np.object)  
  
from sklearn.preprocessing import LabelEncoder
```

Fakat aslında Keras ve scikit-learn paketlerinde bu Iris verilerini bu biçimde bize veren sınıflar ve fonksiyonlar zaten bulunmaktadır. Yani benzer işlemler scikit-learn'de şöyle de yapabilirdik:

```
from sklearn.datasets import load_iris  
  
iris = load_iris()  
  
dataset_x = iris.data  
dataset_y = iris.target
```

Biz burada verileri dataset_x ve dataset_y biçiminde iki gruba ayırdık. Ancak tabii hedefimiz bir eğitim uygulamak değildir. dataset_y'yi biz eğitimsiz kümelemenin başarısını test etmek için ayrıca kullanacağız.

Şimdi de KMeans sınıfını kullanarak bu verileri üç kümeye ayıralım:

```
from sklearn.cluster import KMeans  
  
km = KMeans(n_clusters=3)  
km.fit(dataset_x)  
  
for i in range(3):  
    cluster = dataset_x[km.labels_ == i]  
    print('-----')  
    print('Cluster {}'.format(i))  
    print('-----')  
    print(cluster)
```

Şimdi de bir zambak bilgi uydurup onun hangi sınıfa ilişkin olabileceğini kestirelim:

```
data = np.array([[4.9, 3.1, 1.5, 0.2]])  
result = km.predict(data)
```

Tabii burada bize predict ile verilen yalnızca bir küme numarasıdır. Hangi küme numarasının hangi zambak çeşidine karşılık geldiğini algoritma bileyem. Bunu bizim belirlememiz gereklidir. Başka bir deyişle K-Means algoritması sayısal bilgilere göre kümeleme yapmaktadır. O kümelerin ne anlam ifade ettiği hakkında bize bir bilgi vermemeştir.

Bazen uygulamacılar fit işlemi sonrasında labels_ özniteliğine bakmak yerine doğrudan asıl veri kümesiyle predict işlemi de yapabilmektedir. Örneğin:

```
result = km.fit(dataset_x).predict(dataset_x)
```

Burada result ile elde edilen sonuç zaten labels_ özniteliği ile de alınmaktadır. fit metodunun KMeans nesnesinin kendisine geri döndüğünü anımsayınız.

Pekiyi zambakların mademki elimizde onların gerçek sınıfları (dataset_y) var. O halde biz K-Means algoritmasından elde ettiğimiz sonucun başarısını test edebiliriz. Ancak burada şu noktaya dikkat etmemiz gereklidir: KMeans sınıfı ile elde ettiğimiz sınıf numaralarının dataset_y'deki sınıf numaralarıyla aynı olması gerekebilmektedir. Önce gözle bakarak bu sınıf numaralarını dönüştürmemiz gereklidir.

```
from sklearn.preprocessing import LabelEncoder  
  
le = LabelEncoder()  
dataset_y = le.fit_transform(dataset_y)  
total = np.sum(dataset_y == km.labels_)  
ratio = total / len(dataset_y)  
print(ratio)
```

Burada 0.893 gibi bir oran elde ettik. Yani bu örneğimizde zambakların %89'u doğru bir biçimde kümelere ayrılmıştır.

K-Means Kümeleme Yönteminde Ölçekleme (Normalizasyon)

K-Means kümeleme yönteminde satırındaki bilgi çok fazla sütundan oluşuyor olabilir. Bu durumda iki nokta arasındaki uzaklık vektörel olarak hesaplandığında skalası büyük olan sütunlardaki toplam etki artacaktır. Bu da kümelemenin yanlış oluşmasına yol açabilecektir. Örneğin sütunlardan birisi on binli değerlerden diğerini tek basamaklı değerlerden oluşuyor olsun. Uzaklık hesabında bu on binli değerlere ilişkin sütun çok baskın hale gelecektir. İşte bu tür durumlarda sütunlardaki büyülüklüklerin birbirlerine yaklaştırılması için ölçekleme (normalizasyon) gerekebilmektedir. Biz yapay sinir ağları ile ilgili bölümde bu tür ölçeklemeleri hazır sınıflar kullanarak ya da manuel biçimde yapmıştık. Yukarıdaki Iris verilerinde böyle bir ölçeklemeye gerek duymadık. Çünkü örneklerimizdeki sütun değerleri zaten birbirlerine yakındır. Peki ölçekleme yapıp yapmama konusunda nasıl bir plan belirlemeliyiz? Normal olarak veri kümesindeki sütunlara göz atarak bu sütunlar arasında skala farklılığının olup olmadığına bakabiliriz. Ya da veri kümesi üzerinde K-Means için her zaman ölçeklendirme yapabiliriz. Çünkü gerekmediği durumlarda da ölçeklendirme yapmak bir soruna yol açmamaktadır. Şimdi biz -her ne kadar gerekmeyen olsa da- pratik yapmak amacıyla bir Iris verileri üzerinde bir ölçekleme yapalım:

```
import numpy as np  
from sklearn.preprocessing import StandardScaler  
  
ss = StandardScaler()  
ss.fit(dataset_x)  
transformed_dataset_x = ss.transform(dataset_x)  
  
from sklearn.cluster import KMeans  
  
km = KMeans(n_clusters=3, n_init=100)  
km.fit(transformed_dataset_x)  
for i in range(3):  
    cluster = dataset_x[km.labels_ == i]  
    print('-----')  
    print('Cluster {}'.format(i))  
    print('-----')  
    print(cluster)  
  
from sklearn.preprocessing import LabelEncoder  
  
le = LabelEncoder()
```

```

dataset_y = le.fit_transform(dataset_y)
total = np.sum(dataset_y == km.labels_)
ratio = total / len(dataset_y)
print(ratio)

```

Bu değerlerle ölçeklenmemiş değerler karşılaştırıldığında 11 eleman farklı olduğu görülmüştür. Başarı oranı ise 0.933 oalrak elde edilmiştir. Yani Iris veri kümelerinde ölçekleme daha iyi bir sonucun elde edilmesine yol açmıştır. Bir kez daha belirtmek gerekirse "ölçekleme yapmamak başarıyı azaltabilmektedir ancak gereksiz ölçekleme yapmak soruna yol açmamaktadır. Bu durumda her zaman ölçekleme yapılması tavsiye edilir.

Mnist Verileri İle K-Means Kümeleme Uygulaması

Şimdi de rakamları sınıflandırmak için kullandığımız Mnist örneğini K-Means kümelemesiyle yapmaya çalışalım.

```

from tensorflow.keras.datasets import mnist

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) = mnist.load_data()

training_dataset_x = training_dataset_x.reshape(-1, 28 * 28)
training_dataset_x = training_dataset_x / 255

from sklearn.cluster import KMeans

km = KMeans(n_clusters=10)
km.fit(training_dataset_x)

print(km.labels_)

```

Şimdi bazı sınıfların grafiklerini çizerek gözlemleyelim:

```

import matplotlib.pyplot as plt
import numpy as np

for i in range(10):
    clustered_result = training_dataset_x[km.labels_ == i]
    clustered_result = clustered_result.reshape(-1, 28, 28)
    print('-----')
    print('Cluster {}'.format(i))
    print('-----')
    for k in range(10):
        rindex = np.random.randint(len(clustered_result))
        plt.imshow(clustered_result[rindex], cmap='gray', interpolation='none')
        plt.pause(1)

```

Burada önemli noktalardan biri şudur: K-Means algoritmasından elde ettiğimiz km.labels_ değerlerinin training_set_y'deki değerlerle aynı olması gerekmemektedir. Çünkü K-Means bir denetimsiz öğrenme yöntemidir ve verileri uzaklıklarına göre sınıflandırmaktadır. Bu sınıflara da 0'dan başlayarak birer numara vermektedir. Bu numaraların rakamsal numaralarla ya da training_set_y'deki numaralarla örtüşmesi beklenmemelidir. Ancak yine de burada genel olarak başarı yüzdesinin düşük olduğu görülmektedir. Bu yüzdeyi artırmak için çeşitli önişlemler (preprocessing) kullanılabilmektedir.

Kümelemede Küme Sayılarının İşin Başında Tahmin Edilmesi

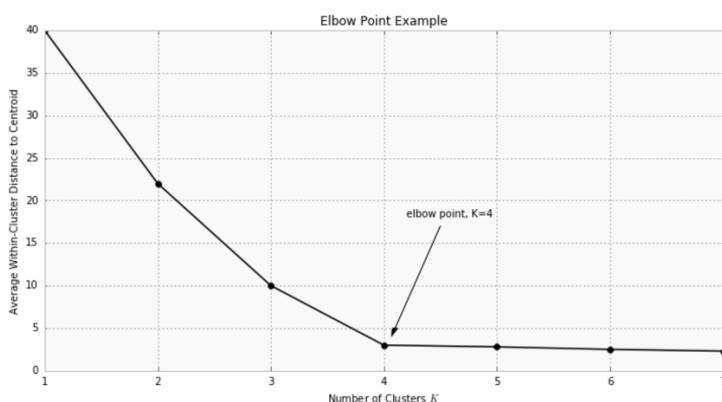
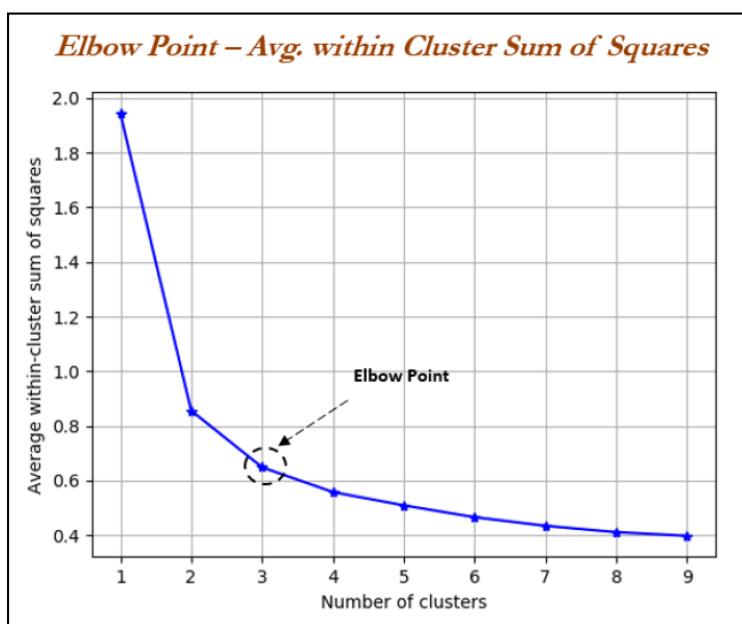
Biz K-Means yöntemini kullanırken verileri işin başında kaç kümeye ayıracagımızı biliyor olmamız gerekmektedir. Zaten bu bilgi çoğu durumda problemin kendisinde bulunmaktadır. Örneğin birtakım meyve fotoğrafları olabilir. Bu fotoğraftaki meyveleri sınıflandırmak isteyebiliriz. İşte eğer fotoğraflarda toplam kaç çeşit meyve olduğunu biliyorsak küme sayısını da biliyoruz demektir. Ancak toplamda kaç çeşit meyve olduğunu hiç bilmeyebiliriz. İşte bu tür

durumlarda K-Means işlemine başlamadan önce veriler incelenerek toplam kaç küme olabileceği çıkarımı yapılabilmektedir.

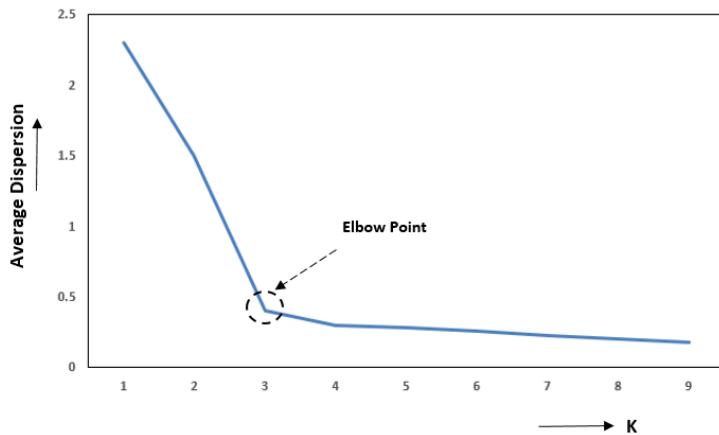
K-Means yönteminde işin başında küme sayısının bulunması temelde iki yöntemle yapılmaktadır:

- 1) Dirsek (Elbow) Yöntemi
- 2) Silhouette Yöntemi

Her iki yöntemde de biz deneysel olarak küme sayılarını 1'den belli bir sayıya kadar oluşturarak problemi çözeriz. Sonra birtakım değerleri karşılaştırırız. En uygun küme sayısını bu karşılaştırma sonrasında belirleriz. Dirsek yönteminde her kümedeki elemanın kendi kümelerinin ağırlık merkezlerine uzaklıklarının toplamı hesaplanır. (Buna "atalet (inertia)" demişti.) Sonra bu toplam uzaklıkların grafiği çizilir. Bu grafikte eğim düşümlerinin azaldığı yere ilişkin olan küme sayısı (bu dirseğin bulunduğu yerdir) en iyi küme sayısı olarak seçilir. Bu yöntem görsel biçimde karar vermeye dayalıdır. Aşağıda örnek dirsek grafikleri görülmektedir:



Elbow Method for selection of optimal “K” clusters



Grafiklerden de görüldüğü gibi dirsek noktası eğrinin kırılıp yataylaşmaya başladığı noktadır. Şimdi biz ilk örneğimizdeki noktalardan küme sayısını tahmin etmek için dirsek yöntemini kullanalım. Noktalarımız şöyledi:

```
import numpy as np

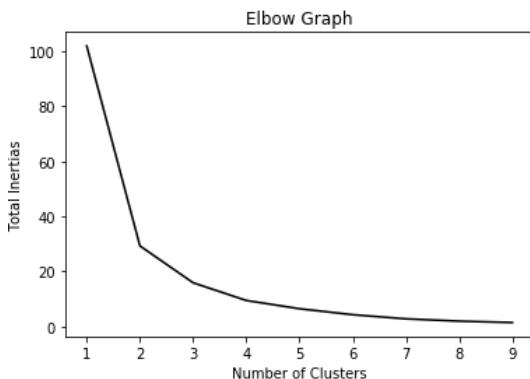
dataset = np.loadtxt('points.csv', delimiter=',', dtype=np.float32)

from sklearn.cluster import KMeans

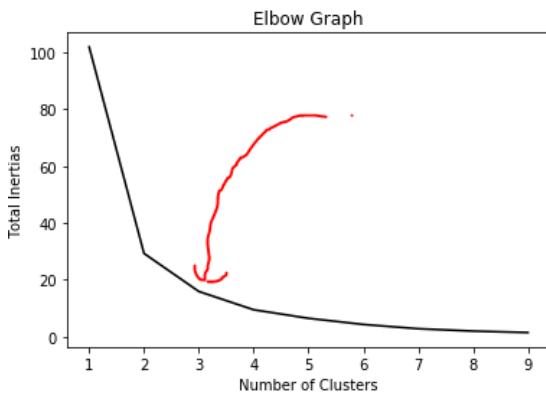
total_inertias = [KMeans(n_clusters=i).fit(dataset).inertia_ for i in range(1, 10)]

import matplotlib.pyplot as plt

plt.title('Elbow Graph')
plt.xlabel('Number of Clusters')
plt.ylabel('Total Inertias')
plt.plot(range(1, 10), total_inertias, c='black')
```



Bu grafiğe baktığımızda yataya geçme eğiliminin 3'ten başladığı görülmektedir. Yani grafikteki eğrinin kırılma noktası 3 gibi gözükmektedir. O halde dirsek yöntemine göre bu noktalar üç kümeye ayrılmalıdır.



Şimdi noktaları üç kümeye ayırarak çözümü yeniden bulalım:

```
import numpy as np

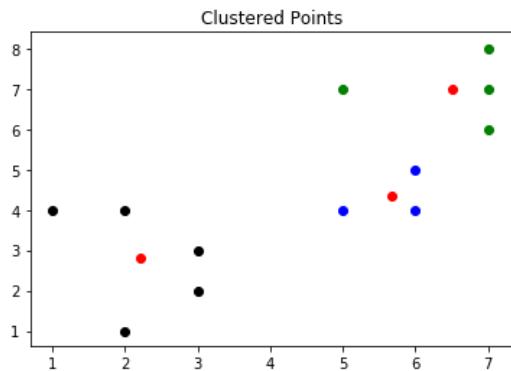
dataset = np.loadtxt('points.csv', delimiter=',', dtype=np.float32)

from sklearn.cluster import KMeans

km = KMeans(n_clusters=3)
km.fit(dataset)

import matplotlib.pyplot as plt

plt.title('Clustered Points')
plt.scatter(dataset[km.labels_ == 0, 0], dataset[km.labels_ == 0, 1], c='black')
plt.scatter(dataset[km.labels_ == 1, 0], dataset[km.labels_ == 1, 1], c='blue')
plt.scatter(dataset[km.labels_ == 2, 0], dataset[km.labels_ == 2, 1], c='green')
plt.scatter(km.cluster_centers_[:, 0], km.cluster_centers_[:, 1], c='red')
```



Şimdi de Iris verileri için dirsek grafiğini çizelim:

```
from sklearn import datasets

iris = datasets.load_iris()
dataset_x = iris.data
dataset_y = iris.target

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
ss.fit(dataset_x)
transformed_dataset_x = ss.transform(dataset_x)

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
```

```

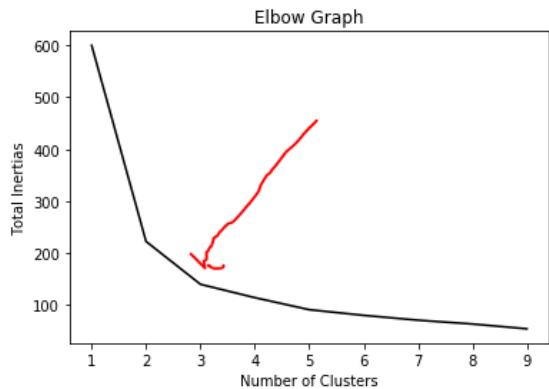
from sklearn.cluster import KMeans

total_inertias = [KMeans(n_clusters=i).fit(transformed_dataset_x).inertia_ for i in range(1, 10)]

import matplotlib.pyplot as plt

plt.title('Elbow Graph')
plt.xlabel('Number of Clusters')
plt.ylabel('Total Inertias')
plt.plot(range(1, 10), total_inertias, c='black')

```



Grafikten de görüldüğü gibi eğrinin yataya geçtiği küme sayısı 3'tür.

Silhouette yöntemi nümerik bir yöntemdir. Yöntemin ayrıntıları burada ele alınmayacaktır. Ancak bu yöntemi uygulayan hazır bir fonksiyon vardır. Bu fonksiyon sklearn.metrics modülündeki silhouette_score isimli fonksiyondur. Bu yöntemde kümeler için tek tek silhouette_score değerleri bulunur. Bu değerin en yüksek olduğu küme sayısından 1 fazlası nihai küme sayısı olarak tespit edilir. Yukarıdaki "point.csv" dosyasındaki veriler için en uygun küme sayısını bu kez Silhouette yöntemiyle bulmaya çalışalım:

```

import numpy as np

dataset = np.loadtxt('points.csv', delimiter=',', dtype=np.float32)

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

kms = [KMeans(n_clusters=i).fit(dataset) for i in range(2, 10)]
sscores = [silhouette_score(dataset, km.labels_) for km in kms]

for k, sscore in enumerate(sscores):
    print('{0}---->{1}'.format(k + 2, sscore))
print(np.argmax(sscores) + 3)

```

Elde edilen çıktı şöyledir:

```

2---->0.5544097423553467
3---->0.47607627511024475
4---->0.4601795971393585
5---->0.4254012405872345
6---->0.3836685121059418
7---->0.29372671246528625
8---->0.21625618636608124
9---->0.114889957010746
3

```

Göründüğü gibi burada 2 kümeli çözümdeki silhouette değeri en yüksektir (0.5544). O halde en uygun küme sayısı bu değerden bir fazla yani 3 olacaktır. Yukarıdaki örnekte de gördüğünüz gibi küme sayıları 2'den başlatılmıştır. 1 kğme için silhouette değeri söz konusu olmaz.

Örneğin Iris örneği için Silhouette değeri şöyle hesaplanabilir:

```
from sklearn import datasets

iris = datasets.load_iris()
dataset_x = iris.data
dataset_y = iris.target

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
ss.fit(dataset_x)
transformed_dataset_x = ss.transform(dataset_x)

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

kms = [KMeans(n_clusters=i).fit(transformed_dataset_x) for i in range(2, 10)]
sscores = [silhouette_score(transformed_dataset_x, km.labels_) for km in kms]

for k, sscore in enumerate(sscores):
    print('{}---->{}'.format(k + 2, sscore))

print(np.argmax(sscores) + 3)
```

Bu örnekte 2'den başlayarak 10'a kadar tek tek küme sayıları için silhouette_score değerleri hesaplanmıştır. Değerler şu biçimdedir:

```
2---->0.5817500491982808
3---->0.45994823920518635
4---->0.41511334907493763
5---->0.34551099599809465
6---->0.3304816878968672
7---->0.32456665698942505
8---->0.34135763346672265
9---->0.3316593073430383
3
```

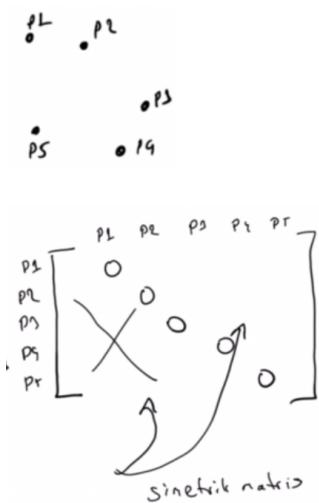
Sonra bu değerler içerisindeki en yüksek değer bulunur. Onun küme sayısının bir fazlası (örneğimizde 3) en iyi küme sayısı olarak tespit edilir.

Hiyerarşik Kümeleme (Hierarchical Clustering) Yöntemleri

K-Means kümeleme yönteminin yanı sıra daha pek çok kümeleme yöntemi de vardır. Bazı yöntemler bazı yöntemlerin biraz değiştirilmiş ve probleme göre genişletilmiş biçimleridir. Hiyerarşik kümeleme ise grup olarak özgün bir yöntem grubudur. Hiyerarşik kümeleme kendi aralarında "Agglomerative" ve "Divisive" olmak üzere ikiye ayrılmaktadır. Agglomerative yöntem tümevarımsal, divisive yöntem tümenden gelimseldir. Daha çok uygulamada Agglomerative yöntem tercih edilmektedir. Agglomerative hiyerarşik kümeleme algoritması şöyle işletilmektedir:

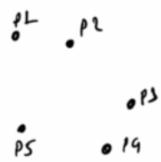
- 1) Önce her nokta ayrı bir küme varsayılarak işleme başlanır.
- 2) Sonra her noktanın her noktadan uzaklığı hesaplanır. Uzaklık hesaplamaada çeşitli yöntemler ve metrikler kullanılabilmektedir. Burada yöntem uzaklık hesaplamanın neye dayalı olarak yapılacağını belirtir. Metrik ise hesaplama biçimini belirtmektedir. Örneğin "öklit uzaklığı" bir yöntem değil metriktir. Ancak iki kümenin en yakın noktalarını hesaplamada kullanmak bir yöntemdir. Başka bir deyişle "yöntem birden fazla noktanın tek nokta olarak

İfade edilmesiyle ilgilidir, metrik ise iki noktanın uzaklığının hesaplanması biçimidir". İleride ele alınacağı gibi uzaklık hesaplamada çeşitli yöntemler tercih edilebilmektedir. Bu adımın sonunda bir uzaklık matrisi elde edilir. Örneğin 5 nokta söz konusu olsun. Uzaklık matrisi şöyle bir yapıda olacaktır:

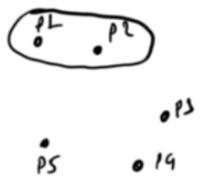


3) Uzaklık matrisinde en yakın iki nokta tespit edilir ve bu iki nokta yeni bir küme oluşturacak biçimde birleştirilir. Sonra yeniden tüm uzaklıklar yeni duruma göre hesaplanır. Aynı işlemler devam ettirilir. Her yinelemede bir eleman diğer bir küme ile birleştirilecektir. Bu işlem azrzu edilen küme sayısına kadar devam ettirilir.

Örneğin işin başında aşağıdaki gibi 5 nokta olsun:



Burada tüm noktalar arasındaki uzaklıklar hesaplandığında P1 ile P2'nin en yakın noktalar olduğunu varsayılmı. Bu durumda bu iki nokmayı tek bir küme olarak birleştiririz.



Artık burada 4 küme vardır. Bu 4 kümenin birbirlerine uzaklışı hesaplanarak yeni bir uzaklık matrisi oluşturulur. Buradaki sorunlardan biri birden fazla noktadan oluşan kümeler için uzaklık yerinin neresi alınacağıdır. İşte bunun için izleyen kısımda açıklayacak olduğumuz birkaç yöntem kullanılmaktadır. Örneğin bu kümelerin en yakın noktalarının kullanılması yöntemlerden biridir. Bu haliyle en yakın iki nokta P3 ve P4 noktalarıdır. O halde bu noktalar da birleştirilir:



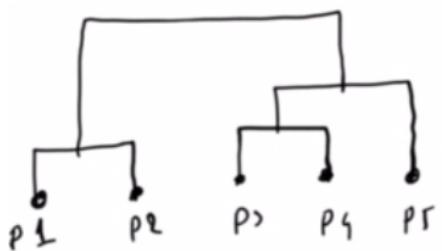
Bu yeni durumda P5 ile {P3, P4} kümelerin mesafelerinin en yakın olduğunu varsayılmı. O halde bunlardan yeni bir küme yapılacaktır:



Artık burada iki tane kümeli kalmıştır. Bu aşamada onlar da birleştirilir:



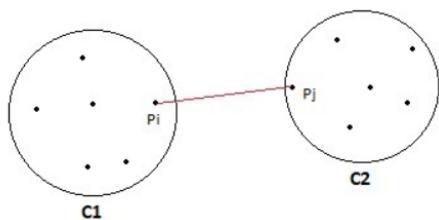
Hangi kümelerin hangi kümelerle birleştirildiğini gösteren grafiğe "dendrogram" denilmektedir.



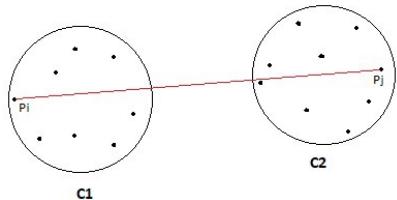
Divisive yöntemde tamamen işlemler tersten yapılmaktadır. Yani işin başında sanki tek bir kümeli varmış gibi başlanır. Sonra her defasında bu kümeden ayırtımalar yapılır. Başka bir deyişle agglomerative yöntem tümeyerimsal (bottom-up), Divisive yöntem ise tümdeğelimsel (top-down) biçimdedir.

Birden fazla noktadan oluşan kümeler için uzaklık hesaplamasında şu yöntemler kullanılmaktadır:

Min Yöntemi: Burada uzaklık olarak iki kümenin en yakın iki noktası arasındaki uzaklık alınır. Örneğin:



Max Yöntemi: Burada uzaklık olarak iki kümenin en uzak iki elemanı arasındaki uzaklık alınır.



Grup Ortalaması Yöntemi: Bu yöntemde iki kümenin tüm iki noktalarının uzaklıklarının ortalaması alınır. Örneğin kümelerden birinin 10 noktası diğerinin 5 noktası olsun. Mümkün uzaklıkların sayısı $10 * 5 = 50$ 'dir. Bu 50 uzaklık toplanıp 50'ye bölünür.

Ward Yöntemi: Bu yöntem grup ortalaması yönteminin çok benzeridir. Tek farkı noktalar arasındaki uzaklıkların karelerinin ortalamasının alınmasıdır. Uygulamaların çoğu default olarak bu yöntem tercih edilmektedir.

Hiyerarşik Kümeleme Yönteminin Scikit-Learn Kullanılarak Gerçekleştirilmesi

Hiyerarşik kümeleme için Scikit-Learn içerisindeki `sklearn.cluster` modülündeki `AgglomerativeClustering` isimli sınıf kullanılmaktadır. Sınıfın `__init__` metodunun parametrik yapısı şöyledir:

```
sklearn.cluster.AgglomerativeClustering(n_clusters=2, affinity='euclidean', memory=None, connectivity=None, compute_full_tree='auto', linkage='ward', pooling_func='deprecated', distance_threshold=None)
```

Fonksiyonun `n_clusters` parametresi ayrılacek küme sayısını, `linkage` parametresi kullanılacak yöntemi belirtir. Bunun default olarak 'ward' olduğuna dikkat ediniz. `affinity` parametresi kullanılacak metriki belirtmektedir. Bu parametre de default olarak 'euclidean' biçimindedir. Fonksiyon bize bir sınıf nesnesi verir. Buradan alınan nesneyle tipki KMeans sınıfında olduğu gibi `fit` metodu çağrılır. Benzer biçimde yine kümeler sınıfın `labels_` isimli özniteligidinden elde edilmektedir. Şimdi yukarıdaki "points.csv" örneğini üç sınıf için `AgglomerativeClustering` sınıfını kullanarak çözelim:

```
import numpy as np

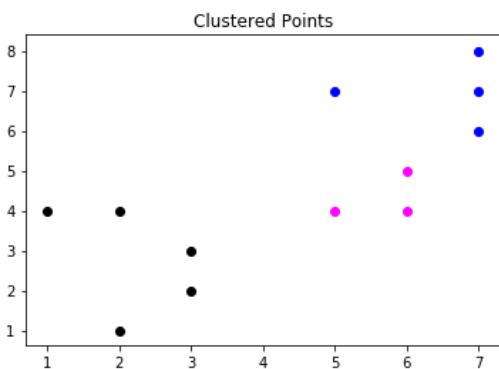
points = np.loadtxt('points.csv', delimiter=',', dtype=np.float32)

from sklearn.cluster import AgglomerativeClustering

ac = AgglomerativeClustering(n_clusters=3)
ac.fit(points)

import matplotlib.pyplot as plt

plt.title('Clustered Points')
plt.scatter(points[ac.labels_ == 0, 0], points[ac.labels_ == 0, 1], c='black')
plt.scatter(points[ac.labels_ == 1, 0], points[ac.labels_ == 1, 1], c='blue')
plt.scatter(points[ac.labels_ == 2, 0], points[ac.labels_ == 2, 1], c='magenta')
```



`AgglomerativeClustering` sınıfının `clusters_centers_` biçiminde bir örnek özniteliği yoktur. Çünkü hiyerarşik kümelemede ağırlık merkezi (centroid) kullanılmamaktadır. Benzer biçimde yine bu sınıfın `predict` metodu da yoktur. Çünkü hiyerarşik sınıflamada bu anlamda bir kestirim yapılamaz. Yeni bir nokta için tüm işlemlerin baştan başlatılması gerekmektedir.

Şimdi de Iris örneğini hem K-Means yönetimi ile hem de hiyerarşik yöntemle çözüp sonuçları kontrol edelim.

```
from sklearn.datasets import load_iris

iris = load_iris()
dataset_x = iris.data
dataset_y = iris.target

from sklearn.preprocessing import StandardScaler
```

```

ss = StandardScaler()
transformed_dataset_x = ss.fit_transform(dataset_x)

from sklearn.cluster import KMeans, AgglomerativeClustering

km = KMeans(n_clusters=3)
km.fit(transformed_dataset_x)

ac = AgglomerativeClustering(n_clusters=3)
ac.fit(transformed_dataset_x)

print(km.labels_)
print(ac.labels_)

km_labels = [{1: 0, 2: 1, 0: 2}[label] for label in km.labels_]
ac_labels = [{1: 0, 2: 1, 0: 2}[label] for label in ac.labels_]

import numpy as np

km_success_ratio = np.sum(km_labels == dataset_y) / len(dataset_y)
ac_success_ratio = np.sum(ac_labels == dataset_y) / len(dataset_y)

print('KMeans success ration: {}'.format(km_success_ratio))
print('Agglomerative success ration: {}'.format(ac_success_ratio))

ratio = np.sum(km_labels == ac.labels_) / len(ac.labels_)
print('Identical class ratio: {}'.format(ratio))

```

Bu programda biz K-Means yöntemi ile Agglomerative yöntemin çıktılarını karşılaştırdık. Buradaki başarı oranı şöyle bulunmuştur:

```

KMeans success ration: 0.8333333333333334
Agglomerative success ration: 0.8266666666666667

```

Göründüğü gibi KMeans çok az daha başarılı olmuşsa da değerler birbirlerine çok yakındır.

Agglomerative Yöntemde Dendrogram Çizilmesi

Anımsanacağı gibi dendrogram hiyerarşik agglomerative kğmelemede hangi noktaların hangi noktalarla birleştirildiğini gösteren bir ağaç grafiği idi. Dendrogram denilen ağaç grafiğinin oluşturulması için scipy kütüphanesinde scipy.cluster.hierarchy modülünde dendrogram isimli bir fonksiyon bulunmaktadır. Bu fonksiyon çizimin kendisini yapar. Ancak bu fonksiyon çizimin nasıl yapılacağını parametresiyle aldığı matrise bakarak tespit etmektedir. Bu matris de aynı modüldeki linkage isimli fonksiyon tarafından elde edilmektedir. Yani linkage fonksiyonu aslında agglomerative kümeleme işlemini yapıp birleştirme bilgilerini bize matris olarak verir. Biz de bu matrisi dendrogram fonksiyonuna veririz.

Points.csv verileri üzerinde dendrogram şöyleden çizilebilir:

```

import numpy as np

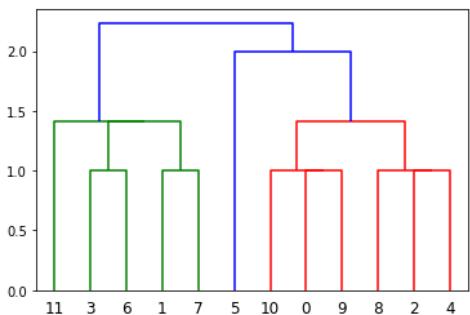
points = np.loadtxt('points.csv', delimiter=',', dtype=np.float32)

from scipy.cluster.hierarchy import linkage, dendrogram

import matplotlib.pyplot as plt

m = linkage(points)
dendrogram(m)

```



Iris verileri için de dendogram şöyle çizilebilir:

```
from sklearn import datasets
from scipy.cluster.hierarchy import linkage, dendrogram

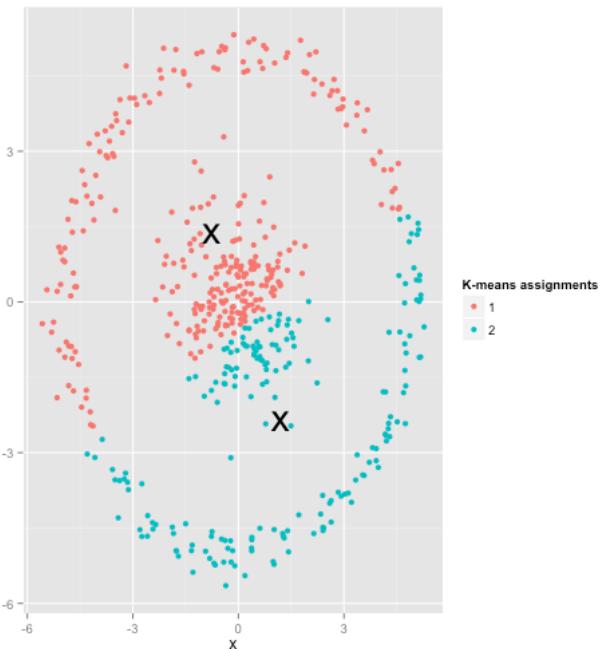
import matplotlib.pyplot as plt

plt.figure(figsize=(25, 10))
m = linkage(transformed_dataset_x)
dendrogram(m)
```

K-Means Yöntemiyle Hiyerarşik (Agglomerative) Yöntemin Karşılaştırılması

Aslında her iki yöntemin de başarısı birbirine benzer olabilmektedir. Ancak iki yöntemin arasındaki farklılıklar şöyle özetlenebilir:

- Yüksek sayıda nokta söz konusu olduğunda hiperarşik (agglomerative) yöntem daha fazla zaman almaktadır. Çünkü işin başında tüm noktaların ayrı kümeler olarak ele alınması ve gitgide birleştirilmesi uzun hesaplama zamanına yol açmaktadır.
- Hiperarşik yöntemde algoritma rastgele bir biçimde başlatılmamaktadır. Dolayısıyla algoritmanın her çalıştırılmasında aynı sonucun bulunmasına yol açar. Halbuki K-Means yönteminde başlangıç ağırlık merkezleri rastgele alındığı için algoritmanın her çalıştırılmasında farklı sonuçlar elde edilebilmektedir.
- Hiperarşik yöntem daha esnektir. Çünkü kümeler arasındaki uzaklık hesaplama yöntemi (similarity) değiştirilebilmektedir. Oysa K-Means yönteminde yalnızca metrik değer değiştirilebilmektedir.
- Hiperarşik yöntemde dendrogram çizilebilmekte dolayısıyla hangi kümelerin hanki kümelerle birleştirildiği gözle görülebilmektedir. Dendrograma bakılarak küme uzaklıklarına dayalı küme sayıları bulunabilmektedir.
- KMeans yöntemi küresel (spherical) olmayan nokta dağılımlarında iyi çalışmaz. Küresel nokta dağılımları noktaların bir merkez etrafında toplanması biçiminde oluşan dağılımlardır. Küresel olmayan nokta dağılımlarında hiperarşik yöntem daha iyi sonuç vermektedir. Aslında küresel olmayan noktalar için en iyi yöntem grubu yoğunlu tabanlı (density based) olanlardır. Aşağıda küresel olmayan nokta dağılımlarındaki kmeans kümelerini görürsünüz:

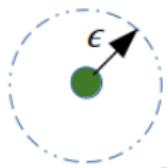


Şekil <http://varianceexplained.org/r/kmeans-free-lunch/> makalesinden alınmıştır.

Burada belki de dış dairesel noktaların ayrı bir küme, iç dairesel noktaların ayrı bir kme olarak değerlerndirilmesi daha uygun olabilecektir. Ancak KMeans hiçbir zaman buna yönelik bir kümeleme yapamamaktadır.

DBSCAN Kümeleme Yöntemi

DBSCAN (Density-Based Spatial Slustering of Applications with Noise) yoğunluk tabanlı kümeleme yöntemlerinin en çok kullanılanlarından biridir. Algoritma 90'lı yıllarda son haline getirilmiştir. DBSCAN yönteminde yoğunluk (density) en önemli unsurdur. Yoğunluk belli bir alanda bulunan nokta sayısı ile ilgilidir. Yani belli bir alanda çok nokta varsa o alan yoğundur, az nokta varsa o alan yoğun değildir. Yoğunluk için kullanılan alan genellikle daireseldir ve bu dairenin yarı çapı epsilon ile gösterilmektedir.



Şekil <https://towardsdatascience.com/dbscan-algorithm-complete-guide-and-application-with-python-scikit-learn-d690cbae4c5d> makalesinden alınmıştır.

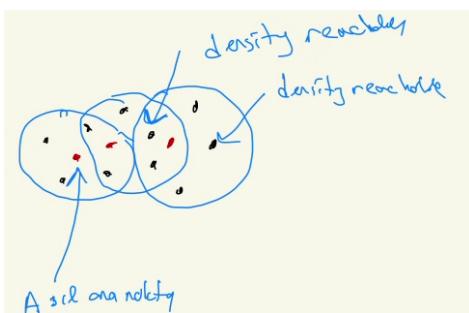
DBSCAN algoritmasında iki önemli parametre vardır: "Yarıçap (epsilon)" ve "en az nokta sayısı (minPts)". Yarıçap yukarıda belirtildiği gibi ilgili noktanın çevresindeki çemberin yarıçapıdır. En az nokta sayısı ise bir alanın yoğun kabul edilebilmesi için o çemberin içerisinde en az kaç noktanın bulunması gerektiğini belirtir. Örneğin "epsilon = 0.5, minPts = 15" demek, 0.5 yarıçaplı çizilen çemberin içerisinde 15 ya da daha fazla nokta varsa orası yoğun" demektir. Algoritmada yoğunluk noktası temelinde değerlendirilmektedir. Yani bir noktası çemberin merkezi varsayılarak o noktanın oluşturduğu alanın yoğun olup olmadığına bakılmaktadır.

DBSCAN algoritmesinde anlatımı kolaylaştırmak amacıyla birkaç kavram kullanılmaktadır. Şimdi bunları açıklayalım:

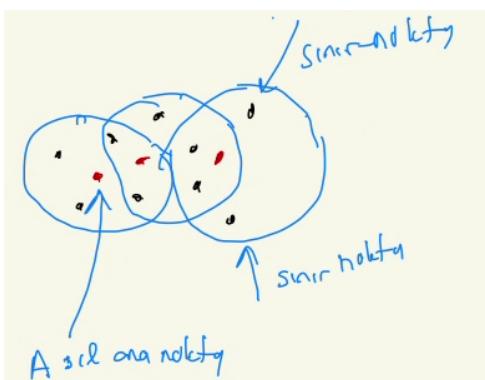
Ana Noktalar (Core Points): Eğer bir noktanın epsilon yarıçaplı çemberi içerisinde minPts ya da daha fazla noktası kalıyorsa, yani o noktanın belirttiği alan yoğun ise böyle noktalara ana noktalar (core points) denilmektedir.

Doğrudan Erişilebilir Noktalar (Direct Reachable Points): Bir ana noktanın oluşturduğu çemberin içerisindeki noktalara o ana noktanın doğrudan erişilebilen noktaları denilmektedir.

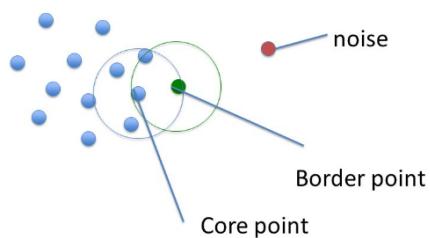
Yoğunluk Yoluyla Erişilebilir Noktalar (Density Reachable Points): Bir ana noktanın doğrudan erişilebilen (yani onun çemberi içerisinde kalan) bir noktası da ana nokta ise o noktanın doğrudan erişilenen noktası asıl noktanın "yoğunluk yoluyla erişilebilen" noktasıdır. Yani yoğunluk yoluyla erişilebilen noktalar "arkadaşımın arkadaşı arkadaşımdır" önermesine benzetilebilir. Tabii geçişlik özelliği devam etmektedir. Yani Ana noktanın doğrudan erişilebilen ana noktasının doğrudan erişilebilen ana noktaları da ilk noktanın yoğunluk yoluyla erişilebilen noktalarıdır. (Yani "arkadaşımın arkasının arkadaşları da benin arkadaşımdır.") Örneğin:



Sınır Noktalar (Border Points): Bir noktanın yoğunluk yoluyla erişilebilen ancak ana nokta olmayan noktalarına sınır noktaları (border points) denilmektedir. Buradaki sınır noktaları ismi bu noktaların ilgili kümenin üç noktaları olması nedeniyle verilmiştir. Örneğin:



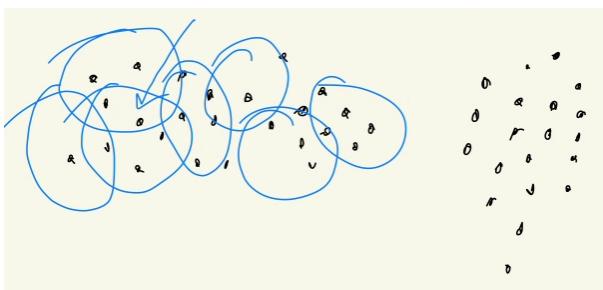
Gürültü Noktaları (Noise Points): Ana nokta ve sınır nokta olmayan noktalara gürültü noktaları denilmektedir. Bu noktalar bir kümenin içerisine dahil edilmezler ve anomali olarak değerlendirilirler. Anomali tespit yöntemlerinde buna benzer gürültü noktaları üzerinde durulmaktadır. Örneğin:



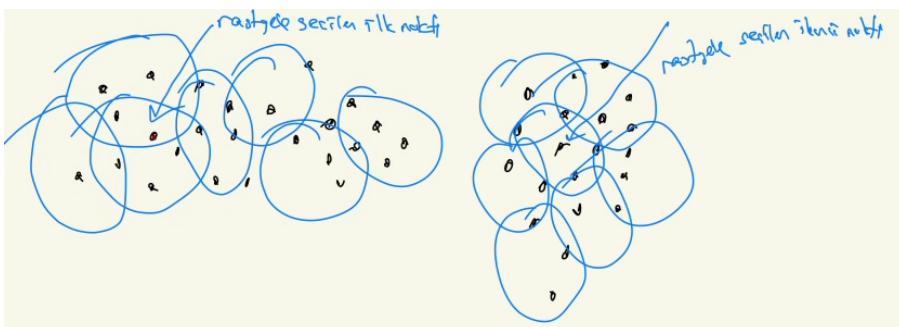
Bu kavramlardan sonra şimdi de DBSCAN algoritmasını açıklayalım. Epsilon ve minPts değerlerinin algoritma girdisi olarak verildiğini varsayıyalım:

- 1) Kalan noktalar kümesinde (başlangıçta tüm noktalar kalan noktalar kümesindedir) rastgele bir nokta alınır. Bu noktanın ana nokta olup olmadığına (yani etrafının yoğun olup olmadığına) bakılır. Eğer bu nokta ana nokta değilse gürültü noktası biçiminde işaretlenir. Tabii önce gürültü noktası biçiminde işaretlenen bir nokta daha sonda bir kümeye dahil edilebilmektedir.
- 2) Eğer rastgele seçilen nokta bir ana noktaya onun doğrudan erişilebilen (yani onun çemberi içerisinde kalan) noktaları bir kişiye yaratılarak ona dahil edilir. Sonra bu noktalar gözden geçirilir. Doğrudan erişilebilen noktalar içerisinde ana noktalar olup olmadığına bakılır. Doğrudan erişilebilen noktaların içerisindeki ana noktaların doğrudan

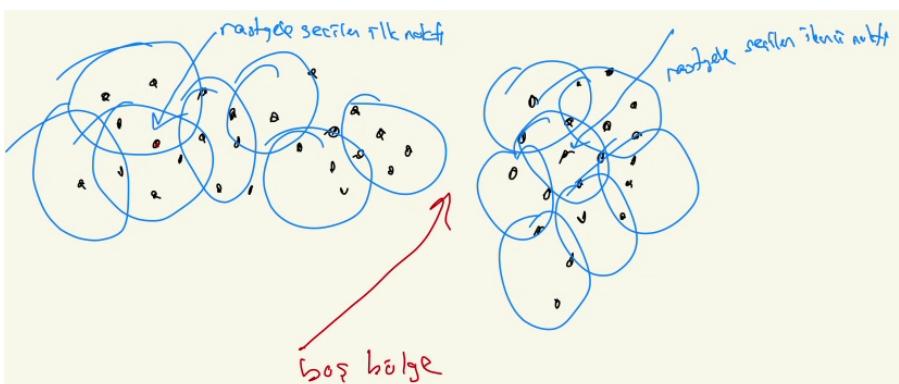
erişilebilen noktaları da kümeye dahil edilir. Böylece özyineleemeli biçimde işlemler devam ettirilerek ilk kümenin bütün noktaları belirlenmiş olur. Yani artık ilk noktanın yoğunluk yoluyla erişilebilen noktalarındaki bütün ana noktalar kümeye dahil edilmiştir.



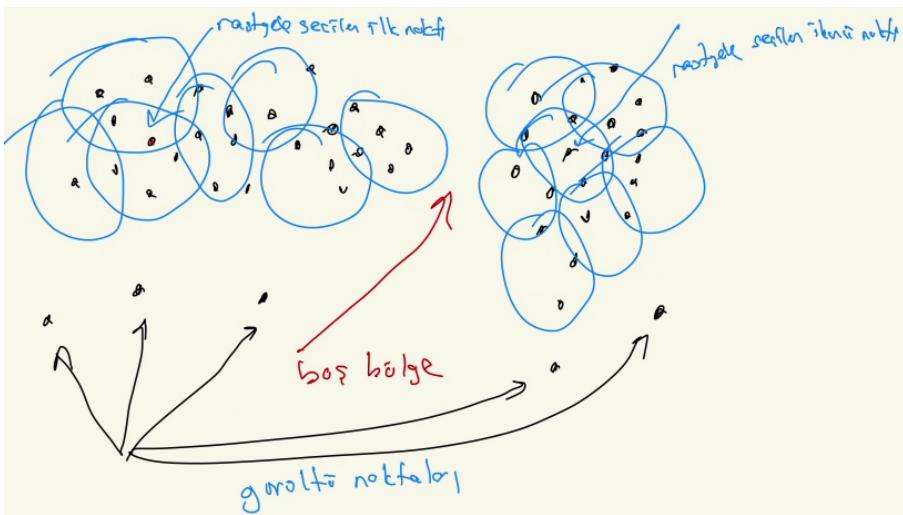
Bu işlem yalnızca bir kümeyi tespit edebilmiştir. Burada algoritma 1'inci adıma geri dönerken bir kümeye sokulmamış ve gürültü olarak işaretlenmemiş noktalar arasından yeni rastgele bir nokta seçer. Aynı şeyleri o nokta için de yapar. Böylece diğer kümeler elde edilmiş olur.



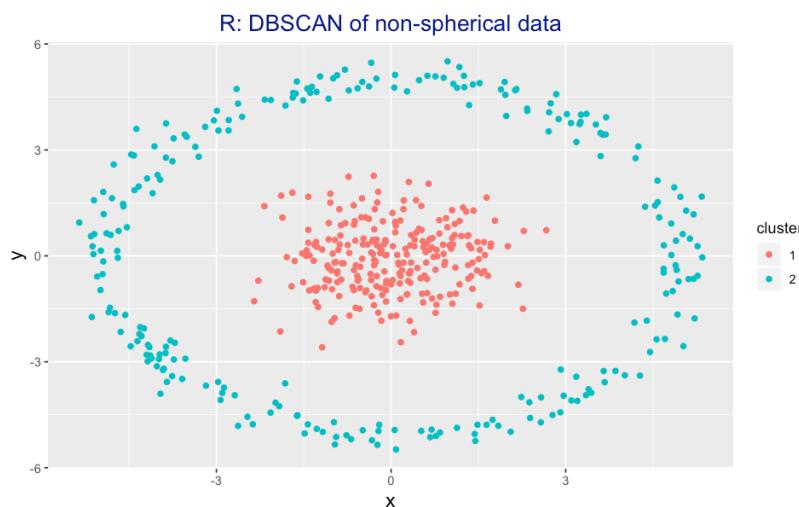
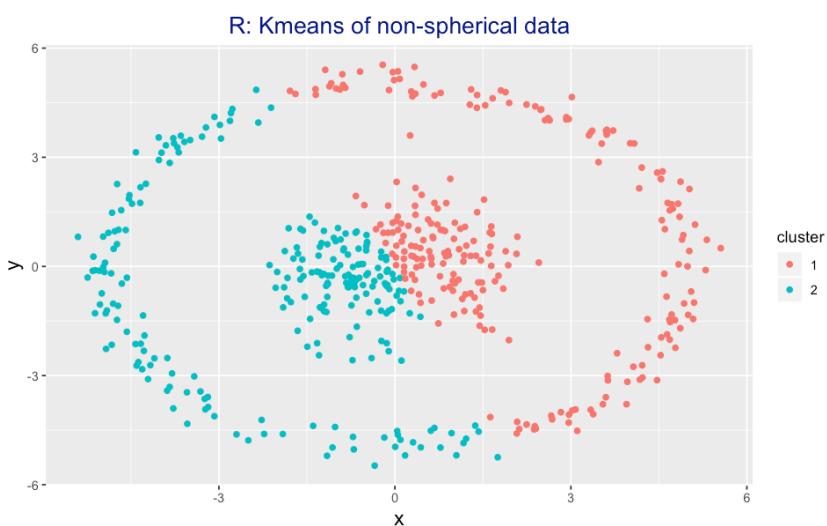
Burada görüldüğü gibi kümelerin birbirlerinden ayrılması için arada boş bir bölgenin olması gereklidir:



Algoritmanın sonunda hiçbir kümeye dahil edilememiş gürültü noktaları kalabilir. Örneğin:



Burada birkaç noktaya dikkatinizi çekmek istiyoruz. DBSCAN algoritmasında biz algoritma kümeye sayıını vermemekteyiz. Algoritma kümeleri epsilon ve minPts değerlerine dayalı olarak kendisi belirlemektedir. Diğer önemli bir nokta da algoritmanın tüm noktaları kümelendirmediği gürültü noktalarını kümelerin dışında bıraktığıdır. Bu iki durum daha önce görmüş olduğumuz KMeans ve Agglomerative (hiyerarşik) kümleme yöntemlerinden farklıdır. Ayrıca DBSCAN küresel olmayan noktaları da iyi biçimde sınıflandırabilmektedir. Örneğin:



Şekiller https://datascience-enthusiast.com/Python/DBSCAN_Kmeans.html makalesinden alınmıştır.

DBSCAN Kümeleme Yönteminin Scikit-Learn Kütüphanesi Kullanılarak Uygulanması

DBSCAN kümeleme yöntemi scikit-learn kütüphanesindeki DBSCAN isimli sınıfı temsil edilmiştir. Sınıfın `__init__` metodunun parametrik yapısı şöyledir:

```
class sklearn.cluster.DBSCAN(eps=0.5, *, min_samples=5, metric='euclidean', metric_params=None, algorithm='auto', leaf_size=30, p=None, n_jobs=None)
```

Fonksiyonun iki önemli parametresi `eps` ve `min_samples` parametreleridir. `eps` parametresi epsilon değerini, `min_samples` parametresi de minPts değerini belirtir. Bunların default değer aldığına dikkat ediniz. Ayrıca fonksiyonun `metric` parametresi bir noktanın çember içerisinde kalıp kalmadığını anlamak için kullanılan uzaklık hesap biçimini belirtmektedir. Bu değerin default olarak "euclidean" alındığına dikkat ediniz. Diğer yöntemlerde olduğu gibi burada da DBSCAN nesnesi yaratıldıktan sonra `fit` metoduyla algoritmayı çalışmak gereklidir. `fit` metodu kümelenen verileri parametre olarak almaktadır. Kümeleme yapıldıktan sonra hangi noktaların hangi kümelere dahil edildiği yine sınıfın `labels_` isimli özniteliğinden elde edilmektedir. `labels_` özniteliğinde kümeler 0'dan başlanarak numaralandırılmıştır. -1 değeri gürültü noktaları anlamına gelmektedir.

Şimdi "points.csv" dosyasındaki noktaları DBSCAN algoritmasıyla kümelendirelim:

```
import numpy as np

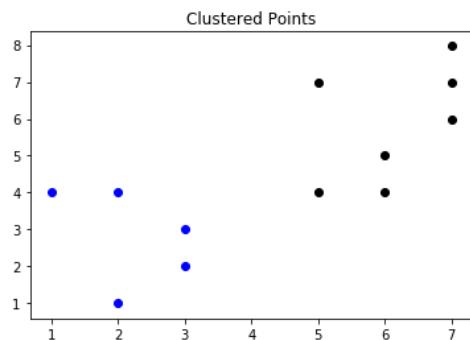
points = np.loadtxt('points.csv', delimiter=',', dtype=np.float32)

from sklearn.cluster import DBSCAN

dbs = DBSCAN(eps=1.5, min_samples=3)
dbs.fit(points)

import matplotlib.pyplot as plt

plt.title('Clustered Points')
plt.scatter(points[dbs.labels_ == 0, 0], points[dbs.labels_ == 0, 1], c='black')
plt.scatter(points[dbs.labels_ == 1, 0], points[dbs.labels_ == 1, 1], c='blue')
plt.scatter(points[dbs.labels_ == 2, 0], points[dbs.labels_ == 2, 1], c='magenta')
```



DBSCAN'in `eps = 1.5`, `min_sample = 3` parametreleriyle noktaları iki kümeye ayırdığını görüyoruz.

Şimdi de Iris verileri ile DBSCAN yöntemini kullanalım:

```
from sklearn.datasets import load_iris

iris = load_iris()
dataset_x = iris.data
dataset_y = iris.target

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
transformed_dataset_x = ss.fit_transform(dataset_x)
```

```
from sklearn.cluster import DBSCAN

dbs = DBSCAN(eps=1.5, min_samples=3)
dbs.fit(transformed_dataset_x)

print(dbs.labels_)
```

İşlemler sonucunda elde edilen değerler şöyledir:

Göründüğü gibi DBSCAN epsilon=1.5, minPts=3 parametreleriyle Iris verilerini iki sınıfa ayırmıştır. Bu parametrelerle hiç gürültü oluşmadığını dikkat ediniz. Şimdi algoritmayı default parametreler olan epsilon=0.5, minPts=5 değerleriyle çalıştırıp sonucuna bakalım:

Çemberi genişletip yoğunluk koşulunu 5'e çektiğimizde gürültü noktalarının arttığını görüyorsunuz. Biz daha önce kullandığımız yöntemlerde Iris verilerini 3 kümeye ayırmıştık. DBSCAN yönteminde küme sayısının belirtilmediğini epsilon ve minPts değerlerine göre otomatik oluşturulduğunu animasyonuz.

Varyans, Kovaryans ve Korelasyon Kavramları

Varyans istatistikteki en önemli kavramlardan biridir. Genel olarak dağılımdaki değerlerin ortalamadan uzaklığını ölçmek için kullanılır. Örneğin ortalamaları aynı olan değerlerin varyanslarına bakıldığından varyansı yüksek olan daha fazla yayılmıştır. Varyansı düşük olanın değerleri ortalamaya daha yakındır. Ya da örneğin ortalama yıllık gelirin 10000\$ olduğu iki ülkeden hangisinde varyans daha o ülkede gelir adeletsizliği diğerinden daha yüksektir. Varyansın karekökü standart sapmayı verir. Bu anlamda varyans ile standart sapma aynı amaçla kullanılmaktadır. Varyans değerlerin ortalamadan farklarının karelerinin toplamının ortalamasıyla hesaplanır. Varyans formülünde ortalama alınırken genel olarak ana kütle (population) için n 'e örneklemeler için $n - 1$ 'e bölme uygulanır.

$$\frac{\sum (x_i - \bar{x})^2}{n-1} \quad \frac{\sum (x_i - \hat{x})^2}{N}$$

Değerlerin ortalamadan uzaklıklarını hesaplamak için kare aldığımızda değerleri negatiflikten kurtarmış olmaktadır. Kare almak yerine mutlak değer kullanmak da bir yöntem olabilmektedir. Ancak kare almak burada açıklayamayacağımız bazı nedenlerden dolayı çok daha etkin bir yöntemdir.

Varyans ve standart sapma için Python'ın standart kütüphanesindeki `statistics` modülünün içinde bulunan `stddev`, `variance`, `pstdev`, `pvariance` fonksiyonları kullanılmaktadır. Bu fonksiyonlar dolaşılabilir herhangi bir nesneyi parametre olarak alabilmektedir. `stddev` ve `variance` örneklem için (yeni $n - 1$ bölümü yapmaktadır) `pstdev` ve `pvariance` ise ana kütle (population) için (yani N 'ye bölme yapmaktadır) kullanılmaktadır. Örneğin:

```

>>> import statistics
>>> a = [1, 3, 7, 2, 4]
>>> statistics.variance(a)
5.3
>>> statistics.pvariance(a)
4.24
>>> statistics.stdev(a)
2.3021728866442674
>>> statistics.pstdev(a)
2.0591260281974

```

Numpy kütüphendesinde ise varyans ve standart sapma için var ve std fonksiyonları kullanılmaktadır. Bu fonksiyonların parametrelerindeki ddof (delta degrees of freedom) parametresi ile bölmektedir. Bu parametre 0 olarak girilirse (default durum) bölme N'ye göre 1 olarak girilirse (n – 1)'e göre yapılmaktadır. Örneğin:

```

>>> import numpy as np
>>> a = [1, 3, 7, 2, 4]
>>> np.var(a)
4.24
>>> np.var(a, ddof=1)
5.300000000000001
>>> np.std(a)
2.0591260281974
>>> np.std(a, ddof=1)
2.302172886644268

```

Tabii numpy kütüphanesinde mean, var, std gibi fonksiyonların hepsi çok boyutlu dizilerle de çalışabilmektedir. Bu durumda axis parametresi ile satırsal ya da sütunsal işlem yapılacağı belirlenir. axis=0, sütunsal, axis=1 satırsal işlem anlamına gelmektedir. Eğer axis değeri None yapılrsa (default durum) bu durumda tüm değerler üzerinde işlemler yapılmaktadır. Örneğin:

```

a = np.array([[1, 2, 3], [4, 5, 2], [5, 9, 1]], dtype=np.float32)
np.var(a, axis=0)
Out[21]: array([2.888889, 8.222222, 0.6666667], dtype=float32)
np.var(a, axis=1)
Out[23]: array([ 0.6666667, 1.5555557, 10.666667], dtype=float32)
np.var(a)
Out[24]: 5.8024693

```

Kovaryans birden fazla değişkenin olduğu durumda (multivariate) bu değişkenlerin birbirleri arasındaki varyansı hesaplamak için yani birindeki değişimin diğerini etkileme biçimini incelemek için kullanılmaktadır.

x ve y'nin kovaryansları olan Cov(x, y) şu formülle hesaplanmaktadır:

$$\frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{N} \quad (\text{ya da } n-1)$$

Cov(x, y) ile Cov(y, x) değerlerinin aynı olmasına dikkat ediniz. Ayrıca Cov(x, x) ile Var(x) tamamen eşdeğer, benzer biçimde Cov(y, y) ile Var(y) değerleri de tamamen aynıdır. Örneğin iki değişken için manuel kovaryans işlemni şöyle yapabiliriz:

```

>>> covxy = np.sum((x - np.mean(x)) * (y - np.mean(y))) / (len(x) - 1)
>>> covxy
24.450000000000003

```

Kovaryans hesabı numpy kütüphanesinde cov isimli fonksiyonla yapılmaktadır. Biz cov fonksiyonunda iki değişken için ilk iki parametreyi kullanabiliriz. Fakat daha fazla değişken söz konusu olursa birinci parametreyi çok boyutlu dizi biçiminde alabiliriz. cov fonksiyonu çok boyutlu dizinin her bir satırını ayrı bir değişken biçiminde ele almaktadır. (Bu fonksiyon axis parametresi kullanmamaktadır.) Birden çok değişken söz konusu olduğunda kovaryans hesabı hepsinin birbirlerine göre ikişerli kovaryanslarını içerecek biçimde matrisel bir hal almaktadır. Örneğin:

X	y	z
5	3	1
7	4	6
2	8	5
1	9	5
3	3	7

Burada kovaryans hesabı $\text{Cov}(x, y)$, $\text{Cov}(x, z)$, $\text{Cov}(y, z)$ biçiminde ayrı ayrı yapılmaktadır. Bu kovaryans değerleri genellikle simetrik bir matris ile ifade edilir. Bu matrise kovaryans matrisi denilmektedir:

$$\begin{bmatrix} \text{cov}(x,x) & \text{cov}(x,y), \text{cov}(x,z) \\ \text{cov}(y,x) & \text{cov}(y,y), \text{cov}(y,z) \\ \text{cov}(z,x) & \text{cov}(z,y), \text{cov}(z,z) \end{bmatrix}$$

İki değişken için kovaryans matrisi de şöyle olacaktır:

$$\begin{bmatrix} \text{cov}(x,x) & \text{cov}(x,y) \\ \text{cov}(y,x) & \text{cov}(y,y) \end{bmatrix}$$

İşte numpy kütüphanesindeki cov isimli fonksiyon bize ürün olarak kovaryans matrisini vermektedir. Yani başka bir deyişle cov fonksiyonu bize tüm değişkenlerin birbirlerine göre kovaryanslarını bir matris olarak verir. cov fonksiyonu kovaryans hesaplamasında default olarak n - 1'e bölme işlemini uygular. Ancak yine fonksiyonun ddof parametresi 0 yapılarak n'e bölme işlemi uygulanabilir. Örneğin:

```
x = [5, 7, 2, 1, 3]
y = [3, 4, 8, 9, 3]
z = [1, 6, 5, 4, 7]

a = np.array([x, y, z], dtype=np.float32)

cov = np.cov(a)
print(cov)
```

Sonuç şöyledir:

```
[[ 5.8 -5.05 -0.2 ]
 [-5.05  8.3 -0.05]
 [-0.2 -0.05  5.3 ]]
```

Matrisin simetrik olduğuna dikkat ediniz. Örneğin matrisin [0, 3] indeksli elemanı Cov(x, z) değerini vermektedir. Bu değeri şöyle sinyayabiliriz:

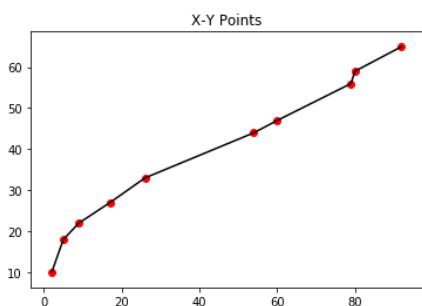
```
result = np.sum((x - np.mean(x)) * (z - np.mean(z))) / (len(x) - 1)
print(result)
```

Kovaryans iki değişkenin birlikte değişimlerinin biçimini anlamakta kullanılıktadır. Yani değişkenlerden birisi artarken diğerine ne olmaktadır? İki değişkenin kovaryansları pozitif bir değerde ise bunların değişimleri aynı yöndedir. Yani birisi artarken diğer de artmaktadır. Kovaryans negatif ise bunların değişimleri farklı yönlerdedir. Yani biri artarken diğer azalmaktadır. Aşağıdaki x, y noktalarının grafiğini çizelim:

```
x = [2, 5, 9, 17, 26, 54, 60, 79, 80, 92]
y = [10, 18, 22, 27, 33, 44, 47, 56, 59, 65]
```

```
import matplotlib.pyplot as plt
```

```
plt.title('X-Y Points')
plt.plot(x, y, c='black')
plt.scatter(x, y, c='red')
```



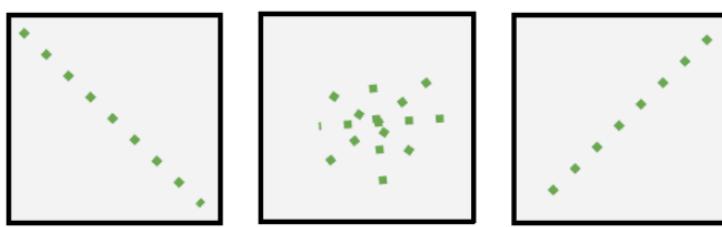
Şimdi Cov(x,y) değerine bakalım:

```
cov = np.cov([x,y])
print(cov)
```

Sonuç şöyledir:

```
[[1190.93333333 643.4
 [ 643.4 355.21111111]]]
```

Göründüğü gibi Cov(x, y) = 643.4 gibi pozitif bir değerdedir. O halde x artarken y de artmaktadır. Eğer iki değişken arasında artmalı azalmalı bir ilişki yoksa kovaryans değeri 0'a yaklaşmaktadır. Kovaryans değerinin yüksekliği x ve y arasında artış ilişkisinin bağlantısı ile ilgilidir. Ancak bu değer doğrudan yorumlanabilecek bir değer değildir.



Large Negative Covariance

Nearly Zero Covariance

Large Positive Covariance

Şekil <https://www.geeksforgeeks.org/mathematics-covariance-and-correlation/> makalesinden alınmıştır.

Korelasyon kovaryansın normalize edilmiş bir biçimidir. Korelasyon da tipki kovaryans gibi iki değişken arasındaki ilişkiye açığa çıkartır. Ancak korelasyonda değerler -1 ile +1 arasında normalize edilmiştir. Yani bir değişken

yükselirken diğer de benzer biçimde yükseliyorsa bunların arasındaki korelasyon 1'e yaklaşır fakat bir değişken yükselirken diğer düşüyorsa bunlar arasındaki korelasyon -1'e yaklaşmaktadır. Korelasyonun 0 civarında olması demek iki değişken arasında ilişkinin çok zayıf olması demektir. Tabii iki değişken arasında korelasyon olması bu iki değişkenin neden sonuç ilişkisi içerisinde olduğu anlamına gelmemektedir. Örneğin dondurma yeme miktarı ile boğulma miktarı arasında pozitif bir korelasyon bulunuyor olabilir. Ancak bu durum dondurma yemenin boğulmaya yol açacağı anlamına gelmez. Korelasyon katsayısının hesaplanması için değişik yöntemler kullanılabilmektedir. Ancak en yaygın kullanılan yöntem "Pearson korelasyon katsayı yöntemi"dir. Bu yöntem aslında iki değişkenin kovaryanslarının bu iki değişkenin standart sapmalarının çarpımı bölümü ile hesaplanır.

$$\text{Cor}(x,y) = \frac{\text{Cov}(x,y)}{\text{std}(x) * \text{std}(y)}$$

Numpy'da korelasyon katsayısı corrcoef isimli fonksiyonla hesaplanmaktadır. Bu fonksiyon bize yine matrisel bir sonuç verir. Örneğin:

```
x = [2, 5, 9, 17, 26, 54, 60, 79, 80, 92]
y = [10, 18, 22, 27, 33, 44, 47, 56, 59, 65]
```

```
import matplotlib.pyplot as plt

plt.title('X-Y Points')
plt.plot(x, y, c='black')
plt.scatter(x, y, c='red')

corr = np.corrcoef([x, y])
print(corr)
```

Burada x ve y'nin korelasyonlarının 0.98 gibi çok yüksek bir değerde olduğunu görüyoruz. Şimdi aynı işlemi kovaryans hesabı ile yapalım:

```
cov = np.cov([x, y])
corr = cov / (np.std(x, ddof=1) * np.std(y, ddof=1))
print(corr)
```

Korelasyon matrisinde matrisin ana köşegenindeki değerinin 1 olmadığını dikkat ediniz. Bu değerlerin 1 olması için bölümde aynı değişkenin standart sapmasının kullanılması gereklidir. Sosyal bilimlerde iki değişkenin arasındaki korelasyon için şunlar söylemektedir:

- 0 - 0.2 ise çok zayıf ilişki ya da korelasyon yok
- 0.2-0.4 arasında ise zayıf korelasyon
- 0.4-0.6 arasında ise orta şiddette korelasyon
- 0.6-0.8 arasında ise yüksek korelasyon
- 0.8 - 1 > ise çok yüksek korelasyon

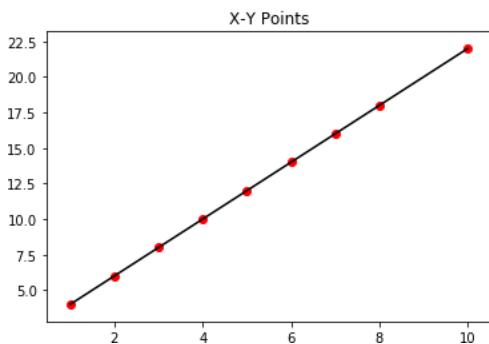
Doğrusal bir ilişkinin mükemmel bir ilişki olduğunu söyleyebiliriz. Örneğin:

```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 10], dtype=np.float32)
y = 2 * x + 2

import matplotlib.pyplot as plt

plt.title('X-Y Points')
plt.plot(x, y, c='black')
plt.scatter(x, y, c='red')

cor = np.corrcoef([x, y])
print(cor)
```



Elde edilen korelasyon sonucu şöyledir:

```
[[1. 1.]
 [1. 1.]]
```

Örneğin:

```
import numpy as np

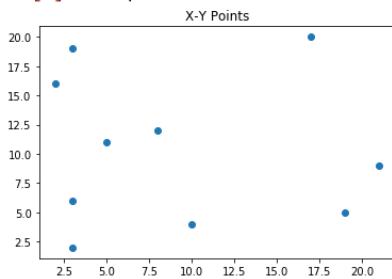
x = np.array([3, 5, 8, 3, 17, 19, 3, 2, 10, 21])
y = np.array([2, 11, 12, 19, 20, 5, 6, 16, 4, 9])

cor = np.corrcoef([x, y])
print(cor)

import matplotlib.pyplot as plt

plt.title('X-Y Points')
plt.scatter(x, y)
```

```
[[ 1.          -0.04398404]
 [-0.04398404  1.        ]]
Out[4]: <matplotlib.collections.PathCollection at 0x23ed6d11bc8>
```



Buradaki grafikten x ve y arasındaki korelasyonun çok düşük olduğu zaten görülmektedir. Örneğin:

```
x = np.array([3, 8, 10, 14, 23, 62, 71, 78, 80, 82])
y = np.array([2, 11, 12, 19, 20, 30, 40, 42, 47, 49])

cor = np.corrcoef([x, y])
print(cor)

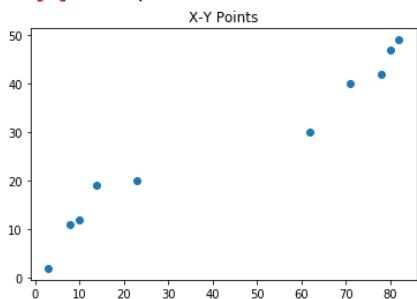
import matplotlib.pyplot as plt

plt.title('X-Y Points')
plt.scatter(x, y)
```

```

[[1.          0.97306728]
 [0.97306728 1.          ]]
Out[5]: <matplotlib.collections.PathCollection at 0x23ed6d88408>

```



Buradaki grafikten de x ve y arasında kuvvetli bir korelasyon olduğu anlaşılmaktadır.

Numpy'da Lineer Cebir İşlemleri

Numpy'da lineer cebir işlemleri için doğrudan numpy içerisindeki bazı fonksiyonlar ve linalg modülündeki fonksiyonlar kullanılmaktadır. Önemli fonksiyonlar şunlardır:

- numpy.dot isimli fonksiyon "dot product" işlemi yapmaktadır. Örneğin:

```

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

c = np.dot(a, b)
print(c)

```

- numpy.matmul fonksiyonu matris çarpması yapmaktadır. Örneğin:

```

import numpy as np

a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype='float32')
b = np.array([[1, 1, 2], [2, 2, 3], [4, 5, 2]], dtype='float32')

c = np.matmul(a, b)
print(c)

[[17. 20. 14.]
 [38. 44. 35.]
 [59. 68. 56.]]

```

- numpy.linalg.det fonksiyonu determinant hesaplamakta kullanılır. Örneğin:

```

import numpy as np

a = np.array([[1, 20, 3], [4, 50, 6], [7, 12, 9]], dtype='float32')
c = np.linalg.det(a)
print(c)

```

- numpy.linalg.inv fonksiyonu matris tersi almakta kullanılır. Örneğin:

```

import numpy as np

a = np.array([[1, 2, 3], [3, 2, 4], [1, 5, 3]], dtype=np.float32)
b = np.linalg.inv(a)
print(b)
print()
c = np.matmul(a, b)
print(c)

```

```

[[ -9.3333334e-01  6.0000002e-01  1.3333334e-01]
 [ -3.3333334e-01  1.2810266e-18  3.3333334e-01]
 [  8.6666667e-01 -2.0000000e-01 -2.6666668e-01]]]

[[ 1.0000000e+00  1.4901161e-08 -2.9802322e-08]
 [ 0.0000000e+00  1.0000000e+00  0.0000000e+00]
 [-1.1920929e-07  1.4901161e-08  1.0000000e+00]]

```

- numpy.linalg.solve fonksiyonu lineer denklem sistemini çözmektedir. Örneğin:

$$\begin{aligned} 3x_1 + 5x_2 - x_3 &= 10 \\ -x_1 - 3x_2 + x_3 &= 8 \\ x_1 + x_2 + x_3 &= 6 \end{aligned}$$

```

import numpy as np

a = np.array([[3, 2, -5], [2, -2, 7], [1, 1, 1]], dtype='float32')
b = np.array([10, 12, 5])
c = np.linalg.solve(a, b)
print(c)

```

```

[[1.]
 [2.]
 [3.]]

```

Öz Değerler ve Öz Vektörler (Eigen Values and Eigen Vectors)

Öz değerler ve öz vektörler konusu pek çok alanda kullanılmaktadır. Öz değer ve öz vektör bir kare matrise dayalı olarak hesaplanmaktadır. Hesaplama için genel eşitlik öyledir:

$$AX = \lambda X$$

Burada A ilgili kare matrisi belirtmektedir. X bir sütun vektördür. Bu X vektörüne öz vektör denir. Lambda ise bir skalerdir. Bu lambda skalerine de öz değer denilmektedir. Buradaki eşitliğin daha belirgin gösterimi şöyle yapılabilir:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix} = \lambda \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix},$$

Yukarıdaki eşitlikten şöyle bir anlam çıkmaktadır: Biz bir vektörü (eşitlikteki x) bir matrisle çarptığımızda yine bir vektör elde ederiz. Ancak elde ettiğimiz vektör ilkiyle aynı doğrultuda yalnızca büyülüklük olarak farklıdır. Yani biz bir vektörü bir matrisle çarptığımızda vektör doğrultu değiştirmemektedir. Yalnızca vektörün büyülüğü değişmemektedir. Tabii biz bir vektörü her türlü matrisle çarparsak böyle bir sonuç elde edemeyiz. Bu vektörün matrise uygun seçilmesi gereklidir. İşte bu vektöre matrisin öz vektörü buradaki skaler lambda değerine de matrisin özdeğeri denilmektedir. Yalnızca kare matrislerin öz değerleri ve öz vektörleri vardır. Ayrıca her kare matrisin de öz değer ve öz vektörü olmak zorunda değildir. Genel olarak nxn'lik bir kare matrisin n tane öz değeri ve öz vektörü vardır.

Öz değerler ve öz vektörler lineer cebirde aşağıdaki biçimde elde edilmektedir. Bu yöntemde önce Lambda değerlerini (yani öz değerleri) buluruz.

$$\begin{aligned}
 Ax &= \lambda x \\
 A x - \lambda x &= \emptyset \\
 (A - \lambda I) x &= \emptyset \\
 (A - \lambda I) x &= \emptyset \\
 A - \lambda I &= \emptyset
 \end{aligned}$$

Bu denklem aslında A $n \times n$ 'lik bir kare matris olmak üzere n 'inci dereceden bir denklem oluşturur. Bu n 'inci dereceden denklemin de n tane kökü vardır. Örneğin aşağıdaki matris için lambda öz değerlerini bulmak isteyelim:

$$A = \begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix}$$

Örneğin:

$$\begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \emptyset$$

$$\begin{bmatrix} 2-\lambda & 3 \\ 2 & 1-\lambda \end{bmatrix} = \emptyset$$

$$(2-\lambda)(1-\lambda) - 6 = \emptyset$$

$$2 - 2\lambda - \lambda + \lambda^2 - 6 = \emptyset$$

$$\lambda^2 - 3\lambda - 4 = \emptyset$$

$$\lambda = \{-1, 4\}$$

Göründüğü gibi buradan öz değerler 4 ve -1 biçiminde bulunmuştur. Aslında biz ödeğer bulma işlemini numpy.linalg modülündeki eigvals fonksiyonuyla da yapabiliyoruz:

```

>>> import numpy as np
>>> a = np.array([[2, 3], [2, 1]])
>>> np.linalg.eigvals(a)
array([ 4., -1.])
    
```

Pekiyi öz vektörler nasıl elde edilmektedir? İşte elimizde öz değerler (yani lambda'lar) varsa biz buradan öz vektörleri elde edebiliriz:

$$\begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \lambda \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$2x_1 + 3x_2 = \lambda x_1$$

$$2x_1 + x_2 = \lambda x_2$$

$$(2-\lambda)x_1 + 3x_2 = \emptyset$$

$$2x_1 + (1-\lambda)x_2 = \emptyset$$

Şimdi burada lambda yerine 4 koyarak denklemi çözmeye çalışalım:

$$\begin{aligned} -2x_1 + 3x_2 &= \emptyset \\ 2x_1 - 3x_2 &= \emptyset \end{aligned}$$

Göründüğü gibi denklemin sonsuz sayıda kökü vardır. Bunlardan birisi $[3, 2]$ kökleridir. İşte $[3, 2]$ vektörü bir öz vektördür. Ama bunun gibi sonsuz sayıda öz vektör vardır. Lineer cebirde bu denklemi sağlayan bütün vektörlere öz vektör uzayı denilmektedir. Fakat pratikte biz bu sonsuz sayıda öz vektörle ilgilenmeyiz. Mademki öz vektör bizim için bir doğrultu belirtmektedir. Biz uzunluğu 1 olan normalize edilmiş öz vektörlerle işlemlerimizi yaparız. Bir vektörü normalize etmek için pisagor teoreminden hareketle şu işlemi uygularız:

$$x_1 / \sqrt{x_1^2 + x_2^2}$$

$$x_2 / \sqrt{x_1^2 + x_2^2}$$

Tabii burada nokta sayısı 3 tane olsaydı biz karekök içerisine üçüncü bileşenin de karesini ekleyecektik. Normalize edilmiş değerler şöyledir:

```
>>> 3 / np.sqrt(3 ** 2 + 2 ** 2)
0.8320502943378437
>>> 2 / np.sqrt(3 ** 2 + 2 ** 2)
0.5547001962252291
```

Şimdi aynı işlemi -1 için yapalım:

$$\begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = -1 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{aligned} 2x_1 + 3x_2 &= -x_1 \\ 2x_1 + x_2 &= -x_2 \\ 3x_1 + 3x_2 &= \emptyset \\ 2x_1 + 2x_2 &= \emptyset \end{aligned}$$

$$1\sqrt{2}, -1\sqrt{2}$$

```
>>> 1 / np.sqrt(1 ** 2 + (-1) ** 2)
0.7071067811865475
>>> -1 / np.sqrt(1 ** 2 + (-1) ** 2)
-0.7071067811865475
```

Aslında bu normalize edilmiş öz vektörleri biz Numpy'ın linalg modülündeki eig fonksiyonuyla elde edebilmekteyiz. Örneğin:

```
>>> a = np.array([[2, 3], [2, 1]])
>>> np.linalg.eig(a)
(array([ 4., -1.]), array([[ 0.83205029, -0.70710678],
   [ 0.5547002 ,  0.70710678]]))
```

eig fonksiyonun bize verdiği vektörler sütun vektörü biçimindedir. Yani yukarıdaki çözümdeki vektörler aslında şöyledir:

$$\begin{bmatrix} 0.8320512 \\ 0.5547002 \end{bmatrix} \begin{bmatrix} -0.70710678 \\ 0.70710678 \end{bmatrix}$$

Öz vektörler bir birine diktir.

BOYUTSAL ÖZELLİK İNDİRİGEMESİ (Dimensionality Feature Reduction)

Makine öğrenmesindeki veri tabloları sütunlardan oluşmaktadır. Pek çok uygulamada sütun sayısı çok fazla olabilmektedir. Ancak bu sütunlar analiz edildiğinde aslında örneğin bazı sütunun birbirleriyle ilişkili olduğu görülebilmektedir. Örneğin iki sütun söz konusu olsun. Biri diğerinin iki katı değerlere sahip olsun. Aslında öğrenme algoritmaları için bu iki sütunun aynı tabloda bulunması bir kazanç sağlamayacağı gibi "overfitting" gibi, "zaman kaybı", ve "işlem miktarı" gibi dezavantajlar doğurmaktadır. Boyutsal özellik indirgemesi demek veri tablosundaki n tane sütun yerine bu n tane sütunu temsil edebilecek k < n tane sütun oluşturmak demektir. Örneğin üç sütunlu bir tabloda bu üç sütun yerine bu üç sütunu temsil edeceğine inandığımız iki sütun oluşturabiliriz.

Boyutsal özellik indirgemesi aynı zamanda çok sütunlu öğelerin grafiklerinin çizilmesinde de tercih edilmektedir. İnsan olarak bizler iki değişkenli yani iki boyutlu grafikleri güzel bir biçimde algılayabilmekteyiz. Çünkü iki boyutlu grafikler kağıt üzerinde görüntülenebilmektedir. Üç boyutlu grafiklerin kağıt üzerinde görüntülenmesi mümkün olabiliyorsa da çok zahmetlidir. Üç boyuttan yüksek boyuta sahip olan veri tablolarının grafikleri çizilemez. İşte özellik indirgemesi yardımıyla örneğin biz 4 boyutlu (sütunlu) bir veri tablosunu iki boyuta indirgeyip onun grafiğini iki boyutlu olarak çizebiliriz.

Özetlersek boyutsal özellik indirgemesi makine öğrenmesinde şu amaçlarla kullanılmaktadır:

- Veri tablosundaki sütunların (özelliklerin) sayısını azaltarak hesaplamalar için gereken zamanı ve bellek alanını azaltmak yani performans kazancı sağlamak.
- Overfitting durumunu azaltmak
- Veri tablosunun grafiksel olarak görüntülenmesini sağlamak.

Boyutsal veri indirgemesi için pek çok yöntem kullanılabilmektedir. Örneğin:

- Missing Value Ratio
- Low Variance Filter
- High Correlation Filter
- Random Forest
- Backward Feature Elimination
- Forward Feature Selection
- Factor Analysis
- Principal Component Analysis
- Independent Component Analysis
- Methods Based on Projections
- t-Distributed Stochastic Neighbor Embedding (t-SNE)
- UMAP

Bu yöntemlerden en fazla kullanılan temel bileşenler analizi (principal component analysis) dir.

Boyutsal özellik indirgemesi yöntemleri çeşitli bakımlardan gruplandırılabilmektedir. Biz burada bu yöntemleri iki gruba ayıracagız:

1) Orijinal n tane sütundan bazı sütunları atarak k tane ($k < n$) sütun elde etmeye çalışan yöntemler.

2) Orijinal n tane sütunun yerine bunlarla ilişkili olan yeni k tane sütun ($k < n$) elde etmeye çalışan yöntemler.

Birinci grup yöntemlerde n tane sütundaki bazı sütunlar çeşitli istatistiksel ölçütler temelinde atılmaktadır. Kalan sütunlar orijinal tablodaki k tane sütundur. İkinci grup yöntemlerde n tane sütunun tamamı başka bir k tane sütunla değiştirilmektedir. Örneğin temel bileşenler analizi (principal component analysis) ikinci grup yöntemlere bir örnektir. Temel bileşenler analizinde biz n tane sütun yerine onları temsil edecek k tane ($k < n$) yeni sütunlar elde ederiz. Şimdi bu iki grup yöntemlerden bazlarını kısaca burada tanıtacağız. Ancak yukarıda da belirtildiği gibi bu alanda en çok tercih edilen yöntem "temel bileşenler analizi (principal component analysis)" yöntemidir. Bu yöntem üzerinde ayrı bir başlıkta daha geniş duracağız.

Eksik Değerli Sütunların Atılması Yöntemi (Missing Value Ratio): Veri tablosundaki bazı sütunlar eksik veriler içerebilmektedir. Örneğin çok kişisel ya da politik birtakım soruların yanıtlarını insanlar vermek istemeyebilirler. Bu durumda bu sorulara ilişkin sütunlarda bilgi eksikliği bulunabilir. İşte bilgi eksikliği belirli bir oranda olan sütunlar tümenden atılıp boyutssal bir özellik indirgemesi oluşturulabilmektedir.

Düşük Varyans Filtrelemesi (Low Variance Filtering): Veri tablosundaki bir sütundaki bilgilerin hep aynı olduğunu düşünelim. Böyle bir sütunun tabloda bulunmasının bir faydası olabilir mi? Tabii ki olmaz. Tüm değerleri aynı olan sütunun varyansı 0'dır. Demekki bir sütunun kestirimde bir faydasının olup olmaması o sütunun varyansıyla da ilgilidir. İşte bu yöntemde bir eşik değeri belirleyip varyansı düşük olan sütunlar tablodan atılabilir. Tabii varyanslar için bir eşik değerinin oluşturulması ayrı bir problemdir. Bu tür durumlarda işlemlerin kolay yürütülebilmesi için önce bir ölçeklendirme (normalizasyon) yapılabilmektedir. Örneğin elimizde aşağıdaki gibi bir data.csv dosyası bulunuyor olsun:

```
10,180,3,1345  
10,100,3,3456  
13,125,2,2340  
12,200,9,5250  
10.01,170,7,1980  
10,160,8,2900  
10,120,5,5200
```

Şimdi biz sütunları MinMax ölçeklendirmesine göre ölçeklendirip varyanslarına bakalım:

```
import numpy as np  
  
dataset = np.loadtxt('data.csv', delimiter=',', dtype=np.float32)  
  
from sklearn.preprocessing import MinMaxScaler  
  
mms = MinMaxScaler()  
  
transformed_dataset = mms.fit_transform(dataset)  
col_vars = np.var(transformed_dataset, axis=0)  
print(col_vars)
```

Aşağıdaki gibi bir sonuç elde edilmiştir:

```
[0.14943445 0.11316326 0.13244483 0.1314027 ]
```

Burada atılmaaya en uygun sütun birinci indeksli en küçük varyansa sahip sütundur. Tabii iki sütun atmak isersek birinci ve üçüncü indeksli sütunları atmalıyız. Örneğin:

```
reduced_dataset = np.delete(transformed_dataset, [1, 3], axis=1)  
print(reduced_dataset)
```

```

[[0.          0.14285716]
 [0.          0.14285716]
 [1.          0.          ]
 [0.66666665  1.0000001 ]
 [0.003333333 0.71428573]
 [0.          0.8571429 ]
 [0.          0.42857143]]

```

Bu yöntem scikit-learn kütüphanesinde VarianceThreshold isimli sınıf ile gerçekleştirilmiştir. Bu fonksiyon 0 ile 1 arasında elimine edilecek sütunların düşük varyans limitini parametre olarak almaktadır. Daha sonra elde edilen sınıf nesnesi ile fit ve transform işlemleri yapılabilir. (Bu işlemler tek adımda fit_transform metoduyla da yapılabilmektedir.) Örneğin:

```

import numpy as np

dataset = np.loadtxt('data.csv', delimiter=',', dtype=np.float32)
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
transformed_dataset = mms.fit_transform(dataset)

from sklearn.feature_selection import VarianceThreshold

vt = VarianceThreshold(0.13)
reduced_dataset = vt.fit_transform(transformed_dataset)
print(vt.variances_)
print(reduced_dataset)

[0.14943445 0.11316326 0.13244484 0.1314027 ]
[[0.          0.14285716 0.          ]
 [0.          0.14285716 0.540589  ]
 [1.          0.          0.25480154]
 [0.66666665  1.0000001  1.          ]
 [0.003333333 0.71428573 0.16261205]
 [0.          0.8571429  0.39820746]
 [0.          0.42857143 0.98719597]]

```

VarianceThreshold sınıfının kendisi ölçeklendirme yapmamaktadır. Dolayısıyla düşük varyanslı sütunların karşılaşmalıdır olarak elimine edilmesi için bizim ölçeklendirme yaptıktan sonra bu sınıfı kullanmamız gereklidir.

Yüksek Korelasyon Filtrelemesi Yöntemi (High Correlation Filtering): Yüksek korelasyona sahip iki sütunun aynı tabloda bulunması makine öğrenmesi modellerinde önemli bir fayda sağlamamaktadır. Örneğin bir sütun diğer sütunun iki katından dört fazla olsun. Bu durumda bu iki sütun arasında doğrusal bir ilişki vardır. Dolayısıyla bu iki sütunun korelasyon katsayısı 1'dir. Bu iki sütunun tablomuzda bir arada bulunmasının kestirimler için bir faydası yoktur. Bunlardan biri tablodan atılabilir. Yüksek korelasyon demekle ne kastedildiği veri bilimcisine bağlıdır. Genellikle burada kastedilen 0.90 ve yukarısıdır. Bu yöntemi Python'da şöyle uygulayabiliriz. Aşağıdaki gibi bir "data.csv" dosyası bulunuyor olsun:

```

10,180,3,302,20,21
10,1790,3,205,20.1,20.2
26,600,2,200,26.2,53
12,1925,9,920,24.1,25
10.1,192,7,710,20.100,20.3
18,28,8,820,20,27
10,890,5,520,20,21

import numpy as np

dataset = np.loadtxt('data.csv', delimiter=',', dtype=np.float32)

```

```

corr = np.corrcoef(dataset, rowvar=False)

reduced_features = []

for i in range(corr.shape[0]):
    for k in range(corr.shape[1]):
        if i != k and i not in reduced_features and np.abs(corr[i, k]) > 0.90:
            reduced_features.append(i)

print(reduced_features)

reduced_dataset = np.zeros((dataset.shape[0], len(reduced_features)))

for i, col in enumerate(reduced_features):
    reduced_dataset[:, i] = dataset[:, col]

print(reduced_dataset)

```

Burada reduced_features listesi içerisindeki sütun numaraları indirgenmiş olan sütunları belirtmektedir. Yani biz buradaki reduced_features sütunlarını alıp bunlardan yeni bir dataset oluşturmalıyız. Yukarıdaki programdan elde edilen sonuçlar şöyledir:

```

[0, 2, 3, 5]
[[ 10.      3.      302.      21.      ]
 [ 10.      3.      205.      20.20000076]
 [ 26.      2.      200.      53.      ]
 [ 12.      9.      920.      25.      ]
 [ 10.10000038 7.      710.      20.29999924]
 [ 18.      8.      820.      27.      ]
 [ 10.      5.      520.      21.      ]]

```

Göründüğü gibi orijinal dataset'teki iki sütun diğer iki sütunla yüksek bir korelasyona sahip olduğu için atılmıştır. scikit-learn kütüphanesinde doğrudan yüksek varyans indirmesini yapan bir sınıf ya da fonksiyon bulunmaktadır.

Rassal Ormalar Yöntemi (Random Forests): Rassal ormanlar yöntemi kursumuzda ilerideki konularda ele alınmaktadır.

Geriye Doğru Özellik İndirmesi Yöntemi (Backward Feature Elimination): Bu yöntem eğitimli öğrenme modellerinde uygulanabilmektedir. Bu yöntemde n tane sütun ile modelin başarısı test edilir. Sonra sütunlardan bir tanesi atılarak yeniden modelin başarısına bakılır. Eğer sütun atıldıktan sonra modelin başarısı çok değişmemişse bu sütunun gerçekten atılabilecek bir sütun olduğunu karar verilir. Bu biçimde tüm sütunlar gözden geçirilir. Tabii bu yöntem çok uzun bir bilgisayar zamanına mal olmaktadır. Bu nedenle pratikte pek tercih edilmemektedir.

İleriye Doğru Özellik İndirmesi Yöntemi (Forward Feature Elimination): Bu yöntem geriye doğru özellik indirmesi yönteminin tersidir. Yani modelin başarısı tek bir sütunla başlatılır. Sonra başarıyı dah çok yükselten sütun dahil edilir. Tabii yöntem de eğitimli öğrenmede kullanılabilecek bir yöntemdir. Yine bu yöntem de yüksek bir bilgisayar zamanına yol açmaktadır.

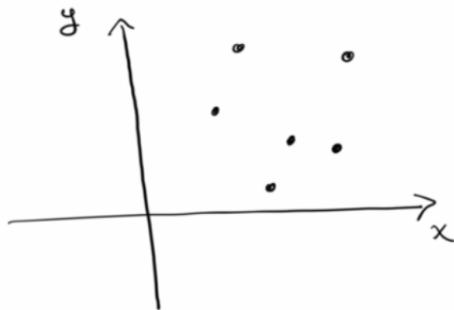
Faktör Analizi Yöntemi (Factor Analysis): Faktör analizi istatistikte özellikle sosyal bilimlerde çok kullanılan yöntemlerden biridir. Faktör analizinde bir olguyu açıklayabilecek n tane özellik aralarındaki kovaryanslara göre k tane özelliğe indirgenmektedir. Yani birbirlerine benzeyen özellikler atılarak onlar yerine o özellikleri temsil eden bir özellik sınıfı oluşturulmaktadır. Faktör analizi ayrı ve önemli bir konudur. Temel Bileşenler Analizi yöntemi de aslında bir bakıma faktör analizi yöntemi olarak değerlendirilebilmektedir.

Biz kursumuzda boyutsal özellik indirmesi yöntemi olarak "temel bileşenler analizi (principal component analysis) üzerinde ağırlıklı biçimde duracağz. Bu yöntem izleyen başlıkta açıklanmaktadır.

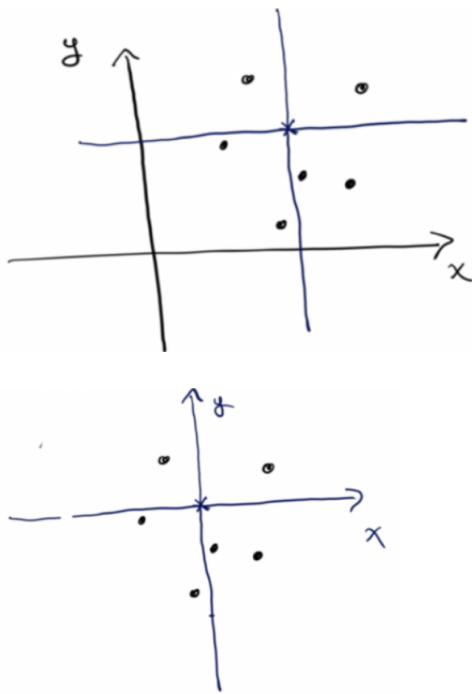
Temel Bileşenler Analizi (Principal Component Analysis)

Yukarıda da belirtildiği gibi boyutsal özellik indirgemesi (feature reduction) için en yaygın kullanılan yöntem "temel bileşenler analizi"dir. Numpy'da temel bileşenler analizini uygulayan yetenekli bir sınıf vardır. Bu sınıf sayesinde aslında veri bilimcisi burada açıklanacak manuel hesapları yapmak zorunda kalmaz. Ancak biz burada temel bileşen analizinin manuel bir biçimde nasıl yapıldığı üzerinde biraz duracağız.

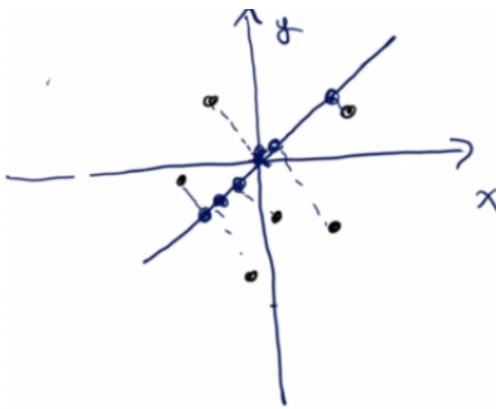
Örneğin iki sütunlu bir veri tablomuz olsun ve bu veri tablosunu tek sütuna indirmek isteyelim. Bu işlemi nasıl yapabiliriz? Buradaki temel mantık iki sütunlu (yani iki bileşene sahip) veri yerine onu temsil edebilecek tek sütunlu (yani tek bileşene sahip) bir veri tablosu oluşturmaktır. Bu yapılrken de orijinal noktaların yeni eksendeki projeksiyonlarının varyansının yüksek tutulması gerekmektedir. Örneğin:



Biz burada x ve y değerlerine sahip iki sütunlu bilgilerin nokta grafiğini çizdik. Amacımız da bu iki sütunlu bilgi yerine tek sütunlu (yani tek eksenli) değerleri oluşturmak. Bunun için uygun bir eksenin seçilmesi gereklidir. İlk yapılacak şey eksenlerin orta noktaya ötelebilmesidir. Bu ötemele sonucunda grafiğin şu hale geldiğini varsayıyalım:



Bundan sonra iki boyutlu noktaların tek boyutlu eksende oluşturulması için (buna "noktaların projekte edilmesi" denilmektedir) uygun bir eksenin seçilmesi gereklidir. Orijin noktasından geçen sonsuz sayıda eksen olabilir. Bizim seçeceğimiz eksenin söyle bir özelliği olmalıdır: Noktalar bu eksene projekte edildiğinde toplam varyansların en yüksek olması istenir. Yani noktaların orijine olan uzaklıklarının toplamının maksimum olması arzu edilmektedir. Böylece iki boyutlu bilginin daha az kayıpla tek boyuta indirgenmesi sağlanmış olur. Aşağıda böyle bir eksen çizilmiştir:



Pekiyi böyle bir doğrunun denklemi nedir ve projeksiyon bu doğru üzerine nasıl yapılacaktır? İşte bu işlemler daha önce görmüş olduğumuz kovaryans matrisi ve öz değer vektörleri kullanılarak yapılmaktadır.

Örneğin N tane sütuna sahip bir veri tablosundan k tane sütuna sahip ($k < n$) bir veri tablosunu temel bileşenler analizi yöntemiyle elde etmek isteyelim. İşlem adımları şöyle gerçekleştirilir:

- 1) N sütunlu veriler üzerinde özellik ölçeklendirmesi (normalizasyon) uygulanır.
- 2) N sütunlu matristen $N \times N'$ lik kovaryans matrisi elde edilir.
- 3) Bu kovaryans matrisinden N tane öz vektör bulunur.
- 4) Bu N tane özvektör arasından k tane'si seçilerek (seçimin nasıl yapılacağı belirtilecektir) asıl matrisle çarpılır ve böylece sonuçta k tane sütuna indirgenmiş veri tablosu elde edilir.

Şimdi bu işlemleri adım adım Python'da yapalım. Bu örneğimizde iki sütunlu tabloyu tek sütuna indirmeye çalışalım. İki sutunlu tablonun bilgileri şöyle olsun:

x1	x2
0.72	0.13
0.18	0.23
2.5	2.3
0.45	0.16
0.04	0.44
0.13	0.24
0.30	0.03
2.65	2.1
0.91	0.91
0.46	0.32

Bu bilgiler "data.csv" dosyası içerisinde bulunmaktadır.

Şimdi ilk yapılacak şey normalize etmek yani orijin noktasını değerlerin ortasına kaydırmaktadır. Bu işlem şöyle yapılabilir:

```
import numpy as np

dataset = np.loadtxt('data.csv', delimiter=',', dtype=np.float32)
dataset = dataset - np.mean(dataset, axis=0)
```

Şimdi bu normalize edilmiş 2 boyutlu tablodan 2×2 'lik kovaryans matrisi elde edilir:

```
cmat = np.cov(dataset, rowvar=False)
print(cmat)
```

Elde edilen covaryans matrisi şöyledir:

```
array([[0.91316003, 0.75931776],  
       [0.75931776, 0.69689329]])
```

Bundan sonra bu kovaryans matrisinin öz değerleri ve öz vektörleri bulunur:

```
evals, evecs = np.linalg.eig(cmat)  
print(evals)  
print(evecs)
```

Elde edilen öz değerler ve öz vektörler şöyledir:

```
[1.57200534 0.03804799]  
[[ 0.75530992 -0.65536778]  
 [ 0.65536778  0.75530992]]
```

Şimdi bizim projeksiyon işlemini yapmamız gereklidir. Biz asıl ölçeklendirilmiş asıl matrisimizi (biz bu örnekte ölçeklendirme yapmadık, çünkü gerekmemi) nxn'lik özvektör matrisinin k tanesiyle çarptığımızda artık k tane sütunlu bir matris elde ederiz. Buradaki amacımız iki sütunu tek sütuna indirgemekti. Demek ki biz tek bir özvektörle çarpma yaparak tek sütunumuzu elde edeceğiz. Pekiyi n tane öz vektör arasından hangi k tane özvektörü bu çarpma işlemeye sokmalıyız? İşte öz değeri yüksek vektörlerin bu işlem için seçilmesi gerekmektedir. Çünkü öz değeri yüksek olan öz vektörler daha yüksek varyansa yol açmaktadır. Yukarıdaki örneğimizde birinci özvektör (1.57 olan) diğerinden daha yüksektir. O halde biz birinci öz vektörü asıl matrisle çarpmalıyız:

```
reduced_dataset = np.matmul(dataset, evecs[:, 0].reshape((-1, 1)))
```

Elde edilen değerler şöyledir:

```
[[ -0.45048978]  
 [-0.79282036]  
 [ 2.3161099 ]  
 [-0.63476248]  
 [-0.76093652]  
 [-0.8240322 ]  
 [-0.83325676]  
 [ 2.29833288]  
 [ 0.20420599]  
 [-0.5223505 ]]
```

Aslında temel bileşenler analizi scikit-learn kütüphanesinde decomposition modülündeki PCA sınıfıyla kolay bir biçimde yapılabilmektedir. PCA sınıfının `__init__` metodunun parametrik yapısı şöyledir:

```
sklearn.decomposition.PCA(n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.  
0, iterated_power='auto', random_state=None)
```

Buradaki `n_components` indirgenme sonucunda elde edilecek sütun sayısını belirtmektedir. Diğer parametrelerin çok önemi yoktur.

PCA sınıfının `fit` metodunu gerçek hesaplamayı yapmaktadır. Veri tablosu bu metoda parametre olarak girilir. Ancak `fit` metodunu indirgenmiş değerleri bize vermez. (Bu metot PCA nesnesinin yine kendisini bize vermektedir.) İndirgenmiş değerleri almak için ayrıca PCA sınıfının `transform` metodunun çağrılması gerekmektedir. Fakat `fit_transform` isimli metod bu ikisini zaten bir arada yapmaktadır. `inverse_transform` metod ise işlemin tersini yapmaktadır. Yani bu metod az sütunlu bir tabloyu çok sütunlu hale getirmektedir. Bu işlemin amaçsız olduğu düşünülebilir. Ancak bazı uygulamalarda (örneğin anomalileri tespit etme uygulamalarında) bu metod kullanılmaktadır. Sınıfın iki önemli özniteligi `explained_variance_` ve `explained_varience_ratio` öznitelikleridir. Bu öznitelikler belirlenen sütun sayısına göre asıl tablonun varyansının ne kadarının indirgenmiş tabloya yansıtıldığını belirtmektedir. `explained_variance_` bir değer olarak bunu verirken `explained_variance_ratio` bir yüzde olarak bunu vermektedir.

Şimdi yukarıda elle yaptığımız örneği PCA sınıfı ile yapalım:

```
import numpy as np

dataset = np.loadtxt('data.csv', delimiter=',', dtype=np.float32)
print(dataset, '\n')

from sklearn.decomposition import PCA

pca = PCA(n_components=1)
reduced_dataset = pca.fit_transform(dataset)
print(reduced_dataset, '\n')
print('Explained variance: {}'.format(pca.explained_variance_))
print('Explained variance ratio: {}'.format(pca.explained_variance_ratio_))
```

Burada elde edilen sonuçlar şöyledir:

```
[[ 0.72  0.13]
 [ 0.18  0.23]
 [ 2.5   2.3 ]
 [ 0.45  0.16]
 [ 0.04  0.44]
 [ 0.13  0.24]
 [ 0.3   0.03]
 [ 2.65  2.1 ]
 [ 0.91  0.91]
 [ 0.46  0.32]]

[[ -0.45048997]
 [ -0.7928204 ]
 [  2.31611   ]
 [ -0.6347626 ]
 [ -0.7609365 ]
 [ -0.82403225]
 [ -0.8332568 ]
 [  2.298333  ]
 [  0.20420602]
 [ -0.52235055]]
```

```
Explained variance: [1.572005]
Explained variance ratio: [0.9763685]
```

Açıklanan varyans oranının %97 civarında olduğu görülmektedir. Bu %97 değeri kabaca şu anlamaya gelmektedir: Biz iki sütunlu bu tabloyu tek sütuna indirgediğimizde yalnızca %3'lük bir kayıp oluşturduk. Tabii şüphesiz indirgeme sonrasında elde edilecek sütun sayısı azaltıldıkça bu oran düşecektir. Bu oranın iyi bir noktada kalacağı biçimde asıl tablonun kaç sütuna indirgenmesi gereğiği izleyen başlıkta ele alınmaktadır.

Anımsanacağı gibi boyutsal özellik indirgemesi çok boyutlu olguların iki ya da üç boyuta indirgenerek grafiklerinin çizilmesi için de kullanılıyordu. Şimdi biz Iris veritabanında sınıflandırma sonucunda elde edilen 4 bsütunlu verileri iki sütuna indirgerek grafiğini çizelim:

```
import numpy as np

from sklearn.datasets import load_iris

iris = load_iris()

dataset_x = iris.data
dataset_y = iris.target

from sklearn.cluster import KMeans

km = KMeans(n_clusters=3)
km.fit(dataset_x)
```

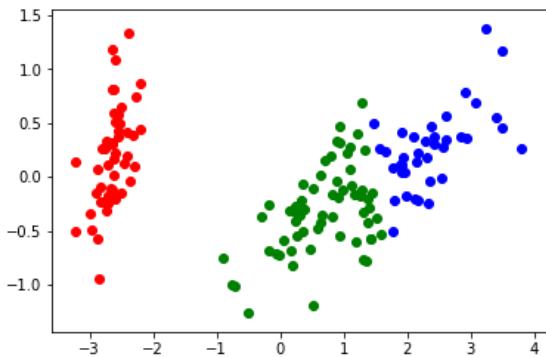
```

from sklearn.decomposition import PCA
pca = PCA(n_components=2)
reduced_dataset_x = pca.fit_transform(dataset_x)

import matplotlib.pyplot as plt

plt.scatter(reduced_dataset_x[km.labels_ == 0][:, 0], reduced_dataset_x[km.labels_ == 0][:, 1],
            color='red')
plt.scatter(reduced_dataset_x[km.labels_ == 1][:, 0], reduced_dataset_x[km.labels_ == 1][:, 1],
            color='green')
plt.scatter(reduced_dataset_x[km.labels_ == 2][:, 0], reduced_dataset_x[km.labels_ == 2][:, 1],
            color='blue')

```



Biz bu örnekte aslında 4 özelliğe (sütuna) sahip olan Iris verilerini 3 kümeye ayırdık. 4 özelliğin grafiğini çizemeyeceğimizden dolayı onu PCA yöntemiyle iki özelliğe indirgeyip grafiğini çizdik.

PCA İşleminde İndirgenecek Sütun Sayısının Belirlenmesi

Elimizde n sütunlu bir veri tablosu olsun. Biz n sütunu k sütuna indirmek isteyelim. Peki bu k değeri ne olmalıdır? Yani örneğin elimizde 100 sütunlu bir veri tablosu varsa biz bunu kaç sütuna önemli bir kayıp olmadan indirmeliyiz? Şüphesiz sezgisel olarak bu 100 sütunun 99 sütuna indirgenmesi ile 10 sütuna indirgenmesi arasında bir farklılık olduğu anlaşılabilir. Biz kabaca indirgeme ne kadar yüksek yapılıyorsa orijinal verilerin o kadar bozulacağını söyleyebiliriz. O halde "optimum indirgeme için sütun sayısını ne olmalıdır" sorusu gündeme gelmektedir. İşte bunu belirleyebilmek için iki yöntem önerilmektedir:

- 1) Açıklanan varyans oranının en çok düşüğü sütun sayısını bulmak. Yani eğrinin eğiminin atay eksene stabil olarak kaldığı noktanın başlangıcı bulmak.
- 2) Açıklanan varyansın kabaca arzu edilen yüzdede olduğu (örneğin %80) sütun sayısını bulmak.

Birinci yöntem büyük ölçüde gözle kontrol edebilecek bir yöntemdir. İkinci yöntemi programlamak çok kolaydır. İkinci yöntemdeki yüzde değeri duruma göre değiştirilebilir. Örneğin yüzde değeri %80 alınırsa bu durum "orijinal verilerdeki bozulmanın %20 düzeyinde kaldığı en çok sütunlu durumun elde edileceği" anlamına gelmektedir. Şimdi biz bu yöntemleri Mnist verilerinde kullanmaya çalışalım.

```

from tensorflow.keras.datasets import mnist

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) = mnist.load_data()
dataset_x = training_dataset_x.reshape(-1, 28 * 28)
dataset_x = dataset_x / 255

import numpy as np
from sklearn.decomposition import PCA

ratios = []
optimal_feature = None
for i in range(100):

```

```

pca = PCA(n_components=i)
pca.fit(dataset_x)
ratios.append(np.sum(pca.explained_variance_ratio_))
if not optimal_feature and ratios[-1] >= 0.80:
    optimal_feature = i
print('{}---> {}'.format(i, ratios[-1]))

import matplotlib.pyplot as plt

plt.figure(figsize=(30, 10))
plt.plot(ratios, 'r')
plt.plot(ratios, 'bs')
plt.xticks(range(1, 100))

plt.pause(1)

print(optimal_feature)

```

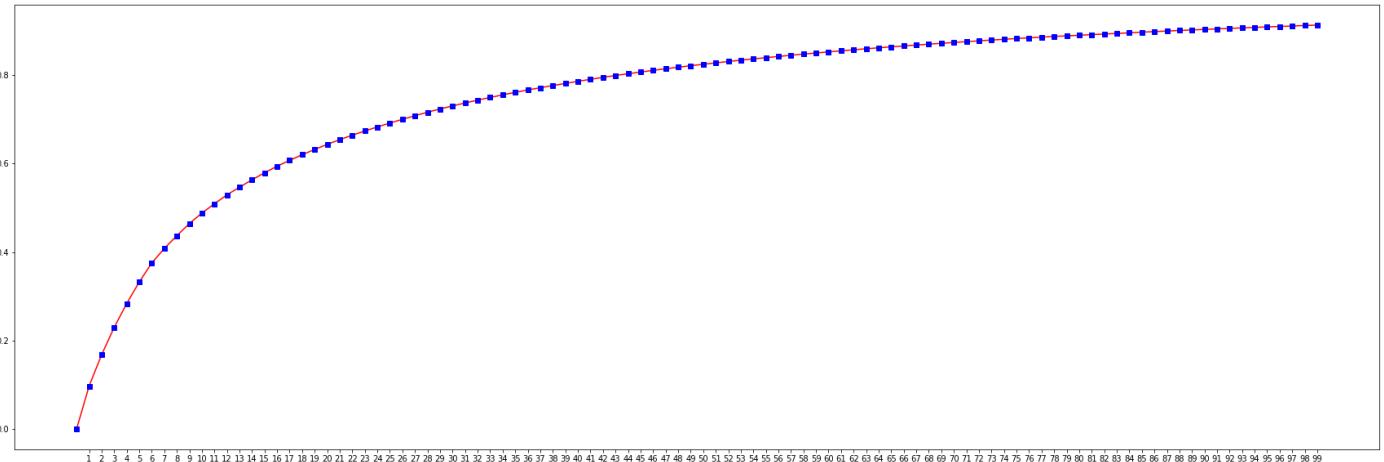
Bu programdan elde edilen veriler aşağıdaki gibidir:

```

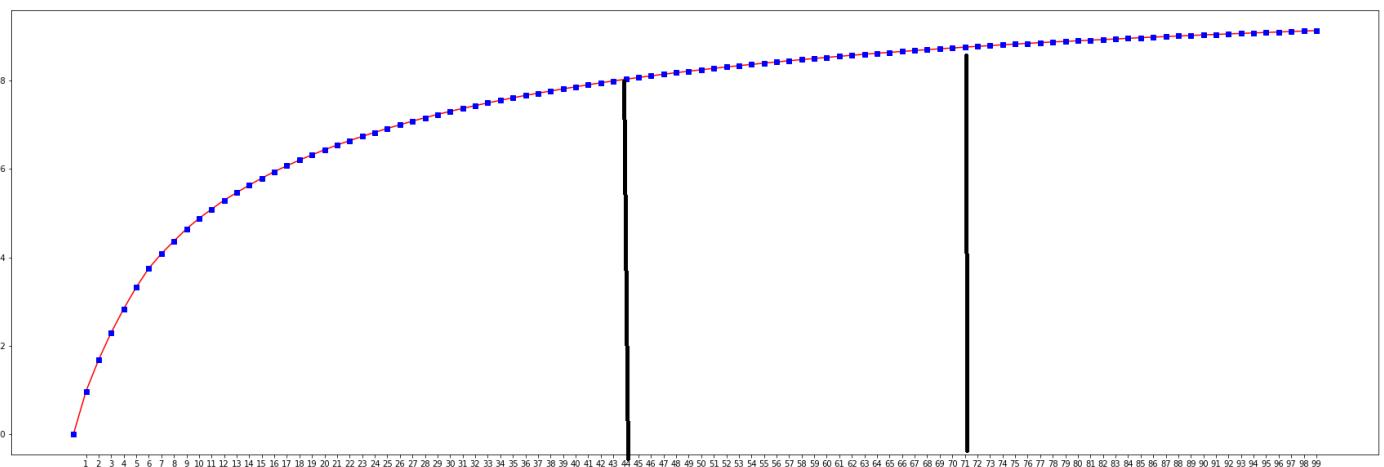
0---> 0.0
1---> 0.09704664359411821
2---> 0.16800588405555902
3---> 0.22969677170660122
4---> 0.28359096140493656
5---> 0.33227893102047446
6---> 0.3754012485672624
.....
25---> 0.6917966534434756
26---> 0.7001939613296784
27---> 0.7083190671384518
28---> 0.7161844162745432
29---> 0.723623901698142
30---> 0.7305381991707998
31---> 0.737103904740151
.....
43---> 0.7993004114124839
44---> 0.8032580490108351
45---> 0.8071143109151905
46---> 0.8108497261629446
47---> 0.81449252518249
48---> 0.8180094304518465
.....
59---> 0.850151959089285
60---> 0.8525806719701093
61---> 0.8550143536639044
62---> 0.8573916039952458
63---> 0.8597069516137059
64---> 0.8619171933388575
.....
80---> 0.8902214119703338
81---> 0.8916782708014838
82---> 0.892996569814878
83---> 0.8943441647586491
84---> 0.8957344823217974
85---> 0.8972051043024051
86---> 0.8985277299960197
.....
95---> 0.9091421162637744
96---> 0.9101542304308556
97---> 0.9111446067892256
98---> 0.9123344433204897
99---> 0.9132516893488988

```

Burada %80 varyans için elde edilen sütun sayısı 44 olarak bulunmuştur. Elde edilen grafik de şöyledir:



Gözle tespitin yapıldığı birinci yöntem bu grafikte nasıl uygulanabilir? Grafikte artık kazancın iyici düşüğü ve eğrinin eğiminin yatay eksene göre stabil hale geldiği nokta gözle tespit edilebilir. Aşağıdaki çizimde gözle tespit edilen yer ile %80'lik yer gösterilmiştir:



Aslında optimal sütun sayısının elde edilmesi için şöyle bir fonksiyon da yazabiliriz:

```
def optimal_reduction(x, ratio=0.80):
    for i in range(1, x.shape[1]):
        pca = PCA(n_components=i)
        pca.fit(x)
        if np.sum(pca.explained_variance_ratio_) >= ratio:
            return i
    return x.shape[1]

result = optimal_reduction(dataset_x)
print(result)
```

Mnist Örneğinde PCA İşleminden sonra K-Means Kümeleme Yönteminin Kullanılması

PCA bir önişlem (preprocessing) faaliyeti olarak değerlendirilebilir. Yani biz sütun indirmemesi yaptıktan sonra başka diğer işlemlere geçebiliriz. Burada Mnist verileri üzerinde PCA işleminden sonra K-Means kğmelemesi yapacağız.

```
import numpy as np
from tensorflow.keras.datasets import mnist
```

```

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) = mnist.load_data()

dataset_x = training_dataset_x.reshape(-1, 28 * 28).astype(np.float32)
dataset_x = dataset_x / 255.0

from sklearn.decomposition import PCA

pca = PCA(n_components=44)
reduced_dataset_x = pca.fit_transform(dataset_x)

from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=10)
kmeans.fit(reduced_dataset_x)
print(kmeans.labels_)

```

ANOMALİLERİN TESPİT EDİLMESİ (Anomaly Detection)

Denetimsiz makine öğrenmesinde popüler konulardan biri de anomalilerin tespit edilmesidir. Bu sayede hiçbir niteliksel bilgiye sahip olmadan hileli işlemleri ve verileri diğerlerinden belli bir güvenilirlikle ayırt edebilmektedir. Dolayısıyla anomalilerin tespit edilmesi şüpheli işlemler, bilgisayar güvenliği, bilgisayar virüsleri, kötü niyetli yazılımların belirlenmesi gibi işlemlerde kullanılabilmektedir. Anomalilerin tespit edilmesi anlamına gelen İngilizce pek çok terim uydurulmuştur. Örneğin: "Anomaly detection", "outliers", "novelties", "noise", "deviations", "exceptions".

Anomalilerin tespit edilmesi için pek çok yöntem kullanılmaktadır. Bunlardan bazıları şunlardır:

- Yoğunluk Tabanlı Yöntemler (Isolation Forest, K-Nearest Neighbor, vs.)
- Destek Vektör makineleri (support vector machines)
- Bayesian Networks
- Saklı Markov Modelleri (Hidden Markov Models)
- Kümeleme Esasına Dayanan Yöntemler (K-Means, DBSCAN vs.)
- Bulanık Mantık Kullanılan Yöntemler

DBSCAN Kümeleme Yöntemiyle Anomalilerin Tespit Edilmesi

Anımsanacağı gibi DBSCAN yönteminde yoğunluk esas alınmaktadır. Böylece bu yöntem küresel kümelerin dışında örneğin eliptik gibi diğer kümeleri de tespit edebiliyordu. DBSCAN yönteminde noktaların ilişkilendirileceği küme sayısını biz vermiyoruz. Bu küme sayısı zaten algoritmanın işleyışı sırasında otomatik olarak belirleniyordu. Yine bu yöntemde bazı noktalar hiçbir kümeye dahil edilemiyordu. Bu noktalara biz "gürültü (noise)" diyoruz. İşte DBSCAN yöntemiyle anomali tespitinde uygulamacı sanki bir kümeleme işlemi yapıyormuş gibi işlemlerini yürütür. Gürültü noktaları anomali olarak değerlendirilir.

DBSCAN yönteminin uygulamak için make_blobs fonksiyonuyla rastgele veri elde edelim. make_blobs fonksiyonu sklearn.datasets modülü içerisinde yer almaktadır. Bu fonksiyon kümeleme işlemleri için istenilen küme sayısına uygun istenilen mikarda rastgele nokta üretmektedir. Biz burada anomali tespiti için küme sayısını 1 alacağız. Örneğin:

```

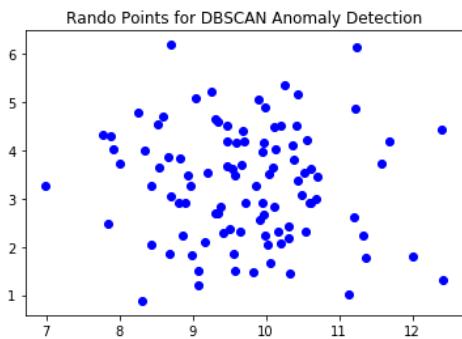
from sklearn.datasets import make_blobs

dataset_x, dataset_y = make_blobs(100, centers=1, cluster_std=1)

import matplotlib.pyplot as plt

plt.title('Rando Points for DBSCAN Anomaly Detection')
plt.scatter(dataset_x[:, 0], dataset_x[:, 1], color='blue')

```



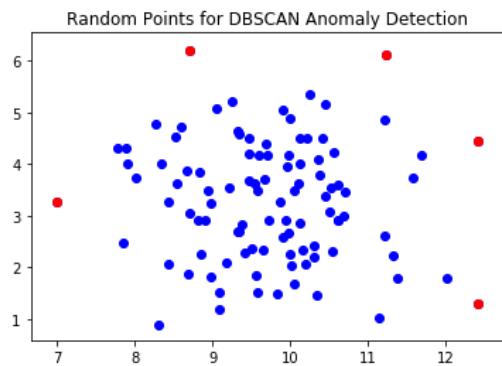
Şimdi DBSCAN algoritmasını çalıştırarak gürültü noktalarını belirleyelim:

```
from sklearn.cluster import DBSCAN

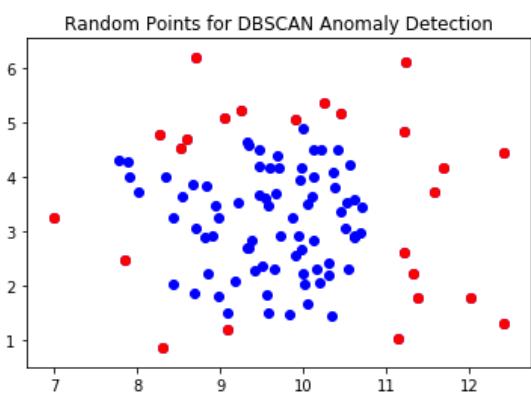
dbs = DBSCAN(eps=1)
dbs.fit(dataset_x)

plt.pause(1)

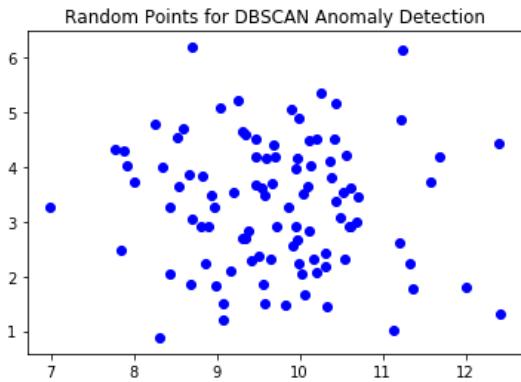
plt.title('Random Points for DBSCAN Anomaly Detection')
plt.scatter(dataset_x[:, 0], dataset_x[:, 1], color='blue')
plt.scatter(dataset_x[dbs.labels_ == -1, 0], dataset_x[dbs.labels_ == -1, 1], color='red')
```



Tabii DBSCAN yöntemini uygularken yoğun belirten epsilon değerinin iyi bir biçimde belirlenmesi gereklidir. Anımsanacağı gibi epsilon değeri küçüldükçe gürültü noktalarının sayıları artmaktadır, epsilon değeri yükseldikçe gürültü noktalarının sayıları azalmaktadır. Yukarıdaki örnekte epsilon 1 alınmıştır. Epsilon değerini 0.5'e küçültüp örneği bir daha çalıştıralım:



Şimdi de epsilon'u 1.5 alarak kodumuzu çalıştıralım:



Göründüğü gibi epsilon parametresinin uygulamacı tarafından ayarlanması gerekmektedir. Epsilon değeri küçültüldükçe şüpheli durumların sayısı artacaktır. Bu da şüpheli durumların değerlendirilmesi için daha fazla kaynağın harcanmasına yol açacaktır.

Şimdi de DBSCAN yerine K-Means algoritmasıyla anomalileri belirlemeye çalışalım. Anımsanacağı gibi K-Means yönteminde algoritmanın noktaları ayıracığı küme sayısını başlangıçta biz veriyorduk. Algoritma da tüm noktaları bir kümeye dahil ediyordu. (Yani noktalar ağırlık merkezlerine çok uzak olsalar bile K-Means algoritması yine de bu noktaları bir kümeye dahil etmektedir.) K-Means algoritmasının 1 küme için çalıştırılması anlamsız gelebilmektedir. Gerçekten de önceki konularda bizim yazdığımız K-Means kodu tek küme için tek bir iterasyonla işlemini sonlandırmaktadır. Ancak K-Means algoritmasının ince versiyonları vardır. Örneğin scikit-learn kütüphanesinin kullandığı K-Means algoritmasında küme sayısı 1 olsa bile algoritma o tek küme için rastgele aldığı ağırlık merkezini iyileştirmektedir. İşte K-Means yoluyla anomali tespitinde algoritma 1 küme için çalıştırılır. Algoritma tüm noktaları o tek kümeye dahil etmekle birlikte o kümenin ağırlık merkezini iyi bir noktaya çeker. Böylece uygulamacı tüm noktaların bu ağırlık merkezine uzaklıklarını hesaplayarak en uzak n tane noktayı anomali olarak belirleyebilir.

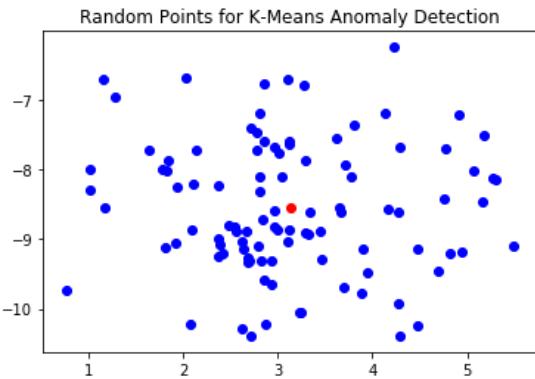
Örneğin make_blobs fonksiyonuyla bir küme için ağırlık merkezini bularak noktaların grafiğini çizdirelim:

```
from sklearn.datasets import make_blobs
dataset_x, dataset_y = make_blobs(100, centers=1, cluster_std=1)
import matplotlib.pyplot as plt
plt.title('Random Points for DBSCAN Anomaly Detection')
plt.scatter(dataset_x[:, 0], dataset_x[:, 1], color='blue')

from sklearn.cluster import KMeans
km = KMeans(n_clusters=1)
km.fit(dataset_x)

plt.pause(1)

plt.title('Random Points for k-MEANS Anomaly Detection')
plt.scatter(dataset_x[:, 0], dataset_x[:, 1], color='blue')
plt.scatter(km.cluster_centers_[0][0], km.cluster_centers_[0][1], color='red')
```



Grafkte noktalar ve ağırlık merkezi gösterilmiştir. Uygulamacının tüm noktaların ağırlık merkezine olan uzaklıklarını hesaplayıp en uzak n tane noktayı anomali olarak tespit etmesi gereklidir. Aslında KMeans sınıfının transform metodu zaten verdiğimiz noktaların ağırlık merkezlerine uzaklığını bize vermektedir. Yani biz noktaların ağırlık merkezlerine uzaklıkları için doğrudan bu metottan faydalanabiliriz:

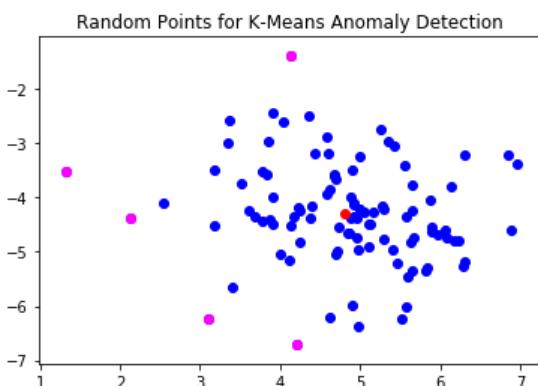
```
import numpy as np

distances = km.transform(dataset_x).ravel()
sorted_distances = np.argsort(distances)
outliers = sorted_distances[-5:]
```

ndarray sınıfının ravel metodu çok boyutlu vektörü tek boyut haline getirmektedir. np.argsort fonksiyonu ise asıl vektörü (yani örneğimizdeki distances vektörünü) değil o vektörün indislerini sıraya dizmektedir. Böylece sorted_distances vektöründeki son n tane eleman aslında ağırlık merkezlerine en uzak n noktanın indisleridir. Şimdi biz bu noktaları da grafikte belirtelim:

```
plt.pause(1)

plt.title('Random Points for K-Means Anomaly Detection')
plt.scatter(dataset_x[:, 0], dataset_x[:, 1], color='blue')
plt.scatter(km.cluster_centers_[0][0], km.cluster_centers_[0][1], color='red')
plt.scatter(dataset_x[outliers, 0], dataset_x[outliers, 1], color='magenta')
```



Şimdi de özellik indirgeme ve yükseltilmesi yöntemiyle anomali tespiti üzerinde duralım. Bu yöntemde n tane sütundan oluşan veri tabloları üzerinde özellik indirgeme yapılarak bu n tane sütun k tane sütuna indirgenir. Sonra bu k tane sütun ters dönüşümle yeniden n tane sütuna yükseltilir. Daha sonra orijinal veri tablosuyla indirgenmiş yükseltilmiş veri tablosu arasında karşılaştırma yapılır. Bu yöntemde önce indirgeme sonra yükselme yapıldığında anomali içeren satırlarda daha ciddi bir farklılaşma olmaktadır. Yöntemde indirgeme için PCA ya da diğer bazı indirgeme yöntemleri de kullanılabilir.

Pekiyi indirgenip yükseltilmiş verilerle orijinal verilerin arasındaki farka nasıl bakılmaktadır? Yaygın kullanılan bir yöntem iki vektörel nokta arasındaki farkların karelerinin toplamlarını [0, 1] arasında normalize ederek karşılaştırmaktadır. Örneğin aşağıdaki gibi bir uzaklık fonksiyonu yazılabilir:

```

def anomaly_scores(original_data, manipulated_data):
    loss = np.sum((original_data - manipulated_data) ** 2, axis=1)
    loss = (loss - np.min(loss)) / (np.max(loss) - np.min(loss))
    return loss

```

Fonksiyonun sonunda değerleri 0 ile 1 arasına getirdik. Bu işlem yapılmasa da olurdu. Buradan amaç oransal bir değerin elde edilmesidir. Bu da seçim işleminde faydalı olabilmektedir.

Şimdi özellik indirmemesi ve yükseltemesi yoluyla anomali tespit edilmesi yöntemini kredi kart işlemlerine ilişkin bir veri tablosu üzerinde uygulayalım. Veri tablosunu "Kaggle" sitesinden indirerek kullanacağız. Siz de bu tabloyu CSV dosyası olarak <https://www.kaggle.com/mlg-ulb/creditcardfraud> adresinden indirebilirsiniz. (Kursumuzda bu dosya "creditcard.csv" ismiyle indirilmiştir.) "creditcard.csv" dosyası 30 sütundan oluşmaktadır. Dosyanın başında bir başlık kısmı vardır. Eğer dosya doğrudan numpy.loadtxt fonksiyonuyla okunacaksa bu başlık kısmı geçilmelidir. Dosyanın 30'uncu sütunu iki tırmak içerisinde "0" ya da "1" olan yazılarından oluşmaktadır. Bu yazılar ilgili işlemin geçerli mi (0 değeri) yoksa geçersiz mi (1 değeri) olduğunu belirtmektedir. Dolayısıyla bu sütun diğer sütunlardan ayırtılmalıdır. Toplam 284807 tane işlem arasında sahte olanların sayısı 492 tanedir. "creditcard.csv" dosyasının yüklenip kullanımına hazır hale getirilmesi şöyle yapılabilir:

```

import numpy as np

dataset_x = np.loadtxt('creditcard.csv', dtype=np.float32, delimiter=',', skiprows=1,
usecols=range(30))
dataset_y = np.loadtxt('creditcard.csv', dtype='str', delimiter=',', skiprows=1, usecols=[30])

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
dataset_x = ss.fit_transform(dataset_x)

```

Buradan dataset_x ve dataset_y isminden iki ndarray elde etmiş olduk. Şimdi dataset_x verilerini PCA ile indirgeyip sonra yükseltilim:

```

from sklearn.decomposition import PCA
pca = PCA(n_components=19)
reduced_dataset_x = pca.fit_transform(dataset_x)
inversed_dataset_x = pca.inverse_transform(reduced_dataset_x)

```

Biz şimdi orijinal veri tablosunu (dataset_x) önce 19 sütuna indirgeyip sonra yine 30 sütuna yükselttik. Bu işlemden inversed_dataset_x tablosunu elde ettik. Artık orijinal dataset_x tablosu ile inversed_dataset_x tablosunu yukarıda yazmış olduğumuz anomaly_scores fonksiyonuna sokarak anomali değerlerini elde edebiliriz:

```
scores = anomaly_scores(dataset_x, inversed_dataset_x)
```

Şimdi bu skorlar arasında en yüksek n tanesini (örneğin 1000 tanesini) alalım:

```

sorted_scores = np.argsort(scores)
anomaly_results = dataset_x[sorted_scores[-1000:]]

anomaly = np.sum(dataset_y[sorted_scores[-fraud_count:]])
print(anomaly)

```

Burada görüldüğü gibi önce indirgeme sonra yükselme yapılmıştır orijinal verilerle satır temelinde uzaklık hesaplanmıştır. Elde edilen scores vektörü tek boyutludur ve yalnızca satırların (yani kredi kartı işlemlerinin) anomali değerini barındırır. Biz en yüksek değere sahip olan işlemlerin anomali içeriği varsayımda bulunmuştur. Bu durumda buradaki scores vektörünü sıraya dizersek en kötü n tane değeri kolayca elde edebiliriz. Ancak bu vektörün sıraya dizilmesi orijinal verilerdeki satır ilişkisini bozacaktır. Bu nedenle örneğimizde bu vektör sort etmek yerine bu vektörün indislerini sort ettik. Sonuçta n_components = 19 değeri için en kötü 1000 anomali değeri arasında 314

tane anomalili işlem, en kötü 100 arasında ise 77 tane anomalili işlem yakalanmıştır. Ancak programın her çalıştırılmasında farklı değerler bulunabilmektedir. 1000 tane değer arasında 314 tane şüpheli işlemin yakalanması aslında oldukça başarılıdır. Çünkü zaten veri kümelerindeki 284807 işlemin yalnızca 492 tanesi sahtecilik içermektedir.

Şimdi de bu kredi kart işlemlerini K-Means yöntemiyle ele alalım:

```
import numpy as np

dataset_x = np.loadtxt('creditcard.csv', dtype=np.float32, delimiter=',', skiprows=1,
usecols=range(30))
dataset_y = np.loadtxt('creditcard.csv', dtype='str', delimiter=',', skiprows=1, usecols=[30])
dataset_y = np.array([int(s[1]) for s in dataset_y], dtype=np.float32)

from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
dataset_x = ss.fit_transform(dataset_x)

from sklearn.cluster import KMeans

km = KMeans(n_clusters=1)
km.fit(dataset_x)

import numpy as np

distances = km.transform(dataset_x).ravel()
sorted_distances = np.argsort(distances)

outliers = sorted_distances[-1000:]

outliers_count = np.sum(dataset_y[outliers])
print(outliers_count)
```

K-Means yöntemini kullanarak yaptığımiz anomali tespitinde en kötü skora sahip 1000 işlem içerisinde bu yöntem 170 tane şüpheli işlemi tespit etmiştir.

İSTATİKSEL YÖNETMLERLE GERÇEKLEŞTİRİLEN REGRESYON ANALİZLERİ

Daha önce de belirttiğimiz gibi regresyon girdi ile çıktı arasında ilişki kurma sürecidir. Bağımsız değişken birden fazlaysa buna çoklu regresyon (multiple regression), bağımlı değişken birden fazlaysa buna da "çok değişkenli (multivariate) regresyon" denilmektedir. Örneğin x_1, x_2, x_3, x_4, x_5 değerlerinden hareketle bir y değerini bulmak isteyelim. Bu çoklu regresyona bir örnektir. Çünkü burada bulunmak istenen fonksiyon $y = f(x_1, x_2, x_3, x_4, x_5)$ gibi bir fonksiyondur. Ancak x_1, x_2, x_3, x_4, c_5 değerlerinden hareketle biz y_1 ve y_2 değerlerini bulmak istersek bu çok değişkenli regresyona örnektir. Çünkü burada biz $(y_1, y_2) = f(x_1, x_2, x_3, x_4, x_5)$ gibi bir fonksiyonu bulmaya çalışırız. Çok değişkenli regresyon ayrı ayrı yapılan çoklu regresyonla aynı şey değildir. Örneğin biz x_1, x_2, x_3, x_4, x_5 değerlerinden hareketle y_1 ve y_2 değerlerini bulmaya çalışalım. İlk bakışta bu y_1 ve y_2 değerlerinin iki ayrı çoklu regresyon ile mümkün gibi görülmektedir. Ancak ayrı ayrı yapılan çoklu regresyonla tek hamlede yapılan çok değişkenli regresyon aynı sonuçları vermez. Çünkü çok değişkenli regresyonda bağımlı değişkenler de birbirlerini etkileyebilmektedir. Bu durumda bağımlı değişkenler arasındaki ilişkiyi görmezden gelerek ayrı ayrı çoklu regresyon uygulamak farklı sonuçların elde edilmesine yol açmaktadır.

Regresyonlar aynı zamanda bulunacak fonksiyonun biçimine göre de "doğrusal", "polinomsal", "üstel" gibi sınıflara ayrılmaktadır. En yaygın kullanılan regresyon "çoklu doğrusal regresyon" (multiple linear regression)".

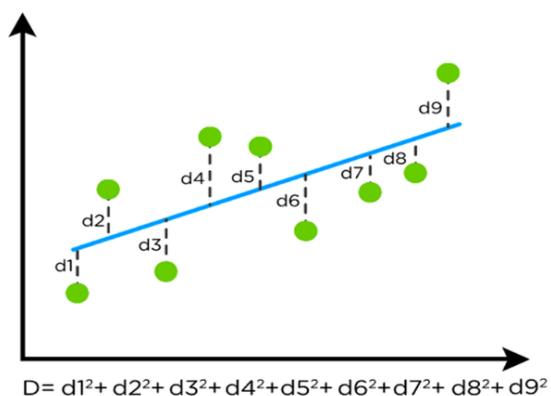
Regresyon ile korelasyon arasında önemli bir ilişki vardır. Aralarında korelasyonun sağlam olmadığı değişkenlerde regresyon analizi yapmak (özellikle doğrusal regresyon) genellikle uygun olmaz. Bu tür durumlarda regresyon sonucuna dayanılarak yapılan kestirimler başarısız olmaktadır. Bu nedenle nümerik regresyonun uygunluğunu anlamak için önce değerlerin korelasyon katsayısına bakmak gereklidir.

Daha önce biz regresyon problemlerini yapay sinir ağlarıyla çözmüştük. Korelasyonu zayıf olan olgular için yapay sinir ağları oldukça iyi bir yöntem haline gelmektedir. Maalesef karşılaşılan gerçek problemlerin büyük kısmında nümerik regresyon uygulanamayacak derecede düşük korelasyonlar bulunmaktadır. Bu durumda aralarında düşük korelasyon ilişkisi bulunan değişkenler için kestirim yöntemi olarak yapay sinir ağları tercih edilmelidir. Yani yapay sinir ağlarıyla regresyon uygulamalarında değişkenler arasında anlamlı bir korelasyonun bulunuyor olması gerekmektedir.

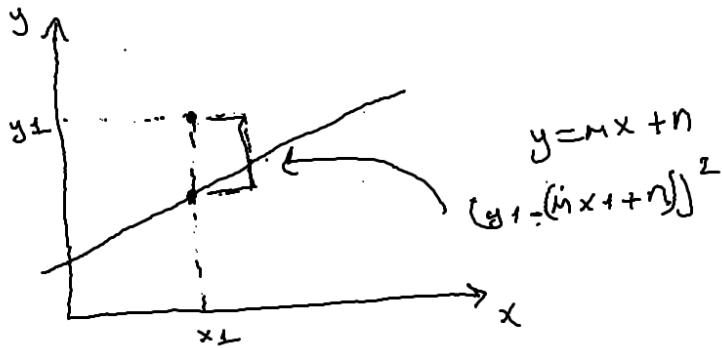
Doğrusal Regresyon

Yukarıda da belirtildiği gibi doğrusal regresyon birtakım noktaları temsil edebilecek bir doğru denklemin elde edilmesi sürecidir. Örneğin tek bağımsız değişkenli doğrusal regresyonda elde edilmek istenen doğru $y = a_0 + a_1x$ biçimindedir. Burada a_1 aynı zamanda doğrunun eğimidir., a_0 da doğrunun y eksenini kestiği noktadır. (Bu doğru denklemi genellikle $y = mx + n$ biçiminde de ifade edilmektedir.) Çoklu doğrusal regresyon ile tekli doğrusal regresyon arasında aslında hesaplamalar bakımından pek bir farklılık yoktur. Tekli doğrusal regresyon iki boyutlu düzlemden bir doğru denkleminin elde edilmesi süreci çoklu doğrusal regresyon n boyutlu uzayda bir doğru denkleminin elde edilmesi sürecidir. Çok boyutlu uzaydaki bu genel doğru denklemi de $y = a_0 + a_1x_1 + a_2x_2 + a_3x_3 + \dots + a_nx_n$ biçimindedir. Bu durumda n bağımsız değişken için regresyon analizi sonucunda bizim $n+1$ 'tanın değer elde etmemiz gerekmektedir. Mademki tek bağımsız değişkenli doğrusal regresyon hesabı ile çoklu doğrusal regresyon hesabı arasında bir farklılık yoktur, o halde biz doğrusal regresyonun genel ifadesini elde etmek için tek değişkenli modelden faydalananabiliriz.

Tek bağımsız değişkenli doğrusal regresyondaki katsayı değerlerinin elde edilmesi için kullanılan formülüne "en küçük kareler (least squares)" formülü denilmektedir. $y = mx + n$ doğrusundaki m ve n değerlerini veren en küçük karaler formülü aslında söz konusu noktaları en iyi biçimde ortalayan doğrunun m ve n değerlerini bulmaktadır. Burada noktaları en iyi biçimde ortalayan doğru demekle noktaların doğruya uzunlukları toplamını en küçükleşen doğru kastedilmektedir. Bir noktanın koordinatları (x_1, y_1) ise bu noktadan doğruya uzaklık $y_1 - mx_1 + n$ 'dır. $((x_1, y_1)$ noktasının doğru üzerindeki izdüşümündeki x değerinin de x_1 olduğuna dikkat ediniz. Yani ilgili noktadan doğruya dikme indirilmemektedir.)



Burada en küçüklemeye çalışılan şey noktaların doğruya uzaklıklarının toplamıdır. Ancak bu uzaklıklar negatif olabileceğinden dolayı negatiflikten onları kurtarmak için kare alma işlemi uygulanır. Belli bir noktanın buradaki doğruya uzunluğu basit biçimde şöyle hesaplanabilmektedir:



Amaç tüm noktaların doğruya toplam uzaklıklarının en küçüklenmesi olduğuna göre X ve Y noktaları belirten vektörler olmak üzere en küçüklenecek ifade şöyle oluşturulabilir:

$$\sum (Y - (mX + n))^2$$

Bizim buradaki asıl amacımız m ve n değerlerini bulmaktır. Bu işlem matematiksel olarak m 'ye ve n 'ye göre birinci kismi türevlerinin sıfırlanması ile gerçekleştirilir. O halde biz de yukarıdaki ifadenin m 'ye ve n 'ye göre kismi türevlerini alıp sıfırlayalım. Buradan da m 'yi ve n 'yi çekelim:

$$\frac{\partial f}{\partial m} = -2 \sum (Y - n - mX)X = 0$$

$$\frac{\partial f}{\partial n} = -2 \sum (Y - n - mX) = 0$$

$$m = \frac{\sum (X - \bar{X})(Y - \bar{Y})}{\sum (X - \bar{X})^2}$$

$$n = \frac{\sum Y - m \sum X}{\sum (X - \bar{X})}$$

Aynı eşitlikler şöyle de ifade edilebilirdi:

$$m = \frac{N \sum(xy) - \sum x \sum y}{N \sum(x^2) - (\sum x)^2}$$

$$b = \frac{\sum y - m \sum x}{N}$$

Tabii aslında bu tek bağımsız değişken için elde edilen formül çok bağımsız değişken için de benzer biçimde uygulanmaktadır. Örneğin yukarıda bizim çıkarttığımız formülde x aslında tek bir değeri değil bir vektörü temsil ediyor olabilir. Bu durumda x ortalama (x üstü çizgi) da aslında her değişkene ilişkin sütunun kendi ortalamasıdır. O halde çok değişken için aslında m değeri de tek bir değer değil bir vektör olarak elde edilecektir. Bu vektördeki değerler $a_0 + a_1x_1 + a_2x_2 + a_3x_3 + \dots + a_nx_n$ doğru denklemindeki $a_1, a_2, a_3, \dots, a_n$ değerleri olacaktır. Başka bir deyişle

Çoklu regresyon durumunda yukarıda çıkarttığımız formüldeki m eşitliğinde bulunan x ve x ortalama (\bar{x} üzeri çizgi) aslında birer vektördür.

$$m = \sum (x - \bar{x})(y - \bar{y})$$

$$\sum (x - \bar{x})$$

$$[a_1, a_2, a_3]$$

$$[x_1, x_2, x_3]$$

$$[\bar{x}_1, \bar{x}_2, \bar{x}_3]$$

Şimdi doğrusal regresyon için m ve n değerlerini bulan bir fonksiyon yazalım:

```
import numpy as np

def linear_regression(x, y):
    a = np.sum((x - np.mean(x)) * (y - np.mean(y)))
    b = np.sum((x - np.mean(x)) ** 2)
    m = a / b
    n = (np.sum(y) - m * np.sum(x)) / len(x)
    return m, n
```

Şimdi de "test.csv" içersindeki noktalar doğrusal regresyon grafiğini çizelim. "test.csv" dosyasının içeriği şöyledir:

```
2,4
3,5
5,7
7,10
7,8
8,12
9.5,10.5
9,15
10,17
13,18
```

Program şöyle oluşturulabilir:

```
import numpy as np

def linear_regression(x, y):
    m = np.sum((x - np.mean(x)) * (y - np.mean(y))) / np.sum((x - np.mean(x)) ** 2)
    n = np.sum((np.sum(y) - m * np.sum(x))) / len(x)

    return m, n

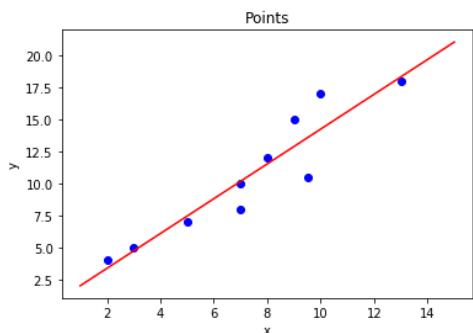
dataset = np.loadtxt('test.csv', dtype=np.float32, delimiter=',')
m, n = linear_regression(dataset[:, 0], dataset[:, 1])

x = np.linspace(1, 15, 100)
y = m * x + n

import matplotlib.pyplot as plt

plt.title('Points')
plt.xlabel('x')
plt.ylabel('y')
```

```
plt.scatter(dataset[:, 0], dataset[:, 1], color='blue')
plt.plot(x, y, color='red')
```



Şimdi de interpolasyonla kestirimde bulmaya çalışalım. Örneğin $x = 11$ için y 'nin değeri nedir?

```
xval = 11
yval = m * xval + n
print('{} --> {}'.format(xval, yval))

11 --> 15.611846947669983
```

Doğrusal regresyon scikit-learn'de `linear_model` isimli modülde `LinearRegression` isimli sınıfı gerçekleştirilmiştir. Bu sınıf hem tekli, hem çoklu hem de çok değişkenli regresyon için kullanılabilir. Sınıfın `__init__` metodu şöyledir:

```
class sklearn.linear_model.LinearRegression(fit_intercept=True, normalize=False, copy_X=True, n_jobs=None)
```

`LinearRegression` nesnesi yaratıldıkten sonra sınıfın `fit` metodu ile regresyon uygulanır. Sınıfın `predict` metodu ise değerleri doğru denkleminde yerine koymak bize sonucu vermektedir. Doğru denklemindeki katsayıları sınıfın `coef_` isimli özniteligiden, y ekseni kesim noktasını da `intercept_` isimli özniteligiden elde edebiliriz. Anımsanacağı gibi çoklu regresyonlarda doğru katsayıları bir tane değil n tanedir. Örneğin tekli regresyon için doğru denklemi şöyledir:

$$y = a_0 + a_1 x$$

Burada a_1 doğrunun eğimi (yani m), a_0 ise ekseni kesim noktasıdır. Dolayısıyla biz tekli regresyonda sınıfın `coef_` özniteligiden bir tane katsayı elde ederiz. Bu da a_1 katsayıdır. Sınıfın `coef_` özniteliği ise bize a_0 değerini vermektedir. Çoklu regresyonda ise doğru denklemi şu biçimde olacaktır:

$$y = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + \dots + a_n x_n$$

Burada ise sınıfın `coef_` özniteliği bize n elemanlı bir katsayı vektörü verecektir. Bu n elemanlı katsayı vektörü sırasıyla $a_1, a_2, a_3, \dots, a_n$ değerlerine karşılık gelmektedir. Sınıfın `intercept_` özniteliği ise yine bize 1 tane değer verecektir. Bu da genel gösterimdeki a_0 değeridir.

Şimdi yukarıda manuel olarak yaptığımız doğrusal regresyon örneğini `LinearRegression` sınıfı ile yapalım:

Örneğin:

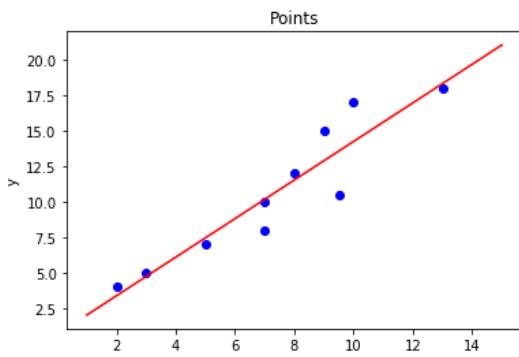
```
import numpy as np
dataset = np.loadtxt('test.csv', dtype=np.float32, delimiter=',')
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(dataset[:, 0].reshape(-1, 1), dataset[:, 1])
x = np.linspace(1, 15, 100)
y = lr.coef_[0] * x + lr.intercept_
```

```

import matplotlib.pyplot as plt

plt.title('Points')
plt.xlabel('x')
plt.ylabel('y')
plt.scatter(dataset[:, 0], dataset[:, 1], color='blue')
plt.plot(x, y, color='red')

```



Burada fit metodunun iki boyutlu bir matris istediğine dikkat ediniz. Çünkü LinearRegression sınıfı çoklu regresyonlarda da kullanılabilmektedir. Yukarıdaki örnekte reshape fonksiyonu bu amaçla kullanılmıştır. Yukarıdaki örneğimizde x değerleri reshape yapıldıktan sonra aşağıdaki duruma getirilmiştir:

```

array([[ 2. ],
       [ 3. ],
       [ 5. ],
       [ 7. ],
       [ 7. ],
       [ 8. ],
       [ 9.5],
       [ 9. ],
       [10. ],
       [13. ]], dtype=float32)

```

Doğrusal regresyon işleminde yapılan regresyonun noktaları ne kadar iyi temsil edebildiğine yönlek R^2 denilen bir ölçüt kullanılmaktadır. R^2 değeri ne kadar yüksekse noktaların doğruya uzunlukları o kadar kısadır. Yani regresyon doğrusu noktalara o kadar yakındır. R^2 değeri düştükçe regresyon doğrusunun noktaları temsil etme yeteneği azalmaktadır. Yani yüksek bir R^2 değeri söz konusu olduğunda bizim yapacağımız kestirim de o kadar iyi olacaktır. R^2 değeri noktaların doğruya olan uzaklıklarının karelerinin toplamının, noktaların kendi ortalamalarına uzaklıklarının karelerinin toplamına oranıdır. Biz burada R^2 değerinin nasıl hesaplanacağı üzerinde durmayacağız. LinearRegression sınıfının score isimli metodu bize R^2 değerini vermektedir. Örneğin yukarıdaki "test.csv" dosyasındaki noktalar için R^2 değeri şöyle elde edilebilir:

```

rsquare = lr.score(dataset[:, 0].reshape(-1, 1), dataset[:, 1])
print(rsquare)

```

0.871811303294605

Şimdi de LinearRegression sınıfını kullanarak çoklu regresyon örneği verelim. Biz daha önce Boston'daki ev fiyatlarını tahmin eden örneği yapay sinir ağlarıyla gerçekleştirmiştik. Şimdi Boston örneğini nümerik olarak çoklu regresyon ile gerçekleştirelim. Boston verileri keras kütüphanesinin yanı sıra scikit-learn içerisinde de datasets modülünde bulunmaktadır. Ancak bu Boston örneğindeki sütunların hepsini değil yalnızca 5'inci ve 12'inci sütunları biz regresyonda kullanacağız. Çünkü tüm sütunlar işleme sokulduğunda düşük korelasyon oluşmaktadır. (Animsanacağı gibi bu tür düşük korelasyonlu durumlarda nümerik regresyon değil yapay sinir ağları ile regresyon çok daha uygun olmaktadır.)

```
from sklearn.datasets import load_boston
```

```

boston_dataset = load_boston()

dataset_x = boston_dataset.data[:, (5, 12)]
dataset_y = boston_dataset.target

from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(dataset_x, dataset_y)

print('Coefficients: {}'.format(lr.coef_))
print('Intercept: {}'.format(lr.intercept_))

```

Katsayılar ve eksen kesim noktası şöyle bulunmuştur:

```

Coefficients: [ 5.09478798 -0.64235833]
Intercept: -1.358272811874489

```

Şimdi x değerleri için bir y değerini bulalım. Bunu iki biçimde yapabiliriz. Birincisi doğru denkleminde yerine koyarak. İkincisi doğrudan predict metodunu kullanarak:

```

import numpy as np

xvals = np.array([10, 12], dtype=np.float32)
yval = np.dot(lr.coef_, xvals) + lr.intercept_
print('{} ---> {}'.format(xvals, yval))

```

Elde edilen değerler şöyledir:

```
[10. 12.] ---> 41.88130702056141
```

Şimdi de aynı işlemi predict metoduyla yapalım:

```

yval = lr.predict(xvals.reshape(1, -1))
print('{} ---> {}'.format(xvals, yval))

```

Elde edilen değerler şöyledir:

```
[10. 12.] ---> [41.88130702]
```

Biz bu örnekte iki değişkenli çoklu regresyon uyguladık. Regresyon sonucunda bulduğumuz doğru denkleminin genel biçimini şöyledir:

$$y = a_0 + a_1x_1 + a_2x_2$$

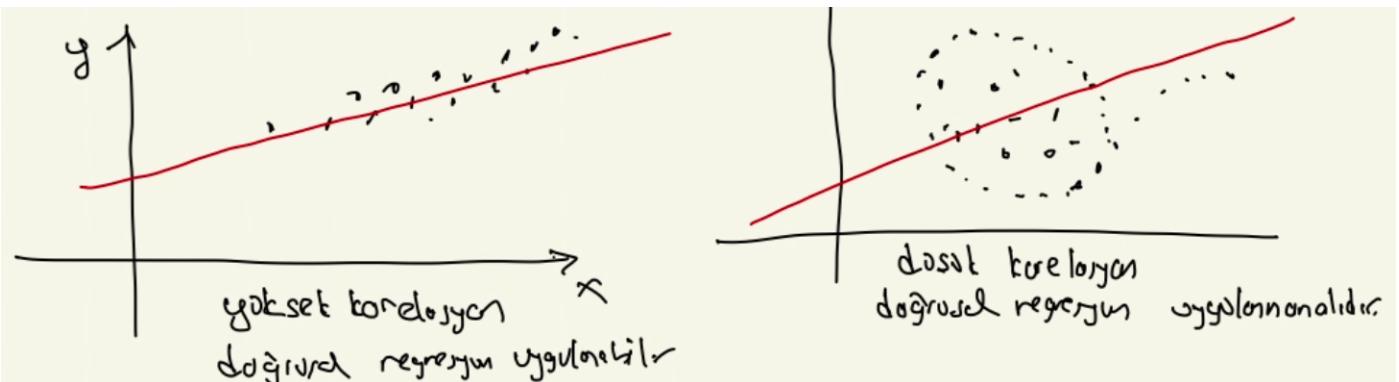
Burada a_0 değeri sınıfın intercept_ özniteliğinden elde edilmiştir. a_1 ve a_2 değerleri ise sınıfın coef_ özniteliğinden elde edilmiştir. Yani bizim bulduğumuz doğru denklemi şöyledir:

$$y = -1.358272811874489 + 5.09478798 \times x_1 - 0.64235833 \times x_2$$

Çok değişkenli doğrusal regresyonlarda da yine aynı LinearRegression sınıfı kullanılmaktadır. Burada fit işlemi ugulanırken y verilerinin bir matris biçiminde olması gereklidir.

İstatistiksel Doğrusal Regresyon Yönetmiyle Yapay Sinir Ağlarıyla Regresyon Yönteminin Karşılaştırılması

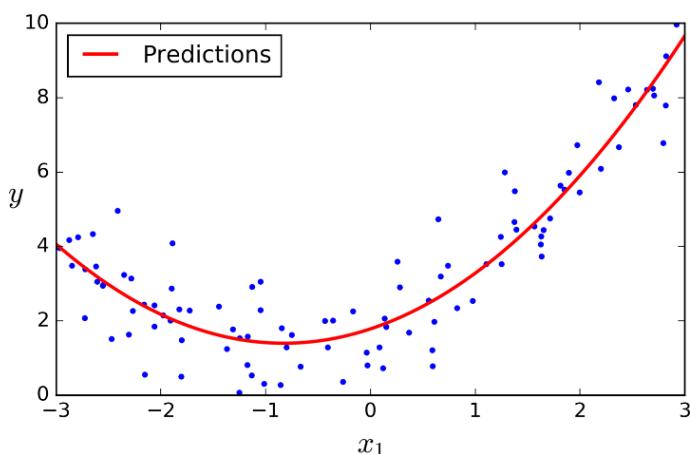
İstatistiksel doğrusal regresyon yukarıda da belirtildiği gibi korelasyonun yüksek olduğu verilere uygulanabilmektedir. x verileriyle y arasındaki korelasyon düşükse bu verileri bir doğru ile temsil etmek iyi bir fikir değildir. Örneğin:



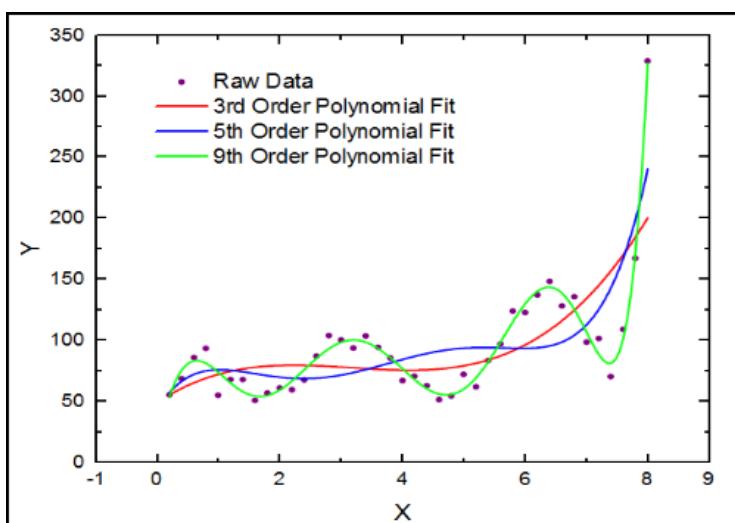
Yapay sinir ağları için böyle bir korelasyon koşulu yoktur. Yapay sinir ağları korelasyonu düşük verilerde de uygun kalıpları bulup bir tahminde bulunmamıza olanak sağlamaktadır. Öte yandan yapay sinir ağlarının eğitilmesi için nispeten yüksek sayıda veriye gereksinim vardır. Eğer eğitimde kullanılacak veri sayısı düşükse yapay sinir ağları çok zayıf bir performans göstermektedir. Halbuki doğrusal regresyon az sayıda verinin korele bir biçimde bulunduğu durumlarda çok daha pratik ve uygundur.

Polinomsal Regresyon

Polinomsal regresyon noktaları temsil edebilecek bir polinomsal eğrinin elde edilmesi sürecidir. Doğrusal olmayan bir regresyon çeşididir. Bazı durumlarda tercih edilebilmektedir. Örneğin:



Burada doğrusal bir regresyonun uygulanmasının uygun olmadığı grafikten görülmektedir. Pekiyi uygulanacak polinom kaçinci dereceden olmalıdır? Örneğin:



Polinomsal regresyon scikit-learn kütüphanesinde preprocessing modülündeki `PolynomialFeatures` sınıfının yardımıyla yapılmaktadır. Sınıfın `__init__` metodunun parametrik yapısı şöyledir:

```
sklearn.preprocessing.PolynomialFeatures(degree=2, interaction_only=False, include_bias=True, order='C')
```

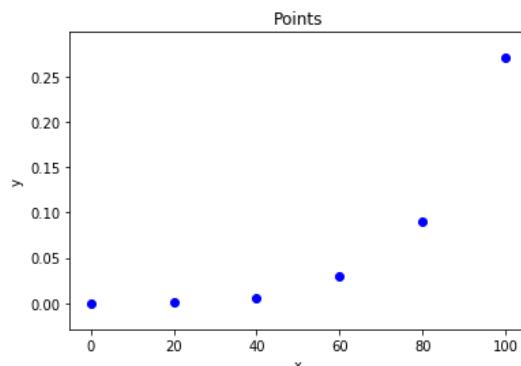
Fonksiyonun degree parametresi kaçinci derece bir polinomla ilgilenildiğini belirtir. Nesne yaratıldıktan sonra önce fit sonra da transform işlemlerini yapmak gereklidir. İki işlem bir arada fit_transform fonksiyonuyla da yapılabilmektedir. transform ya da fit_transform bize dönüştürülmüş bir matris verir. Burada biz öncelikle bize verilen bu dönüştürülmüş matrisin anlamı üzerinde duracağız. Örneğin aşağıdaki gibi tek bağımsız değişken ve tek bağımlı değişkenden oluşan bir verimiz olsun:

```
test.csv
```

```
0,0.0002  
20,0.0012  
40,0.0060  
60,0.0300  
80,0.0900  
100,0.27
```

Bu noktaların serpilme grafiğini çizelim:

```
import numpy as np  
  
dataset = np.loadtxt('test.csv', dtype=np.float32, delimiter=',')  
  
import matplotlib.pyplot as plt  
plt.title('Points')  
plt.xlabel('x')  
plt.ylabel('y')  
plt.scatter(dataset[:, 0], dataset[:, 1], color='blue')
```



Grafikten de görüldüğü gibi x ve y arasında ikinci derece polinomsal bir ilişkinin varlığı anlaşılmaktadır. O halde biz tahminleme yapmak için ikinci derece polinomsal bir regresyon analizi uygulayabiliriz. Pekiyi biz noktalara doğrusal regresyon uygulamak istesek elde edeceğimiz doğru ne kadar uygun olabilir? Deneyelim:

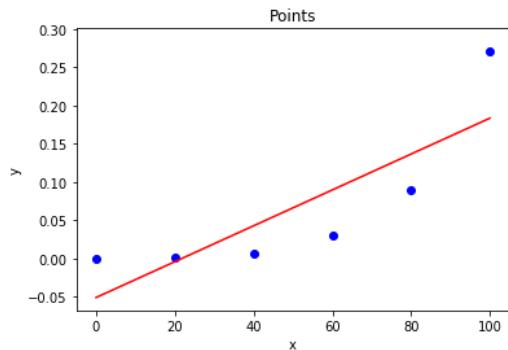
```
import numpy as np  
  
dataset = np.loadtxt('test.csv', dtype=np.float32, delimiter=',')  
  
from sklearn.linear_model import LinearRegression  
  
lr = LinearRegression()  
lr.fit(dataset[:, 0].reshape(-1, 1), dataset[:, 1])  
  
x = np.linspace(0, 100, 100)  
y = lr.coef_[0] * x + lr.intercept_  
  
import matplotlib.pyplot as plt  
plt.title('Points')
```

```

plt.xlabel('x')
plt.ylabel('y')
plt.scatter(dataset[:, 0], dataset[:, 1], color='blue')
plt.plot(x, y, color='red')

```

Elde edilen grafik şöyledir:



Burada elde edilen doğrunun noktaları iyi temsil etmediği gözle görülmektedir. Şimdi de R^2 değerine bakalım:

```

rsquared = lr.score(dataset[:, 0].reshape(-1, 1), dataset[:, 1])
print(rsquared)

```

0.6903499390522619

R^2 değerinin düşük olduğunu görüyorsunuz. Yukarıda da belirtildiği gibi bu noktalar için bir doğru geçirmeye çalışmak yerine ikinci dereceden bir polinom eğrisi geçirmeye çalışmak daha uygun olacaktır.

Scikit-learn kütüphanesinde polinomsal regresyon iki aşamada uygulanmaktadır. Birinci aşamada `sklearn.preprocessing` modülündeki `PolynomialFeatures` sınıfı kullanılarak noktalar transpoze edilir. İkinci aşamada transpoze edilmiş noktalar yoluyla doğrusal regresyon uygulanır. İzleyen paragraflarda da açıklandığı gibi polinomsal regresyon aslında çoklu doğrusal regresyonla aynı anlamdadır. Şimdi biz `PolynomialFeatures` sınıfı ile noktalarımızı transpoze edelim:

```

import numpy as np

dataset = np.loadtxt('test.csv', dtype=np.float32, delimiter=',')
dataset_x = dataset[:, 0].reshape(-1, 1)
dataset_y = dataset[:, 1]

from sklearn.preprocessing import PolynomialFeatures

pf = PolynomialFeatures(degree=2)
transformed_dataset_x = pf.fit_transform(dataset_x)
print(transformed_dataset_x)

```

Burada `fit_transform` işleminden elde edilen matrisin şekline bakınız:

```

[[1.0e+00  0.0e+00  0.0e+00]
 [1.0e+00  2.0e+01  4.0e+02]
 [1.0e+00  4.0e+01  1.6e+03]
 [1.0e+00  6.0e+01  3.6e+03]
 [1.0e+00  8.0e+01  6.4e+03]
 [1.0e+00  1.0e+02  1.0e+04]]

```

Buradaki matrisin 6×3 'luk olduğuna dikkat ediniz. Buradaki 6 değer, 6 tane noktanın olmasından gelmektedir. Pekiyi 3 değeri nereden gelmektedir? Buradaki 3 sutun tek değişkenli (örneğimizde x) ikinci derece polinomun tüm terimlerinin sayısı ilgilidir. Tek değişkenli ikinci derece polinomun genel ifadesi şöyledir:

$$y = a_0 + a_1x + a_2x^2$$

Göründüğü gibi burada toplam 3 tane tahmin edilmesi istenilen a_i katsayıları vardır. Transform edilen matrisin birinci sütunu x^0 değerini, ikinci sütunu x^1 değerini, üçüncü sütunu ise x^2 değerini belirtmektedir.

Aslında bizim tahmin etmeye çalıştığımız değerler bu a_i değerleridir. Pekiyi bağımsız değişken sayısı 2 tane olsaydı ikinci derece polinomun genel biçimini nasıl olurdu?

$$y = a_0 + a_1x_1 + a_2x_2 + a_3x_1x_2 + a_4x_1^2 + a_5x_2^2$$

Göründüğü gibi burada tahmin edilmeye çalışılacak 6 tane a_i değeri bulunmaktadır. Buradaki sütunlar da sırasıyla $x_1^0x_2^0$, x_1 , x_2 , x_1x_2 , x_1^2 , x_2^2 biçimindedir. Yani bizim "test.csv" dosyasımızdaki verilerde iki tane x değeri olsaydı bizim transform ettiğimiz matrisin 6 tane sütunu olacaktı:

$$\left[\begin{array}{cccccc} x_1^0 & x_1 & x_2 & x_1x_2 & x_1^2 & x_2^2 \\ - & - & - & - & - & - \\ - & - & - & - & - & - \\ \dots & \dots & \dots & \dots & \dots & \dots \end{array} \right]$$

`PolynomialFeatures` sınıfının `fit_transform` metodundan elde ettiğimiz matris aslında yukarıda açıklanan sütun değerlerine ilişkindir. Pekiyi bu durumun polinomsal regresyonla ne ilgisi vardır?

İşte aslında polinomsal regresyon çoklu (multiple) doğrusal regresyon aynı anlamdadır. Örneğin aslında tek değişkenli ikinci derece polinomsal regresyon üç değişkenli çoklu doğrusal regresyonla aynı anlama gelmektedir. İki değişkenli ikinci derece polinomsal regresyon da 6 değişkenli çoklu doğrusal regresyonla aynı anlamdadır. Örneğin tek değişkenli ikinci derece bir polinom söz konusu olsun. Bu poliomun genel biçimini şöyledir:

$$y = a_0 + a_1x + a_2x^2$$

Burada aslında doğrusal regresyon bağlamında x ve x^2 'ler değişken değil katsayı olarak ele alınabilir. Yani biz polinomu şöyle doğrusal bir biçimde dönüştürebiliriz:

$$y = a_0 + xa_1 + x_2a^2$$

Burada aslında x değerleri bizim gözlediğimiz değerlerdir. Biz burada a_i değerlerini bulmaya çalışmaktaiz. Şimdi değişken sayısı iki olduğunda da benzer biçimde bu da 6 terimli bir çoklu doğrusal regresyon modeline dönüştürülebilir.

$$y = a_0 + x_1a_1 + x_2a_2 + x_1x_2a_2 + x_1^2a_4 + x_2^2a_5$$

Başka bir deyişle aslında:

$$y = AX$$

birimde matisel formda gösterilmiş bir polinomsal deneklem,

$$y = XA$$

biçiminde de ifade edilebilir. Buradan hareketle bizim yapacağımız şey aslında transform edilmiş çok sütunlu değerleri doğrusal regresyona sokmak olacaktır. Örneğin:

```
import numpy as np

dataset = np.loadtxt('test.csv', dtype=np.float32, delimiter=',')
dataset_x = dataset[:, 0].reshape(-1, 1)
dataset_y = dataset[:, 1]

from sklearn.preprocessing import PolynomialFeatures

pf = PolynomialFeatures(degree=2)
transformed_dataset_x = pf.fit_transform(dataset_x)
print(transformed_dataset_x)

from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(transformed_dataset_x, dataset_y)

a0 = lr.coef_[0]
a1 = lr.coef_[1]
a2 = lr.coef_[2]

print('a0 = {}, a1 = {}, a2 = {}'.format(a0, a1, a2))

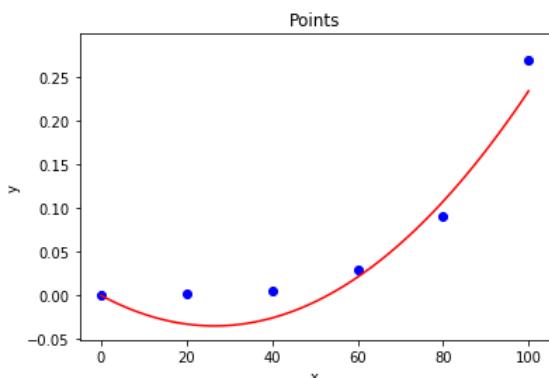
x = np.linspace(0, 100, 100)
y = a0 + a1 * x + a2 * x ** 2

import matplotlib.pyplot as plt
plt.title('Points')
plt.xlabel('x')
plt.ylabel('y')
plt.scatter(dataset[:, 0], dataset[:, 1], color='blue')
plt.plot(x, y, color='red')
```

Buradan elde ettiğimiz a0, a1, ve a2 katsayıları şöyledir:

a0 = 0.0, a1 = -0.0026392594445496798, a2 = 4.9812791985459626e-05

Elde ettiğimiz polinomun noktalarla birlikte grafiği de şöyledir:



Şimdi burada yapılanların anlamını bir daha değerlendirelim. Başlangıçta bizim elimizde tek değişkenli ikinci derece bir polinomun tahmin edilmesi problemi vardı:

$$y = a_0 + a_1 x + a_2 x^2$$

Burada bizim tahmin etmeye çalıştığımız değerler a_0 , a_1 ve a_2 değerleridir. Biz bu polinomu aşağıdaki gibi doğrusal hale getirdik:

$$y = x_0 + x_1 a_0 + x_2 a_1 + x_3 a_2$$

Aslında biz LinearRegression sınıfının predict metodu ile de elde etmeye çalıştığımız polinoma ilişkin y değerlerini bulabiliyoruz. Yani grafiğimizi şöyle de çizebilirdik:

```
x = np.linspace(0, 100, 100).reshape(-1, 1)
transformed_x = pf.fit_transform(x)
y = lr.predict(transformed_x)

import matplotlib.pyplot as plt
plt.title('Points')
plt.xlabel('x')
plt.ylabel('y')
plt.scatter(dataset[:, 0], dataset[:, 1], color='blue')
plt.plot(x, y, color='red')
```

Doğrusal regresyondan elde ettiğimiz katsayıları inceleyiniz:

```
a0 = 0.0, a1 = -0.0026392594445496798, a2 = 4.9812791985459626e-05
```

Yani aslında bizim elde ettiğimiz polinom şöyledir:

$$y = -0.0026 x + 4.98 x^2$$

Şimdi de polinomsal regresyonu doğrusal regresyona dönüştürdükten sonra elde ettiğimiz R^2 değerine bakalım:

```
rsquared = lr.score(transformed_dataset_x, dataset_y)
print(rsquared)
```

Elde edilen sonuç şöyledir:

```
0.9568460873349596
```

Anımsanacağı gibi biz daha önce bu noktalardan doğru geçirdiğimizde 0.6903499390522619 değeri elde etmiştık. Göründüğü gibi bu noktalardan ikinci bir derece bir polinomun geçirilmesi modeli çok daha iyileştirmiştir.

Şimdi de n sütunlu bir veri kümelerinin k'inci dereceden bir polioma nasıl uydurulacağını özet olarak verelim:

1) Önce PolynomialFeatures sınıfı türünden bir nesne yaratılır. Sonra sınıfın fit_transform metodu çağrılarak transform edilmiş katsayı matrisi elde edilir:

```
pf = PolynomialFeatures(degree=k)
transformed_dataset_x = pf.fit_transform(dataset_x)
```

2) Sonra transform edilmiş değerleri LinearRegression sınıfında kullanıp fit işlemi yapmalıyız.

```
lr = LinearRegression()
lr.fit(transformed_dataset_x, dataset_y)
```

3) Artık her şey hazırır. LinearRegression sınıfının predict metoduna x değerleri verilirse y değeri elde edilir. Örneğin:

```
x_transformed = pf.fit_transform(x)
y = lr.predict(x_transformed)
```

4) k'inci derece polinomun katsayıları da istenirse lr.coef_ özniteliğinden alınır. lr.intercept özniteliğinin orijinal polinomun katsayılarıyla bir ilgisi yoktur. Yani bir deyişle aslında elde edilen polinomun katsayılı şöyledir:

```
y = lr.coef_[0] + lr.coef_[1] * x + lr.coef_[2] * x ** 2 + .... + lr.coef_[k] * x ** k
```

Pekiyi biz elimizdeki noktalardan bir doğru mu, yoksa bir polinom mu geçireceğimizi nasıl belirleyeceğiz? Ya da eğer polinom geçireceksek bunun kaçinci dereceden polinom olacağını nasıl belirleyeceğiz? Aslında bu kararı otomatik bir biçimde veren scikit-learn içerisinde bir fonksiyon bulunmamaktadır. Uygulamacı noktaların grafiine bakarak sezgisel biçimde bunu belirleyebilir. Gerçekten de biz yukarıdaki örnekte noktalardan ikinci derece bir polinomun geçirilebileceğini grafiğe bakarak sezgisel biçimde belirledik. Tabii noktaların grafiğine gözle bakabilmek için değişken sayısının 1 tane olması gereklidir. Halbuki gerçek uygulamalarda değişken sayısı 1'den fazla olmaktadır. Kaçinci dereceden bir polinomun uygun olacağını belirlemek aslında deneme yanılma yoluyla yapılabilir. Yani uygulamacı ikinci derece, üçüncü derece, dördüncü derece gibi farklı derecelerde polinom geçirerek R^2 değerine bakabilir. Bu R^2 değerinin en iyi olduğu dereceyi belirleyebilir. Örneğin yukarıda üzerinde alışığımız noktalardan üçüncü derece bir polinom geçirmeye çalışalım:

```
import numpy as np

dataset = np.loadtxt('test.csv', dtype=np.float32, delimiter=',')
dataset_x = dataset[:, 0].reshape(-1, 1)
dataset_y = dataset[:, 1]

from sklearn.preprocessing import PolynomialFeatures

pf = PolynomialFeatures(degree=3)
transformed_dataset_x = pf.fit_transform(dataset_x)
print(transformed_dataset_x)

from sklearn.linear_model import LinearRegression

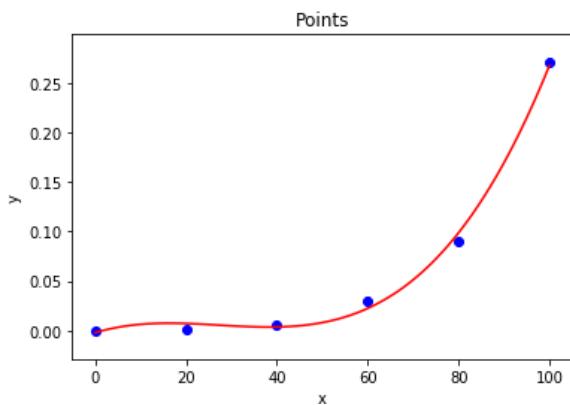
lr = LinearRegression()
lr.fit(transformed_dataset_x, dataset_y)

x = np.linspace(0, 100, 100).reshape(-1, 1)
transformed_x = pf.fit_transform(x)
y = lr.predict(transformed_x)

import matplotlib.pyplot as plt
plt.title('Points')
plt.xlabel('x')
plt.ylabel('y')
plt.scatter(dataset[:, 0], dataset[:, 1], color='blue')
plt.plot(x, y, color='red')

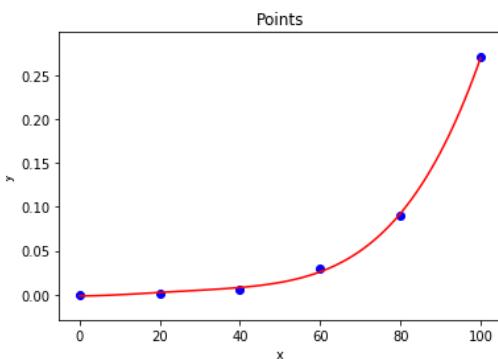
rsquared = lr.score(transformed_dataset_x, dataset_y)
print(rsquared)
```

Buradan elde ettiğimiz grafik ve R^2 değeri şöyledir:



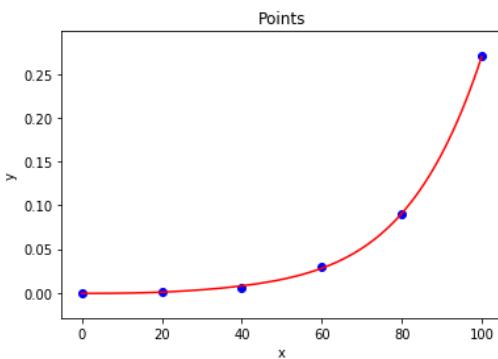
0.9966690525829125

O halde bu noktalar için üçüncü derece bir polinom aslında daha uygundur. Dördüncü derece polinom için elde edilen grafik ve R^2 değerleri de şöyledir:



0.9994477004504332

Göründüğü gibi R^2 değeri çok az daha iyileşmiştir. Ancak yüksek dereceli polinomlardan genel olarak kaçınmak gereklidir. Şimdi 5'inci derece polinom için grafik ve R^2 değerine bakalım:



0.9998423761893278

Yukarıda da belirttiğimiz gibi az bir iyileşme için derece yükselmesi iyi bir teknik değildir. Burada üçüncü derece bir polinom uygun olabilir.

GRADIENT ASCENT ve GRADIENT DESCENT ALGORİTMALARININ ANLAMI

Gradient ascent ve gradient descent algoritmaları makine öğrenmesinde çok sık kullanılmaktadır. Anımsanacağı gibi biz de yapay sinir ağlarında optimizasyon algoritması olarak "stochastic gradient descent (sgd)" algoritmasını kullanmıştık. Stochastic gradient descent algoritması gradient descent algoritmasının bir türüdür. Pekiyi bu algoritmaların çalışma biçimi nasıldır? Aslında bu algoritmalar bir noktadan başlayarak uygun bir doğrultuda yavaş yavaş ilerleme ile karakterize olmaktadır. Yani gradient ascent ve gradient descent algoritmaları iteratifdir, en yüksek

ya da en düşük optimal noktaya yavaş yavaş erişmeyi hedeflemektedir. Algoritmada doğrultu hedefe varmayı sağlayacak biçimde hesaplanmaktadır. Matematiksel anlamda eğer bir maksimizasyon problemi söz konusu ise "gradient ascent", bir minimizasyon söz konusu ise "gradient descent" yöntemi kullanılır.

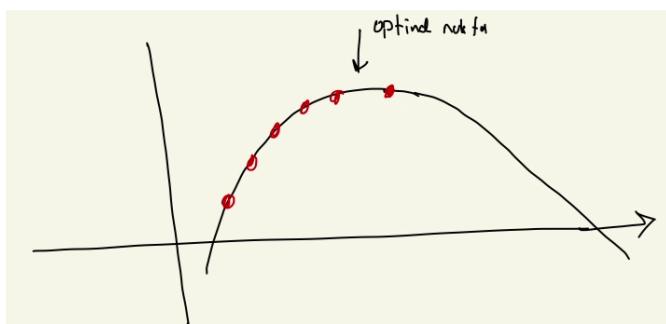
Bu yöntemlerde önce bir amaç fonksiyonu tespit edilmelidir. Yani ilerlendikçe ne hedeflenmektedir? Neyin maksimize ya da minimize edilmesi istenmektedir? Bunun bir biçimde matematiksel ifadesi ortaya konmalıdır. İkinci olarak hedefe varılıp varılmadığının tespit edilmesi gereklidir. Yani amaç fonksiyonu bir doğrultu belirtir. Fakat tatmin edici bir noktaya gelinip gelinmediği "malliyet fonksiyonu (cost function)" ya da "kayıp fonksiyonu (loss function)" denilen bir fonksiyonla belirlenir.

Yukarıda da belirtildiği gibi gradient ascent ve gradient descent algoritmalarında bir amaç, bu amaca uygun bir doğrultunun tespit edilmesi gerekmektedir. Sonra bu doğrultuda ilerlenir ve tatmin edici bir noktaya gelinip gelinmediği kontrol edilir. Matematiksel olarak doğrultu belirleme işlemi amaç fonksiyonun birinci türevi alınarak yapılmaktadır. Eğer amaç fonksiyonu birden fazla değişkenden oluşuyorsa bu durumda her değişken için parçalı türevler alınır. Bu parçalı türevlerin oluşturduğu vektöre de gradient vektör denilmektedir. Gradient vektör ters üçgenle temsil edilmektedir. Örneğin:

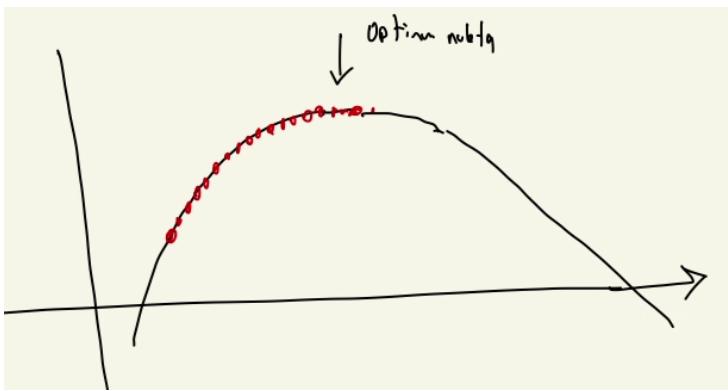
$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right]$$

Pekiyi amaç fonksiyonu nasıl tespit edilecektir? Amaç fonksiyonunun spesifik probleme dayalı olarak tespit edilmesi gereklidir. Bu nedenle her problemin amaç fonksiyonu farklı olabilir. Benzer biçimde maaliyet (ya da kayıp) fonksiyonu da probleme dayalı biçimde tespit edilebilir. Pekiyi neden bu tür problemler denklem yoluyla değil de iteratif doğrultuda ilerleme yoluyla çözümek istenmektedir? Bunun nedeni bazı tür problemlerde denklem çözümünün mümkün olmamasıdır.

Pek çok iteratif yöntemde ilerleme belli bir çarpansal değere orantılı olarak yapılmaktadır. Buna "öğrenme hızı" (learning rate) denilmektedir. Öğrenme hızı değer olarak yükseltilirse noktalar arasındaki sıçramalar daha yüksek olur. Hedefe daha hızlı yaklaşılır. Ancak öğrenme hızı yüksek olduğunda hedefe yaklaşım hızlı olsa da hedefin hassas bir biçimde elde edilmesi zorlaşmaktadır. Benzer biçimde öğrenme hızı düşürüldüğünde hedefin daha hassas bir biçimde bulunması sağlanır. Ancak hedefin yanına yaklaşma uzun zaman alabilmektedir. Örneğin:



Burada öğrenme hızı yüksek tutulduğu için hedefe (maksimum noktaya) hızlı yaklaşılmış ancak hedef hassas bir biçimde belirlenememiştir. Fakat örneğin:



Burada hedefe (maksimum nokta) daha uzun sürede yaklaşılmıştır. Ancak hedef daha hassas bir biçimde belirlenebilmektedir. Biz makine öğrenmesinde "öğrenme hızı (learning rate)" biçiminde bir parametreyle karşılaşlığımızda şu çıkarımda bulunmalıyız: "Bu parametrenin değeri yükseltilirse ben daha kısa zamanda hedefin yakınılarına gelebilirim. Böylece hedefe yaklaşmam daha az bilgisayar zamanıyla gerçekleşir. Ama tam hedefi daha az bir hassasiyetle belirleyebilirim. Eğer ben bu parametrenin değerini düşürürsem hedefe yaklaşmam daha uzun bir zaman alır ama ben hedefi daha hassas belirleyebilirim." Öğrenme hızı algoritmayı kuranın belirlediği görelî bir değerdir. Mutlak değer değildir.

Şimdi biz gradient descent yöntemiyle bir parabolün en küçük y değerini bulmak isteyelim. Bunun için bizim bir x değerinden başlayıp belli bir doğrultuda giderek en düşük değeri bulmamız gereklidir. Parabolün genel fonksiyonu şöyledir:

$$y = ax^2 + bx + c$$

Burada amacımız y değerinin en küçük yapılmasıdır. Amaç fonksiyonumuz da $ax^2 + bx + c$ fonksiyonudur. Buradaki maliyet fonksiyonu amaç fonksiyonundan elde edilen değer ile varmak istenilen değer (örneğimizde 0 olabilir) arasındaki fark olarak tanımlanabilir. Biz bu farkı belirlediğimiz bir epsilon değerinden küçük hale getirmeye çalışabiliyoruz. Gideceğimiz doğrultuyu belirlemek için gradient vektör kullanılır. Zaten bu örneğimizde tek bir değişken olduğundan gradient vektörümüzde amaç fonksiyonun türevinden oluşan tek elemanlı bir vektördür. Bu durumda gradient vektör de tek bir elemandan oluşacaktır. Örneğin:

```
def gradient_descend(a, b, c, y, learning_rate):
    x = 10
    count = 0

    while True:
        yguess = a * x ** 2 + b * x + c
        if abs(yguess - y) < 0.000001: # Loss function
            break
        x += (-2 * a * x - b) * learning_rate
        count += 1

    return x, count

x, count = gradient_descend(1, 0, -4, -4, 0.001)
print(x, count)
y = x ** 2 - 4
print(y)
```

Şimdi de doğrusal regresyon problemini gradient descent algoritmasıyla çözelim. Anımsanacağı gibi doğrusal regresyonda amaç $y = mx + n$ doğrusundaki m ve n değerlerini uygun biçimde belirlemekti. Buradaki amaç fonksiyonu şöyledi:

$$E = \frac{1}{N} \sum_{i=0}^n (y_i - (mx_i + n))^2$$

Şimdi gradient descent algoritması için bu amaç fonksiyonundaki iki değişken olan m ve n için türev alarak gradient vektörü bulmaya çalışalım:

$$\frac{\partial E}{\partial n} = \frac{-2}{N} \sum_{i=0}^N x_i (y_i - mx_i - n)$$

$$\frac{\partial E}{\partial m} = \frac{-2}{N} \sum_{i=0}^N (y_i - mx_i - n)$$

Şimdi artık malzeme fonksiyonu olmadan gradient vektördeki doğrultularda belirli bir miktarda iteratif olarak ilerleyeceğimde kodu yazabilirim. Aşağıdaki fonksiyonda x ve y değerleri noktaların değerleridir. epoch uygulanacak iterasyon sayısını belirtmektedir. learning_rate ise ilerleme adımlarının büyüklüğünü belirtir. Biz burada gradient vektörde belirtilen doğrultuda küçük adımlarla doğruya oluşturan m ve n değerlerini güncelleyeceğiz. Şüphesiz doğrusal regresyonun bu biçimde iteratif olarak çözülmesine gerek yoktur. Zaten doğrusal regresyon için "en küçük karaler" denilen kesin bir formül bulunmaktadır. Ancak biz burada bu örneği yalnızca gradient algoritmaların çalışma biçimini anlatabilmek için veriyoruz:

```

import numpy as np

dataset = np.loadtxt('test.csv', dtype=np.float32, delimiter=',')
dataset_x = dataset[:, 0]
dataset_y = dataset[:, 1]

def linear_regressin_gradient(x, y, epoch, learning_rate):
    N = len(x)
    m = 0
    n = 0

    for i in range(epoch):
        ypred = m * x + n
        dfm = (-2 / N) * np.sum(x * (y - ypred))
        dfn = (-2 / N) * np.sum(y - ypred)
        m = m - learning_rate * dfm
        n = n - learning_rate * dfn

    return m, n

m, n = linear_regressin_gradient(dataset_x, dataset_y, 10000, 0.001)

x = np.linspace(1, 15, 100)
y = m * x + n

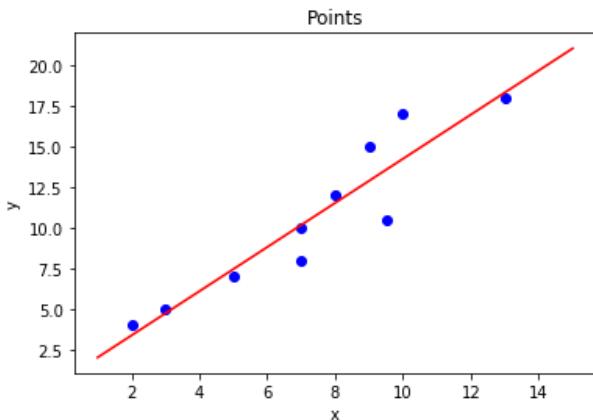
import matplotlib.pyplot as plt

plt.title('Points')
plt.xlabel('x')
plt.ylabel('y')
plt.scatter(dataset[:, 0], dataset[:, 1], color='blue')
plt.plot(x, y, color='red')

```

```
plt.pause(1)
print(m, n)
```

Elde edilen grafik ve doğrunun m ve n değerleri şöyledir:



1.3620186817169206 0.6356675564765912

Şimdi de problemi iteratif olmayan biçimde en küçük kareler yöntemiyle LinearRegression sınıfıyla çözelim:

```
import numpy as np

dataset = np.loadtxt('test.csv', dtype=np.float32, delimiter=',')
from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(dataset[:, 0].reshape(-1, 1), dataset[:, 1])

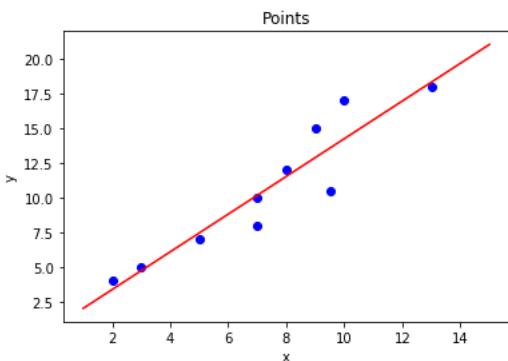
x = np.linspace(1, 15, 100)
y = lr.coef_[0] * x + lr.intercept_

import matplotlib.pyplot as plt

plt.title('Points')
plt.xlabel('x')
plt.ylabel('y')
plt.scatter(dataset[:, 0], dataset[:, 1], color='blue')
plt.plot(x, y, color='red')

plt.pause(1)
print(lr.coef_[0], lr.intercept_)
```

Elde edilen grafik ve doğrunun m ve n değerleri şöyledir:



1.3594104 0.6583328

Stokastik Gradient ve Mini Batch Yöntemleri

Gradient Ascent ve Gradient Descent yöntemlerinde doğrultuda ilerleme veri kümesindeki tüm vektör işleme sokularak yapılmaktadır. Bu ilerleme de belli miktarda bir döngüyle (epoch) devam ettirilir. Halbuki stokastik gradient ascent ve stokastik gradient descent yöntemlerinde doğrultuda ilerleme veri kümesindeki her eleman için tek tek ya da rastgele elemanlar için (stokastik sözcüğü bu nedenden kullanılmıştır) tek tek yapılır. Tüm veri setinin bu biçimde tek tek işleme sokulması bir kez yapılabileceği gibi n kez de yapılmaktadır. Böylece toplam iş miktarı stokastik versiyonda azaltılmış olmaktadır.

Veri kümesindeki tüm vektörün işleme sokulmasına "batch" işlem denilmektedir. Mini batch yöntemi bu bakımdan Gradient yöntemleriyle stokastik gradient yöntemlerinin bir ortalaması gibidir. Mini batch yönteminde tüm veriler içerisindeki belli miktarda kümeler oluşturularak bunlar işleme sokulmaktadır. Örneğin tüm veriler 1000 tane olsun. Biz bu yöntemde 1000 taneyi tek hamlede değil, tek tek de değil belirlediğimiz bir miktarda (örneğin 10'arlık) grup grubu işleme sokarız. Yine bu işlem toplamda baştan sona bir kez ya da n kez yapılmaktadır.

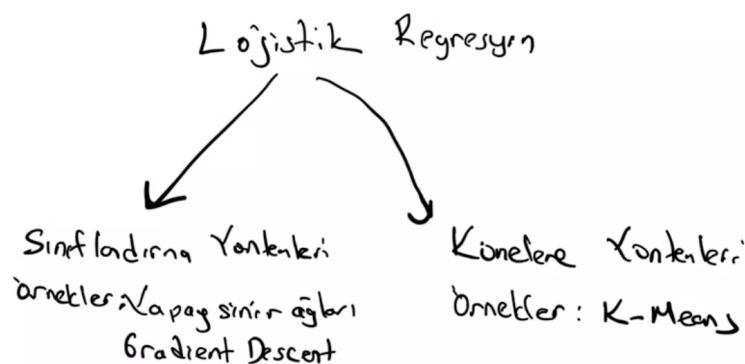
Örneğin veri kümesinde 100000 eleman bulunuyor olsun. Gradient yönteminde her iterasyonda bu 100000 eleman işleme sokulup bundan sonra ağırlıklar güncellenir. Bu işlemler de belli bir sayıda (örneğin 10000 kere) devam ettirilir. Stokastik gradient yönteminde ise bu 100000 eleman tek tek (ya da rastgele elemanlar) işleme sokulur. Bu işlem toplamda bir kez ya da n kez (örneğin 10 kere) tekrar ettirilir. Mini Batch yönteminde ise bu elemanlar küçük gruplar (örneğin 100'erli) halinde işleme sokulmaktadır. Duruma göre bu de bir kez ya da n kez tekrarlanmaktadır.

Genel olarak stokastik gradient descent yöntemi maksimum ya da minimum değere çok daha hızlı yakınsamaktadır. Ancak gradient yöntem stokastik yönteme göre daha iyi sonuç verir. Veri miktarının az olduğu durumlarda stokastik gradient yerine normal gradient yöntem daha uygun kaçmaktadır. (Örneğin 5 noktadan oluşan doğrusal regresyonda stokastik gradient normal gradient yönteme göre çok daha kötü sonuç vermektedir.) Ayrıca veri miktarı az ise stokastik gradient yönteminde learning_rate değerini yükseltmek daha iyi sonuç verebilmektedir.

Gradient Descent Sınıflandırma Yöntemiyle Lojistik Regresyon

Anımsanacağı gibi çıktıları iki değerden oluşan ve sınıflandırma amacıyla kullanılan regresyon modeline lojistik regresyon modeli denilmektedir. (Sonraları çıktı değerinin ikiden fazla olduğu modeller de lojistik regresyon olarak adlandırılabilir.)

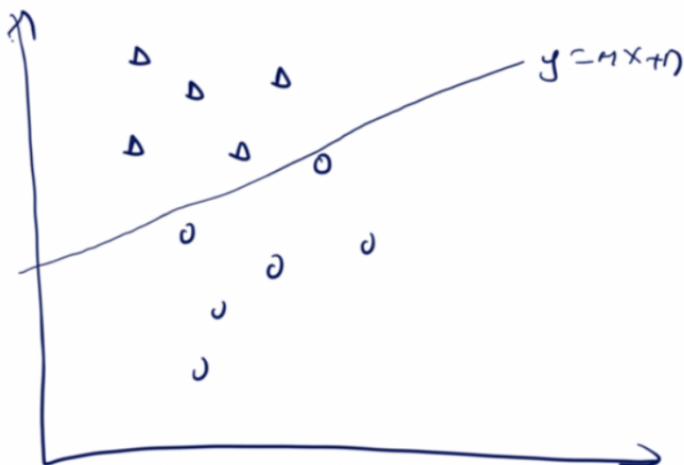
Lojistik regresyon problemlerinin çözümü için genel olarak iki ana yöntem grubundan faydalankmaktadır: Sınıflandırma (classification) ve Kümeleme (clustering). Daha önce görmüş olduğumuz sinir ağları sınıflandırma yöntemine bir örnek oluştururken K-Means ya da DBSCAN yöntemleri kümeleme yöntemlerine örnek oluşturmaktadır.



Sınıflandırma tarzı lojistik regresyon yöntemleri genel olarak denetimli (supervised) yöntemlerdir. Halbuki kümeleme tarzı lojistik regresyon yöntemleri denetimsiz (unsupervised) yöntemlerdir.

Lojistik regresyon problemleri yapay sinir ağları ve kümeleme yöntemi kullanılmadan denetimli biçimde gradient descent yöntemiyle de çözülebilmektedir.

Aslında lojistik regresyon iki kümeyi ayıran bir doğru denkleminin belirlenmesi ile de yapılabilir. Örneğin:



İki kümeyi en iyi biçimde ayıran doğru denklemi elde edilirse hem bir sınıflandırma yapılmış olur hem de yeni değerlerin hangi sınıf içerisinde gireceği yönünde bir kestirim aracı elde edilmiş olur. Kestirim sınıfı belirlenecek noktanın doğrunun yukarısında mı yoksa aşağısında mı kaldığına göre yapılmaktadır. Aslında buradaki yaklaşım yukarıda yaptığımız doğrusal regresyon örneğine benzemektedir.

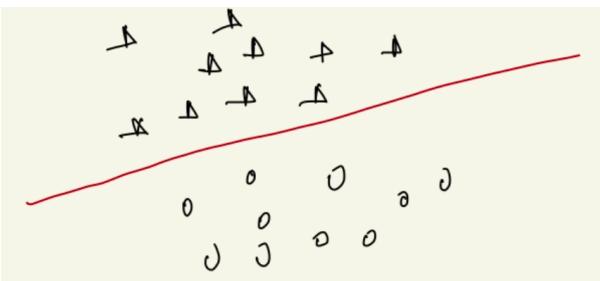
Bu durumda biz birtakım değerleri benzerliklerine göre grupperlendirmek bu kursta gördüğümüz yöntemler şunlar olacaktır:

- Yapay Sinir Ağları (Eğitimli – Sınıflandırma Tarzı)
- K-Means (Eğitimsiz – Kümeleme Tarzı)
- DBScan (Eğitimsiz – Kümeleme Tarzı)
- Gradient Descent (Eğitimli – Sınıflandırma Tarzı)

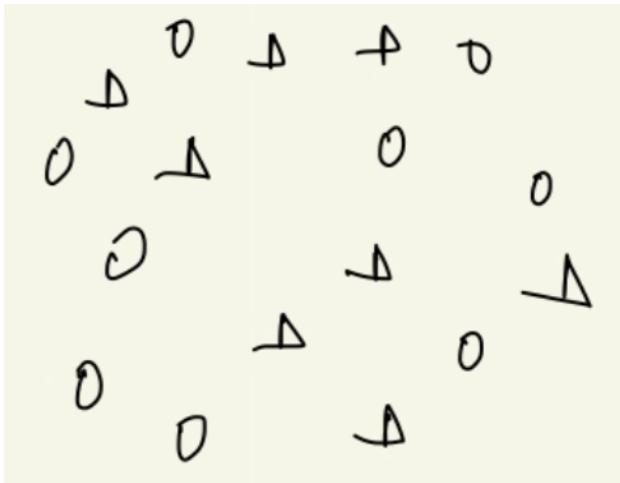
Pekiyi grupperlendirme amacıyla biz bu yöntemlerden hangisini ne zaman tercih etmeliyiz?

- Eğer elimizde yalnızca dataset_x varsa ama bunların hangi grplara girdiğine yönelik dataset_y yoksa biz mecburen K-Means ya da DBSCAN gibi kümeleme yöntemlerini kullanırız.
- Eğer elimizde dataset_x verilerinin yanı sıra bunların zaten hangi grplarda olduğunu belirten bir dataset_y verileri de varsa biz yapay sinir ağları ve gradient descent sınıflandırma yöntemlerini kullanabiliriz. (Tabii aslında eğitim için gereken dataset_y verileri olsa da biz bunlardan faydalananmayıp yine K-Means ve DBSCAN gibi kümeleme yöntemlerini kullanabiliriz. Ancak genel olarak eğitimli sınıflandırma yöntemleri bu durumda kümeleme yöntemlerine göre tercih edilmektedir.)

Pekiyi elimizde eğitimli öğrenme için kullanabileceğimiz dataset_x ve dataset_y verileri bulunuyor olsun. Bu durumda yapay sinir ağları mı yoksa izleyen bölümde göreceğimiz gradient descent yöntemi mi tercih edilmelidir? İşte eğer veriler "doğrusal olarak ayrılabilen (linearly separable)" biçimdeyse yani bir doğru ile iki sınıf birbirlerinden ayrılabiliriyorsa gradient descent yöntemi daha hızlı ve uygun olmaktadır. Ancak veriler bir doğru ile iki kümeye ayrılamıyorsa bu durumda gradient descent yönteminin performansı düşmektedir. Yapay sinir ağları bu tür durumlarda daha uygun olmaktadır. Örneğin aşağıdaki veriler doğrusal olarak ayrılabilen biçimdedir:

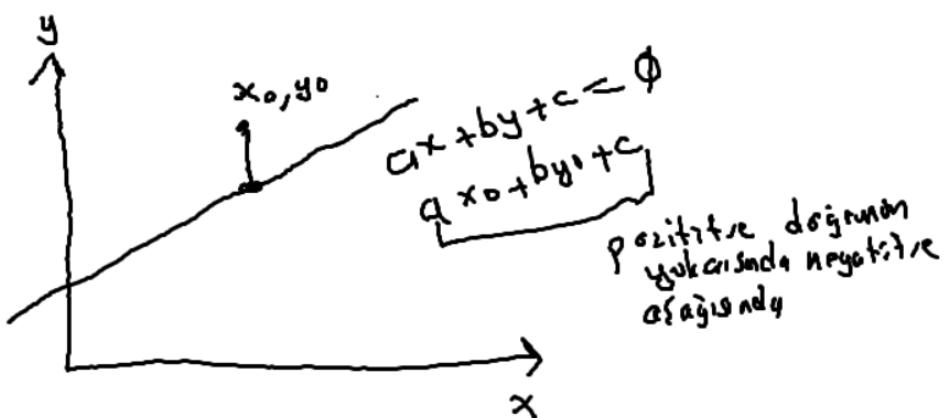


Burada biz yapay sinir ağı yerine gradient descent yöntemini tercih edebiliriz. Aşağıdaki veriler ise doğrusal olarak ayrılabilir olmayan biçimdedir:



Burada biz bir doğru denklemi elde ederek verileri iyi bir biçimde sınıflandıramayız. İşte bu tür durumlarda yapay sinir ağıları tercih edilmelidir. Maalesef sınıflandırma problemlerinin önemli bir bölümü "doğrusal olarak ayrılabilir" biçimde değildir.

Pekiyi iki sınıfı birbirinden ayırmak için elde edeceğimiz doğru denkleminde amaç fonksiyonu ne olmalıdır? Yani biz iterasyonlar sırasında neyi küçültmeye çalışmalıyız? İşte burada iki kümeyi ayıran doğru denkleminin genel biçimini $ax + by + c = 0$ olarak ifade edilebilir. Bir noktayı bu doğru denkleminde x ve y olarak yerleştirdiğimizde çıkan değer bu noktanın doğruya uzunluğunu verir.



Biz bu yöntemde denetimli öğrenme uygulayacağımıza göre gerçek değerlerle buradaki uzunluk arasında bir ilişki kurmamız gereklidir. Bunun da en basit yolu sigmoid fonksiyonunu kullanmaktır. Yani biz noktanın doğruya uzunluğunu sigmoid fonksiyonuna sokarsak bu fonksiyon bize 0 ile 1 arasında bir değer verecektir. Ve aynı zamanda söz konusu olan nokta doğrunun ne kadar yukarıındaysa 1'e o kadar yaklaşacak, ne kadar aşağıındaysa 0'a o kadar yaklaşacaktır. O halde amaç fonksiyonu gerçek değerlerden bu sigmoid değerinin çıkartılması biçiminde

oluşturulabilir. Biz de bu amaç fonksiyonunu minimize etmeye çalışabiliriz. Tabii bu işlemi yavaş yavaş bir doğrultu biçiminde gradient descent yöntemiyle yaparız. Çözüme bu biçimde varan bir program şöyle yazılabilir:

```
import numpy as np

dataset = np.loadtxt('test.csv', skiprows=1, delimiter=',', dtype=np.float32)
dataset_x = dataset[:, :2]
dataset_y = dataset[:, 2].reshape(-1, 1)

dataset_x = np.append(dataset_x, np.ones((dataset_x.shape[0], 1)), axis=1)

def sigmoid(x):
    return 1.0 / (1 + np.exp(-x))

def gradeint_descent_logistic(x, y, learning_rate=0.001, epoch=50000):
    weights = np.ones((x.shape[1], 1))

    for k in range(epoch):
        h = sigmoid(np.matmul(x, weights))
        error = y - h
        weights = weights + learning_rate * np.matmul(x.transpose(), error)

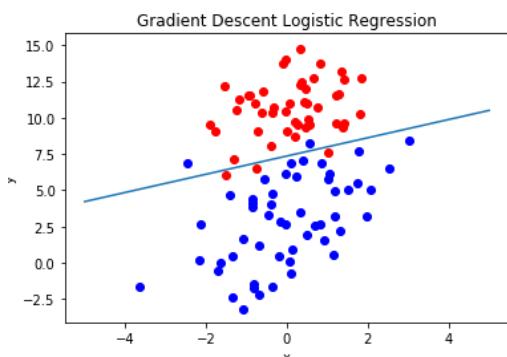
    return weights

weights = gradeint_descent_logistic(dataset_x, dataset_y)

import matplotlib.pyplot as plt

plt.title("Gradient Descent Logistic Regression")
plt.scatter(dataset[dataset[:, 2] == 1, 0], dataset[dataset[:, 2] == 1, 1], color='b')
plt.scatter(dataset[dataset[:, 2] == 0, 0], dataset[dataset[:, 2] == 0, 1], color='r')
plt.xlabel("x")
plt.ylabel("y")

x = np.linspace(-5, 5, 100)
y = (-weights[2] - weights[0] * x) / weights[1]
plt.plot(x, y)
```



Buradaki "test.csv" dosyasının içeriği şöyledir:

```
x,y,class
-0.017612,14.053064,0
-1.395634,4.662541,1
-0.752157,6.538620,0
-1.322371,7.152853,0
0.423363,11.054677,0
0.406704,7.067335,1
0.667394,12.741452,0
-2.460150,6.866805,1
0.569411,9.548755,0
-0.026632,10.427743,0
0.850433,6.920334,1
```

1.347183,13.175500,0
1.176813,3.167020,1
-1.781871,9.097953,0
-0.566606,5.749003,1
0.931635,1.589505,1
-0.024205,6.151823,1
-0.036453,2.690988,1
-0.196949,0.444165,1
1.014459,5.754399,1
1.985298,3.230619,1
-1.693453,-0.557540,1
-0.576525,11.778922,0
-0.346811,-1.678730,1
-2.124484,2.672471,1
1.217916,9.597015,0
-0.733928,9.098687,0
-3.642001,-1.618087,1
0.315985,3.523953,1
1.416614,9.619232,0
-0.386323,3.989286,1
0.556921,8.294984,1
1.224863,11.587360,0
-1.347803,-2.406051,1
1.196604,4.951851,1
0.275221,9.543647,0
0.470575,9.332488,0
-1.889567,9.542662,0
-1.527893,12.150579,0
-1.185247,11.309318,0
-0.445678,3.297303,1
1.042222,6.105155,1
-0.618787,10.320986,0
1.152083,0.548467,1
0.828534,2.676045,1
-1.237728,10.549033,0
-0.683565,-2.166125,1
0.229456,5.921938,1
-0.959885,11.555336,0
0.492911,10.993324,0
0.184992,8.721488,0
-0.355715,10.325976,0
-0.397822,8.058397,0
0.824839,13.730343,0
1.507278,5.027866,1
0.099671,6.835839,1
-0.344008,10.717485,0
1.785928,7.718645,1
-0.918801,11.560217,0
-0.364009,4.747300,1
-0.841722,4.119083,1
0.490426,1.960539,1
-0.007194,9.075792,0
0.356107,12.447863,0
0.342578,12.281162,0
-0.810823,-1.466018,1
2.530777,6.476801,1
1.296683,11.607559,0
0.475487,12.040035,0
-0.783277,11.009725,0
0.074798,11.023650,0
-1.337472,0.468339,1
-0.102781,13.763651,0
-0.147324,2.874846,1
0.518389,9.887035,0
1.015399,7.571882,0
-1.658086,-0.027255,1
1.319944,2.171228,1
2.056216,5.019981,1
-0.851633,4.375691,1
-1.510047,6.061992,0
-1.076637,-3.181888,1
1.821096,10.283990,0
3.010150,8.401766,1
-1.099458,1.688274,1
-0.834872,-1.733869,1
-0.846637,3.849075,1
1.400102,12.628781,0
1.752842,5.468166,1
0.078557,0.059736,1

```

0.089392, -0.715300, 1
1.825662, 12.693808, 0
0.197445, 9.744638, 0
0.126117, 0.922311, 1
-0.679797, 1.220530, 1
0.677983, 2.556666, 1
0.761349, 10.693862, 0
-2.168791, 0.143632, 1
1.388610, 9.341997, 0
0.317029, 14.739025, 0

```

Bu programda yapılanlar şunlardır:

- Önce "test.csv" dosyasından değerler okunmuştur. Bu işlemin sonucunda dataset_x ve dataset_y biçiminde iki ndarray elde edilmiştir. dataset_x iki sütunlu aşağıdaki biçimde bir matriztir:

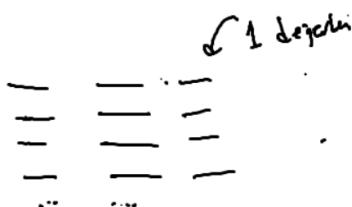


dataset_y ise tek sütunlu aşağıdaki biçimsel görünümüne sahiptir:



dataset_y 0 ve 1 değerlerinden oluşmaktadır. Bu değerler sınıfları belirtmektedir.

- Bizim eğitimden amacımız aslında toplamda $ax + by + c$ doğrusundaki a, b, c değerlerini bulmaktır. Bunun için dataset_x'ye bir sütun daha ekleyip (c için) onu şu görünümüne getirdik:



- Bundan sonra programda a, b, c değerleri için weights isminde 3 elemanlı bir sütun vektörü oluşturduk.



- Daha sonra programda noktaların weights katsayılarıyla belirtilen doğruya uzunlukları hesaplanmış, bunlar sigmoid fonksiyonuna sokulup gerçek y değerleri ile farkları hesaplanmıştır:

```

for k in range(epoch):
    h = sigmoid(np.matmul(x, weights))
    error = y - h
    weights = weights + learning_rate * np.matmul(x.transpose(), error)

```

- Buradaki amaç fonksiyonu (yani minimize edilecek fonksiyon) $y - h$ yani error fonksiyonudur. Görüldüğü gibi her yinelemede x matrisi ile error değeri çarpılarak weights güncellenmiştir.

Pekiyi yukarıdaki örnekte sistemin başarısını nasıl test edebiliriz. Bunun için ilk akla gelen yöntem başarılı olanların oranını tespit etmektir. Bu tespit şöyle yapılabilir:

```
result = np.matmul(dataset_x, weights)
success_ratio = np.sum((result > 0) == dataset_y) / len(dataset_y)
print(success_ratio)
```

Buradan 0.95 değeri elde edilmiştir.

Yukarıdaki yöntem stochastic gradient descent uygulanarak aşağıdaki biçimde dönüştürülebilir:

```
import numpy as np

dataset = np.loadtxt('test.csv', skiprows=1, delimiter=',', dtype=np.float32)
dataset_x = dataset[:, :2]
dataset_y = dataset[:, 2].reshape(-1, 1)

dataset_x = np.append(dataset_x, np.ones((dataset_x.shape[0], 1)), axis=1)

def sigmoid(x):
    return 1.0 / (1 + np.exp(-x))

def stochastic_gradient_descent_logistic(x, y, learning_rate=0.001, epoch=150):
    weights = np.ones(x.shape[1])

    for i in range(epoch):
        for k in range(x.shape[0]):
            learning_rate = 4 / (1 + i + k) + 0.01
            index = int(np.random.randint(0, x.shape[0]))
            h = sigmoid(np.matmul(x[index], weights))
            error = y[index] - h
            weights = weights + learning_rate * error * x[index]

    return weights

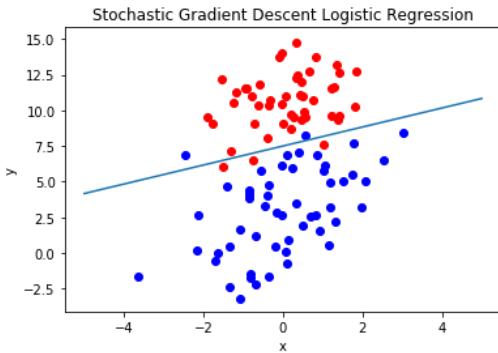
weights = stochastic_gradient_descent_logistic(dataset_x, dataset_y)

import matplotlib.pyplot as plt

plt.title("Stochastic Gradient Descent Logistic Regression")
plt.scatter(dataset[dataset[:, 2] == 1, 0], dataset[dataset[:, 2] == 1, 1], color='b')
plt.scatter(dataset[dataset[:, 2] == 0, 0], dataset[dataset[:, 2] == 0, 1], color='r')
plt.xlabel("x")
plt.ylabel("y")

x = np.linspace(-5, 5, 100)
y = (-weights[2] - weights[0] * x) / weights[1]
plt.plot(x, y)

result = np.matmul(dataset_x, weights)
success_ratio = np.sum((result > 0) == dataset_y) / len(dataset_y)
print(success_ratio)
```



Buradan elde edilen başarı oranı %50.4 bulunmuştur. Şimdi bu yöntemde learning_rate'in de belli bir aralıkta değiştirilmesiyle iyileştirilmiş biçimiyle oluşturulabilir:

```

import numpy as np

dataset = np.loadtxt('test.txt')
dataset_x = dataset[:, :2]
dataset_y = dataset[:, 2].reshape(-1, 1)

dataset_x = np.append(dataset_x, np.ones((dataset_x.shape[0], 1)), axis=1)

def sigmoid(x):
    return 1.0 / (1 + np.exp(-x))

def gradeint_descent_logistic(x, y, epoch=50):
    weights = np.ones(x.shape[1])

    for i in range(epoch):
        for k in range(x.shape[0]):
            learning_rate = 4 / (1 + i + k) + 0.01
            index = int(np.random.randint(0, x.shape[0]))
            h = sigmoid(np.matmul(x[index], weights))
            error = y[index] - h
            weights = weights + learning_rate * error * x[index]

    return weights

weights = gradeint_descent_logistic(dataset_x, dataset_y)

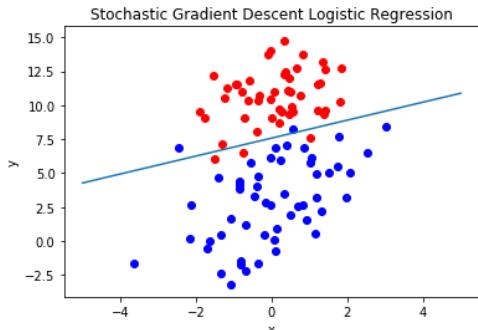
import matplotlib.pyplot as plt

plt.title("Stochastic Gradient Descent Logistic Regression")
plt.scatter(dataset[dataset[:, 2] == 1, 0], dataset[dataset[:, 2] == 1, 1], color='b')
plt.scatter(dataset[dataset[:, 2] == 0, 0], dataset[dataset[:, 2] == 0, 1], color='r')
plt.xlabel("x")
plt.ylabel("y")

x = np.linspace(-5, 5, 100)
y = (-weights[2] - weights[0] * x) / weights[1]
plt.plot(x, y)

result = np.matmul(dataset_x, weights)
success_ratio = np.sum(result > 0 == dataset_y) / len(dataset_y)
print(success_ratio)

```



Buradaki başarı değeri 9.95 bulunmuştur.

Scikit-Learn'deki Hazır LogisticRegression Sınıfının Kullanılması

Aslında Scikit-Learn içerisinde zaten gradient descent yöntemiyle lojistik regresyon işlemini yapan LogisticRegression isimli yetenekli bir sınıf vardır. Bu sınıfın kullanılması oldukça kolaydır. Tek yapacağımız şey LogisticRegression sınıfı türünden bir nesne yaratmak ve bu nesnelerle fit işlemini uygulamaktır. Bundan sonra artık predict metoduyla tahminleme yapabiliriz. Sınıfın score isimli metodu yapılan regresyonun başarı yüzdesini bize vermektedir. Yine burada elde edilen doğru denkleminin katsayılarını biz sınıfın coef_ isimli örnek özniteligidenden, eksen kesim noktasını da sınıfın intercept_ örnek özniteligidenden elde edebiliriz. Yukarıdaki örneğin LogisticRegression sınıfıyla gerçekleştirimi söyle olabilir:

```
import numpy as np

dataset = np.loadtxt('test.csv', skiprows=1, delimiter=',', dtype=np.float32)
dataset_x = dataset[:, :2]
dataset_y = dataset[:, 2].reshape(-1, 1)

from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
lr.fit(dataset_x, dataset_y)

import matplotlib.pyplot as plt

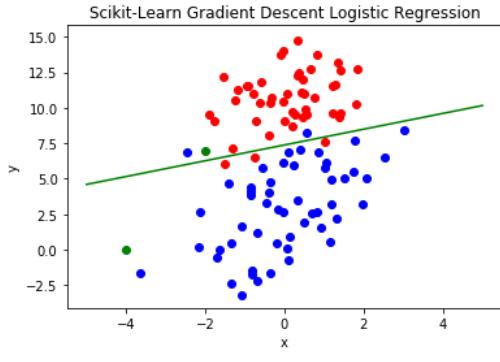
plt.title("Scikit-Learn Gradient Descent Logistic Regression")
plt.xlabel("x")
plt.ylabel("y")
plt.scatter(dataset[dataset[:, 2] == 1, 0], dataset[dataset[:, 2] == 1, 1], color='b')
plt.scatter(dataset[dataset[:, 2] == 0, 0], dataset[dataset[:, 2] == 0, 1], color='r')
x = np.linspace(-5, 5, 100)
y = (-lr.coef_[0, 0] * x - lr.intercept_[0]) / lr.coef_[0, 1]
plt.plot(x, y, color='green')

predict_points = np.array([[-2, 7], [-4, 0]])

plt.scatter(predict_points[:, 0], predict_points[:, 1], color='g')

plt.pause(1)
score = lr.score(dataset_x, dataset_y)
print('score = {}'.format(score))

result = lr.predict(predict_points)
print('{} point prediction = {}'.format(predict_points, result))
```



```
score = 0.95
[[ -2  7]
 [ -4  0]] point prediction = [0. 1.]
```

LogisticRegression nesnesini yaratırken aslında `__init__` metodunda pek çok argüman da girebiliriz. Bu argümanların hepsi default değer almış durumdadır. Aşağıda fonksiyonun parametre listesi görülmektedir:

```
class sklearn.linear_model.LogisticRegression(penalty='L2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='Lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None)
```

Buradaki `max_iter` parametresi gradient descent uygulanırken kaç yineleme yapılacağını (yani bizim örneğimizdeki epoch) belirtmektedir. Bu parametrenin default değerinin 100 olduğunu dikkat ediniz. Fonksiyonun `solver` parametresi çözüm algoritmasını belirlemek için kullanılmaktadır. Küçük veri kümesi için bu parametrenin 'liblinear' girilmesi uygun olabilmektedir.

Lojistik regresyon uygulamakarında veri bilimcisi gerçek değerlerle tahmin edilen değerler arasındaki farkı görmek isteyebilir. Örneğin gerçekte 1 olması gereken değerlerin kaç tanesi 1 kaç tanesi 0 olmuştur gibi. İşte bunu gösteren matrise "confusion matrix" denilmektedir. Confusion matrix şüphesiz manuel biçimde oluşturulabilir. Ancak Scikit-Learn içerisinde `sklearn.metrics` modülünde bu matrisi oluşturan hazır bir `confusion_matrix` isimli fonksiyon vardır. Bu fonksiyon bizden iki parametre ister. Birinci parametre gerçek y değerlerini ikinci parametre ise tahmin edilmiş olan y değerlerini belirtmektedir. Yukarıdaki örneğimiz için confusion matrix şöyleden oluşturulmuştur:

```
from sklearn.metrics import confusion_matrix

result = confusion_matrix(dataset_y, lr.predict(dataset_x))
print(result)

[[44  3]
 [ 2 51]]
```

Burada sınıflandırmada kullanılacak sınıf sayısı iki olduğu için confusion matrix 2x2 boyutundadır. Genel olarak sınıf sayısı n olmak üzere bu matris nxn boyutunda olur. Confusion matrix'te (i, j) hücresi gerçek durumun i olması halinde bunun j biçiminde tespit edildiği nokta sayısını vermektedir. Örneğin yukarıdaki matriste gerçek sonuç 0 iken 0 sonucu 44 kez elde edilmiştir. Öte yandan gerçek sonuç 0 iken 1 sonucu 3 kez elde edilmiştir. Benzer biçimde gerçek sonuç 1 iken 0 sonucu 2 kez, gerçek sonuç 1 iken 1 sonucu 51 kez elde edilmiştir. Gerçekten de grafik incelendiğinde kırmızıların 3 tanesinin doğrunun altında olduğu, mavilerin de 2 tanesinin doğrunun üzerinde olduğu görülmektedir.

Şüphesiz biz bu confusion matrix'i nokta sayısına bölersek oransal bir matris elde edebiliriz:

```
result_ratio = result / len(dataset_y)
print(result_ratio)

[[0.44 0.03]
 [0.02 0.51]]
```

Scikit-Learn kütüphanesindeki hazır LogisticRegression sınıfı aslında çok daha genel gerçekleştirilmiştir. Biz bu sınıfla sonraki konularda da ele alınacağı gibi çok sütundan oluşan ve birden fazla sınıftan ve etiketten oluşan lojistik regresyon problemlerini de çözebiliriz. Önceki konularda da belirtildiği gibi lojistik regresyon denildiğinde ikili sınıflandırma anlaşılmaktadır. Fakat LogisticRegression sınıfı aslında çok çok sınıflı lojistik regresyon modellerinde de kullanılabilmektedir. İzleyen bölümlerde böyle çok sınıflı lojistik regresyon örnekleri verilemektedir. Ayrıca Scikit-Learn kütüphanesindeki LogisticRegression sınıfı çok etiketli (çok değişkenli) sınıflandırmalar için de kullanılabilmektedir.

Çok Sınıflı Gradient Descent Lojistik Regresyon İçin IRIS Örneği

Anımsanacağı gibi Iris (Zambak) veri kümesi 4 özellikten hareketle zambakların türünü 3 seçenekten biri biçiminde belirleyebilmek için oluşturulmuş bir örnek veri kümesidir. Biz daha önce bu veri kümesini doğrudan dosyadan yükleyerek kullanmıştık. Aslında bu veri kümesi hazır biçimde Scikit-Learn içerisinde de bulunmaktadır. Aşağıda Iris veri kümesi için lojistik regresyon örneği verilmiştir:

```
from sklearn.datasets import load_iris

iris = load_iris()
dataset_x = iris.data
dataset_y = iris.target

from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.25)

from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
lr.fit(training_dataset_x, training_dataset_y)

result = lr.score(test_dataset_x, test_dataset_y)
print('score = {}'.format(result))

from sklearn.metrics import confusion_matrix

predicted_test_y = lr.predict(test_dataset_x)
cm = confusion_matrix(test_dataset_y, predicted_test_y)
print(cm)
```

Buradan elde edilen skor ve confusion_matrix şöyledir:

```
score = 0.9736842105263158
[[13  0  0]
 [ 0 12  1]
 [ 0  0 12]]
```

Göründüğü gibi test verileri ile yapılan denemede yalnızca 1 numaralı tür 1 kere 2 olarak tespit edilmiştir.

Iris örneğindeki %97'lik bu başarı Iris verilerinin "doğrusal olarak ayırtılabilir (linearly separable)" olduğunu göstermektedir. Eğer Iris verileri doğrusal olarak ayırtılabilir olmasaydı bizim başarımız düşük kalırdı. Böylece biz burada gradient descent lojistik regresyon kullanmamız gerektiğini anladık ve yapay sinir ağları ya da ileride göreceğimiz "destek vektör makineleri (support vector machines)" gibi alternatif yöntemleri denememiz uygun olurdu.

Buradaki Iris örneğinde coef_ ve intercept_ örnek özniteliklerini yazdırduğumuzda şunları görmekteyiz:

```

print(lr.coef_)
print()
print(lr.intercept_)

[[-0.44695305  0.7897291 -2.34012927 -0.95547692]
 [ 0.60398165 -0.30810544 -0.19988568 -0.93823562]
 [-0.15702859 -0.48162367  2.54001496  1.89371254]]

[ 9.86891014  1.52164148 -11.39055162]

```

Burada katsayı matrisinin 3×4 'lük olduğunu görürsünüz. Matrisin 3 satırdan oluşması sonraki konuda açıklandığı gibi 3 sınıfın 3 doğru ile ayrıştırılmasından kaynaklanmaktadır. Yani 3 sınıfı bir lojistik regresyonda biz bu 3 sınıfı 3 ayrı doğruya ayrırtırabilmekteyiz. Katsayı matrisinin 4 sütundan oluşması ise regresyonda 4 tane değişkenin (yani sütunun) bulunuyor olmasındandır. Eksen kesim matrisinin 1×3 'lük olduğunu da dikkat ediniz. Her doğrunun bir tane eksen kesim noktası bulunmaktadır.

Çok Sınıflı Lojistik Regresyon İçin Mnist Örneği

Anımsanacağı gibi Mnist veri kümesi her biri 28×28 pixelden oluşan gray-scale resimlerden oluşuyordu. Bu resimlerde 0-9 arasındaki sayılar bulunmaktadır. Bizim amacımız da yeni bir resmi verdığımızda bunun üzerindeki sayının belirlenmesiydi. Biz Mnist örneğini daha önce yapay sinir ağları konusunda yapmıştık. Şimdi aynı örneği LogisticRegression sınıfıyla yapalım:

```

from tensorflow.keras.datasets import mnist

(training_dataset_x, training_dataset_y), (test_dataset_x, test_dataset_y) = mnist.load_data()

training_dataset_x = training_dataset_x.reshape(-1, 28 * 28)[:2000, :]
training_dataset_y = training_dataset_y[:2000]
test_dataset_x = test_dataset_x.reshape(-1, 28 * 28)

from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(verbose=2)
lr.fit(training_dataset_x, training_dataset_y)

result = lr.score(test_dataset_x, test_dataset_y)
print('score = {}'.format(result))

```

Elde ettiğimiz başarı skoru şöyledir:

```
score = 0.9255
```

Bu yüksek başarı değerinden yine Mnist verilerinin "doğrusal olarak ayırtılabilir (linearly separable)" olduğunu görüyorsunuz. Confusion matrix ise şöyledir:

```

from sklearn.metrics import confusion_matrix

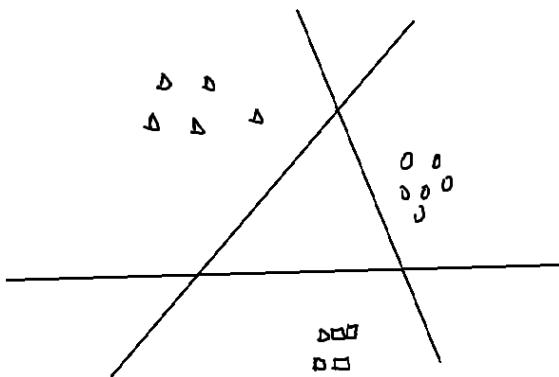
predicted_test_y = lr.predict(test_dataset_x)
cm = confusion_matrix(test_dataset_y, predicted_test_y)
print(cm)

```

```
[[ 963     0     0     3     1     3     4     4     2     0]
 [  0 1112     4     2     0     1     3     2    11     0]
 [  3   10   926    15     6     4    15     8    42     3]
 [  4   1   21   916     1    26     3     9    22     7]
 [  1   1     7     3   910     0     9     7    10    34]
 [ 11   2     1    33    11   776    11     6    35     6]
 [  9   3     7     3     7    16   910     2     1     0]
 [  1   6    24     5     7     1     0   951     3    30]
 [  8   7     6    23     6    26    10    10   869     9]
 [  9   7     0    11    25     6     0    22     7   922]]
```

Çok Sınıflı Lojistik Regresyonlarda Regresyon Doğruları

Daha önce çok sınıflı (yani sınıf sayısı 2'den fazla olan) lojistik regresyonlar için sınıf sayısı kadar doğru oluşturulduğunu belirtmiştim. Örneğin Iris verilerinde zambaklar 3 sınıfından birine sokulmak istenmektedir. Yani Iris örneğindeki sınıf sayısı 3'tür. Bu durumda LogisticRegression sınıfı bize 3 tane doğru vermelidir. Gerçekten de anımsanacağı gibi bu problemde `coef_matrix` 3X4'lük boyuttaydı. Tabii her doğru eksenleri kestiğine göre her doğru için bir `intercept` olması gereklidir. Bu durumda Iris örneğinde bize verilen `intercept` vektörü de 3 elemanlıdır. Peki neden n sınıf için n tane doğru gerekmektedir? Aslında bu n doğrunun her biri bir sınıfı diğer tümünden ayırmak için kullanılmaktadır.



Çok sınıflı lojistik regresyon problemlerinin "doğrusal olarak ayrılabilir (linearly separable)" olması aslında her bir sınıfın yalnızca bir doğru ile diğerlerinden ayrılabilir olması anlamına gelmektedir. Daha önce belirttiğim gibi doğrusal olarak ayırtılamayan sınıflandırma problemlerinde gradient descent yöntemiyle lojistik regresyonun başarısı düşük olacaktır.

Şimdi biz bu doğruları elde edebilmek için 2 sütunlu 3 sınıfı bir lojistik regresyon örneği yapalım:

```
import numpy as np

from sklearn.datasets import make_blobs

dataset_x, dataset_y = make_blobs(n_samples=100, n_features=2, centers=3, cluster_std=1)

from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
lr.fit(dataset_x, dataset_y)
from sklearn.datasets import make_blobs

dataset_x, dataset_y = make_blobs(n_samples=100, n_features=2, centers=3,
cluster_std=0.5)

from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
```

```

lr.fit(dataset_x, dataset_y)

score = lr.score(dataset_x, dataset_y)
print(score)

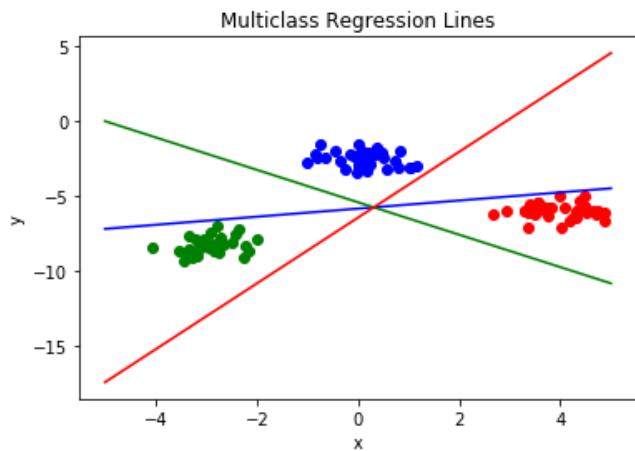
import numpy as np
import matplotlib.pyplot as plt

plt.title('Multiclass Regression Lines')
plt.xlabel('x')
plt.ylabel('y')

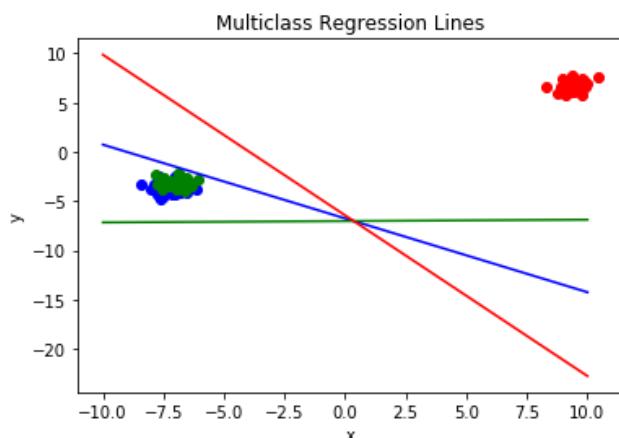
plt.scatter(dataset_x[dataset_y == 0, 0], dataset_x[dataset_y == 0, 1], color='blue')
plt.scatter(dataset_x[dataset_y == 1, 0], dataset_x[dataset_y == 1, 1], color='green')
plt.scatter(dataset_x[dataset_y == 2, 0], dataset_x[dataset_y == 2, 1], color='red')

x = np.linspace(-5, 5, 100)
for i in range(3):
    y = (-lr.coef_[i, 0] * x - lr.intercept_[i]) / lr.coef_[i, 1]
    plt.plot(x, y, color=['blue', 'green', 'red'][i])

```



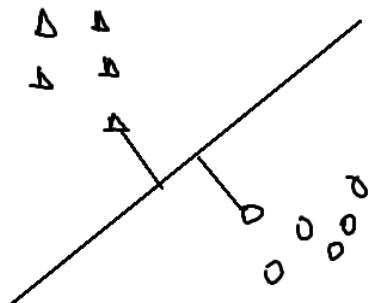
Buradan elde edilen score 1'dir. Bu mükemmel bir ayrıştırma anlamına gelmektedir. Şimdi doğrusal olarak ayrılabilir olmayan bir örnek verelim:



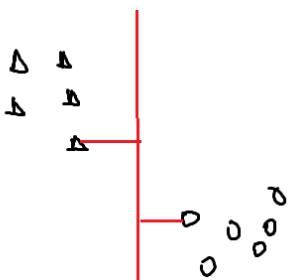
Buradaki skor 0.78 olarak elde edilmiştir.

DESTEK VEKTÖR MAKİNELERİ (Support Vector Machines)

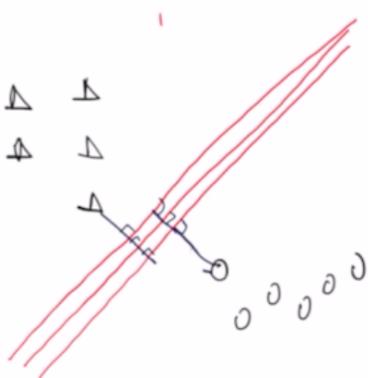
Destek vektör makineleri de eğitimli (supervised) sınıflandırma (classification) yöntemlerinden biridir. Destek vektör makinelerinde de aslında sınıfları ayırmak için doğrular kullanılmaktadır. Ancak bu doğrular en yakın noktaları ayırmaya amacıyla oluşturulmaktadır. Bu en yakın noktalara destek vektörleri (support vectors) denilmektedir. Örneğin:



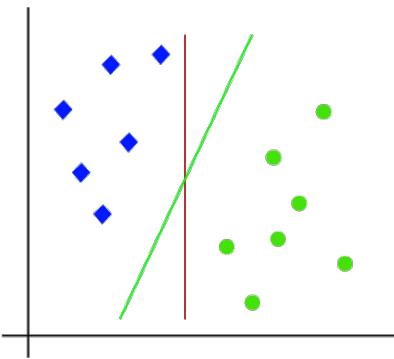
Burada iki sınıfı ayırmak için öyle bir doğru oluşturulur ki bu doğru şu özelliğe sahiptir: "Bu doğuya en yakın iki sınıfındaki noktalar ele alındığında bu noktalar ile doğru arasındaki toplam uzaklık diğer alternatif doğrulara göre daha fazla olmalıdır". Buradaki noktanın doğuya uzaklığında dikme uzaklığı kullanılmaktadır. Örneğin aşağıdaki diğer alternatif doğruya göz önüne alalım:



Burada alternatif bir bir doğru görülmektedir. Pekiyi yukarıdaki hangi iki doğru yukarıda tanımladığımız tanıma daha yakındır? İşte hedefimiz doğuya iki sınıfındaki en yakın noktaların uzakları toplamını maksimum yapan doğruya elde etmektir. Doğuya en yakın noktalarla doğru arasındaki toplam uzaklığa marjin denir. Pekiyi marjini aynı olan sonsuz sayıda paralel doğru elde edilebileceğine göre bunlardan hangisi seçilecektir. İşte en yakın iki noktaya uzaklıklar eşit olan marjini en yüksek doğru seçilmektedir.



. Aşağıda iki sınıflı bir model için alternatif iki destek doğrusu görüyorsunuz:



Burada gözle de görüldüğü gibi yeşil doğru kırmızı doğrudan daha iyidir.

Destek vektör makinelерinde uygun doğrunun elde edilmesi süreci teorik bakımdan biraz karışıkır. Biz burada bu doğrunun nasıl elde edileceği üzerinde durmayacağız. Bunun için başka kaynaklara başvurulabilir.

Destek vektör makineleri Scikit-Learn içerisindeki SVC sınıfıyla temsil edilmektedir. Sınıfın `__init__` metodunun parametrik yapısı şöyledir:

```
class sklearn.svm.SVC(C=1.0, kernel='rbf', degree=3, gamma='scale', coef0=0.0, shrinking=True,
probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1,
decision_function_shape='ovr', break_ties=False, random_state=None)
```

Aslında destek vektör makineleri yalnızca sınıflandırma amacıyla değil aynı zamanda regresyon amacıyla kullanılmaktadır. sklearn.svm modülündeki bu SVC (Support Vector Classification) sınıflandırma amacıyla SVR (Support Vector Regression) sınıfı da regresyon amacıyla kullanılmaktadır.

Şimdi destek vektör makineleri ile sınıflandırmaya bir örnek verelim. Bu sınıflandırma örneğinde scikit-learn içerisinde bulunan kanser tarama verilerinin gerçekten kanser olup olmadığını belirlemeye çalışacağız. Scikit-Learn'deki bu veri kümesinde kişilerden meme kanseri riski için önemli olabilecek bazı biyomedikal veriler elde edilmiştir. Amaç bu verilere dayanarak kişideki lezyonun iyi huylu mu (benign) yoksa kötü huylu mu (malign) olduğuna karar vermektir. Bu sınıflandırma probleminin destek vektör makineleri ile çözümü şöyledir:

```
from sklearn.datasets import load_breast_cancer

bc = load_breast_cancer()
print('Sütun isimleri: {}'.format(bc.feature_names))
print('Sınıf İsimleri: {}'.format(bc.target_names))

dataset_x = bc.data
dataset_y = bc.target

from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.3)

from sklearn.svm import SVC

svc = SVC(kernel='linear')
svc.fit(dataset_x, dataset_y)

score = svc.score(test_dataset_x, test_dataset_y)
print(score)

result = svc.predict(test_dataset_x)
ratio = sum(result == test_dataset_y) / len(test_dataset_y)
print(ratio)
```

Burada SVC nesnesi yaratılırken kernel olarak 'linear' verildiğine dikkat ediniz. Bu kernel doğrusal olarak ayrılabilir veri kümesi için en iyi yöntemdir. Anımsanacağı gibi doğrusal olarak ayrılabilir veriler için lojistik regresyon da kullanılabilmektedir. Fakat eğer veriler doğrusal olarak ayırtılabilir değilse destek vektör makinelerinde ismine "kernel trick" denilen yöntemle boyut yükseltilmesi yapılarak sorun çözülmektedir. Halbuki lojistik regresyonda böyle bir "kernel trick" durumu yoktur. (Kernel trick aslında doğrusal olarak ayırtılamayan veri kümelerini boyut yükselterek doğrusal olarak ayırtılabilir hale getiren bir yöntemdir.)

SVC sınıfının predict metodu ile kestirim yapılabilir. Yukarıdaki örnekte test_dataset_x verileri üzerinde predict ile kestirimde bulunulmuştur. score fonksiyonu yine kestirim yapıp başarı yüzdesini bize verir. Tabii biz bu başarı yüzdesini yukarıdaki örnekte olduğu gibi manuel bir biçimde de elde edebiliriz. Söz konusu kanser verileri için başarı oranı %96.4 bulunmuştur. Şimdi aynı örneği Lojistik regresyon ile yapalım:

```
from sklearn.datasets import load_breast_cancer

bc = load_breast_cancer()
print('Sütun isimleri: {}'.format(bc.feature_names))
print('Sınıf İsimleri: {}'.format(bc.target_names))

dataset_x = bc.data
dataset_y = bc.target

from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.3)

from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
lr.fit(dataset_x, dataset_y)

score = lr.score(test_dataset_x, test_dataset_y)
print(score)

result = lr.predict(test_dataset_x)
ratio = sum(result == test_dataset_y) / len(test_dataset_y)
print(ratio)
```

Elde edilen sonuçlar benzerdir.

Şimdi de kendimiz make_blobs fonksiyonuyla rastgele veri üreterek destek vektör makineleriyle elde edilen doğrula lojistik regresyondan elde edilen doğrulu grafiksel biçimde göstermeye çalışalım. Aşağıda böyle bir kod görüyorsunuz:

```
import numpy as np

from sklearn.datasets.samples_generator import make_blobs

dataset_x, dataset_y = make_blobs(n_samples=100, n_features=2, centers=2, cluster_std=0.8)

from sklearn.svm import SVC

svc = SVC(kernel='linear')
svc.fit(dataset_x, dataset_y)

from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
lr.fit(dataset_x, dataset_y)

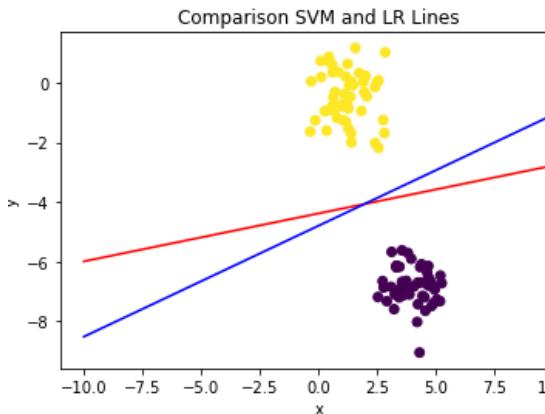
import matplotlib.pyplot as plt
```

```

plt.title('Comparison SVM and LR Lines')
plt.xlabel('x')
plt.ylabel('y')

plt.scatter(dataset_x[:, 0], dataset_x[:, 1], c=dataset_y)
x = np.linspace(-10, 10, 100)
y = (-svc.coef_[0][0] * x - svc.intercept_[0]) / svc.coef_[0][1]
plt.plot(x, y, c='red')
y = (-lr.coef_[0][0] * x - lr.intercept_[0]) / lr.coef_[0][1]
plt.plot(x, y, c='blue')

```



Burada kırmızı destek vektör makineleri ile mavi ise lojistik regresyon ile elde edilmiş olan doğrulardır.

Pekiyi eğer veri kümesi doğrusal olarak ayırtılabilir bir küme ise lojistik regresyon mu yoksa destek vektör makineleri mi daha iyi sonuç bermektedir? Bu karşılaştırma sonraki başlık altında değerlendirilecektir.

Pekiyi veri kümesi doğrusal olarak ayırtılabilir değilse ne yapmalıyız? İşte lojistik regresyonda yapılacak bir şey yoktur. Bu durumda yapay sinir ağları ya da destek vektör makineleri kullanılabilir. Yukarıda da belirtildiği gibi destek vektör makinelerinde kernel değiştirilerek boyut yükselmesi ile doğrusal olarak ayırtılabilir olmayan sınıflandırma problemleri de çözülebilmiştir. Bunun için destek vektör makineelerinde birkaç hazır kernel bulunmaktadır. Şimdi doğrusal olarak ayırtılabilir olmayan dairesel bir veri kümesi için "radial kernel" kullanımını örnek olarak verelim. Bunun için önce dairesel iki sınıfı rasgele noktalar oluşturalım. Bu işlem `sklearn.datasets` modülü içerisindeki `make_circles` fonksiyonuyla yapılabilmektedir. (Animasyonu gibi `make_blobs` fonksiyonu doğrusal olarak ayırtılabilir tarzda rastgele veriler üretiyor.)

```

from sklearn.datasets import make_circles

dataset_x, dataset_y = make_circles(n_samples=100)

from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.3)

import matplotlib.pyplot as plt

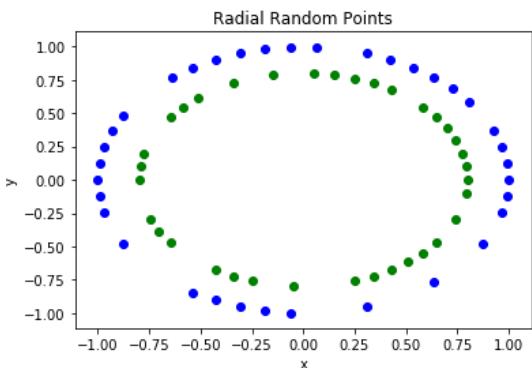
plt.scatter(training_dataset_x[training_dataset_y == 0, 0],
            training_dataset_x[training_dataset_y == 0, 1], color='blue')

plt.title('Radial Random Points')
plt.xlabel('x')
plt.ylabel('y')
plt.scatter(training_dataset_x[training_dataset_y == 1, 0],
            training_dataset_x[training_dataset_y == 1, 1], color='green')

plt.pause(1)

```

Bu işlemler sonucunda aşağıdaki gibi rastgele 2 sınıfı dairesel noktalar elde ettik:



Göründüğü gibi bu iki sınıfı dairesel noktalar bir doğru ile ayırtılabilirler. Olsa olsa bu noktalar dairesel bir eğri ile ayırtılabilirler. İşte SVC sınıfındaki "radiak kernel (rbf)" bunu sağlamaktadır:

```
from sklearn.svm import SVC

svc = SVC(kernel='rbf')
svc.fit(training_dataset_x, training_dataset_y)

result = svc.predict(test_dataset_x)
ratio = sum(result == test_dataset_y) / len(test_dataset_y)
print('Başarı oranı: {}'.format(ratio))
print('Score: {}'.format(svc.score(training_dataset_x, training_dataset_y)))
```

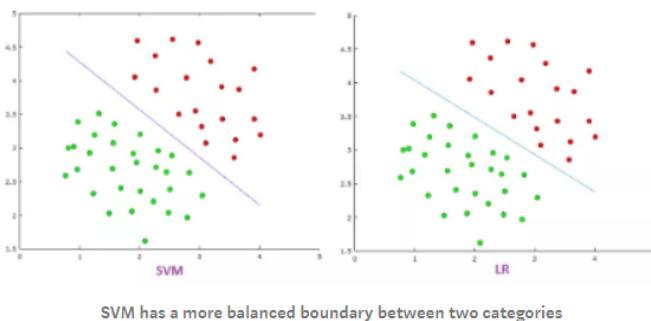
Buradan şu sonuç elde edilmiştir:

```
Başarı oranı: 1.0
Score: 1.0
```

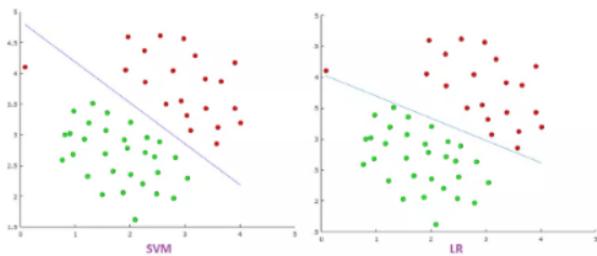
SVC sınıfındaki "poly" kernel ise sınıfları bir doğru ile ya da dairesel olarak değil polinomsal bir fonksiyonla ayırtırmaya çalışmaktadır.

Lojistik Regresyon, Destek Vektör Makineleri, Kümeleme ve Yapay Sinir Ağları Yöntemlerinin Karşılaştırması

Eğer veri kümesi doğrusal olarak ayırtılabilir değilse zaten lojistik regresyon kullanılamaz. Bu durumda destek vektör makineleri (kernel değiştirerek), kümeleme yöntemleri ya da yapay sinir ağları kullanılmalıdır. Eğer veri kümesi doğrusal olarak ayırtılabilir ise lojistik regresyon, destek vektör makineleri ve yapay sinir ağları yöntemlerinin her biri kullanılabilir. Genel olarak destek vektör makinelerinin daha adil (ortalayacak biçimde) bir sınır tespiti yaptığı söylenebilir.



Öte yandan lojistik regresyon üç değerler olması durumunda bu üç değerleri dikkate alacak biçimde daha iyi bir sınıflandırma yapabilmektedir:



LR's boundary change by including one new example. In reality this may not be a good idea so being not sensitive like SVM tends to be better.

Büyük veriler için lojistik regresyon destek vektör makinelerinden daha fazla işlem gerektirmektedir. Dolayısıyla daha yavaş olma eğilimindedir.

Yapay sinir ağları her türlü verilerde çalışabilen genel bir yöntem olmasına karşın eğitim konusunda sorunları olabilmektedir. Yapay sinir ağlarından performans elde edebilmek için belli miktarda veri ile eğitmek gerekmektedir. Halbuki lojistik regresyon ve destek vektör makinelerinde makul çözüm az sayıda veri ile elde edilebilmektedir.

Kümeleme yöntemlerinin denetimli (supervised) değil denetimsiz (unsupervised) yöntemler olduğunu anımsayınız. Dolayısıyla elimizde yalnızca dataset_x varsa fakat dataset_y yoksa mecburen kümeleme yöntemlerini kullanız.

DOĞRUSAL KARAR MODELLERİNİN ÇÖZÜMÜ

Doğrusal programlama "yöneylem araştırması (operational research)" denilen alanın en önemli konularından biridir. Belli doğrusal kısıtlar altında maksimizasyon ve minimizasyon işlemlerine yöneliktir. Doğrusal programlama 2. Dünya Savaşı sürecinde geliştirilmiştir. Makine öğrenmesi işlemlerinde belli adımlarda doğrusal programlama ile optimizasyonların yapılması gerekmektedir. Yani bu konu doğrudan makine öğrenmesi içeresine girmese de bir biçimde makine öğrenmesi konusuyla ilişkilidir. Doğrusal programlamadaki doğrusallık kullanılan değişkenlerin birinci derece olması ile ilgilidir. Eğer kısıtları ve amaç fonksiyonunu oluşturan değişkenler birinci dereceden değilse buna da "doğrusal olmayan programlama (nonlinear programming)" denilmektedir.

Doğrusal karar modellerinde kısıtlar (constraints) ve amaç fonksiyonu (objective function) vardır. Doğrusal programlama faaliyetinde kısıtları sağlayan amaç fonksiyonunun en büyük ya da en küçük değeri elde edilmeye çalışılmaktadır. Kısıtlar bir çözüm alanı oluşturur. Bu çözüm alanı içerisinde (kısıtlar sağlanacak biçimde) amaç fonksiyonu en büyütlenmeye ya da en küçütlenmeye çalışılmaktadır. Doğrusal karar modellerinin çözüm alanı konveks bir kümedir. Bu konveks kümede uç noktalar (extreme points) bulunur. Eniyi çözüm de bu uç noktalardan birindedir. Doğrusal karar modellerin çözümü için yalnızca uç noktaları dolaşan algoritmalar önerilmiştir. En etkin algoritmaların biri "Simplex" denilen algoritmadır. Simplex algoritmasının revize edilmiş biçimine de "Revised Simplex" denilmektedir. Tabii biz burada Simplex algoritmasının kendisini görmeyeceğiz. Yalnızca doğrusal karar modelinin oluşturulmasını ve çözümünde kullanılan hazır sınıfları ve fonksiyonları göreceğiz.

Doğrusal karar modelleri matematiksel olarak iki biçimde ifade edilebilmektedir. Bunlardan birine kanonik biçim diğerine standart biçim denilmektedir. Kanonik biçimde tüm kısıtlar \leq biçimine ya da \geq biçimine getirilir. Standart biçimde ise kısıtlar $=$ biçiminde bulunmaktadır. Kanonik biçimin genel hali şöyledir:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \dots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2n}x_n &\leq b_2 \\ \vdots & \\ a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 + \dots + a_{mn}x_n &\leq b_m \end{aligned}$$

$$Z_{\text{Max}} = c_1x_1 + c_2x_2 + \dots + c_nx_n$$

Aslında bu modeli matrisel biçimde aşağıdaki gibi de ifade edebiliriz:

$$AX \leq b$$

$$x \geq 0$$

$$Z_{\max} = CX$$

Standart form ise matrisel biçimde şöyle gösterilebilir:

$$AX = b$$

$$x \geq 0$$

$$Z_{\max} = CX$$

Doğrusal karar problemleri genellikle bize metinsel olarak gelir. Biz bu problemin matematiksel modelini yukarıda belirtildiği gibi kanonik ya da standart biçimde ifade ederiz. Sonra da bunu Python'ın hazır modülleriyle çözeriz. Örnek bir doğrusal karar modeli problemi şöyle olabilir:

Bir çiftçinin buğday, mısır ve arpa ürünlerinin ekimi için 300 hektarlık arazisi vardır. Çiftçi hektar başına buğdaydan 150 TL, mısırda 220 TL, arpadan da 180 TL kar beklemektedir. İşgücü yüzünden çiftçi buğday için 150 hektardan ve arpa için 120 hektardan daha fazla yer ayırmamalıdır. Verimlilik yönünden ise en az 80 hektar buğday için yer ayırmalı ve mısır ekimi için de toplam arazinin %30'undan daha fazla yer ayırmamalıdır. Çiftçi karını en büyüklemek istemektedir. Çiftçinin hangi ürün için ne kadar alan ayırması gereklidir?

Bu problemde x_1 buğday için ayrılan alan, x_2 mısır için ayrılan alan ve x_3 de arpa için ayrılan alan olmak üzere kısıtlar ve amaç fonksiyonu şöyle modellenebilir:

$$x_1 + x_2 + x_3 = 300$$

$$x_1 \leq 150$$

$$x_3 \leq 120$$

$$x_1 \geq 80$$

$$x_2 \leq 90$$

$$x_1, x_2, x_3 \geq 0$$

$$Z_{\max} = 150 x_1 + 220 x_2 + 180 x_3$$

Göründüğü gibi burada problem matematiksel terimlerle ifade edilmiştir. Artık biz bu modelden hareketle Python'daki hazır modülleri kullanarak problemi çözebilir. Diğer bir problem de şöyle olabilir:

Bir boyacı fabrikası hem iç hem dış boyacı üretiyor. Boyacı üretiminde A ve B olmak üzere iki tip hammadde kullanılıyor. Bir günde A hammaddesinden en çok 6 ton, B hammaddesinden en çok 8 ton kullanılabilir. Günlük hammadde ihtiyacı iç ve dış boyacı için ton olarak aşağıdaki tabloda verilmiştir.

Hammadde	Gereken Hammadde		Mevcut Miktar (ton)
	Miktarı (ton)		
	Dış Boyacı	İç Boyacı	
A	1	2	6
B	2	1	8

Bir pazar araştırması iç boyanın günlük isteminin, dış boyacı istemini 1 tondan fazla aşmadığını ve iç boyanın en büyük isteminin 2 ton olduğunu gösteriyor. İç boyanın toplam satış fiyatı bir tonda 2 br, dış boyanın toplam satış fiyatı bir tonda 3 br dir. Şirket toplam geliri en büyükleyecek biçimde kaç ton iç boyacı kaç ton dış boyacı üretimi yapması gerektiğini belirlemek istemektedir.

Bu problemde iki değişken vardır: İç boyacı ve dış boyacı miktarı. Bunları x_1 ve x_2 biçiminde temsil edebiliriz. Amaç fonksiyonu bir maksimizasyon biçimindedir. Kısıtlar şu biçimde ifade edilebilir:

$$\begin{aligned}
 X_1 + 2X_2 &\leq 6 \quad (\text{A hammadde kısıtı}) \\
 2X_1 + X_2 &\leq 8 \quad (\text{B hammadde kısıtı}) \\
 X_2 - X_1 &\leq 1 \quad (\text{İç boyanın günlük istemi dış boyayı 1 tondan fazla aşmayacak}) \\
 X_2 &\leq 2 \quad (\text{İç boyanın en büyük istemi 2 ton}) \\
 X_1, X_2 &\geq 0
 \end{aligned}$$

$$\max f(\mathbf{X}) = \max Z = 3X_1 + 2X_2$$

Diger bir örnek de şöyle olabilir:

Bir kişi sadece et, süt ve yumurta yiyerek diyet yapmaktadır. Bu kişinin günde en az 15 mg A vitamini, 30 mg C vitamini ve 10 mg D vitamini alması gerekmektedir. Buna karşılık besinlerle aldığı kolesterol 80 br/gün'ü geçmemelidir. 1 lt sütte 1 mg A, 100 mg C 10 mg D ve 70 br kolesterol vardır ve sütün litresi 800 TL dir. 1 kg ette 1 mg A, 10 mg C, 100 mg D vitamini ve 50 br kolesterol vardır. Etin kilosu 3700 TL dir. Yumurtanın düznesinde 10 mg A, 10 mg C ve 10 mg D vitamini ile 120 br kolesterol bulunmakta olup, yumurtanın düznesi 275 TL dir. Kişiin istediği, bu diyeti en ucuz yolla gerçekleştirmektir. Buna göre problemin DP modelini oluşturunuz.

Problemin matematiksel modeli şöyle oluşturulabilir:

- x1: Bir günde tüketilecek süt miktarı (litre)
- x2: Bir günde tüketilecek et miktarı (kg)
- x3: Bir günde tüketilecek yumurta miktarı (tane)

$$\min Z = 800x_1 + 3700x_2 + 275x_3$$

Kısıtlar şunlardır:

$$\begin{aligned}
 x_1 + x_2 + 10x_3 &\geq 15 && (\text{A vitamini kısıtı}) \\
 100x_1 + 10x_2 + 10x_3 &\geq 30 && (\text{C vitamini, kısıtı}) \\
 10x_1 + 100x_2 + 10x_3 &\geq 10 && (\text{D vitamini kısıtı}) \\
 70x_1 + 50x_2 + 120x_3 &\leq 80 \\
 x_1, x_2, x_3 &\geq 0
 \end{aligned}$$

Diger örnek şöyle olabilir:

Bir oyuncak imalatçısı model otomobil ve uçak üretimi yapmayı planlamaktadır. Şirket bu iki imalatını iki ayrı işlemin yapıldığı I ve II nolu atölyelerinde gerçekleştirmektedir. Çizelgede bir adet model otomobil ile model uçak imali için atölye işlem süreleri ve atölye kapasiteleri verilmiştir. Bir model otomobil satışından 45 TL, bir model uçak satışından ise 55 TL kar elde edilecektir. Maksimum kar için her bir üründen ne kadar imal edilmeli?

Atölyeler	Mallar		Kapasite (saat)
	Otomobil	Uçak	
	İşlem zamanı (saat/ad.)		
I	6	4	120
II	3	10	180

Örnek https://kemaisonmez.weebly.com/uploads/2/4/7/1/24712761/sm_ders_3_dogrusal_programlama_genel.pdf URL'sinden alınmıştır.

Problemin değişkenleri şöyle ifade edilebilir:

- x1: Model otomobil miktarı (adet)
- x2: Model uçak miktarı (adet olarak)

Problemin amaç fonksiyonu şöyledir:

$$\text{Max } Z = 45x_1 + 55x_2$$

Problemin kısıtları şöyledir:

$$\begin{aligned} 6x_1 + 4x_2 &\leq 120 \\ 3x_1 + 10x_2 &\leq 180 \\ x_1, x_2 &\geq 0 \end{aligned}$$

Doğrusal Programlama Modelinin SciPy linprog İle Çözülmesi

Doğrusal karar modellerinin çözümü için `scipy.optimize` modülü içerisindeki `linprog` sınıfı kullanılabilir. Bu sınıfın `__init__` metodunun parametrik yapısı şöyledir:

```
scipy.optimize.linprog(c, A_ub=None, b_ub=None, A_eq=None, b_eq=None, bounds=None, method='simpLex', callback=None, options=None)
```

Bu fonksiyonun kullanılabilmesi için amaç fonksiyonun minimizasyon biçiminde olması gereklidir. Eğer amaç fonksiyon makismızasysa bizim onu negatif değerlerle çarpıp minimizasyon haline getirmemiz gereklidir. Çünkü amaç fonksiyonunun en büyütlenmesi onun negatifinin en küçütlenmesi ile aynı anlamladır. Fonksiyondaki `A_ub` (upper bound) parametresi modeldeki " \leq " kısıtlarının A katsayı matrisini belirtir. Eğer kısıtlar içerisinde " \geq " olanlar varsa eşitsizliğin her iki tarafı negatif ile çarpılıp bu eşitsizlik " \leq " haline getirilmelidir. `A_eq` parametresi " $=$ " olan kısıtların A katsayı matrisini belirtmektedir. Fonksiyondaki `bounds` parametresi kısıtlardaki değişkenlerin alabileceği değer aralığını belirlemek için kullanılır. Bu parametre için argüman girilmezse default durum " ≥ 0 " biçimindedir. Eğer değişkenler özel bir kısıt içeriyoysa burada tüm değişkenler sırasıyla bir dolaşılabilir nesne içerisinde demetler biçiminde oluşturulmalıdır. Örneğin x_1 koşulunun " ≥ 10 " olduğunu, x_2 koşulunun da " ≥ -5 " ve " ≤ 20 " olduğunu varsayıyalım. Burada `bounds` parametresi şöyle düzenlenmelidir.

```
bounds = [(10, None), (-5, 20)]
```

Buradaki `None` değeri minimal için $-\infty$, Maksimum için $+\infty$ anlamına gelmektedir.

`linprog` fonksiyonunun geri dönüş değeri bir sınıf türündendir. Bu sınıfın `__str__` ve `__repr__` metotları karar modelini sonucunu bize yazı olarak verir. Fakat bu sonuçları biz de şu örnek özniteliklerden ayrı ayrı nümerik biçiminde alabiliriz:

fun: Amaç fonksiyonun değeri (Eğer amaç fonksiyonu maksimizasyonsa bu değerin negatif dikkate alınmalıdır.)
x: Karar değişkenlerinin optimum değerleri
success: Optimal çözüm var mı, yok mu? bool bir değer
nit: Optimal çözüm için gereken iterasyon sayısı

Şimdi bir örnek yapalım:

$$Z_{\max} = 3x + 2y$$

Kısıtlar:

$$\begin{aligned} 2x + y &\leq 18 \\ 2x + 3y &\leq 42 \\ 3x + y &\leq 24 \\ x, y &\geq 0 \end{aligned}$$

Çözüm şöyle yapılabilir:

```
import numpy as np
from scipy.optimize import linprog

c = np.array([-3, -2], dtype=np.float32)
```

```

aub = np.array([[2, 1], [2, 3], [3, 1]], dtype=np.float32)
bub = np.array([18, 42, 24])
lp = linprog(c, A_ub=aub, b_ub=bub)

print(lp)
print('-----')
print('Variables: {}'.format(lp.x))
print('Max Value: {}'.format(-lp.fun))

```

Sonuç şöyle lede edilmiştir:

```

con: array([], dtype=float64)
fun: -32.9999980043574
message: 'Optimization terminated successfully.'
nit: 4
slack: array([9.60302629e-08, 3.18105720e-07, 3.0000009e+00])
status: 0
success: True
x: array([ 3.0000001, 11.9999989])
-----
Variables: [ 3.0000001 11.9999989]
Max Value: 32.9999980043574

```

Şimdi daha önce verdiğimiz atölye problemini çözelim:

```

import numpy as np
from scipy.optimize import linprog

c = np.array([-3, -2], dtype=np.float32)
aub = np.array([[2, 1], [2, 3], [3, 1]], dtype=np.float32)
bub = np.array([18, 42, 24])
lp = linprog(c, A_ub=aub, b_ub=bub)

print('variables: {}'.format(lp.x))
print('Max Value: {}'.format(-lp.fun))

con: array([], dtype=float64)
fun: -1274.999989391895
message: 'Optimization terminated successfully.'
nit: 4
slack: array([9.93566687e-08, 1.50681728e-07])
status: 0
success: True
x: array([ 9.9999999, 14.9999999])
-----
Variables: [ 9.9999999 14.9999999]
Max Value: 1274.999989391895

```

Doğrusal Programlama Modeli İçin Pulp Kütüphanesinin Kullanımı

Pulp kütüphanesi linprog fonksiyonundan daha detaylıdır. Ancak gösterim daha doğal bir formattadır. Ancak bu kütüphane default olarak Anaconda içerisinde yoktur. Kütüphane aşağıdakilerden biriyle kurulabilir:

```

pip install pulp
python -m pip install pulp

```

Anaconda için conda programıyla kurulum da şöyle yapılabilir:

```

conda install -c conda-forge pulp

```

Kurulum sırasında hangi python sürümünün pip ya da python programının çalıştığını dikkat ediniz.

Pulp kütüphanesini dokümantasyonuna aşağıdaki adresten erişebilirsiniz:

<https://coin-or.github.io/pulp/>

Pulp kütüphanesinin kullanımı şöyledir:

1) Önce pulp modülündeki LpProblem sınıfı türünden bir nesne yaratılır. Bu nesne yaratılırken problemin ismi ve maksimizasyon mu, minimizasyon mu olduğu belirtilir.

```
pulp.LpProblem(name='NoName', sense=1)
```

Örneğin:

```
import pulp  
lp = pulp.LpProblem('My Problem', pulp.LpMaximize)
```

2) Bundan sonra değişkenleri oluşturmak gereklidir. Değişkenler pulp.LpVariable isimli bir sınıfta temsil edilmişlerdir. Değişkenler oluşturulurken onlara isimler ve sınır değerler verilebilir. Burada lowBound ve upBound parametreleri default olarak None değerini almıştır. Bu None değeri lowBound için $-\infty$, upBound için $+\infty$ biçimindedir.

```
pulp.LpVariable(name, lowBound=None, upBound=None, cat='Continuous', e=None)
```

Örneğin:

```
import pulp  
lp = pulp.LpProblem('My Problem', pulp.LpMaximize)  
  
x = pulp.LpVariable("x", lowBound=0)  
y = pulp.LpVariable("y", lowBound=0)
```

3) Şimdi sıra amaç fonksiyonunu ve kısıtları oluşturmaya gelmiştir. Amaç fonksiyonu LpProblem sınıfının `+=` operatör metoduyla oluşturulmaktadır. Örneğin:

```
lp += 3 * x + 2 * y
```

Kısıtlar da tamamen aynı yöntemle oluşturulmaktadır. Ancak kısıtlarda "`<=`", "`>=`" ya da "`==`" operatörleri de kullanılmalıdır.

```
lp += 3 * x + 2 * y  
lp += 2 * x + y <= 18  
lp += 2 * x + 3 * y <= 42  
lp += 3 * x + y <= 24
```

Sınıfın `__str__` ve `__repr__` metodları modelin matematiksel temsilini yazı olarak bize vermektedir.

4) Bundan sonra sıra modeli çözmeye gelmiştir. Çözme işlemi LpProblem sınıfının `solve` isimli metoduyla yapılmaktadır.

```
solve(solver=None, **kwargs)
```

Programcı isterse çözüm algoritmasını ya da çözüm yazılımını değiştirebilmek için solver parametresini kullanabilir.

LpProblem sınıfının `objective`, `constraints`, ve `status` örnek öznitelikleri bize modele ilişkin bilgileri vermektedir.

5) Çözüm sonucunda değişkenlerin optimal değerlerini alabilmek için `pulp.value` fonksiyonu kullanılmaktadır. Örneğin:

```

print('x = {}'.format(pulp.value(x)))
print('y = {}'.format(pulp.value(y)))

```

Amaç fonksiyonunun değeri de yine pulp.value metoduyla constraints özniteliği verilerek elde edilebilir. Örneğin:

```
print('Objective: {}'.format(pulp.value(lp.objective)))
```

Çözümün tüm kodları şöyledir:

```

import pulp

lp = pulp.LpProblem('MyProblem', pulp.LpMaximize)
x = pulp.LpVariable('x', lowBound=0)
y = pulp.LpVariable('y', lowBound=0)

lp += 3 * x + 2 * y

lp += 2 * x + y <= 18
lp += 2 * x + 3 * y <= 42
lp += 3 * x + y <= 24

print(lp)

lp.solve()
print('x = {}'.format(pulp.value(x)))
print('y = {}'.format(pulp.value(y)))
print('Objective: {}'.format(pulp.value(lp.objective)))

```

LpProblem sınıfının writeLP metodu modeli bir text dosyanın içerisinde yazar. Ancak maalesef pulp modülünde modülü okuyan bir fonksiyon yoktur.

Şimdi son olarak daha çok değişkenli bir doğrusal programlama modelini metinden hareketle çözmeye çalışalım:

Bir şehirde her biri 1200 kişilik öğrenci kapasitesi olan üç ayrı lise bulunmaktadır. Şehir yönetimi şehri Kuzey, Güney, Doğu, Batı ve Merkez olarak beş ayrı bölgeye ayırmıştır. Bazı öğrencilerin dışındaki okullara gitmek durumundadır. Bu durumda okul yönetimleri öğrencilerin alacağı toplam mesafeyi en az indiröök istemektedir. Gerekli olan doğrusal modeli kurunuz. Okullar bve mesafeleri aşağıdaki tabloda verilmiştir:

Bölgeler	Merkez Lisesi	Batı Lisesi	Güney Lisesi	Öğrenci Sayısı
Kuzey	8	11	14	700
Güney	12	9	0	300
Doğu	9	16	10	900
Batı	8	0	9	600
Merkez	0	8	12	500

Bu problemde bölgelerden liselere kaç kişinin gideceği karar değişkenlerini oluşturmaktadır. X_{ij} karar değişkeni j 'inci bölgeden i 'inci okula kaç kişinin gideceğini belirtmektedir. Örneğin X_{13} karar değişkeni Doğu bölgesinden Merkez liseye kaç kişinin gideceğini belirtmektedir. Model şöyle kurulabilir:

$$\text{Min } Z = 8X_{11} + 12X_{12} + 9X_{13} + 8X_{14} + 0 * X_{15} + 11X_{21} + 9X_{22} + 16X_{23} + 0X_{24} + 8X_{25} + 14X_{31} + 0X_{32} + 10X_{33} + 9X_{34} + 12X_{35}$$

Kısıtlar:

$$X_{11} + X_{21} + X_{31} = 700 \quad (\text{Kuzey bölgesindeki toplam öğrenci kısıtı})$$

$$X_{12} + X_{22} + X_{32} = 300 \quad (\text{Güney bölgesindeki toplam öğrenci kısıtı})$$

$$X_{13} + X_{23} + X_{33} = 900 \quad (\text{Doğu bölgesindeki toplam öğrenci kısıtı})$$

$X_{14} + X_{24} + X_{34} = 600$ (Batı bölgesindeki toplam öğrenci kısıtı)
 $X_{15} + X_{25} + X_{35} = 500$ (Merkez bölgesindeki toplam öğrenci kısıtı)

$X_{11} + X_{12} + X_{13} + X_{14} + X_{15} \leq 1200$ (Merkez lisesinin kapasite kısıtı)
 $X_{21} + X_{22} + X_{23} + X_{24} + X_{25} \leq 1200$ (Batı lisesinin kapasite kısıtı)
 $X_{31} + X_{32} + X_{33} + X_{34} + X_{35} \leq 1200$ (Güney lisesinin kapasite kısıtı)

$X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, X_{31}, X_{32}, X_{33}, X_{34}, X_{35} \geq 0$

$x_{ik} \rightarrow i^{\text{nci}} \text{ okuldan } k^{\text{inci}} \text{ bölgeye gidecek öğrenci sayısı}$

Problemi önce linprog ile çözelim:

```
import numpy as np
from scipy.optimize import linprog

c = np.array([8, 12, 9, 8, 0, 11, 9, 16, 0, 8, 14, 0, 10, 9, 12])
A_ub = np.array([[1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                 [0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
                 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]])
b_ub = np.array([1200, 1200, 1200])

A_eq = np.array([[1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
                 [0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                 [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0],
                 [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0],
                 [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0]])
b_eq = np.array([700, 300, 900, 600, 500])

result = linprog(c=c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq)
print(result)
```

Elde edilen sonuç şöyledir:

```
fun: 14600.0
message: 'Optimization terminated successfully.'
    nit: 14
    slack: array([ 0., 600., 0.])
status: 0
success: True
    x: array([700., 0., 0., 0., 500., 0., 0., 0., 600., 0., 0.,
       300., 900., 0., 0.])
```

Şimdi aynı problemi pulp ile çözelim:

```
import pulp

lp = pulp.LpProblem('District_School_Problem', pulp.LpMinimize)

x11 = pulp.LpVariable("x11", lowBound=0)
x12 = pulp.LpVariable("x12", lowBound=0)
x13 = pulp.LpVariable("x13", lowBound=0)
x14 = pulp.LpVariable("x14", lowBound=0)
x15 = pulp.LpVariable("x15", lowBound=0)

x21 = pulp.LpVariable("x21", lowBound=0)
x22 = pulp.LpVariable("x22", lowBound=0)
x23 = pulp.LpVariable("x23", lowBound=0)
x24 = pulp.LpVariable("x24", lowBound=0)
x25 = pulp.LpVariable("x25", lowBound=0)
```

```

x31 = pulp.LpVariable("x31", lowBound=0)
x32 = pulp.LpVariable("x32", lowBound=0)
x33 = pulp.LpVariable("x33", lowBound=0)
x34 = pulp.LpVariable("x34", lowBound=0)
x35 = pulp.LpVariable("x35", lowBound=0)

variables = [x11, x12, x13, x14, x15, x21, x22, x23, x24, x25, x31, x32, x33, x34, x35]

lp += 8 * x11 + 12 * x12 + 9 * x13 + 8 * x14 + 0 * x15 + 11 * x21 + 9 * x22 + 16 * x23 + 0 *
x24 + 8 * x25 + 14 * x31 + 0 * x32 + 10 * x33 + 9 * x34 + 12 * x35

lp += x11 + x12 + x13 + x14 + x15 <= 1200
lp += x21 + x22 + x23 + x24 + x25 <= 1200
lp += x31 + x32 + x33 + x34 + x35 <= 1200

lp += x11 + x21 + x31 == 700
lp += x12 + x22 + x32 == 300
lp += x13 + x23 + x33 == 900
lp += x14 + x24 + x34 == 600
lp += x15 + x25 + x35 == 500

lp.solve()

for variable in variables:
    print('{} = {}'.format(variable, pulp.value(variable)))

```

Elde edilen sonuçlar şöyledir:

```

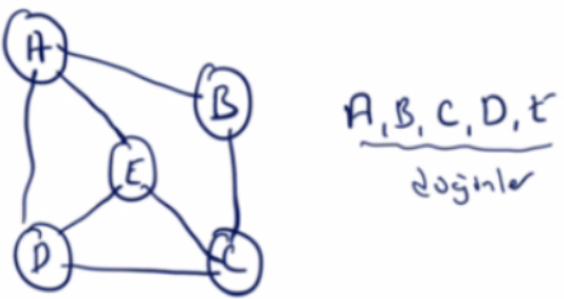
x11 = 700.0
x12 = 0.0
x13 = 0.0
x14 = 0.0
x15 = 500.0
x21 = 0.0
x22 = 0.0
x23 = 0.0
x24 = 600.0
x25 = 0.0
x31 = 0.0
x32 = 300.0
x33 = 900.0
x34 = 0.0
x35 = 0.0

```

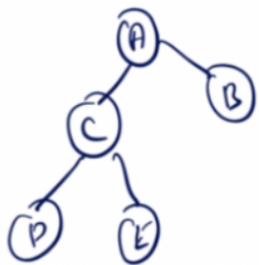
GRAFLAR ÜZERİNDE İŞLEMLER

Graflar optimizasyon uygulamalarında en çok karşılaşılan veri yapılarından biridir. Biz bu bölümde graf veri yapısının Python'da nasıl kullanılacağı ve bu veri yapısıyla temel graf optimizasyon problemlerinin nasıl çözüleceği üzerinde duracağız.

Graflar düğümlerden(nodes) ve kenarlardan (edges) oluşmaktadır. Genellikle şekilsel olarak düğümler yuvarlaklarla, kenarlar da çizgilerle gösterilirler. Örneğin:

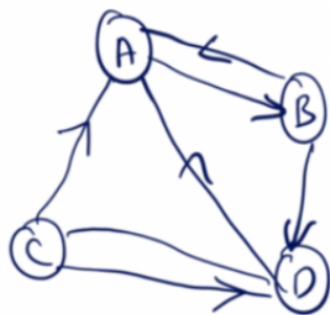


Veri yapılarında bir düğüme birden fazla yolla ulaşılabilirse buna graf ancak bir düğüme tek bir yolla ulaşılıyorsa buna da ağaç (tree) denilmektedir. Ağaçlar grafların özel bir durumudur. Aşağıda bir ağaç veri yapısı görülmektedir:



Graflarda düğümlere ve kenarlara bilgiler ilişirilebilir. Örneğin Mecidiyeköy'den Beşiktaş'a gidilebilecek en kısa yolu bulmak isteyelim. Bu bir graf problemidir. Yani problemin öncelikle bir graf olarak modellenmesi gereklidir. Bu modelleme yapılırken kavşak noktaları düğüm olarak alınır. Düğümler arasındaki tüm yollar kenarlarla grafta belirtilir. Sonra da graf üzerinde en kısa yol problemi uygulanır.

Graflar yönsüz (undirected) ve yönlü (directed) olmak üzere ikiye ayrılmaktadır. Yönsüz graflar aslında çift yönlü graflar biçiminde değerlendirilebilir. Yönsüz graflarda A ve B düğümü arasında bir kenar varsa bu durum hem A'dan B'ye hem de B'den A'ya gidiş olduğu anlamına gelir. Yönlü graflarda ise A'dan B'ye gidiş olması B'den A'ya gidiş olacağının anlamına gelmemektedir. Yönlü graflar genellikle kenarlarda oklarla belirtilirler. Örneğin:



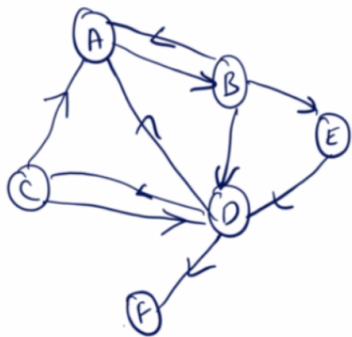
Örneğin karayollarını temsil eden graflar yönlüdür. Fakat sosyal ağlardaki arkadaşlık grafları yosuzdır.

Yapay zeka uygulamaları bazen graf veri yapısı ilgili olabilmektedir. Örneğin sosyal ağlar graf veri yapısı ile temsil edilip bunun üzerinde yapay zeka algoritmaları uygulanabilemektedir.

Graf veri yapıları tipik olarak iki yöntemle gerçekleştirilmektedir: Komşuluk Matrisi (Adjacency Matrix) ve Komşuluk Listeleri (Adjacency Lists). Komşuluk matrisi yöntemi daha seyrek kullanılmaktadır. Bu yöntemde hangi düğümden hangi düğüme yol olduğu bir matrisle belirlenir. Örneğin yukarıdaki graf için komşuluk matrisi şöyle oluşturulabilir:

	A	B	C	D
A	1	1	0	0
B	1	1	0	1
C	1	0	1	1
D	1	0	1	1

Komşuluk listesi yöntemi en sık kullanılan yöntemdir. Bu yöntemde her düğümün hangi düğümlerle bağlı olduğu bir listede tutulur. Tabii bağlı düğümlerin sayısı fazlaysa liste yerine aramayı hızlandırmak için sözlük tarzı bir veri yapısı tercih edilmektedir. Komşuluk listesi yöntemi Python'da sözlük nesneleriyle basit biçimde oluşturulabilir. Örneğin aşağıdaki yönlü grafi komşuluk listesi yöntemiyle Python'da oluşturmak isteyelim:



```
graph = {A: {B}, B: {A, D, E}, C: {A, D}, D: {A, F, C}, E: {D}, F: {}}
```

Fakat graflarda düğümlere ve kenarlara da bilgiler ilişirilebilmektedir. Peki bu durumda nasıl bir genelleştirme yapılabilir? Örneğin Networkx kütüphanesi grafi aşağıdaki gibi temsil etmiştir:

```

graph = {
    "nodes": {
        "A": {"name": "Ankara", "size": 1300}, 
        "B": {"name": "Istanbul", "size": 2100}, 
        "C": {"name": "Izmir", "size": 1300}
    },
    "edges": [
        {"A": "B", {"length": 1200}}, 
        {"A": "C", {"length": 1300}}
    ]
}
  
```

Göründüğü gibi graf düğümlerden ve kenarlardan oluşmaktadır. Düğümler bir sözlük içerisindeydir. Ama sözlüğün anahtarına karşılık gelen değer de sözlüktür. Çünkü bir düğüme atanacak birden fazla özellik olabilir. Benzer biçimde kenarlar da birer sözlük biçimindedir. Her kenara ilişirilecek bilgi de bir sözlükle temsil edilmiştir. Bu bilgiler de birden fazla olabilirler.

Graf veri yapısı için Python'ın standart kütüphanelerinde bir modül yoktur. Ancak graf işlemleri için üçüncü parti çeşitli kütüphaneler bulunmaktadır. Bunların arasından en çok tercih edilenlerden ikisi Networkx ve igraph kütüphaneleridir. Biz buarada Networkx kütüphanesini ele alacağız.

Anaconda sürümünde Networkx default olarak yüklenmiş biçimde gelmektedir. Diğer sürümlerde kütüphane manuel biçimde pip yoluyla yüklenebilir. Biz kütüphaneyi nx ismiyle import ederek kullanacağız. Örneğin:

```
import networkx as nx
```

Kütüphanenin dokümantasyonuna şuradan erişilebilir:

Networkx Kütüphanesinin Kullanımı

Kütüphanenin tipik kullanımı şöyledir:

1) Kütüphanede toplam dört farklı graph veri yapısı bulunmaktadır. Graph isimli sınıf yönsüz (undirected) graflar için, DiGraph isimli sınıf yönlü graflar için, MultiGraph yönsüz paralel kenarlara izin veren graflar için, MultiDiGraph yönlü paralel kenarlara izin veren graflar için kullanılmaktadır. Tipik olarak önce bu sınıflar türünden bir graph nesnesi oluşturulur. Graph ve DiGraph sınıfının `__init__` metodlarının parametrik yapıları şöyledir:

```
Graph(incoming_graph_data=None, **attr)  
Graph(incoming_graph_data=None, **attr)
```

Bu fonksiyonla `**` parametrisini kullanarak grafa istediğimiz bilgileri iliştirebiliriz. Bu bilgiler grafi temsil eden herhangi şeyler olabilir. `incoming_graph_data` grafi yaratırken hemen düğüm ve kenar girmek için kullanılmaktadır. Bu girişin bazı biçimleri vardır. Bunun için dokümanlara başvurabilirsiniz. Örneğin:

```
import networkx as nx  
  
graph = nx.Graph()
```

Burada biz boş bir graph yattık. Tabii istersek graph nesnesine kendi istediğimiz birtakım bilgileri `**` parametresine karşı gelecek biçimde isimli olarak girebiliriz. Örneğin:

```
import networkx as nx  
  
graph = nx.Graph(name='MyGraph', date='19/07/2020')
```

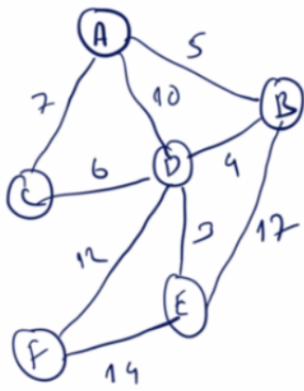
Graf kenarlar verilerek dolu bir biçimde de yaratılabilir:

```
import networkx as nx  
  
edges = [('A', 'B'), ('A', 'C'), ('C', 'B')]  
graph = nx.Graph(edges)
```

2) Yaratılmış grafa biz `add_edge` metoduna kenar, `add_node` metoduyla düğüm ekleyebiliriz. Önce düğüm ekleyip sonra kenar eklemek zorunlu değildir. Bir kenar eklenirken kenara konu olan düğümler yoksa zaten bu düğümler de garafa eklenmektedir.

```
add_edge(u_of_edge, v_of_edge, **attr)
```

Fonksiyonun ilk iki parametresi başlangıç ve bitiş düğümlerini belirtir. Düğümler hash'lenebilir (hashable) herhangi bir türden oabilir. (Örneğin, int, float, str gibi.) `**` parametresi bizim ilgili kenara istediğimiz gibi atayabileceğimiz özelliklerini belirtmektedir. Örneğin aşağıdaki grafın yollarını `add_edge` ile oluşturmaya çalışalım:



Buradaki sayılar ilgili yolların uzunluğu olsun. Networkx graf veri yapısı şöyle oluşturulabilir:

```
import networkx as nx

g = nx.Graph(title='Road Graph')

g.add_edge('A', 'B', length=5)
g.add_edge('A', 'C', lenght=7)
g.add_edge('A', 'D', lenght=10)
g.add_edge('D', 'B', lenght=4)
g.add_edge('D', 'C', lenght=6)
g.add_edge('D', 'F', lenght=12)
g.add_edge('D', 'E', lenght=3)
g.add_edge('B', 'E', lenght=17)
g.add_edge('F', 'E', lenght=10)
```

Göründüğü gibi düğüm eklenmesine hiç gerek duyulmamıştır. `add_edge` metodunda belirtilen düğümler zaten varsa olan düğümler kullanılır. Eğer `add_edge` metodunda belirtilen düğümler henüz yoksa zaten `add_edge` tarafından yaratılmaktadır. Tabii bazen yalnızca düğüm de yaratmak isteyebiliriz. Yani hiç kenarı olmayan bir düğüm de söz konusu olabilir. Ya da önce düğümleri yaratıp onlara birtakım bilgiler ilişirip ondan sonra kenarları yaratabilirim. Düğüm yaratma işlemi `add_node` metoduyla yapılmaktadır:

```
add_node(node_for_adding, **attr)
```

Fonksiyonun birinci parametresi eklenecek düğümü belirtir. Yine bu düğüm hash'lenebilir herhangi bir nesne olabilir. Buradaki `**` parametresi düğüme ilişirilecek bilgileri belirtmektedir. Grafi `add_edge` ile oluştururken düğüme bilgi ilişiremediğimize dikkat ediniz. Eğer düğümlere bilgiler ilişirilmek isteniyorsa önce düğümler `add_node` ile eklenenmelidir. Tabii aslında bir düğüme yaratıldıktan sonra da bilgiler ilişirebiliriz.

Graf sınıflarının `number_of_nodes` ve `number_of_edges` isimli metodları graftaki düğüm ve kenar sayılarını bize verir. Örneğin:

```
print(g.number_of_edges())
print(g.number_of_nodes())
```

Graftaki tüm düğümleri `nodes` örnek özniteligi ile, tüm kenarları da `edges` örnek özniteligi ile alabiliriz. Bu sınıflar bize dolaşılabilir bir nesne verirler. Bu nesneler dolaşıldığında bizim düğümler için verdigimiz nesneler (yukarıdaki örnekte 'A', 'B' biçiminde harfler) kenarlar dolaşıldığında da iki düğümden oluşan demetler elde edilmektedir. Örneğin:

```
for node in g.nodes:
    print(node, '--->', type(node))

for edge in g.edges:
    print(edge, '--->', type(edge))
```

Kodun ekran çıktısı şöyle olacaktır:

```
A ---> <class 'str'>
B ---> <class 'str'>
C ---> <class 'str'>
D ---> <class 'str'>
F ---> <class 'str'>
E ---> <class 'str'>
('A', 'B') ---> <class 'tuple'>
('A', 'C') ---> <class 'tuple'>
('A', 'D') ---> <class 'tuple'>
('B', 'E') ---> <class 'tuple'>
('B', 'D') ---> <class 'tuple'>
('C', 'D') ---> <class 'tuple'>
('D', 'F') ---> <class 'tuple'>
('D', 'E') ---> <class 'tuple'>
('F', 'E') ---> <class 'tuple'>
```

Dolaşılabilir sınıfın `__str__` metodu bize düğümleri ve kenarları bir demet yazısı biçiminde vermektedir. O halde biz doğrudan nodes ve edges örnek özniteliklerini print fonksiyonuyla da yazdırabiliriz.

```
print(g.nodes)
print(g.edges)
```

Kodun çıktısı şöyle olacaktır:

```
['A', 'B', 'C', 'D', 'F', 'E']
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'E'), ('B', 'D'), ('C', 'D'), ('D', 'F'), ('D', 'E'), ('F', 'E')]
```

Graftan herhangi bir düğüm `remove_node` metodu ile silinebilir. Bir düğüm silindiğinde o düğüme bağlı olan kenarların hepsi de silinmektedir. Benzer biçimde bir kenar da `remove_edge` metoduyla silinebilmektedir. Ancak kenar silindiğinde kenara ilişkin düğümler silinmemektedir.

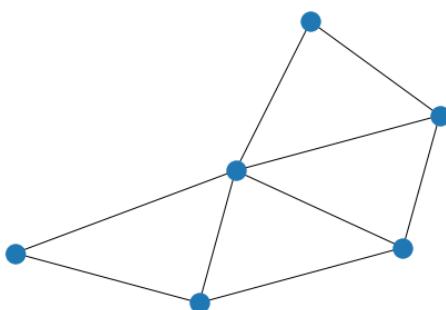
Graftaki belli bir düğümün komşularını (yani o düğüme bağlı kenarları) elde etmek için `[..]` operatörü kullanılabilir. Bu işlem bize sözlük tarzı (AtlasView isimli bir sınıf) dolaşılabilir bir nesne vermektedir. Örneğin:

```
for s in g['A']:
    print(s)
```

Burada A'nın komşuları yazı olarak elde edilmektedir. Eğer biz A ile B arasındaki kenar özelliklerini elde etmek istiyorsak `g['A']['B']` ifadesini kullanmalıyız.

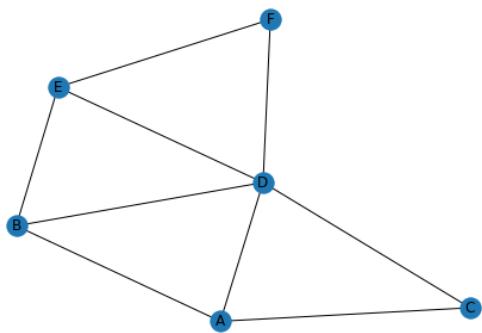
Oluşturulan grafın çizdirilmesi oldukça kolaydır. Tek yapılacak şey networkx modülündeki `draw` fonksiyonunu kullanmaktır. Örneğin:

```
nx.draw(g)
```



Düğümlerin üzerinde isimlerin yazılması isteniyorsa with_labels parametresi True geçilmelidir. Örneğin:

```
nx.draw(g, with_labels=True)
```

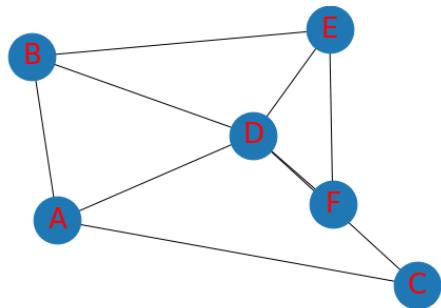


Fonksiyonun pek çok isimli parametresi vardır. Bu parametreler aşağıdaki link'ten incelenebilir:

https://networkx.github.io/documentation/stable/reference/generated/networkx.drawing.nx_pylab.draw_networkx.html?highlight=draw_networkx#networkx.drawing.nx_pylab.draw_networkx

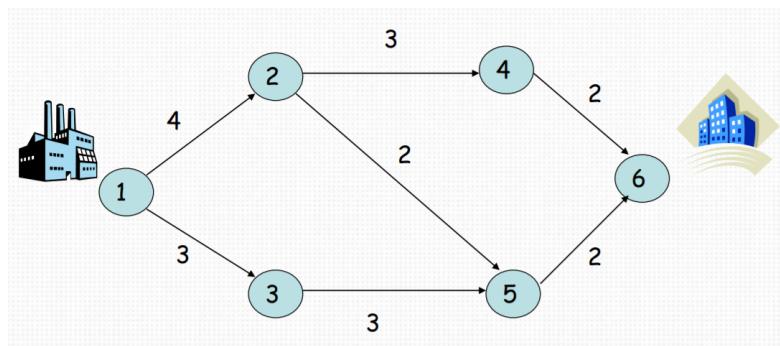
Örneğin:

```
nx.draw(g, with_labels=True, font_size=26, node_size=2000, font_color='red')
```



Tabii aslında graflar birtakım optimizasyon tarzı problemleri çözmek için kullanılmaktadır. Yani graflar aslında yalnızca veri yapısıdır. Önemli olan bu veri yapısı üzerinde yararlı işlemler yapabilen algoritmaların çalıştırılmasıdır. İşte networkx kütüphanesinde de graflar üzerinde işlemler yapabilen pek çok hazır fonksiyon vardır.

Yukarıda da belirtildiği gibi Graph sınıfı yönsüz (yani çift yönlü) graf oluşturmaktadır. Halbuki DiGraph sınıfı yönlü graf oluşturur. Örneğin aşağıdaki yönlü grafi oluşturmak isteyelim:



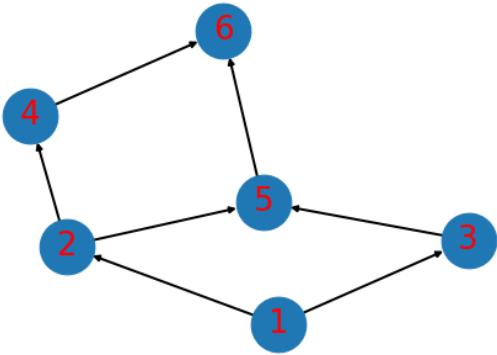
```
import networkx as nx  
  
g = nx.DiGraph(title='Road Graph')
```

```

g.add_edge(1, 2, length=4)
g.add_edge(1, 3, length=3)
g.add_edge(3, 5, length=3)
g.add_edge(5, 6, length=2)
g.add_edge(2, 4, length=3)
g.add_edge(2, 5, length=2)
g.add_edge(4, 6, length=2)

nx.draw(g, with_labels=True, font_size=24, font_color='white', node_size=1000)

```



Aslında graf bilgileri bazı uygulamalarda komşuluk matrisi biçiminde de ifade edilebilmektedir. Örneğin yukarıdaki grafın komşuluk matrisi şöyledir:

```

0 4 3 0 0 0
0 0 0 3 2 0
0 0 0 0 3 0
0 0 0 0 0 2
0 0 0 0 0 2
0 0 0 0 0 0

```

Bu komşuluk matrisinden yönlü graf elde eden fonksiyon basit biçimde şöyle yazılabilir:

```

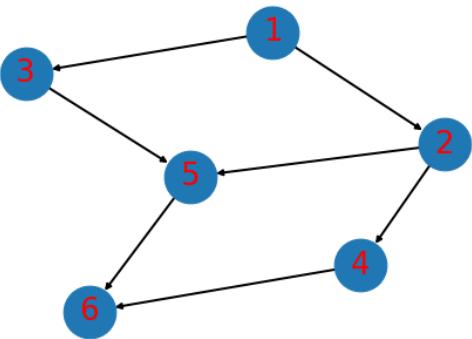
import numpy as np
import networkx as nx

def adjacency_matrix_to_graph(path):
    g = nx.DiGraph()
    data = np.loadtxt(path, dtype=np.float32)
    for i in range(len(data)):
        for k in range(len(data)):
            if data[i, k] != 0:
                g.add_edge(i + 1, k + 1, length=data[i, k])

    return g

g = adjacency_matrix_to_graph('graph.txt')
nx.draw(g, with_labels=True, font_size=26, node_size=2000, font_color='red', width=2)

```



Aslında bu işi yapan networkx kütüphanesinde `from_numpy_matrix` isimli bir fonksiyon da vardır. Örneğin:

```
data = np.loadtxt('graph.txt', dtype=np.float32)
g = nx.from_numpy_matrix(data)
nx.draw(g, with_labels=True, font_size=26, node_size=2000, font_color='red', width=2)
```

Tabii bunun tersini yapan `adjacency_matrix` isimli bir fonksiyon da vardır:

```
data2 = nx.adjacency_matrix(g)
print(data2)
print(data2.toarray())
```

Graflar üzerinde çeşitli optimizasyon işlemleri için `networkx.algorithms` modülündeki fonksiyonlar kullanılmaktadır. Networkx kütüphanesi bu bakımdan çok zengindir. Burada bazı graf problemleri örnek olarak verilecektir.

Graflar üzerindeki en klasik problem "en kısa yol (shortest path)" problemidir. Bu problemde bir düğümden bir düğüme gidebilmek için gereken en kısa yol bulunmaya çalışılır. En kısa yol problemi için `algorithms.shortest_paths.generic.shortest_path` fonksiyonu kullanılmaktadır. Bu fonksiyona biz başlangıç düğümünü, bitiş düğümünü ve neye göre uzunluk belirleneceğine belirten kenar özelliğini veriririz. O da bize en kısa yolu düğümlerde oluşan bir liste olarak verir. Örneğin:

```
import networkx as nx

g = nx.DiGraph(title='Road Graph')

g.add_edge(1, 2, length=4)
g.add_edge(1, 3, length=3)
g.add_edge(1, 2, length=4)
g.add_edge(2, 4, length=3)
g.add_edge(2, 5, length=2)
g.add_edge(3, 5, length=3)
g.add_edge(4, 6, length=2)
g.add_edge(5, 6, length=4)

print(g.number_of_edges())
print(g.number_of_nodes())

nx.draw(g, with_labels=True, font_size=26, node_size=2000, font_color='red', width=2)

from networkx.algorithms.shortest_paths.generic import shortest_path
path = shortest_path(g, 1, 6, weight='length')
print(path)
```

Programın çıktısı şöyle olacaktır:

[1, 3, 5, 6]

Graflar üzerinde turlama (cycling) işlemleri en çok kullanılan işlemleridir. Belli bir düğümden başlanarak tüm düğümlerin ya da kenarların elde edilmesi gerekebilir. Örneğin simple_cycles isimli fonksiyon bize graf içerisindeki kısa ya da uzun bütün turları vermektedir:

```
import networkx as nx

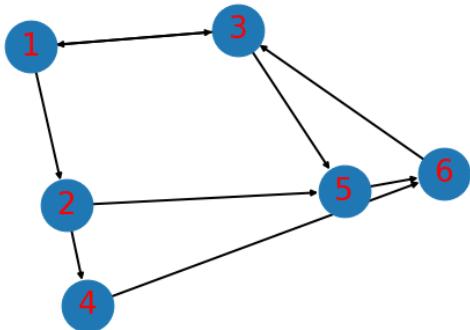
g = nx.DiGraph(title='Road Graph')

g.add_edge(1, 2, length=4)
g.add_edge(1, 3, length=3)
g.add_edge(1, 2, length=4)
g.add_edge(2, 4, length=3)
g.add_edge(2, 5, length=2)
g.add_edge(3, 5, length=3)
g.add_edge(4, 6, length=2)
g.add_edge(5, 6, length=4)
g.add_edge(6, 3, length=7)
g.add_edge(3, 1, length=3)

print(g.number_of_edges())
print(g.number_of_nodes())

nx.draw(g, with_labels=True, font_size=26, node_size=2000, font_color='red', width=2)

cycles = nx.simple_cycles(g)
print(list(cycles))
```



```
[[1, 3], [1, 2, 5, 6, 3], [1, 2, 4, 6, 3], [3, 5, 6]]
```

Yukarıda da belirtildiği gibi graflarda pek çok algoritmik problem vardır. networkx kütüphanesi bu problemlerin çoğunu çözecek fonksiyonlara sahiptir. Örneğin "en küçük örten ağaç (minimum spanning tree)" yönsüz bir graftaki bütün düğümleri içeren en kısa ağacın oluşturulma problemidir. Su boruları ya da elektrik telleri gibi alt yapı hizmetlerinde önemli kullanımı vardır. (Örneğin tüm evlere su götürecek en kısa uzunluklu boru hattının oluşturulması örneği gibi.) Bu algoritma networkx.algorithms.tree.mst modülündeki minimum_spanning_edges fonksiyonuyla çözülmektedir. Örneğin:

```
import networkx as nx

g = nx.Graph(title='Road Graph')

g.add_edge(1, 2, length=4)
g.add_edge(1, 3, length=3)
g.add_edge(1, 2, length=4)
g.add_edge(2, 4, length=3)
g.add_edge(2, 5, length=2)
g.add_edge(3, 5, length=3)
g.add_edge(4, 6, length=2)
g.add_edge(5, 6, length=4)
g.add_edge(6, 3, length=7)
```

```

print(g.number_of_edges())
print(g.number_of_nodes())

nx.draw(g, with_labels=True, font_size=26, node_size=2000, font_color='red', width=2)

from networkx.algorithms.tree.mst import minimum_spanning_edges
se = minimum_spanning_edges(g)
print(list(se))

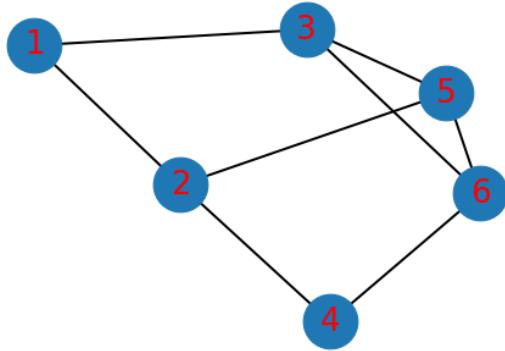
```

Programın çıktısı şöyle olacaktır:

```

8
6
[(1, 2, {'length': 4}), (1, 3, {'length': 3}), (2, 4, {'length': 3}), (2, 5,
{'length': 2}), (3, 6, {'length': 7})]

```



Diğer klasik problemlerden biri de "maksimum akış (maximum flow)" problemidir. Bu problemde belli bir düğümden belli bir düğüme maksimum kapasite aktarımı yapmak istenmektedir. Tipik durumda bu kapasite aktarımı borulardan suların aktarılması biçiminde düşünülebilir. Problemden amaç çeşitli yollar kullanarak maksimum akışın elde edilmesidir. Şüphesiz iki düğüm arasındaki akış o düğümlerdeki yollarda bulunan boruların en küçük kapasitelisi ile ilgilidir. Örnek bir problem şöyle verilebilir:

```

import networkx as nx

g = nx.DiGraph()
g.add_edge('x','a', capacity=3.0)
g.add_edge('x','b', capacity=1.0)
g.add_edge('a','c', capacity=3.0)
g.add_edge('b','c', capacity=5.0)
g.add_edge('b','d', capacity=4.0)
g.add_edge('d','e', capacity=2.0)
g.add_edge('c','y', capacity=2.0)
g.add_edge('e','y', capacity=3.0)

nx.draw(g, with_labels=True, font_size=26, node_size=2000, font_color='red', width=2)

from networkx.algorithms.flow import maximum_flow

flow_value, flow_dict = maximum_flow(g, 'x', 'y')
print(flow_value)
print(flow_dict)

```

Graf Çizme İçin Graphviz Kütüphanesinin Kullanımı

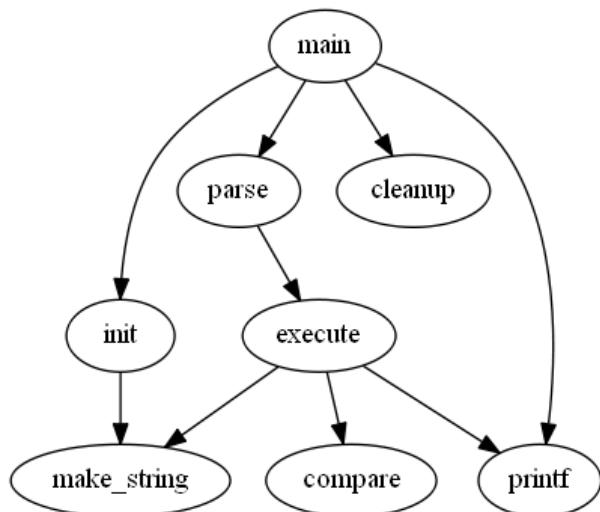
Graphviz aslında genel amaçlı bir graf çizim kütüphanesidir. Bu kütüphane komut satırından ya da pek çok dilden kullanılabilir. Tipik kullanımda kullanıcı uzantısı .dot ya da .gv olacak biçimde bir text dosyasına Graphviz dilini kullanarak grafını yazışal biçimde oluşturur. Sonra dot isimli programı çalıştırır. Bu program Graphviz kaynak

dosyasını okuyarak buradan hareketle bir çizim dosyası oluşturmaktadır. Örneğin aşağıdaki gibi bir sample.gv dosyası oluşturmuş olalım:

```
digraph G {  
    main -> parse -> execute;  
    main -> init;  
    main -> cleanup;  
    execute -> make_string;  
    execute -> printf  
    init -> make_string;  
    main -> printf;  
    execute -> compare;  
}
```

Aşağıdaki gibi bir komutla bir png dosyası elde edebiliriz:

```
dot -Tpng sample.gv > test.jpg
```



Graphviz kütüphanesinin dokümantasyonuna aşağıdaki bağlantıdan erişilebilir:

<https://graphviz.gitlab.io/documentation/>

Graphviz Kütüphanesi aslında pek çok dilden kullanılabilimektedir. Biz burada Python'daki kullanımı üzerinde duracağız. Kütüphanenin Python için genel dokümantasyonuna aşağıdaki bağlantıdan erişilebilir:

<https://graphviz.readthedocs.io/en/stable/manual.html>

Graphviz kütüphanesi Python'dan kullanımı tipik olarak şöyle kullanılmaktadır:

1) Önce graph ilgili sınıf kullanılarak (yönsüz için Graph, yönlü için DiGraph) yaratılır. Örneğin:

```
import graphviz as gv
```

```
dot = gv.Digraph('My Graph')
```

2) Graf için düğümler yaratılır. Bu işlem Graph sınıflarının node isimli metodlarıyla yapılmaktadır. Bu metodlar bizden düğümün ismini ve etiketini parametre olarak alırlar. Etiket verilmek zorunda değildir. Örneğin:

```
import graphviz as gv
```

```
dot = gv.Digraph('My Graph')
```

```
dot.node('A', 'Adana')
dot.node('E', 'Eskişehir')
dot.node('K', 'Kastamonu')
dot.node('O', 'Ordu')
dot.node('İ', 'İstanbul')
```

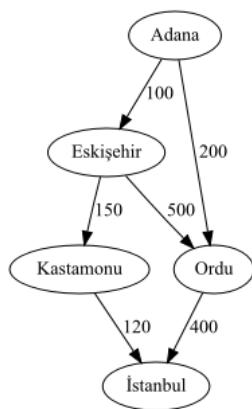
3) Bundan sonra kenarlar edge metoduyla eklenir. Örneğin:

```
dot.edge('A', 'E', label='100')
dot.edge('A', 'O', label='200')
dot.edge('E', 'K', label='150')
dot.edge('O', 'İ', label='400')
dot.edge('K', 'İ', label='120')
dot.edge('E', 'O', label='500')
```

Sınıfın source isimli örnek özniteliği bize çizilecek grafın script dosya içeriğini verir. (Yani biz bu sctipt'i bir dosyaya yapıp dot programıyla da görüntü dosyası oluşturabiliriz.) Komut satırında grafi çizdirmek için tek yapılacak şey nesne ismini yazarak ENTER tuşuna basmaktadır. Örneğin:

In [2]: dot

Out[2]:



Sınıfın render isimli metodu grafik dosyasını oluşturmakta kullanılabilir. render metoduyla default durumda bu grafik dosyası hem oluşturulmakta hem de çizdirilmektedir. Örneğin:

```
dot.render('test.gv', format='png', view=True)
```

Elimizde dot diliyye yazılmış bir script yazısı varsa biz graphviz içerisindeki Source fonksiyonuyla bir Graph ya da Digraph nesnesi elde edebiliriz. Örneğin:

```
g = gv.Source(text)
```

Aslında Source bir sınıfıtır. Bu sınıfın from_file metoduyla da biz doğrudan dosyanın yol ifadesini vererek graf nesnesini elde edebiliriz.

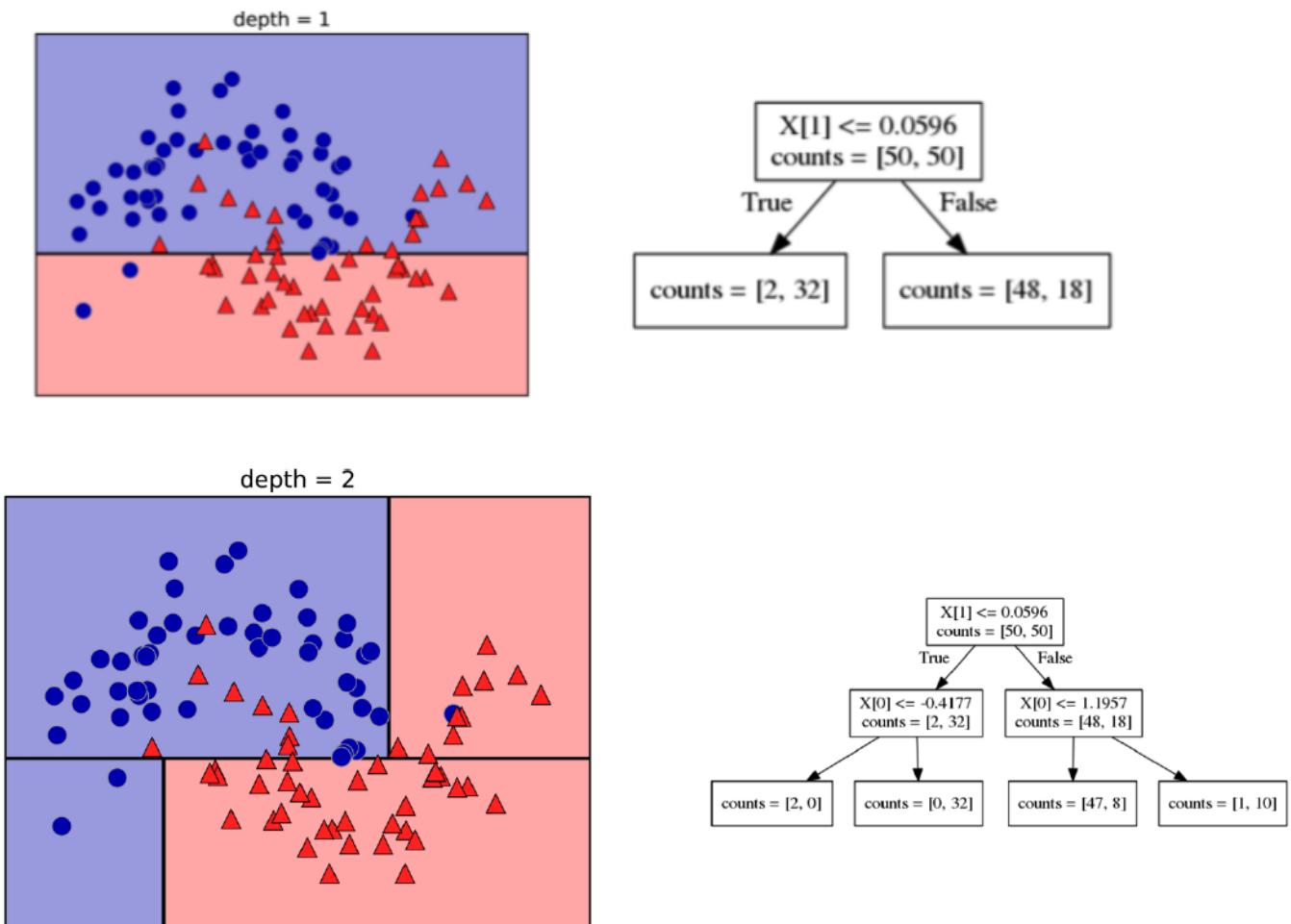
```
g = gv.Source.from_file(path)
```

Karar Ağaçları (Decision Trees)

Karar ağaçları sınıflandırma ve regresyon tarzı problemlerde kullanılan alternatif yöntemlerden biridir. Bu bakımdan bu yöntemin diğer alternatif yöntemlere göre bazı avantajları ve dezavantajları söz konusu olabilmektedir. Karar ağaçlarının çalışma mekanizması şöyle bir oyunla anlatılabilir: Oyunda oyunculardan biri aklından bir nesne tutar. Diğer Doğru-Yanlış yanıt beklenen sorular sorarak bu nesneyi bulmaya çalışır. Örnek sorular şöyle olabilir:

- Canlı mı cansız mı?
- Hafif mi ağır mı?
- Hacim olarak büyük mü küçük mü?
- Hareketli mi sabit mi?
-

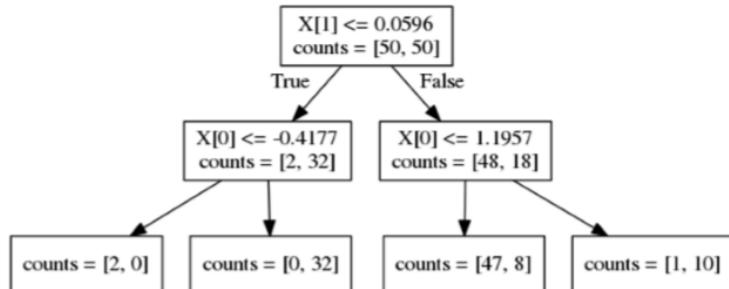
İşte bu soruların her birinde aslında biz gitgide alanı daraltarak sonuca yaklaşmış oluruz. Daha önce biz sınıflandırma problemlerini eğer doğrusal olarak ayırtılabilir (linearly separable) ise uygun doğruyla ayırtırmaya çalıştık. İşte karar ağaçları yöntemi de aslında yine doğrularla ayırtırmaya çalışır fakat bu yöntemde ayırtırma için tek bir doğru kullanılmamaktadır. Birden fazla doğruyla sınıflandırma yapılmaya çalışılır. Aşağıdaki şekilde iki sütunlu (x ve y) bir veri kümesinde sürekli bölme yöntemiyle uygun sınıfı elde etmenin grafiksel çizimi görülmektedir:



Cizimler ve karar ağıacı resimleri "Introduction To Machine Learning With Python" kitabından alınmıştır.

Burada görüldüğü gibi birinci soruda sınıflandırma iyi bir düzeyde yapılamamıştır. İkinci soruda sınıflandırma daha iyi bir düzeyde yapılmıştır. Burada iki sütunlu (x ve y) bir veri tablosu söz konusudur. Önce sütunlardan birine ilişkin (y sütunu) soru sorulmuş bu soruya göre ağaç ikiye ayrılmıştır. Sonra her iki alt ağaç için diğer bir soru (x sütunu) sorularak toplam dört bölge elde edilmiştir.

Yukarıdaki örnekte iki kademe bölme sonucunda elde edilen ağacı inceleyiniz:



Bu ağaçta kök düğümde sorulan soru $X[1] \leq 0.0596$ 'dır. Bu soruya göre bu koşula uyan ve uymayan satırlar elde edilmiştir. Ağaçtaki counts bilgisi ilgili bölgede kaç tane A sınıfından (yuvarlak mvi) kaç tane B sınıfından (üçgen kırmızı) nokta olduğunu belirtmektedir. Örneğin counts = [2, 32] o bölgede 2 tane A sınıfından 32 tane de B sınıfından nokta olduğunu anlatmaktadır. Bizim karar ağaçları yönteminde amacımız bölmelere devam etmek ve en sonunda homojen bölgeler elde etmektir. Bölgelerin homojen olması demek yalnızca tek gruptan nokta bulunması demektir. Yukarıdaki şekilde en soldaki yapraktaki counts değeri [2, 0] biçiminde olduğuna dikkat ediniz. Yani o bölgede 2 tane A sınıfından nokta vardır fakat hiç B sınıfından nokta yoktur. İşte bu homojen bir durumdur.

Karar ağaçları denetimli (supervised) bir öğrenme teknigini kullanmaktadır. Peki eğitimden sonra nasıl kestirim yapılacaktır? Kestirim oldukça kolaydır. Kestirilecek değerlere ağacın tepesinden itibaren sorular sorulur. Soruların yanıtlarına göre sola ya da sağa sapılarak bir yaprağa varılır. İşte o yaprağın durumu da bize sınıflandırmanın sonucunu verecektir.

Pekiyi karar ağaçları oluştururken hangi sorular hangi sırayla sorulacaktır? İşte bu noktada bilgi teorisi (information theory) konuları devreye girmektedir. Önce hangi, sorunun sorulması gereği ve bu sorunun ne olması gerekiği çeşitli matematsel yöntemlerle belirlenebilmektedir. Tabii konuda tek bir algoritmik yöntem yoktur. Shannon etropisi, Gini yönetimi en çok kullanılan yöntemlerdendir.

Scikit-Learn Karar Ağaçlarında Sınıflandırma

Scikit-Learn kütüphanesinde karar ağaçları için hazır sınıflar bulunmaktadır. DecisionTreeClassifier isimli sınıf karar ağaçlarıyla sınıflandırma için kullanılmaktadır. Bu sınıfın kullanımı aslında daha önce gördüğümüz sınıfların kullanımına çok benzerdir. Önce DecisionTreeClassifier türünden bir nesne yaratılır. Sonra fit işlemi uygulanır. Kestirim için yine sınıfın predict metodunu kullanılmaktadır. Sınıflandırmanın başarısı diğer sınıflarda olduğu gibi sınıfın score fonksiyonuyla elde edilmektedir.

Şimdi bu sınıfı daha önce çalıştığımız meme kanseri veri tablosu ile kullanalım.

```
from sklearn.datasets import load_breast_cancer

bc = load_breast_cancer()

print('Sütun isimleri: {}'.format(bc.feature_names))
print('Sınıf İsimleri: {}'.format(bc.target_names))

dataset_x = bc.data
dataset_y = bc.target

from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.2)

from sklearn.tree import DecisionTreeClassifier
```

```

dtc = DecisionTreeClassifier()
dtc.fit(training_dataset_x, training_dataset_y)
score = dtc.score(test_dataset_x, test_dataset_y)
print(score)

```

Burada default değerlerle `DecisionTreeClassifier` nesnesi yaratılmıştır. Daha sonra fit işlemi yapılmıştır. Ağacın oluşturulması fit işlemi sırasında gerçekleştirilmektedir. `score` metodu bir çeşit test metodudur. Biz de burada test veri kümemizle işlemin başarısını tespit etmiş olduk. `score` işlemi sırasında her test verisi ağaca sokularak ağaçtan bulunan değer gerçek sonuçla karşılaştırılmış ve sınıflama için bir başarı yüzdesi hesaplanmıştır. Yukarıdaki örnek için başarı yüzdesi %92-%96 civarındadır. Yine sınıfın `predict` metodu kestirim yapmaka kullanılmaktadır. Yani biz ağacı oluşturdukten sonra kestirimde bulunabiliriz.

```

import random

r = random.randint(0, len(test_dataset_x) - 1)
result = dtc.predict(test_dataset_x[r].reshape((1, -1)))

print(bc.target_names[result[0]])
print(test_dataset_y[r] == result[0])

```

`DecisionTreeClassifier` sınıfı ile oluşturulan ağacın graphiz yardımıyla grafi çizdirilebilir. Bunun için `sklearn.tree` modülündeki `export_graphviz` metodu kullanılmaktadır. Bu metot çizimi yapmaz fakat çizim için gereken `.dot` ya da `.gv` dosysını oluşturur. Örneğin:

```

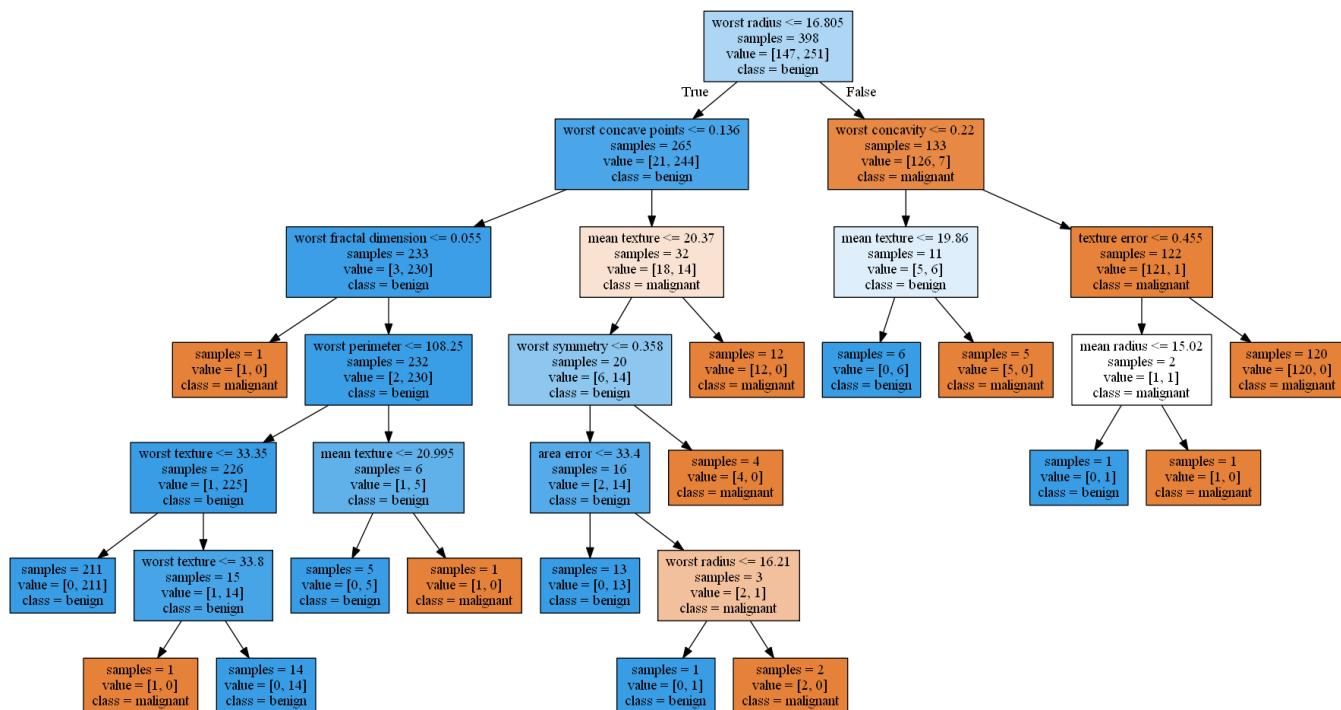
from sklearn.tree import export_graphviz
import graphviz

export_graphviz(dtc, out_file='dt.gv', class_names=['malignant', 'benign'],
feature_names=bc.feature_names, impurity=False, filled=True)

g = graphviz.Source.from_file('dt.gv')

```

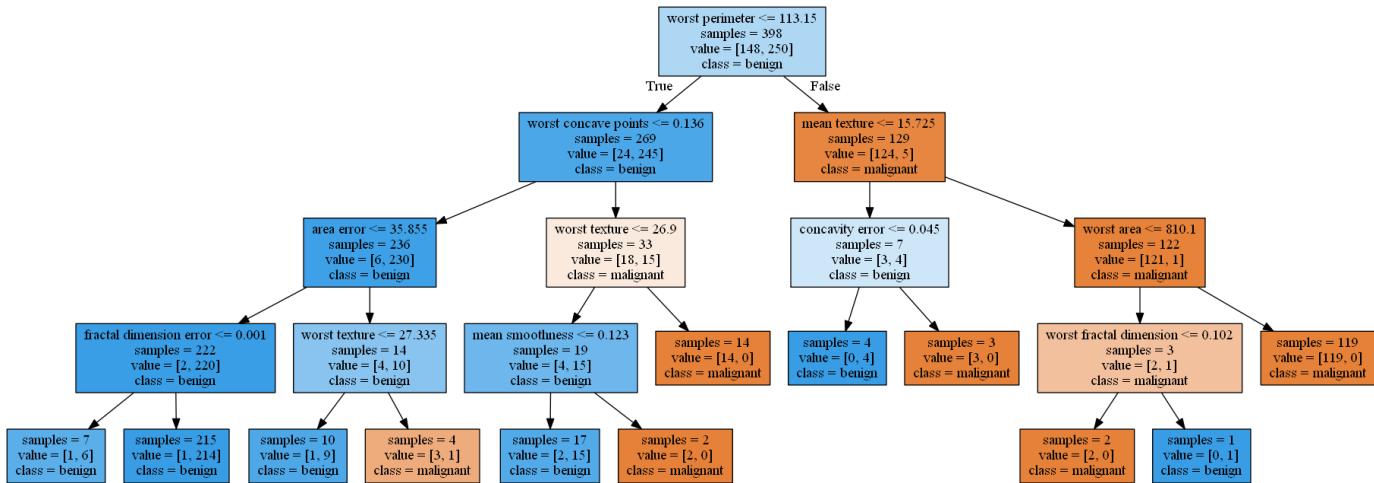
Elde edilen graf aşağıdaki gibidir:



Burada yaprakaların homojen (pure) durumda olduğuna dikkat ediniz. Toplam özellik sayısı 30 olduğu halde eğitimde 6 derinlikte sonuca ulaşılmıştır. Biz DecisionTreeClassifier tür fonksiyonunda bu ağacın oluşturulması ile ilgili çeşitli belirlemeler yapabilmekteyiz. DecisionTreeClassifier sınıfının `__init__` metodunun parametrik yapısı şöyledir:

```
sklearn.tree.DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, class_weight=None, presort='deprecated', ccp_alpha=0.0)
```

Buradaki criterion parametresi 'gini' ya da 'entropy' biçiminde girilebilmektedir. Karar ağaçlarında dallanma için iki temel algoritmanın kullanılabilirliğini belirtmiştir. `max_depth` parametresi ağacın maksimum derinliğini belirlemekte kullanılır. Örneğin bu parametreyi 4 yaparak kodu yeniden çalışıralım.



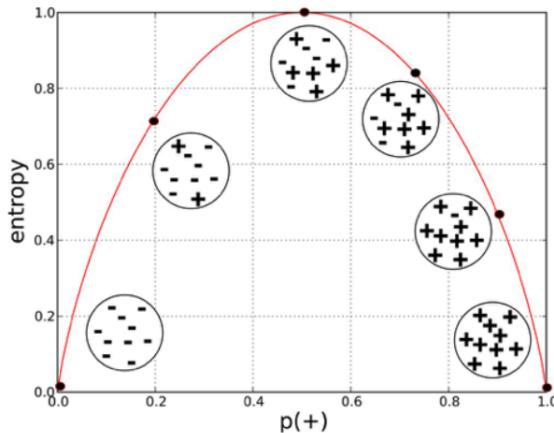
Burada ağacın 4 derinliği olduğu (okların sayısına bakınız) görülmektedir. Yaprakların "pure" olmadığına dikkat ediniz.

Karar Ağaçlarının Algoritmik Temeli

Karar ağaçlarının algoritmik temeli entropy kavramına dayanmaktadır. Entropy sistemdeki düzensizliğin bir ölçüsüdür. Eğer veriler rastgele ise yüksek bir entropy, düzenli ise düşük bir entropy söz konusudur. Karar ağaçlarında entropi hesaplamak için en çok kullanılan yöntem "shannon entropisi" yöntemidir. Shannon entropisi şöyle hesaplanmaktadır:

$$E(S) = \sum_{i=1}^c -p_i \log_2 p_i$$

Buradaki p_i ilgili özellik değerinin veri kümesindeki ortaya çıkma olasılığıdır. Bu olasılık ilgili özelliğin adet toplamının veri kümesindeki satır sayısına bölümü ile elde edilebilir. Entropi düzensizliğin yanı heterojenliğin bir ölçüsüdür. Homojenlik arttıkça entropi azalır, homojenlik azaldıkça entropi artar. Örneğin + ve - karakterlerinin veri kümesi içerisindeki durumlarına göre örnek bir entropi diyagramı şöyle çizilebilir:



Burada görüldüğü gibi homojenlik arttıkça entropi düşmüş, heterojenlik arttıkça entropi artmıştır. Düşey eksende entropi miktarı gösterilmektedir. Entropi miktarı genel olarak $[0, 1]$ aralığında bir gerçek sayı biçiminde ifade edilir. Şekildeki yatay eksende + şekillerinin olasılıkları bulunmaktadır.

Karar ağaçlarında bölünecek en iyi sütunun tespit edilmesi entropi ile ilgilidir. Bölünecek en iyi sütunun tespit edilmesi kabaca şöyle yapılmaktadır: Bütün alternatif sütunlardan bölme yapılır. Her sütun bölündüğünde iki alt düğümün entropileri toplamı hesaplanır. (Alt düğümlerin entropileri hesaplanırken ayrıca Shannon entropy değeri alt düğümün olasılığı ile çarpılmaktadır.) Bu iki alt düğümün entropileri toplamı ana düğümün entropisinden çıkartılır. Buna kazanç (gain) denilmektedir.

$\text{Kazanç} = \text{Üst Düğümün Entropisi} - \text{Alt Düğümlerin Entropileri Toplamı}$

Böylece en büyük kazanca yol açabilecek sütun tespit edilir. Bir sütundaki kazanç ne kadar yüksekse o sütundan bölme yapıldığında o kadar homojen bir alt düğümler elde edilmektedir. Başka bir deyişle kazancın en yüksek olduğu sütun aslında eğer ordan bölme yapılırsa entropinin en fazla düşeceği sütundur." Tabii her bölmeden sonra işlemler yine diğer sütunlar temel alınarak devam ettirilir. Tabii bu işlemin sonucunda aynı sütundan da birden fazla kez bölme yapılıyor olabilir.

Şimdi numpy kullanmadan örnek bir entropy hesabı yapan fonksiyon yazalım. Bu fonksiyonun girdisi çok boyutlu bir liste olsun. Bu listenin son sütun elemanı sınıf değerleri olsun. Örneğin:

```
dataset = [['a', 'x', 'yes'], ['b', 'x', 'yes'], ['a', 'y', 'yes'], ['a', 'x', 'no'], ['a', 'y', 'yes']]
```

Burada iki sütunlu bir veri kümesi vardır. Son sütun "yes", "no" biçiminde iki sınıf belirtmektedir. Bu veri kümesinde karar ağıacı oluşturarak bir tahminlemenin yapılmak istendiğini varsayıyalım. Buradaki dataset'in entropisi nasıl hesaplanacaktır? Entropi hesaplanırken yalnızca sonuçlara bakılır. Yani burada entropi hesabına yalnızca son sütun sokulacaktır. Entropi hesaplayan fonksiyon şöyle yazılabılır:

```
from math import log

def calc_shannon_entropy(dataset):
    nrows = len(dataset)
    label_dict = {}
    for row in dataset:
        label = row[-1]
        if label not in label_dict:
            label_dict[label] = 0
        label_dict[label] += 1
    entropy = 0
    for key in label_dict:
        prob = label_dict[key] / nrows
        entropy += - prob * log(prob, 2)

    return entropy
```

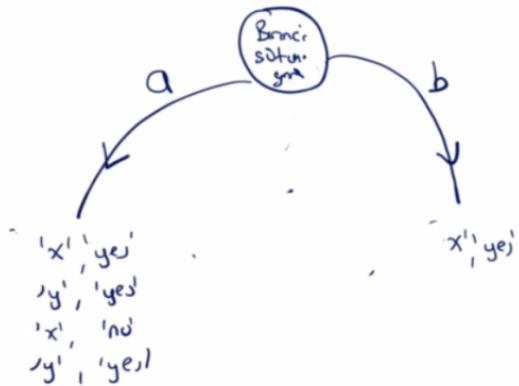
```

dataset = [['a', 'x', 'yes'], ['b', 'x', 'yes'], ['a', 'y', 'yes'], ['a', 'x', 'no'], ['a', 'y', 'yes']]

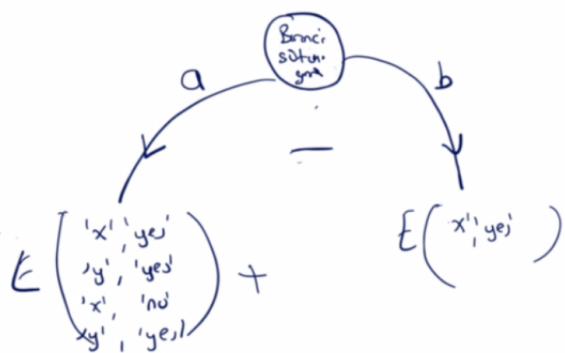
result = calc_shannon_entropy(dataset)
print(result)

```

Şimdi bizim hangi sütuna göre ikiye ayırma yapacağımızı belirlememiz gereklidir. İki tane aday sütun vardır: Birinci sütun ve ikinci sütun. Önce bizim bu iki aday sütuna göre bölme yapıp kazançları hesaplamamız gereklidir. Sonra kazancı en yüksek olan sütun gerçek bölmenin yapılacağı sütun olarak belirlenecek ve o sütun bölünecektir. Biz birinci sütuna göre bölme yaparsak sol ve sağ düğümler şöyle olur:



Biz burada kazancı şöyle hesaplarız:



Şimdi alt düğümlerin entropilerini ve kazancı hesaplayalım:

```

result = calc_shannon_entropy(dataset)
print(result)

first_left = [['x', 'yes'], ['y', 'yes'], ['x', 'no'], ['y', 'yes']]
first_right = [['x', 'yes']]

prob = len(first_left) / len(dataset)
first_left_result = prob * calc_shannon_entropy(first_left)

prob = len(first_right) / len(dataset)
first_right_result = prob * calc_shannon_entropy(first_right)

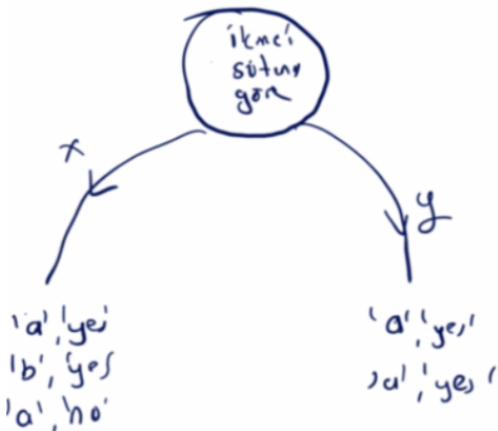
first_parent_result = calc_shannon_entropy(dataset)
first_gain = first_parent_result - (first_left_result + first_right_result)

print(first_gain)

```

Buradan elde edilen sonuç 0.072'dir.

Şimdi ikinci sütundan bölme yaptığımızı düşünelim:



Şimdi alt düğümlerin entropilerini ve kazancı hesaplayalım:

```
second_left = [['a', 'yes'], ['b', 'yes'], ['a', 'no']]
second_right = [['a', 'yes'], ['a', 'yes']]

prob = len(second_left) / len(dataset)
second_left_result = prob * calc_shannon_entropy(second_left)

prob = len(second_right) / len(dataset)
second_right_result = prob * calc_shannon_entropy(second_right)

second_parent_result = calc_shannon_entropy(dataset)
second_gain = second_parent_result - (second_left_result + second_right_result)

print(second_gain)
```

Buradan 0.170 sonucu elde edilmiştir. Göründüğü gibi $0.072 < 0.170$ biçimindedir. Bu durumda bölme ikinci sütuna göre yapılacaktır.

İşte algoritmada alt düğümler de benzer biçimde bölünerek ilerlenir. Eğer bir düğüm tamamen homojen (pure) biçimde gelirse (yani entropisi 0 olursa) artık o düğüm bir yaprak durumundadır ve daha fazla bölme yapılmaz.

Aslında bölme işlemini yapan bir fonksiyon da benzer biçimde yazılabılır. Örneğin:

```
def split_dataset(dataset, column, value):
    result = []
    for row in dataset:
        if row[column] == value:
            result.append(row[:column] + row[column + 1:])

    return result
```

Şimdi de en iyi sütunun seçilmesini yapan fonksiyonu yazalım:

```
def find_best_feature(dataset):
    nfeatures = len(dataset[0]) - 1
    parent_entropy = calc_shannon_entropy(dataset)
    max_gain = float('-inf')
    max_feature = -1
    for i in range(nfeatures):
        feature_list = [row[i] for row in dataset]
        uniques = set(feature_list)
```

```

sub_entropy = 0
for value in uniques:
    sub_dataset = split_dataset(dataset, i, value)
    sub_entropy += calc_shannon_entropy(sub_dataset)
gain = parent_entropy - sub_entropy
if gain > max_gain:
    max_gain = gain
    max_feature = i

return max_feature, max_gain

dataset = [['a', 'x', 'yes'], ['b', 'x', 'yes'], ['a', 'y', 'yes'], ['a', 'x', 'no'], ['a', 'y', 'yes']]

```

result = find_best_feature(dataset)
print(result)

Şimdi de yukarıdaki örneği scikit-learn ile çözelim. scikit-learn aslında sütunların kategorik olup olmadığını bilmez. Her şeyi sanki sayısal bilgiymiş gibi yapar. Bizim kategorik alanları sayısal biçimde dönüştürmemiz gereklidir. (Bunun için LabelEncoder sınıfının kullanıldığıni anımsayınız.)

```

dataset = [['a', 'x', 'yes'], ['b', 'x', 'yes'], ['a', 'y', 'yes'], ['a', 'x', 'no'], ['a', 'y', 'yes']]

result = find_best_feature(dataset)
print(result)

import numpy as np
from sklearn.tree import DecisionTreeClassifier

training_dataset_x = np.array([[0, 0], [1, 0], [0, 1], [0, 0], [0, 1]])
training_dataset_y = np.array([1, 1, 1, 0, 1])

dt = DecisionTreeClassifier(criterion='entropy')
dt.fit(training_dataset_x, training_dataset_y)

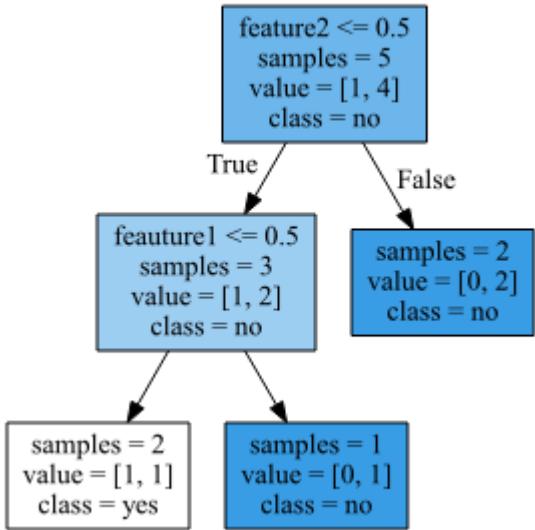
from sklearn.tree import export_graphviz
import graphviz

export_graphviz(dt, out_file='dt.gv', class_names=['yes', 'no'], feature_names=['feature1', 'feature2'], impurity=False, filled=True)

g = graphviz.Source.from_file('dt.gv')

```

Elde edilen grafik şöyledir:



Sınıflandırma İşlemlerinde Lojistik Regresyon İle Karar Ağaçlarının Karşılaştırılması

Lojistik regresyon ile karar ağaçları toplamda birbirlerine yakın performans göstermektedir. Ancak aralarındaki belirgin farklılıklar şöyle özetlenebilir:

- Karar ağaçları insan düşüncesine daha yakındır. Dolayısıyla verilen kararın izlediği yol sözcüklerle daha iyi açıklanabilmektedir.
- Lojistik regresyon parametrik, karar ağaçları ise parametrik olmayan bir modelde sahiptir.
- Lojistik regresyonda verilerin dağılımı önemli olabilmektedir. Halbuki karar ağaçlarında dağılımin önemi yoktur.
- Lojistik regresyonda üç değerler sonucu karar ağaçlarına göre daha olumsuz etkilemektedir.
- Lojistik regresyon özellikle sürekli değişkenler üzerinde uygulanmaktadır. Değişkenler kesikli ise karar ağaçları daha iyi performans göstermektedir.
- Genel olarak karar ağaçlarının overfit durumuna yatkınlığı lojistik regresyondan fazla olma eğilimindedir. Yani karar ağaçlarının eğitimde uygulanan kümeye bağlı olarak sınıflandırmayı yanlış öğrenme potansiyeli lojistik regresyona göre daha fazladır.

Karar Ağaçları İle Lojistik Olmayan Regresyon İşlemleri

Karar ağaçları yalnızca lojistik regresyon (sınıflandırma) amacıyla değil lojistik olmayan regresyon amacıyla da kullanılabilir. Kullanım mantığı çok benzerdir. Burada biz scikit-learn ile lojistik olmayan karar ağaç regresyonlarının nasıl olduğu yapılabileceği üzerinde duracağız. Karar ağaçlarının lojistik olmayan regresyon amaçlı kullanılması için scikit-learn içerisindeki DecisionTreeRegressor sınıfı kullanılmaktadır. Sınıfın `__init__` metodunun parametrik yapısı şöyledir:

```
sklearn.tree.DecisionTreeRegressor(criterion='mse', splitter='best', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None,
random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None,
presort='deprecated', ccp_alpha=0.0)
```

Buradaki criterion 'mse', 'friedman_mse' ya da 'mae' olabilir. Yine max_depth parametresi ağacın maksimum derinliğini ayarlamakta kullanılır. DecisionTreeRegressor nesnesi yaratıldiktan sonra yine fit metoduyla eğitim yapılpredict metodu ile tahminleme yapılmaktadır. Aşağıda Boston ev fiyatlarıyla ilgili bir karar ağaç regresyonu görülmektedir:

```

import numpy as np

from sklearn.datasets import load_boston

boston = load_boston()
dataset_x = boston.data
dataset_y = boston.target

from sklearn.tree import DecisionTreeRegressor

dtr = DecisionTreeRegressor()
dtr.fit(dataset_x, dataset_y)

predict_x = np.array([[3.2370e-02, 0.0000e+00, 2.1800e+00, 0.0000e+00, 5.8800e-01,
                      6.9980e+00, 5.5800e+01, 6.0622e+00, 3.0000e+00, 2.2200e+02,
                      1.8700e+01, 3.9463e+02, 2.9400e+00]]))

result = dtr.predict(predict_x)
print(result)

from sklearn.tree import export_graphviz
import graphviz

export_graphviz(dtr, out_file='dtr.gv', impurity=False, filled=True)

g = graphviz.Source.from_file('dtr.gv')

```

Karar ağaçlarıyla regresyon işlemi aslında sınıflandırma işlemine çok benzemektedir. Ağaçtan elde edilen yapraklar eğitim veri kümesindeki değerler olmaktadır. Tabii birden fazla aynı değere ilişkin yaprak karar ağacında bulunabilmektedir. Şüphesiz bu yöntemde fazla sayıda veri ile eğitim uygulamak kestirimi iyileştirecektir. Mademki bu yöntemde yapraklar aslında gerçek sonuç gözlem değerleridir o halde bu değerlerin dışında bir sonuç elde edilmesi beklenmemelidir. Yani eğitim veri kümesinde olmayan sonuçların tahminlemesi bu yöntemde mümkün değildir. Karar ağaçlarıyla lojistik olmayan regresyonu n eğitimdeki veri kğmesi kadar sınıfı sahip lojistik regresyon gibi düşünebilirsiniz.

Karar Ağaçlarında Paketleme (Bagging)

Yukarıda da belirtildiği gibi karar ağaçlarının en önemli sorunu "overfit" durumuna yatkınlıktır. Overfit aslında yüksek varsayınsa kendini göstermektedir. Örneğin biz eğitim veri kümesini iki yarıya bölüp iki karar ağıacı oluşturursak bu ağaçlar bu iki kümenin o andaki durumuna bağlı olarak farklı olabilecektir. Bu durum yüksek varyansı anlatmaktadır. Oysa lojistik regresyonlarda karar ağaçlarına göre varyans düşüktür. Ancak lojistik regresyonlar da üç değerlerden kötü bir biçimde etkilenme eğilimindedir. O halde karar ağaçlarını iyileştirmek için yapılacak şey varyansı düşürmektir. Varyansı düşürmenin ilk akla gelen makul yöntemi ise tek bir ağaç değil birden fazla ağaç kullanmaktır. (Birden fazla ağaca bu terminolojide "orman (forest)" da denilmektedir.)

Paketleme işleminde eğitim veri kümesi içerisindeki belli miktarda satırlardan ağaçlar oluşturulmaktadır. Örneğin eğitim veri kümesinde 100 tane satır olsun. Biz 30'ar satırdan oluşan 500 tane örnek elde edip 500 tane karar ağıacı oluşturabiliriz. Tabii örnekler (samples) rastgele bir biçimde seçilmelidir. Örneklerde genellikle iadelî seçim uygulanmaktadır. (Yani böylece bir örneklem aynı elemanlara sahip olabilmektedir.) Paketleme yönteminde satırlardan örnekler oluşturulmakla birlikte sütunların (yani değişkenlerin) hepsi alınmaktadır.

Paketleme işlemi sonucunda toplam n tane karar ağıacı elde edilmektedir. Pekiyi bu ağaçlardan nasıl faydalılaşacaktır? İşte varyansı düşürebilmek amacıyla buradan bir ortalama değer hesaplanır. Ortalama değer sınıflandırma problemleri için en yüksek sınıf temelinde yapılmaktadır. Lojistik olmayan regresyon problemlerinde ise gerçekten bir aritmetik ortalama hesaplanabilmektedir. Böylece lojistik olmayan regresyon problemlerinde eğitim veri kümesinde olmayan değerler de elde edilebilmektedir.

Örneğin sınıflandırma problemi için 500 tane karar ağacının oluşturulduğu düşünelim. Sonra da kestirim (prediction) işlemi yapmak isteyelim. İşte yöntemde bu kestirilecek veri tüm bu 500 ağaca da sokulmaktadır. Bu 500 ağacın sonucunda elde edilen değerlere bakılarak en yaygın sınıf kestirimini sonucu olarak belirlenmektedir. Yine lojistik olmayan bir regresyon problemi için 500 karar ağacının oluşturulduğunu düşünelim. Kestirim yapıılırken kestirilecek değer bu 500 karar ağacına da sokulacak ve elde edilen değerlerin aritmetik ortalaması regresyon sonucu olarak belirlenecektir.

Şüphesiz paketleme yöntemi normal karar ağacı yöntemine göre çok daha fazla bilgisayar zamanının kullanılmasına yol açmaktadır. Ancak bu yöntemin performansı normal karar ağacı yöntemine göre daha iyidir.

Pekiyi bu yöntemde eğitim veri kümesindeki toplam satır sayısı n olmak üzere kaçarlı kaç ağaç (yani alt veri kümesi) oluşturulmalıdır? Şüphesiz ağaç sayısı ne kadar fazla olursa o kadar iyidir. Ancak belirli sayıda ağaçtan sonra bilgisayar zamanı artmasına karşın iyileşme oranı da azalmaktadır. Örneğin ağaç sayısı için 100, 200, 300 gibi değerler kullanılabilir. Örneklem için kullanılacak satır sayıları eğitim veri kümesinin miktarına da bağlı olarak belirlenebilmektedir. Yüksek miktardaki eğitim veri kümesi için örneklem veri miktarı sınıflandırma problemlerinde \sqrt{n} , regresyon problemlerinde $n / 2$ ya da $n / 3$ gibi değerde tutulabilir. Fakat düşük miktarda eğitim veri kümesinde örneklem büyülüğu daha yüksek olabilir.

Scikit-learn kütüphanesinde sklearn.ensemble modülündeki BaggingClassifier sınıfı paketleme yöntemi ile karar ağacı sınıflandırması yapmakta kullanılır. Benzer biçimde aynı modüldeki BaggingRegressor sınıfı da paketleme yöntemiyle lojistik olmayan karar ağacı regresyonu için kullanılmaktadır. BaggingClassifier sınıfının `__init__` metodunun parametrik yapısı şöyledir:

```
class sklearn.ensemble.BaggingClassifier(base_estimator=None, n_estimators=10, max_samples=1.0, max_features=1.0, bootstrap=True, bootstrap_features=False, oob_score=False, warm_start=False, n_jobs=None, random_state=None, verbose=0)
```

Fonksiyonun birinci parametresi olan `base_estimator` kullanılacak yöntemi belirtmektedir. Bu parametre için argüman girilmezse default durum karar ağacılarının kullanılmasıdır. Fonksiyonun `n_estimators` parametresi toplamda oluşturulacak karar ağacılarının sayısını belirtir. Fonksiyonun `max_samples` parametresi tamsayı olarak girilirse örneklem büyülüğini float olarak girilse örneklem oranını belirtmektedir. `bootstrap` parametresi ise örneklemi iadelı mi iadesiz mi yapılacağını belirtmektedir. Default durum iadelı örneklemidir.

BaggingClassifier sınıfının `fit` ve `predict` metotları diğer sınıflarda olduğu gibidir. Şimdi BaggingClassifier sınıfını ile meme kanseri veri ile kullanalım.

```
from sklearn.datasets import load_breast_cancer

bc = load_breast_cancer()
print('Sütun isimleri: {}'.format(bc.feature_names))
print('Sınıf Isimleri: {}'.format(bc.target_names))

dataset_x = bc.data
dataset_y = bc.target

from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.3)

from sklearn.ensemble import BaggingClassifier

dt = BaggingClassifier(n_estimators=300, max_samples=0.4)
dt.fit(training_dataset_x, training_dataset_y)
score = dt.score(test_dataset_x, test_dataset_y)
print(score)
```

Buradaki sonuçlar önceki normal karar ağacından elde ettiğimiz sonuçlardan genel olarak %3 civarında daha iyidir.

Paketleme ile lojistik olmayan regresyon işlemi de yapılabilmektedir. Bunun için ise BaggingRegressor sınıfı kullanılmaktadır. Aşağıda Boston verileriyle böyle bir lojistik olmayan regresyon örneği görüyorsunuz:

```
import numpy as np

from sklearn.datasets import load_boston
from sklearn.ensemble import BaggingRegressor

boston = load_boston()
dtr = BaggingRegressor(n_estimators=300, max_samples=0.4)
dtr.fit(boston.data, boston.target)

predict_x = np.array([[3.2370e-02, 0.0000e+00, 2.1800e+00, 0.0000e+00, 5.8800e-01,
                      6.9980e+00, 5.5800e+01, 6.0622e+00, 3.0000e+00, 2.2200e+02,
                      1.8700e+01, 3.9463e+02, 2.9400e+00]]))

result = dtr.predict(predict_x)
print(result)
```

RASSAL ORMANLAR (Random Forest)

Aslında rassal orman denilen yöntem paketleme (bagging) yönteminin bir ileri aşamasıdır. Anımsanacağı gibi paketleme yönteminde rastgele seçilen belli miktarda satırlardan karar ağaçları oluşturuluyordu. Ancak bu ağaçlarda sütunların hepsi kullanılıyordu. İşte paketleme yönteminde tüm sütunların (yani değişkenlerin) ağaca dahil edilmesi oluşturulan farklı ağaçların birbirlerine benzemesine yol açabilmektedir. Bu da varyansı yükseltten bir faktör durumuna gelebilmektedir. Ayrıca paketleme yönteminde tüm sütunların işleme sokulması ağırlığı yüksek sütunların tüm ağaçlarda etkili olmasına yol açmaktadır. İşte rassal orman yönteminde yalnızca eğitim veri kümelerindeki satırlar değil aynı zamanda sütunlar da örneklenmektedir. Bu nedenle rassal orman yöntemi paketleme yöntemine göre daha üstün kabul edilmektedir.

Rassal orman yöntemi için scikit-learn kütüphanesindeki ensemble modülünde RandomForestClassifier ve RandomForestRegressor isimli sınıflar bulunmaktadır. RandomForestClassifier sınıfının `__init__` metodunun parametrik yapısı şöyledir:

```
class sklearn.ensemble.RandomForestClassifier(n_estimators=100, criterion='gini', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto',
max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False,
n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None)
```

Yine fonksiyonun `n_estimator` parametresi toplam ağaç sayısını belirtir. `criterion` parametresi karar ağaçları için kullanılacak yöntemi belirtmektedir. Bu parametre '`gini`', ya da '`entropy`' biçiminde girilebilmektedir. Fonksiyonun pek çok parametresi ağaçın üretiminin durdurulması ile ilgilidir. Fonksiyonun `max_features` parametresi `int`, `float` ya da `str` türünden değer alabilmektedir. `int` değer doğrudan rastgele alınacak sütun sayısını belirtir. `Float` değer yine oran belirtmektedir. Eğer bu parametre yazışal biçimde '`sqrt`' olarak girilirse sütun sayısı toplam sütunların karekökü kadar alınmaktadır. Zaten bu parametre için argüman girilmezse default olarak '`auto`' alındığına dikkat ediniz. '`auto`' taamen '`sqrt`' ile aynı anlama gelmektedir.

`RandomForestClassifier` kullanımına şöyle bir örnek verebiliriz:

```
from sklearn.datasets import load_breast_cancer

bc = load_breast_cancer()
print('Sütun isimleri: {}'.format(bc.feature_names))
print('Sınıf İsimleri: {}'.format(bc.target_names))

dataset_x = bc.data
dataset_y = bc.target
```

```

from sklearn.model_selection import train_test_split

training_dataset_x, test_dataset_x, training_dataset_y, test_dataset_y =
train_test_split(dataset_x, dataset_y, test_size=0.3)

from sklearn.ensemble import RandomForestClassifier

dt = RandomForestClassifier(n_estimators=300, max_features=0.6)
dt.fit(training_dataset_x, training_dataset_y)
score = dt.score(test_dataset_x, test_dataset_y)
print(score)

```

Benzer biçimde rassal ormanlarla regresyon uygulaması da yapılabilmektedir. Genel prensip sınıflandırma problemleriyle aynıdır. Bunun için sklearn.ensemble modülündeki RandomForestRegressor sınıfı kullanılmaktadır. RandomForestRegressor sınıfının `__init__` metodunun parametrik yapısı şöyledir:

```

class sklearn.ensemble.RandomForestRegressor(n_estimators=100, criterion='mse', max_depth=None,
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto',
max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False,
n_jobs=None, random_state=None, verbose=0, warm_start=False, ccp_alpha=0.0, max_samples=None)

```

Aşağıda RandomForestRegressor sınıfı ile Boston örneği verilmiştir:

```

import numpy as np

from sklearn.datasets import load_boston
from sklearn.ensemble import RandomForestRegressor

boston = load_boston()
rfr = RandomForestRegressor(n_estimators=300, max_features=0.6)
rfr.fit(boston.data, boston.target)

predict_x = np.array([[3.2370e-02, 0.0000e+00, 2.1800e+00, 0.0000e+00, 5.8800e-01,
6.9980e+00, 5.5800e+01, 6.0622e+00, 3.0000e+00, 2.2200e+02,
1.8700e+01, 3.9463e+02, 2.9400e+00]]))

result = rfr.predict(predict_x)
print(result)

```

Pekiştirmeli Öğrenme (Reinforcement Learning)

Yapay zekadaki öğrenme (reinforcement learning) psikolojideki edimsel koşullanma (operant conditioning) kuramı temel alınarak oluşturulmuştur. Psikolojide edimsel koşullanma "ödül getiren davranışların tekrarlandığı" fikrine dayanır. Bu konudaki ilk çalışmalar Edward Thorndike tarafından yapılmıştır. Thorndike kedileri belli bir alana kapatıp belli bir düğmeye bastıklarında onların çıkışmasına olanak sağlayan bir düzenek oluşturmuştur. Sonra kediler bu düğmeye tesadüfen bastıklarında ödül olarak dışarı çıkmışlardır. Thorndike bu deneyi aynı kediler üzerinde tekrarladıkça kediler düğmeye basma işini çok daha hızlı yapmaya başlamıştır. Thorndike buna "etki yasası (law of effect)" demiştir. Fakat edimsel koşullamayı kuramsallaştıran asıl kişi B.F. Skinner'dır. Skinner Thorndike'in deneylerini genişleterek bu alanı çok daha kapsamlı hale getirmiştir. İşte "pekiştirmeye (reinforcement)" terimi de Skinner tarafından uydurulmuştur. Skinner organizmada hazır uyandıran edimlerin (eylemlerin) yinelendiğini pek çok deneyle göstermiştir. Buna göre organizma birtakım faaliyetlerde bulunur. Bazı faaliyetleronda hazır uyandırmaktadır. Başka bir deyişle organizma bazı faaliyetlerden bazı ödüller (rewards) elde etmektedir. Yapılan faaliyetler karşısında ödül elde etme sürecine "pekiştirmeye (reinforcement)"i burada verilen ödülü de "pekiştireç (reinforcer)" denir. Edimsel koşullanma davranış değiştirmede ve davranışın şekillendirilmesinde en etkili yöntemlerden biri olarak kabul edilmektedir.

Edimsel koşullanmada davranışı tekrarlatmak için verilen uyarana pekiştireç (reinforcer) dendiğini belirtmişik. Pekiştireçler de "pozitif" ve "negatif" olmak üzere ikiye ayrılmaktadır. Organizmaya doğrudan hazır veren pekiştireçlere pozitif pekiştireçler denir. (Örneğin bir çocuğa ödül olarak verilen şeker gibi, bir çalışana ödül olarak verilen para gibi.) Ortamda olumsuz bir uyarani (yani organizmada gerginlik yaratılan bir uyarani) ortadan kaldırarak organizmanın yine hazır elde etmesi sağlanabilir. İşte bu biçimde ortamdan acı verici bir uyarının kaldırılması durumunda bu uyarana da "negatif pekiştireç" denilmektedir. Hem pozitif hem de negatif pekiştireçler sonuç olarak organizmada hazır uyandırmaktadır. Edimsel koşullanmada önemli bir olgu da "ceza (punishment)"dır. Organizmada hoş gitmeyen bir etki oluşturan uyararlara bu bağlamda "ceza" denilmektedir. Ceza da bir pekiştireçtir. Ancak araştırmalar davranış değiştirmede ve davranışın kalıcılığını sağlamada cezaların çok iyi iş görmediğini göstermektedir. Ceza çabuk etki göstermesi bakımından pratik bir yöntemdir ancak genellikle kalıcı bir davranış değişikliğine yol açmaz ve saldırganlığı artırmak gibi dolaylı yan etikileri vardır. Cezaalar da "birinci" ve "ikincil" olmak üzere ikiye ayrılmaktadır. Organizamada doğrudan acı veren uyarının uygulanmasına birincil ceza, organizmada hazır uyandıran bir uyarının ortadan kaldırılması biçiminde verilen cezaya da ikincil ceza verilmektedir. Ceza verilecekse ikincil cezalar tercih edilmelidir.

Pekiştirmeye sürecinin diğer bir türü de "dolaylı pekiştirmeye (vicarious reinforcement)" denilen biçimdir. Dolaylı pekiştirmeye başlarını izleyerek ve başlarının ödül aldığı ya da ceza aldığına göre bundan bir sonuç çıkartma sürecidir. Tabii burada bazı akıl yürütme faaliyetleri de deveye girdiği için sürecin bilişsel tarafı da vardır. Bu nedenle bu öğrenme modeline "sosyal öğrenme" ya da "sosyal bilişsel öğrenme" denilmektedir.

Pekiştirmeli Makine Öğrenmesinin Ana Fikri ve Temelleri

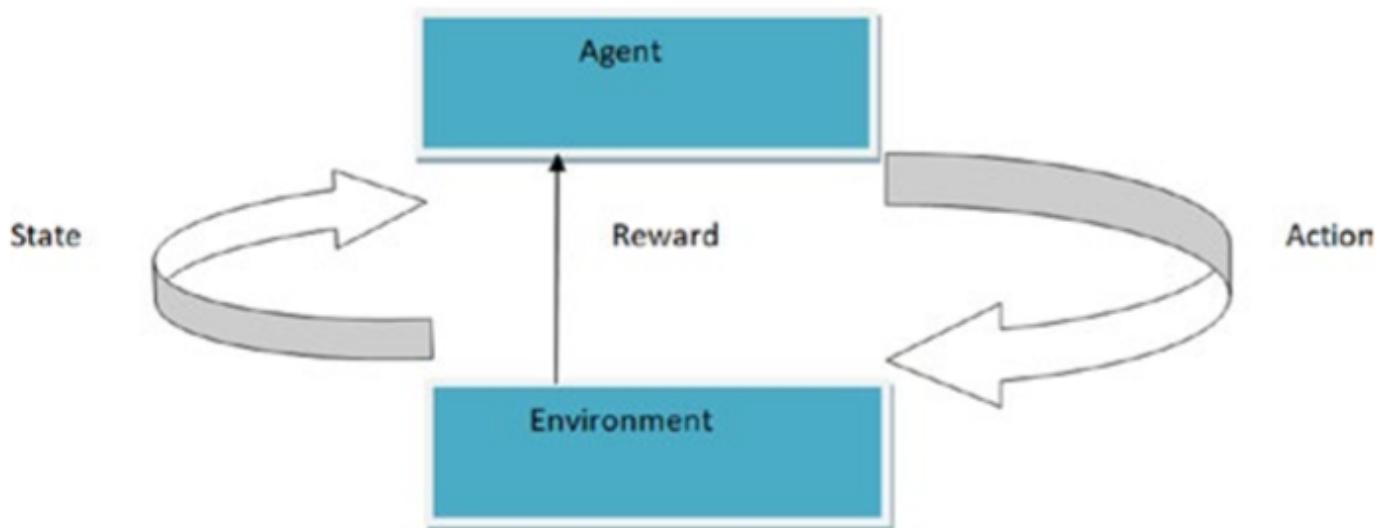
Pekiştirmeli öğrenmede ana fikir belli bir edimin (action) çevre (environment) ile etkileşime sokulması bunun sonucunda bir ödül ceza sisteminin uygulanması ve gitgide bu ödüllerin maksimize edilmeye çalışılmasıdır. Pekiştirmeli makine öğrenmesi sürecinde bazı terimler sıkça kullanılmaktadır:

- Yazılımsal Etmen (Software Agent): Pekiştirmeli öğrenmeyi gerçekleştiren yazılım ve algoritmalarla "yazılımsal etmen" denilmektedir. Genellikle "yazılımsal etmen" yerine yalnızca "etmen" terimi kullanılmaktadır. Biz de burada "yazılımsal etmen" yerine "etmen" terimini kullanacağız.
- Çevre (Environment): Etmenin faaliyette bulunduğu ortama denilmektedir. Pekiştirmeli öğrenmede çevreyi oluşturmak başka bir deyişle simüle etmek önemli bir problemdir. Bunun için hazır platformlardan faydalananmaktadır.
- Faaliyet (Action) Yazılımsal etmenin yaptığı edimlere faaliyet (action) denilmektedir.

- Durum (Status): Durum yazılımsal etmenin belli bir andaki verisel özellikleridir. Yazılımsal etmen bir faaliyette bulunduğu zaman durum değişikliği oluşur. İşte yazılımsal etmenin bir durumdan diğerine geçmesi söz konusudur. Bu haliyle konu "automata" teorisile ve durum makineleriyle (state machines) ilgili olmaktadır.

- Ödül / Ceza (Reward /Punishment): Yazılımsal etmen bir faaliyette bulunduğuanda algoritmik olarak ona bir ödül ya da ceza verilmektedir. Şüphesiz bu kavram pekiştirmeli öğrenmenin en önemli prensibini oluşturmaktadır.

Bu durumda yukarıdaki kavamlarla tipik bir pekiştirmeli öğrenme modeli aşağıdaki gibi bir şekilde betimlenebilir:



(Şekil "Reinforcement Learning With Open AI, TensorFlow and Keras Using Python" kitabından alınmıştır.

Bu şekilde anlatılmak istenen "etmenin çevre ile etkileşiminde bir faaliyette bulunduğu, bu faaliyet sonucunda bir durum değişikliğinin olduğu ve yeni durum için bir ödül/ceza sisteminin devreye sokulduğu"dur.

Etmenin içinde bulunduğu çevrenin özellikleri aşağıdakilere ilişkin olabilir:

- Deterministik Olan ya da Deterministik Olmayan Çevre: Eğer bir durumda gidilebilecek tek bir durumsal seçenek varsa ya da birden fazla durumsal seçeneklerin hangi seçileceği eldeki bilgilerle belirlenebiliyorsa bu tür çevrelere deterministik çevre denilmektedir. Deterministik çevrelerde aynı durumlardan aynı sonuçlar elde edilmektedir.

Ayırık ya da Sürekli Çevre: Ayırık çevreden kastedilen belli bir durumda olabilecek seçeneklerin sınırlı sayıda olmasıdır. Yani etmen belli bir durumda sınırlı sayıda faaliyette (action) bulunabilir. Bu matematikteki "ayırık değişken" kavramından biraz farklıdır. Sürekli çevrede ise bir durumda sonsuz miktarda olası faaliyet (action) söz konusu olmaktadır. Örneğin bir labirentte belli bir noktada sapılacak yolların sayısı sınırlıdır. Bu labirent ayırık bir çevre oluşturmaktadır.

Gözlemlenebilme (Observability) Derecesine Göre Çevreler: Gözlemlenebilir çevre demekle çevrenin bütünsel durumunun bilinmesi anlatılmak istenmektedir. Yani biz çevredeki tüm öğelerin yerlerini, değerlerini vs. biliyorsak bu gözlemlenebilir bir çevredir. Eğer biz çevreyi oluşturan öğelerin durumlarını tam olarak bilmiyorsak bu çevre gözlemlenebilir değildir. Tabii gözlemlenebilirlik bir derece iştir. Yani bir çevrede bazı öğeler gözlemlenebilirken bazıları gözlemlenebilir olmayıabilir.

Pekiştirmeli Öğrenmenin Uygulama Alanları

Pekiştirmeli öğrenmenin en önemli uygulama alanları şunlardır:

- Otomatik Kontrol Sistemleri: Otomatik kontrol sistemleri çevredeki değişimlere göre uygun konum alabilen sistemlerdir. Pekiştirmeli öğrenmenin tipik uygulama alanlarından birini oluşturmaktadır. Örneğin adaptif trafik

kontrollerinde kırmızı ışığın ne süreyle yanıp söndürüleceği bu biçimde sisteme öğretilebilir. Benzer biçimde havadaki nem, basınç gibi faktörlere bağlı olarak birtakım denegelme işlemlerinin yapılması da otomatik kontrol sistemlerine örnek olarak verilebilir.

- Finansal Problemler: Finansal süreçler tipik olarak birtakım alternatifler arasında uygun bir alternatifin belirlenmesi ile ilgilidir. Burada pekiştirmeli öğrenme "öğrenilecek şeyin net olarak bilinmediği" durumlarda kullanılabilketedir.

- Üretim Problemleri: Üretimdeki bazı problemler faaliyet ve durum değişikliği ile ilişkilidir. Bu tür durumlarda pekiştirmeli öğrenmeler kullanılabilir.

- Stok Yönetimi Problemleri: Optimal stok miktarlarının belirlenmesi üretimde önemli bir sorundur. Bu tür sistemler pek çok faktörü olan bu sorun için iyi bir seçenek sunabilmektedir.

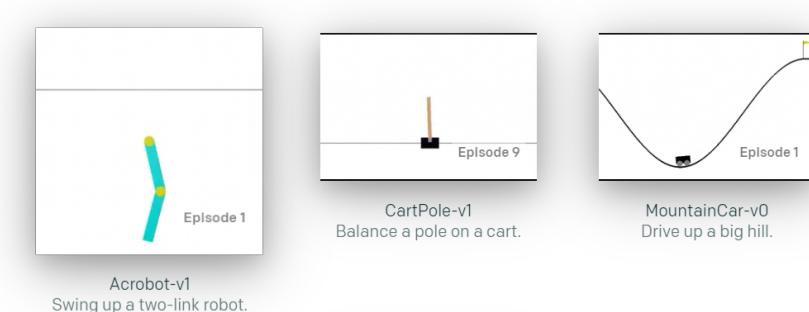
- Dağıtım Problemleri: Genel olarak bir grup ürünün ne zaman ve hangi rotayla dağıtılacagına ilişkin problemlerde de pekiştirmeli öğrenme sıkça kullanılmaktadır.

OpenAI GYM Ortamı

OpenAI yapay zeka konusunda faaliyet gösteren bir kurumdur. OpenAI'nın gym denilen ortamı (gym.openai.com) pekiştirmeli öğrenme için iyi bir simülasyon ortamı sunmaktadır. Tabii gym dışında başka ortamlar da mevcuttur. Örneğin benzer özellikler Google'ın DeepMind ortamında da bulunmaktadır. Biz kursumuzda OpenAI'nın Gym ortamını kullanacağımız.

OpenAI Gym ortamında çeşitli konulara ayrılmış çeşitli simülatörler vardır. Bu simülatörlerde pek çok şey hazırlıdır. Bütün bunlaraslında kullanıcıların algoritma geliştirmeleri için hazırlanmıştır. GYM ortamındaki simülatörler çeşitli kategorilere ayrılmıştır. Örneğin "Classical Control" kategorisindeki simülatörler şunlardır:

Classic control
Control theory problems from the classic RL literature.



Acrobot-v1
Swing up a two-link robot.

CartPole-v1
Balance a pole on a cart.

MountainCar-v0
Drive up a big hill.



MountainCarContinuous-v0
Drive up a big hill with
continuous control.



Pendulum-v0
Swing up a pendulum.

Ortamı kurmak için aşağıdaki komut uygulanabilir:

```
pip install gym
```

Gym CartPole Simülatörü

Cartpole ("Cart" araba, "pole" ise direk anlamına gelmektedir) simülatöründe bir küçük arabanın tepesine bir direk yerleştirilmiştir. Bu direk arabaya katı biçimde monte edilmemiştir ve mafsallardan oynayabilmektedir. Araba hareket ederse bu direk sola ya da sağa düşer. Bu simülatörde kullanıcı arabaya soldan ya da sağdan bir kuvvet uygular. (Kullanıcı arabanın hızını doğrudan belirlemez. Yalnızca kuvvet uygular.) Uygulanan kuvvet sabittir. Eğer bu kuvvet aynı biçimde uygulanmaya devam ederse şüphesiz araba ivmeli bir biçimde yanı gittikçe hızlanarak hareket edecektir. Oyundan amaç arabaya monte edilmiş direğin devrilmeden +15, -15 derece arasında dik tutulmasıdır. Aynı zamanda arabanın merkezden 2.4 birim uzaklaşmaması gerekmektedir. Yani burada başarısızlığın iki nedeni vardır: Direğin +15, -15'lik dereceden aşağıda olması ve arabanın merkezden 2.4 birim uzaklaşmış olmasıdır. Özette modelin parametrik bilgileri şöyledir:

Faaliyetler (actions): İki faaliyet (action) tanımlıdır: Sola ya da sağa kuvvet uygulamak.

Gözlem Uzayı (Observation Space): Her durumda (state) ortamdan elde edilen gözlem bilgileri şunlardır:

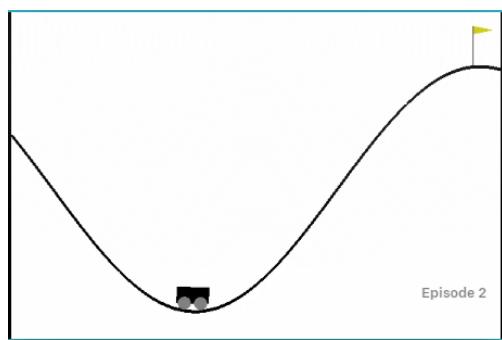
- Arabanın konumu (merkez 0 olmak üzere)
- Arabanın o andaki hızı
- Direğin açısı
- Direğin açısal hızı

Durumlar (States): Buradaki durumlar zamansal biçimde birbirlerinden ayrılmıştır. Zaman sürekli bir olgudur. Ancak burada belli zaman aralıklarındaki durumlar ele alınmaktadır. Yani bir durumdan (state) diğerine geçiş bir t zamanı sonra olmaktadır. Başka bir deyişle belli bir zaman aralığı sonucunda ortamda değişim yeni bir duruma karşı gelmektedir.

Ödül Sistemi: Bu simülatörde ödül sistemi de simülatörün kendisi tarafından verilmektedir. Tabii programcı başka ödül sistemlerini de kullanabilir. Ancak genel olarak ödül değeri direk -15, +15 açısı arasında ise +1 olarak değilse 0 olarak verilmektedir.

Gym MountainCar Simülatörü

MountainCar simülatöründe bir araba bir tepeyi aşmaya çalışmaktadır. Arabayı idare edenin yapabileceği üç şey vardır (action space): İleri doğru gaz vermek, geriye doğru gaz vermek ve hiç gaz vermemek.

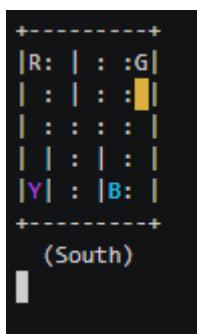


Araba ileri doğru gaz verildiğinde yokuştan dolayı tepeyi aşamaz. Tepe aşması için hızının artırılması gerekmektedir. Burada insanların (makinelerin değil) aklına gelen en uygun strateji önce ileri doğru gaz verip arabanın çıkışını sağlama, sonra geriye gaz verip ters yönde potansiyel enerji kazanmak ve böyle devam ederek tepeyi aşmasını sağlamaktır. Simülatör bize gözlem değeri olarak (observation space) arabanın konumunu ve hızını vermektedir. MountainCar simülatöründe tepe aşılırsa ödül olarak 1 puan verilmekte aşılmazsa 0 puan verilmektedir.

Gym Taxi Simülatörü

Bu simülatörde taksi için dört adet indirme bindirme yeri vardır. Amaç müşteriyi bu noktalardan birinden alıp diğerine bırakmaktır. Taksi eğer engel yoksa sola, sağa, yukarıya, aşağıya gidebilmektedir. Burada yolcu uygun yere bırakılırsa +20 ödül alınmaktadır. Her adımda -1 ceza verilmektedir. (Yolcu alımında herhangi bir puan verilmediği

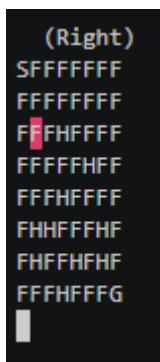
gibi -1 ceza verilmektedir.) Yolcunun olmadığı yerden yolcu almak ya da hedef olmayan durağa yolcu bırakmak -10 ceza puanına yol açmaktadır. Bu simülörde taksiyi idare edenin yapabileceği 5 eylem vardır: Aşağıya gitmek, yukarıya gitmek, sağa gitmek, sola gitmek, yolcu almak ve yolcu bırakmak.



Taksi simülöründe bir tane gözlem değeri (observation space) vardır. O da taksinin hangi noktada olduğunu belirtir.

FrozenLake Simülörü

Bu simülörde donmuş bir gölde bir kişi bir noktadan (S noktası) başka bir noktaya (G noktası) gitmeye çalışmaktadır. Ancak yolu üzerinde delikler (yani çatılar) vardır (H noktaları). Kişi de ancak donmuş yüzeylerden (F noktaları) yürüyebilmektedir.



Eğer kişi hedef noktaya (G noktası) varrsa 1 puan ödül almaktadır. Ancak hedefe varamadığı her harekette 0 puan alır. Kişinin yapabileceği hareketler (action space): Yukarı gitmek, sağa gitmek, aşağı gitmek ve sola gitmektir.

Gym Simülörlerinin Çalıştırılması

Gym simülörlerinin çalıştırılması için önce modüldeki global make isimli fonksiyonun simülör ismi argüman biçiminde verilerek çağrılmazı gereklidir. Örneğin:

```
import gym  
env = gym.make('CartPole-v0')
```

Bundan sonra simülör nesnenin reset metoduyla başlangıç konumuna araylanır. Örneğin:

```
import gym  
env = gym.make('CartPole-v0')  
obs = env.reset()
```

reset işlemi ile araba ve direk rastgele bir konumdan başlatılmaktadır. reset metodu bize geri dönüş değeri olarak "gözlem nesnesini (observation object)" verir. Gözlem nesnesi ortamdaki gözlemlenebilir tüm öğelerin (observation space) oluşturduğu bir numpy dizisidir. Bu dizinin elemanları "CartPole-v0" için sırasıyla arabanın konumu, arabanın hızı, direğin açısı ve direğin açısal hızıdır. Örneğin:

```
In [22]: obs  
Out[22]: array([-0.04124306, -0.0085189 ,  0.01014184, -0.01133194])
```

reset işleminden sonra artık durumlar arasında geçiş oluşturmak gereklidir. Bunun için de faaliyet (action) lazımdır. İşte bu işlem step metoduyla yapılmaktadır. Simülatör sınıflarının step metotları faaliyet alanındaki (action space) bir faaliyeti argüman olarak alıp yeni oluşan çevre durumuna ilişkin bazı değerleri bize vermektedir. CartPole simülöründe step metodu 0 ve 1 olmak üzere iki değeri argüman olarak almaktadır. 0 sola, 1 sağa kuvvet uygulanacağı anlamına gelmektedir. step metodu dörtlü bir demete geri dönmektedir. Bu dörtlü demetin ilk elemanı o faaliyet uygulandıktan sonraki çevrenin gözlem değerleridir. Bu değerler yine CartPole örneğinde bize 4'lü numpy dizisi biçiminde verilmektedir. Demetin ikinci elemanı önerilen ödül miktarıdır. Üçüncü eleman oyunun bitip bitmediğini belirten bir flag niteliğindedir. Örneğin CartPole simülöründe bitiş koşulları iki tanedir. Son parametre bir sözlük nesnesidir. Bize başka bazı bilgileri vermektedir.

Gym simülörleri oluşan değerleri bize vermenin yanı sıra ortamı da görüntüleyebilmektedir. Bunun için ilgiki simülör nesnesinin render metodu kullanılır. Şimdi örnek olarak arabayı bir sağa bir sola hareket ettirelim:

Örneğin:

```
import gym  
  
env = gym.make('CartPole-v1')  
obs = env.reset()  
  
for i in range(1000):  
    obs, reward, done, info = env.step(0)  
    if done:  
        break  
    env.render()  
    obs, reward, done, info = env.step(1)  
    if done:  
        break  
    env.render()  
  
print(i)  
  
env.close()
```

Burada araba bir sağa bir sola hareket ettirilmiştir. Bitiş koşulları sağlandığında döngüden çıkışmıştır.

Pekiyi pekiştirmeli öğrenmede bu simülörle programcı ne yapacaktır? İşte bu tür OpenAI simülörlerinde programcı aslında gözlem değerlerini alıp faaliyeti (action) belirleyecek algoritmik yapıyı kurmaya çalışır. Bu simülörlerin temel amacı bu çalışmaların yapılabilmesi için ortam oluşturmaktır.

Genel olarak Gym nesnelerinin action_space elemanın sample metodunu bize faaliyet uzayındaki rastgele bir değeri vermektedir. Örneğin arabayı rasgele biçimde 500 adım şöyle ilerletebiliriz:

```
import gym  
  
env = gym.make('CartPole-v0')  
obs = env.reset()  
  
for i in range(500):  
    action = env.action_space.sample()  
    obs, reward, done, info = env.step(action)  
    env.render()  
  
env.close()
```

Biz bu simülatörleri aslında pekiştirmeli öğrenme algoritmalarını gerçekleştirmek ve denemek için kullanacağız. Örneğin aşağıda yalnızca direğin açısına göre faaliyeti ayarlayan bir strateji uygulanmıştır. Bu stratejinin başarısı olacağı belliidir:

```
import gym

def get_action(obs, reward):
    return 0 if obs[2] > 0 else 1

env = gym.make('CartPole-v0')
obs = env.reset()

reward = 0
for i in range(500):
    action = get_action(obs, reward)
    obs, reward, done, info = env.step(action)
    env.render()

env.close()
```

Simülatör sınıflarının `action_space` örnek öznitelikleri faaliyet kümesi hakkında bize bilgi vermektedir. Bu örnek özniteligidenden Discrete türünden bir nesne elde edilmişse bu nesne bize faaliyet elemanlarının ayrık değerlerden oluştuğunu söylemektedir. Discrete sınıfının `n` örnek özniteliği bu değerlerin kaç tane olduğunu verir. Benzer biçimde simülatör sınıflarının `oservation_space` örnek özniteliği de gözlemlerin özelliklerini bize vermektedir.

Şimdi MountanCar simülatörünü çalıştırıralım. Örneğin bu simülatörden doğrudan arabaya ileriye doğru gaz verelim:

```
import gym

env = gym.make('MountainCar-v0')
obs = env.reset()
print(obs)
while True:
    obs, reward, done, info = env.step(2)
    if done:
        break
    env.render()
    print(obs)

env.close()
```

Göründüğü gibi arabaya sürekli gaz vermek tepeyi aşması için yeterli olmamaktadır. Burada izlenecek en uygun strateji arabanın hızı sıfıra düşene kadar tepe yönünde gaz vermek, hız sıfıra düşünce ters yönde gaz vermek, böylece hız her sıfıra düştüğünde ters yönde gaz vererek tepeyi aşmasını sağlamaktır:

```
import gym

env = gym.make('MountainCar-v0')
obs = env.reset()
print(obs)
while True:
    if obs[1] > 0:
        action = 2
    else:
        action = 0
    obs, reward, done, info = env.step(action)
    if done:
        break
    env.render()
    print(obs)

env.close()
```

Şimdi de Taksi simülöründe taksiye rastgele işlemler yapalım:

```
import gym

env = gym.make('Taxi-v3')
obs = env.reset()
print(obs)
env.render()

for i in range(10):
    action = env.action_space.sample()
    obs, reward, done, info = env.step(action)
    if done:
        break
    env.render()

env.close()
```

Pekiştirmeli Öğrenmede Kullanılan Algoritmalar

Pekiştirmeli öğrenmede pek çok algoritmik yöntemden ve teknikten faydalana bilabilmektedir. Örneğin yapay sinir ağları da başka tekniklerle birlikte pekiştirmeli öğrenmede kullanılabilmektedir. Biz kursumuzun bu bölümünde en yaygın kullanılan algoritmik yöntemler üzerinde duracağız.

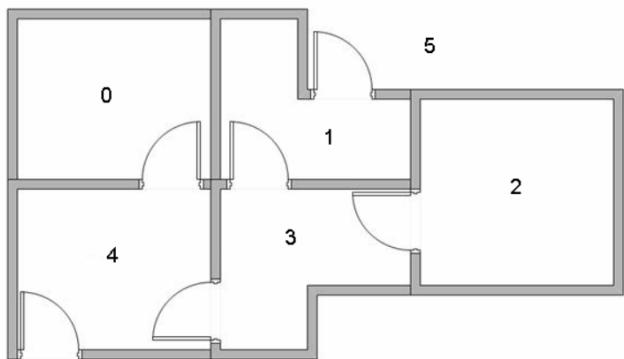
Q-Learning Algoritması

Q-Learning pekiştirmeli öğrenmede en fazla kullanılan algoritmik yöntem ve yaklaşımındır. Bu yaklaşımın sınır ağı versiyonu da vardır. Buna da "Deep Q Learning (DQN)" denilmektedir.

Q Learning algoritması aslında "Markov Zincirleri" denilen algoritmik modelle yakından ilişkilidir. Gerek Markov zincirlerinde gerekse Q-Learning algoritmasında en önemli unsur belli bir noktadaki kararın daha önceki veriler ve durumlar dikkate alınarak verilmesidir.

Q-Learning algoritmasında üç önemli olgu vardır: Durum (State), Eylem (Action) ve Ödül (Reward). Yöntemde öncelikle karşılaşılan problemden durumlar (states) oluşturulmaktadır. Yani içinde bulunulan koşullar durumlara ayırtılmalıdır. Durumlar ayrık olgulardır. Dolayısıyla sürekli olguların ayrık hale getirilmesi gereklidir. Örneğin MountainCar simülasyonunda arabanın içinde bulunduğu durum konum ve hızla belirlenmektedir. Fakat konum da hız da ayrık oldular değildir. O halde bizim bu konum ve hızı aralık temelli bir biçimde ayrık hale getirmemiz gereklidir. Sürekli olduların ayrık hale getirilmesiyle elde edilen durumların sayısı çok yüksek olabilmektedir. Uygulamacı durum sayılarını kendi olanaklarına göre makul bir biçimde belirlemelidir. (Pekiştirmeli öğrenmede oluşturulan durumların sayısı fazla olabileceğinden dolayı bu biçimde yapay öğrenme yöntemleri de yüksek bir bilgisayar zamanı ve kaynağının kullanılmasına yol açmaktadır.) Tabii bazen durumlar zaten ayrıktır. Bu durumda onların ayrık hale getirilmesine gerek kalmaz. Örneğin Taxi simülöründe ya da FrozenLake simülöründe durum zaten ayrıktır.

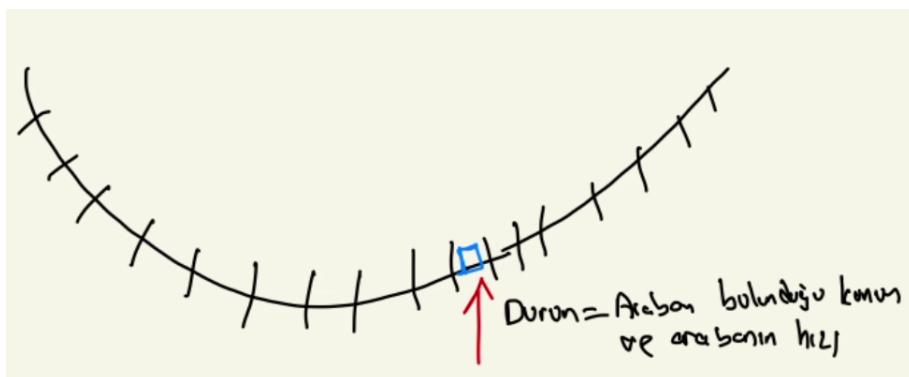
Örneğin bir labirentte odaların her biri bir durum olarak modellenebilir. Böylece labirentte kişinin bulunduğu yer zaten ayrık bir durum oluşturur.



(Şekil <http://mnemstudio.org/path-finding-q-learning-tutorial.htm> sitesinden alınmıştır.)

Burada durumlar 0, 1, 2, 3, 4, 5'ten oluşmaktadır. Bu labirentten amaç 5 noktasına ulaşmaktadır.

Süphesiz durum (state) tek değişkenli bir bilgi olmak zorunda değildir. Çünkü belli bir durumu ifade etmek için birden fazla değişkene ihtiyaç duyulabilmektedir. Örneğin MountainCar simülasyonunda durum yalnızca arabanın x eksenindeki yeri değildir. Arabanın o andaki hızı da durumun bir parçasıdır. Bu örnekte x eksenini 20 parça hızı da 20 parçaaya ayırsak toplamda 20x20'lük bir durum matrisi elde ederiz.



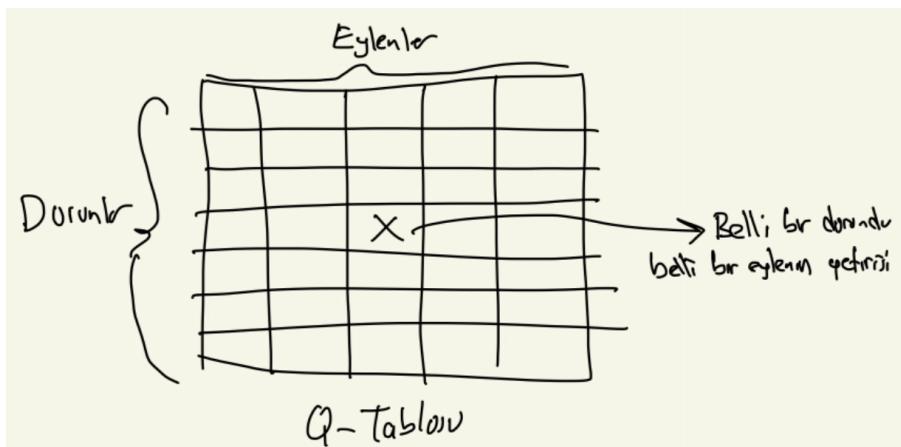
Göründüğü gibi MountainCar simülasyonunda arabanın belli bir konumda olması durum bilgisi için yetmemektedir. Araba aynı konumda bulunduğu halde o sırada farklı hızlarda olabilir. O halde buradaki durumlar konum ve hızı barındırmalıdır. Biz konum ve hızı 20 parçaya bölgerek ayrı hale getirdiğimizde toplam 20x20'lük bir durum matrisi elde ederiz. Tabii eğer problemde durumsal bilgi 2 değişkenli değil 3 değişkenli olsaydı bizim durum matrisimiz de 3 boyutlu olacaktı. O halde genel olarak n değişkenli bir durumsal problemde durum matrisi de n boyutlu olacaktır.

Q-Learning algoritmada ikinci önemli nokta belli bir durumdaki eylemlerin (actions) neler olduğunu söylemektedir. Yani belli bir durumda olan etmen (agent) o durumu değiştirmek için hangi geçerli eylemleri yapabilecektir? Eylemlerin sayısı ve genel olarak listesi çok fazla olabilir. Üstelik de eylemler içinde bulunulan duruma göre farklılaşabilir. Bu durumda programcının eylemleri uygun ve makul bir veri yapısıyla ifade etmesi gereklidir. Yukarıdaki labirent örneğinde eylem bir odadan (yani durumdan) diğerine geçmektir. Ancak her odadan diğerine bir geçişin olmadığı görülmektedir. Tabii bu tür gerçek problemlerde hangi odalardan hangi odalara geçişin olduğu başlangıçta biliniyor olabilir ya da süreç sırasında öğreniliyor olabilir. Yukarıdaki problemde labirentin durumunun önceden bilindiğini varsayırsak hangi durumlarda hangi eylemlerin yapılabileceğini bir sözlük nesnesiyle ya da matrisle temsil edebiliriz.

Pekiştirmeli öğrenmede bir ödül mekanizmasının olması gerekligidenden bahsetmiştik. Çünkü öğrenme aslında ödülü maksimize etmeye çalışmakla gerçekleşmektedir. Ödül mekanizması ortamı iyi bilenler tarafından makul bir biçimde oluşturulabilir. Ancak bazen ortam problem çözücü tarafından da tam olarak bilinmeyebilir. Bu durumda programcı açık birtakım durumlar için ödül koyabilir. Örneğin yukarıdaki labirentte ödül yalnızca labirentten çıkış için veriliyor olabilir. Yani herhangi iki oda arasındaki geçiş eyleminden 0 ödül verilirken bir odadan dışarıya çıkış için örneğin 100'lük ödül verilebilir.

Q Learning algoritmasında ismine "Q-Tablosu (Q-Table)" denilen bir tablo oluşturulmaktadır. Bu tabloda satırlar durumları (states) sütunlar ise eylemleri (actions) belirtir. Q tabloları sözlükler ya da matrisler biçiminde

oluşturulabilmektedir. Çünkü tablodan amaç belli durumda iken hangi eylemlerin uygun olacağına ilişkin bir puanlama oluşturmaktır. Yani algoritmada etmen belli bir durumdayken tabloya başvurarak o durumda hangi eylemin yapılması gerekiğine karar verir. Q-Tablosu başlangıçta 0'larla doldurulur.



Tabii Q-Tablosunu oluşturan durumların da aslında değişken sayısına bağlı olarak matrisel bir biçimde olabileceğine dikkat ediniz. Örneğin MountainCar simülasyonunda durum aslında 20×20 'lik bir matristir. Bu durumda MountainCar simülasyonundaki eylemler de 3 tane olduğuna göre bu simülasyon için oluşturacak Q-Tablosu aslında $20 \times 20 \times 3$ 'luk bir matris durumunda olacaktır. Tabii yukarıda da belirttiğimiz gibi Q-Tablosu bir matris biçiminde değil bir sözlük biçiminde de oluşturulabilir. Örneğin yukarıdaki labirent için Q-Tablosu aşağıdaki gibi bir sözlük biçiminde oluşturulabilir:

```
maze = {0: {4: 0}, 1: {3: 0, 5: 0}, 2: {3: 0}, 3: {1: 0, 2: 0, 4: 0}, 4: {0: 0, 3: 0, 5: 0}, 5: {5: 0}}
```

Burada bir sözlüğün her elemanı sözlük yapılmıştır. Bu tür problemlerde matris yerine sözlüklerin tercih edilmesinin en önemli nedeni her durumda yapılacak eylemlerin farklı olmasıdır.

Aynı labirent için Q tablosu aşağıdaki gibi bir matrisle de oluşturulabilir:

```
maze = np.zeros((6, 6))
```

		Eylemler					
		0	1	2	3	4	5
Durumlar	0	0	0	0	0	0	0
	1	0	0	0	0	0	0
	2	0	0	0	0	0	0
	3	0	0	0	0	0	0
	4	0	0	0	0	0	0
	5	0	0	0	0	0	0

Q Tablosu

Q tablosu öğrenme süreci sırasında her eylemden sonra güncellenen bir tablodur. Öğrenme gerçekleştikçe tablo gitgide güncellenir. Öğrenme süreci bittiğinde tablo hangi durumlarda hangi eylemlerin en yararlı olduğunu belirten bir yapıya kavuşur.

Q-Learning algoritmasının incelikleri yavaş yavaş ele alınacaktır. Ancak algoritma baştan rastgele bir durumdan başlatılır. İşin başında mademki etmen (agent) hiçbir şey bilmemektedir. O halde rastgele bir eylemde bulunur. İşte etmen belli s durumunda belli bir a hareketini yarlığında matrisin $[s, a]$ elemanı bir formülle güncellenmektedir. Böylece bir durumda bir eylem yapıldığında gittikçe tablo elemanları güncellenmiş olacaktır. Bu eylemler arttıkça tablo elemanları daha uygun değerlerle doldurulur. En sonunda Q-Tablosu tatminkar değerlerle doldurulduğunda artık

artık etmen belli bir durumda bırakıldığında ödülü maksimize etmek için ne yapması gerektiğini öğrenmiş olacaktır. Burada şüphesiz iki nokta önemlidir:

- 1) Tablo elemanı hangi formüle göre güncellenmektedir?
- 2) Belli bir durumda etmen hangi eylemi gerçekleştirecektir?

Tablo elemanlarının güncellenmesi aşağıdaki formüle göre yapılmaktadır:

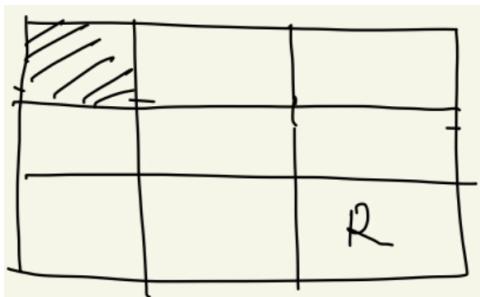
$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\underbrace{r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a)}_{\substack{\text{learned value} \\ \text{reward} + \text{discount factor} \cdot \text{estimate of optimal future value}}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Bu formülde iki önemli sabit vardır: Alfa (learning rate) ve gama (discount). Bu faktörler 0 ile 1 arasında bir değer olarak alınabilmektedir ve daha sonra bunların etkisinden bahsedilecektir. Buradaki $t+1$ alt indisler "sonraki" anlamına gelmektedir. Örneğin r_{t+1} eski durumdan yeni duruma geçiş için verilecek ödülü anlatmaktadır. (yani s_t 'den s_{t+1} 'e geçiş için verilecek ödül.) Formülün max ile ilgili kısmı şu anlamda gelmektedir: s_t durumundan s_{t+1} durumuna geçiş planlandığı zaman kendimizin s_{t+1} durumunda olduğunu varsayırsak oradan başka bir duruma geçiş için elde edilecek maksimum kalite (ya da memnuniyet diyebiliriz) bulunmak istenmektedir. Çünkü Q matrisi aslında nihayetinde her durumdan mümkün her eylemin amaç doğrultusundaki kalitesini (Quality) belirtir. Yani Q matrisinde $[s, a]$ elemanı s durumundayken a eylemi gerçekleştirildiğinde bundan ne kadar memnuniyet duyulacağını belirtmektedir.

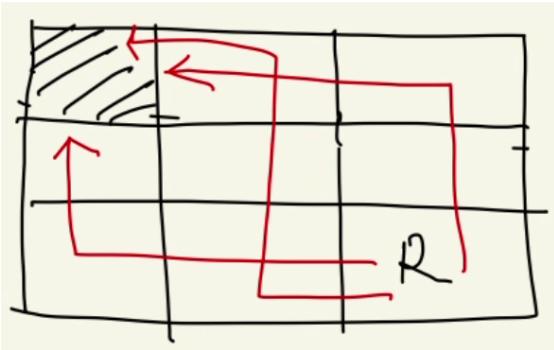
Peki her eylemden sonra Q -Tablosu güncelleneceğine göre öğrenme süreci toplamda nasıl olacaktır? İşte öğrenme süreci kabaca şöyle bir algoritmayla gerçekleştiriliyor:

- 1) Öğrenme süreci bölüm bölüm (episode) yapılmaktadır. Her bölüm rastgele bir durumdan başlatılır.
- 2) Her durumda en iyi eylem gerçekleştirilmeye çalışılır. Bunun için alternatifler arasındaki en iyi Q değeri seçilir. Tabii bu Q değeri yalnızca tablodan değil yine yukarıdaki formülden elde edilmelidir.
- 3) İşin başında Q tablosu tamamen boş olduğuna göre rastgele bir eylem seçilebilir. Tabloda duruma ilişkin satırın tüm elemanları 0 ise rastgele bir eylemin seçilmesi uygun olabilmektedir. Bazen alternatif Q değerleri aynı da olabilmektedir. Bu durumda yine rastgele bir eylem seçilebilir.

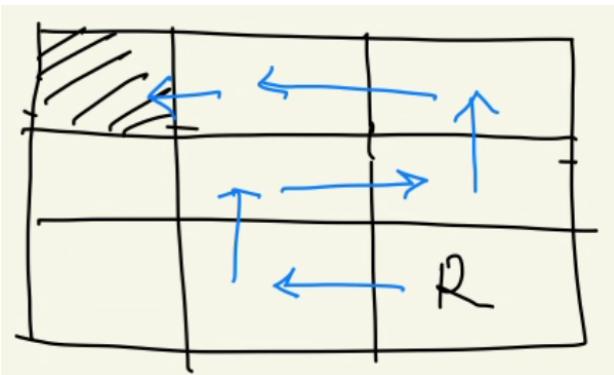
Şimdi çok basit bir problem üzerinde düşünelim. Etmenimiz bir robot olsun. Bu robot 3X3'lük bir matriste sola, sağa, yukarı aşağı gidebilsin. Ama çapraz olarak gidemesin. Burada robotun amacı belli bir hücreye hızlı (en az yol kat edecek biçimde) varmak olsun:



Burada taralı hücre hedefi belirtiyor olsun. Göründüğü gibi hedefe varmanın en kısa yolu 4 adımındır. Bu yol farklı biçimlerde oluşturulabilmektedir. Örneğin:



Ancak aşağıdaki gibi bir rota hedefe daha yavaş varmaktadır:



Böyle bir problemi rastgele hareketlerle çözmeye çalışabiliriz. Her hedefe varlığında yolu hesaplayabiliriz. Ancak bu bir öğrenmeye yol açmaz. Yani robot her defasında rastgele hareketler yapar. Halbuki istediğimiz şey robotu eğitmek ondan sonra onu herhangi bir hücreye koyduğumuzda en kısa yolu bulmasını sağlamaktır. Öğrenmenin "davranışta kalıcı bir değişiklik oluşturan süreçler" biçiminde tanımlandığını anımsayınız. Rastgele hareketler daha sonra kullanılabilecek bir davranış değişikliğine yol açmamaktadır.

Burada öğretici olarak robota yaptığı hareketlerden ödül (ya da ceza) vermemiz gereklidir. Peki nasıl bir ödül mekanizması oluşturulabilir? Akla gelen makul ödül mekanizması hedefe yaklaştıkça artan, uzaklaştıkça azalan ya da negatif hale gelen bir mekanizma olabilir. Tabii bunun için bizim robotun durumuyla hedef arasındaki ilişkiyi biliyor olmamız gereklidir. Ancak her uygulamada buna benzer bir bilgiyi öğretici bilmiyor olabilir. Bu nedenle ödül sistemi genellikle daha basit bir biçimde oluşturulmaktadır. Örneğin burada en basit bir ödül mekanizması "hedefe varan hareketin 1, hedefe varmayan hareketin 0" olarak ödüllendirilmesidir. (Tabii 0 ödül aslında ödül vermemek negatif ödül ise ceza vermek anlamındadır.)

Bu problemi Q-Learning algoritmasıyla çözebilme için yukarıda da belirtildiği gibi bir Q-Tablosunun oluşturulması gereklidir. Q-Tablosu satırlarda durumların sütunlarda eylemlerin (actions) olduğu bir tablodur. Tipik olarak matrisel bir biçimde oluşturulabilen gibi sözlük gibi veri yapılarıyla da oluşturulabilmektedir. Q-Tablosunu oluşturabilmek için ise şüphesiz bizim önce problemi durumlar (states) ve eylemler (actions) biçiminde tanımlamamız gereklidir. Eylemler ise robotun hareketleridir. Biz her hücreyi bir durumla temsil edebiliriz:

S0	S1	S2
S3	S4	S5
S6	S7	R S8

Robotun bir durumdayken başka bir duruma geçmesi biçiminde tanımlanan eylemleri de dört tanedir: Yukarı, Sağ, Aşağı, Sola gitmek. Tabii bu problemde robotun her durumda her eylemi yapamayacağı da görülmektedir. Örneğin robot S8 durumunda sağa ve aşağı gidememektedir. Bu gidemeyiş programlamada bir bir biçimde oluşturulmalıdır. Bazen mümkün hareketler sANKI mümkün olmuş gibi düşünülüp ona yüksek cezalar da verilebilmektedir. Bu durumda Q-Tablosu aşağıdaki gibi ifade edilebilir:

	Yukarı	Sağ	Aşağı	Sola
S0				
S1				
S2				
S3				
S4				
S5				
S6				
S7				
S8				

Q - Tablosu

Q-Learning algoritmasında tablodaki hücreleri güncellerken kullanılan iki parametrik değerini olduğunu görüyorsunuz. Bunlar alfa ve gama değerleridir. Alfa değerine "öğrenme hızı (learning rate)" gama değerine ise "indirim (discount)" denilmektedir. Bu iki parametre de 0 ile 1 arasında bir değer almaktadır. Biz bu örneğimizde alfa değerini 1, gama değerini 0.90 olarak alalım:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\underbrace{r_{t+1} + \gamma \max_a Q(s_{t+1}, a)}_{\substack{\text{learned value} \\ \text{reward} \\ \text{discount factor} \\ \text{estimate of optimal future value}}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Q-Learning algoritmasında oluşturulan Q tablosu baştan 0'larla doldurulmuştur. Q-Tablosu yukarıda da belirtildiği gibi aslında hangi durumdayken hangi eylemlerin daha iyi (kaliteli) olduğunu göstermektedir. İşin başında biz hangi eylemlerin hangi iyilikte olduğunu bilememekteyiz. Bu nedenle Q-Tablosu da 0'larla doludur.

Şimdi robotumuzun S1'de olduğunu ve S0'a gitmek istedğini düşünelim. Eğer robot S1'de ise ve S0'a gitmek istiyorsa bu eylem sonucunda bizim tablonun S1-Sola hücresini güncellememiz gereklidir. Bu güncelleme yukarıdaki formüle göre yapılacaktır. Alfa için 1, gama için 0.90 değerlerini uygun görmüştük. Formüldeki r_{t+1} S1'den Sola gidildiğinde elde edilecek ödülü belirtmektedir. Bu ödülün 1 olduğunu belirlemiştik. Formüldeki $\max Q(s_{t+1}, a)$ bulunulan

durumdan belli durumlara gidildiğiindeki o durumlara özgü en yüksek tablo değerini belirtir. Yani örneğin biz S1'deyken sağa gitmek istediğimizde kendimizi bu yeni S2'de olduğumuzu düşünerek burada yapabielceğimiz en iyi eylemi bulmak istemekteyiz.

Biz S1'de olalım ve S0'a gitmek isteyelim. Bu durumda tabloda güncelleyeceğimiz hücre Q(s1, sola) hücresi olacaktır. Yukarıdaki formüle göre güncellememizi yapalım:

$$Q(s1, \text{sola}) = Q(s1, \text{sola}) + 1 * (1 + 0.90 * 0 - Q(s1, \text{sola}))$$

Buradan 1 değeri elde edilecektir. S1'den sola gidildiğinde varılacak yer hedef olduğu için formülün max elemanından 0 elde edildiğini varsayıyoruz. Bu durumda Q-Tablosu şu hale gelecektir:

	Kısa	Sağ	Aşağı	Sola
S0	0	0	0	0
S1	0	0	0	1
S2	0	0	0	0
S3	0	0	0	0
S4	0	0	0	0
S5	0	0	0	0
S6	0	0	0	0
S7	0	0	0	0
S8	0	0	0	0

(Q - Tablosu)

Şimdi robotumuzun S2'de olduğunu varsayıyalım. S2'den sola gitmek isteyelim. Bu durumda biz tablomuzun Q(s2, sola) hücresini güncelleriz. Formülü uygulayalım:

$$Q(s2, \text{sola}) = Q(s2, \text{sola}) + 1 * (0 + 0.90 * 1 - Q(s1, \text{sağa}))$$

Buradan 0.90 değeri elde edilmektedir. Algoritmamızın kritik nokta formüldedek max değeridir. Biz S2'den sola gitmek istedik. Hedef olarak varacağımız yer S1'dir. İşte bu max değeri robotun S1'de olduğu fikriyle oradaki eylemlere ilişkin en büyük tablo değerini belirtir. Bu da 1'dir.

	Kısa	Sağ	Aşağı	Sola
S0	0	0	0	0
S1	0	0	0	1
S2	0	0	0	0
S3	0	0	0	0
S4	0	0	0	0
S5	0	0	0	0
S6	0	0	0	0
S7	0	0	0	0
S8	0	0	0	0

(Q - Tablosu)

güncellenecek hücre

S1'den S2'e gidildiğinde
S1'deki tablo değerinden en
büyük

Q-Tablosunun yeni durumu şöyledir:

	Vulcan	Sağrı	Aşağı	Solu
S0	0	0	0	0
S1	0	0	0	1
S2	0	0	0	0.90
S3	0	0	0	0
S4	0	0	0	0
S5	0	0	0	0
S6	0	0	0	0
S7	0	0	0	0
S8	0	0	0	0

Q - Tablosu

Şimdi robotun S1'de olduğunu ve S2'ye gitmek istediğini düşünelim. Şüphesiz bu iyi bir haraket değildir. Fakat Q(s1, sağa) hücresinin formüle göre güncelleylelim:

$$Q(s1, \text{sağa}) = 0 + 1 * (0 + 0.90 * 0.90 - 0)$$

Buradan 0.81 değeri elde edilir. Buradaki max elemanın 0.90 değerinde olduğuna dikkat ediniz. Çünkü robot s1'den s2'ye girmek istediğiinde s2'de olduğu fikriyle oradaki eylemlerin en yüksek tablo değeri 0.90'dır:

	Vulcan	Sağrı	Aşağı	Solu
S0	0	0	0	0
S1	0	0	0	1
S2	0	0	0	0.90
S3	0	0	0	0
S4	0	0	0	0
S5	0	0	0	0
S6	0	0	0	0
S7	0	0	0	0
S8	0	0	0	0

Q - Tablosu

Robotun S2'de olduğu
fikriyle orada yapabileceğii,
eylemler en yüksek değer

güncellenecek hücre

Q-Tablosunun yeni hali şöyledir:

	Yukarı	Sağ	Aşağı	Sol
S0	0	0	0	0
S1	0	0.81	0	1
S2	0	0	0	0.90
S3	0	0	0	0
S4	0	0	0	0
S5	0	0	0	0
S6	0	0	0	0
S7	0	0	0	0
S8	0	0	0	0

Q - Tablosu

Şimdi robotumuzun S5'te olduğunu varsayıyalım ve robotumuz yukarıya gitmek istesi. Bu durumda Q-Tablosundaki Q(s5, yukarıya) hücresi güncellenecektir. Formülü uygulayalımlı:

$$Q(s5, \text{yukarıya}) = 0 + 1 * (0 + 0.90 * 0.90 - 0)$$

Buradan elde edilecek değer 0.81'dır. Çünkü S5'ten yukarıya gittiğimizde S2 durumuna geçeriz. Robotumuzun S2'de olduğu fikriyle oradan elde edebileceğim en yüksek değer (s2 satırından) 0.90'dır. O halde tablomuzun yeni durumu şöyledir:

	Yukarı	Sağ	Aşağı	Sol
S0	0	0	0	0
S1	0	0.81	0	1
S2	0	0	0	0.90
S3	0	0	0	0
S4	0	0	0	0
S5	0.81	0	0	0
S6	0	0	0	0
S7	0	0	0	0
S8	0	0	0	0

Q - Tablosu

Şimdi de robotumuzun S8'de olduğunu varsayıyalım ve yukarıya gitmek istediğimi düşünelim. Bu durumda Q-Tablosunun Q(s8, yukarıya) elemanı güncellenecektir. Formülü uygulayalımlı:

$$Q(s8, \text{yukarıya}) = 0 + 1 * (0 + 0.90 * 0.81 - 0)$$

Buradan elde edilecek değer 0.729'dur. Tablomuzun yeni hali şöyledir:

	Yukarı	Sağ	Aşağı	Sola
S8	0	0	0	0
S1	0	0.81	0	1
S2	0	0	0	0.90
S3	0	0	0	0
S4	0	0	0	0
S5	0.81	0	0	0
S6	0	0	0	0
S7	0	0	0	0
S8	0.729	0	0	0

Q - Tablosu

Şimdi tablomuzu biraz daha doldurmaya çalışalım. Örneğin robotumuzun S8'de olduğunu ve sola gitmek istedığını varsayıyalım:

$$Q(s8, sola) = 0 + 1 * (0 + 0.90 * 0 - 0)$$

Bu aradan 0 değeri elde edilecektir. Aslında S8'den sola gitmek izlenebilecek bir yoldur. Ancak tablo henüz boş olduğu için bu değer 0 olarak elde edilir. Burada biz başlangıçta boş olan tablonun ödül almadıkça anlamlı değerlerle doldurulamadığını görüyoruz. Tablonun doldurulma stratejisi Q-Learning algoritmasında önemli bir konudur.

Şimdi robotumuzun S3'te olduğunu varsayıyalım ve yukarıya gitmek isteyelim:

$$Q(s3, yukarıya) = 0 + 1 * (1 + 0.90 * 0 - 0)$$

Burada 1 değeri elde edilecektir. Tablomuzun hali şöyledir:

	Yukarı	Sağ	Aşağı	Sola
S8	0	0	0	0
S1	0	0.81	0	1
S2	0	0	0	0.90
S3	1	0	0	0
S4	0	0	0	0
S5	0.81	0	0	0
S6	0	0	0	0
S7	0	0	0	0
S8	0.729	0	0	0

Q - Tablosu

Şimdi sırasıyla birkaç elemanı güncelleylelim:

$$Q(s4, yukarıya) = 0 + 1 * (0 + 0.90 * 1 - 0) = 0.90$$

$$Q(s6, yukarıya) = 0 + 1 * (0 + 0.90 * 1 - 0) = 0.90$$

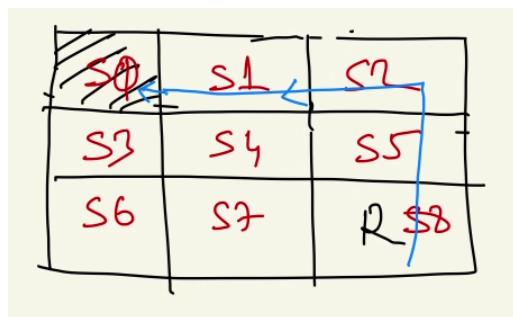
$$Q(s7, sağa) = 0 + 1 * (0 + 0.90 * 0.729 - 0) = 0.656$$

$$Q(s7, \text{yukarıya}) = 0 + 1 * (0 + 0.90 * 0.90 - 0) = 0.81$$

	Kukarı	Sağrı	Aşağı	Sola
S0	0	0	0	0
S1	0	0.81	0	1
S2	0	0	0	0.90
S3	1	0	0	0
S4	0.90	0	0	0
S5	0.81	0	0	0
S6	0.90	0	0	0
S7	0.81	0.656	0	0
S8	0.729	0	0	0

Q - Tablosu

Böylesi işlemlerle tablonun doldurulduğunu ve hücrelerinde güncellendiğini düşünelim. En son varacağımız nokta ne olacaktır? İşte en sonunda biz robotumuzu herhangi bir hücreye bıraktığımızda robotumuz bu tabloya bakarak hedefe (S0 hedef) en kısa sürede ulaşacaktır. Doldurulmuş ve güncellenmiş bir Q-Tablosunda etmenin hedefe gitmesi şöyle gerçekleşir: Etmen hangi konumdaysa her zaman o konumdaki en yüksek tablo değerine ilişkin eylemi gerçekleştirir. Sonra yeni durumda da aynı işlemleri yapar. Örneğin yukarıda zayıf doldurulmuş tabloda robotun S8'de olduğunu düşünelim. S8 satırındaki en yüksek puanlı eylem yukarı gitmektir. Robot yukarı gidince S5 durumuna gelmiş olur. S5 satırındaki en yüksek puanlı eylem yukarı gitmektir. Böylece robot S2'ye gelir. S2 satırındaki en yüksek puanlı eylem ise sola gitmektir. Böylece robot S1'ye gelmiş olur. S1 satırındaki en yüksek puanlı eylem ise sola gitmektir. Böylece robot hedefe ulaşmış olur.



Formüldeki indirim (discount) parametresi ne anlam ifade ettiğine yönelik bir ip ucu elde etmiş olabilirsiniz. Bu indirim parametresi hedeften uzaklaşıkça puan düşürmeye etkildir. 1'den küçük olursa puan o kadar çabuk düşürülür.

Gym Taxi-V3 Simülasyonu

Bu simülasyonda 5×5 'lik bir alanda bir taksi bulunmaktadır. Her biri ayrı bir renkle gösterilen 4 ayrı indirme-bindirme durağı vardır. Burada öğrenilecek iş taksinin yolcuya belirlenen bir duraktan alıp başka bir belirlenen durağa bırakmasıdır. Durakların yerleri hep sabit olmakla birlikte yolcu bu duraklardan herhangi birinde bulunuyor olabilir ve bu duraklardan herhangi birine gitmek istiyor olabilir. Alan içerisindeki matriste duvarlar da vardır. Dolayısıyla duvara çarptığında diğer tarafa geçilememektedir. Çizdirilen şekilde Mavi her zaman yolcunun alınacağı durağı, mor ise yolcunun bırakılacağı durağı belirtmektedir. Arabanın içerisinde yolcu yoksa araba kırmızı, yolcu varsa yeşil olarak gösterilmektedir. Örneğin:

```
+-----+
|R: | : : G| |
| : | : : |
| : | : : |
| : | : : |
|Y| : |B: |
+-----+
```

Burada yolcu sol üst duraktan alınacak, sol alt durağa bırakılacaktır.

Simülasyonu başlatmak için gym.make fonksiyonu ile simülatör ismi verilerek simülatör nesnesi elde edilir. Nesne reset edilerek başlangıç konumu oluşturulur. Örneğin:

```
import gym

env = gym.make('Taxi-v3')
obs = env.reset()
env.render()
```

```
+-----+
|R: | : : G| |
| : | : : |
| : | : : |
| : | : : |
|Y| : |B: |
+-----+
```

Simülasyonda toplam 6 tane eylem (action) vardır:

```
In [103]: env.action_space.n
Out[103]: 6
```

Bu eylemler taksinin yukarı, sola, sağa, aşağıya gitmesi, yolcu alma ve yolcu bırakma eylemleridir. Toplam durum sayısı (observation space) 500 tanedir. Örneğin:

```
In [105]: env.observation_space.n
Out[105]: 500
```

Bu sayı ilk bakışta size biraz tuhaf gelebilir. Gerçekten de bir t anında burada farklı 500 durum vardır. Taksi 5 X 5'lik alanda 25 farklı yerden birinde olabilir. Yolcu da 5 farklı yerde bulunabilir. (4 tane durak ve taksinin içi). Nihayetinde yolcunun bırakılacağı 4 farklı yer bulunmaktadır. Bunların çarpımı 500'dür. 5X5'lik matristeki hücre numaralandırmaları ekran koordinat sisteminde olduğu gibi yapılmıştır. Yani Sol-Üst köşe (0, 0) koordinatını belirtmektedir.

O anda bulunulan durum observation nesnesinden tek bir sayı olarak söyle elde edilmektedir: Yolcunun bırakılacağı yer + yolcunun konumu * 4 + arabanın sütun numarası * 20 + arabanın satır numarası * 100. Yolcunun bırakılacağı yer RGYB durumlarına göre 0, 1, 2, 3 ile, yolcunun konumu ise RGYBl'ye göre 0, 1, 2, 3, 4 biçiminde kodlanmıştır. I (4) yolcunun taksinin içinde olduğu anlamına gelmektedir. Taksinin bulunduğu konum satır ve sütun olarak kodlanmıştır. Bu durumda taksi 25 yerden birinde olabilir.

Örneğin yukarıdaki reset durumunun observation değeri şöyledir:

```
In [8]: obs
Out[8]: 453
```

Buradaki 453 değeri söyle elde edilmiştir:

$1 (\text{yolcunun bırakılacağı yer}) + 4 * 3 (\text{yolcunun konumu}) + 20 * 2 (\text{arabanın sütun numarası}) + 100 * 4 (\text{arabanın satır numarası}) = 453$. simülatör nesnesinin encode isimli metodu sırasıyla satır no, sütun no, yolcunun konumu ve yolcunun bırakılacağı koordinatları parametre olarak alıp bize simülasyonun konumunu tek bir değer olarak verir.

Örneğin:

In [23]: env.encode(4, 2, 3, 1)

Out[23]: 453

Bunun tersini yapan decode isimli metot da vardır. decode metodu dolaşılabilir bir nesne vermektedir. Örneğin:

In [26]: list(env.decode(453))

Out[26]: [4, 2, 3, 1]

Her step işleminde simülatör tarafaından verilen ödül mekanizması şöyledir:

- Yolcu uygun yere bırakılırsa +20 ödül alınmaktadır.
- Her adımda -1 ceza verilmektedir.
- Yolcunun olmadığı yerden yolcu almak ya da hedef olmayan durağa yere yolcu bırakmak -10 ceza puanına yol açmaktadır.
- Yolcunun doğru yerden almışında herhangi bir puan verilmediği gibi yine -1 ceza verilmektedir.

Belli bir durumda yapılabilecek 6 eylemin sayısal temsilleri şöyledir:

0: Aşağısı

1: Yukarı

2: Sağ

3: Sola

4: Yolcu alma

5: Yolcu bırakma

Bir eylem yine step metoduyla gerçekleştirilmektedir. Bu metot yine bize 4'lü bir demet verir. reset ve action metodlarından elde ettiğimiz observation nesnesi o andaki ortamın durumunu vermektedir. Örneğin:

In [27]: obs, reward, done, info = env.step(1)

In [28]: obs

Out[28]: 353

Burada araba yukarı yönlü hareket ettirilmiştir. Yeni konumu 353'tür. Bu değeri decode edip yeni çizimi yapalım:

In [30]: list(env.decode(obs))

Out[30]: [3, 2, 3, 1]

In [31]: env.render()

R:		:	:G
	:		:
	:	:	:
	:	:	:
	:	B:	
(North)			

Şimdi arabaya rastgele 10 hareket yaptıran örnek bir program oluşturalım:

```
import gym
```

```

env = gym.make('Taxi-v3')
obs = env.reset()
env.render()

total_rewards = 0
for i in range(10):
    action = env.action_space.sample()
    obs, reward, done, _ = env.step(action)
    total_rewards += reward
    env.render()

print(total_rewards)

```

Şimdi de biz Q-Learning algoritmasını kullanarak taksinin yolcuyu alıp bırakma işini öğrenmesini sağlayalım. Bu problemdeki toplam durum sayısı 500'dür. Toplam eylemlerin sayısı da 6'dır. Bu durumda bizm Q tablomuz 500 X 6'lık bir matris olmalıdır. Eğitim işlemi aşağıdaki gibi yapılabilir:

```

NEPOCHS = 100000
MAX_ITER = 1000
EPSILON = 0.1
DISCOUNT = 0.6
LEARNING_RATE = 0.1

def train(env, nepochs=NEPOCHS, max_iter=MAX_ITER, epsilon=EPSILON,
learning_rate=LEARNING_RATE, discount=DISCOUNT):
    q_table = np.zeros((env.observation_space.n, env.action_space.n))

    for i in range(nepochs):
        obs = env.reset()
        for j in range(max_iter):
            if np.random.uniform(0, 1) < epsilon:
                action = np.random.choice(env.action_space.n)
            else:
                action = np.argmax(q_table[obs])

            next_obs, reward, done, _ = env.step(action)
            if done:
                break
            q_table[obs, action] = q_table[obs, action] + learning_rate * (
                reward + discount * np.max(q_table[next_obs]) - q_table[obs, action])
            obs = next_obs

    return q_table

```

Buradaki train fonksiyonu Q-Tablosunu oluşturarak oluşturulmuş olan tabloyla geri döner. Fonksiyonda her biri en fazla MAX_ITER (1000) kadar hareket yapan NEPOCHS (1000000) kadar deneme yapılmıştır. Yani bir deneme ortam reset edilerek rastgele bir yerden başlatılır, işlem bitene kadar ya da en fazla MAX_ITER kadar devam ettirilir. Bu denemeler de toplamda NEPOCHS kadar yinelenmektedir.

Q-Learning yönteminde tabloyu doldurmak için Q-Tablosundaki en iyi eylemlerin yanı sıra arada rastgele birtakım eylemlerin de yapılması gerekmektedir. Arada rastgele eylemlerin yapılmasına "keşif" (exploration), tabloya bakarak en iyi Q değerine ilişkin eylemi uygulamaya ise "işletme" (exploitation) denilmektedir. İşte ne zaman keşif (exploration) ne zaman işletme (exploitation) uygulanacağına yönelik "keşif stratejileri (exploration strategy)" vardır. En basit keşif stratejilerinden biri "epsilon greedy" stratejisidir. Burada çoğu kez tablodaki en iyi Q değerine ilişkin eylem seçilir. Ancak arada bir (epsilon olasılıkla) rastgele bir eylem seçilir. Biz de örneğimizde keşif stratejisi olarak "epsilon greedy" yöntemini kullandık:

```

if np.random.uniform(0, 1) < epsilon:
    action = np.random.choice(env.action_space.n)
else:
    action = np.argmax(q_table[obs])

```

Burada np.random.uniform fonksiyonu 0 ile 1 arasında rastgele noktalı bir değer üretmektedir. Bu değer epsilon değerinden küçükse rastgele bir eylem seçilmiştir. Epsilon için default değerin 0.1 olduğuna dikkat ediniz. O halde eğitim sırasında her 10 eylemin ortalama 1 tanesi rastgele alınmaktadır.

train fonksiyonunda Q-Tablosunun yukarı vermiş olduğumuz formüle göre güncellendiğini görüyorsunuz:

```
q_table[obs, action] = q_table[obs, action] + learning_rate * (reward + discount *  
np.max(q_table[next_obs]) - q_table[obs, action])
```

Burada obs ve next_obs önceki sonraki durumları belirtmektedir. Yani eylem uygulanıp yeni bir durum elde edilip eski değer güncellenmiştir.

Aşağıda oluşturulmuş olan Q-Tablosunu kullanarak problemi çözen fonksiyonu görüyorsunuz:

```
def run_episode(env, q_table):  
    count = 0  
    obs = env.reset()  
    env.render()  
  
    done = False  
    while not done:  
        action = np.argmax(q_table[obs])  
        obs, reward, done, _ = env.step(action)  
        env.render()  
        count += 1  
  
    return count
```

Göründüğü gibi bu fonksiyon doldurulmuş Q-Tablosundaki en yüksek değerler eşliğinde belli bir durumdaki en iyi eylemlemi belirlemektedir. Şimdi aşağıdaki gibi bir test işlemi yapalım:

```
env = gym.make('Taxi-v3')  
q_table = train(env)  
  
result = run_episode(env, q_table)  
print(result)
```

Çıkan sonuç aşağıdaki gibidir:

R:	:	:	:G
Y:	:	:	
:	:	:	
:	:	:	
Y	:	:	B:

R:	:	:	:G
Y:	:	:	
:	:	:	
:	:	:	
Y	:	:	B:

(North)

G:	:	:	:G
Y:	:	:	
:	:	:	
:	:	:	
Y	:	:	B:

(Pickup)

R:	:	:	:G
G:	:	:	
Y:	:	:	
:	:	:	
Y	:	:	B:

(South)

R:	:	:	:G
Y:	:	:	
G:	:	:	
Y:	:	:	
Y	:	:	B:

(South)

R:	:	:	:G
Y:	:	:	
Y:	:	:	
G:	:	:	
Y	:	:	B:

(East)

R:	:	:	:G
Y:	:	:	
Y:	:	:	
Y:	:	:	
Y	:	:	B:

(East)

R:	:	:	:G
Y:	:	:	
Y:	:	:	
Y:	:	:	
Y	:	:	B:

(East)

```

(East)
+-----+
|R: | : :G|
| : | : : |
| : : : : |
| : | : |B:|
+-----+
(South)
+-----+
|R: | : :G|
| : | : : |
| : : : : |
| : | : |B:|
+-----+
(Y) : |B:|
+-----+
(South)
+-----+
|R: | : :G|
| : | : : |
| : : : : |
| : | : |B:|
+-----+
(Y) : |B:|
+-----+
(Dropoff)
10

```

Burada araç önce maviye gidip müşteriyi alıp sonra mora gidip müşteriyi bırakıyor. Bu işlemi toplam 10 adımda bitirebilmektedir. Siz de gözle bilişsel yeteneklerinizi kullanarak bu işlemin kaç adımda yapılabileceğini hesaplayınız. Şimdi run_episode fonksiyonunu yeniden çalıştırıp oluşan durumları yan yana görüntüleyelim:

```

(South)
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+
|R: | : :G| |R: | : :G| |R: | : :G| |R: | : :G| |R: | : :G| |R: | : :G| |R: | : :G| |R: | : :G| |R: | : :G| |
| : | : : | | : | : : | | : | : : | | : | : : | | : | : : | | : | : : | | : | : : | | : | : : |
| : | : : | | : | : : | | : | : : | | : | : : | | : | : : | | : | : : | | : | : : | | : | : : |
| : | : |B:| | : | : |B:| | : | : |B:| | : | : |B:| | : | : |B:| | : | : |B:| | : | : |B:|
|Y| : |B:| +-----+ |Y| : |B:| +-----+ |Y| : |B:| +-----+ |Y| : |B:| +-----+ |Y| : |B:| +-----+ |Y| : |B:|
+-----+ (West) +-----+ (South) +-----+ (South) +-----+ (Pickup) +-----+ (North) +-----+ (North) +-----+ (North) +-----+ (North)

```

Bu işlemin hareketli bir biçimde görüntülenmesi için şöyle bir yol izlenebilir:

```

def run_episode_record(env, q_table):
    count = 0
    obs = env.reset()

    records = []
    done = False
    while not done:
        records.append(env.render(mode='ansi'))
        action = np.argmax(q_table[obs])
        obs, reward, done, _ = env.step(action)
        count += 1

    return count, records

count, records = run_episode_record(env, q_table)

```

Burada run_episode_record problemi çözmekle birlikte oradaki görüntüyü bir listede biriktirmektedir. render metodunun mode parametresinin 'ansi' biçiminde girildiğine dikkat ediniz. Bu parametre sayesinde çıktı ekrana basılmaz. ANSI terminal sürücüsünün anlayabileceği ESC dizimleri biçiminde görüntüyü yazışal biçimde bize verir. Yani aslında fonksiyonda birtakım yazılar biriktirilmektedir. Bu yazılar zaman aralıklarıyla aynı yere bastırılırsa sanki görüntü hareketlimiş gibi bir etki oluşmaktadır. Ekranın silinmesi için 'x1b[1J' özel yazısının yazdırılması yeterli olmaktadır. Eylemleri hareketli görüntülemek için de şöyle bir fonksiyon yazılabilir:

```

def disp_records(records):
    for record in records:
        print('\x1b[1J')
        print(record)

```

```

    time.sleep(0.5)
    print(count)

count, records = run_episode_record(env, q_table)
disp_records(records)

```

Gym FrozenLake8x8-v0 Simülasyonu

Gym'in bu simülasyonunda bir kişi buzlu bir zemin üzerinde bir yerden (S) bir yere (G) varmak istemektedir. Ancak buzlu zeminde çatıtlaklar (delikler) vardır. Bu çatıtlaklara düşündüğünde oyun biter. Hedefe varlığında 1 puan ödül verilmektedir. Düşme ya da hedefe varmamaya yol açan her hareket 0 ödülün verilmesine yol açmaktadır. Simülasyon şu biçimde başlatılabilir:

```

import gym

env = gym.make('FrozenLake8x8-v0', is_slippery=False)
obs = env.reset()
env.render()

```

```

SFFFFFFF
FFFFFFF
FFFHFFF
FFFFFHFF
FFFHFFFF
FHHFFFHF
FHFFHFHF
FFFHFFFFG

```

Buradaki karakterler şu anlamlara gelmektedir:

- S: Başlangıç noktası (Source)
- F: Donmuş yer (Frozen)
- H: Delik (Hole)
- G: Hedef (Goal)

Bu simülasyondaki toplam durumların sayısı yani observation_space 64'tür. Çünkü gözlemlenecek tek şey 8X8'lük matristir. Eylemlerin sayısı yani action_space ise 4'tür. Eylemler kişinin gideceği yönü (yukarı, aşağı, sola, sağa) belirtmektedir. Yön belirten değerler şunlardır:

- 0: Left
- 1: Aşağı
- 2: Sağ
- 3: Yukarı

Oyunun iki versiyonu vardır: Kaygan mod ve kaygan olmayan mod. Default durum kaygan moddur. Kaygan modda kişi her zaman kayganlıktan dolayı istenilen yönde hareket edemeyebilmektedir. (Yani örneğin sağa gitmek isterken kaayarak aşağıya gidebilmektedir.) Eğer make fonksiyonunda is_slippery isimli parametresi False geçilirse kaygan olmayan mod seçilmiş olur. Bu durumda kişi her zaman istediği yönde hareket edebilmektedir.

Oyundaki ödül mekanizması hedefe varlığında 1 puan diğer bütün eylemlerde 0 puan biçimindedir. Yani oyunda yalnızca hedefe varlığında ödül alınmaktadır.

Aşağıda rasgele hareketler yapan örnek bir program vardır:

```

import gym
import time

env = gym.make('FrozenLake8x8-v0', is_slippery=False)

```

```

obs = env.reset()

count = 0
for i in range(1000):
    action = env.action_space.sample()
    obs, reward, done, info = env.step(action)
    print('\x1b[1J')
    env.render()
    print(count)
    count += 1
    if done:
        break
    time.sleep(0.2)

(Down)
SFFFFFFF
FFFFFFFF
FFFHFFFF
FFFFFHFF
FFFHFFFF
FHFFFHF
FHFFHFHF
FFFHFFFFG
12

```

Şimdi de Q-Learning algoritmasıyla problemi çözelim:

```

import gym
import time

import numpy as np

NEPOCHS = 5000
MAX_ITER = 100
DISCOUNT = 0.9
LEARNING_RATE = 0.6

def train(env, nepochs=NEPOCHS, max_iter=MAX_ITER, learning_rate=LEARNING_RATE,
discount=DISCOUNT):
    q_table = np.zeros((env.observation_space.n, env.action_space.n))

    for i in range(nepochs):
        obs = env.reset()
        for j in range(max_iter):
            action = np.argmax(q_table[obs] + np.random.randn(1, env.action_space.n) * (1. / (i + 1)))
            next_obs, reward, done, _ = env.step(action)

            q_table[obs, action] = q_table[obs, action] + learning_rate * (
                reward + discount * np.max(q_table[next_obs]) - q_table[obs, action])
            obs = next_obs
            if done:
                break

    return q_table

```

Eğitimde keşif stratejisi olarak normal dağılım yöntemi kullanılmıştır. Bu yöntemde Q-Tablosunun ilgili satırındaki eylemlerin Q değerlerine normal dağılıma uygun rastgele değerler toplanmıştır. Bu toplamın en yüksek Q değeri eylem olarak seçilmiştir:

```
action = np.argmax(q_table[obs] + np.random.randn(1, env.action_space.n) * (1. / (i + 1)))
```

Aynı zamanda her yinelemede bu rastgele normal dağıl değerlerinin etkisinin $1 / (i + 1)$ çarpanı ile düşürüldüğünü görüyorsunuz. Şimdi de simülatörü çalıştırıp hedefe bulan fonksiyonu yazalım:

```
def run_episode(env, q_table):
    count = 0
    obs = env.reset()
    done = False
    while not done:
        action = np.argmax(q_table[obs])
        obs, reward, done, _ = env.step(action)
        print('\x1b[1J')
        env.render()
        time.sleep(0.4)
        count += 1
    return count
```

Programı şöyle çalıştırabiliriz:

```
env = gym.make('FrozenLake8x8-v0', is_slippery=False)
q_table = train(env)

count = run_episode(env, q_table)
print(count)
```

Bu simülatörde simülatör reset edildiğinde her zaman başlangıç S pozisyonu (sol-üst köşe) olmaktadır. Kişinin başlangıç başlangıç pozisyonunun değiştirilmesine yönelik bir dokümantasyon yoktur. Ancak sınıfların yapısı incelendiğinde pozisyon bilgisinin env.env.s değişkeninde turulduğu görülmektedir. Yani biz bu değişkendeki değeri değiştirerek simülatörün S dışında başka bir yerden çalışmaya başlamasını sağlayabiliriz. Aşağıdaki örnekte eğitim sırasında her defasında rastgele bir başlangıç noktası seçilmiştir:

```
import gym
import time

import numpy as np

NEPOCHS = 50000
MAX_ITER = 1000
DISCOUNT = 0.8
LEARNING_RATE = 0.1

def train_random(env, nepochs=NEPOCHS, max_iter=MAX_ITER, learning_rate=LEARNING_RATE,
discount=DISCOUNT):
    q_table = np.zeros((env.observation_space.n, env.action_space.n))

    for i in range(nepochs):
        obs = env.reset()
        env.env.s = np.random.randint(env.observation_space.n)
        for j in range(max_iter):
            action = np.argmax(q_table[obs] + np.random.randn(1, env.action_space.n) * (1. / (i + 1)))
            next_obs, reward, done, _ = env.step(action)

            q_table[obs, action] = q_table[obs, action] + learning_rate * (
                reward + discount * np.max(q_table[next_obs]) - q_table[obs, action])
            obs = next_obs
            if done:
                break

    return q_table
```

```

def run_episode_random(env, q_table):
    count = 0
    obs = env.reset()
    env.env.s = np.random.randint(env.observation_space.n)
    done = False
    while not done:
        action = np.argmax(q_table[obs])
        obs, reward, done, _ = env.step(action)
        print('\x1b[1J')
        env.render()
        time.sleep(0.4)
        count += 1
    return count

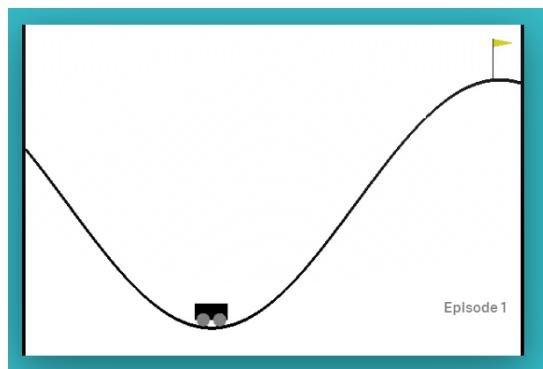
env = gym.make('FrozenLake8x8-v0', is_slippery=False)
q_table = train_random(env)

count = run_episode_random(env, q_table)
print(count)

```

MountainCar Örneğinin Q Learning Algoritmasıyla Çözümü

Anımsanacağı gibi OpenAI Gym içerisindeki MountainCar simülatöründe bir araba ileri ya da geri gidebiliyor ya da olduğu yerde durabiliyordu. Belli bir t anında bu hareketlerden yalnızca birini yapabiliyordu. Amaç ise onun tepeyi aşmasıydı. Daha önceden de gördüğümüz gibi araba sürekli ileri gitmeye çalışırsa tepeyi aşmaya gücü yetmemektedir. Biz daha önce bu problemi sezgisel biçimde çözmüştük. Arabayı hız 0'a düşene kadar ileri, daha sonra yine hız sıfıra düşene kadar geri yönde hareket ettirerek tepeyi aşırılmıştık. Burada ise arabanın tepeyi aşması için yapması gereken şeyler Q-Learning algoritmasıyla pekiştirmeli bir biçimde arabaya öğretilecektir.



Anımsanacağı gibi bu problemdeki mevcut eylemler (action space) 0, 1 ve 2'dir. 0 "geri git", 1 "gaz verme (motoru durdur)", 2 ise "ileri git" anlamına gelmektedir. Problemdeki gözlem değerleri (observation_space) ise iki elemanlı bir numpy dizisi olarak bize verilmektedir. Dizinin 0'inci indisli elemanı arabanın x eksenindeki konumu, 1'inci indisli elemanı ise arabanın o andaki hızını belirtmektedir. Burada işlemin başarısını belirten done koşulu iki duruma bağlıdır: Birincisi arabanın tepeyi aşması, ikincisi ise maksimum deneme sayısı olan 200 denemenin bitmesidir.

Bu problemin Q-Learning algoritmasıyla çözümünde ilk yapılacak şey durumların (states) oluşturulmasıdır. Çünkü gözlemlerdeki konum ve hız değerleri sürekli değerlerdir. Bizim bu değerleri ayrık hale getirip durumsallaştırmamız gereklidir. Sürekli gözlem değerlerini ayrık biçimde durumsallaştırırken bir bölge değerini önceden belirleyebiliriz. Tabii aslında her gözlemin diğer gözlemlerle aynı sayıda parçaaya ayrılması kolay bir tasarımdır. Örneğin MountainCar simülasyonunda parça sayısını 20 olarak belirlediğimizi düşünelim. Bu durumda gözlem değerlerinden biri olan konum maksimum ve minimum değerler dikkate alınarak 20 parçaaya bölünecektir. Benzer şekilde arabanın hızı da maksimum ve minimum değerler dikkate alınarak yine 20 parçaaya bölünebilir. Arabanın içinde bulunduğu durum (state) böylece aslında 20×20 durumdan bir tanesi olacaktır. Matrisel biçimde düşünüldüğünde bunun için 20×20 'lik iki boyutlu bir matris oluşturulabilir. Arabanın içinde bulunduğu durum konum ve hız ile temsil ediliyor olsa da biz bu durumu matrisel

birimde iki indeksle temsil edebiliriz. (Tabii bu tür örneklerde observation_space daha fazla elemana sahipse bu durumda matris de daha fazla boyuttan oluşacaktır. Bizim durumsal indekslerimiz de daha fazla bileşene sahip olacaktır. Örneğin CartPole simülasyonundaki gözlem değişkenleri 4 tanedir.) Aşağıda MountainCar simülasyonu için bir gözlemi indekslerden oluşan durumsal bir bilgiye dönüştüren fonksiyon örneği verilmiştir:

```
import gym

env = gym.make('MountainCar-v0')

DISCRETE_STATES = 20

def obs_to_state(env, obs):
    state_intervals = (env.observation_space.high - env.observation_space.low) / DISCRETE_STATES
    return ((obs - env.observation_space.low) / state_intervals).astype(int)
```

Şimdi Q-Tablosunu oluşturmaaya çalışalım. MountainCar örneğindeki eylemlerin sayısı 3 olduğuna göre oluşturacağımız Q-Tablosu da $20 \times 20 \times 3$ boyutlarında olacaktır. Q-Tablosu istenilen boyutta şöyle oluşturulabilir:

```
q_table = np.zeros([20, 20, 3])
```

Buradaki boyutu ileride ele alacağımız bir neden dolayı bir fazla olarak oluşturacağız. Bu durumda Q-Tablosu env nesnesinden hareketle daha parametrik bir biçimde şöyle de oluşturulabilir:

```
index_list = [DISCRETE_STATES + 1] * env.observation_space.shape[0] + [env.action_space.n]
q_table = np.zeros(index_list)
```

Şimdi çeşitli denemeler yaptırarak sistemi öğrenme sürecine sokmaya çalışalım. Burada yaptırılan denemeler Q-Tablosunun uygun biçimde doldurulmasına yol açacaktır. Bu örnekte keşif stratejisi (exploration strategy) olarak "epsilon greedy" yönteminin bir varyasyonunu kullanacağız. Kullanacağımız yöntemde EPSILON olasılıkla tamamen rastgele bir eylem seçimi yapacağız. Ancak diğer durumlarda birbirlerine benzer Q değerlerinin arasında birz rassallık da oluşturacağız.

```
import gym

NEPOCHS = 500
MAX_ITER = 1000
LEARNING_RATE = 0.01
EPSILON = 0.2
DISCOUNT = 0.90
DISCRETE_STATES = 15

def obs_to_state(env, obs):
    state_intervals = (env.observation_space.high - env.observation_space.low) / DISCRETE_STATES
    return ((obs - env.observation_space.low) / state_intervals).astype(int)

import numpy as np

def train(env):
    print('Training starts...')

    index_list = [DISCRETE_STATES + 1] * env.observation_space.shape[0] + [env.action_space.n]
    q_table = np.zeros(index_list)
    for i in range(NEPOCHS):
        obs = env.reset()
        for k in range(MAX_ITER):
            a, b = obs_to_state(env, obs)
            if np.random.uniform(0, 1) < EPSILON:
                action = np.random.choice(env.action_space.n)
            else:
```

```

logits = q_table[a, b]
logits_exp = np.exp(logits)
probs = logits_exp / np.sum(logits_exp)
action = np.random.choice(env.action_space.n, p=probs)

next_obs, reward, done, _ = env.step(action)
a_next, b_next = obs_to_state(env, next_obs)
q_table[a, b, action] = q_table[a, b, action] + LEARNING_RATE * (
    reward + DISCOUNT * np.max(q_table[a_next, b_next]) - q_table[a, b,
action])
obs = next_obs
print(i, end=' ')
print()

return q_table

```

Buradaki train fonksiyonu hakkında bazı açıklamalar yapmak istiyoruz:

- Fonksiyonda iç içe iki döngü vardır. Dış döngü NEPOCHS kadar dönmektedir. NEPOCHS toplam yapılan denemelerin sayısını belirtir. İç döngü ise MAX_ITER kadar dönmektedir. MAX_ITER de bir denemede en fazla kaç hareket yapılacağını belirtir. Teorik olarak NEPOCHS ve MAX_ITER sayıları ne kadar fazla olursa öğrenmenin de o kadar iyi olacaktır.

- Denemeler sırasında seçilecek eylem (action) aşağıdaki gibi bir koşulla belirlenmiştir:

```

if np.random.uniform(0, 1) < EPSILON:
    action = np.random.choice(env.action_space.n)
else:
    logits = q_table[a, b]
    logits_exp = np.exp(logits)
    probs = logits_exp / np.sum(logits_exp)
    action = np.random.choice(env.action_space.n, p=probs)

```

Yukarıdaki if deyiminde eğer rastgele üretilen bir değer EPSILON'dan (%2) küçükse bu durumda rastgele bir eylem seçilmektedir. Eğer bu rastgele değer EPSILON'dan küçük değilse Q-Tablosundaki en iyi eylemler arasında belli oranlarda rastgelelik ile eylem seçimi yapılmıştır. (Yani örneğin Q tablosundaki eylemlerin puanları farklıysa yüksek puanın olasılığı yüksek olacak biçimde bir rastgelelik oluşturulmuştur.) Bu sayede Q değerleri birbirlerine yakın olan eylemler arasında her zaman en yüksek Q değerine sahip olan değil alternatifler de değerlendirilmektedir.

- Döгüdeki Q-Tablo değerlerinin güncellemesi şöyle yapılmıştır:

```

next_obs, reward, done, _ = env.step(action)
a_next, b_next = obs_to_state(env, next_obs, state_intervals)
q_table[a, b, action] = q_table[a, b, action] + learning_rate * (
    reward + GAMMA * np.max(q_table[a_next, b_next]) - q_table[a, b, action])

```

Burada konuya girişte açıkladığımız Q değerinin hesaplanma işlemi ve tablo elemanın güncelleme işlemi yapılmaktadır.

- train fonksiyonun oluşturulan Q tablosuna geri döndüğüne dikkat ediniz.

train fonksiyonu yalnızca Q tablosunu doldurarak öğrenme işlemini gerçekleştirmektedir. Yani bu öğrenme sürecinin sonunda artık biz hangi durumdan (state) başlarsak başlayalım Q tablosundaki en iyi değerlerden hareketle eylemler peşi sıra yapılacaktır. İşin bu kısmını run fonksiyonu yapmaktadır:

```

def run(env, q_table):
    count = 0
    obs = env.reset()
    while True:

```

```

env.render()
a, b = obs_to_state(env, obs)
action = np.argmax(q_table[a, b])
obs, reward, done, _ = env.step(action)
count += 1
if done:
    break

env.close()

return count

```

Fonksiyonun yaptığı tek şey aslında Q tablosundan hareketler durum geçişleri yapmaktadır. Q tablosunda bir duruma ilişkin en iyi eylem şu ifadeyle bulunmuştur:

```
action = np.argmax(q_table[a, b])
```

`np.argmax` fonksiyonunun numpy dizisi içerisinde en yüksek değerin indeksini verdiğini anımsayınız. O halde yukarıdaki ifade aslında bize o durum için en yüksek Q değerli eylemi vermektedir.

Nihayet oyunu şöyle çalıştırabiliriz:

```

env = gym.make('MountainCar-v0')
q_table = train(env)

count = run(env, q_table)
print(count)

```

CartPole Simülasyonunun Q Learning Algoritmasıyla Çözümü

Cartpole Simülasyonu aşağıdaki gibi bir Q Learning algoritması kullanılabılır:

```

import numpy as np
import gym

NEPOCH = 2000
MAX_ITER = 1000
EPS = 0.02
GAMMA = 1.5
LEARNING_RATE = 0.001
MIN_LR = 0.003
INITIAL_LR = 0.003
DISCRETE_STATES = 70

def obs_to_state(env, obs, state_intervals):
    states = ((obs - env.observation_space.low) / state_intervals).astype(int)
    states[states > DISCRETE_STATES] = DISCRETE_STATES
    states[states < 0] = 0

    return states

def train(env):
    index_list = [DISCRETE_STATES + 1] * env.observation_space.shape[0] + [env.action_space.n]
    q_table = np.zeros(index_list)
    state_intervals = (env.observation_space.high - env.observation_space.low) /
DISCRETE_STATES
    for i in range(NEPOCH):
        obs = env.reset()
        learning_rate = LEARNING_RATE
        for j in range(MAX_ITER):
            a, b, c, d = obs_to_state(env, obs, state_intervals)

```

```

if np.random.uniform(0, 1) < EPS:
    action = np.random.choice(env.action_space.n)
else:
    logits = q_table[a, b, c, d]
    logits_exp = np.exp(logits)
    probs = logits_exp / np.sum(logits_exp)
    action = np.random.choice(env.action_space.n, p=probs)
obs, reward, done, _ = env.step(action)
a_next, b_next, c_next, d_next = obs_to_state(env, obs, state_intervals)
q_table[a, b, c, d, action] = q_table[a, b, c, d, action] + learning_rate * (
    reward + GAMMA * np.max(q_table[a_next, b_next, c_next, d_next]) -
q_table[a, b, c, d, action])
print('.', end='')

print()
return q_table

def run_episode(env, q_table, render=True, rnd=False):
    state_intervals = (env.observation_space.high - env.observation_space.low) /
DISCRETE_STATES
    obs = env.reset()
    for step in range(MAX_ITER):
        a, b, c, d = obs_to_state(env, obs, state_intervals)
        if not rnd:
            action = np.argmax(q_table[a, b, c, d])

        else:
            action = env.action_space.sample()

        obs, reward, done, _ = env.step(action)
        if render:
            env.render()
        if done:
            break
    return step + 1

env = gym.make('CartPole-v0')

env.observation_space.low[1] = -10
env.observation_space.high[1] = 10

env.observation_space.low[3] = -10
env.observation_space.high[3] = 10

q_table = train(env)

total_step = run_episode(env, q_table, render=False, rnd=False)
print(total_step)

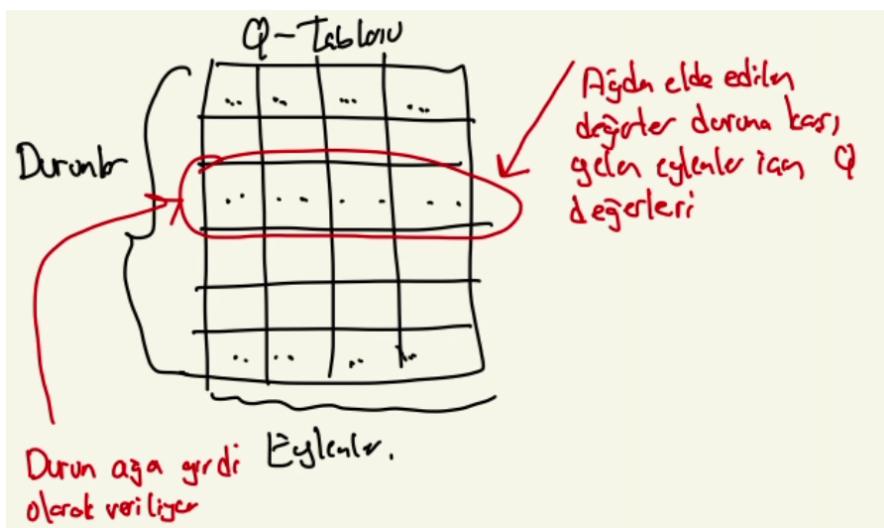
env.close()

```

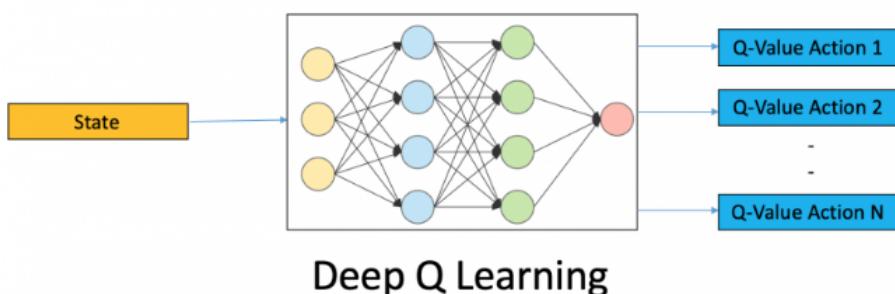
Pekiştirmeli Öğrenmede Deep Q-Learning (DQN) Yöntemi

Q Learning en çok tercih edilen pekiştirmeli öğrenme algoritmasıdır. Ancak bu algoritma durum sayısının (yani observation space'in) çok fazla olduğu özellikle de sürekli (continuous) gözlem değerlerinin söz konusu olduğu durumlarda bazı sorunlar oluşturmaktadır. Bu sorunlardan en önemlisi oluşturulacak Q-Tablosunun büyülüğu dolayısıyla kaplayacağı alandır. Aynı zamanda bu yöntemde Q-Tablosunun doldurulması için iyi bir eğitimin sağlanması gerekmektedir. Maalesef bu yöntemde çok fazla durum söz konusu olduğunda yetersiz bir eğitimde Q-Tablosunun önemli bir kısmı boş kalmamıştır. Bu da çözümde bazı durumlarda etmenin (agent) ne yapacağını bilememesine yol açmaktadır. İşte Deep Q-Learning (DQN) yöntemi aslında yapay sinir ağları ile Q-Learning algoritmasının hibrit bir biçimidir.

Anımsanacağı gibi Q-Learning algoritmasında önce bir eğitim süreci gerçekleştiriliyor durumlar için en iyi eylemlerin ne olacağını belirten bir Q-Tablosu oluşturuluyordu. Sonra da bu Q-Tablosuna bakılarak bir rota izleniyordu. İşte Deep Q-Learning yönteminde bir Q-Tablosu oluşturulup ona başvurulmak yerine belli durumda en iyi eylemin ne olacağı yapay sinir ağından elde edilmektedir. Deep Q-Learning yönteminde oluşturulacak yapay sinir ağının girdilerini durum bilgisi oluşturmaktadır. Bu yapay sinir ağının çıktıları ise belli bir durumdaki Q değerleridir.



Deep Q-Learning yönteminde kullanılan yapay sinir ağının ilgili durum bilgisini alır, çıktı olarak da her eylem için kestirilen Q değerlerini vermektedir. Biz de çıktı Q değerlerinin en büyüğünü bulup ona karşı gelen eylemin en iyi olduğu sonucunu elde ederiz. Bu durumda buradaki ağın girdi katmanında `len(env.observation_space)` kadar nöron, çıktı katmanında da `env.action_space.n` kadar nöron olacaktır.



Sekil <https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/> sitesinden alınmıştır.

Deep Q-Learning yönteminde sinir ağının bir regresyon modeli oluşturduğu için bizim sürekli durum değerlerini ayrı hale getirmemize de gerek kalmamaktadır. Bu sürekli durum değerleri doğrudan ağa girdi olarak verilebilir.

Örneğin CartPole simülasyonunda bu ağın girdi katmanında 4 tane nöron olacaktır. (Arabanın x koordinatı, arabanın hızı, direğin açısı ve direğin açısal hızı). Çıktı katmanında da 2 nöron bulunacaktır. (Araba sola mı sağa mı gidecek?) Bu kestirim ağı bir lojistik regresyon değil normal çok değişkenli (multivariate) bir regresyondur. Dolayısıyla tipik olarak ağı oluştururken gizli katmanların aktivasyon fonksiyonları "relu", loss fonksiyonu "mean_squared_error" ya da "mean_absolute_error", optimizasyon algoritması ise "adam" ya da "rmsprop" ya da "sgd" alınabilir. Çıktı katmanın aktivasyon fonksiyonu da "linear" olmalıdır.

Bilindiği gibi regresyon modellerindeki sinir ağları "denetimli (supervised)" bir öğrenme teknigi kullanmaktadır. O halde Deep Q-Learning yöntemindeki sinir ağı da bir eğitim sürecine sahip olmak zorundadır. Peki biz işin başında sistem hakkında bir şey bilmemiğimize göre eğitimi nasıl uygulayacağız? İşte tipik olarak bu yöntemde eğitimle kestirim çoğu kez bir arada yürütülmektedir. Programcı önce bazı gözlem kümesi için Q değerlerini bulur, bunları eğitimde kullanır. Sonra da kestirim için sinir ağından faydalananır.

Deep Q-Learning Yöntemi kabaca şöyle gerçekleştirilmektedir:

- 1) Gözlemlere karşılık eylemler için Q değerlerini verecek bir yapay sinir ağı modeli oluşturulur.
- 2) Q-Learning algoritması uygulanarak belli sayıda gözle ve q değerleri elde edilir. Bu Q değerleriyle ağı eğitilir. Genellikle ağın eğitilmesi için bu Q değerlerinin belli bir sayıya (batch) erişmesi beklenir. Bunun için Q değerleri bir bellek alanında saklanabilmektedir. Bu Q değerleri belli sayıya eriştikten sonra artık iadelî seçimlerle ağıtım başlatılmaktadır. Tabii aslında eğitim hiç biriktirme yapmadan hemen de başlatılabilir.
- 3) Q-Learning algoritması işletilirken Q formülü yine uygulanır. Ancak bu formüldeki tablonun en yüksek elemanları yapay sinir ağından tahminleme yoluyla elde edilmektedir.

Deep Q-Learning Yönteminin CartPole Simülasyonunda Kullanılması

CartPole simülasyonunda Deep Q-Learning yöntemini kullanırken önce yapay sinir ağını oluştururuz. Sonra da bir keşif stratejisi ile ağı eğitiriz. Ağı yeterince eğitildikten sonra da en iyi Q değerlerinden hareketle eylemleri uygularız. Biz burada ağı eğitmırken her zaman belirleyeceğimiz bir BATCH miktarı kadar veriyi fit işlemeye sokacağız. Bu BATCH miktarı olusana kadar beklenecek sonra eğitime başlanacaktır. Bir kez biriktirdiğimiz değerler bu BATCH miktarına erişince artık her defasında biriktirilen değerler içerisinde BATCH miktarı kadarı rastgele seçilerek eğitime devam edilecektir.

```
import numpy as np
import gym
import random

DISCOUNT = 0.95
LEARNING_RATE = 0.001
BATCH_SIZE = 32
EPSILON_MAX = 1.0
EPSILON_MIN = 0.01
EPSILON_DECAY = 0.995

env = gym.make('CartPole-v1')
exploration_rate = EPSILON_MAX

observation_space = env.observation_space.shape[0]
action_space = env.action_space.n

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

def train(env, epoch):
    model = Sequential()
    model.add(Dense(64, input_dim=observation_space, activation='relu', name='Hidden-1'))
    model.add(Dense(64, activation='relu', name='Hidden-2'))
    model.add(Dense(action_space, activation='linear', name='Output'))
    model.compile(optimizer=Adam(lr=LEARNING_RATE), loss='mse')

    epsilon = EPSILON_MAX
    memory = []

    for i in range(epoch):
        obs = env.reset()
        obs = obs.reshape((1, -1))

        count = 0
        while True:
            if np.random.rand() < epsilon:
                action = np.random.choice(action_space)
```

```

else:
    q_values = model.predict(obs)
    action = np.argmax(q_values[0])

obs_next, reward, done, _ = env.step(action)

env.render()

reward = reward if not done else -reward

obs_next = obs_next.reshape((1, -1))
memory.append((obs, action, reward, obs_next, done))

if len(memory) >= BATCH_SIZE:
    batch = random.sample(memory, BATCH_SIZE)
    for obs_mem, action_mem, reward_mem, obs_next_mem, done_mem in batch:
        q_update = reward_mem
        if not done_mem:
            q_update = (reward_mem + DISCOUNT *
np.amax(model.predict(obs_next_mem)[0]))
            q_values = model.predict(obs_mem)
            q_values[0, action_mem] = q_update
            model.fit(obs_mem, q_values, verbose=0)
    epsilon *= EPSILON_DECAY
    epsilon = max(EPSILON_MIN, exploration_rate)

obs = obs_next
count += 1

if done:
    break

print(f'{i} => {count}')

return model

def run_episode(env, model):
    obs = env.reset()
    count = 0
    while True:
        obs = obs.reshape(1, -1)
        action = np.argmax(model.predict(obs))
        obs, reward, done, info = env.step(action)
        if done:
            break
        env.render()
        count += 1

    return count

env = gym.make('CartPole-v1')
model = train(env, 300)
print('training ends...')

result = run_episode(env, model)
print(result)
env.close()

```

Burada kabaca şunlar yapılmıştır:

- Regresyon temelli bir sinir ağı modeli kurulmuştur. Bu sinir ağına CartPole gözlemleri girdi olarak verilip eylemlerin (actions) Q değerleri çıktı olarak alınmaktadır. Dolayısıyla ağ eğitildikten sonra qğın çıktısından elde edilen Q

değerlerinin en yüksek olanı eylem olarak tercih edilecektir. Programda ayrik hale getirme gibi bir işlem yapılmamıştır. Çünkü buna gerek yoktur. Zaten sinir ağı sürekli değerler için bir tahmin yapabilmektedir.

- Programda keşif stratejisi olarak "epsilon greedy" yöntemi kullanılmıştır. Ancak rassallık gitgide azaltılmıştır. Rassallık önce EPSILON_MAX değerinden başlatılır. Her yinelemede EPSILON_DECAY kadar azaltılmaktadır. Ancak rassalık en fazla EPSILON_MIN değerine kadar azaltılmış olmaktadır.

- Ağın eğitilmesi şöyle sağlanmaktadır: Önce BATCH_SIZE kadar veri toplanana kadar hiç eğitim uygulanmaz. İlk eğitim BATCH_SIZE bilgi toplandığında başlatılır. Bundan sonra da artık her yeni bilgi oluştuğunda yeniden rastgele bir BATCH_SIZE kadar bilgi çekilerek eğitim devam ettirilmektedir. Örneğimizde BATCH_SIZE 32 olarak alınmıştır. Bu durumda ilk 32 değer için hiçbir eğitim yapılmamakta sonra hep rasgele 32 değer seçilerek fit işlemi yapılmaktadır.

Bu biçimdeki modelin başarısı neye bağlıdır? Seçilen bazı parametrik değerlerin, sinir ağının genel yapısının ve diğer parametrelerin öğrenme hızında ve verilen kararlarda etkisinin olduğu açıklıdır. Programcı bu tür durumlarda deneme yanılma yöntemlerine de başvurabilir. Ancak ağın eğitilmesi için çok denemenin yapılması gerekebilir.

Pekiştirmeli Öğrenmede Kullanılabilecek Yüksek Seviyeli Kütüphaneler

Pekiştirmeli öğrenme problemleri genellikle probleme özgü bir biçimde tasarlanıp eğitilmektedir. Bu da model geliştirme zamanını artırmaktadır. İşte bu tür problemlerdeki geliştirme zamanını azaltmak için son birkaç yıldır yüksek seviyeli kütüphaneler oluşturulmaya başlanmıştır. Henüz bu kütüphaneler çok olgun bir noktada değildir. Ancak zaman içerisinde bunlar da daha iyi hale getirilecektir. Bu tür kütüphaneler arasında uygun olanını seçmek de henüz kolay bir karar değildir. Çünkü hangi kütüphanelerin ne kadar süre içerisinde olgunlaşacağını tahmin etmek zordur. Böyle bir seçimde göz önüne alınması gereken ölçütlerden bazıları şunlar olmalıdır:

- Kütüphanenin genel yeteneği (örneğin kaç algoritma destekleniyor gibi)
- Kütühanenin dokümantasyonu
- Kütüphanenin genel tasarıtı. Bazı kütüphaneler daha esnek tasarıma sahiptir.
- Kütüphaneyi oluşturan proje grubunun büyülüğu.
- Kütüphaneyi oluşturan proje grubunun destek aldığı sponsorlar.
- Kütüphanenin kolay kullanılabilir olması
- Projenin güncellenme aralığı

Kursun yapıldığı zaman aralığındaki belli başlı kütüphaneler şunlardır:

- KerasRL
- Tensorforce
- OpenAI-Baselines
- Stable Baselines
- ChainerRL
- Coach
- RLLib

Stable-Baselines Kütüphanesinin Kullanımı

Stable-Baselines Kütüphanesi aslında OpenAI kurumunun Baselines kütüphanesinin çatallanmış (fork edilmiş) bir versiyonudur. Ancak bu versiyonda orijinal BaseLines kütüphanesi pek çok bakımdan iyileştirilmiştir. Kütüphanein dokümantasyonu <https://stable-baselines.readthedocs.io/en/master/> adresinde bulunmaktadır. Kütüphane pip ile şöyle install edilebilir:

```
pip install stable-baselines
```

Kütüphane içerisindeki bazı algoritmalar paralel veri işleme yaptığı için OpenMPI kütüphanesinin de bunlar için yüklenmesi gerekmektedir. Ancak bu yalnızca kısıtlı birkaç algoritma için söz konusudur.

Stable-baselines kütüphanesi arka planda Tensorflow kullanmaktadır. Ancak kütüphanenin kurs yapıldığındaki güncel versiyonları Tensorflow'un 2'li versiyonlarını desteklememektedir. Bu nedenle Tensorflow'un 1.8.0 ve 1.15.0 arası versiyonlarının yüklenmesi gerekmektedir. Downgrade işlemi şöyle yapılabilir:

```
pip uninstall tensorflow
pip install tensorflow==1.15.0
```

Kütüphanelerin hangi versiyonlarının yüklü olduğunu anlamak için pip komutunda show seçeneği kullanılabilir. Örneğin:

```
(base) C:\Users\CSD>pip show tensorflow
Name: tensorflow
Version: 1.15.0
Summary: TensorFlow is an open source machine learning framework for everyone.
Home-page: https://www.tensorflow.org/
Author: Google Inc.
Author-email: packages@tensorflow.org
License: Apache 2.0
Location: c:\programdata\anaconda3\lib\site-packages
Requires: protobuf, google-pasta, wrapt, astor, six, opt-einsum, gast, numpy, keras-applications, tensorboard, termcolor, grpcio, wheel, tensorflow-estimator, keras-preprocessing, absl-py
Required-by:
```

Tensorflow'u downgrade yapmak yerine Python'da "sanal ortam (virtual environment)" da oluşturulabilir. Sanal ortamın nasıl oluşturulacağını Internet'teki çeşitli kaynaklardan öğrenebilirisiniz.

Stable-Baselines kütüphanesinin kullanımı oldukça basittir. Tipik çalışma şöyle yapılmaktadır:

- 1) Önce bir ortam (environment) nesnesi yaratılır. Ortam nesneleri doğrudan gym.make ile oluşturulabilmektedir. Tabii programcı isterse kendi problemi için kendi ortam nesnelerini de oluşturabilir (custom environment).
- 2) Uygulanacak algoritma belirlenir. Algoritmaların çoğu geneldir. Yani pek çok ortam için kullanılabilmektedir. En çok kullanılan algoritmalar DQN (Deep Q-Learning), PPO1 ve PPO2 (Proximal Policy Optimization), TRPO (Trust Region Policy Optimization), ACKTR (Actor Critic using Kronecker-Factored Trust Region), DDPG (Deep Deterministic Policy Gradient) algoritmalarıdır. Her algoritma aynı isimli sınıflarla temsil edilmiştir. Bu sınıf nesnelerinde genel olarak model nesneleri denilmektedir. Algoritma sınıflarına ilişkin __init__ metodlarının pek çok parametresi vardır. Parametrelerin çoğu default değerler almaktadır. Zorunlu parametrelerin birisi kullanılacak politikadır. Tipik politikalar şunlardır: MlpPolicy, LnMlpPolicy, CnnPolicy, LnCnnPolicy. En çok kullanılan politikalar MlpPolicy (Multi Layer Perceptron, 64 nörondan oluşan iki katmanlı ağ) ve CnnPolicy (Convolutional Neural Network, 64 nörondan oluşan iki katmanlı ağ)
- 3) model nesnelerinin learn metotları ile eğitim gerçekleştirilir. Artık eğitilmiş bir model elde edilmiş olur.
- 4) Eğitilmiş model gerçek durumlara uygulanır. Bunun için model sınıflarının predict metotları kullanılır. Bu metotlar parametre olarak bizden durumu (observation) alırlar ve bize uygulacak eylemi verirler biz de bu işlemleri bir döngü içerisinde yaparak etmeni uygun biçimde çalıştırımız oluruz.

Örneğin:

```
import gym

from stable_baselines.deepq.policies import MlpPolicy
from stable_baselines import DQN

env = gym.make('CartPole-v1')

model = DQN(MlpPolicy, env, verbose=0)
model.learn(total_timesteps=100000)

obs = env.reset()
for i in range(2000):
```

```

action, _states = model.predict(obs)
obs, reward, done, info = env.step(action)
env.render()
print(i, end=' ')
env.close()

```

Burada algoritma olarak DQN (Deep Q-Learning) uygulanmıştır. Kullanılan politika MlpPolicy biçimindedir. Bu politika iki katmanlı her biri 64 nörondan oluşan bir yapay sinir ağı kullanmaktadır. learn fonksiyonunda eğitim için toplam 1000000 yinelemenin yapıldığını görüyoruz. Modelimn parametreleri (epsilon, gamma vs.) default biçimde seçilmiştir. Eğitimden sonra örnek bir deneme yapılmıştır.

Şimdi de MountainCar örneğini uygulayalım. MountainCar için de yukarıdaki programın neredeyse aynısı kullanılabilir. Örneğin:

```

import gym

from stable_baselines.deepq.policies import MlpPolicy
from stable_baselines import DQN

env = gym.make('MountainCar-v0')

model = DQN(MlpPolicy, env, verbose=0)
model.learn(total_timesteps=100000)

obs = env.reset()
i = 0
while True:
    action, _states = model.predict(obs)
    obs, reward, done, info = env.step(action)
    if done:
        break
    env.render()
    print(i, end=' ')
    i += 1

env.close()

```

Model eğitildikten sonra saklanıp geri yüklenebilir. Modeli saklamak için model sınıfının save isimli metodu kullanılır. Örneğin:

```
model.save('mountaincar')
```

Bu metot modeli bir .zip dosyası içerisinde saklamaktadır. Modeli geri yüklemek için ise algoritma sınıflarının statik load metodları kullanılmaktadır. Örneğin:

```
model = DQN.load('mountaincar')
```

Örneğin yukarıdaki MountainCar simülasyonunda eğitim sonraki değerleri saklayıp geri yükleyelim:

```

import gym

from stable_baselines.deepq.policies import MlpPolicy
from stable_baselines import DQN

env = gym.make('MountainCar-v0')

model = DQN(MlpPolicy, env, verbose=0)
model.learn(total_timesteps=100000)

model.save('mountaincar')

```

```

del model # model nesnesi kasti olarak siliniyor.

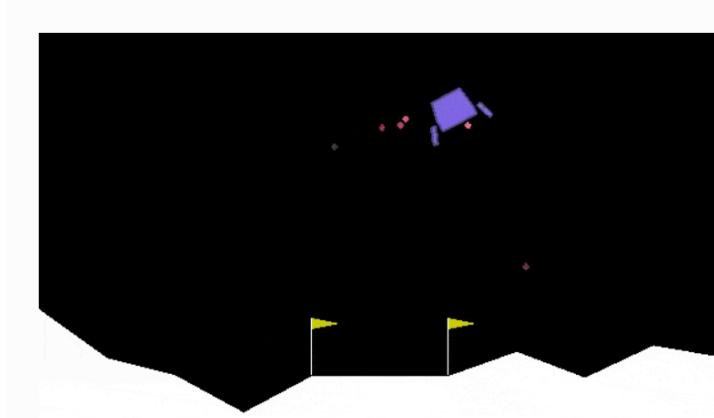
model = DQN.load('mountaincar')

obs = env.reset()
i = 0
while True:
    action, _states = model.predict(obs)
    obs, reward, done, info = env.step(action)
    if done:
        break
    env.render()
    print(i, end=' ')
    i += 1

env.close()

```

Stable-Baselines dokğanlarında içerisinde çeşitli örnekler vardır. Örneğin LunarLanding simülatörü ayda inmeye çalışan örümceği takit eder. Bu simülatörde amaç örümceğin iki bayrak arasına indirilmesidir. Örümcek sağa sola hareket ettirilebilmektedir.



Modelin eğitimi ve örnek uygulaması şöyle yapılmaktadır:

```

import gym
from stable_baselines import DQN
from stable_baselines.common.evaluation import evaluate_policy

# Create environment
env = gym.make('LunarLander-v2')

# Instantiate the agent
model = DQN('MlpPolicy', env, learning_rate=1e-3, prioritized_replay=True, verbose=1)
# Train the agent
model.learn(total_timesteps=int(2e5))

# Evaluate the agent
mean_reward, std_reward = evaluate_policy(model, model.get_env(), n_eval_episodes=10)
print(mean_reward, std_reward)

# Enjoy trained agent
obs = env.reset()
for i in range(1000):
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    env.render()

```

```
env.close()
```

Stable-Baselines İçin Custom Environment Yazımı

Örneklerden de görüldüğü gibi stable-baselines OpenAI gym ortamını temel almaktadır. Gerçekten de model sınıfının learn metodu bizden bir environment nesnesi istemektedir. Pekiyi biz kendi problemlerimizi bu kütüphaneyle nasıl çözebiliriz? İşte bunun için bizim gym uyumlu environment nesnesi oluşturmamız gereklidir.

Gym uyumlu environment nesnesi elde edebilmek için öncelikle gym.Env sınıfından türetme yapılarak bir environment sınıfının oluşturulması gereklidir. Örneğin:

```
import gym

class MyEnv(gym.Env):
    pass
```

Bizim bu kendi environment sınıfımız için birkaç metod ve örnek özniteliğini yazmamız gereklidir. Bunlar şöyledir:

Örnek öznitelikleri: action_space, observation_space
Metotlar: __init__, reset, step, render

Örneğin:

```
import gym

class MyEnv(gym.Env):
    metadata = {'render.modes': ['human']}

    def __init__(self):
        super().__init__()
        # ....
        self.action_space = < actions >
        self.observation_space = < observations >

    def reset(self):
        pass

    def step(self):
        pass

    def render(self):
        pass
```

Internet'te çeşitli custom environment örnekleri bulunmaktadır. Gym'in üçüncü parti simülatörleri de bu amaçla incelenebilir.