EX ERTINS 1HT

# Makine Öğrenme PyTorch ile and Scikit-Lea n

Makine öğrenimi ve derin öğrenme geliştirin
Python ile modeller

Sebastian Raschka
Yuxi (Hayden) Liu
Vahid Mirjalili

Packt

olimpikbarbell@gmail.com

# PyTorch ve Scikit-Learn ile Makine Öğrenimi

Python ile makine öğrenimi ve derin öğrenme modelleri geliştirme

**Sebastian Raschka**
**Yuxi (Hayden) Liu**
**Vahid Mirjalili**

Packt
BIRMINGHAM-MUMBAI

"Python" ve Python Logosu, Python Software Foundation'ın ticari markalarıdır.

6587191CA584E

# PyTorch ve Scikit-Learn ile Makine Öğrenimi

Telif Hakkı © 2022 Packt Publishing

*Tüm hakları saklıdır*. Bu kitabın hiçbir bölümü, eleştirel makalelere veya incelemelere yerleştirilmiş kısa alıntılar dışında, yayıncının önceden yazılı izni olmadan çoğaltılamaz, bir erişim sisteminde saklanamaz veya herhangi bir biçimde veya herhangi bir yolla iletilemez.

Bu kitabın hazırlanmasında sunulan bilgilerin doğruluğunu sağlamak için her türlü çaba gösterilmiştir. Ancak, bu kitapta yer alan bilgiler açık veya zımni herhangi bir garanti olmaksızın satılmaktadır. Ne yazarlar ne de Packt Publishing veya bayileri ve distribütörleri, bu kitabın doğrudan veya dolaylı olarak neden olduğu veya neden olduğu iddia edilen herhangi bir zarardan sorumlu tutulamaz.

Packt Publishing, bu kitapta adı geçen tüm şirketler ve ürünler hakkında ticari marka bilgilerini uygun büyük harf kullanımıyla sağlamaya çalışmıştır. Ancak Packt Publishing bu bilgilerin doğruluğunu garanti edemez.

**Yapımcı** Tushar Gupta

**Satın Alma Editörü - Hakem Değerlendirmeleri:** Saby Dsilva

**Proje Editörü:** Janice Gonsalves

**İçerik Geliştirme Editörü:** Bhavesh Amin

**Metin Editörü:** Safis Editing

**Teknik Editör:** Aniket Shetty

Düzeltmen**:** Safis Editing

**İndeksleyici:** Tejal Daruwale Soni

**Sunum Tasarımcısı:** Pranit Padwal İlk

yayın tarihi: Şubat 2022 Üretim

referansı: 5151122

Packt Publishing Ltd. tarafından
yayınlanmıştır. Livery Place
35 Livery Street
Birmingham
B3 2PB,
Birleşik
Krallık.

ISBN 978-1-80181-931-2

www.packt.com

olimpikbarbell@gmail.com

# Önsöz

Son yıllarda, makine öğrenimi yöntemleri, büyük miktarda veriyi anlamlandırma ve kararları otomatikleştirme yetenekleriyle, sağlık, robotik, biyoloji, fizik, tüketici ürünleri, internet hizmetleri ve diğer çeşitli sektörlerde yaygın uygulamalar bulmuştur.

Bilimdeki dev sıçramalar genellikle güçlü fikirler ve harika araçların bir araya gelmesiyle ortaya çıkar. Makine öğrenimi de bir istisna değildir. Veriye dayalı öğrenme yöntemlerinin başarısı, alanın 60 yıllık geçmişi boyunca binlerce yetenekli araştırmacının dahiyane fikirlerine dayanmaktadır. Ancak son zamanlardaki popülerlikleri, onları ölçeklenebilir ve erişilebilir kılan donanım ve yazılım çözümlerinin evriminden de besleniyor. NumPy ve scikit-learn gibi Python etrafında inşa edilen sayısal hesaplama, veri analizi ve makine öğrenimi için mükemmel kütüphanelerin ekosistemi, araştırma ve endüstride geniş bir benimseme kazanmıştır. Bu, Python'un en popüler programlama dili olmasına büyük ölçüde yardımcı oldu.

Bilgisayarla görme, metin, konuşma ve diğer görevlerde son zamanlarda derin öğrenme tekniklerinin getirdiği muazzam gelişmeler bu temayı örneklendirmektedir. Yaklaşımlar, GPU'lar ve yüksek düzeyde optimize edilmiş hesaplama rutinleri ile birlikte dikkate değer ölçüde iyi çalışmaya başlayan son kırk yılın sinir ağı teorisine dayanmaktadır.

Geçtiğimiz beş yıl boyunca PyTorch'u oluştururken amacımız, araştırmacılara derin öğrenme algoritmalarını ifade etmek için en esnek aracı sunmak ve aynı zamanda altta yatan mühendislik karmaşıklıklarıyla ilgilenmek oldu. Mükemmel Python ekosisteminden faydalandık. Buna karşılık, çok yetenekli insanlardan oluşan bir topluluğun PyTorch üzerinde çeşitli alanlarda gelişmiş derin öğrenme modelleri oluşturduğunu gördüğümüz için şanslıyız. Bu kitabın yazarları da onlar arasındaydı.

Sebastian'ı bu sıkı sıkıya bağlı topluluk içinde birkaç yıldır tanıyorum. Bilgiyi kolayca açıklama ve karmaşık olanı erişilebilir kılma konusunda eşsiz bir yeteneğe sahip. Sebastian, yaygın olarak kullanılan birçok makine öğrenimi yazılım paketine katkıda bulundu ve derin öğrenme ve veri görselleştirme üzerine düzinelerce mükemmel öğretici yazı yazdı.

Makine öğrenimini pratikte uygulamak için hem fikirlere hem de araçlara hakim olmak gerekir. Başlamak, teorik kavramları anlamlandırmaktan hangi yazılım paketlerinin yükleneceğini bulmaya kadar göz korkutucu gelebilir.

6587191CA584E

Neyse ki elinizde tuttuğunuz kitap, bu yolculukta size rehberlik etmek için makine öğrenimi kavramlarını ve pratik mühendislik adımlarını birleştirerek güzel bir iş çıkarıyor. Veri güdümlü tekniklerin temellerinden en yeni derin öğrenme mimarilerine kadar keyifli bir yolculuk sizi bekliyor. Her bölümde, tanıtılan yöntemleri pratik bir göreve uygulayan somut kod örnekleri bulacaksınız.

İlk baskı 2015 yılında çıktığında, *ML ve Python* kitap kategorisi için çok yüksek bir çıta belirledi. Ancak mükemmellik bununla sınırlı kalmadı. Her baskıda Sebastian ve ekibi, derin öğrenme devrimi yeni alanlarda ortaya çıktıkça materyali geliştirmeye ve iyileştirmeye devam etti. Bu yeni PyTorch baskısında, dönüştürücü mimarileri ve çizge sinir ağları hakkında yeni bölümler bulacaksınız. Bu yaklaşımlar derin öğrenmenin en ileri noktasındadır ve son iki yılda metin anlama ve moleküler yapı alanlarında fırtına gibi esmiştir. Hugging Face, PyTorch Lightning ve PyTorch Geometric gibi ekosistemdeki yeni ancak yaygın olarak popüler yazılım paketlerini kullanarak bunları uygulayacaksınız.

Bu kitabın teori ve pratik arasında kurduğu mükemmel denge, yazarların ileri düzey araştırma uzmanlığı ve uygulamalı problem çözme deneyimlerinin bir araya gelmesiyle ortaya çıkan bir sürpriz değildir. Sebastian Raschka ve Vahid Mirjalili, bilgisayarla görme ve hesaplamalı biyoloji için derin öğrenme araştırmalarındaki geçmişlerinden yararlanmaktadır. Hayden Liu, makine öğrenimi yöntemlerini olay tahmini, tavsiye sistemleri ve sektördeki diğer görevlere uygulama deneyimini getiriyor. Tüm yazarlar eğitim için derin bir tutkuyu paylaşıyor ve bu, kitabın basitten ileri seviyeye doğru yaklaşılabilir bir şekilde ilerlemesine yansıyor.

Bu kitabı hem heyecan verici makine öğrenimi alanına geniş bir genel bakış hem de pratik bilgiler hazinesi olarak çok değerli bulacağınıza eminim. Umarım, sorun alanınız ne olursa olsun, makine öğrenimini daha iyi bir şekilde uygulamanız için size ilham verir.

*Dmytro Dzhulgakov*

*PyTorch Çekirdek*

*Bakımcısı*

olimpikbarbell@gmail.com

# Katkıda Bulunanlar

## Yazarlar hakkında

**Dr. Sebastian Raschka,** Wisconsin-Madison Üniversitesi'nde makine öğrenimi ve derin öğrenme üzerine odaklanan bir İstatistik Doçentidir. Son araştırmaları, sınırlı verilerle çalışmak için birkaç atışla öğrenme ve sıralı hedefler için derin sinir ağları geliştirme gibi genel zorluklara odaklanmıştır. Sebastian aynı zamanda hevesli bir açık kaynak katılımcısıdır ve Grid.ai'deki Baş Yapay Zeka Eğitmeni olarak yeni görevinde, insanların makine öğrenimi ve yapay zekaya girmelerine yardımcı olma tutkusunu takip etmeyi planlamaktadır.

---

*Dönüşümler ve çizge sinir ağları ile ilgili yeni bölümler üzerinde birlikte çalışmaktan memnuniyet duyduğum Jitian Zhao ve Ben Kaufman'a çok teşekkür ederim. Ayrıca Hayden ve Vahid'in yardımları için de minnettarım; sizler olmasaydınız bu kitap mümkün olmazdı. Son olarak, Andrea Panizza, Tony Gitter ve Adam Bielski'ye makalenin bazı bölümleri üzerindeki yararlı tartışmaları için teşekkür etmek istiyorum.*

---

**Yuxi (Hayden) Liu,** Google'da makine öğrenimi yazılım mühendisidir ve çeşitli veri odaklı alanlarda makine öğrenimi bilimcisi olarak çalışmıştır. Hayden, bir dizi makine öğrenimi kitabının yazarıdır. İlk kitabı *Python Machine Learning By Example*, 2017 ve 2018 yıllarında Amazon'da kendi kategorisinde en çok satanlar listesinde 1. sırada yer almış ve birçok dile çevrilmiştir. Diğer kitapları arasında R Deep Learning Projects, *Hands-On Deep Learning Architectures with Python* ve *PyTorch 1.x Reinforcement Learning Cookbook bulunmaktadır*.

---

*Birlikte çalıştığım tüm harika insanlara, özellikle de ortak yazarlarıma, Packt'teki editörlerime ve hakemlerime teşekkür etmek istiyorum. Onlar olmasaydı bu kitabı okumak ve gerçek dünya sorunlarına uygulamak daha zor olurdu. Son olarak, beni bu çok satan ML kitabının PyTorch baskısını yazmaya teşvik eden tüm okuyuculara destekleri için teşekkür etmek istiyorum.*

---

6587191CA584E

**Dr. Vahid Mirjalili** bilgisayarla görme uygulamalarına odaklanan bir derin öğrenme araştırmacısıdır. Va- hid, Michigan Eyalet Üniversitesi'nden hem Makine Mühendisliği hem de Bilgisayar Bilimleri alanında doktora derecesi almıştır. Doktora yolculuğu sırasında, gerçek dünya problemlerini çözmek için yeni bilgisayarla görme algoritmaları geliştirmiş ve bilgisayarla görme topluluğunda yüksek oranda atıf alan çeşitli araştırma makaleleri yayınlamıştır.

## Diğer katkıda bulunanlar

**Benjamin Kaufman,** Wisconsin-Madison Üniversitesi'nde Biyomedikal Veri Bilimi alanında doktora adayıdır. Araştırmaları, ilaç keşfi için makine öğrenimi yöntemlerinin geliştirilmesi ve uygulanmasına odaklanmaktadır. Bu alandaki çalışmaları, çizge sinir ağlarının daha iyi anlaşılmasını sağlamıştır.

**Jitian Zhao,** Wisconsin-Madison Üniversitesi'nde doktora öğrencisidir ve burada büyük ölçekli dil modellerine olan ilgisini geliştirmiştir. Hem gerçek dünya uygulamaları hem de teorik destek geliştirme konusunda derin öğrenme konusunda tutkuludur.

---

*Aileme destekleri için teşekkür etmek istiyorum. Beni her zaman hayallerimin peşinden gitmeye teşvik ettiler ve iyi bir insan olmam için beni motive ettiler.*

---

olimpikbarbell@gmail.com

# Eleştirmen hakkında

**Roman Tezikov,** gelişmiş bilgisayarla görme, NLP ve MLOps alanlarında dört yılı aşkın deneyime sahip bir endüstriyel araştırma mühendisi ve derin öğrenme meraklısıdır. ML-REPA topluluğunun ortak yaratıcısı olarak, ML'nin yeniden üretilebilirliği ve boru hattı otomasyonu hakkında çeşitli atölye çalışmaları ve buluşmalar düzenledi. Mevcut çalışma zorluklarından biri, moda endüstrisinde bilgisayarla görmenin kullanılmasını içermektedir. Roman ayrıca hızlandırılmış derin öğrenme için bir PyTorch çerçevesi olan Catalyst'in çekirdek geliştiricisiydi.

6587191CA584E

# Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

olimpikbarbell@gmail.com

# İçindekiler

**Önsöz** xxiii

**Bölüm 1: Bilgisayarlara Verilerden Öğrenme Yeteneği Kazandırma** 1

**Verileri bilgiye dönüştürmek için akıllı makineler oluşturma** ..... 1
**Üç farklı makine öğrenimi türü** ..... 2
Denetimli öğrenme ile gelecek hakkında tahminlerde bulunma - 3
*Sınıf etiketlerini tahmin etmek için sınıflandırma - 4 Sürekli sonuçları tahmin etmek için regresyon - 5*
Pekiştirmeli öğrenme ile etkileşimli problemleri çözme - 6 Denetimsiz öğrenme ile gizli yapıları keşfetme - 7
*Kümeleme ile alt grupların bulunması - 8 Veri sıkıştırma için boyut indirgeme - 8*
**Temel terminoloji ve notasyonlara giriş** ..... 9
Bu kitapta kullanılan gösterim ve kurallar - 9
Makine öğrenimi terminolojisi - 11
**Makine öğrenimi sistemleri oluşturmak için bir yol haritası** ..... 12
Ön işleme - verilerin şekillendirilmesi - 13
Tahmine dayalı bir modelin eğitilmesi ve seçilmesi - 13
Modelleri değerlendirme ve görünmeyen veri örneklerini tahmin etme - 14
**Makine öğrenimi için Python kullanma** ..... 14
Python Paket Dizini'nden Python ve paketlerini yükleme - 14
Anaconda Python dağıtımını ve paket yöneticisini kullanma - 15
Bilimsel hesaplama, veri bilimi ve makine öğrenimi için paketler - 16
**Özet** ..... 17

6587191CA584E

## Bölüm 2: Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi 19

**Yapay nöronlar - makine öğreniminin erken tarihine kısa bir bakış** ..... 19
Yapay nöronun resmi tanımı - 20 Perceptron
öğrenme kuralı - 22
**Python'da bir perceptron öğrenme algoritması uygulama** ..... 25
Nesne yönelimli bir algılayıcı API - 25
Iris veri kümesi üzerinde bir perceptron modelinin eğitilmesi - 29
**Uyarlanabilir doğrusal nöronlar ve öğrenmenin yakınsaması** ..... 35
Gradyan inişi ile kayıp fonksiyonlarını minimize etme
- 37 Adaline'i Python'da uygulama - 39
Özellik ölçeklendirme yoluyla gradyan inişini iyileştirme - 43
Büyük ölçekli makine öğrenimi ve stokastik gradyan inişi - 45
**Özet** ..... 51

## Bölüm 3: Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu 53

**Bir sınıflandırma algoritması seçme** ..... 53
**Scikit-learn ile ilk adımlar - bir perceptronun eğitimi** ..... 54
**Sınıf olasılıklarının lojistik regresyon ile modellenmesi** ..... 59
Lojistik regresyon ve koşullu olasılıklar - 60 Lojistik kayıp
fonksiyonu aracılığıyla model ağırlıklarının öğrenilmesi - 63
Adaline uygulamasını lojistik regresyon için bir algoritmaya dönüştürme - 66 Scikit-
learn ile bir lojistik regresyon modelini eğitme - 70
Düzenli hale getirme yoluyla aşırı uyumla mücadele - 73
**Destek vektör makineleri ile maksimum marjlı sınıflandırma** ..... 76
Maksimum marj sezgisi - 77
Slack değişkenleri kullanarak doğrusal olmayan ayrılabilir bir durumla başa
çıkma - 77 Scikit-learn'de alternatif uygulamalar - 79
**Çekirdek DVM kullanarak doğrusal olmayan problemleri çözme** ..... 80
Doğrusal olarak ayrıştırılamayan veriler için çekirdek yöntemleri - 80
Yüksek boyutlu bir uzayda ayırıcı hiper düzlemleri bulmak için çekirdek hilesini kullanma - 82
**Karar ağacı öğrenimi** ..... 86
IG'yi maksimize etmek - paranızın karşılığını en iyi şekilde
almak - 88 Bir karar ağacı oluşturmak - 92
Rastgele ormanlar aracılığıyla çoklu karar ağaçlarının birleştirilmesi - 95
**K-en yakın komşular - tembel bir öğrenme algoritması** ..... 98
**Özet** ..... 102

olimpikbarbell@gmail.com

## Bölüm 4: İyi Eğitim Veri Setleri Oluşturma - Veri Önişleme 105

**Eksik verilerle başa çıkma** ..... 105
Tablo verilerinde eksik değerleri belirleme - 106
Eksik değerlere sahip eğitim örneklerini veya özellikleri eleme - 107
Eksik değerleri b e l i r l e m e - 108
Scikit-learn tahmin edici API'sini anlama - 109
**Kategorik verilerin işlenmesi** ..... 111
Pandas ile kategorik veri kodlama - 111 Sıralı
özellikleri eşleme - 111
Sınıf etiketlerini kodlama - 112
Nominal özellikler üzerinde tek seferlik kodlama gerçekleştirme - 113
*İsteğe bağlı: sıralı özelliklerin kodlanması - 116*
**Bir veri kümesini ayrı eğitim ve test veri kümelerine bölme** ..... 117
**Özellikleri aynı ölçeğe getirme** ..... 119
**Anlamlı özelliklerin seçilmesi** ..... 122
Model karmaşıklığına karşı cezalar olarak L1 ve L2 düzenlemesi -
122 L2 düzenlemesinin geometrik bir yorumu - 123
L1 düzenlemesi ile seyrek çözümler - 125 Sıralı
özellik seçimi algoritmaları - 128
**Rastgele ormanlar ile özellik öneminin değerlendirilmesi** ..... 134
**Özet** ..... 137

## Bölüm 5: Boyut Azaltma Yoluyla Verileri Sıkıştırma 139

**Temel bileşen analizi ile denetimsiz boyut azaltma** ..... 139
Temel bileşen analizindeki ana adımlar - 140 Temel
bileşenlerin adım adım çıkarılması - 142 Toplam ve
açıklanan varyans - 144
Özellik dönüşümü - 146
Scikit-learn'de temel bileşen analizi - 149 Özellik
katkılarının değerlendirilmesi - 152
**Doğrusal diskriminant analizi ile denetimli veri sıkıştırma** ..... 154
Doğrusal diskriminant analizine karşı temel bileşen analizi - 154
Doğrusal diskriminant analizinin iç işleyişi - 156
Dağılım matrislerinin hesaplanması - 156
Yeni özellik alt uzayı için doğrusal ayırıcıların seçilmesi - 158
Örneklerin yeni özellik uzayına yansıtılması - 161

6587191CA584E

Scikit-learn aracılığıyla LDA - 162
**Doğrusal olmayan boyut indirgeme ve görselleştirme ....163**
Neden doğrusal olmayan boyut indirgeme düşünülmelidir? - 164
Verilerin t-dağılımlı stokastik komşu gömme yoluyla görselleştirilmesi - 165
**Özet ....169**

## Bölüm 6: Model Değerlendirme için En İyi Uygulamaların Öğrenilmesi ve Hiperparametre Ayarlama 171

**İş akışlarını boru hatları ile kolaylaştırma ....171**
Meme Kanseri Wisconsin veri setinin yüklenmesi - 172
Dönüştürücülerin ve tahmin edicilerin bir ardışık düzende birleştirilmesi - 173
**Model performansını değerlendirmek için k-kat çapraz doğrulamayı kullanma ....175**
Bekletme yöntemi - 175
K-kat çapraz doğrulama - 176
**Öğrenme ve doğrulama eğrileri ile algoritmalarda hata ayıklama ....180**
Öğrenme eğrileriyle yanlılık ve varyans sorunlarını teşhis etme - 180
Doğrulama eğrileriyle aşırı ve yetersiz uyumu ele alma - 183
**Makine öğrenimi modellerine ızgara arama yoluyla ince ayar yapma ....185**
Izgara arama yoluyla hiperparametrelerin ayarlanması - 186
Rastgele arama ile hiperparametre konfigürasyonlarını daha geniş çapta keşfetme - 187
Ardışık yarılama ile daha kaynak verimli hiperparametre arama - 189 İç içe çapraz doğrulama ile algoritma seçimi - 191
**Farklı performans değerlendirme ölçütlerine bakmak ....193**
Karışıklık matrisini okuma - 193
Bir sınıflandırma modelinin kesinliğini ve geri çağırmasını optimize etme - 195 Alıcı işletim karakteristiğini çizme - 198
Çok sınıflı sınıflandırma için puanlama metrikleri - 200
Sınıf dengesizliği ile başa çıkma - 201
**Özet ....203**

## Bölüm 7: Topluluk Öğrenmesi için Farklı Modellerin Birleştirilmesi 205

**Topluluklarla öğrenme ....205**
**Sınıflandırıcıları çoğunluk oyu ile birleştirme ....209**
Basit bir çoğunluk oyu sınıflandırıcısının uygulanması - 209
Tahmin yapmak için çoğunluk oylaması ilkesini kullanma - 214
Topluluk sınıflandırıcısını değerlendirme ve ayarlama - 217

olimpikbarbell@gmail.com
**Bagging - bootstrap örneklerinden bir sınıflandırıcı topluluğu oluşturma.....223**
Özetle torbalama - 224
Şarap veri kümesindeki örnekleri sınıflandırmak için torbalama uygulaması - 225
**Uyarlanabilir güçlendirme yoluyla zayıf öğrenicilerden yararlanma .....229**
Uyarlanabilir boosting nasıl çalışır - 229
AdaBoost'u scikit-learn kullanarak
uygulama - 233
**Gradyan artırma - kayıp gradyanlarına dayalı bir topluluğun eğitimi.....237**
AdaBoost'un gradyan artırma ile karşılaştırılması - 237
Genel gradyan artırma algoritmasının ana hatları - 237
Sınıflandırma için gradyan artırma algoritmasının açıklanması - 239
Sınıflandırma için gradyan artırmanın gösterilmesi - 241
XGBoost Kullanımı - 243
**Özet.....245**

**Bölüm 8: Makine Öğrenimini Duygu Analizine Uygulama 247**

**IMDb film inceleme verilerini metin işleme için hazırlama .....247**
Film inceleme veri setinin elde edilmesi - 248
Film veri setini daha uygun bir formata dönüştürmek için ön işleme - 248
**Kelime torbası modelinin tanıtılması .....250**
Kelimeleri özellik vektörlerine dönüştürme - 250
Terim sıklığı-ters belge sıklığı aracılığıyla kelime alaka düzeyinin değerlendirilmesi -
252 Metin verilerinin temizlenmesi - 254
Belgeleri belirteçlere dönüştürme - 256
**Belge sınıflandırması için lojistik regresyon modeli eğitimi .....258**
**Daha büyük verilerle çalışma - çevrimiçi algoritmalar ve çekirdek dışı öğrenme.....260**
**Gizli Dirichlet tahsisi ile konu modelleme.....264**
Metin belgelerinin LDA ile ayrıştırılması - 264
scikit-learn ile LDA - 265
**Özet.....268**

**Bölüm 9: Regresyon Analizi ile Sürekli Hedef Değişkenlerin Tahmin Edilmesi 269**

**Doğrusal regresyonun tanıtılması .....269**
Basit doğrusal regresyon - 270
Çoklu doğrusal regresyon - 271
**Ames Konut veri setini keşfetme .....272**
Ames Konut veri kümesinin bir DataFrame'e yüklenmesi - 272

6587191CA584E

Bir veri kümesinin önemli özelliklerini görselleştirme - 274
Korelasyon matrisi kullanarak ilişkilere bakma - 276
**Sıradan en küçük kareler doğrusal regresyon modelinin uygulanması....278**
Gradyan inişi ile regresyon parametreleri için regresyon çözme - 278 Scikit-learn ile bir regresyon modelinin katsayısını tahmin etme - 283
**RANSAC kullanarak sağlam bir regresyon modeli uydurma....285**
**Doğrusal regresyon modellerinin performansının değerlendirilmesi....288**
**Regresyon için düzenlenmiş yöntemlerin kullanılması....292**
**Doğrusal bir regresyon modelini eğriye dönüştürme - polinom regresyonu....294**
Scikit-learn kullanarak polinom terimleri ekleme - 294
Ames Konut veri setinde doğrusal olmayan ilişkilerin modellenmesi - 297
**Rastgele ormanlar kullanarak doğrusal olmayan ilişkilerle başa çıkma....299**
Karar ağacı regresyonu - 300
Rastgele orman regresyonu - 301
**Özet....304**

## Bölüm 10: Etiketsiz Verilerle Çalışma - Kümeleme Analizi 305

**K-ortalamalar kullanarak nesneleri benzerliğe göre gruplama....305**
scikit-learn kullanarak k-ortalamalar kümeleme - 305
k-means++ kullanarak ilk küme merkezlerini yerleştirmenin daha akıllı bir yolu - 310 Sert ve yumuşak kümeleme - 311
Optimum küme sayısını bulmak için dirsek yönteminin kullanılması - 313
Siluet grafikleri aracılığıyla kümeleme kalitesinin ölçülmesi - 314
**Kümelerin hiyerarşik bir ağaç olarak düzenlenmesi....319**
Kümeleri aşağıdan yukarıya doğru gruplama - 320
Uzaklık matrisi üzerinde hiyerarşik kümeleme gerçekleştirme - 321 Dendrogramları ısı haritasına ekleme - 325
Scikit-learn aracılığıyla aglomeratif kümeleme uygulama - 327
**DBSCAN ile yüksek yoğunluklu bölgelerin bulunması....328**
**Özet....334**

## Bölüm 11: Sıfırdan Çok Katmanlı Yapay Sinir Ağının Uygulanması 335

**Yapay sinir ağları ile karmaşık fonksiyonların modellenmesi....335**
Tek katmanlı sinir ağı özeti - 337
Çok katmanlı sinir ağı mimarisinin tanıtılması - 338 İleri yayılım yoluyla bir sinir ağının etkinleştirilmesi - 340

olimpikbarbell@gmail.com

**El yazısı rakamların sınıflandırılması** ..... **343**
MNIST veri setinin elde edilmesi ve hazırlanması
- 343 Çok katmanlı bir algılayıcının uygulanması -
347 Sinir ağı eğitim döngüsünün kodlanması - 352
Sinir ağı performansının değerlendirilmesi - 357
**Yapay sinir ağının eğitilmesi** ..... **360**
Kayıp fonksiyonunun hesaplanması - 360
Geriye yayılım anlayışınızı geliştirme - 362 Geriye yayılım
yoluyla sinir ağlarını eğitme - 363
**Sinir ağlarında yakınsama hakkında** ..... **367**
**Sinir ağı uygulaması hakkında son birkaç söz** ..... **368**
**Özet** ..... **368**

## Bölüm 12: PyTorch ile Sinir Ağı Eğitimini Paralelleştirme 369

**PyTorch ve eğitim performansı** ..... **369**
Performans zorlukları - 369
PyTorch nedir? - 371
PyTorch'u nasıl öğreneceğiz - 372
**PyTorch ile ilk adımlar** ..... **372**
PyTorch'u Yükleme - 372
PyTorch'ta tensör oluşturma - 373
Bir tensörün veri türünü ve şeklini değiştirme - 374
Tensörlere matematiksel işlemler uygulama - 375
Tensörleri bölme, yığma ve birleştirme - 376
**PyTorch'ta girdi boru hatları oluşturma** ..... **378**
Mevcut tensörlerden bir PyTorch DataLoader oluşturma - 378 İki
tensörü ortak bir veri kümesinde birleştirme - 379
Shuffie, batch ve tekrar - 380
Yerel depolama diskinizdeki dosyalardan bir veri kümesi oluşturma - 382
torchvision.datasets kütüphanesinden kullanılabilir veri kümelerini getirme - 386
**PyTorch'ta bir NN modeli oluşturma** ..... **389**
PyTorch sinir ağı modülü (torch.nn) - 390 Doğrusal
regresyon modeli oluşturma - 390
torch.nn ve torch.optim modülleri aracılığıyla model eğitimi - 394
Iris veri kümesindeki çiçekleri sınıflandırmak için çok katmanlı bir algılayıcı
oluşturma - 395 Eğitilen modelin test veri kümesi üzerinde değerlendirilmesi -
398

6587191CA584E

Eğitilmiş modeli kaydetme ve yeniden yükleme - 399
**Çok katmanlı sinir ağları için aktivasyon fonksiyonlarının seçilmesi.....400**
Lojistik fonksiyon özeti - 400
Softmax fonksiyonu ile çok sınıflı sınıflandırmada sınıf olasılıklarının tahmin edilmesi - 402
Hiperbolik tanjant kullanarak çıktı spektrumunun genişletilmesi - 403
Doğrultulmuş lineer ünite aktivasyonu - 405
**Özet.....406**

## Bölüm 13: Daha Derine İnmek - PyTorch'un Mekaniği 409

**PyTorch'un temel özellikleri.....410**
**PyTorch'un hesaplama grafikleri.....410**
Hesaplama grafiklerini anlama - 410 PyTorch'ta
grafik oluşturma - 411
**Model parametrelerini saklamak ve güncellemek için PyTorch tensör nesneleri.....412**
**Otomatik türevlendirme yoluyla gradyanların hesaplanması.....415**
Eğitilebilir değişkenlere göre kaybın gradyanlarını hesaplama - 415
Otomatik farklılaştırmayı anlama - 416
Çekişmeli örnekler - 416
**torch.nn modülü aracılığıyla yaygın mimarilerin uygulamalarının basitleştirilmesi.....417**
nn.Sequential tabanlı modeller uygulama - 417 Kayıp
fonksiyonu seçme - 418
XOR sınıflandırma problemini çözme - 419
Model oluşturmayı nn.Module ile daha esnek hale getirme -
424 PyTorch'ta özel katmanlar yazma - 426
**Birinci proje - bir otomobilin yakıt verimliliğinin tahmin edilmesi .....431**
Özellik sütunlarıyla çalışma - 431 Bir
DNN regresyon modelini eğitme - 435
**Proje iki - MNIST el yazısı rakamlarının sınıflandırılması .....436**
**Daha üst düzey PyTorch API'leri: PyTorch-Lightning'e kısa bir giriş.....439**
PyTorch Lightning modelinin kurulması - 440
Lightning için veri yükleyicilerin kurulması - 443
PyTorch Lightning Trainer sınıfını kullanarak modeli eğitme - 444
TensorBoard kullanarak modeli değerlendirme - 445
**Özet.....449**

## Bölüm 14: Derin Evrişimli Sinir Ağları ile Görüntülerin Sınıflandırılması 451

**CNN'lerin yapı taşları.....451**

olimpikbarbell@gmail.com

CNN'leri ve özellik hiyerarşilerini anlama - 452 Ayrık
konvolüsyonlar gerçekleştirme - 454
*Tek boyutta ayrık konvolüsyonlar - 454*
*Çıktı özellik haritalarının boyutunu kontrol etmek için girdileri doldurma - 457 Konvolüsyon çıktısının boyutunu belirleme - 458*
*2B'de ayrık konvolüsyon gerçekleştirme - 459*
Alt örnekleme katmanları - 463
**Her şeyi bir araya getirme - bir CNN uygulama....464**
Birden fazla giriş veya renk kanalıyla çalışma - 464 L2
düzenlemesi ve bırakma ile bir NN'nin düzenlenmesi - 467
Sınıflandırma için kayıp fonksiyonları - 471
**PyTorch kullanarak derin bir CNN uygulama....473**
Çok katmanlı CNN mimarisi - 473 Veri
yükleme ve ön işleme - 474
torch.nn modülünü kullanarak bir CNN uygulama - 476
*PyTorch'ta CNN katmanlarını yapılandırma - 476*
*PyTorch'ta bir CNN oluşturma - 477*
**CNN kullanarak yüz görüntülerinden gülümseme sınıflandırması....482**
CelebA veri kümesinin yüklenmesi - 483
Görüntü dönüştürme ve veri artırma - 484 Bir CNN
gülümseme sınıflandırıcısının eğitimi - 490
**Özet....497**

**Bölüm 15: Tekrarlayan Sinir Ağları Kullanarak Sıralı Verilerin Modellenmesi 499**

**Sıralı verilerin tanıtılması....499**
Sıralı verilerin modellenmesi - sıra önemlidir - 500
Sıralı verilere karşı zaman serisi verileri - 500
Sıraları temsil etme - 500
Farklı sekans modelleme kategorileri - 501
**Dizileri modellemek için RNN'ler....502**
RNN'lerde veri akışını anlama - 502 Bir RNN'de
aktivasyonları hesaplama - 504

Gizli yinelemeye karşı çıktı yinelemesi - 506
Uzun menzilli etkileşimleri öğrenmenin zorlukları - 509
Uzun kısa süreli bellek hücreleri - 511
**PyTorch'ta dizi modelleme için RNN'lerin uygulanması .....513**
Birinci proje - IMDb film incelemelerinin duyarlılığını tahmin etme - 513
*Film inceleme verilerinin hazırlanması - 513*
*Cümle kodlaması için katmanların gömülmesi*
*- 517 Bir RNN modeli oluşturma - 520*
*Duygu analizi görevi için bir RNN modeli oluşturma - 521*
Proje iki - PyTorch'ta karakter düzeyinde dil modelleme - 525
*Veri kümesini ön işleme - 526*
*Karakter düzeyinde bir RNN modeli oluşturma - 531*
*Değerlendirme aşaması - yeni metin pasajları*
*oluşturma - 533*
**Özet .....537**

## Bölüm 16: Dönüştürücüler - Doğal Dil İşlemeyi İyileştirme Dikkat Mekanizmaları ile 539

**RNN'lere bir dikkat mekanizması ekleme.....540**
Dikkat RNN'lerin bilgiye erişmesine yardımcı olur - 540
RNN'ler için orijinal dikkat mekanizması - 542 Çift
yönlü bir RNN kullanarak girdileri işleme - 543 Bağlam
vektörlerinden çıktılar üretme - 543 Dikkat ağırlıklarını
hesaplama - 544
**Öz-dikkat mekanizmasının tanıtılması .....544**
Kendine dikkat etmenin temel bir biçimiyle başlamak - 545
Öz-dikkat mekanizmasının parametrelendirilmesi: ölçeklendirilmiş nokta-çarpım dikkati - 549
**İhtiyacımız olan tek şey dikkat: orijinal transformatör mimarisinin tanıtılması .....552**
Çok kafalı dikkat yoluyla bağlam katıştırmalarını kodlama - 554
Bir dil modeli öğrenme: kod çözücü ve maskelenmiş çoklu kafa dikkati -
558 Uygulama ayrıntıları: konumsal kodlamalar ve katman normalizasyonu
- 559
**Etiketsiz verilerden yararlanarak büyük ölçekli dil modelleri oluşturma.....561**
Dönüştürücü modellerinin ön eğitimi ve ince ayarı - 561
GPT ile etiketlenmemiş verilerden yararlanma - 563
Yeni metin oluşturmak için GPT-2 kullanımı -
566 BERT ile çift yönlü ön eğitim - 569 Her
iki dünyanın en iyisi: BART - 572

dilimpikbarbell@gmail.com

**PyTorch'ta bir BERT modeline ince ayar yapma ..... 574**
IMDb film incelemesi veri kümesinin yüklenmesi
- 575 Veri kümesinin simgeleştirilmesi - 577
Önceden eğitilmiş bir BERT modelini yükleme ve ince ayar yapma - 578
Trainer API kullanarak bir transformatöre daha kolay ince ayar yapma - 582
**Özet ..... 586**

**Bölüm 17: Yeni Verilerin Sentezlenmesi için Üretken Çekişmeli Ağlar 589**

**Üretken çekişmeli ağların tanıtılması ..... 589**
Otomatik kodlayıcılarla başlangıç - 590
Yeni verilerin sentezlenmesi için üretken modeller - 592
GAN'lar ile yeni örneklerin üretilmesi - 593
Bir GAN modelindeki üretici ve ayırıcı ağların kayıp fonksiyonlarını anlama - 594
**Sıfırdan bir GAN uygulamak ..... 596**
Google Colab üzerinde GAN modellerinin eğitimi - 596
Üreteç ve ayırıcı ağların uygulanması - 600 Eğitim veri setinin tanımlanması - 604
GAN modelinin eğitimi - 605
**Konvolüsyonel ve Wasserstein GAN kullanarak sentezlenmiş görüntülerin kalitesinin iyileştirilmesi 612**
Transpoze konvolüsyon - 612
Toplu normalizasyon - 614
Üreteç ve ayırıcının uygulanması - 616 İki dağılım arasındaki farklılık ölçütleri - 624 GAN'lar için pratikte
EM mesafesinin kullanılması - 627
Gradyan cezası - 628
DCGAN modelini eğitmek için WGAN-GP'nin uygulanması - 629 Mod çöküşü - 633
**Diğer GAN uygulamaları ..... 635**
**Özet ..... 635**

**Bölüm 18: Bağımlılıkları Yakalamak için Çizge Sinir Ağları Grafik Yapılı Verilerde 637**

**Grafik verilerine giriş ..... 638**
Yönlendirilmemiş Çizgeler
- 638 Yönlendirilmiş
Çizgeler - 639

6587191CA584E


Etiketli grafikler - 640
Moleküllerin grafik olarak gösterilmesi - 640
**Grafik konvolüsyonlarını anlama ....641**
Çizge konvolüsyonlarının kullanılmasının ardındaki motivasyon - 641 Temel bir çizge konvolüsyonunun uygulanması - 644
**PyTorch'ta sıfırdan bir GNN uygulama ....648**
NodeNetwork modelinin tanımlanması - 649
NodeNetwork'ün grafik konvolüsyon katmanının kodlanması - 650
Değişken grafik boyutlarıyla başa çıkmak için küresel bir havuzlama katmanı ekleme - 652 DataLoader'ı hazırlama - 655
Tahminler yapmak için NodeNetwork'ü kullanma - 658
**PyTorch Geometrik kütüphanesini kullanarak bir GNN uygulama ....659**
**Diğer GNN katmanları ve son gelişmeler ....665**
Spektral grafik konvolüsyonları - 665
Havuzlama - 667
Normalleştirme - 668
Gelişmiş çizge sinir ağı literatürüne işaretçiler - 669
**Özet ....671**

## Bölüm 19: Karar Verme Sürecinde Takviyeli Öğrenme Karmaşık Ortamlar 673

**Giriş - deneyimlerden öğrenme ....674**
Pekiştirmeli öğrenmeyi anlama - 674
Bir pekiştirmeli öğrenme sisteminin etmen-çevre arayüzünün tanımlanması - 675
**RL'nin teorik temelleri ....676**
Markov karar süreçleri - 677
*Markov karar süreçlerinin matematiksel formülasyonu - 677 Bir Markov sürecinin görselleştirilmesi - 679*
Epizodik ve sürekli görevler - 679
RL terminolojisi: getiri, politika ve değer fonksiyonu - 680
*Geri dönüş - 680*
*Politika - 682*
*Değer fonksiyonu - 682*
Bellman denklemini kullanarak dinamik programlama - 684

dilimpikbarbell@gmail.com

**Takviyeli öğrenme algoritmaları .....684**
Dinamik programlama - 685
*Politika değerlendirmesi - dinamik programlama ile değer fonksiyonunun tahmin edilmesi - 686 Tahmin edilen değer fonksiyonunu kullanarak politikanın iyileştirilmesi - 686*
*Politika yinelemesi - 687*
*Değer yinelemesi - 687*
Monte Carlo ile pekiştirmeli öğrenme - 687 MC kullanarak *durum-değer fonksiyonu tahmini - 688 MC kullanarak eylem-değer fonksiyonu tahmini - 688 MC kontrolü kullanarak optimal bir politika bulma - 688*
*Politika iyileştirme - eylem-değer fonksiyonundan açgözlü politikanın hesaplanması - 689*
Zamansal fark öğrenimi - 689
*TD tahmini - 689*
*Poliçe üzerinde TD kontrolü (SARSA) - 691*
*Politika dışı TD kontrolü (Q-öğrenme) - 691*
**İlk RL algoritmamızın uygulanması.....691**
OpenAI Gym araç setinin tanıtımı - 692
*OpenAI Gym'de mevcut ortamlarla çalışma - 692 Bir grid dünyası örneği - 694*
*OpenAI Gym'de ızgara dünya ortamının uygulanması - 694*
Q-öğrenme ile ızgara dünya problemini çözme - 701
**Derin Q-öğrenmeye bir bakış.....706**
Q-öğrenme algoritmasına göre bir DQN modelinin eğitilmesi - 706
*Tekrar oynatma belleği - 707*
*Kaybın hesaplanması için hedef değerlerin belirlenmesi - 708*
Derin bir Q-öğrenme algoritmasının uygulanması - 710
**Bölüm ve kitap özeti .....714**

**Hoşunuza Gidebilecek Diğer Kitaplar 719**

**Dizin 723**

6587191CA584E

olimpikbarbell@gmail.com

# Önsöz

Haberleri ve sosyal medyayı takip ettiğinizde, makine öğreniminin çağımızın en heyecan verici teknolojilerinden biri haline geldiğini muhtemelen biliyorsunuzdur. Microsoft, Google, Meta, Apple, Amazon, IBM ve çok daha fazlası gibi büyük şirketler, makine öğrenimi araştırmalarına ve uygulamalarına iyi nedenlerle yoğun bir şekilde yatırım yapmaktadır. Makine öğrenimi günümüzün ve çağımızın moda sözcüğü haline gelmiş gibi görünse de, bu kesinlikle bir aldatmaca değildir. Bu heyecan verici alan yeni olasılıkların önünü açıyor ve günlük hayatımızın vazgeçilmezi haline geldi. Akıllı telefonlarımızdaki sesli asistanla konuşmak, müşterilerimiz için doğru ürünü önermek, kredi kartı dolandırıcılığını önlemek, e-posta gelen kutularımızdaki spam'leri filtrelemek, tıbbi hastalıkları tespit ve teşhis etmek, liste uzayıp gidiyor.

Eğer bir makine öğrenimi uygulayıcısı, daha iyi bir problem çözücü olmak istiyorsanız, hatta makine öğrenimi araştırmalarında kariyer yapmayı düşünüyorsanız, bu kitap tam size göre! Ancak yeni başlayanlar için makine öğreniminin arkasındaki teorik kavramlar oldukça zorlayıcı olabilir. Yine de, son yıllarda yayınlanan birçok pratik kitap, güçlü öğrenme algoritmalarını uygulayarak makine öğrenimine başlamanıza yardımcı olacaktır.

Pratik kod örneklerine maruz kalmak ve makine öğreniminin örnek uygulamaları üzerinde çalışmak, bu alana dalmak için harika bir yoldur. Somut örnekler, öğrenilen materyali doğrudan eyleme geçirerek daha geniş kavramları göstermeye yardımcı olur. Ancak, büyük gücün büyük sorumluluk getirdiğini unutmayın! Python ve Python tabanlı makine öğrenimi kütüphanelerini kullanarak makine öğrenimi ile ilgili uygulamalı deneyim sunmanın yanı sıra, bu kitap makine öğrenimini başarılı bir şekilde kullanmak için gerekli olan makine öğrenimi algoritmalarının arkasındaki matematiksel kavramları da tanıtmaktadır. Bu nedenle, bu kitap tamamen pratik bir kitaptan farklıdır; makine öğrenimi kavramlarıyla ilgili gerekli ayrıntıları tartışan, makine öğrenimi algoritmalarının nasıl çalıştığı, nasıl kullanılacağı ve en önemlisi en yaygın tuzaklardan nasıl kaçınılacağı konusunda sezgisel ancak bilgilendirici açıklamalar sunan bir kitaptır.

Bu kitapta, bu alanda bir başlangıç yapmanızı sağlayacak tüm temel konu ve kavramları kapsayan heyecan verici bir yolculuğa çıkacağız. Bilgiye olan açlığınızın tatmin olmadığını fark ederseniz, bu kitap bu alandaki temel atılımları takip etmek için kullanabileceğiniz birçok yararlı kaynağa atıfta bulunmaktadır.

# Bu kitap kimler için

Bu kitap, makine öğrenimi ve derin öğrenmenin çok çeşitli görevlere ve veri kümelerine nasıl uygulanacağını öğrenmek için ideal bir arkadaştır. Teknolojideki son trendlere ayak uydurmak isteyen bir programcıysanız, bu kitap kesinlikle sizin için. Ayrıca, öğrenciyseniz veya kariyerinizde bir geçiş yapmayı düşünüyorsanız, bu kitap makine öğrenimi dünyasına hem bir giriş hem de kapsamlı bir rehber olacaktır.

# Bu kitap neleri kapsıyor

*Bilgisayarlara Verilerden Öğrenme Yeteneği Kazandırma* başlıklı *1. Bölüm,* çeşitli problem görevlerinin üstesinden gelmek için makine öğreniminin ana alt alanlarını tanıtmaktadır. Buna ek olarak, sonraki bölümlerde bize rehberlik edecek tipik bir makine öğrenimi modeli oluşturma hattının oluşturulması için gerekli adımları tartışmaktadır.

*Bölüm 2*, *Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi, makine öğreniminin* kökenlerine geri dönmekte ve ikili algılayıcı sınıflandırıcıları ve uyarlanabilir doğrusal nöronları tanıtmaktadır. Bu bölüm, örüntü sınıflandırmanın temellerine nazik bir giriş niteliğindedir ve optimizasyon algoritmaları ile makine öğrenimi arasındaki etkileşime odaklanmaktadır.

*Bölüm 3*, *Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu,* sınıflandırma için temel makine öğrenimi algoritmalarını açıklamakta ve en popüler ve kapsamlı açık kaynaklı makine öğrenimi kütüphanelerinden biri olan scikit-learn'i kullanarak pratik örnekler sunmaktadır.

*Bölüm 4*, *İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme,* eksik veriler gibi işlenmemiş veri kümelerindeki en yaygın sorunlarla nasıl başa çıkılacağını tartışmaktadır. Ayrıca, veri kümelerindeki en bilgilendirici özellikleri belirlemek için çeşitli yaklaşımlar tartışılmakta ve makine öğrenimi algoritmaları için uygun girdiler olarak farklı türlerdeki değişkenlerin nasıl hazırlanacağı öğretilmektedir.

*Bölüm 5*, *Boyutluluk Azaltma Yoluyla Verilerin Sıkıştırılması*, bir veri kümesindeki özelliklerin sayısını, yararlı ve ayırt edici bilgilerinin çoğunu koruyarak daha küçük kümelere indirgemek için gerekli teknikleri açıklamaktadır. Temel bileşen analizi yoluyla boyut azaltmaya yönelik standart yaklaşımı tartışmakta ve bunu denetimli ve doğrusal olmayan dönüştürme teknikleriyle karşılaştırmaktadır.

*6. Bölüm*, *Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Öğrenme Uygulamaları,* tahmine dayalı modellerin performanslarını tahmin etmek için yapılması ve yapılmaması gerekenleri tartışmaktadır. Ayrıca, modellerimizin performansını ölçmek için farklı metrikler ve makine öğrenimi algoritmalarına ince ayar yapmak için teknikler tartışılmaktadır.

*Bölüm 7*, *Topluluk Öğrenimi için Farklı Modelleri Birleştirme,* size birden fazla öğrenme algoritmasını etkili bir şekilde birleştirmenin farklı kavramlarını tanıtır. Bireysel öğrenicilerin zayıflıklarının üstesinden gelmek için uzman topluluklarının nasıl oluşturulacağını öğreterek daha doğru ve güvenilir tahminler elde etmenizi sağlar.

*Makine Öğrenimini Duygu Analizine Uygulama* başlıklı *8. Bölümde,* makine öğrenimi algoritmalarının insanların yazılarına dayalı görüşlerini tahmin edebilmesi için metinsel verileri anlamlı temsillere dönüştürmeye yönelik temel adımlar ele alınmaktadır.

*Bölüm 9*, *Regresyon Analizi ile Sürekli Hedef Değişkenlerin Tahmini,* sürekli bir ölçekte tahminler yapmak için hedef ve yanıt değişkenleri arasındaki doğrusal ilişkileri modellemeye yönelik temel teknikleri tartışmaktadır. Farklı doğrusal modeller tanıtıldıktan sonra, polinom regresyon ve ağaç tabanlı yaklaşımlardan da bahsedilmektedir.

*Bölüm 10*, *Etiketsiz Verilerle Çalışma - Kümeleme Analizi,* odağı makine öğreniminin farklı bir alt alanı olan denetimsiz öğrenmeye kaydırmaktadır. Belirli bir benzerlik derecesini paylaşan nesne gruplarını bulmak için üç temel kümeleme algoritması ailesinden algoritmalar uyguluyoruz.

*Bölüm 11*, *Sıfırdan Çok Katmanlı Yapay Sinir Ağı Uygulaması*, Python'da popüler geri yayılım algoritmasına dayalı güçlü, çok katmanlı sinir ağları oluşturmak için ilk olarak *Bölüm 2, Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi'nde* tanıttığımız gradyan tabanlı optimizasyon kavramını genişletmektedir.

*Bölüm 12*, *PyTorch ile Sinir Ağı Eğitimini Paralelleştirme*, sinir ağlarını daha verimli bir şekilde eğitmek için size pratik bir rehber sağlamak üzere önceki bölümdeki bilgiler üzerine inşa edilmiştir. Bu bölümün odak noktası, modern GPU'ların çoklu çekirdeklerini kullanmamıza ve kullanıcı dostu ve esnek bir API aracılığıyla ortak yapı taşlarından derin sinir ağları oluşturmamıza olanak tanıyan açık kaynaklı bir Python kütüphanesi olan PyTorch'tur.

*Bölüm 13*, *Going Deeper - The Mechanics of PyTorch, bir* önceki bölümün kaldığı yerden devam ediyor ve PyTorch'un daha gelişmiş kavramlarını ve işlevlerini tanıtıyor. PyTorch olağanüstü geniş ve sofistike bir kütüphanedir ve bu bölüm size dinamik hesaplama grafikleri ve otomatik farklılaştırma gibi kavramlar üzerinden yol gösterir. Ayrıca, karmaşık sinir ağlarını uygulamak için PyTorch'un nesne yönelimli API'sini nasıl kullanacağınızı ve PyTorch Lightning'in en iyi uygulamalar ve şablon kodunu en aza indirme konusunda size nasıl yardımcı olduğunu öğreneceksiniz.

*Bölüm 14*, *Derin Evrişimli Sinir Ağları ile Görüntülerin Sınıflandırılması,* **evrişimli sinir ağlarını** (**CNN'ler**) tanıtmaktadır. CNN, görüntü veri kümeleriyle çalışmak için özellikle uygun olan belirli bir derin sinir ağı mimarisi türünü temsil eder. Geleneksel yaklaşımlara kıyasla üstün performansları nedeniyle, CNN'ler artık çeşitli görüntü tanıma görevlerinde en gelişmiş sonuçları elde etmek için bilgisayarla görmede yaygın olarak kullanılmaktadır. Bu bölüm boyunca, konvolüsyonel katmanların görüntü sınıflandırması için güçlü özellik çıkarıcılar olarak nasıl kullanılabileceğini öğreneceksiniz.

*Bölüm 15*, *Tekrarlayan Sinir Ağları Kullanarak Sıralı Verilerin Modellenmesi*, derin öğrenme için özellikle metin ve diğer sıralı veri türleri ve zaman serisi verileriyle çalışmak için çok uygun olan bir başka popüler sinir ağı mimarisini tanıtmaktadır. Bir ısınma egzersizi olarak, bu bölümde film eleştirilerinin duyarlılığını tahmin etmek için tekrarlayan sinir ağları tanıtılmaktadır. Daha sonra, tekrarlayan ağlara tamamen yeni bir metin oluşturmak için kitaplardaki bilgileri sindirmeyi öğreteceğiz.

*Bölüm 16*, *Dönüştürücüler - Dikkat Mekanizmaları ile Doğal Dil İşlemenin Geliştirilmesi*, doğal dil işlemedeki en son trendlere odaklanmakta ve dikkat mekanizmalarının uzun dizilerdeki karmaşık ilişkilerin modellenmesine nasıl yardımcı olduğunu açıklamaktadır. Bu bölümde özellikle etkili dönüştürücü mimarisi ve BERT ve GPT gibi son teknoloji dönüştürücü modelleri açıklanmaktadır.

6587191CA584E


*Bölüm 17*, *Yeni Verilerin Sentezlenmesi için Üretken Çekişmeli Ağlar,* yeni, gerçekçi görünümlü görüntüler üretmek için kullanılabilecek sinir ağları için popüler bir çekişmeli eğitim rejimini tanıtmaktadır. Bölüm, veri sıkıştırma için kullanılabilen özel bir sinir ağı mimarisi türü olan oto kodlayıcılara kısa bir girişle başlamaktadır. Bölüm daha sonra bir otomatik kodlayıcının kod çözücü kısmının gerçek ve sentezlenmiş görüntüler arasında ayrım yapabilen ikinci bir sinir ağı ile nasıl birleştirileceğini göstermektedir. İki sinir ağının düşmanca bir eğitim yaklaşımında birbirleriyle rekabet etmesine izin vererek, yeni el yazısı rakamları üreten üretken bir düşmanca ağ uygulayacaksınız.

*Bölüm 18*, *Grafik Yapılı Verilerde Bağımlılıkları Yakalamak için Grafik Sinir Ağları,* tablo veri kümeleri, görüntüler ve metinlerle çalışmanın ötesine geçmektedir. Bu bölümde, sosyal medya ağları ve moleküller gibi grafik yapılı veriler üzerinde çalışan grafik sinir ağları tanıtılmaktadır. Graf konvolüsyonlarının temellerini açıkladıktan sonra, bu bölüm moleküler veriler için tahmin modellerinin nasıl uygulanacağını gösteren bir öğretici içermektedir.

*Bölüm 19*, *Karmaşık Ortamlarda Karar Verme için Takviyeli Öğrenme*, robotları ve diğer otonom sistemleri eğitmek için yaygın olarak kullanılan bir makine öğrenimi alt kategorisini kapsamaktadır. Bu bölüm, etmen/çevre etkileşimlerine, RL sistemlerinin ödül sürecine ve deneyimlerden öğrenme kavramına aşina olmak için **pekiştirmeli öğrenmenin** (RL) temellerini tanıtarak başlar. RL'nin ana kategorilerini öğrendikten sonra, Q-öğrenme algoritmasını kullanarak bir ızgara dünya ortamında gezinebilen bir ajan uygulayacak ve eğiteceksiniz. Son olarak, bu bölümde Q-öğrenmenin derin sinir ağları kullanan bir çeşidi olan derin Q-öğrenme algoritması tanıtılmaktadır.

# Bu kitaptan en iyi şekilde yararlanmak için

İdeal olarak, çeşitli algoritmaları ve modelleri hem göstermek hem de uygulamak için verdiğimiz kod örneklerini takip etmek için Python'da programlama konusunda zaten rahatsınız. Bu kitaptan en iyi şekilde yararlanmak için, matematiksel notasyonun sağlam bir şekilde kavranması da yardımcı olacaktır.

Sıradan bir dizüstü veya masaüstü bilgisayar bu kitaptaki kodların çoğunu çalıştırmak için yeterli olacaktır ve ilk bölümde Python ortamınız için talimatlar veriyoruz. Daha sonraki bölümlerde, ihtiyaç duyulduğunda ek kütüphaneler ve kurulum önerileri tanıtılacaktır.

Yeni bir **grafik işlem birimi** (**GPU**) daha sonraki derin öğrenme bölümlerinde kod çalışma zamanlarını hızlandırabilir. Bununla birlikte, bir GPU gerekli değildir ve ücretsiz bulut yeniden kaynaklarını kullanmak için talimatlar da veriyoruz.

# Örnek kod dosyalarını indirin

Tüm kod örnekleri GitHub üzerinden https://github.com/rasbt/machine- learning-book adresinden indirilebilir. Ayrıca zengin kitap ve video kataloğumuzdan başka kod paketlerimiz de https://github.com/PacktPublishing/ adresinde mevcuttur. Onlara da göz atın!

Kodu etkileşimli olarak çalıştırmak için Jupyter Notebook kullanmanızı tavsiye etsek de, tüm kod örnekleri hem bir Python betiğinde (örneğin, ch02/ch02.py) hem de bir Jupyter Notebook formatında (örneğin, ch02/ch02.ipynb) mevcuttur. Ayrıca, ek bilgi ve güncellemeler için her bir bölüme eşlik eden README.md dosyasını incelemenizi öneririz

## Renkli görselleri indirin

Ayrıca bu kitapta kullanılan ekran görüntülerinin/diyagramların renkli görüntülerini içeren bir PDF dosyası da sunuyoruz. Buradan indirebilirsiniz: https://static.packt-cdn.com/downloads/9781801819312_ColorImages. pdf. Ayrıca, daha düşük çözünürlüklü renkli görüntüler, bu kitabın örnek kod dosyalarıyla birlikte gelen kod not defterlerine gömülüdür.

## Konvansiyonlar

Bu kitap boyunca kullanılan bir dizi metin kuralı vardır.

İşte bu stillerin bazı örnekleri ve anlamlarının açıklaması. Metin içindeki kod sözcükleri aşağıdaki gibi gösterilir: "Ve zaten kurulu olan paketler --upgrade bayrağı ile güncellenebilir."

Bir kod bloğu aşağıdaki gibi ayarlanır:

```
def init (self, eta=0.01, n_iter=50, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state = random_state
```

Python yorumlayıcısındaki herhangi bir girdi aşağıdaki gibi yazılır (>>> sembolüne dikkat edin). Beklenen çıktı >>> sembolü olmadan gösterilecektir:

```
>> v1 = np.array([1, 2, 3])
>> v2 = 0,5 * v1
>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...                np.linalg.norm(v2))
0.0
```

Herhangi bir komut satırı girdisi veya çıktısı aşağıdaki gibi yazılır:

```
pip install gym==0.20
```

**Yeni terimler** ve **önemli sözcükler** kalın olarak gösterilir. Ekranda, örneğin menülerde veya iletişim kutularında gördüğünüz sözcükler metinde aşağıdaki gibi görünür: "**İleri** düğmesine tıkladığınızda bir sonraki ekrana geçersiniz."

Uyarılar veya önemli notlar aşağıdaki gibi bir kutuda görünür.

İpuçları ve püf noktaları bu şekilde görünür.

6587191CA584E


# İletişime geçin

Okuyucularımızın geri bildirimlerine her zaman açığız.

**Genel geri bildirim**: feedback@packtpub.com adresine e-posta gönderin ve mesajınızın konusunda kitabın başlığından bahsedin. Bu kitabın herhangi bir yönü hakkında sorularınız varsa, lütfen questions@packtpub. com adresinden bize e-posta gönderin.

**Hatalar**: İçeriğimizin doğruluğunu sağlamak için her türlü özeni göstermemize rağmen, hatalar olabilir. Bu kitapta bir hata bulduysanız, bunu bize bildirirseniz minnettar oluruz. Lütfen http://www.packtpub.com/submit-errata adresini ziyaret edin, kitabınızı seçin, Errata Gönderim Formu bağlantısına tıklayın ve ayrıntıları girin.

**Korsanlık**: Çalışmalarımızın herhangi bir şekilde internette yasadışı kopyalarına rastlarsanız, bize konum adresini veya web sitesi adını bildirirseniz minnettar oluruz. Lütfen materyalin linki ile birlikte copyright@packtpub.com adresinden bize ulaşın.

**Eğer yazar olmakla ilgileniyorsanız**: Uzman olduğunuz bir konu varsa ve bir kitap yazmak ya da bir kitaba katkıda bulunmakla ilgileniyorsanız, lütfen http://authors.packtpub.com adresini ziyaret edin.

# Düşüncelerinizi paylaşın

*PyTorch ve Scikit-Learn ile Makine Öğrenimi'ni* okuduktan sonra, düşüncelerinizi duymak isteriz! Lütfen
Bu kitabın Amazon inceleme sayfasına doğrudan gitmek ve görüşlerinizi paylaşmak için buraya tıklayın.

Değerlendirmeniz bizim ve teknoloji topluluğu için önemlidir ve mükemmel kalitede içerik sunduğumuzdan emin olmamıza yardımcı olacaktır.

# Bu kitabın Ücretsiz PDF kopyasını indirin

Bu kitabı satın aldığınız için teşekkürler!

Hareket halindeyken kitap okumayı seviyorsunuz ancak basılı kitaplarınızı her yere taşıyamıyor musunuz? Satın aldığınız e-Kitap seçtiğiniz cihazla uyumlu değil mi?

Endişelenmeyin, artık her Packt kitabıyla birlikte o kitabın DRM'siz PDF sürümüne hiçbir ücret ödemeden sahip olacaksınız.

Her yerde, her yerde, her cihazda okuyun. Favori teknik kitaplarınızdaki kodları arayın, kopyalayın ve doğrudan uygulamanıza yapıştırın.

Avantajlar bununla da bitmiyor, her gün gelen kutunuzda indirimlere, haber bültenlerine ve harika ücretsiz içeriklere özel erişim elde edebilirsiniz

Avantajlardan yararlanmak için bu basit adımları izleyin:

1. QR kodunu tarayın veya aşağıdaki bağlantıyı ziyaret edin

https://packt.link/free-ebook/9781801819312

2. Satın alma belgenizi gönderin
3. İşte bu kadar! Ücretsiz PDF'inizi ve diğer avantajlarınızı doğrudan e-postanıza göndereceğiz

6587191CA584E

olimpikbarbell@gmail.com

# 1

# Bilgisayarlara Verilerden Öğrenme Yeteneği Kazandırma

Verileri anlamlandıran algoritmaların uygulaması ve bilimi olan **makine öğrenimi,** bence tüm bilgisayar bilimleri arasında en heyecan verici alan! Verinin bolca geldiği bir çağda yaşıyoruz; makine öğrenimi alanından kendi kendine öğrenen algoritmaları kullanarak bu verileri bilgiye dönüştürebiliriz. Son yıllarda geliştirilen birçok güçlü açık kaynak kütüphanesi sayesinde, makine öğrenimi alanına girmek ve verilerdeki örüntüleri tespit etmek ve gelecekteki olaylar hakkında tahminlerde bulunmak için güçlü algoritmaların nasıl kullanılacağını öğrenmek için muhtemelen daha iyi bir zaman olmamıştır.

Bu bölümde, makine öğreniminin ana kavramları ve farklı türleri hakkında bilgi edineceksiniz. İlgili terminolojiye temel bir girişle birlikte, pratik problem çözme için makine öğrenimi tekniklerini başarılı bir şekilde kullanmanın temelini atacağız.

Bu bölümde aşağıdaki konuları ele alacağız:

- Makine öğreniminin genel kavramları
- Üç öğrenme türü ve temel terminoloji
- Makine öğrenimi sistemlerinin başarılı bir şekilde tasarlanması için yapı taşları
- Veri analizi ve makine öğrenimi için Python'un kurulması ve ayarlanması

## Verileri bilgiye dönüştürmek için akıllı makineler oluşturma

İçinde bulunduğumuz modern teknoloji çağında, bolca sahip olduğumuz bir kaynak var: büyük miktarda yapılandırılmış ve yapılandırılmamış veri. Makine öğrenimi, 20. yüzyılın ikinci yarısında, tahminlerde bulunmak için verilerden bilgi türeten kendi kendine öğrenen algoritmaları içeren **yapay zekanın** (**AI**) bir alt alanı olarak gelişti.

6587191CA584E

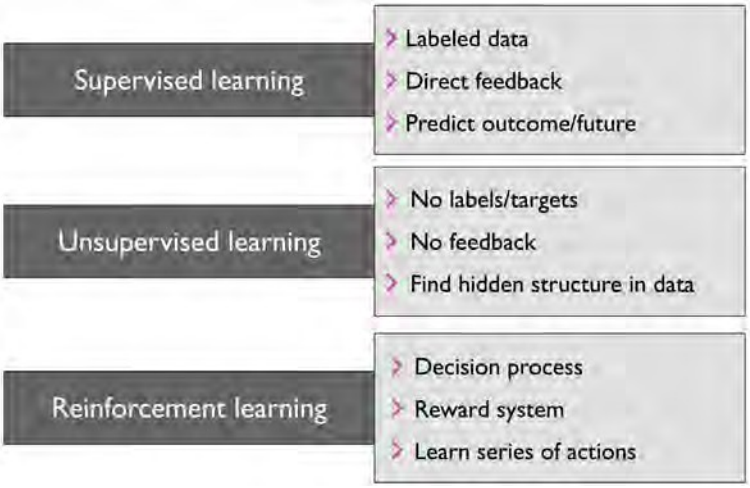

İnsanların büyük miktarda veriyi analiz ederek manuel olarak kurallar türetmesini ve modeller oluşturmasını gerektirmek yerine, makine öğrenimi, tahmine dayalı modellerin performansını kademeli olarak iyileştirmek ve veriye dayalı kararlar almak için verilerdeki bilgiyi yakalamak için daha verimli bir alternatif sunar.

Makine öğrenimi sadece bilgisayar bilimleri araştırmalarında giderek daha önemli hale gelmekle kalmıyor, aynı zamanda günlük hayatımızda da giderek daha büyük bir rol oynuyor. Makine öğrenimi sayesinde e-posta spam filtrelerinden, kullanışlı metin ve ses tanıma yazılımlarından, güvenilir web arama motorlarından, izlenecek eğlenceli film önerilerinden, mobil çek yatırma işlemlerinden, tahmini yemek teslim sürelerinden ve çok daha fazlasından yararlanıyoruz. Umarım yakında bu listeye güvenli ve verimli sürücüsüz otomobilleri de ekleyeceğiz. Ayrıca tıbbi uygulamalarda da kayda değer ilerlemeler kaydedilmiştir; örneğin araştırmacılar derin öğrenme modellerinin cilt kanserini insana yakın bir doğrulukla tespit edebildiğini göstermiştir (https://www.nature.com/ articles/nature21056). Bir başka dönüm noktası da yakın zamanda DeepMind'daki araştırmacılar tarafından gerçekleştirilmiştir; araştırmacılar derin öğrenmeyi kullanarak 3D protein yapılarını tahmin etmiş ve fizik temelli yaklaşımları önemli bir farkla geride bırakmıştır (https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year- old-grand-challenge-in-biology). Doğru 3D protein yapısı tahmini biyolojik ve farmasötik araştırmalarda önemli bir rol oynarken, son zamanlarda sağlık alanında makine öğreniminin başka birçok önemli uygulaması olmuştur. Örneğin araştırmacılar, COVID-19 hastalarının oksijen ihtiyacını dört güne kadar önceden tahmin ederek hastanelerin ihtiyacı olanlara kaynak ayırmasına yardımcı olacak sistemler tasarladı (https://ai.facebook.com/blog/new-ai-research-to-help-predict-covid-19- resource-needs-from-a-series-of-x-rays/). Günümüzün bir diğer önemli konusu da en büyük ve en kritik zorluklardan birini teşkil eden iklim değişikliğidir. Günümüzde birçok çaba, bununla mücadele etmek için akıllı sistemler geliştirmeye yöneliktir (https://www.forbes.com/sites/ robtoews/2021/06/20/these-are-the-startups-applying-ai-to-tackle-climate-change). İklim değişikliğiyle mücadeleye yönelik birçok yaklaşımdan biri de yeni ortaya çıkan hassas tarım alanıdır. Burada araştırmacılar, gübre kullanımını ve israfını en aza indirmek amacıyla kaynak dağıtımını optimize etmek için bilgisayarla görme tabanlı makine öğrenimi sistemleri tasarlamayı amaçlamaktadır.

# Üç farklı makine türü öğrenme

Bu bölümde, üç tür makine öğrenimine göz atacağız: **denetimli öğrenme**, **denetimsiz öğrenme** ve **pekiştirmeli öğrenme**. Üç farklı öğrenme türü arasındaki temel farklılıklar hakkında bilgi edineceğiz ve kavramsal örnekler kullanarak, uygulanabilecekleri pratik problem alanlarına ilişkin bir anlayış geliştireceğiz:

 olimpikbarbell@gmail.com 

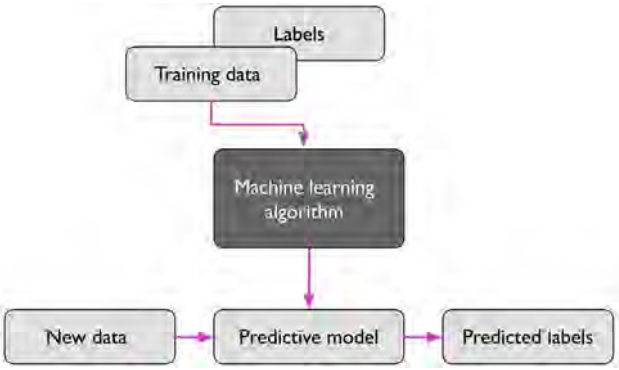



*Şekil 1.1: Üç farklı makine öğrenimi türü*

## Denetimli öğrenme ile gelecek hakkında tahminlerde bulunma

Denetimli öğrenmedeki temel amaç, etiketli eğitim verilerinden görünmeyen veya gelecekteki veriler hakkında tahminler yapmamızı sağlayan bir model öğrenmektir. Burada "denetimli" terimi, istenen çıktı sinyallerinin (etiketlerin) zaten bilindiği bir dizi eğitim örneğini (veri girdileri) ifade eder. Denetimli öğrenme daha sonra veri girdileri ve etiketler arasındaki ilişkiyi modelleme sürecidir. Dolayısıyla, denetimli öğrenmeyi "etiket öğrenmesi" olarak da düşünebiliriz.

*Şekil 1.2* tipik bir denetimli öğrenme iş akışını özetlemektedir; burada etiketli eğitim verileri, yeni, etiketsiz veri girdileri üzerinde tahminler yapabilen bir tahmin modelinin uydurulması için bir makine öğrenimi algoritmasına aktarılır:



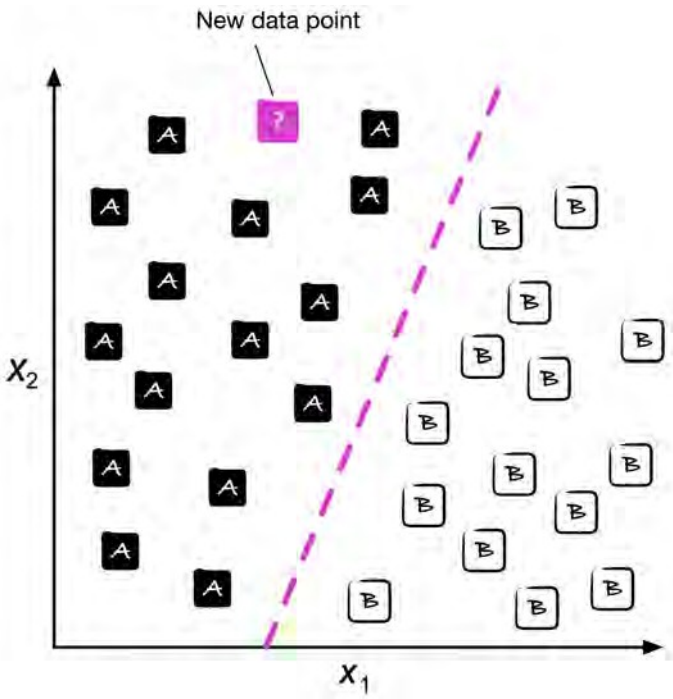

*Şekil 1.2: Gözetimli öğrenme süreci*

E-posta spam filtreleme örneğini göz önünde bulundurarak, yeni bir e-postanın iki kategoriden birine ait olup olmadığını tahmin etmek için spam veya spam olmayan olarak doğru şekilde işaretlenmiş etiketli e-postalardan oluşan bir külliyat üzerinde denetimli bir makine öğrenimi algoritması kullanarak bir modeli eğitebiliriz. Önceki e-posta spam filtreleme örneğinde olduğu gibi ayrık sınıf etiketlerine sahip bir denetimli öğrenme görevi, **sınıflandırma görevi** olarak da adlandırılır. Denetimli öğrenmenin bir diğer alt kategorisi, sonuç sinyalinin sürekli bir değer olduğu **regresyondur.**

## Sınıf etiketlerini tahmin etmek için sınıflandırma

Sınıflandırma, geçmiş gözlemlere dayanarak yeni örneklerin veya veri noktalarının kategorik sınıf etiketlerini tahmin etmeyi amaçlayan denetimli öğrenmenin bir alt kategorisidir. Bu sınıf etiketleri, veri noktalarının grup üyelikleri olarak anlaşılabilecek ayrık, sırasız değerlerdir. Daha önce bahsedilen e-posta spam tespiti örneği, makine öğrenimi algoritmasının iki olası sınıf arasında ayrım yapmak için bir dizi kural öğrendiği ikili sınıflandırma görevinin tipik bir örneğini temsil eder: spam ve spam olmayan e-postalar.

*Şekil 1.3,* 30 eğitim örneği verilen bir ikili sınıflandırma görevi kavramını göstermektedir; 15 eğitim örneği A sınıfı ve 15 eğitim örneği B sınıfı olarak etiketlenmiştir. Bu senaryoda, veri setimiz iki boyutludur, yani her örneğin kendisiyle ilişkili iki değeri vardır: $x_1$ ve $x_2$ . Şimdi, bu iki sınıfı ayırabilen ve $x_1$ *ve* $x_2$ değerleri göz önüne alındığında yeni verileri bu iki kategorinin her birine sınıflandırabilen bir kuralı (kesikli çizgi olarak temsil edilen karar sınırı) öğrenmek için denetimli bir makine öğrenimi algoritması kullanabiliriz:




*Şekil 1.3: Yeni bir veri noktasının sınıflandırılması*

 olimpikbarbell@gmail.com 

Ancak, sınıf etiketleri kümesinin ikili bir yapıda olması gerekmez. Denetimli bir öğrenme algoritması tarafından öğrenilen tahmin modeli, eğitim veri kümesinde sunulan herhangi bir sınıf etiketini yeni, etiketsiz bir veri noktasına veya örneğe atayabilir.

**Çok sınıflı sınıflandırma** görevinin tipik bir örneği el yazısı karakter tanımadır. Alfabedeki her harfin birden fazla el yazısı örneğinden oluşan bir eğitim veri kümesi toplayabiliriz. Harfler ("A," "B," "C," vb.) tahmin etmek istediğimiz farklı sırasız kategorileri veya sınıf etiketlerini temsil edecektir. Şimdi, bir kullanıcı bir giriş cihazı aracılığıyla yeni bir el yazısı karakteri sağlarsa, tahmin modelimiz alfabedeki doğru harfi belirli bir doğrulukla tahmin edebilecektir. Ancak, makine öğrenimi sistemimiz, eğitim veri kümesinin bir parçası değillerse, örneğin 0 ile 9 arasındaki rakamlardan herhangi birini doğru bir şekilde tanıyamayacaktır.

## Sürekli sonuçları tahmin etmek için regresyon

Önceki bölümde sınıflandırma görevinin örneklere kategorik, sırasız etiketler atamak olduğunu öğrendik. İkinci bir denetimli öğrenme türü, **regresyon analizi olarak** da adlandırılan sürekli sonuçların tahminidir. Regresyon analizinde, bize bir dizi tahmin edici (**açıklayıcı**) değişken ve sürekli bir yanıt değişkeni (**sonuç**) verilir ve bu değişkenler arasında bir sonucu tahmin etmemizi sağlayan bir ilişki bulmaya çalışırız.

Makine öğrenimi alanında, tahmin edici değişkenlerin genellikle "özellikler" olarak adlandırıldığını ve yanıt değişkenlerine genellikle "hedef değişkenler" olarak atıfta bulunulduğunu unutmayın. Bu kitap boyunca bu kuralları benimseyeceğiz.

Örneğin, öğrencilerin matematik SAT puanlarını tahmin etmekle ilgilendiğimizi varsayalım. (SAT, Amerika Birleşik Devletleri'nde üniversite kabullerinde sıklıkla kullanılan standartlaştırılmış bir sınavdır). Sınava çalışmak için harcanan zaman ile nihai puanlar arasında bir ilişki varsa, bu sınava girmeyi planlayan gelecekteki öğrencilerin sınav puanlarını tahmin etmek için çalışma süresini kullanan bir modeli öğrenmek için bunu eğitim verisi olarak kullanabiliriz.

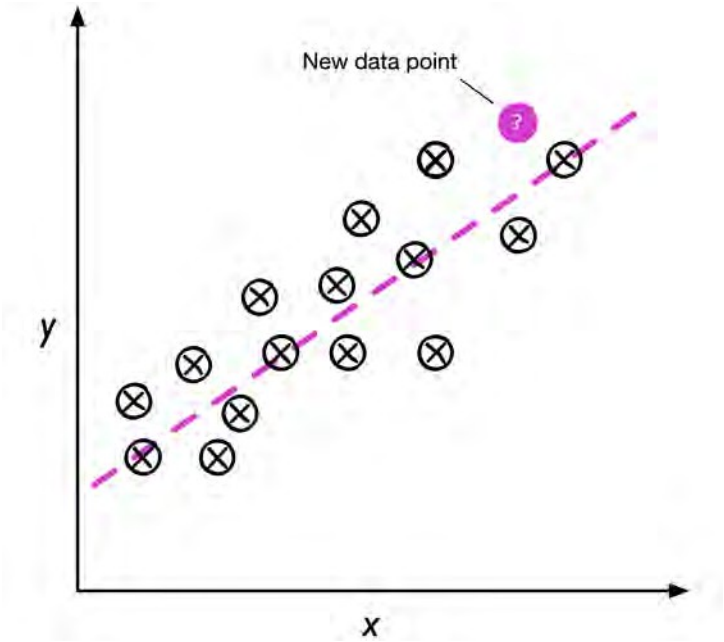
**Ortalamaya doğru gerileme**

"Regresyon" terimi Francis Galton tarafından 1886'da *Kalıtsal Boyda Vasatlığa Doğru Regresyon adlı* makalesinde ortaya atılmıştır. Galton, bir popülasyondaki boy varyansının zaman içinde artmadığı biyolojik olgusunu tanımlamıştır.

Ebeveynlerin boylarının çocuklarına geçmediğini, bunun yerine çocuklarının boylarının popülasyon ortalamasına doğru gerilediğini gözlemledi.

*Şekil 1.4* doğrusal regresyon kavramını göstermektedir. Bir özellik değişkeni (*x)* ve bir hedef değişkeni (*y)* verildiğinde, bu verilere, veri noktaları ile uydurulan doğru arasındaki mesafeyi (genellikle ortalama karesel mesafe) en aza indiren düz bir doğru uydururuz.

Artık yeni verilerin hedef değişkenini tahmin etmek için bu verilerden öğrenilen kesişim ve eğimi kullanabiliriz:



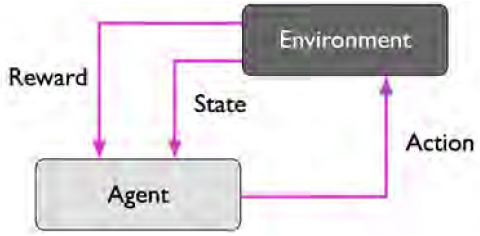

*Şekil 1.4: Doğrusal bir regresyon örneği*

## Pekiştirmeli öğrenme ile etkileşimli problemleri çözme

Makine öğreniminin bir başka türü de **takviyeli öğrenmedir**. Takviyeli öğrenmede amaç, çevreyle etkileşimlere dayalı olarak performansını artıran bir sistem (**ajan**) geliştirmektir. Çevrenin mevcut durumu hakkındaki bilgiler tipik olarak **ödül sinyali** de içerdiğinden, pekiştirmeli öğrenmeyi denetimli öğrenmeyle ilgili bir alan olarak düşünebiliriz. Ancak takviyeli öğrenmede bu geri bildirim doğru temel gerçek etiketi veya değeri değil, eylemin bir ödül fonksiyonu tarafından ne kadar iyi ölçüldüğünün bir ölçüsüdür. Bir ajan, çevreyle etkileşimi sayesinde, keşif amaçlı bir deneme-yanılma yaklaşımı veya kasıtlı planlama yoluyla bu ödülü en üst düzeye çıkaran bir dizi eylemi öğrenmek için pekiştirmeli öğrenmeyi kullanabilir.

Takviyeli öğrenmenin popüler bir örneği bir satranç programıdır. Burada ajan, tahtanın (ortamın) durumuna bağlı olarak bir dizi hamleye karar verir ve ödül, oyunun sonunda **kazanmak** veya **kaybetmek** olarak tanımlanabilir:

 olimpikbarbell@gmail.com 




*Şekil 1.5: Pekiştirmeli öğrenme süreci*

Pekiştirmeli öğrenmenin birçok farklı alt türü vardır. Bununla birlikte, genel bir şema, pekiştirmeli öğrenmedeki ajanın çevre ile bir dizi etkileşim yoluyla ödülü en üst düzeye çıkarmaya çalışmasıdır. Her durum pozitif veya negatif bir ödülle ilişkilendirilebilir ve ödül, bir satranç oyununu kazanmak veya kaybetmek gibi genel bir hedefe ulaşmak olarak tanımlanabilir. Örneğin, satrançta her hamlenin sonucu çevrenin farklı bir durumu olarak düşünülebilir.

Satranç örneğini daha fazla incelemek için, satranç tahtasındaki belirli konfigürasyonları ziyaret etmenin kazanma olasılığı daha yüksek olan durumlarla ilişkili olduğunu düşünelim - örneğin, rakibin satranç taşını tahtadan kaldırmak veya veziri tehdit etmek gibi. Ancak diğer konumlar, bir sonraki turda rakibe bir satranç taşı kaybetmek gibi oyunu kaybetme olasılığı daha yüksek olan durumlarla ilişkilidir. Şimdi, satranç oyununda, ödül (oyunu kazanmak için olumlu ya da kaybetmek için olumsuz) oyunun sonuna kadar verilmeyecektir. Buna ek olarak, nihai ödül rakibin nasıl oynadığına da bağlı olacaktır. Örneğin, rakip veziri feda edebilir ama sonunda oyunu kazanabilir.

Özetle, pekiştirmeli öğrenme, bir eylemi gerçekleştirdikten hemen sonra ya da *gecikmeli* geri bildirim yoluyla kazanılabilecek toplam ödülü en üst düzeye çıkaran bir dizi eylemi seçmeyi öğrenmekle ilgilidir.

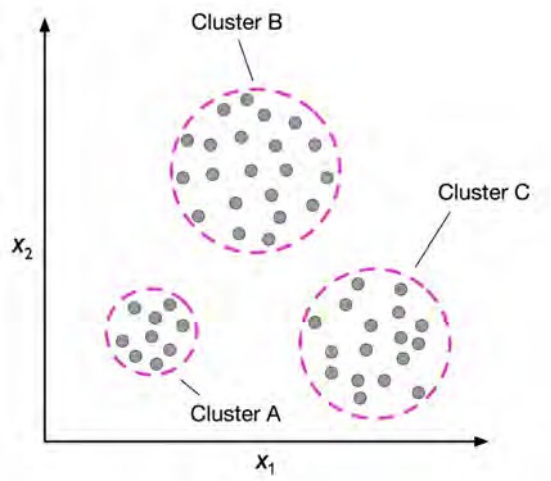
## Denetimsiz öğrenme ile gizli yapıların keşfedilmesi

Denetimli öğrenmede, bir modeli eğitirken doğru cevabı (etiket veya hedef değişken) önceden biliriz ve pekiştirmeli öğrenmede, aracı tarafından gerçekleştirilen belirli eylemler için bir ödül ölçüsü tanımlarız. Ancak denetimsiz öğrenmede, etiketlenmemiş verilerle ya da yapısı bilinmeyen verilerle uğraşırız. Denetimsiz öğrenme tekniklerini kullanarak, bilinen bir sonuç değişkeni veya ödül fonksiyonunun rehberliği olmadan anlamlı bilgiler elde etmek için verilerimizin yapısını keşfedebiliriz.

## Kümeleme ile alt grupları bulma

**Kümeleme,** grup üyelikleri hakkında önceden herhangi bir bilgiye sahip olmadan bir bilgi yığınını anlamlı alt gruplar (**kümeler**) halinde düzenlememize olanak tanıyan bir keşifsel veri analizi veya örüntü keşif tekniğidir. Analiz sırasında ortaya çıkan her küme, belirli bir benzerlik derecesini paylaşan ancak diğer kümelerdeki nesnelerden daha farklı olan bir grup nesneyi tanımlar; bu nedenle kümeleme bazen **denetimsiz sınıflandırma olarak** da adlandırılır. Kümeleme, bilgiyi yapılandırmak ve verilerden anlamlı ilişkiler türetmek için harika bir tekniktir. Örneğin, pazarlamacıların farklı pazarlama programları geliştirmek için ilgi alanlarına göre müşteri gruplarını keşfetmelerini sağlar.

*Şekil 1.6,* etiketlenmemiş verileri, özelliklerinin benzerliğine dayalı olarak üç farklı grup veya küme (A, B ve C, keyfi sırayla) halinde düzenlemek için kümelemenin nasıl uygulanabileceğini göstermektedir, $x_1$ ve $x_2$ :



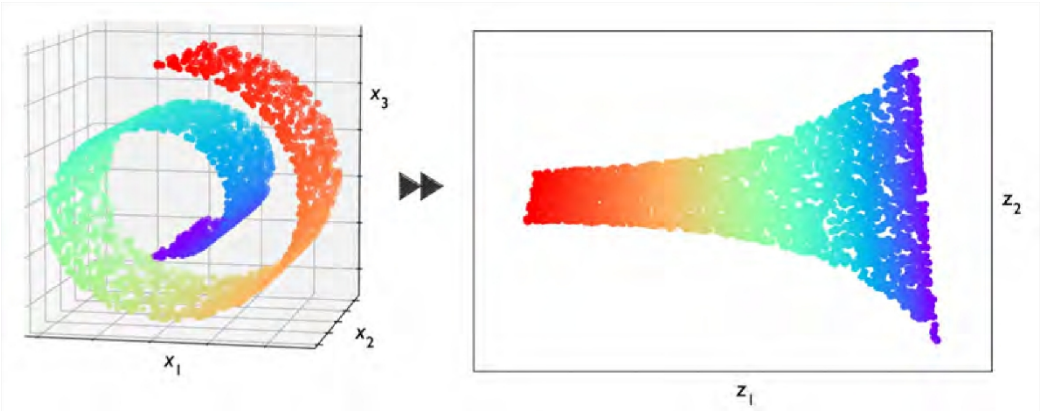

*Şekil 1.6: Kümeleme nasıl çalışır?*

## Veri sıkıştırma için boyut azaltma

Denetimsiz öğrenmenin bir diğer alt alanı da **boyut indirgemedir**. Genellikle, sınırlı depolama alanı ve makine öğrenimi algoritmalarının hesaplama performansı için zorluk oluşturabilecek yüksek boyutlu verilerle (her gözlem çok sayıda ölçümle birlikte gelir) çalışıyoruz. Denetimsiz boyutsallık azaltma, belirli algoritmaların tahmin performansını düşürebilen verilerdeki gürültüyü gidermek için özellik ön işlemede yaygın olarak kullanılan bir yaklaşımdır. Boyutsallık azaltma, ilgili bilgilerin çoğunu korurken verileri daha küçük boyutlu bir alt uzaya sıkıştırır.

 olimpikbarbell@gmail.com 

Bazen boyut indirgeme, verileri görselleştirmek için de faydalı olabilir; örneğin, yüksek boyutlu bir özellik seti, 2D veya 3D dağılım grafikleri veya histogramlar aracılığıyla görselleştirmek için bir, iki veya üç boyutlu özellik uzaylarına yansıtılabilir. *Şekil 1.7,* 3D İsviçre rulosunu yeni bir 2D özellik alt uzayına sıkıştırmak için doğrusal olmayan boyutsallık azaltmanın uygulandığı bir örneği göstermektedir:




*Şekil 1.7: Üç boyuttan iki boyuta indirgenmiş bir boyutluluk örneği*

# Temel terminolojiye ve notasyonlarına giriş

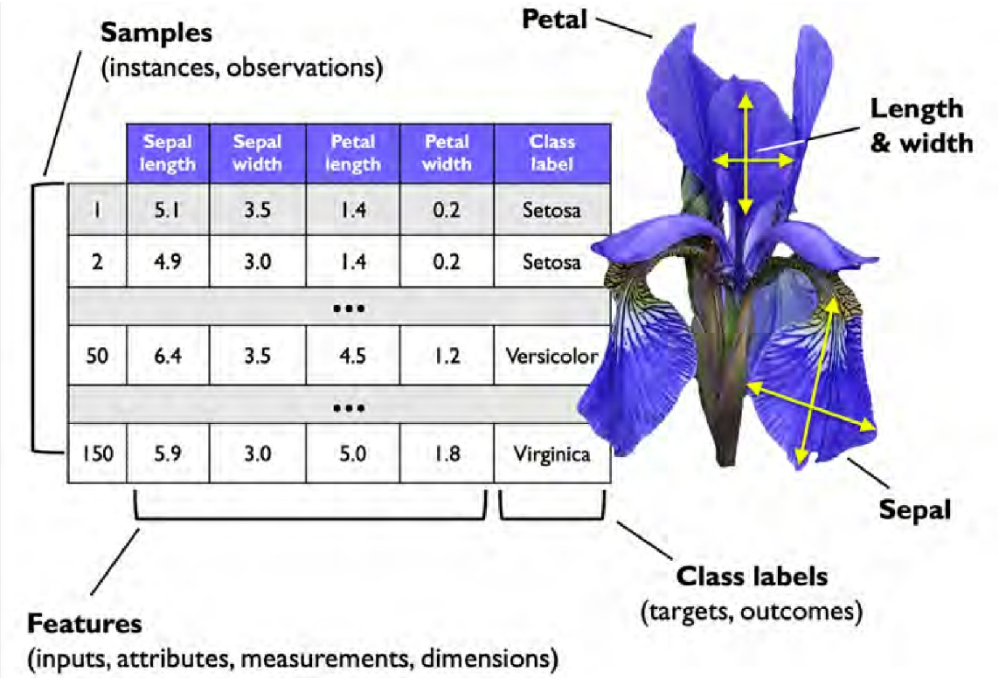
Makine öğreniminin üç geniş kategorisini -denetimli, denetimsiz ve pekiştirmeli öğrenme- tartıştığımıza göre, bu kitap boyunca kullanacağımız temel terminolojiye bir göz atalım. Aşağıdaki alt bölüm, bir veri kümesinin farklı yönlerine atıfta bulunurken kullanacağımız ortak terimlerin yanı sıra daha kesin ve verimli bir şekilde iletişim kurmak için matematiksel notasyonu da kapsamaktadır.

Makine öğrenimi çok geniş ve disiplinler arası bir alan olduğundan, aynı kavramlara atıfta bulunan birçok farklı terimle er ya da geç karşılaşmanız garantidir. İkinci alt bölüm, makine öğrenimi literatüründe en sık kullanılan terimlerin birçoğunu toplar ve makine öğrenimi yayınlarını okurken bir referans bölümü olarak işinize yarayabilir.

## Bu kitapta kullanılan gösterim ve kurallar

*Şekil 1.8,* makine öğrenimi alanında klasik bir örnek olan Iris veri setinden bir alıntıyı göstermektedir (daha fazla bilgi https://archive.ics.uci.edu/ml/datasets/iris adresinde bulunabilir). Iris veri kümesi, üç farklı türden (Setosa, Versicolor ve Virginica) 150 Iris çiçeğinin ölçümlerini içerir.

Burada her bir çiçek örneği veri kümemizde bir satırı temsil etmekte ve çiçek ölçümleri de veri kümesinin **özellikleri** olarak adlandırdığımız sütunlar olarak saklanmaktadır:




| | Sepal length | Sepal width | Petal length | Petal width | Class label |
|---|---|---|---|---|---|
| 1 | 5.1 | 3.5 | 1.4 | 0.2 | Setosa |
| 2 | 4.9 | 3.0 | 1.4 | 0.2 | Setosa |
| ... | | | | | |
| 50 | 6.4 | 3.5 | 4.5 | 1.2 | Versicolor |
| ... | | | | | |
| 150 | 5.9 | 3.0 | 5.0 | 1.8 | Virginica |

*Şekil 1.8: Iris veri seti*

Notasyonu ve uygulamayı basit ama verimli tutmak için lineer cebirin bazı temellerinden yararlanacağız. İlerleyen bölümlerde, verilerimize atıfta bulunmak için bir matris gösterimi kullanacağız. Her bir örneğin, her bir özelliğin ayrı bir sütun olarak depolandığı bir özellik matrisi olan ***X'te*** ayrı bir satır olarak temsil edilmesine yönelik yaygın geleneği takip edeceğiz.

Bu durumda 150 örnek ve dört özellikten oluşan Iris veri seti 150×4 matris olarak yazılabilir ve resmi olarak $XX\ X\ X^{150\times 4}$ şeklinde gösterilir:

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

 olimpikbarbell@gmail.com 

**Notasyonel konvansiyonlar**

Bu kitabın çoğu bölümünde, aksi belirtilmedikçe, $i$ üst simgesi $i$'inci eğitim örneğini ve $j$ alt simgesi j'inci eğitim veri kümesi boyutunu ifade etmek için kullanılacaktır.

Vektörlere atıfta bulunmak için küçük, kalın harfler ($\boldsymbol{x}$) ve matrislere atıfta bulunmak için büyük, kalın harfler ($\boldsymbol{X}$) kullanacağız. Bir vektör veya matristeki tek bir elemana atıfta bulunmak için harfleri italik olarak yazacağız (sırasıyla $x^{(n)}$ veya $x^{(n)}$).

Örneğin, $x_1^{(150)}$ çiçek örneği 150'nin ilk boyutu olan çanak yaprak uzunluğunu ifade eder. $\boldsymbol{X}$ matrisindeki her satır bir çiçek örneğini temsil eder ve dört boyutlu bir satır vektörü olarak yazılabilir, $\boldsymbol{x}^{(i)} \in \mathbb{R}^{1\times 4}$:

$$\boldsymbol{x}^{(i)} = [x_1^{(i)}\ x_2^{(i)}\ x_3^{(i)}\ x_4^{(i)}]$$

Ve her bir özellik boyutu 150 boyutlu bir sütun vektörüdür, $\boldsymbol{x}_j^{(i)} \in \mathbb{R}^{150\times 1}$. Örneğin:

$$\boldsymbol{x}_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}$$

Benzer şekilde, hedef değişkenleri (burada sınıf etiketleri) 150 boyutlu bir sütun vektörü olarak temsil edebiliriz:

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(150)} \end{bmatrix} \text{ burada } y^{(i)} \in \{\text{Setosa}, \text{Versicolor}, \text{Virginica}\}$$

## Makine öğrenimi terminolojisi

Makine öğrenimi çok geniş bir alandır ve aynı zamanda diğer araştırma alanlarından birçok bilim insanını bir araya getirdiği için disiplinler arasıdır. Hal böyle olunca, birçok terim ve kavram yeniden keşfedilmiş veya yeniden tanımlanmıştır ve size zaten tanıdık gelebilir ancak farklı isimler altında görünebilir. Size kolaylık sağlamak için, aşağıdaki listede, bu kitabı ve genel olarak makine öğrenimi literatürünü okurken faydalı bulabileceğiniz yaygın olarak kullanılan terimlerden ve eşanlamlılarından bir seçki bulabilirsiniz:

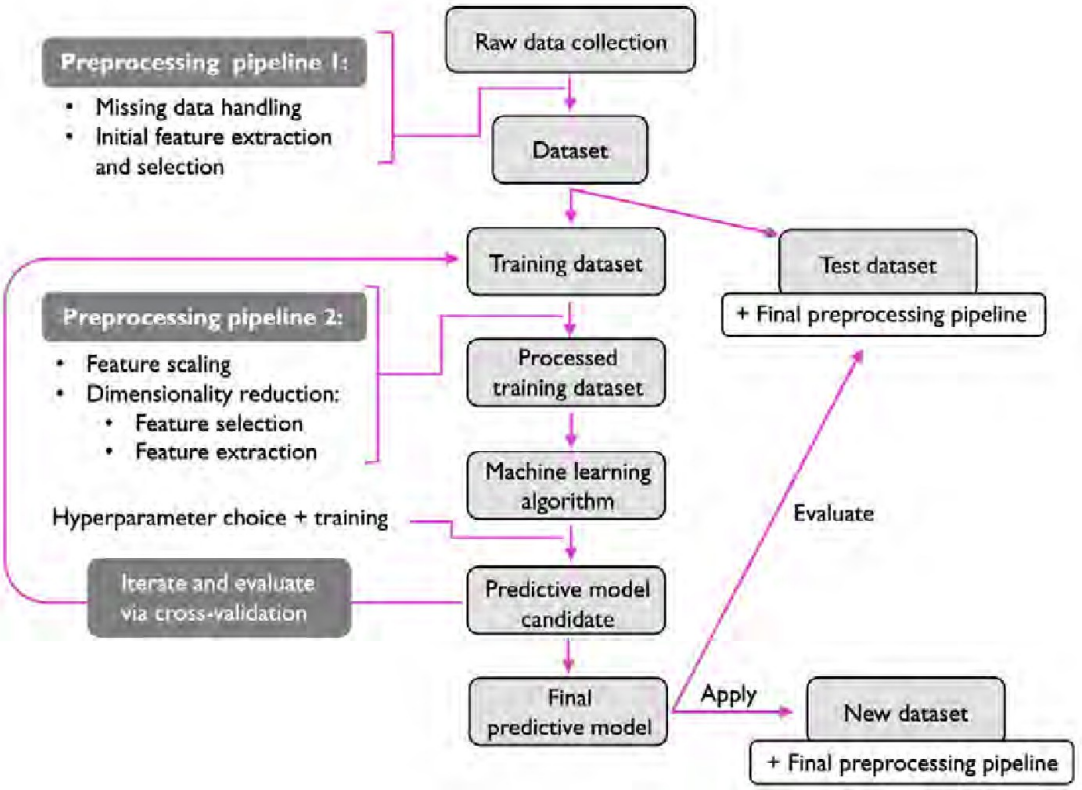
- **Eğitim örneği**: Veri kümesini temsil eden ve gözlem, kayıt, örnek veya numune ile eşanlamlı olan bir tablodaki satır (çoğu bağlamda numune, bir eğitim örnekleri koleksiyonunu ifade eder).
- **Eğitim**: Parametre tahminine benzer parametrik modeller için model uydurma.

- **Özellik, kısaltma *x***: Bir veri tablosundaki veya veri (tasarım) matrisindeki bir sütun. Yordayıcı, değişken, girdi, nitelik veya ortak değişken ile eş anlamlıdır.
- **Hedef, kısaltma *y***: Sonuç, çıktı, yanıt değişkeni, bağımlı değişken, (sınıf) etiketi ve temel gerçek ile eş anlamlıdır.
- **Kayıp fonksiyonu**: Genellikle *maliyet* fonksiyonu ile eş anlamlı olarak kullanılır. Bazen kayıp fonksiyonuna *hata* fonksiyonu da denir. Bazı literatürde, "kayıp" terimi tek bir veri noktası için ölçülen kaybı ifade eder ve maliyet, tüm veri kümesi üzerindeki kaybı (ortalama veya toplanmış) hesaplayan bir ölçümdür.

# Makine öğrenimi sistemleri oluşturmak için bir yol haritası

Önceki bölümlerde, makine öğreniminin temel kavramlarını ve üç farklı öğrenme türünü tartıştık. Bu bölümde, bir makine öğrenimi sisteminin öğrenme algoritmasına eşlik eden diğer önemli parçalarını tartışacağız.

*Şekil 1.9,* aşağıdaki alt bölümlerde tartışacağımız tahmine dayalı modellemede makine öğrenimini kullanmaya yönelik tipik bir iş akışını göstermektedir:




*Şekil 1.9: Tahmine dayalı modelleme iş akışı*

 olimpikbarbell@gmail.com 

## Ön işleme - verileri şekillendirme

Makine öğrenimi sistemleri oluşturmanın yol haritasını tartışarak başlayalım. Ham veriler nadiren bir öğrenme algoritmasının optimum performansı için gerekli biçim ve şekilde gelir. Bu nedenle, verilerin ön işlemden geçirilmesi, herhangi bir makine öğrenimi uygulamasındaki en önemli adımlardan biridir.

Önceki bölümdeki İris çiçeği veri kümesini örnek olarak alırsak, ham verileri anlamlı özellikler çıkarmak istediğimiz bir dizi çiçek görüntüsü olarak düşünebiliriz. Faydalı özellikler, çiçeklerin rengi veya çiçeklerin yüksekliği, uzunluğu ve genişliği etrafında merkezlenebilir.

Birçok makine öğrenimi algoritması, optimum performans için seçilen özelliklerin aynı ölçekte olmasını gerektirir; bu da genellikle [0, 1] aralığındaki özelliklerin veya daha sonraki bölümlerde göreceğimiz gibi sıfır ortalama ve birim varyansa sahip standart bir normal dağılımın dönüştürülmesiyle elde edilir.

Seçilen özelliklerden bazıları yüksek oranda ilişkili ve dolayısıyla belirli bir dereceye kadar gereksiz olabilir. Bu gibi durumlarda, boyut azaltma teknikleri özellikleri daha düşük boyutlu bir alt uzaya sıkıştırmak için kullanışlıdır. Özellik uzayımızın boyutluluğunun azaltılması, daha az depolama alanı gerekmesi ve öğrenme algoritmasının çok daha hızlı çalışabilmesi avantajına sahiptir. Bazı durumlarda, veri kümesi çok sayıda alakasız özellik (veya gürültü) içeriyorsa, yani veri kümesi düşük bir sinyal-gürültü oranına sahipse, boyutluluk azaltma bir modelin tahmin performansını da artırabilir.

Makine öğrenimi algoritmamızın yalnızca eğitim veri kümesinde iyi performans gösterip göstermediğini değil, aynı zamanda yeni verilere iyi genelleme yapıp yapmadığını belirlemek için, veri kümesini rastgele ayrı eğitim ve test veri kümelerine bölmek istiyoruz. Eğitim veri setini makine öğrenimi modelimizi eğitmek ve optimize etmek için kullanırken, test veri setini son modeli değerlendirmek için en sona saklıyoruz.

## Tahmine dayalı bir modelin eğitilmesi ve seçilmesi

İlerleyen bölümlerde göreceğiniz gibi, farklı problemleri çözmek için birçok farklı makine öğrenimi algoritması geliştirilmiştir. David Wolpert'in ünlü *No free lunch teoremlerinden* özetlenebilecek önemli bir nokta, öğrenmeyi "bedavaya" elde edemeyeceğimizdir (*The Lack of A Priori Distinctions Between Learning Algorithms,* D.H. Wolpert, 1996; *No free lunch theorems for optimization,* D.H. Wolpert ve W.G. Macready, 1997). Bu kavramı popüler *bir deyişle* ilişkilendirebiliriz: *Elinizdeki tek alet bir çekiçse, her şeye bir çiviymiş gibi davranmak caziptir* (Abraham Maslow, 1966). Örneğin, her sınıflandırma algoritmasının kendine özgü önyargıları vardır ve görev hakkında herhangi bir varsayımda bulunmazsak hiçbir sınıflandırma modeli üstünlük sağlamaz. Bu nedenle pratikte, en iyi performans gösteren modeli eğitmek ve seçmek için en az birkaç farklı öğrenme algoritmasını karşılaştırmak çok önemlidir. Ancak farklı modelleri karşılaştırmadan önce, performansı ölçmek için bir metriğe karar vermemiz gerekir. Yaygın olarak kullanılan bir metrik, doğru sınıflandırılmış örneklerin oranı olarak tanımlanan sınıflandırma doğruluğudur.

Sorulması gereken meşru bir soru şudur: Bu test veri setini model seçimi için kullanmazsak, ancak nihai model değerlendirmesi için tutarsak, hangi modelin nihai test veri setinde ve gerçek dünya verilerinde iyi performans gösterdiğini nasıl bilebiliriz? Bu soruda yer alan sorunu ele almak için "çapraz doğrulama" olarak özetlenen farklı teknikler kullanılabilir. Çapraz doğrulamada, modelin

6587191CA584E


genelleme performansını tahmin etmek için bir veri kümesini eğitim ve doğrulama alt kümelerine ayırırız.

 olimpikbarbell@gmail.com 

Son olarak, yazılım kütüphaneleri tarafından sağlanan farklı öğrenme algoritmalarının varsayılan parametrelerinin özel problem görevimiz için en uygun olmasını da bekleyemeyiz. Bu nedenle, ilerleyen bölümlerde modelimizin performansına ince ayar yapmamıza yardımcı olan hiperparametre optimizasyon tekniklerini sık sık kullanacağız.

Bu hiperparametreleri, verilerden öğrenilmeyen ancak bir modelin performansını artırmak için çevirebileceğimiz düğmelerini temsil eden parametreler olarak düşünebiliriz. Bu, ilerleyen bölümlerde gerçek örnekleri gördüğümüzde çok daha açık hale gelecektir.

## Modelleri değerlendirme ve görünmeyen veri örneklerini tahmin etme

Eğitim veri kümesi üzerine yerleştirilmiş bir model seçtikten sonra, *genelleme hatası olarak* adlandırılan bu modelin görünmeyen veriler üzerinde ne kadar iyi performans gösterdiğini tahmin etmek için test veri kümesini kullanabiliriz. Performansından memnun kalırsak, artık bu modeli yeni, gelecekteki verileri tahmin etmek için kullanabiliriz. Özellik ölçeklendirme ve boyut azaltma gibi daha önce bahsedilen prosedürlerin parametrelerinin yalnızca eğitim veri setinden elde edildiğini ve aynı parametrelerin daha sonra test veri setinin yanı sıra yeni veri örneklerini dönüştürmek için yeniden uygulandığını belirtmek önemlidir; aksi takdirde test verileri üzerinde ölçülen performans aşırı iyimser olabilir.

# Makine öğrenimi için Python kullanma

Python, veri bilimi için en popüler programlama dillerinden biridir ve çok aktif geliştirici ve açık kaynak topluluğu sayesinde, bilimsel iletişim ve makine öğrenimi için çok sayıda yararlı kütüphane geliştirilmiştir.

Python gibi yorumlanmış dillerin hesaplama yoğun görevler için performansı alt düzey programlama dillerinden daha düşük olsa da, çok boyutlu diziler üzerinde hızlı vektörize işlemler için alt katman Fortran ve C uygulamalarını temel alan NumPy ve SciPy gibi uzantı kütüphaneleri geliştirilmiştir.

Makine öğrenimi programlama görevleri için, çoğunlukla şu anda en popüler ve erişilebilir açık kaynaklı makine öğrenimi kütüphanelerinden biri olan scikit-learn kütüphanesine başvuracağız. İlerleyen bölümlerde, *derin öğrenme* adı verilen makine öğrenmesinin bir alt alanına odaklandığımızda, *derin sinir ağı* modellerini grafik kartlarını kullanarak çok verimli bir şekilde eğitme konusunda uzmanlaşmış PyTorch kütüphanesinin en son sürümünü kullanacağız.

## Python ve paketlerini Python Paket Dizini'nden yükleme

Python üç büyük işletim sistemi (Microsoft Windows, macOS ve Linux) için de kullanılabilir ve yükleyicinin yanı sıra belgeler de resmi Python web sitesinden indirilebilir: https://www.python.org.

Bu kitapta verilen kod örnekleri Python 3.9 için yazılmış ve test edilmiştir ve genellikle Python 3'ün mevcut olan en son sürümünü kullanmanızı öneririz. Kodun bir kısmı Python 2.7 ile de uyumlu olabilir, ancak Python 2.7 için resmi destek 2019'da sona erdiğinden ve açık kaynaklı kütüphanelerin çoğu Python 2.7'yi desteklemeyi çoktan bıraktığından (https://python3statement.org), Python 3.9 veya daha yenisini kullanmanızı şiddetle tavsiye ederiz.

Python sürümünüzü şu komutu çalıştırarak kontrol edebilirsiniz

```
python --version
```

veya

```
python3 --version
```

terminalinizde (veya Windows kullanıyorsanız PowerShell'de).

Bu kitap boyunca kullanacağımız ek paketler, Python 3.3'ten beri Python Standart Kütüphanesi'nin bir parçası olan pip yükleyici programı aracılığıyla kurulabilir. Pip hakkında daha fazla bilgi https://docs.python.org/3/installing/index.html adresinde bulunabilir.

Python'u başarıyla kurduktan sonra, ek Python paketlerini kurmak için terminalden pip komutunu çalıştırabiliriz:

```
pip install SomePackage
```

Halihazırda kurulu olan paketler --upgrade bayrağı ile güncellenebilir:

```
pip install SomePackage --upgrade
```

## Anaconda Python dağıtımını ve paket yöneticisini kullanma

Bilimsel hesaplama bağlamlarında Python yüklemek için şiddetle tavsiye edilen açık kaynaklı bir paket yönetim sistemi Continuum Analytics tarafından sunulan conda'dır. Conda ücretsizdir ve izin verilen bir açık kaynak lisansı altında lisanslanmıştır. Amacı, farklı işletim sistemlerinde veri bilimi, matematik ve mühendislik için Python paketlerinin kurulumuna ve sürüm yönetimine yardımcı olmaktır. Conda'yı kullanmak istiyorsanız, Anaconda, Miniconda ve Miniforge gibi farklı tatlarda gelir:

- Anaconda birçok bilimsel hesaplama paketi önceden yüklenmiş olarak gelir. Anaconda installer https://docs.anaconda.com/anaconda/install/ adresinden indirilebilir ve Anaconda hızlı başlangıç kılavuzu https://docs.anaconda.com/anaconda/user-guide/ getting-started/ adresinde mevcuttur.
- Miniconda, Anaconda'ya (https://docs.conda.io/en/latest/miniconda. html) göre daha yalın bir alternatiftir. Esasen, Anaconda'ya benzer, ancak önceden yüklenmiş herhangi bir paket olmadan, birçok insanın (yazarlar dahil) tercih ettiği.
- Miniforge, Miniconda'ya benzer ancak topluluk tarafından korunur ve Miniconda ve Anaconda'dan farklı bir paket deposu (conda-forge) kullanır. Miniforge'un Miniconda'ya harika bir alternatif olduğunu gördük. İndirme ve kurulum talimatlarını https://github.com/conda-forge/miniforge adresindeki GitHub deposunda bulabilirsiniz.

Anaconda, Miniconda veya Miniforge aracılığıyla conda'yı başarıyla kurduktan sonra, aşağıdaki komutu kullanarak yeni Python paketlerini yükleyebiliriz:

```
conda install SomePackage
```

 olimpikbarbell@gmail.com 

Mevcut paketler aşağıdaki komut kullanılarak güncellenebilir:

```
conda güncelleme SomePackage
```

Resmi conda kanalı aracılığıyla kullanılamayan paketler, topluluk destekli conda-forge projesi (https://conda-forge.org) aracılığıyla kullanılabilir.
--channel conda-forge bayrağı. Örneğin:

```
conda install SomePackage --channel conda-forge
```

Varsayılan conda kanalı veya conda-forge aracılığıyla kullanılamayan paketler şu yolla yüklenebilir daha önce açıklandığı gibi pip. Örneğin:

```
pip install SomePackage
```

## Bilimsel hesaplama, veri bilimi ve makine öğrenimi için paketler

Bu kitabın ilk yarısı boyunca, verileri depolamak ve işlemek için çoğunlukla NumPy'nin çok boyutlu dizilerini kullanacağız. Zaman zaman, tablo verileriyle çalışmayı daha da kolay hale getiren ek üst düzey veri işleme araçları sağlayan NumPy'nin üzerine inşa edilmiş bir kütüphane olan pandas'ı kullanacağız. Öğrenme deneyiminizi artırmak ve anlamlandırmak için genellikle son derece yararlı olan nicel verileri görselleştirmek için, çok özelleştirilebilir Matplotlib kütüphanesini kullanacağız.

Bu kitapta kullanılan ana makine öğrenimi kütüphanesi scikit-learn'dir (*3* ila *11. Bölümler*). *Bölüm 12*, *PyTorch ile Sinir Ağı Eğitiminin Paralelleştirilmesi*, daha sonra derin öğrenme için PyTorch kütüphanesini tanıtacaktır.

Bu kitabı yazmak için kullanılan başlıca Python paketlerinin sürüm numaraları aşağıdaki listede belirtilmiştir. Kod örneklerinin doğru çalıştığından emin olmak için lütfen kurulu paketlerinizin sürüm numaralarının ideal olarak bu sürüm numaralarına eşit olduğundan emin olun:

- NumPy 1.21.2
- SciPy 1.7.0
- Scikit-learn 1.0
- Matplotlib 3.4.3
- pandas 1.3.2

Bu paketleri kurduktan sonra, örneğin Python'da paketi içe aktararak ve sürüm niteliğine erişerek kurulu sürümü iki kez kontrol edebilirsiniz:

```
>> import numpy
>>> numpy. sürüm
'_1.21.2'
```

Size kolaylık sağlamak için, bu kitabın https://github.com/rasbt/machine-learning-book adresindeki uyumlu kod deposuna bir python-environment-check.py betiği ekledik, böylece bu betiği çalıştırarak hem Python sürümünüzü hem de paket sürümlerini kontrol edebilirsiniz.

Bazı bölümler ek paketler gerektirecek ve kurulumlar hakkında bilgi verecektir. Örneğin, bu noktada PyTorch'u kurma konusunda endişelenmeyin. *Bölüm 12,* ihtiyaç duyduğunuzda ipuçları ve talimatlar sağlayacaktır.

Kodunuz bölümdeki kodla birebir eşleşmesine rağmen hatalarla karşılaşırsanız, hata ayıklama için daha fazla zaman harcamadan veya yayıncı ya da yazarlara ulaşmadan önce temel paketlerin sürüm numaralarını kontrol etmenizi öneririz. Bazen, kütüphanelerin yeni sürümleri bu hataları açıklayabilecek geriye dönük uyumsuz değişiklikler getirmektedir.

Ana Python kurulumunuzdaki paket sürümünü değiştirmek istemiyorsanız, bu kitapta kullanılan paketleri kurmak için sanal bir ortam kullanmanızı öneririz. Eğer Python'u conda yöneticisi olmadan kullanıyorsanız, yeni bir sanal ortam oluşturmak için venv kütüphanesini kullanabilirsiniz. Örneğin, aşağıdaki iki komutla sanal ortamı oluşturabilir ve

```
python3 -m venv /Users/sebastian/Desktop/pyml-book
source /Users/sebastian/Desktop/pyml-book/bin/activate
```

etkinleştirebilirsiniz:

Her yeni terminal veya Power- Shell açtığınızda sanal ortamı etkinleştirmeniz gerektiğini unutmayın. venv hakkında daha fazla bilgiyi https://docs.python.org/3/library/venv.html adresinde bulabilirsiniz.

Anaconda'yı conda paket yöneticisi ile kullanıyorsanız, aşağıdaki şekilde bir sanal ortam oluşturabilir ve etkinleştirebilirsiniz:

```
conda create -n pyml python=3.9
conda activate pyml
```

# Özet

Bu bölümde, makine öğrenimini çok üst düzeyde inceledik ve sonraki bölümlerde daha ayrıntılı olarak inceleyeceğimiz büyük resmi ve ana kavramları tanıdık. Denetimli öğrenmenin iki önemli alt alandan oluştuğunu öğrendik: sınıflandırma ve regresyon. Sınıflandırma modelleri nesneleri bilinen sınıflara ayırmamızı sağlarken, hedef değişkenlerin sürekli sonuçlarını tahmin etmek için regresyon analizini kullanabiliriz. Denetimsiz öğrenme yalnızca etiketsiz verilerdeki yapıları keşfetmek için yararlı teknikler sunmakla kalmaz, aynı zamanda özellik ön işleme adımlarında veri sıkıştırma için de yararlı olabilir.

İlerleyen bölümlerde daha derin tartışmalar ve uygulamalı örnekler için bir temel olarak kullanacağımız makine öğrenimini problem görevlerine uygulamak için tipik yol haritasını kısaca gözden geçirdik. Son olarak, Python ortamımızı kurduk ve makine öğrenimini iş başında görmeye hazır olmak için gerekli paketleri yükledik ve güncelledik.

Bu kitabın ilerleyen bölümlerinde, makine öğreniminin kendisine ek olarak, farklı makine öğrenimi algoritmalarından en iyi performansı almanıza yardımcı olacak bir veri kümesini önceden işlemek için farklı teknikler tanıtacağız. Kitap boyunca sınıflandırma algoritmalarını oldukça kapsamlı bir şekilde ele alırken, regresyon analizi ve kümeleme için farklı teknikleri de keşfedeceğiz.

olimpikbarbell@gmail.com

Önümüzde makine öğreniminin geniş alanındaki birçok güçlü tekniği kapsayan heyecan verici bir yolculuk var. Ancak, bu kitabın bölümleri boyunca bilgilerimizi aşamalı olarak geliştirerek makine öğrenimine adım adım yaklaşacağız. Bir sonraki bölümde, bu yolculuğa sınıflandırma için en eski makine öğrenimi algoritmalarından birini uygulayarak başlayacağız, bu da bizi scikit-learn açık kaynaklı makine öğrenimi kütüphanesini kullanarak daha gelişmiş makine öğrenimi algoritmalarını ele alacağımız *Bölüm 3*, Scikit-Learn *Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu'na* hazırlayacak.

## Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.com

# 2

# Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi

Bu bölümde, sınıflandırma için algoritmik olarak tanımlanan ilk makine öğrenimi algoritmalarından ikisini kullanacağız: perceptron ve uyarlanabilir doğrusal nöronlar. Python'da adım adım bir perceptron uygulayarak ve Iris veri kümesindeki farklı çiçek türlerini sınıflandırmak için onu eğiterek başlayacağız. Bu, sınıflandırma için makine öğrenimi algoritmaları kavramını ve Python'da nasıl verimli bir şekilde uygulanabileceklerini anlamamıza yardımcı olacaktır.

Uyarlanabilir doğrusal nöronlar kullanarak optimizasyonun temellerini tartışmak, daha sonra *Bölüm 3, Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu*'nda scikit-learn makine öğrenimi kütüphanesi aracılığıyla daha sofistike sınıflandırıcıları kullanmak için zemin hazırlayacaktır.

Bu bölümde ele alacağımız konular aşağıdaki gibidir:

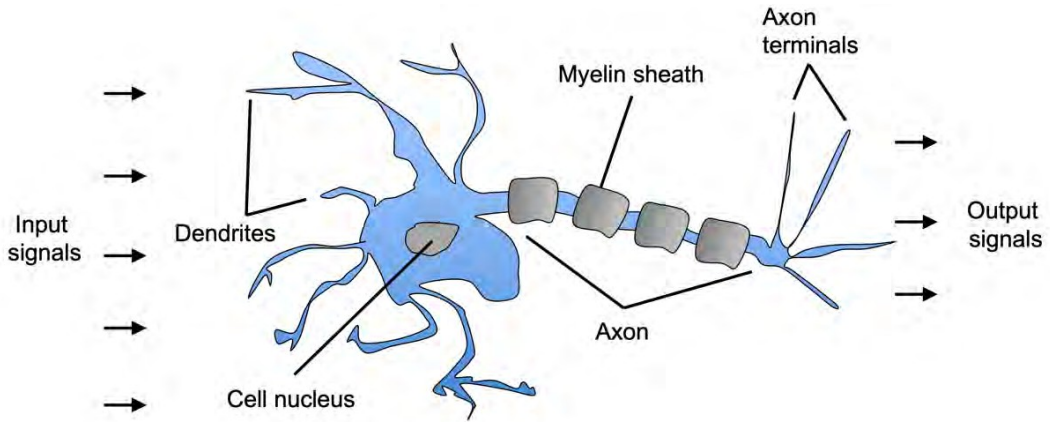
- Makine öğrenimi algoritmalarının anlaşılması
- Verileri okumak, işlemek ve görselleştirmek için pandas, NumPy ve Matplotlib kullanma
- Python'da 2 sınıflı problemler için doğrusal sınıflandırıcılar uygulama

## Yapay nöronlar - makinenin erken tarihine kısa bir bakış öğrenme

Algılayıcı ve ilgili algoritmaları daha ayrıntılı olarak tartışmadan önce, makine öğreniminin başlangıcında kısa bir tura çıkalım. Bir **yapay zekaya** (**YZ**) işaret etmek için biyolojik beynin nasıl çalıştığını anlamaya çalışan Warren McCulloch ve Walter Pitts, **McCulloch-Pitts** (**MCP**) nöronu olarak adlandırılan basitleştirilmiş bir beyin hücresinin ilk konseptini 1943 yılında yayınladı (*A Logical Calculus of the Ideas Immanent in Nervous Activity* by *W. S. McCulloch* and *W. Pitts*, *Bulletin of Mathematical Biophysics*, 5(4): 115-133, 1943).

6587191CA584E

Biyolojik nöronlar, *Şekil 2.1'*de gösterildiği gibi kimyasal ve elektrik sinyallerinin işlenmesi ve iletilmesinde rol oynayan beyindeki birbirine bağlı sinir hücreleridir:




*Şekil 2.1: Kimyasal ve elektrik sinyallerini işleyen bir nöron*

McCulloch ve Pitts böyle bir sinir hücresini ikili çıkışları olan basit bir mantık kapısı olarak tanımlamıştır; dendritlere birden fazla sinyal ulaşır, bunlar daha sonra hücre gövdesine entegre edilir ve biriken sinyal belirli bir eşiği aşarsa, akson tarafından iletilecek bir çıkış sinyali üretilir.

Sadece birkaç yıl sonra Frank Rosenblatt, MCP nöron modeline dayanan algılayıcı öğrenme kuralının ilk konseptini yayınladı (*The Perceptron: A Perceived and Recognizing Automaton* by *F. Rosenblatt*, *Cornell Aeronautical Laboratory*, 1957). Perceptron kuralı ile Rosenblatt, bir nöronun ateşlenip ateşlenmeyeceğine (bir sinyal iletip iletmeyeceğine) karar vermek için girdi özellikleriyle çarpılacak olan optimum ağırlık katsayılarını otomatik olarak öğrenecek bir algoritma önermiştir. Denetimli öğrenme ve sınıflandırma bağlamında, böyle bir algoritma daha sonra yeni bir veri noktasının bir sınıfa mı yoksa diğerine mi ait olduğunu tahmin etmek için kullanılabilir.

## Yapay bir nöronun resmi tanımı

Daha resmi olarak, **yapay nöronların** arkasındaki fikri iki sınıflı bir ikili sınıflandırma görevi bağlamına yerleştirebiliriz: 0 ve 1. Daha sonra, belirli girdi değerlerinin (*x)* doğrusal bir kombinasyonunu ve buna karşılık gelen bir ağırlık vektörünü (*w*) alan bir karar fonksiyonu, $\sigma\sigma z\sigma$, tanımlayabiliriz; burada *z,* net girdi olarak adlandırılır $z = w\,x_{11} + w\,x_{22} + \ldots + w\,x_{mm}$ :

$$\begin{array}{l} \boldsymbol{wwww}\ \boldsymbol{w}\ \begin{matrix} ww1 \\ \vdots \\ m \end{matrix}\ ],\ \boldsymbol{xx}\ \boldsymbol{w}\ \boldsymbol{w}\ \begin{matrix} x1 \\ \vdots \\ ww_{xm} \end{matrix}\ ] \end{array}$$

Şimdi, belirli bir örneğin net girdisi, $\boldsymbol{x}^{(i)}$ , tanımlanmış bir eşikten ($\theta\theta$) büyükse, sınıf 1'i, aksi takdirde sınıf 0'ı tahmin ederiz. Perceptron algoritmasında karar fonksiyonu, $\sigma(\bullet)$, **birim adım fonksiyonunun** bir çeşididir:

$$\sigma\sigma z\sigma\ \sigma\ \sigma \begin{matrix} z\ z\ zz \text{ ise } 1 \\ 0 \text{ aksi takdirde} \end{matrix}$$

 olimpikbarbell@gmail.com 

Daha sonra kod uygulamasını basitleştirmek için, bu kurulumu birkaç adımla değiştirebiliriz. İlk olarak, eşiği, $\theta\theta$, denklemin sol tarafına taşıyoruz:

$$z\ zz$$

$$z\ z\ zz\ zz$$

İkinci olarak, $b\ b\ bbb$ olarak adlandırılan bir önyargı *birimi* tanımlıyoruz ve bunu net girdinin bir parçası haline getiriyoruz:

$$z = w\ x_{11} + \ldots + w\ x_{mm} + b = \boldsymbol{w}^T \mathrm{x} + b$$

Üçüncü olarak, önyargı biriminin tanıtılması ve yukarıdaki net girdi $z'$nin yeniden tanımlanması göz önüne alındığında, karar fonksiyonunu aşağıdaki gibi yeniden tanımlayabiliriz:

$$\sigma\sigma z\sigma\ \sigma\ \sigma \begin{matrix} z\ zz \text{ ise } 1 \\ z \text{ aksi takdirde} \end{matrix}$$

**Doğrusal cebir temelleri: nokta çarpımı ve matris transpozu**

İlerleyen bölümlerde sık sık lineer cebirdeki temel notasyonlardan yararlanacağız. Örneğin, $\boldsymbol{x}$ ve $\boldsymbol{w}'$deki değerlerin çarpımlarının toplamını bir vektör nokta çarpımı kullanarak kısaltacağız, üst simge $T$ *ise* bir sütun vektörünü bir satır vektörüne veya tersine dönüştüren bir işlem olan transpoze anlamına gelir. Örneğin, aşağıdaki iki sütun vektörüne sahip olduğumuzu varsayalım:

$$a\ \boldsymbol{a}\ aa2]\ ,\ b\ \boldsymbol{a}\ \ \boldsymbol{a}b\ ]_2 \quad \begin{matrix} a_1 \\ a_3 \end{matrix} \quad \begin{matrix} b_1 \\ b_3 \end{matrix}$$

Daha sonra, $\boldsymbol{a}$ vektörünün transpozesini $\boldsymbol{a}^T = [_{a1\ a2\ a3}]$ olarak yazabilir ve nokta çarpımını şu şekilde yazabiliriz

$$a^T\ b\ \boldsymbol{b}\ \ \boldsymbol{b}\ a\ b_{ii}\ \boldsymbol{b}\ a_1 \cdot b_1 + a \cdot b_{22} + a_3 \cdot b_3$$

$$i$$

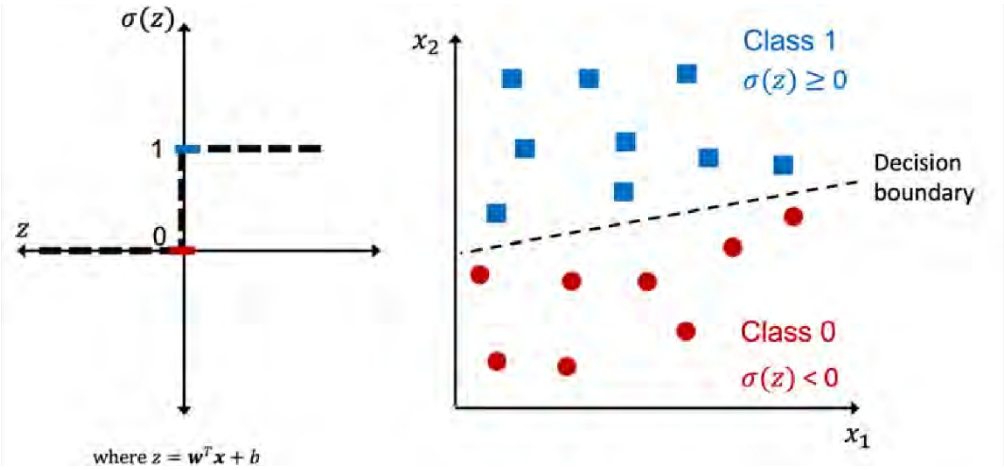
Ayrıca, transpoze işlemi, örneğin köşegeni üzerinden yansıtmak için matrislere de uygulanabilir:

$$\begin{matrix} 1\ 2 \\ [3\ 4] \\ 5\ 6 \end{matrix}^T = [\begin{matrix} 1\ 3\ 5 \\ 2\ 4\ 6 \end{matrix}]$$

Transpoze işleminin yalnızca matrisler için tanımlandığını lütfen unutmayın; ancak makine öğrenimi bağlamında "vektör" terimini kullandığımızda $n \times 1$ veya $1 \times m$ matrislere atıfta bulunuyoruz.

Bu kitapta, lineer cebirden sadece çok temel kavramları kullanacağız; ancak, hızlı bir tazelemeye ihtiyacınız varsa, lütfen Zico Kolter'in http://www.cs.cmu.edu/~zkolter/course/ linalg/linalg_notes.pdf adresinde ücretsiz olarak bulunan mükemmel *Lineer Cebir İnceleme ve Referansına* bir göz atın.

*Şekil 2.2'de* net girdi $z = \boldsymbol{w}^T x + b$'nin algılayıcının karar fonksiyonu tarafından nasıl ikili bir çıktıya (0 veya 1) sıkıştırıldığı (sol alt şekil) ve doğrusal bir karar sınırıyla ayrılabilen iki sınıf arasında ayrım yapmak için nasıl kullanılabileceği (sağ alt şekil) gösterilmektedir:




*Şekil 2.2: İkili sınıflandırma problemi için doğrusal bir karar sınırı üreten bir eşik fonksiyonu*

## Algılayıcı öğrenme kuralı

MCP nöronunun ve Rosenblatt'ın *eşikli* perceptron modelinin arkasındaki tüm fikir, beyindeki tek bir nöronun nasıl çalıştığını taklit etmek için yeniden indirgemeci bir yaklaşım kullanmaktır: ya *ateşlenir* ya da ateşlenmez. Dolayısıyla, Rosenblatt'ın klasik algılayıcı kuralı oldukça basittir ve algılayıcı algoritması aşağıdaki adımlarla özetlenebilir:

1. Ağırlıkları ve sapma birimini 0'a veya küçük rastgele sayılara başlatın
2. Her bir eğitim örneği için, $\boldsymbol{x}^{(i)}$ :
    a. Çıkış değerini hesaplayın, $yy^{(ii}$
    b. Ağırlıkları ve yanlılık birimini güncelleyin

Burada, çıkış değeri daha önce tanımladığımız birim adım fonksiyonu tarafından tahmin edilen sınıf etiketidir ve önyargı biriminin ve ağırlık vektöründeki her bir ağırlığın, $w_j$ , eşzamanlı güncellemesi, $\boldsymbol{w}$, daha resmi olarak şu şekilde yazılabilir:

$$_{wj} := _{wj} + \Delta_{wj}$$

ve $b\ bb\ b + \Delta b$

Güncelleme değerleri ("deltalar") aşağıdaki şekilde hesaplanır:

$$\Delta_{wwjj} = \eta\eta y^{(i)} - yy^{(i)})x^{(i)}_{ij}$$

ve $\Delta b = \eta\eta y^{(i)} - yy\ )^{(i)}$

Sapma biriminden farklı olarak, her bir ağırlığın, $w_j$ , veri kümesindeki bir özelliğe, $x_j$ , karşılık geldiğini ve yukarıda tanımlanan güncelleme değerinin, $\Delta ww_{jj}$ , belirlenmesinde yer aldığını unutmayın. Ayrıca, $\eta$ **öğrenme oranıdır** (genellikle 0,0 ile 1,0 arasında bir sabittir), $y^{(i)}$ birinci eğitim örneğinin **gerçek sınıf** etiketidir ve
$yy^{(i)}$ **tahmin** edilen **sınıf etiketidir**. Ön yargı biriminin ve ağırlık vektöründeki tüm ağırlıkların aynı anda güncellendiğine dikkat etmek önemlidir, bu da tahmin edilen etiketi yeniden hesaplamadığımız anlamına gelir,
$yy^{(i)}$ , önyargı birimi ve tüm ağırlıklar ilgili güncelleme değerleri aracılığıyla güncellenmeden önce, $\Delta ww_{jj}$ ve $\Delta b$. Somut olarak, iki boyutlu bir veri kümesi için güncellemeyi şu şekilde yazabiliriz:

$$\Delta ww1 = \eta(y^{(ii} - çıktı^{(ii} ix^{(ii}_{1};$$
$$\Delta ww2 = \eta(y^{(ii} - çıktı^{(ii} ix^{(ii}_{2};$$
$$\Delta b = \eta(y^{(ii} - çıkış^{(ii} i$$

Python'da perceptron kuralını uygulamadan önce, bu öğrenme kuralının gerçekten ne kadar güzel bir şekilde basit olduğunu göstermek için basit bir düşünce deneyinden geçelim. Perceptron'un sınıf etiketini doğru tahmin ettiği iki senaryoda, güncelleme değerleri 0 olduğu için önyargı birimi ve ağırlıklar değişmeden kalır:

(1) $y\ y^{(i)} = 0, yy^{(i)} = 0, \Delta ww_{jj} = \eta(0 - 0)x_{ij}^{(i)} = 0, \Delta b = \eta(0 - 0) = 0$

(2) $y\ y^{(i)} = 1, yy^{(i)} = 1, \Delta ww_{jj} = \eta(1 - 1)x_{ij}^{(i)} = 0, \Delta b = \eta(1 - 1) = 0$

Ancak, yanlış bir tahmin durumunda, ağırlıklar pozitif veya negatif hedef sınıf yönüne doğru itilir:

(3) $y\ y^{(i)} = 1, yy^{(i)} = 0, \Delta ww_{jj} = \eta(1 - 0)x_{jj}^{(i)} = \eta x_{jj}^{(i)}, \Delta b = \eta(1 - 0) = \eta$

(4) $y\ y^{(i)} = 0, yy^{(i)} = 1, \Delta ww_{jj} = \eta(0 - 1)x_{jj}^{(i)} = -\eta x_{jj}^{(i)}, \Delta b = \eta(0 - 1) = -\eta$

Çarpımsal bir faktör olarak özellik değerini daha iyi anlamak için, $x_j^{(i)}$ , an- üzerinden gidelim diğer basit örnek, nerede:

$$y\ y^{(i)} = 1, yy^{(i)} = 0, \eta = 1$$

$x_j^{(i)} = 1{,}5$ olduğunu ve bu örneği *0 sınıfı* olarak yanlış sınıflandırdığımızı varsayalım. Bu durumda, ilgili ağırlığı toplamda 2,5 artırırız, böylece net girdi, $z\ z\ x^{(i)} \times ww + b_{jj}$, daha pozitif olur
Bu örnekle bir sonraki karşılaşmamızda, örneği *sınıf 1* olarak sınıflandırmak için birim adım fonksiyonunun eşiğinin üzerinde olma olasılığı daha yüksektir:

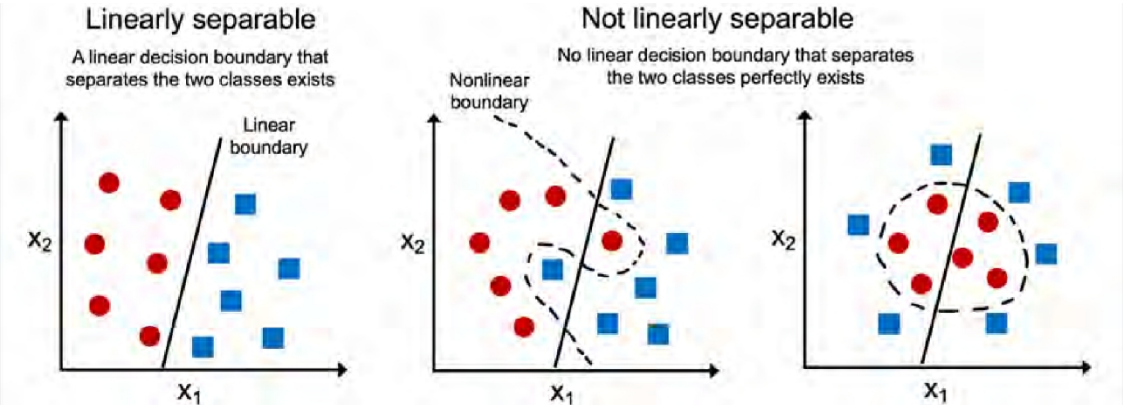
$\Delta wwjj = (1 - 0)1.5 = 1.5, \Delta b = (1 - 0) = 1$

Ağırlık güncellemesi, $\Delta ww_{jj}$ , $x^{(i)}_j$'in değeri ile orantılıdır. Örneğin, başka bir örneğimiz varsa, $x_j^{(i)} = 2$, yani yanlışlıkla *0 sınıfı* olarak sınıflandırılmışsa, bu örneği bir sonraki sefer doğru sınıflandırmak için karar sınırını daha da büyük ölçüde zorlayacağız:

$\Delta wwjj = (1 - 0)2 = 2, \Delta b = (1 - 0) = 1$

6587191CA584E

Algılayıcının yakınsamasının yalnızca iki sınıf doğrusal olarak ayrılabilirse garanti edildiğini belirtmek önemlidir, bu da iki sınıfın doğrusal bir karar sınırı ile mükemmel bir şekilde ayrılabileceği anlamına gelir. (İlgilenen okuyucular yakınsama kanıtını ders notlarımda bulabilirler: https:// sebastianraschka.com/pdf/lecture-notes/stat453ss21/L03_perceptron_slides.pdf). *Şekil 2.3* doğrusal olarak ayrılabilir ve doğrusal olarak ayrılamaz senaryoların görsel örneklerini göstermektedir:



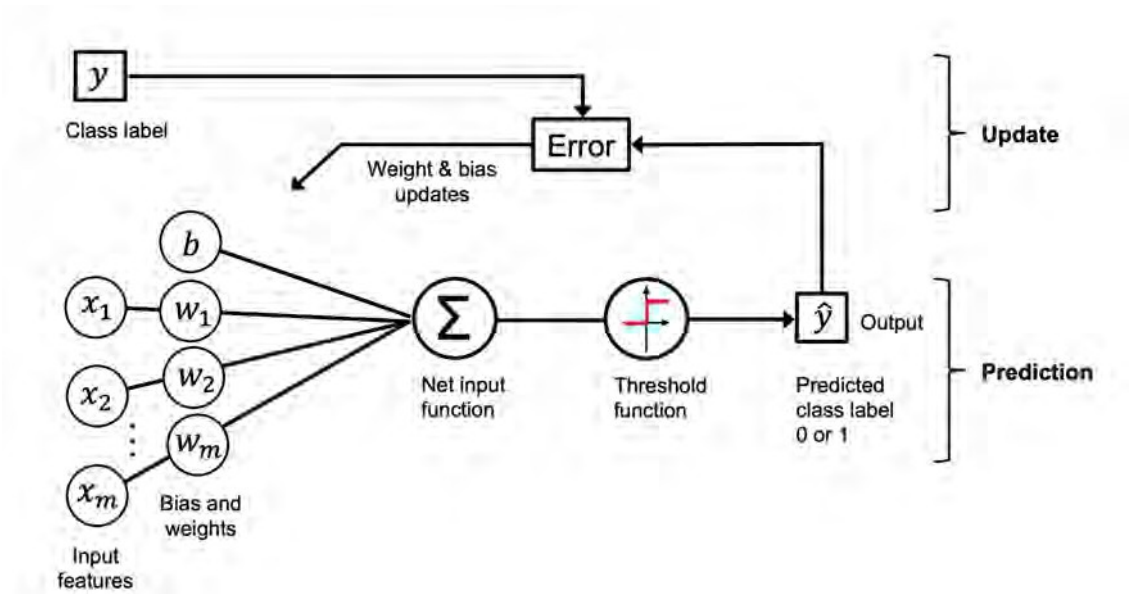

*Şekil 2.3: Doğrusal ve doğrusal olmayan ayrılabilir sınıf örnekleri*

İki sınıf doğrusal bir karar sınırı ile ayrılamıyorsa, eğitim veri kümesi (e**poklar**) üzerinde maksimum geçiş sayısı ve/veya tolere edilen yanlış sınıflandırma sayısı için bir eşik belirleyebiliriz - aksi takdirde algılayıcı ağırlıkları güncellemeyi asla durdurmaz. Bu bölümün ilerleyen kısımlarında, doğrusal karar sınırları üreten ve sınıflar tamamen doğrusal olarak ayrılabilir olmasa bile yakınsayan Adaline algoritmasını ele alacağız. *Bölüm 3*'te, doğrusal olmayan karar sınırları üretebilen algoritmalar hakkında bilgi edineceğiz.

**Örnek kodu indirme**

Bu kitabı doğrudan Packt'ten satın aldıysanız, örnek kod dosyalarını http://www.packtpub.com adresindeki hesabınızdan indirebilirsiniz. Bu kitabı başka bir yerden satın aldıysanız, tüm kod örneklerini ve veri kümelerini doğrudan https://github.com/ rasbt/machine-learning-book adresinden indirebilirsiniz.

Şimdi, bir sonraki bölümde uygulamaya geçmeden önce, az önce öğrendiklerinizi algılayıcının genel konseptini gösteren basit bir diyagramda özetleyebilirsiniz:




*Şekil 2.4: Modelin ağırlıkları ve yanlılığı hata fonksiyonuna göre güncellenir*

Yukarıdaki şema, algılayıcının bir örneğin (***x***) girdilerini nasıl aldığını ve net girdiyi hesaplamak için bunları önyargı birimi (***b***) ve ağırlıklarla (***w***) nasıl birleştirdiğini göstermektedir. Ağ girdisi daha sonra eşik fonksiyonuna aktarılır ve bu fonksiyon 0 ya da 1'den oluşan ikili bir çıktı -örneğin tahmin edilen sınıf etiketi- üretir. Öğrenme aşamasında bu çıktı, tahminin hatasını hesaplamak ve ağırlıklar ile önyargı birimini güncellemek için kullanılır.

# Python'da bir perceptron öğrenme algoritması uygulama

Önceki bölümde Rosenblatt'ın perceptron kuralının nasıl çalıştığını öğrendik; şimdi bunu Python'da uygulayalım ve *Bilgisayarlara Verilerden Öğrenme Yeteneği Vermek* başlıklı *1. Bölümde* tanıttığımız Iris veri kümesine uygulayalım.

## Nesne yönelimli bir algılayıcı API'si

Perceptron arayüzünü bir Python sınıfı olarak tanımlamak için nesne yönelimli bir yaklaşım benimseyeceğiz, bu da bir `fit` yöntemi aracılığıyla verilerden öğrenebilen ve ayrı bir `predict` yöntemi aracılığıyla tahminler yapabilen yeni `Perceptron` nesnelerini başlatmamıza olanak tanıyacaktır. Bir kural olarak, nesnenin başlatılmasıyla oluşturulmayan niteliklere bir alt çizgi (_) ekleriz, ancak bunu nesnenin diğer yöntemlerini çağırarak yaparız, örneğin `self.w_`.

**Python'un bilimsel hesaplama yığını için ek kaynaklar**

Python'un bilimsel kütüphanelerine henüz aşina değilseniz veya tazelemeye ihtiyacınız varsa, lütfen aşağıdaki kaynaklara bakın:

- NumPy: https://sebastianraschka.com/blog/2020/numpy-intro.html
- pandas: https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html
- Matplotlib: https://matplotlib.org/stable/tutorials/introductory/usage.html

Aşağıda Python'da bir perceptron uygulaması yer almaktadır:

```
import numpy as np
class Perceptron:
    """Perceptron sınıflandırıcı.

    Parametrel
    ------------
    er eta :

    float
      Öğrenme oranı (0,0 ile 1,0 arasında)
    n_iter : int
      Eğitim veri kümesinin üzerinden
    geçer. random_state : int
      Rastgele ağırlık başlatma için rastgele sayı üreteci
      tohumu.
    -----------
    Nitelikler

    w_ : 1d-dizi
      Fitting sonrası
    ağırlıklar. b_ : Skaler
      Montajdan sonra öngerilim ünitesi.

    errors_ : liste
      Her bir dönemdeki yanlış sınıflandırma (güncelleme) sayısı.

    """
    def init (self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
```

 olimpikbarbell@gmail.com 

```
    def fit(self, X, y):
        """Eğitim verilerini
        sığdır.

        Parametreler

        X : {dizi benzeri}, shape = [n_examples, n _ f e a t u r e s ]
          Eğitim vektörleri, burada n_examples örnek sayısı ve
          n_features özellik sayısıdır.
        y : dizi benzeri, shape = [n_examples]
          Hedef değerler.

        Geri dönüşler

        self : nesne

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,
                              size=X.shape[1])
        self.b_ = np.float_(0.)
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            zip(X, y) içindeki xi, hedef için:
                güncelleme = self.eta * (hedef - self.predict(xi))
                self.w_ += güncelleme * xi
                self.b_ += güncelle
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_girdi(self, X): """Net
        girdiyi hesapla"""
        return np.dot(X, self.w_) + self.b_

    def predict(self, X):
        """Birim adımından sonra sınıf etiketini
        döndür""" return np.where(self.net_input(X)
        >= 0.0, 1, 0)
```

Bu perceptron uygulamasını kullanarak, şimdi yeni Perceptron nesnelerini belirli bir öğrenme oranı, eta ($\eta$) ve epok sayısı, n_iter (eğitim veri kümesi üzerinden geçişler) ile başlatabiliriz.

Fit yöntemi aracılığıyla, self.b_ önyargısını 0 başlangıç değerine ve self.w_ içindeki ağırlıkları $\mathbb{R}^m$ vektörüne başlatırız; burada *m*, veri kümesindeki boyutların (özelliklerin) sayısını temsil eder.

İlk ağırlık vektörünün rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1]) aracılığıyla 0.01 standart sapmayla normal dağılımdan çekilen küçük rastgele sayılar içerdiğine dikkat edin; burada rgen, istenirse önceki sonuçları yeniden üretebilmemiz için kullanıcı tarafından belirlenen rastgele bir tohumla tohumladığımız bir NumPy rastgele sayı üretecidir.

Teknik olarak, ağırlıkları sıfır olarak başlatabiliriz (aslında bu orijinal algılayıcı algoritmasında yapılır). Ancak bunu yaparsak, $\eta$ (eta) öğrenme oranının karar sınırı üzerinde hiçbir etkisi olmayacaktır. Tüm ağırlıklar sıfır olarak başlatılırsa, öğrenme oranı parametresi, eta, ağırlık vektörünün yönünü değil, yalnızca ölçeğini etkiler. Trigonometriye aşinaysanız, aşağıdaki kod parçacığında gösterildiği gibi, v1 ile v2 = 0,5 × *v1 vektörü* arasındaki açının tam olarak sıfır olacağı

```
>> v1 = np.array([1, 2, 3])
>> v2 = 0,5 * v1
>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...                 np.linalg.norm(v2))
0.0
```

v1 =[1 2 3] vektörünü düşünün:

Burada, np.arccos trigonometrik ters kosinüs ve np.linalg.norm bir vektörün uzunluğunu hesaplayan bir fonksiyondur. (Rastgele sayıları rastgele bir normal dağılımdan çekme kararımız - örneğin tekdüze bir dağılım yerine- ve 0.01'lik bir standart sapma kullanma kararımız keyfiydi; unutmayın, daha önce tartışıldığı gibi tamamen sıfır vektörlerin özelliklerinden kaçınmak için sadece küçük rastgele değerlerle ilgileniyoruz).

Bu bölümü okuduktan sonra isteğe bağlı bir alıştırma olarak self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1]) değerini self.w_ = np.zeros(X.shape[1]) olarak değiştirebilir ve bir sonraki bölümde sunulan perceptron eğitim kodunu eta için farklı değerlerle çalıştırabilirsiniz. Karar sınırının değişmediğini gözlemleyeceksiniz.

**NumPy dizi indeksleme**

Tek boyutlu diziler için NumPy indeksleme, köşeli ayraç ([]) gösterimini kullanan Python listelerine benzer şekilde çalışır. İki boyutlu diziler için, ilk indeksleyici satır numarasını ve ikinci indeksleyici sütun numarasını ifade eder. Örneğin, iki boyutlu bir dizi olan X'in üçüncü satırını ve dördüncü sütununu seçmek için X[2, 3] kullanırız.

 olimpikbarbell@gmail.com 

Ağırlıklar başlatıldıktan sonra, `fit` yöntemi eğitim veri kümesindeki tüm bireysel örnekler üzerinde döngü yapar ve önceki bölümde tartıştığımız perceptron öğrenme kuralına göre ağırlıkları günceller.

Sınıf etiketleri, ağırlık güncellemesi için sınıf etiketini almak üzere eğitim sırasında `fit` yönteminde çağrılan `predict` `yöntemi` tarafından tahmin edilir; ancak `predict`, modelimizi yerleştirdikten sonra yeni verilerin sınıf etiketlerini tahmin etmek için de kullanılabilir. Ayrıca, daha sonra algılayıcımızın eğitim sırasında ne kadar iyi performans gösterdiğini analiz edebilmek için `self.errors_` listesindeki her epok sırasında yanlış sınıflandırmaların sayısını da topluyoruz. `net_input` yönteminde kullanılan `np.dot` fonksiyonu basitçe vektör nokta çarpımını hesaplar, $\boldsymbol{w}^T \mathrm{x} + b$.

**Vektörleştirme: For döngülerini vektörleştirilmiş kod ile değiştirme**

İki dizi (`a` ve `b`) arasındaki vektör nokta çarpımını `a.dot(b)` veya `np.dot(a, b )` aracılığıyla hesaplamak için NumPy kullanmak yerine, hesaplamayı saf Python'da `sum([i * j for i, j in zip(a, b )] )` aracılığıyla da gerçekleştirebiliriz. Ancak, NumPy kullanmanın klasik Python döngü yapılarına `göre` avantajı, aritmetik işlemlerinin vektörleştirilmesidir. Vektörleştirme, temel bir aritmetik işlemin otomatik olarak bir dizideki tüm elemanlara uygulanması anlamına gelir. Aritmetik işlemlerimizi, her bir eleman için bir seferde bir dizi işlem yapmak yerine, bir dizi üzerinde bir dizi yapı olarak formüle ederek, **tek komut, çoklu veri** (**SIMD**) desteğine sahip modern **merkezi işlem birimi (CPU**) mimarilerimizi daha iyi kullanabiliriz. Ayrıca NumPy, C veya Fortran'da yazılmış olan **Temel Lineer Cebir Alt Programları** (**BLAS)** ve **Lineer Cebir Paketi** (**LAPACK)** gibi yüksek düzeyde optimize edilmiş lineer cebir kütüphanelerini kullanır. Son olarak NumPy, vektör ve matris nokta çarpımları gibi doğrusal cebirin temellerini kullanarak kodumuzu daha kompakt ve sezgisel bir şekilde yazmamıza da olanak tanır.

## Iris veri kümesi üzerinde bir perceptron modelinin eğitilmesi

Algılayıcı uygulamamızı test etmek için, bu bölümün geri kalanında aşağıdaki analizleri ve örnekleri iki özellik değişkeniyle (boyutla) sınırlayacağız. Algılayıcı kuralı iki boyutla sınırlı olmasa da, yalnızca iki özelliği, çanak yaprak uzunluğu ve taç yaprak uzunluğunu dikkate almak, eğitilen modelin karar bölgelerini öğrenme amacıyla bir dağılım grafiğinde görselleştirmemize olanak sağlayacaktır.

Pratik nedenlerden dolayı Iris veri kümesinden sadece iki çiçek sınıfını, setosa ve versicolor, dikkate alacağımızı unutmayın - perceptron'un ikili bir sınıflandırıcı olduğunu unutmayın. Bununla birlikte, perceptron algoritması çok sınıflı sınıflandırmaya genişletilebilir - örneğin, **bire karşı hepsi** (**OvA**) tekniği.

**Çok sınıflı sınıflandırma için OvA yöntemi**

Bazen **one-versus-rest (OvR) olarak** da adlandırılan OvA, herhangi bir ikili sınıflandırıcıyı çok sınıflı problemlere genişletmemizi sağlayan **bir** tekniktir. OvA kullanarak, belirli bir sınıfın pozitif sınıf olarak ele alındığı ve diğer tüm sınıflardan örneklerin negatif sınıflar olarak kabul edildiği her sınıf için bir sınıflandırıcı eğitebiliriz. Yeni, etiketsiz bir veri örneğini sınıflandıracak olsaydık, *n sınıf* etiketlerinin sayısı olmak üzere n sınıflandırıcımızı kullanır ve sınıflandırmak istediğimiz belirli örneğe en yüksek güvene sahip sınıf etiketini atardık. Algılayıcı durumunda, en büyük mutlak net girdi değeriyle ilişkili sınıf etiketini seçmek için OvA'yı kullanırız.

İlk olarak, `pandas` kütüphanesini kullanarak Iris veri setini doğrudan *UCI Machine Learning Repository'd*en bir `DataFrame` nesnesine yükleyeceğiz ve verilerin doğru yüklendiğini kontrol etmek için `tail` yöntemi aracılığıyla son beş satırı yazdıracağız:

```
>> import os
>> import pandas as pd
>>> s = 'https://archive.ics.uci.edu/ml/'\
...        'machine-learning-databases/iris/iris.data'
>> print('URL'den:', s)
URL'den: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.
data
>> df = pd.read_csv(s,
...                          header=None,
...                          encoding='utf-8')
>> df.tail()
```

Önceki kodu çalıştırdıktan sonra, Iris veri kümesinin son beş satırını gösteren aşağıdaki çıktıyı görmeliyiz:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **145** | 6.7 | 3.0 | 5.2 | 2.3 | Iris-virginica |
| **146** | 6.3 | 2.5 | 5.0 | 1.9 | Iris-virginica |
| **147** | 6.5 | 3.0 | 5.2 | 2.0 | Iris-virginica |
| **148** | 6.2 | 3.4 | 5.4 | 2.3 | Iris-virginica |
| **149** | 5.9 | 3.0 | 5.1 | 1.8 | Iris-virginica |

*Şekil 2.5: Iris veri setinin son beş satırı*

 olimpikbarbell@gmail.com 

**Iris veri kümesini yükleme**

Iris veri setinin (ve bu kitapta kullanılan diğer tüm veri setlerinin) bir kopyasını bu kitabın kod paketinde bulabilirsiniz; bu kopyayı ofis dışında çalışıyorsanız veya https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data adresindeki UCI sunucusu geçici olarak kullanılamıyorsa kullanabilirsiniz. Örneğin, Iris veri setini yerel bir dizinden yüklemek için şu satırı değiştirebilirsiniz,

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/iris/iris.data',
    header=None, encoding='utf-8')
```

bir sonraki ile:

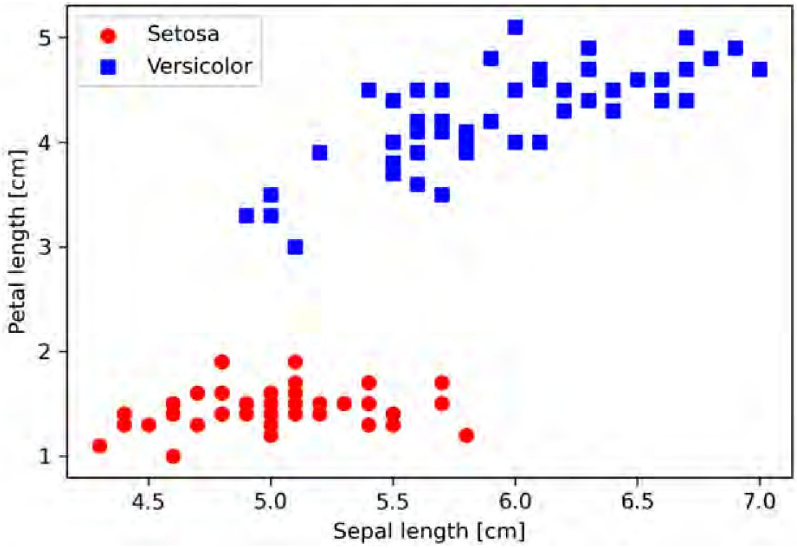
```
df = pd.read_csv(
    'your/local/path/to/iris.data',
    header=None, encoding='utf-8')
```

Ardından, 50 Iris-setosa ve 50 Iris-versicolor çiçeğine karşılık gelen ilk 100 sınıf etiketini çıkarıyoruz ve sınıf etiketlerini, pandas `DataFrame`'in `values` yönteminin karşılık gelen NumPy temsilini verdiği `y` vektörüne atadığımız `1` (versicolor) ve `0` (setosa) olmak üzere iki tamsayı sınıf etiketine dönüştürüyoruz.

Benzer şekilde, bu 100 eğitim örneğinin ilk özellik sütununu (sepal uzunluk) ve üçüncü özellik sütununu (petal uzunluk) çıkarıyoruz ve bunları iki boyutlu bir dağılım grafiği aracılığıyla görselleştirebileceğimiz bir özellik matrisine (`X`) atıyoruz:

```
>> import matplotlib.pyplot as plt
>> import numpy as np
>> # setosa ve versicolor'ı seçin
>> y = df.iloc[0:100, 4].values
>> y = np.where(y == 'Iris-setosa', 0, 1)
>> # çanak yaprak uzunluğunu ve taç yaprak uzunluğunu çıkarın
>>> X = df.iloc[0:100, [0, 2]].values
>> # veri çizimi
>> plt.scatter(X[:50, 0], X[:50, 1],
...                color='red', marker='o', label='Setosa')
>> plt.scatter(X[50:100, 0], X[50:100, 1],
...                color='blue', marker='s', label='Versicolor')
>> plt.xlabel('Sepal uzunluk [cm]')
>> plt.ylabel('Petal uzunluğu [cm]')
>> plt.legend(loc='sol üst')
>> plt.show()
```

Yukarıdaki kod örneğini çalıştırdıktan sonra, aşağıdaki dağılım grafiğini görmeliyiz:



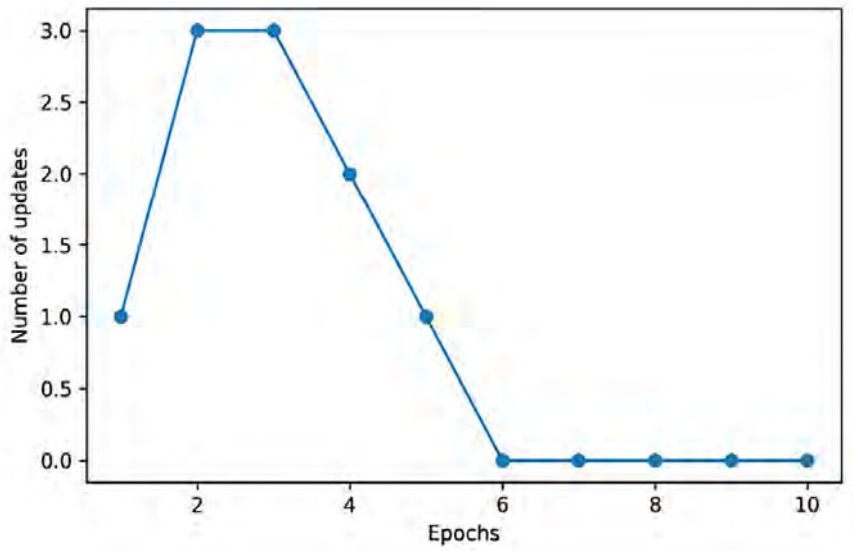

*Şekil 2.6: Setosa ve versicolor çiçeklerinin çanak yaprak ve taç yaprak uzunluğuna göre dağılım grafiği*

*Şekil 2.6,* Iris veri kümesindeki çiçek örneklerinin iki özellik ekseni boyunca dağılımını göstermektedir: taç yaprak uzunluğu ve çanak yaprak uzunluğu (santimetre cinsinden ölçülmüştür). Bu iki boyutlu özellik alt uzayında, setosa ile versicolor çiçeklerini ayırmak için doğrusal bir karar sınırının yeterli olması gerektiğini görebiliriz. Dolayısıyla, perceptron gibi doğrusal bir sınıflandırıcı bu veri kümesindeki çiçekleri mükemmel bir şekilde sınıflandırabilmelidir.

Şimdi, perceptron algoritmamızı yeni çıkardığımız Iris veri alt kümesi üzerinde eğitmenin zamanı geldi. Ayrıca, algoritmanın yakınsayıp yakınsamadığını ve iki Iris çiçek sınıfını ayıran bir karar sınırı bulup bulmadığını kontrol etmek için her epok için yanlış sınıflandırma hatasını çizeceğiz:

```
>> ppn = Perceptron(eta=0.1, n_iter=10)
>> ppn.fit(X, y)
>> plt.plot(range(1, len(ppn.errors_) + 1),
...              ppn.errors_, marker='o')
>>  plt.xlabel('Çağlar')
>> plt.ylabel('Güncelleme sayısı')
>> plt.show()
```

Algılayıcı ağırlıkları ve önyargısı bir örneği her yanlış sınıflandırdığında güncellendiğinden, yanlış sınıflandırma hatalarının sayısı ile güncelleme sayısının aynı olduğuna dikkat edin. Önceki kodu çalıştırdıktan sonra, *Şekil 2.7'de* gösterildiği gibi epok sayısına karşı yanlış sınıflandırma hatalarının grafiğini görmeliyiz:




*Şekil 2.7: Epok sayısına karşı yanlış sınıflandırma hatalarının grafiği*

*Şekil 2.7'de* görebileceğimiz gibi, algılayıcımız altıncı epoktan sonra yakınsamıştır ve artık eğitim örneklerini mükemmel bir şekilde sınıflandırabilmelidir. İki boyutlu veri kümeleri için karar sınırlarını görselleştirmek üzere küçük bir kolaylık fonksiyonu uygulayalım:

```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, sınıflandırıcı, çözünürlük=0.02):
    # işaretleyici oluşturucuyu ve renk haritasını ayarlayın
    işaretleyiciler = ('o', 's', '^', 'v', '<')
    colors = ('red', ' blue', ' lightgreen', ' gray', ' cyan') cmap
    = ListedColormap(colors[:len(np.unique(y))])

    # karar yüzeyini çizin
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, çözünürlük),
                           np.arange(x2_min, x2_max, çözünürlük))
    lab = classifier.predict(np.array([xx1.ravel(), x x 2 . ravel()]).T) lab
    = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
```

6587191CA584E

```
# plot sınıfı örnekleri
for idx, cl in
    enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0],
                y=X[y == cl, 1],
                alpha=0.8,
                c=colors[idx],
                marker=markers[idx],
                label=f'Class {cl}',
                edgecolor='black')
```

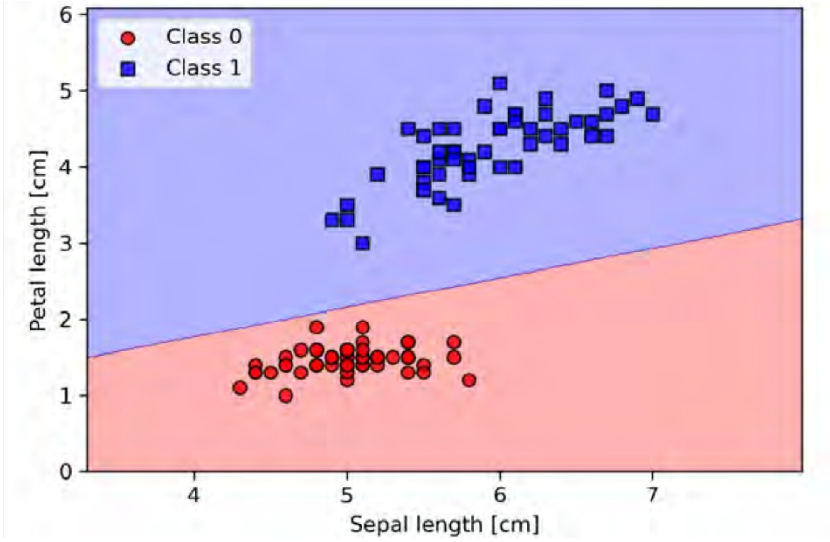
İlk olarak, bir dizi renk ve işaretleyici tanımlıyoruz ve ListedColormap aracılığıyla renk listesinden bir renk haritası oluşturuyoruz. Ardından, iki özellik için minimum ve maksimum değerleri belirleriz ve bu özellik vektörlerini NumPy meshgrid işlevi aracılığıyla xx1 ve xx2 adlı bir çift ızgara dizisi oluşturmak için kullanırız. Algılayıcı sınıflandırıcımızı iki özellik boyutunda eğittiğimiz için, ızgara dizilerini düzleştirmemiz ve Iris eğitim alt kümesiyle aynı sayıda sütuna sahip bir matris oluşturmamız gerekir, böylece ilgili ızgara noktalarının sınıf etiketlerini (lab) tahmin etmek için tahmin yöntemini kullanabiliriz.

Tahmin edilen sınıf etiketlerini, lab, xx1 ve xx2 ile aynı boyutlara sahip bir ızgarada yeniden şekillendirdikten sonra, artık Matplotlib'in contourf işlevi aracılığıyla, farklı karar bölgelerini ızgara dizisindeki tahmin edilen her sınıf için farklı renklerle eşleştiren bir kontur grafiği çizebiliriz:

```
>>> plot_decision_regions(X, y, classifier=ppn)
>> plt.xlabel('Sepal uzunluk [cm]')
>> plt.ylabel('Petal uzunluğu [cm]')
>> plt.legend(loc='sol üst')
>> plt.show()
```

Önceki kod örneğini çalıştırdıktan sonra, şimdi *Şekil 2.8'*de gösterildiği gibi karar bölgelerinin bir grafiğini görmeliyiz:

 olimpikbarbell@gmail.com 




*Şekil 2.8: Algılayıcının karar bölgelerinin bir grafiği*

Grafikte de görebileceğimiz gibi, algılayıcı Iris eğitim alt kümesindeki tüm çiçek örneklerini mükemmel bir şekilde sınıflandırabilen bir karar sınırı öğrenmiştir.

**Perceptron yakınsaması**

Algılayıcı iki İris çiçeği sınıfını mükemmel bir şekilde sınıflandırmasına rağmen, yakınsama algılayıcının en büyük sorunlarından biridir. Rosenblatt, iki sınıf doğrusal bir hi-perplane ile ayrılabiliyorsa algılayıcı öğrenme kuralının yakınsadığını matematiksel olarak kanıtlamıştır. Ancak, sınıflar böyle bir doğrusal karar sınırı ile mükemmel bir şekilde ayrılamazsa, maksimum sayıda epok belirlemediğimiz sürece ağırlıkların güncellenmesi asla durmayacaktır. İlgilenen okuyucular ispatın bir özetini https:// sebastianraschka.com/pdf/lecture-notes/stat453ss21/L03_perceptron_slides. pdf adresindeki ders notlarımda bulabilirler.

# Uyarlanabilir doğrusal nöronlar ve öğrenmenin yakınsaması

Bu bölümde, başka bir tek katmanlı **sinir ağı** (**NN**) türüne göz atacağız: **ADAptive LInear NEuron** (**Adaline**). Adaline, Rosenblatt'ın perceptron algoritmasından sadece birkaç yıl sonra Bernard Widrow ve doktora öğrencisi Tedd Hoff tarafından yayınlanmıştır ve ikincisinin bir gelişimi olarak kabul edilebilir (*An Adaptive "Adaline" Neuron Using Chemical "Memistors", Technical Report Number 1553-2* by *B. Widrow and colleagues*, *Stanford Electron Labs, Stanford,* CA, *October* 1960).

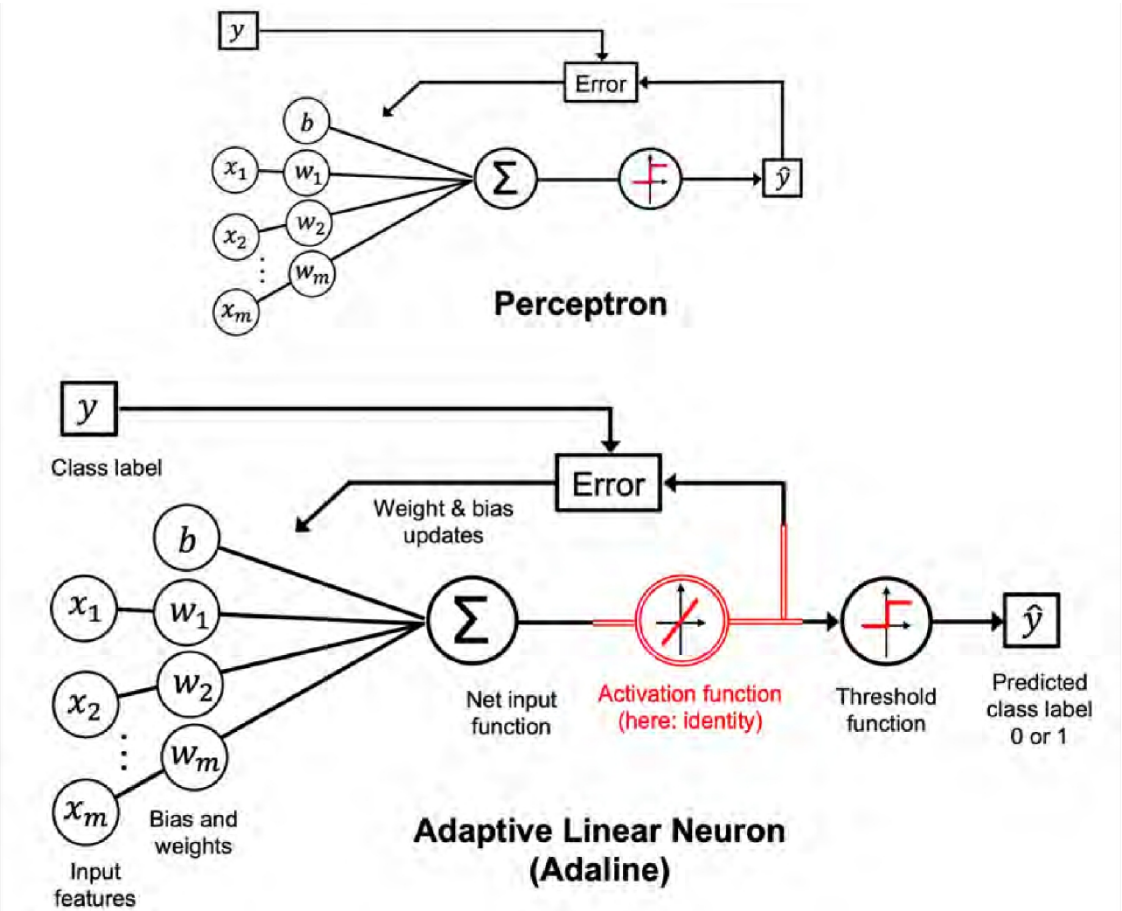
6587191CA584E

Adaline algoritması özellikle ilginçtir çünkü sürekli kayıp fonksiyonlarının tanımlanması ve minimize edilmesine ilişkin temel kavramları göstermektedir. Bu, lojistik regresyon, destek vektör makineleri ve çok katmanlı sinir ağları gibi sınıflandırma için diğer makine öğrenimi algoritmalarının yanı sıra gelecek bölümlerde tartışacağımız doğrusal regresyon modellerini anlamak için zemin hazırlar.

Adaline kuralı (**Widrow-Hoff kuralı** olarak da bilinir) ile Rosenblatt'ın algılayıcısı arasındaki temel fark, ağırlıkların algılayıcıda olduğu gibi bir birim adım fonksiyonu yerine doğrusal bir aktivasyon fonksiyonuna dayalı olarak güncellenmesidir. Adaline'de bu doğrusal aktivasyon fonksiyonu, $\sigma(z)$, basitçe ağ girdisinin kimlik fonksiyonudur, böylece $\sigma(z) = z$ olur.

Ağırlıkları öğrenmek için doğrusal aktivasyon fonksiyonu kullanılırken, nihai tahmini yapmak için daha önce ele aldığımız birim adım fonksiyonuna benzer bir eşik fonksiyonu kullanmaya devam ediyoruz.

Perceptron ve Adaline algoritması arasındaki temel farklar *Şekil 2.9'da* vurgulanmıştır:




*Şekil 2.9: Algılayıcı ve Adaline algoritması arasında bir karşılaştırma*

*Şekil 2.9*'da gösterildiği gibi, Adaline algoritması model hatasını hesaplamak ve ağırlıkları güncellemek için gerçek sınıf etiketlerini doğrusal aktivasyon fonksiyonunun sürekli değerli çıktısıyla karşılaştırır. Bunun aksine, algılayıcı gerçek sınıf etiketlerini tahmin edilen sınıf etiketleriyle karşılaştırır.

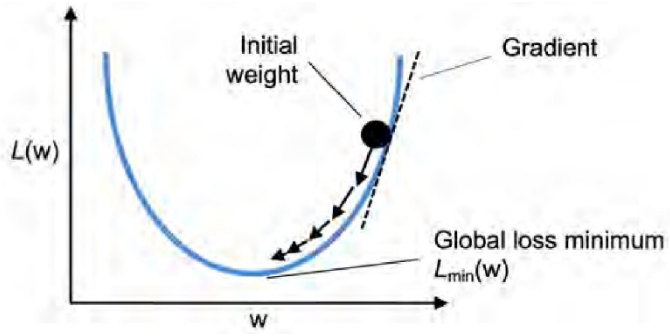
## Gradyan inişi ile kayıp fonksiyonlarını minimize etme

Denetimli makine öğrenimi algoritmalarının temel bileşenlerinden biri, öğrenme süreci sırasında optimize edilecek olan tanımlanmış bir **amaç fonksiyonudur.** Bu amaç fonksiyonu genellikle minimize etmek istediğimiz bir kayıp veya maliyet fonksiyonudur. Adaline örneğinde, model parametrelerini öğrenmek için $L$ kayıp fonksiyonunu, hesaplanan sonuç ile gerçek sınıf etiketi arasındaki **ortalama karesel hata** (**MSE**) olarak tanımlayabiliriz:

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \phi(z^{(i)})\right)^2$$

Bu sürekli doğrusal aktivasyon fonksiyonunun temel avantajı, birim adım fonksiyonunun aksine, kayıp fonksiyonunun türevlenebilir hale gelmesidir. Bu kayıp fonksiyonunun bir diğer güzel özelliği de dışbükey olmasıdır; böylece Iris veri kümesindeki örnekleri sınıflandırmak için kayıp fonksiyonumuzu en aza indiren ağırlıkları bulmak için **gradyan inişi** adı verilen çok basit ama güçlü bir optimizasyon algoritması kullanabiliriz.

*Şekil 2.10*'da gösterildiği gibi, gradyan inişinin arkasındaki ana fikri, yerel veya küresel bir minimum kayba ulaşılana kadar bir *tepeden aşağı tırmanmak* olarak tanımlayabiliriz. Her iterasyonda, gradyanın ters yönünde bir adım atarız, burada adım boyutu öğrenme oranının değeri ve gradyanın eğimi tarafından belirlenir (basitlik için, aşağıdaki şekil bunu yalnızca tek bir ağırlık, $w$ için görselleştirmektedir):




*Şekil 2.10: Gradyan inişi nasıl çalışır?*

Gradyan inişi kullanarak, artık kayıp fonksiyonumuz $L(\boldsymbol{w}, b)$'nin gradyanının, $\nabla L$ $\boldsymbol{w}$ $b$, ters yönünde bir adım atarak model parametrelerini güncelleyebiliriz:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}, \quad b := b + \Delta b$$

Parametre değişiklikleri, $\Delta w$ ve $\Delta b$, öğrenme oranı $\eta$ ile çarpılan negatif gradyan olarak tanımlanır:

$$\Delta w = -\eta \nabla_w L(w, b) \quad \Delta b = -\eta \nabla_b L(w, b)$$

Kayıp fonksiyonunun gradyanını hesaplamak için, kayıp fonksiyonunun her bir ağırlığa göre kısmi türevini hesaplamamız gerekir, $w_j$ :

$$\frac{\partial L}{\partial w_j} = -\frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) x_j^{(i)}$$

Benzer şekilde, kaybın önyargıya göre kısmi türevini şu şekilde hesaplarız:

$$\frac{\partial L}{\partial b} = -\frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)}))$$

Lütfen yukarıdaki paydaki 2'nin sadece sabit bir ölçeklendirme faktörü olduğunu ve algoritmayı etkilemeden bunu çıkarabileceğimizi unutmayın. Ölçekleme faktörünü kaldırmak, öğrenme oranını 2 kat değiştirmekle aynı etkiye sahiptir. Aşağıdaki bilgi kutusu bu ölçekleme faktörünün nereden kaynaklandığını açıklamaktadır.

Böylece ağırlık güncellemesini şu şekilde yazabiliriz:

$$\Delta w_j = -\eta \frac{\partial L}{\partial w_j} \text{ ve } \Delta b = -\eta \frac{\partial L}{\partial b}$$

Tüm parametreleri aynı anda güncellediğimiz için, Adaline öğrenme kuralımız şu hale gelir:

$$w = w + \Delta w \quad b = b + \Delta b$$

**Ortalama karesel hata türevi**

Eğer kalkülüs bilginiz varsa, MSE kayıp fonksiyonunun *j.* ağırlığa göre kısmi türevi aşağıdaki gibi elde edilebilir:

$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{n} \sum_i (y^{(i)} - \sigma(z^{(i)}))^2$$

$$= \frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) \frac{\partial}{\partial w_j} (y^{(i)} - \sigma(z^{(i)}))$$

$$= \frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) \frac{\partial}{\partial w_j} (y^{(i)} - \sum_j (w_j x_j^{(i)} + b))$$

$$= \frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) (-x_j^{(i)}) = -\frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) x_j^{(i)}$$

Aynı yaklaşım kısmi türevleri bulmak için de kullanılabilir $\frac{\partial L}{\partial b}$ ancak $\frac{\partial}{\partial b}(y^{(i)} - \sum_j (w_j x_j^{(i)} + b))$ -1'e eşittir ve böylece son adım şu şekilde basitleşir $-\frac{2}{n}\sum_i (y^{(i)} - \sigma(z^{(i)}))$.

 olimpikbarbell@gmail.com 

Adaline öğrenme kuralı perceptron kuralıyla aynı görünse de, $\sigma\sigma z^{(i)}$ ) ile $z^{(ii} = \boldsymbol{w}\boldsymbol{w}^T \boldsymbol{x}\boldsymbol{x}^{(ii} + b$ tamsayı bir sınıf etiketi değil, gerçek bir sayıdır. Ayrıca, ağırlık güncellemesi eğitim veri kümesindeki tüm örneklere göre hesaplanır (her eğitim örneğinden sonra parametreleri artımlı olarak güncellemek yerine), bu nedenle bu yaklaşım toplu **gradyan inişi** olarak da adlandırılır. Daha açık olmak ve bu bölümde ve bu kitapta daha sonra ilgili kavramlardan bahsederken karışıklığı önlemek için, bu süreci **tam yığın gradyan inişi** olarak adlandıracağız.

## Adaline'in Python'da Uygulanması

Perceptron kuralı ve Adaline çok benzer olduğundan, daha önce tanımladığımız perceptron uygulamasını alacağız ve `uyum` yöntemini değiştireceğiz, böylece ağırlık ve sapma parametreleri artık gradyan inişi yoluyla kayıp fonksiyonunu en aza indirerek güncellenecektir:

```
sınıf AdalineGD:
    """ADAptive LInear NEuron sınıflandırıcı.

    Parametrel
    ------------
    er eta :

    float
        Öğrenme oranı (0,0 ile 1,0 arasında)
    n_iter : int
        Eğitim veri kümesinin üzerinden geçer.
    random_state : int
        Rastgele ağırlık başlatma için rastgele sayı üreteci tohumu.
    -----------
    Nitelikler

    w_ : 1d-dizi
        Fitting sonrası
    ağırlıklar. b_ : Skaler
        Montajdan sonra sapma
    birimi. losses_ : liste
      Her bir epoktaki ortalama karesel hata kayıp fonksiyonu değerleri.

    """
    def init (self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Eğitim verilerini sığdırın.
```

```
        Parametreler
        ----------
        X : {dizi benzeri}, şekil = [n_örnekler, n_özellikler]
            Eğitim vektörleri, burada n_examples
            örnek sayısı ve
            n_özellikler, özelliklerin sayısıdır. y :
        dizi benzeri, şekil = [n_örnekler]
            Hedef değerler.

        Geri dönüşler
        -------
        self : nesne

        """
        rgen = np.random.RandomState(self.random_state) self.w_
        = rgen.normal(loc=0.0, scale=0.01,
                                  size=X.shape[1])
        self.b_ = np.float_(0.)
        self.losses_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            çıktı = self.aktivasyon(net_girdi)
            hatalar = (y - çıktı)
            self.w_ += s e l f . eta * 2.0 * X.T.dot(errors) / X . shape[0]
            self.b_ += s e l f . eta * 2.0 * errors.mean()
            kayıp = (hatalar**2).ortalama()
            self.kayıp_.append(kayıp)
        return self

    def net_girdi(self, X): """Net
        girdiyi hesapla"""
        return np.dot(X, self.w_) + self.b_

    def aktivasyon(self, X):
        """Doğrusal aktivasyonu
        hesapla""" dönüş X

    def predict(self, X):
        """Birim adımından sonra sınıf etiketini döndür"""
```

 olimpikbarbell@gmail.com 

```
        return np.where(self.activation(self.net_input(X))
                        >= 0.5, 1, 0)
```

Algılayıcıda olduğu gibi her bir eğitim örneğini değerlendirdikten sonra ağırlıkları güncellemek yerine, gradyanı tüm eğitim veri kümesine göre hesaplarız. Bias birimi için bu self aracılığıyla yapılır. eta * 2.0 * e r r o r s .mean(), burada errors kısmi türev değerlerini içeren bir dizidir $\frac{\partial}{\partial b}$.

Benzer şekilde, ağırlıkları da güncelleriz. Ancak ağırlık güncellemelerinin kısmi t ü r e v l e r aracılığıyla yapıldığına dikkat edin $\frac{\partial L}{\partial w_j}$

her bir ağırlık için hataları her bir özellik değeri i l e çarparak hesaplayabileceğimiz $x_j$ özellik değerlerini içerir:

```
        for w_j in range(self.w_.shape[0]):
            self.w_[w_j] += s e l f . eta *
                (2.0 * (X[:, w_j]*hatalar)).mean()
```

Ağırlık güncellemesini bir for döngüsü kullanmadan daha verimli bir şekilde uygulamak için, bunun yerine özellik matrisimiz ile hata vektörü arasında bir matris-vektör çarpımı kullanabiliriz:

```
self.w_ += self.eta * 2.0 * X.T.dot(errors) / X . shape[0]
```

Lütfen aktivasyon yönteminin kod üzerinde hiçbir etkisi olmadığını unutmayın çünkü bu sadece bir kimlik fonksiyonudur. Burada, bilginin tek katmanlı bir NN'de nasıl aktığı ile ilgili genel konsepti göstermek için aktivasyon fonksiyonunu (aktivasyon yöntemi ile hesaplanan) ekledik: giriş verilerinden özellikler, ağ girişi, aktivasyon ve çıkış.

Bir sonraki bölümde, özdeş olmayan, doğrusal olmayan bir aktivasyon fonksiyonu kullanan bir lojistik regresyon sınıflandırıcısı hakkında bilgi edineceğiz. Lojistik regresyon modelinin Adaline ile yakından ilişkili olduğunu, tek farkın aktivasyon ve kayıp fonksiyonu olduğunu göreceğiz.

Şimdi, önceki perceptron uygulamasına benzer şekilde, kayıp değerlerini bir self.losses_ algoritmanın eğitimden sonra yakınsayıp yakınsamadığını kontrol etmek için liste.

**Matris çarpımı**

Bir matris çarpımı gerçekleştirmek, matristeki her satırın tek bir satır vektörü olarak ele alındığı bir vektör nokta çarpımını hesaplamaya benzer. Bu vektörleştirilmiş yaklaşım daha kompakt bir gösterimi temsil eder ve NumPy kullanarak daha verimli bir hesaplama sağlar. Örneğin:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 2 \times 8 + 3 \times 9 \\ 4 \times 7 + 5 \times 8 + 6 \times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

Yukarıdaki denklemde, bir matrisi matematiksel olarak tanımlanmamış bir vektörle çarptığımızı lütfen unutmayın. Ancak, bu önceki vektörün 3×1 matris olarak kabul edildiği konvansiyonu kullandığımızı unutmayın.

Pratikte, optimum uyum için iyi bir öğrenme oranı, $\eta$, bulmak genellikle bazı deneyler gerektirir. Bu nedenle, başlangıç olarak $\eta\ \eta\ \eta\eta 1$ ve $\eta\ \eta\ \eta\eta\eta\eta\eta 1$ olmak üzere iki farklı öğrenme oranı seçelim ve Adaline uygulamasının eğitim verilerinden ne kadar iyi öğrendiğini görmek için epok sayısına karşı kayıp fonksiyonlarını çizelim.

**Hiperparametreler**

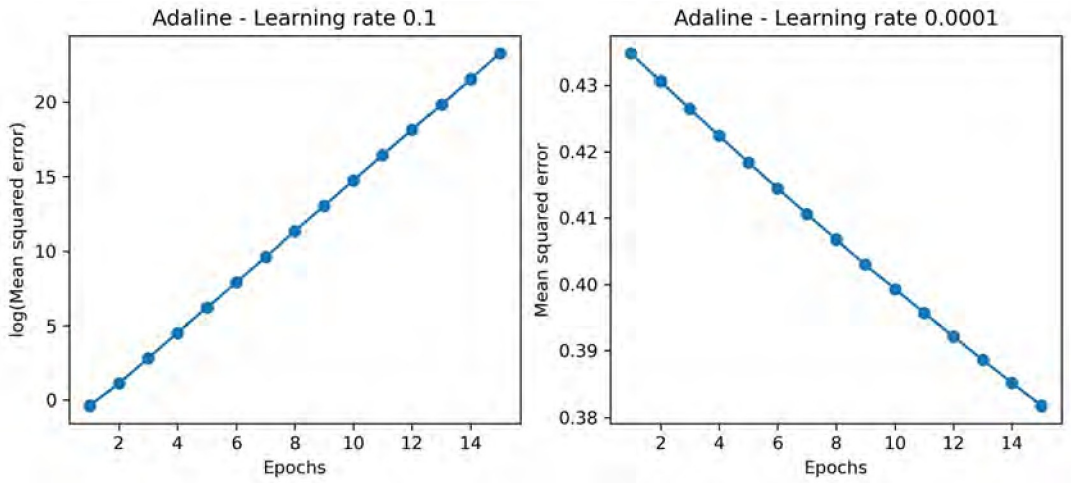
Öğrenme oranı, $\eta$ (eta) ve epok sayısı (n_iter), perceptron ve Adaline öğrenme algoritmalarının hiper parametreleri (veya ayar parametreleri) olarak adlandırılır. *Bölüm 6, Model Değerlendirme ve Hiper Parametre Ayarlama için En İyi Öğrenme Uygulamaları'n*da, sınıflandırma modelinin optimum performansını sağlayan farklı hiper parametrelerin değerlerini otomatik olarak bulmak için farklı tekniklere göz atacağız.

Şimdi iki farklı öğrenme oranı için epok sayısına karşı kayıp grafiğini çizelim:

```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
>> ada1 = AdalineGD(n_iter=15, eta=0.1).fit(X, y)
>> ax[0].plot(range(1, len(ada1.losses_) + 1),
...                np.log10(ada1.losses_), marker='o')
> >   ax[0].set_xlabel('Epochs')
>> ax[0].set_ylabel('log(Ortalama karesel hata)')
>> ax[0].set_title('Adaline - Öğrenme oranı 0.1')
>> ada2 = AdalineGD(n_iter=15, eta=0.0001).fit(X, y)
>> ax[1].plot(range(1, len(ada2.losses_) + 1),
...                ada2.losses_, marker='o')
> >   ax[1].set_xlabel('Epochs')
>> ax[1].set_ylabel('Ortalama karesel hata')
>> ax[1].set_title('Adaline - Öğrenme oranı 0,0001')
>> plt.show()
```

Ortaya çıkan kayıp fonksiyonu grafiklerinde de görebileceğimiz gibi, iki farklı türde sorunla karşılaştık. Soldaki grafik, çok büyük bir öğrenme oranı seçersek neler olabileceğini gösteriyor. Kayıp fonksiyonunu minimize etmek yerine, MSE her epokta daha da büyür, çünkü global minimumun *üzerine çıkmış* oluruz. Öte yandan, sağ grafikte kaybın azaldığını, ancak seçilen öğrenme oranının azaldığını görebiliriz,

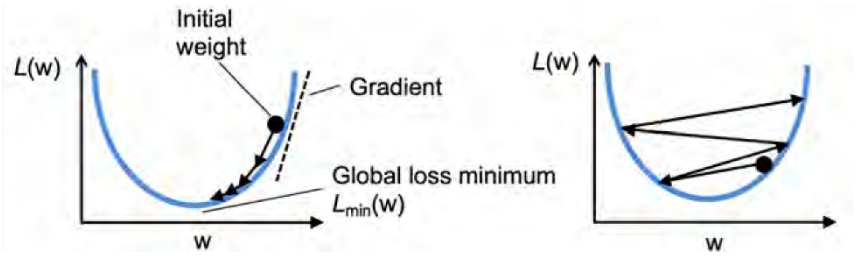
$\eta\ \ \eta\ \ \eta\eta\eta\eta 1$, o kadar küçüktür ki algoritmanın küresel kayıp minimumuna yakınsaması için çok fazla sayıda epok gerekir:




*Şekil 2.11: Optimal altı öğrenme oranları için hata grafikleri*

*Şekil 2.12,* kayıp fonksiyonunu en aza indirmek için belirli bir ağırlık parametresinin değerini değiştirirsek neler olabileceğini göstermektedir, *L*. Sol alt şekil, kaybın küresel minimum yönünde hareket ederek kademeli olarak azaldığı, iyi seçilmiş bir öğrenme oranı durumunu göstermektedir.

Bununla birlikte, sağdaki alt şekil, çok büyük bir öğrenme oranı seçersek ne olacağını göstermektedir - küresel minimumun üzerine çıkarız:




*Şekil 2.12: İyi seçilmiş bir öğrenme oranı ile çok büyük bir öğrenme oranının karşılaştırılması*

## Özellik ölçeklendirme yoluyla gradyan inişini iyileştirme

Bu kitap boyunca karşılaşacağımız birçok makine öğrenimi *algoritması,* optimum performans için *bir* tür özellik ölçeklendirmesi gerektirir; bu konuyu *Bölüm 3, Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu* ve *Bölüm 4*, *İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme* bölümlerinde daha ayrıntılı olarak ele alacağız.

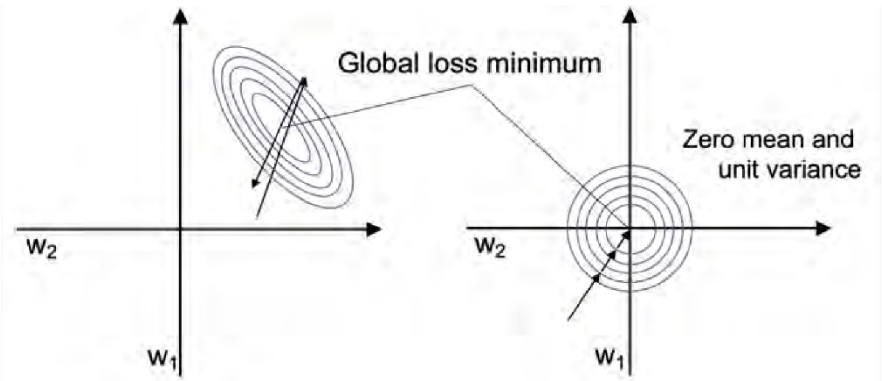
6587191CA584E

Gradyan inişi, özellik ölçeklendirmeden yararlanan birçok algoritmadan biridir. Bu bölümde, **standardizasyon** adı verilen bir özellik ölçeklendirme yöntemi kullanacağız. Bu normalleştirme prosedürü, gradyan inişli öğrenmenin daha hızlı yakınsamasına yardımcı olur; ancak orijinal veri kümesini normal dağılımlı hale getirmez. Standardizasyon, her bir özelliğin ortalamasını sıfır merkezli olacak ve her bir özelliğin standart sapması 1 (birim varyans) olacak şekilde kaydırır. Örneğin, j. özelliği standartlaştırmak için, her eğitim örneğinden örnek ortalamasını, $\mu_{jj}$ , çıkarabilir ve standart sapmasına, $\sigma_j$, bölebiliriz:

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

Burada, $x_j$ tüm eğitim örneklerinin *j.* özellik değerlerinden oluşan bir vektördür, *n,* ve bu standartlaştırma tekniği veri setimizdeki her bir özelliğe, *j,* uygulanır.

Standardizasyonun gradyan inişli öğrenmeye yardımcı olmasının nedenlerinden biri, tüm ağırlıklar (ve yanlılık) için iyi çalışan bir öğrenme oranı bulmanın daha kolay olmasıdır. Özellikler çok farklı ölçeklerdeyse, bir ağırlığı güncellemek için iyi çalışan bir öğrenme oranı, diğer ağırlığı eşit derecede iyi güncellemek için çok büyük veya çok küçük olabilir. Genel olarak, standartlaştırılmış özellikler kullanmak eğitimi dengeleyebilir, böylece optimize edicinin iyi veya optimal bir çözüm (global kayıp minimum) bulmak için daha az adımdan geçmesi gerekir. *Şekil 2.13,* ölçeklendirilmemiş özellikler (solda) ve standartlaştırılmış özellikler (sağda) ile olası gradyan güncellemelerini göstermektedir; burada eş merkezli daireler, iki boyutlu bir sınıflandırma probleminde iki model ağırlığının bir fonksiyonu olarak kayıp yüzeyini temsil etmektedir:




*Şekil 2.13: Gradyan güncellemelerinde ölçeklenmemiş ve standartlaştırılmış özelliklerin karşılaştırılması*

Standartlaştırma, yerleşik NumPy yöntemleri mean ve std kullanılarak kolayca gerçekleştirilebilir:

```
>> X_std = np.copy(X)
>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

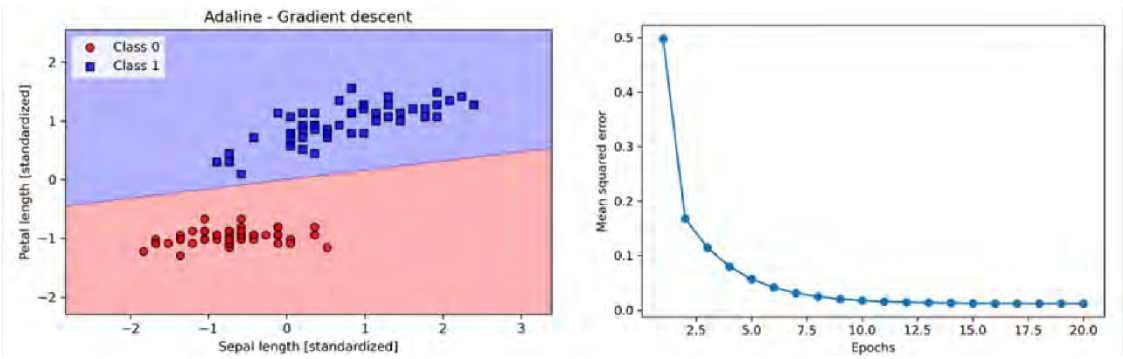
Standardizasyondan sonra, Adaline'i tekrar eğiteceğiz ve şimdi $\eta$ $\eta$ $\eta\eta$ öğrenme oranını kullanarak az sayıda epoktan sonra yakınsadığını göreceğiz:

```
>> ada_gd = AdalineGD(n_iter=20, eta=0.5)
>> ada_gd.fit(X_std, y)
```

6587191CA584E

```
>>> plot_decision_regions(X_std, y, classifier=ada_gd)
>> plt.title('Adaline - Gradyan inişi')
>> plt.xlabel('Sepal uzunluk [standartlaştırılmış]')
>> plt.ylabel('Petal uzunluğu [standartlaştırılmış]')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
>> plt.plot(range(1, len(ada_gd.losses_) + 1),
...              ada_gd.losses_, marker='o')
>> plt.xlabel('Çağlar')
>> plt.ylabel('Ortalama karesel hata')
>> plt.tight_layout()
>> plt.show()
```

Bu kodu çalıştırdıktan sonra, *Şekil 2.14'te* gösterildiği gibi, karar bölgelerinin bir şeklini ve declining kaybının bir grafiğini görmeliyiz:




*Şekil 2.14: Adaline'in karar bölgelerinin ve MSE'nin epok sayısına göre grafikleri*

Grafiklerde de görebileceğimiz gibi, Adaline standartlaştırılmış özellikler üzerinde eğitim aldıktan sonra artık yakınsamıştır. Ancak, tüm çiçek örnekleri doğru sınıflandırılmış olsa bile MSE'nin sıfır olmadığını unutmayın.

## Büyük ölçekli makine öğrenimi ve stokastik gradyan inişi

Önceki bölümde, tüm eğitim veri kümesinden hesaplanan kayıp gradyanının ters yönünde bir adım atarak bir kayıp fonksiyonunun nasıl en aza indirileceğini öğrendik; bu nedenle bu yaklaşım bazen tam yığın gradyan inişi olarak da adlandırılır. Şimdi milyonlarca veri noktasından oluşan çok büyük bir veri kümesine sahip olduğumuzu düşünün, ki bu birçok makine öğrenimi uygulamasında nadir değildir. Tam toplu gradyan inişini çalıştırmak bu tür senaryolarda hesaplama açısından oldukça maliyetli olabilir, çünkü global minimuma doğru her adım attığımızda tüm eğitim veri kümesini yeniden değerlendirmemiz gerekir.

Toplu gradyan inişi algoritmasına popüler bir alternatif, bazen yinelemeli veya çevrimiçi gradyan inişi olarak da adlandırılan **stokastik gradyan** inişidir (**SGD**). Ağırlıkları tüm eğitim örnekleri üzerinde biriken hataların toplamına göre güncellemek yerine, $\boldsymbol{x}^{(i)}$ :

$$\Delta w_{ij} = \frac{2\eta}{n} \sum_i (y^{(i)} - \sigma z^{(i)})) x_{ij}^{(i)}$$

Örneğin, her eğitim örneği için parametreleri aşamalı olarak güncelleriz:

$$\Delta w_{ij} = \eta \, \eta y^{(i)} - \sigma z^{(i)})) x_{ij}^{(i)} , \Delta b = \eta \, \eta y^{(i)} - \sigma z))^{(i)}$$

SGD, gradyan inişinin bir yaklaşımı olarak düşünülebilse de, daha sık ağırlık güncellemeleri nedeniyle tipik olarak çok daha hızlı uzlaşmaya ulaşır. Her *bir gradyan* tek bir eğitim örneğine dayalı olarak hesaplandığından, hata yüzeyi gradyan inişine göre daha gürültülüdür, bu da daha sonra *Bölüm 11, Sıfırdan Çok Katmanlı Yapay Sinir Ağı Uygulaması*'nda göreceğimiz gibi, doğrusal olmayan kayıp fonksiyonlarıyla çalışıyorsak SGD'nin sığ yerel minimumlardan daha kolay kaçabilmesi avantajına sahip olabilir. SGD ile tatmin edici sonuçlar elde etmek için, eğitim verilerini rastgele bir sırada sunmak önemlidir; ayrıca, döngüleri önlemek için her dönem için eğitim veri kümesini karıştırmak istiyoruz.

**Eğitim sırasında öğrenme oranının ayarlanması**

SGD uygulamalarında, sabit öğrenme oranı, $\eta$, genellikle zaman içinde azalan uyarlanabilir bir öğrenme o r a n ı ile değiştirilir, örneğin:

$$\frac{c_1}{[\text{iterasyon sayısı}] + c_2}$$

Burada $c_1$ ve $c_2$ sabitlerdir. SGD'nin global kayıp minimumuna değil, ona çok yakın bir alana ulaştığını unutmayın. Ve uyarlanabilir bir öğrenme oranı kullanarak, minimum kayıp için daha fazla tavlama elde edebiliriz.

SGD'nin bir diğer avantajı da **çevrimiçi öğrenme** için kullanabilmemizdir. Çevrimiçi öğrenmede, modelimiz yeni eğitim verileri geldikçe anında eğitilir. Bu, özellikle web uygulamalarındaki müşteri verileri gibi büyük miktarlarda veri biriktiriyorsak kullanışlıdır. Çevrimiçi öğrenmeyi kullanarak, sistem değişikliklere anında adapte olabilir ve depolama alanı bir sorunsa, model güncellendikten sonra eğitim verileri atılabilir.

**Mini toplu gradyan inişi**

Tam toplu gradyan inişi ile SGD arasında bir uzlaşma **mini toplu gradyan inişi** olarak adlandırılır. Mini toplu gradyan inişi, eğitim verilerinin daha küçük alt kümelerine, örneğin bir seferde 32 eğitim örneğine tam toplu gradyan inişi uygulanması olarak anlaşılabilir. Tam yığın gradyan inişine göre avantajı, daha sık ağırlık güncellemeleri nedeniyle mini yığınlar aracılığıyla yakınsamaya daha hızlı ulaşılmasıdır. Ayrıca, mini toplu öğrenme, SGD'deki eğitim örnekleri üzerindeki for döngüsünü, doğrusal cebir kavramlarından yararlanan vektör izli işlemlerle değiştirmemize olanak tanır (örneğin, bir nokta çarpımı aracılığıyla ağırlıklı bir toplam uygulamak), bu da öğrenme algoritmamızın hesaplama verimliliğini daha da artırabilir.

Adaline öğrenme kuralını gradyan inişi kullanarak zaten uyguladığımız için, öğrenme algoritmasını SGD aracılığıyla ağırlıkları güncelleyecek şekilde değiştirmek için yalnızca birkaç ayarlama yapmamız gerekiyor. Fit yönteminin içinde, artık her eğitim örneğinden sonra ağırlıkları güncelleyeceğiz. Ayrıca, çevrimiçi öğrenme için ağırlıkları yeniden başlatmayan ek bir partial_fit yöntemi uygulayacağız. Algoritmamızın eğitimden sonra yakınsayıp yakınsamadığını kontrol etmek için, kaybı her epoktaki eğitim örneklerinin ortalama kaybı olarak hesaplayacağız. Ayrıca, kayıp fonksiyonunu optimize ederken tekrarlayan döngülerden kaçınmak için her epoktan önce eğitim verilerini karıştırmak için bir seçenek ekleyeceğiz; random_state parametresi aracılığıyla,

```
sınıf AdalineSGD:
    """ADAptive LInear NEuron sınıflandırıcı.

    Parametrel
    ------------
    er eta :

    float
        Öğrenme oranı (0,0 ile 1,0 arasında)
    n_iter : int
        Eğitim veri kümesinin üzerinden
    geçer. shuffle : bool (varsayılan:
    True)
        Döngüleri önlemek için True ise her epokta eğitim
        verilerini karıştırır.
    random_state : int
        Rastgele ağırlık başlatma için rastgele sayı üreteci tohumu.
```

tekrarlanabilirlik için rastgele bir tohumun belirlenmesine izin veriyoruz:

 olimpikbarbell@gmail.com 

```
    Nitelikler
    -----------
    w_ : 1d-dizi
        Fitting sonrası
    ağırlıklar. b_ : Skaler
        Montajdan sonra sapma
    birimi. losses_ : liste
        Her epoktaki tüm eğitim örneklerinin ortalaması alınmış
        ortalama karesel hata kayıp fonksiyonu değeri.


    """
    def init (self, eta=0.01, n_iter=10,
                    shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        self.random_state = random_state

    def fit(self, X, y):
        """ Eğitim verilerini

        sığdırın. Parametreler
        ----------
        X : {dizi benzeri}, shape = [n_examples, n _ f e a t u r e s ]
            Eğitim vektörleri, burada n_examples örnek sayısı ve
            n_features özellik sayısıdır.
        y : dizi benzeri, shape = [n_examples]
            Hedef değerler.

        Geri dönüşler
        -------
        self : nesne

        """
        self._initialize_weights(X.shape[1])
        self.losses_ = []
        for i in range(self.n_iter):
            if self.shuffle:
```

```
                X, y = self._shuffle(X, y)
            kayıplar = []
            for xi, target in zip(X, y):
                losses.append(self._update_weights(xi, target))
            avg_kayıp =
            np.mean(kayıplar)
            self.kayıplar_.append(avg_kayı
            p)
        return self

    def partial_fit(self, X, y):
        """Ağırlıkları yeniden başlatmadan eğitim verilerini
        sığdır""" if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y): self._update_weights(xi,
                target)
        Başka:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """Eğitim verilerini
        karıştır"""
        r = self.rgen.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m ):
        """Ağırlıkları küçük rastgele sayılarla başlatın""" self.rgen =
        np.random.RandomState(self.random_state) self.w_ =
        s e l f . rgen.normal(loc=0.0, scale=0.01,
                                        size=m)
        self.b_ = np.float_(0.)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """Ağırlıkları güncellemek için Adaline öğrenme kuralını
        uygulayın""" output = self.activation(self.net_input(xi))
        hata = (hedef - çıktı)
        self.w_ += s e l f . eta * 2.0 * xi * (hata)
        self.b_ += s e l f . eta * 2.0 * hata
        kayıp = hata**2
        dönüş kaybı
```

 olimpikbarbell@gmail.com 

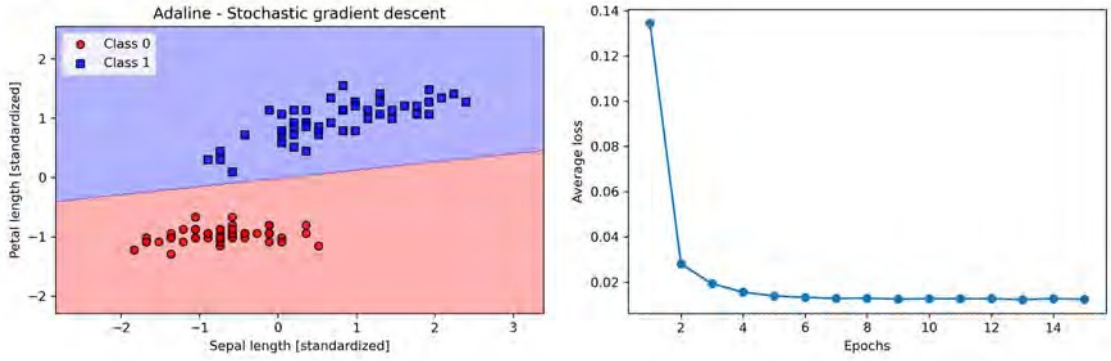
```
    def net_girdi(self, X): """Net
        girdiyi hesapla"""
        return np.dot(X, self.w_) + self.b_

    def aktivasyon(self, X):
        """Doğrusal aktivasyonu hesapla"""
        dönüş X

    def predict(self, X):
        """Birim adımından sonra sınıf etiketini döndür"""
        return np.where(self.activation(self.net_input(X))
                        >= 0.5, 1, 0)
```

Şu anda AdalineSGD sınıflandırıcısında kullandığımız _shuffle yöntemi şu şekilde çalışır: np.random'daki permütasyon işlevi aracılığıyla, 0 ila 100 aralığında benzersiz sayılardan oluşan rastgele bir dizi üretiriz. Bu sayılar daha sonra özellik matrisimizi ve sınıf etiketi vektörümüzü karıştırmak için indeks olarak kullanılabilir.

Daha sonra AdalineSGD sınıflandırıcısını eğitmek için fit yöntemini kullanabilir ve plot_decision_regions eğitim sonuçlarımızı çizmek için:

```
>> ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>> ada_sgd.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada_sgd)
>> plt.title('Adaline - Stokastik gradyan inişi')
>> plt.xlabel('Sepal uzunluk [standartlaştırılmış]')
>> plt.ylabel('Petal uzunluğu [standartlaştırılmış]')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
>> plt.plot(range(1, len(ada_sgd.losses_) + 1), ada_sgd.losses_,
...             marker='o')
>>  plt.xlabel('Çağlar')
>> plt.ylabel('Ortalama kayıp')
>> plt.tight_layout()
>> plt.show()
```

Önceki kod örneğini çalıştırarak elde ettiğimiz iki grafik *Şekil 2.15*'te gösterilmektedir:




*Şekil 2.15: SGD kullanarak bir Adaline modelini eğiten karar bölgeleri ve ortalama kayıp grafikleri*

Gördüğünüz gibi, ortalama kayıp oldukça hızlı bir şekilde düşüyor ve 15 epoktan sonraki nihai karar sınırı toplu gradyan inişi Adaline'e benziyor. Modelimizi güncellemek istersek, örneğin veri akışı olan bir çevrimiçi öğrenme senaryosunda, tek tek eğitim örnekleri üzerinde partial_fit yöntemini çağırabiliriz - örneğin, `ada_sgd.partial_fit(X_std[0, :], y[0])`.

# Özet

Bu bölümde, süper-görüntülü öğrenme için doğrusal sınıflandırıcıların temel kavramlarını iyi bir şekilde anladık. Bir algılayıcıyı uyguladıktan sonra, gradyan inişinin vektörleştirilmiş bir uygulaması ve SGD aracılığıyla çevrimiçi öğrenme yoluyla uyarlanabilir doğrusal nöronları nasıl verimli bir şekilde eğitebileceğimizi gördük.

Artık Python'da basit sınıflandırıcıların nasıl uygulanacağını gördüğümüze göre, h e m akademide hem de endüstride yaygın olarak kullanılan daha gelişmiş ve güçlü makine öğrenimi sınıflandırıcılarına erişmek için Python scikit-learn makine öğrenimi kütüphanesini kullanacağımız bir sonraki bölüme geçmeye hazırız.

Perceptron ve Adaline algoritmalarını uygulamak için kullandığımız nesne yönelimli yaklaşım, bu bölümde kullandığımız aynı temel kavramlara dayalı olarak uygulanan scikit-learn API'sini anlamamıza yardımcı olacaktır: fit ve predict yöntemleri. Bu temel kavramlara dayanarak, sınıf olasılıklarını modellemek için lojistik regresyon ve doğrusal olmayan karar sınırlarıyla çalışmak için destek vektör makineleri hakkında bilgi edineceğiz. Ayrıca, genellikle sağlam topluluk sınıflandırıcılarında birleştirilen farklı bir denetimli öğrenme algoritması sınıfı olan ağaç tabanlı algoritmaları tanıtacağız.

 olimpikbarbell@gmail.com 

# Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.com

# 3

# Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu

Bu bölümde, hem akademide hem de endüstride yaygın olarak kullanılan popüler ve güçlü makine öğrenimi algoritmaları arasında bir tura çıkacağız. Sınıflandırma için çeşitli denetimli öğrenme algoritmaları arasındaki farkları öğrenirken, aynı zamanda bireysel güçlü ve zayıf yönleri hakkında bir takdir geliştireceğiz. Ayrıca, bu algoritmaları verimli ve üretken bir şekilde kullanmak için kullanıcı dostu ve tutarlı bir arayüz sunan scikit-learn kütüphanesi ile ilk adımlarımızı atacağız.

Bu bölüm boyunca ele alınacak konular aşağıdaki gibidir:

- Lojistik regresyon, destek vektör makineleri, karar ağaçları ve *k-en yakın* komşular gibi sınıflandırma için sağlam ve popüler algoritmalara giriş
- Kullanıcı dostu bir Python API'si aracılığıyla çok çeşitli makine öğrenimi algoritmaları sağlayan scikit-learn makine öğrenimi kütüphanesini kullanan örnekler ve açıklamalar
- Doğrusal ve doğrusal olmayan karar sınırlarına sahip sınıflandırıcıların güçlü ve zayıf yönleri hakkında tartışmalar

## Bir sınıflandırma seçme algoritması

Belirli bir problem görevi için uygun bir sınıflandırma algoritması seçmek pratik ve deneyim gerektirir; her algoritmanın kendine özgü tuhaflıkları vardır ve belirli varsayımlara dayanır. David H. Wolpert'in **ücretsiz öğle yemeği teoremini** açıklamak gerekirse, tek bir sınıflandırıcı tüm olası senaryolarda en iyi şekilde çalışmaz (*The Lack of A Priori Distinctions Between Learning Algorithms*, *Wolpert, David H*, *Neural Computation 8.7* (1996): 1341-1390). Uygulamada, belirli bir sorun için en iyi modeli seçmek üzere en az birkaç farklı öğrenme algoritmasının performansını karşılaştırmanız her zaman tavsiye edilir; bunlar özellik veya örnek sayısı, veri kümesindeki gürültü miktarı ve sınıfların doğrusal olarak ayrılabilir olup olmaması açısından farklılık gösterebilir.

6587191CA584E

Sonuç olarak, bir sınıflandırıcının performansı (hesaplama performansının yanı sıra tahmin gücü) büyük ölçüde öğrenme için mevcut olan temel verilere bağlıdır. Denetimli bir makine öğrenimi algoritmasının eğitiminde yer alan beş ana adım aşağıdaki gibi özetlenebilir:

1. Özelliklerin seçilmesi ve etiketli eğitim örneklerinin toplanması
2. Bir performans ölçütü seçme
3. Bir öğrenme algoritması seçme ve bir modeli eğitme
4. Modelin performansının değerlendirilmesi
5. Algoritma ayarlarının değiştirilmesi ve modelin ayarlanması.

Bu kitabın yaklaşımı makine öğrenimi bilgisini adım adım inşa etmek olduğundan, bu bölümde temel olarak farklı algoritmaların ana kavramlarına odaklanacağız ve kitabın ilerleyen bölümlerinde daha ayrıntılı tartışmalar için özellik seçimi ve ön işleme, performans ölçümleri ve hiperparametre ayarlama gibi konuları tekrar ziyaret edeceğiz.

# Scikit-learn ile ilk adımlar - bir perceptron eğitimi

*Bölüm 2*'de, *Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi'nde*, Python ve NumPy'de kendi başımıza uyguladığımız sınıflandırma için iki ilgili öğrenme algoritması, **perceptron** kuralı ve **Adaline** hakkında bilgi edindiniz. Şimdi, belirtildiği gibi, kullanıcı dostu ve tutarlı bir arayüz ile çeşitli sınıflandırma algoritmalarının son derece optimize edilmiş bir uygulamasını birleştiren scikit-learn API'sine bir göz atacağız. Scikit-learn kütüphanesi yalnızca çok çeşitli öğrenme algoritmaları sunmakla kalmaz, aynı zamanda verileri önceden işlemek ve modellerimizde ince ayar yapmak ve değerlendirmek için birçok kullanışlı işlev sunar. Bunu, temel kavramlarla birlikte *Bölüm 4*, *İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme* ve *Bölüm 5*, *Boyut Azaltma Yoluyla Verileri Sıkıştırma'da* daha ayrıntılı olarak tartışacağız.

Scikit-learn kütüphanesini kullanmaya başlamak için, *Bölüm 2*'de uyguladığımıza benzer bir perceptron modelini eğiteceğiz. Basitlik açısından, aşağıdaki bölümler boyunca zaten aşina olduğumuz **Iris veri setini** kullanacağız. Iris veri kümesi, algoritmaları test etmek ve denemek için sıklıkla kullanılan basit ancak popüler bir veri kümesi olduğu için scikit-learn aracılığıyla zaten mevcuttur. Önceki bölüme benzer şekilde, görselleştirme amacıyla Iris veri setinden yalnızca iki özellik kullanacağız.

Bu 150 çiçek örneğinin taç yaprak uzunluğu ve taç yaprak genişliğini özellik matrisi `X`'e ve çiçek türlerinin karşılık gelen sınıf etiketlerini de vektör dizisi `y`'ye atayacağız:

```
>> from sklearn import veri kümeleri
>> import numpy as np
>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>> y = iris.target
```

olimpikbarbell@gmail.com

```
>> print('Sınıf etiketleri:',
np.unique(y)) Sınıf etiketleri: [0 1 2]
```

np.unique(y) işlevi iris.target dosyasında saklanan üç benzersiz sınıf etiketini döndürür ve görebileceğimiz gibi, Iris çiçek sınıfı adları olan Iris-setosa, Iris-versicolor ve Iris-virginica zaten tam sayı olarak saklanmaktadır (burada: 0, 1, 2). Birçok scikit-learn fonksiyonu ve sınıf metodu string formatındaki sınıf etiketleriyle de çalışsa da, teknik aksaklıklardan kaçınmak ve daha küçük bellek ayak izi nedeniyle hesaplama performansını artırmak için tamsayı etiketlerinin kullanılması önerilen bir yaklaşımdır; ayrıca, sınıf etiketlerinin tamsayı olarak kodlanması çoğu makine öğrenimi kütüphanesi arasında yaygın bir gelenektir.

Eğitilmiş bir modelin görünmeyen veriler üzerinde ne kadar iyi performans gösterdiğini değerlendirmek için veri kümesini ayrı eğitim ve test veri kümelerine ayıracağız. *Bölüm 6*'da, *Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Uygulamaları Öğrenme bölümünde,* model değerlendirme ile ilgili en iyi uygulamaları daha ayrıntılı olarak tartışacağız. Scikit-learn'in model_selection modülündeki train_test_split işlevini kullanarak X ve y dizilerini rastgele bir şekilde

```
>> from sklearn.model_selection import train_test_split
>> X_train, X_test, y_train, y_test = train_test_split(
...         X, y, test_size=0.3, random_state=1, stratify=y
... )
```

yüzde 30 test verisine (45 örnek) ve yüzde 70 eğitim verisine (105 örnek) böldük:

train_test_split fonksiyonunun bölme işleminden önce eğitim veri kümelerini dahili olarak karıştırdığını unutmayın; aksi takdirde, sınıf 0 ve sınıf 1'deki tüm örnekler eğitim veri kümelerinde yer alırdı ve test veri kümesi sınıf 2'den 45 örnekten oluşurdu. random_state parametresi aracılığıyla, bölme işleminden önce veri kümelerini karıştırmak için kullanılan dahili sözde rastgele sayı üreteci için sabit bir rastgele tohum (random_state=1) sağladık. Böyle sabit bir random_state kullanmak sonuçlarımızın tekrarlanabilir olmasını sağlar.

Son olarak, stratify=y aracılığıyla katmanlaştırma için yerleşik destekten yararlandık. Bu bağlamda, tabakalama, train_test_split yönteminin, girdi veri kümesiyle aynı oranlarda sınıf etiketine sahip eğitim ve test alt kümeleri döndürmesi anlamına gelir. Bunun gerçekten böyle olduğunu doğrulamak için NumPy'nin bir dizideki her bir değerin oluşum sayısını sayan bincount

```
>> print('y'deki etiket sayıları:', np.bincount(y))
y'deki etiket sayıları: [50 50 50]
>> print('y_train'deki etiket sayıları:', np.bincount(y_train))
y_train'deki etiket sayıları: [35 35 35]
>> print('y_test içindeki etiket sayıları:',
np.bincount(y_test)) Etiketler y_test içinde sayılır: [15 15
15]
```

fonksiyonunu kullanabiliriz:

*Bölüm 2*'deki **gradyan inişi** örneğinde gördüğümüz gibi, birçok makine öğrenimi ve optimizasyon algoritması da optimum performans için özellik ölçeklendirmesi gerektirir. Burada, scikit-learn'in ön işleme modülündeki StandardScaler sınıfını kullanarak özellikleri standartlaştıracağız:

```
>> from sklearn.preprocessing import StandardScaler
>> sc = StandardScaler()
>> sc.fit(X_train)
>> X_train_std = sc.transform(X_train)
>> X_test_std = sc.transform(X_test)
```

Önceki kodu kullanarak, StandardScaler sınıfını ön işleme modülünden yükledik ve sc değişkenine atadığımız yeni bir StandardScaler nesnesini başlattık. StandardScaler, fit yöntemini kullanarak eğitim verilerinden her bir özellik boyutu için $\mu$ (örnek ortalaması) ve $\sigma$ (standart sapma) parametrelerini tahmin etti. Dönüştürme yöntemini çağırarak, eğitim verilerini bu tahmini parametreleri ($\mu$ ve $\sigma$) kullanarak standartlaştırdık. Test veri setini standartlaştırmak için aynı ölçeklendirme parametrelerini kullandığımızı unutmayın, böylece hem eğitim hem de test veri setindeki değerler birbiriyle karşılaştırılabilir.

Eğitim verilerini standartlaştırdıktan sonra artık bir perceptron modelini eğitebiliriz. Scikit-learn'deki çoğu algoritma, varsayılan olarak **bire karşı dinlenme** (**OvR**) yöntemi aracılığıyla çok sınıflı sınıflandırmayı zaten desteklemektedir; bu da üç çiçek sınıfını **tek seferde** perceptron'a beslememize

```
>> from sklearn.linear_model import Perceptron
>> ppn = Perceptron(eta0=0.1, random_state=1)
>> ppn.fit(X_train_std, y_train)
```

olanak tanır. Kod aşağıdaki gibidir:

Scikit-learn arayüzü size *Bölüm 2*'deki perceptron uygulamamızı hatırlatacaktır. Perceptron sınıfını linear_model modülünden yükledikten sonra, yeni bir Perceptron nesnesi başlattık ve modeli fit yöntemiyle eğittik. Burada, model parametresi, eta0, kendi perceptron uygulamamızda kullandığımız öğrenme oranı, eta, ile eşdeğerdir.

*Bölüm 2*'den hatırlayacağınız gibi, uygun bir öğrenme oranı bulmak biraz deneysel çalışma gerektirir. Eğer öğrenme oranı çok büyükse, algoritma global kayıp minimumunu aĞacaktır. Öğrenme oranı çok küçükse, algoritma yakınsamaya kadar daha fazla epok gerektirecektir, bu da öğrenmeyi yavaşlatabilir-özellikle büyük veri kümeleri için. Ayrıca, her epoktan sonra eğitim veri kümesinin ilk karıştırılmasının tekrarlanabilirliğini sağlamak için random_state parametresini kullandık.

Scikit-learn'de bir modeli eğittikten sonra, tıpkı *Bölüm 2*'deki kendi perceptron uygulamamızda olduğu gibi predict yöntemiyle tahminler yapabiliriz. Kod aşağıdaki gibidir:

```
>> y_pred = ppn.predict(X_test_std)
>> print('Yanlış sınıflandırılmış örnekler: d' % (y_test != y_pred).sum()) Yanlış
sınıflandırılmış örnekler: 1
```

Kodu çalıştırdığımızda, algılayıcının 45 çiçek örneğinden 1'ini yanlış sınıflandırdığını görebiliriz. Dolayısıyla, test veri kümesindeki yanlış sınıflandırma hatası yaklaşık 0,022 veya yüzde 2,2'dir[1] ≈

olimpikbarbell@gmail.com


0,022 .

$\left(\frac{}{45}\right)-$

**Doğruluğa karşı sınıflandırma hatası**

Birçok makine öğrenimi uygulayıcısı, yanlış sınıflandırma hatası yerine, basitçe aşağıdaki şekilde hesaplanan bir modelin sınıflandırma doğruluğunu rapor eder:

*1*- hata = 0,978 veya yüzde 97,8

Sınıflandırma hatasını mı yoksa doğruluğu mu kullanacağımız sadece bir tercih meselesidir.

Scikit-learn'in metrikler modülü aracılığıyla kullanılabilen çok çeşitli farklı performans metriklerini de uyguladığını unutmayın. Örneğin, perceptron'un test veri kümesi üzerindeki sınıflandırma doğruluğunu aşağıdaki gibi hesaplayabiliriz:

```
>> from sklearn.metrics import accuracy_score
>> print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
Doğruluk: 0.978
```

Burada, y_test gerçek sınıf etiketleri ve y_pred daha önce tahmin ettiğimiz sınıf etiketleridir. Doğal olarak, scikit-learn'deki her sınıflandırıcı, burada gösterildiği gibi, tahmin çağrısını accuracy_score ile birleştirerek bir sınıflandırıcının tahmin doğruluğunu hesaplayan bir puan yöntemine sahiptir:

```
>> print('Doğruluk: %.3f' % ppn.score(X_test_std, y_test))
Doğruluk: 0.978
```

**Aşırı Uyum**

Modellerimizin performansını bu bölümdeki test veri setine göre değerlendireceğimizi unutmayın. *Bölüm 6*'da, aşırı uyumu tespit etmek ve önlemek için öğrenme eğrileri gibi grafiksel analizler de dahil olmak üzere faydalı teknikler hakkında bilgi edineceksiniz. Bu bölümün ilerleyen kısımlarında tekrar değineceğimiz aşırı uyum, modelin eğitim verilerindeki örüntüleri iyi yakaladığı ancak görülmeyen verilere iyi genelleme yapamadığı anlamına gelir.

Son olarak, yeni eğitilmiş perceptron modelimizin **karar bölgelerini** çizmek ve farklı çiçek örneklerini ne kadar iyi ayırdığını görselleştirmek için *Bölüm 2'deki* plot_decision_regions fonksiyonumuzu kullanabiliriz. Ancak, test veri kümesindeki veri örneklerini küçük dairelerle vurgulamak için küçük bir değişiklik ekleyelim:

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
def plot_decision_regions(X, y, sınıflandırıcı, test_idx=None,
                          çözünürlük=0.02):
    # işaretleyici oluşturucuyu ve renk
    haritasını ayarlayın işaretleyiciler = ('o',
    's', '^', 'v', '<')
    colors = ('red', ' blue', ' lightgreen', ' gray', ' cyan') cmap
    = ListedColormap(colors[:len(np.unique(y))])
```

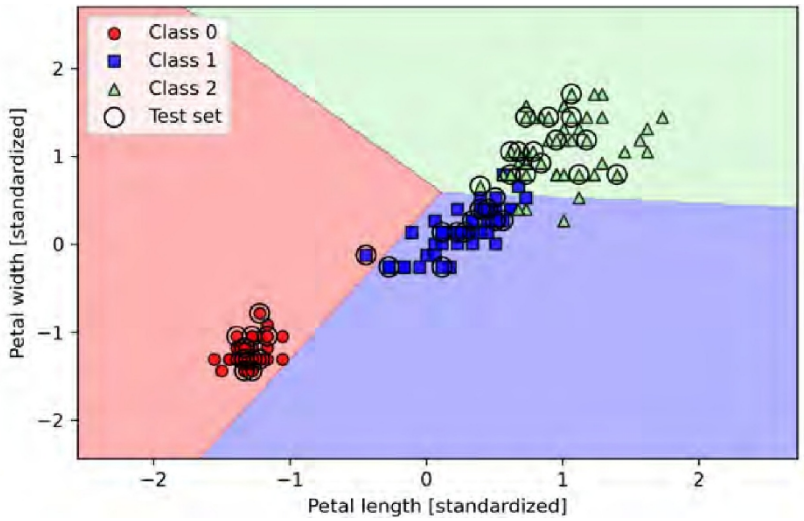
 olimpikbarbell@gmail.com 

```
# karar yüzeyini çizin
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, çözünürlük),
                       np.arange(x2_min, x2_max, çözünürlük))
lab = classifier.predict(np.array([xx1.ravel(), x x 2 . ravel()]).T) lab
= lab.reshape(xx1.shape)
plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())

# plot sınıfı örnekleri
for idx, cl in
    enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0],
                y=X[y == cl, 1],
                alpha=0.8,
                c=colors[idx],
                marker=markers[idx],
                label=f'Class {cl}',
                edgecolor='black')
# test örneklerini vurgulayın
eğer test_idx:
    # tüm örnekleri çizin
    X_test, y_test = X[test_idx, :], y[test_idx]

    plt.scatter(X_test[:, 0], X_test[:, 1],
                c='none', edgecolor='black', alpha=1.0,
                linewidth=1, marker='o',
                s=100, label='Test seti')
```

plot_decision_regions fonksiyonunda yaptığımız küçük değişiklikle, artık elde edilen grafiklerde işaretlemek istediğimiz örneklerin indekslerini belirleyebiliyoruz. Kod aşağıdaki gibidir:

```
>> X_combined_std = np.vstack((X_train_std, X_test_std))
>> y_combined = np.hstack((y_train, y_test))
> > >  plot_decision_regions(X=X_combined_std,
...                          y=y_combined,
...                          sınıflandırıcı=ppn,
...                          test_idx=range(105, 150))
>> plt.xlabel('Petal uzunluğu [standartlaştırılmış]')
>> plt.ylabel('Petal genişliği [standartlaştırılmış]')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

Ortaya çıkan grafikte de görebileceğimiz gibi, üç çiçek sınıfı doğrusal bir karar sınırı ile mükemmel bir şekilde ayrılamaz:




*Şekil 3.1: Iris veri setine takılan çok sınıflı bir algılayıcı modelin karar sınırları*

Ancak, *Bölüm 2*'deki tartışmamızdan perceptron algoritmasının mükemmel doğrusal olarak ayrılabilir olmayan veri kümelerinde asla yakınsamadığını hatırlayın, bu nedenle perceptron algoritmasının kullanımı pratikte genellikle önerilmez. İlerleyen bölümlerde, sınıflar tamamen doğrusal olarak ayrılabilir olmasa bile minimum kayba yakınsayan daha güçlü doğrusal sınıflandırıcılara bakacağız.

**Ek perceptron ayarları**

Perceptron'un yanı sıra diğer scikit-learn fonksiyonları ve sınıfları, genellikle açıklık için atladığımız ek parametrelere sahiptir. Python'daki yardım işlevini kullanarak (örneğin, help(Perceptron)) veya http://scikit-learn.org/stable/ adresindeki mükemmel scikit-learn çevrimiçi belgelerini inceleyerek bu parametreler hakkında daha fazla bilgi edinebilirsiniz.

# Sınıf olasılıklarının lojistik regresyonu ile modellenmesi

Perceptron kuralı, sınıflandırma için makine öğrenimi algoritmalarına güzel ve kolay bir giriş sunsa da, en büyük dezavantajı, sınıflar mükemmel bir şekilde erken ayrılabilir değilse asla yakınsamamasıdır. Önceki bölümdeki sınıflandırma görevi böyle bir senaryoya örnek teşkil eder. Bunun nedeni, her epokta her zaman en az bir yanlış sınıflandırılmış eğitim örneği bulunduğundan ağırlıkların sürekli olarak güncellenmesidir. Elbette, öğrenme oranını değiştirebilir ve epok sayısını artırabilirsiniz, ancak algılayıcının bu veri kümesinde asla yakınsamayacağı konusunda uyarılmalıdır.

Zamanımızı daha iyi kullanmak için, şimdi doğrusal ve ikili sınıflandırma problemleri için başka bir basit ama daha güçlü algoritmaya göz atacağız: **lojistik regresyon**. Adına rağmen lojistik regresyonun regresyon değil sınıflandırma için bir model olduğunu unutmayın.

 olimpikbarbell@gmail.com 

## Lojistik regresyon ve koşullu olasılıklar

Lojistik regresyon, uygulaması çok kolay olan ve doğrusal olarak ayrılabilen sınıflarda çok iyi performans gösteren bir sınıflandırma modelidir. Endüstride sınıflandırma için en yaygın kullanılan algoritmalardan biridir. Perceptron ve Adaline'e benzer şekilde, bu bölümdeki lojistik regresyon modeli de ikili sınıflandırma için doğrusal bir modeldir.

**Çoklu sınıflar için lojistik regresyon**

Lojistik regresyonun, **multinomial lojistik regresyon** veya **softmax regresyon** olarak bilinen çok sınıflı ortamlara kolayca genelleştirilebileceğini unutmayın. Multinomial lojistik regresyonun daha ayrıntılı kapsamı bu kitabın kapsamı dışındadır, ancak ilgilenen okuyucu https://sebastianraschka. com/pdf/lecture-notes/stat453ss21/L08_logistic slides.pdf adresindeki ders notlarımda veya https://www. youtube.com/watch?v=L0FU8NFpx4E adresinde daha fazla bilgi bulabilir.

Lojistik regresyonu çok sınıflı ortamlarda kullanmanın b i r başka yolu da daha önce tartıştığımız OvR tekniğidir.

İkili sınıflandırma için olasılıksal bir model olarak lojistik regresyonun arkasındaki ana mekaniği açıklamak için, önce olasılıkları tanıtalım: belirli bir olayın lehine olan olasılıklar. Oranlar şöyle yazılabilir

$\frac{p}{(1-p)}$ Burada $p$ pozitif olayın olasılığını temsil eder. "Olumlu olay" terimi şu anlama gelmez olarak "iyi" anlamına gelmek zorunda değildir, ancak tahmin etmek istediğimiz olayı, örneğin belirli semptomlar göz önüne alındığında bir hastanın belirli bir hastalığa sahip olma olasılığını ifade eder; pozitif olayı $y = 1$ sınıf etiketi ve semptomları $\boldsymbol{x}$ özellikleri olarak düşünebiliriz. Bu nedenle, kısalık için, $p$ olasılığını $p := p(y = 1|\boldsymbol{x})$, belirli bir örneğin $\boldsymbol{x}$ özellikleri göz önüne alındığında belirli bir sınıf 1'e ait olma koşullu olasılığı olarak tanımlayabiliriz.

Daha sonra, basitçe olasılıkların logaritması (log-odds) olan **logit** fonksiyonunu tanımlayabiliriz:

$$\text{logit}(p) = \log \frac{p}{(1 - p)}$$

*Log'un* doğal logaritma anlamına geldiğini ve bilgisayar bilimlerinde yaygın olarak kullanıldığını unutmayın. *logit* işlevi 0 ile 1 aralığındaki giriş değerlerini alır ve bunları tüm gerçek sayı aralığındaki değerlere dönüştürür.

Lojistik model altında, ağırlıklı girdiler (*Bölüm 2*'de net girdiler olarak anılacaktır) ile log-odds arasında doğrusal bir ilişki olduğunu varsayıyoruz:

$$\text{logit}(p) = w_1 x_1 + \cdots + w_m x_m + b = \sum_i w_j x_j + b = \boldsymbol{w}^T \boldsymbol{x} + b$$

Yukarıda log-odds ve net girdiler arasındaki doğrusal ilişki hakkında yaptığımız bir varsayım açıklanırken, aslında ilgilendiğimiz şey $p$ olasılığı, yani özellikleri verilen bir örneğin sınıf üyesi olma olasılığıdır. Logit fonksiyonu olasılığı bir gerçek sayı aralığına eşlerken, gerçek sayı aralığını $p$ olasılığı için bir [0, 1] aralığına geri eşlemek için bu fonksiyonun tersini

6587191CA584E

düşünebiliriz.

Logit fonksiyonunun bu tersi tipik olarak **lojistik sigmoid fonksiyonu** olarak adlandırılır ve karakteristik S şekli nedeniyle bazen basitçe **sigmoid** fonksiyonu olarak kısaltılır:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

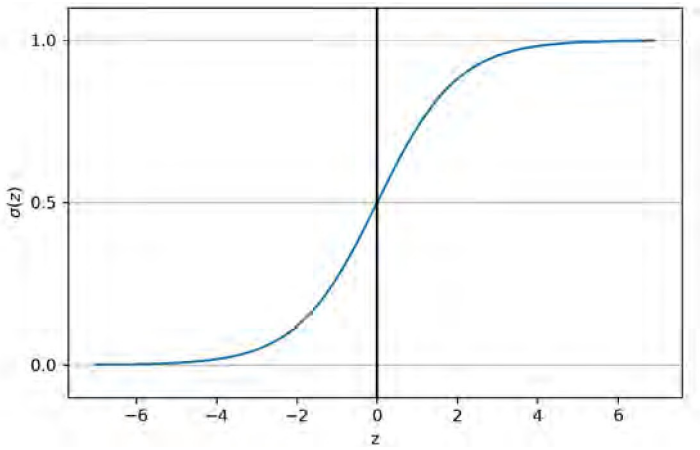
Burada, $z$ net girdi, ağırlıkların doğrusal kombinasyonu ve girdilerdir (yani, eğitim örnekleriyle ilişkili özelliklerdir):

$$z = \boldsymbol{w}^T \mathrm{x} + b$$

Şimdi, nasıl göründüğünü görmek için sigmoid fonksiyonunu -7 ila 7 aralığındaki bazı değerler için çizelim:

```
>> import matplotlib.pyplot as plt
>> import numpy as np
>> def sigmoid(z):
...         return 1.0 / (1.0 + np.exp(-z))
>> z = np.arange(-7, 7, 0.1)
>> sigma_z = sigmoid(z)
>> plt.plot(z, sigma_z)
>> plt.axvline(0.0, color='k')
>> plt.ylim(-0.1, 1.1)
>> plt.xlabel('z')
>> plt.ylabel('$\sigma (z)$')
>> # y ekseni tikleri ve kılavuz çizgisi
>> plt.yticks([0.0, 0.5, 1.0])
>> ax = plt.gca()
>> ax.yaxis.grid(True)
>> plt.tight_layout()
>> plt.show()
```

Önceki kod örneğinin çalıştırılmasının bir sonucu olarak, şimdi S-şekilli (sigmoidal) eğriyi görmeliyiz:

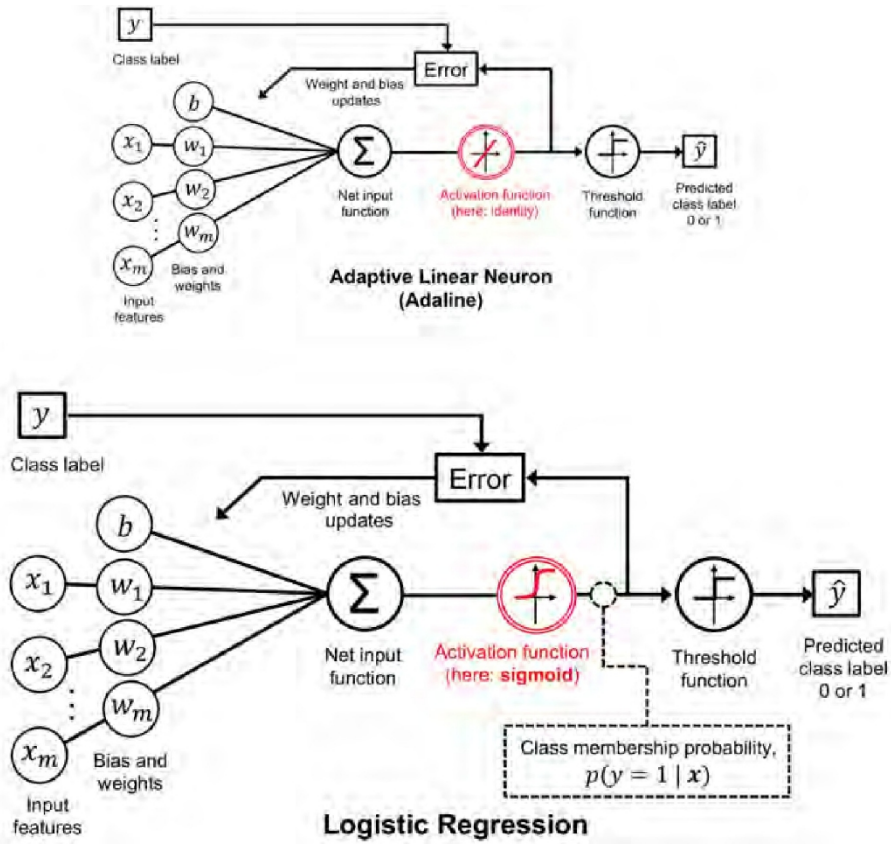



*Şekil 3.2: Lojistik sigmoid fonksiyonunun bir grafiği*

z sonsuza ($z \to \infty$) doğru giderse $\sigma(z)$'nin 1'$e$ yaklaştığını görebiliriz, çünkü $e^{-z}$ z'nin büyük değerleri için çok küçük hale gelir. Benzer şekilde, $\sigma(z)$ $z \to -\infty$ için giderek büyüyen paydanın bir sonucu olarak 0'a doğru gider. Böylece, bu sigmoid fonksiyonun girdi olarak gerçek sayı değerlerini aldığı ve bunları $\sigma(0) = 0.5$'te bir kesişme ile [0, 1] aralığındaki değerlere dönüştürdüğü sonucuna varabiliriz.

Lojistik regresyon modeli hakkında bir anlayış oluşturmak için bunu *Bölüm 2* ile ilişkilendirebiliriz. Adaline'de aktivasyon fonksiyonu olarak $\sigma(z) = z$ özdeşlik fonksiyonunu kullandık. Lojistik regresyonda bu aktivasyon fonksiyonu basitçe daha önce tanımladığımız sigmoid fonksiyonuna dönüşür.

Adaline ve lojistik regresyon arasındaki fark, tek farkın aktivasyon fonksiyonu olduğu aşağıdaki şekilde gösterilmektedir:




*Şekil 3.3: Adaline ile karşılaştırmalı lojistik regresyon*

Sigmoid fonksiyonunun çıktısı daha sonra belirli bir örneğin sınıf 1'e ait olma olasılığı olarak yorumlanır, $\sigma(z) = p(y = 1|\boldsymbol{x}\ \boldsymbol{x}\ b)$ , $x$ özellikleri göz önüne alındığında ve $w$ ve $b$ ağırlıkları ve yanlılığı ile parametrelendirilir. Örneğin, belirli bir akış örneği için $\sigma(z) = 0{,}8$ hesaplarsak, bu örneğin bir `Iris-versicolor çiçeği olma olasılığının` yüzde 80 olduğu anlamına gelir. Bu nedenle, bu çiçeğin bir `Iris-setosa` çiçeği olma olasılığı $p(y = \boldsymbol{0}|\boldsymbol{x}; \boldsymbol{w}, b) = 1 - p(y = \boldsymbol{1}|\boldsymbol{x}; \boldsymbol{w}, b) =$ 0,2 veya yüzde 20 olarak hesaplanabilir.

 olimpikbarbell@gmail.com 

Tahmin edilen olasılık daha sonra bir eşik fonksiyonu aracılığıyla basitçe ikili bir sonuca dönüştürülebilir:

$$\hat{y} = \begin{cases} 1 & \text{eğer } \sigma(z) \geq 0{,}5 \text{ ise} \\ 0 & \text{aksi takdirde} \end{cases}$$

Sigmoid fonksiyonunun önceki grafiğine bakarsak, bu aşağıdakine eşdeğerdir:

$$\hat{y} = \begin{cases} 1 & \text{ise } z \geq 0 \\ 0 & \text{aksi takdirde} \end{cases}$$

Aslında, yalnızca tahmin edilen sınıf etiketleriyle ilgilenmediğimiz, aynı zamanda sınıf üyeliği olasılığının tahmin edilmesinin özellikle yararlı olduğu (eşik fonksiyonunu uygulamadan önce sigmoid fonksiyonunun çıktısı) birçok uygulama vardır. Lojistik regresyon, örneğin hava tahmininde sadece belirli bir günde yağmur yağıp yağmayacağını tahmin etmek için değil, aynı zamanda yağmur yağma olasılığını bildirmek için de kullanılır. Benzer şekilde lojistik regresyon, belirli semptomlar göz önüne alındığında bir hastanın belirli bir hastalığa sahip olma olasılığını tahmin etmek için kullanılabilir, bu nedenle lojistik regresyon tıp alanında büyük bir popülerliğe sahiptir.

## Lojistik kayıp fonksiyonu aracılığıyla model ağırlıklarının öğrenilmesi

Lojistik regresyon modelini olasılıkları ve sınıf etiketlerini tahmin etmek için nasıl kullanabileceğimizi öğrendiniz; şimdi kısaca modelin parametrelerini, örneğin ağırlıkları ve yanlılık birimini, *w* ve *b,* nasıl uydurduğumuzdan bahsedelim:

$$L(\mathbf{w}, b \mid \mathbf{x}) = \sum_i \frac{1}{2}\left(\sigma(z^{(i)}) - y^{(i)}\right)^2$$

Adaline sınıflandırma modelimizin parametrelerini öğrenmek için bu fonksiyonu minimize ettik. Lojistik regresyon için kayıp fonksiyonunu nasıl türetebileceğimizi açıklamak için, öncelikle veri kümemizdeki tek tek örneklerin birbirinden bağımsız olduğunu varsayarak bir lojistik regresyon modeli oluşturduğumuzda maksimize etmek istediğimiz olasılığı, $\mathcal{L}$, tanımlayalım. Formül aşağıdaki gibidir:

$$\mathcal{L}(\mathbf{w}, b \mid \mathbf{x}) = p(\mathbf{y} \mid \mathbf{x}; \mathbf{w}, b) = \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}; \mathbf{w}, b) = \prod_{i=1}^{n} \sigma(z^{(i)})^{y^{(i)}} \left(1 - \sigma(z^{(i)})\right)^{1-y^{(i)}}$$

Pratikte, bu denklemin **log-olabilirlik** olarak adlandırılan (doğal) **logunu** maksimize etmek daha kolaydır fonksiyon:

$$l(\mathbf{w}, b \mid \mathbf{x}) = \log \mathcal{L}(\mathbf{w}, b \mid \mathbf{x}) = \sum_{i=1}^{n} \left[y^{(i)} \log(\sigma(z^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))\right]$$

İlk olarak, log fonksiyonunun uygulanması, olasılıkların çok küçük olması durumunda ortaya çıkabilecek sayısal düşük akış potansiyelini azaltır. İkinci olarak, faktörlerin çarpımını faktörlerin toplamına dönüştürebiliriz, bu da kalkülüsten hatırlayabileceğiniz gibi toplama hilesi yoluyla bu fonksiyonun türevini elde etmeyi kolaylaştırır.

**Olabilirlik fonksiyonunun türetilmesi**

Veriler göz önüne alındığında modelin olabilirliği için $\mathcal{L}(\mathbf{www}\ b \mid \mathbf{xx})$ ifadesini aşağıdaki gibi elde edebiliriz. Sınıf etiketleri 0 ve 1 olan ikili bir sınıflandırma problemimiz olduğu göz önüne alındığında, 1 etiketini bir Bernoulli değişkeni olarak düşünebiliriz - 1 olma olasılığı $p$ olan 0 ve 1 olmak üzere iki değer alabilir: *YYBernYpY* . Tek bir veri noktası için bu olasılığı şöyle yazabiliriz
*PPY P* 1 *P X P*(ii ) *P σPz*(ii ) ve *PPY P P X P P*(ii ) *P* 1 - *σPz*(ii ).

Bu iki ifadeyi bir araya getirerek ve stenografi kullanarak
*PPY P y*(ii | *X P X*(ii ) *P pPy*(ii | *X*(ii ), Bernoul- li değişkeninin olasılık kütle fonksiyonunu elde ederiz:

$$ppy^{(ii)} \mid x^{(ii)}) = (\sigma pz^{(ii)}))^{y^{(ii)}} (1 - \sigma pz^{(ii)}))^{1-y^{(ii)}}$$

Tüm eğitim örneklerinin bağımsız olduğu varsayımı altında, tüm olayların gerçekleşme olasılığını hesaplamak için çarpma kuralını kullanarak eğitim etiketlerinin olasılığını aşağıdaki gibi yazabiliriz:

$$\mathcal{L}(\mathbf{www}\ b \mid \mathbf{xx}) = \prod_{ii=1}^{n} ppy^{(i)} \mid \mathbf{xx}^{(i)}; \mathbf{www}\ b\mathbf{w}$$

Şimdi, Bernoulli değişkeninin olasılık kütle fonksiyonunu yerine koyarak, model parametrelerini değiştirerek maksimize etmeye çalıştığımız olabilirlik ifadesine ulaşıyoruz:

$$\mathcal{L}(\mathbf{ww}\ b \mid \mathbf{xx}) = \prod_{ii=1}^{n} (\sigma\sigma z^{(i)}))^{y^{(i)}} (1 - \sigma\sigma z^{(i)}))^{1-y^{(i)}}$$

Şimdi, bu log-likelihood fonksiyonunu maksimize etmek için gradient ascent gibi bir optimizasyon algoritması kullanabiliriz. (Gradient ascent, *Bölüm 2'de* açıklanan gradient descent ile tamamen aynı şekilde çalışır, ancak gradient ascent bir fonksiyonu minimize etmek yerine maksimize eder). Alternatif olarak, log-likelihood'u *Bölüm 2'*de olduğu gibi gradyan inişi kullanılarak minimize edilebilen bir kayıp fonksiyonu, $L$, olarak yeniden yazalım:

$$L(\mathbf{www}b) = \sum_{ii=1}^{n} [-y^{(i)} \log(\sigma(z^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))]$$

Bu kayıp fonksiyonunu daha iyi kavramak için, tek bir eğitim örneği için hesapladığımız kayba bir göz atalım:

$$L(\sigma(z), yyy, b) = -y\ yy(\sigma(z)) - (1 - yy)\ yyy(1 - \sigma(z))$$

Denkleme baktığımızda, $y = 0$ ise ilk terimin sıfır olduğunu ve $y = 1$ ise ikinci terimin sıfır olduğunu görebiliriz:

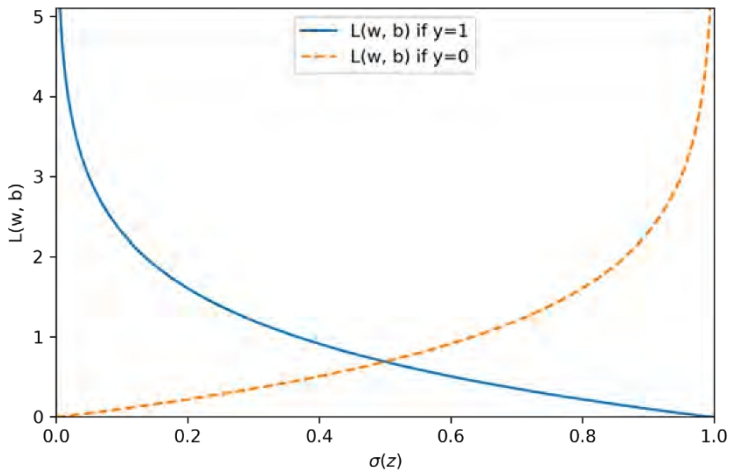
$$L(\sigma(z), yyy, b) = \begin{cases} -\log(\sigma(z)) & \text{eğer } y = 1 \text{ ise} \\ -\log(1 - \sigma z(\ )) & \text{eğer } y = y \end{cases}$$

$\sigma(z)$'nin farklı değerleri için tek bir eğitim örneğini sınıflandırma kaybını gösteren bir çizim

```
>> def kayıp_1(z):
...         return - np.log(sigmoid(z))
>> def kayıp_0(z):
...         return - np.log(1 - sigmoid(z))
>> z = np.arange(-10, 10, 0.1)
>> sigma_z = sigmoid(z)
>>> c1 = [loss_1(x) for x in z]
>> plt.plot(sigma_z, c1, label='L(w, b) if y=1')
>>> c0 = [loss_0(x) for x in z]
>> plt.plot(sigma_z, c0, linestyle='--', label='L(w, b) if y=0')
>> plt.ylim(0.0, 5.1)
>> plt.xlim([0, 1])
> >   plt.xlabel('$\sigma(z)$')
>> plt.ylabel('L(w, b)')
>>  plt.legend(loc='best')
>> plt.tight_layout()
>> plt.show()
```

oluşturmak için kısa bir kod parçacığı yazalım:

Ortaya çıkan çizim, $x$ ekseninde 0 ila 1 aralığında sigmoid aktivasyonunu (sigmoid fonksiyonuna girdiler -10 ila 10 aralığında $z$ değerleriydi) ve $y$ ekseninde ilişkili lojistik kaybı göstermektedir:




*Şekil 3.4: Lojistik regresyonda kullanılan kayıp fonksiyonunun bir grafiği*

Bir örneğin sınıf 1'e ait olduğunu doğru tahmin edersek kaybın 0'a yaklaştığını görebiliriz (sürekli çizgi). Benzer şekilde, $y$ ekseninde $y$ = 0'ı doğru tahmin edersek kaybın da 0'a yaklaştığını görebiliriz (kesikli çizgi). Ancak, tahmin yanlışsa, kayıp sonsuza doğru gider. Ana nokta, yanlış tahminleri giderek daha büyük bir kayıpla cezalandırmamızdır.

## Bir Adaline uygulamasının lojistik regresyon için bir algoritmaya dönüştürülmesi

Lojistik regresyonu kendimiz uygulayacak olsaydık, *Bölüm 2'*deki Adaline uygulamamızdaki kayıp fonksiyonu $L$'yi yeni kayıp fonksiyonu ile değiştirebilirdik:

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} \left[ -y^{(i)} \log(\sigma(z^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(z^{(i)})) \right]$$

Bunu epok başına tüm eğitim örneklerini sınıflandırma kaybını hesaplamak için kullanırız. Ayrıca, doğrusal aktivasyon fonksiyonunu sigmoid ile değiştirmemiz gerekir. Adaline kodunda bu değişiklikleri yaparsak, çalışan bir lojistik regresyon uygulaması elde ederiz. Aşağıda tam yığın gradyan inişi için bir uygulama verilmiştir (ancak aynı değişikliklerin stokastik gradyan inişi versiyonunda da yapılabileceğini unutmayın):

```
sınıf LogisticRegressionGD:
    """Gradyan iniş tabanlı lojistik regresyon sınıflandırıcısı.

    Parametrel
    ------------
    er eta :

    float
      Öğrenme oranı (0,0 ile 1,0 arasında)
    n_iter : int
      Eğitim veri kümesinin üzerinden
    geçer. random_state : int
      Rastgele ağırlık başlatma için rastgele sayı üreteci tohumu.


    Nitelikler--

    w_ : 1d-dizi
      Eğitim sonrası
    ağırlıklar. b_ : Skaler
      Montajdan sonra sapma
    birimi. losses_ : liste
      Her bir epoktaki ortalama karesel hata kayıp fonksiyonu değerleri.

    """
    def init (self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
```

 olimpikbarbell@gmail.com 

```
    def fit(self, X, y):
        """ Eğitim verilerini

        sığdırın. Parametreler
        ----------
        X : {dizi benzeri}, şekil = [n_örnekler, n_özellikler]
          Eğitim vektörleri, burada n_examples örnek sayısı
          ve n_features özellik sayısıdır.
        y : dizi benzeri, shape = [n_examples]
          Hedef değerler.

        Geri dönüşler

        self : LogisticRegressionGD örneği """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.float_(0.)
        self.losses_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            çıktı = self.aktivasyon(net_girdi)
            hatalar = (y - çıktı)
            self.w_ += s e l f . eta * 2.0 * X.T.dot(errors) / X . shape[0]
            self.b_ += s e l f . eta * 2.0 * errors.mean()
            kayıp = (-y.dot(np.log(çıktı))
                     - ((1 - y).dot(np.log(1 - çıktı))
                      / X.shape[0])
            self.losses_.append(loss)
        return self

    def net_girdi(self, X): """Net
        girdiyi hesapla"""
        return np.dot(X, self.w_) + self.b_

    def aktivasyon(self, z):
        """Lojistik sigmoid aktivasyonunu hesapla"""
        1 döndürür. / (1. + np.exp(-np.clip(z, -250, 250))
```

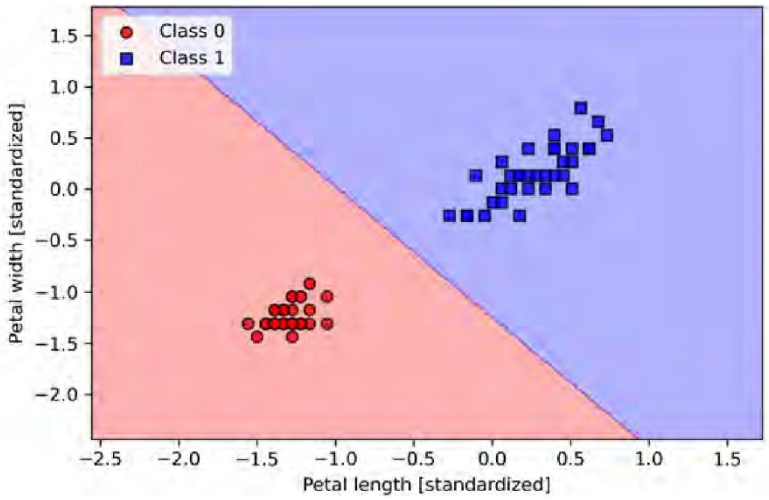
```
    def predict(self, X):
        """Birim adımından sonra sınıf etiketini döndür"""
        return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```

Bir lojistik regresyon modeli kurduğumuzda, bunun yalnızca ikili sınıflandırma görevleri için işe yaradığını aklımızda tutmamız gerekir.

Şimdi sadece setosa ve versicolor çiçeklerini (sınıf 0 ve 1) ele alalım ve lojistik regresyon uygulamamızın çalışıp çalışmadığını kontrol edelim:

```
>> X_train_01_subset = X_train_std[(y_train == 0) | (y_train == 1)]
>> y_train_01_subset = y_train[(y_train == 0) | (y _ t r a i n  == 1)]
>> lrgd = LogisticRegressionGD(eta=0.3,
...                                n_iter=1000,
...                                random_state=1)
> >  lrgd.fit(X_train_01_subset,
...            y_train_01_subset)
> > >  plot_decision_regions(X=X_train_01_subset,
...                           y=y_train_01_subset,
...                           sınıflandırıcı=lrgd)
>> plt.xlabel('Petal uzunluğu [standartlaştırılmış]')
>> plt.ylabel('Petal genişliği [standartlaştırılmış]')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

Ortaya çıkan karar bölgesi grafiği aşağıdaki gibi görünür:




*Şekil 3.5: Lojistik regresyon modeli için karar bölgesi grafiği*

 olimpikbarbell@gmail.com 

**Lojistik regresyon için gradyan inişli öğrenme algoritması**

Önceki koddaki LogisticRegressionGD ile *Bölüm 2'*deki AdalineGD kodunu karşılaştırdıysanız, ağırlık ve yanlılık güncelleme kurallarının değişmediğini fark etmiş olabilirsiniz (ölçekleme faktörü 2 hariç). Kalkülüs kullanarak, gradyan inişi yoluyla parametre güncellemelerinin lojistik regresyon ve Adaline için gerçekten de benzer olduğunu gösterebiliriz. Bununla birlikte, gradyan inişi öğrenme kuralının aşağıdaki türetiminin, lojistik regresyon için gradyan inişi öğrenme kuralının arkasındaki matematiksel kavramlarla ilgilenen okuyuculara yönelik olduğunu lütfen unutmayın. Bu bölümün geri kalanını takip etmek için gerekli değildir.

*Şekil 3.6,* log-likelihood fonksiyonunun *j.* ağırlığa göre kısmi türevini nasıl hesaplayabileceğimizi özetlemektedir:

$$\frac{\partial L}{\partial w_j} = \underbrace{\frac{\partial L}{\partial a}\frac{da}{dz}\frac{\partial z}{\partial w_j}}_{\text{Apply chain rule}} \quad \text{where } a = \sigma(z) = \frac{1}{1+e^{-z}}$$

1) Derive terms separately: 2) Combine via chain rule and simplify:

$$\frac{\partial L}{\partial a} = \frac{a-y}{a\cdot(1-a)}$$

$$\frac{da}{dz} = \frac{e^{-z}}{(1+e^{-z})^2} = a\cdot(1-a)$$

$$\frac{\partial z}{w_j} = x_j$$

$$\frac{\partial L}{\partial z} = a - y$$

$$\frac{\partial L}{\partial w_j} = (a-y)x_j = -(y-a)x_j$$

*Şekil 3.6: Log-likelihood fonksiyonunun kısmi türevinin hesaplanması*

Kısalık için eğitim örneklerinin ortalamasını almadığımızı unutmayın.

*Bölüm 2'*den gradyanın tersi yönünde adımlar attığımızı hatırlayın. Dolayısıyla, $\partial L$ = -($y$ - $a$)$x$'i çevririz ve j. ağırlığı aşağıdaki gibi güncelleriz

ing oranı $\eta$:

$$w_j := w_j + \eta(y - a)x_j$$

Kayıp fonksiyonunun önyargı birimine göre kısmi türevi gösterilmese de, önyargı türetme zincir kuralı kullanılarak aynı genel konsepti takip eder ve aşağıdaki güncelleme kuralı ile sonuçlanır:

$$b := b + \eta(y - a)$$

Hem ağırlık hem de önyargı birimi güncellemeleri *Bölüm 2*'deki Adaline güncellemelerine eşittir.

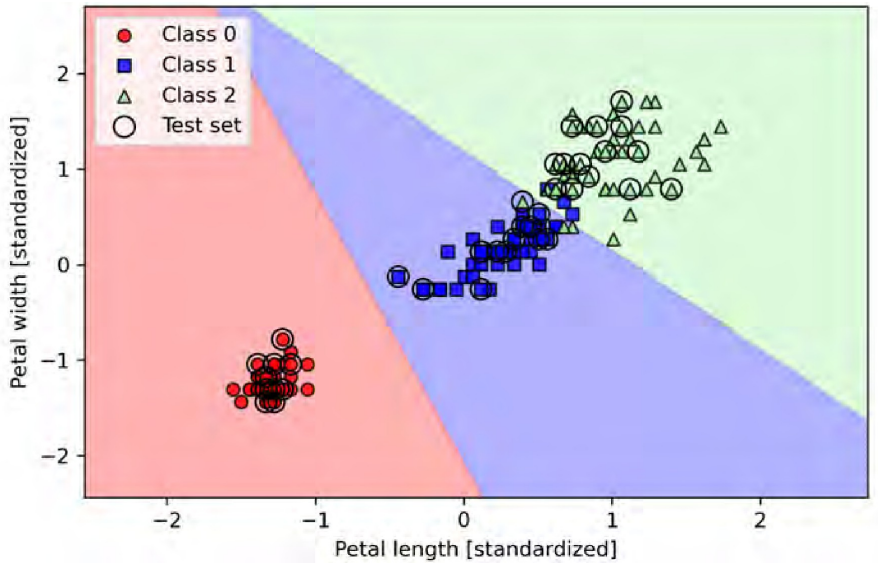
## Scikit-learn ile lojistik regresyon modeli eğitimi

Önceki alt bölümde, Adaline ve lojistik regresyon arasındaki kavramsal farklılıkları göstermeye yardımcı olan yararlı kodlama ve matematik alıştırmalarından geçtik. Şimdi, scikit-learn'in çok sınıflı ayarları da destekleyen lojistik regresyonun daha optimize edilmiş uygulamasını nasıl kullanacağımızı öğrenelim. Scikit-learn'in son sürümlerinde, çok sınıflı sınıflandırma, multinomial veya OvR için kullanılan tekniğin otomatik olarak seçildiğini unutmayın. Aşağıdaki kod örneğinde, sklearn.linear_model.LogisticRegression sınıfını ve modeli standartlaştırılmış çiçek eğitim veri kümesindeki üç sınıfın tümü üzerinde eğitmek için bilinen **fit** yöntemini kullanacağız. Ayrıca, örnekleme amacıyla multi_class='ovr' olarak ayarladık. Okuyucu için bir alıştırma olarak, sonuçları multi_class='multinomial' ile karşılaştırmak isteyebilirsiniz. Multinomial ayarının artık scikit- learn'in LogisticRegression sınıfında varsayılan seçim olduğunu ve uygulamada Iris veri setinde bulunanlar gibi birbirini dışlayan sınıflar için önerildiğini unutmayın. Burada "birbirini dışlayan", her eğitim örneğinin yalnızca tek bir sınıfa ait olabileceği anlamına gelir (bir eğitim örneğinin birden fazla sınıfın üyesi olabildiği çok etiketli sınıflandırmanın aksine).

Şimdi kod örneğine bir göz atalım:

```
>> from sklearn.linear_model import LogisticRegression
>> lr = LogisticRegression(C=100.0, solver='lbfgs',
...                         multi_class='ovr')
>> lr.fit(X_train_std, y_train)
> > >  plot_decision_regions(X_combined_std,
...                        y_combined,
...                        sınıflandırıcı=lr,
...                        test_idx=range(105, 150))
>> plt.xlabel('Petal uzunluğu [standartlaştırılmış]')
>> plt.ylabel('Petal genişliği [standartlaştırılmış]')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

Modeli eğitim verilerine uydurduktan sonra, *Şekil 3.7*'de gösterildiği gibi karar bölgelerini, eğitim örneklerini ve test örneklerini çizdik:




*Şekil 3.7: Scikit-learn'in çok sınıflı lojistik regresyon modeli için karar bölgeleri*

**Konveks optimizasyon için algoritmalar**

Optimizasyon problemlerini çözmek için birçok farklı algoritma olduğunu unutmayın. Lojistik regresyon kaybı gibi dışbükey kayıp fonksiyonlarını en aza indirmek için, normal **stokastik gradyan inişinden** (**SGD**) daha gelişmiş yaklaşımların kullanılması önerilir. Aslında scikit-learn, çözücü parametresi aracılığıyla belirtilebilen 'newton-cg', 'lbfgs', 'liblinear', 'sag' ve 'saga' gibi bir dizi optimizasyon algoritmasını uygular.

Lojistik regresyon kaybı dışbükey olsa da, çoğu optimizasyon algoritması küresel minimum kayba kolaylıkla yakınsamalıdır. Bununla birlikte, bir algoritmayı diğerine göre kullanmanın bazı avantajları vardır. Örneğin, önceki sürümlerde (örneğin, v 0.21), scikit-learn varsayılan olarak çok terimli kaybı işleyemeyen ve çok sınıflı sınıflandırma için OvR şemasıyla sınırlı olan 'liblinear' kullanıyordu. Ancak scikit-learn v 0.22'de varsayılan çözücü, sınırlı bellekli **Broyden-Fletcher-Goldfarb-Shanno** (**BFGS**) algoritması (https://en.wikipedia.org/ wiki/Limited-memory_BFGS) anlamına gelen ve bu konuda daha esnek olan 'lbfgs' olarak değiştirilmiştir.

LogisticRegression modelini eğitmek için kullandığımız önceki koda baktığınızda, şimdi "Bu gizemli C parametresi nedir?" diye merak ediyor olabilirsiniz. Bu parametreyi, aşırı uyum ve düzenlileştirme kavramlarını tanıtacağımız bir sonraki alt bölümde tartışacağız. Ancak, bu konulara geçmeden önce, sınıf üyeliği olasılıkları hakkındaki tartışmamızı bitirelim.

Eğitim örneklerinin belirli bir sınıfa ait olma olasılığı predict_ proba yöntemi kullanılarak hesaplanabilir. Örneğin, test veri kümesindeki ilk üç örneğin olasılıklarını aşağıdaki gibi tahmin edebiliriz:

```
>> lr.predict_proba(X_test_std[:3, :])
```

Bu kod parçacığı aşağıdaki diziyi döndürür:

```
array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
       [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
       [8.48831425e-01, 1 . 51168575e-01, 2.62277619e-14]])
```

İlk satır ilk çiçeğin sınıf üyeliği olasılıklarına, ikinci satır ikinci çiçeğin sınıf üyeliği olasılıklarına karşılık gelir ve bu böyle devam eder. Her satırdaki sütun bazında toplamın beklendiği gibi 1 olduğuna dikkat edin. (Bunu lr.predict_ proba(X_test_std[:3, : ]).sum(axis=1) komutunu çalıştırarak doğrulayabilirsiniz).

İlk satırdaki en yüksek değer yaklaşık 0,85'tir, bu da ilk örneğin yüzde 85'lik bir tahmin olasılığı ile sınıf 3'e (Iris-virginica) ait olduğu anlamına gelir. Dolayısıyla, daha önce fark etmiş olabileceğiniz gibi, NumPy'nin argmax fonksiyonunu kullanarak her satırdaki en büyük sütunu belirleyerek tahmin edilen sınıf etiketlerini elde edebiliriz:

```
>> lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

Döndürülen sınıf indeksleri burada gösterilmektedir (Iris-virginica, Iris-setosa ve Iris-setosa):

```
array([2, 0, 0])
```

Önceki kod örneğinde, koşullu olasılıkları hesapladık ve bunları NumPy'nin argmax işlevini kullanarak manuel olarak sınıf etiketlerine dönüştürdük. Pratikte, scikit-learn kullanırken sınıf etiketlerini elde etmenin daha uygun yolu doğrudan tahmin yöntemini çağırmaktır:

```
>> lr.predict(X_test_std[:3, :])
array([2, 0, 0])
```

Son olarak, tek bir çiçek örneğinin sınıf etiketini tahmin etmek istiyorsanız bir uyarı: scikit-learn veri girişi olarak iki boyutlu bir dizi bekler; bu nedenle, önce tek bir satır dilimini böyle bir biçime dönüştürmemiz gerekir. Tek bir satır girdisini iki boyutlu bir veri dizisine dönüştürmenin bir yolu, burada gösterildiği gibi yeni bir boyut eklemek için NumPy'nin reshape yöntemini kullanmaktır:

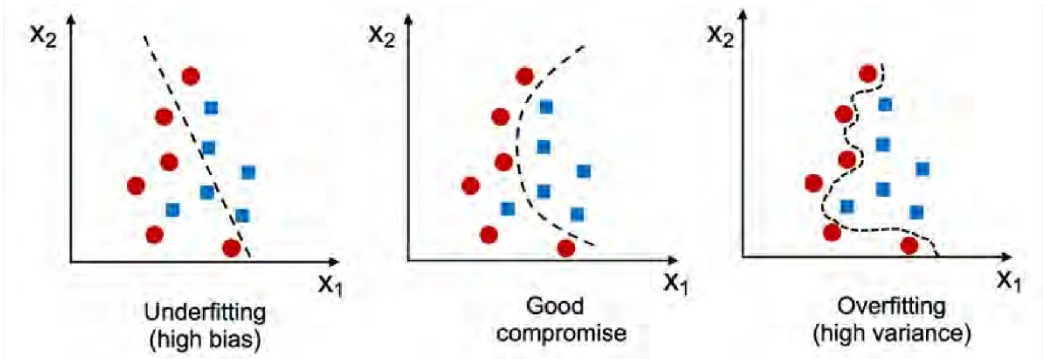
```
>> lr.predict(X_test_std[0, :].reshape(1, -1))
array([2])
```

 olimpikbarbell@gmail.com 

## Düzenli hale getirme yoluyla aşırı uyumla mücadele

Aşırı uyum, bir modelin eğitim verilerinde iyi performans gösterdiği ancak görülmeyen verilere (test verileri) iyi genelleme yapamadığı makine öğreniminde yaygın bir sorundur. Bir model aşırı uyumdan muzdaripse, modelin yüksek bir varyansa sahip olduğunu da söyleriz; bu, çok fazla parametreye sahip olmaktan kaynaklanabilir ve altta yatan veriler göz önüne alındığında çok karmaşık bir modele yol açabilir. Benzer şekilde, modelimiz **yetersiz uyumdan** (yüksek sapma) da muzdarip olabilir, bu da modelimizin eğitim verilerindeki örüntüyü iyi yakalamak için yeterince karmaşık olmadığı ve bu nedenle görünmeyen verilerde düşük performanstan muzdarip olduğu anlamına gelir.

Şimdiye kadar sınıflandırma için yalnızca doğrusal modellerle karşılaşmış olsak da, aşırı uyum ve yetersiz uyum sorunları, *Şekil 3.8'de* gösterildiği gibi doğrusal bir karar sınırını daha karmaşık, doğrusal olmayan karar sınırlarıyla karşılaştırarak en iyi şekilde gösterilebilir:




*Şekil 3.8: Yetersiz uyarlanmış, iyi uyarlanmış ve aşırı uyarlanmış model örnekleri*

**Sapma-varyans değiş tokuşu**

Araştırmacılar genellikle bir modelin performansını tanımlamak için "yanlılık" ve "varyans" ya da "yanlılık-varyans değiş tokuşu" terimlerini kullanırlar; yani, insanların bir modelin "yüksek varyansa" ya da "yüksek yanlılığa" sahip olduğunu söylediği konuşmalara, kitaplara ya da makalelere rastlayabilirsiniz. Peki, bu ne anlama geliyor? Genel olarak, "yüksek varyansın" aşırı uyumla, "yüksek yanlılığın" ise yetersiz uyumla orantılı olduğunu söyleyebiliriz.

Makine öğrenimi modelleri bağlamında varyans, örneğin eğitim veri kümesinin farklı alt kümeleri üzerinde modeli birden fazla kez yeniden eğittiğimizde, belirli bir örneği sınıflandırmak için model tahmininin tutarlılığını (veya değişkenliğini) ölçer. Modelin eğitim verilerindeki rastgeleliğe duyarlı olduğunu söyleyebiliriz. Buna karşılık önyargı, modeli farklı eğitim veri kümeleri üzerinde birden çok kez yeniden oluşturduğumuzda tahminlerin genel olarak doğru değerlerden ne kadar uzak olduğunu ölçer; önyargı, rastgelelikten kaynaklanmayan sistematik hatanın ölçüsüdür.

"Bias" ve "variance" terimlerinin teknik özellikleri ve türetilmesiyle ilgileniyorsanız, bunu ders notlarımda yazmıştım: https://sebastianraschka.com/ pdf/lecture-notes/stat451fs20/08-model-eval-1-intro notes.pdf.

6587191CA584E

İyi bir sapma-varyans ödünleşimi bulmanın bir yolu, modelin karmaşıklığını düzenli hale getirme yoluyla ayarlamaktır. Düzenli hale getirme, eş doğrusallığı (özellikler arasında yüksek korelasyon) ele almak, verilerdeki gürültüyü filtrelemek ve nihayetinde aşırı uyumu önlemek için çok yararlı bir yöntemdir.

Düzenli hale getirmenin arkasındaki kavram, aşırı parametre (ağırlık) değerlerini cezalandırmak için ek bilgi sağlamaktır. **Düzenli hale** getirmenin en yaygın biçimi, aşağıdaki gibi yazılabilen **L2 düzenli hale** getirmedir (bazen L2 küçültme veya ağırlık azaltma olarak da adlandırılır):

$$\frac{\lambda}{2n}\|\boldsymbol{w}\|^2 = \frac{\lambda}{2n}\sum_{j=1}^{m} w_j^2$$

Burada, $\lambda$ **düzenlileştirme parametresi** olarak adlandırılır. Paydadaki 2'nin yalnızca bir ölçeklendirme faktörü olduğuna dikkat edin, öyle ki kayıp gradyanı hesaplanırken iptal olur. Örneklem büyüklüğü *n, düzenlileştirme* terimini kayba benzer şekilde ölçeklendirmek için eklenir.

**Düzenli hale getirme ve özellik normalizasyonu**

Düzenli hale getirme, standartlaştırma gibi özellik ölçeklendirmenin önemli olmasının bir başka nedenidir. Düzenlemenin düzgün çalışması için, tüm özelliklerimizin karşılaştırılabilir ölçeklerde olduğundan emin olmamız gerekir.

Lojistik regresyon için kayıp fonksiyonu, model eğitimi sırasında ağırlıkları küçültecek basit bir düzenleme terimi eklenerek düzenlenebilir:

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left[-y^{(i)}\log(\sigma(z^{(i)})) - (1 - y^{(i)})\log(1 - \sigma(z^{(i)}))\right] + \frac{\lambda}{2n}\|\boldsymbol{w}\|^2$$

Düzenlenmemiş kaybın kısmi türevi şu şekilde tanımlanır:

$$\frac{\partial L(\boldsymbol{w}, b)}{\partial w_j} = \left(\frac{1}{n}\sum_{i=1}^{n}(\sigma(\boldsymbol{w}^T\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)}\right)$$

Düzenli hale getirme teriminin kayba eklenmesi kısmi türevi aşağıdaki forma dönüştürür:

$$\frac{\partial L(\boldsymbol{w}, b)}{\partial w_j} = \left(\frac{1}{n}\sum_{i=1}^{n}(\sigma(\boldsymbol{w}^T\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{n}w_j$$

 olimpikbarbell@gmail.com 

Düzenlilik parametresi $\lambda$ aracılığıyla, ağırlıkları küçük tutarken eğitim verilerine ne kadar yakından uyduğumuzu kontrol edebiliriz. $\lambda$ değerini artırarak, düzenlileştirme gücünü artırırız. *Bölüm 2'*de öğrendiğimiz gibi, esasen bir kesişme terimi veya negatif eşik olan bias biriminin genellikle düzenli hale getirilmediğini lütfen unutmayın.

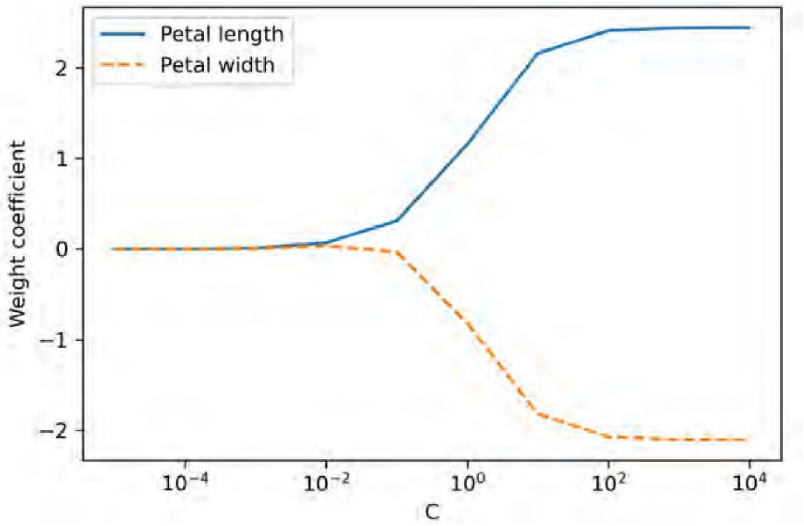
Scikit-learn'de `LogisticRegression` sınıfı için uygulanan `C` parametresi, bir sonraki bölümün konusu olacak olan destek vektör makinelerindeki bir kuraldan gelmektedir. `C` terimi, düzenlileştirme parametresi olan $\lambda$ ile ters orantılıdır. Sonuç olarak, ters düzenlileştirme parametresi `C`'nin değerini azaltmak, iki ağırlık katsayısı için L2 düzenlileştirme yolunu çizerek

```
>>> weights, params = [], []
>> for c in np.arange(-5, 5):
...         lr = LogisticRegression(C=10.**c,
...                                 multi_class='ovr')
...         lr.fit(X_train_std, y_train)
...         weights.append(lr.coef_[1])
...         params.append(10.**c)
>>> ağırlıklar = np.array(ağırlıklar)
>> plt.plot(params, weights[:, 0],
...                 label='Petal length')
>> plt.plot(params, weights[:, 1], linestyle='--',
...                 label='Petal width')
>> plt.ylabel('Ağırlık katsayısı')
>> plt.xlabel('C')
>> plt.legend(loc='sol üst')
>> plt.xscale('log')
>> plt.show()
```

görselleştirebileceğimiz düzenlileştirme gücünü artırdığımız anlamına gelir:

Önceki kodu çalıştırarak, ters-düzenleme parametresi `C` için farklı değerlere sahip 10 lojistik regresyon modeli kurduk. Örnekleme amacıyla, tüm sınıflandırıcılara karşı yalnızca sınıf `1`'in (burada, veri kümesindeki ikinci sınıf: `Iris-versicolor`) ağırlık katsayılarını topladık - çok sınıflı sınıflandırma için OvR tekniğini kullandığımızı unutmayın.

Ortaya çıkan grafikte görebileceğimiz gibi, C parametresini azaltırsak, yani düzenlileştirme gücünü artırırsak ağırlık katsayıları küçülür:




*Şekil 3.9: Ters düzenlileştirme gücü parametresi C'nin L2 düzenlileştirilmiş model sonuçları üzerindeki etkisi*

Düzenli hale getirme gücünü artırmak aşırı uyumu azaltabilir, bu nedenle neden varsayılan olarak tüm modelleri güçlü bir şekilde düzenli hale getirmediğimizi sorabiliriz. Bunun nedeni, düzenlileştirme gücünü ayarlarken dikkatli olmamız gerektiğidir. Örneğin, düzenlileştirme gücü çok yüksekse ve ağırlık katsayıları sıfıra yaklaşırsa, *Şekil 3.8*'de gösterildiği gibi, model yetersiz uyum nedeniyle çok kötü performans gösterebilir.

**Lojistik regresyon üzerine ek bir kaynak**

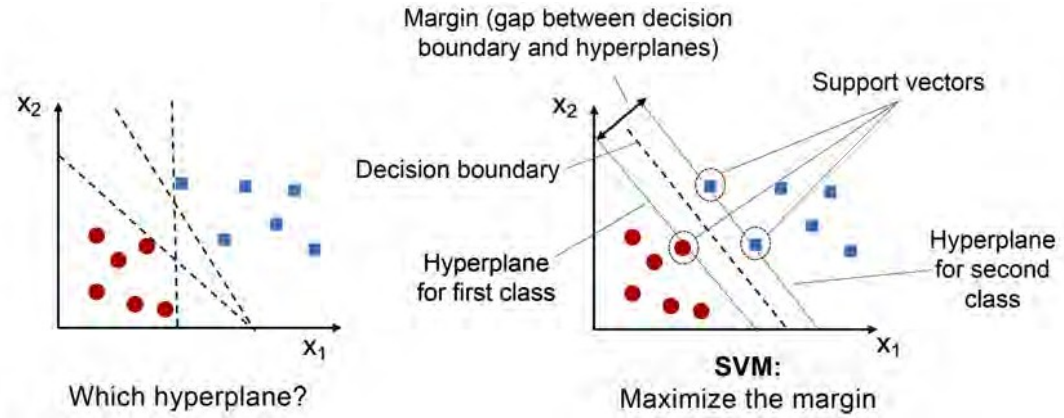
Her bir sınıflandırma algoritmasının derinlemesine ele alınması bu kitabın kapsamını aştığından, *Logistic Regression: From Introductory to Advanced Concepts and Applications*, *Dr. Scott Menard*, *Sage Publications, 2009*, lojistik regresyon hakkında daha fazla bilgi edinmek isteyen okuyuculara tavsiye edilir.

# Destek vektörü ile maksimum marjlı sınıflandırma makineleri

Bir diğer güçlü ve yaygın olarak kullanılan öğrenme algoritması ise perceptronun bir uzantısı olarak düşünülebilecek **destek vektör makinesidir** (**SVM**). Perceptron algoritmasını kullanarak yanlış sınıflandırma hatalarını en aza indirdik. Ancak DVM'lerde optimizasyon amacımız marjini maksimize etmektir. Marjin, ayırıcı hiper düzlem (karar sınırı) ile bu hiper düzleme en yakın olan ve **destek vektörleri olarak** adlandırılan eğitim örnekleri arasındaki mesafe olarak tanımlanır.

 olimpikbarbell@gmail.com 

Bu durum *Şekil 3.10*'da gösterilmektedir:




*Şekil 3.10: DVM, karar sınırı ile eğitim veri noktaları arasındaki marjı maksimize eder*

## Maksimum marj sezgisi

Büyük marjlara sahip karar sınırlarının arkasındaki mantık, daha düşük genelleme hatasına sahip olma eğiliminde olmalarıdır, oysa küçük marjlara sahip modeller aşırı uyuma daha yatkındır.

Ne yazık ki, DVM'lerin arkasındaki ana sezgi nispeten basit olsa da, arkasındaki matematik oldukça ileri düzeydedir ve kısıtlı optimizasyon konusunda sağlam bilgi gerektirir.

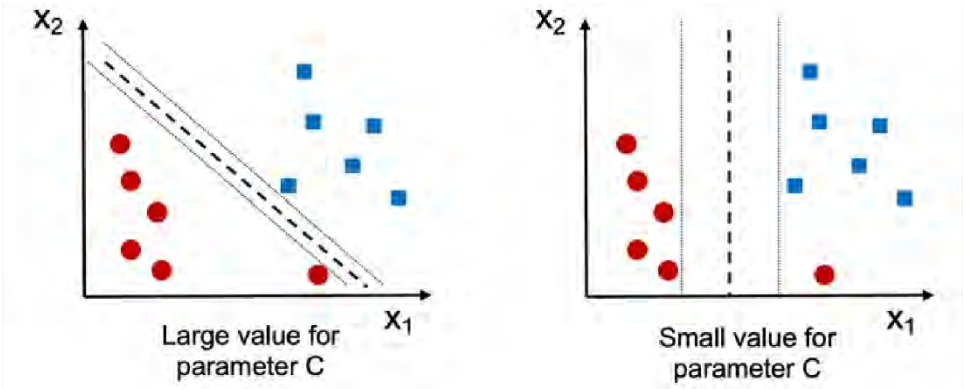
Bu nedenle, DVM'lerde maksimum marj optimizasyonunun arkasındaki ayrıntılar bu kitabın kapsamı dışındadır. Ancak, daha fazla bilgi edinmek istiyorsanız aşağıdaki kaynakları tavsiye ederiz:

- Chris J.C. Burges'in *A Tutorial on Support Vector Machines for Pattern Recognition* (Data Mining and Knowledge Discovery, 2(2): 121-167, 1998)
- Vladimir Vapnik'in kitabı *The Nature of Statistical Learning Theory*, Springer Science+Business Media, Vladimir Vapnik, 2000
- Andrew Ng'nin çok detaylı ders notlarına https://see.stanford.edu/materials/aimlcs229/cs229-notes3.pdf adresinden ulaşabilirsiniz.

## Doğrusal olmayan ayrılabilir bir durumla gevşek değişkenler kullanarak başa çıkma

Maksimum marjlı sınıflandırmanın arkasındaki daha karmaşık matematiksel kavramlara daha fazla dalmak istemesek de, 1995 yılında Vladimir Vapnik tarafından ortaya atılan ve **yumuşak marjlı sınıflandırmaya** yol açan sözde *gevşek değişkenden* kısaca bahsedelim. Gevşek değişkeni tanıtmanın motivasyonu, SVM optimizasyon hedefindeki doğrusal kısıtlamaların, uygun kayıp cezalandırması altında yanlış sınıflandırmaların varlığında optimizasyonun yakınsamasına izin vermek için doğrusal olmayan şekilde ayrılabilir veriler için gevşetilmesi gerektiğiydi.

Slack değişkeninin kullanımı, SVM bağlamlarında genellikle *C* olarak adlandırılan değişkeni ortaya çıkarır. *C'yi* yanlış sınıflandırma cezasını kontrol etmek için bir hiper parametre olarak düşünebiliriz. *C'*nin büyük değerleri büyük hata cezalarına karşılık gelirken, *C* için daha küçük değerler seçersek yanlış sınıflandırma hataları konusunda daha az katı oluruz. Daha sonra marjın genişliğini kontrol etmek ve dolayısıyla *Şekil 3.11'*de gösterildiği gibi yanlılık-varyans dengesini ayarlamak için *C* parametresini kullanabiliriz:




*Şekil 3.11: Ters düzenlileştirme gücü C'nin büyük ve küçük değerlerinin sınıflandırma üzerindeki etkisi*

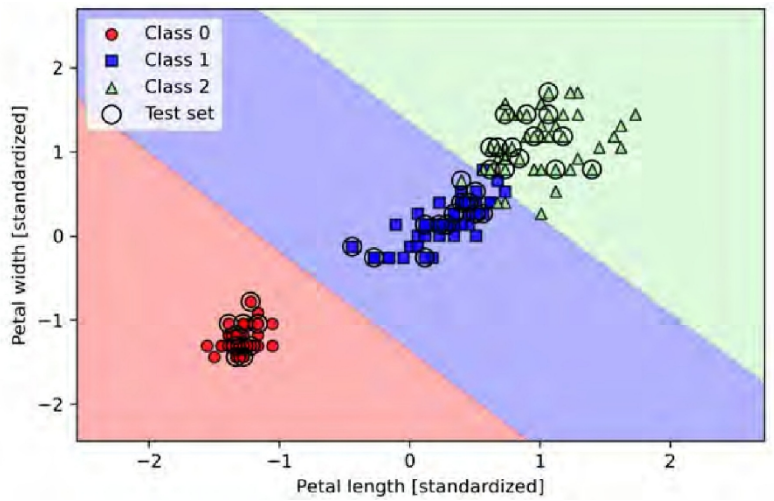
Bu kavram, önceki bölümde düzenli regresyon bağlamında tartıştığımız, C değerinin azaltılmasının modelin önyargısını artırdığı (yetersiz uyum) ve varyansını düşürdüğü (aşırı uyum) düzenli hale getirme ile ilgilidir.

Doğrusal SVM'nin arkasındaki temel kavramları öğrendiğimize göre, Iris veri kümemizdeki farklı çiçekleri sınıflandırmak için bir SVM modeli eğitelim:

```
>> from sklearn.svm import SVC
>> svm = SVC(kernel='linear', C=1.0, random_state=1)
>> svm.fit(X_train_std, y_train)
> > >   plot_decision_regions(X_combined_std,
...                             y_combined,
...                             sınıflandırıcı=svm,
...                             test_idx=range(105, 150))
>> plt.xlabel('Petal uzunluğu [standartlaştırılmış]')
>> plt.ylabel('Petal genişliği [standartlaştırılmış]')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

 olimpikbarbell@gmail.com 

Bir önceki kod örneğini çalıştırarak Iris veri kümesi üzerinde sınıflandırıcıyı eğittikten sonra görselleştirilen DVM'nin üç karar bölgesi *Şekil 3.12'de* gösterilmektedir:




*Şekil 3.12: DVM'nin karar bölgeleri*

**SVM'lere karşı lojistik regresyon**

Pratik sınıflandırma görevlerinde, doğrusal lojistik regresyon ve doğrusal DVM'ler genellikle çok benzer sonuçlar verir. Lojistik regresyon, eğitim verilerinin koşullu olasılıklarını en üst düzeye çıkarmaya çalışır, bu da onu çoğunlukla karar sınırına en yakın noktalarla (destek vektörleri) ilgilenen DVM'lere göre aykırı değerlere daha yatkın hale getirir. Öte yandan, lojistik regresyon daha basit bir model olma avantajına sahiptir ve daha kolay uygulanabilir ve matematiksel olarak açıklanması daha kolaydır. Ayrıca, lojistik regresyon modelleri kolayca güncellenebilir, bu da akış verileriyle çalışırken caziptir.

## Scikit-learn'de alternatif uygulamalar

Önceki bölümlerde kullandığımız scikit-learn kütüphanesinin `LogisticRegression` sınıfı, `solver='liblinear'` olarak ayarlanarak LIBLINEAR kütüphanesinden yararlanabilir. LIBLINEAR, Ulusal Tayvan Üniversitesi'nde geliştirilen yüksek düzeyde optimize edilmiş bir C/C++ kütüphanesidir (http://www.csie.ntu.edu.tw/~cjlin/ liblinear/).

Benzer şekilde, bir DVM'yi eğitmek için kullandığımız `SVC` sınıfı, DVM'ler için özelleştirilmiş eşdeğer bir C/C++ kütüphanesi olan LIBSVM'yi kullanır (http://www.csie.ntu.edu.tw/~cjlin/libsvm/).

LIBLINEAR ve LIBSVM kullanmanın, örneğin yerel Python uygulamalarına göre avantajı, büyük miktarlarda doğrusal sınıflandırıcıların son derece hızlı bir şekilde eğitilmesine izin vermeleridir. Ancak, bazen veri kümelerimiz bilgisayar belleğine sığmayacak kadar büyüktür. Bu nedenle scikit-learn, partial_fit yöntemiyle çevrimiçi öğrenmeyi de destekleyen SGDClassifier sınıfı aracılığıyla alternatif uygulamalar da sunar. SGDClassifier sınıfının arkasındaki konsept, *Bölüm 2*'de Adaline için uyguladığımız stokastik gradyan algoritmasına benzer.

Algılayıcının SGD versiyonunu (loss='perceptron'), lojistik regresyonu (loss='log') ve varsayılan parametrelere sahip bir SVM'yi (loss='hinge') aşağıdaki gibi başlatabiliriz:

```
>> from sklearn.linear_model import SGDClassifier
>>  ppn = SGDClassifier(loss='perceptron')
>> lr = SGDClassifier(loss='log')
>>  svm = SGDClassifier(loss='hinge')
```

# Bir çekirdek kullanarak doğrusal olmayan problemleri çözme DVM

DVM'lerin makine öğrenimi uygulayıcıları arasında yüksek popülerliğe sahip olmasının bir başka nedeni de doğrusal olmayan sınıflandırma problemlerini çözmek için kolayca **çekirdekleştirilebilmeleridir. DVM'**lerin en yaygın çeşidi olan **çekirdek DVM'**nin arkasındaki ana fikri tartışmadan önce, böyle bir doğrusal olmayan sınıflandırma probleminin neye benzeyebileceğini görmek için ilk olarak sentetik bir veri kümesi oluşturalım.

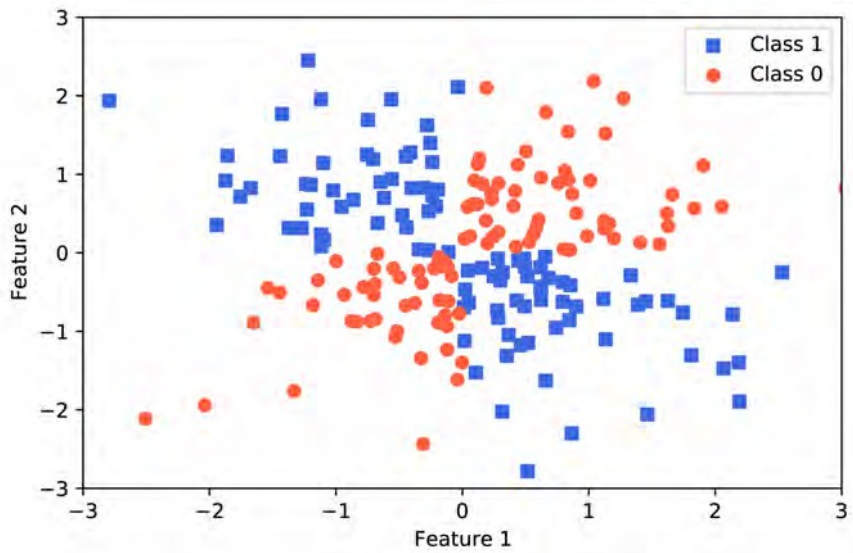
## Doğrusal olarak ayrıştırılamayan veriler için çekirdek yöntemleri

Aşağıdaki kodu kullanarak, NumPy'nin logical_xor işlevini kullanarak bir XOR kapısı biçiminde basit bir veri kümesi oluşturacağız; burada 100 örneğe 1 sınıf etiketi ve 100 örneğe -1 sınıf etiketi atanacaktır:

```
>> import matplotlib.pyplot as plt
>> import numpy as np
>> np.random.seed(1)
>> X_xor = np.random.randn(200, 2)
>> y_xor = np.logical_xor(X_xor[:, 0] > 0,
...                        X_xor[:, 1] > 0)
>> y_xor = np.where(y_xor, 1, 0)
>> plt.scatter(X_xor[y_xor == 1, 0],
...            X_xor[y_xor == 1, 1],
...            c='royalblue', marker='s',
...            label='Sınıf 1')
```

```
>> plt.scatter(X_xor[y_xor == 0, 0],
...            X_xor[y_xor == 0, 1],
...            c='domates', marker='o',
...            label='Sınıf 0')
>> plt.xlim([-3, 3])
>> plt.ylim([-3, 3])
>> plt.xlabel('Özellik 1')
>> plt.ylabel('Özellik 2')
>> plt.legend(loc='best')
>> plt.tight_layout()
>> plt.show()
```

Kodu çalıştırdıktan sonra, *Şekil 3.13*'te gösterildiği gibi rastgele gürültü içeren bir XOR veri kümesine sahip olacağız:



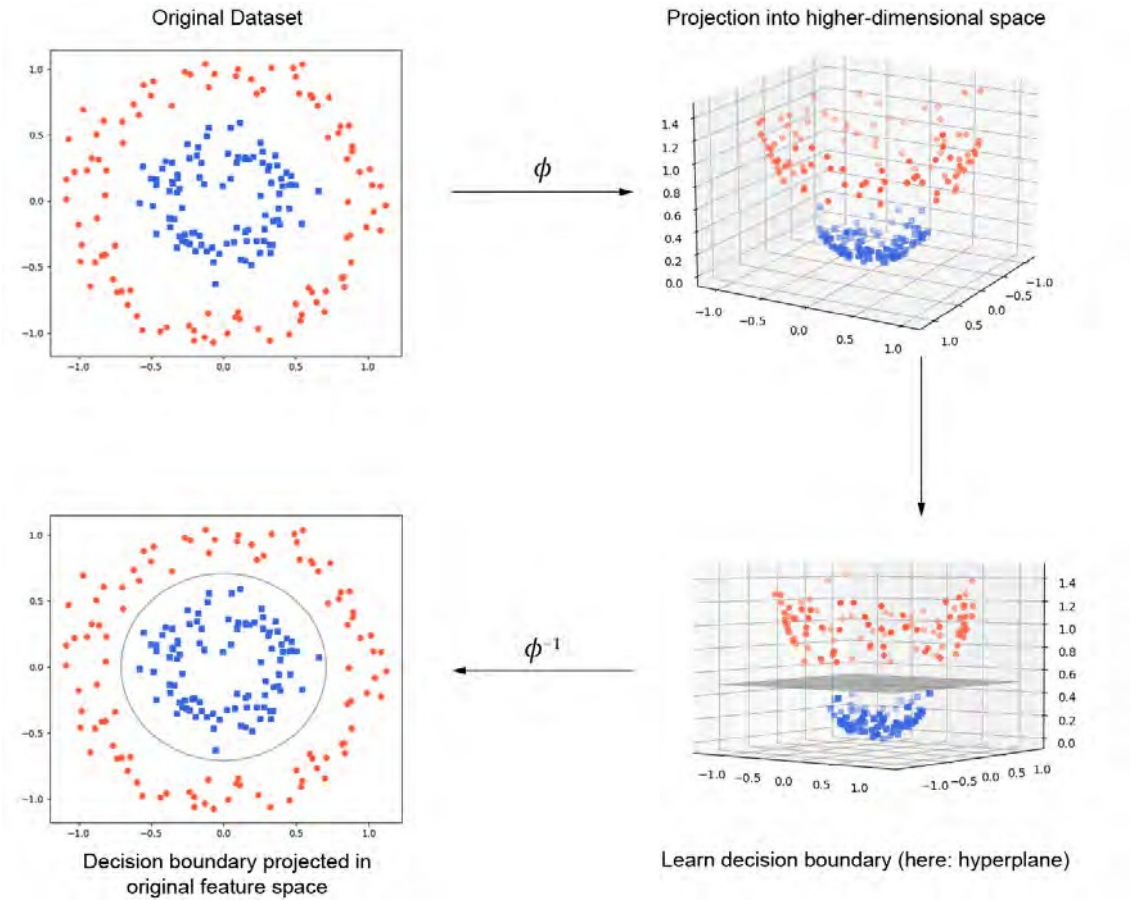

*Şekil 3.13: XOR veri kümesinin bir grafiği*

Açıkçası, daha önceki bölümlerde tartıştığımız doğrusal lojistik regresyon veya doğrusal SVM modeli aracılığıyla karar sınırı olarak doğrusal bir hiper düzlem kullanarak örnekleri pozitif ve negatif sınıftan çok iyi ayıramayız.

Bu tür doğrusal olarak ayrılamayan verilerle başa çıkmak için **çekirdek yöntemlerinin** arkasındaki temel fikir, orijinal özelliklerin doğrusal olmayan kombinasyonlarını oluşturmak ve bunları, verilerin doğrusal olarak ayrılabilir hale geldiği bir eşleme fonksiyonu ($\phi$) aracılığıyla daha yüksek boyutlu bir uzaya yansıtmaktır. *Şekil 3.14*'te gösterildiği gibi, iki boyutlu bir veri kümesini, sınıfların aşağıdaki projeksiyon yoluyla ayrılabilir hale geldiği yeni bir üç boyutlu özellik uzayına dönüştürebiliriz:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

Bu, aşağıdaki eşmerkezli daire veri kümesinde gösterildiği gibi, orijinal özellik uzayına geri yansıttığımızda doğrusal olmayan bir karar sınırı haline gelen doğrusal bir hiper düzlem aracılığıyla çizimde gösterilen iki sınıfı ayırmamızı sağlar:




*Şekil 3.14: Doğrusal olmayan verilerin kernel yöntemleri kullanılarak sınıflandırılması süreci*

## Yüksek boyutlu bir uzayda ayırıcı hiper düzlemleri bulmak için çekirdek hilesini kullanma

Doğrusal olmayan bir problemi DVM kullanarak çözmek için, eğitim verilerini bir eşleme fonksiyonu ($\phi$) aracılığıyla daha yüksek boyutlu bir özellik uzayına dönüştürür ve verileri bu yeni özellik uzayında sınıflandırmak için doğrusal bir DVM modeli eğitiriz. Daha sonra, aynı eşleme fonksiyonunu, $\phi$, yeni, görülmemiş verileri doğrusal DVM modelini kullanarak sınıflandırmak üzere dönüştürmek için kullanabiliriz.

Bununla birlikte, bu eşleme yaklaşımıyla ilgili bir sorun, özellikle yüksek boyutlu verilerle uğraşıyorsak, yeni özelliklerin oluşturulmasının hesaplama açısından çok pahalı olmasıdır. İşte bu noktada sözde **çekirdek hilesi** devreye girmektedir.

Bir DVM'yi eğitmek için ikinci dereceden programlama görevinin nasıl çözüleceği konusunda fazla ayrıntıya girmemiş olsak da, pratikte $\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}$ nokta çarpımını $\phi(\boldsymbol{x}^{(i)})^T\phi(\boldsymbol{x}^{(j)})$ ile değiştirmemiz yeterlidir. İki nokta arasındaki bu nokta çarpımını açıkça hesaplamak gibi pahalı bir adımdan tasarruf etmek için, **çekirdek fonksiyonu olarak** adlandırılan bir **fonksiyon** tanımlıyoruz:

$$\kappa(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \phi(\boldsymbol{x}^{(i)})^T\phi(\boldsymbol{x}^{(j)})$$

En yaygın kullanılan çekirdeklerden biri, basitçe **Gauss** çekirdeği olarak adlandırılabilecek **radyal temel fonksiyon** (**RBF**) çekirdeğidir:

$$\kappa(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \exp\left(-\frac{\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|^2}{2\sigma^2}\right)$$

Bu genellikle şu şekilde basitleştirilir:

$$\kappa(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \exp\left(-\gamma\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|^2\right)$$

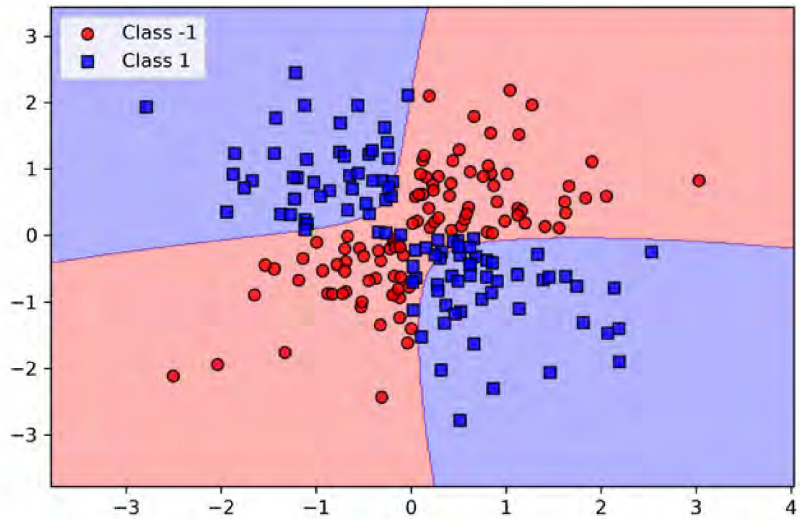
İşte, $\gamma = \frac{1}{2\sigma^2}$ optimize edilecek serbest bir parametredir.

Kabaca konuşmak gerekirse, "kernel" terimi bir çift örnek arasındaki benzerlik **fonksiyonu** olarak yorumlanabilir. Eksi işareti, mesafe ölçüsünü bir benzerlik puanına dönüştürür ve üstel terim nedeniyle, ortaya çıkan benzerlik puanı 1 (tam olarak benzer örnekler için) ile 0 (çok farklı örnekler için) arasında bir aralığa düşecektir.

Çekirdek numarasının arkasındaki büyük resmi ele aldığımıza göre, XOR verilerini iyi bir şekilde ayıran doğrusal olmayan bir karar sınırı çizebilen bir çekirdek SVM'yi eğitip eğitemeyeceğimizi görelim. Burada, scikit-learn'den daha önce içe aktardığımız SVC sınıfını kullanacağız ve kernel='linear' parametresini kernel='rbf' ile değiştireceğiz:

```
>> svm = SVC(kernel='rbf', random_state=1, gamma=0.10, C=10.0)
>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

Ortaya çıkan grafikte de görebileceğimiz gibi, çekirdek DVM XOR verilerini nispeten iyi bir şekilde ayırmaktadır:




*Şekil 3.15: Kernel yöntemi kullanılarak XOR verileri üzerinde karar sınırı*

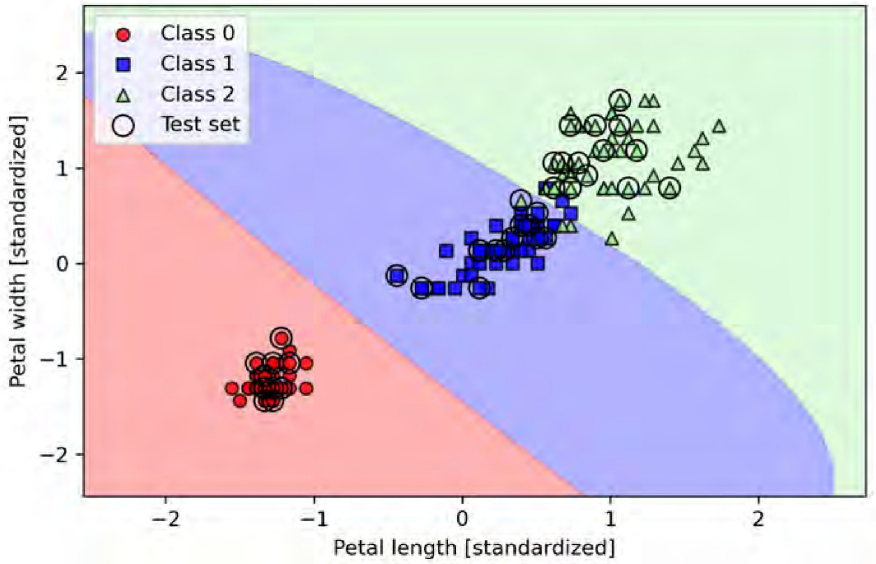
Gamma=0.1 olarak ayarladığımız $\gamma\gamma$ parametresi, Gauss küresi için bir kesme parametresi olarak anlaşılabilir. Eğer $\gamma\gamma$ değerini artırırsak, eğitim örneklerinin etkisini veya erişimini artırırız, bu da daha sıkı ve daha inişli çıkışlı bir karar sınırına yol açar. $\gamma\gamma$ değerini daha iyi anlamak için, Iris çiçek

```
>> svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
>> svm.fit(X_train_std, y_train)
> > >   plot_decision_regions(X_combined_std,
...                              y_combined, sınıflandırıcı=svm,
...                              test_idx=range(105, 150))
>> plt.xlabel('Petal uzunluğu [standartlaştırılmış]')
>> plt.ylabel('Petal genişliği [standartlaştırılmış]')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

veri kümemize bir RBF çekirdek SVM uygulayalım:

 olimpikbarbell@gmail.com 

$\gamma\gamma$ için nispeten küçük bir değer seçtiğimizden, RBF çekirdek DVM modelinin ortaya çıkan karar sınırı *Şekil 3.16'da* gösterildiği gibi nispeten yumuşak olacaktır:



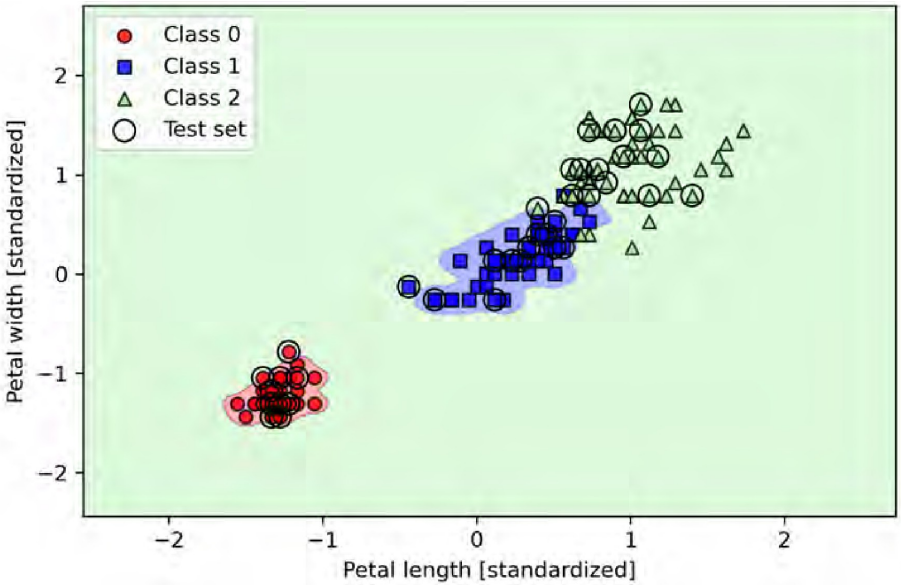

*Şekil 3.16: Küçük bir RBF çekirdekli DVM modeli kullanılarak Iris veri kümesi üzerindeki karar sınırları $\gamma\gamma$ değeri*

Şimdi, $\gamma\gamma$ değerini artıralım ve karar sınırı üzerindeki etkisini gözlemleyelim:

```
>> svm = SVC(kernel='rbf', random_state=1, gamma=100.0, C=1.0)
>> svm.fit(X_train_std, y_train)
> > >   plot_decision_regions(X_combined_std,
...                              y_combined, sınıflandırıcı=svm,
...                              test_idx=range(105,150))
>> plt.xlabel('Petal uzunluğu [standartlaştırılmış]')
>> plt.ylabel('Petal genişliği [standartlaştırılmış]')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

6587191CA584E

*Şekil 3.17*'de, 0 ve 1 sınıfları etrafındaki karar sınırının nispeten büyük bir $\gamma\gamma$ değeri kullanılarak çok daha sıkı olduğunu görebiliriz:




*Şekil 3.17: Büyük bir RBF çekirdekli DVM modeli kullanılarak Iris veri kümesi üzerindeki karar sınırları $\gamma\gamma$ değeri*

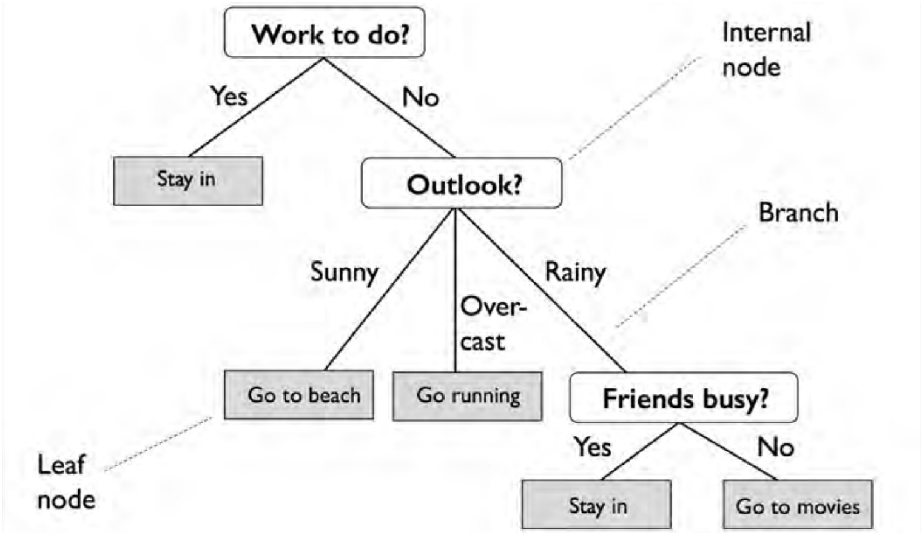
Model eğitim veri setine çok iyi uysa da, böyle bir sınıflandırıcı muhtemelen görünmeyen veriler üzerinde yüksek bir genelleme hatasına sahip olacaktır. Bu da $\gamma\gamma$ parametresinin, algoritma eğitim veri kümesindeki dalgalanmalara karşı çok hassas olduğunda aşırı uyum veya varyansı kontrol etmede önemli bir rol oynadığını göstermektedir.

# Karar ağacı öğrenme

Yorumlanabilirliği önemsiyorsak **karar ağacı** sınıflandırıcıları cazip modellerdir. "Karar ağacı" adından da anlaşılacağı gibi, bu modeli bir dizi soru sorarak bir karar vererek verilerimizi parçalara ayırmak olarak düşünebiliriz.

 olimpikbarbell@gmail.com 

Belirli bir günde bir faaliyete karar vermek için bir karar ağacı kullandığımız aşağıdaki örneği ele alalım:




*Şekil 3.18: Bir karar ağacı örneği*

Eğitim veri kümemizdeki özelliklere dayanarak, karar ağacı modeli örneklerin sınıf etiketlerini çıkarmak için bir dizi soru öğrenir. *Şekil 3.18* kategorik değişkenlere dayalı bir karar ağacı kavramını gösterse de, aynı kavram Iris veri kümesinde olduğu gibi özelliklerimiz gerçek sayılar olduğunda da geçerlidir. Örneğin, **sepal genişlik** özelliği ekseni boyunca bir kesme değeri tanımlayabilir ve ikili bir soru sorabiliriz: "Sepal genişlik ≥ 2,8 cm mi?"

Karar algoritmasını kullanarak, ağaç kökünden başlarız ve verileri, bir sonraki bölümde daha ayrıntılı olarak açıklanacak olan en büyük **bilgi kazancı** (**IG**) ile sonuçlanan özellik üzerinde böleriz. Yinelemeli bir süreçte, bu bölme prosedürünü yapraklar saf olana kadar her bir çocuk düğümde tekrarlayabiliriz. Bu, her düğümdeki eğitim örneklerinin hepsinin aynı sınıfa ait olduğu anlamına gelir. Pratikte bu, çok sayıda düğüme sahip çok derin bir ağaçla sonuçlanabilir ve bu da kolayca aşırı uyuma yol açabilir. Bu nedenle, genellikle ağacın maksimum derinliği için bir sınır belirleyerek ağacı **budamak** isteriz.

## IG'yi maksimize etmek - paranızın karşılığını en iyi şekilde almak

Düğümleri en bilgilendirici özelliklere bölmek için, ağaç öğrenme algoritması aracılığıyla optimize edilecek bir amaç fonksiyonu tanımlamamız gerekir. Burada amaç fonksiyonumuz, aşağıdaki gibi tanımladığımız her bir bölünmede IG'yi maksimize etmektir:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

Burada, $f$ bölme işlemini gerçekleştirecek özelliktir; $D_p$ ve $D_j$ *ana ve j.* çocuk düğümün veri kümesidir; $I$ **safsızlık** ölçütümüzdür; $N_p$ ana düğümdeki toplam eğitim örneği sayısıdır; ve $N_j$ j. çocuk düğümdeki örnek sayısıdır. Gördüğümüz gibi, bilgi kazancı basitçe ana düğümün safsızlığı ile çocuk düğüm safsızlıklarının toplamı arasındaki farktır - çocuk düğümlerin safsızlığı ne kadar düşükse, bilgi kazancı o kadar büyüktür. Bununla birlikte, basitlik ve kombinatoryal arama alanını azaltmak için çoğu kütüphane (scikit-learn dahil) ikili karar ağaçları uygular. Bu, her ana düğümün $D_{left}$ ve $D_{right}$ olmak üzere iki çocuk düğüme bölündüğü anlamına gelir:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

İkili karar ağaçlarında yaygın olarak kullanılan üç **safsızlık** ölçütü veya bölme kriteri **Gini safsızlığı** ($I_G$ **), entropi** ($I_H$ ) ve **sınıflandırma hatasıdır** ($I_E$ **). Boş olmayan** tüm sınıflar için entropi tanımı ile başlayalım ($p(i|t) \neq 0$):

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t)$$

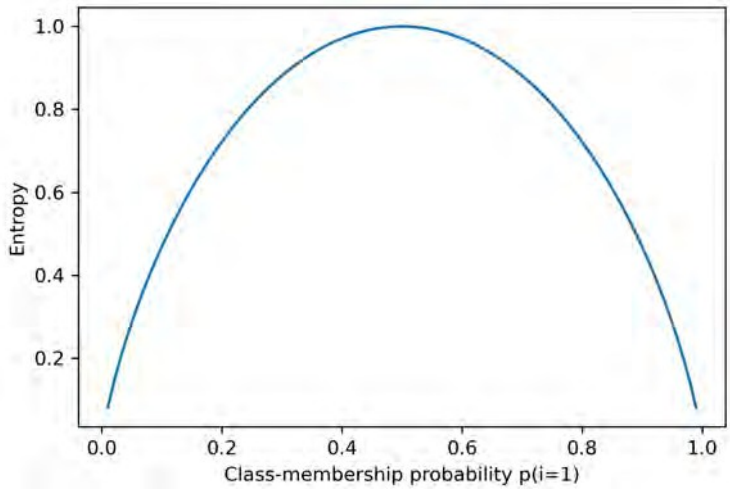
Burada, $p(i|t)$ belirli bir düğüm, $t$ için $i$ sınıfına ait örneklerin oranıdır. Bu nedenle, bir düğümdeki tüm örnekler aynı sınıfa aitse entropi 0'dır ve tekdüze bir sınıf dağılımımız varsa entropi maksimumdur. Örneğin, ikili sınıf ortamında, $p(i=1|t) = 1$ veya $p(i=0|t) = 0$ ise entropi 0'dır. $p(i=1|t) = 0{,}5$ ve $p(i=0|t) = 0{,}5$ olacak şekilde sınıflar eşit olarak dağıtılırsa entropi 1 olur. Bu nedenle, entropi kriterinin ağaçtaki karşılıklı bilgiyi maksimize etmeye çalıştığını söyleyebiliriz.

Görsel bir sezgi sağlamak için, aşağıdaki kod aracılığıyla farklı sınıf dağılımları için entropi değerlerini görselleştirelim:

```
>> def entropi(p):
...     return - p * np.log2(p) - (1 - p) * np.log2((1 - p ) )
>> x = np.arange(0.0, 1.0, 0 . 01)
>> ent = [entropi(p) if p != 0 else None for p in x]
>>  plt.ylabel('Entropi')
>> plt.xlabel('Sınıf-üyelik olasılığı p(i=1)')
>> plt.plot(x, ent)
>> plt.show()
```

 olimpikbarbell@gmail.com 

Aşağıdaki *Şekil 3.19,* önceki kod tarafından üretilen grafiği göstermektedir:




*Şekil 3.19: Farklı sınıf-üyelik olasılıkları için entropi değerleri*

Gini safsızlığı, yanlış sınıflandırma olasılığını en aza indirmek için bir kriter olarak anlaşılabilir:

$$IG(t) = \sum_{ü1}^{c} p(i|t)\ (1 - p(i|t)) = 1 - \sum_{ü1}^{c} p(i|t)^2$$

Entropiye benzer şekilde, Gini safsızlığı, sınıflar mükemmel bir şekilde karışıksa, örneğin ikili sınıf ortamında ($c = 2$) maksimumdur:

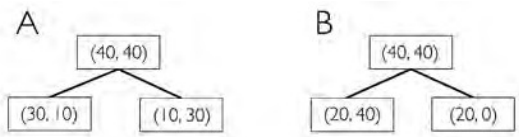
$$IG(t) = 1 - \sum_{ü1}^{c} 0{,}5^2 = 0{,}5$$

Ancak, pratikte hem Gini safsızlığı hem de entropi tipik olarak çok benzer sonuçlar verir ve farklı budama kesme değerlerini denemek yerine farklı safsızlık kriterleri kullanarak ağaçları değerlendirmek için fazla zaman harcamaya değmez. Aslında, daha sonra *Şekil 3.21'*de göreceğiniz gibi, hem Gini safsızlığı hem de entropi benzer bir şekle sahiptir.

Bir başka safsızlık ölçütü de sınıflandırma hatasıdır:

$$IE(t) = 1 - \text{maks}\{p(i|t)\}$$

Bu, budama için faydalı bir kriterdir, ancak düğümlerin sınıf olasılıklarındaki değişikliklere daha az duyarlı olduğu için bir karar ağacını büyütmek için önerilmez. Bunu *Şekil 3.20'*de gösterilen iki olası bölme senaryosuna bakarak gösterebiliriz:




*Şekil 3.20: Karar ağacı veri bölünmeleri*

Ana düğümde, sınıf 1'den 40 örnek ve sınıf 2'den 40 örnek içeren ve $D_{leß}$ *ve* $D_{right}$ olmak üzere iki veri kümesine böldüğümüz bir veri kümesi olan $D_p$ ile başlıyoruz. Bölme kriteri olarak sınıflandırma hatasını kullanan bilgi kazancı, *A* ve *B* olmak üzere her iki senaryoda da aynı olacaktır ($IG_E$ = 0,25):

$I_{E(}Dp) = 1 - 0{,}5 = 0{,}5$

*AA* $I_{E(Dleft)} = 1 - \frac{3}{4} = 0{,}25$

*AA* $IE(_{Drirrht}) = 1 - \frac{3}{4} = 0{,}25$

*AA* $_{IGE} = 0{,}5 - \frac{4}{8}\,0{,}25 - \frac{4}{8}\,0{,}25 = 0{,}25$

*AB* $I_{E(Dleft)} = 1 - \frac{4}{6} = \frac{1}{3}$

*AB* $IE(_{Drirrht}) = 1 - 1 = 0$

*AB* $_{IGE} = 0{,}5 - \frac{6}{8} \times \frac{1}{3} - 0 = 0{,}25$

Ancak, Gini safsızlığı *B* senaryosundaki ($IG_G = 0{,}1\overline{6}$) bölünmeyi, gerçekten daha saf olan *A* senaryosuna ($IG_G$ = 0,125) tercih edecektir:

$I_{G(}Dp) = 1 - (0{,}5^2 + 0{,}5^2) = 0{,}5$

*AA* $I_{G(Dleft)} = 1 - ((\frac{3}{4})^2 + (\frac{1}{4})^2) = \frac{3}{8} = 0.375$

*AA* $IG(_{Drirrht}) = 1 - ((\frac{1}{4})^2 + (\frac{3}{4})^2) = \frac{3}{8} = 0.375$

*AA* $_{IGG} = 0{,}5 - \frac{4}{8}\,0{,}375 - \frac{4}{8}\,0{,}375 = 0{,}125$

*AB* $I_{G(Dleft)} = 1 - ((\frac{2}{6})^2 + (\frac{4}{6})^2) = \frac{4}{9} = 0.\overline{4}$

*AB* $IG(_{Drirrht}) = 1 - (1^2 + 0^2) = 0$

*AB* $_{IGG} = 0.5 - \frac{6}{8}\,0.\overline{4} - 0 = 0.16$

 olimpikbarbell@gmail.com 

Benzer şekilde, entropi kriteri de B senaryosunu ($IG_H = 0{,}31$) A senaryosuna ($IG_H = 0{,}19$) tercih edecektir:

$$_{IH}(Dp) = -(0{,}5\ \log_2(0{,}5) + 0{,}5\ \log_2(0{,}5)) = 1$$

$$_{AAIH}(Dleft) = -(\ 4\ \ \frac{3}{4}\log_2(\frac{3}{4}) + \frac{1}{4}\log_2(\frac{1}{4})) = 0.81$$

$$AA\ \ _{IH}(Drirrht) = -(\frac{1}{4}\log_2(\frac{1}{4}) + \frac{3}{4}\log_2(\frac{3}{4})) = 0.81$$

$$AA\ \ _{IGH} = 1 - \frac{4}{8}0{,}81 - \frac{4}{8}0{,}81 = 0{,}19$$

$$_{BAIH}(Dleft) = -(\ 6\ \ \frac{2}{6}\log_2(\frac{2}{6}) + \frac{4}{6}\log_2(\frac{4}{6})) = 0.92$$

$$AB\ \ _{IH}(Drirrht) = 0$$

$$BA\ _{IGH} = 1 - \frac{6}{8}0{,}92 - 0 = 0{,}31$$

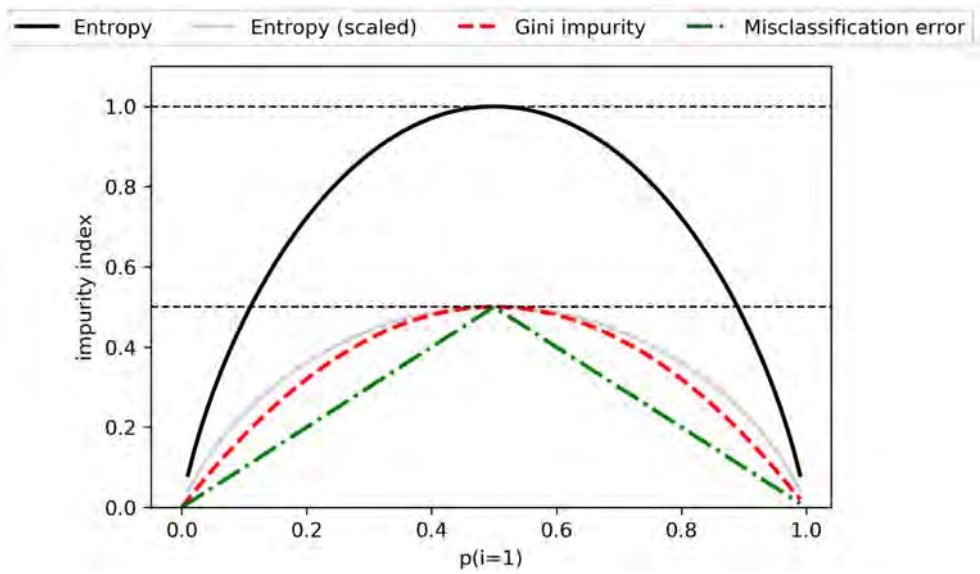
Daha önce tartıştığımız üç farklı safsızlık kriterinin daha görsel bir karşılaştırması için, sınıf 1 için [0, 1] olasılık aralığı için safsızlık endekslerini çizelim. Gini safsızlığının entropi ve sınıflandırma hatası arasında bir ara ölçü olduğunu gözlemlemek için entropinin ölçeklendirilmiş bir versiyonunu (entropi / 2) da ekleyeceğimizi unutmayın. Kod aşağıdaki gibidir:

```
>> import matplotlib.pyplot as plt
>> import numpy as np
>> def gini(p):
...         p*(1 - p ) + (1 - p)*(1 - (1-p)) döndür
>> def entropi(p):
...         return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>> def error(p):
...         return 1 - np.max([p, 1 - p])
>> x = np.arange(0.0, 1.0, 0 . 01)
>> ent = [entropi(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>> ax = plt.subplot(111)
```

6587191CA584E

```
>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                          ['Entropi', 'Entropi (ölçeklendirilmiş)',
...                           'Gini kirliliği',
...                           'Yanlış sınıflandırma hatası'],
...                          ['-', '-', '--', '-.'],
...                          ['siyah', 'açık gri',
...                           'kırmızı', 'yeşil', 'camgöbeği']):
...       line = ax.plot(x, i, label=lab,
...                      linestyle=ls, lw=2, color=c)
>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...           ncol=5, fancybox=True, shadow=False)
>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>> plt.ylim([0, 1.1])
>> plt.xlabel('p(i=1)')
>> plt.ylabel('safsızlık endeksi')
>> plt.show()
```

Yukarıdaki kod örneği tarafından üretilen çizim aşağıdaki gibidir:




*Şekil 3.21: 0 ile 1 arasında farklı sınıf-üyelik olasılıkları için farklı safsızlık endeksleri*

## Karar ağacı oluşturma

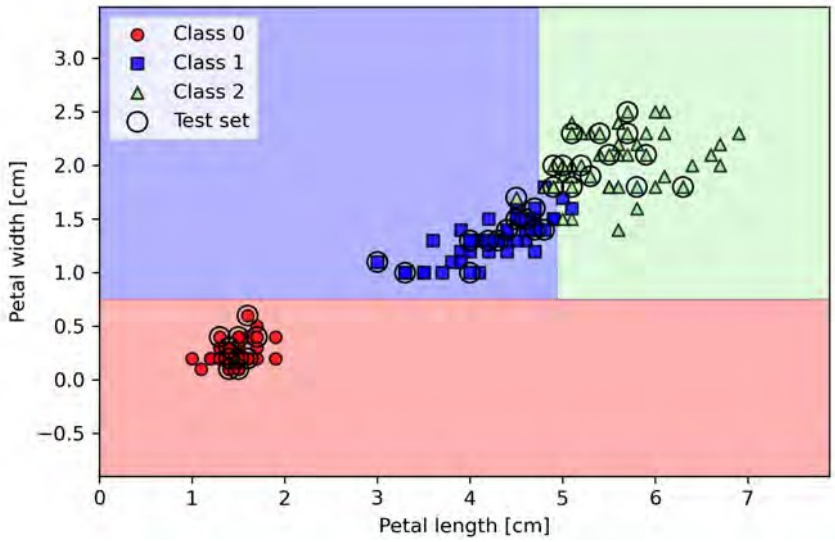
Karar ağaçları, özellik uzayını dikdörtgenlere bölerek karmaşık karar sınırları oluşturabilir. Ancak, karar ağacı ne kadar derin olursa, karar sınırı o kadar karmaşık hale geleceğinden ve bu da kolayca aşırı uyuma neden olabileceğinden dikkatli olmalıyız. Şimdi scikit-learn kullanarak, safsızlık kriteri olarak Gini safsızlığını kullanarak maksimum derinliği 4 olan bir karar ağacı eğiteceğiz.

 olimpikbarbell@gmail.com 

Görselleştirme amacıyla özellik ölçeklendirme istenebilse de, özellik ölçeklendirmenin karar ağacı algoritmaları için bir gereklilik olmadığını unutmayın. Kod aşağıdaki gibidir:

```
>> from sklearn.tree import DecisionTreeClassifier
> >    tree_model = DecisionTreeClassifier(criterion='gini',
...                                        max_depth=4,
...                                        random_state=1)
>> tree_model.fit(X_train, y_train)
>> X_combined = np.vstack((X_train, X_test))
>> y_combined = np.hstack((y_train, y_test))
> > >  plot_decision_regions(X_combined,
...                          y_combined,
...                          sınıflandırıcı=tree_model,
...                          test_idx=range(105, 150))
>> plt.xlabel('Petal uzunluğu [cm]')
>> plt.ylabel('Petal genişliği [cm]')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

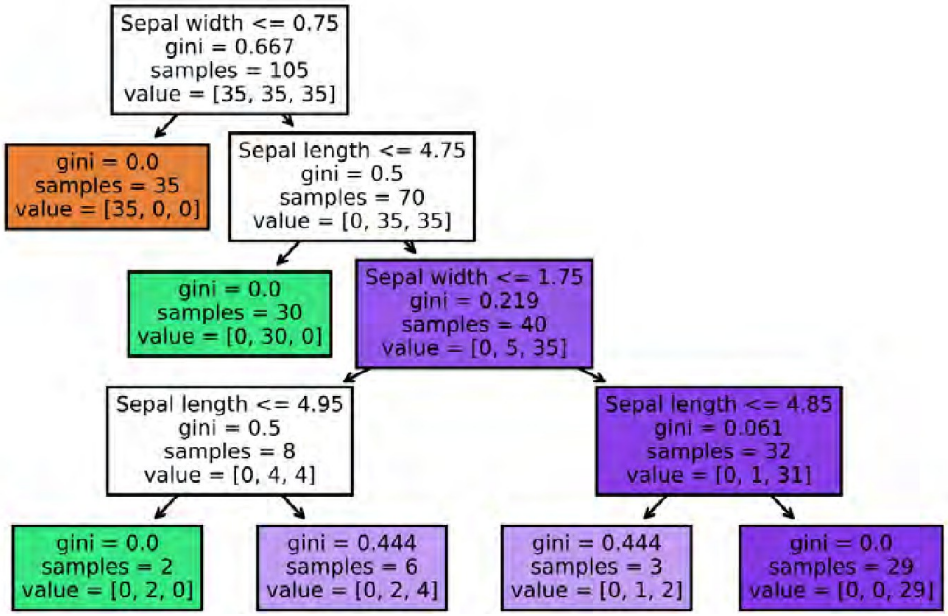
Kod örneğini çalıştırdıktan sonra, karar ağacının tipik eksen-paralel karar sınırlarını elde ederiz:




*Şekil 3.22: Karar ağacı kullanılarak Iris verilerinin karar sınırları*

Scikit-learn'deki güzel bir özellik, aşağıdaki kod aracılığıyla eğitimden sonra karar ağacı modelini kolayca görselleştirmemize izin vermesidir:

```
>>> from sklearn import ağaç
>> özellik_adları = ['Sepal uzunluk', 'Sepal genişlik',
...                  'Petal uzunluğu', 'Petal genişliği']
>> tree.plot_tree(tree_model,
...                feature_names=özellik_adları,
...                filled=True)
>> plt.show()
```




*Şekil 3.23: Iris veri setine uygun bir karar ağacı modeli*

Çağırdığımız plot_tree fonksiyonunda filled=True ayarı, düğümleri o düğümdeki çoğunluk sınıf etiketine göre renklendirir. https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html adresindeki belgelerde bulabileceğiniz birçok ek seçenek mevcuttur.

Karar ağacı şekline baktığımızda, artık karar ağacının eğitim veri setimizden çıkardığı bölünmeleri güzel bir şekilde izleyebiliriz. Her bir düğümdeki özellik bölme kriteriyle ilgili olarak, soldaki dalların "Doğru "ya, sağdaki dalların ise "Yanlış "a karşılık geldiğine dikkat edin.

 olimpikbarbell@gmail.com 

Kök düğüme bakıldığında, en üstte 105 örnekle başlamaktadır. İlk bölme işlemi, kök düğümü 35 örnekli (sol çocuk düğüm) ve 70 örnekli (sağ çocuk düğüm) iki çocuk düğüme bölmek için çanak genişliği kesme ≤ 0,75 cm kullanır. İlk bölmeden sonra, sol alt düğümün zaten saf olduğunu ve yalnızca Iris-setosa sınıfından örnekler içerdiğini görebiliriz (Gini safsızlığı = 0). Sağdaki diğer bölmeler daha sonra Iris-versicolor ve Iris-virginica sınıfından örnekleri ayırmak için kullanılır.

Bu ağaca ve ağacın karar bölgesi grafiğine baktığımızda, karar ağacının çiçek sınıflarını ayırma konusunda çok iyi bir iş çıkardığını görebiliriz. Ne yazık ki, scikit-learn şu anda bir karar ağacını manuel olarak sonradan budamak için işlevsellik uygulamamaktadır. Bununla birlikte, önceki kod örneğimize geri dönebilir, karar ağacımızın max_depth değerini 3 olarak değiştirebilir ve mevcut modelimizle karşılaştırabiliriz, ancak bunu ilgilenen okuyucu için bir alıştırma olarak bırakıyoruz.

Alternatif olarak, scikit-learn karar ağaçları için otomatik bir maliyet karmaşıklığı sonrası budama prosedürü sağlar. İlgilenen okuyucular bu daha gelişmiş konu hakkında daha fazla bilgiyi aşağıdaki eğitimde bulabilirler: https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning. html.

## Rastgele ormanlar aracılığıyla çoklu karar ağaçlarının birleştirilmesi

Topluluk yöntemleri, iyi sınıflandırma performansları ve aşırı uyuma karşı sağlamlıkları nedeniyle son on yılda makine öğrenimi uygulamalarında büyük popülerlik kazanmıştır. **Bagging** ve **boosting** dahil olmak üzere farklı topluluk yöntemlerini daha sonra *Bölüm 7'de (Topluluk Öğrenmesi için Farklı Modelleri Birleştirme*) ele alacak olsak da, iyi ölçeklenebilirliği ve kullanım kolaylığı ile bilinen karar ağacı tabanlı rastgele **orman** algoritmasını tartışalım. Rastgele orman, bir karar ağaçları **topluluğu** olarak düşünülebilir. Rastgele ormanın arkasındaki fikir, daha iyi bir genelleme performansına sahip olan ve aşırı uyuma daha az duyarlı olan daha sağlam bir model oluşturmak için bireysel olarak yüksek varyanstan muzdarip birden fazla (derin) karar ağacının ortalamasını almaktır. Rastgele orman algoritması dört basit adımda özetlenebilir:

1. *n* boyutunda rastgele bir **bootstrap** örneği çizin (eğitim veri kümesinden *n* örneği değiştirerek rastgele seçin).
2. Bootstrap örneğinden bir karar ağacı oluşturun. Her düğümde:
    a. Değiştirmeden rastgele *d* özellik seçin.
    b. Amaç fonksiyonuna göre en iyi bölmeyi sağlayan özelliği kullanarak düğümü bölün, örneğin bilgi kazancını maksimize edin.
3. *Adım 1-2 k* kez tekrarlayın.
4. Sınıf etiketini **oy çokluğuyla** atamak için her ağacın tahminini toplayın. Çoğunluk oylaması *Bölüm 7'*de daha ayrıntılı olarak ele alınacaktır.

Bireysel karar ağaçlarını eğitirken *2. adımda* küçük bir değişikliğe dikkat etmeliyiz: her düğümde en iyi bölünmeyi belirlemek için tüm özellikleri değerlendirmek yerine, yalnızca bunların rastgele bir alt kümesini dikkate alıyoruz.

**Değişimli ve değişimsiz örnekleme**

"Değiştirmeli" ve "değiştirmesiz" örnekleme terimlerine aşina değilseniz, basit bir düşünce deneyi üzerinden gidelim. Bir kupadan rastgele sayılar çektiğimiz bir piyango oyunu oynadığımızı varsayalım. 0, 1, 2, 3 ve 4 olmak üzere beş benzersiz sayı içeren bir kupa ile başlıyoruz ve her turda tam olarak bir sayı çekiyoruz. İlk turda, küpten belirli bir sayı çekme şansı 1/5 olacaktır. Şimdi, değiştirmesiz örneklemede, her turdan sonra sayıyı küpe geri koymayız. Sonuç olarak, bir sonraki turda kalan sayılar kümesinden belirli bir sayıyı çekme olasılığı bir önceki tura bağlıdır. Örneğin, elimizde 0, 1, 2 ve 4 sayılarından oluşan bir kalan sayı kümesi varsa, bir sonraki turda 0 sayısını çekme olasılığı 1/4 olacaktır.

Ancak, değiştirmeli rastgele örneklemede, çekilen sayıyı her zaman küpe geri göndeririz, böylece her turda belirli bir sayıyı çekme olasılığı değişmez; aynı sayıyı birden fazla kez çekebiliriz. Başka bir deyişle, *değiştirmeli* örneklemede örnekler (sayılar) bağımsızdır ve kovaryansları sıfırdır. Örneğin, beş tur rastgele sayı çekme işleminden elde edilen sonuçlar aşağıdaki gibi görünebilir:

- Değiştirme olmaksızın rastgele örnekleme: 2, 1, 3, 4, 0
- Değiştirme ile rastgele örnekleme: 1, 3, 3, 4, 1

Rastgele ormanlar karar ağaçlarıyla aynı düzeyde yorumlanabilirlik sunmasa da, rastgele ormanların en büyük avantajı, iyi hiperparametre değerleri seçme konusunda çok fazla endişelenmemize gerek olmamasıdır. Topluluk modeli, bireysel karar ağaçları arasındaki tahminlerin ortalamasından kaynaklanan gürültüye karşı oldukça sağlam olduğundan, genellikle rastgele ormanı budamamız gerekmez. Uygulamada dikkat etmemiz gereken tek parametre, rastgele orman için seçtiğimiz ağaç sayısı, *k,* (*adım 3*). Tipik olarak, ağaç sayısı arttıkça, hesaplama maliyetinin artması pahasına rastgele orman sınıflandırıcısının performansı da artar.

Uygulamada daha az yaygın olmasına rağmen, rastgele orman sınıflandırıcısının optimize edilebilen diğer hiper parametreleri - *Bölüm 6*, *Model Değerlendirme ve Hiper Parametre* Ayarlama *için En İyi Uygulamaları Öğrenme'*de tartışacağımız teknikler kullanılarak - sırasıyla bootstrap örneğinin boyutu, *n* (*adım 1*) ve her bölünme için rastgele seçilen özellik sayısı, *d'dir* (*adım 2a*). Bootstrap örnekleminin örneklem büyüklüğü *n* aracılığıyla, rastgele ormanın yanlılık-varyans dengesini kontrol ederiz.

Bootstrap örneğinin boyutunun küçültülmesi, belirli bir eğitim örneğinin bootstrap örneğine dahil edilme olasılığı daha düşük olduğu için bireysel ağaçlar arasındaki çeşitliliği artırır. Bu nedenle, bootstrap örneklerinin boyutunun küçültülmesi rastgele ormanın rastgeleliğini artırabilir ve aşırı uyumun etkisini azaltmaya yardımcı olabilir. Bununla birlikte, daha küçük bootstrap örnekleri tipik olarak rastgele ormanın genel performansının daha düşük olmasına ve eğitim ile test performansı arasında küçük bir farka, ancak genel olarak düşük bir test performansına neden olur. Tersine, önyükleme örneğinin boyutunun artırılması aşırı uyum derecesini artırabilir. Bootstrap örnekleri ve dolayısıyla bireysel karar ağaçları birbirlerine daha çok benzediğinden, orijinal eğitim veri kümesine daha yakından uymayı öğrenirler.

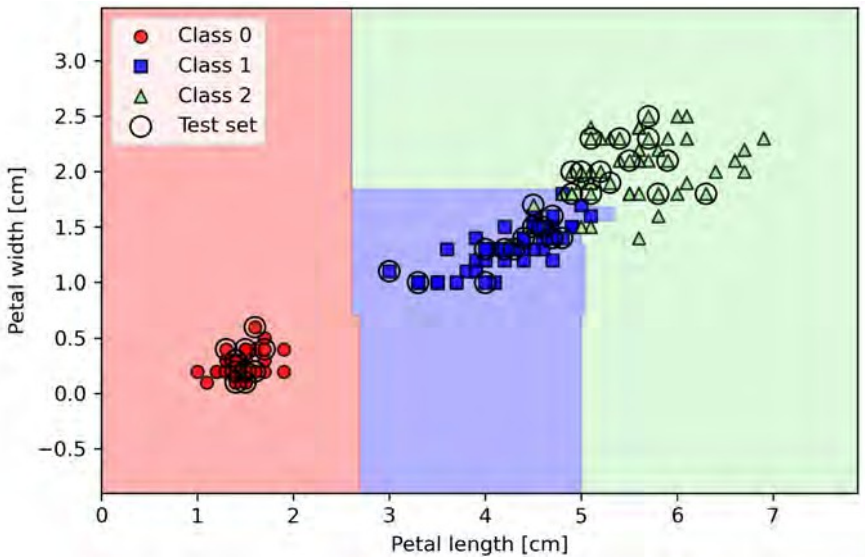
Scikit-learn'deki `RandomForestClassifier` uygulaması da dahil olmak üzere çoğu uygulamada, bootstrap örneğinin boyutu orijinal eğitim veri kümesindeki eğitim örneklerinin sayısına eşit olacak şekilde seçilir ve bu da genellikle iyi bir yanlılık-varyans dengesi sağlar. Her bölünmede özellik sayısı (*d)* için, eğitim veri kümesindeki toplam özellik sayısından daha küçük bir değer seçmek istiyoruz. Scikit-learn ve diğer uygulamalarda kullanılan makul bir varsayılan değer $d$ $d$ $\sqrt{m}$'dir; burada $m$, eğitim veri kümesindeki özellik sayısıdır.

Elverişli bir şekilde, scikit-learn'de kullanabileceğimiz bir uygulama zaten mevcut olduğundan, bireysel karar ağaçlarından rastgele orman sınıflandırıcısını kendimiz oluşturmak zorunda değiliz:

```
>> from sklearn.ensemble import RandomForestClassifier
>>  orman = RandomForestClassifier(n_estimators=25,
...                                 random_state=1,
...                                 n_jobs=2)
>> forest.fit(X_train, y_train)
>>>  plot_decision_regions(X_combined, y_combined,
...                        sınıflandırıcı=orman, test_idx=aralık(105,150))
>> plt.xlabel('Petal uzunluğu [cm]')
>> plt.ylabel('Petal genişliği [cm]')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

6587191CA584E

Önceki kodu çalıştırdıktan sonra, *Şekil 3.24'te* gösterildiği gibi rastgele ormandaki ağaçlar topluluğu tarafından oluşturulan karar bölgelerini görmeliyiz:




*Şekil 3.24: Rastgele orman kullanarak Iris veri kümesi üzerinde karar sınırları*

Önceki kodu kullanarak, n_estimators parametresi aracılığıyla 25 karar ağacından rastgele bir orman eğittik. Varsayılan olarak, düğümleri bölmek için bir ölçüt olarak Gini safsızlık ölçütünü kullanır. Çok küçük bir eğitim veri kümesinden çok küçük bir rastgele orman yetiştiriyor olsak da, bilgisayarımızın birden fazla çekirdeğini (burada iki çekirdek) kullanarak model eğitimini paralelleştirmemize olanak tanıyan n_jobs pa- rametresini tanıtım amacıyla kullandık. Bu kodda hatalarla karşılaşırsanız, bilgisayarınız çoklu işlemeyi desteklemiyor olabilir. n_jobs parametresini atlayabilir veya n_jobs=None olarak ayarlayabilirsiniz.

# K-en yakın komşular - tembel bir öğrenme algoritması

Bu bölümde tartışmak istediğimiz son denetimli öğrenme algoritması **k-en yakın komşu (KNN)** sınıflandırıcısıdır ve şu ana kadar tartıştığımız öğrenme algoritmalarından temelde farklı olduğu için özellikle ilginçtir.

KNN, **tembel öğrenicinin** tipik bir örneğidir. Görünürdeki basitliği nedeniyle değil, eğitim verilerinden ayırt edici bir işlev öğrenmediği, bunun yerine eğitim veri kümesini ezberlediği için "tembel" olarak adlandırılır.

 olimpikbarbell@gmail.com 

**Parametrik ve parametrik olmayan modeller**

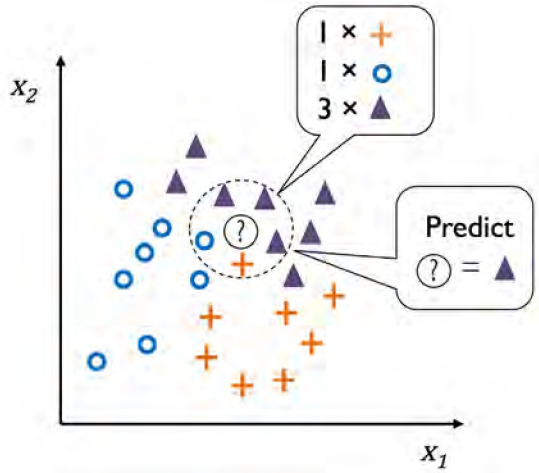
Makine öğrenimi algoritmaları parametrik ve parametrik olmayan modeller olarak gruplandırılabilir. Parametrik modelleri kullanarak, artık orijinal eğitim veri setine ihtiyaç duymadan yeni veri noktalarını sınıflandırabilen bir işlevi öğrenmek için eğitim veri setinden parametreleri tahmin ederiz. Parametrik modellerin tipik örnekleri perceptron, lojistik regresyon ve doğrusal SVM'dir. Buna karşılık, parametrik olmayan modeller sabit bir parametre seti ile karakterize edilemez ve parametre sayısı eğitim verisi miktarına göre değişir. Şimdiye kadar gördüğümüz iki parametrik olmayan model örneği, karar ağacı sınıflandırıcısı / rastgele orman ve çekirdek (ancak doğrusal değil) SVM'dir.

KNN, örnek tabanlı öğrenme olarak tanımlanan parametrik olmayan modellerin bir alt kategorisine aittir. Örnek tabanlı öğrenmeye dayalı modeller, eğitim veri setinin ezberlenmesi ile karakterize edilir ve tembel öğrenme, öğrenme süreci sırasında hiçbir (sıfır) maliyetle ilişkilendirilmeyen örnek tabanlı öğrenmenin özel bir durumudur.

KNN algoritmasının kendisi oldukça basittir ve aşağıdaki adımlarla özetlenebilir:

1. *k* sayısını ve bir uzaklık metriği seçin
2. Sınıflandırmak istediğimiz veri kaydının k-en *yakın* komşularını bulun
3. Sınıf etiketini oy çokluğuyla atayın

*Şekil 3.25,* yeni bir veri noktasına (**?**) en yakın beş komşusu arasındaki çoğunluk oylamasına dayalı olarak üçgen sınıf etiketinin nasıl atandığını göstermektedir:




*Şekil 3.25: k-en yakın komşular nasıl çalışır?*

KNN algoritması, seçilen mesafe metriğine bağlı olarak, eğitim veri kümesinde sınıflandırmak istediğimiz noktaya en yakın (en benzer) *k* örneği bulur. Veri noktasının sınıf etiketi daha sonra *k* en yakın komşusu arasında oy çokluğuyla belirlenir.

**Bellek tabanlı yaklaşımların avantajları ve dezavantajları**

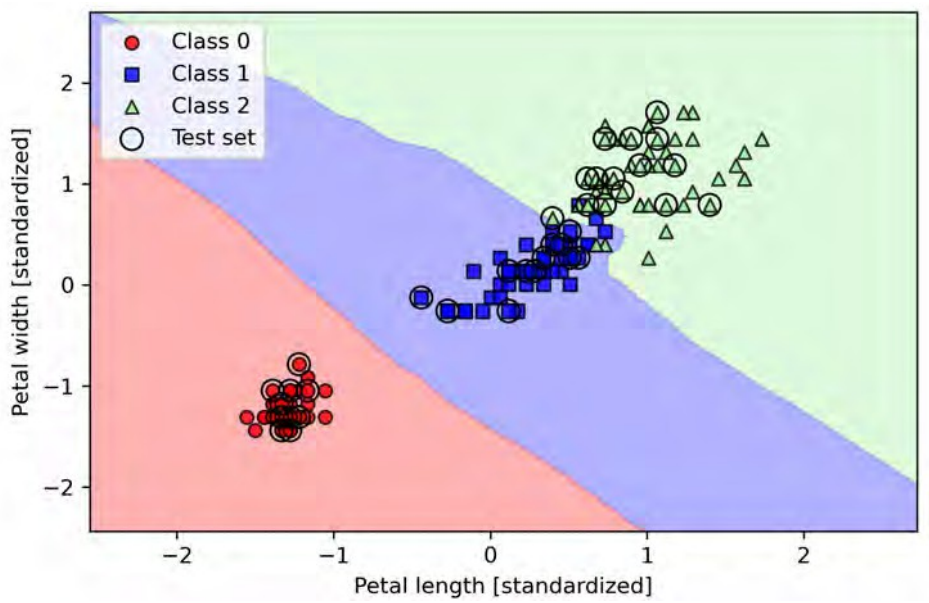
Bu tür bellek tabanlı bir yaklaşımın ana avantajı, yeni eğitim verileri topladıkça sınıflandırıcının hemen uyum sağlamasıdır. Bununla birlikte, veri kümesi çok az boyuta (özelliğe) sahip değilse ve algoritma eğitim verilerini daha etkili bir şekilde sorgulamak için verimli veri yapıları kullanılarak uygulanmamışsa, yeni örnekleri sınıflandırmak için hesaplama karmaşıklığının en kötü senaryoda eğitim veri kümesindeki örnek sayısıyla doğrusal olarak artmasıdır. Bu tür veri yapıları arasında scikit-learn'de desteklenen k-d ağacı (https://en.wikipedia.org/wiki/K-d_tree) ve top ağacı (https://en.wikipedia. org/wiki/Ball_tree) bulunmaktadır. Ayrıca, veri sorgulama için hesaplama maliyetlerinin yanı sıra, büyük veri kümeleri sınırlı depolama kapasiteleri açısından da sorunlu olabilir.

Bununla birlikte, nispeten küçük ve orta ölçekli veri setleriyle çalıştığımız birçok durumda, bellek tabanlı yöntemler iyi tahmin ve hesaplama performansı sağlayabilir ve bu nedenle birçok gerçek dünya problemine yaklaşmak için iyi bir seçimdir. En yakın komşu yöntemlerinin kullanımına ilişkin son örnekler arasında farmasötik ilaç hedeflerinin özelliklerinin tahmin edilmesi yer almaktadır (*Machine Learning to Identify Flexibility Signatures of Class A GPCR Inhibition,* Bio- molecules, 2020, Joe Bemister-Buffington, Alex J. Wolf, Sebastian Raschka, and Leslie A. Kuhn, https://www.mdpi.com/2218-273X/10/3/454) ve son teknoloji dil modelleri (*Efficient Nearest Neighbor Language Models,* 2021, Junxian He, Graham Neubig, and Taylor Berg-Kirkpatrick, https://arxiv.org/abs/2109.04212).

Aşağıdaki kodu çalıştırarak, şimdi scikit-learn'de bir Öklid mesafesi metriği kullanarak bir KNN modeli uygulayacağız:

```
>> from sklearn.neighbors import KNeighborsClassifier
>>  knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                                    metric='minkowski')
>> knn.fit(X_train_std, y_train)
>>>  plot_decision_regions(X_combined_std, y_combined,
...                                 sınıflandırıcı=knn, test_idx=aralık(105,150))
>> plt.xlabel('Petal uzunluğu [standartlaştırılmış]')
>> plt.ylabel('Petal genişliği [standartlaştırılmış]')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

Bu veri seti için KNN modelinde beş komşu belirleyerek, *Şekil 3.26*'da gösterildiği gibi nispeten düzgün bir karar sınırı elde ederiz:




*Şekil 3.26: Iris veri setinde k-en yakın komşuların karar sınırları*

**Bağların çözülmesi**

Eşitlik durumunda, KNN algoritmasının scikit-learn uygulaması, sınıflandırılacak veri kaydına daha yakın mesafeye sahip komşuları tercih edecektir. Komşuların benzer mesafelere sahip olması durumunda algoritma, eğitim veri kümesinde ilk sırada gelen sınıf etiketini seçecektir.

*Doğru k* seçimi, aşırı uyum ve yetersiz uyum arasında iyi bir denge bulmak için çok önemlidir. Ayrıca veri kümesindeki özelliklere uygun bir mesafe ölçütü seçtiğimizden de emin olmalıyız. Genellikle, santimetre cinsinden ölçülen özelliklere sahip Iris veri kümemizdeki çiçekler gibi gerçek değerli örnekler için basit bir Öklid mesafe ölçüsü kullanılır. Ancak, bir Öklid mesafe ölçüsü kullanıyorsak, her özelliğin mesafeye eşit katkıda bulunması için verileri standartlaştırmak da önemlidir. Önceki kodda kullandığımız minkowski mesafesi, aşağıdaki gibi yazılabilen Öklid ve Manhattan mesafesinin bir genellemesidir:

$$d(x^{(i)}, x^{(j)}) = \sqrt[p]{\sum_k |x_k^{(i)} - x_k^{(j)}|^p}$$

Parametreyi p=2 olarak ayarlarsak Öklid mesafesi veya p=1'de Manhattan mesafesi olur. Scikit-learn'de başka birçok mesafe ölçütü mevcuttur ve metrik parametresine eklenebilir. Bunlar https://scikit-learn.org/stable/modules/generated/sklearn.metrics adresinde listelenmiştir. DistanceMetric.html.

Son olarak, KNN'**nin boyutsallık laneti** nedeniyle aşırı uyum sağlamaya çok yatkın olduğunu belirtmek önemlidir. Boyutluluk laneti, sabit boyutlu bir eğitim veri kümesinin artan boyut sayısı için özellik uzayının giderek seyrekleştiği olgusunu tanımlar. En yakın komşuların bile yüksek boyutlu bir uzayda iyi bir tahmin veremeyecek kadar uzakta olduğunu düşünebiliriz.

Düzenli hale getirme kavramını lojistik regresyonla ilgili bölümde aşırı uyumdan kaçınmanın bir yolu olarak tartışmıştık. Ancak, karar ağaçları ve KNN gibi düzenli hale getirmenin uygulanabilir olmadığı modellerde, boyutluluk lanetinden kaçınmamıza yardımcı olması için özellik seçimi ve boyutluluk azaltma tekniklerini kullanabiliriz. Bu konu sonraki iki bölümde daha ayrıntılı olarak ele alınacaktır.

**GPU destekli alternatif makine öğrenimi uygulamaları**

Büyük veri kümeleriyle çalışırken, k-en yakın komşuları çalıştırmak veya birçok tahminciyle rastgele ormanlar kurmak önemli bilgi işlem kaynakları ve işlem süresi gerektirebilir. NVIDIA'nın CUDA kütüphanesinin son sürümleriyle uyumlu bir NVIDIA GPU ile donatılmış bir bilgisayarınız varsa RAPIDS ekosistemini (https://docs.rapids.ai/api) değerlendirmenizi öneririz. Örneğin, RAPIDS'in cuML (https://docs.rapids.ai/api/ cuml/stable/) kütüphanesi, scikit-learn'in makine öğrenimi algoritmalarının çoğunu işlem hızlarını artırmak için GPU desteği ile uygular. cuML ile ilgili bir tanıtımı https://docs.rapids.ai/api/cuml/stable/estimator_intro.html adresinde bulabilirsiniz. RAPIDS ekosistemi hakkında daha fazla bilgi edinmek istiyorsanız, lütfen RAPIDS ekibiyle işbirliği içinde yazdığımız serbestçe erişilebilen dergi makalesine de bakın: *Python'da Makine Öğrenimi: Veri Bilimi, Makine Öğrenimi ve Yapay Zeka Alanındaki Temel Gelişmeler ve Teknoloji Trendleri* (https://www.mdpi.com/2078-2489/11/4/193).

# Özet

Bu bölümde, doğrusal ve doğrusal olmayan problemlerin üstesinden gelmek için kullanılan birçok farklı makine öğrenimi algoritması hakkında bilgi edindiniz. Yorumlanabilirliği önemsiyorsak karar ağaçlarının özellikle çekici olduğunu gördünüz. Lojistik regresyon sadece SGD aracılığıyla çevrimiçi öğrenme için kullanışlı bir model olmakla kalmaz, aynı zamanda belirli bir olayın olasılığını tahmin etmemize de olanak tanır.

DVM'ler, çekirdek hilesi aracılığıyla doğrusal olmayan problemlere genişletilebilen güçlü doğrusal modeller olmalarına rağmen, iyi tahminler yapmak için ayarlanması gereken birçok parametreye sahiptirler. Buna karşılık, rastgele ormanlar gibi topluluk yöntemleri çok fazla parametre ayarı gerektirmez ve karar ağaçları kadar kolay aşırı uyum sağlamaz, bu da onları birçok pratik problem alanı için çekici modeller haline getirir. KNN sınıflandırıcı, herhangi bir model eğitimi olmadan, ancak hesaplama açısından daha pahalı bir tahmin adımı ile tahminler yapmamızı sağlayan tembel öğrenme yoluyla sınıflandırmaya alternatif bir yaklaşım sunar.

 olimpikbarbell@gmail.com 

Ancak, uygun bir öğrenme algoritmasının seçiminden daha da önemli olan, eğitim veri setimizdeki mevcut verilerdir. Hiçbir algoritma, bilgilendirici ve ayırt edici özellikler olmadan iyi tahminler yapamaz.

Bir sonraki bölümde, güçlü makine öğrenimi modelleri oluşturmamız gerektiği anlamına gelen verilerin ön işlenmesi, özellik seçimi ve boyut azaltma ile ilgili önemli konuları tartışacağız. Daha sonra, *Bölüm 6'da, Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Uygulamaları Öğrenme bölümünde*, modellerimizin performansını nasıl değerlendirip karşılaştırabileceğimizi göreceğiz ve farklı algoritmalara ince ayar yapmak için yararlı püf noktaları öğreneceğiz.

# Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E

olimpikbarbell@gmail.com

# 4

# İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme

Verinin kalitesi ve içerdiği faydalı bilgi miktarı, bir makine öğrenimi algoritmasının ne kadar iyi öğrenebileceğini belirleyen temel faktörlerdir. Bu nedenle, bir makine öğrenimi algoritmasına beslemeden önce bir veri kümesini incelediğimizden ve ön işlemden geçirdiğimizden emin olmak kesinlikle çok önemlidir. Bu bölümde, iyi makine öğrenimi modelleri oluşturmamıza yardımcı olacak temel veri ön işleme tekniklerini tartışacağız.

Bu bölümde ele alacağımız konular aşağıdaki gibidir:

- Veri setinden kayıp değerlerin çıkarılması ve yüklenmesi
- Kategorik verilerin makine öğrenimi algoritmaları için şekillendirilmesi
- Model yapımı için ilgili özelliklerin seçilmesi

## Eksik verileriyle başa çıkma

Gerçek dünya uygulamalarında eğitim örneklerimizde çeşitli nedenlerle bir veya daha fazla değerin eksik olması nadir değildir. Örneğin, veri toplama sürecinde bir hata olmuş olabilir, belirli ölçümler uygulanabilir olmayabilir veya bir ankette belirli alanlar boş bırakılmış olabilir. Eksik değerleri genellikle veri tablomuzda boş alanlar olarak ya da "sayı değil" anlamına gelen `NaN` veya `NULL` (ilişkisel veritabanlarında bilinmeyen değerlerin yaygın olarak kullanılan bir göstergesi) gibi yer tutucu dizeler olarak görürüz. Ne yazık ki, çoğu hesaplama aracı bu tür eksik değerlerle başa çıkamaz veya bunları göz ardı edersek öngörülemeyen sonuçlar üretir. Bu nedenle, daha ileri analizlere geçmeden önce bu eksik değerlerle ilgilenmemiz çok önemlidir.

Bu bölümde, veri kümemizdeki girdileri yeniden taşıyarak veya diğer eğitim örneklerinden ve özelliklerden eksik değerleri yükleyerek eksik değerlerle başa çıkmak için birkaç pratik teknik üzerinde çalışacağız.

## Tablo verilerinde eksik değerleri belirleme

Kayıp değerlerle başa çıkmak için çeşitli teknikleri tartışmadan önce, basit bir örnek oluşturalım Sorunu daha iyi kavramak için **virgülle ayrılmış değerler** (**CSV**) dosyasından DataFrame:

```
>> import pandas as pd
>> from io import StringIO
>> csv_data = \
... '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''
>> # Python 2.7 kullanıyorsanız, şunlara ihtiyacınız vardır
>> # dizeyi unicode'a dönüştürmek için:
>> # csv_data = unicode(csv_data)
>>> df = pd.read_csv(StringIO(csv_data))
>> df
      A     B     C    D
0   1.0   2.0   3.0  4.0
1   5.0   6.0   NaN  8.0
2  10.0  11.0  12.0  NaN
```

Önceki kodu kullanarak, CSV formatındaki verileri read_csv fonksiyonu aracılığıyla bir pandas DataFrame'e okuduk ve eksik iki hücrenin NaN ile değiştirildiğini fark ettik. Önceki kod örneğindeki StringIO fonksiyonu sadece örnekleme amacıyla kullanıldı. csv_data'ya atanan dizeyi, sabit diskimizdeki normal bir CSV dosyasıymış gibi bir pandas DataFrame'e okumamızı sağladı.

Daha büyük bir DataFrame için, eksik değerleri manuel olarak aramak sıkıcı olabilir; bu durumda, bir hücrenin sayısal bir değer içerip içermediğini (False) veya verilerin eksik olup olmadığını (True) gösteren Boolean değerlerine sahip bir DataFrame döndürmek için isnull yöntemini kullanabiliriz. sum yöntemini kullanarak, sütun başına eksik değer sayısını aşağıdaki gibi döndürebiliriz:

```
>>> df.isnull().sum()
A    0
B    0
C    1
D    1
dtype: int64
```

Bu şekilde, sütun başına eksik değer sayısını sayabiliriz; aşağıdaki alt bölümlerde, bu eksik verilerle nasıl başa çıkılacağına ilişkin farklı stratejilere göz atacağız.

 alinpikbarbell@gmail.com

**Pandas'ın DataFrame'i ile kullanışlı veri işleme**

scikit-learn başlangıçta yalnızca NumPy dizileriyle çalışmak için geliştirilmiş olsa da, bazen pandas'ın DataFrame'ini kullanarak verileri önceden işlemek daha uygun olabilir. Günümüzde çoğu scikit-learn fonksiyonu DataFrame nesnelerini girdi olarak desteklemektedir, ancak NumPy dizi işleme scikit-learn API'sinde daha olgun olduğundan, mümkün olduğunda NumPy dizilerinin kullanılması önerilir. Bir DataFrame'in altında yatan NumPy dizisine, bir scikit-learn tahmin edicisine be[illegible] yla her zaman erişebileceğinizi unutmayın:

```
>>> df.values
dizi([[ 1.,   2.,   3.,   4.],
      [ 5.,   6.,  nan,   8.],
      [ 10.,  11.,  12.,  nan]])
```

## Eksik değerlere sahip eğitim örneklerinin veya özelliklerin elenmesi

Eksik verilerle başa çıkmanın en kolay yollarından biri, ilgili özellikleri (sütunlar) veya eğitim örneklerini (satırlar) veri kümesinden tamamen çıkarmaktır; eksik değerlere sahip satırlar dropna yöntemiyle kolayca çıkarılabilir:

```
> >   df.dropna(axis=0) AB
            C      D
0    1.0 2 .0 3 . 0   4 . 0
```

Benzer şekilde, eksen bağımsız değişkenini 1 olarak ayarlayarak herhangi bir satırda en az bir NaN içeren sütunları düşürebiliriz:

```
>> df.dropna(axis=1) A  B
0    1.0       2.0
1    5.0       6.0
2 10.0       11.0
```

dropna yöntemi, kullanışlı olabilecek birkaç ek parametreyi destekler:

```
>> # yalnızca tüm sütunların NaN olduğu satırları bırakın
>> # (burada dizinin tamamını döndürür çünkü
>> # tüm değerleri NaN olan bir satıra sahip olun)
>> df.dropna(how='all')

        A       B       C       D
0    1.0     2.0     3.0     4.0
1    5.0     6.0     NaN     8.0
2   10.0    11.0    12.0     NaN
```

```
>> # 4'ten az gerçek değere sahip satırları bırak
>> df.dropna(thresh=4)
        A       B       C       D
0     1.0     2.0     3.0     4.0
>> # yalnızca belirli sütunlarda NaN görünen satırları bırakın (burada: 'C')
> >   df.dropna(subset=['C']) A
                B       C       D
0     1.0     2.0     3.0     4.0
2 10.0       11.0    12.0     NaN
```

Kayıp verilerin kaldırılması uygun bir yaklaşım gibi görünse de bazı dezavantajları da beraberinde getirir; örneğin, çok fazla örneği kaldırarak güvenilir bir analizi imkansız hale getirebiliriz. Ya da çok fazla özellik sütununu kaldırırsak, sınıflandırıcımızın sınıflar arasında ayrım yapmak için ihtiyaç duyduğu değerli bilgileri kaybetme riskiyle karşı karşıya kalırız. Bir sonraki bölümde, eksik değerlerle başa çıkmak için en yaygın kullanılan alternatiflerden birine bakacağız: enterpolasyon teknikleri.

## Kayıp değerleri belirleme

Çoğu zaman, eğitim örneklerinin kaldırılması veya tüm özellik sütunlarının atılması mümkün değildir, çünkü çok fazla değerli veri kaybedebiliriz. Bu durumda, veri kümemizdeki diğer eğitim örneklerinden eksik değerleri tahmin etmek için farklı enterpolasyon teknikleri kullanabiliriz. En yaygın enterpolasyon tekniklerinden biri, eksik değeri tüm özellik sütununun **ortalama değeriyle** değiştirdiğimiz ortalama **imputasyondur**. Bunu başarmanın uygun bir yolu, aşağıdaki kodda gösterildiği gibi scikit-learn'in SimpleImputer sınıfını kullanmaktır:

```
>> from sklearn.impute import SimpleImputer
>> import numpy as np
>> imr = SimpleImputer(missing_values=np.nan, strategy='mean')
>> imr = imr.fit(df.values)
>> imputed_data = imr.transform(df.values)
>>> imputed_data
    dizi([[ 1.,    2.,    3.,    4.],
          [ 5.,    6.,   7.5,    8.],
         [ 10.,   11.,   12.,    6.]])
```

Burada, her NaN değerini, her özellik sütunu için ayrı ayrı hesaplanan karşılık gelen ortalama ile değiştirdik. Strateji parametresi için diğer seçenekler medyan veya most_frequent'tir; bunlardan ikincisi eksik değerleri en sık rastlanan değerlerle değiştirir. Bu, örneğin kırmızı, yeşil ve mavi gibi renk adlarının kodlamasını depolayan bir özellik sütunu gibi kategorik özellik değerlerinin yüklenmesi için kullanışlıdır. Bu bölümün ilerleyen kısımlarında bu tür veri örnekleriyle karşılaşacağız.

 alinpikbarbell@gmail.com 

Alternatif olarak, eksik değerleri impute etmenin daha da kullanışlı bir yolu pandas'ın fillna yöntemini kullanmak ve argüman olarak bir imputation yöntemi sağlamaktır. Örneğin, pandas kullanarak, aşağıdaki komut aracılığıyla doğrudan DataFrame nesnesinde aynı ortalama imputasyonunu elde

```
>> df.fillna(df.mean())
```

edebiliriz:

| | A | B | C | D |
|---|---|---|---|---|
| **0** | 1.0 | 2.0 | 3.0 | 4.0 |
| **1** | 5.0 | 6.0 | 7.5 | 8.0 |
| **2** | 10.0 | 11.0 | 12.0 | 6.0 |

*Şekil 4.1: Verilerdeki eksik değerlerin ortalama ile değiştirilmesi*

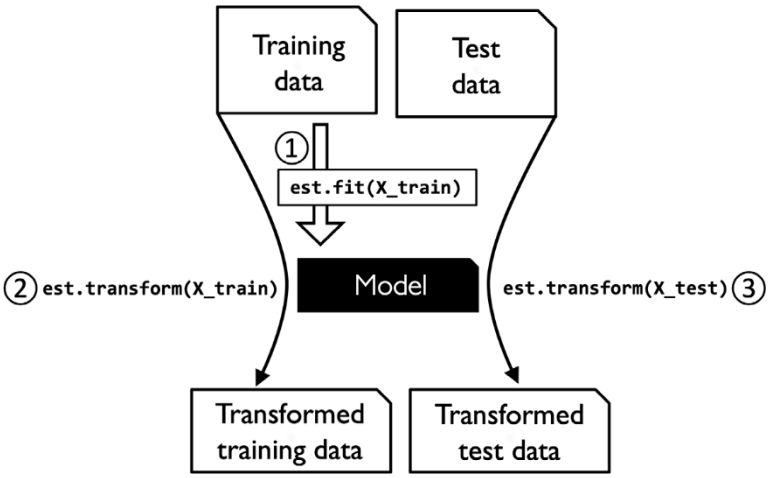
**Kayıp veriler için ek imputasyon yöntemleri**

Eksik özellikleri en yakın komşularla impute etmek için k-en yakın komşular yaklaşımına dayalı KNNImputer dahil olmak üzere ek imputasyon teknikleri için https://scikit-learn.org/stable/ modules/impute.html adresindeki scikit-learn imputasyon belgelerini öneririz.

## Scikit-learn tahmin edici API'sini anlama

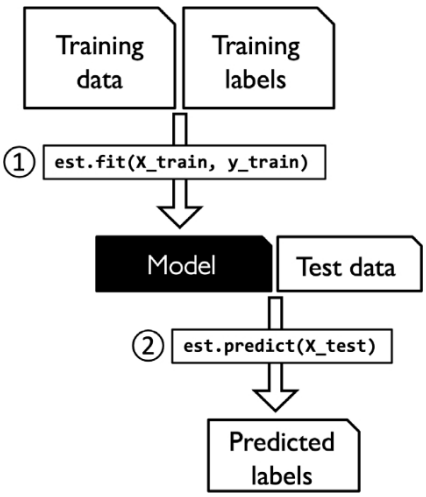
Önceki bölümde, veri setimizdeki eksik değerleri impute etmek için scikit-learn'deki SimpleImputer sınıfını kullandık. SimpleImputer sınıfı, veri dönüşümüyle ilgili Python sınıflarını uygulamak için kullanılan scikit-learn'deki **transformer** API'nin bir parçasıdır. (Lütfen scikit-learn dönüştürücü API'sinin doğal dil işlemede kullanılan dönüştürücü mimarisi *ile* karıştırılmaması gerektiğini unutmayın; bu konuyu *Bölüm 16, Dönüştürücüler - Doğal Dil İşlemeyi Dikkat Mekanizmalarıyla İyileştirme* bölümünde daha ayrıntılı olarak ele alacağız). Bu tahmin edicilerin iki temel yöntemi uyum ve dönüşümdür. Uydurma yöntemi, eğitim verilerinden parametreleri öğrenmek için kullanılır ve dönüştürme yöntemi, verileri dönüştürmek için bu parametreleri kullanır. Dönüştürülecek herhangi bir veri dizisinin, modeli oturtmak için kullanılan veri dizisiyle aynı sayıda özelliğe sahip olması gerekir.

*Şekil 4.2,* eğitim verileri üzerine yerleştirilen bir scikit-learn dönüştürücü örneğinin, bir eğitim veri kümesinin yanı sıra yeni bir test veri kümesini dönüştürmek için nasıl kullanıldığını göstermektedir:




*Şekil 4.2: Veri dönüşümü için scikit-learn API kullanımı*

*Bölüm 3*, *Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu'nda* kullandığımız sınıflandırıcılar, scikit-learn'deki **tahmin ediciler olarak** adlandırılan ve kavramsal olarak scikit- learn dönüştürücü API'sine çok benzeyen bir API'ye aittir. Tahmin ediciler bir `tahmin` yöntemine sahiptir ancak bu bölümün ilerleyen kısımlarında göreceğiniz gibi bir `dönüştürme yöntemine` de sahip olabilirler. Hatırlayacağınız gibi, bu tahmin edicileri sınıflandırma için eğitirken bir modelin parametrelerini öğrenmek için `fit` yöntemini de kullanmıştık. Ancak, denetimli öğrenme görevlerinde, modelin oturtulması için ek olarak sınıf etiketlerini sağlarız ve bunlar daha sonra *Şekil 4.3'te* gösterildiği gibi tahmin yöntemi aracılığıyla yeni, etiketsiz veri örnekleri hakkında `tahminler` yapmak için kullanılabilir:




*Şekil 4.3: Sınıflandırıcılar gibi tahmine dayalı modeller için scikit-learn API'sini kullanma*

 alinpikbarbell@gmail.com 

# Kategorik verilerinin işlenmesi

Şimdiye kadar sadece sayısal değerlerle çalıştık. Ancak, gerçek dünya veri kümelerinin bir veya daha fazla kategorik özellik sütunu içermesi nadir değildir. Bu bölümde, sayısal hesaplama kütüphanelerinde bu tür verilerle nasıl başa çıkılacağını görmek için basit ama etkili örneklerden yararlanacağız.

Kategorik verilerden bahsederken, sıralı ve **nominal** özellikler arasında daha fazla ayrım yapmamız gerekir. Ordinal özellikler, sıralanabilen veya sipariş edilebilen kategorik değerler olarak anlaşılabilir. Örneğin, tişört bedeni sıralı bir özellik olacaktır, çünkü bir sıra tanımlayabiliriz: Bunun aksine, nominal özellikler herhangi bir sıralamayı ima etmez; önceki örnekle devam edersek, tişört rengini nominal bir özellik olarak düşünebiliriz, çünkü örneğin kırmızının maviden daha büyük olduğunu söylemek tipik olarak mantıklı değildir.

## Pandas ile kategorik veri kodlama

Bu tür kategorik verileri işlemek için farklı teknikleri keşfetmeden önce, yeni bir DataFrame oluşturalım sorunu göstermek için:

```
>> import pandas as pd
>> df = pd.DataFrame([
...                 ['yeşil', 'M', 10.1, 'class2'],
...                 ['kırmızı', 'L', 13.5,
...                 'class1'],
...                 ['mavi', 'XL', 15.3, 'class2']])
>> df.columns = ['color', 'size', 'price', 'classlabel']
>> df
     renk beden fiyat classlabel
0    yeşil      M  10.1    sınıf2
1  kırmızı      L  13.5    sınıf1
2     mavi     XL  15.3    sınıf2
```

Önceki çıktıda görebileceğimiz gibi, yeni oluşturulan DataFrame bir nominal özellik (renk), bir sıralı özellik (boyut) ve bir sayısal özellik (fiyat) sütunu içerir. Sınıf etiketleri (denetimli öğrenme görevi için bir veri kümesi oluşturduğumuzu varsayarsak) son sütunda saklanır. Bu kitapta tartıştığımız sınıflandırma için öğrenme algoritmaları, sınıf etiketlerinde sıralı bilgileri kullanmaz.

## Sıralı özelliklerin haritalanması

Öğrenme algoritmasının sıralı özellikleri doğru yorumladığından emin olmak için kategorik dize değerlerini tam sayılara dönüştürmemiz gerekir. Ne yazık ki, boyut özelliğimizin etiketlerinin doğru sırasını otomatik olarak türetebilecek uygun bir işlev yoktur, bu nedenle eşlemeyi manuel olarak tanımlamamız gerekir. Aşağıdaki basit örnekte, özellikler arasındaki sayısal farkı bildiğimizi varsayalım, örneğin $XL = L + 1 = M + 2$:

```
>>> size_mapping = {'XL': 3,
...                 'L': 2,
...                 'M': 1}
```

```
>> df['boyut'] = d f ['boyut'].map(boyut_eşleme)
>> df
     renk beden fiyat classlabel
0    yeşil      1    10.1      sınıf2
1      kırmızı  2    13.5      sınıf1
2     mavi      3    15.3      sınıf2
```

Daha sonraki bir aşamada tamsayı değerlerini orijinal dize gösterimine geri dönüştürmek istersek, basitçe bir ters eşleme sözlüğü tanımlayabiliriz, inv_size_mapping = {v: k for k, v in size_ mapping.items()}, bu daha sonra dönüştürülmüş özellik sütununda pandas map yöntemi aracılığıyla kullanılabilir ve daha önce kullandığımız size_mapping sözlüğüne benzer. Bunu

```
>> inv_size_mapping = {v: k for k, v in size_mapping.items()}
> > >   df['size'].map(inv_size_mapping)
0    M
1    L
2    XL
ad: size, tür: nesne
```

aşağıdaki gibi kullanabiliriz:

## Sınıf etiketlerini kodlama

Birçok makine öğrenimi kütüphanesi, sınıf etiketlerinin tamsayı değerleri olarak kodlanmasını gerektirir. Scikit-learn'deki sınıflandırma tahmincilerinin çoğu sınıf etiketlerini dahili olarak tamsayılara dönüştürse de, teknik aksaklıklardan kaçınmak için sınıf etiketlerini tamsayı dizileri olarak sağlamak iyi bir uygulama olarak kabul edilir. Sınıf etiketlerini kodlamak için, daha önce tartışılan sıralı özelliklerin eşlenmesine benzer bir yaklaşım kullanabiliriz. Sınıf etiketlerinin sıralı *olmadığını* ve belirli bir dize etiketine hangi tam sayıyı atadığımızın önemli olmadığını

```
>> import numpy as np
>> class_mapping = {label: idx for idx, label in
...                    enumerate(np.unique(df['classlabel']))}
>> class_mapping
{'class1': 0, 'class2': 1}
```

hatırlamamız gerekir. Böylece, 0'dan başlayarak sınıf etiketlerini basitçe sıralayabiliriz:

Daha sonra, sınıf etiketlerini tam sayılara dönüştürmek için eşleme sözlüğünü kullanabiliriz:

```
>> df['classlabel'] = df['classlabel'].map(class_mapping)
>> df
     renk beden fiyat classlabel
0    yeşil      1    10.1        1
1      kırmızı  2    13.5        0
2     mavi      3    15.3        1
```

 alinpikbarbell@gmail.com 

Dönüştürülen sınıf etiketlerini orijinal dize gösterimine geri eşlemek için eşleme sözlüğündeki anahtar-değer çiftlerini aşağıdaki gibi tersine çevirebiliriz:

```
>> inv_class_mapping = {v: k for k, v in class_mapping.items()}
>> df['classlabel'] = df['classlabel'].map(inv_class_mapping)
>> df
     renk beden fiyat classlabel
0    yeşil      1    10.1      sınıf2
1    kırmızı    2    13.5      sınıf1
2    mavi       3    15.3      sınıf2
```

Alternatif olarak, bunu başarmak için doğrudan scikit-learn'de uygulanan uygun bir LabelEncoder sınıfı vardır:

```
>> from sklearn.preprocessing import LabelEncoder
>> class_le = LabelEncoder()
> >  y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([1, 0, 1])
```

fit_transform yönteminin fit ve transform'u ayrı ayrı çağırmak için bir kısayol olduğunu ve tamsayı sınıf etiketlerini orijinal dize gösterimlerine geri dönüştürmek için inverse_transform yöntemini kullanabileceğimizi unutmayın:

```
> >  class_le.inverse_transform(y)
array(['class2', 'class1', 'class2'], dtype=object)
```

## Nominal özellikler üzerinde tek seferlik kodlama gerçekleştirme

Önceki Sıralı *özellikleri eşleme* bölümünde, sıralı boyut özelliğini tam sayılara dönüştürmek için basit bir sözlük eşleme yaklaşımı kullandık. Scikit-learn'in sınıflandırma tahmin edicileri, sınıf etiketlerini herhangi bir sıralamayı (nominal) ima etmeyen kategorik veriler olarak ele aldığından, dize etiketlerini tamsayılara kodlamak için kullanışlı LabelEncoder'ı kullandık. Veri setimizin nominal renk sütununu dönüştürmek için benzer bir yaklaşımı aşağıdaki gibi

```
>>> X = df[['renk', 'boyut', 'fiyat']].values
>> color_le = LabelEncoder()
>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
 array([[1, 1, 10.1],
        [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

kullanabiliriz:

Önceki kodu çalıştırdıktan sonra, NumPy dizisinin ilk sütunu olan X artık yeni rengi tutar değerleri, aşağıdaki gibi kodlanır:

- mavi = 0
- yeşil = 1
- kırmızı = 2

Bu noktada durur ve diziyi sınıflandırıcımıza beslersek, kategorik verilerle uğraşırken en yaygın hatalardan birini yapmış oluruz. Sorunu görebiliyor musunuz? Renk değerleri belirli bir sırada gelmese de, önceki bölümlerde ele alınanlar gibi yaygın sınıflandırma modelleri artık yeşilin maviden, kırmızının da yeşilden daha büyük olduğunu varsayacaktır. Bu varsayım yanlış olsa da, bir sınıflandırıcı yine de faydalı sonuçlar üretebilir. Ancak bu sonuçlar optimal olmayacaktır.

Bu sorun için yaygın bir geçici çözüm, **tek seferlik kodlama** adı verilen bir teknik kullanmaktır. Bu yaklaşımın arkasındaki fikir, nominal özellik sütunundaki her benzersiz değer için yeni bir kukla özellik oluşturmaktır. Burada, renk özelliğini üç yeni özelliğe dönüştürürüz: mavi, yeşil ve kırmızı. İkili değerler daha sonra bir örneğin belirli rengini belirtmek için kullanılabilir; örneğin, mavi bir örnek mavi=1, yeşil=0, kırmızı=0 olarak kodlanabilir. Bu dönüşümü gerçekleştirmek için scikit-learn'in ön işleme modülünde uygulanan OneHotEncoder'ı kullanabiliriz:

```
>> from sklearn.preprocessing import OneHotEncoder
>>> X = df[['renk', 'boyut', 'fiyat']].values
>> color_ohe = OneHotEncoder()
>> color_ohe.fit_transform(X[:, 0].reshape(-1, 1)).toarray()
    array([[0., 1., 0.],
           [0., 0., 1.],
           [1., 0., 0.]])
```

Dizideki diğer iki sütunun da değiştirilmesini önlemek için OneHotEncoder'ı yalnızca tek bir sütuna (X[:, 0].reshape(-1, 1)) uyguladığımıza dikkat edin. Çok özellikli bir dizideki sütunları seçerek dönüştürmek istiyorsak, aşağıdaki gibi bir (name, transformer, column(s)) listesi kabul eden ColumnTransformer'ı kullanabiliriz:

```
>> from sklearn.compose import ColumnTransformer
>>> X = df[['renk', 'boyut', 'fiyat']].values
>> c_transf = Sütun Dönüştürücü([
...        ('onehot', OneHotEncoder(), [0]),
...        ('nothing', 'passthrough', [1, 2])
--- ])
>> c_transf.fit_transform(X).astype(float)
    array([[0.0, 1.0, 0.0, 1, 10.1],
           [0.0, 0.0, 1.0, 2, 13.5],
           [1.0, 0.0, 0.0, 3, 15.3]])
```

 olimpikbarbell@gmail.com 

Önceki kod örneğinde, 'passthrough' bağımsız değişkeni aracılığıyla yalnızca ilk sütunu değiştirmek ve diğer iki sütuna dokunmamak istediğimizi belirttik.

Tek seferlik kodlama yoluyla bu sahte özellikleri oluşturmanın daha da kullanışlı bir yolu, pandas'ta uygulanan get_dummies yöntemini kullanmaktır. Bir DataFrame'e uygulanan get_dummies yöntemi yalnızca dize sütunlarını dönüştürür ve diğer tüm sütunları değiştirmeden

```
>> pd.get_dummies(df['price', 'color', 'size']])
    price size color_blue color_green color_red
0    10.1    1          0           1         0
1    13.5    2          0           0         1
2    15.3    3          1           0         0
```

bırakır:

Tek seferlik kodlama veri kümeleri kullanırken, bunun belirli yöntemler (örneğin, matris ters çevirme gerektiren yöntemler) için sorun teşkil edebilecek çoklu doğrusallığı beraberinde getirdiğini unutmamalıyız. Özellikler yüksek oranda korelasyon gösteriyorsa, matrisleri tersine çevirmek hesaplama açısından zordur ve bu da sayısal olarak istikrarsız tahminlere yol açabilir. Değişkenler arasındaki korelasyonu azaltmak için, tek vuruşla kodlanmış diziden bir özellik sütununu kaldırabiliriz. Yine de bir özellik sütununu kaldırarak önemli bir bilgiyi kaybetmediğimizi unutmayın; örneğin, color_blue sütununu kaldırırsak, color_green=0 ve color_red=0 gözlemlersek, bu gözlemin mavi olması gerektiği anlamına geldiğinden, özellik bilgisi hala korunur.

Eğer get_dummies fonksiyonunu kullanırsak, fonksiyona True argümanını geçirerek ilk sütunu düşürebiliriz. drop_first parametresi, aşağıdaki kod örneğinde gösterildiği gibi:

```
>> pd.get_dummies(df['fiyat', 'renk', 'boyut']],
...                    drop_first=True)
    price size color_green color_red
0    10.1    1           1         0
1    13.5    2           0         1
2    15.3    3           0         0
```

OneHotEncoder aracılığıyla gereksiz bir sütunu bırakmak için drop='first' ayarını yapmamız ve categories='auto' aşağıdaki gibi:

```
>> color_ohe = OneHotEncoder(categories='auto', drop='first')
>> c_transf = Sütun Dönüştürücü([
...             ('onehot', color_ohe, [0]),
...             ('nothing', 'passthrough', [1, 2])
... ])
> > >   c_transf.fit_transform(X).astype(float)
  dizi([  1. ,  0. ,  1. ,  10.1],
      [  0. ,  1. ,  2. ,  13.5],
      [  0. ,  0. ,  3. ,  15.3]])
```

**Nominal veriler için ek kodlama şemaları**

Tek vuruşlu kodlama, sırasız kategorik değişkenleri kodlamanın en yaygın yolu olsa da, birkaç alternatif yöntem mevcuttur. Bu tekniklerden bazıları, yüksek kardinaliteye (çok sayıda benzersiz kategori etiketi) sahip kategorik özelliklerle çalışırken faydalı olabilir. Örnekler şunları içerir:

- İkili kodlama, bir-sıcak kodlamaya benzer çoklu ikili özellikler üretir, ancak daha az özellik sütunu gerektirir, yani $K$ - 1 yerine $\log_2(K)$, burada $K$ benzersiz kategori sayısıdır. İkili kodlamada, sayılar önce ikili gösterimlere dönüştürülür ve ardından her ikili sayı konumu yeni bir özellik sütunu oluşturur.
- Her bir kategorinin etiketini, eğitim setinde kaç kez veya hangi sıklıkta geçtiğiyle değiştiren sayım veya sıklık kodlaması.

Bu yöntemlerin yanı sıra ek kategorik kodlama şemaları scikit-learn uyumlu category_encoders kütüphanesi aracılığıyla kullanılabilir: https://contrib.scikit-learn. org/category_encoders/.

Bu yöntemlerin model performansı açısından tek vuruşlu kodlamadan daha iyi performans göstereceği garanti edilmese de, kategorik kodlama şeması seçimini model performansını iyileştirmek için ek bir "hiperparametre" olarak düşünebiliriz.

## İsteğe bağlı: sıralı özelliklerin kodlanması

Sıralı özellik kategorileri arasındaki sayısal farklardan emin değilsek veya iki sıralı değer arasındaki fark tanımlanmamışsa, bunları 0/1 değerleriyle bir eşik kodlaması kullanarak da kodlayabiliriz. Örneğin, M, L ve XL değerlerine sahip özellik boyutunu x > M ve x > L olmak üzere iki yeni özelliğe bölebiliriz:

```
>> df = pd.DataFrame([['yeşil', 'M', 10.1,
...                     'class2'],
...                    ['kırmızı', 'L', 13.5,
...                     'class1'],
...                    ['mavi', 'XL', 15.3,
...                     'class2']])
>> df.columns = ['color', 'size', 'price',
...               'classlabel']
>> df
```

Bu değişkenleri değer-eşik yaklaşımını kullanarak kodlamak amacıyla özel lambda ifadeleri yazmak için pandas'ın DataFrame'inin apply yöntemini kullanabiliriz:

```
>> df['x > M'] = df['boyut'].apply(
...     lambda x: 1 if x in {'L', 'XL'} else 0)
>> df['x > L'] = df['size'].apply(
...     lambda x: 1 if x == 'XL' else 0)
```

 alinpikbarbell@gmail.com 

```
>> del df['size']
>> df
```

# Bir veri kümesini ayrı eğitim ve test veri kümelerine bölme

Bir veri kümesini eğitim ve test için ayrı veri kümelerine bölme kavramını *Bölüm 1*, *Bilgisayarlara Verilerden Öğrenme* Yeteneği Kazandırma ve *Bölüm 3*, *Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu'nda* kısaca tanıttık. Tahminleri test setindeki gerçek etiketlerle karşılaştırmanın, modelimizi gerçek dünyaya salmadan önce tarafsız performans değerlendirmesi olarak anlaşılabileceğini unutmayın. Bu bölümde, yeni bir veri kümesi olan **Wine** veri kümesini hazırlayacağız. Veri kümesini önceden işledikten sonra, bir veri kümesinin boyutluluğunu azaltmak için özellik seçimine yönelik farklı teknikleri keşfedeceğiz.

Şarap veri kümesi, UCI makine öğrenimi re-pozitifinden (https://archive.ics.uci.edu/ml/datasets/Wine) elde edilebilen bir başka açık kaynak veri kümesidir; farklı kimyasal özelliklerini tanımlayan 13 özelliğe sahip 178 şarap örneğinden oluşur.

**Şarap veri setinin elde edilmesi**

Wine veri setinin (ve bu kitapta kullanılan diğer tüm veri setlerinin) bir kopyasını bu kitabın kod paketinde bulabilirsiniz; bu kopyayı ofis dışında çalışıyorsanız veya `https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data` adresindeki veri setine UCI sunucusunda geçici olarak erişilemiyorsa kullanabilirsiniz. Örneğin, Wine veri setini yerel bir dizinden yüklemek için şu satırı değiştirebilirsiniz:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/wine/wine.data',
    header=None
)
```

bir sonraki ile:

```
df = pd.read_csv( 'your/local/path/to/wine.data',
    header=None
)
```

Pandas kütüphanesini kullanarak, UCI makine öğrenimi deposundan açık kaynaklı Wine veri setini doğrudan okuyacağız:

```
>>  df_wine = pd.read_csv('https://archive.ics.uci.edu/'
...                        'ml/machine-learning-databases/'
...                        'wine/wine.data', header=None)
```

```
>> df_wine.columns = ['Sınıf etiketi', 'Alkol',
...                   'Malik asit', 'Kül',
...                   'Külün alkalinitesi', 'Magnezyum',
...                   'Toplam fenoller', 'Flavanoidler',
...                   'Nonflavanoid fenoller',
...                   "Proantosiyaninler",
...                   'Renk yoğunluğu', 'Ton',
...                   'Seyreltilmiş şarapların OD280/OD315'i',
...                   'Proline']
>> print('Sınıf etiketleri', np.unique(df_wine['Sınıf
etiketi'])) Sınıf etiketleri [1 2 3]
>> df_wine.head()
```

Şarap veri kümesinde yer alan ve 178 şarap örneğinin kimyasal özelliklerini tanımlayan 13 farklı özellik aşağıdaki tabloda listelenmiştir:

| | Class label | Alcohol | Malic acid | Ash | Alcalinity of ash | Magnesium | Total phenols | Flavanoids | Nonflavanoid phenols | Proanthocyanins | Color intensity | Hue | OD280/OD315 of diluted wines | Proline |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 14.23 | 1.71 | 2.43 | 15.6 | 127 | 2.80 | 3.06 | 0.28 | 2.29 | 5.64 | 1.04 | 3.92 | 1065 |
| 1 | 1 | 13.20 | 1.78 | 2.14 | 11.2 | 100 | 2.65 | 2.76 | 0.26 | 1.28 | 4.38 | 1.05 | 3.40 | 1050 |
| 2 | 1 | 13.16 | 2.36 | 2.67 | 18.6 | 101 | 2.80 | 3.24 | 0.30 | 2.81 | 5.68 | 1.03 | 3.17 | 1185 |
| 3 | 1 | 14.37 | 1.95 | 2.50 | 16.8 | 113 | 3.85 | 3.49 | 0.24 | 2.18 | 7.80 | 0.86 | 3.45 | 1480 |
| 4 | 1 | 13.24 | 2.59 | 2.87 | 21.0 | 118 | 2.80 | 2.69 | 0.39 | 1.82 | 4.32 | 1.04 | 2.93 | 735 |

*Şekil 4.4: Şarap veri setinden bir örnek*

Örnekler, veri kümesi özetinde (https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.names) açıklandığı gibi, İtalya'da aynı bölgede yetiştirilen ancak farklı şarap çeşitlerinden elde edilen üç farklı üzüm türüne atıfta bulunan 1, 2 ve 3 olmak üzere üç farklı sınıftan birine aittir.

Bu veri kümesini ayrı test ve eğitim veri kümelerine rastgele bölmenin uygun bir yolu, scikit-learn'in model_selection alt modülündeki train_test_split işlevini kullanmaktır:

```
>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>> X_train, X_test, y_train, y_test =\
...        train_test_split(X, y,
...                         test_size=0.3,
...                         random_state=0,
...                         stratify=y)
```

İlk olarak, 1-13 özellik sütunlarının NumPy dizi gösterimini X değişkenine atadık ve ilk sütundaki sınıf etiketlerini y değişkenine atadık. Ardından, X ve y'yi ayrı eğitim ve test veri kümelerine rastgele bölmek için train_test_split işlevini kullandık.

 alinpikbarbell@gmail.com 

`Test_size=0,3` olarak ayarlandığında, şarap örneklerinin yüzde 30'u `X_test` ve `y_test`'e atanırken, örneklerin kalan yüzde 70'i sırasıyla `X_train` ve `y_train`'e atanmıştır. Sınıf etiketi dizisi `y`'nin `stratify` için bir argüman olarak sağlanması, hem eğitim hem de test veri kümelerinin orijinal veri kümesiyle aynı sınıf oranlarına sahip olmasını sağlar.

**Bir veri kümesini eğitim ve test veri kümelerine bölmek için uygun bir oran seçme**

Bir veri setini eğitim ve test veri setlerine bölüyorsak, öğrenme algoritmasının faydalanabileceği değerli bilgileri sakladığımızı aklımızda tutmalıyız. Bu nedenle, test kümesine çok fazla bilgi ayırmak istemeyiz. Bununla birlikte, test seti ne kadar küçük olursa, genelleme hatasının tahmini de o kadar yanlış olur. Bir veri kümesini eğitim ve test veri kümelerine bölmek tamamen bu dengeyi sağlamakla ilgilidir. Uygulamada, en yaygın kullanılan bölünmeler, başlangıçtaki veri kümesinin boyutuna bağlı olarak 60:40, 70:30 veya 80:20'dir. Bununla birlikte, büyük veri kümeleri için 90:10 veya 99:1 bölmeleri de yaygın ve uygundur. Örneğin, veri kümesi 100.000'den fazla eğitim örneği içeriyorsa, genelleştirme performansının iyi bir tahminini elde etmek için test için yalnızca 10.000 örneği alıkoymak iyi olabilir. Daha fazla bilgi ve örnek, `https://arxiv.org/pdf/1811.12808.pdf` adresinden ücretsiz olarak erişilebilen *Makine öğreniminde model değerlendirme, model seçimi ve algoritma seçimi* başlıklı makalemin birinci bölümünde bulunabilir. Ayrıca, model değerlendirme konusunu tekrar ele alacağız ve *Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Uygulamaları Öğrenme* başlıklı *6. Bölümde* daha ayrıntılı olarak tartışacağız.

Ayrıca, model eğitimi ve değerlendirmesinden sonra tahsis edilen test verilerini atmak yerine, modelin tahmin performansını artırabileceğinden, bir sınıflandırıcıyı tüm veri kümesi üzerinde yeniden eğitmek yaygın bir uygulamadır. Bu yaklaşım genellikle tavsiye edilmekle birlikte, örneğin veri kümesi küçükse ve test veri kümesi aykırı değerler içeriyorsa daha kötü genelleme performansına yol açabilir. Ayrıca, modeli tüm veri kümesine yeniden yerleştirdikten sonra, performansını değerlendirmek için elimizde bağımsız veri kalmaz.

# Özellikleri aynı ölçeğine getirme

**Özellik ölçeklendirme,** ön işleme hattımızda kolayca unutulabilen çok önemli bir adımdır. **Karar ağaçları** ve **rastgele ormanlar,** özellik ölçeklendirme konusunda endişelenmemize gerek olmayan çok az sayıdaki makine öğrenimi algoritmasından ikisidir. Bu algoritmalar ölçekten bağımsızdır. Ancak, **gradyan inişi** optimizasyon algoritmasını uyguladığımız *Bölüm 2, Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi*'nde gördüğümüz gibi, makine öğrenimi ve **optimizasyon** algoritmalarının çoğu, özellikler aynı ölçekte olduğunda çok daha iyi davranır.

Özellik ölçeklendirmenin önemi basit bir örnekle gösterilebilir. Bir özelliğin 1'den 10'a kadar bir ölçekte ölçüldüğü ve ikinci özelliğin sırasıyla 1'den 100.000'e kadar bir ölçekte ölçüldüğü iki özelliğimiz olduğunu varsayalım.

*Bölüm 2*'deki Adaline'deki karesel hata fonksiyonunu düşündüğümüzde, algoritmanın çoğunlukla ikinci özellikteki daha büyük hatalara göre ağırlıkları optimize etmekle meşgul olacağını söylemek mantıklıdır. Bir başka örnek de Öklid mesafe ölçüsüne sahip **k-en yakın komşu** (**KNN**) algoritmasıdır: örnekler arasında hesaplanan mesafeler ikinci özellik ekseni tarafından domine

6587191CA584E


edilecektir.

 olimpikbarbell@gmail.com 

Şimdi, farklı özellikleri aynı ölçeğe getirmek için iki yaygın yaklaşım vardır: **normalleştirme** ve **standartlaştırma**. Bu terimler genellikle farklı alanlarda oldukça gevşek bir şekilde kullanılır ve anlamları bağlamdan türetilmelidir. Çoğu zaman normalleştirme, özelliklerin [0, 1] aralığında yeniden ölçeklendirilmesi anlamına gelir ki bu da **min-max ölçeklendirmenin** özel bir durumudur. Verilerimizi normalleştirmek için basitçe şunları yapabiliriz

her bir özellik sütununa min-maks ölçeklendirmesi uygular, burada bir örneğin yeni değeri, $x^{(i)}_{norm}$, $x^{(i)}$, aşağıdaki gibi hesaplanabilir:

$$x^{(i)}_{norm} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}$$

Burada, $x^{(i)}$ belirli bir örnektir, $x_{min}$ bir özellik sütunundaki en küçük değerdir ve $x_{max}$ en büyük değerdir.

Min-max ölçeklendirme prosedürü scikit-learn'de uygulanmaktadır ve aşağıdaki şekilde kullanılabilir:

```
>> from sklearn.preprocessing import MinMaxScaler
>> mms = MinMaxScaler()
>> X_train_norm = mms.fit_transform(X_train)
>> X_test_norm = mms.transform(X_test)
```

Min-max ölçeklendirme yoluyla normalleştirme, sınırlı bir aralıkta değerlere ihtiyaç duyduğumuzda yararlı olan yaygın olarak kullanılan bir teknik olsa da, standartlaştırma, özellikle gradyan inişi gibi optimizasyon algoritmaları için birçok makine öğrenimi algoritması için daha pratik olabilir. Bunun nedeni, *Bölüm 3*'teki lojistik regresyon ve SVM gibi birçok doğrusal modelin ağırlıkları 0'a veya 0'a yakın küçük rastgele değerlere başlatmasıdır. Standartlaştırmayı kullanarak, özellik sütunlarını ortalama 0 ve standart sapma 1'de ortalarız, böylece özellik sütunları standart normal dağılımla aynı parametrelere sahip olur (sıfır ortalama ve birim varyans), bu da ağırlıkların öğrenilmesini kolaylaştırır. Ancak, standardizasyonun dağılımın şeklini değiştirmediğini ve normal dağılmayan verileri normal dağılan verilere dönüştürmediğini vurgulamalıyız. Verileri sıfır ortalama ve birim varyansa sahip olacak şekilde ölçeklendirmenin yanı sıra, standartlaştırma aykırı değerler hakkında faydalı bilgileri korur ve verileri sınırlı bir değer aralığına ölçeklendiren min-max ölçeklendirmenin aksine algoritmayı bunlara karşı daha az hassas hale getirir.

Standardizasyon prosedürü aşağıdaki denklemle ifade edilebilir:

$$x^{(i)}_{std} = \frac{x^{(i)} - \mu x}{\sigma x}$$

Burada, $\mu_x$ belirli bir özellik sütununun örnek ortalaması ve $\sigma_x$ ilgili standart sapmadır.

Aşağıdaki tablo, yaygın olarak kullanılan iki özellik ölçeklendirme tekniği olan standardizasyon ve normalizasyon arasındaki farkı, 0 ila 5 sayılarından oluşan basit bir örnek veri kümesi üzerinde göstermektedir:

| Giriş | Standartlaştırılmış | Min-maks normalleştirilmiş |
|---|---|---|
| 0.0 | -1.46385 | 0.0 |
| 1.0 | -0.87831 | 0.2 |
| 2.0 | -0.29277 | 0.4 |
| 3.0 | 0.29277 | 0.6 |
| 4.0 | 0.87831 | 0.8 |
| 5.0 | 1.46385 | 1.0 |

*Tablo 4.1: Standartlaştırma ve min-max normalleştirme arasında bir karşılaştırma*

Aşağıdaki kod örneklerini çalıştırarak tabloda gösterilen standartlaştırma ve normalleştirmeyi manuel olarak gerçekleştirebilirsiniz:

```
>> ex = np.array([0, 1, 2, 3, 4, 5])
>> print('standardized:', (ex - e x .mean()) / e x . std())
standardized: [-1.46385011 -0.87831007 - 0 . 29277002
0 . 29277002
0.8 7 8 3 1007 1.46385011]
>> print('normalized:', (ex - ex.min()) / (ex.max() - ex.min())) normalized: [
0. 0.2 0 . 4 0.6 0 . 8 1. ]
```

MinMaxScaler sınıfına benzer şekilde, scikit-learn de standartlaştırma için bir sınıf uygular:

```
>> from sklearn.preprocessing import StandardScaler
>> stdsc = StandardScaler()
>>  X_train_std = stdsc.fit_transform(X_train)
>> X_test_std = stdsc.transform(X_test)
```

 alinpikbarbell@gmail.com 

Yine, StandardScaler sınıfını eğitim verilerine yalnızca bir kez uydurduğumuzu ve bu parametreleri test veri kümesini veya herhangi bir yeni veri noktasını dönüştürmek için kullandığımızı vurgulamak da önemlidir.

Özellik ölçeklendirme için RobustScaler gibi diğer, daha gelişmiş yöntemler scikit-learn'de mevcuttur. RobustScaler, özellikle çok sayıda aykırı değer içeren küçük veri kümeleriyle çalışıyorsak yararlı olur ve önerilir. Benzer şekilde, bu veri kümesine uygulanan makine öğrenimi algoritması **aşırı uyum** sağlamaya yatkınsa, RobustScaler iyi bir seçim olabilir. Her bir özellik sütununda bağımsız olarak çalışan RobustScaler, medyan değeri kaldırır ve veri kümesini veri kümesinin 1. ve 3. çeyrek dilimine (yani sırasıyla 25. ve 75. çeyrek dilimine) göre ölçeklendirir, böylece daha uç değerler ve aykırı değerler daha az belirgin hale gelir. İlgilenen okuyucu RobustScaler hakkında daha fazla bilgiyi https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html adresindeki resmi scikit-learn belgelerinde bulabilir.

# Anlamlı özelliklerinin seçilmesi

Bir modelin eğitim veri kümesinde test veri kümesinden çok daha iyi performans gösterdiğini fark edersek, bu durum aşırı uyumun güçlü bir göstergesidir. *Bölüm 3*, *Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu*'nda tartıştığımız gibi, aşırı uyum, modelin eğitim veri kümesindeki belirli gözlemlerle ilgili olarak parametrelere çok yakın uyduğu, ancak yeni verilere iyi genelleme yapmadığı anlamına gelir; modelin **yüksek** bir **varyansa** sahip olduğunu söylüyoruz. Aşırı uyumun nedeni, modelimizin verilen eğitim verileri için çok karmaşık olmasıdır. Genelleme hatasını azaltmak için yaygın çözümler aşağıdaki gibidir:

- Daha fazla eğitim verisi toplayın
- Düzenli hale getirme yoluyla karmaşıklık için bir ceza getirin
- Daha az parametre içeren daha basit bir model seçin
- Verilerin boyutluluğunu azaltın

Daha fazla eğitim verisi toplamak genellikle uygulanabilir değildir. *Bölüm 6*, *Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Uygulamaları Öğrenme'd*e, daha fazla eğitim verisinin yararlı olup olmadığını kontrol etmek için yararlı bir teknik öğreneceğiz. İlerleyen bölümlerde, verilere daha az parametre uydurulmasını gerektirerek daha basit modellere yol açan özellik seçimi yoluyla düzenlileştirme ve boyutsallık azaltma yoluyla aşırı uyumu azaltmanın yaygın yollarını inceleyeceğiz. Daha sonra, *Bölüm 5'te, Boyutsallık Azaltma Yoluyla Verileri Sıkıştırma,* ek özellik çıkarma tekniklerine göz atacağız.

# Model karmaşıklığına karşı cezalar olarak L1 ve L2 düzenlemesi

*Bölüm 3'*ten hatırlayacağınız üzere **L2 düzenlemesi,** büyük bireysel ağırlıkları cezalandırarak bir modelin karmaşıklığını azaltmaya yönelik bir yaklaşımdır. Ağırlık vektörümüz $\boldsymbol{w}$'nin kareli L2 normunu aşağıdaki gibi tanımladık:

$$LLL\ \|ww\|_2^2 = \sum_{jj1}^{m} ww_{jj}^2$$

Model karmaşıklığını azaltmaya yönelik bir diğer yaklaşım da ilgili **L1 düzenlemesidir**:

$$L1L \; \|ww\|_1 = \sum_{jj1}^{m} |wwjj|$$

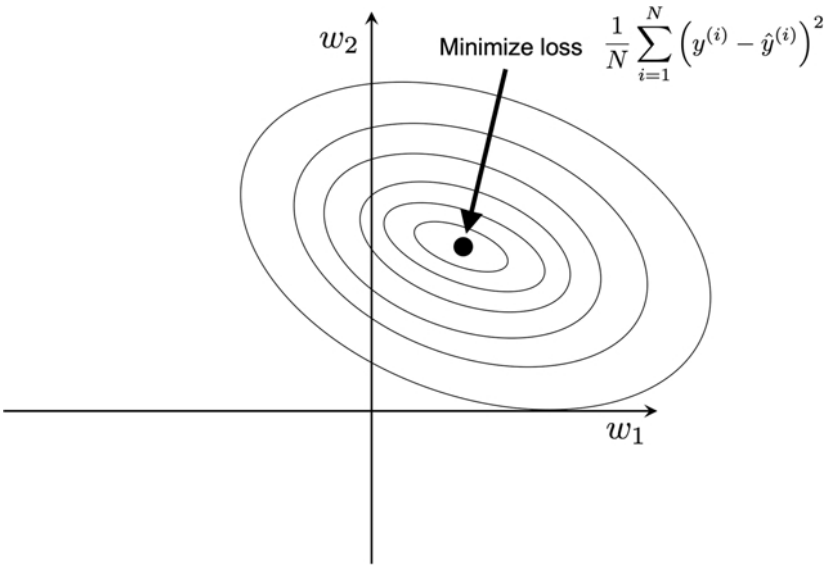
Burada, basitçe ağırlıkların karesini ağırlıkların mutlak değerlerinin toplamıyla değiştirdik. L2 düzenlileştirmenin aksine, L1 düzenlileştirme genellikle seyrek özellik vektörleri verir ve çoğu özellik ağırlığı sıfır olacaktır. Seyreklik, özellikle eğitim örneklerinden daha fazla alakasız boyuta sahip olduğumuz durumlarda, alakasız birçok özelliğe sahip yüksek boyutlu bir veri kümemiz varsa pratikte yararlı olabilir. Bu anlamda, L1 düzenlemesi özellik seçimi için bir teknik olarak anlaşılabilir.

## L2 düzenlileştirmenin geometrik bir yorumu

Önceki bölümde belirtildiği gibi, L2 düzenlemesi, kayıp fonksiyonuna bir ceza terimi ekleyerek, düzensizleştirilmiş bir kayıp fonksiyonu ile eğitilen bir modele kıyasla daha az aşırı ağırlık değerleriyle etkili bir şekilde sonuçlanır.

L1 düzenlileştirmenin seyrekliği nasıl teşvik ettiğini daha iyi anlamak için bir adım geri gidelim ve düzenlileştirmenin geometrik yorumuna bir göz atalım. $w_1$ ve $w_2$ olmak üzere iki ağırlık katsayısı için dışbükey bir kayıp fonksiyonunun konturlarını çizelim.

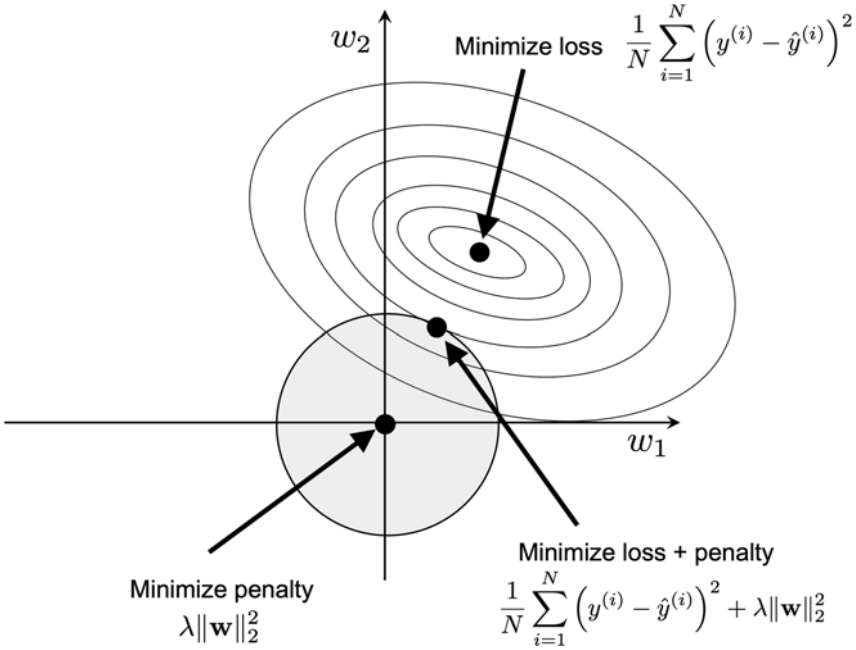
Burada, *Bölüm 2'*de Adaline için kullandığımız, eğitim setindeki tüm $N$ örneklerin ortalaması alınarak gerçek ve tahmin edilen sınıf etiketleri ($y$ ve $yy$) arasındaki **karesel** uzaklıkları hesaplayan **ortalama karesel hata (MSE)** kayıp fonksiyonunu ele alacağız. MSE küresel olduğu için lojistik regresyonun kayıp fonksiyonuna göre çizimi daha kolaydır; ancak aynı kavramlar geçerlidir. Amacımızın, *Şekil 4.5'*te gösterildiği gibi (elipslerin merkezindeki nokta) eğitim verileri için kayıp fonksiyonunu en aza indiren ağırlık katsayıları kombinasyonunu bulmak olduğunu unutmayın:




*Şekil 4.5: Ortalama karesel hata kayıp fonksiyonunun minimize edilmesi*

 alinpikbarbell@gmail.com 

Düzenliliği, daha küçük ağırlıkları teşvik etmek için kayıp fonksiyonuna bir ceza terimi eklemek olarak düşünebiliriz; başka bir deyişle, büyük ağırlıkları cezalandırırız. Böylece, düzenlileştirme parametresi $\lambda$ aracılığıyla düzenlileştirme gücünü artırarak, ağırlıkları sıfıra doğru küçültür ve modelimizin eğitim verilerine olan bağımlılığını azaltırız. Bu kavramı L2 ceza terimi için aşağıdaki şekilde gösterelim:

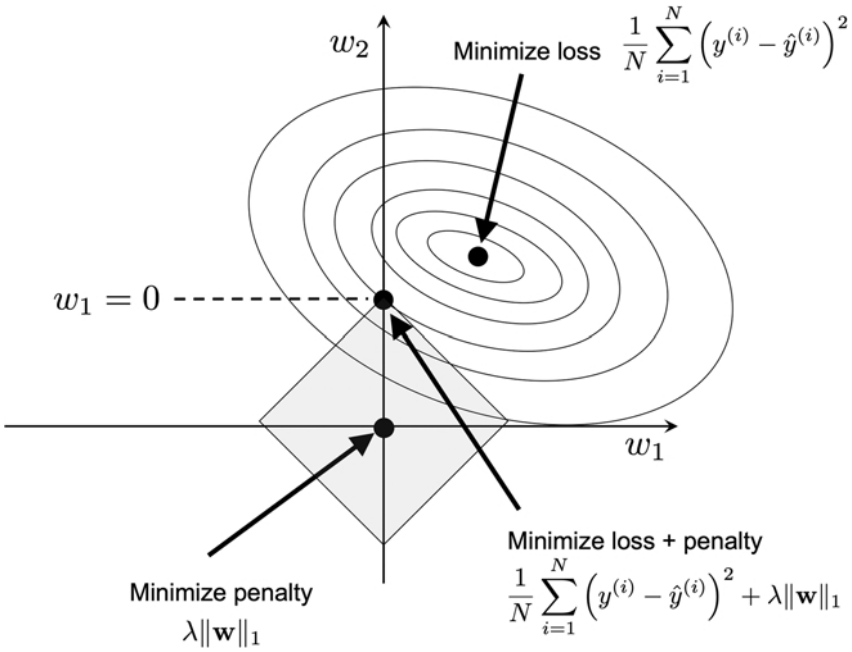



*Şekil 4.6: Kayıp fonksiyonuna L2 düzenlemesi uygulanması*

İkinci dereceden L2 düzenlileştirme terimi gölgeli top ile temsil edilir. Burada, ağırlık katsayılarımız düzenlileştirme bütçemizi aşamaz - ağırlık katsayılarının kombinasyonu gölgeli alanın dışına düşemez. Öte yandan, yine de kayıp fonksiyonunu minimize etmek istiyoruz. Ceza kısıtlaması altında, en iyi çabamız L2 topunun cezalandırılmamış kayıp fonksiyonunun konturlarıyla kesiştiği noktayı seçmektir. Düzenlilik parametresi $\lambda$ ne kadar büyük olursa, cezalandırılmış kayıp o kadar hızlı büyür ve bu da daha dar bir L2 topuna yol açar. Örneğin, düzenlilik parametresini sonsuza doğru artırırsak, ağırlık katsayıları etkin bir şekilde sıfır olur ve L2 topunun merkezi ile gösterilir. Örnekteki ana mesajı özetlemek gerekirse, amacımız cezalandırılmamış kayıp artı ceza teriminin toplamını en aza indirmektir; bu, modele uyacak yeterli eğitim verisi olmadığında varyansı azaltmak için önyargı eklemek ve daha basit bir modeli tercih etmek olarak anlaşılabilir.

## L1 düzenlemesi ile seyrek çözümler

Şimdi L1 düzenlileştirme ve seyrekliği tartışalım. L1 düzenliliğinin arkasındaki ana kavram, önceki bölümde tartıştıklarımıza benzer. Ancak, L1 cezası mutlak ağırlık katsayılarının toplamı olduğundan (L2 teriminin ikinci dereceden olduğunu unutmayın), bunu *Şekil 4.7*'de gösterildiği gibi elmas şeklinde bir bütçe olarak temsil edebiliriz:




*Şekil 4.7: Kayıp fonksiyonuna L1 düzenlemesi uygulanması*

Yukarıdaki şekilde, kayıp fonksiyonunun konturunun L1 elmasına $w_1 = 0$'da temas ettiğini görebiliriz. L1 düzenlenmiş bir sistemin konturları keskin olduğundan, optimumun - yani kayıp fonksiyonunun elipsleri ile L1 elmasının sınırı arasındaki kesişimin - eksenler üzerinde yer alması daha olasıdır, bu da seyrekliği teşvik eder.

**L1 düzenlemesi ve seyreklik**

L1 regülarizasyonunun neden seyrek çözümlere yol açabileceğinin matematiksel detayları bu kitabın kapsamı dışındadır. Eğer ilgileniyorsanız, L2 ve L1 regülarizasyonunun mükemmel bir açıklaması *Bölüm 3.4*'te bulunabilir, *The Elements of Statistical Learning*, *Trevor Hastie, Robert Tibshirani,* ve *Jerome Friedman, Springer Science+Business Media,* 2009.

 alinpikbarbell@gmail.com 

Scikit-learn'de L1 düzenlileştirmeyi destekleyen düzenlenmiş modeller için cezayı basitçe ayarlayabiliriz parametresini 'l1' olarak ayarlayarak seyrek bir çözüm elde edin:

```
>> from sklearn.linear_model import LogisticRegression
> >   LogisticRegression(penalty='l1',
...                          çözücü='liblinear',
...                          multi_class='ovr')
```

'lbfgs' şu anda L1-düzenlenmiş kayıp optimizasyonunu desteklemediğinden, farklı bir optimizasyon algoritması (örneğin, solver='liblinear') seçmemiz gerektiğini de unutmayın. Standartlaştırılmış Şarap verilerine uygulandığında, L1 düzenlenmiş lojistik regresyon aşağıdaki seyrek çözümü verecektir:

```
>> lr = LogisticRegression(penalty='l1',
...                              C=1.0,
...                              çözücü='liblinear',
...                              multi_class='ovr')
>> # C=1.0'ın varsayılan değer olduğunu unutmayın. Artırabilirsiniz
>> # veya düzenli hale getirme etkisi yaratmak için azaltın
>> # sırasıyla daha güçlü veya daha zayıf.
>> lr.fit(X_train_std, y_train)
>> print('Eğitim doğruluğu:', lr.score(X_train_std, y_train))
Eğitim doğruluğu: 1,0
>> print('Test doğruluğu:', lr.score(X_test_std, y_test))
Test doğruluğu: 1.0
```

Hem eğitim hem de test doğrulukları (her ikisi de yüzde 100) modelimizin her iki veri kümesinde de mükemmel bir iş çıkardığını göstermektedir. Kesişim terimlerine lr.intercept_ niteliği aracılığıyla eriştiğimizde, dizinin üç değer döndürdüğünü görebiliriz:

```
>> lr.intercept_
     array([-1.26317363, -1.21537306, -2.37111954])
```

LogisticRegression nesnesini çok sınıflı bir veri kümesine **bire karşı dinlenme** (**OvR**) yaklaşımıyla uydurduğumuz için, ilk kesişim değeri 1. sınıfa karşı 2. ve 3. sınıflara uyan modele aittir, ikinci değer 2. sınıfa karşı 1. ve 3. sınıflara uyan modelin kesişim değeridir ve üçüncü değer 3. sınıfa karşı 1. ve 2. sınıflara uyan modelin kesişim değeridir:

```
>> lr.coef_
array([ 1.24647953, 0 . 18050894, 0 . 74540443, -1.16301108,
          0.          ,0.          , 1.16243821, 0.          ,
          0.          , O .          , O.          , O .55620267,
          2.50890638],
        [-1.53919461, -0.38562247, -0.99565934, 0 . 36390047,
         -0.05892612, 0.          , 0.66710883, 0.          ,
          0.          , -1.9318798 , 1.23775092, 0.          ,
```

```
        -2.23280039],
       [ 0.13557571, 0.16848763, 0.35710712, 0.        ,
         0.        , 0.        , -2.43804744, 0.        ,
         0.        , 1.56388787, -0.81881015, -0.49217022,
         0.        ]])
```

lr.coef_ özniteliği aracılığıyla eriştiğimiz ağırlık dizisi, her sınıf için bir ağırlık vektörü olmak üzere üç sıra ağırlık katsayısı içerir. Her satır 13 ağırlıktan oluşur ve net girdiyi hesaplamak için her ağırlık 13 boyutlu Wine veri kümesindeki ilgili özellik ile çarpılır:

$$z = w_1 x_1 + \cdots + w_m x_m + b = \sum_{j=1}^{m} x_j w_j + b = w^T x + b$$

**Scikit-learn tahmin edicilerinin yanlılık birimi ve ağırlık parametrelerine erişim**

Scikit-learn'de intercept_ önyargı birimine, coef_ ise $w_j$ değerlerine karşılık gelir.

Daha önce de belirtildiği gibi, özellik seçimi için bir yöntem olarak hizmet eden L1 düzenlemesi sonucunda, bu veri kümesindeki potansiyel olarak alakasız özelliklere karşı dayanıklı bir model eğittik. Açıkçası, önceki örnekteki ağırlık vektörlerinin seyrek olması gerekmez çünkü sıfırdan daha fazla sıfır olmayan girdi içerirler. Bununla birlikte, düzenlileştirme gücünü daha da artırarak, yani C parametresi için daha düşük değerler seçerek seyrekliği (daha fazla sıfır girişi) zorlayabiliriz.

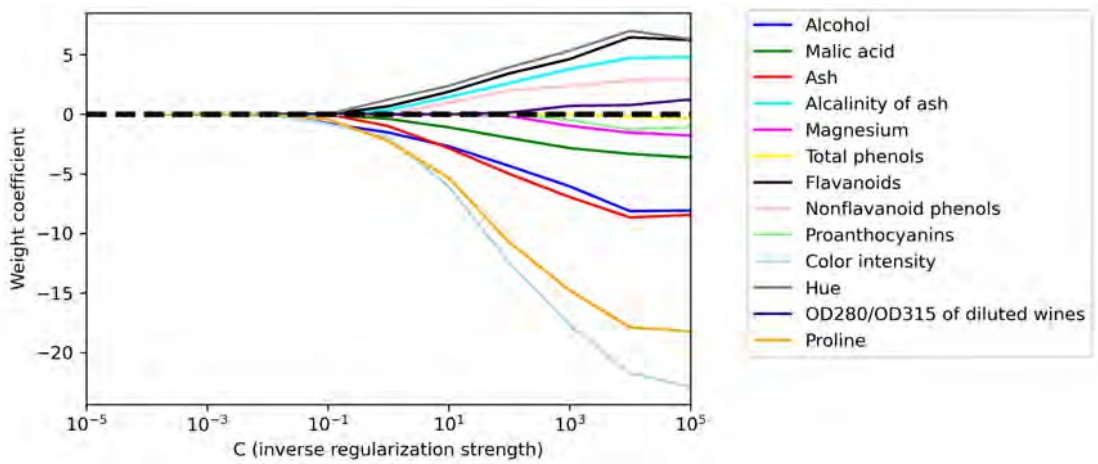
Bu bölümdeki düzenlileştirme ile ilgili son örnekte, düzenlileştirme gücünü değiştireceğiz ve farklı düzenlileştirme güçleri için düzenlileştirme yolunu - farklı özelliklerin ağırlık katsayılarını - çizeceğiz:

```
>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>> ax = plt.subplot(111)
>>> renkler = ['mavi', 'yeşil', 'kırmızı', 'camgöbeği',
...            'macenta', 'sarı', 'siyah',
...            'pembe', 'açık yeşil', 'açık mavi',
...            'gri', 'çivit mavisi', 'turuncu']
>>> weights, params = [], []
>> for c in np.arange(-4., 6.):
...     lr = LogisticRegression(penalty='l1', C=10.**c,
...                             çözücü='liblinear',
...                             multi_class='ovr', random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
```

 alinpikbarbell@gmail.com 

```
>>> ağırlıklar = np.array(ağırlıklar)
>> for sütun, renk in zip(range(weights.shape[1]), renkler):
...         plt.plot(params, weights[:, column],
...                  label=df_wine.columns[column + 1],
...                  color=color)
>> plt.axhline(0, color='black', linestyle='--', linewidth=3)
>> plt.xlim([10**(-5), 10**5])
>> plt.ylabel('Ağırlık katsayısı')
>> plt.xlabel('C (ters regülasyon gücü)')
>> plt.xscale('log')
>> plt.legend(loc='sol üst')
>> ax.legend(loc='upper center',
...                bbox_to_anchor=(1.38, 1.03),
...                ncol=1, fancybox=True)
>> plt.show()
```

Ortaya çıkan grafik bize L1 düzenlileştirmenin davranışı hakkında daha fazla bilgi sağlar. Gördüğümüz gibi, modeli güçlü bir düzenlilik parametresiyle ($C < 0{,}01$) cezalandırırsak tüm özellik ağırlıkları sıfır olacaktır; *C, düzenlilik parametresinin* tersidir, $\lambda$:




*Şekil 4.8: Düzenli hale getirme gücü hiper parametresi C'nin değerinin etkisi*

## Sıralı özellik seçim algoritmaları

Modelin karmaşıklığını azaltmanın ve aşırı uyumdan kaçınmanın alternatif bir yolu, özellikle düzensiz modeller için yararlı olan özellik seçimi yoluyla **boyut azaltmadır.** Boyut azaltma tekniklerinin iki ana kategorisi vardır: **özellik seçimi** ve **özellik çıkarma**. Özellik seçimi ile orijinal özelliklerin bir alt kümesini *seçerken, özellik çıkarımında* yeni bir özellik alt uzayı oluşturmak için özellik kümesinden bilgi *türetiriz.*

Bu bölümde, klasik bir özellik seçimi algoritması ailesine göz atacağız. Bir sonraki bölümde, *Bölüm 5, Boyut Azaltma Yoluyla Verileri Sıkıştırma,* bir veri kümesini daha düşük boyutlu bir özellik alt uzayına sıkıştırmak için farklı özellik çıkarma teknikleri hakkında bilgi edineceğiz.

Sıralı özellik seçimi algoritmaları, başlangıçtaki d *boyutlu* bir özellik uzayını $k<d$ olan k boyutlu bir özellik alt uzayına indirgemek için kullanılan bir açgözlü arama algoritmaları ailesidir. Özellik seçimi algoritmalarının arkasındaki motivasyon, problemle en alakalı özelliklerin bir alt kümesini otomatik olarak seçmek, hesaplama verimliliğini artırmak veya ilgisiz özellikleri veya gürültüyü kaldırarak modelin genelleme hatasını azaltmaktır; bu, düzenli hale getirmeyi desteklemeyen algoritmalar için yararlı olabilir.

Klasik bir sıralı özellik seçimi algoritması, hesaplama verimliliğini artırmak için sınıflandırıcının performansında minimum bir düşüşle başlangıç özellik alt uzayının boyutluluğunu azaltmayı amaçlayan sıralı **geriye doğru seçimdir** (**SBS).** Bazı durumlarda SBS, bir modelin aşırı uyumdan muzdarip olması halinde modelin tahmin gücünü bile artırabilir.

**Açgözlü arama algoritmaları**

Açgözlü algoritmalar, bir kombinatoryal arama probleminin her aşamasında yerel olarak en uygun seçimleri yapar ve tüm olası kombinasyonları değerlendiren ve en uygun çözümü bulması garanti edilen kapsamlı arama algoritmalarının aksine, genellikle probleme en uygun olmayan bir çözüm verir. Bununla birlikte, pratikte, kapsamlı bir arama genellikle hesaplama açısından uygun değildir, oysa açgözlü algoritmalar daha az karmaşık, hesaplama açısından daha verimli bir çözüme izin verir.

SBS algoritmasının arkasındaki fikir oldukça basittir: SBS, yeni özellik alt uzayı istenen sayıda özellik içerene kadar tam özellik alt kümesinden özellikleri sırayla kaldırır. Her aşamada hangi özelliğin kaldırılacağını belirlemek için, minimize etmek istediğimiz kriter fonksiyonunu ($J$) tanımlamamız gerekir.

Kriter fonksiyonu tarafından hesaplanan kriter, basitçe sınıflandırıcının belirli bir özelliğin çıkarılmasından önceki ve sonraki performans farkı olabilir. Daha sonra, her aşamada çıkarılacak özellik basitçe bu kriteri maksimize eden özellik olarak tanımlanabilir; ya da daha basit bir ifadeyle, her aşamada çıkarıldıktan sonra en az performans kaybına neden olan özelliği ortadan kaldırırız. SBS'nin yukarıdaki tanımına dayanarak, algoritmayı dört basit adımda özetleyebiliriz:

1. Algoritmayı $k = d$ ile başlatın, burada $d$ tam özellik uzayının boyutluluğudur, $\boldsymbol{X}_d$ .
2. Kriteri maksimize eden özelliği, $\boldsymbol{x}^-$ , belirleyin: $\boldsymbol{x}^- = \text{argmax } J(\boldsymbol{X}_k - \boldsymbol{x})$, burada $\boldsymbol{x} \in \boldsymbol{X}_k$ .
3. Özellik kümesinden $\boldsymbol{x}^-$ özelliğini kaldırın: $\boldsymbol{X}_{k-1} = \boldsymbol{X}_k - \boldsymbol{x}^-$ ; $k = k - 1$.
4. $k$ istenen özellik sayısına eşitse sonlandırın; aksi takdirde *adım 2*'ye gidin.

 alinpikbarbell@gmail.com

**Sıralı özellik algoritmaları üzerine bir kaynak**

*F. Ferri*, *P. Pudil*, *M. Hatef* ve *J. Kittler* tarafından yazılan *Compara- tive Study of Techniques for Large-Scale Feature Selection* (*Büyük Ölçekli Özellik Seçimi için Tekniklerin Karşılaştırmalı Çalışması*), sayfa 403-413, 1994'te çeşitli sıralı özellik algoritmalarının ayrıntılı bir değerlendirmesini bulabilirsiniz.

Kodlama becerilerimizi ve kendi algoritmalarımızı uygulama yeteneğimizi geliştirmek için, devam edelim ve bunu Python'da sıfırdan uygulayalım:

```
from sklearn.base import clone from
itertools import combinations
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

SBS sınıfı:
    def init (self, tahmin edici, k_özellikler,
                    scoring=accuracy_score,
                    test_size=0.25, random_state=1):
        self.s c o r i n g  = puanlama
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state
    def fit(self, X, y):
        X_train, X_test, y_train, y_test = \
            train_test_split(X, y, test_size=self.test_size,
                                random_state=self.random_state)

        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                                    X_test, y_test, self.indices_)
        self.scores_ = [ score] while
        dim > self.k_features:
            puanlar = [] alt
            kümeler = []

            for p in combinations(self.indices_, r=dim - 1):
                score = self._calc_score(X_train, y_train,
```

```
                                                X_test, y_test, p)
                    scores.append(score)
                    subsets.append(p)

                best = np.argmax(scores)
                self.indices_ = subsets[best]
                self.subsets_.append(self.indices_)
                dim -= 1

                self.scores_.append(scores[best])
            self.k_score_ = s e l f . s c o r e s _ [-1]

            return self

        def transform(self, X):
            return X[:, self.indices_]

        def _calc_score(self, X_train, y_train, X_test, y_test, indices):
            self.estimator.fit(X_train[:, indices], y_train)
            y_pred = self.estimator.predict(X_test[:, indices])
            score = self.scoring(y_test, y_pred)
            dönüş puanı
```

Önceki uygulamada, döndürmek istediğimiz özellik sayısını belirtmek için k_features parametresini tanımladık. Varsayılan olarak, özellik alt kümeleri üzerinde bir modelin (sınıflandırma için bir tahmin edici) performansını değerlendirmek için scikit-learn'den accuracy_score kullanırız.

Fit yönteminin while döngüsü içinde, itertools.combination işlevi tarafından oluşturulan özellik alt kümeleri değerlendirilir ve özellik alt kümesi istenen boyutluluğa sahip olana kadar azaltılır. Her iterasyonda, en iyi alt kümenin doğruluk puanı, son olarak oluşturulan test veri kümesi X_test'e dayalı olarak self.scores_ adlı bir listede toplanır. Bu puanları daha sonra sonuçları değerlendirmek için kullanacağız. Nihai özellik alt kümesinin sütun indisleri self.indices_'e atanır; bu indisleri, seçilen özellik sütunlarını içeren yeni bir veri dizisi döndürmek için transform yöntemi aracılığıyla kullanabiliriz. Kriteri fit yöntemi içinde açıkça hesaplamak yerine, en iyi performans gösteren özellik alt kümesinde yer almayan özelliği kaldırdığımızı unutmayın.

Şimdi, scikit-learn'den KNN sınıflandırıcısını kullanarak SBS uygulamamızı çalışırken görelim:

```
>> import matplotlib.pyplot as plt
>> from sklearn.neighbors import KNeighborsClassifier
>> knn = KNeighborsClassifier(n_neighbors=5)
>> sbs = SBS(knn, k_features=1)
>> sbs.fit(X_train_std, y_train)
```

 olimpikbarbell@gmail.com 

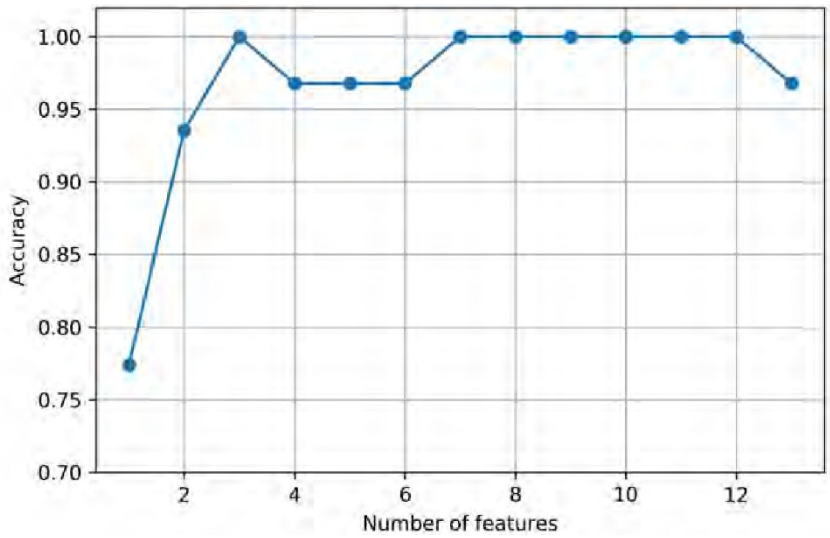
SBS uygulamamız veri setini `fit` fonksiyonu içinde test ve eğitim veri setine bölmüş olsa da, biz yine de eğitim veri seti olan `X_train`'i algoritmaya besledik. SBS `fit` yöntemi daha sonra test (doğrulama) ve eğitim için yeni eğitim alt kümeleri oluşturacaktır, bu nedenle bu test kümesine **doğrulama veri kümesi** de denir. Bu yaklaşım, *orijinal* test setimizin eğitim verilerinin bir parçası haline gelmesini önlemek için gereklidir.

SBS algoritmamızın her aşamada en iyi özellik alt kümesinin puanlarını topladığını unutmayın, bu nedenle uygulamamızın daha heyecan verici kısmına geçelim ve KNN sınıflandırıcısının doğrulama veri kümesi üzerinde hesaplanan sınıflandırma doğruluğunu çizelim. Kod aşağıdaki

```
>>> k_feat = [len(k) for k in sbs.subsets_]
>> plt.plot(k_feat, sbs.scores_, marker='o')
>> plt.ylim([0.7, 1.02])
>>  plt.ylabel('Doğruluk')
>> plt.xlabel('Özellik sayısı')
>> plt.grid()
>> plt.tight_layout()
>> plt.show()
```

gibidir:

*Şekil 4.9*'da görebileceğimiz gibi, KNN sınıflandırıcısının doğruluğu, *Bölüm 3*'te KNN algoritması bağlamında tartıştığımız **boyutluluk** lanetindeki azalmaya bağlı olarak, özellik sayısını azalttıkça doğrulama veri kümesinde iyileşmiştir. Ayrıca, aşağıdaki grafikte sınıflandırıcının $k$ = {3, 7, 8, 9, 10, 11, 12} için yüzde 100 doğruluk elde ettiğini görebiliriz:




*Şekil 4.9: Özellik sayısının model doğruluğu üzerindeki etkisi*

Kendi merakımızı gidermek için, doğrulama veri kümesinde bu kadar iyi bir performans sağlayan en küçük özellik alt kümesinin (*k=3*) neye benzediğini görelim:

```
>>> k3 = list(sbs.subsets_[10])
> >  print(df_wine.columns[1:][k3])
Index(['Alkol', 'Malik asit', ' Seyreltilmiş şarapların OD280/OD315'i'],
dtype='object')
```

Önceki kodu kullanarak, sbs.subsets_ özniteliğindeki 11. konumdan üç özellikli alt kümenin sütun indekslerini elde ettik ve pandas Wine DataFrame'in sütun indeksinden karşılık gelen özellik adlarını döndürdük.

Daha sonra, KNN sınıflandırıcısının performansını orijinal test veri kümesi üzerinde değerlendirelim:

```
>> knn.fit(X_train_std, y_train)
>> print('Eğitim doğruluğu:', knn.score(X_train_std, y_train)) Eğitim
doğruluğu: 0.967741935484
>> print('Test doğruluğu:', knn.score(X_test_std, y_test)) Test
doğruluğu: 0.962962962963
```

Önceki kod bölümünde, tüm özellik kümesini kullandık ve eğitim veri kümesinde yaklaşık yüzde 97 ve test veri kümesinde yaklaşık yüzde 96 doğruluk elde ettik; bu da modelimizin yeni verilere zaten iyi genelleme yaptığını gösterir. Şimdi, seçilen üç özellikli alt kümeyi kullanalım ve KNN'nin ne kadar iyi performans gösterdiğini görelim:

```
>> knn.fit(X_train_std[:, k3], y_train)
>> print('Eğitim doğruluğu:',
...          knn.score(X_train_std[:, k3], y_train))
Eğitim doğruluğu: 0.951612903226
>> print('Test doğruluğu:',
...          knn.score(X_test_std[:, k3], y_test))
Test doğruluğu: 0.925925925926
```

Şarap veri setindeki orijinal özelliklerin dörtte birinden daha azı kullanıldığında, test veri setindeki tahmin doğruluğu biraz düşmüştür. Bu, söz konusu üç özelliğin orijinal veri kümesinden daha az ayırt edici bilgi sağlamadığını gösterebilir. Bununla birlikte, Wine veri kümesinin küçük bir veri kümesi olduğunu ve rastgeleliğe (veri kümesini eğitim ve test alt kümelerine bölme şeklimiz ve eğitim veri kümesini bir eğitim ve doğrulama alt kümesine daha fazla bölme şeklimiz) çok duyarlı olduğunu da aklımızda tutmalıyız.

Özellik sayısını azaltarak KNN modelinin performansını artırmasak da veri kümesinin boyutunu küçülttük, bu da pahalı veri toplama adımları içerebilecek gerçek dünya uygulamalarında faydalı olabilir. Ayrıca, özellik sayısını önemli ölçüde azaltarak, yorumlanması daha kolay olan daha basit modeller elde ettik.

 alinpikbarbell@gmail.com

**Scikit-learn'de özellik seçimi algoritmaları**

Daha önce Python mlxtend paketinde uyguladığımız basit SBS ile ilgili birkaç farklı sıralı özellik seçme uygulamasını http://rasbt.github.io/mlxtend/user_guide/feature_selection/ SequentialFeatureSelector/ adresinde bulabilirsiniz. mlxtend uygulamamız birçok zil ve ıslık ile birlikte gelirken, son v0.24 sürümünün bir parçası olan basitleştirilmiş, kullanıcı dostu bir sürümü uygulamak için scikit-learn ekibiyle işbirliği yaptık. Kullanımı ve işleyişi bu bölümde uyguladığımız SBS koduna çok benzemektedir. Daha fazla bilgi edinmek isterseniz, lütfen https://scikit-learn.org/stable/ modules/generated/sklearn.feature_selection.SequentialFeatureSelector. html adresindeki belgelere bakın.

Scikit-learn aracılığıyla kullanılabilen daha birçok özellik seçme algoritması vardır. Bunlar arasında özellik ağırlıklarına dayalı özyinelemeli geriye doğru eleme, özellikleri öneme göre seçmek için ağaç tabanlı yöntemler ve tek değişkenli istatistiksel testler bulunmaktadır. Farklı özellik seçim yöntemlerinin kapsamlı bir tartışması bu kitabın kapsamı dışındadır, ancak açıklayıcı örneklerle birlikte iyi bir özet http://scikit-learn.org/stable/modules/ feature_selection.html adresinde bulunabilir.

# Rastgele ormanları ile özellik öneminin değerlendirilmesi

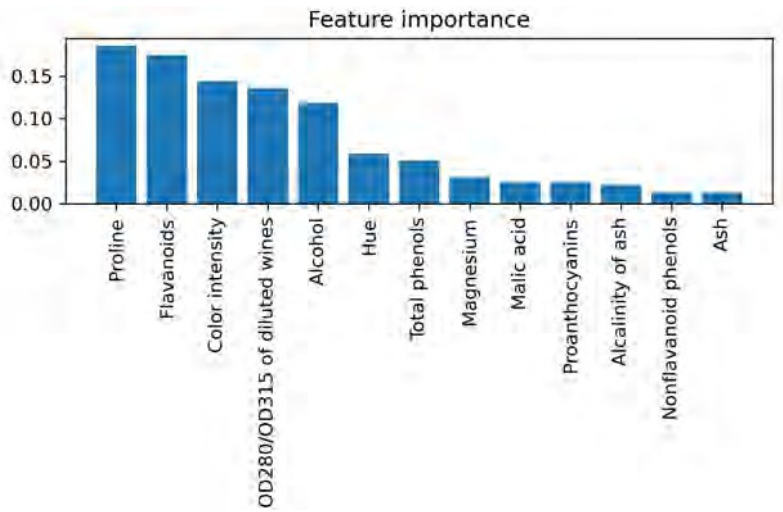
Önceki bölümlerde, lojistik regresyon yoluyla ilgisiz özellikleri sıfırlamak için L1 düzenlileştirmenin nasıl kullanılacağını ve özellik seçimi için SBS algoritmasının nasıl kullanılacağını ve bir KNN algoritmasına nasıl uygulanacağını öğrendiniz. Bir veri kümesinden ilgili özellikleri seçmek için bir başka yararlı yaklaşım da *Bölüm 3'te* tanıtılan en- semble teknik olan **rastgele orman** kullanmaktır. Rastgele orman kullanarak, verimizin doğrusal olarak ayrılabilir olup olmadığı hakkında herhangi bir varsayımda bulunmadan, özelliğin önemini ormandaki tüm karar ağaçlarından hesaplanan ortalama safsızlık azalması olarak ölçebiliriz. Uygun bir şekilde, scikit-learn'deki rastgele orman uygulaması zaten bizim için özellik önem değerlerini toplar, böylece bir RandomForestClassifier yerleştirdikten sonra feature_importances_ özelliği aracılığıyla bunlara erişebiliriz. Aşağıdaki kodu çalıştırarak, şimdi Wine veri kümesi üzerinde 500 ağaçtan oluşan bir ormanı eğiteceğiz ve 13 özelliği kendi önem ölçülerine göre sıralayacağız - *Bölüm 3*'teki tartışmamızdan ağaç tabanlı modellerde

```
>> from sklearn.ensemble import RandomForestClassifier
>> feat_labels = df_wine.columns[1:]
>>  orman = RandomForestClassifier(n_estimators=500,
...                                 random_state=1)
>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_
>>> indices = np.argsort(importances)[::-1]
>> for f in range(X_train.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[indices[f]],
```

standartlaştırılmış veya normalleştirilmiş özellikler kullanmamız gerekmediğini hatırlayın:

```
...                                    importances[indices[f]]))
>> plt.title('Özellik önemi')
> >  plt.bar(range(X_train.shape[1]),
...            importances[indices],
...            align='center')
> >   plt.xticks(range(X_train.shape[1]),
...                 feat_labels[indices], rotation=90)
>> plt.xlim([-1, X_train.shape[1]])
>> plt.tight_layout()
>> plt.show()
 1) Prolin                          0.185453
 2) Flavanoidler                    0.174751
 3) Renk yoğunluğu                  0.143920
 4) Seyreltilmiş şarapların         0.136162
 OD280/OD315'i
 5) Alkol                           0.118529
 6) Hue                             0.058739
 7) Toplam fenoller                 0.050872
 8) Magnezyum                       0.031357
 9) Malik asit                      0.025648
 10) Proantosiyaninler              0.025570
 11) Külün alkalinitesi             0.022366
 12) Nonflavanoid fenoller          0.013354
 13) Kül                            0.013279
```

Kodu çalıştırdıktan sonra, Wine veri kümesindeki farklı özellikleri göreceli önemlerine göre sıralayan bir çizim oluşturduk; özellik önem değerlerinin 1,0'a toplanacak şekilde normalleştirildiğine dikkat edin:




*Şekil 4.10: Şarap veri setinin rastgele orman tabanlı özellik önemi*

 alinpikbarbell@gmail.com 

Prolin ve flavonoid seviyeleri, renk yoğunluğu, OD280/OD315 kırınımı ve şarabın alkol konsantrasyonunun, 500 karar ağacındaki ortalama safsızlık azalmasına dayalı olarak veri kümesindeki en ayırt edici özellikler olduğu sonucuna varabiliriz. İlginç bir şekilde, grafikteki en üst sıradaki özelliklerden ikisi aynı zamanda önceki bölümde uyguladığımız SBS algoritmasının üç özellikli alt küme seçiminde de yer almaktadır (alkol konsantrasyonu ve seyreltilmiş şarapların OD280/OD315'i).

Bununla birlikte, yorumlanabilirlik söz konusu olduğunda, rastgele orman tekniği bahsetmeye değer önemli bir *sorunla* birlikte gelir. Eğer iki ya da daha fazla özellik yüksek oranda korelasyon gösteriyorsa, bir özellik çok üst sıralarda yer alırken diğer özellik(ler) hakkındaki bilgi tam olarak yakalanamayabilir. Öte yandan, özellik önem değerlerinin yorumlanmasından ziyade sadece bir modelin tahmin performansıyla ilgileniyorsak bu sorunla ilgilenmemize gerek yoktur.

Özellik önem değerleri ve rastgele ormanlar hakkındaki bu bölümü sonlandırmak için, scikit-learn'in model uydurma işleminden sonra kullanıcı tarafından belirlenen bir eşiğe dayalı olarak özellikleri seçen bir SelectFromModel nesnesi de uyguladığını belirtmek gerekir; bu, RandomForestClassifier'ı bir özellik seçici ve bir scikit-learn Pipeline nesnesinde ara adım olarak kullanmak istiyorsak kullanışlıdır; bu, *Bölüm 6, Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Öğrenme Uygulamalarında* göreceğiniz gibi, farklı ön işleme adımlarını bir tahmin ediciye bağlamamıza olanak tanır. Örneğin, aşağıdaki kodu kullanarak veri kümesini en önemli beş özelliğe indirgemek için eşiği 0,1 olarak ayarlayabiliriz:

```
>> from sklearn.feature_selection import SelectFromModel
>>> sfm = SelectFromModel(forest, threshold=0.1, prefit=True)
>> X_selected = sfm.transform(X_train)
>> print('Bu eşiği karşılayan özelliklerin sayısı',
...          'criterion:', X_selected.shape[1])
Bu eşik kriterini karşılayan özellik sayısı: 5
>> for f in range(X_selected.shape[1]):
...       print("%2d) %-*s %f" % (f + 1, 30,
...                                 feat_labels[indices[f]],
...                                 importances[indices[f]]))

 1) Prolin                          0.185453
 2) Flavanoidler                    0.174751
 3) Renk yoğunluğu                  0.143920
 4) Seyreltilmiş şarapların         0.136162
OD280/OD315'i
 5) Alkol                           0.118529
```

6587191CA584E


# Özet

Bu bölüme eksik verileri doğru şekilde ele aldığımızdan emin olmak için faydalı tekniklere bakarak başladık. Verileri bir makine öğrenimi algoritmasına beslemeden önce, kategorik değişkenleri doğru şekilde kodladığımızdan da emin olmalıyız ve bu bölümde, sıralı ve nominal özellik değerlerini tamsayı temsillerine nasıl eşleyebileceğimizi gördük.

Ayrıca, bir modelin karmaşıklığını azaltarak aşırı uyumdan kaçınmamıza yardımcı olabilecek L1 düzenlileştirmeyi kısaca tartıştık. Alakasız özellikleri kaldırmaya alternatif bir yaklaşım olarak, bir veri kümesinden anlamlı özellikleri seçmek için sıralı bir özellik seçme algoritması kullandık.

Bir sonraki bölümde, boyut azaltmaya yönelik bir başka faydalı yaklaşımı öğreneceksiniz: özellik çıkarma. Özellik seçiminde olduğu gibi özellikleri tamamen kaldırmak yerine, özellikleri daha düşük boyutlu bir alt uzaya sıkıştırmamızı sağlar.

# Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.com

olimpikbarbell@gmail.com

# 5

# Boyut Azaltma Yoluyla Veri Sıkıştırma

*Bölüm 4*, *İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme*'de, farklı özellik seçimi teknikleri kullanarak bir veri kümesinin boyutluluğunu azaltmaya yönelik farklı yaklaşımlar hakkında bilgi edindiniz. Boyut azaltma için özellik seçimine alternatif bir yaklaşım da **özellik çıkarımıdır**. Bu bölümde, bir veri kümesinin bilgi içeriğini orijinalinden daha düşük boyutlu yeni bir özellik alt uzayına dönüştürerek özetlemenize yardımcı olacak iki temel teknik hakkında bilgi edineceksiniz. Veri sıkıştırma, makine öğreniminde önemli bir konudur ve modern teknoloji çağında üretilen ve toplanan artan miktarda veriyi depolamamıza ve analiz etmemize yardımcı olur.

Bu bölümde aşağıdaki konuları ele alacağız:

- Denetimsiz veri sıkıştırma için temel bileşen analizi
- Sınıf ayrılabilirliğini en üst düzeye çıkarmak için denetimli bir boyut azaltma tekniği olarak doğrusal diskriminant analizi
- Veri görselleştirme için doğrusal olmayan boyut azaltma tekniklerine ve t-dağılımlı stokastik komşu gömme işlemine kısa bir bakış

## temel bileşen analizi aracılığıyla denetimsiz boyut azaltma

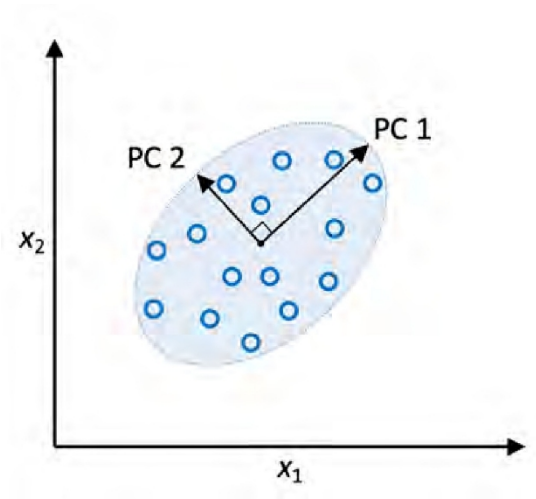
Özellik seçimine benzer şekilde, bir veri kümesindeki özelliklerin sayısını azaltmak için farklı özellik çıkarma teknikleri kullanabiliriz. Özellik seçimi ve özellik çıkarımı arasındaki fark, **sıralı arka koğuş seçimi** gibi özellik seçimi algoritmalarını kullandığımızda orijinal özellikleri korurken, verileri yeni bir özellik uzayına dönüştürmek veya yansıtmak için özellik çıkarımını kullanmamızdır.

Boyutsallık azaltma bağlamında, özellik çıkarma, ilgili bilgilerin çoğunu korumak amacıyla veri sıkıştırmaya yönelik bir yaklaşım olarak anlaşılabilir. Uygulamada, özellik çıkarma yalnızca depolama alanını veya öğrenme algoritmasının hesaplama verimliliğini artırmak için kullanılmaz, aynı zamanda **boyut lanetini** azaltarak tahmin performansını da artırabilir **- özellikle de** düzenli olmayan modellerle çalışıyorsak.

## Temel bileşen analizindeki ana adımlar

Bu bölümde, özellikle özellik çıkarma ve boyut azaltma için farklı alanlarda yaygın olarak kullanılan denetimsiz bir doğrusal dönüştürme tekniği olan **temel bileşen analizini** (**PCA**) tartışacağız. PCA'nın diğer popüler uygulamaları arasında keşifsel veri analizi ve borsa ticaretinde sinyallerin denoize edilmesi ve biyoinformatik alanında genom verilerinin ve gen ifade seviyelerinin analizi yer almaktadır.

PCA, özellikler arasındaki korelasyona dayalı olarak verilerdeki örüntüleri tanımlamamıza yardımcı olur. Özetle, PCA yüksek boyutlu verilerde maksimum varyans yönlerini bulmayı amaçlar ve verileri orijinaline eşit veya daha az boyutlu yeni bir alt uzaya yansıtır. Yeni alt uzayın ortogonal eksenleri (temel bileşenler), *Şekil 5.1*'de gösterildiği gibi, yeni özellik eksenlerinin birbirine ortogonal olduğu kısıtlaması göz önüne alındığında maksimum varyans yönleri olarak yorumlanabilir:




*Şekil 5.1: Bir veri kümesindeki maksimum varyans yönlerini bulmak için PCA kullanımı*

*Şekil 5.1*'de $x_1$ ve $x_2$ orijinal özellik eksenleri, **PC 1** ve **PC 2** ise temel bileşenlerdir. Boyut azaltma

için PCA kullanırsak, d×k *boyutlu* bir dönüşüm matrisi oluştururuz,
***W***, eğitim örneğinin özelliklerinin bir vektörünü, ***x,*** orijinal d-boyutlu özellik uzayından daha az boyuta sahip yeni bir *k-boyutlu* özellik alt uzayına eşlememizi sağlar. Örneğin, süreç aşağıdaki gibidir. Bir özellik vektörümüz olduğunu varsayalım, ***x***:

$$xx\,x\,[_{x1},\,_{x2},\ldots,\,_{xd}],\,xx\,x\quad \boldsymbol{x}^{\;d}$$

daha sonra bir dönüşüm matrisi olan ***WW W*** $^{Wdddd}$ ile dönüştürülür:

$$\boldsymbol{xW} = z$$

sonuçta çıktı vektörü elde edilir:

$$z\,z\,[z1,\,_{z2},\ldots,\,_{zkk}],\,z\,z\,z^{kk}$$

 olimpikbarbell@gmail.com 

Orijinal d *boyutlu* verilerin bu yeni k *boyutlu* alt uzaya (tipik olarak $k << d$) dönüştürülmesinin bir sonucu olarak, ilk temel bileşen mümkün olan en büyük varyansa sahip olacaktır. Ardından gelen tüm temel bileşenler, bu bileşenlerin diğer temel bileşenlerle ilişkisiz (ortogonal) olduğu kısıtlaması göz önüne alındığında en büyük varyansa sahip olacaktır - girdi özellikleri ilişkili olsa bile, ortaya çıkan temel bileşenler karşılıklı olarak ortogonal (ilişkisiz) olacaktır. PCA yönlerinin veri ölçeklendirmesine karşı oldukça hassas olduğunu ve özellikler farklı ölçeklerde ölçüldüyse ve tüm özelliklere eşit önem atamak istiyorsak PCA'dan önce özellikleri standartlaştırmamız gerektiğini unutmayın.

Boyut azaltma için PCA algoritmasına daha detaylı bakmadan önce, yaklaşımı birkaç basit adımda özetleyelim:

1. d-boyutlu veri kümesini standartlaştırın.
2. Kovaryans matrisini oluşturun.
3. Kovaryans matrisini özvektörlerine ve özdeğerlerine ayrıştırın.
4. Karşılık gelen özvektörleri sıralamak için özdeğerleri azalan sıraya göre sıralayın.
5. En büyük *k* özdeğere karşılık gelen *k* özvektör seçin; burada *k,* yeni özellik altuzayının (*k kd*) boyut-alitesidir.
6. "En iyi" *k* öz vektörden bir projeksiyon matrisi, ***W***, oluşturun.
7. Yeni *k-boyutlu* özellik alt uzayını elde etmek için projeksiyon matrisi ***W***'yi kullanarak d-boyutlu giriş veri kümesi ***X***'i dönüştürün.

İlerleyen bölümlerde, bir öğrenme alıştırması olarak Python kullanarak adım adım bir PCA gerçekleştireceğiz. Daha sonra, scikit-learn kullanarak nasıl daha kolay bir PCA gerçekleştirebileceğimizi göreceğiz.

**Eigendecomposition: Bir Matrisin Özvektörlere ve Özdeğerlere Ayrıştırılması**

**Özdeğer** ayrıştırması, bir kare matrisin **özdeğerler** ve **özdeğerler** olarak adlandırılan **özvektörler**, bu bölümde açıklanan PCA prosedürünün merkezinde yer almaktadır.

Kovaryans matrisi kare matrisin özel bir durumudur: simetrik bir matristir, bu da matrisin transpozesine eşit olduğu anlamına gelir, $A = A^T$ .

Böyle bir simetrik matrisi ayrıştırdığımızda, özdeğerler gerçek (karmaşık değil) sayılardır ve özvektörler birbirlerine ortogonaldir (diktir). Ayrıca, özdeğerler ve özvektörler çiftler halinde gelir. Bir kovaryans matrisini özvektörlerine ve özdeğerlerine ayırırsak, en yüksek özdeğer ile ilişkili özvektörler veri kümesindeki maksimum varyans yönüne karşılık gelir. Burada, bu "yön" veri kümesinin özellik sütunlarının doğrusal bir dönüşümüdür.

Özdeğerler ve özvektörler hakkında daha ayrıntılı bir tartışma bu kitabın kapsamı dışında olmakla birlikte, ek kaynaklara işaret eden nispeten kapsamlı bir inceleme Wikipedia'da https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors adresinde bulunabilir.

## Temel bileşenlerin adım adım çıkarılması

Bu alt bölümde, bir PCA'nın ilk dört adımını ele alacağız:

1. Verilerin standartlaştırılması
2. Kovaryans matrisinin oluşturulması
3. Kovaryans matrisinin özdeğerlerinin ve özvektörlerinin elde edilmesi
4. Özvektörleri sıralamak için özdeğerleri azalan sıraya göre sıralama

İlk olarak, *Bölüm 4, İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme*'de çalıştığımız Wine veri kümesini yükleyerek başlayacağız:

```
>> import pandas as pd
>> df_wine = pd.read_csv(
...         'https://archive.ics.uci.edu/ml/'
...         'machine-learning-databases/wine/wine.data',
...         header=None
... )
```

**Şarap veri setinin elde edilmesi**

Wine veri setinin (ve bu kitapta kullanılan diğer tüm veri setlerinin) bir kopyasını bu kitabın kod paketinde bulabilirsiniz; bu kopyayı ofis dışında çalışıyorsanız veya https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data adresindeki UCI sunucusu geçici olarak kullanılamıyorsa kullanabilirsiniz. Örneğin, Wine veri setini yerel bir dizinden yüklemek için aşağıdaki satırları değiştirebilirsiniz:

```
df = pd.read_csv(
        'https://archive.ics.uci.edu/ml/'
        'machine-learning-databases/wine/wine.data',
        header=None
)
```

bunlarla:

```
df = pd.read_csv(
        'your/local/path/to/wine.data',
        header=None
)
```

 olimpikbarbell@gmail.com 

Daha sonra, Şarap verilerini ayrı eğitim ve test veri kümeleri olarak işleyeceğiz - sırasıyla verilerin yüzde 70'ini ve yüzde 30'unu kullanarak - ve birim varyansa göre standartlaştıracağız:

```
>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y, test_size=0.3,
...                      stratify=y,
...                      random_state=0)
>> # özellikleri standartlaştırın
>> from sklearn.preprocessing import StandardScaler
>> sc = StandardScaler()
>> X_train_std = sc.fit_transform(X_train)
>> X_test_std = sc.transform(X_test)
```

Önceki kodu çalıştırarak zorunlu ön işlemeyi tamamladıktan sonra, ikinci adıma geçelim: kovaryans matrisini oluşturmak. Simetrik *d×d boyutlu kovaryans matrisi,* burada *d* veri kümesindeki boyut sayısıdır, farklı özellikler arasındaki ikili kovaryansları saklar. Örneğin, iki özellik arasındaki kovaryans, $x_j$ ve $x_k$ , popülasyon düzeyinde aşağıdaki denklemle hesaplanabilir:

$$\sigma_{jj} = \frac{1}{n-1} \sum_{i=1}^{n} (x_j^{(i)} - \mu_j)(x_j^{(i)} - \mu_j)$$

Burada, $\mu_{jj}$ ve $\mu_{kk}$ sırasıyla *j* ve *k* özelliklerinin örnek ortalamalarıdır. Veri setini standartlaştırdığımızda örnek ortalamalarının sıfır olduğunu unutmayın. İki özellik arasındaki pozitif kovaryans, özelliklerin birlikte arttığını veya azaldığını gösterirken, negatif kovaryans özelliklerin zıt yönlerde değiştiğini gösterir. Örneğin, üç özelliğin kovaryans matrisi aşağıdaki gibi yazılabilir (Σ'n i n Yunanca büyük harf sigma olduğuna ve toplama sembolü ile karıştırılmaması gerektiğine dikkat edin):

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

Kovaryans matrisinin özvektörleri temel bileşenleri (maksimum varyansın yönleri) temsil ederken, karşılık gelen özdeğerler bunların büyüklüğünü tanımlayacaktır. Şarap veri seti söz konusu olduğunda, 13×13 boyutlu kovaryans matrisinden 13 özvektör ve özdeğer elde ederiz.

Şimdi, üçüncü adımımız için, kovaryans matrisinin öz çiftlerini elde edelim. Eğer lineer cebir dersi aldıysanız, bir özvektörün, $\boldsymbol{v}$, aşağıdaki koşulu sağladığını öğrenmiş olabilirsiniz:

$$\Sigma v \; \boldsymbol{v} \; \lambda v$$

Burada, $\lambda$ bir skalerdir: özdeğer. Özvektörlerin ve özdeğerlerin manuel olarak hesaplanması biraz sıkıcı ve ayrıntılı bir iş olduğundan, Şarap kovaryans matrisinin özçiftlerini elde etmek için NumPy'den `linalg.eig` işlevini kullanacağız:

```
>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>> print('\nEigenvalues \n', eigen_vals) Özdeğerler


[ 4.84274532   2.41602459   1.5 4 8 4 5 8 2 5          0.84166161
                            0.96120438
  0.6620634    0.51828472   0.3 4 6 5 0377 0.3131368    0.10754642
  0.21357215   0.15362835   0.1808613 ]
```

numpy.cov fonksiyonunu kullanarak, standartlaştırılmış eğitim veri setinin kovaryans matrisini hesapladık. `Linalg.eig` fonksiyonunu kullanarak, 13 özdeğerden oluşan bir vektör (`eigen_vals`) ve 13×13 boyutlu bir matriste (`eigen_vecs`) sütunlar olarak depolanan karşılık gelen özvektörleri veren eigendecomposition işlemini gerçekleştirdik.

**NumPy'de Eigendecomposition**

numpy.linalg.eig fonksiyonu hem simetrik hem de simetrik olmayan kare matrisler üzerinde çalışmak üzere tasarlanmıştır. Ancak, bazı durumlarda karmaşık özdeğerler döndürdüğünü görebilirsiniz.

İlgili bir fonksiyon olan numpy.linalg.eigh, kovaryans matrisi gibi simetrik matrislerle çalışmak için sayısal olarak daha kararlı bir yaklaşım olan Hermetian matrislerini ayrıştırmak için uygulanmıştır; numpy.linalg.eigh her zaman gerçek özdeğerler döndürür.

## Toplam ve açıklanan varyans

Veri setimizi yeni bir özellik alt uzayına sıkıştırarak boyutluluğunu azaltmak istediğimizden, yalnızca bilginin (varyansın) çoğunu içeren özvektörlerin (temel bileşenler) alt kümesini seçiyoruz. Özdeğerler, özvektörlerin büyüklüğünü tanımlar, bu nedenle özdeğerleri azalan büyüklüğe göre sıralamamız gerekir; karşılık gelen özdeğerlerinin değerlerine göre en iyi $k$ özvektörle ilgileniyoruz. Ancak en bilgilendirici $k$ özvektörü toplamadan önce, özdeğerlerin **açıklanan varyans oranlarını** çizelim. Bir özdeğerin açıklanan varyans oranı,
$_{jj}\lambda$ , basitçe bir özdeğerin, $\lambda_{jj}$ , ve özdeğerlerin toplamının kesridir:

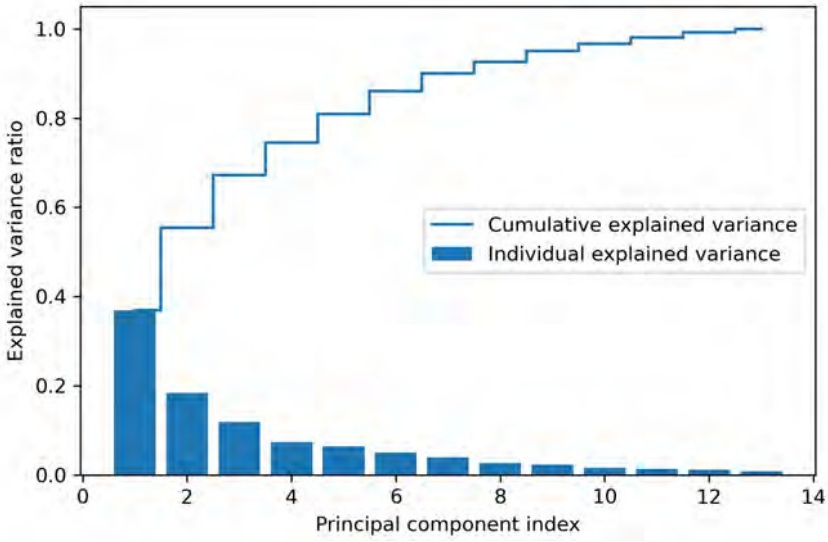
$$\text{Açıklanan varyans oranı} = \frac{\lambda_{jj}}{\sum_{j=1}^{d} \lambda_{jj}}$$

NumPy cumsum fonksiyonunu kullanarak, daha sonra Matplotlib'in step fonksiyonu aracılığıyla çizeceğimiz açıklanan varyansların kümülatif toplamını hesaplayabiliriz:

```
>>> tot = sum(eigen_vals)
>> var_exp = [(i / tot) for i in
...                 sorted(eigen_vals, reverse=True)]
>> cum_var_exp = np.cumsum(var_exp)
>> import matplotlib.pyplot as plt
>> plt.bar(range(1,14), var_exp, align='center',
...             label='Bireysel açıklanan varyans')
>> plt.step(range(1,14), cum_var_exp, where='mid',
...              label='Kümülatif açıklanmış varyans')
>> plt.ylabel('Açıklanan varyans oranı')
>> plt.xlabel('Temel bileşen endeksi')
>>  plt.legend(loc='best')
>> plt.tight_layout()
>> plt.show()
```

Ortaya çıkan grafik, birinci temel bileşenin tek başına varyansın yaklaşık yüzde 40'ını açıkladığını göstermektedir.

Ayrıca, ilk iki temel bileşenin veri setindeki varyansın neredeyse yüzde 60'ını açıkladığını görebiliriz:




*Şekil 5.2: Temel bileşenler tarafından yakalanan toplam varyans oranı*

Açıklanan varyans grafiği bize *Bölüm 4*, *İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme'd*e rastgele ormanlar aracılığıyla hesapladığımız özellik önem değerlerini hatırlatsa da, PCA'nın denetimsiz bir yöntem olduğunu, yani sınıf etiketleri hakkındaki bilgilerin göz ardı edildiğini kendimize hatırlatmalıyız. Rastgele orman, düğüm safsızlıklarını hesaplamak için sınıf üyeliği bilgisini kullanırken, varyans bir özellik ekseni boyunca değerlerin yayılımını ölçer.

## Özellik dönüşümü

Kovaryans matrisini başarılı bir şekilde öz çiftlere ayrıştırdığımıza göre, Şarap veri setini yeni temel bileşen eksenlerine dönüştürmek için son üç adıma geçelim. Bu bölümde ele alacağımız temel adımlar şunlardır:

1. En büyük *k* özdeğere karşılık gelen *k* özvektör seçin; burada *k,* yeni özellik altuzayının (*k kd*) boyut-alitesidir.
2. "En iyi" *k* öz vektörden bir projeksiyon matrisi, ***W***, oluşturun.
3. Yeni *k-boyutlu* özellik alt uzayını elde etmek için projeksiyon matrisi ***W***'yi kullanarak d-boyutlu giriş veri kümesi ***X***'i dönüştürün.

Ya da daha az teknik bir ifadeyle, öz çiftleri öz değerlerin azalan sırasına göre sıralayacağız, seçilen öz vektörlerden bir projeksiyon matrisi oluşturacağız ve verileri daha düşük boyutlu alt uzaya dönüştürmek için projeksiyon matrisini kullanacağız.

Öz değerlerin azalan sırasına göre öz çiftleri sıralayarak başlıyoruz:

```
>> # (özdeğer, özvektör) çiftlerinin bir listesini oluşturun
>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
...                    for i in range(len(eigen_vals))]
>> # (özdeğer, özvektör) tuple'larını yüksekten düşüğe doğru sıralayın
>> eigen_pairs.sort(key=lambda k: k[0], reverse=True)
```

Daha sonra, bu veri setindeki varyansın yaklaşık yüzde 60'ını yakalamak için en büyük iki özdeğere karşılık gelen iki özvektörü topluyoruz. Bu alt bölümün ilerleyen kısımlarında verileri iki boyutlu bir dağılım grafiği ile çizeceğimiz için iki özvektörün gösterim amacıyla seçildiğini unutmayın. Uygulamada, temel bileşenlerin sayısı, hesaplama verimliliği ile sınıflandırıcının performansı arasındaki dengeye göre belirlenmelidir:

```
>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                    eigen_pairs[1][1][:, np.newaxis]))
>> print('Matris W:\n', w) Matris
W:

[[-0.13724218  0.50303478]
 [ 0.24724326  0.16487119]
 [-0.02545159  0.24456476]
 [ 0.20694508 -0.11352904]
 [-0.15436582  0.28974518]
```

 olimpikbarbell@gmail.com 

```
 [-0.39376952  0.05080104]
 [-0.41735106 -0.02287338]
 [ 0.30572896  0.09048885]
 [-0.30668347  0.00835233]
 [ 0.07554066  0.54977581]
 [-0.32613263 -0.20716433]
 [-0.36861022 -0.24902536]
 [-0.29669651  0.38022942]]
```

Önceki kodu çalıştırarak, en üstteki iki özvektörden 13×2 boyutlu bir projeksiyon matrisi, ***W***, oluşturduk.

**Aynalı projeksiyonlar**

NumPy ve LAPACK'in hangi sürümlerini kullandığınıza bağlı olarak, ***W*** matrisini işaretleri ters çevrilmiş olarak elde edebilirsiniz. Lütfen bunun bir sorun olmadığını unutmayın; ***v*** bir matrisin özvektörü ise, Σ, sahip oluruz:

$$\Sigma v \; \boldsymbol{v} \; \lambda v$$

Burada, ***v özvektördür*** ve -v ***de*** bir özvektördür, bunu aşağıdaki şekilde gösterebiliriz. Temel cebir kullanarak, denklemin her iki tarafını bir skaler olan $\alpha$ ile çarpabiliriz:

$$\alpha\alpha v \; \alpha \; \alpha\lambda v$$

Matris çarpımı skaler çarpım için birleşik olduğundan, bunu aşağıdaki şekilde yeniden düzenleyebiliriz:

$$\Sigma(\alpha\alpha\alpha) = \lambda\lambda(\alpha\alpha\alpha)$$

Şimdi, $\alpha v$'nin hem $\alpha \; \alpha$ 1 hem de $\alpha$ 1 için aynı özdeğere, $\lambda$, sahip bir özvektör olduğunu görebiliriz.

$\alpha \; \alpha \; \alpha$1. Dolayısıyla, hem ***v*** hem de ***-v*** özvektörlerdir.

Projeksiyon matrisini kullanarak, artık bir örneği, ***x'i*** (13 boyutlu bir satır vektörü olarak temsil edilir) PCA altuzayına (temel bileşenler bir ve iki) dönüştürebiliriz ve şimdi iki yeni özellikten oluşan iki boyutlu bir örnek vektörü olan ***x'*** elde edebiliriz:

$$x' = \boldsymbol{xW}$$

```
>> X_train_std[0].dot(w)
array([ 2.38299011, 0 .45458499])
```

Benzer şekilde, 124×13 boyutlu eğitim veri setinin tamamını matris nokta çarpımını hesaplayarak iki temel bileşene dönüştürebiliriz:

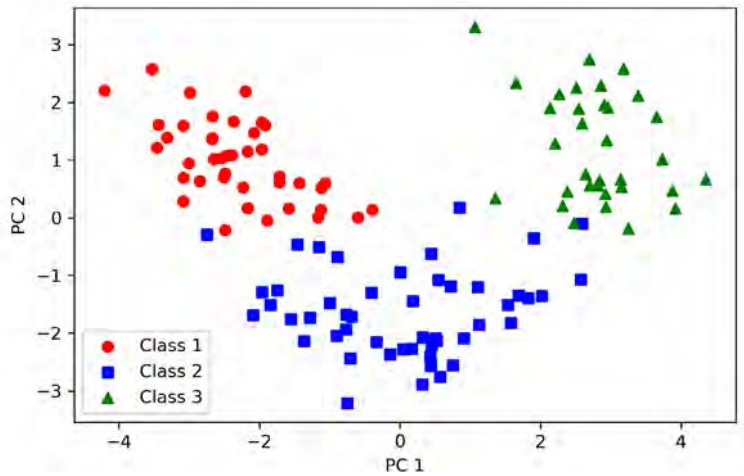
$$X' = \boldsymbol{XW}$$

```
>> X_train_pca = X_train_std.dot(w)
```

Son olarak, şimdi 124×2 boyutlu bir matris olarak depolanan dönüştürülmüş Wine eğitim veri setini iki boyutlu bir dağılım grafiğinde görselleştirelim:

```
>>> renkler = ['r', 'b', 'g']
>>> işaretleyiciler = ['o', 's', '^']
>> for l, c, m in zip(np.unique(y_train), colors, markers):
...         plt.scatter(X_train_pca[y_train==l, 0],
...                     X_train_pca[y_train==l, 1],
...                     c=c, label=f'Class {l}', marker=m)
>> plt.xlabel('PC 1')
>> plt.ylabel('PC 2')
>> plt.legend(loc='sol alt')
>> plt.tight_layout()
>> plt.show()
```

*Şekil 5.3*'te görebileceğimiz gibi, veriler birinci temel bileşen (*x* ekseni) boyunca ikinci temel bileşenden (*y* ekseni) daha fazla yayılmıştır; bu da önceki alt bölümde oluşturduğumuz açıklanan varyans oranı grafiğiyle tutarlıdır. Bununla birlikte, doğrusal bir sınıflandırıcının sınıfları muhtemelen iyi bir şekilde ayırabileceğini söyleyebiliriz:




*Şekil 5.3: Şarap veri setindeki veri kayıtlarının PCA aracılığıyla 2 boyutlu bir özellik uzayına yansıtılması*

Önceki dağılım grafiğinde örnekleme amacıyla sınıf etiketi bilgisini kodlamış olsak da, PCA'nın herhangi bir sınıf etiketi bilgisi kullanmayan denetimsiz bir teknik olduğunu unutmamalıyız.

 olimpikbarbell@gmail.com 

## Scikit-learn'de temel bileşen analizi

Önceki alt bölümdeki ayrıntılı yaklaşım PCA'nın iç işleyişini takip etmemize yardımcı olsa da, şimdi scikit-learn'de uygulanan PCA sınıfının nasıl kullanılacağını tartışacağız.

PCA sınıfı, scikit-learn'in dönüştürücü sınıflarından bir diğeridir; bu sınıfla, aynı model parametrelerini kullanarak hem eğitim verilerini hem de test veri kümesini dönüştürmeden önce ilk olarak eğitim verilerini kullanarak modele uyum sağlarız. Şimdi, scikit-learn'in PCA sınıfını Wine eğitim veri kümesinde kullanalım, dönüştürülmüş örnekleri lojistik regresyon ile sınıflandıralım ve *Bölüm 2, Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi*'nde tanımladığımız plot_decision_regions fonksiyonu ile karar bölgelerini

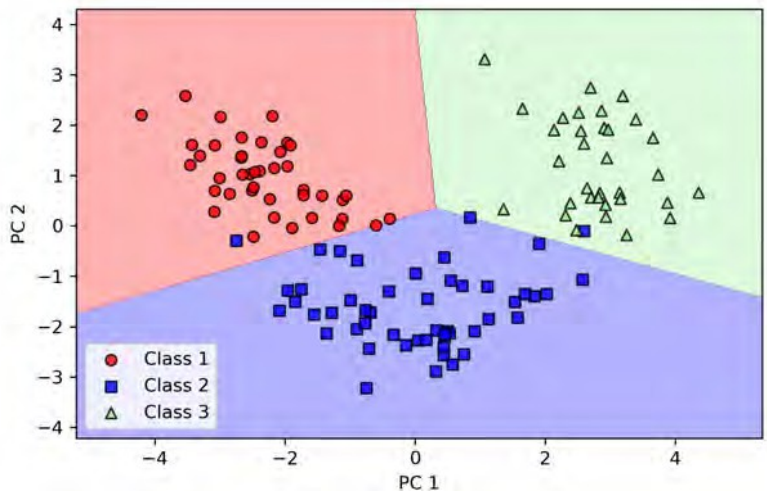
```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, sınıflandırıcı, test_idx=None, çözünürlük=0.02):

    # işaretleyici oluşturucuyu ve renk haritasını ayarlayın
    işaretleyiciler = ('o', 's', '^', 'v', '<')
    colors = ('red', ' blue', ' lightgreen', ' gray', ' cyan') cmap
    = ListedColormap(colors[:len(np.unique(y))])

    # karar yüzeyini çizin
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, çözünürlük),
                           np.arange(x2_min, x2_max, çözünürlük))
    lab = classifier.predict(np.array([xx1.ravel(), x x 2 . ravel()]).T) lab
    = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot sınıfı örnekleri
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')
```

görselleştirelim:

Size kolaylık sağlamak için, önceki plot_decision_regions kodunu geçerli çalışma dizininizdeki ayrı bir kod dosyasına, örneğin plot_decision_regions_script.py, yerleştirebilir ve geçerli Python oturumunuza aktarabilirsiniz:

```
>> from sklearn.linear_model import LogisticRegression
>>  from sklearn.decomposition import PCA
>> # PCA transformatörünün başlatılması ve
>> # lojistik regresyon tahmincisi:
>> pca = PCA(n_components=2)
>> lr = LogisticRegression(multi_class='ovr',
...                                 random_state=1,
...                                 çözücü='lbfgs')
>> # boyutsallık azaltma:
>>  X_train_pca = pca.fit_transform(X_train_std)
>> X_test_pca = pca.transform(X_test_std)
>> # lojistik regresyon modelinin indirgenmiş veri kümesine uydurulması:
>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train, classifier=lr)
>> plt.xlabel('PC 1')
>> plt.ylabel('PC 2')
>> plt.legend(loc='sol alt')
>> plt.tight_layout()
>> plt.show()
```

Bu kodu çalıştırdığımızda, artık eğitim verileri için karar bölgelerinin iki temel bileşen eksenine indirgendiğini görmeliyiz:




*Şekil 5.4: Boyut indirgeme için scikit-learn'in PCA'sını kullanan eğitim örnekleri ve lojistik regresyon karar bölgeleri*

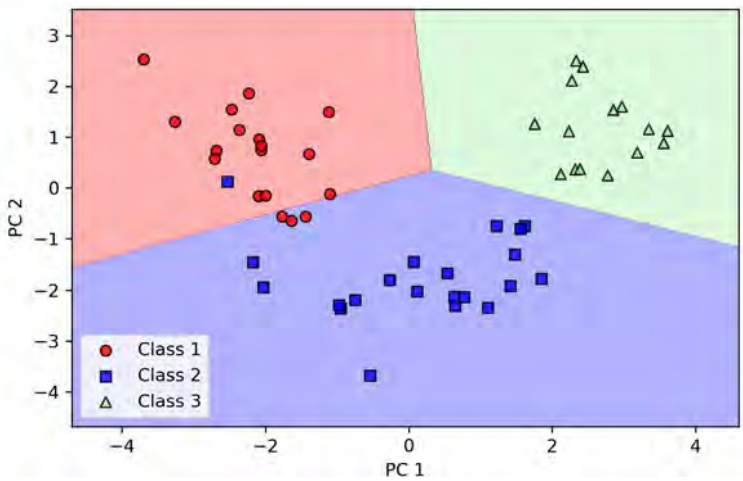
olimpikbarbell@gmail.com

Scikit-learn aracılığıyla PCA projeksiyonlarını kendi PCA uygulamamızla karşılaştırdığımızda, ortaya çıkan grafiklerin birbirinin ayna görüntüsü olduğunu görebiliriz. Bunun bu iki uygulamadaki bir hatadan kaynaklanmadığını unutmayın; bu farkın nedeni, öz çözücüye bağlı olarak, öz vektörlerin negatif veya pozitif işaretlere sahip olabilmesidir.

Önemli olduğundan değil, ancak istersek verileri -1 ile çarparak ayna görüntüsünü tersine çevirebiliriz; özvektörlerin tipik olarak birim uzunluk 1'e ölçeklendirildiğini unutmayın. Bütünlük adına, sınıfları iyi ayırıp ayıramadığını görmek için dönüştürülmüş test veri kümesi üzerinde lojistik regresyonun karar bölgelerini çizelim:

```
>>> plot_decision_regions(X_test_pca, y_test, classifier=lr)
>> plt.xlabel('PC 1')
>> plt.ylabel('PC 2')
>> plt.legend(loc='sol alt')
>> plt.tight_layout()
>> plt.show()
```

Önceki kodu çalıştırarak test veri kümesi için karar bölgelerini çizdikten sonra, lojistik regresyonun bu küçük iki boyutlu özellik alt uzayında oldukça iyi performans gösterdiğini ve test veri kümesinde yalnızca birkaç örneği yanlış sınıflandırdığını görebiliriz:




*Şekil 5.5: PCA tabanlı özellik uzayında lojistik regresyon karar bölgeleri ile test veri noktaları*

Farklı temel bileşenlerin açıklanan varyans oranlarıyla ilgileniyorsak, PCA sınıfını n_components parametresi None olarak ayarlanmış şekilde başlatabiliriz, böylece tüm temel bileşenler tutulur ve açıklanan varyans oranına explained_variance_ratio_ niteliği aracılığıyla erişilebilir:

```
>>> pca = PCA(n_components=None)
>> X_train_pca = pca.fit_transform(X_train_std)
>> pca.explained_variance_ratio_
array([ 0.36951469, 0.18434927, 0.11815159, 0.07334252,
```

```
        0.06422108, 0 . 05051724, 0 . 03954654, 0 . 02643918,
        0.02389319, 0 . 01629614, 0 . 01380021, 0 . 01172226,
        0.00820609])
```

PCA sınıfını başlattığımızda n_components=None olarak ayarladığımıza dikkat edin, böylece boyutsallık azaltma gerçekleştirmek yerine tüm temel bileşenleri sıralı bir düzende döndürür.

## Özellik katkılarının değerlendirilmesi

Bu bölümde, orijinal özelliklerin temel bileşenlere katkılarını nasıl değerlendirebileceğimize kısaca göz atacağız. Öğrendiğimiz gibi, PCA aracılığıyla, özelliklerin doğrusal kombinasyonlarını temsil eden temel bileşenler oluşturuyoruz. Bazen, her bir orijinal özelliğin belirli bir temel bileşene ne kadar katkıda bulunduğunu bilmek isteriz. Bu katkılar genellikle **yük olarak** adlandırılır.

Faktör yükleri, özvektörlerin özdeğerlerin karekökü ile ölçeklendirilmesiyle hesaplanabilir. Elde edilen değerler daha sonra orijinal özellikler ile temel bileşen arasındaki korelasyon olarak yorumlanabilir. Bunu göstermek için, ilk temel bileşen için yükleri çizelim.

İlk olarak, özvektörleri özdeğerlerin karekökü ile çarparak 13×13 boyutlu yük matrisini hesaplıyoruz:

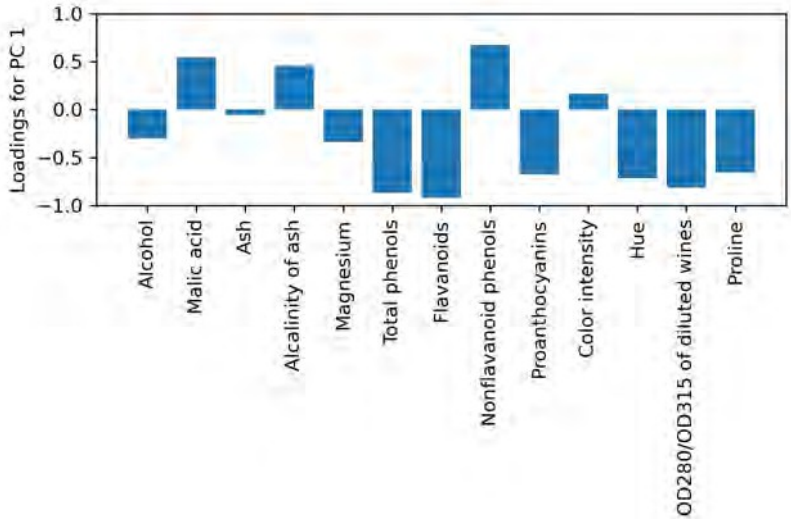
```
>>> yükler = eigen_vecs * np.sqrt(eigen_vals)
```

Ardından, bu matristeki ilk sütun olan ilk temel bileşenin yüklerini, loadings[:, 0], çiziyoruz:

```
>> fig, ax = plt.subplots()
>> ax.bar(range(13), loadings[:, 0], align='center')
>> ax.set_ylabel('PC 1 için yükler')
>>  ax.set_xticks(range(13))
>>  ax.set_xticklabels(df_wine.columns[1:], rotation=90)
>>  plt.ylim([-1, 1])
>> plt.tight_layout()
>> plt.show()
```

*Şekil 5.6'*da, örneğin **Alkolün** birinci temel bileşenle negatif bir korelasyona (yaklaşık -0,3), **Malik asidin** ise pozitif bir korelasyona (yaklaşık 0,54) sahip olduğunu görebiliriz. 1 değerinin mükemmel bir pozitif korelasyonu tanımladığını, -1 değerinin ise mükemmel bir negatif korelasyona karşılık geldiğini unutmayın:

 olimpikbarbell@gmail.com 




*Şekil 5.6: Birinci temel bileşen ile özellik korelasyonları*

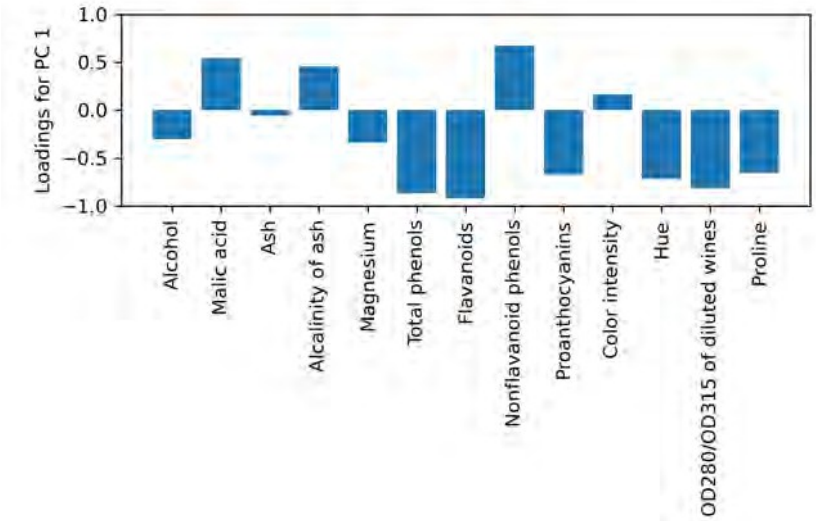
Önceki kod örneğinde, kendi PCA uygulamamız için faktör yüklerini hesaplıyoruz. Yükleri, pca.components_ öğesinin özvektörleri ve pca.explained_variance_ öğesinin özdeğerleri temsil ettiği uygun bir scikit-learn PCA nesnesinden benzer bir şekilde elde edebiliriz:

```
>> sklearn_loadings = p c a .components_.T * np.sqrt(pca.explained_variance_)
```

Scikit-learn PCA yüklerini daha önce oluşturduklarımızla karşılaştırmak için benzer bir çubuk grafiği oluşturalım:

```
>> fig, ax = plt.subplots()
>> ax.bar(range(13), sklearn_loadings[:, 0], align='center')
>> ax.set_ylabel('PC 1 için yükler')
>> ax.set_xticks(range(13))
>> ax.set_xticklabels(df_wine.columns[1:], rotation=90)
>> plt.ylim([-1, 1])
>> plt.tight_layout()
>> plt.show()
```

Gördüğümüz gibi, çubuk grafikler aynı görünüyor:




*Şekil 5.7: scikit-learn kullanarak ilk temel bileşenle özellik korelasyonları*

PCA'yı denetimsiz bir özellik çıkarma tekniği olarak inceledikten sonra, bir sonraki bölümde sınıf etiketi bilgisini dikkate alan **doğrusal** bir dönüşüm tekniği olan doğrusal **diskriminant analizi** (**LDA**) tanıtılacaktır.

# Doğrusal diskriminant yoluyla denetimli veri sıkıştırma analizi

LDA, hesaplama verimliliğini artırmak ve düzenli olmayan modellerde boyutluluk laneti nedeniyle aşırı uyum derecesini azaltmak için özellik çıkarma tekniği olarak kullanılabilir. LDA'nın arkasındaki genel kavram PCA'ya çok benzer, ancak PCA bir veri kümesinde maksimum varyansın ortogonal bileşen eksenlerini bulmaya çalışırken, LDA'daki amaç sınıf ayrılabilirliğini optimize eden özellik alt uzayını bulmaktır. İlerleyen bölümlerde, LDA ve PCA arasındaki benzerlikleri daha ayrıntılı olarak tartışacağız ve LDA yaklaşımını adım adım inceleyeceğiz.

## Temel bileşen analizine karşı doğrusal diskriminant analizi

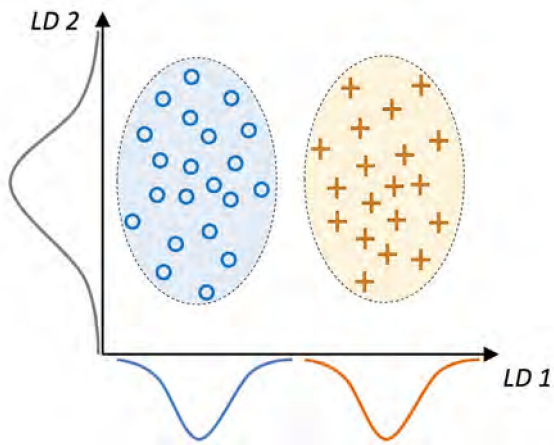
Hem PCA hem de LDA, bir veri kümesindeki boyut sayısını azaltmak için kullanılabilen *doğrusal dönüşüm* tekniklerdir; birincisi denetimsiz bir algoritma iken, ikincisi denetimlidir. Bu nedenle, LDA'nın sınıflandırma görevleri için PCA'ya kıyasla daha üstün bir özellik çıkarma tekniği olduğunu düşünebiliriz. Bununla birlikte, A.M. Martinez, PCA ile ön işlemenin belirli durumlarda, örneğin her sınıfın yalnızca az sayıda örnekten oluşması durumunda, bir görüntü tanıma görevinde daha iyi sınıflandırma sonuçları verme eğiliminde olduğunu bildirmiştir (*PCA Versus LDA* by *A. M. Martinez* and *A. C. Kak*, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 23(2): 228-233, 2001).

olimpikbarbell@gmail.com

**Fisher LDA**

LDA bazen **Fisher'ın LDA'sı olarak** da adlandırılır. Ronald A. Fisher ilk olarak 1936'da iki sınıflı sınıflandırma problemleri için Fisher'ın *Doğrusal Ayrıştırıcısını* formüle etmiştir (*The Use of Multiple Mea- surements in Taxonomic Problems*, *R. A. Fisher*, *Annals of Eugenics*, 7(2): 179-188, 1936). 1948'de Fisher'in doğrusal diskriminantı, C. Radhakrishna Rao tarafından eşit sınıf kovaryansları ve normal dağılımlı s ı n ı f l a r varsayımı altında çok sınıflı problemler için genelleştirildi ve şimdi buna LDA diyoruz (*The Utilization of Multiple Measurements in Problems of Biological Classification* by *C. R. Rao*, *Journal of the Royal Statistical Society*. Seri B (Metodolojik), 10(2): 159-203, 1948).

*Şekil 5.8,* iki sınıflı bir problem için LDA kavramını özetlemektedir. Sınıf 1'den örnekler daireler ve sınıf 2'den örnekler çarpı işaretleri ile gösterilmiştir:




*Şekil 5.8: İki sınıflı bir problem için LDA kavramı*

*X* ekseninde (*LD 1*) gösterildiği gibi doğrusal bir diskriminant iki normal dağılımlı sınıfı iyi bir şekilde ayıracaktır. *Y* ekseninde gösterilen örnek doğrusal diskriminant (*LD 2*) veri kümesindeki varyansın çoğunu yakalamasına rağmen, sınıf ayırt edici bilgilerin hiçbirini yakalayamadığı için iyi bir doğrusal diskriminant olarak başarısız olacaktır.

LDA'daki varsayımlardan biri verilerin normal dağıldığıdır. Ayrıca, sınıfların aynı kovaryans matrislerine sahip olduğunu ve eğitim örneklerinin istatistiksel olarak birbirinden bağımsız olduğunu varsayarız. Bununla birlikte, bu varsayımlardan biri veya daha fazlası (hafifçe) ihlal edilse bile, boyut azaltma için LDA hala makul ölçüde iyi çalışabilir (*Pattern Classification 2nd Edition* by *R. O. Duda, P. E. Hart*, and *D. G. Stork*, *New York*, 2001).

## Doğrusal diskriminant analizinin iç işleyişi

Kod uygulamasına geçmeden önce, LDA'yı gerçekleştirmek için gereken ana adımları kısaca özetleyelim:

1. d-boyutlu veri kümesini standartlaştırın (*d* özellik sayısıdır).
2. Her sınıf için d-boyutlu ortalama vektörü hesaplayın.
3. Sınıflar arası dağılım matrisini, $\boldsymbol{S}_B$ , ve sınıf içi dağılım matrisini, $\boldsymbol{S}_W$ oluşturun.
4. Matrisin özvektörlerini ve karşılık gelen özdeğerlerini hesaplayın, $S^{-1}\ S_B$ .W
5. Karşılık gelen özvektörleri sıralamak için özdeğerleri azalan sıraya göre sıralayın.
6. Bir d×k-di- mensional dönüşüm matrisi $\boldsymbol{W}$ oluşturmak için en büyük *k* özdeğere karşılık gelen *k* özvektörü seçin; özvektörler bu matrisin sütunlarıdır.
7. $\boldsymbol{W}$ dönüşüm matrisini kullanarak örnekleri yeni özellik altuzayına yansıtın.

Gördüğümüz gibi LDA, matrisleri yeni düşük boyutlu özellik uzayını oluşturacak olan özdeğerlere ve özvektörlere ayrıştırma anlamında PCA'ya oldukça benzemektedir. Ancak, daha önce de belirtildiği gibi, LDA, *2. adımda* hesaplanan ortalama vektörler şeklinde temsil edilen sınıf etiketi bilgisini dikkate alır. İlerleyen bölümlerde, bu yedi adımı açıklayıcı kod uygulamaları eşliğinde daha ayrıntılı olarak tartışacağız.

## Dağılım matrislerinin hesaplanması

Bu bölümün başındaki PCA bölümünde Şarap veri setinin özelliklerini zaten standartlaştırdığımız için, ilk adımı atlayabilir ve sırasıyla sınıf içi dağılım matrisini ve sınıflar arası dağılım matrisini oluşturmak için kullanacağımız ortalama vektörlerin hesaplanmasıyla devam edebiliriz. Her ortalama vektör, $\boldsymbol{m}_i$ , *i* sınıfının örneklerine göre ortalama özellik değerini, $\mu_m$ , saklar:

$$mi = \frac{1}{n_i} \sum_{xxxDi} xxm$$

Bu da üç ortalama vektörle sonuçlanır:

$$mi = \begin{bmatrix} \mu iialcohol \\ \mu iimalic\ acid \\ \vdots \\ \mu iiproline \end{bmatrix}^T \quad i\ \ i \in \{1i2i3\}$$

Bu ortalama vektörler, üç etiketin her biri için bir ortalama vektör hesapladığımız aşağıdaki kodla hesaplanabilir:

```
>>  np.set_printoptions(precision=4)
>>> mean_vecs = []
>> for label in range(1,4):
...         mean_vecs.append(np.mean(
...                     X_train_std[y_train==label], axis=0))
...         print(f'MV {label}: {mean_vecs[label - 1]}\n')
```

 olimpikbarbell@gmail.com 

```
MV 1: [ 0,9066    - 0 . 3497 0.3201 - 0 . 7189 0 . 5056 0.8807 0 . 9589 -0.5516
0.5416 0 .2338    0.5897 0 . 6563 1 . 2075]
MV 2: [-0.8749    - 0 . 2848 -0.3735 0.3157 - 0 . 3848 - 0 . 0433 0 . 0635 -0.0946
0.0703 -0.8286     0.3144 0 . 3608 -0.7253]
MV 3: [ 0.1992 0 . 866 0 . 1682 0 . 4148 -0.0451 -1.0286 - 1 . 2876 0 . 8287
-0.7795 0 . 9649 -1.209 -1.3622 -0.4013]
```

Ortalama vektörleri kullanarak artık sınıf içi dağılım matrisini hesaplayabiliriz, $\boldsymbol{S}_W$ :

$$S_W = \sum_{i=1}^{c} S_i$$

Bu, her bir $i$ sınıfının bireysel dağılım matrislerinin, $\boldsymbol{S}_i$ , toplanmasıyla hesaplanır:

$$S_i = \sum_{\boldsymbol{x} \in D_i} (\boldsymbol{x} - \boldsymbol{m}_i)(\boldsymbol{x} - \boldsymbol{m}_i)^T$$

```
>> d = 13 # özellik sayısı
>>> S_W = np.zeros((d, d))
>> for label, mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.zeros((d, d))
...     for row in X_train_std[y_train == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (satır - mv).dot((satır - mv).T)
...     S_W += class_scatter
>> print('Sınıf içi dağılım matrisi: '
...         f'{S_W.shape[0]}x{S_W.shape[1]}')
Sınıf içi dağılım matrisi: 13x13
```

Dağılım matrislerini hesaplarken yaptığımız varsayım, eğitim veri kümesindeki sınıf etiketlerinin eşit dağıldığıdır. Ancak, sınıf etiketlerinin sayısını yazdırırsak, bu varsayımın ihlal edildiğini görürüz:

```
>> print('Sınıf etiket dağılımı:',
...         np.bincount(y_train)[1:])
Sınıf etiketi dağılımı: [41 50 33]
```

Bu nedenle, bireysel dağılım matrislerini, $\boldsymbol{S}_i$ , dağılım matrisi, $\boldsymbol{S}_W$ olarak toplamadan önce ölçeklendirmek istiyoruz. Dağılım matrislerini sınıf-örnek sayısına, $n_i$ , böldüğümüzde, dağılım matrisini birleştirmenin aslında kovaryans matrisini, $\Sigma_i$ hesaplamakla aynı olduğunu görebiliriz - kovaryans matrisi dağılım matrisinin normalleştirilmiş bir versiyonudur:

$$\Sigma_i = \frac{1}{n_i} S_i = \frac{1}{n_i} \sum_{\boldsymbol{x} \in D_i} (\boldsymbol{x} - \boldsymbol{m}_i)(\boldsymbol{x} - \boldsymbol{m}_i)^T$$

Ölçeklendirilmiş sınıf içi dağılım matrisini hesaplamak için kullanılan kod aşağıdaki gibidir:

```
>> d = 13 # özellik sayısı
>>> S_W = np.zeros((d, d))
>> for label,mv in zip(range(1, 4), mean_vecs):
...         class_scatter = np.cov(X_train_std[y_train==label].T)
...         S_W += class_scatter
>> print('Ölçeklendirilmiş sınıf içi dağılım matrisi: '
...             f'{S_W.shape[0]}x{S_W.shape[1]}')
Ölçeklendirilmiş sınıf içi dağılım matrisi:
13x13
```

Ölçeklendirilmiş sınıf içi dağılım matrisini (veya kovaryans matrisini) hesapladıktan sonra, bir sonraki adıma geçebilir ve sınıflar arası dağılım matrisini ***S hesaplayabiliriz:***$_B$

$$_{SB} = \sum_{ii1}^{c} {}_{ni}(_{mi} - m)(mi - m)^T$$

Burada ***m,*** tüm ***c*** sınıflarından örnekler dahil olmak üzere hesaplanan genel ortalamadır:

```
>> mean_overall = np.mean(X_train_std, axis=0)
>> mean_overall = mean_overall.reshape(d, 1)

>> d = 13 # özellik sayısı
>>> S_B = np.zeros((d, d))
>> for i, mean_vec in enumerate(mean_vecs):
...         n = X_train_std[y_train == i + 1, :].shape[0]
...         mean_vec = mean_vec.reshape(d, 1) # sütun vektörü yap
...         S_B += n * (mean_vec - mean_overall).dot(
...         (mean_vec - mean_overall).T)
>> print('Sınıflar arası dağılım matrisi: '
...             f'{S_B.shape[0]}x{S_B.shape[1]}')
Sınıflar arası dağılım matrisi: 13x13
```

## Yeni özellik alt uzayı için doğrusal ayırıcıların seçilmesi

LDA'nın geri kalan adımları PCA'nın adımlarına benzer. Ancak, kovaryans matrisi üzerinde özdeğer ayrıştırması yapmak yerine, matrisin genelleştirilmiş özdeğer problemini çözüyoruz, $S_W^{-1} S_B$ :

```
>> eigen_vals, eigen_vecs =\
...         np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

Özçiftleri hesapladıktan sonra, özdeğerleri azalan sırada sıralayabiliriz:

```
>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...                     for i in range(len(eigen_vals))]
```

 olimpikbarbell@gmail.com 

```
>> eigen_pairs = sorted(eigen_pairs,
...                     key=lambda k: k[0], reverse=True)
>> print('Azalan sırada özdeğerler:\n')
>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])
Azalan sırada özdeğerler:
349.617808906
172.76152219
3.78531345125e-14
2.11739844822e-14
1.51646188942e-14
1.51646188942e-14
1.35795671405e-14
1.35795671405e-14
7.58776037165e-15
5.90603998447e-15
5.90603998447e-15
2.25644197857e-15
0.0
```

LDA'da doğrusal ayırıcıların sayısı en fazla $c$ - 1'dir, burada $c$ sınıf etiketlerinin sayısıdır, çünkü aradaki dağılım matrisi, $\boldsymbol{S}_B$ , rütbesi bir veya daha az olan $c$ matrislerinin toplamıdır. Gerçekten de sadece iki sıfır olmayan özdeğere sahip olduğumuzu görebiliriz (3-13 özdeğerleri tam olarak sıfır değildir, ancak bu NumPy'deki kayan nokta aritmetiğinden kaynaklanmaktadır).

**Eşdoğrusallık**

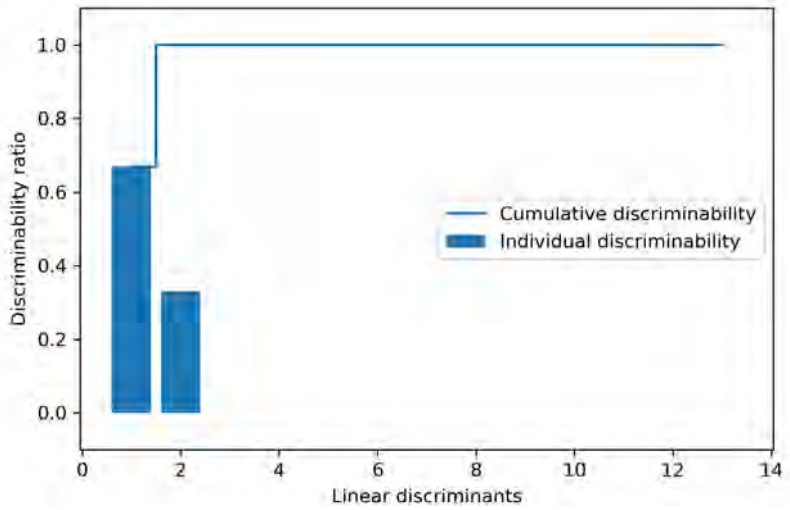
Nadir görülen mükemmel eşdoğrusallık durumunda (tüm hizalanmış örnek noktalar düz bir çizgi üzerine düşer), kovaryans matrisinin rütbesinin bir olacağını ve bunun da sıfır olmayan bir özdeğere sahip yalnızca bir özvektörle sonuçlanacağını unutmayın.

Sınıf-ayırıcı bilginin ne kadarının doğrusal ayırıcılar (özvektörler) tarafından yakalandığını ölçmek için, PCA bölümünde oluşturduğumuz açıklanan varyans grafiğine benzer şekilde, doğrusal ayırıcıları azalan özdeğerlere göre çizelim. Basit olması açısından, sınıf ayrıştırıcı bilginin içeriğini **ayrıştırılabilirlik** olarak adlandıracağız:

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real,
...                                    reverse=True)]
>>> cum_discr = np.cumsum(discr)
>> plt.bar(range(1, 14), discr, align='center',
...         label='Bireysel ayırt edilebilirlik')
```

```
>> plt.step(range(1, 14), cum_discr, where='mid',
...                 label='Kümülatif ayırt edilebilirlik')
>> plt.ylabel('"Ayrıştırılabilirlik" oranı')
>> plt.xlabel('Doğrusal Diskriminantlar')
>> plt.ylim([-0.1, 1.1])
>> plt.legend(loc='best')
>> plt.tight_layout()
>> plt.show()
```

*Şekil 5.9'*da görebileceğimiz gibi, ilk iki doğrusal ayırıcı tek başına Wine eğitim veri kümesindeki yararlı bilgilerin yüzde 100'ünü yakalamaktadır:




*Şekil 5.9: İlk iki diskriminant yararlı bilgilerin yüzde 100'ünü yakalar*

Şimdi dönüşüm matrisini oluşturmak için en ayırt edici iki özvektör sütununu üst üste koyalım, ***W***:

```
>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                 eigen_pairs[1][1][:, np.newaxis].real))
>> print('Matris W:\n', w) Matris
W:
 [[-0.1481   -0.4092]
  [ 0.0908   -0.1577]
  [-0.0168   -0.3537]
  [ 0.1484    0.3223]
  [-0.0163   -0.0817]
  [ 0.1913    0.0842]
  [-0.7338    0.2823]
  [-0.075    -0.0102]
```

 olimpikbarbell@gmail.com 

```
 [ 0.0018    0.0907]
 [ 0.294    -0.2152]
 [-0.0328    0.2747]
 [-0.3547   -0.0124]
 [-0.3915   -0.5958]]
```

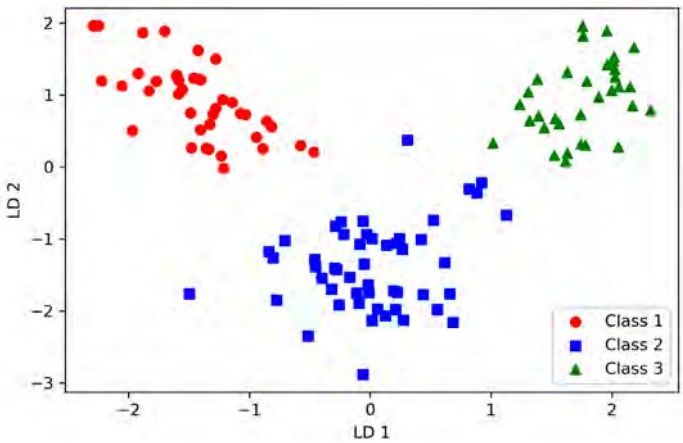
## Örneklerin yeni özellik uzayına yansıtılması

Önceki alt bölümde oluşturduğumuz $\boldsymbol{W}$ dönüşüm matrisini kullanarak, matrisleri çarpmak suretiyle eğitim veri setini dönüştürebiliriz:

$$X' = \boldsymbol{XW}$$

```
>> X_train_lda = X_train_std.dot(w)
>>> renkler = ['r', 'b', 'g']
>>> işaretleyiciler = ['o', 's', '^']
>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0],
...                 X_train_lda[y_train==l, 1] * (-1),
...                 c=c, label= f'Class {l}', marker=m)
>> plt.xlabel('LD 1')
>> plt.ylabel('LD 2')
>> plt.legend(loc='sağ alt')
>> plt.tight_layout()
>> plt.show()
```

*Şekil 5.10'da* görebileceğimiz gibi, üç Şarap sınıfı artık yeni özellik alt uzayında mükemmel şekilde doğrusal olarak ayrılabilir:




*Şekil 5.10: Verilerin ilk iki ayırıcıya yansıtılmasıyla şarap sınıfları mükemmel şekilde ayrılabilir*

## Scikit-learn aracılığıyla LDA

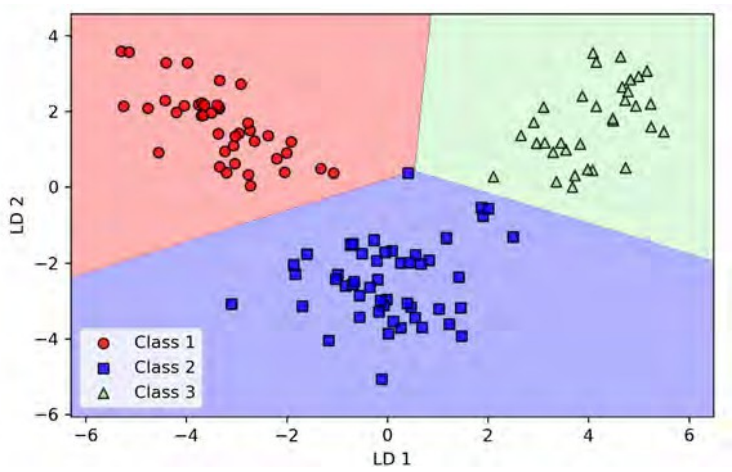
Bu adım adım uygulama, LDA'nın iç işleyişini anlamak ve LDA ile PCA arasındaki farkları anlamak için iyi bir alıştırmaydı. Şimdi, scikit-learn'de uygulanan LDA sınıfına bakalım:

```
>> # aşağıdaki import deyimi tek satırdır
>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
>> lda = LDA(n_components=2)
>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

Ardından, lojistik regresyon sınıflandırıcısının LDA dönüşümünden sonra daha düşük boyutlu eğitim veri kümesini nasıl ele aldığını görelim:

```
>> lr = LogisticRegression(multi_class='ovr', random_state=1,
...                         çözücü='lbfgs')
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>> plt.xlabel('LD 1')
>> plt.ylabel('LD 2')
>> plt.legend(loc='sol alt')
>> plt.tight_layout()
>> plt.show()
```

*Şekil 5.11'e* baktığımızda, lojistik regresyon modelinin sınıf 2'deki örneklerden birini yanlış sınıflandırdığını görebiliriz:



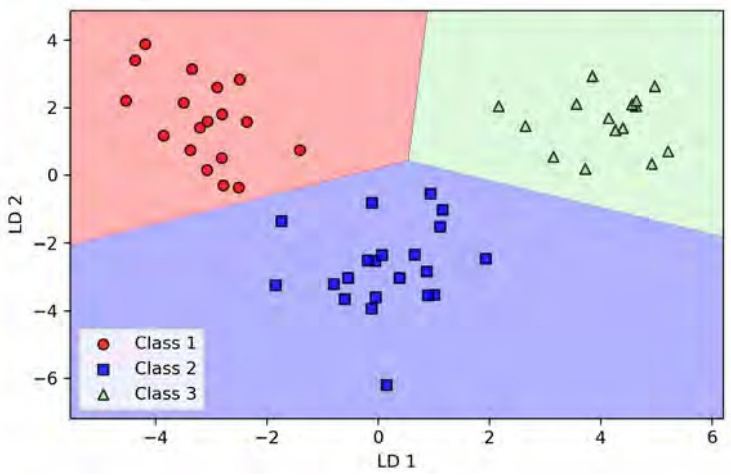

*Şekil 5.11: Lojistik regresyon modeli sınıflardan birini yanlış sınıflandırıyor*

 olimpikbarbell@gmail.com 

Düzenli hale getirme gücünü düşürerek, lojistik regresyon modelinin eğitim veri kümesindeki tüm örnekleri doğru sınıflandırması için karar sınırlarını muhtemelen kaydırabiliriz. Ancak, daha da önemlisi, test veri kümesindeki sonuçlara bir göz atalım:

```
>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>> plt.xlabel('LD 1')
>> plt.ylabel('LD 2')
>> plt.legend(loc='sol alt')
>> plt.tight_layout()
>> plt.show()
```

*Şekil 5.12*'de görebileceğimiz gibi, lojistik regresyon sınıflandırıcısı, orijinal 13 Şarap özelliği yerine yalnızca iki boyutlu bir özellik alt uzayı kullanarak test veri kümesindeki örnekleri sınıflandırmak için mükemmel bir doğruluk puanı alabilmektedir:




*Şekil 5.12: Lojistik regresyon modeli test verileri üzerinde mükemmel çalışıyor*

# Doğrusal olmayan boyut indirgeme ve görselleştirme

Bir önceki bölümde, özellik çıkarımı için PCA ve LDA gibi doğrusal dönüşüm tekniklerini ele aldık. Bu bölümde, doğrusal olmayan boyutsallık azaltma tekniklerini dikkate almanın neden faydalı olabileceğini tartışacağız.

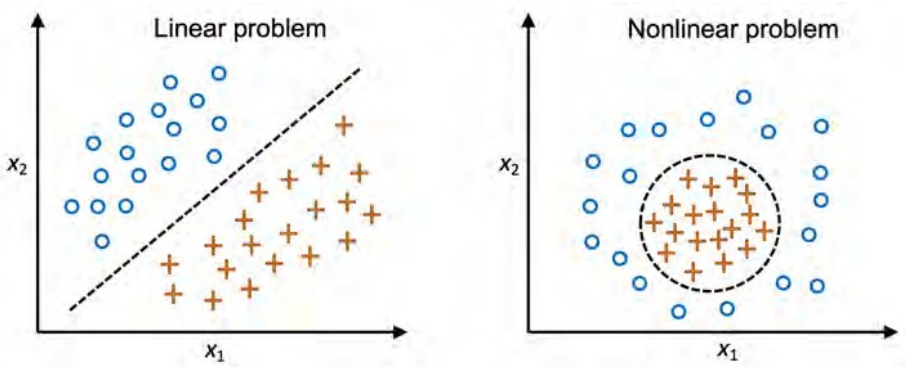
Literatürde yüksek boyutlu veri kümelerini iki veya üç boyutlu olarak görselleştirmek için sıklıkla kullanıldığından, özellikle vurgulanmaya değer bir doğrusal olmayan boyut azaltma tekniği **t-dağıtılmış stokastik komşu gömme** (**t-SNE**) tekniğidir. El yazısı görüntülerini 2 boyutlu bir özellik uzayında çizmek için t-SNE'yi nasıl uygulayabileceğimizi göreceğiz.

## Neden doğrusal olmayan boyut indirgeme düşünülmelidir?

Birçok makine öğrenimi algoritması, girdi verilerinin doğrusal ayrılabilirliği hakkında varsayımlarda bulunur.

Algılayıcının başarılı olmak için mükemmel doğrusal olarak ayrılabilir eğitim verisine bile ihtiyaç duyduğunu öğrendiniz. Şimdiye kadar ele aldığımız diğer algoritmalar, mükemmel doğrusal ayrılabilirlik eksikliğinin gürültüden kaynaklandığını varsaymaktadır: Adaline, lojistik regresyon ve (standart) SVM bunlardan sadece birkaçıdır.

Bununla birlikte, gerçek dünya uygulamalarında oldukça sık karşılaşabileceğimiz doğrusal olmayan problemlerle uğraşıyorsak, PCA ve LDA gibi boyut azaltma için doğrusal dönüşüm teknikleri en iyi seçim olmayabilir:



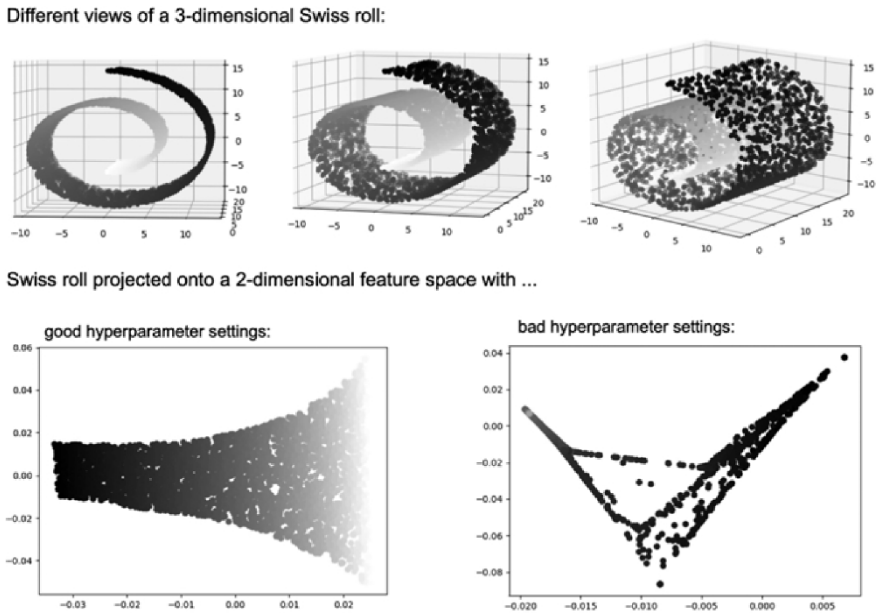

*Şekil 5.13: Doğrusal ve doğrusal olmayan problemler arasındaki fark*

Scikit-learn kütüphanesi, doğrusal olmayan boyut indirgeme için bu kitabın kapsamı dışında kalan bir dizi gelişmiş teknik uygulamaktadır. İlgilenen okuyucu, scikit-learn'deki mevcut uygulamaların açıklayıcı örneklerle tamamlanmış güzel bir özetini http://scikit- learn.org/stable/modules/manifold.html adresinde bulabilir.

Doğrusal olmayan boyut azaltma tekniklerinin geliştirilmesi ve uygulanması genellikle manifold öğrenme olarak da adlandırılır; burada manifold, yüksek boyutlu bir uzaya gömülü daha düşük boyutlu bir topolojik uzayı ifade eder. Manifold öğrenme algoritmaları, veri noktaları arasındaki ilişkinin korunduğu daha düşük boyutlu bir uzaya yansıtmak için verilerin uyumlu yapısını yakalamak zorundadır.

 olinpikbarbell@gmail.com 

Manifold öğrenmenin klasik bir örneği *Şekil 5.14'te* gösterilen 3 boyutlu İsviçre rulosudur:




*Şekil 5.14: Daha düşük, iki boyutlu bir uzaya yansıtılmış üç boyutlu İsviçre rulosu*

Doğrusal olmayan boyut azaltma ve manifold öğrenme algoritmaları çok güçlü olmakla birlikte, bu tekniklerin kullanımının zor olduğunu ve ideal olmayan hiperparametre seçimleriyle yarardan çok zarara neden olabileceklerini belirtmeliyiz. Bu zorluğun ardında yatan neden, genellikle kolayca görselleştiremediğimiz ve yapının belirgin olmadığı yüksek boyutlu veri kümeleriyle çalışıyor olmamızdır (*Şekil 5.14'*teki İsviçre rulosu örneğinin aksine). Dahası, veri kümesini iki veya üç boyuta yansıtmadığımız sürece (ki bu genellikle daha karmaşık ilişkileri yakalamak için yeterli değildir), sonuçların kalitesini değerlendirmek zor, hatta imkansızdır. Bu nedenle, birçok kişi hala boyut azaltma için PCA ve LDA gibi daha basit tekniklere güvenmektedir.

## Verilerin t-dağılımlı stokastik komşu gömme yoluyla görselleştirilmesi

Doğrusal olmayan boyut indirgemeyi tanıttıktan ve bazı zorluklarını tartıştıktan sonra, karmaşık veri kümelerini iki veya üç boyutlu olarak görselleştirmek için sıklıkla kullanılan t-SNE'yi içeren uygulamalı bir örneğe göz atalım.

Özetle, t-SNE veri noktalarını yüksek boyutlu (orijinal) özellik uzayında çift-bilge uzaklıklarına göre modellemektedir. Ardından, yeni, daha düşük boyutlu uzayda, orijinal uzaydaki ikili mesafelerin olasılık dağılımına yakın bir ikili mesafe olasılık dağılımı bulur. Başka bir deyişle, t-SNE veri noktalarını orijinal uzaydaki ikili mesafeler korunacak şekilde daha düşük boyutlu bir uzaya yerleştirmeyi öğrenir. Bu yöntem hakkında daha fazla ayrıntıyı *Maaten ve Hinton* tarafından yazılan *Visualizing data using t-SNE adlı* orijinal araştırma makalesinde bulabilirsiniz, *Jour- nal of Machine Learning Research,* 2018 (https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf). Ancak, araştırma makalesinin başlığından da anlaşılacağı üzere, t-SNE projeksiyon için tüm veri kümesini gerektirdiğinden görselleştirme amaçlı bir tekniktir. Noktaları doğrudan yansıttığı için (PCA'nın aksine, bir projeksiyon matrisi içermez), t-SNE'yi yeni veri noktalarına uygulayamayız.

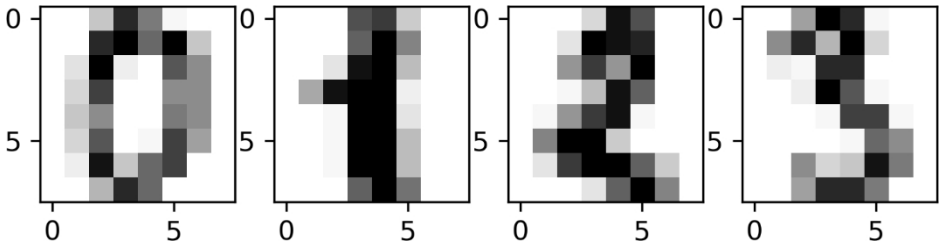
Aşağıdaki kod, t-SNE'nin 64 boyutlu bir veri kümesine nasıl uygulanabileceğinin hızlı bir gösterimini göstermektedir. İlk olarak, düşük çözünürlüklü el yazısı on rakamdan (0-9 rakamları) oluşan Digits veri kümesini scikit-learn'den yüklüyoruz:

```
>> from sklearn.datasets import load_digits
>>> digits = load_digits()
```

Rakamlar 8×8 gri tonlamalı görüntülerdir. Aşağıdaki kod, toplam 1.797 görüntüden oluşan veri kümesindeki ilk dört görüntüyü çizer:

```
>> fig, ax = plt.subplots(1, 4)
>> for i in range(4):
>>>     ax[i].imshow(digits.images[i], cmap='Greys')
>> plt.show()
```

*Şekil 5.15*'te görebileceğimiz gibi, görüntüler nispeten düşük çözünürlüklüdür, 8×8 piksel (yani görüntü başına 64 piksel):




*Şekil 5.15: El yazısı rakamların düşük çözünürlüklü görüntüleri*

digits.data niteliğinin, örneklerin satırlarla temsil edildiği ve sütunların piksellere karşılık geldiği bu veri kümesinin tablo biçimindeki bir sürümüne erişmemizi sağladığını unutmayın:

```
>>> digits.data.shape
(1797, 64)
```

 olinpikbarbell@gmail.com 

Ardından, özellikleri (pikselleri) yeni bir X_digits değişkenine ve etiketleri de başka bir y_digits değişkenine atayalım:

```
>> y_digits = d i g i t s .target
>> X_digits = d i g i t s .data
```

Ardından, scikit-learn'den t-SNE sınıfını içe aktarıyoruz ve yeni bir tsne nesnesi uyduruyoruz. fit_transform kullanarak, t-SNE uydurma ve veri dönüşümünü tek bir adımda gerçekleştiriyoruz:

```
>> from sklearn.manifold import TSNE
>> tsne = TSNE(n_components=2, init='pca',
...                random_state=123)
>>> X_digits_tsne = tsne.fit_transform(X_digits)
```

Bu kodu kullanarak 64 boyutlu veri kümesini 2 boyutlu bir uzaya yansıttık. *Kobak* ve *Linderman*'ın *Nature Biotechnology Cilt 39,* sayfa 156-157, 2021 (https://www.nature.com/articles/ s41587-020-00809-z) adlı araştırma makalesinde önerildiği gibi PCA kullanarak t-SNE gömülmesini *başlatan* init='pca' değerini belirledik.

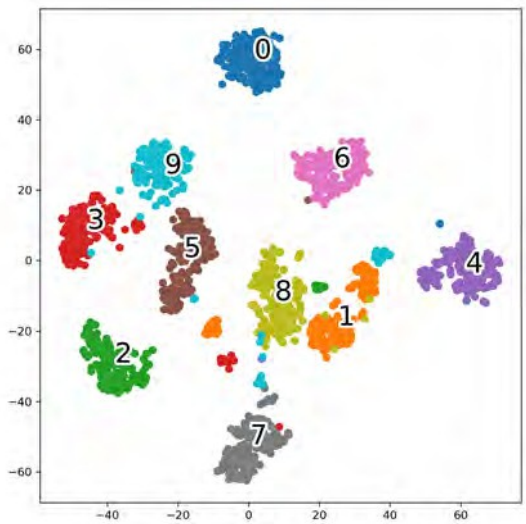
t-SNE'nin, örnekte atladığımız (scikit-learn varsayılan değerlerini kullandık) perplexity ve öğrenme oranı (genellikle **epsilon olarak** adlandırılır) gibi ek hiperparametreler içerdiğini unutmayın. Uygulamada, bu parametreleri de keşfetmenizi öneririz. Bu parametreler ve bunların sonuçlar üzerindeki etkileri hakkında daha fazla bilgi *Watten- berg*, *Viegas* ve *Johnson, Distill*, 2016 (https://distill.pub/2016/misread-tsne/) tarafından yazılan *How to Use t-SNE Effectively* adlı mükemmel makalede bulunabilir.

Son olarak, aşağıdaki kodu kullanarak 2D t-SNE gömülerini görselleştirelim:

```
>> import matplotlib.patheffects as PathEffects
>> def plot_projection(x, colors):

...     f = plt.figure(figsize=(8, 8))
...     ax = plt.subplot(aspect='equal')
...     for i in range(10):
...         plt.scatter(x[renkler == i,
...         0],
                        x[renkler == i,
                        1])
...     for i in range(10):
...         xtext, ytext = np.median(x[colors == i, :], axis=0) txt
...         = ax.text(xtext, ytext, str(i), fontsize=24)
...         t x t . set_path_effects([
...             PathEffects.Stroke(linewidth=5, foreground="w"),
...             PathEffects.Normal()])
```

```
>>> plot_projection(X_digits_tsne, y_digits)
>> plt.show()
```

PCA gibi, t-SNE de denetimsiz bir yöntemdir ve önceki kodda, y_digits (0-9) sınıf etiketlerini yalnızca functions color argümanı aracılığıyla görselleştirme amacıyla kullanıyoruz. Matplotlib'in PathEffects'i görsel amaçlar için kullanılır, öyle ki sınıf etiketi ilgili her bir basamağa ait veri noktalarının merkezinde (np.median aracılığıyla) görüntülenir. Ortaya çıkan çizim aşağıdaki gibidir:




*Şekil 5.16: t-SNE'nin el yazısı rakamlarını 2B özellik uzayına nasıl yerleştirdiğini gösteren bir görselleştirme*

Gördüğümüz gibi, t-SNE farklı basamakları (sınıfları) mükemmel olmasa da güzel bir şekilde ayırabilmektedir. Hiperparametreleri ayarlayarak daha iyi bir ayrıştırma elde etmek mümkün olabilir. Ancak, okunaksız el yazısı nedeniyle belirli bir derecede sınıf karışımı kaçınılmaz olabilir. Örneğin, tek tek görüntüleri inceleyerek, 3 rakamının bazı örneklerinin gerçekten de 9 rakamına benzediğini görebiliriz.

**Düzgün manifold yaklaşımı ve projeksiyon**

Bir diğer popüler görselleştirme tekniği de tekdüze **manifold yaklaşımı ve projeksiyonudur (UMAP)**. UMAP, t-SNE ile benzer şekilde iyi sonuçlar üretebilirken (örneğin, daha önce atıfta bulunulan Kobak ve Linderman makalesine bakın), tipik olarak daha hızlıdır ve yeni verileri yansıtmak için de kullanılabilir, bu da onu PCA'ya benzer bir makine öğrenimi bağlamında boyutluluk azaltma tekniği olarak daha çekici hale getirir. İlgilenen okuyucular UMAP hakkında daha fazla bilgiyi orijinal makalede bulabilirler: *UMAP: Uniform manifold approximation and projection for dimension reduction* by *McInnes, Healy,* and *Melville,* 2018 (https://arxiv.org/abs/1802.03426). UMAP'ın scikit-learn uyumlu bir uygulaması https://umap-learn.readthedocs.io adresinde bulunabilir.

 olimpikbarbell@gmail.com 

# Özet

Bu bölümde, özellik çıkarımı için iki temel boyutsallık azaltma tekniğini öğrendiniz: PCA ve LDA. PCA kullanarak, sınıf etiketlerini göz ardı ederken ortogonal özellik eksenleri boyunca varyansı en üst düzeye çıkarmak için verileri daha düşük boyutlu bir alt uzaya yansıttık. LDA, PCA'nın aksine, denetimli bir boyut azaltma tekniğidir, yani doğrusal bir özellik uzayında sınıf ayrılabilirliğini en üst düzeye çıkarmaya çalışmak için eğitim veri kümesindeki sınıf bilgilerini dikkate alır. Son olarak, verileri iki veya üç boyutta görselleştirmek için kullanılabilen doğrusal olmayan bir özellik çıkarma tekniği olan t-SNE hakkında da bilgi edindiniz.

Temel veri ön işleme teknikleri olarak PCA ve LDA ile donatılmış olarak, artık farklı ön işleme tekniklerini verimli bir şekilde dahil etmek ve bir sonraki bölümde farklı modellerin performansını değerlendirmek için en iyi uygulamaları öğrenmeye hazırsınız.

# Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E

olimpikbarbell@gmail.com

# 6

# Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Uygulamaları Öğrenme

Önceki bölümlerde, sınıflandırma için temel makine öğrenimi algoritmalarını ve bu algoritmalara beslemeden önce verilerimizi nasıl şekillendireceğimizi öğrendik. Şimdi, algoritmalara ince ayar yaparak ve modellerin performansını değerlendirerek iyi makine öğrenimi modelleri oluşturmanın en iyi uygulamalarını öğrenmenin zamanı geldi. Bu bölümde, aşağıdakileri nasıl yapacağımızı öğreneceğiz:

- Makine öğrenimi modellerinin performansını değerlendirme
- Makine öğrenimi algoritmalarının ortak sorunlarını teşhis etme
- Makine öğrenimi modellerine ince ayar yapın
- Farklı performans ölçütleri kullanarak tahmin modellerini değerlendirme

## İş akışlarını boru hatları ile kolaylaştırma

Önceki bölümlerde, *Bölüm 4, İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme'de* özellik ölçekleme için standartlaştırma veya *Bölüm 5*, *Boyut Azaltma Yoluyla Verileri Sıkıştırma'da* veri sıkıştırma için temel bileşen analizi gibi farklı önişleme tekniklerini uyguladığımızda, ayrı test veri kümesindeki örnekler gibi yeni verileri ölçeklemek ve sıkıştırmak için eğitim verilerinin uydurulması sırasında elde edilen parametreleri yeniden kullanmamız gerektiğini öğrendiniz. Bu bölümde, scikit-learn'deki `Pipeline` sınıfı gibi son derece kullanışlı bir araç hakkında bilgi edineceksiniz. İsteğe bağlı sayıda dönüşüm adımı içeren bir modeli uydurmamıza ve yeni veriler hakkında tahminler yapmak için uygulamamıza olanak tanır.

6587191CA584E

## Meme Kanseri Wisconsin veri kümesinin yüklenmesi

Bu bölümde, 569 kötü huylu ve iyi huylu tümör hücresi örneği içeren Meme Kanseri Wisconsin veri kümesi ile çalışacağız. Veri kümesindeki ilk iki sütun, sırasıyla örneklerin benzersiz kimlik numaralarını ve karşılık gelen tanıları (M = kötü huylu, B = iyi huylu) saklamaktadır. Sütun 3-32, hücre çekirdeklerinin sayısallaştırılmış görüntülerinden hesaplanan ve bir tümörün iyi huylu mu yoksa kötü huylu mu olduğunu tahmin etmek için bir model oluşturmak üzere kullanılabilecek 30 gerçek değerli özellik içerir. Meme Kanseri Wisconsin veri kümesi UCI Makine Öğrenimi Deposuna yerleştirilmiştir ve bu veri kümesi hakkında daha ayrıntılı bilgi https://archive.ics.uci.edu/ ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic) adresinde bulunabilir.

**Meme Kanseri Wisconsin veri setinin elde edilmesi**

Bu veri setinin (ve bu kitapta kullanılan diğer tüm veri setlerinin) bir kopyasını bu kitabın kod paketinde bulabilirsiniz; bu kopyayı ofis dışında çalışıyorsanız veya https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data adresindeki UCI sunucusu geçici olarak kullanılamıyorsa kullanabilirsiniz. Örneğin, veri kümesini yerel bir dizinden yüklemek için aşağıdaki satırları değiştirebilirsiniz:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu
    /ml/' 'machine-learning-databases'
    '/breast-cancer-wisconsin/wdbc.data',
    header=None
)
```

bunlarla:

```
df = pd.read_csv(
    'your/local/path/to/wdbc.data',
    header=None
)
```

Bu bölümde, veri kümesini okuyacağız ve üç basit adımda eğitim ve test veri kümelerine ayıracağız:

1. Pandas kullanarak veri setini doğrudan UCI web sitesinden okuyarak başlayacağız:

```
>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                  'machine-learning-databases'
...                  '/breast-cancer-wisconsin/wdbc.data',
...                  header=None)
```

 olimpikbarbell@gmail.com 

2. Daha sonra, 30 özelliği bir NumPy dizisi olan X'e atayacağız. Bir LabelEncoder nesnesi kullanarak, sınıf etiketlerini orijinal dize gösterimlerinden ('M' ve 'B') tam sayılara

```
>> from sklearn.preprocessing import LabelEncoder
>> X = df.loc[:, 2:].values
>> y = df.loc[:, 1].values
>> le = LabelEncoder()
>> y = le.fit_transform(y)
>> le.classes_
array(['B', 'M'], dtype=object)
```

dönüştüreceğiz:

3. Sınıf etiketlerini (tanı) bir dizide (y) kodladıktan sonra, kötü huylu tümörler artık sırasıyla sınıf 1 ve iyi huylu tümörler sınıf 0 olarak temsil edilmektedir. Bu eşleştirmeyi, iki kukla sınıf etiketi üzerinde uygun LabelEncoder'ın dönüştürme yöntemini çağırarak iki kez kontrol edebiliriz:

```
>> le.transform(['M', 'B']) array([1,
0])
```

4. Aşağıdaki alt bölümde ilk model hattımızı oluşturmadan önce, veri kümesini ayrı bir eğitim veri kümesine (verilerin yüzde 80'i) ve ayrı bir test veri kümesine (verilerin yüzde 20'si) ayıralım:

```
>> from sklearn.model_selection import train_test_split
>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                      test_size=0.20,
...                      stratify=y,
...                      random_state=1)
```

## Boru hattında dönüştürücüleri ve tahmin edicileri birleştirme

Önceki bölümde, birçok öğrenme algoritmasının optimum performans için aynı ölçekte girdi özelliklerine ihtiyaç duyduğunu öğrendiniz. Meme Kanseri Wisconsin veri kümesindeki özellikler çeşitli farklı ölçeklerde ölçüldüğünden, bunları lojistik regresyon gibi doğrusal bir sınıflandırıcıya beslemeden önce Meme Kanseri Wisconsin veri kümesindeki sütunları standartlaştıracağız. Ayrıca, *Bölüm 5'te* tanıtılan boyut azaltma için bir özellik çıkarma tekniği olan **temel bileşen analizi (PCA)** yoluyla verilerimizi ilk 30 boyuttan daha düşük iki boyutlu bir alt uzaya sıkıştırmak istediğimizi varsayalım.

6587191CA584E

Eğitim ve test veri kümeleri için model uydurma ve veri dönüştürme adımlarını ayrı ayrı uygulamak yerine StandardScaler, PCA ve LogisticRegression nesnelerini bir ardışık düzende zincirleyebiliriz:

```
>> from sklearn.preprocessing import StandardScaler
>>  from sklearn.decomposition import PCA
>> from sklearn.linear_model import LogisticRegression
>> from sklearn.pipeline import make_pipeline
>>  pipe_lr = make_pipeline(StandardScaler(),
...                         PCA(n_components=2),
...                         LogisticRegression())
>> pipe_lr.fit(X_train, y_train)
>> y_pred = pipe_lr.predict(X_test)
>> test_acc = pipe_lr.score(X_test, y_test)
>> print(f'Test doğruluğu: {test_acc:.3f}')
Test doğruluğu: 0,956
```

make_pipeline işlevi, isteğe bağlı sayıda scikit-learn dönüştürücüsünü (fit ve transform yöntemlerini girdi olarak destekleyen nesneler) ve ardından fit ve tahmin yöntemlerini uygulayan bir scikit-learn tahmin edicisini alır. Önceki kod örneğimizde, bu nesnelerden bir scikit-learn Pipeline nesnesi oluşturan make_pipeline işlevine girdi olarak StandardScaler ve PCA olmak üzere iki scikit-learn dönüştürücüsü ve bir LogisticRegression tahmin edicisi sağladık.

Bir scikit-learn Pipeline'ı bu bireysel dönüştürücüler ve tahmin ediciler etrafında bir meta-tahmin edici veya sarmalayıcı olarak düşünebiliriz. Pipeline'ın fit yöntemini çağırırsak, veriler tahmin edici nesnesine (pipeline'daki son öğe) ulaşana kadar bu ara adımlardaki fit ve transform çağrıları aracılığıyla bir dizi dönüştürücüden geçirilecektir. Tahmin edici daha sonra dönüştürülmüş eğitim verilerine uydurulacaktır.

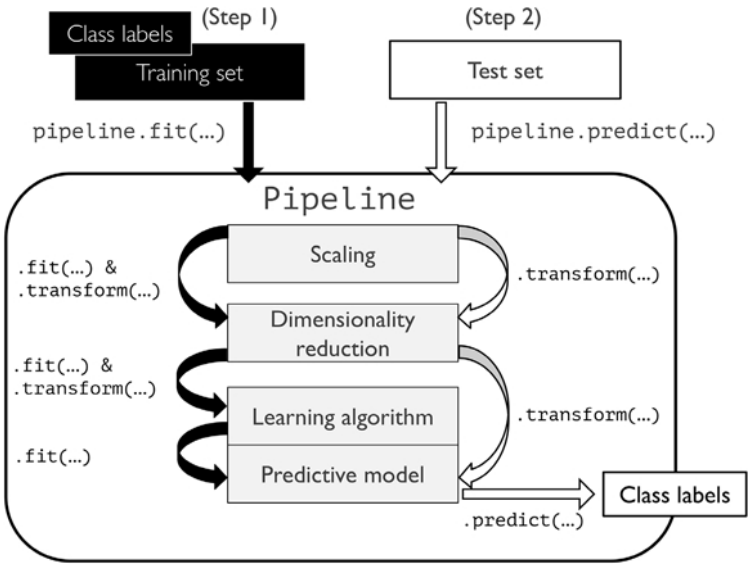
Önceki kod örneğinde pipe_lr ardışık düzeninde fit yöntemini çalıştırdığımızda, StandardScaler ilk olarak eğitim verileri üzerinde fit ve transform çağrıları gerçekleştirmiştir. İkinci olarak, dönüştürülen eğitim verileri ardışık düzende bir sonraki nesne olan PCA'ya aktarıldı. Önceki adıma benzer şekilde, PCA da ölçeklendirilmiş giriş verileri üzerinde fit ve transform işlemlerini gerçekleştirmiş ve bunları boru hattının son öğesi olan tahmin ediciye aktarmıştır.

Son olarak, LogisticRegression tahmin edicisi, StandardScaler ve PCA aracılığıyla dönüştürme işlemlerinden geçirildikten sonra eğitim verilerine uydurulmuştur. Yine, bir boru hattındaki ara adımların sayısında bir sınır olmadığını belirtmeliyiz; ancak, boru hattını tahmin görevleri için kullanmak istiyorsak, son boru hattı öğesi bir tahmin edici olmalıdır.

Bir boru hattında fit çağrısına benzer şekilde, boru hatlarındaki son adım bir tahmin edici ise boru hatları da bir tahmin yöntemi uygular. Bir Pipeline nesne örneğinin predict çağrısına bir veri kümesi beslersek, veriler dönüşüm çağrıları aracılığıyla ara adımlardan geçecektir. Son adımda, tahmin edici nesnesi dönüştürülmüş veriler üzerinde bir tahmin döndürecektir.

olimpikbarbell@gmail.com

Scikit-learn kütüphanesinin boru hatları, bu kitabın geri kalanında sıkça kullanacağımız son derece yararlı sarmalayıcı araçlardır. `Pipeline` nesnesinin nasıl çalıştığını iyice anladığınızdan emin olmak için, lütfen önceki paragraflardaki tartışmamızı özetleyen *Şekil 6.1*'e yakından bakın:




*Şekil 6.1: Boru Hattı nesnesinin iç işleyişi*

# Modelin performansını değerlendirmek için k-kat çapraz doğrulamayı kullanma

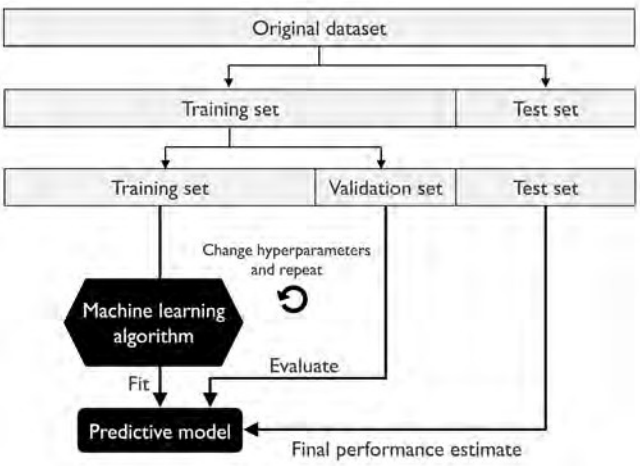
Bu bölümde, modelin genelleme performansı, yani modelin görülmeyen veriler üzerinde ne kadar iyi performans gösterdiği konusunda güvenilir tahminler elde etmemize yardımcı olabilecek yaygın çapraz doğrulama teknikleri **holdout çapraz doğrulama** ve **k-kat çapraz doğrulama** hakkında bilgi edineceksiniz.

## Bekletme yöntemi

Makine öğrenimi modellerinin genelleme performansını tahmin etmek için kullanılan klasik ve popüler bir yaklaşım da tutma yöntemidir. Bekletme yöntemini kullanarak ilk veri kümemizi ayrı eğitim ve test veri kümelerine ayırırız; bunlardan ilki model eğitimi için, ikincisi ise genelleme performansını tahmin etmek için kullanılır. Bununla birlikte, tipik makine öğrenimi uygulamalarında, görünmeyen veriler üzerinde tahminler yapma performansını daha da artırmak için farklı parametre ayarlarını ayarlamak ve karşılaştırmakla da ilgileniriz. Bu sürece **model seçimi** adı verilir ve bu ad, *ayarlama parametrelerinin* (**hiperparametreler** olarak da adlandırılır) *optimum* değerlerini seçmek istediğimiz belirli bir sınıflandırma problemine atıfta bulunur. Ancak, model seçimi sırasında aynı test veri setini tekrar tekrar kullanırsak, eğitim verilerimizin bir parçası haline gelecek ve böylece modelin aşırı uyum sağlama olasılığı artacaktır. Bu soruna rağmen, birçok kişi hala model seçimi için test veri setini kullanmaktadır ki bu iyi bir makine öğrenimi uygulaması değildir.

6587191CA584E

Model seçimi için tutma yöntemini kullanmanın daha iyi bir yolu, verileri üç parçaya ayırmaktır: bir eğitim veri kümesi, bir doğrulama veri kümesi ve bir test veri kümesi. Eğitim veri kümesi farklı modellere uyum sağlamak için kullanılır ve doğrulama veri kümesindeki performans daha sonra model seçimi için kullanılır. Eğitim ve model seçimi adımları sırasında modelin daha önce görmediği bir test veri kümesine sahip olmanın avantajı, modelin yeni verilere genelleme yeteneğine ilişkin daha az yanlı bir tahmin elde edebilmemizdir. *Şekil 6.2'de,* farklı hiperparametre değerleri kullanarak eğitimden sonra modelin performansını tekrar tekrar değerlendirmek için bir doğrulama veri kümesi kullandığımız holdout cross-validation kavramı gösterilmektedir. Hiperparametre değerlerinin ayarlanmasından memnun olduğumuzda, modelin test veri kümesi üzerindeki genelleme performansını tahmin ederiz:




*Şekil 6.2: Eğitim, doğrulama ve test veri kümeleri nasıl kullanılır?*

Bekletme yönteminin bir dezavantajı, performans tahmininin eğitim veri kümesini eğitim ve doğrulama alt kümelerine nasıl böldüğümüze çok duyarlı olabilmesidir; tahmin, verilerin farklı örnekleri için değişecektir. Bir sonraki alt bölümde, performans tahmini için daha sağlam bir teknik olan ve eğitim verilerinin k alt kümesi üzerinde tutma yöntemini k kez tekrarladığımız k-kat çapraz doğrulama yöntemine göz atacağız.

## K-kat çapraz doğrulama

K-kat çapraz doğrulamada, eğitim veri kümesini değiştirmeden rastgele *k* kata böleriz. Burada, *eğitim* katları olarak adlandırılan *k* - 1 kat model eğitimi için, *test katları* olarak adlandırılan bir kat ise performans değerlendirmesi için kullanılır. Bu prosedür *k* kez tekrarlanır, böylece *k* model ve performans tahmini elde edilir.

**Değişimli ve değişimsiz örnekleme**

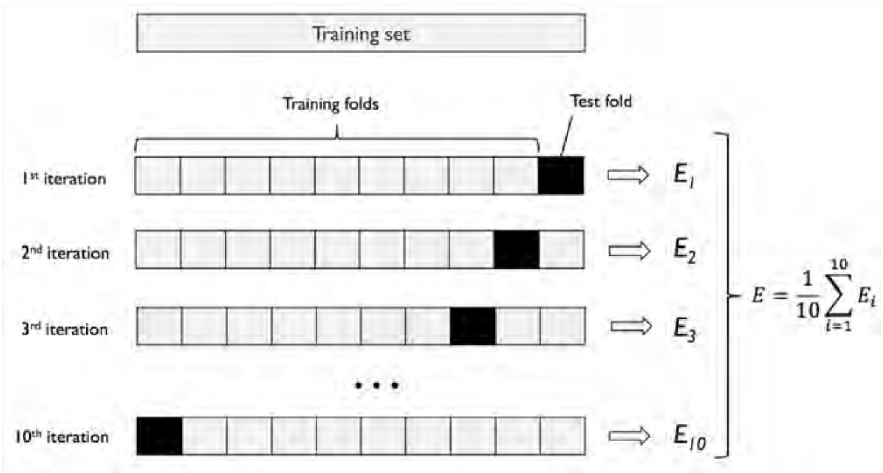
*Bölüm'*de yedekli ve yedeksiz örneklemeyi göstermek için bir örnek inceledik *3*. Bu bölümü okumadıysanız veya tazelemek istiyorsanız, *Rastgele ormanlar aracılığıyla birden fazla karar ağacını birleştirme* bölümündeki *Değiştirmeli ve değiştirmesiz örnekleme* başlıklı bilgi kutusuna bakın.

Daha sonra, bekletme yöntemine kıyasla eğitim verilerinin alt bölümlendirmesine daha az duyarlı bir performans tahmini elde etmek için farklı, bağımsız test katlarına dayalı modellerin ortalama performansını hesaplarız. Tipik olarak, model ayarlama için k-kat çapraz doğrulama kullanırız, yani test katlarındaki model performansını değerlendirerek tahmin edilen tatmin edici bir genelleme performansı sağlayan optimum hiperparametre değerlerini buluruz.

Tatmin edici hiperparametre değerleri bulduktan sonra, modeli tüm eğitim veri kümesi üzerinde yeniden eğitebilir ve bağımsız test veri kümesini kullanarak nihai bir performans tahmini elde edebiliriz. Bir modeli k-kat çapraz doğrulamadan sonra tüm eğitim veri setine uydurmanın arkasındaki mantık, ilk olarak, tipik olarak tek bir nihai modelle (*k* ayrı modele karşı) ilgilenmemiz ve ikincisi, bir öğrenme algoritmasına daha fazla eğitim örneği sağlamanın genellikle daha doğru ve sağlam bir modelle sonuçlanmasıdır.

K-kat çapraz doğrulama değiştirmesiz bir yeniden örnekleme tekniği olduğundan, bu yaklaşımın avantajı her iterasyonda her örneğin tam olarak bir kez kullanılacak olması ve eğitim ve test katlarının birbirinden ayrı olmasıdır. Ayrıca, tüm test katları birbirinden ayrıdır; yani test katları arasında hiçbir çakışma yoktur. *Şekil 6.3,* k = 10 ile *k-kat* çapraz doğrulamanın arkasındaki kavramı özetlemektedir. Eğitim veri kümesi 10 kata bölünür ve 10 iterasyon sırasında 9 kat eğitim için kullanılır ve 1 kat model değerlendirmesi için test veri kümesi olarak kullanılır.

Ayrıca, her bir kat için tahmin edilen performanslar, $E_i$ (örneğin, sınıflandırma doğruluğu veya hatası), daha sonra modelin tahmin edilen ortalama performansını, $E$, hesaplamak için kullanılır:




*Şekil 6.3: k-kat çapraz doğrulama nasıl çalışır?*

Özetle, k-kat çapraz doğrulamada tüm veri noktaları değerlendirme için kullanıldığından, k-kat çapraz doğrulama veri kümesini bir doğrulama kümesine sahip tutma yönteminden daha iyi kullanır.

Deneysel kanıtların da gösterdiği gibi, k-kat çapraz doğrulamada *k* için iyi bir standart değer 10'dur. Örneğin, Ron Kohavi tarafından çeşitli gerçek dünya veri kümeleri üzerinde yapılan deneyler, 10 katlı çapraz doğrulamanın *yanlılık ve* varyans arasında en iyi dengeyi sunduğunu göstermektedir (*A Study of Cross-Validation and Bootstrap for Accuracy Esti- mation and Model Selection* by *Kohavi, Ron*, *International Joint Conference on Artificial Intelligence (IJCAI)*, 14 ( 12): 1137-43, 1995,

6587191CA584E


https://www.ijcai.org/Proceedings/95-2/Papers/016.pdf).

 olimpikbarbell@gmail.com 

Ancak, nispeten küçük eğitim setleriyle çalışıyorsak, kat sayısını artırmak faydalı olabilir. Eğer $k$ değerini artırırsak, her iterasyonda daha fazla eğitim verisi kullanılacak ve bu da bireysel model tahminlerinin ortalamasını alarak genelleme performansını tahmin etmeye yönelik daha az kötümser bir önyargıya neden olacaktır. Bununla birlikte, büyük $k$ değerleri çapraz doğrulama algoritmasının çalışma süresini de artıracak ve eğitim katları birbirine daha benzer olacağı için daha yüksek varyanslı tahminler verecektir. Öte yandan, büyük veri kümeleriyle çalışıyorsak, $k$ için daha küçük bir değer seçebiliriz, örneğin $k = 5$ ve modelin ortalama performansının doğru bir tahminini elde ederken, modeli farklı katlarda yeniden yerleştirme ve değerlendirmenin hesaplama maliyetini azaltabiliriz.

**Tek seferlik çapraz doğrulama**

K-kat çapraz doğrulamanın özel bir durumu, **tek başına bırakarak çapraz doğrulama** (**LOOCV**) yöntemidir. LOOCV'de, kat sayısını eğitim örneklerinin sayısına ($k = n$) eşit olarak ayarlarız, böylece her iterasyon sırasında test için yalnızca bir eğitim örneği kullanılır, bu da çok küçük veri kümeleriyle çalışmak için önerilen bir yaklaşımdır.

Standart k-kat çapraz doğrulama yaklaşımına göre hafif bir iyileştirme olan katmanlı k-kat çapraz doğrulama, özellikle sınıf oranlarının eşit olmadığı durumlarda daha iyi yanlılık ve varyans tahminleri sağlayabilir, bu da bu bölümde daha önce atıfta bulunulan Ron Kohavi tarafından yapılan aynı çalışmada gösterilmiştir. Tabakalı çapraz doğrulamada, her katın eğitim veri kümesindeki sınıf oranlarını temsil etmesini sağlamak için sınıf etiketi oranları her katta korunur; bunu scikit-learn'deki `StratifiedKFold` yineleyicisini kullanarak göstereceğiz:

```
>> import numpy as np
>> from sklearn.model_selection import StratifiedKFold
>>> kfold = StratifiedKFold(n_splits=10).split(X_train, y_train)
>>> puanlar = []
>> for k, (train, test) in enumerate(kfold):
...     pipe_lr.fit(X_train[train], y_train[train])
...     score = pipe_lr.score(X_train[test], y_train[test])
...     scores.append(score)
...     print(f'Fold: {k+1:02d}, '
...           f'Sınıf distr.: {np.bincount(y_train[train])}, '
...           f'Acc.: {score:.3f}')
Kat: 01, Sınıf dağılımı: [256 153], Acc.: 0.935
Kat: 02, Sınıf dağılımı: [256 153], Acc.: 0.935
Kat: 03, Sınıf dağılımı: [256 153], Acc.: 0.957
Kat: 04, Sınıf dağılımı: [256 153], Acc.: 0.957
```

6587191CA584E

```
Kat: 05, Sınıf dağılımı: [256 153], Acc.: 0.935
Kat: 06, Sınıf dağılımı: [257 153], Acc.: 0.956
Kat: 07, Sınıf dağılımı: [257 153], Acc.: 0.978
Kat: 08, Sınıf dağılımı: [257 153], Acc.: 0.933
Kat: 09, Sınıf dağılımı: [257 153], Acc.: 0.956
Kat: 10, Sınıf dağılımı: [257 153], Acc.: 0.956
>>> mean_acc = np.mean(scores)
>> std_acc = np.std(scores)
>> print(f'\nCV doğruluğu: {mean_acc:.3f} +/- {std_acc:.3f}')
CV doğruluğu: 0.950 +/- 0.014
```

İlk olarak, sklearn.model_selection modülündeki StratifiedKFold yineleyicisini eğitim veri kümesindeki y_train sınıf etiketleri ile başlattık ve n_splits parametresi aracılığıyla kat sayısını belirttik. K katlar arasında döngü yapmak için kfold yineleyicisini kullandığımızda, bu bölümün başında kurduğumuz lojistik regresyon ardışık düzenine uyması için train içinde döndürülen indeksleri kullandık. pipe_lr ardışık düzenini kullanarak, örneklerin her yinelemede uygun şekilde ölçeklendirildiğinden (örneğin, standartlaştırıldığından) emin olduk. Daha sonra modelin doğruluk puanını hesaplamak için test endekslerini kullandık ve bu endeksleri ortalama doğruluğu ve tahminin standart sapmasını hesaplamak için puanlar listesinde topladık.

Önceki kod örneği k-kat çapraz doğrulamanın nasıl çalıştığını göstermek için yararlı olsa da, scikit-learn ayrıca katmanlı k-kat çapraz doğrulama kullanarak modelimizi daha az ayrıntılı bir şekilde değerlendirmemizi sağlayan bir k-kat çapraz doğrulama puanlayıcısı da uygular:

```
>> from sklearn.model_selection import cross_val_score
>>> scores = cross_val_score(estimator=pipe_lr,
...                          X=X_train,
...                          y=y_train,
...                          cv=10,
...                          n_jobs=1)
>> print(f'CV doğruluk puanları: {scores}')
CV doğruluk puanları: [ 0.93478261  0.93478261  0.95652174
                        0.95652174  0.93478261  0.95555556
                        0.97777778  0.93333333  0.95555556
                        0.95555556]
>> print(f'CV doğruluğu: {np.mean(scores):.3f} '
...        f'+/- {np.std(scores):.3f}')
CV doğruluğu: 0,950 +/- 0,014
```

Cross_val_score yaklaşımının son derece kullanışlı bir özelliği, farklı katların değerlendirilmesini makinemizdeki birden fazla **merkezi işlem birimine** (**CPU**) dağıtabilmemizdir. Eğer n_jobs parametresini 1 olarak ayarlarsak, daha önce StratifiedKFold örneğimizde olduğu gibi, performansları değerlendirmek için sadece bir CPU kullanılacaktır. Ancak, n_jobs=2 olarak ayarlayarak, 10 tur çapraz doğrulamayı iki CPU'ya dağıtabiliriz (makinemizde varsa) ve n_jobs=-1 olarak ayarlayarak, hesaplamayı paralel olarak yapmak için makinemizdeki mevcut tüm CPU'ları kullanabiliriz.

**Genelleme performansının tahmin edilmesi**

Çapraz doğrulamada genelleme performansının varyansının nasıl tahmin edildiğine ilişkin ayrıntılı bir tartışmanın bu kitabın kapsamı dışında olduğunu lütfen unutmayın, ancak model değerlendirme ve çapraz doğrulama hakkında https://arxiv.org/abs/1811.12808 adresinde paylaştığımız kapsamlı bir makaleye (Model *Evalua- tion, Model Selection, and Algorithm Selection in Machine Learning* by *S. Raschka*) başvurabilirsiniz. Bu makalede ayrıca .632 ve .632+ bootstrap çapraz doğrulama yöntemleri gibi alternatif çapraz doğrulama teknikleri de tartışılmaktadır.

Ayrıca, M. Markatou ve diğerlerinin mükemmel bir makalesinde ayrıntılı bir tartışma bulabilirsiniz (*Analysis of Variance of Cross-validation Estimators of the Generalization Error*, *M. Markatou, H. Tian*, *S. Biswas,* and *G. M. Hripcsak, Journal of Machine Learning Research*, 6: 1127- 1168, 2005), bu makaleye https://www.jmlr.org/papers/v6/markatou05a.html adresinden ulaşabilirsiniz.

# Öğrenme ve doğrulama ile algoritmalarda hata ayıklama eğrileri

Bu bölümde, bir öğrenme algoritmasının performansını iyileştirmemize yardımcı olabilecek çok basit ancak güçlü iki teşhis aracına göz atacağız: **öğrenme eğrileri** ve **doğrulama eğrileri**. Sonraki alt bölümlerde, bir öğrenme algoritmasının aşırı uyum (yüksek varyans) veya yetersiz uyum (yüksek yanlılık) sorunu olup olmadığını teşhis etmek için öğrenme eğrilerini nasıl kullanabileceğimizi tartışacağız. Ayrıca, öğrenme algoritmalarının yaygın sorunlarını ele almamıza yardımcı olabilecek doğrulama eğrilerine de bir göz atacağız.

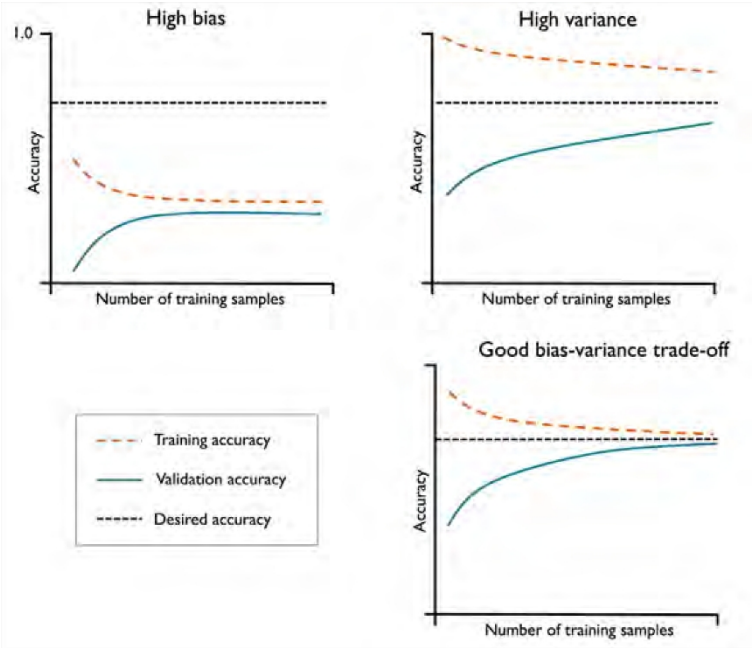
## Öğrenme eğrileri ile yanlılık ve varyans sorunlarının teşhisi

Bir model belirli bir eğitim veri kümesi için çok karmaşıksa (örneğin, çok derin bir karar ağacı düşünün), model eğitim verilerine aşırı uyma eğilimindedir ve görülmeyen verilere iyi genelleme yapmaz. Genellikle, aşırı uyum derecesini azaltmak için daha fazla eğitim örneği toplamak yardımcı olabilir.

Ancak pratikte, daha fazla veri toplamak genellikle çok pahalı olabilir veya mümkün olmayabilir. Model eğitim ve doğrulama doğruluklarını eğitim veri kümesi boyutunun fonksiyonu olarak çizerek, modelin yüksek varyans veya yüksek yanlılıktan muzdarip olup olmadığını ve daha fazla veri toplamanın bu sorunu çözmeye yardımcı olup olamayacağını kolayca tespit edebiliriz.

6587191CA584E

Ancak scikit-learn'de öğrenme eğrilerinin nasıl çizileceğini tartışmadan önce, aşağıdaki örnek üzerinden giderek bu iki yaygın model sorununu tartışalım:




*Şekil 6.4: Yaygın model sorunları*

Sol üstteki grafik yüksek yanlılığa sahip bir modeli göstermektedir. Bu model hem düşük eğitim hem de çapraz doğrulama doğruluğuna sahiptir, bu da eğitim verilerine yetersiz uyum sağladığını gösterir. Bu sorunu ele almanın yaygın yolları, örneğin ek özellikler toplayarak veya oluşturarak model parametrelerinin sayısını artırmak veya örneğin **destek vektör makinesi** (**SVM**) veya lojistik regresyon sınıflandırıcılarında düzenlilik derecesini azaltmaktır.

Sağ üstteki grafik, eğitim ve çapraz doğrulama doğruluğu arasındaki büyük farkla gösterilen yüksek varyanstan muzdarip bir modeli göstermektedir. Bu aşırı uyum sorununu ele almak için, örneğin daha fazla eğitim verisi toplayabilir, modelin karmaşıklığını azaltabilir veya düzenleme parametresini artırabiliriz.

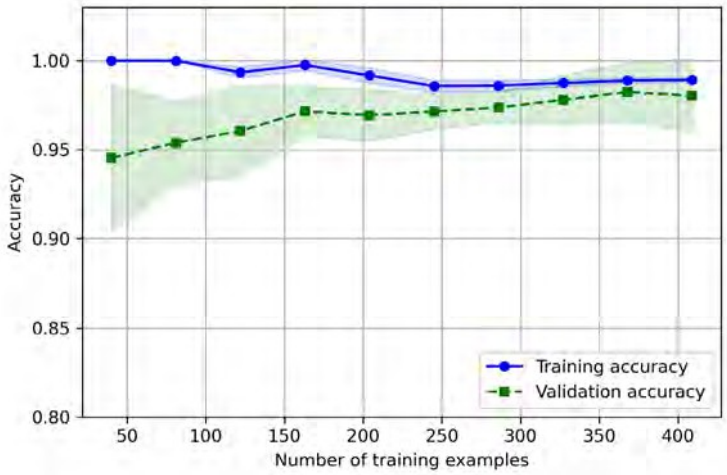
Düzensizleştirilmiş modellerde, aşırı uyum derecesini azaltmak için özellik seçimi (*Bölüm 4*) veya özellik çıkarma (*Bölüm 5)* yoluyla özellik sayısını azaltmak da yardımcı olabilir. Daha fazla eğitim verisi toplamak genellikle aşırı uyum olasılığını azaltma eğiliminde olsa da, örneğin eğitim verileri aşırı gürültülüyse veya model zaten optimale çok yakınsa, her zaman yardımcı olmayabilir.

Bir sonraki alt bölümde, doğrulama eğrilerini kullanarak bu model sorunlarını nasıl ele alacağımızı göreceğiz, ancak önce modeli değerlendirmek için scikit-learn'in öğrenme eğrisi işlevini nasıl kullanabileceğimizi görelim:

```
>> import matplotlib.pyplot as plt
>> from sklearn.model_selection import learning_curve
>>  pipe_lr = make_pipeline(StandardScaler(),
...                                LogisticRegression(penalty='l2',
...                                                   max_iter=10000))
>> train_sizes, train_scores, test_scores =\
...                     learning_curve(estimator=pipe_lr,
...                                    X=X_train,
...                                    y=y_train,
...                                    train_sizes=np.linspace(
...                                                   0.1, 1.0, 10),
...                                    cv=10,
...                                    n_jobs=1)
>> train_mean = np.mean(train_scores, axis=1)
>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>  plt.plot(train_sizes, train_mean,
...             color='blue', marker='o',
...             markersize=5, label='Eğitim doğruluğu')
> >   plt.fill_between(train_sizes,
...                      train_mean + train_std,
...                      train_mean - train_std,
...                      alpha=0.15, color='blue')
>>  plt.plot(train_sizes, test_mean,
...             color='green', linestyle='--',
...             marker='s', markersize=5,
...             label='Doğrulama doğruluğu')
> >   plt.fill_between(train_sizes,
...                      test_ortalama + test_std,
...                      test_mean - test_std,
...                      alpha=0.15, color='green')
>> plt.grid()
>> plt.xlabel('Eğitim örneklerinin sayısı')
>>  plt.ylabel('Doğruluk')
>> plt.legend(loc='sağ alt')
>> plt.ylim([0.8, 1.03])
>> plt.show()
```

6587191CA584E

LogisticRegression nesnesini (varsayılan olarak 1.000 yineleme kullanır) örneklendirirken, daha küçük veri kümesi boyutları veya aşırı düzenleme parametresi değerleri (bir sonraki bölümde ele alınmıştır) için yakınsama sorunlarından kaçınmak amacıyla ek bir bağımsız değişken olarak max_iter=10000 değerini geçtiğimizi unutmayın. Önceki kodu başarıyla uyguladıktan sonra, aşağıdaki öğrenme eğrisi grafiğini elde edeceğiz:




*Şekil 6.5: Eğitim örneklerinin sayısına göre eğitim ve doğrulama veri kümesi doğruluğunu gösteren bir öğrenme eğrisi*

learning_curve işlevindeki train_sizes parametresi aracılığıyla, öğrenme eğrilerini oluşturmak için kullanılan eğitim örneklerinin mutlak veya göreceli sayısını kontrol edebiliriz. Burada, eğitim veri kümesi boyutları için 10 eşit aralıklı, göreceli aralık kullanmak üzere train_sizes=np.linspace(0.1, 1.0, 10) olarak ayarladık. Varsayılan olarak, learning_curve işlevi bir sınıflandırıcının çapraz doğrulama doğruluğunu hesaplamak için katmanlı k-kat çapraz doğrulama kullanır ve 10-kat katmanlı çapraz doğrulama için cv parametresi aracılığıyla $k = 10$ olarak ayarladık.

Ardından, Matplotlib'in plot işlevini kullanarak çizdiğimiz eğitim veri kümesinin farklı boyutları için döndürülen çapraz doğrulamalı eğitim ve test puanlarından ortalama doğrulukları hesapladık. Ayrıca, tahminin varyansını belirtmek için fill_between işlevini kullanarak ortalama doğruluğun standart sapmasını grafiğe ekledik.

Önceki öğrenme eğrisi grafiğinde görebileceğimiz gibi, modelimiz eğitim sırasında 250'den fazla örnek görmüşse hem eğitim hem de doğrulama veri kümelerinde oldukça iyi performans göstermektedir. Ayrıca, 250'den az örnek içeren eğitim veri kümeleri için eğitim doğruluğunun arttığını ve doğrulama ile eğitim doğruluğu arasındaki farkın genişlediğini görebiliriz - bu da artan derecede aşırı uyumun bir göstergesidir.

## Doğrulama eğrileri ile aşırı ve yetersiz uyumun ele alınması

Doğrulama eğrileri, aşırı uyum veya yetersiz uyum gibi sorunları ele alarak bir modelin performansını iyileştirmek için yararlı bir araçtır. Doğrulama eğrileri öğrenme eğrileriyle ilişkilidir, ancak eğitim ve test doğruluklarını örneklem büyüklüğünün fonksiyonları olarak çizmek yerine, model parametrelerinin değerlerini, örneğin lojistik regresyondaki ters düzenleme parametresi

olimpikbarbell@gmail.com

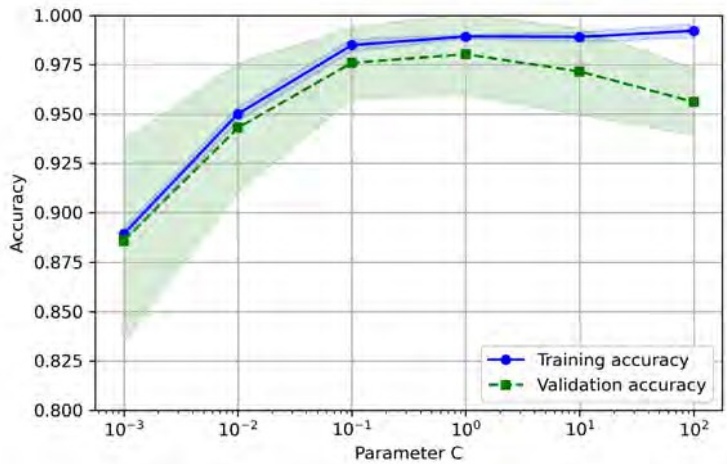

C'yi değiştiririz.

6587191CA584E

Devam edelim ve scikit-learn aracılığıyla doğrulama eğrilerini nasıl oluşturduğumuzu görelim:

```
>> from sklearn.model_selection import validation_curve
>> param_range = [0.001, 0 . 01, 0 . 1, 1.0, 10.0, 100.0]
>> train_scores, test_scores = validation_curve(
...                                    tahminci=pipe_lr,
...                                    X=X_train,
...                                    y=y_train,
...                                    param_name='logisticregression C',
...                                    param_range=param_range,
...                                    cv=10)
>> train_mean = np.mean(train_scores, axis=1)
>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>  plt.plot(param_aralığı, train_mean,
...              color='blue', marker='o',
...              markersize=5, label='Eğitim doğruluğu')
>> plt.fill_between(param_range, train_mean + train_std,
...                         train_mean - train_std, alpha=0.15,
...                         color='blue')
>>  plt.plot(param_range, test_mean,
...              color='green', linestyle='--',
...              marker='s', markersize=5,
...              label='Doğrulama doğruluğu')
> >   plt.fill_between(param_range,
...                         test_ortalama + test_std,
...                         test_mean - test_std,
...                         alpha=0.15, color='green')
>> plt.grid()
>> plt.xscale('log')
>> plt.legend(loc='sağ alt')
>> plt.xlabel('Parametre C')
> >  plt.ylabel('Doğruluk')
>> plt.ylim([0.8, 1.0])
>> plt.show()
```

Önceki kodu kullanarak, C parametresi için doğrulama eğrisi grafiğini elde ettik:




*Şekil 6.6: DVM hiper parametresi C için bir doğrulama eğrisi grafiği*

learning_curve fonksiyonuna benzer şekilde, validation_curve fonksiyonu da sınıflandırıcının performansını tahmin etmek için varsayılan olarak katmanlı k-kat çapraz değerlendirmeyi kullanır. validation_curve fonksiyonunun içinde, değerlendirmek istediğimiz parametreyi belirttik. Bu durumda, param_range parametresi aracılığıyla belirlediğimiz belirli bir değer aralığı için scikit-learn işlem hattı içindeki LogisticRegression nesnesine erişmek için 'logisticregression C' olarak yazdığımız LogisticRegression sınıflandırıcısının ters düzenleme parametresi olan C'dir. Önceki bölümdeki öğrenme eğrisi örneğine benzer şekilde, ortalama eğitim ve çapraz doğrulama doğruluklarını ve bunlara karşılık gelen standart sapmaları çizdik.

C'nin değişen değerleri için doğruluktaki farklılıklar ince olsa da, düzenlileştirme gücünü artırdığımızda (C'nin küçük değerleri) modelin verilere biraz daha az uyduğunu görebiliriz. Bununla birlikte, C'nin büyük değerleri için, düzenliliğin gücünü düşürmek anlamına gelir, bu nedenle model verilere biraz fazla uyma eğilimindedir. Bu durumda, tatlı nokta C değerinin 0,1 ile 1,0'ı arasında görünmektedir.

# Makine öğrenimi modellerine ızgara araması yoluyla ince ayar yapma

Makine öğreniminde iki tür parametre vardır: eğitim verilerinden öğrenilenler, örneğin lojistik regresyondaki ağırlıklar ve bir öğrenme algoritmasının ayrı olarak çalıştırılan parametreleri. İkincisi, bir modelin ayarlama parametreleridir (veya hiperparametreleridir), örneğin lojistik regresyondaki düzenleme parametresi veya bir karar ağacının maksimum derinlik parametresi.

Önceki bölümde, hiperparametrelerinden birini ayarlayarak bir modelin performansını artırmak için doğrulama eğrilerini kullandık. Bu bölümde, hiperparametre değerlerinin *optimum* kombinasyonunu bularak bir modelin performansını iyileştirmeye daha fazla yardımcı olabilecek **ızgara arama** adı verilen popüler bir hiperparametre optimi- zasyon tekniğine göz atacağız.

## Izgara arama yoluyla hiperparametrelerin ayarlanması

Izgara arama yaklaşımı oldukça basittir: farklı hiperparametreler için bir değerler listesi belirlediğimiz ve bilgisayarın bu kümeden en uygun değer kombinasyonunu elde etmek için her kombinasyon için model performansını değerlendirdiği kaba kuvvet kapsamlı bir arama

```
>> from sklearn.model_selection import GridSearchCV
>> from sklearn.svm import SVC
>>  pipe_svc = make_pipeline(StandardScaler(),
...                                 SVC(random_state=1))
>> param_aralığı = [0.0001, 0.001, 0.01, 0.1,
...                  1.0, 10.0, 100.0, 1000.0]
>> param_grid = [{'svc C': param_range,
...                  'svc kernel': ['linear']},
...                 {'svc C': param_range,
...                  'svc gamma': param_range,
...                  'svc kernel': ['rbf']}]
>>> gs = GridSearchCV(tahmin edici=pipe_svc,
...                       param_grid=param_grid,
...                       scoring='accuracy',
...                       cv=10,
...                       refit=True,
...                       n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>> print(gs.best_score_)
0.9846153846153847
>> print(gs.best_params_)
{'svc C': 100.0, 'svc gamma': 0.001, 'svc kernel': 'rbf'}
```

paradigmasıdır:

Önceki kodu kullanarak, bir SVM boru hattını eğitmek ve ayarlamak için sklearn.model_selection modülünden bir GridSearchCV nesnesini başlattık. Ayarlamak istediğimiz parametreleri belirtmek için GridSearchCV'nin param_grid parametresini bir sözlük listesine ayarladık. Doğrusal DVM için yalnızca ters düzenleme parametresi olan C'yi değerlendirdik; **radyal tabanlı fonksiyon (RBF)** çekirdek DVM için hem svc C hem de svc gama parametrelerini ayarladık. svc gama parametresinin çekirdek DVM'lere özgü olduğunu unutmayın.

GridSearchCV, farklı hiperparametrelerle eğitilen modelleri karşılaştırmak için k-kat çapraz doğrulama kullanır. cv=10 ayarı aracılığıyla, 10 katlı çapraz doğrulama gerçekleştirecek ve model performansını değerlendirmek için bu 10 kat boyunca ortalama doğruluğu (scoring='accuracy' aracılığıyla) hesaplayacaktır. GridSearchCV'nin modelleri farklı katlara paralel olarak uydurarak ızgara aramasını hızlandırmak için tüm işlem çekirdeklerimizi kullanabilmesi için n_jobs=-1 olarak ayarladık, ancak makineniz bu ayarla ilgili sorun yaşıyorsa, tek işlem için bu ayarı n_jobs=None olarak değiştirebilirsiniz.

Izgara aramasını gerçekleştirmek için eğitim verilerini kullandıktan sonra, best_score_ özniteliği aracılığıyla en iyi performans gösteren modelin puanını elde ettik ve best_params_ özniteliği

 olimpikbarbell@gmail.com 

aracılığıyla erişilebilen parametrelerine baktık. Bu özel durumda, `svc C = 100,0` ile RBF çekirdek SVM modeli en iyi k-kat çapraz doğrulama doğruluğunu verdi: yüzde 98,5.

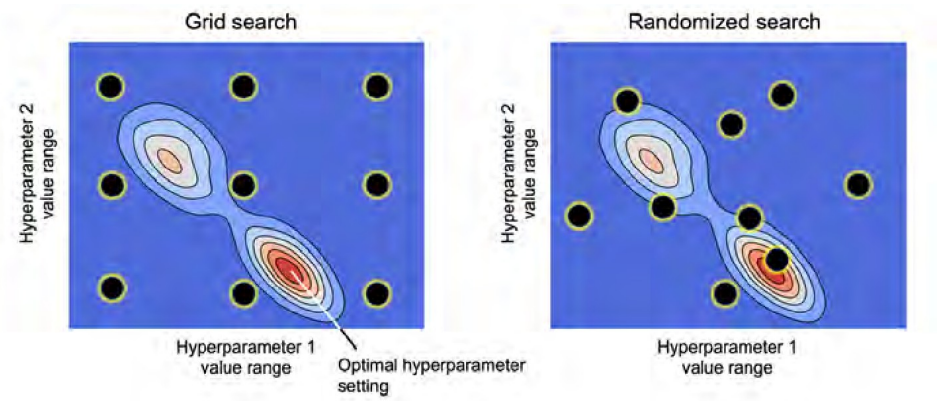
Son olarak, GridSearchCV nesnesinin best_estimator_ niteliği aracılığıyla erişilebilen en iyi seçilmiş modelin performansını tahmin etmek için bağımsız test veri kümesini kullanıyoruz:

```
>> clf = gs.best_estimator_
>> clf.fit(X_train, y_train)
>> print(f'Test doğruluğu: {clf.score(X_test, y_test):.3f}')
Test doğruluğu: 0,974
```

Izgara aramasını tamamladıktan sonra clf.fit(X_train, y_train) aracılığıyla manuel olarak eğitim setine en iyi ayarlarla (gs.best_estimator_) bir model uydurmanın gerekli olmadığını lütfen unutmayın. GridSearchCV sınıfı, refit=True (varsayılan) olarak ayarlarsak gs.best_estimator_'ı tüm eğitim setine otomatik olarak yeniden uyarlayacak bir refit parametresine sahiptir.

## Rastgele arama ile hiperparametre konfigürasyonlarını daha geniş çapta keşfetme

Izgara araması kapsamlı bir arama olduğundan, kullanıcı tarafından belirtilen parametre ızgarasında yer alıyorsa optimum hiperparametre konfigürasyonunu bulması garanti edilir. Ancak, büyük hiperparametre ızgaraları belirlemek ızgara aramasını pratikte çok pahalı hale getirir. Farklı parametre kombinasyonlarını örneklemek için alternatif bir yaklaşım rastgele aramadır. Rastgele aramada, hiperparametre konfigürasyonlarını dağılımlardan (veya ayrık kümelerden) rastgele çekeriz. Izgara aramanın aksine, rastgele arama hiperparametre uzayı üzerinde kapsamlı bir arama yapmaz. Yine de, daha geniş bir hiperparametre değeri ayar aralığını daha maliyet ve zaman etkin bir şekilde keşfetmemizi sağlar. Bu kavram, dokuz hiperparametre ayarından oluşan sabit bir ızgaranın ızgara arama ve rastgele arama yoluyla arandığını gösteren *Şekil 6.7'de gösterilmektedir:*




*Şekil 6.7: Her biri dokuz farklı hiper parametre konfigürasyonunu örneklemek için ızgara arama ve rastgele aramanın karşılaştırılması*

Ana çıkarım, ızgara aramanın yalnızca ayrık, kullanıcı tarafından belirlenen seçenekleri araştırırken, arama alanı çok azsa iyi hiperparametre konfigürasyonlarını kaçırabileceğidir. İlgilenen okuyucular, aşağıdaki makalede ampirik çalışmalarla birlikte rastgele arama hakkında ek ayrıntılar bulabilirler: *Hiper-Parametre Optimizasyonu* için *Rastgele Arama*, *J. Bergstra*, *Y. Bengio, Journal of Machine Learning Research,* s. 281-305, 2012, https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.

 olimpikbarbell@gmail.com 

Bir SVM'yi ayarlamak için rastgele aramayı nasıl kullanabileceğimize bakalım. Scikit-learn, bir önceki alt bölümde kullandığımız GridSearchCV'ye benzer bir RandomizedSearchCV sınıfı uygular. Temel fark, parametre ızgaramızın bir parçası olarak dağılımları belirtebilmemiz ve değerlendirilecek toplam hiperparametre konfigürasyonu sayısını belirtebilmemizdir. Örneğin, önceki bölümdeki ızgara arama örneğinde DVM'yi ayarlarken birkaç hiperparametre için kullandığımız hiperparametre aralığını ele alalım:

```
>> import scipy.stats
>> param_aralığı = [0.0001, 0.001, 0.01, 0.1,
...                 1.0, 10.0, 100.0, 1000.0]
```

RandomizedSearchCV, kategorik hiperparametreleri göz önünde bulundururken yararlı olan parametre ızgarası için girdi olarak benzer ayrık değer listelerini kabul edebilirken, asıl gücü bu listeleri örneklemek için dağılımlarla değiştirebileceğimiz gerçeğinde yatmaktadır. Böylece, örneğin, önceki listeyi SciPy'den aşağıdaki dağılımla değiştirebiliriz:

```
>> param_range = scipy.stats.loguniform(0.0001, 1000.0)
```

Örneğin, normal bir tekdüze dağılım yerine loguniform bir dağılım kullanmak, yeterince çok sayıda denemede [0.0001, 0.001] aralığından, örneğin [10.0, 100.0] aralığıyla aynı sayıda örnek çekilmesini sağlayacaktır. Davranışını kontrol etmek için, burada gösterildiği gibi rvs(10) yöntemiyle bu dağılımdan 10 rastgele örnek çekebiliriz:

```
>> np.random.seed(1)
>> param_range.rvs(10)
array([8.30145146e-02, 1.10222804e+01, 1.00184520e-04, 1.30715777e-02,
       1.06485687e-03, 4.42965766e-04, 2.01289666e-03, 2.62376594e-02,
       5.98924832e-02, 5.91176467e-01])
```

**Dağıtımları belirleme**

RandomizedSearchCV, rvs() yöntemini çağırarak onlardan örnek alabildiğimiz sürece rastgele dağılımları destekler. Şu anda scipy. stats aracılığıyla kullanılabilen tüm dağılımların bir listesi burada bulunabilir: https://docs.scipy.org/doc/scipy/reference/stats. html#probability-distributions.

Şimdi RandomizedSearchCV'yi çalışırken görelim ve önceki bölümde GridSearchCV ile yaptığımız gibi bir SVM'yi ayarlayalım:

```
>> from sklearn.model_selection import RandomizedSearchCV
>> pipe_svc = make_pipeline(StandardScaler(),
...                         SVC(random_state=1))
```

6587191CA584E

```
>> param_grid = [{'svc C': param_range,
...                    'svc kernel': ['linear']},
...                   {'svc C': param_range,
...                    'svc gamma': param_range,
...                    'svc kernel': ['rbf']}]
>>> rs = RandomizedSearchCV(tahminci=pipe_svc,
...                             param_distributions=param_grid,
...                             scoring='accuracy',
...                             refit=True,
...                             n_iter=20,
...                             cv=10,
...                             random_state=1,
...                             n_jobs=-1)

>>> rs = rs.fit(X_train, y_train)
>> print(rs.best_score_)
0.9670531400966184

>> print(rs.best_params_)
{'svc C': 0.05971247755848464, 'svc kernel': 'linear'}
```

Bu kod örneğine dayanarak, kullanımın GridSearchCV'ye çok benzediğini görebiliriz, ancak parametre aralıklarını belirtmek için dağılımları kullanabilir ve n_iter=20 ayarını yaparak yineleme sayısını-20 yineleme olarak belirleyebiliriz.

## Ardışık yarılama ile daha kaynak verimli hiperparametre arama

Randomize arama fikrini bir adım öteye taşıyan scikit-learn, uygun hiperparametre konfigürasyonlarını bulmayı daha verimli hale getiren HalvingRandomSearchCV adlı ardışık yarılama varyantını uygular. Ardışık yarılama, geniş bir aday konfigürasyon kümesi verildiğinde, yalnızca bir konfigürasyon kalana kadar taviz vermeyen hiperparametre konfigürasyonlarını art arda atar. Prosedürü aşağıdaki adımlarla özetleyebiliriz:

1. Rastgele örnekleme yoluyla geniş bir aday konfigürasyon seti çizin
2. Modelleri sınırlı kaynaklarla, örneğin eğitim verilerinin küçük bir alt kümesiyle eğitin (tüm eğitim setini kullanmak yerine)
3. Tahmini performansa dayalı olarak en alttaki yüzde 50'yi atın
4. Mevcut kaynakların miktarını artırarak *2. adıma* geri dönün

 olimpikbarbell@gmail.com 

Adımlar yalnızca bir hiperparametre yapılandırması kalana kadar tekrarlanır. Izgara arama varyantı için HalvingGridSearchCV adı verilen ve *1. adımda* rastgele örnekler yerine belirtilen tüm hiperparametre konfigürasyonlarının kullanıldığı bir ardışık yarılama uygulaması da olduğunu unutmayın.

Scikit-learn 1.0'da, HalvingRandomSearchCV hala deneyseldir, bu yüzden önce onu etkinleştirmemiz gerekir:

```
>> from sklearn.experimental import enable_halving_search_cv
```

(Yukarıdaki kod çalışmayabilir veya gelecekteki sürümlerde desteklenmeyebilir).

Deneysel desteği etkinleştirdikten sonra, aşağıda gösterildiği gibi ardışık yarılama ile rastgele aramayı kullanabiliriz:

```
>> from sklearn.model_selection import HalvingRandomSearchCV

>>> hs = HalvingRandomSearchCV(pipe_svc,
...                            param_distributions=param_grid,
...                            n_candidates='exhaust',
...                            resource='n_samples',
...                            faktör=1.5,
...                            random_state=1,
...                            n_jobs=-1)
```

resource='n_samples' (varsayılan) ayarı, eğitim seti boyutunu turlar arasında değiştirdiğimiz yeniden kaynak olarak kabul ettiğimizi belirtir. Faktör parametresi aracılığıyla, her turda kaç adayın eleneceğini belirleyebiliriz. Örneğin, faktör=2 olarak ayarlandığında adayların yarısı elenir ve faktör=1,5 olarak ayarlandığında adayların yalnızca %100/1,5 ≈ %66'sı bir sonraki tura geçer. RandomizedSearchCV'de olduğu gibi sabit sayıda iterasyon seçmek yerine, n_candidates='exhaust' (varsayılan) olarak ayarlıyoruz, bu da son turda maksimum sayıda kaynak (burada: eğitim örnekleri) kullanılacak şekilde hiperparametre konfigürasyonlarının sayısını örnekleyecektir.

Daha sonra RandomizedSearchCV'ye benzer bir arama gerçekleştirebiliriz:

```
>>> hs = hs.fit(X_train, y_train)
>> print(hs.best_score_)
0.9617647058823529

>> print(hs.best_params_)
{'svc C': 4.934834261073341, 'svc kernel': 'linear'}
```

6587191CA584E

```
>> clf = hs.best_estimator_
>> print(f'Test doğruluğu: {hs.score(X_test, y_test):.3f}')
Test doğruluğu: 0,982
```

Önceki iki alt bölümdeki GridSearchCV ve RandomizedSearchCV sonuçlarını HalvingRandomSearchCV modeliyle karşılaştırırsak, ikincisinin test setinde biraz daha iyi performans gösteren bir model verdiğini görebiliriz (yüzde 97,4'e karşılık yüzde 98,2 doğruluk).

**Hyperopt ile hiperparametre ayarlama**

Hiperparametre optimizasyonuna yönelik bir diğer popüler kütüphane de hiperopt'tur (https://github. com/hyperopt/hyperopt) ve hiperparametre optimizasyonu için rastgele arama ve **Ağaç Yapılandırılmış Parzen Es- timatörleri** (**TPE**) yöntemi dahil olmak üzere birkaç farklı yöntem uygular. TPE, geçmiş hiperparametre değerlendirmelerine ve bu değerlendirmeleri bağımsız olaylar olarak görmek yerine ilgili performans puanlarına dayalı olarak sürekli güncellenen olasılıksal bir modele dayanan bir Bayesian optimizasyon yöntemidir. TPE hakkında daha fazla bilgiyi *Hiper-Parametre Optimizasyonu için Algoritmalar* bölümünde bulabilirsiniz. *Bergstra J, Bardenet R, Bengio Y, Kegl B. NeurIPS 2011*. s. 2546-2554, https://dl.acm. org/doi/10.5555/2986459.2986743.

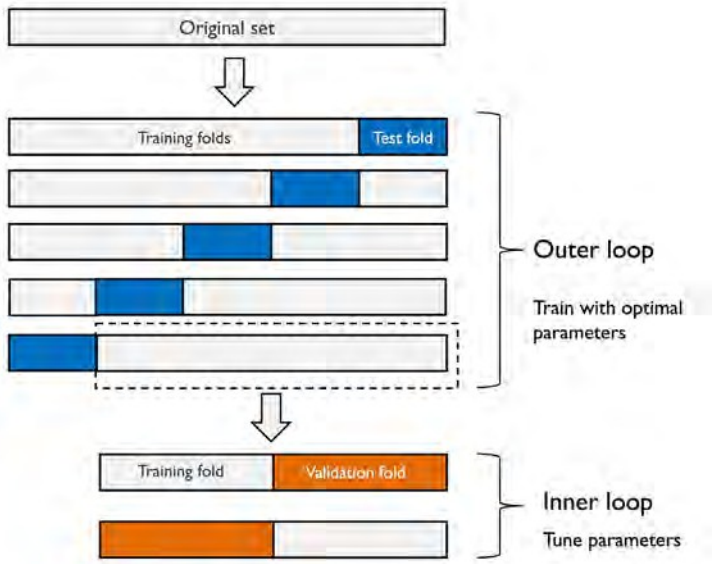
Hyperopt hiperparametre optimizasyonu için genel amaçlı bir arayüz sağlarken, ek kolaylık için hyperopt-sklearn adlı scikit-learn'e özgü bir paket de bulunmaktadır: https://github.com/hyperopt/hyperopt-sklearn.

## İç içe çapraz doğrulama ile algoritma seçimi

Önceki alt bölümlerde gördüğümüz gibi, k-kat çapraz doğrulamayı ızgara arama veya rastgele arama ile birlikte kullanmak, hiperparametrelerinin değerlerini değiştirerek bir makine öğrenimi modelinin performansına ince ayar yapmak için kullanışlı bir yaklaşımdır. Farklı makine öğrenimi algoritmaları arasında seçim yapmak istiyorsak, önerilen bir diğer yaklaşım da iç içe **çapraz doğrulamadır**. Hata tahminindeki yanlılık üzerine güzel bir çalışmada Sudhir Varma ve Richard Simon, iç içe çapraz doğrulama kullanıldığında tahminin gerçek hatasının test veri setine göre neredeyse yansız olduğu sonucuna varmıştır (*Bias in Error Estimation When Using Cross-Validation for Model Selection* by *S. Varma* and *R. Si- mon*, *BMC Bioinformatics,* 7(1): 91, 2006, https://bmcbioinformatics.biomedcentral.com/articl es/10.1186/1471-2105-7-91).

 olimpikbarbell@gmail.com 

İç içe çapraz doğrulamada, verileri eğitim ve test katlarına ayırmak için bir dış k-kat çapraz doğrulama döngümüz vardır ve eğitim katında k-kat çapraz doğrulama kullanarak modeli seçmek için bir iç döngü kullanılır. Model seçiminden sonra, model performansını değerlendirmek için test katlaması kullanılır. *Şekil 6.8,* hesaplama performansının önemli olduğu büyük veri kümeleri için faydalı olabilecek, yalnızca beş dış ve iki iç katlamalı iç içe çapraz doğrulama kavramını açıklamaktadır; bu özel iç içe çapraz doğrulama türü **5×2 çapraz doğrulama** olarak da bilinir:




*Şekil 6.8: İç içe çapraz doğrulama kavramı*

Scikit-learn'de grid arama ile iç içe çapraz doğrulamayı aşağıdaki gibi gerçekleştirebiliriz:

```
>> param_aralığı = [0.0001, 0.001, 0.01, 0.1,
...                 1.0, 10.0, 100.0, 1000.0]
>> param_grid = [{'svc C': param_range,
...                 'svc kernel': ['linear']},
...                {'svc C': param_range,
...                 'svc gamma': param_range,
...                 'svc kernel': ['rbf']}]
>>> gs = GridSearchCV(tahmin edici=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=2)
```

```
>>> scores = cross_val_score(gs, X_train, y_train,
...                          scoring='accuracy', cv=5)
>> print(f'CV doğruluğu: {np.mean(scores):.3f} '
...       f'+/- {np.std(scores):.3f}')
CV doğruluğu: 0,974 +/- 0,015
```

Geri dönen ortalama çapraz doğrulama doğruluğu, bir modelin hiperparametrelerini ayarlarsak ve onu görünmeyen veriler üzerinde kullanırsak ne beklememiz gerektiği konusunda bize iyi bir tahmin verir.

Örneğin, bir SVM modelini basit bir karar ağacı sınıflandırıcısıyla karşılaştırmak için iç içe çapraz doğrulama yaklaşımını kullanabiliriz; basitlik için sadece derinlik parametresini ayarlayacağız:

```
>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(
...     estimator=DecisionTreeClassifier(random_state=0),
...     param_grid=[{'max_depth': [1, 2, 3, 4, 5, 6, 7, Yok]}],
...     scoring='accuracy',
...     cv=2
... )
>>> scores = cross_val_score(gs, X_train, y_train,
...                          scoring='accuracy', cv=5)
>> print(f'CV doğruluğu: {np.mean(scores):.3f} '
...       f'+/- {np.std(scores):.3f}')
CV doğruluğu: 0,934 +/- 0,016
```

Gördüğümüz gibi, DVM modelinin iç içe çapraz doğrulama performansı (yüzde 97,4), karar ağacının performansından (yüzde 93,4) önemli ölçüde daha iyidir ve bu nedenle, bu özel veri kümesiyle aynı popülasyondan gelen yeni verileri sınıflandırmak için daha iyi bir seçim olmasını bekleriz.

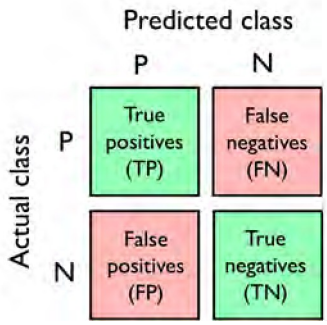
# Farklı performans değerlendirmelerine bakmak metrikler

Önceki bölümlerde ve kısımlarda, genel olarak bir modelin performansını ölçmek için kullanışlı bir metrik olan tahmin doğruluğunu kullanarak farklı makine öğrenimi modellerini değerlendirdik. Ancak, bir modelin uygunluğunu ölçmek için kullanılabilecek hassasiyet, geri çağırma, **F1 puanı** ve **Matthews korelasyon katsayısı** (**MCC**) gibi başka performans ölçütleri de vardır.

## Karışıklık matrisini okuma

Farklı puanlama metriklerinin ayrıntılarına girmeden önce, bir öğrenme algoritmasının performansını ortaya koyan bir **matris** olan **karışıklık matrisine** bir göz atalım.

Karışıklık matrisi, *Şekil 6.9*'da gösterildiği gibi, bir sınıflandırıcının **doğru pozitif** (**TP**), **doğru negatif** (**TN**), **yanlış pozitif** (**FP**) ve **yanlış negatif** (**FN**) tahminlerinin sayılarını bildiren basit bir kare matristir:




*Şekil 6.9: Karışıklık matrisi*

Bu metrikler gerçek ve tahmin edilen sınıf etiketlerini karşılaştırarak manuel olarak kolayca hesaplanabilse de, scikit-learn aşağıdaki gibi kullanabileceğimiz kullanışlı bir confusion_matrix işlevi

```
>> from sklearn.metrics import confusion_matrix
>> pipe_svc.fit(X_train, y_train)
>> y_pred = pipe_svc.predict(X_test)
>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
>> print(confmat)
[[71 1]
[ 2 40]]
```

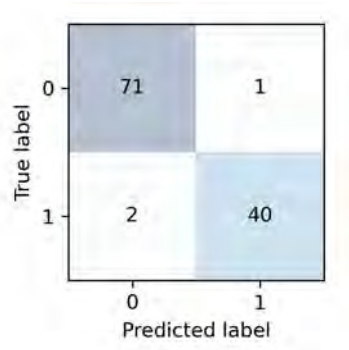
sağlar:

Kod çalıştırıldıktan sonra döndürülen dizi bize sınıflandırıcının test veri kümesinde yaptığı farklı hata türleri hakkında bilgi sağlar. Bu bilgileri Matplotlib'in matshow fonksiyonunu kullanarak *Şekil 6.9*'daki karışıklık matrisi çizimine eşleyebiliriz:

```
>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i, s=confmat[i, j],
...                 va='center', ha='center')
> >  ax.xaxis.set_ticks_position('bottom')
>> plt.xlabel('Tahmin edilen etiket')
>> plt.ylabel('Gerçek etiket')
>> plt.show()
```

6587191CA584E

Şimdi, eklenen etiketlerle birlikte aşağıdaki karışıklık matrisi grafiği, sonuçların yorumlanmasını biraz daha kolaylaştıracaktır:




*Şekil 6.10: Verilerimiz için bir karışıklık matrisi*

Bu örnekte sınıf 1'in (kötü huylu) pozitif sınıf olduğunu varsayan modelimiz, sırasıyla sınıf 0'a (TN) ait örneklerin 71'ini ve sınıf 1'e (TP) ait örneklerin 40'ını doğru sınıflandırmıştır. Bununla birlikte, modelimiz sınıf 1'den iki örneği sınıf 0 (FN) olarak yanlış sınıflandırmış ve bir örneğin iyi huylu bir tümör (FP) olmasına rağmen kötü huylu olduğunu tahmin etmiştir. Bir sonraki alt bölümde, bu bilgileri çeşitli hata metriklerini hesaplamak için nasıl kullanabileceğimizi öğreneceğiz.

## Bir sınıflandırma modelinin kesinliğini ve geri çağırmasını optimize etme

Hem tahmin **hatası** (**ERR**) hem de **doğruluk** (**ACC**) kaç örneğin yanlış sınıflandırıldığı hakkında genel bilgi sağlar. Hata, tüm yanlış tahminlerin toplamının toplam tahmin sayısına bölünmesi olarak anlaşılabilir ve doğruluk, sırasıyla, doğru tahminlerin toplamının toplam tahmin sayısına bölünmesi olarak hesaplanır:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

Tahmin doğruluğu daha sonra doğrudan hatadan hesaplanabilir:

$$ACC\ A\ \frac{TP + TN}{FP + FN + TP + TN}\ A\ 1 - ERR$$

**Gerçek pozitif oranı** (**TPR**) ve **yanlış pozitif oranı** (**FPR),** özellikle dengesiz sınıf problemleri için yararlı olan performans ölçütleridir:

$$FPR\ F\ \frac{FP}{N}\ F\ \frac{FP}{FP + TN}$$

$$TPR\ F\ \frac{TP}{P}\ F\ \frac{TP}{FN + TP}$$

 olimpikbarbell@gmail.com 

Örneğin tümör teşhisinde, hastaya uygun tedavi konusunda yardımcı olmak için kötü huylu tümörlerin tespit edilmesiyle daha çok ilgileniyoruz. Ancak, hastaları gereksiz yere endişelendirmemek için yanlışlıkla kötü huylu (FP) olarak sınıflandırılan iyi huylu tümörlerin sayısını azaltmak da önemlidir. FPR'nin aksine TPR, toplam pozitifler (P) havuzundan doğru şekilde tanımlanan pozitif (veya ilgili) örneklerin oranı hakkında faydalı bilgiler sağlar.

**Kesinlik** (**PRE**) ve **geri çağırma** (**REC)** performans ölçümleri bu TP ve TN oranlarıyla ilişkilidir ve aslında REC, TPR ile eş anlamlıdır:

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

Başka bir deyişle, geri çağırma, ilgili kayıtlardan (pozitifler) kaç tanesinin bu şekilde yakalandığını (gerçek pozitifler) ölçer. Kesinlik ise ilgili olarak tahmin edilen kayıtların (doğru ve yanlış pozitiflerin toplamı) kaçının gerçekten ilgili olduğunu (doğru pozitifler) ölçer:

$$PRE\ P\frac{TP}{TP + FP}$$

Kötü huylu tümör tespiti örneğine dönecek olursak, geri çağırma için optimizasyon yapmak kötü huylu bir tümörün tespit edilememe olasılığını en aza indirmeye yardımcı olur. Ancak bu, hastalar sağlıklı olmasına rağmen (yüksek sayıda FP) hastalarda kötü huylu tümörlerin tahmin edilmesi pahasına gerçekleşir. Öte yandan, öngörü için optimizasyon yaparsak, bir hastanın kötü huylu tümörü olduğunu tahmin edersek doğruluğu vurgularız. Ancak bunun bedeli kötü huylu tümörleri daha sık gözden kaçırmaktır (yüksek sayıda FN).

PRE ve REC optimizasyonunun yukarı ve aşağı yönlerini dengelemek için, PRE ve REC'in harmonik ortalaması, yani F1 puanı kullanılır:

$$F1\ F\ F\frac{P\ RE\ P\ REC}{PRE + REC}$$

**Kesinlik ve geri çağırma hakkında daha fazla okuma**

Kesinlik ve geri çağırma gibi farklı performans ölçütleri hakkında daha kapsamlı bir tartışma ile ilgileniyorsanız David M. W. Powers'ın *Evaluation adlı* teknik raporunu okuyun: *Hassasiyet, Geri Çağırma ve F-Faktöründen ROC, Bilgilendirilmişlik, İşaretlenmişlik ve Korelasyona,* https://arxiv.org/abs/2010.16061 adresinden ücretsiz olarak erişilebilir.

Son olarak, bir karışıklık matrisini özetleyen bir ölçü, özellikle biyolojik araştırma bağlamlarında popüler olan MCC'dir. MCC aşağıdaki gibi hesaplanır:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

6587191CA584E

PRE, REC ve F1 skorunun aksine, MCC -1 ile 1 arasında değişir ve bir karışıklık matrisinin tüm öğelerini hesaba katar - örneğin, F1 skoru TN'yi içermez. MCC değerlerinin yorumlanması F1 skorundan daha zor olsa da, aşağıdaki makalede açıklandığı gibi daha üstün bir metrik olarak kabul edilmektedir: *Matthews korelasyon katsayısının (MCC) F1 skoruna* göre *avantajları ve ikili sınıflandırma değerlendirmesinde doğruluk*, *D. Chicco* ve *G. Jurman, BMC Genomics.* s. 281-305, 2012, https://bmcgenomics.biomedcentral.com/articles/10.1186/s12864-019-6413-7.

Bu puanlama metriklerinin tümü scikit-learn'de uygulanmaktadır ve aşağıdaki kod parçasında gösterildiği gibi sklearn. metrics modülünden içe aktarılabilir:

```
>> from sklearn.metrics import precision_score
>> from sklearn.metrics import recall_score, f1_score
>> from sklearn.metrics import matthews_corrcoef

>> pre_val = precision_score(y_true=y_test, y_pred=y_pred)
>> print(f'Hassasiyet: {pre_val:.3f}')
Hassasiyet: 0,976
>>> rec_val = recall_score(y_true=y_test, y_pred=y_pred)
>> print(f'Recall: {rec_val:.3f}')
Recall: 0.952
>> f1_val = f1_score(y_true=y_test, y_pred=y_pred)
>> print(f'F1: {f1_val:.3f}') F1:
0.964
>> mcc_val = matthews_corrcoef(y_true=y_test, y_pred=y_pred)
>> print(f'MCC: {mcc_val:.3f}')
MCC: 0.943
```

Ayrıca, GridSearchCV'de scoring parametresi aracılığıyla doğruluktan farklı bir puanlama ölçütü kullanabiliriz. Puanlama parametresi tarafından kabul edilen farklı değerlerin tam bir listesi http://scikit-learn.org/stable/modules/model_evaluation.html adresinde bulunabilir.

Scikit-learn'deki pozitif sınıfın sınıf 1 olarak etiketlenen sınıf olduğunu unutmayın. Farklı bir *pozitif etiket* belirtmek istersek, make_scorer fonksiyonu aracılığıyla kendi puanlayıcımızı oluşturabiliriz ve bu puanlayıcıyı daha sonra doğrudan GridSearchCV'deki puanlama parametresine bir argüman olarak sağlayabiliriz (bu örnekte, f1_score'u bir metrik olarak kullanarak):

```
>> from sklearn.metrics import make_scorer
>> c_gamma_range = [0.01, 0.1, 1.0, 10.0]
>> param_grid = [{'svc C': c_gamma_range,
...                 'svc kernel': ['linear']},
...                {'svc C': c_gamma_range,
...                 'svc gamma': c_gamma_range,
...                 'svc kernel': ['rbf']}]
```

olimpikbarbell@gmail.com

```
>>> skorer = make_scorer(f1_score, pos_label=0)
>>> gs = GridSearchCV(tahmin edici=pipe_svc,
...                   param_grid=param_grid,
...                   skor=skorer,
...                   cv=10)
>>> gs = gs.fit(X_train, y_train)
>> print(gs.best_score_) 0.986202145696
>> print(gs.best_params_)
{'svc C': 10.0, 'svc gamma': 0.01, 'svc kernel': 'rbf'}
```

## Alıcı işletim karakteristiği çizimi

**Alıcı işletim karakteristiği** (**ROC**) grafikleri, sınıflandırıcının karar eşiğinin kaydırılmasıyla hesaplanan FPR ve TPR'ye göre performanslarına dayalı olarak sınıflandırma için model seçmek için yararlı araçlardır. Bir ROC grafiğinin köşegeni *rastgele tahmin* olarak yorumlanabilir ve köşegenin altına düşen sınıflandırma modelleri rastgele tahminden daha kötü olarak kabul edilir. Mükemmel bir sınıflandırıcı, 1 TPR ve 0 FPR ile grafiğin sol üst köşesine düşecektir. ROC eğrisine dayanarak, bir sınıflandırma modelinin performansını karakterize etmek için **eğri altındaki ROC alanını** (ROC **AUC**) hesaplayabiliriz.

ROC eğrilerine benzer şekilde, bir sınıflandırıcının farklı olasılık eşikleri için kesinlik-geri **çağırma eğrilerini** hesaplayabiliriz. Bu hassasiyet-hatırlama eğrilerini çizmek için bir fonksiyon da scikit-learn'de uygulanmaktadır ve http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html adresinde belgelenmiştir.

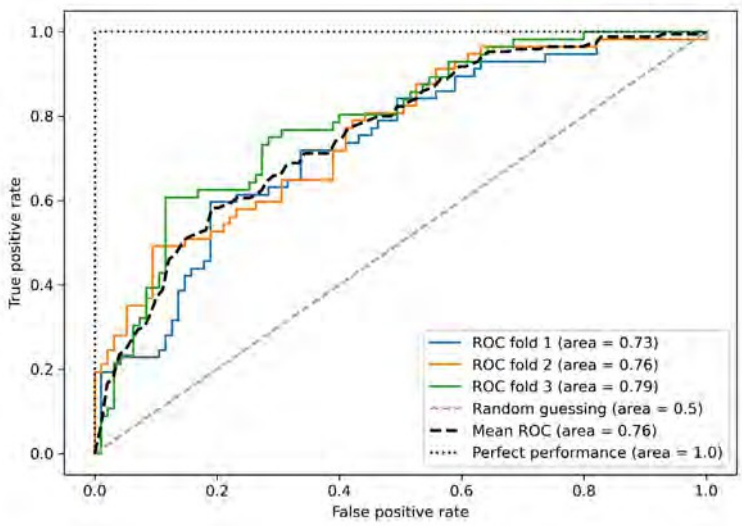
Aşağıdaki kod örneğini uygulayarak, bir tümörün iyi huylu mu yoksa kötü huylu mu olduğunu tahmin etmek için Meme Kanseri Wisconsin veri kümesinden yalnızca iki özellik kullanan bir sınıflandırıcının ROC eğrisini çizeceğiz. Daha önce tanımladığımız aynı lojistik regresyon hattını kullanacak olsak da, bu sefer sadece iki özellik kullanıyoruz. Bunun amacı, diğer özelliklerde bulunan faydalı bilgileri saklayarak sınıflandırma görevini sınıflandırıcı için daha zorlu hale getirmektir, böylece ortaya çıkan ROC eğrisi görsel olarak daha ilginç hale gelir. Benzer nedenlerden dolayı, StratifiedKFold doğrulayıcısındaki kat sayısını da üçe indiriyoruz. Kod

```
>> from sklearn.metrics import roc_curve, auc
>> from numpy import interp
>> boru_lr = make_pipeline(
...     StandardScaler(),
...     PCA(n_components=2),
...     LogisticRegression(penalty='l2', random_state=1,
...                        solver='lbfgs', C=100.0)
... )
>> X_train2 = X_train[:, [4, 14]]
```

aşağıdaki gibidir:

```
>>  cv = list(StratifiedKFold(n_splits=3).split(X_train, y_train))
>>> fig = plt.figure(figsize=(7, 5))
>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>> all_tpr = []
>> for i, (train, test) in enumerate(cv):
...         probas = pipe_lr.fit(
...             X_train2[train],
...             y_train[train]
...         ).predict_proba(X_train2[test])
...         fpr, tpr, thresholds = roc_curve(y_train[test],
...                                          probas[:, 1],
...                                          pos_label=1)
...         mean_tpr += interp(mean_fpr, fpr, tpr)
...         mean_tpr[0] = 0.0
...         roc_auc = auc(fpr, tpr)
...         plt.plot(fpr,
...                  tpr,
...                  label=f'ROC fold {i+1} (area = {roc_auc:.2f})')
>> plt.plot([0, 1],
...             [0, 1],
...             linestyle='--',
...             color=(0.6, 0.6, 0.6),
...             label='Rastgele tahmin (alan=0.5)')
>>> mean_tpr /= len(cv)
>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
>> plt.plot(mean_fpr, mean_tpr, 'k--',
...             label=f'Ortalama ROC (alan = {mean_auc:.2f})', lw=2)
>> plt.plot([0, 0, 1],
...             [0, 1, 1],
...             linestyle=':',
...             color='black',
...             label='Mükemmel performans (alan=1.0)')
>> plt.xlim([-0.05, 1.05])
>> plt.ylim([-0,05, 1,05])
>> plt.xlabel('Yanlış pozitif oranı')
>> plt.ylabel('Gerçek pozitif oranı')
>> plt.legend(loc='sağ alt')
>> plt.show()
```

Önceki kod örneğinde, scikit-learn'den zaten aşina olduğumuz StratifiedKFold sınıfını kullandık ve sklearn.metrics modülündeki roc_curve fonksiyonunu kullanarak pipe_lr boru hattımızdaki LogisticRegression sınıflandırıcısının ROC performansını her iterasyon için ayrı ayrı hesapladık. Ayrıca, NumPy'den içe aktardığımız interp fonksiyonu aracılığıyla üç kattan elde edilen ortalama ROC eğrisini enterpole ettik ve auc fonksiyonu aracılığıyla eğrinin altındaki alanı hesapladık. Elde edilen ROC eğrisi, farklı katlar arasında belirli bir derecede varyans olduğunu ve ortalama ROC AUC'nin (0,76) mükemmel bir puan (1,0) ile rastgele tahmin (0,5) arasında olduğunu göstermektedir:




*Şekil 6.11: ROC grafiği*

Sadece ROC AUC skoru ile ilgileniyorsak, önceki bölümlerde tanıtılan diğer skorlama fonksiyonlarına (örneğin, precision_score) benzer şekilde kullanılabilen roc_auc_ score fonksiyonunu sklearn.metrics alt modülünden doğrudan içe aktarabileceğimizi de unutmayın.

Bir sınıflandırıcının performansının ROC AUC olarak raporlanması, sınıflandırıcının dengesiz örneklere göre performansı hakkında daha fazla bilgi sağlayabilir. Bununla birlikte, doğruluk puanı bir ROC eğrisi üzerinde tek bir kesme noktası olarak yorumlanabilirken, A. P. Bradley, ROC AUC ve doğruluk ölçümlerinin çoğunlukla birbiriyle uyumlu olduğunu göstermiştir: *The Use of the Area Under the ROC Curve in the Evaluation of Ma- chine Learning Algorithms* by *A. P. Bradley*, *Pattern Recognition,* 30(7): 1145-1159, 1997, https://reader. elsevier.com/reader/sd/pii/S0031320396001422.

## Çok sınıflı sınıflandırma için puanlama metrikleri

Şimdiye kadar tartıştığımız puanlama metrikleri ikili sınıflandırma sistemlerine özgüdür. Ancak scikit-learn, bu puanlama metriklerini **bire karşı hepsi (OvA)** sınıflandırması yoluyla çok sınıflı problemlere genişletmek için makro ve mikro ortalama yöntemlerini de uygular. Mikro ortalama, sistemin bireysel TP'leri, TN'leri, FP'leri ve FN'lerinden hesaplanır. Örneğin, *k-sınıflı* bir sistemdeki kesinlik puanının mikro ortalaması aşağıdaki gibi hesaplanabilir:

$$PREmicro = \frac{TP1 + \cdots + TPkk}{TP_1 + \cdots + TPkk + FP_1 + \cdots + FPkk}$$

Makro-ortalama basitçe farklı sistemlerin ortalama puanları olarak hesaplanır:

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

Mikro-ortalama, her bir örneği veya tahmini eşit olarak ağırlıklandırmak istediğimizde kullanışlıdır; makro-ortalama ise bir sınıflandırıcının genel performansını en sık kullanılan sınıf etiketlerine göre değerlendirmek için tüm sınıfları eşit olarak ağırlıklandırır.

Scikit-learn'de çok sınıflı sınıflandırma modellerini değerlendirmek için ikili performans ölçütleri kullanıyorsak, varsayılan olarak makro-ortalamanın normalleştirilmiş veya ağırlıklandırılmış bir çeşidi kullanılır. Ağırlıklı makro ortalama, ortalama hesaplanırken her bir sınıf etiketinin puanının gerçek örnek sayısına göre ağırlıklandırılmasıyla hesaplanır. Ağırlıklı makro-ortalama, sınıf dengesizlikleri, yani her etiket için farklı sayıda örnekle uğraşıyorsak kullanışlıdır.

Ağırlıklı makro ortalama, scikit-learn'deki çok sınıflı problemler için varsayılan olsa da, `sklearn.metrics` modülünden içe aktardığımız farklı puanlama fonksiyonlarının içindeki ortalama parametresi aracılığıyla `ortalama` yöntemini belirleyebiliriz, örneğin `precision_score` veya

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                          pos_label=1,
...                          greater_is_better=True,
...                          average='micro')
```

`make_scorer` fonksiyonları:

## Sınıf dengesizliği ile başa çıkma

Bu bölüm boyunca birkaç kez sınıf dengesizliklerinden bahsettik, ancak bu tür senaryoların ortaya çıkması durumunda bunlarla nasıl başa çıkılacağını tartışmadık. Sınıf dengesizliği, gerçek dünya verileriyle çalışırken oldukça yaygın bir sorundur - bir sınıftan veya birden fazla sınıftan örneklerin bir veri kümesinde aşırı temsil edilmesi. Bunun meydana gelebileceği spam filtreleme, dolandırıcılık tespiti veya hastalık taraması gibi çeşitli alanlar düşünebiliriz.

Bu bölümde üzerinde çalıştığımız Meme Kanseri Wisconsin veri setinin yüzde 90 oranında sağlıklı hastalardan oluştuğunu düşünün. Bu durumda, denetimli bir makine öğrenimi algoritmasının yardımı olmadan tüm örnekler için sadece çoğunluk sınıfını (iyi huylu tümör) tahmin ederek test veri kümesinde yüzde 90 doğruluk elde edebiliriz. Dolayısıyla, böyle bir veri kümesi üzerinde yaklaşık yüzde 90 test doğruluğu elde eden bir modeli eğitmek, modelimizin bu veri kümesinde sağlanan özelliklerden yararlı bir şey öğrenmediği anlamına gelir.

Bu bölümde, dengesiz veri setlerine yardımcı olabilecek bazı tekniklerin üzerinden kısaca geçeceğiz. Ancak bu soruna yaklaşmak için farklı yöntemleri tartışmadan önce, başlangıçta 357 iyi huylu tümör (sınıf `0`) ve 212 kötü huylu tümörden (sınıf `1`) oluşan veri kümemizden dengesiz bir veri kümesi oluşturalım:

```
>> X_imb = np.vstack((X[y == 0], X[y == 1][:40]))
>> y_imb = np.hstack((y[y == 0], y[y == 1][:40]))
```

 olimpikbarbell@gmail.com 

Bu kod parçacığında, 357 iyi huylu tümör örneğinin tamamını aldık ve bunları ilk 40 kötü huylu örnekle üst üste koyarak keskin bir sınıf dengesizliği yarattık. Her zaman çoğunluk sınıfını (iyi huylu, sınıf 0) tahmin eden bir modelin doğruluğunu hesaplayacak olsaydık, yaklaşık yüzde 90'lık bir tahmin doğruluğu elde ederdik:

```
>> y_pred = np.zeros(y_imb.shape[0])
>>> np.mean(y_pred == y_imb) * 100
89.92443324937027
```

Bu nedenle, sınıflandırıcıları bu tür veri kümelerine yerleştirdiğimizde, farklı modelleri karşılaştırırken doğruluktan ziyade hassasiyet, geri çağırma, ROC eğrisi gibi uygulamamızda en çok önemsediğimiz diğer ölçütlere odaklanmak mantıklı olacaktır. Örneğin, önceliğimiz ek bir tarama önermek için kötü huylu kanserli hastaların çoğunu tespit etmek olabilir, bu nedenle geri çağırma tercih ettiğimiz metrik olmalıdır. Sistem çok emin değilse e-postaları spam olarak etiketlemek istemediğimiz spam filtrelemede, hassasiyet daha uygun bir ölçüt olabilir.

Sınıf dengesizliği, makine öğrenimi modellerini değerlendirmenin yanı sıra, model uydurma sırasında bir öğrenme algoritmasını da etkiler. Makine öğrenimi algoritmaları tipik olarak, uydurma sırasında gördüğü eğitim örnekleri üzerinden bir toplam olarak hesaplanan bir ödül veya kayıp fonksiyonunu optimize ettiğinden, karar kuralı muhtemelen çoğunluk sınıfına doğru önyargılı olacaktır.

Başka bir deyişle, algoritma, eğitim sırasında kaybı en aza indirmek veya ödülü en üst düzeye çıkarmak için veri kümesindeki en bol sınıfa dayalı tahminleri optimize eden bir modeli dolaylı olarak öğrenir.

Model uydurma sırasında dengesiz sınıf oranlarıyla başa çıkmanın bir yolu, azınlık sınıfındaki yanlış tahminlere daha büyük bir ceza atamaktır. Scikit-learn aracılığıyla, böyle bir cezayı ayarlamak, class_weight parametresini class_weight='balanced' olarak ayarlamak kadar kolaydır, bu da çoğu sınıflandırıcı için uygulanmaktadır.

Sınıf dengesizliği ile başa çıkmak için kullanılan diğer popüler stratejiler arasında azınlık sınıfını yukarı örnekleme, çoğunluk sınıfını aşağı örnekleme ve sentetik eğitim örnekleri oluşturma yer alır. Ne yazık ki, farklı sorun alanlarında en iyi sonucu veren evrensel olarak en iyi çözüm veya teknik yoktur. Bu nedenle, pratikte, belirli bir problem üzerinde farklı stratejilerin denenmesi, sonuçların değerlendirilmesi ve en uygun görünen tekniğin seçilmesi önerilir.

Scikit-learn kütüphanesi, veri kümesinden değiştirme ile yeni örnekler çekerek azınlık sınıfının yukarı örneklenmesine yardımcı olabilecek basit bir yeniden örnekleme işlevi uygular. Aşağıdaki kod, dengesiz Meme Kanseri Wisconsin veri setimizden (burada sınıf 1) azınlık sınıfını alacak ve sınıf etiketi 0 ile aynı sayıda örnek içerene kadar tekrar tekrar yeni örnekler çekecektir:

```
>> from sklearn.utils import resample
>> print('Daha önce sınıf 1 örneklerinin sayısı:',
...           X_imb[y_imb == 1].shape[0])
Daha önce sınıf 1 örneklerinin sayısı: 40
>> X_upsampled, y_upsampled = resample(
...               X_imb[y_imb == 1],
```

6587191CA584E

```
...             y_imb[y_imb == 1],
...             replace=True,
...             n_samples=X_imb[y_imb == 0].shape[0],
...             random_state=123)
>> print('1. sınıftan sonraki örnek sayısı:',
...           X_upsampled.shape[0])
Sonra sınıf 1 örneklerinin sayısı: 357
```

Yeniden örneklemeden sonra, aşağıdaki gibi dengeli bir veri kümesi elde etmek için orijinal sınıf 0 örneklerini yukarı örneklenmiş sınıf 1 alt kümesiyle yığabiliriz:

```
>>> X_bal = np.vstack((X[y == 0], X_upsampled))
>>> y_bal = np.hstack((y[y == 0], y_upsampled))
```

Sonuç olarak, oy çokluğuna dayalı bir tahmin kuralı yalnızca yüzde 50 doğruluk sağlayacaktır:

```
>> y_pred = np.zeros(y_bal.shape[0])
>> np.mean(y_pred == y_bal) * 100 50
```

Benzer şekilde, veri kümesinden eğitim örneklerini kaldırarak çoğunluk sınıfını aşağı örnekleyebiliriz. Yeniden örnekleme işlevini kullanarak aşağı örnekleme yapmak için, önceki kod örneğinde sınıf 1 etiketini sınıf 0 ile değiştirebiliriz veya tam tersini yapabiliriz.

**Sınıf dengesizliğini gidermek için yeni eğitim verileri oluşturma**

Sınıf dengesizliği ile başa çıkmak için kullanılan bir diğer teknik de bu kitabın kapsamı dışında olan sentetik eğitim örneklerinin oluşturulmasıdır. Muhtemelen sentetik eğitim verisi üretimi için en yaygın kullanılan algoritma Sentetik **Azınlık Aşırı Örnekleme Tekniğidir** (**SMOTE**) ve bu teknik hakkında daha fazla bilgiyi *Nitesh Chawla ve* diğerlerinin orijinal araştırma makalesinde bulabilirsiniz: SMOTE: Synthetic *Minority Over-sampling Technique*, *Journal of Artificial Intelligence Research,* 16: 321-357, 2002, https://www.jair. org/index.php/jair/article/view/10302 adresinde mevcuttur. Ayrıca, SMOTE uygulaması da dahil olmak üzere tamamen dengesiz veri kümelerine odaklanan bir Python kütüphanesi olan imbalanced-learn'e göz atmanız şiddetle tavsiye edilir. Imbalanced-learn hakkında daha fazla bilgiyi https://github.com/scikit-learn-contrib/imbalanced-learn adresinden edinebilirsiniz.

# Özet

Bu bölümün başında, makine öğrenimi modellerini daha verimli bir şekilde eğitmemize ve değerlendirmemize yardımcı olan uygun model ardışık düzenlerinde farklı dönüştürme tekniklerini ve sınıflandırıcıları nasıl zincirleyeceğimizi tartıştık. Daha sonra, model seçimi ve değerlendirmesi için temel tekniklerden biri olan k-kat çapraz doğrulamayı gerçekleştirmek için bu ardışık düzenleri kullandık. K-kat çapraz doğrulamayı kullanarak, öğrenme algoritmalarının aşırı uyum ve yetersiz uyum gibi yaygın sorunlarını teşhis etmek için öğrenme ve doğrulama eğrilerini çizdik.

 olimpikbarbell@gmail.com 

Izgara arama, rastgele arama ve ardışık yarılama yöntemlerini kullanarak modelimize daha da ince ayar yaptık. Daha sonra, belirli problem görevleri için bir modelin performansını değerlendirmek ve optimize etmek için karışıklık matrislerini ve çeşitli performans ölçümlerini kullandık. Son olarak, birçok gerçek dünya uygulamasında yaygın bir sorun olan dengesiz verilerle başa çıkmak için farklı yöntemleri tartışarak bu bölümü sonlandırdık. Artık, sınıflandırma için denetimli makine öğrenimi modellerini başarılı bir şekilde oluşturmak için gerekli tekniklerle iyi bir şekilde donatılmış olmalısınız.

Bir sonraki bölümde, topluluk yöntemlerine bakacağız: bir makine öğrenimi sisteminin tahmin performansını daha da artırmak için birden fazla modeli ve sınıflandırma algoritmasını birleştirmemize olanak tanıyan yöntemler.

## Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.

# 7

# Topluluk Öğrenmesi için Farklı Modellerin Birleştirilmesi

Bir önceki bölümde, sınıflandırma için farklı modellerin ayarlanması ve değerlendirilmesine yönelik en iyi uygulamalara odaklandık. Bu bölümde, bu teknikler üzerine inşa edeceğiz ve genellikle tek tek üyelerinden herhangi birinden daha iyi bir tahmin performansına sahip olabilecek bir sınıflandırıcı kümesi oluşturmak için farklı yöntemleri keşfedeceğiz. Aşağıdakileri nasıl yapacağımızı öğreneceğiz:

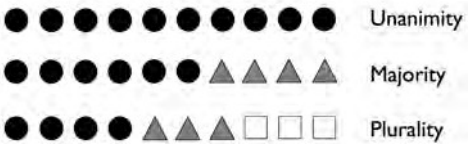
- Çoğunluk oylamasına dayalı tahminlerde bulunma
- Tekrarlama ile eğitim veri kümesinin rastgele kombinasyonlarını çizerek aşırı uyumu azaltmak için torbalama kullanın
- Hatalarından ders çıkaran zayıf öğrenicilerden güçlü modeller oluşturmak için güçlendirme uygulayın

## toplulukları ile öğrenme

**Topluluk yöntemlerinin** amacı, farklı sınıflandırıcıları tek başına her bir sınıflandırıcıdan daha iyi genelleme performansına sahip bir meta sınıflandırıcıda birleştirmektir. Örneğin, 10 uzmanın tahminlerini bir araya getirdiğimizi varsayarsak, topluluk yöntemleri, her bir uzmanın tahminlerinden daha doğru ve sağlam bir tahmin elde etmek için 10 uzmanın tahminlerini stratejik olarak birleştirmemize olanak tanıyacaktır. Bu bölümün ilerleyen kısımlarında göreceğiniz gibi, bir sınıflandırıcı topluluğu oluşturmak için birkaç farklı yaklaşım vardır. Bu bölümde, toplulukların nasıl çalıştığına ve neden genellikle iyi bir genelleme performansı sağladıkları kabul edildiğine dair temel bir açıklama sunulacaktır.

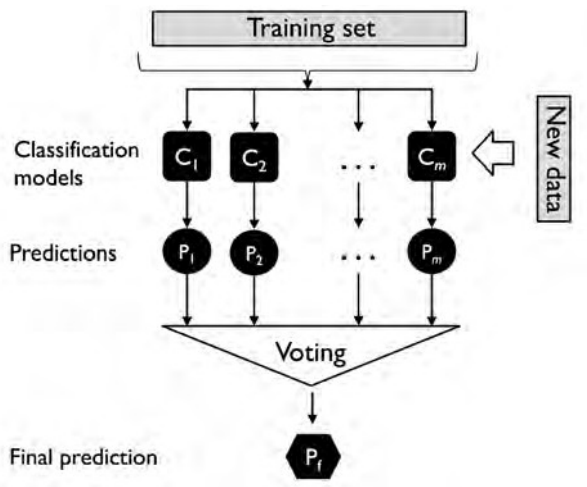
Bu bölümde, **çoğunluk oylaması** prensibini kullanan en popüler topluluk yöntemlerine odaklanacağız. Çoğunluk oylaması basitçe, sınıflandırıcıların çoğunluğu tarafından tahmin edilen, yani oyların yüzde 50'sinden fazlasını alan sınıf etiketini seçmemiz anlamına gelir. Açıkçası, "çoğunluk oylaması" terimi yalnızca ikili sınıf ayarlarını ifade eder. Bununla birlikte, çoğunluk oylaması ilkesini çok sınıflı ortamlara genellemek kolaydır, bu da **çoğulluk oylaması olarak** bilinir. (Birleşik Krallık'ta insanlar çoğunluk ve çokluk oylamasını sırasıyla "mutlak" ve "göreceli" çoğunluk terimleriyle birbirinden ayırmaktadır).

Burada, en çok oyu alan sınıf etiketini (mod) seçiyoruz. *Şekil 7.1,* her bir benzersiz sembolün (üçgen, kare ve daire) benzersiz bir sınıf etiketini temsil ettiği 10 sınıflandırıcıdan oluşan bir topluluk için çoğunluk ve çokluk oylaması kavramını göstermektedir:




*Şekil 7.1: Farklı oylama konseptleri*

Eğitim veri kümesini kullanarak, *m* farklı sınıflandırıcıyı ($C_1$ , ..., $C_m$ ) eğiterek başlarız. Tekniğe bağlı olarak, topluluk farklı sınıflandırma algoritmalarından oluşturulabilir, örneğin karar ağaçları, destek vektör makineleri, lojistik regresyon sınıflandırıcıları vb. Alternatif olarak, eğitim veri kümesinin farklı alt kümelerine uyan aynı temel sınıflandırma algoritmasını da kullanabiliriz. Bu yaklaşımın öne çıkan örneklerinden biri, *Bölüm 3, Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu'nda* ele aldığımız farklı karar ağacı sınıflandırıcılarını birleştiren rastgele orman *algoritmasıdır. Şekil 7.2,* çoğunluk oylamasını kullanan genel bir topluluk yaklaşımı kavramını göstermektedir:




*Şekil 7.2: Genel bir topluluk yaklaşımı*

Basit çoğunluk veya çokluk oylaması yoluyla bir sınıf etiketini tahmin etmek için, her bir sınıflandırıcının tahmin edilen sınıf etiketlerini, $C_j$ , birleştirebilir ve en çok oyu alan sınıf etiketini, *yy*, seçebiliriz:

$$yy\ yyyy\{C1(xx)\ ,\ _{C2}(xx)\ ,\ldots\ ,\ _{Cm}(xx)\}$$

(İstatistikte mod, bir kümede en sık görülen olay veya sonuçtur. Örneğin, mod{1, 2, 1, 1, 2, 4, 5, 4} = 1.)

Örneğin, sınıf1 = -1 ve sınıf2 = +1 olan ikili bir sınıflandırma görevinde, çoğunluk oyu tahminini aşağıdaki gibi yazabiliriz:

$$C(x) = \text{sign}\left[\sum_{j}^{m} C_j(x)\right] = \begin{cases} \sum_j C_j(x) \geq 0 \text{ ise } 1 \\ -1 \text{ aksi takdirde} \end{cases}$$

Topluluk yöntemlerinin neden tek başına sınıflandırıcılardan daha iyi çalışabileceğini göstermek için bazı kombinatorik kavramları uygulayalım. Aşağıdaki örnekte, ikili sınıflandırma görevi için tüm *n-tabanlı* sınıflandırıcıların eşit hata oranına sahip olduğu varsayımını yapacağız, $\varepsilon$. Ayrıca, sınıflandırıcıların bağımsız olduğunu ve hata oranlarının ilişkili olmadığını varsayacağız. Bu varsayımlar altında, bir temel sınıflandırıcılar topluluğunun hata olasılığını basitçe bir binom dağılımının olasılık kütle fonksiyonu olarak ifade edebiliriz:

$$P(y \; y_k) = \sum_{k}^{n} \binom{n}{k} \varepsilon^{k} (1 - \varepsilon)^{n-k} = \varepsilon_{\text{TOPLULUK}}$$

Burada $\binom{n}{k}$ *n'nin k'yı seçtiği* binom katsayısıdır. Başka bir deyişle, topluluğun tahmininin yanlış olma olasılığını hesaplarız. Şimdi, her sınıflandırıcının 0,25 hata oranına sahip olduğu ($\varepsilon = 0.25$) 11 temel sınıflandırıcıdan ($n = 11$) oluşan daha somut bir örneğe bakalım:

$$P(y \; y_k) = \sum_{k=k}^{11} \binom{11}{k} 0.25^{k} (1 - 0.25)^{11-k} = 0,034$$

**Binom katsayısı**

Binom katsayısı, $n$ büyüklüğündeki bir kümeden $k$ sırasız elemanın alt kümelerini seçebileceğimiz yolların sayısını ifade eder; bu nedenle genellikle "n k'yı seç" olarak adlandırılır. Burada sıra önemli olmadığından, binom katsayısı bazen *kombinasyon* veya *kombinatoryal sayı* olarak da adlandırılır ve kısaltılmamış haliyle aşağıdaki gibi yazılır:

$$\frac{n!}{(n-k)!\,k!}$$

Burada (!) sembolü faktöriyel anlamına gelir - örneğin, 3! = 3×2×1 = 6.

Gördüğünüz gibi, tüm varsayımlar karşılandığında topluluğun hata oranı (0,034) her bir bağımsız sınıflandırıcının hata oranından (0,25) çok daha düşüktür. Bu basitleştirilmiş örnekte, çift sayıda sınıflandırıcı ($n$) tarafından yapılan 50-50'lik bir bölünmenin hata olarak değerlendirildiğini, oysa bunun yalnızca zamanın yarısı için doğru olduğunu unutmayın. Böyle idealist bir topluluk sınıflandırıcıyı farklı temel hata oranları aralığında bir temel sınıflandırıcıyla karşılaştırmak için

```
>> from scipy.special import tarak
>> import math
>> def ensemble_error(n_classifier, error):
```

Python'da olasılık kütle fonksiyonunu uygulayalım:

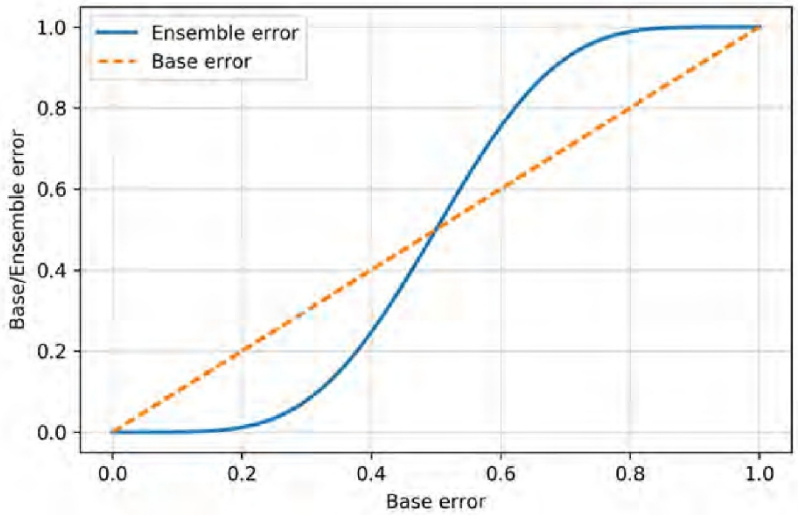
```
...     k_start = int(math.ceil(n_classifier / 2.))
...     probs = [comb(n_classifier, k) * hata**k
...                 *
...                 (1-error)**(n_classifier - k)
...                 for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>  ensemble_error(n_classifier=11, error=0.25)
0.03432750701904297
```

ensemble_error işlevini uyguladıktan sonra, topluluk ve temel hatalar arasındaki ilişkiyi bir çizgi grafikte görselleştirmek için 0,0 ila 1,0 arasındaki farklı temel hatalar için topluluk hata oranlarını hesaplayabiliriz:

```
>> import numpy as np
>> import matplotlib.pyplot as plt
>> hata_aralığı = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...                 for error in error_range]
>> plt.plot(error_range, ens_errors,
...             label='Topluluk hatası',
...             çizgi genişliği=2)
>> plt.plot(hata_aralığı, hata_aralığı,
...             linestyle='--', label='Temel hata',
...             çizgi genişliği=2)
>> plt.xlabel('Temel hata')
>> plt.ylabel('Base/Ensemble error')
>> plt.legend(loc='sol üst')
>>  plt.grid(alpha=0.5)
>> plt.show()
```

Ortaya çıkan grafikte görebileceğiniz gibi, temel sınıflandırıcılar rastgele tahminden ($\varepsilon\varepsilon$ $\varepsilon$ $\varepsilon\varepsilon\varepsilon$) daha iyi performans gösterdiği sürece, bir topluluğun hata olasılığı her zaman tek bir temel sınıflandırıcının hatasından daha iyidir.

*Y* ekseninin temel hatayı (noktalı çizgi) ve topluluk hatasını (sürekli çizgi) gösterdiğine dikkat edin:




*Şekil 7.3: Topluluk hatasının temel hataya karşı grafiği*

# Sınıflandırıcıları çoğunluk oyu ile birleştirme

Önceki bölümde topluluk öğrenmesine kısa bir giriş yaptıktan sonra, bir ısınma alıştırması ile başlayalım ve Python'da çoğunluk oylaması için basit bir topluluk sınıflandırıcısı uygulayalım.

**Çoğunluk oylaması**

Bu bölümde tartışacağımız çoğunluk oylama algoritması, çoklu oylama yoluyla çok sınıflı ortamlara da genelleştirilebilse de, literatürde sıklıkla olduğu gibi basitlik açısından "çoğunluk oylama" terimi kullanılacaktır.

## Basit bir çoğunluk oyu sınıflandırıcısının uygulanması

Bu bölümde uygulayacağımız algoritma, güven için bireysel ağırlıklarla ilişkili farklı sınıflandırma algoritmalarını birleştirmemize olanak tanıyacaktır. Amacımız, belirli bir veri kümesi üzerinde bireysel sınıflandırıcıların zayıflıklarını dengeleyen daha güçlü bir meta sınıflandırıcı oluşturmaktır. Daha kesin matematiksel terimlerle, ağırlıklı çoğunluk oylamasını aşağıdaki gibi yazabiliriz:

$$yy\,yy_{i}\,yy\sum_{jj1}^{m} wwj_{xxA}(Cj(xx)\,y\,ii$$

Burada, $w_j$ bir temel sınıflandırıcı ile ilişkili bir ağırlıktır, $C_j$ ; *yy*, topluluğun tahmin edilen sınıf etiketidir; *A,* benzersiz sınıf etiketleri kümesidir; $\chi\chi_A$ (Yunanca chi), *j.* sınıflandırıcının tahmin edilen sınıfı *i ile* eşleşirse ($C_j$ ***(x)*** = *i*) 1 döndüren karakteristik işlev veya gösterge işlevidir. Eşit ağırlıklar için bu denklemi basitleştirebilir ve aşağıdaki gibi yazabiliriz:

$$yy\ yyyy\{C1(xx)\ ,\ C2(xx)\ ,\ldots\ ,\ Cm\ (xx)\}$$

*Ağırlıklandırma* kavramını daha iyi anlamak için şimdi daha somut bir örneğe göz atacağız. $C_{jj}$ (*jj* {1, 2, 3}) olmak üzere üç temel sınıflandırıcıdan oluşan bir topluluğumuz olduğunu ve verilen bir ***x*** örneğinin $C_{jj}$ (*xx*) ∈ {0, 1} sınıf etiketini tahmin etmek istediğimizi varsayalım. Üç temel sınıflandırıcıdan ikisi 0 sınıf etiketini tahmin ederken biri, $C_3$ , örneğin 1 sınıfına ait olduğunu tahmin eder. Her bir temel sınıflandırıcının tahminlerini eşit olarak ağırlıklandırırsak, çoğunluk oyu örneğin 0 sınıfına ait olduğunu tahmin eder:

$$C1(xx) \rightarrow 0,\ C\ 2\ (xx) \rightarrow 0,\ C\ 3\ (xx) \rightarrow 1$$

$$yy\ yyyy\{0,\ 0,\ 1\}\ yy\ 0$$

Şimdi $C_3$ 'a 0,6 ağırlık atayalım ve $C_1$ ve $C_2$ 'yi 0,2 katsayısı ile ağırlıklandıralım:

$$yy\ yy\ yy \sum_{i}^{m} wwj\ \chi\chi_A(C_j(xx)\ y\ ii$$

$$y\ \ \underset{i}{yyyy}[0{,}2 \times {}_{i0} + 0{,}2 \times {}_{i0},\ 0{,}6 \times {}_{i1}]\ y\ 1$$

Daha basitçe, 3×0.2 = 0.6 olduğundan, $C_3$ tarafından yapılan tahminin $C_1$ veya $C_2$ tarafından yapılan tahminlerden üç kat daha fazla ağırlığa sahip olduğunu söyleyebiliriz, bunu aşağıdaki gibi yazabiliriz:

$$yy\ yyyy\{0,\ 0,\ 1,\ 1,\ 1\}\ yy\ 1$$

Ağırlıklı çoğunluk oyu kavramını Python koduna çevirmek için NumPy'nin uygun argmax ve bincount fonksiyonlarını kullanabiliriz, burada bincount her sınıf etiketinin oluşum sayısını sayar. Daha sonra argmax fonksiyonu, çoğunluk sınıf etiketine karşılık gelen en yüksek sayımın indeks konumunu döndürür (bu, sınıf etiketlerinin 0'dan başladığını varsayar):

```
>> import numpy as np
>> np.argmax(np.bincount([0, 0, 1],
...                 weights=[0.2, 0.2, 0.6]))
1
```

*Bölüm 3*'teki lojistik regresyon tartışmasından hatırlayacağınız gibi, scikit-learn'deki belirli sınıflandırıcılar predict_proba yöntemi aracılığıyla tahmin edilen bir sınıf etiketinin olasılığını da döndürebilir. Çoğunluk oylaması için sınıf etiketleri yerine tahmin edilen sınıf olasılıklarını kullanmak, topluluğumuzdaki sınıflandırıcılar iyi kalibre edilmişse faydalı olabilir. Olasılıklardan sınıf etiketlerini tahmin etmek için çoğunluk oylamasının değiştirilmiş versiyonu aşağıdaki gibi yazılabilir:

$$m$$

olimpikbarbell@gmail.com

$$yy\,yy\,\underset{i}{yy}\,\sum_{jj1} wwjjpijj$$

Burada, $p_{ij}$ j. sınıflandırıcının $i$. sınıf etiketi için öngörülen olasılığıdır.

Önceki örneğimize devam edecek olursak, sınıf etiketleri $i$ $i\{0, 1\}$ olan ikili bir sınıflandırma problemimiz ve üç sınıflandırıcıdan oluşan bir topluluğumuz olduğunu varsayalım, $C_{jj}$ $(jj\ \{1, 2, 3\})$. $C_j$ sınıflandırıcılarının belirli bir $\boldsymbol{x}$ örneği için aşağıdaki sınıf üyelik olasılıklarını döndürdüğünü varsayalım:

$$_{C1}(xx) \rightarrow [0.9, 0.1],\ _{C2}(xx) \rightarrow [0.8, 0.2],\ C3(xx) \rightarrow [0.4, 0.6]$$

Daha önce olduğu gibi aynı ağırlıkları (0.2, 0.2 ve 0.6) kullanarak, bireysel sınıf olasılıklarını aşağıdaki gibi hesaplayabiliriz:

$$p(i0|xx) = 0{,}2 \times 0{,}9 + 0{,}2 \times 0{,}8 + 0{,}6 \times 0{,}4 = 0{,}58$$

$$p(i1|xx) = 0{,}2 \times 0{,}1 + 0{,}2 \times 0{,}2 + 0{,}6 \times 0{,}6 = 0{,}42$$

$$yy = \underset{i}{yyy}\ yyy[p(i0|xx)\ , p(i1|xx)] = 0$$

Sınıf olasılıklarına dayalı ağırlıklı çoğunluk oylamasını uygulamak için, np.average ve np.argmax kullanarak yine NumPy'den yararlanabiliriz:

```
>> ex = np.array([[0.9, 0.1],
...                [0.8, 0.2],
...                [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0 . 2, 0.6])
>>> p
array([0.58, 0.42])
>> np.argmax(p) 0
```

Her şeyi bir araya getirerek, şimdi MajorityVoteClassifier'ı Python'da uygulayalım:

```
from s k l e a r n .base import BaseEstimator
from sklearn.base import ClassifierMixin
from s k l e a r n .preprocessing import LabelEncoder
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
ithalat operatörü
class MajorityVoteClassifier(BaseEstimator, ClassifierMixin):
    def init (self, classifiers, vote='classlabel', weights=None):

        self.classifiers = classifiers
        self.named_classifiers = {
            anahtar: anahtar için değer,
            value in _name_estimators(classifiers)
        }
        self.vote = vote
```

 olimpikbarbell@gmail.com 

```
        self.weights = weights

    def fit(self, X, y):
        if self.vote not in ('probability', 'classlabel'):
            raise ValueError(f "vote must be 'probability'
            "
                                f "or 'classlabel'"
                                f"; got (vote={self.vote})")
        eğer self.weights ve
        len(self.weights) != len(self.classifiers):
            raise ValueError(f'Sınıflandırıcı sayısı ve'
                                f' ağırlıkları eşit olmalıdır'
                                f'; got {len(self.weights)} weights,'
                                f' {len(self.classifiers)} classifiers')
        # Sınıf etiketlerinin # 0 ile başlamasını sağlamak
        için LabelEncoder kullanın, bu np.argmax için
        önemlidir
        # self.predict içinde çağrı
        self.lablenc_ = LabelEncoder()
        self.lablenc_.fit(y)
        self.classes_ = self.lablenc_.classes_
        self.classifiers_ = []
        for clf in self.classifiers:
            fitted_clf = clone(clf).fit(X,
                                    self.lablenc_.transform(y))
            self.classifiers_.append(fitted_clf)
        return self
```

Her bir parçayı açıklamak için koda çok sayıda yorum ekledik. Ancak, kalan yöntemleri uygulamadan önce, kısa bir ara verelim ve ilk başta kafa karıştırıcı görünebilecek bazı kodları tartışalım. Sınıflandırıcının parametrelerini ayarlamak ve döndürmek için get_params ve set_params yöntemlerinin yanı sıra tahmin doğruluğunu hesaplamak için score yöntemi de dahil olmak üzere bazı temel işlevleri *ücretsiz* olarak elde etmek için BaseEstimator ve ClassifierMixin ana sınıflarını kullandık.

Daha sonra, yeni bir MajorityVoteClassifier nesnesini vote='classlabel' ile başlatırsak, sınıf etiketlerini temel alan bir çoğunluk oylaması yoluyla sınıf etiketini tahmin etmek için predict yöntemini ekleyeceğiz. Alternatif olarak, sınıf üyeliği olasılıklarına dayalı olarak sınıf etiketini tahmin etmek için topluluk sınıflandırıcısını vote='probability' ile başlatabileceğiz. Ayrıca, **eğri altındaki alıcı işletim karakteristiği alanını** (**ROC AUC**) hesaplarken yararlı olan ortalama olasılıkları döndürmek için bir predict_proba yöntemi de ekleyeceğiz:

```
    def predict(self, X):
        if self.vote == 'probability':
            maj_vote = np.argmax(self.predict_proba(X), axis=1)
        else: # 'classlabel' oy
```

```
            # clf.predict çağrılarının sonuçlarını toplayın
            tahminler = np.asarray([
                clf.predict(X) for clf in self.classifiers_
            ]).T

            maj_vote = np.apply_along_axis(
                lambda x: np.argmax(
                    np.bincount(x, weights=self.weights)
                ),
                axis=1, arr=tahminler
            )
        maj_vote = self.lablenc_.inverse_transform(maj_vote)
        return maj_vote

    def predict_proba(self, X):
        probas = np.asarray([clf.predict_proba(X)
                             for clf in self.classifiers_])
        avg_proba = np.average(probas, axis=0,
                               weights=self.weights)
        return avg_proba

    def get_params(self,
        deep=True): if not deep:
            return super().get_params(deep=False)
        else:
            out = self.named_classifiers.copy()
            for name, step in self.named_classifiers.items(): for
                key, value in step.get_params(
                        deep=True).items():
                    out[f'{name} {key}'] = value
            geri dön
```

Ayrıca, topluluktaki bireysel sınıflandırıcıların parametrelerine erişmek için _name_ estimators işlevini kullanmak üzere get_params yönteminin kendi değiştirilmiş versiyonunu tanımladığımızı unutmayın; bu ilk başta biraz karmaşık görünebilir, ancak sonraki bölümlerde hiperpara- rametre ayarı için ızgara aramasını kullandığımızda mükemmel bir anlam ifade edecektir.

 olimpikbarbell@gmail.com 

**scikit-learn'de VotingClassifier**

MajorityVoteClassifier uygulaması gösterim amacıyla çok kullanışlı olsa da, bu kitabın ilk baskısındaki uygulamayı temel alarak scikit-learn'de bu çoğunluk oy sınıflandırıcısının daha sofistike bir versiyonunu uyguladık. Topluluk sınıflandırıcısı scikit-learn sürümünde sklearn.ensemble.VotingClassifier olarak mevcuttur 0.17 ve daha yeni sürümler. VotingClassifier hakkında daha fazla bilgiyi https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html adresinde bulabilirsiniz.

## Tahminlerde bulunmak için çoğunluk oylaması ilkesinin kullanılması

Şimdi sıra bir önceki bölümde uyguladığımız MajorityVoteClassifier'ı harekete geçirmeye geldi. Ama önce, üzerinde test edebileceğimiz bir veri kümesi hazırlayalım. CSV dosyalarından veri kümelerini yükleme tekniklerine zaten aşina olduğumuz için, kısa yoldan Iris veri kümesini scikit-learn'in veri kümeleri modülünden yükleyeceğiz. Ayrıca, örnekleme amacıyla sınıflandırma görevini daha zorlu hale getirmek için yalnızca iki özellik, *sepal genişlik* ve *pet- al uzunluğu* seçeceğiz. MajorityVoteClassifier'ımız çok sınıflı problemlere genelleştirilebilse de, yalnızca daha sonra ROC AUC'yi hesaplayacağımız Iris-versicolor ve Iris-virginica sınıflarından çiçek örneklerini sınıflandıracağız. Kod aşağıdaki gibidir:

```
>> from sklearn import veri kümeleri
>> from sklearn.model_selection import train_test_split
>> from sklearn.preprocessing import StandardScaler
>> from sklearn.preprocessing import LabelEncoder
>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>> le = LabelEncoder()
>> y = le.fit_transform(y)
```

**Karar ağaçlarından sınıf üyelik olasılıkları**

Scikit-learn'in ROC AUC puanını hesaplamak için predict_proba yöntemini (varsa) kullandığını unutmayın. *Bölüm 3*'te lojistik regresyon modellerinde sınıf olasılıklarının nasıl hesaplandığını gördük. Karar ağaçlarında olasılıklar, eğitim zamanında her düğüm için oluşturulan bir frekans vektöründen hesaplanır. Bu vektör, o düğümdeki sınıf etiketi dağılımından hesaplanan her bir sınıf etiketinin frekans değerlerini toplar. Daha sonra frekanslar, toplamları 1 olacak şekilde normalleştirilir. Benzer şekilde, k-en yakın komşu algoritmasında normalleştirilmiş sınıf etiketi frekanslarını döndürmek için k-en yakın komşuların sınıf etiketleri toplanır. Hem karar ağacı hem de k-en yakın komşu sınıflandırıcısı tarafından döndürülen normalleştirilmiş olasılıklar bir lojistik regresyon modelinden elde edilen olasılıklara benzese de, aslında bunların olasılık kütle fonksiyonlarından türetilmediğinin farkında olmalıyız.

Ardından, Iris örneklerini yüzde 50 eğitim ve yüzde 50 test verisi olarak ayıracağız:

```
>> X_train, X_test, y_train, y_test =\
...     train_test_split(X, y,
...                      test_size=0.5,
...                      random_state=1,
...                      stratify=y)
```

Eğitim veri kümesini kullanarak, şimdi üç farklı sınıflandırıcıyı eğiteceğiz:

- Lojistik regresyon sınıflandırıcı
- Karar ağacı sınıflandırıcısı
- k-en yakın komşu sınıflandırıcı

Daha sonra, her bir sınıflandırıcının model performansını, bunları bir topluluk sınıflandırıcıda birleştirmeden önce eğitim veri kümesi üzerinde 10 kat çapraz doğrulama yoluyla

```
>> from sklearn.model_selection import cross_val_score
>> from sklearn.linear_model import LogisticRegression
>> from sklearn.tree import DecisionTreeClassifier
>> from sklearn.neighbors import KNeighborsClassifier
>> from sklearn.pipeline import Pipeline
>> import numpy as np
>> clf1 = LogisticRegression(penalty='l2',
...                           C=0.001,
...                           çözücü='lbfgs',
...                           random_state=1)
>> clf2 = DecisionTreeClassifier(max_depth=1,
...                              criterion='entropy',
...                              random_state=0)
>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                            p=2,
...                            metric='minkowski')
>> pipe1 = Boru Hattı([['sc', StandardScaler()],
...                    ['clf', clf1]])
>> pipe3 = Boru Hattı([['sc', StandardScaler()],
...                    ['clf', clf3]])
>> clf_labels = ['Lojistik regresyon', 'Karar ağacı', 'KNN']
>> print('10 kat çapraz doğrulama:\n')
>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
...                              X=X_train,
...                              y=y_train,
```

değerlendireceğiz:

 olimpikbarbell@gmail.com 

```
...                                 cv=10,
...                                 scoring='roc_auc')
...         print(f'ROC AUC: {scores.mean():.2f} '
...               f'(+/- {scores.std():.2f}) [{label}]')
```

Aldığımız çıktı, aşağıdaki parçacıkta gösterildiği gibi, bireysel sınıflandırıcıların tahmin performanslarının neredeyse eşit olduğunu göstermektedir:

```
10 kat çapraz doğrulama:
ROC AUC: 0,92 (+/- 0,15) [Lojistik regresyon] ROC
AUC: 0.87 (+/- 0.18) [Karar ağacı]
ROC AUC: 0,85 (+/- 0,13) [KNN]
```

Lojistik regresyon ve k-en yakın komşu sınıflandırıcısını neden bir boru hattının parçası olarak eğittiğimizi merak ediyor olabilirsiniz. Bunun nedeni, *Bölüm 3'te* tartışıldığı gibi, hem lojistik regresyon hem de k-en yakın komşu algoritmalarının (Öklid mesafesi metriğini kullanarak) karar ağaçlarının aksine ölçekle değişmez olmamasıdır. İris özelliklerinin hepsi aynı ölçekte (cm) ölçülmüş olsa da, standartlaştırılmış özelliklerle çalışmak iyi bir alışkanlıktır.

Şimdi, daha heyecan verici kısma geçelim ve çoğunluk kuralı oylaması için ayrı sınıflandırıcıları MajorityVoteClassifier'da birleştirelim:

```
>> mv_clf = MajorityVoteClassifier(
...         sınıflandırıcılar=[pipe1, clf2, pipe3]
... )
>> clf_labels += ['Çoğunluk oylaması']
>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>> for clf, label in zip(all_clf, clf_labels):
...         scores = cross_val_score(estimator=clf,
...                                  X=X_train,
...                                  y=y_train,
...                                  cv=10,
...                                  scoring='roc_auc')
...         print(f'ROC AUC: {scores.mean():.2f} '
...               f'(+/- {scores.std():.2f}) [{label}]')
ROC AUC: 0,92 (+/- 0,15) [Lojistik regresyon]
ROC AUC: 0,87 (+/- 0,18) [Karar ağacı] ROC
AUC: 0,85 (+/- 0,13) [KNN]
ROC AUC: 0,98 (+/- 0,05) [Çoğunluk oylaması]
```

Gördüğünüz gibi, MajorityVotingClassifier'ın performansı 10 kat çapraz doğrulama değerlendirmesinde bireysel sınıflandırıcılara göre iyileşmiştir.

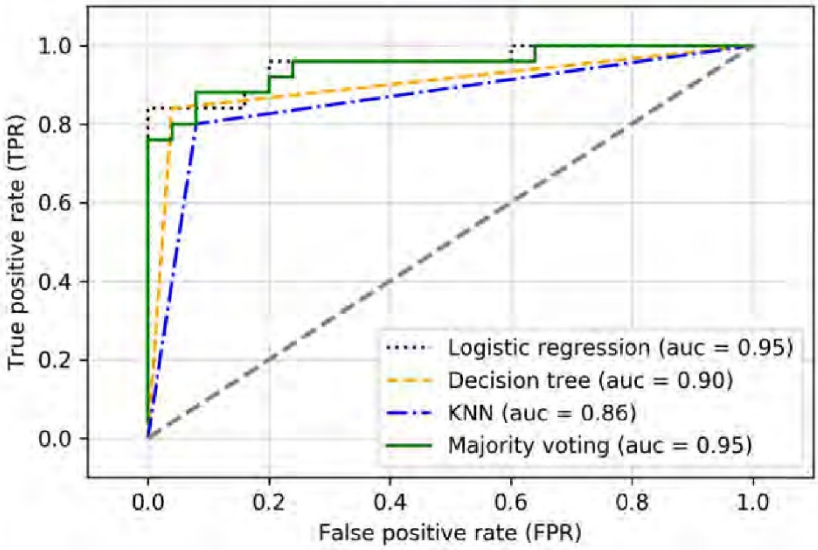
## Topluluk sınıflandırıcısının değerlendirilmesi ve ayarlanması

Bu bölümde, MajorityVoteClassifier'ın görünmeyen verilerle iyi genelleme yapıp yapmadığını kontrol etmek için test veri setinden ROC eğrilerini hesaplayacağız. Test veri setinin model seçimi için kullanılmayacağını unutmamalıyız; amacı sadece bir sınıflandırıcı sistemin genelleme performansının tarafsız bir tahminini rapor etmektir:

```
>> from sklearn.metrics import roc_curve
>> from sklearn.metrics import auc
>>> renkler = ['siyah', 'turuncu', 'mavi', 'yeşil']
>>> linestyles = [':', '--', '-.', '-']
>> for clf, label, clr, ls \
...       zip(all_clf, clf_labels, colors, linestyles) içinde:
...       # pozitif sınıfın etiketinin 1 olduğunu varsayarak
...       y_pred = clf.fit(X_train,
...                        y_train).predict_proba(X_test)[:, 1]
...       fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                        y_score=y_pred)
...       roc_auc = auc(x=fpr, y=tpr)
...       plt.plot(fpr, tpr,
...                color=clr,
...                linestyle=ls,
...                label=f'{label} (auc = {roc_auc:.2f})')
>> plt.legend(loc='sağ alt')
>> plt.plot([0, 1], [0, 1],
...          linestyle='--',
...          color='gray',
...          çizgi genişliği=2)
>> plt.xlim([-0.1, 1.1])
>> plt.ylim([-0.1, 1.1])
>>  plt.grid(alpha=0.5)
>> plt.xlabel('Yanlış pozitif oran (FPR)')
>> plt.ylabel('Gerçek pozitif oran (TPR)')
>> plt.show()
```

olimpikbarbell@gmail.com

Elde edilen ROC'de görebileceğiniz gibi, topluluk sınıflandırıcısı test veri setinde de iyi performans göstermektedir (ROC AUC = 0,95). Bununla birlikte, lojistik regresyon sınıflandırıcısının aynı veri kümesinde benzer şekilde iyi performans gösterdiğini görebilirsiniz, bu muhtemelen veri kümesinin küçük boyutu göz önüne alındığında yüksek varyanstan (bu durumda veri kümesini nasıl böldüğümüzün hassasiyeti) kaynaklanmaktadır:




*Şekil 7.4: Farklı sınıflandırıcılar için ROC eğrisi*

Sınıflandırma örnekleri için yalnızca iki özellik seçtiğimizden, topluluk sınıflandırıcısının karar bölgesinin gerçekte neye benzediğini görmek ilginç olacaktır.

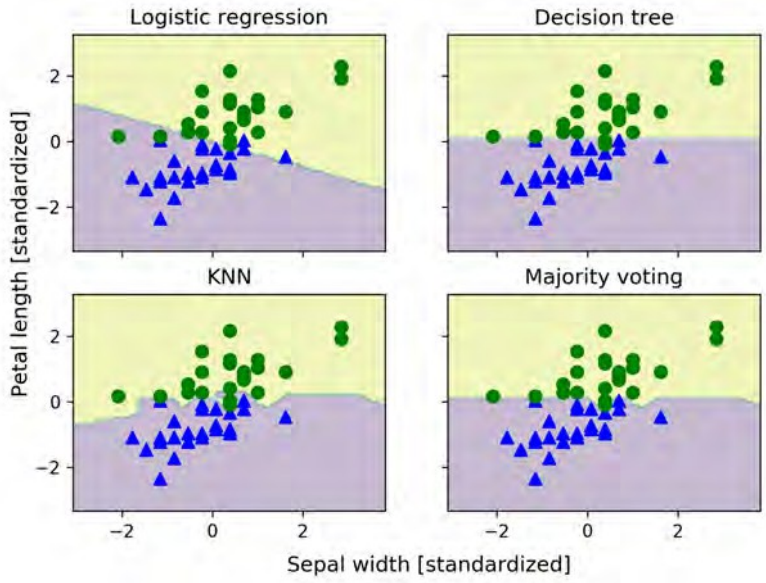
Model uydurmadan önce eğitim özelliklerini standartlaştırmak gerekli olmasa da, lojistik regresyon ve k-en yakın komşular boru hatlarımız bunu otomatik olarak halledeceğinden, eğitim veri setini standartlaştıracağız, böylece karar ağacının karar bölgeleri görsel amaçlar için aynı ölçekte olacaktır. Kod aşağıdaki gibidir:

```
>> sc = StandardScaler()
>> X_train_std = sc.fit_transform(X_train)
>> from itertools import product
>> x_min = X_train_std[:, 0].min() - 1
>> x_max = X_train_std[:, 0].max() + 1
>>> y_min = X_train_std[:, 1].min() - 1
>>>
>>> y_max = X_train_std[:, 1].max() + 1
```

```
>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(7, 5))
>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                         all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
...                                   X_train_std[y_train==0, 1],
...                                   c='mavi',
...                                   marker='^',
...                                   s=50)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
...                                   X_train_std[y_train==1, 1],
...                                   c='yeşil',
...                                   marker='o',
...                                   s=50)
...     axarr[idx[0], idx[1]].set_title(tt)
>> plt.text(-3.5, -5.,
...          s='Sepal genişlik [standartlaştırılmış]',
...          ha='center', va='center', fontsize=12)
>> plt.text(-12.5, 4.5,
...          s='Petal uzunluğu [standartlaştırılmış]',
...          ha='orta', va='orta',
...          fontsize=12, rotation=90)
>> plt.show()
```

İlginç ama aynı zamanda beklendiği gibi, topluluk sınıflandırıcısının karar bölgeleri, bireysel sınıflandırıcılardan gelen karar bölgelerinin bir hibrid köprüsü gibi görünmektedir. İlk bakışta, çoğunluk oyu karar sınırı, *sepal genişlik* $\geq 1$ için $y$ eksenine ortogonal olan karar ağacı kütüğünün kararına çok benziyor.

Bununla birlikte, k-en yakın komşu sınıflandırıcısından gelen doğrusal olmayanlığın da karıştığını fark edebilirsiniz:




*Şekil 7.5: Farklı sınıflandırıcılar için karar sınırları*

Topluluk sınıflandırması için bireysel sınıflandırıcının parametrelerini ayarlamadan önce, bir GridSearchCV nesnesi içindeki bireysel parametrelere nasıl erişebileceğimize dair temel bir fikir edinmek için get_ params yöntemini çağıralım:

```
>> mv_clf.get_params()
{'decisiontreeclassifier':
 DecisionTreeClassifier(class_weight=None, criterion='entropy',
                        max_depth=1, max_features=None,
                        max_leaf_nodes=None, min_samples_leaf=1,
                        min_samples_split=2,
                        min_weight_fraction_leaf=0.0,
                        random_state=0, splitter='best'),
 'decisiontreeclassifier class_weight': None,
 'decisiontreeclassifier criterion': ' entropy',
 [...]
 'decisiontreeclassifier random_state': 0,
 'decisiontreeclassifier splitter': ' best',
 'pipeline-1':
 Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
                                       with_std=True)),
                 ('clf', LogisticRegression(C=0.001,
```

```
                                                class_weight=None,
                                                dual=False,
                                                fit_intercept=True,
                                                intercept_scaling=1,
                                                max_iter=100,
                                                multi_class='ovr',
                                                penalty='l2',
                                                random_state=0,
                                                solver='liblinear',
                                                tol=0.0001,
                                                verbose=0))]),
'boru hattı-1 clf':
LogisticRegression(C=0.001, class_weight=None, dual=False,
                   fit_intercept=True, intercept_scaling=1,
                   max_iter=100, multi_class='ovr',
                   penalty='l2', random_state=0,
                   solver='liblinear', tol=0.0001, verbose=0),
'boru hattı-1 clf C': 0.001, 'pipeline-
1 clf class_weight': Yok, 'boru hattı-
1 clf dual': False,
[...]
'pipeline-1 sc with_std': True,
'boru hattı-2':
Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
                                      with_std=True)),
                ('clf', KNeighborsClassifier(algorithm='auto',
                                             leaf_size=30,
                                             metric='minkowski',
                                             metric_params=None,
                                             n_neighbors=1,
                                             p=2,
                                             weights='uniform')]),
'pipeline-2 clf':
KNeighborsClassifier(algorithm='auto', leaf_size=30,
                     metric='minkowski', metric_params=None,
                     n_neighbors=1, p=2, weights='uniform'),
'boru hattı-2 clf algoritması':
'otomatik', [...]
'pipeline-2 sc with_std': True}
```

 olimpikbarbell@gmail.com 

get_params yöntemi tarafından döndürülen değerlere dayanarak, artık her bir sınıflandırıcının özniteliklerine nasıl erişeceğimizi biliyoruz. Şimdi lojistik regresyon sınıflandırıcısının ters regülasyon parametresi C'yi ve karar ağacı derinliğini gösterim amacıyla bir ızgara aramasıyla

```
>> from sklearn.model_selection import GridSearchCV
>>> params = {'decisiontreeclassifier max_depth': [1, 2],
...            'boru hattı-1 clf C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(tahmin edici=mv_clf,
...                     param_grid=params,
...                     cv=10,
...                     scoring='roc_auc')
>> grid.fit(X_train, y_train)
```

ayarlayalım:

Izgara araması tamamlandıktan sonra, farklı hiperparametre değeri kombinasyonlarını ve 10 katlı çapraz doğrulama yoluyla hesaplanan ortalama ROC AUC puanlarını aşağıdaki gibi yazdırabiliriz:

```
>> for r, _ in enumerate(grid.cv_results_['mean_test_score']):
...     mean_score = grid.cv_results_['mean_test_score'][r]
...     std_dev = grid.cv_results_['std_test_score'][r]
...     params = grid.cv_results_['params'][r]
...     print(f'{mean_score:.3f} +/- {std_dev:.2f} {params}')
0.983 +/- 0.05 {'decisiontreeclassifier max_depth': 1,
                'boru hattı-1 clf C': 0.001}
0,983 +/- 0,05 {'decisiontreeclassifier max_depth': 1,
                'boru hattı-1 clf C': 0.1}
0,967 +/- 0,10 {'decisiontreeclassifier max_depth': 1,
                'boru hattı-1 clf C': 100.0}
0,983 +/- 0,05 {'decisiontreeclassifier max_depth': 2,
                'boru hattı-1 clf C': 0.001}
0,983 +/- 0,05 {'decisiontreeclassifier max_depth': 2,
                'boru hattı-1 clf C': 0.1}
0,967 +/- 0,10 {'decisiontreeclassifier max_depth': 2,
                'boru hattı-1 clf C': 100.0}
>> print(f'En iyi parametreler: {grid.best_params_}')
En iyi parametreler: {'decisiontreeclassifier max_depth': 1,
                      'boru hattı-1 clf C': 0.001}
>> print(f'ROC AUC : {grid.best_score_:.2f}')
ROC AUC: 0.98
```

Gördüğünüz gibi, daha düşük bir düzenlileştirme gücü (C=0.001) seçtiğimizde en iyi çapraz doğrulama sonuçlarını elde ederken, ağaç derinliği performansı hiç etkilemiyor gibi görünmektedir, bu da bir karar kütüğünün verileri ayırmak için yeterli olduğunu göstermektedir. Model değerlendirmesi için test veri setini birden fazla kez kullanmanın kötü bir uygulama olduğunu kendimize hatırlatmak için, bu bölümde ayarlanmış hiperparametrelerin genelleme performansını tahmin etmeyeceğiz. Hızlıca topluluk öğrenmesi için alternatif bir yaklaşıma geçeceğiz: **torbalama**.

**Yığınlama kullanarak topluluklar oluşturma**

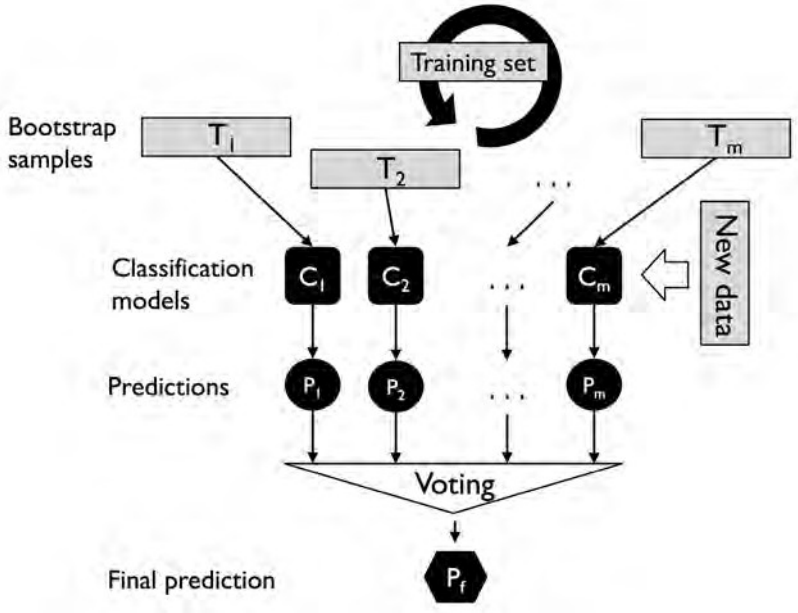
Bu bölümde uyguladığımız çoğunluk oyu yaklaşımı istifleme ile karıştırılmamalıdır. Yığınlama algoritması iki seviyeli bir topluluk olarak anlaşılabilir; ilk seviye, tahminlerini ikinci seviyeye besleyen bireysel sınıflandırıcılardan oluşur; burada başka bir sınıflandırıcı (tipik olarak lojistik regresyon), nihai tahminleri yapmak için birinci seviye sınıflandırıcı tahminlerine uyarlanır. Yığınlama hakkında daha fazla bilgi için aşağıdaki kaynaklara bakın:

- Yığın algoritması David H. Wolp- ert tarafından *Stacked generalization*, *Neural Networks,* 5(2):241-259, 1992'de daha ayrıntılı olarak açıklanmıştır (https://www.sciencedirect.com/science/article/pii/S0893608005800231).
- İlgilenen okuyucular, istifleme ile ilgili video eğitimimizi YouTube'da şu adreste bulabilirler https://www.youtube.com/watch?v=8T2emza6g80.
- Yığın sınıflandırıcının scikit-learn uyumlu bir sürümü mlxtend'den edinilebilir: http://rasbt.github.io/mlxtend/user_guide/classifier/StackingCVClassifier/.
- Ayrıca, scikit-learn'e yakın zamanda bir StackingClassifier eklenmiştir (0.22 ve daha yeni sürümlerde mevcuttur); daha fazla bilgi için lütfen https://scikit-learn.org/stable/modules/generated/sklearn.ensemble adresindeki belgelere bakın. StackingClassifier.html.

# Torbalama - bootstrap örneklerinden bir sınıflandırıcı topluluğu oluşturma

Torbalama, önceki bölümde uyguladığımız MajorityVoteClassifier ile yakından ilişkili olan bir topluluk öğrenme tekniğidir. Bununla birlikte, topluluktaki bireysel sınıflandırıcılara uyması için aynı eğitim veri kümesini kullanmak yerine, ilk eğitim veri kümesinden bootstrap örnekleri (değiştirme ile rastgele örnekler) çekeriz, bu nedenle torbalama bootstrap *toplama* olarak da bilinir.

Torbalama kavramı *Şekil 7.6'*da özetlenmiştir:

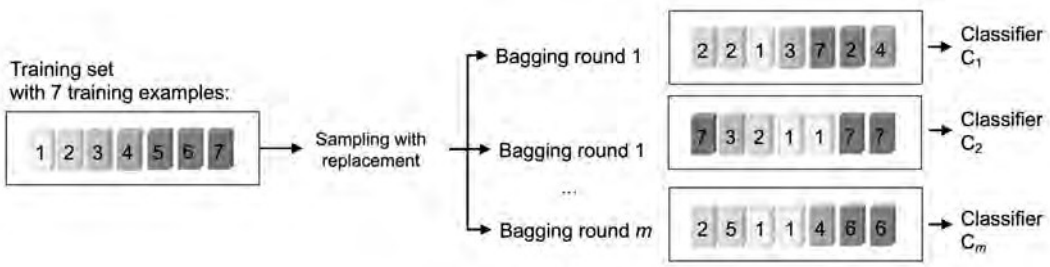



*Şekil 7.6: Torbalama kavramı*

Aşağıdaki alt bölümlerde, elle basit bir torbalama örneği üzerinde çalışacağız ve şarap örneklerini sınıflandırmak için scikit-learn kullanacağız.

## Özetle torbalama

Bir torbalama sınıflandırıcısının bootstrap toplamasının nasıl çalıştığına dair daha somut bir örnek vermek için *Şekil 7.7'*de gösterilen örneği ele alalım. Burada, torbalama işleminin her turunda rastgele ve değiştirilerek örneklenen yedi farklı eğitim örneğimiz (indeks 1-7 olarak belirtilmiştir) bulunmaktadır. Her bir bootstrap örneği daha sonra, tipik olarak budanmamış bir karar ağacı olan $C_j$ sınıflandırıcısını yerleştirmek için kullanılır:




*Şekil 7.7: Bir torbalama örneği*

*Şekil 7.7*'den de görebileceğiniz gibi, her sınıflandırıcı eğitim veri kümesinden rastgele bir örnek alt kümesi alır. Torbalama yoluyla elde edilen bu rastgele örnekleri Torbalama *turu 1*, *Torbalama turu 2* vb. olarak adlandırıyoruz. Her alt küme belirli bir oranda kopya içerir ve orijinal örneklerin bir kısmı, değiştirme ile örnekleme nedeniyle yeniden örneklenmiş veri kümesinde hiç görünmez. Bireysel sınıflandırıcılar bootstrap örneklerine uygun hale getirildikten sonra, tahminler çoğunluk oylaması kullanılarak birleştirilir.

Torbalama işleminin *Bölüm 3*'te tanıttığımız rastgele orman sınıflandırıcı ile de ilişkili olduğunu unutmayın. Aslında rastgele ormanlar, bireysel karar ağaçlarını yerleştirirken rastgele özellik alt kümelerini de kullandığımız özel bir torbalama durumudur.

**Torbalama kullanarak model toplulukları**

Torbalama ilk olarak 1994 yılında Leo Breiman tarafından teknik bir raporda önerilmiştir; Breiman ayrıca torbalamanın kararsız modellerin doğruluğunu artırabileceğini ve aşırı uyum derecesini azaltabileceğini göstermiştir. Torbalama hakkında daha fazla bilgi edinmek için *L. Breiman*'ın *Torbalama Tahmincileri*, *Makine Öğrenimi,* 24(2):123-140, 1996 adlı çevrimiçi olarak ücretsiz olarak erişilebilen araştırmasını okumanızı şiddetle tavsiye ederiz.

## Wine veri kümesindeki örnekleri sınıflandırmak için torbalama uygulamak

Torbalama işlemini çalışırken görmek için, *Bölüm 4, İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme'de* tanıtılan Şarap veri kümesini kullanarak daha karmaşık bir sınıflandırma problemi oluşturalım. Burada, yalnızca Şarap sınıfları 2 ve 3'ü dikkate alacağız ve iki özellik seçeceğiz - Alkol ve seyreltilmiş şarapların OD280/OD315'i:

```
>> import pandas as pd
> >   df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                          'machine-learning-databases/'
...                          'wine/wine.data',
...                          header=None)
>> df_wine.columns = ['Sınıf etiketi', 'Alkol',
...                   "Malik asit", "Kül",
...                   "Külün alkalinitesi",
...                   'Magnezyum', 'Toplam fenoller',
...                   'Flavanoidler', 'Flavanoid olmayan fenoller',
...                   "Proantosiyaninler",
...                   'Renk yoğunluğu', 'Ton',
...                   'Seyreltilmiş şarapların OD280/OD315'i',
...                   'Proline']
>> # 1 sınıf düşür
>> df_wine = df_wine[df_wine['Sınıf etiketi'] != 1]
>> y = df_wine['Sınıf etiketi'].değerler
>>> X = df_wine[['Alkol',
...              'Seyreltilmiş şarapların OD280/OD315']].values
```

 olimpikbarbell@gmail.com 

Ardından, sınıf etiketlerini ikili formatta kodlayacağız ve veri kümesini yüzde 80 eğitim ve yüzde 20 test veri kümelerine böleceğiz:

```
>> from sklearn.preprocessing import LabelEncoder
>> from sklearn.model_selection import train_test_split
>> le = LabelEncoder()
>> y = le.fit_transform(y)
>> X_train, X_test, y_train, y_test =\
...             train_test_split(X, y,
...                              test_size=0.2,
...                              random_state=1,
...                              stratify=y)
```

**Şarap veri setinin elde edilmesi**

Wine veri setinin (ve bu kitapta kullanılan diğer tüm veri setlerinin) bir kopyasını bu kitabın kod paketinde bulabilirsiniz; bu kopyayı ofis dışında çalışıyorsanız veya https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data adresindeki UCI sunucusu geçici olarak kullanılamıyorsa kullanabilirsiniz. Örneğin, Wine veri setini yerel bir dizinden yüklemek için aşağıdaki satırları kullanın:

```
df = pd.read_csv('https://archive.ics.uci.edu/ml/'
                 'machine-learning-databases'
                 '/wine/wine.data',
                 header=None)
```

ve bunları bunlarla değiştirin:

```
df = pd.read_csv('your/local/path/to/wine.data',
                 header=None)
```

Topluluk alt modülünden içe aktarabileceğimiz bir BaggingClassifier algoritması scikit-learn'de zaten uygulanmaktadır. Burada, temel sınıflandırıcı olarak budanmamış bir karar ağacı kullanacağız ve eğitim veri kümesinin farklı bootstrap örneklerine uyan 500 karar ağacından

```
>> from sklearn.ensemble import BaggingClassifier
>> tree = DecisionTreeClassifier(criterion='entropy',
...                              random_state=1,
...                              max_depth=None)
>> bag = BaggingClassifier(base_estimator=tree,
...                        n_estimators=500,
...                        max_samples=1.0,
...                        max_features=1.0,
...                        bootstrap=True,
```

oluşan bir topluluk oluşturacağız:

```
...                                 bootstrap_features=Fals
...                                 e,
...                                 n_jobs=1,
                                    random_state=1)
```

Ardından, torbalama sınıflandırıcısının performansını tek bir budanmamış karar ağacının performansıyla karşılaştırmak için eğitim ve test veri kümeleri üzerindeki tahminin doğruluk

```
>> from sklearn.metrics import accuracy_score
>> tree = tree.fit(X_train, y_train)
>> y_train_pred = tree.predict(X_train)
>> y_test_pred = tree.predict(X_test)
>> tree_train = accuracy_score(y_train, y_train_pred)
>> tree_test = accuracy_score(y_test, y_test_pred)
>> print(f'Karar ağacı eğitim/test doğrulukları '
...          f'{tree_train:.3f}/{tree_test:.3f}')
Karar ağacı eğitim/test doğrulukları
1.000/0.833
```

puanını hesaplayacağız:

Burada yayınladığımız doğruluk değerlerine göre, budanmamış karar ağacı eğitim örneklerinin tüm sınıf etiketlerini doğru tahmin etmektedir; ancak, test doğruluğunun önemli ölçüde düşük olması modelin yüksek varyansına (aşırı uyum) işaret etmektedir:

```
>>> bag = bag.fit(X_train, y_train)
>> y_train_pred = bag.predict(X_train)
>> y_test_pred = bag.predict(X_test)
>> bag_train = accuracy_score(y_train, y_train_pred)
>> bag_test = accuracy_score(y_test, y_test_pred)
>> print(f'Bagging train/test accuracies '
...          f'{bag_train:.3f}/{bag_test:.3f}')
Torbalama eğitme/test doğrulukları
1.000/0.917
```

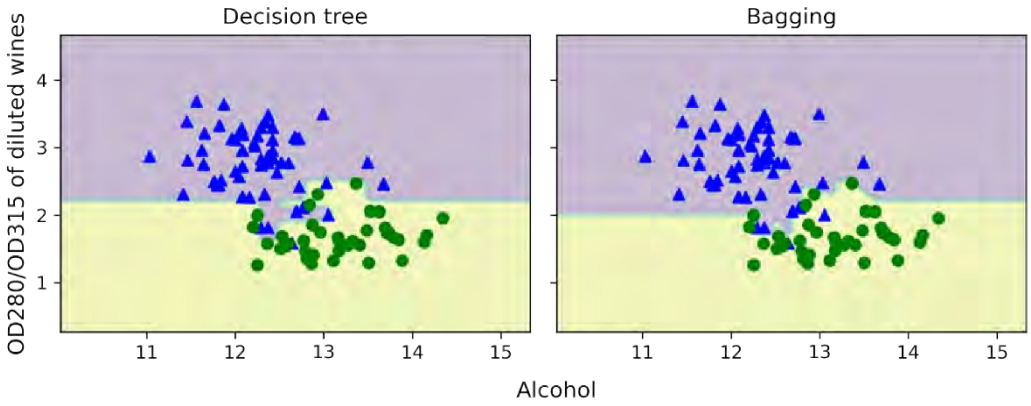
Eğitim veri kümesinde karar ağacı ve torbalama sınıflandırıcısının eğitim doğrulukları benzer olsa da (her ikisi de yüzde 100), test veri kümesinde tahmin edildiği gibi torbalama sınıflandırıcısının biraz daha iyi bir genelleme performansına sahip olduğunu görebiliriz. Daha sonra, karar ağacı ve torbalama sınıflandırıcısı arasındaki karar bölgelerini karşılaştıralım:

```
>> x_min = X_train[:, 0].min() - 1
>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                          np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                             sharex='col',
...                             sharey='row',
```

olimpikbarbell@gmail.com

```
...                                 figsize=(8, 3))
>> for idx, clf, tt in zip([0, 1],
...                                 [ağaç, çanta],
...                                 ['Karar ağacı',
...         'Torbalama']): clf.fit(X_eğitim, y_eğitim)
...
...         Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...         Z = Z.reshape(xx.shape)
...         axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...         axarr[idx].scatter(X_train[y_train==0, 0],
...                            X_train[y_train==0, 1],
...                            c='blue', marker='^')
...         axarr[idx].scatter(X_train[y_train==1, 0],
...                            X_train[y_train==1, 1],
...                            c='green', marker='o')
...         axarr[idx].set_title(tt)
>> axarr[0].set_ylabel('Seyreltilmiş şarapların OD280/OD315',
fontsize=12)
>> plt.tight_layout()
>> plt.text(0, -0.2,
...             s='Alkol',
...             ha='center',
...             va='center',
...             fontsize=12,
...             transform=axarr[1].transAxes)
>> plt.show()
```

Ortaya çıkan grafikte görebileceğimiz gibi, üç düğümlü derin karar ağacının parçalı doğrusal karar sınırı, torbalama topluluğunda daha düzgün görünüyor:




*Şekil 7.8: Torbalamaya karşı bir karar ağacının parçalı doğrusal karar sınırı*

Bu bölümde sadece çok basit bir torbalama örneğine baktık. Uygulamada, daha karmaşık sınıflandırma görevleri ve bir veri kümesinin yüksek boyutluluğu, tek karar ağaçlarında kolayca aşırı uyuma yol açabilir ve bu, torbalama algoritmasının gerçekten güçlü olduğu yerdir. Son olarak, torbalama algoritmasının bir modelin varyansını azaltmak için etkili bir yaklaşım olabileceğini belirtmeliyiz. Ancak torbalama, model yanlılığını, yani verilerdeki eğilimleri iyi yakalayamayacak kadar basit olan modelleri azaltmada etkisizdir. Bu nedenle, örneğin budanmamış karar ağaçları gibi düşük önyargılı bir sınıflandırıcı topluluğu üzerinde torbalama yapmak istiyoruz.

# Uyarlanabilir boosting aracılığıyla zayıf öğrenicilerden yararlanma

Topluluk yöntemleriyle ilgili bu son bölümde, en yaygın uygulamasına özel olarak odaklanarak **boosting'**i tartışacağız: **Uyarlanabilir Boosting** (**AdaBoost**).

**AdaBoost tanıma**

AdaBoost'un arkasındaki orijinal fikir 1990 yılında Robert E. Schapire tarafından *The Strength of Weak Learnability*, *Machine Learning*, 5(2)'de formüle edilmiştir: 197-227, 1990, URL: http://rob. schapire.net/papers/strengthofweak.pdf. Robert Schapire ve Yoav Freund'un AdaBoost algoritmasını *Proceedings of the Thirteenth International Conference'*de *(*ICML 1996) sunmasının ardından AdaBoost, takip eden yıllarda en yaygın kullanılan topluluk yöntemlerinden biri haline gelmiştir (*Experiments with a New Boosting Algorithm* by *Y. Freund*, *R. E. Schapire*, and others, *ICML*, volume 96, 148-156, 1996). Freund ve Schapire, çığır açan çalışmaları için 2003 yılında, bilgisayar bilimleri alanında en çok ses getiren yayınlara verilen prestijli bir ödül olan Gödel Ödülü'nü aldılar.

Güçlendirmede, topluluk, genellikle **zayıf öğreniciler** olarak da adlandırılan ve rastgele tahminlere göre genellikle yalnızca küçük bir performans avantajına sahip olan çok basit temel sınıflandırıcılardan oluşur - zayıf bir öğrenicinin tipik bir örneği karar ağacı kütüğüdür. Güçlendirmenin arkasındaki temel kavram, sınıflandırılması zor olan eğitim örneklerine odaklanmak, yani zayıf öğrenicilerin daha sonra topluluğun performansını artırmak için yanlış sınıflandırılmış eğitim örneklerinden öğrenmesine izin vermektir.

Aşağıdaki alt bölümler genel boosting ve AdaBoost kavramının arkasındaki algoritmik prosedürü tanıtacaktır. Son olarak, pratik bir sınıflandırma örneği için scikit-learn kullanacağız.

## Uyarlanabilir güçlendirme nasıl çalışır?

Torbalama algoritmasının aksine, boosting algoritmasının ilk formülasyonu, eğitim veri setinden değiştirilmeden alınan eğitim örneklerinin rastgele alt kümelerini kullanır; orijinal boosting prosedürü aşağıdaki dört temel adımda özetlenebilir:

1. Zayıf bir öğrenici olan $C_1$ 'yi eğitmek için eğitim veri kümesi $D'$den değiştirmeden rastgele bir eğitim örnekleri alt kümesi (örnek), $d_1$ , çekin.
2. Eğitim veri kümesinden değiştirmeden ikinci bir rastgele eğitim alt kümesi, $d_2$ , çekin ve zayıf bir öğreniciyi, $C_2$ , eğitmek için daha önce yanlış sınıflandırılmış örneklerin yüzde

olimpikbarbell@gmail.com

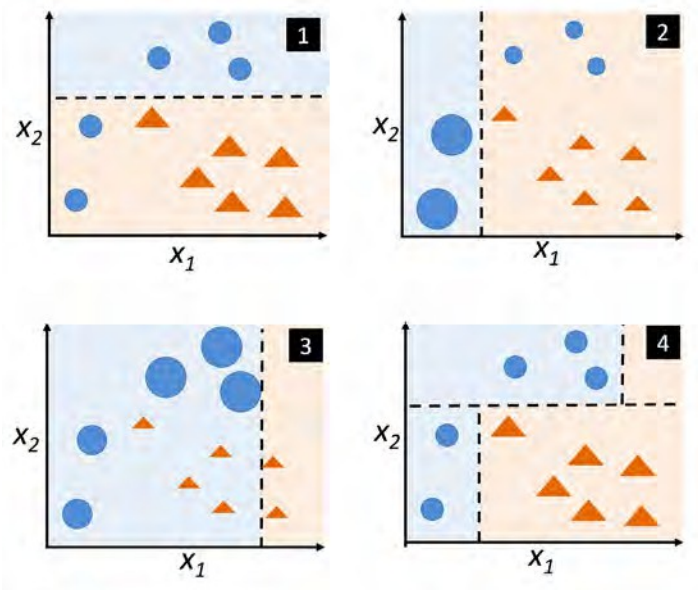
50'sini ekleyin.

3. Üçüncü bir zayıf öğreniciyi, $C_3$ , eğitmek için eğitim veri kümesinde, $D$, $C_1$ ve $C_2$ 'nin üzerinde anlaşamadığı eğitim örneklerini, $d_3$ , bulun.
4. Zayıf öğreniciler $C_1$ , $C_2$ ve $C_3$ 'yi çoğunluk oylaması yoluyla birleştirin.

Leo Breiman tarafından tartışıldığı üzere (*Bias, variance*, *and arcing classifiers,* 1996), boosting, bagging modellerine kıyasla hem bias hem de varyansta azalmaya yol açabilir. Ancak uygulamada, AdaBoost gibi boosting algoritmaları yüksek varyanslarıyla, yani eğitim verilerine aşırı uyum sağlama eğilimleriyle de bilinmektedir (*An improvement of AdaBoost to avoid overfitting* by *G. Raetsch*, *T. Onoda*, and *K. R. Mueller*. *Proceedings of the International Conference on Neural Information Processing*, *CiteSeer*, 1998).

Burada açıklanan orijinal boosting prosedürünün aksine AdaBoost, zayıf öğrenicileri eğitmek için tüm eğitim veri setini kullanır; burada eğitim örnekleri, topluluktaki önceki zayıf öğrenicilerin hatalarından öğrenen güçlü bir sınıflandırıcı oluşturmak için her yinelemede yeniden ağırlıklandırılır.

AdaBoost algoritmasının özel ayrıntılarına girmeden önce *Şekil 7.9*'a bir göz atalım AdaBoost'un arkasındaki temel kavramı daha iyi kavramak için:




*Şekil 7.9: Zayıf öğrenicileri iyileştirmek için AdaBoost kavramı*

AdaBoost örneğini adım adım incelemek için, tüm eğitim örneklerine eşit ağırlıkların atandığı ikili sınıflandırma için bir eğitim veri kümesini temsil eden alt şekil **1** ile başlayacağız. Bu eğitim veri kümesine dayanarak, iki sınıfın örneklerini (üçgenler ve daireler) sınıflandırmaya çalışan ve muhtemelen kayıp fonksiyonunu (veya karar ağacı topluluklarının özel durumunda safsızlık puanını) en aza indiren bir karar kütüğünü (kesikli çizgi olarak gösterilir) eğitiriz.

Bir sonraki tur için (alt şekil **2**), önceden yanlış sınıflandırılmış iki örneğe (daireler) daha büyük bir ağırlık atarız. Ayrıca, doğru sınıflandırılmış örneklerin ağırlığını düşürüyoruz. Bir sonraki karar kütüğü artık en büyük ağırlığa sahip olan eğitim örneklerine, yani sınıflandırılması zor olduğu varsayılan eğitim örneklerine daha fazla odaklanacaktır.

Alt şekil **2'de** gösterilen zayıf öğrenici, daire sınıfından üç farklı örneği yanlış sınıflandırır ve bu örneklere alt şekil **3'te** gösterildiği gibi daha büyük bir ağırlık atanır.

AdaBoost topluluğumuzun yalnızca üç tur güçlendirmeden oluştuğunu varsayarsak, alt şekil **4'**te gösterildiği gibi, farklı yeniden ağırlıklandırılmış eğitim alt kümeleri üzerinde eğitilen üç zayıf öğreniciyi ağırlıklı çoğunluk oylamasıyla birleştiririz.

AdaBoost'un temel konseptini daha iyi anladığımıza göre, sözde kod kullanarak algoritmaya daha ayrıntılı bir göz atalım. Anlaşılır olması için, eleman-bilge çarpımını çarpı sembolü (×) ile ve iki vektör arasındaki nokta-çarpımını nokta sembolü (·) ile göstereceğiz:

1. Ağırlık vektörünü, $\boldsymbol{w}$, $\sum ww = 1$ olacak şekilde tek tip ağırlıklara ayarlayın.
2. $m$ artırma turundaki $j$ için aşağıdakileri yapın:
   a. Ağırlıklı bir zayıf öğreniciyi eğitin: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
   b. Sınıf etiketlerini tahmin edin: $\hat{y} = \text{tahmin}(C_{jj}, \boldsymbol{XXX}$.
   c. Ağırlıklı hata oranını hesaplayın: $\varepsilon\varepsilon\ \varepsilon\ \varepsilon\varepsilon\varepsilon\ (\hat{y} \neq y)$.
   d. Katsayıyı hesaplayın: $\alpha_{jj} = 0{,}5 \log \frac{1-s}{s}$
   e. Ağırlıkları güncelleyin: $\boldsymbol{ww}\ \boldsymbol{w}\ \boldsymbol{ww}_{\boldsymbol{w}\ \boldsymbol{w}\text{xp}\boldsymbol{ww}\alpha_{jj}}\boldsymbol{w}\ \hat{y}\ \boldsymbol{w}\ \boldsymbol{yy}$.
   Ağırlıkları toplamı 1 olacak şekilde normalleştirin: $\boldsymbol{ww}\ \boldsymbol{ww}\boldsymbol{ww}\sum ww$ .
3. Nihai tahmini hesaplayın: $\hat{y} = (\sum_{jj1}^{m} (\alpha_{jj} \times \text{predict}(C_{jj}, \boldsymbol{XXXX}\ \boldsymbol{XX}$.

*Adım 2c'*deki $(\hat{y} \neq y)$ ifadesinin 1'ler ve 0'lardan oluşan ikili bir vektörü ifade ettiğine dikkat edin; burada tahmin yanlışsa 1, değilse 0 atanır.

AdaBoost algoritması oldukça basit gibi görünse de, *Şekil 7.10'*da gösterildiği gibi 10 eğitim örneğinden oluşan bir eğitim veri kümesi kullanarak daha somut bir örnek üzerinden gidelim:

| Index | x | y | Weights | $\hat{y}$(x <= 3.0)? | Correct? | Updated weights |
|---|---|---|---|---|---|---|
| 1 | 1.0 | 1 | 0.1 | 1 | Yes | 0.072 |
| 2 | 2.0 | 1 | 0.1 | 1 | Yes | 0.072 |
| 3 | 3.0 | 1 | 0.1 | 1 | Yes | 0.072 |
| 4 | 4.0 | -1 | 0.1 | -1 | Yes | 0.072 |
| 5 | 5.0 | -1 | 0.1 | -1 | Yes | 0.072 |
| 6 | 6.0 | -1 | 0.1 | -1 | Yes | 0.072 |
| 7 | 7.0 | 1 | 0.1 | -1 | No | 0.167 |
| 8 | 8.0 | 1 | 0.1 | -1 | No | 0.167 |
| 9 | 9.0 | 1 | 0.1 | -1 | No | 0.167 |
| 10 | 10.0 | -1 | 0.1 | -1 | Yes | 0.072 |

*Şekil 7.10: AdaBoost algoritması aracılığıyla 10 eğitim örneğinin çalıştırılması*

Tablonun ilk sütunu 1'den 10'a kadar olan eğitim örneklerinin indekslerini göstermektedir. İkinci sütunda, bunun tek boyutlu bir veri kümesi olduğunu varsayarak, tek tek örneklerin özellik değerlerini görebilirsiniz. Üçüncü sütun, her bir eğitim örneği için gerçek sınıf etiketini, $y_i$ , gösterir, $x_i$ , burada $y_i \in \{1, -1\}$. Başlangıç ağırlıkları dördüncü sütunda gösterilmektedir; ağırlıkları tekdüze olarak başlatıyoruz (aynı sabit değeri atayarak) ve toplamları 1 olacak şekilde normalleştiriyoruz. 10 örnekli eğitim veri kümesi durumunda, bu nedenle her ağırlığa 0.$_i$Tahmin edilen sınıf etiketleri, $\hat{y}$, beşinci sütunda, bölme kriterimizin $x$ $x$ $xx$ olduğu varsayılarak gösterilmektedir. Tablonun son sütunu, sözde kodda tanımladığımız güncelleme kurallarına dayalı olarak güncellenen ağırlıkları göstermektedir.

Ağırlık güncellemelerinin hesaplanması ilk başta biraz karmaşık görünebileceğinden, şimdi hesaplamayı adım adım takip edeceğiz. *Adım 2c'de* açıklandığı gibi ağırlıklı hata oranını, $\varepsilon\varepsilon$ (epsilon),

```
>> y = np.array([1, 1, 1, -1, -1, -1, 1, 1, 1, -1])
>> yhat = np.array([1, 1, 1, -1, -1, -1, -1, -1, -1])
>>> doğru = (y == yhat)
>>> ağırlıklar = np.full(10, 0.1)
>> print(ağırlıklar)
[0.1 0.1 0.1 0 . 1 0.1 0.1 0.1 0 . 1   0 . 1 0 . 1 ]
>>> epsilon = np.mean(~doğru)
>> print(epsilon)
0.3
```

hesaplayarak başlayacağız:

correct değerinin True ve False değerlerinden oluşan bir Boolean dizisi olduğunu ve True değerinin bir tahminin doğru olduğunu gösterdiğini unutmayın. correct aracılığıyla diziyi ters çeviririz, böylece np.mean(~correct) yanlış tahminlerin oranını (True değeri 1 ve False değeri 0 olarak sayılır), yani sınıflandırma hatasını hesaplar.

Daha sonra, adım *2d'de* gösterilen $\alpha_{jj}$ katsayısını hesaplayacağız; *bu* katsayı daha sonra *adım 2e'de* ağırlıkları

```
>>> alpha_j = 0,5 * np.log((1-epsilon) / epsilon)
>> print(alpha_j) 0.42364893019360184
```

güncellemek ve çoğunluk oyu tahminindeki ağırlıklar için kullanılacaktır (adım *3*):

Katsayıyı hesapladıktan sonra, $\alpha_{jj}$ (alpha_j), şimdi aşağıdaki denklemi kullanarak ağırlık vektörünü güncelleyebiliriz:

$$ww\ w\ ww_{w\ w\mathrm{xp}ww\alpha jj}w\ \hat{y}\ w\ yy$$

Burada, $\hat{y}$ × $y$ sırasıyla tahmin edilen ve gerçek sınıf etiketlerinin vektörleri arasındaki eleman-bilge

```
>> update_if_correct = 0.1 * np.exp(-alpha_j * 1 * 1)
>> print(update_if_correct)
0.06546536707079771
```

olimpikbarbell@gmail.com

çarpımıdır. Böylece, bir tahmin, $yy_i$ , doğruysa, $yy_i \times yy_i$ pozitif bir işarete sahip olacaktır, böylece $\alpha_{jj}$ da pozitif bir sayı olduğundan, ith ağırlığını azaltırız:

Benzer şekilde, $yy_i$ etiketi yanlış tahmin ettiyse birinci ağırlığı şu şekilde artıracağız:

```
>> update_if_wrong_1 = 0.1 * np.exp(-alpha_j * 1 * -1)
>> print(update_if_wrong_1)
0.1527525231651947
```

Alternatif olarak, şöyle de olabilir:

```
>> update_if_wrong_2 = 0.1 * np.exp(-alpha_j * -1 * 1)
>> print(update_if_wrong_2)
0.1527525231651947
```

Ağırlıkları güncellemek için bu değerleri aşağıdaki gibi kullanabiliriz:

```
>>> ağırlıklar = np.where(doğru == 1,
...                       update_if_correct,
...                       update_if_wrong_1)
>> print(ağırlıklar)
array([0.06546537, 0.06546537, 0 . 0 6 5 4 6 5 3 7 ,  0 . 06546537, 0 . 06546537,
       0.06546537, 0 . 15275252, 0.15275252, 0.15275252, 0 . 06546537])
```

Yukarıdaki kod update_if_correct değerini tüm doğru tahminlere ve update_if_ wrong_1 değerini tüm yanlış tahminlere atamıştır. Basit olması için update_if_wrong_2 değerini kullanmadık, çünkü zaten update_if_wrong_1 değerine benzemektedir.

Ağırlık vektöründeki her bir ağırlığı güncelledikten sonra, toplamları 1 olacak şekilde ağırlıkları normalleştiririz (*adım 2f*):

$$ii\sum \frac{www}{ww}$$

Kod olarak bunu aşağıdaki şekilde gerçekleştirebiliriz:

```
>>> normalize_ağırlıklar = ağırlıklar / np.sum(ağırlıklar)
> >   print(normalized_weights)
[0.07 1 4 2 8 5 7  0 . 07142857 0.07142857 0.07142857 0.07142857 0 . 07142857
 0.16666667 0.16666667 0.16666667 0 . 07142857]
```

Böylece, doğru sınıflandırılmış bir örneğe karşılık gelen her ağırlık, bir sonraki güçlendirme turu için başlangıç değeri olan 0,1'den 0,0714'e düşürülecektir. Benzer şekilde, yanlış sınıflandırılmış örneklerin ağırlıkları 0,1'den 0,1667'ye yükselecektir.

## AdaBoost'u scikit-learn kullanarak uygulama

Önceki alt bölümde AdaBoost kısaca tanıtıldı. Daha pratik kısma geçerek, şimdi scikit-learn aracılığıyla bir AdaBoost topluluk sınıflandırıcısını eğitelim. Torbalama meta sınıflandırıcısını eğitmek için önceki bölümde kullandığımız aynı Şarap alt kümesini kullanacağız.

 olimpikbarbell@gmail.com 

base_estimator özelliği aracılığıyla AdaBoostClassifier'ı 500 karar ağacı kütüğü üzerinde eğiteceğiz:

```
>> from sklearn.ensemble import AdaBoostClassifier
>>  tree = DecisionTreeClassifier(criterion='entropy',
...                                  random_state=1,
...                                  max_depth=1)
>> ada = AdaBoostClassifier(base_estimator=tree,
...                              n_estimators=500,
...                              learning_rate=0.1,
...                              random_state=1)
>> tree = tree.fit(X_train, y_train)
>> y_train_pred = tree.predict(X_train)
>> y_test_pred = tree.predict(X_test)
>> tree_train = accuracy_score(y_train, y_train_pred)
>> tree_test = accuracy_score(y_test, y_test_pred)
>> print(f'Karar ağacı eğitim/test doğrulukları '
...          f'{tree_train:.3f}/{tree_test:.3f}')
Karar ağacı eğitim/test doğrulukları
0,916/0,875
```

Gördüğünüz gibi, karar ağacı kütüğü, önceki bölümde gördüğümüz budanmamış karar ağacının aksine eğitim verilerine yeterince uymuyor gibi görünüyor:

```
>> ada = ada.fit(X_train, y_train)
>> y_train_pred = ada.predict(X_train)
>> y_test_pred = ada.predict(X_test)
>> ada_train = accuracy_score(y_train, y_train_pred)
>> ada_test = accuracy_score(y_test, y_test_pred)
>> print(f'AdaBoost train/test accuracies '
...          f'{ada_train:.3f}/{ada_test:.3f}')
AdaBoost eğitme/test doğrulukları
1.000/0.917
```

Burada, AdaBoost modelinin eğitim veri kümesinin tüm sınıf etiketlerini doğru tahmin ettiğini ve ayrıca karar ağacı kütüğüne kıyasla biraz daha iyi bir test veri kümesi performansı gösterdiğini görebilirsiniz. Bununla birlikte, model önyargısını azaltma girişimimizle ek varyans getirdiğimizi de görebilirsiniz - eğitim ve test performansı arasında daha büyük bir boşluk.

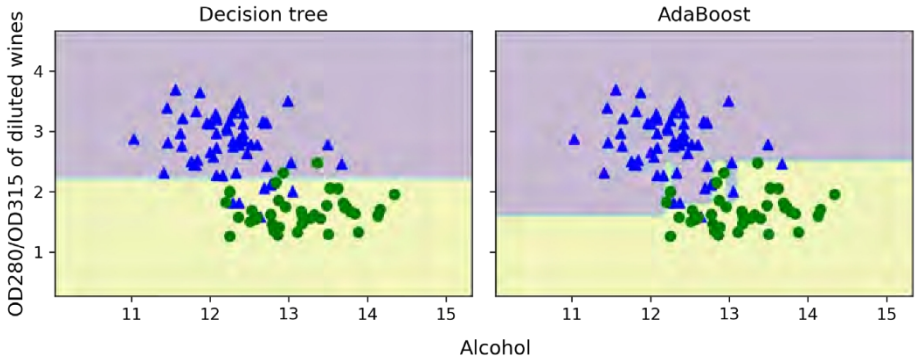
Gösterim amacıyla başka bir basit örnek kullanmamıza rağmen, AdaBoost sınıflandırıcısının performansının karar kütüğüne kıyasla biraz daha iyi olduğunu ve önceki bölümde eğittiğimiz torbalama sınıflandırıcısıyla çok benzer doğruluk puanları elde ettiğini görebiliriz. Ancak, test veri kümesinin tekrarlanan kullanımına dayalı bir model seçmenin kötü bir uygulama olarak kabul edildiğini belirtmeliyiz. Genelleme performansının tahmini aşırı iyimser olabilir, bu konuyu *Bölüm 6, Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Uygulamaları Öğrenme* bölümünde daha ayrıntılı olarak ele aldık.

Son olarak, karar bölgelerinin neye benzediğini kontrol edelim:

```
>> x_min = X_train[:, 0].min() - 1
>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                     np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>> for idx, clf, tt in zip([0, 1],
...                         [ağaç, ada],
...                         ['Karar ağacı', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='mavi',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='yeşil',
...                        marker='o')
...     axarr[idx].set_title(tt)
...     axarr[0].set_ylabel('Seyreltilmiş şarapların OD280/OD315', fontsize=12)
>> plt.tight_layout()
>> plt.text(0, -0.2,
...         s='Alkol',
...         ha='center',
...         va='center',
...         fontsize=12,
...         transform=axarr[1].transAxes)
>> plt.show()
```

olimpikbarbell@gmail.com

Karar bölgelerine bakarak, AdaBoost modelinin karar sınırının, karar kütüğünün karar sınırından önemli ölçüde daha karmaşık olduğunu görebilirsiniz. Ayrıca, AdaBoost modelinin özellik uzayını önceki bölümde eğittiğimiz torbalama sınıflandırıcısına çok benzer şekilde ayırdığına dikkat edin:




*Şekil 7.11: AdaBoost'a karşı karar ağacının karar sınırları*

Topluluk teknikleri ile ilgili son söz olarak, topluluk öğrenmesinin bireysel sınıflandırıcılara kıyasla hesaplama karmaşıklığını artırdığını belirtmek gerekir. Uygulamada, tahmin performansında genellikle nispeten mütevazı bir iyileşme için artan hesaplama maliyetlerinin bedelini ödemek isteyip istemediğimiz konusunda dikkatli düşünmemiz gerekir.

Bu değiş tokuşun sıkça atıfta bulunulan bir örneği, topluluk teknikleri kullanılarak kazanılan ünlü 1 milyon dolarlık *Netflix Ödülü'dür*. Algoritma ile ilgili ayrıntılar *A. Toescher*, *M. Jahrer* ve *R. M. Bell* tarafından *The BigChaos Solution to the Netflix Grand Prize* (Netflix *Büyük Ödülüne* BigChaos *Çözümü*), *Netflix Prize documentation,* 2009, adresinde yayınlanmıştır. http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf. Kazanan ekip 1 milyon dolarlık büyük para ödülünü aldı; ancak Netflix, karmaşıklığı nedeniyle modellerini hiçbir zaman uygulamadı ve bu da onu gerçek dünya uygulaması için uygunsuz hale getirdi:

> *"Yeni yöntemlerden bazılarını ofis dışında değerlendirdik, ancak ölçtüğümüz ek doğruluk kazanımları, bunları üretim ortamına getirmek için gereken mühendislik çabasını haklı çıkaracak gibi görünmüyordu."*
>
> http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-yıldızlar.html

# Gradyan artırma - kayıp gradyanlarına dayalı bir topluluğun eğitimi

Gradient boosting, önceki bölümde tanıtılan boosting konseptinin bir başka çeşididir, yani güçlü bir topluluk oluşturmak için zayıf öğrenicileri art arda eğitmektir. Gradient boosting son derece önemli bir konudur çünkü Kaggle yarışmalarını kazanmasıyla tanınan XGBoost gibi popüler makine öğrenimi algoritmalarının temelini oluşturur.

Gradyan artırma algoritması ilk başta biraz ürkütücü görünebilir. Bu nedenle, aşağıdaki alt bölümlerde, genel bir bakışla başlayarak adım adım ele alacağız. Ardından, gradyan artırmanın sınıflandırma için nasıl kullanıldığını göreceğiz ve bir örnek üzerinden yürüyeceğiz. Son olarak, gradyan artırmanın temel kavramlarını tanıttıktan sonra, XGBoost gibi popüler uygulamalara kısaca göz atacağız ve gradyan artırmayı pratikte nasıl kullanabileceğimizi göreceğiz.

## AdaBoost ile gradyan artırmanın karşılaştırılması

Temel olarak gradyan artırma, bu bölümde daha önce ele aldığımız AdaBoost'a çok benzer. AdaBoost, karar ağacı kütüklerini bir önceki karar ağacı kütüğünün hatalarına dayalı olarak eğitir. Özellikle, hatalar her turda örnek ağırlıklarını hesaplamak ve her bir karar ağacı kütüğünü bir toplulukta birleştirirken her bir karar ağacı kütüğü için bir sınıflandırıcı ağırlığı hesaplamak için kullanılır. Maksimum iterasyon sayısına (karar ağacı kütükleri) ulaşıldığında eğitimi durduruyoruz. AdaBoost gibi, gradyan artırma da tahmin hatalarını kullanarak karar ağaçlarını yinelemeli bir şekilde uyarlar. Bununla birlikte, gradyan artırma ağaçları genellikle karar ağacı kütüklerinden daha derindir ve tipik olarak maksimum 3 ila 6 derinliğe (veya maksimum 8 ila 64 yaprak düğümü sayısına) sahiptir. Ayrıca, AdaBoost'un aksine, gradyan artırma örnek ağırlıklarını atamak için tahmin hatalarını kullanmaz; bunlar doğrudan bir sonraki ağaca uyacak hedef değişkeni oluşturmak için kullanılır. Ayrıca, AdaBoost'ta olduğu gibi her ağaç için ayrı bir ağırlıklandırma terimine sahip olmak yerine, gradyan artırma her ağaç için aynı olan global bir öğrenme oranı kullanır.

Gördüğünüz gibi, AdaBoost ve gradyan artırma birçok benzerliği paylaşmakta ancak bazı temel noktalarda farklılık göstermektedir. Aşağıdaki alt bölümde, gradyan artırma algoritmasının genel taslağını çizeceğiz.

## Genel gradyan artırma algoritmasının ana hatları

Bu bölümde, sınıflandırma için gradyan güçlendirmeye bakacağız. Basitlik açısından, ikili sınıflandırma örneğine bakacağız. İlgilenen okuyucular *Bölüm 4.6'da* lojistik kayıp ile çok sınıflı ortama genelleştirmeyi bulabilirler. *Çok sınıflı lojistik regresyon ve* 2001 yılında Friedman tarafından yazılan orijinal gradyan artırma makalesinin *sınıflandırılması, Greedy fonksiyon yaklaşımı: A gradient boosting ma- chine*, https://projecteuclid.org/journals/annals-of-statistics/volume-29/issue-5/Greedy- function-approximation-A-gradient-boostingmachine/10.1214/aos/1013203451.full.

 olimpikbarbell@gmail.com 

**Regresyon için gradyan artırma**

Gradyan artırmanın arkasındaki prosedürün AdaBoost'tan biraz daha karmaşık olduğunu unutmayın. Friedman'ın makalesinde verilen daha basit bir regresyon örneğini kısa olması için atlıyoruz, ancak ilgilenen okuyucuların https://www.youtube.com/watch?v=zblsrxc7XpM adresinde bulunan regresyon için gradyan artırma hakkındaki tamamlayıcı video eğitimimi de dikkate almaları önerilir.

Temelde, gradyan artırma bir dizi ağaç oluşturur ve her ağaç bir önceki ağacın hatasına (etiket ile tahmin edilen değer arasındaki fark) uyar. Her turda, küçük güncellemelerle her ağacı doğru yönde daha fazla dürttüğümüz için ağaç topluluğu gelişir. Bu güncellemeler bir kayıp gradyanına dayanır ve gradyan artırma adını bu şekilde almıştır.

Aşağıdaki adımlar gradyan artırmanın arkasındaki genel algoritmayı tanıtacaktır. Ana adımları gösterdikten sonra, bazı bölümlerini daha ayrıntılı olarak inceleyeceğiz ve sonraki alt bölümlerde uygulamalı bir örnek üzerinden yürüyeceğiz.

1. Sabit bir tahmin değeri döndürmek için bir modeli başlatın. Bunun için bir karar ağacı kök düğümü, yani tek bir yaprak düğümü olan bir karar ağacı kullanıyoruz. Ağaç tarafından döndürülen değeri şu şekilde gösteriyoruz *yy*, ve bu değeri daha sonra tanımlayacağımız farklılaştırılabilir bir kayıp fonksiyonu *L'yi* minimize ederek buluyoruz:

$$F_0(x) = \arg\min_{y} \sum_{i=1}^{n} L(y_i, \hat{y})$$

Burada *n,* veri setimizdeki *n* eğitim örneğini ifade etmektedir.

2. Her ağaç için $m = 1, ..., M,$ burada *M* kullanıcı tarafından belirlenen toplam ağaç sayısıdır, aşağıdaki *2a* ila *2d adımlarında* özetlenen hesaplamaları gerçekleştiririz:

   a. Tahmin edilen değer $F(x_i) = yy_i$ ile sınıf etiketi $y_i$ arasındaki farkı hesaplayın. Bu değere bazen sözde *yanıt* veya *sözde artık* denir. Daha resmi olarak, bu sözde kalıntıyı, tahmin edilen değerlere göre kayıp fonksiyonunun negatif gradyanı olarak yazabiliriz:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{için } i = 1, 2, \ldots, n$$

   Yukarıdaki gösterimde $F(x)$'in bir önceki ağacın tahmini olduğunu unutmayın, $F_{m-1}(x)$. Dolayısıyla, ilk turda bu, 1. adımdaki ağaçtan (tek yaprak düğümü) elde edilen sabit değeri ifade eder.

   b. Sözde kalıntılara bir ağaç uydurun $r_{im}$ . $j = 1$ ... belirtmek için $R_{jm}$ gösterimini kullanıyoruz. $J_m$ *m* iterasyonunda ortaya çıkan ağacın yaprak düğümleri.

c. Her bir yaprak düğüm $R_{jm}$ için aşağıdaki çıktı değerini hesaplarız:

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

Bir sonraki alt bölümde, kayıp fonksiyonunu minimize ederek bu $\gamma_{jm}$'nin nasıl hesaplandığını daha derinlemesine inceleyeceğiz. Bu noktada, $R_{jm}$ yaprak düğümlerinin birden fazla eğitim örneği içerebileceğini, dolayısıyla toplamayı not edebiliriz.

d. $_m$ $\gamma$ çıktı değerlerini önceki ağaca ekleyerek modeli güncelleyin:

$$F_m(x) = F_{m-1}(x) + \eta\gamma_m$$

Bununla birlikte, mevcut $\gamma_m$ ağacının tam tahmin edilen değerlerini önceki $F_{m-1}$ ağacına eklemek yerine, $\gamma_m$ ağacını tipik olarak aşağıdaki değerler arasında küçük bir değer olan $\eta$ öğrenme oranıyla ölçeklendiririz
0,01 ve 1. Başka bir deyişle, modeli küçük adımlar atarak aşamalı olarak güncelliyoruz, bu da aşırı uyumu önlemeye yardımcı oluyor.

Şimdi, gradyan güçlendirmenin genel yapısına baktıktan sonra, sınıflandırma için gradyan güçlendirmeyi incelemek üzere bu mekaniği benimseyeceğiz.

## Sınıflandırma için gradyan artırma algoritmasının açıklanması

Bu alt bölümde, ikili sınıflandırma için gradyan artırma algoritmasının uygulanmasına ilişkin ayrıntıların üzerinden geçeceğiz. Bu bağlamda, *Bölüm 3, Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu*'nda lojistik regresyon için tanıttığımız lojistik kayıp fonksiyonunu kullanacağız. Tek bir eğitim örneği için lojistik kaybı aşağıdaki gibi belirleyebiliriz:

$$L_i = -y_i \log p_i + (1 - y_i) \log(1 - p_i)$$

*Bölüm 3*'te log(odds) değerini de tanıtmıştık:

$$y_{odds} = \log\left(\frac{p}{1-p}\right)$$

Daha sonra anlamlı olacak nedenlerden dolayı, lojistik fonksiyonu aşağıdaki gibi yeniden yazmak için bu log(odds) değerlerini kullanacağız (burada ara adımları atlıyoruz):

$$L_i = \log(1 + e^{\hat{y}_i}) - y_i\hat{y}_i$$

Şimdi, kayıp fonksiyonunun bu log(odds)'a göre kısmi türevini tanımlayabiliriz, $y_{odds}$. Bu kayıp fonksiyonunun log(odds)'a göre türevi şöyledir:

$$\frac{\partial L_i}{\partial y_{odds}} = \frac{e^{y_{odds}}}{1 + e^{y_{odds}}} - y_i = p_i - y_i$$

Bu matematiksel tanımları belirttikten sonra, şimdi bir önceki bölümde yer alan *1'den 2d'ye* kadar olan genel gradyan artırma *adımlarını* tekrar gözden geçirelim ve bu ikili sınıflandırma senaryosu için yeniden formüle edelim.

1. Lojistik kaybı en aza indiren bir kök düğüm oluşturun. Kök düğüm log(odds), *yy* değerini döndürürse kaybın en aza indirildiği ortaya çıkar.
2. Her ağaç için $m = 1, ..., M,$ burada $M$ kullanıcı tarafından belirlenen toplam ağaç sayısıdır, *2a* ila *2d adımlarında* özetlenen aşağıdaki hesaplamaları gerçekleştiririz:

   a. Lojistik regresyonda (*Bölüm 3*'te) kullandığımız tanıdık lojistik fonksiyonunu kullanarak log(odds) değerini bir olasılığa dönüştürürüz:

   $$p\,p\,\frac{1}{1+e^{-\hat{y}}}$$

   Ardından, kaybın log(odds)'a göre negatif kısmi türevi olan ve sınıf etiketi ile tahmin edilen olasılık arasındaki fark olarak ortaya çıkan sözde kalıntıyı hesaplarız:

   $$-\frac{\partial L_i}{\partial \hat{y}} = y_i - p_i$$

   b. Sözde kalıntılara yeni bir ağaç yerleştirin.

   c. Her yaprak düğümü için $R_{jm}$ , lojistik kayıp fonksiyonunu en aza indiren bir $\gamma_{jm}$ değeri hesaplayın. Bu, birden fazla eğitim örneği içeren yaprak düğümleriyle başa çıkmak için bir özetleme adımı içerir:

   $$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

   $$= \log(1 + e^{\hat{y}_i + \gamma}) - y_i(\hat{y}_i + \gamma)$$

   Ara matematiksel ayrıntıları atlarsak, bu aşağıdaki sonucu verir:

   $$\gamma_{jm} = \frac{\sum_i y_i - p_i}{\sum_i p_i(1 - p_i)}$$

   Buradaki toplamın yalnızca $R_{jm}$ yaprak düğümüne karşılık gelen düğümdeki örnekler üzerinde olduğunu ve tüm eğitim setini kapsamadığını unutmayın.

   d. *Adım 2c'*deki gama değerini $\eta$ öğrenme oranıyla ekleyerek modeli güncelleyin:

   $$F_m(x) = F_{m-1}(x) + \eta\gamma_{jm}$$

**Olasılıklara karşı log(odds) çıktısı alma**

Ağaçlar neden olasılıkları değil de log(odds) değerlerini döndürüyor? Çünkü sadece olasılık değerlerini toplayıp anlamlı bir sonuca ulaşamayız. (Yani, teknik olarak konuşursak, sınıflandırma için gradyan artırma regresyon ağaçlarını kullanır).

Bu bölümde, genel gradyan artırma algoritmasını benimsedik ve örneğin genel kayıp fonksiyonunu lojistik kayıpla ve tahmin edilen değerleri log(odds) ile değiştirerek ikili sınıflandırma için belirledik. Bununla birlikte, bireysel adımların çoğu hala çok soyut görünebilir ve bir sonraki bölümde bu adımları somut bir örneğe uygulayacağız.

## Sınıflandırma için gradyan güçlendirmeyi gösterme

Önceki iki alt bölümde ikili sınıflandırma için gradyan artırma algoritmasının yoğunlaştırılmış matematiksel ayrıntılarının üzerinden geçilmiştir. Bu kavramları daha açık hale getirmek için, küçük bir oyuncak örneğe, yani *Şekil 7.12*'de gösterilen aşağıdaki üç örnekten oluşan bir eğitim veri kümesine uygulayalım:

| | Feature $x_1$ | Feature $x_2$ | Class label y |
|---|---|---|---|
| 1 | 1.12 | 1.4 | 1 |
| 2 | 2.45 | 2.1 | 0 |
| 3 | 3.54 | 1.2 | 1 |

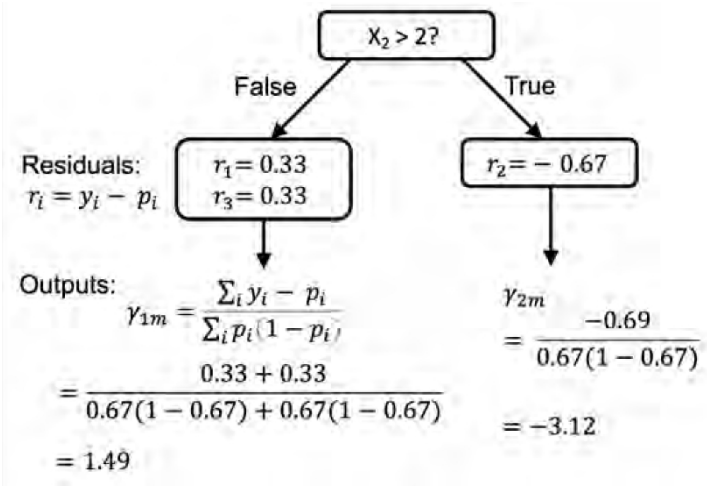
*Şekil 7.12: Gradyan güçlendirmeyi açıklamak için oyuncak veri seti*

*Adım 1 ile* başlayalım, kök düğümü oluşturalım ve log(odds) değerini hesaplayalım ve *adım 2a* ile log(odds) değerini sınıf-üyelik olasılıklarına dönüştürelim ve sözde kalıntıları hesaplayalım. *Bölüm 3*'te öğrendiklerimize dayanarak, oranların başarı sayısının başarısızlık sayısına bölünmesiyle hesaplanabileceğini unutmayın. Burada, 1 etiketini başarı ve 0 etiketini başarısızlık olarak kabul ediyoruz, bu nedenle olasılıklar şu şekilde hesaplanır: odds = 2/1. Adım *1* ve *2a'yı* uyguladığımızda, *Şekil 7.13*'te gösterilen aşağıdaki sonuçları elde ederiz:

| | Feature $x_1$ | Feature $x_2$ | Class label y | Step 1: $\hat{y} = \log(\text{odds})$ | Step 2a: $p = \frac{1}{1+e^{-\hat{y}}}$ | Step 2a: $r = y - p$ |
|---|---|---|---|---|---|---|
| 1 | 1.12 | 1.4 | 1 | 0.69 | 0.67 | 0.33 |
| 2 | 2.45 | 2.1 | 0 | 0.69 | 0.67 | -0.67 |
| 3 | 3.54 | 1.2 | 1 | 0.69 | 0.67 | 0.33 |

*Şekil 7.13: Adım 1 ve Adım 2a'nın ilk tur uygulamasından elde edilen sonuçlar*

Daha sonra, *adım 2b'*de, sözde kalıntılar *r* üzerine yeni bir ağaç yerleştiririz. *2c adımında, Şekil 7.14'te* gösterildiği gibi bu ağaç için $\gamma\gamma$ çıktı değerlerini hesaplarız:

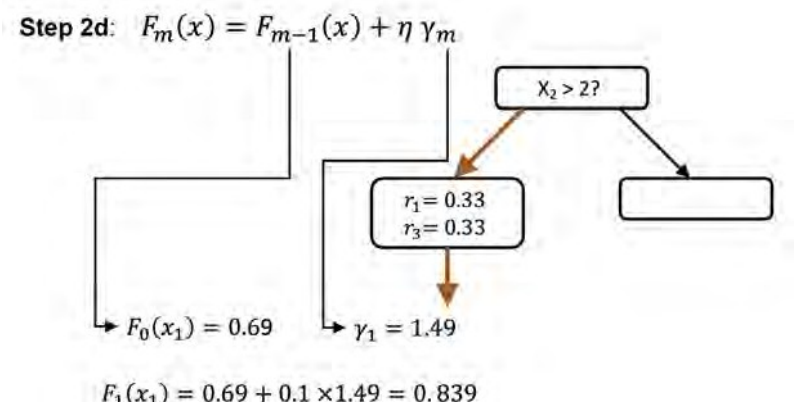



*Şekil 7.14: Kalıntılara bir ağaç uyduran ve her yaprak düğümü için çıktı değerlerini hesaplayan 2b ve 2c adımlarının gösterimi*

(Ağacı yapay olarak sadece iki yaprak düğümü olacak şekilde sınırlandırdığımızı unutmayın, bu da bir yaprak düğümünün birden fazla örnek içermesi durumunda ne olacağını göstermeye yardımcı olur).

Ardından, son *adım 2d'*de, önceki modeli ve mevcut modeli güncelleriz. Öğrenme oranının $\eta$ $\eta$ $\eta\eta 1$ olduğu varsayıldığında, ilk eğitim örneği için elde edilen tahmin *Şekil 7.15*'te gösterilmektedir:

| | Feature $x_1$ | Feature $x_2$ | Class label y |
|---|---|---|---|
| 1 | 1.12 | 1.4 | 1 |
| 2 | 2.45 | 2.1 | 0 |
| 3 | 3.54 | 1.2 | 1 |




*Şekil 7.15: İlk eğitim örneği bağlamında gösterilen önceki modelin güncellenmesi*

Artık $m = 1$ olan ilk turun *2a* ila *2d adımlarını* tamamladığımıza göre, $m = 2$ olan ikinci tur için *2a* ila *2d adımlarını* uygulamaya devam edebiliriz. İkinci turda, güncellenmiş model tarafından döndürülen log(odds) değerini, örneğin $F_1\ (x_1\ ) = 0{,}839$, *2A adımına* girdi olarak kullanırız. İkinci turda elde ettiğimiz yeni değerler *Şekil 7.16'da* gösterilmektedir:

| | $x_1$ | $x_2$ | y | Step 1: $F_0(x) = \hat{y}$ = log(odds) | Step 2a: $p = \frac{1}{1+e^{-\hat{y}}}$ | Step 2a: $r = y - p$ | New log(odds) $\hat{y} = F_1(x)$ | Step 2a: $p$ | Step 2a: $r$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.12 | 1.4 | 1 | 0.69 | 0.67 | 0.33 | 0.839 | 0.698 | 0.302 |
| 2 | 2.45 | 2.1 | 0 | 0.69 | 0.67 | -0.67 | 0.378 | 0.593 | -0.593 |
| 3 | 3.54 | 1.2 | 1 | 0.69 | 0.67 | 0.33 | 0.839 | 0.698 | 0.302 |

Round $m = 1$ Round $m = 2$

*Şekil 7.16: İlk turdaki değerlerin yanında ikinci turdaki değerler*

Tahmin edilen olasılıkların pozitif sınıf için daha yüksek ve negatif sınıf için daha düşük olduğunu zaten görebiliyoruz. Sonuç olarak, artıklar da küçülmektedir. *2a'dan 2d'ye* kadar olan *adımların M* ağaç sığdırana ya da artıklar kullanıcı tarafından belirlenen bir eşik değerinden daha küçük *olana* kadar tekrarlandığını unutmayın. Ardından, gradyan artırma algoritması tamamlandığında, *Bölüm 3'*teki lojistik regresyon gibi nihai modelin olasılık değerlerini $F_M\ (x)$ 0,5'te eşikleyerek sınıf etiketlerini tahmin etmek için kullanabiliriz. Ancak, lojistik regresyonun aksine, gradyan artırma birden fazla ağaçtan oluşur ve doğrusal olmayan karar sınırları üretir. Bir sonraki bölümde, gradyan güçlendirmenin eylemde nasıl göründüğüne bakacağız.

## XGBoost Kullanımı

Gradyan artırmanın arkasındaki ince ayrıntıları ele aldıktan sonra, son olarak gradyan artırma kod uygulamalarını nasıl kullanabileceğimize bakalım.

Scikit-learn'de gradyan artırma sklearn.ensemble.GradientBoostingClassifier olarak uygulanır (bkz. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html daha fazla ayrıntı için). Gradient boosting'in eğitilmesi yavaş olabilen sıralı bir süreç olduğuna dikkat etmek önemlidir. Bununla birlikte, son yıllarda gradyan artırmanın daha popüler bir uygulaması, yani XGBoost ortaya çıkmıştır.

XGBoost, eğitim sürecini önemli ölçüde hızlandıran çeşitli hileler ve yaklaşımlar önermiştir. Bu nedenle, XGBoost adı aşırı gradyan artırma anlamına gelir. Dahası, bu yaklaşımlar ve hileler çok iyi tahmin performansları ile sonuçlanmaktadır. Aslında, XGBoost birçok Kaggle yarışmasında kazanan çözüm olduğu için popülerlik kazanmıştır.

XGBoost'un yanı sıra, LightGBM ve CatBoost gibi diğer popüler gradyan artırma uygulamaları da vardır. LightGBM'den esinlenen scikit-learn artık orijinal gradyan artırma sınıflandırıcısından (GradientBoostingClassifier) daha performanslı olan bir HistGradientBoostingClassifier da uygulamaktadır.

Bu yöntemler hakkında daha fazla bilgiye aşağıdaki kaynaklardan ulaşabilirsiniz:

- XGBoost: https://xgboost.readthedocs.io/en/stable/
- LightGBM: https://lightgbm.readthedocs.io/en/latest/
- CatBoost: https://catboost.ai
- **HistGradientBoostingClassifier**: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.HistGradientBoostingClassifier.html

Bununla birlikte, XGBoost hala en popüler gradyan artırma uygulamaları arasında yer aldığından, pratikte nasıl kullanabileceğimizi göreceğiz. Öncelikle, örneğin pip aracılığıyla yüklememiz

```
pip install xgboost
```

**XGBoost'u Yükleme**

Bu bölüm için, üzerinden yüklenebilen XGBoost 1.5.0 sürümünü kullandık:

```
pip install XGBoost==1.5.0
```

Kurulum detayları hakkında daha fazla bilgiyi https://xgboost adresinde bulabilirsiniz.
readthedocs.io/tr/stable/install.html

gerekiyor:

Neyse ki, XGBoost'un XGBClassifier'ı scikit-learn API'sini takip eder. Bu yüzden kullanımı nispeten kolaydır:

```
>> import xgboost as xgb
>>> model = xgb.XGBClassifier(n_estimators=1000, learning_rate=0.01,
...                           max_depth=4, random_state=1,
...                           use_label_encoder=False)

>> gbm = model.fit(X_train, y_train)
>> y_train_pred = gbm.predict(X_train)
>> y_test_pred = gbm.predict(X_test)

>> gbm_train = accuracy_score(y_train, y_train_pred)
>> gbm_test = accuracy_score(y_test, y_test_pred)
>> print(f'XGboost train/test accuracies '
...       f'{gbm_train:.3f}/{gbm_test:.3f}')
XGboost eğitim/test doğrulukları
0,968/0,917
```

Burada, gradyan artırma sınıflandırıcısını 1.000 ağaç (tur) ve 0,01 öğrenme oranı ile uyarlarız. Genellikle 0,01 ile 0,1 arasında bir öğrenme oranı önerilir. Ancak, öğrenme oranının her bir turdan elde edilen tahminleri ölçeklendirmek için kullanıldığını unutmayın. Dolayısıyla, sezgisel olarak, öğrenme oranı ne kadar düşük olursa, doğru tahminler elde etmek için o kadar fazla tahminci gerekir.

Ardından, bireysel karar ağaçları için max_depth değerini 4 olarak belirledik. Hala zayıf öğrenicileri güçlendirdiğimiz için, 2 ile 6 arasında bir değer makuldür, ancak veri kümesine bağlı olarak daha büyük değerler de işe yarayabilir.

Son olarak, use_label_encoder=False, kullanıcıları XGBoost'un artık varsayılan olarak etiketleri dönüştürmediği konusunda bilgilendiren bir uyarı mesajını devre dışı bırakır ve kullanıcıların etiketleri 0 etiketi ile başlayan bir tamsayı biçiminde sağlamasını bekler (Bu kitap boyunca bu biçimi takip ettiğimiz için burada endişelenecek bir şey yok).

Daha birçok ayar mevcuttur ve ayrıntılı bir tartışma bu kitabın kapsamı dışındadır. Ancak, ilgilenen okuyucular https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.XGBClassifier adresindeki orijinal belgelerde daha fazla ayrıntı bulabilirler.

# Özet

Bu bölümde, topluluk öğrenimi için en popüler ve yaygın olarak kullanılan tekniklerden bazılarını inceledik. Topluluk yöntemleri, farklı sınıflandırma modellerini birleştirerek bireysel zayıflıklarını ortadan kaldırır ve bu da genellikle endüstriyel uygulamalar ve makine öğrenimi yarışmaları için çok cazip olan istikrarlı ve iyi performans gösteren modellerle sonuçlanır.

Bu bölümün başında, sınıflandırma için farklı algoritmaları birleştirmemizi sağlayan MajorityVoteClassifier'ı Python'da uyguladık. Daha sonra, eğitim veri setinden rastgele bootstrap örnekleri çekerek ve ayrı ayrı eğitilmiş sınıflandırıcıları çoğunluk oyu ile birleştirerek bir modelin varyansını azaltmak için yararlı bir teknik olan bagging'i inceledik. Son olarak, daha sonra hatalardan öğrenen zayıf öğrenicileri eğitmeye dayalı algoritmalar olan AdaBoost ve gradient boosting şeklinde boosting hakkında bilgi edindik.

Önceki bölümler boyunca farklı öğrenme algoritmaları, ayarlama ve değerlendirme teknikleri hakkında çok şey öğrendik. Bir sonraki bölümde, internet ve sosyal medya çağında ilginç bir konu haline gelen makine öğrenmesinin özel bir uygulamasına, duygu analizine bakacağız.

# Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.com

olimpikbarbell@gmail.com

# 8

# Makine Öğrenimini Duygu Analizine Uygulama

Modern internet ve sosyal medya çağında, insanların görüşleri, yorumları ve tavsiyeleri siyaset bilimi ve işletmeler için değerli bir kaynak haline geldi. Modern teknolojiler sayesinde artık bu tür verileri en verimli şekilde toplayabiliyor ve analiz edebiliyoruz. Bu bölümde, **doğal dil işlemenin** (**NLP**) **duygu analizi** adı verilen bir alt alanını inceleyecek ve belgeleri duygularına göre sınıflandırmak için makine öğrenimi algoritmalarının nasıl kullanılacağını öğreneceğiz: yazarın tutumu. Özellikle, **Internet Movie Database**'den (**IMDb**) 50.000 film eleştirisinden oluşan bir veri kümesi ile çalışacağız ve olumlu ve olumsuz eleştirileri ayırt edebilen bir tahminci oluşturacağız.

Bu bölümde ele alacağımız konular aşağıdakileri içermektedir:

- Metin verilerinin temizlenmesi ve hazırlanması
- Metin belgelerinden özellik vektörleri oluşturma
- Olumlu ve olumsuz film eleştirilerini sınıflandırmak için bir makine öğrenimi modelinin eğitilmesi
- Çekirdek dışı öğrenme kullanarak büyük metin veri kümeleriyle çalışma
- Kategorizasyon için belge koleksiyonlarından konu çıkarımı

## IMDb film inceleme verilerini metin işleme için hazırlama

Daha önce de belirtildiği gibi, bazen **fikir madenciliği** olarak da adlandırılan duygu analizi, daha geniş bir NLP alanının popüler bir alt disiplinidir; belgelerin duygularını analiz etmekle ilgilenir. Duygu analizindeki popüler bir görev, yazarların belirli bir konuyla ilgili ifade edilen görüşlerine veya duygularına dayalı olarak belgelerin sınıflandırılmasıdır.

Bu bölümde, Andrew Maas ve diğerleri tarafından IMDb'den toplanan film eleştirilerinden oluşan büyük bir veri kümesi ile çalışacağız (*Learning Word Vectors for Sentiment Analysis*, *A. L. Maas*, *R. E. Daly*, *P. T. Pham*, *D. Huang*, *A. Y. Ng,* and *C. Potts, Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, sayfa 142-150, Portland, Oregon, ABD, Association for Computational Linguistics, Haziran 2011). Film incelemesi veri kümesi, olumlu veya olumsuz olarak etiketlenmiş 50.000 kutupsal film incelemesinden oluşur; burada olumlu, bir filmin IMDb'de altıdan fazla yıldızla derecelendirildiği ve olumsuz, bir filmin IMDb'de beşten az yıldızla derecelendirildiği anlamına gelir. İlerleyen bölümlerde, veri kümesini indireceğiz, makine öğrenimi araçları için kullanılabilir bir biçime getireceğiz ve belirli bir eleştirmenin bir filmi beğenip beğenmediğini tahmin edebilecek bir makine öğrenimi modeli oluşturmak için bu film incelemelerinin bir alt kümesinden anlamlı bilgiler çıkaracağız.

## Film inceleme veri kümesinin elde edilmesi

Film inceleme veri setinin sıkıştırılmış bir arşivi (84,1 MB) http:// ai.stanford.edu/~amaas/data/sentiment/ adresinden gzip sıkıştırmalı bir tarball arşivi olarak indirilebilir:

- Linux veya macOS ile çalışıyorsanız, yeni bir terminal penceresi açabilir, indirme dizinine cd gönderebilir ve veri kümesini açmak için tar -zxf aclImdb_v1.tar.gz komutunu çalıştırabilirsiniz.
- Windows ile çalışıyorsanız, dosyaları indirme arşivinden çıkarmak için 7-Zip (http:// www.7-zip.org) gibi ücretsiz bir arşivleyici indirebilirsiniz.
- Alternatif olarak, gzip ile sıkıştırılmış tarball arşivini doğrudan Python'da aşağıdaki şekilde açabilirsiniz:

```
>> import tarfile
>> with tarfile.open('aclImdb_v1.tar.gz', 'r:gz') as tar:
...     tar.extractall()
```

## Film veri setini daha uygun bir formata dönüştürmek için ön işleme

Veri kümesini başarıyla ayıkladıktan sonra, şimdi sıkıştırılmış indirme arşivindeki metin belgelerini tek bir CSV dosyasında bir araya getireceğiz. Aşağıdaki kod bölümünde, film incelemelerini bir pandas DataFrame nesnesine okuyacağız, bu işlem standart bir masaüstü bilgisayarda 10 dakika kadar sürebilir.

İlerlemeyi ve tamamlanana kadar tahmini süreyi görselleştirmek için, birkaç yıl önce bu tür amaçlar için geliştirilen **Python Progress Indica- tor** (**PyPrind**, https://pypi.python.org/pypi/PyPrind/) paketini kullanacağız. PyPrind, pip install pyprind komutu çalıştırılarak kurulabilir:

```
>> import pyprind
>> import pandas as pd
>> import os
>> import sys
>> # 'basepath'i dizin olarak değiştir
>> # sıkıştırılmış film veri kümesini aç
>>> basepath = 'aclImdb'
```

 olimpikbarbell@gmail.com 

```
>>>
>>> labels = {'pos': 1, 'neg': 0}
>> pbar = pyprind.ProgBar(50000, stream=sys.stdout)
>> df = pd.DataFrame()
>> for s in ('test', 'train'):
...     for l in ('pos', 'neg'):
...         path = os.path.join(basepath, s, l)
...         for dosya in sorted(os.listdir(path)):
...             ile open(os.path.join(path, dosya),
...                         'r', encoding='utf-8') as infile:
...                 txt = infile.read()
...             df = df.append([[txt, labels[l]]],
...                             ignore_index=True)
...             pbar.update()
>> df.columns = ['review', 'sentiment'] 0%
                                100%
[################################] | ETA: 00:00:00
Geçen toplam süre: 00:00:25
```

Önceki kodda, ilk olarak yeni bir ilerleme çubuğu nesnesi olan pbar'ı, okuyacağımız belge sayısı olan 50.000 yineleme ile başlattık. İç içe for döngülerini kullanarak, ana aclImdb dizinindeki train ve test alt dizinlerini yineledik ve sonunda df pandas DataFrame'e eklediğimiz pos ve neg alt dizinlerinden ayrı ayrı metin dosyalarını bir tamsayı sınıf etiketiyle (1 = pozitif ve 0 = negatif) birlikte okuduk.

Birleştirilmiş veri kümesindeki sınıf etiketleri sıralandığından, şimdi np.random alt modülündeki permütasyon işlevini kullanarak DataFrame'i karıştıracağız - bu, verileri doğrudan yerel sürücümüzden aktaracağımız sonraki bölümlerde veri kümesini eğitim ve test veri kümelerine bölmek için yararlı olacaktır.

Kendi rahatlığımız için, birleştirilmiş ve karıştırılmış film inceleme veri kümesini bir CSV dosyası olarak da saklayacağız:

```
>> import numpy as np
>> np.random.seed(0)
>>>  df = d f .reindex(np.random.permutation(df.index))
>> df.to_csv('movie_data.csv', index=False, encoding='utf-8')
```

Bu veri kümesini bu bölümün ilerleyen kısımlarında kullanacağımız için, CSV'yi okuyarak ve ilk üç örneğin bir alıntısını yazdırarak verileri doğru formatta başarılı bir şekilde kaydettiğimizi hızlıca doğrulayalım:

```
>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
>> # aşağıdaki sütun yeniden adlandırması bazı bilgisayarlarda gereklidir:
>> df = df.rename(columns={"0": " review", " 1": " sentiment"})
>> df.head(3)
```

Kod örneklerini bir Jupyter not defterinde çalıştırıyorsanız, şimdi *Şekil 8.1*'de gösterildiği gibi veri kümesinin ilk üç örneğini görmelisiniz:

| | review | sentiment |
|---|---|---|
| 0 | In 1974, the teenager Martha Moxley (Maggie Gr... | 1 |
| 1 | OK... so... I really like Kris Kristofferson a... | 0 |
| 2 | ***SPOILER*** Do not read this, if you think a... | 0 |

*Şekil 8.1: Film inceleme veri kümesinin ilk üç satırı*

Bir sonraki bölüme geçmeden önce akıl sağlığı kontrolü olarak `DataFrame`'in 50.000 satırın tamamını içerdiğinden emin olalım:

```
>> df.shape
(50000, 2)
```

# Kelime torbası modeliyle tanışın

*İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme* başlıklı *4. Bölüm'd*en hatırlayacağınız üzere, metin veya sözcükler gibi kategorik verileri makine öğrenimi algoritmasına aktarmadan önce sayısal bir forma dönüştürmemiz gerekir. Bu bölümde, metni sayısal özellik vektörleri olarak temsil etmemizi sağlayan **kelime torbası** modelini tanıtacağız. Kelime torbasının arkasındaki fikir oldukça basittir ve aşağıdaki gibi özetlenebilir:

1. Tüm belge kümesinden benzersiz belirteçlerden (örneğin, kelimeler) oluşan bir kelime dağarcığı oluşturuyoruz.
2. Her belgeden, her bir kelimenin söz konusu belgede ne sıklıkta geçtiğine dair sayıları içeren bir özellik vektörü oluşturuyoruz.

Her belgedeki benzersiz sözcükler, sözcük dağarcığındaki tüm sözcüklerin yalnızca küçük bir alt kümesini temsil ettiğinden, özellik vektörleri çoğunlukla sıfırlardan oluşacaktır, bu yüzden onlara **seyrek** diyoruz. Bu kulağa çok soyut geliyorsa endişelenmeyin; aşağıdaki alt bölümlerde basit bir kelime torbası modeli oluşturma sürecini adım adım inceleyeceğiz.

## Kelimeleri özellik vektörlerine dönüştürme

İlgili belgelerdeki kelime sayılarına dayalı bir kelime torbası modeli oluşturmak için scikit-learn'de uygulanan `CountVectorizer` sınıfını kullanabiliriz. Aşağıdaki kod bölümünde göreceğiniz gibi `CountVectorizer`, belge veya cümle olabilen bir dizi metin verisi alır ve bizim için kelime torbası modelini oluşturur:

```
>> import numpy as np
>>  from sklearn.feature_extraction.text import CountVectorizer
>> count = CountVectorizer()
>>> docs = np.array(['Güneş parlıyor',
```

```
...                    'Hava çok güzel',
...                    'Güneş parlıyor, hava tatlı,' 've bir ve bir iki
...                    eder'])
>>> bag = count.fit_transform(docs)
```

CountVectorizer üzerinde fit_transform yöntemini çağırarak, kelime çantası modelinin kelime dağarcığını oluşturduk ve aşağıdaki üç cümleyi seyrek özellik vektörlerine dönüştürdük:

- 'Güneş parlıyor'
- 'Hava çok güzel'
- 'Güneş parlıyor, hava güzel ve bir ve bir iki eder'

Şimdi, altta yatan kavramları daha iyi anlamak için kelime dağarcığının içeriğini yazdıralım:

```
>>  print(count.vocabulary_)
{'ve': 0,
'iki': 7,
'parlıyor': 3,
'bir': 2,
"Güneş": 4,
'hava durumu': 8,
'the': 6,
'tatlı': 5,
'is': 1}
```

Önceki komutun çalıştırılmasından da görebileceğiniz gibi, kelime dağarcığı, benzersiz kelimeleri tamsayı indeksleriyle eşleyen bir Python sözlüğünde saklanır. Daha sonra, yeni oluşturduğumuz özellik vektörlerini yazdıralım:

```
>> print(bag.toarray())
[[0 1 0 1 0 1 0 0 ]
 [0 1 0 0 0 1 1 0 1 ]
 [2 3 2 1 1 1 2 1 1 ]]
```

Burada gösterilen özellik vektörlerindeki her bir indeks konumu CountVectorizer sözlüğünde sözlük öğeleri olarak saklanan tamsayı değerlerine karşılık gelmektedir. Örneğin, 0 indeks konumundaki ilk özellik, yalnızca son belgede geçen 'and' kelimesinin sayısına benzemektedir ve 1 indeks konumundaki 'is' kelimesi (belge vektörlerindeki ikinci özellik) üç cümlede de geçmektedir. Özellik vektörlerindeki bu değerlere **ham terim frekansları** da denir: *tf*(*t, d*)-bir terimin, *t*, bir belgede, *d*, kaç kez geçtiği. Kelime çantası modelinde, bir cümle veya belgedeki kelime veya terim sırasının önemli olmadığı unutulmamalıdır. Terim frekanslarının özellik vektöründe görünme sırası, genellikle alfabetik olarak atanan kelime dağarcığı indekslerinden türetilir.

**N-gram modelleri**

Az önce oluşturduğumuz kelime torbası modelindeki öğeler dizisine 1-gram ya da unigram modeli de denir - kelime dağarcığındaki her öğe ya da belirteç tek bir kelimeyi temsil eder. Daha genel olarak, NLP'deki bitişik öğe dizilerine - sözcükler, harfler veya semboller - *n-gramlar* da denir. N-gram modelindeki *n* sayısının seçimi belirli bir uygulamaya bağlıdır; örneğin, Ioannis Kanaris ve diğerleri tarafından yapılan bir çalışma, 3 ve 4 boyutlu n-gramların e-posta mesajlarının anti-spam filtrelemesinde iyi performans verdiğini ortaya koymuştur (*Words versus character n-grams for anti-spam filtering* by *Ioannis Kanaris*, *Konstantinos Kanaris, Ioannis Houvardas*, and *Efstathios Stamatatos, International Journal on Artificial Intelligence Tools*, *World Scientific Publishing Company,* 16(06): 1047-1067, 2007).

N-gram gösterim kavramını özetlemek gerekirse, ilk belgemiz olan "güneş parlıyor" ifadesinin 1-gram ve 2-gram gösterimleri aşağıdaki gibi oluşturulacaktır:

- 1-gram: "the", "sun", "is", "shining"
- 2-gram: "güneş", "güneş", "parlıyor"

Scikit-learn'deki `CountVectorizer` sınıfı, `ngram_range` parametresi aracılığıyla farklı n-gram modelleri kullanmamıza olanak tanır. Varsayılan olarak 1-gram temsili kullanılırken, yeni bir `CountVectorizer` örneğini `ngram_range=(2,2)` ile başlatarak 2-gram temsiline geçebiliriz.

## Terim sıklığı-ters belge sıklığı aracılığıyla kelime alaka düzeyinin değerlendirilmesi

Metin verilerini analiz ederken, genellikle her iki sınıftan birden fazla belgede geçen kelimelerle karşılaşırız. Sıklıkla karşılaşılan bu kelimeler genellikle yararlı veya ayırt edici bilgiler içermez. Bu alt bölümde, özellik vektörlerinde bu sık geçen kelimelerin ağırlığını azaltmak için kullanılabilecek, **terim sıklığı içinde belge sıklığı (tf-idf)** adı verilen faydalı bir teknik hakkında bilgi edineceksiniz. tf-idf, terim frekansı ile ters belge frekansının çarpımı olarak tanımlanabilir:

$$tf\text{-}idf(t, d) = tf(t, d) \times idf(t, d)$$

Burada, $tf(t, d)$ önceki bölümde tanıttığımız terim frekansıdır ve $idf(t, d)$ aşağıdaki gibi hesaplanabilen ters belge frekansıdır:

$$idf(t, d) = \log \frac{n_d}{1 + df(d, t)}$$

Burada, $n_d$ toplam belge sayısıdır ve $df(d, t)$ $t$ terimini içeren belge sayısıdır ($d$). 1 sabitinin paydaya eklenmesinin isteğe bağlı olduğunu ve eğitim örneklerinin hiçbirinde geçmeyen terimlere sıfır olmayan bir değer atama amacına hizmet ettiğini unutmayın; *log*, düşük belge frekanslarına çok fazla ağırlık verilmemesini sağlamak için kullanılır.

olimpikbarbell@gmail.com

Scikit-learn kütüphanesi, CountVectorizer sınıfından ham terim frekanslarını girdi olarak alan ve bunları tf-idfs'ye dönüştüren başka bir dönüştürücü olan TfidfTransformer sınıfını uygular:

```
>> from sklearn.feature_extraction.text import TfidfTransformer
>> tfidf = TfidfTransformer(use_idf=True,
...                         norm='l2',
...                         smooth_idf=True)
>> np.set_printoptions(precision=2)
>> print(tfidf.fit_transform(count.fit_transform(docs))
...        .toarray())
[[ 0.    0.43  0.    0.56  0.56  0.    0.43  0.    0.  ]
 [ 0.    0.43  0.    0.    0.    0.56  0.43  0.    0.56]
 [ 0.5   0.45  0.5   0.19  0.19  0.19  0.3   0.25  0.19]]
```

Önceki alt bölümde gördüğünüz gibi, 'is' kelimesi üçüncü belgede en sık geçen kelime olarak en büyük terim frekansına sahipti. Ancak, aynı özellik vektörünü tf-idf'lere dönüştürdükten sonra, 'is' kelimesi artık üçüncü belgede nispeten küçük bir tf-idf (0,45) ile ilişkilidir, çünkü birinci ve ikinci belgede de mevcuttur ve bu nedenle herhangi bir yararlı ayırt edici bilgi içermesi olası değildir.

Ancak, özellik vektörlerimizdeki tek tek terimlerin tf-idf'lerini manuel olarak hesaplamış olsaydık, TfidfTransformer'ın tf-idf'leri daha önce tanımladığımız standart ders kitabı denklemlerine kıyasla biraz farklı hesapladığını fark ederdik. Scikit-learn'de uygulanan ters belge sıklığı denklemi aşağıdaki gibi hesaplanır:

$$idf(t, d) = \log \frac{1 + n_d}{1 + df(d, t)}$$

Benzer şekilde, scikit-learn'de hesaplanan tf-idf, daha önce tanımladığımız varsayılan denklemden biraz sapmaktadır:

$$tf\text{-}idf(t, d) = tf(t, d) \times (idf(t, d) + 1)$$

Önceki idf denklemindeki "+1 "in, önceki kod örneğinde smooth_idf=True olarak ayarlanmasından kaynaklandığını unutmayın; bu, tüm belgelerde geçen terimlere sıfır ağırlık (yani $idf(t, d) = \log(1) = 0$) atanmasına yardımcı olur.

Tf-idfs'yi hesaplamadan önce ham terim frekanslarını normalleştirmek daha tipik olsa da, TfidfTransformer sınıfı tf-idfs'yi doğrudan normalleştirir. *Varsayılan olarak* (norm='l2'), scikit-learn'in TfidfTransformer'ı, normalleştirilmemiş bir özellik *vektörünü (v)* L2-normuna bölerek 1 uzunluğunda bir *vektör* döndüren L2-normalleştirmesini uygular:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}} = \frac{v}{\left(\sum_{i=1}^{n} v_i^2\right)^{1/2}}$$

TfidfTransformer'ın nasıl çalıştığını anladığımızdan emin olmak için bir örnek üzerinden gidelim ve üçüncü belgedeki `'is'` kelimesinin tf-idf'sini hesaplayalım. `'is'` sözcüğü üçüncü belgede 3 terim sıklığına ($tf = 3$) sahiptir ve 'is' terimi her üç belgede de geçtiği için bu terimin belge sıklığı 3'tür ($df = 3$). Böylece, ters belge sıklığını aşağıdaki gibi hesaplayabiliriz:

$$idf(\text{"dır"}, d_3) = \log \frac{1+3}{1+3} = 0$$

Şimdi, tf-idf'yi hesaplamak için, ters belge frekansına 1 eklememiz ve bunu terim frekansı ile çarpmamız gerekir:

$$tf\text{-}idf(\text{"dır"}, d_3) = 3 \times (0+1) = 3$$

Bu hesaplamayı üçüncü belgedeki tüm terimler için tekrarlarsak, aşağıdaki tf-idf vektörlerini elde ederiz: `[3.39, 3.0, 3.39, 1.29, 1.29, 2.0, 1.69, 1.29]`. Ancak, bu özellik vektöründeki değerlerin daha önce kullandığımız TfidfTransformer'dan elde ettiğimiz değerlerden farklı olduğuna dikkat edin. Bu tf-idf hesaplamasında eksik olan son adım, aşağıdaki şekilde uygulanabilecek L2-normalizasyonudur:

$$tf\text{-}idf(d_3)_{norm} = \frac{[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]}{\sqrt{3.39^2 + 3.0^2 + 3.39^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.0^2 + 1.69^2 + 1.29^2}}$$

$$= [0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19]$$

$$tf\text{-}idf(\text{"dır"}, d_3) = 0{,}45$$

Gördüğünüz gibi, sonuçlar artık scikit-learn'in TfidfTransformer'ı tarafından döndürülen sonuçlarla eşleşiyor ve artık tf-idfs'nin nasıl hesaplandığını anladığınıza göre, bir sonraki bölüme geçelim ve bu kavramları film incelemesi veri kümesine uygulayalım.

## Metin verilerinin temizlenmesi

Önceki alt bölümlerde kelime torbası modeli, terim frekansları ve tf-idfs hakkında bilgi edinmiştik. Ancak, kelime torbası modelimizi oluşturmadan önce ilk önemli adım, metin verilerini istenmeyen tüm karakterlerden arındırarak temizlemektir.

Bunun neden önemli olduğunu göstermek için, yeniden karıştırılmış film incelemesi veri kümesindeki ilk belgenin son 50 karakterini görüntüleyelim:

```
>>> df.loc[0, 'review'][-50:]
'yedi.<br /><br />Başlık (Brezilya): Mevcut değil'
```

Burada görebileceğiniz gibi, metin HTML işaretlemesinin yanı sıra noktalama işaretleri ve harf olmayan diğer karakterleri de içermektedir. HTML işaretlemesi pek çok yararlı anlambilim içermese de, noktalama işaretleri belirli NLP bağlamlarında yararlı, ek bilgileri temsil edebilir. Ancak, basitlik açısından, :) gibi ifade karakterleri hariç tüm noktalama işaretlerini kaldıracağız, çünkü bunlar duygu analizi için kesinlikle yararlıdır.

 olimpikbarbell@gmail.com 

Bu görevi gerçekleştirmek için, burada gösterildiği gibi Python'un **düzenli ifade** (**regex**) kütüphanesi re'yi kullanacağız:

```
>> import re
>> def önişlemci(metin):
...     metin = re.sub('<[^>]*>', '', metin)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
...                            metin)
...                 metin = (re.sub('[\W]+', ' ',
...                                 metin.lower()) +
...             ' '.join(emoticons).replace('-', ''))
...     metin döndür
```

Önceki kod bölümündeki ilk regex olan <[^>]*> aracılığıyla, film incelemelerindeki tüm HTML işaretlemelerini kaldırmaya çalıştık. Birçok programcı genellikle HTML'yi ayrıştırmak için regex kullanımına karşı tavsiyede bulunsa da, bu regex bu özel veri kümesini *temizlemek* için yeterli olacaktır. Yalnızca HTML işaretlemesini kaldırmakla ilgilendiğimiz ve HTML işaretlemesini daha fazla kullanmayı planlamadığımız için, işi yapmak için regex kullanmak kabul edilebilir olmalıdır. Ancak, HTML işaretlemesini metinden kaldırmak için gelişmiş araçlar kullanmayı tercih ederseniz, https://docs. python.org/3/library/html.parser.html adresinde açıklanan Python'un HTML ayrıştırıcı modülüne göz atabilirsiniz. HTML işaretlemesini kaldırdıktan sonra, geçici olarak ifade olarak sakladığımız ifadeleri bulmak için biraz daha karmaşık bir regex kullandık. Ardından, [\W]+ regex'ini kullanarak metindeki tüm kelime olmayan karakterleri kaldırdık ve metni küçük harflere dönüştürdük.

**Kelime büyük harf kullanımı ile başa çıkma**

Bu analiz bağlamında, bir kelimenin büyük harfle yazılmasının - örneğin cümlenin başında yer alıp almamasının - anlamsal olarak ilgili bilgi içermediğini varsayıyoruz. Bununla birlikte, istisnalar olduğunu unutmayın; örneğin, özel isimlerin gösterimini kaldırıyoruz. Ancak yine de, bu analiz bağlamında, harf durumunun duygu analizi ile ilgili bilgi içermediği basitleştirici bir varsayımdır.

Sonunda, geçici olarak saklanan ifadeleri işlenen belge dizesinin sonuna ekledik. Ek olarak, tutarlılık için ifadelerden *burun* karakterini (- in :-)) kaldırdık.

**Düzenli ifadeler**

Düzenli ifadeler bir dize içindeki karakterleri aramak için verimli ve kullanışlı bir yaklaşım sunsa da, aynı zamanda dik bir öğrenme eğrisine de sahiptir. Ne yazık ki, düzenli ifadeler hakkında derinlemesine bir tartışma bu kitabın kapsamı dışındadır. Ancak, Google Developers portalında https://developers.google. com/edu/python/regular-expressions adresinde harika bir öğretici bulabilir veya https://docs.python.org/3.9/library/re.html adresinde Python'un re modülünün resmi belgelerine göz atabilirsiniz.

İfade karakterlerinin temizlenmiş belge dizelerinin sonuna eklenmesi en şık yaklaşım gibi görünmese de, kelime dağarcığımız yalnızca tek kelimelik belirteçlerden oluşuyorsa, kelimelerin sırasının kelime çantası modelimizde önemli olmadığını unutmamalıyız. Ancak belgelerin tek tek terimlere, kelimelere veya belirteçlere bölünmesi hakkında daha fazla konuşmadan önce, önişlemci işlevimizin doğru çalıştığını doğrulayalım:

```
>> preprocessor(df.loc[0, 'review'][-50:])
'is seven title brazil not available'
>> preprocessor("</a>Bu :) :( bir test :-)!") 'bu bir
test : ) :( :)'
```

Son olarak, sonraki bölümlerde *temizlenmiş* metin verilerini tekrar tekrar kullanacağımız için, şimdi önişlemci fonksiyonumuzu DataFrame'imizdeki tüm film incelemelerine uygulayalım:

```
>> df['inceleme'] = d f ['inceleme'].apply(önişlemci)
```

## Belgeleri belirteçlere dönüştürme

Film incelemesi veri kümesini başarıyla hazırladıktan sonra, şimdi metin derlemini tek tek öğelere nasıl böleceğimizi düşünmemiz gerekiyor. Belgeleri *tokenize* etmenin bir yolu, temizlenmiş belgeleri boşluk karakterlerinden bölerek tek tek kelimelere ayırmaktır:

```
>> def tokenizer(metin):
...         return text.split()
>>> tokenizer('runners like running and thus they run')
['runners', ' like', ' running', 'and', ' thus', 'they',
' run']
```

Tokenleştirme bağlamında, bir başka yararlı teknik de bir kelimeyi kök formuna dönüştürme işlemi olan **kelime** köklendirmedir. Birbiriyle ilişkili kelimeleri aynı kökle eşleştirmemizi sağlar. Orijinal stemming algoritması 1979 yılında Martin F. Porter tarafından geliştirilmiştir ve bu nedenle **Porter stemmer** algoritması olarak bilinir (*An algorithm for suffix stripping* by *Martin F. Porter*, *Program: Elektronik Kütüphane ve Bilgi Sistemleri*, 14(3): 130-137, 1980). Python için **Natural Language Toolkit** (**NLTK**, http://www. nltk.org), aşağıdaki kod bölümünde kullanacağımız Porter stemming algoritmasını uygular. NLTK'yı yüklemek için conda install nltk veya pip install nltk komutlarını çalıştırabilirsiniz.

**NLTK çevrimiçi kitap**

NLTK bu bölümün odak noktası olmasa da, NLP'de daha ileri uygulamalarla ilgileniyorsanız NLTK web sitesini ziyaret etmenizi ve http:// www.nltk.org/book/ adresinde ücretsiz olarak bulunan resmi NLTK kitabını okumanızı şiddetle tavsiye ederim.

Aşağıdaki kod Porter stemming algoritmasının nasıl kullanılacağını göstermektedir:

```
>> from nltk.stem.porter import PorterStemmer
>>> porter = PorterStemmer()
```

```
>> def tokenizer_porter(text):
...         return [porter.stem(word) for word in text.split()]
>>> tokenizer_porter('runners like running and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'r u n ' ]
```

nltk paketindeki PorterStemmer'ı kullanarak, sözcükleri kök biçimlerine indirgemek için tokenizer işlevimizi değiştirdik; bu, 'running' sözcüğünün kök biçimi olan 'run'a indirgendiği önceki basit örnekte gösterilmiştir.

**Stemming algoritmaları**

Porter köklendirme algoritması muhtemelen en eski ve en basit köklendirme algoritmasıdır. Diğer popüler stemming algoritmaları arasında yeni Snowball stemmer (Porter2 veya English stemmer) ve Lancaster stemmer (Paice/Husk stemmer) bulunmaktadır. Hem Snowball hem de Lancaster kökleyiciler orijinal Porter kökleyiciden daha hızlı olmakla birlikte, Lan- caster kökleyici Porter kökleyiciden daha agresif olmasıyla da ünlüdür, bu da daha kısa ve daha belirsiz kelimeler üreteceği anlamına gelir. Bu alternatif köklendirme algoritmaları NLTK paketinde de mevcuttur (http://www.nltk.org/api/ nltk.stem.html).

Bir önceki örnekte gösterildiği gibi, köklendirme 'thu' ('thus' kelimesinden) gibi gerçek olmayan kelimeler yaratabilirken, *lemmatizasyon* adı verilen bir teknik, tek tek kelimelerin kanonik (dilbilgisi açısından doğru) biçimlerini elde etmeyi amaçlar - sözde *lemmalar*. Bununla birlikte, lemmatizasyon köklendirmeye kıyasla hesaplama açısından daha zor ve pahalıdır ve pratikte köklendirme ve lemmatizasyonun metin sınıflandırma performansı üzerinde çok az etkisi olduğu gözlemlenmiştir (*Influence of Word Normalization on Text Classification*, *Michal Toman*, *Roman Tesar,* and *Karel Jezek, Proceedings of InSciT*, pages 354-358, 2006).

Kelime torbası modelini kullanarak bir makine öğrenimi modelini eğiteceğimiz bir sonraki bölüme geçmeden önce, **durdurma kelimesi kaldırma** adı verilen başka bir yararlı konudan kısaca bahsedelim. Durdurma sözcükleri, her tür metinde son derece yaygın olan ve muhtemelen farklı belge sınıfları arasında ayrım yapmak için kullanılabilecek hiçbir yararlı bilgi taşımayan (veya çok az taşıyan) sözcüklerdir. Durdurma sözcüklerine örnek *olarak is*, *and*, *has* ve *like verilebilir*. Eğer tf-idfs yerine ham ya da normalize edilmiş terim frekanslarıyla çalışıyorsak, durak sözcükleri kaldırmak yararlı olabilir.

Film incelemelerinden durdurma sözcüklerini kaldırmak için, NLTK kütüphanesinde bulunan ve nltk.download işlevini çağırarak elde edilebilen 127 İngilizce durdurma sözcüğü kümesini

```
>> import nltk
>> nltk.download('stopwords')
```

kullanacağız:

Durdurma kelimeleri setini indirdikten sonra, İngilizce durdurma kelimesi setini aşağıdaki gibi yükleyebilir ve uygulayabiliriz:

```
>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>> [w for w in tokenizer_porter('a runner likes')
... 'koşuyor ve çok koşuyor')
... if w not in stop]
['runner', 'like', 'run', 'run', 'lot']
```

# Belge için lojistik regresyon modeli eğitimi sınıflandırma

Bu bölümde, kelime çantası modeline dayalı olarak film eleştirilerini *olumlu* ve *olumsuz* eleştiriler olarak sınıflandırmak için bir lojistik regresyon modeli eğiteceğiz. İlk olarak, temizlenmiş metin belgelerinden oluşan DataFrame'i eğitim için 25.000 ve test için 25.000 belgeye böleceğiz:

```
>>> X_train = df.loc[:25000, 'review'].values
>> y_train = df.loc[:25000, 'sentiment'].values
>> X_test = df.loc[25000:, 'review'].values
>> y_test = df.loc[25000:, 'sentiment'].values
```

Ardından, 5 katlı tabakalı çapraz doğrulama kullanarak lojistik regresyon modelimiz için en uygun parametre kümesini bulmak için bir GridSearchCV nesnesi kullanacağız:

```
>> from sklearn.model_selection import GridSearchCV
>> from sklearn.pipeline import Pipeline
>> from sklearn.linear_model import LogisticRegression
>>  from sklearn.feature_extraction.text import TfidfVectorizer
>> tfidf = TfidfVectorizer(strip_accents=None,
...                         küçük harf=False,
...                         preprocessor=None)
>> small_param_grid = [
...     {
...         'vect ngram_range': [(1, 1)],
...         'vect stop_words': [Yok],
...         'vect tokenizer': [tokenizer, tokenizer_porter],
...         'clf penalty': ['l2'],
...         "clf C": [1.0, 10.0]
...     },
...     {
...         'vect ngram_range': [(1, 1)],
...         'vect stop_words': [stop, None],
...         'vect tokenizer': [tokenizer],
```

```
...             'vect use_idf':[False],
...             'vect norm':[Yok],
...             'clf ceza': ['l2'],
...             "clf C": [1.0, 10.0]
...         },
... ]
>> lr_tfidf = Boru Hattı([
...         ('vect', tfidf),
...         ('clf', LogisticRegression(solver='liblinear'))
... ])
>> gs_lr_tfidf = GridSearchCV(lr_tfidf, small_param_grid,
...                                 scoring='accuracy', cv=5,
...                                 verbose=2, n_jobs=1)
>> gs_lr_tfidf.fit(X_train, y_train)
```

Lojistik regresyon sınıflandırıcısı için, nispeten büyük veri kümeleri için varsayılan seçimden ('lbfgs') daha iyi performans gösterebileceğinden LIBLINEAR çözücüsünü kullandığımızı unutmayın.

**n_jobs parametresi aracılığıyla çoklu işlem**

Makinenizdeki mevcut tüm çekirdekleri kullanmak ve ızgara aramasını hızlandırmak için n_jobs=-1 (önceki kod örneğinde olduğu gibi n_jobs=1 yerine) ayarını yapmanızı şiddetle tavsiye ettiğimizi lütfen unutmayın. Ancak, bazı Windows kullanıcıları önceki kodu n_jobs=-1 ayarıyla çalıştırırken Windows'ta çoklu işlem için tokenizer ve tokenizer_porter işlevlerinin seçilmesiyle ilgili sorunlar bildirmiştir. Başka bir geçici çözüm de bu iki fonksiyonu, [tokenizer, tokenizer_porter], [str.split] ile değiştirmek olacaktır. Ancak, basit str.split ile değiştirmenin stemming'i desteklemeyeceğini unutmayın.

GridSearchCV nesnesini ve parametre ızgarasını önceki kodu kullanarak başlattığımızda, özellik vektörlerinin sayısı ve büyük kelime dağarcığı ızgara aramasını hesaplama açısından oldukça pahalı hale getirebileceğinden, kendimizi sınırlı sayıda parametre kombinasyonuyla sınırladık. Standart bir masaüstü bilgisayar kullanarak ızgara aramamızın tamamlanması 5-10 dakika sürebilir.

Önceki kod örneğinde, önceki alt bölümdeki CountVectorizer ve TfidfTransformer'ı CountVectorizer ile TfidfTransformer'ı birleştiren TfidfVectorizer ile değiştirdik. param_grid'imiz iki parametre sözlüğünden oluşuyordu. İlk sözlükte, tf-idf'leri hesaplamak için TfidfVectorizer'ı varsayılan ayarlarıyla (use_idf=True, smooth_idf=True ve norm='l2') kullandık; ikinci sözlükte, ham terim frekanslarına dayalı bir modeli eğitmek için bu parametreleri use_idf=False, smooth_idf=False ve norm=None olarak ayarladık. Ayrıca, lojistik regresyon sınıflandırıcısının kendisi için, ceza parametresi aracılığıyla L2 düzenlemesi kullanarak modelleri eğittik ve ters-düzenleme parametresi için bir değer aralığı tanımlayarak farklı düzenleme güçlerini karşılaştırdık C. İsteğe bağlı bir alıştırma olarak, 'clf penalty' seçeneğini değiştirerek parametre tablosuna L1 düzenlemesi eklemeniz de önerilir: ['l2'] 'clf penalty' olarak değiştirin: ['l2', 'l1'].

Izgara araması bittikten sonra, en iyi parametre setini yazdırabiliriz:

```
>> print(f'En iyi parametre seti: {gs_lr_tfidf.best_params_}')
En iyi parametre seti: {'clf C': 10.0, 'clf penalty': 'l2', 'vect ngram_range': (1,
1), 'vect stop_words': None, 'vect tokenizer': <fonksiyon tokenizer at
0x169932dc0>}
```

Önceki çıktıda görebileceğiniz gibi, en iyi ızgara arama sonuçlarını Porter stemming olmadan düzenli tokenizer, stop word kütüphanesi olmadan ve tf-idfs'yi 10.0 düzenli hale getirme gücü C ile L2-düzenli hale getirme kullanan bir lojistik regresyon sınıflandırıcısı ile birlikte kullanarak elde ettik.

Bu ızgara aramasından elde edilen en iyi modeli kullanarak, eğitim veri kümesindeki ortalama 5 katlı çapraz doğrulama doğruluk puanlarını ve test veri kümesindeki sınıflandırma doğruluğunu

```
>> print(f'CV Doğruluğu: {gs_lr_tfidf.best_score_:.3f}') CV
Doğruluğu: 0,897
>>> clf = gs_lr_tfidf.best_estimator_
>> print(f'Test Doğruluğu: {clf.score(X_test, y_test):.3f}')
Test Doğruluğu: 0,899
```

yazdıralım:

Sonuçlar, makine öğrenimi modelimizin bir film eleştirisinin olumlu mu yoksa olumsuz mu olduğunu yüzde 90 doğrulukla tahmin edebildiğini ortaya koyuyor.

**Naïve Bayes sınıflandırıcı**

Metin sınıflandırması için hala çok popüler olan bir sınıflandırıcı, e-posta spam filtreleme uygulamalarında popülerlik kazanan naïve Bayes sınıflandırıcısıdır. Naïve Bayes sınıflandırıcılarının uygulanması kolaydır, hesaplama açısından verimlidir ve diğer algoritmalara kıyasla nispeten küçük veri kümelerinde özellikle iyi performans gösterme eğilimindedir. Bu kitapta naïve Bayes sınıflandırıcılarını tartışmasak da, ilgilenen okuyucu arXiv'de naïve Bayes metin sınıflandırması hakkında ücretsiz olarak erişilebilen bir makale bulabilir (*Naive Bayes and Text Classification I - Intro- duction and Theory* by *S. Raschka*, *Computing Research Repository* (*CoRR*), abs/1410.5329, 2014, http://arxiv.org/pdf/1410.5329v3.pdf). Bu makalede atıfta bulunulan **naïve** Bayes sınıflandırıcılarının farklı versiyonları scikit-learn'de uygulanmaktadır. İlgili kod sınıflarına bağlantılar içeren bir genel bakış sayfasını burada bulabilirsiniz: https://scikit-learn.org/stable/ modules/naive_bayes.html.

# Daha büyük verilerle çalışma - çevrimiçi algoritmalar ve çekirdek dışı öğrenme

Önceki bölümdeki kod örneklerini uyguladıysanız, bir ızgara araması sırasında 50.000 film incelemesi veri kümesi için özellik vektörlerini oluşturmanın hesaplama açısından oldukça pahalı olabileceğini fark etmiş olabilirsiniz. Birçok gerçek dünya uygulamasında, bilgisayarımızın belleğini aşabilecek daha da büyük veri kümeleriyle çalışmak nadir değildir.

 olimpikbarbell@gmail.com 

Herkesin süper bilgisayar olanaklarına erişimi olmadığından, şimdi sınıflandırıcıyı bir veri kümesinin daha küçük gruplarına aşamalı olarak yerleştirerek bu tür büyük veri kümeleriyle çalışmamıza olanak tanıyan **çekirdek dışı öğrenme** adı verilen bir teknik uygulayacağız.

**Tekrarlayan sinir ağları ile metin sınıflandırma**

*Bölüm 15*, Tekrarlayan *Sinir Ağları Kullanarak Sıralı Verileri Modelleme*'de, bu veri kümesini tekrar ziyaret edeceğiz ve IMDb film incelemesi veri kümesindeki incelemeleri sınıflandırmak için derin öğrenme tabanlı bir sınıflandırıcı (tekrarlayan bir sinir ağı) eğiteceğiz. Bu sinir ağı tabanlı sınıflandırıcı, stokastik gradyan iniş optimizasyon algoritmasını kullanarak aynı çekirdek dışı prensibini takip eder, ancak bir kelime torbası modelinin oluşturulmasını gerektirmez.

*Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitilmesi* başlıklı *2. Bölümde,* **sto- chastic gradient descent** kavramı tanıtılmıştı; bu, modelin ağırlıklarını her seferinde bir örnek kullanarak güncelleyen bir optimizasyon algoritmasıdır. Bu bölümde, scikit-learn'deki `SGDClassifier`'ın `partial_fit` işlevini kullanarak belgeleri doğrudan yerel sürücümüzden aktaracağız ve küçük belge gruplarını kullanarak bir lojistik regresyon modeli eğiteceğiz.

İlk olarak, bu bölümün başında oluşturduğumuz `movie_data.csv` dosyasındaki işlenmemiş metin verilerini temizleyen ve durdurma sözcüklerini kaldırırken sözcük belirteçlerine ayıran bir belirteçleyici işlevi tanımlayacağız:

```
>> import numpy as np
>> import re
>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>> def tokenizer(metin):
...     metin = re.sub('<[^>]*>', '', metin)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
...                            metin)
...     metin = re.sub('[\W]+', ' ', metin.lower()) \
...                 + ' '.join(emoticons).replace('-', '')
...     tokenized = [w for w in text.split() if w not in stop]
...     return tokenized
```

Daha sonra, her seferinde bir belge okuyan ve döndüren `stream_docs` adında bir üreteç işlevi tanımlayacağız:

```
>> def stream_docs(path):
...     with open(path, 'r', encoding='utf-8') as csv:
...         next(csv) # başlığı atla
...         csv içindeki satır için:
...             metin, etiket = satır[:-3], int(satır[-
...             2])
...             verim metin, etiket
```

stream_docs işlevimizin doğru çalıştığını doğrulamak için, movie_data.csv dosyasından ilk belgeyi okuyalım; bu, inceleme metninin yanı sıra ilgili sınıf etiketinden oluşan bir tuple döndürmelidir:

```
>> next(stream_docs(path='movie_data.csv'))
('"1974 yılında, genç Martha Moxley ... ',1)
```

Şimdi stream_docs'tan bir belge akışı alacak get_minibatch adlı bir işlev tanımlayacağız
işlevini kullanın ve size parametresiyle belirtilen belirli sayıda belge döndürün:

```
>> def get_minibatch(doc_stream, size):
...     docs, y = [], []
...     dene:
...         for _ in range(size):
...             text, label = next(doc_stream)
...             docs.append(text)
...             y.append(etiket)
...     StopIteration hariç:
...         return Yok, Yok
...     return docs, y
```

Ne yazık ki, CountVectorizer'ı çekirdek dışı öğrenme için kullanamayız çünkü tüm kelime dağarcığını bellekte tutmayı gerektirir. Ayrıca, TfidfVectorizer'ın ters belge frekanslarını hesaplamak için eğitim veri kümesinin tüm özellik vektörlerini bellekte tutması gerekir. Bununla birlikte, scikit-learn'de uygulanan metin işleme için başka bir kullanışlı vektörleştirici HashingVectorizer'dır. HashingVectorizer veriden bağımsızdır ve Austin Appleby'nin 32-bit MurmurHash3 fonksiyonu aracılığıyla hashing hilesini kullanır (MurmurHash hakkında daha fazla bilgiyi https://en.wikipedia.org/wiki/ MurmurHash adresinde bulabilirsiniz):

```
>> from sklearn.feature_extraction.text import HashingVectorizer
>> from sklearn.linear_model import SGDClassifier
>> vect = HashingVectorizer(decode_error='ignore',
...                         n_özellikler=2**21,
...                         önişlemci=Yok,
...                         tokenizer=tokenizer)
>> clf = SGDClassifier(loss='log', random_state=1)
>> doc_stream = stream_docs(path='movie_data.csv')
```

Önceki kodu kullanarak, HashingVectorizer'ı tokenizer işlevimizle başlattık ve özellik sayısını 2**21 olarak ayarladık. Ayrıca, SGDClassifier'ın kayıp parametresini 'log' olarak ayarlayarak bir lojistik regresyon sınıflandırıcısını yeniden başlattık. HashingVectorizer'da çok sayıda özellik seçerek hash çarpışmalarına neden olma olasılığını azalttığımızı, ancak lojistik regresyon modelimizdeki katsayı sayısını da artırdığımızı unutmayın.

 olinpikbarbell@gmail.com 

Şimdi asıl ilginç kısım geliyor - tüm tamamlayıcı fonksiyonları kurduktan sonra, aşağıdaki kodu kullanarak çekirdek dışı öğrenmeyi başlatabiliriz:

```
>> import pyprind
>>> pbar = pyprind.ProgBar(45)
>> sınıflar = np.array([0, 1])
>> for _ in range(45):
...         X_train, y_train = get_minibatch(doc_stream, size=1000)
...         if not X_train:
...             Mola
...         X_train = vect.transform(X_train) clf.partial_fit(X_train,
...         y_train, classes=classes) pbar.update()
...                                 100%
0%
[################################] | ETA: 00:00:00
Geçen toplam süre: 00:00:21
```

Yine, öğrenme algoritmamızın ilerlemesini tahmin etmek için PyPrind paketinden yararlandık. İlerleme çubuğu nesnesini 45 yineleme ile başlattık ve aşağıdaki for döngüsünde, her mini yığının 1.000 belgeden oluştuğu 45 mini belge yığını üzerinde yineleme yaptık. Artımlı öğrenme sürecini tamamladıktan sonra, modelimizin performansını değerlendirmek için son 5.000 belgeyi kullanacağız:

```
>> X_test, y_test = get_minibatch(doc_stream, size=5000)
>> X_test = vect.transform(X_test)
>> print(f'Doğruluk: {clf.score(X_test, y_test):.3f}')
Doğruluk: 0,868
```

**YokTür hatası**

NoneType hatasıyla karşılaşırsanız, X_test, y_test = get_minibatch(...) kodunu iki kez çalıştırmış olabileceğinizi lütfen unutmayın. Önceki döngü aracılığıyla, her biri 1.000 belge getirdiğimiz 45 iterasyonumuz var. Dolayısıyla, test için geriye tam olarak 5.000 belge kalıyor ve bunları şu şekilde atıyoruz:

```
>> X_test, y_test = get_minibatch(doc_stream, size=5000)
```

Bu kodu iki kez çalıştırırsak, üreteçte yeterli belge kalmaz ve X_test None döndürür. Bu nedenle, NoneType hatasıyla karşılaşırsanız, önceki stream_docs(...) kodundan tekrar başlamanız gerekir.

Gördüğünüz gibi, modelin doğruluğu yaklaşık yüzde 87'dir ve önceki bölümde hiperparametre ayarı için ızgara aramasını kullanarak elde ettiğimiz doğruluğun biraz altındadır. Bununla birlikte, çekirdek dışı öğrenme bellek açısından çok verimlidir ve tamamlanması bir dakikadan az sürmüştür.

Son olarak, modelimizi güncellemek için son 5.000 belgeyi kullanabiliriz:

```
>> clf = clf.partial_fit(X_test, y_test)
```

**word2vec modeli**

Kelime torbası modeline daha modern bir alternatif, Google'ın 2013 yılında yayınladığı bir algoritma olan word2vec'tir (*Efficient Estimation of Word Representations in Vector Space, T. Mikolov, K. Chen*, *G. Corrado,* and *J. Dean*, https://arxiv.org/abs/1301.3781).

word2vec algoritması, kelimeler arasındaki ilişkiyi otomatik olarak öğrenmeye çalışan sinir ağlarına dayalı denetimsiz bir öğrenme algoritmasıdır. word2vec'in arkasındaki fikir, benzer anlamlara sahip kelimeleri benzer kümelere yerleştirmektir ve akıllı vektör aralığı sayesinde model, basit vektör matematiği kullanarak belirli kelimeleri yeniden üretebilir, örneğin, *kral - erkek + kadın = kraliçe*.

İlgili makalelere ve alternatif uygulamalara faydalı bağlantılar içeren orijinal C-uygulaması https://code.google.com/p/word2vec/ adresinde bulunabilir.

# Gizli Dirichlet ile konu modelleme tahsisi

**Konu modelleme,** etiketlenmemiş metin belgelerine konu atama gibi geniş bir görevi tanımlar. Örneğin, tipik bir uygulama, gazete makalelerinden oluşan büyük bir metin külliyatındaki belgelerin kategorize edilmesidir. Konu modelleme uygulamalarında, daha sonra bu makalelere, örneğin spor, finans, dünya haberleri, politika ve yerel haberler gibi kategori etiketleri atamayı amaçlarız. Dolayısıyla, *Bilgisayarlara Verilerden Öğrenme Yeteneği Kazandırma* başlıklı *1. Bölümde* ele aldığımız geniş makine öğrenimi kategorileri bağlamında, konu modellemeyi denetimsiz öğrenmenin bir alt kategorisi olan bir kümeleme görevi olarak düşünebiliriz.

Bu bölümde, **latent Dirichlet allocation** (**LDA**) adı verilen popüler bir konu modelleme tekniğini tartışacağız. Ancak, gizli Dirichlet tahsisi genellikle LDA olarak kısaltılsa da, *Bölüm 5, Boyutluluk Azaltma Yoluyla Verileri Sıkıştırma*'da tanıtılan denetimli bir boyutluluk azaltma tekniği olan *doğrusal diskriminant analizi* ile karıştırılmaması gerektiğini unutmayın.

## Metin belgelerini LDA ile ayrıştırma

LDA'nın arkasındaki matematik oldukça karmaşık olduğundan ve Bayesçi çıkarım bilgisi gerektirdiğinden, bu konuya bir uygulayıcı perspektifinden yaklaşacağız ve LDA'yı meslekten olmayanların terimlerini kullanarak yorumlayacağız. Bununla birlikte, ilgilenen okuyucu aşağıdaki araştırma makalesinde LDA hakkında daha fazla bilgi edinebilir: *Latent Dirichlet Allocation*, *David M. Blei, Andrew Y. Ng*, and *Michael I. Jordan, Journal of Machine Learning Research 3* , pages: 993-1022, Ocak 2003, https://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf.

LDA, farklı belgelerde sıkça bir arada görünen kelime gruplarını bulmaya çalışan üretken bir olasılık modelidir. Bu sık görünen kelimeler, her belgenin farklı kelimelerin bir karışımı olduğu varsayılarak konularımızı temsil eder. Bir LDA'nın girdisi, bu bölümde daha önce tartıştığımız kelime torbası modelidir.

 olimpikbarbell@gmail.com 

Girdi olarak bir kelime torbası matrisi verildiğinde, LDA bunu iki yeni matrise ayrıştırır:

- Bir belge-konu matrisi
- Bir kelime-konu matrisi

LDA, kelime torbası matrisini öyle bir şekilde ayrıştırır ki, bu iki matrisi birbiriyle çarparsak, girdi olan kelime torbası matrisini mümkün olan en düşük hatayla yeniden üretebiliriz. Pratikte, LDA'nın kelime torbası matrisinde bulduğu konularla ilgileniyoruz. Tek dezavantajı, konu sayısını önceden tanımlamamız gerekmesi olabilir - konu sayısı, LDA'nın manuel olarak belirtilmesi gereken bir hiper parametresidir.

## Scikit-learn ile LDA

Bu alt bölümde, film incelemesi veri kümesini ayrıştırmak ve farklı konulara kategorize etmek için scikit-learn'de uygulanan LatentDirichletAllocation sınıfını kullanacağız. Aşağıdaki örnekte, analizi 10 farklı konuyla sınırlayacağız, ancak okuyucuların bu veri kümesinde bulunabilecek konuları daha fazla keşfetmek için algoritmanın hiperparametrelerini denemeleri teşvik edilmektedir.

İlk olarak, bu bölümün başında oluşturduğumuz film incelemelerinin yerel movie_data.csv dosyasını kullanarak veri kümesini bir pandas DataFrame'e yükleyeceğiz:

```
>> import pandas as pd
>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
>> # bazı bilgisayarlarda aşağıdakiler gereklidir:
>> df = df.rename(columns={"0": " review", " 1": " sentiment"})
```

Daha sonra, LDA'ya girdi olarak kelime torbası matrisini oluşturmak için zaten tanıdık olan CountVectorizer'ı kullanacağız.

Kolaylık sağlamak için, scikit-learn'in yerleşik İngilizce durdurma sözcüğü kitaplığını stop_words='english' aracılığıyla kullanacağız:

```
>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer(stop_words='english',
...                         max_df=.1,
...                         max_features=5000)
>>> X = count.fit_transform(df['review'].values)
```

Belgeler arasında çok sık geçen kelimeleri hariç tutmak için dikkate alınacak kelimelerin maksimum belge sıklığını yüzde 10 (max_ df=.1) olarak belirlediğimize dikkat edin. Sıklıkla ortaya çıkan kelimelerin çıkarılmasının ardındaki mantık, bunların tüm belgelerde görülen ortak kelimeler olabileceği ve bu nedenle belirli bir belgenin belirli bir konu kategorisiyle ilişkilendirilme olasılıklarının daha düşük olabileceğidir. Ayrıca, LDA tarafından gerçekleştirilen çıkarımı iyileştirmek amacıyla bu veri kümesinin boyutluluğunu sınırlandırmak için dikkate alınacak kelime sayısını en sık meydana gelen 5.000 kelimeyle (max_features=5000) sınırlandırdık. Ancak, hem max_df=.1 hem de max_features=5000 hiperparametre değerleri isteğe bağlı olarak seçilmiştir ve okuyucuların sonuçları karşılaştırırken bunları ayarlamaları önerilir.

Aşağıdaki kod örneği, bir LatentDirichletAllocation tahmincisinin kelime torbası matrisine nasıl uydurulacağını ve belgelerden 10 farklı konunun nasıl çıkarılacağını göstermektedir (model uydurmanın bir dizüstü bilgisayarda veya standart masaüstü bilgisayarda 5 dakika veya daha

```
>> from sklearn.decomposition import LatentDirichletAllocation
>>  lda = LatentDirichletAllocation(n_components=10,
...                                 random_state=123,
...                                 learning_method='batch')
>> X_topics = lda.fit_transform(X)
```

fazla sürebileceğini unutmayın):

learning_method='batch' ayarını yaparak, lda tahmin edicisinin mevcut tüm eğitim verilerine (kelime torbası matrisi) dayalı tahminini tek bir yinelemede yapmasına izin veririz; bu, alternatif 'çevrimiçi' öğrenme yönteminden daha yavaştır, ancak daha doğru sonuçlara yol açabilir (learning_method='online' ayarını yapmak, *Bölüm 2*, *Sınıflandırma için Basit Makine Öğrenimi Algoritmalarını Eğitme* ve daha önce bu bölümde tartıştığımız çevrimiçi veya mini toplu öğrenmeye benzer).

**Beklenti-maksimizasyonu**

Scikit-learn kütüphanesinin LDA uygulaması, parametre tahminlerini yinelemeli olarak güncellemek için **beklenti-maksimizasyon** (**EM**) algoritmasını kullanır. Bu bölümde EM algoritmasını tartışmadık, ancak daha fazla bilgi edinmek istiyorsanız, lütfen Wikipedia'daki mükemmel genel bakışa (https://en.wikipedia.org/wiki/Expectation-maximization_ algoritması) ve Colorado Reed'in *Latent Dirichlet Allocation* adlı eğitiminde LDA'da nasıl kullanıldığına dair ayrıntılı eğiticiye bakın: *Towards a Deeper Understanding*, http://obphio.us/pdfs/lda_tutorial.pdf adresinden ücretsiz olarak edinilebilir.

LDA'yı yerleştirdikten sonra, artık 10 konunun her biri için artan sırada kelime önemini (burada 5000) içeren bir matris saklayan lda örneğinin components_ niteliğine erişebiliriz:

```
>> lda.components_.shape (10,
5000)
```

Sonuçları analiz etmek için, 10 konunun her biri için en önemli beş kelimeyi yazdıralım. Kelime önem değerlerinin artan sırada sıralandığına dikkat edin. Bu nedenle, ilk beş kelimeyi yazdırmak için konu dizisini ters sırada sıralamamız gerekir:

```
>> n_top_words = 5
>> özellik_adları = count.get_feature_names_out()
>> for topic_idx, topic in enumerate(lda.components_):
...     print(f'Konu {(topic_idx + 1)}:')
...     print(' '.join([özellik_adları[i]
...                     for i in topic.argsort()\
...                     [:-n_top_words - 1:-
1]])) Konu 1:
```

 olimpikbarbell@gmail.com 

```
en kötü dakikalar berbat senaryo
aptal Konu 2:
aile anne baba çocuklar kız Konu 3:
amerikan savaşı dvd müzik tv
Konu 4:
insan izleyici sinema sanat anlayışı
Konu 5:
polis adam araba ölü cinayet
Konu 6:
korku evi seks kız kadın Konu
7:
rol performans komedi aktör performansları
Konu 8:
dizi bölüm savaş bölümleri tv Konu
9:
kitap versiyonu orijinal okuma
roman Konu 10:
aksiyon dövüşü adam çocuklar havalı
```

Her bir konu için en önemli beş kelimeyi okuyarak, LDA'nın aşağıdaki konuları belirlediğini tahmin edebilirsiniz:

1. Genel olarak kötü filmler (aslında bir konu kategorisi değil)
2. Aileler hakkında filmler
3. Savaş filmleri
4. Sanat filmleri
5. Suç filmleri
6. Korku Filmleri
7. Komedi filmi incelemeleri
8. Bir şekilde TV dizileriyle ilgili filmler
9. Kitaplardan uyarlanan filmler
10. Aksiyon filmleri

Kategorilerin incelemelere göre anlamlı olduğunu teyit etmek için korku filmi kategorisinden üç film seçelim (korku filmleri 5. indeks konumunda 6. kategoriye aittir):

```
>>> korku = X_topics[:, 5].argsort()[::-1]
>> for iter_idx, movie_idx in enumerate(horror[:3]):
...         print(f'\nKorku filmi #{(iter_idx + 1)}:')
...         print(df['review'][movie_idx][:300], '...')
Korku filmi #1:
House of Dracula, bir yıl önceki House of Frankenstein ile aynı temel önermeden
hareket ediyor; yani Universal'ın en ünlü üç c a n a v a r ı ; Dracula, Frankenstein'ın
Canavarı ve Kurt Adam filmde birlikte yer alıyor. Doğal olarak, film bu nedenle oldukça
dağınıktır, ancak gerçek şu ki ...
```

```
Korku filmi 2:
Tamam, şimdi ne tür bir ÇÖP izliyorum ben? "The Witches' Mountain" gelmiş
geçmiş en tutarsız ve çılgın İspanyol istismar filmlerinden biri olmalı ama aynı
zamanda garip bir şekilde ilgi çekici. Burada kesinlikle mantıklı hiçbir şey yok ve
hatta olduğundan bile şüpheliyim ...
Korku filmi 3:
<br /><br />Korku filmi zamanı, Japon tarzı. Uzumaki/Spiral baştan sona
tam bir ucubeydi. Eğlenceli bir ucubeydi, ancak zaman zaman korkudan
ziyade kitsch'e biraz fazla güveniyordu. Hikayeyi kısa ve öz bir şekilde
özetlemek zor: kaygısız, normal bir genç kız ...
```

Önceki kod örneğini kullanarak, en iyi üç korku filminin ilk 300 karakterini yazdırdık. Tam olarak hangi filme ait olduklarını bilmesek de incelemeler korku filmi incelemelerine benziyor (ancak, 2 numaralı Korku filminin konu kategorisi 1: *Genel olarak kötü filmler* için de uygun olabileceği iddia edilebilir).

# Özet

Bu bölümde, NLP alanında duygu analizinde temel bir görev olan metin belgelerini kutuplarına göre sınıflandırmak için makine öğrenimi algoritmalarının nasıl kullanılacağını öğrendiniz. Kelime torbası modelini kullanarak bir belgeyi bir özellik vektörü olarak nasıl kodlayacağınızı öğrenmekle kalmadınız, aynı zamanda tf-idf kullanarak terim frekansını alaka düzeyine göre nasıl ağırlıklandıracağınızı da öğrendiniz.

Metin verileriyle çalışmak, bu işlem sırasında oluşturulan büyük özellik vektörleri nedeniyle hesaplama açısından oldukça pahalı olabilir; son bölümde, tüm veri kümesini bilgisayarın belleğine yüklemeden bir makine öğrenimi algoritmasını eğitmek için çekirdek dışı veya artımlı öğrenmenin nasıl kullanılacağını ele aldık.

Son olarak, film incelemelerini denetimsiz bir şekilde farklı kategorilere a y ı r m a k için LDA kullanarak konu modelleme kavramıyla tanıştınız.

Bu kitapta şimdiye kadar birçok makine öğrenimi kavramını, en iyi uygulamaları ve sınıflandırma için denetimli modelleri ele aldık. Bir sonraki bölümde, şimdiye kadar çalıştığımız sınıflandırma modellerinin kategorik sınıf etiketlerinin aksine, sonuç değişkenlerini sürekli bir ölçekte tahmin etmemizi sağlayan bir başka denetimli öğrenme alt kategorisi olan *regresyon analizine* bakacağız.

# Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

olimpikbarbell@gmail.com

# 9

# Regresyon Analizi ile Sürekli Hedef Değişkenlerin Tahmin Edilmesi

Önceki bölümler boyunca, **denetimli öğrenmenin** arkasındaki ana kavramlar hakkında çok şey öğrendiniz ve grup üyeliklerini veya kategorik değişkenleri tahmin etmek için sınıflandırma görevleri için birçok farklı modeli eğittiniz. Bu bölümde, denetimli öğrenmenin bir başka alt kategorisine dalacağız: **regresyon analizi**.

Regresyon modelleri, sürekli bir ölçekte hedef değişkenleri tahmin etmek için kullanılır, bu da onları bilimdeki birçok soruyu ele almak için çekici kılar. Ayrıca değişkenler arasındaki ilişkileri anlamak, eğilimleri değerlendirmek veya tahminler yapmak gibi endüstride de uygulamaları vardır. Bir şirketin gelecek aylardaki satışlarının tahmin edilmesi buna bir örnektir.

Bu bölümde, regresyon modellerinin temel kavramlarını tartışacağız ve aşağıdaki konuları ele alacağız:

- Veri kümelerini keşfetme ve görselleştirme
- Doğrusal regresyon modellerinin uygulanmasına yönelik farklı yaklaşımların incelenmesi
- Aykırı değerlere karşı dayanıklı regresyon modellerinin eğitimi
- Regresyon modellerini değerlendirme ve yaygın sorunları teşhis etme
- Doğrusal olmayan verilere regresyon modellerinin uydurulması

## Doğrusal regresyonunun tanıtılması

Doğrusal regresyonun amacı, bir veya birden fazla özellik ile sürekli bir hedef değişken arasındaki ilişkiyi modellemektir. Denetimli öğrenmenin farklı bir alt kategorisi olan sınıflandırmanın aksine, regresyon analizi kategorik sınıf etiketleri yerine sürekli bir ölçekte çıktıları tahmin etmeyi amaçlar.

Aşağıdaki alt bölümlerde, en temel doğrusal regresyon türü olan **basit doğrusal regresyon** ile tanışacak ve bunun daha genel, çok değişkenli durumla (çoklu özellikli doğrusal regresyon) nasıl ilişkilendirileceğini anlayacaksınız.

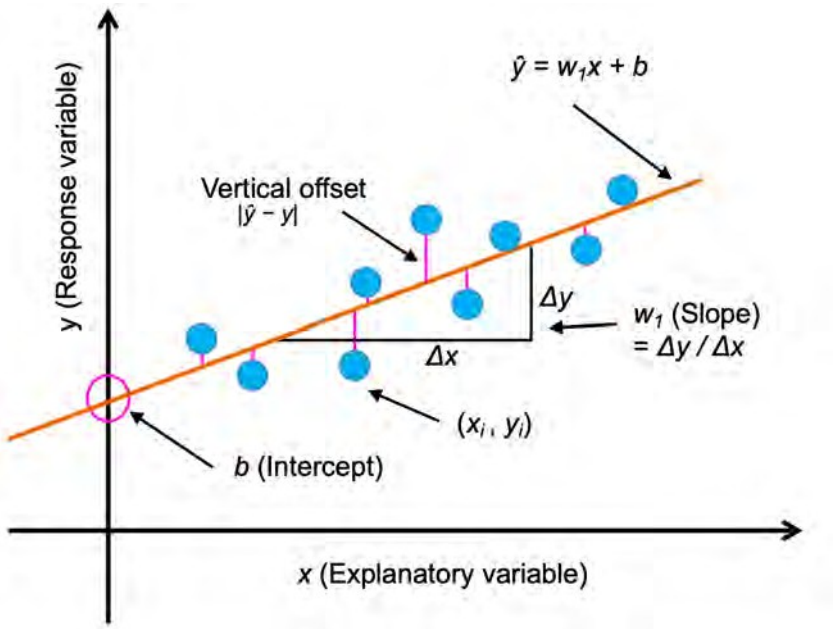
# Basit doğrusal regresyon

Basit (tek **değişkenli**) doğrusal regresyonun amacı, tek bir özellik (**açıklayıcı değişken**, $x$) ile sürekli değerli bir **hedef** (**yanıt değişkeni,** $y$) arasındaki ilişkiyi modellemektir. Tek açıklayıcı değişkenli bir doğrusal modelin denklemi aşağıdaki gibi tanımlanır:

$$y\ y\ yy_1\ x + b$$

Burada, parametre (yanlılık birimi), $b$, $y$ ekseni kesişimini ve $w_1$ açıklayıcı değişkenin ağırlık katsayısını temsil etmektedir. Amacımız, açıklayıcı değişken ile hedef değişken arasındaki ilişkiyi tanımlamak için doğrusal denklemin ağırlıklarını öğrenmek ve daha sonra eğitim veri setinin bir parçası olmayan yeni açıklayıcı değişkenlerin yanıtlarını tahmin etmek için kullanılabilmektir.

Daha önce tanımladığımız doğrusal denkleme dayanarak, doğrusal regresyon *Şekil 9.1*'de gösterildiği gibi eğitim örnekleri boyunca en iyi uyan düz çizgiyi bulmak olarak anlaşılabilir:




*Şekil 9.1: Basit bir tek özellikli doğrusal regresyon örneği*

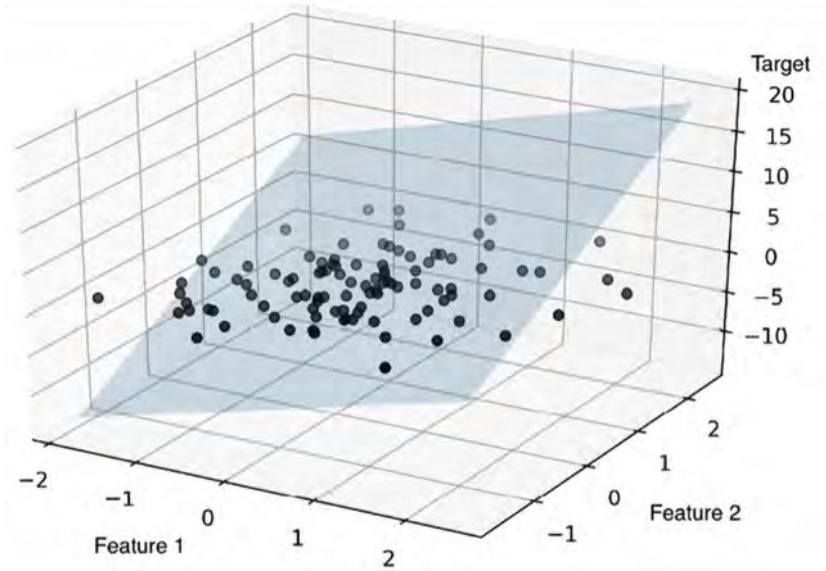
Bu en iyi uyum çizgisine **regresyon çizgisi** de denir ve regresyon çizgisinden eğitim örneklerine kadar olan dikey çizgiler **ofsetler** veya artıklar olarak adlandırılır - tahminimizin hataları.

# Çoklu doğrusal regresyon

Bir önceki bölümde, tek açıklayıcı değişkenli doğrusal regresyonun özel bir durumu olan basit doğrusal regresyon tanıtılmıştır. Elbette, doğrusal regresyon modelini çoklu açıklayıcı değişkenlere de genelleştirebiliriz; bu sürece **çoklu doğrusal regresyon** denir:

$$y_{yy1x1} + \ldots + {}_{yymxm} + b\ y\ \boldsymbol{b}_{\substack{m \\ \ddot{u}1}}{}_{yyi\ xi} + b\ y\ yy^T \boldsymbol{x}\boldsymbol{x} + b$$

*Şekil 9.2,* iki özellikli bir çoklu doğrusal regresyon modelinin iki boyutlu, uygun hiper düzleminin nasıl görünebileceğini göstermektedir:




*Şekil 9.2: İki özellikli doğrusal regresyon modeli*

Gördüğünüz gibi, çoklu doğrusal regresyon hiper düzlemlerinin üç boyutlu bir dağılım grafiğinde görselleştirilmesi, statik şekillere bakıldığında yorumlanması zaten zordur. Bir dağılım grafiğinde iki boyutlu hiper düzlemleri (üç veya daha fazla özelliğe sahip veri kümelerine uyan çoklu doğrusal regresyon modelleri) görselleştirmenin iyi bir yolu olmadığından, bu bölümdeki örnekler ve görselleştirmeler esas olarak basit doğrusal regresyon kullanılarak tek değişkenli duruma odaklanacaktır. Bununla birlikte, basit ve çoklu doğrusal regresyon aynı kavramlara ve aynı değerlendirme tekniklerine dayanmaktadır; bu bölümde tartışacağımız kod uygulamaları da her iki regresyon modeli türüyle uyumludur.

# Ames Housing veri setinin incelenmesi

İlk doğrusal regresyon modelini uygulamadan önce, 2006'dan 2010'a kadar Ames, Iowa'daki bireysel konut mülkleri hakkında bilgi içeren yeni bir veri kümesi olan Ames Konut veri kümesini tartışacağız. Veri seti 2011 yılında Dean De Cock tarafından toplanmıştır ve ek bilgilere aşağıdaki bağlantılardan ulaşılabilir:

- Veri setini açıklayan bir rapor: http://jse.amstat.org/v19n3/decock.pdf
- Veri kümesinin özelliklerine ilişkin ayrıntılı belgeler: http://jse.amstat.org/v19n3/decock/DataDocumentation.txt
- Sekmeyle ayrılmış biçimde veri kümesi: http://jse.amstat.org/v19n3/decock/AmesHousing. txt

Her yeni veri setinde olduğu gibi, neyle çalıştığımızı daha iyi hissetmek için verileri basit bir görselleştirme yoluyla keşfetmek her zaman yararlıdır, aşağıdaki alt bölümlerde yapacağımız da budur.

## Ames Konut veri kümesini bir DataFrame'e yükleme

Bu bölümde, hızlı ve çok yönlü olan ve düz metin biçiminde depolanan tablo verileriyle çalışmak için önerilen bir araç olan pandas read_csv işlevini kullanarak Ames Housing veri kümesini yükleyeceğiz.

Ames Housing veri kümesi 2.930 örnek ve 80 özellikten oluşmaktadır. Basitlik açısından, aşağıdaki listede gösterilen özelliklerin yalnızca bir alt kümesiyle çalışacağız. Ancak, merak ediyorsanız, bu bölümün başında verilen tam veri seti açıklamasına giden bağlantıyı takip edin ve bu bölümü okuduktan sonra bu veri setindeki diğer değişkenleri keşfetmeniz önerilir.

Hedef değişken de dahil olmak üzere birlikte çalışacağımız özellikler aşağıdaki gibidir:

- Genel Kalite: Evin genel malzemesi ve kaplaması için 1'den (çok zayıf) 10'a (mükemmel) kadar bir ölçekte derecelendirme
- Genel Durum: Evin genel durumu için 1'den (çok kötü) 10'a (mükemmel) kadar bir ölçekte derecelendirme
- Gr Yaşam Alanı: Fit kare cinsinden sınıf üstü (zemin) yaşam alanı
- Merkezi Hava: Merkezi klima (N=hayır, Y=evet)
- Toplam Bsmt SF: Bodrum alanının toplam fit karesi
- SalePrice: ABD doları cinsinden satış fiyatı ($)

Bu bölümün geri kalanında, satış fiyatını (SalePrice) hedef değişkenimiz olarak kabul edeceğiz - beş açıklayıcı değişkenden birini veya daha fazlasını kullanarak tahmin etmek istediğimiz değişken. Bu veri kümesini daha fazla incelemeden önce, bir pandas DataFrame'ine yükleyelim:

```
import pandas as pd

columns = ['Overall Qual', 'Overall Cond', 'Gr Liv Area',
           'Central Air', 'Total Bsmt SF', ' SalePrice']
```

 olimpikbarbell@gmail.com 

```
df = pd.read_csv('http://jse.amstat.org/v19n3/decock/AmesHousing.txt',
                 sep='\t',
                 usecols=columns)

df.head()
```

Veri kümesinin başarıyla yüklendiğini doğrulamak için, *Şekil 9.3'te* gösterildiği gibi veri kümesinin ilk beş satırını görüntüleyebiliriz:

| | Overall Qual | Overall Cond | Total Bsmt SF | Central Air | Gr Liv Area | SalePrice |
|---|---|---|---|---|---|---|
| **0** | 6 | 5 | 1080.0 | Y | 1656 | 215000 |
| **1** | 5 | 6 | 882.0 | Y | 896 | 105000 |
| **2** | 6 | 6 | 1329.0 | Y | 1329 | 172000 |
| **3** | 7 | 5 | 2110.0 | Y | 2110 | 244000 |
| **4** | 5 | 5 | 928.0 | Y | 1629 | 189900 |

*Şekil 9.3: Konut veri setinin ilk beş satırı*

Veri kümesini yükledikten sonra, beklenen sayıda satır içerdiğinden emin olmak için DataFrame'in boyutlarını da kontrol edelim:

```
>> df.shape
(2930, 6)
```

Gördüğümüz gibi, DataFrame beklendiği gibi 2.930 satır içeriyor.

Dikkat etmemiz gereken bir diğer husus da *Şekil 9.3'te* görebileceğimiz gibi string tipinde kodlanmış olan 'Central Air' değişkenidir. *Bölüm 4, İyi Eğitim Veri Kümeleri Oluşturma - Veri Ön Hazırlığı'nda* öğrendiğimiz gibi, DataFrame sütunlarını dönüştürmek için .map yöntemini kullanabiliriz. Aşağıdaki kod 'Y' dizesini 1 tam sayısına ve 'N' dizesini 0 tam sayısına

```
>> df['Merkezi Hava'] = df['Merkezi Hava'].map({'N': 0, 'Y': 1})
```

dönüştürecektir:

Son olarak, veri çerçevesi sütunlarından herhangi birinin eksik değer içerip içermediğini kontrol edelim:

```
>>> df.isnull().sum()

Genel Nitelik      0
Genel Durum        0
Toplam Bsmt SF     1
Merkezi Hava       0
Gr Yaşam Alanı     0
SatışFiyatı        0
dtype: int64
```

Gördüğümüz gibi, Total Bsmt SF özellik değişkeni bir eksik değer içermektedir. Nispeten büyük bir veri setimiz olduğundan, bu eksik özellik değeriyle başa çıkmanın en kolay yolu, ilgili örneği veri setinden çıkarmaktır (alternatif yöntemler için lütfen *Bölüm 4*'e bakın):

```
>> df = df.dropna(axis=0)
>>> df.isnull().sum()

Genel Nitelik      0
Genel Durum        0
Toplam Bsmt SF     0
Merkezi Hava       0
Gr Yaşam Alanı     0
SatışFiyatı        0
dtype: int64
```

## Bir veri kümesinin önemli özelliklerini görselleştirme

**Keşifsel veri analizi** (**EDA**), bir makine öğrenimi modelinin eğitiminden önce önemli ve önerilen bir ilk adımdır. Bu bölümün geri kalanında, aykırı değerlerin varlığını, verilerin dağılımını ve özellikler arasındaki ilişkileri görsel olarak tespit etmemize yardımcı olabilecek grafiksel EDA araç kutusundan bazı basit ama kullanışlı teknikler kullanacağız.

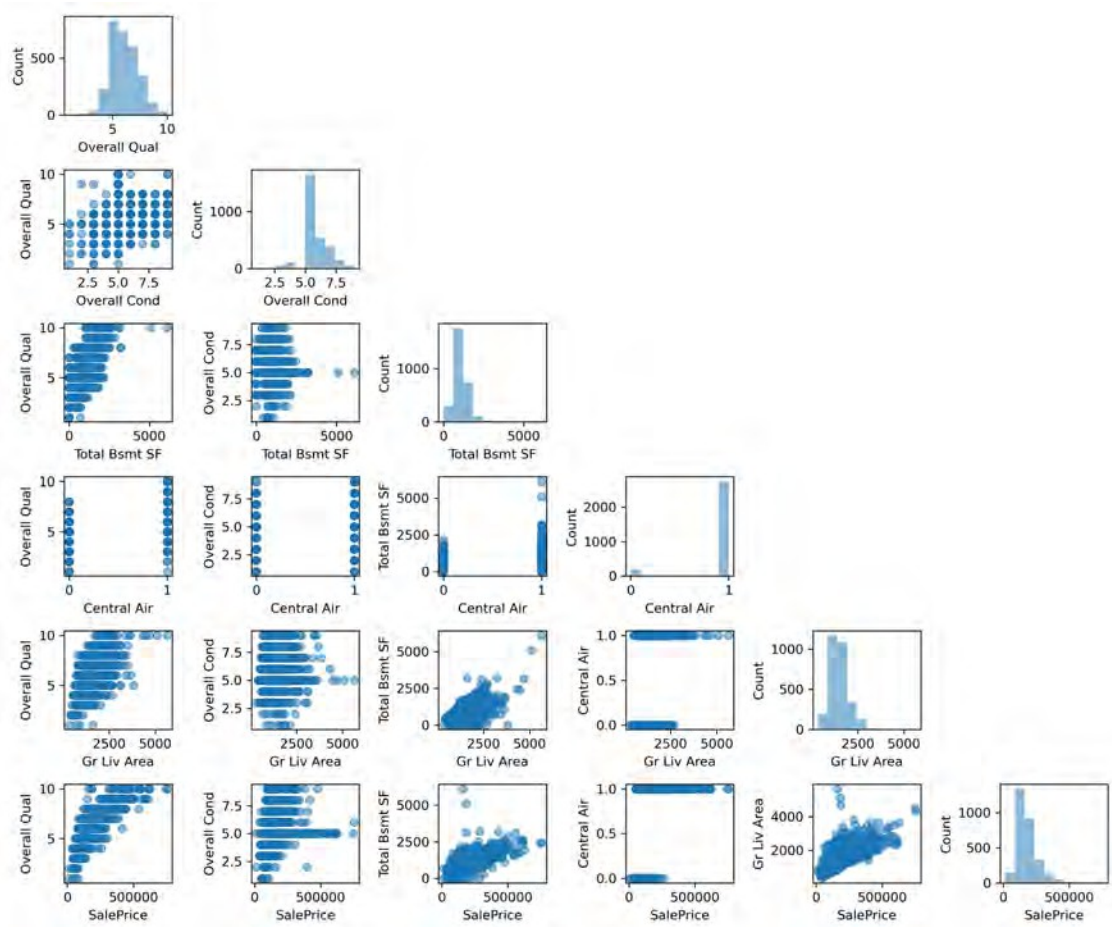
İlk olarak, bu veri kümesindeki farklı özellikler arasındaki ikili korelasyonları tek bir yerde görselleştirmemizi sağlayan bir scatterplot **matrisi** oluşturacağız. Scatterplot matrisini çizmek için, Python'da makine öğrenimi ve veri bilimi uygulamaları için çeşitli kolaylık fonksiyonları içeren bir Python kütüphanesi olan mlxtend kütüphanesinden (http://rasbt.github.io/mlxtend/) scatterplotmatrix fonksiyonunu kullanacağız.

mlxtend paketini conda install mlxtend veya pip install mlxtend aracılığıyla yükleyebilirsiniz. Bu bölüm için mlxtend 0.19.0 sürümünü kullandık.

Kurulum tamamlandıktan sonra, paketi içe aktarabilir ve dağılım matrisini aşağıdaki gibi oluşturabilirsiniz:

```
>> import matplotlib.pyplot as plt
>> from mlxtend.plotting import scatterplotmatrix
>>> scatterplotmatrix(df.values, figsize=(12, 10),
...                   names=df.columns, alpha=0.5)
>> plt.tight_layout()
plt.show()
```

*Şekil 9.4'te* görebileceğiniz gibi, dağılım matrisi bize bir veri kümesindeki ilişkilerin faydalı bir grafik özetini sunar:




*Şekil 9.4: Verilerimizin dağılım matrisi*

Bu dağılım matrisini kullanarak artık verilerin nasıl dağıldığını ve aykırı değerler içerip içermediğini hızlı bir şekilde görebiliriz. Örneğin, (alt satırın solundan beşinci sütun) yer üstündeki yaşam alanının büyüklüğü (Gr Liv Area) ile satış fiyatı (SalePrice) arasında bir miktar doğrusal bir ilişki olduğunu görebiliriz.

Ayrıca, histogramda (dağılım grafiği matrisinde sağ alttaki alt grafik) şunu görebiliriz SalePrice değişkeni birkaç aykırı değer tarafından çarpıtılmış gibi görünüyor.

**Doğrusal regresyonun normallik varsayımı**

Genel kanının aksine, doğrusal bir regresyon modelinin eğitilmesinin açıklayıcı veya hedef değişkenlerin normal dağılımlı olmasını gerektirmediğini unutmayın. Normallik varsayımı yalnızca bu kitabın kapsamı dışında kalan belirli istatistikler ve hipotez testleri için bir gerekliliktir (bu konuda daha fazla bilgi için lütfen *Douglas C. Montgomery*, *Elizabeth A. Peck* ve *G. Geoffrey Vining* tarafından yazılan *Introduction to Linear Regression Analysis*, *Wiley*, s: 318-319, 2012).

## Korelasyon matrisi kullanarak ilişkilere bakmak

Bir önceki bölümde, Ames Konut veri seti değişkenlerinin veri dağılımlarını histogramlar ve dağılım grafikleri şeklinde görselleştirdik. Daha sonra, değişkenler arasındaki doğrusal ilişkileri ölçmek ve özetlemek için bir korelasyon matrisi oluşturacağız. Korelasyon matrisi, *Bölüm 5, Boyutluluk Azaltma Yoluyla Verilerin Sıkıştırılması*'nda *temel bileşen analizi yoluyla denetimsiz boyutluluk azaltma* bölümünde ele aldığımız kovaryans matrisi ile yakından ilişkilidir. Korelasyon matrisini kovaryans matrisinin yeniden ölçeklendirilmiş bir versiyonu olarak yorumlayabiliriz. Aslında, korelasyon matrisi standartlaştırılmış özelliklerden hesaplanan bir kovaryans matrisiyle aynıdır.

Korelasyon matrisi, özellik çiftleri arasındaki doğrusal bağımlılığı ölçen **Pearson momentler çarpımı korelasyon katsayısını** (genellikle **Pearson's r** olarak kısaltılır) içeren bir kare matristir. Korelasyon katsayıları -1 ila 1 aralığındadır. r = 1 ise iki özellik arasında mükemmel bir pozitif korelasyon vardır, r = 0 ise korelasyon yoktur ve $r$ = -1 ise mükemmel bir negatif korelasyon vardır. Daha önce de belirtildiği gibi, Pearson korelasyon katsayısı basitçe iki özellik, $x$ ve $y$ (pay) arasındaki kovaryansın standart sapmalarının çarpımına (payda) bölünmesiyle hesaplanabilir:

$$r \; r \frac{\sum_{ii1}^{n} [(x^{(ii} - \mu xi(y^{(ii} - \mu y)]}{\sqrt{\sum_{ii1}^{n} (x^{(ii} - \mu i^2 \sqrt{\sum_{ii1}^{n} (y^{(ii} - \mu)^2}} \; r \frac{\sigma xy}{\sigma x \sigma y}$$

Burada, $\mu$ ilgili özelliğin ortalamasını, $\sigma_{xy}$ ise $x$ özellikleri arasındaki kovaryansı göstermektedir. ve $y$, ve $\sigma_x$ ve $\sigma_y$ özelliklerin standart sapmalarıdır.

 olimpikbarbell@gmail.com 

**Standartlaştırılmış özellikler için korelasyona karşı kovaryans**

Standartlaştırılmış bir çift özellik arasındaki kovaryansın aslında doğrusal korelasyon katsayısına eşit olduğunu gösterebiliriz. Bunu göstermek için önce $x$ ve $y$ özelliklerini standartlaştırarak sırasıyla $x'$ ve $y'$ *olarak göstereceğimiz* z-skorlarını elde edelim:

$$x' = \frac{x - \mu_x}{\sigma_x}, \quad y' = \frac{y - \mu_y}{\sigma_y}$$

İki özellik arasındaki (popülasyon) kovaryansı aşağıdaki gibi hesapladığımızı unutmayın:

$$\sigma_{xy} = \frac{1}{n}\sum_{i}^{n}(x^{(i)} - \mu_x)(y^{(i)} - \mu_y)$$

Standardizasyon bir özellik değişkenini sıfır ortalamada merkezlediğinden, artık ölçeklendirilmiş özellikler arasındaki kovaryansı aşağıdaki gibi hesaplayabiliriz:

$$\sigma_{x'y'} = \frac{1}{n}\sum_{i}^{n}(x'^{(i)} - 0)(y'^{(i)} - 0)$$

Yeniden yerine koyma yoluyla aşağıdaki sonucu elde ederiz:

$$\sigma_{x'y'} = \frac{1}{n}\sum_{i}^{n}\left(\frac{x - \mu_x}{\sigma_x}\right)\left(\frac{y - \mu_y}{\sigma_y}\right)$$

$$\sigma_{x'y'} = \frac{1}{n\,\sigma_x\,\sigma_y}\sum_{i}^{n}(x^{(i)} - \mu_x)(y^{(i)} - \mu_y)$$

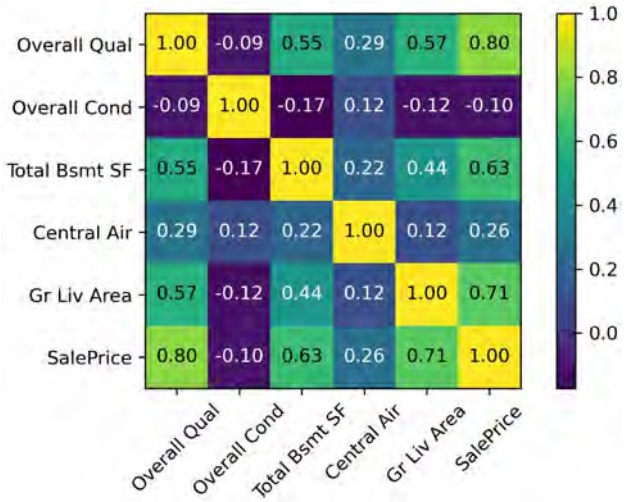
Son olarak, bu denklemi aşağıdaki şekilde basitleştirebiliriz:

$$\sigma_{x'y'} = \frac{\sigma_{xy}}{\sigma_x\,\sigma_y}$$

Aşağıdaki kod örneğinde, daha önce scatterplot matrisinde görselleştirdiğimiz beş özellik sütununda NumPy'nin corrcoef işlevini kullanacağız ve korelasyon matrisi dizisini bir ısı haritası olarak çizmek için mlxtend'in heatmap işlevini kullanacağız:

```
>> import numpy as np
>> from mlxtend.plotting import heatmap


>>  cm = np.corrcoef(df.values.T)
>>> hm = heatmap(cm, row_names=df.columns, column_names=df.columns)
>> plt.tight_layout()
>> plt.show()
```

*Şekil 9.5'te* görebileceğiniz gibi, korelasyon matrisi bize, ilgili doğrusal korelasyonlarına göre özellikleri seçmemize yardımcı olabilecek başka bir yararlı özet grafik sağlar:




*Şekil 9.5: Seçilen değişkenlerin korelasyon matrisi*

Doğrusal bir regresyon modeli kurmak için, hedef değişkenimiz olan `SalePrice` ile yüksek korelasyona sahip olan özelliklerle ilgileniyoruz. Önceki korelasyon matrisine baktığımızda, `SalePrice`'ın `Gr Liv Area` değişkeniyle (0.71) en büyük korelasyonu gösterdiğini görebiliriz; bu da aşağıdaki bölümde basit bir doğrusal regresyon modeli kavramlarını tanıtmak için keşif değişkeni olarak iyi bir seçim gibi görünmektedir.

# Sıradan en küçük kareler doğrusal regresyon modelinin uygulanması

Bu bölümün başında, doğrusal regresyonun eğitim verilerimizin örnekleri üzerinden en iyi uyan düz çizgiyi elde etmek olarak anlaşılabileceğinden bahsetmiştik. Ancak, ne en *iyi uydurma* terimini tanımladık ne de böyle bir modeli uydurmanın farklı tekniklerini tartıştık. Aşağıdaki alt bölümlerde, eğitim örneklerine olan karesel dikey mesafelerin (artıklar veya hatalar) toplamını en aza indiren doğrusal regresyon çizgisinin parametrelerini tahmin etmek için **sıradan en küçük kareler (OLS)** yöntemini (bazen doğrusal **en küçük kareler** olarak da adlandırılır) kullanarak bu bulmacanın eksik parçalarını dolduracağız.

## Gradyan inişi ile regresyon parametreleri için regresyon çözme

*Bölüm 2, Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi'ndeki* **Uyarlanabilir Doğrusal Nöron** (**Adaline**) uygulamamızı düşünün. Yapay nöronun doğrusal bir aktivasyon fonksiyonu kullandığını hatırlayacaksınız. Ayrıca, **gradyan inişi** (**GD**) ve **stokastik gradyan inişi** (**SGD**) gibi optimizasyon algoritmaları aracılığıyla ağırlıkları öğrenmek için minimize ettiğimiz $L(\boldsymbol{w})$ adlı bir kayıp fonksiyonu tanımlamıştık.

Adaline'deki bu kayıp fonksiyonu, OLS için kullandığımız kayıp fonksiyonu ile aynı olan **ortalama karesel hatadır** (**MSE**):

$$L(\boldsymbol{w}, b) = \frac{1}{2n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Burada, $\hat{y}$ tahmin edilen değerdir $\hat{y} = \boldsymbol{w}^T\boldsymbol{x} + b$ ($\frac{1}{2}$ teriminin sadece türetmede kolaylık sağlamak için kullanıldığına dikkat edin GD'nin güncelleme kuralı). Esasen, OLS regresyonu eşiksiz Adaline olarak anlaşılabilir

Böylece 0 ve 1 sınıf etiketleri yerine sürekli hedef değerler elde ederiz. Bunu göstermek için *Bölüm 2*'deki Adaline GD uygulamasını alalım ve ilk doğrusal regresyon modelimizi uygulamak için eşik fonksiyonunu kaldıralım:

```
sınıf DoğrusalRegresyonGD:
    def init (self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.array([0.])
        self.losses_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_ += s e l f . eta * 2.0 * X.T.dot(errors) / X . shape[0]
            self.b_ += s e l f . eta * 2.0 * errors.mean()
            kayıp = (hatalar**2).ortalama()
            self.kayıp_.append(kayıp)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_) + self.b_

    def predict(self, X):
        return self.net_input(X)
```

**Gradyan inişi ile ağırlık güncellemeleri**

Ağırlıkların nasıl güncellendiğine dair bir tazelemeye ihtiyacınız varsa - gradyanın ters yönünde bir adım atarak - lütfen *Bölüm 2*'deki *Uyarlanabilir doğrusal nöronlar ve öğrenmenin yakınsaması bölümünü* tekrar ziyaret edin.

LinearRegressionGD regresörümüzü çalışırken görmek için, açıklayıcı değişken olarak Ames Housing veri kümesinden Gr Living Area (zemin üzerindeki yaşam alanının feet kare cinsinden boyutu) özelliğini kullanalım ve SalePrice'ı tahmin edebilecek bir modeli eğitelim. Ayrıca, GD algoritmasının daha iyi yakınsaması için değişkenleri standartlaştıracağız. Kod aşağıdaki

```
>>> X = df[['Gr Yaşam Alanı']].values
>> y = df['SalePrice'].values
>> from sklearn.preprocessing import StandardScaler
>> sc_x = StandardScaler()
>> sc_y = StandardScaler()
>> X_std = sc_x.fit_transform(X)
>> y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()
>> lr = LinearRegressionGD(eta=0.1)
>> lr.fit(X_std, y_std)
```

gibidir:

np.newaxis ve flatten kullanarak y_std ile ilgili geçici çözüme dikkat edin. Scikit-learn'deki çoğu veri ön işleme sınıfı, verilerin iki boyutlu dizilerde saklanmasını bekler. Önceki kod örneğinde, y[:, n p .newaxis] içinde np.newaxis kullanımı diziye yeni bir boyut eklemiştir. Ardından, StandardScaler ölçeklendirilmiş değişkeni **döndürdükten** sonra, kolaylık sağlamak için **flatten()** yöntemini kullanarak orijinal tek boyutlu dizi temsiline geri dönüştürdük.

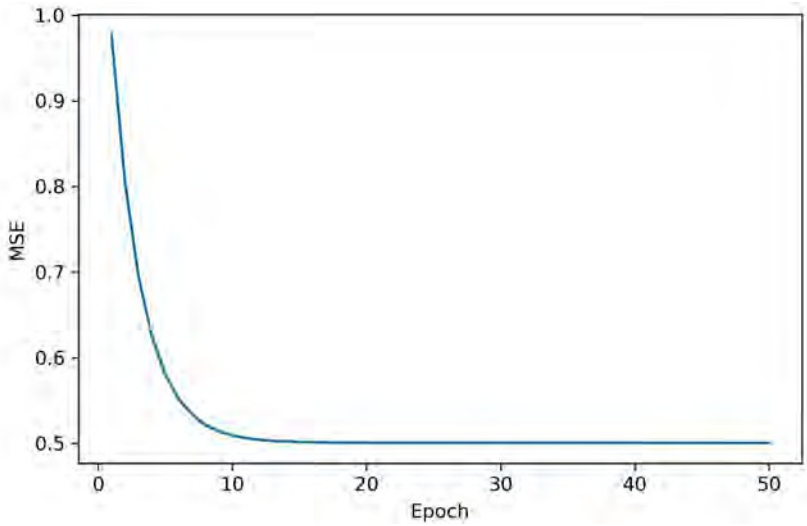
*Bölüm 2*'de, GD gibi optimizasyon algoritmaları kullanırken, algoritmanın bir kayıp minimumuna (burada, *global bir* kayıp minimumuna) yakınsadığını kontrol etmek için kaybı eğitim veri seti üzerindeki epok (tam iterasyon) sayısının bir fonksiyonu olarak çizmenin her zaman iyi bir fikir

```
>> plt.plot(range(1, lr.n_iter+1), lr.losses_)
>> plt.ylabel('MSE')
>>  plt.xlabel('Epoch')
>> plt.show()
```

olduğunu tartışmıştık:

 olimpikbarbell@gmail.com 

*Şekil 9.6*'da görebileceğiniz gibi, GD algoritması yaklaşık olarak onuncu epoktan sonra yakınsamıştır:




*Şekil 9.6: Epok sayısına karşı kayıp fonksiyonu*

Daha sonra, doğrusal regresyon çizgisinin eğitim verilerine ne kadar iyi uyduğunu görselleştirelim. Bunu yapmak için, eğitim örneklerinin bir dağılım grafiğini çizecek ve regresyon çizgisini ekleyecek

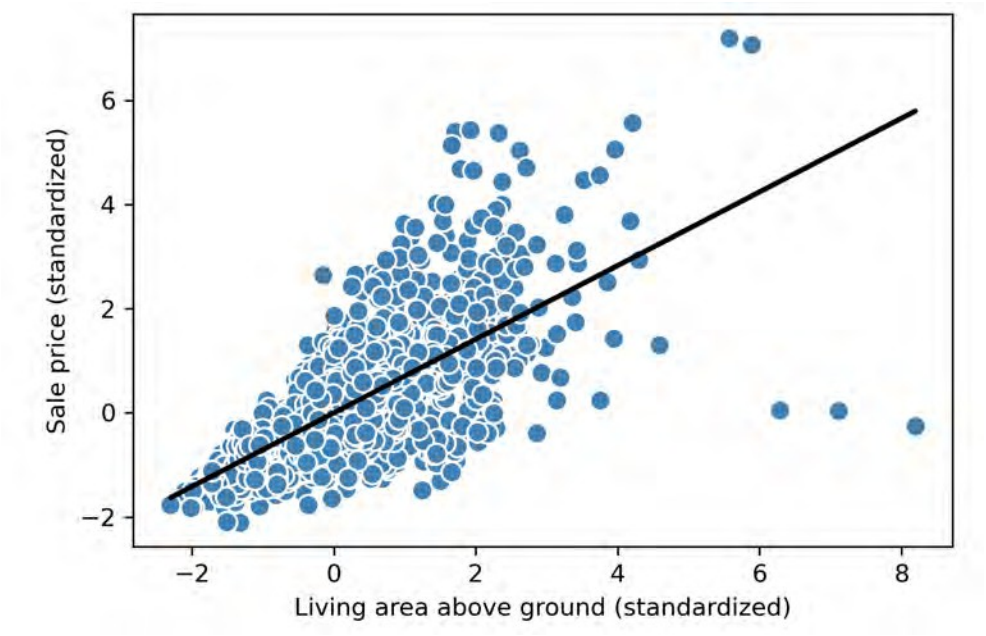
```
>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='steelblue', edgecolor='white', s=70)
...     plt.plot(X, model.predict(X), color='black', lw=2)
```

basit bir yardımcı fonksiyon tanımlayacağız:

Şimdi, bu lin_regplot fonksiyonunu kullanarak yaşam alanını satış fiyatına karşı çizeceğiz:

```
>> lin_regplot(X_std, y_std, lr)
>> plt.xlabel(' Yer üstündeki yaşam alanı (standartlaştırılmış)')
>> plt.ylabel('Satış fiyatı (standartlaştırılmış)')
>> plt.show()
```

*Şekil 9.7*'de görebileceğiniz gibi, doğrusal regresyon çizgisi, konut fiyatlarının yaşam alanının büyüklüğü ile artma eğiliminde olduğu genel eğilimini yansıtmaktadır:




*Şekil 9.7: Yaşam alanı büyüklüğüne karşı satış fiyatlarının doğrusal regresyon grafiği*

Bu gözlem mantıklı olsa da, veriler bize yaşam alanı büyüklüğünün birçok durumda ev fiyatlarını çok iyi açıklamadığını da söylemektedir. Bu bölümün ilerleyen kısımlarında, bir regresyon modelinin performansının nasıl ölçüleceğini tartışacağız. İlginç bir şekilde, örneğin 6'dan büyük standartlaştırılmış yaşam alanına karşılık gelen üç veri noktası gibi birkaç aykırı değer de gözlemleyebiliriz. Aykırı değerlerle nasıl başa çıkabileceğimizi bu bölümün ilerleyen kısımlarında tartışacağız.

Bazı uygulamalarda, tahmin edilen sonuç değişkenlerini orijinal ölçeklerinde raporlamak da önemli olabilir. Tahmin edilen fiyatı *ABD doları cinsinden orijinal fiyat* ölçeğine geri ölçeklendirmek için StandardScaler'ın inverse_transform yöntemini uygulayabiliriz:

```
>> özellik_std = sc_x.transform(np.array([[2500]]))
>>> hedef_std = lr.predict(özellik_std)
>> hedef_ters_dönüştürülmüş = sc_y.inverse_transform(hedef_std.reshape(-1, 1))
>> print(f'Satış fiyatı: ${target_reverted.flatten()[0]:.2f}')
Satış fiyatı: $292507.07
```

Bu kod örneğinde, 2.500 feet kare yer üstü yaşam alanına sahip bir evin fiyatını tahmin etmek için önceden eğitilmiş doğrusal regresyon modelini kullandık. Modelimize göre, böyle bir evin değeri 292.507,07 $ olacaktır.

Bir yan not olarak, standartlaştırılmış değişkenlerle çalışıyorsak teknik olarak kesişim parametresini (örneğin, yanlılık birimi, *b*) güncellememiz gerekmediğini de belirtmek gerekir, çünkü bu durumlarda *y* ekseni kesişimi her zaman 0'dır. Model parametrelerini yazdırarak bunu

```
>> print(f'Eğim: {lr.w_[0]:.3f}')
Eğim: 0.707
>> print(f'Intercept: {lr.b_[0]:.3f}')
Intercept: -0.000
```

hızlıca teyit edebiliriz:

## Scikit-learn ile bir regresyon modelinin katsayısını tahmin etme

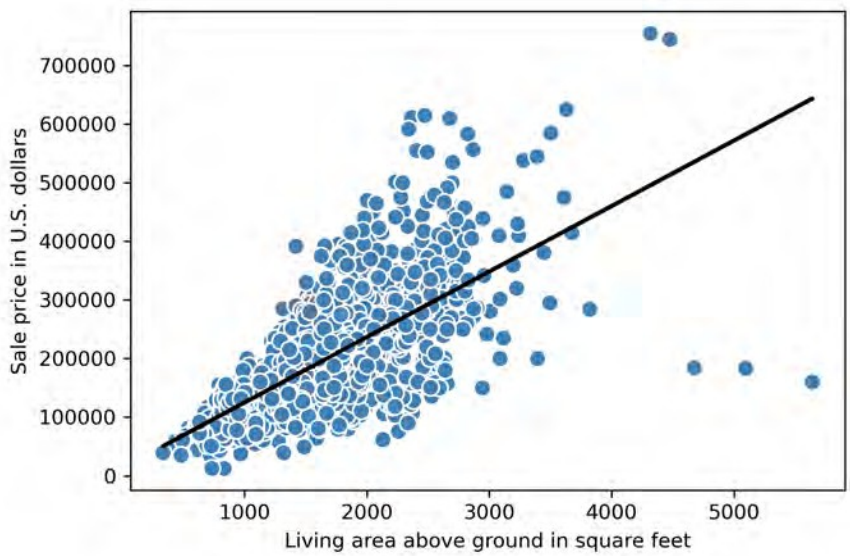
Önceki bölümde, regresyon analizi için çalışan bir model uyguladık; ancak, gerçek dünyadaki bir uygulamada, daha verimli uygulamalarla ilgilenebiliriz. Örneğin, scikit-learn'in regresyon tahmincilerinin çoğu SciPy'deki (scipy.linalg.lstsq) en küçük kareler uygulamasını kullanır ve bu da **Lineer Cebir Paketi**'ne (**LAPACK**) dayalı yüksek düzeyde optimize edilmiş kod optimizasyonlarını kullanır. Scikit-learn'deki doğrusal regresyon uygulaması, (S)GD tabanlı optimizasyon kullanmadığı için standartlaştırılmamış değişkenlerle de (daha iyi) çalışır, böylece standartlaştırma adımını atlayabiliriz:

```
>> from sklearn.linear_model import LinearRegression
>> slr = LinearRegression()
>> slr.fit(X, y)
>> y_pred = slr.predict(X)
>> print(f'Eğim: {slr.coef_[0]:.3f}') Eğim:
111.666
>> print(f'Intercept: {slr.intercept_:.3f}') Intercept:
13342.979
```

Bu kodun çalıştırılmasından da görebileceğiniz gibi, scikit-learn'in standartlaştırılmamış Gr Liv Area ve SalePrice değişkenleriyle donatılmış LinearRegression modeli, özellikler standartlaştırılmadığı için farklı model katsayıları vermiştir. Bununla birlikte, SalePrice'ı Gr Liv Area'ya karşı çizerek GD uygulamamızla karşılaştırdığımızda, niteliksel olarak verilere benzer şekilde iyi uyduğunu görebiliriz:

```
>> lin_regplot(X, y, slr)
>> plt.xlabel('Feet kare cinsinden yer üstündeki yaşam alanı')
>> plt.ylabel('ABD doları cinsinden satış fiyatı')
>> plt.tight_layout()
>> plt.show()
```

Örneğin, genel sonucun GD uygulamamızla aynı göründüğünü görebiliriz:




*Şekil 9.8: scikit-learn kullanarak doğrusal regresyon grafiği*

**Doğrusal regresyonun analitik çözümleri**

Makine öğrenimi kütüphanelerini kullanmaya alternatif olarak, doğrusal denklemler sistemini içeren OLS'yi çözmek için çoğu giriş istatistik ders kitabında bulunabilecek kapalı formda bir çözüm de vardır:

$$w = (X^T X)^{-1} X^T y$$

Bunu Python'da aşağıdaki gibi uygulayabiliriz:

```
# "birler" sütun vektörü ekleme
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print(f'Eğim: {w[1]:.3f}') Eğim:
111.666
>>> print(f'Intercept: {w[0]:.3f}')
Intercept: 13342.979
```

 olimpikbarbell@gmail.com 

Bu yöntemin avantajı, analitik olarak en uygun çözümü bulmasının garanti edilmesidir. Ancak, çok büyük veri kümeleriyle çalışıyorsak, bu formüldeki matrisi (bazen normal denklem olarak da adlandırılır) ters çevirmek hesaplama açısından çok pahalı olabilir veya eğitim örneklerini içeren matris tekil (ters çevrilemez) olabilir, bu nedenle bazı durumlarda iteratif yöntemleri tercih edebiliriz.

Normal denklemlerin nasıl elde edileceği hakkında daha fazla bilgi edinmek isterseniz, Dr. Stephen Pollock'un Leicester Üniversitesi'ndeki derslerinden *Klasik Doğrusal Regresyon Modeli* bölümüne göz atabilirsiniz. http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf adresinden ücretsiz olarak temin edilebilir.

Ayrıca, GD, SGD, kapalı form çözümü, QR çarpanlarına ayırma ve tekil vektör ayrıştırması yoluyla elde edilen doğrusal regresyon çözümlerini karşılaştırmak istiyorsanız, mlxtend'de uygulanan LinearRegression sınıfını kullanabilirsiniz (http://rasbt.github.io/mlxtend/user_guide/regressor/LinearRegression/), kullanıcıların bu seçenekler arasında geçiş yapmasını sağlar. Python'da regresyon modellemesi için önerilebilecek bir diğer harika kütüphane de https://www.statsmodels.org/stable/examples/index.html#regression adresinde gösterildiği gibi daha gelişmiş doğrusal regresyon modelleri uygulayan statsmodels'tir.

# RANSAC kullanarak sağlam bir regresyon modeli uydurma

Doğrusal regresyon modelleri aykırı değerlerin varlığından büyük ölçüde etkilenebilir. Bazı durumlarda, verilerimizin çok küçük bir alt kümesi, tahmin edilen model katsayıları üzerinde büyük bir etkiye sahip olabilir. Aykırı değerleri tespit etmek için birçok istatistiksel test kullanılabilir, ancak bunlar kitabın kapsamı dışındadır. Bununla birlikte, aykırı değerleri ortadan kaldırmak her zaman veri bilimcisi olarak kendi muhakememizi ve alan bilgimizi gerektirir.

Aykırı değerleri atmaya alternatif olarak, **RANdom SAmple Consensus** (**RANSAC**) algoritmasını kullanarak, aykırı değerler **olarak** adlandırılan verilerin bir alt kümesine bir regresyon modeli uyduran sağlam bir regresyon yöntemine bakacağız.

Yinelemeli RANSAC algoritmasını aşağıdaki gibi özetleyebiliriz:

1. Rastgele sayıda örneği aykırı değer olarak seçin ve modeli oturtun.
2. Diğer tüm veri noktalarını uygun modele göre test edin ve belirli bir tolerans dahilinde kalan noktaları aykırı değerlere ekleyin.
3. Tüm inliers kullanarak modeli yeniden kurun.
4. Uyumlu modelin hatasını aykırı değerlere göre tahmin edin.
5. Performans kullanıcı tanımlı belirli bir eşiği karşılıyorsa veya sabit bir iterasyon sayısına ulaşılmışsa algoritmayı sonlandırın; aksi takdirde *adım 1*'e geri dönün.

Şimdi scikit- learn'in RANSACRegressor sınıfında uygulanan RANSAC algoritması ile birlikte doğrusal bir model kullanalım:

```
>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(
...         LinearRegression(),
...         max_trials=100, # varsayılan değer
...         min_samples=0.95,
...         residual_threshold=None, # varsayılan değer
...         random_state=123)
>>> ransac.fit(X, y)
```

RANSACRegressor'un maksimum iterasyon sayısını 100 olarak ayarladık ve min_samples=0.95 kullanarak rastgele seçilen eğitim örneklerinin minimum sayısını veri kümesinin en az yüzde 95'i olacak şekilde ayarladık.

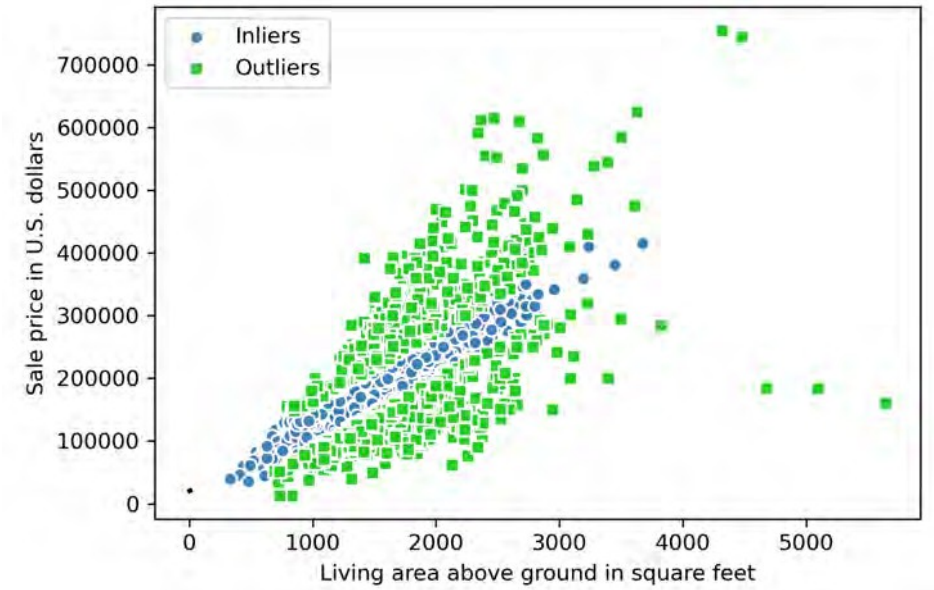
Varsayılan olarak (residual_threshold=None aracılığıyla), scikit-learn aykırı değer eşiğini seçmek için **MAD** tahminini kullanır, burada MAD hedef değerlerin **medyan mutlak sapması** anlamına gelir, y. Ancak, aykırı değer eşiği için uygun bir değerin seçimi probleme özgüdür, bu da RANSAC'ın bir dezavantajıdır.

Son yıllarda otomatik olarak iyi bir inlier eşiği seçmek için birçok farklı yaklaşım geliştirilmiştir. Ayrıntılı bir tartışmayı *R. Toldo* ve *A. Fusiello* tarafından yazılan *Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting*, *Springer,* 2009 (in *Image Analysis and Processing- ICIAP 2009*, pages: 123-131) adlı kitapta bulabilirsiniz.

RANSAC modelini uydurduktan sonra, uydurulan RANSAC doğrusal regresyon modelinden inliers ve outliers değerlerini elde edelim ve bunları doğrusal uyumla birlikte çizelim:

```
>>> inlier_mask = ransac.inlier_mask_
>>>  outlier_mask = np.logical_not(inlier_mask)
>> satır_X = np.arange(3, 10, 1)
>>>  line_y_ransac = ransac.predict(line_X[:, np.newaxis])
>> plt.scatter(X[inlier_mask], y [inlier_mask],
...                 c='steelblue', edgecolor='white',
...                 marker='o', label='Inliers')
>>  plt.scatter(X[outlier_mask], y [outlier_mask],
...                 c='limegreen', edgecolor='white',
...                 marker='s', label='Outliers')
>> plt.plot(line_X, line_y_ransac, color='black', lw=2)
>> plt.xlabel('Feet kare cinsinden yer üstündeki yaşam alanı')
>> plt.ylabel('ABD doları cinsinden satış fiyatı')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

*Şekil 9.9*'da görebileceğiniz gibi, doğrusal regresyon modeli tespit edilen ve daireler şeklinde gösterilen inlier kümesine yerleştirilmiştir:




*Şekil 9.9: RANSAC doğrusal regresyon modeli aracılığıyla belirlenen aykırı değerler ve aykırı değerler*

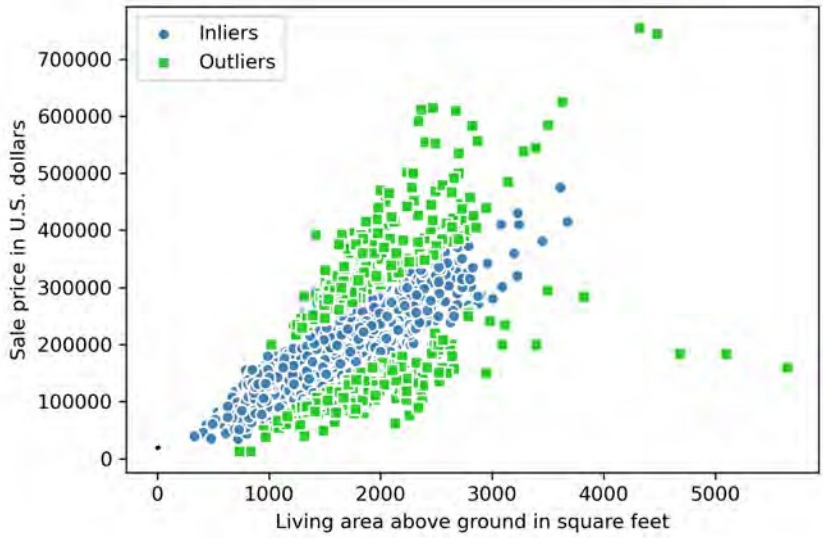
Aşağıdaki kodu çalıştırarak modelin eğimini ve kesişimini yazdırdığımızda, doğrusal regresyon çizgisi bir önceki bölümde RANSAC kullanmadan elde ettiğimiz uyumdan biraz farklı olacaktır:

```
>> print(f'Slope: {ransac.estimator_.coef_[0]:.3f}')
Eğim: 106.348
>> print(f'Intercept: {ransac.estimator_.intercept_:.3f}')
Intercept: 20190.093
```

residual_threshold parametresini None olarak ayarladığımızı, dolayısıyla RANSAC'ın inliers ve outliers'ı işaretleme eşiğini hesaplamak için MAD'yi kullandığını hatırlayın. Bu veri kümesi için MAD aşağıdaki gibi hesaplanabilir:

```
>> def median_absolute_deviation(data):
... return np.median(np.abs(data - np.median(data)))
>>> median_absolute_deviation(y)
37000.00
```

Dolayısıyla, daha az veri noktasını aykırı değer olarak tanımlamak istiyorsak, önceki MAD değerinden daha büyük bir residual_threshold değeri seçebiliriz. Örneğin, *Şekil 9.10'da* 65.000 artık eşik değerine sahip bir RANSAC doğrusal regresyon modelinin aykırı ve aykırı değerleri gösterilmektedir:




*Şekil 9.10: RANSAC doğrusal regresyon modeli tarafından belirlenen ve daha büyük bir yan eşik değerine sahip aykırı değerler*

RANSAC kullanarak bu veri kümesindeki aykırı değerlerin potansiyel etkisini azalttık, ancak bu yaklaşımın görünmeyen veriler için tahmin performansı üzerinde olumlu bir etkisi olup olmayacağını bilmiyoruz. Bu nedenle, bir sonraki bölümde, tahmine dayalı modelleme için sistemler oluşturmanın çok önemli bir parçası olan bir regresyon modelini değerlendirmek için farklı yaklaşımlara bakacağız.

# Doğrusal regresyon modellerinin performansının değerlendirilmesi

Önceki bölümde, bir regresyon modelini eğitim verilerine nasıl uyduracağınızı öğrendiniz. Ancak önceki bölümlerde, genelleme performansına ilişkin daha tarafsız bir tahmin elde etmek için modeli eğitim sırasında görmediği veriler üzerinde test etmenin çok önemli olduğunu keşfettiniz.

*Model Değerlendirmesi ve Hiperparametrelerin Ayarlanması için En İyi Öğrenme Uygulamaları* başlıklı *6. Bölümden* hatırlayacağınız üzere, veri setimizi ayrı eğitim ve test veri setlerine bölmek istiyoruz; burada ilkini modeli oturtmak için, ikincisini ise genelleme performansını tahmin etmek için görülmeyen veriler üzerindeki performansını değerlendirmek için kullanacağız. Basit regresyon modeliyle devam etmek yerine, şimdi veri kümesindeki beş özelliğin tümünü kullanacağız ve bir çoklu regresyon

```
>> from sklearn.model_selection import train_test_split
>> target = 'SalePrice'
>>> özellikler = df.columns[df.columns != target]
>>> X = df[özellikler].değerler
>> y = df[hedef].değerler
```

 olimpikbarbell@gmail.com 

modeli eğiteceğiz:

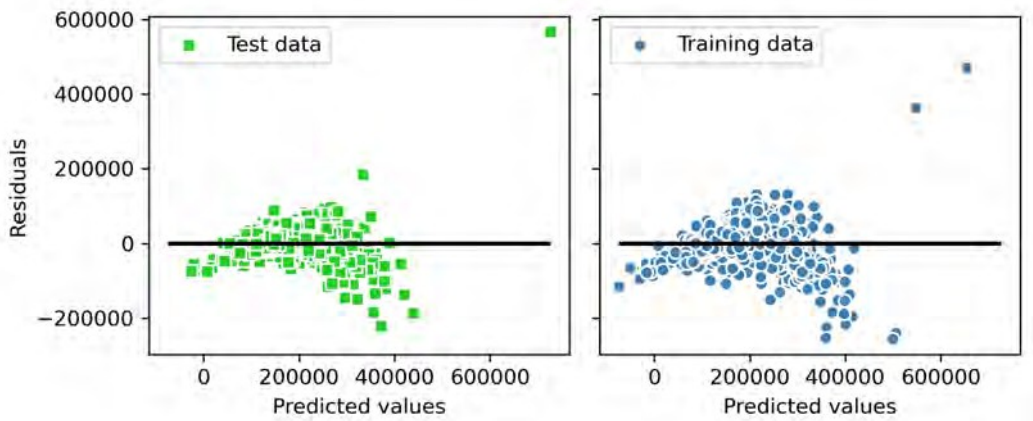
```
>> X_train, X_test, y_train, y_test = train_test_split(
...         X, y, test_size=0.3, random_state=123)
>> slr = LinearRegression()
>> slr.fit(X_train, y_train)
>> y_train_pred = slr.predict(X_train)
>> y_test_pred = slr.predict(X_test)
```

Modelimizde birden fazla açıklayıcı değişken kullanıldığından, doğrusal regresyon çizgisini (ya da daha kesin olmak gerekirse hiper düzlemi) iki boyutlu bir grafikte görselleştiremeyiz, ancak regresyon modelimizi teşhis etmek için tahmin edilen değerlere karşı kalıntıları (gerçek ve tahmin edilen değerler arasındaki farklar veya dikey mesafeler) çizebiliriz. Artık grafikleri, regresyon modellerini teşhis etmek için yaygın olarak kullanılan grafiksel bir araçtır. Doğrusal olmama durumunu ve aykırı değerleri tespit etmeye ve hataların rastgele dağılıp dağılmadığını kontrol etmeye yardımcı olabilirler.

Aşağıdaki kodu kullanarak, şimdi tahmin edilen yanıtlarımızdan gerçek hedef değişkenleri çıkardığımız bir artık grafiği çizeceğiz:

```
>> x_max = np.max(
...         [np.max(y_train_pred), np.max(y_test_pred)])
>> x_min = np.min(
...         [np.min(y_train_pred), np.min(y_test_pred)])

>> fig, (ax1, ax2) = plt.subplots(
...         1, 2, figsize=(7, 3), sharey=True)

>> ax1.scatter(
...         y_test_pred, y_test_pred - y_test,
...         c='limegreen', marker='s',
...         edgecolor='beyaz',
...         label='Test verileri')
>> ax2.scatter(
...         y_train_pred, y_train_pred - y_train,
...         c='steelblue', marker='o', edgecolor='white',
...         label='Eğitim verileri')
> >    ax1.set_ylabel('Artıklar')

>> for ax in (ax1, ax2):
...         ax.set_xlabel('Tahmin edilen değerler')
...         ax.legend(loc='sol üst')
...         ax.hlines(y=0, xmin=x_min-100, xmax=x_max+100,\
...                 color='black', lw=2)
>> plt.tight_layout()
>> plt.show()
```

Kodu çalıştırdıktan sonra, *Şekil 9.11'de* gösterildiği gibi *x* ekseni orijininden geçen bir çizgi ile test ve eğitim veri kümeleri için artık grafikleri görmeliyiz:




*Şekil 9.11: Verilerimizin artık grafikleri*

Mükemmel bir tahmin durumunda, artıklar tam olarak sıfır olacaktır ki gerçekçi ve pratik uygulamalarda muhtemelen bununla hiç karşılaşmayacağız. Ancak, iyi bir regresyon modeli için hataların rastgele dağılmasını ve artıkların merkez çizgisi etrafında rastgele dağılmasını bekleriz. Eğer bir kalıntı grafiğinde örüntüler görürsek, bu, modelimizin, önceki kalıntı grafiğimizde bir dereceye kadar görebileceğiniz gibi, kalıntılara sızmış olan bazı dışsal bilgileri yakalayamadığı anlamına gelir. Ayrıca, merkez çizgisinden büyük sapma gösteren noktalarla temsil edilen aykırı değerleri tespit etmek için de artık grafikleri kullanabiliriz.

Bir modelin performansının bir başka faydalı nicel ölçütü de, doğrusal regresyon modelini uydurmak için minimize ettiğimiz kayıp fonksiyonumuz olarak daha önce tartıştığımız **ortalama karesel hatadır** (**MSE**). MSE

MSE'nin aşağıdaki versiyonu $\frac{1}{2}$ kaybı basitleştirmek için sıklıkla kullanılan ölçekleme faktörü gradyan inişinde türev:

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

Sınıflandırma bağlamındaki tahmin doğruluğuna benzer şekilde, *Bölüm 6'da* tartışıldığı gibi çapraz doğrulama ve model seçimi için MSE'yi kullanabiliriz.

Sınıflandırma doğruluğu gibi, MSE de örneklem büyüklüğüne göre normalleşir, *n*. Bu, farklı örneklem büyüklükleri arasında (örneğin, öğrenme eğrileri bağlamında) karşılaştırma yapmayı da mümkün kılar.

Şimdi eğitim ve test tahminlerimizin MSE'sini hesaplayalım:

```
>> from sklearn.metrics import mean_squared_error
>> mse_train = mean_squared_error(y_train, y_train_pred)
>> mse_test = mean_squared_error(y_test, y_test_pred)
```

```
>> print(f'MSE t r a i n : {mse_train:.2f}')
MSE train: 1497216245.85
>> print(f'MSE testi: {mse_test:.2f}')
MSE testi: 1516565821.00
```

Eğitim veri kümesindeki MSE'nin test kümesindekinden daha az olduğunu görebiliriz; bu da modelimizin bu durumda eğitim verilerine biraz fazla uyum sağladığının bir göstergesidir. Hatayı orijinal birim ölçeğinde (burada *dolar-kare* yerine dolar) göstermenin daha sezgisel olabileceğini unutmayın, bu nedenle MSE'nin karekökünü hesaplamayı seçebiliriz, buna *kök ortalama karesel hata* veya yanlış tahmini biraz daha az vurgulayan **ortalama mutlak hata** (**MAE)** denir:

$$MAE\ M\ \frac{1}{n}\sum_{ii1}^{n} |y^{(ii} - yy^{(ii}|$$

MAE'yi MSE'ye benzer şekilde hesaplayabiliriz:

```
>> from sklearn.metrics import mean_absolute_error
>> mae_train = mean_absolute_error(y_train, y_train_pred)
>> mae_test = mean_absolute_error(y_test, y_test_pred)
>> print(f'MAE train: {mae_train:.2f}')
MAE train: 25983.03
>> print(f'MAE testi: {mae_test:.2f}') MAE
testi: 24921.29
```

Test seti MAE'sine dayanarak, modelin ortalama olarak yaklaşık 25.000 $ hata yaptığını söyleyebiliriz.

Modelleri karşılaştırmak için MAE veya MSE'yi kullandığımızda, örneğin sınıflandırma doğruluğunun aksine bunların sınırsız olduğunun farkında olmamız gerekir. Başka bir deyişle, MAE ve MSE'nin yorumları veri kümesine ve özellik ölçeklendirmesine bağlıdır. Örneğin, satış fiyatları 1.000'in katları olarak sunulmuşsa (K son ekiyle), aynı model ölçeklenmemiş özelliklerle çalışan bir modele kıyasla daha düşük bir MAE verecektir. Bu noktayı daha iyi açıklamak için,

$$|\$500K| - 550K| < |\$500{,}000 - 550{,}000|$$

Bu nedenle, modelin performansının daha iyi yorumlanabilmesi için bazen MSE'nin standartlaştırılmış bir versiyonu olarak anlaşılabilecek **belirleme katsayısını** ($R^2$ ) raporlamak daha faydalı olabilir. Başka bir deyişle, $R^2$ model tarafından yakalanan yanıt varyansının oranıdır. $R^2$ değeri şu şekilde tanımlanır:

$$R^2 = 1 - \frac{SSE}{SST}$$

Burada SSE, MSE'ye benzeyen ancak örneklem büyüklüğü $n$'ye göre normalleştirmeyi içermeyen karesel hataların toplamıdır:

$$SSE = \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2$$

SST ise karelerin toplamıdır:

$$SST = \sum_{i=1}^{n} (y^{(i)} - \mu_y)^2$$

Başka bir deyişle, SST basitçe yanıtın varyansıdır.

Şimdi kısaca $R'nin^2$ gerçekten de MSE'nin yeniden ölçeklendirilmiş bir versiyonu olduğunu gösterelim:

$$R^2 = 1 - \frac{\frac{1}{n} SSE}{\frac{1}{n} SST}$$

$$= \frac{\frac{1}{n}\sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{n}\sum_{i=1}^{n} (y^{(i)} - \mu_y)^2}$$

$$= 1 - \frac{MSE}{Var(y)}$$

Eğitim veri kümesi için $R^2$ 0 ile 1 arasında sınırlandırılmıştır, ancak test veri kümesi için negatif olabilir. Negatif bir $R^2$ regresyon modelinin verilere örnek ortalamasını temsil eden yatay bir çizgiden daha kötü uyduğu anlamına gelir. (Pratikte bu genellikle aşırı uyum durumunda veya test setini eğitim setini ölçeklendirdiğimiz şekilde ölçeklendirmeyi unuttuğumuzda meydana gelir). $R^2 = 1$ ise model, karşılık gelen $MSE = 0$ ile verilere mükemmel bir şekilde uyar.

Eğitim verileri üzerinde değerlendirildiğinde, modelimizin $R^2$ değeri 0,77'dir, bu harika değildir ancak sadece küçük bir özellik kümesiyle çalıştığımız göz önüne alındığında çok da kötü değildir. Bununla birlikte, test veri setindeki $R^2$ 0,75 ile sadece biraz daha küçüktür, bu da modelin sadece

```
>> from sklearn.metrics import r2_score
>> train_r2 = r2_score(y_train, y_train_pred)
>> test_r2 = r2_score(y_test, y_test_pred)
>> print(f'R^2 train: {train_r2:.3f}, {test_r2:.3f}') R^2
train: 0.77, test: 0.75
```

biraz fazla uyum sağladığını gösterir:

# regresyon için düzenli yöntemlerin kullanılması

*Bölüm 3, Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu*'nda tartıştığımız gibi, düzenlileştirme, ek bilgi ekleyerek ve böylece karmaşıklığa karşı bir ceza oluşturmak için modelin parametre değerlerini küçülterek aşırı uyum sorunuyla başa çıkmaya yönelik bir yaklaşımdır. Düzenlenmiş doğrusal regresyona yönelik en popüler yaklaşımlar **ridge regresyonu**, en **küçük mutlak küçültme ve seçim operatörü** (**LASSO**) ve **elastik ağ olarak** adlandırılır.

Ridge regresyonu, ağırlıkların karesel toplamını MSE kayıp fonksiyonuna eklediğimiz L2 cezalı bir modeldir:

$$L(w,b)_{RidgeRegression} = \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \|w\|_2^2$$

Burada, L2 terimi aşağıdaki gibi tanımlanır:

$$\|w\|_2^2 = \sum_{j=1}^{m} w_j^2$$

Hiperparametre $\lambda$ değerini artırarak, düzenlileştirme gücünü artırırız ve böylece modelimizin ağırlıklarını küçültürüz. *Bölüm 3*'te belirtildiği gibi, $b$ önyargı biriminin düzenli hale getirilmediğini lütfen unutmayın.

Seyrek modellere yol açabilecek alternatif bir yaklaşım LASSO'dur. Düzenlileştirme gücüne bağlı olarak, belirli ağırlıklar sıfır olabilir, bu da LASSO'yu denetimli bir özellik seçim tekniği olarak kullanışlı hale getirir:

$$L(w,b)_{Lasso} = \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \|w\|_1$$

Burada, LASSO için L1 cezası, aşağıdaki gibi model ağırlıklarının mutlak büyüklüklerinin toplamı olarak tanımlanır:

$$\|w\|_1 = \sum_{j=1}^{m} |w_j|$$

Bununla birlikte, LASSO'nun bir sınırlaması, $m > n$ ise en fazla $n$ özellik seçmesidir; burada $n$ eğitim örneklerinin sayısıdır. Bu, özellik seçiminin belirli uygulamalarında istenmeyen bir durum olabilir. Ancak pratikte, LASSO'nun bu özelliği genellikle bir avantajdır çünkü doymuş modelleri önler. Bir modelin doygunluğa ulaşması, eğitim örneklerinin sayısının özellik sayısına eşit olması durumunda gerçekleşir ki bu da aşırı parametrelendirmenin bir şeklidir. Sonuç olarak, doymuş bir model eğitim verilerine her zaman mükemmel bir şekilde uyabilir, ancak yalnızca bir enterpolasyon biçimidir ve bu nedenle iyi genelleme yapması beklenmez.

Ridge regresyonu ve LASSO arasında bir uzlaşma, seyreklik oluşturmak için bir L1 cezası ve $m > n$ *ise* $n$'den fazla özellik seçmek için kullanılabilecek bir L2 cezası olan elastik ağdır:

$$L(w,b)_{ElasticNet} = \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2 + \lambda_2 \|w\|_2^2 + \lambda_1 \|w\|_1$$

Bu düzenli regresyon modellerinin tümü scikit-learn aracılığıyla kullanılabilir ve kullanımları, parametre aracılığıyla düzenli hale getirme gücünü belirtmemiz gerekmesi dışında normal regresyon modeline benzer

$\lambda$, örneğin, k-kat çapraz doğrulama ile optimize edilmiştir.

 olimpikbarbell@gmail.com 

Bir ridge regresyon modeli şu yolla başlatılabilir:

```
>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Düzenli hale getirme gücünün, $\lambda$ parametresine benzer olan alpha parametresi tarafından düzenlendiğini unutmayın. Benzer şekilde, linear_model alt modülünden bir LASSO regresörünü başlatabiliriz:

```
>> from sklearn.linear_model import Kement
>> lasso = Kement(alpha=1.0)
```

Son olarak, ElasticNet uygulaması L1 - L2 oranını değiştirmemize olanak tanır:

```
>> from sklearn.linear_model import ElasticNet
>> elanet = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

Örneğin, l1_ratio değerini 1.0 olarak ayarlarsak, ElasticNet regresörü LASSO regresyonuna eşit olacaktır. Doğrusal regresyonun farklı uygulamaları hakkında daha ayrıntılı bilgi için lütfen http://scikit-learn.org/stable/modules/linear_model.html adresindeki belgelere bakın.

# Doğrusal bir regresyon modelini eğriye dönüştürme - polinom regresyon

Önceki bölümlerde, açıklayıcı ve yanıt değişkenleri arasında doğrusal bir ilişki olduğunu varsaymıştık. Doğrusallık varsayımının ihlalini hesaba katmanın bir yolu, polinom terimler ekleyerek bir polinom regresyon modeli kullanmaktır:

$$y\, yy1x + yy2x^2 + \ldots + {}_{yyd}\, x^d + b$$

Burada *d,* polinomun derecesini göstermektedir. Doğrusal olmayan bir ilişkiyi modellemek için polinom regresyonu kullanabilmemize rağmen, doğrusal regresyon katsayıları ***w*** nedeniyle hala çoklu doğrusal regresyon modeli olarak kabul edilir. Aşağıdaki alt bölümlerde, bu tür polinom terimlerini mevcut bir veri kümesine nasıl ekleyebileceğimizi ve bir polinom regresyon modelini nasıl uydurabileceğimizi göreceğiz.

## Scikit-learn kullanarak polinom terimleri ekleme

Şimdi tek açıklayıcı değişkenli basit bir regresyon problemine ikinci dereceden bir terim ($d$ = 2) eklemek için scikit-learn'in PolynomialFeatures dönüştürücü sınıfını nasıl kullanacağımızı öğreneceğiz. Ardından, aşağıdaki adımları izleyerek polinom ile doğrusal uyumu karşılaştıracağız:

1. İkinci dereceden bir polinom terimi ekleyin:

```
>> from sklearn.preprocessing import PolynomialFeatures
>> X = np.array([ 258.0, 270.0, 294.0, 320.0, 342.0,
...                 368.0, 396.0, 446.0, 480.0, 586.0])\
...              [:, np.newaxis]
>> y = np.array([ 236.4, 234.4, 252.8, 298.6, 314.2,
...                 342.2, 360.8, 368.0, 391.2, 390.8])
>> lr = LinearRegression()
>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>  X_quad = quadratic.fit_transform(X)
```

2. Karşılaştırma için basit bir doğrusal regresyon modeli kurun:

```
>> lr.fit(X, y)
>> X_fit = np.arange(250, 600, 10)[:, np.newaxis]
>> y_lin_fit = lr.predict(X_fit)
```

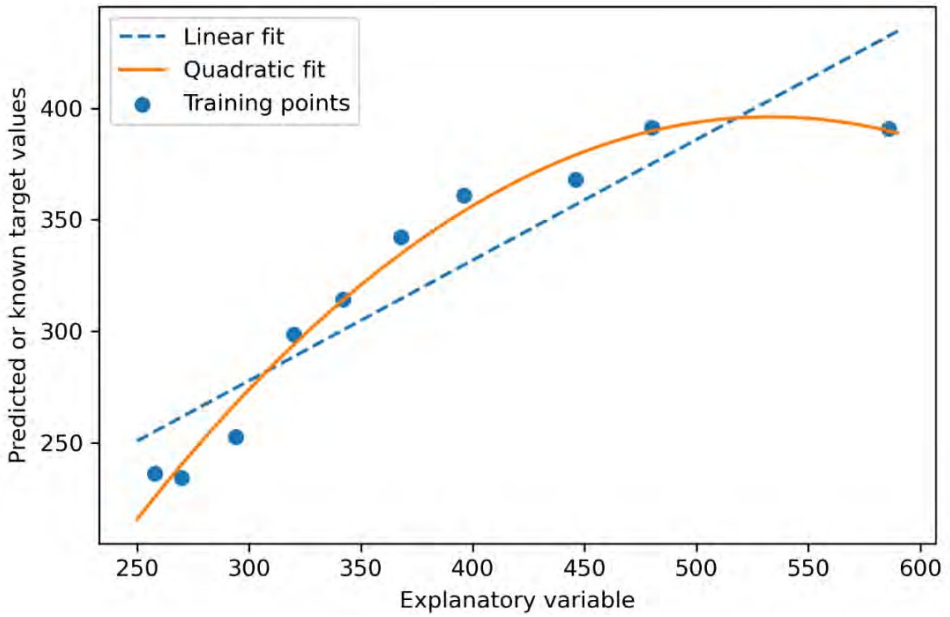
3. Polinom regresyon için dönüştürülmüş özellikler üzerine çoklu regresyon modeli yerleştirin:

```
>> pr.fit(X_quad, y)
>>   y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
```

4. Sonuçları çizin:

```
>> plt.scatter(X, y, label='Eğitim noktaları')
>> plt.plot(X_fit, y_lin_fit,
...          label='Linear fit', linestyle='--')
>> plt.plot(X_fit, y_quad_fit,
...          label='Quadratic fit')
>> plt.xlabel('Açıklayıcı değişken')
>> plt.ylabel('Tahmin edilen veya bilinen hedef değerler')
>> plt.legend(loc='sol üst')
>> plt.tight_layout()
>> plt.show()
```

Ortaya çıkan grafikte, polinom uyumunun yanıt ve açıklayıcı değişkenler arasındaki ilişkiyi doğrusal uyumdan çok daha iyi yakaladığını görebilirsiniz:




*Şekil 9.12: Doğrusal ve kuadratik modelin karşılaştırılması*

Daha sonra, MSE ve $R^2$ değerlendirme metriklerini hesaplayacağız:

```
>> y_lin_pred = lr.predict(X)
>> y_quad_pred = pr.predict(X_quad)
>> mse_lin = mean_squared_error(y, y_lin_pred)
>> mse_quad = mean_squared_error(y, y_quad_pred)
>> print(f'Training MSE linear: {mse_lin:.3f}'
         f', quadratic: {mse_quad:.3f}')
Eğitim MSE doğrusal: 569.780, ikinci dereceden: 61.330
>>> r2_lin = r2_score(y, y_lin_pred)
>> r2_quad = r2_score(y, y_quad_pred)
>> print(f'Training R^2 linear: {r2_lin:.3f}'
         f', quadratic: {r2_quad:.3f}')
Eğitim R^2 doğrusal: 0,832, ikinci dereceden: 0,982
```

Kodu çalıştırdıktan sonra görebileceğiniz gibi, MSE 570'ten (doğrusal uyum) 61'e (ikinci dereceden uyum) düşmüştür; ayrıca belirleme katsayısı, bu özel oyuncak probleminde doğrusal uyumun ($R^2$ = 0,832) aksine ikinci dereceden modelin ($R^2$ = 0,982) daha yakın bir uyumunu yansıtmaktadır.

## Ames Konut veri setinde doğrusal olmayan ilişkilerin modellenmesi

Bir önceki alt bölümde, oyuncak bir problemde doğrusal olmayan ilişkilere uyacak polinom özelliklerinin nasıl oluşturulacağını öğrendiniz; şimdi daha somut bir örneğe bakalım ve bu kavramları Ames Konut veri kümesindeki verilere uygulayalım. Aşağıdaki kodu çalıştırarak, ikinci derece (kuadratik) ve üçüncü derece (kübik) polinomlar kullanarak satış fiyatları ile yer üstündeki yaşam alanı arasındaki ilişkiyi modelleyeceğiz ve bunu doğrusal bir uyumla karşılaştıracağız.

*Şekil 9.8'*de olduğu gibi önceki şekillerde de görebileceğimiz 4.000 feet kareden büyük yaşam alanına sahip üç aykırı değeri kaldırarak başlıyoruz, böylece bu aykırı değerler regresyon

```
>>> X = df[['Gr Yaşam Alanı']].values
>> y = df['SalePrice'].values
>>> X = X[(df['Gr Yaşam Alanı'] < 4000)]
>>> y = y[(df['Gr Yaşam Alanı'] < 4000)]
```

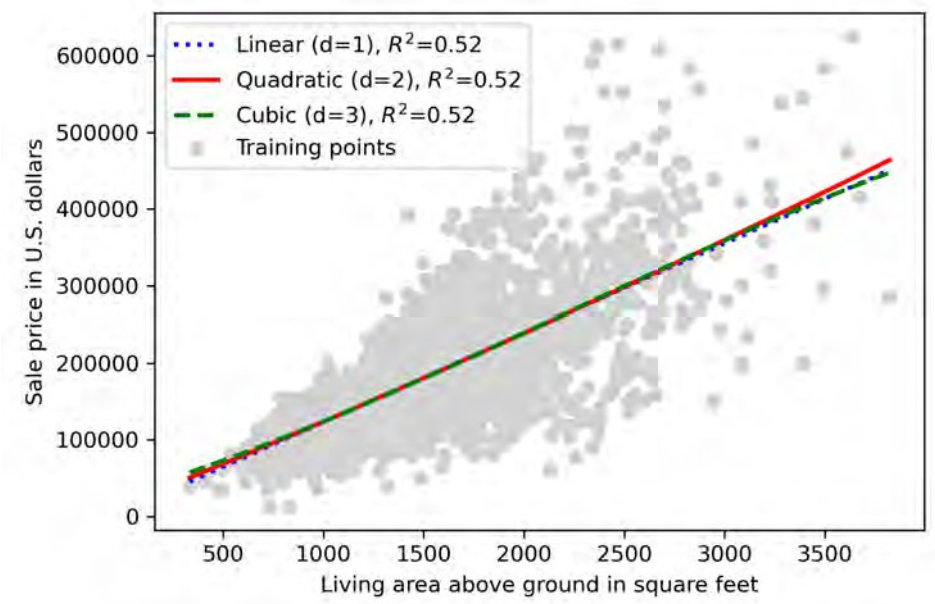
uyumlarımızı çarpıtmıyor:

Ardından, regresyon modellerini kuruyoruz:

```
>> regr = LinearRegression()

>> # kuadratik ve kübik özellikler oluşturun
>>> quadratic = PolynomialFeatures(degree=2)
>> cubic = PolynomialFeatures(degree=3)
> >  X_quad = quadratic.fit_transform(X)
>> X_cubic = cubic.fit_transform(X)

>> # özelliklere sığdır
>> X_fit = np.arange(X.min()-1, X.max()+2, 1)[:, np.newaxis]
>> regr = regr.fit(X, y)
>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))
>> regr = regr.fit(X_quad, y)
> >   y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))
>> regr = regr.fit(X_cubic, y)
> >   y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>> cubic_r2 = r2_score(y, regr.predict(X_cubic))

>> # sonuçları çizin
>> plt.scatter(X, y, label='Eğitim noktaları', color='lightgray')
>> plt.plot(X_fit, y_lin_fit,
...              label=f'Doğrusal (d=1), $R^2$={linear_r2:.2f}',
```

```
...             color='blue',
...             lw=2,
...             linestyle=':')
>> plt.plot(X_fit, y_quad_fit,
...             label=f'İkinci dereceden (d=2),
...             $R^2$={quadratic_r2:.2f}',
...             color='red',
...             lw=2,
...             linestyle='-')
>> plt.plot(X_fit, y_cubic_fit,
...             label=f'Kübik (d=3), $R^2$={cubic_r2:.2f}',
...             color='green',
...             lw=2,
...             linestyle='--')
>> plt.xlabel('Feet kare cinsinden yer üstündeki yaşam alanı')
>> plt.ylabel('ABD doları cinsinden satış fiyatı')
>> plt.legend(loc='sol üst')
>> plt.show()
```

Ortaya çıkan çizim *Şekil 9.13*'te gösterilmektedir:

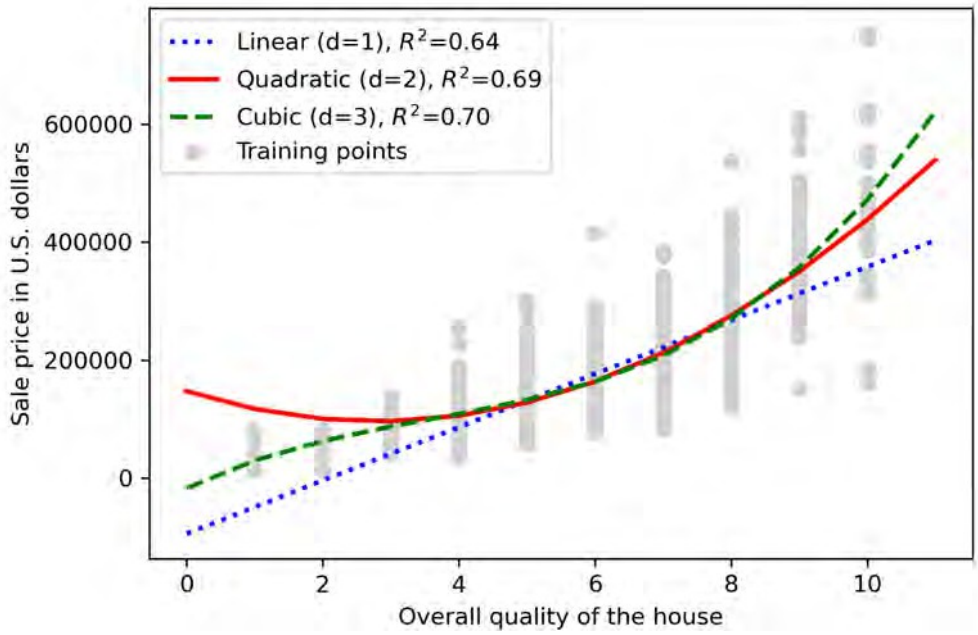



*Şekil 9.13: Satış fiyatı ve yaşam alanı verilerine uydurulan farklı eğrilerin karşılaştırılması*

Gördüğümüz gibi, ikinci dereceden veya kübik özellikler kullanmanın gerçekten bir etkisi yoktur. Bunun nedeni, iki değişken arasındaki ilişkinin doğrusal görünmesidir. Şimdi de başka bir özelliğe, yani Overall Qual'a bakalım. Overall Qual değişkeni, konutların malzeme ve bitiş kalitesinin genelini derecelendirir ve 10'un en iyi olduğu 1'den 10'a kadar bir ölçekte verilir:

```
>>> X = df[['Genel Nitelik']].values
>> y = df['SalePrice'].values
```

X ve y değişkenlerini belirledikten sonra, önceki kodu tekrar kullanabilir ve *Şekil 9.14'*teki grafiği elde edebiliriz:




*Şekil 9.14: Satış fiyatı ve konut kalitesi verileri üzerinde doğrusal, ikinci dereceden ve kübik uyum*

Gördüğünüz gibi, ikinci dereceden ve kübik uyumlar, satış fiyatları ile evin genel kalitesi arasındaki ilişkiyi doğrusal uyumdan daha iyi yakalamaktadır. Ancak, giderek daha fazla polinom özelliği eklemenin bir modelin karmaşıklığını artırdığını ve dolayısıyla aşırı uyum şansını artırdığını bilmelisiniz. Bu nedenle, uygulamada, genelleme performansını tahmin etmek için modelin performansının her zaman ayrı bir test veri kümesi üzerinde değerlendirilmesi önerilir.

# Rastgele ormanları kullanarak doğrusal olmayan ilişkilerle başa çıkma

Bu bölümde, bu bölümdeki önceki regresyon modellerinden kavramsal olarak farklı olan **rastgele orman** regresyonuna bakacağız. Birden fazla **karar ağacından oluşan** bir topluluk olan rastgele orman, daha önce ele aldığımız global doğrusal ve polinom regresyon modellerinin aksine, parçalı doğrusal fonksiyonların toplamı olarak anlaşılabilir. Başka bir deyişle, karar ağacı algoritması aracılığıyla girdi uzayını daha yönetilebilir hale gelen daha küçük bölgelere ayırırız.

## Karar ağacı regresyonu

Karar ağacı algoritmasının bir avantajı, rastgele özelliklerle çalışması ve doğrusal olmayan verilerle uğraşıyorsak özelliklerin herhangi bir dönüşümünü gerektirmemesidir, çünkü karar ağaçları ağırlıklı kombinasyonları hesaba katmak yerine her seferinde bir özelliği analiz eder. (Aynı şekilde, karar ağaçları için özelliklerin normalleştirilmesi veya standartlaştırılması gerekli değildir). *Bölüm 3, Scikit-Learn* Kullanarak *Makine Öğrenimi Sınıflandırıcıları Turu'nda* belirtildiği gibi, yapraklar saf olana veya bir durdurma kriteri karşılanana kadar düğümlerini iteratif olarak bölerek bir karar ağacını büyütürüz. Sınıflandırma için karar ağaçlarını kullandığımızda, hangi özellik bölünmesinin **bilgi kazancını** (**IG**) en üst düzeye çıkardığını belirlemek için entropiyi bir safsızlık ölçüsü olarak tanımladık; bu, ikili bir bölünme için aşağıdaki gibi tanımlanabilir:

$$IG(D_p, x_i) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

Burada, $x_i$ bölme işlemini gerçekleştirecek özellik, $N_p$ ana düğümdeki eğitim örneklerinin sayısı, $I$ safsızlık fonksiyonu, $D_p$ ana düğümdeki eğitim örneklerinin alt kümesi ve $D_{left}$ ve $D_{right}$ bölme işleminden sonra sol ve sağ çocuk düğümlerdeki eğitim örneklerinin alt kümeleridir. Amacımızın bilgi kazancını en üst düzeye çıkaran özellik bölünmesini bulmak olduğunu unutmayın; başka bir deyişle, alt düğümlerdeki safsızlıkları en çok azaltan özellik bölünmesini bulmak istiyoruz. *Bölüm 3'te,* her ikisi de sınıflandırma için yararlı ölçütler olan safsızlık ölçütleri olarak Gini safsızlığı ve entropiyi tartıştık. Ancak regresyon için bir karar ağacı kullanmak için sürekli değişkenlere uygun bir safsızlık ölçütüne ihtiyacımız vardır, bu nedenle bir düğümün safsızlık ölçütünü, *t,* bunun yerine MSE olarak tanımlarız:

$$I(t) = MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} (y^{(i)} - \hat{y}_t)^2$$

Burada, $N_t$ $t$ düğümündeki eğitim örneklerinin sayısı, $D_t$ $t$ düğümündeki eğitim alt kümesi, $y^{(i)}$ gerçek hedef değer ve $\hat{y}_t$ tahmin edilen hedef değerdir (örnek ortalaması):

$$\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} y^{(i)}$$

Karar ağacı regresyonu bağlamında, MSE genellikle **düğüm içi varyans** olarak adlandırılır, bu nedenle bölme kriteri daha çok **varyans azaltma** olarak da bilinir.

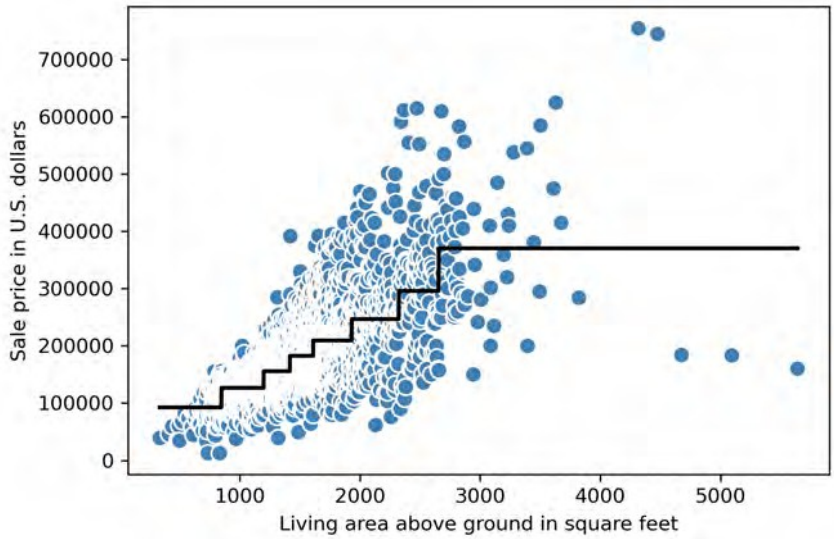
Bir karar ağacının doğru uyumunun neye benzediğini görmek için, SalePrice ve Gr Living Area değişkenleri arasındaki ilişkiyi modellemek üzere scikit-learn'de uygulanan DecisionTreeRegressor'u kullanalım. SalePrice ve Gr Living Area'nın doğrusal olmayan bir ilişkiyi temsil etmediğini unutmayın, ancak bu özellik kombinasyonu yine de bir regresyon ağacının

```
>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['Gr Yaşam Alanı']].values
>> y = df['SalePrice'].values
>> tree = DecisionTreeRegressor(max_depth=3)
>> tree.fit(X, y)
```

genel yönlerini oldukça güzel bir şekilde göstermektedir:

```
>>> sort_idx = X.flatten().argsort()
>> lin_regplot(X[sort_idx], y[sort_idx], ağaç)
>> plt.xlabel('Feet kare cinsinden yer üstündeki yaşam alanı')
>> plt.ylabel('ABD doları cinsinden satış fiyatı')>> p l t . show()
```

Ortaya çıkan grafikte de görebileceğiniz gibi, karar ağacı verilerdeki genel eğilimi yakalamaktadır. Ve bir regresyon ağacının doğrusal olmayan verilerdeki eğilimleri de nispeten iyi yakalayabileceğini hayal edebiliriz. Ancak bu modelin bir sınırlaması, istenen tahminin sürekliliğini ve farklılaşabilirliğini yakalayamamasıdır. Buna ek olarak, verilere aşırı veya yetersiz uyum sağlamamak için ağacın derinliği için uygun bir değer seçme konusunda dikkatli olmamız gerekir; burada üç derinlik iyi bir seçim gibi görünmektedir.




*Şekil 9.15: Bir karar ağacı regresyon grafiği*

Daha derin karar ağaçları denemeniz teşvik edilir. Gr Yaşam Alanı ile SalePrice arasındaki ilişkinin oldukça doğrusal olduğunu unutmayın, bu nedenle karar ağacını bunun yerine Overall Qual değişkenine uygulamanız da önerilir.

Bir sonraki bölümde, regresyon ağaçlarını uydurmanın daha sağlam bir yolunu inceleyeceğiz: rastgele ormanlar.

## Rastgele orman regresyonu

*Bölüm 3*'te öğrendiğiniz gibi, rastgele orman algoritması birden fazla karar ağacını birleştiren bir topluluk tekniğidir. Bir rastgele orman, modelin varyansını azaltmaya yardımcı olan rastgelelik nedeniyle genellikle tek bir karar ağacından daha iyi bir genelleme performansına sahiptir. Rastgele ormanların diğer avantajları, veri kümesindeki aykırı değerlere karşı daha az hassas olmaları ve fazla parametre ayarı gerektirmemeleridir. Rastgele ormanlarda tipik olarak denememiz gereken tek parametre, topluluktaki ağaç sayısıdır. Regresyon için temel rastgele orman algoritması, *Bölüm 3*'te tartıştığımız sınıflandırma için rastgele orman algoritmasıyla neredeyse aynıdır. Tek fark, bireysel karar ağaçlarını büyütmek için MSE kriterini kullanmamız ve tahmin edilen hedef değişkenin tüm karar ağaçlarındaki ortalama tahmin olarak hesaplanmasıdır.

Şimdi, Ames Housing veri *kümesindeki* tüm özellikleri kullanarak örneklerin yüzde 70'*ine* bir rastgele orman regresyon modeli uyduralım ve daha önce *doğrusal regresyon modellerinin* performansını *değerlendirme* bölümünde yaptığımız gibi kalan yüzde 30'daki *performansını değerlendirelim*. Kod

```
>> target = 'SalePrice'
>>> özellikler = df.columns[df.columns != target]
>>> X = df[özellikler].değerler
>> y = df[hedef].değerler
>> X_train, X_test, y_train, y_test = train_test_split(
...         X, y, test_size=0.3, random_state=123)

>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(
...         n_estimators=1000,
...         criterion='squared_error',
...         random_state=1,
...         n_jobs=-1)
>> forest.fit(X_train, y_train)
>> y_train_pred = forest.predict(X_train)
>> y_test_pred = forest.predict(X_test)
>> mae_train = mean_absolute_error(y_train, y_train_pred)
>> mae_test = mean_absolute_error(y_test, y_test_pred)
>> print(f'MAE train: {mae_train:.2f}')
MAE train: 8305.18
>> print(f'MAE testi: {mae_test:.2f}') MAE
testi: 20821.77
>>> r2_train = r2_score(y_train, y_train_pred)
>>> r2_test =r2_score(y_test, y_test_pred)
>> print(f'R^2 train: {r2_train:.2f}')
R^2 train: 0.98
>> print(f'R^2 testi: {r2_test:.2f}')
R^2 testi: 0.85
```

aşağıdaki gibidir:

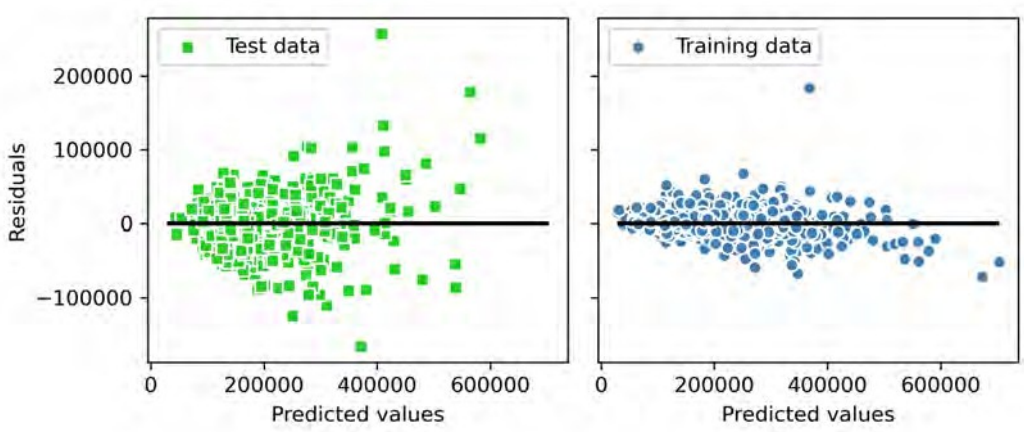
Ne yazık ki, rastgele ormanın eğitim verilerine aşırı uyum sağlama eğiliminde olduğunu görebilirsiniz. Ancak yine de hedef ve açıklayıcı değişkenler arasındaki ilişkiyi nispeten iyi açıklayabilmektedir (test veri setinde $R^2 = 0{,}85$). Karşılaştırma için, önceki bölümdeki doğrusal *regresyon modellerinin performansını değerlendiren ve* aynı veri setine uyan doğrusal model daha az aşırı uyum göstermiş ancak test setinde daha kötü performans göstermiştir ($R^2 = 0{,}75$).

Son olarak, tahminin artıklarına da bir göz atalım:

```
>> x_max = np.max([np.max(y_train_pred), np.max(y_test_pred)])
>>  x_min = np.min([np.min(y_train_pred), np.min(y_test_pred)])

>> fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(7, 3), sharey=True)
```

```
>> ax1.scatter(y_test_pred, y_test_pred - y_test,
...                c='limegreen', marker='s', edgecolor='white',
...                label='Test verileri')
>> ax2.scatter(y_train_pred, y_train_pred - y_train,
...                c='steelblue', marker='o', edgecolor='white',
...                label='Eğitim verileri')
> >   ax1.set_ylabel('Artıklar')

>> for ax in (ax1, ax2):
...        ax.set_xlabel('Tahmin edilen değerler')
...        ax.legend(loc='sol üst')
...        ax.hlines(y=0, xmin=x_min-100, xmax=x_max+100,
...                  color='black', lw=2)

>> plt.tight_layout()
>> plt.show()
```

Daha önce $R^2$ katsayısı ile özetlendiği gibi, $y$ ekseni yönündeki aykırı değerlerin de gösterdiği gibi, modelin eğitim verilerine test verilerinden daha iyi uyduğunu görebilirsiniz. Ayrıca, artıkların dağılımı sıfır merkez noktası etrafında tamamen rastgele görünmemektedir, bu da modelin tüm keşif bilgilerini yakalayamadığını göstermektedir. Bununla birlikte, artık grafiği, bu bölümde daha önce çizdiğimiz doğrusal modelin artık grafiğine göre büyük bir iyileşme göstermektedir.




*Şekil 9.16: Rastgele orman regresyonunun kalıntıları*

 olimpikbarbell@gmail.com 

İdeal olarak, model hatamız rastgele ya da öngörülemez olmalıdır. Başka bir deyişle, tahmin hataları açıklayıcı değişkenlerin içerdiği herhangi bir bilgiyle ilişkili olmamalı, daha ziyade gerçek dünya dağılımlarının veya örüntülerinin rastgeleliğini yansıtmalıdır. Örneğin artık grafiğini inceleyerek tahmin hatalarında örüntüler bulursak, bu artık grafiklerinin tahmine dayalı bilgi içerdiği anlamına gelir. Bunun yaygın bir nedeni, açıklayıcı bilginin bu artıklara sızması olabilir.

Ne yazık ki, kalıntı grafiklerindeki rastgele olmama durumuyla başa çıkmak için evrensel bir yaklaşım yoktur ve deney yapmayı gerektirir. Elimizdeki verilere bağlı olarak, değişkenleri dönüştürerek, öğrenme algoritmasının hiperparametrelerini ayarlayarak, daha basit veya daha karmaşık modeller seçerek, aykırı değerleri kaldırarak veya ek değişkenler ekleyerek modeli iyileştirebiliriz.

# Özet

Bu bölümün başında, tek bir açıklayıcı değişken ile sürekli bir yanıt değişkeni arasındaki ilişkiyi modellemek için basit doğrusal regresyon analizini öğrendiniz. Daha sonra, tahminsel modelleme görevlerinde önemli bir ilk adım olan verilerdeki örüntülere ve anormalliklere bakmak için yararlı bir açıklayıcı veri analizi tekniğinden bahsettik.

İlk modelimizi gradyan tabanlı bir optimizasyon yaklaşımı kullanarak doğrusal regresyon uygulayarak oluşturduk. Daha sonra regresyon için scikit-learn'in doğrusal modellerini nasıl kullanacağınızı ve aykırı değerlerle başa çıkmak için bir yaklaşım olarak sağlam bir regresyon tekniğini (RANSAC) nasıl uygulayacağınızı gördünüz. Regresyon modellerinin tahmin performansını değerlendirmek için ortalama karesel hatalar toplamını ve ilgili $R^2$ metriğini hesapladık. Ayrıca, regresyon modellerinin sorunlarını teşhis etmek için yararlı bir grafiksel yaklaşımı da tartıştık: artık grafiği.

Model karmaşıklığını azaltmak ve aşırı uyumu önlemek için regresyon modellerine düzenli hale getirmenin nasıl uygulanabileceğini keşfettikten sonra, polinom özellik dönüşümü ve rastgele orman regresörleri de dahil olmak üzere doğrusal olmayan ilişkileri modellemek için çeşitli yaklaşımları ele aldık.

Önceki bölümlerde denetimli öğrenme, sınıflandırma ve regresyon analizini ayrıntılı olarak ele aldık. Bir sonraki bölümde, makine öğrenmesinin bir başka ilginç alt alanı olan denetimsiz öğrenme ve hedef değişkenlerin olmadığı durumlarda verilerdeki gizli yapıları bulmak için küme analizinin nasıl kullanılacağı hakkında bilgi edineceğiz.

# Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.com

# 10

# Etiketsiz Verilerle Çalışma - Kümeleme Analizi

Önceki bölümlerde, cevabın zaten bilindiği verileri (eğitim verilerimizde sınıf etiketleri zaten mevcuttu) kullanarak makine öğrenimi modelleri oluşturmak için denetimli öğrenme tekniklerini kullandık. Bu bölümde, vites değiştireceğiz ve doğru cevabı önceden bilmediğimiz verilerdeki gizli yapıları keşfetmemize olanak tanıyan bir **denetimsiz öğrenme** teknikleri kategorisi olan küme analizini keşfedeceğiz. **Kümelemenin** amacı verilerde doğal bir gruplama bulmaktır, böylece aynı kümedeki öğeler farklı kümelerdekilere göre birbirlerine daha çok benzerler.

Keşfedici doğası göz önüne alındığında, kümeleme heyecan verici bir konudur ve bu bölümde, verileri anlamlı yapılar halinde düzenlememize yardımcı olabilecek aşağıdaki kavramlar hakkında bilgi edineceksiniz:

- Popüler **k-ortalamalar** algoritmasını kullanarak benzerlik merkezlerini bulma
- Hiyerarşik kümeleme ağaçları oluşturmak için aşağıdan yukarıya bir yaklaşım benimsemek
- Yoğunluk tabanlı bir kümeleme yaklaşımı kullanarak nesnelerin rastgele şekillerini belirleme

## k- araçlarını kullanarak nesneleri benzerliğe göre gruplama

Bu bölümde, endüstride olduğu kadar akademide de yaygın olarak kullanılan en popüler kümeleme algoritmalarından biri olan k-ortalamalar hakkında bilgi edineceğiz. Kümeleme (veya küme analizi), birbirleriyle diğer gruplardaki nesnelerden daha fazla ilişkili olan benzer nesne gruplarını bulmamızı sağlayan bir tekniktir. Kümelemenin iş odaklı uygulamalarına örnek olarak belgelerin, müziklerin ve filmlerin farklı konulara göre gruplandırılması veya tavsiye motorları için bir temel olarak ortak satın alma davranışlarına dayalı olarak benzer ilgi alanlarını paylaşan müşterilerin bulunması verilebilir.

### scikit-learn kullanarak k-ortalamalar kümeleme

Birazdan göreceğiniz gibi, k-ortalamalar algoritmasının uygulanması son derece kolaydır, ancak aynı zamanda diğer kümeleme algoritmalarına kıyasla genel olarak çok verimlidir, bu da popülaritesini açıklayabilir. K-ortalamalar algoritması **prototip tabanlı kümeleme** kategorisine aittir.

Diğer iki kümeleme kategorisi olan **hiyerarşik** ve **yoğunluk tabanlı kümelemeyi** bu bölümün ilerleyen kısımlarında tartışacağız.

Prototip tabanlı kümeleme, her kümenin bir prototiple temsil edildiği anlamına gelir; bu prototip genellikle ya sürekli özelliklere sahip benzer noktaların centroidi (*ortalaması*) ya da kategorik özellikler söz konusu olduğunda **medoiddir** (en *temsili* veya belirli bir kümeye ait diğer tüm noktalara olan uzaklığı en aza indiren nokta). *K-ortalamalar* küresel bir şekle sahip kümeleri belirlemede çok iyi olsa da, bu kümeleme algoritmasının dezavantajlarından biri, *küme* sayısını *önceden* belirlememiz gerektiğidir. *K* için uygun olmayan bir seçim, kümeleme performansının düşük olmasına neden olabilir. Bu bölümün ilerleyen kısımlarında, optimum *küme* sayısını belirlememize yardımcı olmak üzere bir kümelemenin kalitesini değerlendirmek için yararlı teknikler olan **dirsek** yöntemini ve **siluet grafiklerini** tartışacağız.

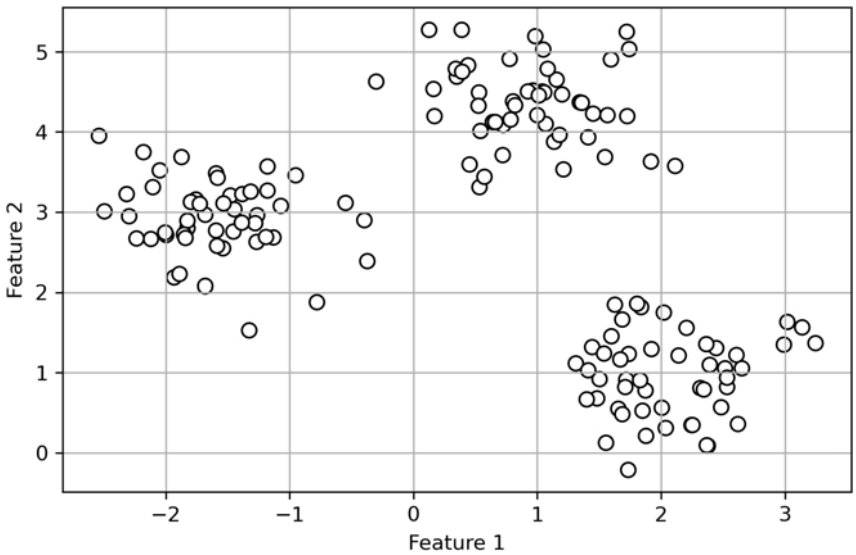
Her ne kadar k-ortalamalar kümelemesi daha yüksek boyutlardaki verilere uygulanabilse de, görselleştirme amacıyla basit bir iki boyutlu veri kümesi kullanarak aşağıdaki örnekleri

```
>> from sklearn.datasets import make_blobs
>> X, y = make_blobs(n_samples=150,
...                   n_özellikler=2,
...                   merkez=3,
...                   cluster_std=0.5,
...                   shuffle=True,
...                   random_state=0)
>> import matplotlib.pyplot as plt
>> plt.scatter(X[:, 0],
...            X[:, 1],
...            c='beyaz',
...            marker='o',
...            edgecolor='siyah',
...            s=50)
>> plt.xlabel('Özellik 1')
>> plt.ylabel('Özellik 2')
>> plt.grid()
>> plt.tight_layout()
>> plt.show()
```

inceleyeceğiz:

olimpikbarbell@gmail.com

Yeni oluşturduğumuz veri kümesi, iki boyutlu bir dağılım grafiği aracılığıyla görselleştirilen, kabaca daha yüksek yoğunluğa sahip üç bölgede gruplandırılmış rastgele oluşturulmuş 150 noktadan oluşmaktadır:




*Şekil 10.1: Etiketsiz veri setimizin dağılım grafiği*

Gerçek dünyadaki kümeleme uygulamalarında, bu örnekler hakkında herhangi bir temel gerçek kategori bilgisine (çıkarımın aksine deneysel kanıt olarak sağlanan oluşum) sahip değiliz; bize sınıf etiketleri verilseydi, bu görev denetimli öğrenme kategorisine girerdi. Bu nedenle, amacımız örnekleri özellik benzerliklerine göre gruplandırmaktır; bu da aşağıdaki dört adımda özetlendiği gibi k-ortalamalar algoritması kullanılarak gerçekleştirilebilir:

1. Başlangıç küme merkezleri olarak örneklerden rastgele *k* merkez noktası seçin
2. Her örneği en yakın merkeze atayın, $\mu^{(jj)}, jj\ j\ \{1,\ldots,k\}$
3. Merkezleri, kendisine atanan örneklerin merkezine taşıyın
4. Küme atamaları değişmeyene veya kullanıcı tanımlı bir toleransa ya da maksimum iterasyon sayısına ulaşılana kadar *2.* ve *3. adımları* tekrarlayın

Şimdi, bir sonraki soru, *nesneler arasındaki benzerliği nasıl ölçeceğimizdir*? Benzerliği uzaklığın tersi olarak tanımlayabiliriz ve sürekli özelliklere sahip örnekleri kümelemek için yaygın olarak kullanılan bir uzaklık, m *boyutlu* uzayda ***x*** ve ***y gibi*** iki nokta arasındaki **kareli Öklid uzaklığıdır:**

$$d(\boldsymbol{xx}\boldsymbol{y})^2 = \sum_{jj1}^{m} (x_{jj} - y_{jj})^2 = \|\boldsymbol{xx} - \boldsymbol{y}\|_2^2$$

Yukarıdaki denklemde j indisinin örnek girdileri olan ***x*** ve ***y'nin*** *j.* boyutunu (özellik sütunu) ifade ettiğine dikkat ediniz. Bu bölümün geri kalanında, *i* ve j üst simgelerini sırasıyla örnek (veri kaydı) indisini ve küme indisini ifade etmek için kullanacağız.

Bu Öklid mesafesi metriğine dayanarak, **k-ortalamalar** algoritmasını basit bir optimizasyon problemi olarak, bazen **küme ataleti** olarak da adlandırılan küme içi **karesel hatalar toplamını (SSE)** en aza indirmeye yönelik iteratif bir yaklaşım olarak tanımlayabiliriz:

$$SSE = \sum_{ij1}^{n} \sum_{iij1}^{kk} ww(iii) \| xx(i) - \mu(ii) \|^2_2$$

Burada, $\mu^{(jj)}$ *j* kümesi için temsili noktadır (centroid). $w^{(i,j)} = 1$ eğer örnek, $\boldsymbol{x}^{(i)}$ , kümede ise *j*, aksi takdirde 0'dır.

$$ww^{(iii)} = \begin{cases} 1i \text{ eğer } xx^{(i)} \in jj \\ 0i \text{ aksi takdirde} \end{cases}$$

Artık basit k-ortalamalar algoritmasının nasıl çalıştığını öğrendiğinize göre, scikit-learn'in cluster modülündeki KMeans sınıfını kullanarak bunu örnek veri kümemize uygulayalım:

```
>> from sklearn.cluster import KMeans
>> km = KMeans(n_clusters=3,
...            init='random',
...            n_init=10,
...            max_iter=300,
...            tol=1e-04,
...            random_state=0)
>>> y_km = km.fit_predict(X)
```

Önceki kodu kullanarak, istenen küme sayısını 3 olarak ayarladık; küme sayısını *önceden* belirtmek zorunda olmak k-ortalamanın sınırlamalarından biridir. k-ortalamalar kümeleme algoritmalarını bağımsız olarak 10 kez çalıştırmak için n_init=10 olarak ayarladık ve son modeli en düşük SSE'ye sahip model olarak seçmek için farklı rastgele merkezler kullandık. max_iter parametresi aracılığıyla her bir çalıştırma için maksimum iterasyon sayısını belirleriz (burada 300). Scikit-learn'deki k-means uygulamasının, maksimum yineleme sayısına ulaşılmadan önce yakınsarsa erken durduğunu unutmayın. Ancak, max_iter için nispeten büyük değerler seçersek, k-means'in belirli bir çalışma için yakınsamaya ulaşmaması mümkündür ve bu sorunlu (hesaplama açısından pahalı) olabilir. Yakınsama sorunlarıyla başa çıkmanın bir yolu, yakınsama ilan etmek için küme içi SSE'deki değişikliklere ilişkin toleransı kontrol eden bir parametre olan tol için daha büyük değerler seçmektir. Önceki kodda, 1e-04 (=0.0001) toleransını seçtik.

K-ortalamalarla ilgili bir sorun, bir veya daha fazla kümenin boş olabilmesidir. Bu sorunun, bu bölümde daha sonra tartışacağımız bir algoritma olan k-medoids veya fuzzy C-means için mevcut olmadığını unutmayın. Ancak, scikit-learn'deki mevcut k-means uygulamasında bu sorun hesaba katılmıştır. Eğer bir küme boşsa, algoritma boş kümenin centroid'inden en uzakta olan örneği arayacaktır. Ardından, bu en uzak noktayı merkez olarak yeniden atayacaktır.

 alinpikbarbell@gmail.com

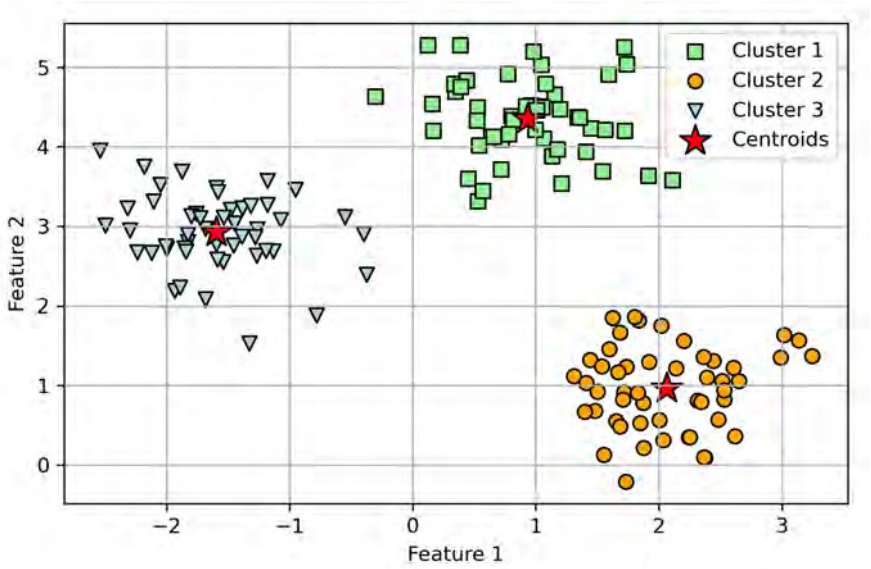
**Özellik ölçeklendirme**

Bir Öklid mesafesi metriği kullanarak gerçek dünya verilerine k-ortalamalar uyguladığımızda, özelliklerin aynı ölçekte ölçüldüğünden emin olmak ve gerekirse z-skor standardizasyonu veya min-max ölçeklendirme uygulamak isteriz.

Küme etiketlerini, y_km, tahmin ettikten ve k-ortalamalar algoritmasının bazı zorluklarını tartıştıktan sonra, şimdi k-ortalamaların veri kümesinde tanımladığı kümeleri küme merkezleriyle birlikte görselleştirelim. Bunlar, takılı KMeans nesnesinin cluster_centers_ niteliği altında saklanır:

```
>> plt.scatter(X[y_km == 0, 0],
...            X[y_km == 0, 1],
...            s=50, c='açık yeşil',
...            marker='s', edgecolor='black',
...            label='Küme 1')
>> plt.scatter(X[y_km == 1, 0],
...            X[y_km == 1, 1],
...            s=50, c='turuncu',
...            marker='o', edgecolor='black',
...            label='Küme 2')
>> plt.scatter(X[y_km == 2, 0],
...            X[y_km == 2, 1],
...            s=50, c='açık mavi',
...            marker='v', edgecolor='black',
...            label='Küme 3')
>> plt.scatter(km.cluster_centers_[:, 0],
...            km.cluster_centers_[:, 1],
...            s=250, marker='*',
...            c='kırmızı', edgecolor='siyah',
...            label='Centroids')
>> plt.xlabel('Özellik 1')
>> plt.ylabel('Özellik 2')
> >  plt.legend(scatterpoints=1)
>> plt.grid()
>> plt.tight_layout()
>> plt.show()
```

*Şekil 10.2'*de, k-ortalamanın her bir kürenin merkezine üç merkezi yerleştirdiğini görebilirsiniz; bu veri kümesi göz önüne alındığında makul bir gruplama gibi görünmektedir:




*Şekil 10.2: k-ortalamalar kümeleri ve merkez noktaları*

Her ne kadar *k-ortalamalar* bu oyuncak veri kümesinde iyi çalışmış olsa da, *küme* sayısını *önceden* belirtmek zorunda olmamızın dezavantajını yaşıyoruz. Gerçek dünya uygulamalarında, özellikle de görselleştirilemeyen daha yüksek boyutlu bir veri kümesiyle çalışıyorsak, seçilecek küme sayısı her zaman bu kadar açık olmayabilir. K-ortalamanın diğer özellikleri kümelerin üst üste binmemesi ve hiyerarşik olmamasıdır ve ayrıca her kümede en az bir öğe olduğunu varsayarız. Bu bölümün ilerleyen kısımlarında, hiyerarşik ve yoğunluk tabanlı kümeleme gibi farklı kümeleme algoritmalarıyla karşılaşacağız. Her iki algoritma türü de küme sayısını önceden belirtmemizi veya veri setimizde küresel yapılar varsaymamızı gerektirmez.

Bir sonraki alt bölümde, **k-means++** adı verilen klasik k-ortalamalar algoritmasının popüler bir varyantını ele alacağız. Bir önceki paragrafta tartışılan k-ortalamalar varsayımlarını ve dezavantajlarını ele almasa da, başlangıç küme merkezlerinin daha akıllıca tohumlanması yoluyla kümeleme sonuçlarını büyük ölçüde iyileştirebilir.

## k-means++ kullanarak ilk küme merkezlerini yerleştirmenin daha akıllı bir yolu

Şimdiye kadar, başlangıç merkezlerini yerleştirmek için rastgele bir tohum kullanan klasik k-ortalamalar algoritmasını tartıştık; bu, başlangıç merkezlerinin kötü seçilmesi durumunda bazen kötü kümelemelere veya yavaş yakınsamaya neden olabilir. Bu sorunu ele almanın bir yolu, k-ortalamalar algoritmasını bir veri kümesi üzerinde birden çok kez çalıştırmak ve SSE açısından en iyi performans gösteren modeli seçmektir.

alinpikbarbell@gmail.com

Bir diğer strateji ise başlangıç merkezlerini k-means++ algoritması aracılığıyla birbirinden uzağa yerleştirmektir; bu da klasik k-means'e göre daha iyi ve tutarlı sonuçlar elde edilmesini sağlar (*k-means++: The Advantag- es of Careful Seeding* by *D. Arthur* and *S. Vassilvitskii* in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms,* pages 1027-1035. *Society for Industrial and Applied Mathematics,* 2007).

k-means++'daki başlatma aşağıdaki gibi özetlenebilir:

1. Seçilen *k* merkezoidi saklamak için boş bir *küme* olan **M'**yi başlatın.
2. Giriş örneklerinden rastgele ilk centroidi, $\mu^{(jj)}$, seçin ve **M'**ye atayın.
3. **M'**de olmayan her örnek için, $\boldsymbol{x}^{(i)}$ **, M'**deki herhangi bir merkeze olan minimum karesel uzaklığı, $d(\boldsymbol{x}^{(i)}, \mathbf{M})^2$ , bulun.
4. Bir sonraki centroidi, $\boldsymbol{\mu}^{(p)}$ , rastgele seçmek için aşağıdakine eşit ağırlıklı bir olasılık dağılımı kullanın $\frac{dd\mu^{(p)}, \mathrm{MM2}}{\sum_i dddd^{(i)}, \mathrm{MM}}$ . Örneğin, bir dizideki tüm noktaları toplarız ve ağırlıklı rastgele bir örneklem seçeriz. öyle ki, karesel mesafe ne kadar büyük olursa, bir noktanın merkez olarak seçilme olasılığı o kadar artar.
5. *k* merkez nokta seçilene kadar *3.* ve *4. adımları* tekrarlayın.
6. Klasik k-ortalamalar algoritması ile devam edin.

scikit-learn'in `KMeans` nesnesi ile k-means++ kullanmak için `init` parametresini `'k-means++'` olarak ayarlamamız yeterlidir. Aslında, `'k-means++'` `init` parametresinin varsayılan argümanıdır ve pratikte şiddetle tavsiye edilir. Önceki örnekte bunu kullanmamamızın tek nedeni, bir kerede çok fazla kavramı tanıtmamaktı. Bu bölümün geri kalanında k-means++ kullanılacaktır, ancak başlangıç küme merkezlerini yerleştirmek için iki farklı yaklaşımı (`init='random'` ile klasik k-means ve `init='k-means++'` ile k-means++) daha fazla denemeniz teşvik edilmektedir.

## Sert ve yumuşak kümeleme

**Sert kümeleme,** bu bölümde daha önce tartıştığımız k-means ve k-means++ algoritmalarında olduğu gibi, bir veri kümesindeki her örneğin tam olarak bir kümeye atandığı bir algoritma ailesini tanımlar. Buna karşılık, **yumuşak kümeleme** algoritmaları (bazen **bulanık kümeleme olarak** da adlandırılır) bir örneği bir veya daha fazla kümeye atar. Yumuşak kümelemenin popüler bir örneği **bulanık C-ortalamalar** (**FCM**) algoritmasıdır (**yumuşak k-ortalamalar** veya **bulanık k-ortalamalar olarak** da adlandırılır). Orijinal fikir, Joseph C. Dunn'ın k-ortalamaları iyileştirmek için bulanık kümelemenin erken bir versiyonunu ilk kez önerdiği 1970'lere dayanmaktadır (*A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters,* 1973). Yaklaşık on yıl sonra James C. Bedzek, günümüzde FCM algoritması olarak bilinen bulanık kümeleme algoritmasının geliştirilmesine yönelik çalışmasını yayınlamıştır (*Pattern Recognition with Fuzzy Objective Function Algorithms*, *Springer Science+Business Media,* 2013).

FCM prosedürü k-ortalamalara çok benzer. Ancak, sabit küme atamasını her bir kümeye ait her bir nokta için olasılıklarla değiştiriyoruz. K-ortalamalar yönteminde, bir *x* örneğinin küme üyeliğini ikili değerlerden oluşan seyrek bir vektörle ifade edebiliriz:

$$x\, x^{\mu(1)} \quad \rightarrow \quad ww^{(iii)} = 0$$

$$[x\ x\ \mu^{(2)} \quad \rightarrow \quad ww^{(iii)} = 1]$$

$$x\ x\ \mu^{(3)} \quad \rightarrow \quad ww^{(iii)} = 0$$

alinpikbarbell@gmail.com

Burada, 1 değerine sahip indeks konumu, örneğin atandığı *küme* merkezini, $\mu^{(jj)}$, gösterir ($k$ = 3, $jj$ {1, 2, 3} olduğu varsayılırsa). Buna karşılık, FCM'deki bir üyelik vektörü aşağıdaki gibi gösterilebilir:

$$x\ x\ \mu^{(1)} \rightarrow ww^{(iiii)} = 0{,}1\ [x$$
$$x\ \mu^{(2)} \rightarrow ww^{(iiii)} = 0{,}85]$$
$$x\ x\ \mu^{(3)} \rightarrow ww^{(iiii)} = 0{,}05$$

Burada her bir değer [0, 1] aralığında yer alır ve ilgili küme merkezinin üyelik olasılığını temsil eder. Belirli bir örnek için üyeliklerin toplamı 1'e eşittir. k-means algoritmasında olduğu gibi, FCM algoritmasını da dört temel adımda özetleyebiliriz:

1. $k$ merkez sayısını belirleyin ve her nokta için küme üyeliklerini rastgele atayın
2. Küme merkezlerini hesaplayın, $\mu^{(jj)}$, $jj$ $j$ {1,... , $k$}
3. Her nokta için küme üyeliklerini güncelleyin
4. Üyelik katsayıları değişmeyene veya kullanıcı tanımlı bir toleransa ya da maksimum iterasyon sayısına ulaşılana kadar *2*. ve *3. adımları* tekrarlayın

FCM'nin amaç fonksiyonu - $J$ olarak kısaltıyoruz$_m$ - k-ortalamada minimize ettiğimiz küme içi SSE'ye çok benziyor:

$$Jm = \sum_{ij1}^{n} \sum_{iij1}^{kk} ww(iii)^m \|xxf - \mu(ii)\|^2 \qquad 2$$

Ancak, üyelik göstergesinin, $w^{(i,j)}$, k-ortalamalarındaki gibi ikili bir değer olmadığını unutmayın ($ww^{(iii)} \in \{0i\ 1\}$), ancak küme üyeliği olasılığını gösteren gerçek bir değerdir ($ww^{(iii)} \in [0i\ 1]$). Ayrıca $w$'ye ek bir üs eklediğimizi fark etmiş olabilirsiniz $^{(i,j)}$ ; $m$ üssü, birden büyük veya eşit herhangi bir sayı (tipik olarak $m$ = 2), **bulanıklık** derecesini kontrol eden **bulanıklık katsayısı** (veya sadece *bulanıklaştırıcı*) olarak adlandırılır.

$m$ değeri ne kadar büyük olursa, küme üyeliği, $w^{(i,j)}$, o kadar küçük olur ve bu da daha bulanık kümelere yol açar. Küme üyeliği olasılığının kendisi aşağıdaki gibi hesaplanır:

$$ww(iii) = \left[\sum_{cc1}^{kk} \left( \frac{\|xx(i) - \mu(ii)\|_2}{\|xx(i) - \mu(c)\|_2} \right)^{\frac{2}{mm1}} \right]^{m1}$$

Örneğin, önceki k-ortalamalar örneğinde olduğu gibi üç küme merkezi seçersek, $\mu^{(jj)}$ kümesine ait $xx^{(i)}$ üyeliğini aşağıdaki gibi hesaplayabiliriz:

$$ww(iii) = \left[\left( \frac{\|xx(i) - \mu(ii)\|_2}{\|xx(i) - \mu(1)\|_2} \right)^{\frac{2}{mm1}} + \left( \frac{\|xx(i) - \mu(ii)\|_2}{\|xx(i) - \mu(2)\|_2} \right)^{\frac{2}{mm1}} + \left( \frac{\|xx(i) - \mu(ii)\|_2}{\|xx(i) - \mu(3)\|_2} \right)^{\frac{2}{mm1}} \right]^{m1}$$

Bir kümenin merkezi, $\mu^{(jj)}$, her örneğin o kümeye ait olma derecesi ile ağırlıklandırılmış tüm örneklerin ortalaması olarak hesaplanır ($ww^{(ii)}{}^{m}$):

$$\mu(jj) = \frac{\sum_{ii1}^{n} ww(iijj)^{m}\, xx(i)}{\sum_{ii1}^{n} ww(iijj)^{m}}$$

Sadece küme üyeliklerini hesaplamak için kullanılan denkleme bakarak, FCM'deki her bir iterasyonun k-ortalamalarındaki bir iterasyondan daha pahalı olduğunu söyleyebiliriz. Öte yandan, FCM yakınsamaya ulaşmak için genellikle daha az iterasyon gerektirir. Bununla birlikte, bir çalışmada açıklandığı üzere, pratikte hem k-ortalamalar hem de FCM'nin çok benzer kümeleme çıktıları ürettiği görülmüştür (*Comparative Analysis of k-means and Fuzzy C-Means Algorithms* by *S. Ghosh* and S. *K. Dubey*, *IJACSA,* 4: 35-38, 2013). Ne yazık ki, FCM algoritması şu anda scikit-learn'de uygulanmamaktadır, ancak ilgilenen okuyucular `https://github. com/scikit-fuzzy/scikit-fuzzy` adresinde bulunan `scikit-fuzzy` paketinden FCM uygulamasını deneyebilirler.

## Optimum küme sayısını bulmak için dirsek yöntemini kullanma

Denetimsiz öğrenmedeki ana zorluklardan biri, kesin cevabı bilmememizdir. Veri setimizde, denetimli bir modelin performansını değerlendirmek için *Bölüm 6*, *Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Öğrenme Uygulamaları'nda* kullandığımız teknikleri uygulamamıza olanak tanıyan temel gerçek sınıf etiketlerine sahip değiliz. Bu nedenle, kümeleme kalitesini ölçmek için, farklı k-ortalamalar kümeleme modellerinin performansını karşılaştırmak üzere küme içi SSE (bozulma) gibi içsel ölçütler kullanmamız gerekir.

Uygun bir şekilde, scikit-learn kullanırken küme içi SSE'yi açıkça hesaplamamız gerekmez, çünkü bir `KMeans` modeli uydurulduktan sonra `inertia_` özelliği aracılığıyla zaten erişilebilir:

```
>> print(f'Distortion: {km.inertia_:.2f}') Distortion:
72.48
```

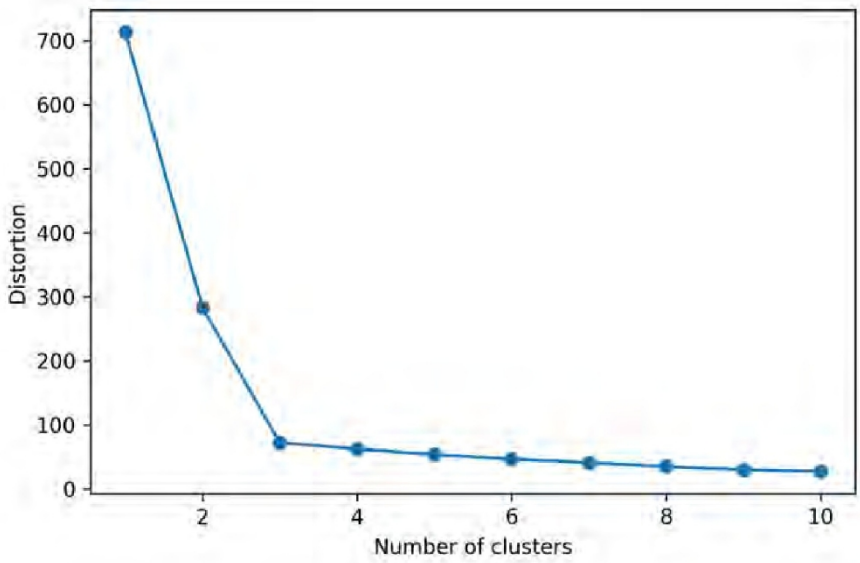
Küme içi SSE'ye dayanarak, belirli bir görev için en uygun *küme* sayısını (*k)* tahmin etmek için **dirsek yöntemi olarak** adlandırılan grafiksel bir araç kullanabiliriz. Eğer *k* artarsa bozulmanın azalacağını söyleyebiliriz. Bunun nedeni, örneklerin atandıkları merkezlere daha yakın olmalarıdır. Dirsek yönteminin arkasındaki fikir, bozulmanın en hızlı şekilde artmaya başladığı *k* değerini belirlemektir; farklı *k* değerleri için bozulmayı çizersek bu daha net hale gelecektir:

```
>>> çarpıtmalar = []
>> for i in range(1, 11):
...     km = KMeans(n_clusters=i,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 random_state=0)
...     km.fit(X)
...     distortions.append(km.inertia_)
```

olimpikbarbell@gmail.com

```
>> plt.plot(range(1,11), distortions, marker='o')
>> plt.xlabel('Küme sayısı')
>>  plt.ylabel('Distortion')
>> plt.tight_layout()
>> plt.show()
```

*Şekil 10.3*'te görebileceğiniz gibi, *dirsek* $k$ = 3'te yer almaktadır, bu nedenle bu, $k$ = 3'ün bu veri kümesi için gerçekten iyi bir seçim olduğuna dair destekleyici bir kanıttır:




*Şekil 10.3: Dirsek yöntemi kullanılarak optimum küme sayısının bulunması*

## Siluet grafikleri aracılığıyla kümeleme kalitesinin ölçülmesi

Bir kümelemenin kalitesini değerlendirmek için bir başka içsel ölçüt de **siluet analizidir** ve bu bölümde daha sonra tartışacağımız k-ortalamalar dışındaki kümeleme algoritmalarına da uygulanabilir. Siluet analizi, kümelerdeki örneklerin ne kadar sıkı gruplandığının bir ölçüsünü çizmek için grafiksel bir araç olarak kullanılabilir. Veri setimizdeki tek bir örneğin **siluet katsayısını** hesaplamak için aşağıdaki üç adımı uygulayabiliriz:

1. **Küme uyumunu**, $a^{(i)}$ , bir örnek, $\boldsymbol{x}^{(i)}$ , ile aynı kümedeki diğer tüm noktalar arasındaki ortalama mesafe olarak hesaplayın.
2. **Küme** ayrımını, $b^{(i)}$ , bir sonraki en yakın kümeden, $\boldsymbol{x}^{(i)}$ örneği ile en yakın kümedeki tüm örnekler arasındaki ortalama mesafe olarak hesaplayın.
3. Silueti, $s^{(i)}$ , burada gösterildiği gibi küme uyumu ve ayrımı arasındaki farkın ikisinden büyük olana bölünmesiyle hesaplayın:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{max\{b^{(i)}\ ,\ a\ \}^{(i)}}$$

Siluet katsayısı -1 ila 1 aralığında sınırlandırılmıştır. Önceki denkleme dayanarak, küme ayrımı ve uyumu eşitse siluet katsayısının 0 olduğunu görebiliriz ($b^{(i)} = a^{(i)}$ ). Ayrıca, $b^{(i)} >> a^{(i)}$ ise ideal siluet katsayısı 1'e yaklaşırız, çünkü $b^{(i)}$ bir örneğin diğer kümelerden ne kadar farklı olduğunu ölçer ve $a^{(i)}$ bize kendi kümesindeki diğer örneklere ne kadar benzer olduğunu söyler.

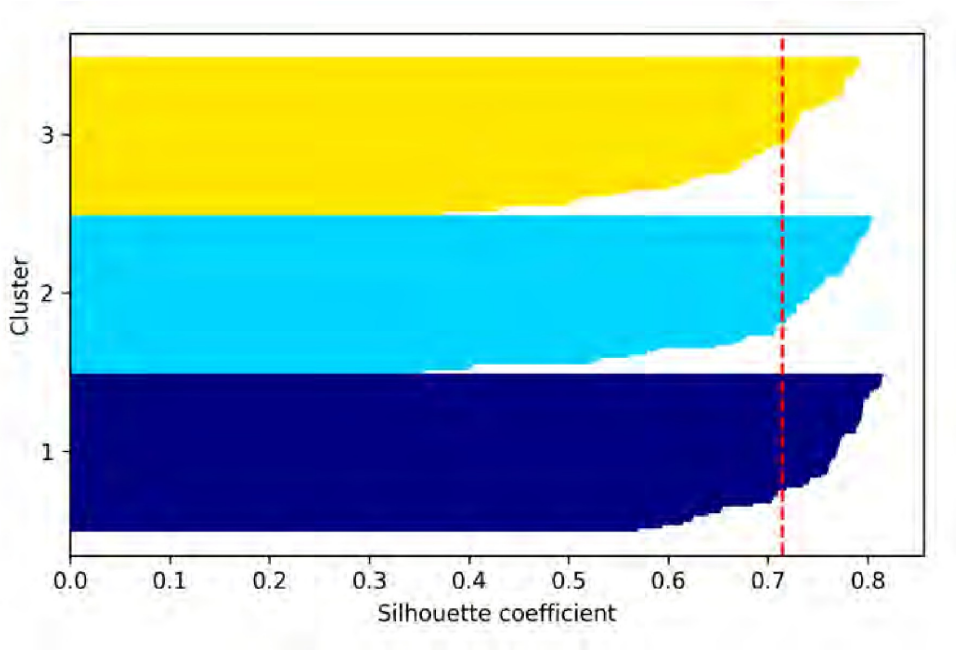
Siluet katsayısı, scikit-learn'in metrik modülünden silhouette_samples olarak mevcuttur ve isteğe bağlı olarak silhouette_scores işlevi kolaylık sağlamak için içe aktarılabilir. silhouette_scores fonksiyonu tüm örneklerdeki ortalama siluet katsayısını hesaplar, bu da numpy. mean(silhouette_samples(...)) ile eşdeğerdir. Aşağıdaki kodu çalıştırarak, şimdi k = 3 ile bir k-ortalamalar kümelemesi için siluet katsayılarının bir grafiğini oluşturacağız:

```
>> km = KMeans(n_clusters=3,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 tol=1e-04,
...                 random_state=0)
>>> y_km = km.fit_predict(X)

>> import numpy as np
>> from matplotlib import cm
>> from sklearn.metrics import silhouette_samples
>> cluster_labels = np.unique(y_km)
>> n_clusters = cluster_labels.shape[0]
>> siluet_vals = siluet_örnekleri(
...       X, y_km, metric='euclidean'
... )
>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>> for i, c in enumerate(cluster_labels):
...       c_siluet_vals = siluet_vals[y_km == c]
...       c_silhouette_vals.sort()
...       y_ax_upper += len(c_silhouette_vals)
...       color = cm.jet(float(i) / n_clusters)
...       plt.barh(range(y_ax_lower, y_ax_upper),
...                c_silhouette_vals,
...                yükseklik=1.0,
...                edgecolor='none',
...                color=color)
...       yticks.append((y_ax_lower + y_ax_upper) / 2.)
...       y_ax_lower += len(c_silhouette_vals)
```

alinpikbarbell@gmail.com

```
>>> siluet_avg = np.mean(siluet_vals)
> >   plt.axvline(silhouette_avg,
...                 color="red",
...                 linestyle="--")
>> plt.yticks(yticks, cluster_labels + 1)
>> plt.ylabel('Küme')
>> plt.xlabel('Siluet katsayısı')
>> plt.tight_layout()
>> plt.show()
```

Siluet grafiğinin görsel olarak incelenmesi yoluyla, farklı kümelerin boyutlarını hızlı bir şekilde inceleyebilir ve *aykırı* değerler içeren kümeleri belirleyebiliriz:




*Şekil 10.4: İyi bir kümelenme örneği için siluet çizimi*

Ancak, önceki siluet grafiğinde de görebileceğiniz gibi, siluet katsayıları 0'a yakın değildir ve bu durumda *iyi bir* kümeleme göstergesi olan ortalama siluet puanından yaklaşık olarak eşit uzaklıktadır. Ayrıca, kümelememizin iyiliğini özetlemek için grafiğe ortalama siluet katsayısını ekledik (noktalı çizgi).

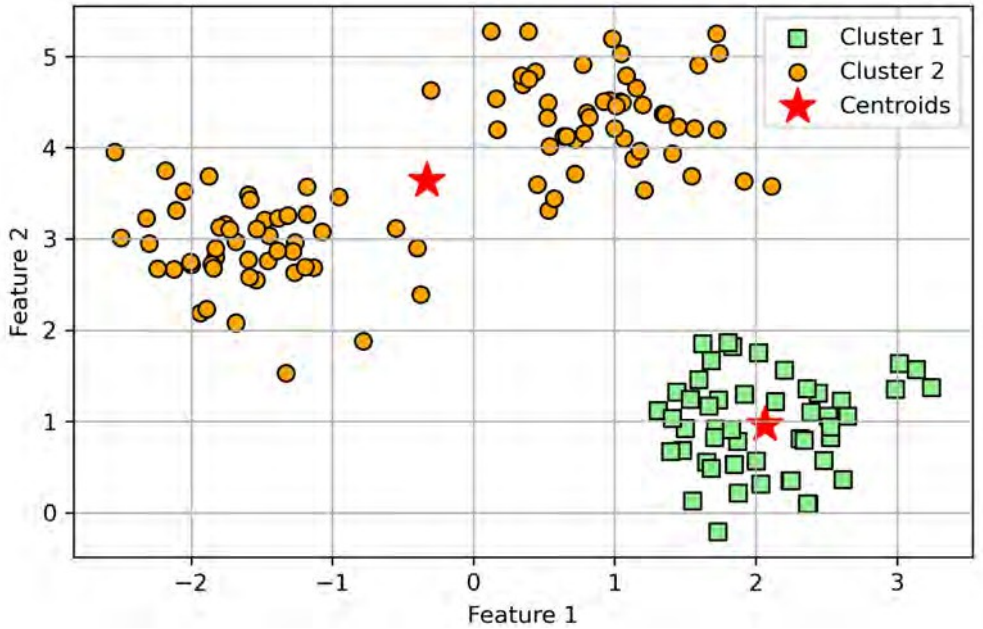
Nispeten *kötü* bir kümeleme için siluet grafiğinin nasıl göründüğünü görmek için, k-ortalamalar algoritmasını yalnızca iki merkezle tohumlayalım:

```
>> km = KMeans(n_clusters=2,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 tol=1e-04,
...                 random_state=0)
>>> y_km = km.fit_predict(X)
>> plt.scatter(X[y_km == 0, 0],
...                 X[y_km == 0, 1],
...                 s=50, c='açık yeşil',
...                 edgecolor='siyah',
...                 marker='s',
...                 label='Küme 1')
>> plt.scatter(X[y_km == 1, 0],
...                 X[y_km == 1, 1],
...                 s=50,
...                 c='turuncu',
...                 edgecolor='siyah',
...                 marker='o',
...                 label='Küme 2')
>> plt.scatter(km.cluster_centers_[:, 0],
...                 km.cluster_centers_[:, 1],
...                 s=250,
...                 marker='*',
...                 c='kırmızı',
...                 label='Centroids')
>> plt.xlabel('Özellik 1')
>> plt.ylabel('Özellik 2')
>> plt.legend()
>> plt.grid()
>> plt.tight_layout()
>> plt.show()
```

*Şekil 10.5*'te görebileceğiniz gibi, merkezlerden biri girdi verilerinin üç küresel gruplamasından ikisinin arasına düşmektedir.

alinpikbarbell@gmail.com

Kümeleme tamamen korkunç görünmese de, optimumun altındadır:



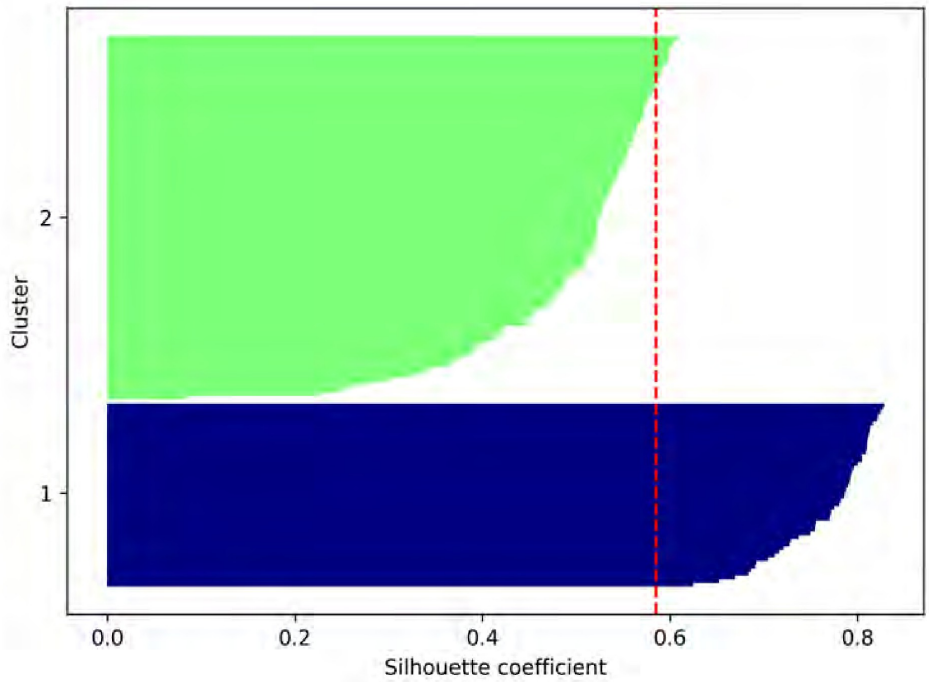

*Şekil 10.5: Optimal olmayan bir kümeleme örneği*

Genellikle daha yüksek boyutlardaki verilerle çalıştığımız için, gerçek dünya problemlerinde veri kümelerini iki boyutlu dağılım grafiklerinde görselleştirme lüksüne sahip olmadığımızı lütfen unutmayın. Bu nedenle, daha sonra sonuçları değerlendirmek için siluet grafiği oluşturacağız:

```
>> cluster_labels = np.unique(y_km)
>> n_clusters = cluster_labels.shape[0]
>> siluet_vals = siluet_örnekleri(
...         X, y_km, metric='euclidean'
... )
>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>> for i, c in enumerate(cluster_labels):
...         c_siluet_vals = siluet_vals[y_km == c]
...         c_silhouette_vals.sort()
...         y_ax_upper += len(c_silhouette_vals)
...         color = cm.jet(float(i) / n_clusters)
...         plt.barh(range(y_ax_lower, y_ax_upper),
...                  c_silhouette_vals,
...                  yükseklik=1.0,
...                  edgecolor='none',
...                  color=color)
```

```
...         yticks.append((y_ax_lower + y_ax_upper) / 2.)
...         y_ax_lower += len(c_silhouette_vals)
>>> siluet_avg = np.mean(siluet_vals)
>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>> plt.yticks(yticks, cluster_labels + 1)
>> plt.ylabel('Küme')
>> plt.xlabel('Siluet katsayısı')
>> plt.tight_layout()
>> plt.show()
```

*Şekil 10.6*'da görebileceğiniz gibi, siluetler artık gözle görülür şekilde farklı uzunluklara ve genişliklere sahiptir; bu da nispeten *kötü* veya en azından *optimal olmayan* bir kümelemenin kanıtıdır:




*Şekil 10.6: Optimal olmayan bir kümeleme örneği için siluet grafiği*

Şimdi, kümelemenin nasıl çalıştığını iyi bir şekilde anladıktan sonra, bir sonraki bölümde k-ortalamalara alternatif bir yaklaşım olarak hiyerarşik kümelemeyi tanıtacağız.

# Kümelerin hiyerarşik olarak düzenlenmesi ağacı

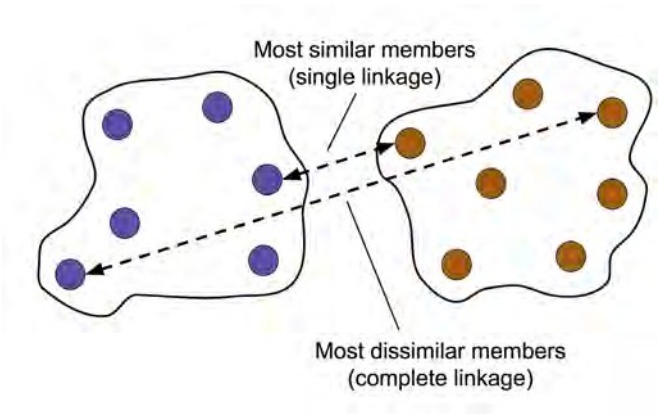
Bu bölümde, prototip tabanlı kümelemeye alternatif bir yaklaşımı inceleyeceğiz: **hiyerarşik kümeleme**. Hiyerarşik kümeleme algoritmasının bir avantajı, anlamlı taksonomiler oluşturarak sonuçların yorumlanmasına yardımcı olabilecek **dendro- gramlar** (ikili hiyerarşik kümelemenin görselleştirmeleri) çizmemize izin vermesidir. Bu hiyerarşik yaklaşımın bir diğer avantajı da küme sayısını önceden belirtmemize gerek olmamasıdır.

alinpikbarbell@gmail.com

Hiyerarşik kümelemeye yönelik iki ana yaklaşım **aglomeratif** ve **bölücü** hiyerarşik kümelemedir. Bölmeli hiyerarşik kümelemede, tüm veri kümesini kapsayan bir küme ile başlarız ve her küme yalnızca bir örnek içerene kadar kümeyi yinelemeli olarak daha küçük kümelere böleriz. Bu bölümde, tam tersi bir yaklaşım benimseyen yığılmalı kümelemeye odaklanacağız. Her örnekle ayrı bir küme olarak başlıyoruz ve sadece bir küme kalana kadar en yakın küme çiftlerini birleştiriyoruz.

## Kümeleri aşağıdan yukarıya doğru gruplama

Aglomeratif hiyerarşik kümeleme için iki standart algoritma **tek bağlantı** ve **tam bağlantıdır**. Tekli bağlantıyı kullanarak, her bir küme çifti için en benzer üyeler arasındaki mesafeleri hesaplar ve en benzer üyeler arasındaki mesafenin en küçük olduğu iki kümeyi birleştiririz. Tam bağlantı yaklaşımı tek bağlantıya benzer, ancak her bir küme çiftindeki en benzer üyeleri karşılaştırmak yerine, birleştirmeyi gerçekleştirmek için en benzemeyen üyeleri karşılaştırırız. Bu *Şekil 10.7*'de gösterilmektedir:




*Şekil 10.7: Tam bağlantı yaklaşımı*

**Alternatif bağlantı türleri**

Aglomeratif hiyerarşik kümeleme için yaygın olarak kullanılan diğer algoritmalar arasında ortalama bağlantı ve Ward's bağlantısı bulunmaktadır. Ortalama bağlantıda, küme çiftlerini iki kümedeki tüm grup üyeleri arasındaki minimum ortalama mesafelere göre birleştiririz. Ward's bağlantısında, toplam küme içi SSE'de minimum artışa yol açan iki küme birleştirilir.

Bu bölümde, tam bağlantı yaklaşımını kullanarak aglomeratif kümelemeye odaklanacağız. Hiyerarşik tam bağlantı kümeleme, aşağıdaki adımlarla özetlenebilecek iteratif bir prosedürdür:

1. Tüm örneklerin çift-bilge uzaklık matrisini hesaplayın.
2. Her veri noktasını tekil bir küme olarak temsil edin.

3. En farklı (uzak) üyeler arasındaki mesafeye göre en yakın iki kümeyi birleştirin.
4. Küme bağlantı matrisini güncelleyin.
5. Tek bir küme kalana kadar *2-4 adımlarını* tekrarlayın.

Daha sonra, mesafe matrisinin nasıl hesaplanacağını tartışacağız (*adım 1*). Ama önce, üzerinde çalışmak için rastgele bir veri örneği oluşturalım. Satırlar farklı gözlemleri (ID'ler 0-4), sütunlar ise bu örneklerin farklı özelliklerini (X, Y, Z) temsil eder:

```
>> import pandas as pd
>> import numpy as np
>> np.random.seed(123)
>>> değişkenler = ['X', 'Y', 'Z']
>>> etiketler = ['ID_0', 'ID_1', 'ID_2', 'ID_3', 'ID_4']
>>> X = np.random.random_sample([5, 3])*10
>> df = pd.DataFrame(X, columns=variables, index=labels)
>> df
```

Önceki kodu yürüttükten sonra, şimdi düzenli olarak oluşturulan örnekleri içeren aşağıdaki DataFrame'i görmeliyiz:

| | X | Y | Z |
|---|---|---|---|
| ID_0 | 6.964692 | 2.861393 | 2.268515 |
| ID_1 | 5.513148 | 7.194690 | 4.231065 |
| ID_2 | 9.807642 | 6.848297 | 4.809319 |
| ID_3 | 3.921175 | 3.431780 | 7.290497 |
| ID_4 | 4.385722 | 0.596779 | 3.980443 |

*Şekil 10.8: Rastgele oluşturulmuş bir veri örneği*

## Bir mesafe matrisi üzerinde hiyerarşik kümeleme gerçekleştirme

Hiyerarşik kümeleme algoritmasının girdisi olarak uzaklık matrisini hesaplamak için SciPy'nin spatial.distance alt modülünden pdist fonksiyonu:

```
>> from scipy.spatial.distance import pdist, squareform
>>> satır_dist = pd.DataFrame(squareform(
...                                pdist(df, metric='euclidean')),
...                                columns=labels, index=labels)
>>> row_dist
```

olinpikbarbell@gmail.com

Önceki kodu kullanarak, X, Y ve Z özelliklerine dayalı olarak veri kümemizdeki her bir girdi örneği çifti arasındaki Öklid mesafesini hesapladık.

Burada gösterildiği gibi, ikili mesafelerin simetrik bir matrisini oluşturmak için kare biçim işlevine girdi olarak pdist tarafından döndürülen yoğunlaştırılmış mesafe matrisini sağladık:

| | ID_0 | ID_1 | ID_2 | ID_3 | ID_4 |
|---|---|---|---|---|---|
| ID_0 | 0.000000 | 4.973534 | 5.516653 | 5.899885 | 3.835396 |
| ID_1 | 4.973534 | 0.000000 | 4.347073 | 5.104311 | 6.698233 |
| ID_2 | 5.516653 | 4.347073 | 0.000000 | 7.244262 | 8.316594 |
| ID_3 | 5.899885 | 5.104311 | 7.244262 | 0.000000 | 4.382864 |
| ID_4 | 3.835396 | 6.698233 | 8.316594 | 4.382864 | 0.000000 |

*Şekil 10.9: Verilerimizin hesaplanan çift-bilge uzaklıkları*

Daha sonra, SciPy'nin cluster.hierarchy alt modülündeki linkage fonksiyonunu kullanarak kümelerimize tam linkage agglomeration uygulayacağız, bu da **linkage matrisi** olarak adlandırılan bir **matris** döndürür.

Ancak, bağlantı işlevini çağırmadan önce, işlev belgelerine dikkatlice bir göz atalım:

```
>> from scipy.cluster.hierarchy import linkage
>> help(linkage)
[...]
Parametreler:
  y : ndarray
    Yoğunlaştırılmış veya gereksiz bir mesafe matrisi.
    Yoğunlaştırılmış bir uzaklık matrisi, uzaklık matrisinin üst
    üçgenini içeren düz bir dizidir. Bu, pdist'in döndürdüğü
    biçimdir. Alternatif olarak, n boyutta m gözlem vektörü
    koleksiyonu m'ye n dizi olarak aktarılabilir.

  yöntem : str, isteğe bağlı
    Kullanılacak bağlantı algoritması. Tam açıklamalar için
    aşağıdaki Bağlantı Yöntemleri bölümüne bakın.

  metric : str, isteğe bağlı
    Kullanılacak mesafe metriği. Geçerli uzaklık ölçütlerinin
    listesi için distance.pdist işlevine bakın.

  İade:
  Z : ndarray
```

```
    Hiyerarşik kümeleme bir bağlantı matrisi olarak
kodlanmıştır. [...]
```

İşlev açıklamasına dayanarak, pdist işlevinden gelen yoğunlaştırılmış bir mesafe matrisini (üçgen başına yukarı) bir giriş niteliği olarak kullanabileceğimizi anlıyoruz. Alternatif olarak, ilk veri dizisini de sağlayabilir ve bağlantıda bir fonksiyon argümanı olarak 'öklid' metriğini kullanabiliriz. Ancak, daha önce tanımladığımız kare biçimli mesafe matrisini kullanmamalıyız, çünkü bu beklenenden farklı mesafe değerleri verecektir. Özetlemek gerekirse, üç olası senaryo burada listelenmiştir:

- **Yanlış yaklaşım**: Aşağıdaki kod parçasında gösterildiği gibi kare biçimli mesafe matrisinin kullanılması yanlış sonuçlara yol açar:

```
>>> row_clusters = linkage(row_dist,
...                        method='complete',
...                        metric='euclidean')
```

- **Doğru yaklaşım**: Aşağıdaki kod örneğinde gösterildiği gibi yoğunlaştırılmış mesafe matrisinin kullanılması doğru bağlantı matrisini verir:

```
>>> row_clusters = linkage(pdist(df, metric='euclidean'),
...                        method='complete')
```

- **Doğru yaklaşım**: Aşağıdaki kod parçasında gösterildiği gibi tam girdi örnek matrisinin (tasarım matrisi olarak adlandırılır) kullanılması da önceki yaklaşıma benzer şekilde doğru bir bağlantı matrisine yol açar:

```
>>> row_clusters = linkage(df.values,
...                        method='complete',
...                        metric='euclidean')
```

Kümeleme sonuçlarına daha yakından bakmak için, bu sonuçları aşağıdaki gibi bir pandas DataFrame'e (en iyi Jupyter not defterinde görüntülenir) dönüştürebiliriz:

```
>> pd.DataFrame(row_clusters,
...             columns=['row label 1',
...                      'satır etiketi 2',
...                      'mesafe',
...                      'kümedeki öğe sayısı'],
...             index=[f'cluster {(i + 1)}' for i in
...                    range(row_clusters.shape[0])])
```

*Şekil 10.10*'da gösterildiği gibi, bağlantı matrisi her bir satırın bir birleşmeyi temsil ettiği birkaç satırdan oluşur. Birinci ve ikinci sütunlar her bir kümedeki en farklı üyeleri gösterir ve üçüncü sütun bu üyeler arasındaki mesafeyi bildirir.

alinpikbarbell@gmail.com

Son sütun, her bir kümedeki üyelerin sayısını verir:

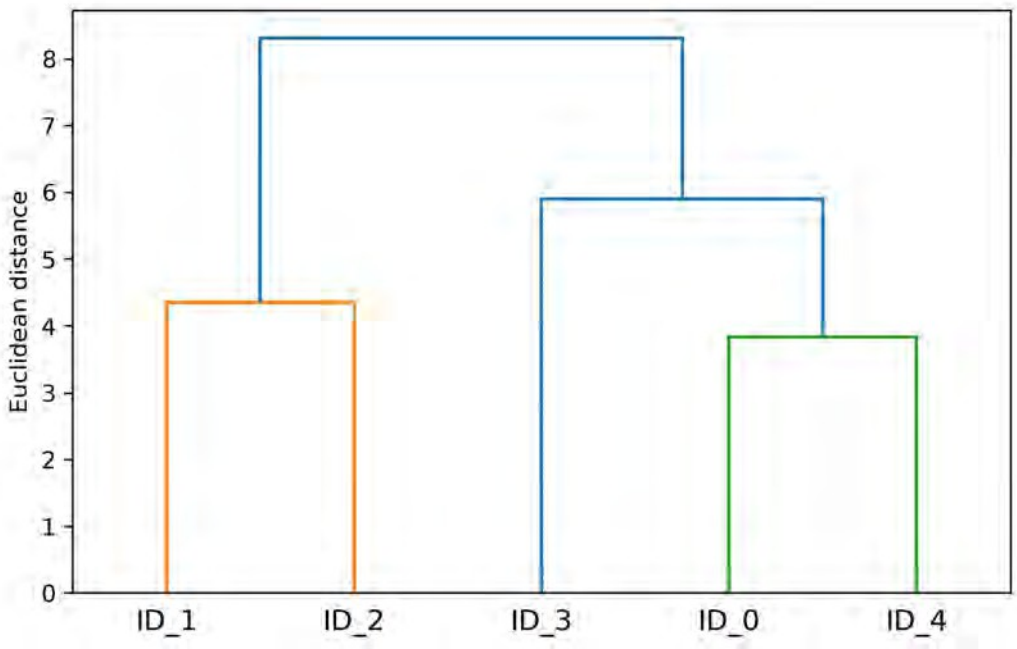
| | row label 1 | row label 2 | distance | no. of items in clust. |
|---|---|---|---|---|
| cluster 1 | 0.0 | 4.0 | 3.835396 | 2.0 |
| cluster 2 | 1.0 | 2.0 | 4.347073 | 2.0 |
| cluster 3 | 3.0 | 5.0 | 5.899885 | 3.0 |
| cluster 4 | 6.0 | 7.0 | 8.316594 | 5.0 |

*Şekil 10.10: Bağlantı matrisi*

Artık bağlantı matrisini hesapladığımıza göre, sonuçları bir dendro- gram şeklinde görselleştirebiliriz:

```
>> from scipy.cluster.hierarchy import dendrogram
>> # dendrogramı siyah yap (bölüm 1/2)
>> # from scipy.cluster.hierarchy import set_link_color_palette
> >   # set_link_color_palette(['black'])
>> satır_dendr = dendrogram(
...         row_clusters,
...         labels=labels,
...         # dendrogramı siyah yap (bölüm 2/2)
...         # color_threshold=np.inf
... )
>> plt.tight_layout()
>> plt.ylabel('Öklid uzaklığı')
>> plt.show()
```

Önceki kodu çalıştırıyorsanız veya bu kitabın bir e-kitap versiyonunu okuyorsanız, ortaya çıkan dendrogramdaki dalların farklı renklerde gösterildiğini fark edeceksiniz. Renk şeması, dendrogramdaki mesafe eşikleri için çevrilen bir Matplotlib renkleri listesinden türetilmiştir. Örneğin, dendrogramları siyah renkte görüntülemek için, önceki koda eklenen ilgili bölümleri kaldırabilirsiniz:




*Şekil 10.11: Verilerimizin bir dendrogramı*

Böyle bir dendrogram, aglomeratif hiyerarşik kümeleme sırasında oluşan farklı kümeleri özetler; örneğin, ID_0 ve ID_4 örneklerinin ardından ID_1 ve ID_2 örneklerinin Öklid mesafe metriğine göre en benzer olanlar olduğunu görebilirsiniz.

## Dendrogramları bir ısı haritasına ekleme

Pratik uygulamalarda, hiyerarşik kümeleme dendrogramları genellikle eğitim örneklerimizi içeren veri dizisi veya matrisindeki tek tek değerleri bir renk koduyla temsil etmemizi sağlayan bir **ısı haritasıyla** birlikte kullanılır. Bu bölümde, bir ısı haritası grafiğine bir dendrogramın nasıl ekleneceğini ve ısı haritasındaki satırların buna göre nasıl sıralanacağını tartışacağız.

 alinpikbarbell@gmail.com 

Bununla birlikte, bir ısı haritasına dendrogram eklemek biraz zor olabilir, bu nedenle bu prosedürü adım adım inceleyelim:

1. Yeni bir şekil nesnesi oluşturuyoruz ve add_axes niteliği aracılığıyla dendrogramın $x$ ekseni konumunu, $y$ ekseni konumunu, genişliğini ve yüksekliğini tanımlıyoruz. Ayrıca, dendrogramı saat yönünün tersine 90 derece döndürüyoruz. Kod aşağıdaki gibidir:

```
>>> fig = plt.figure(figsize=(8, 8), facecolor='white')
>> axd = fig.add_axes([0.09, 0 . 1, 0 . 2, 0 . 6])
>>> row_dendr = dendrogram(row_clusters,
...                                orientation='left')
>> # not: matplotlib < v1.5.1 için lütfen şunu kullanın
> >  # orientation='right'
```

2. Ardından, ilk DataFrame'imizdeki verileri, esasen bir Python sözlüğü olan dendrogram nesnesinden yapraklar anahtarı aracılığıyla erişilebilen kümeleme etiketlerine göre yeniden sıralıyoruz. Kod aşağıdaki gibidir:

```
>> df_rowclust = d f .iloc[row_dendr['leaves'][::-1]]
```

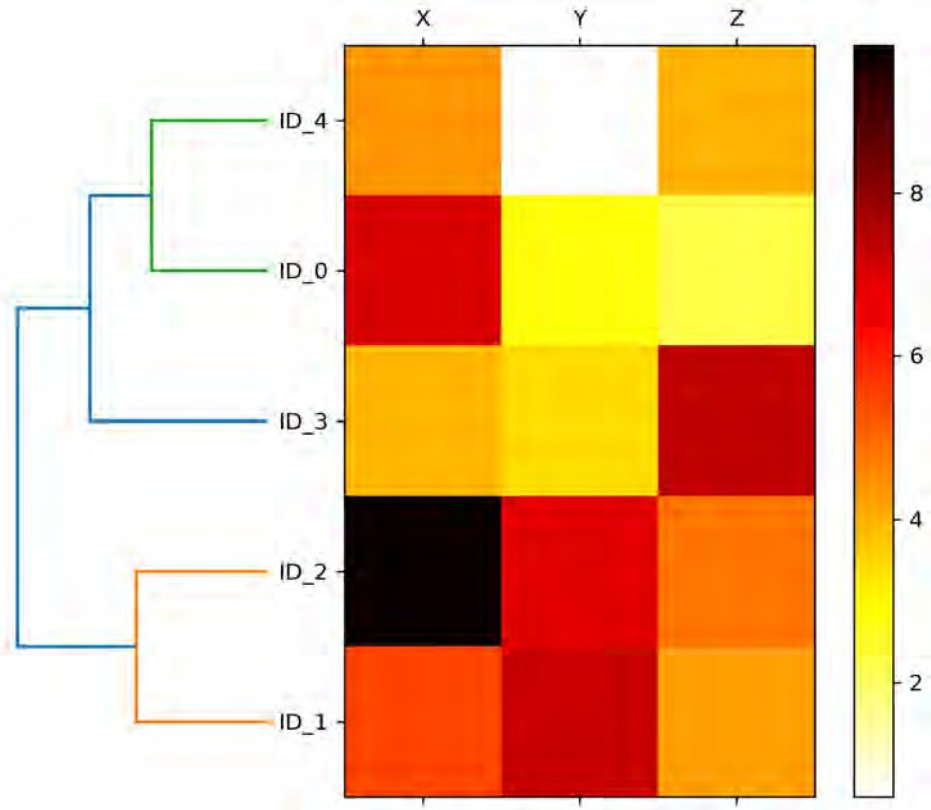
3. Şimdi, yeniden sıralanan DataFrame'den ısı haritasını oluşturuyoruz ve dendrogramın yanına yerleştiriyoruz:

```
>> axm = fig.add_axes([0.23, 0 . 1, 0 . 6, 0 . 6])
>>> cax = axm.matshow(df_rowclust,
...                          interpolasyon='en yakın',
...                          cmap='hot_r')
```

4. Son olarak, eksen işaretlerini kaldırarak ve eksen dikenlerini gizleyerek dendrogramın estetiğini değiştiriyoruz. Ayrıca, bir renk çubuğu ekliyoruz ve sırasıyla $x$ ve $y$ ekseni tik etiketlerine özellik ve veri kaydı adlarını atıyoruz:

```
>> axd.set_xticks([])
>> axd.set_yticks([])
>> for i in axd.spines.values():
...        i.set_visible(False)
>>> fig.colorbar(cax)
>> axm.set_xticklabels(['] + list(df_rowclust.columns))
>> axm.set_yticklabels(['] + list(df_rowclust.index))
>> plt.show()
```

Önceki adımları izledikten sonra, ısı haritası ekli dendrogramla birlikte görüntülenmelidir:




*Şekil 10.12: Verilerimizin ısı haritası ve dendrogramı*

Gördüğünüz gibi, ısı haritasındaki satırların sırası den- drogramdaki örneklerin kümelenmesini yansıtmaktadır. Basit bir dendrograma ek olarak, ısı haritasındaki her örneğin ve özelliğin renk kodlu değerleri bize veri kümesinin güzel bir özetini sunar.

## Scikit-learn aracılığıyla aglomeratif kümeleme uygulamak

Önceki alt bölümde, SciPy kullanarak aglomeratif hiyerarşik kümelemenin nasıl yapıldığını gördünüz. Bununla birlikte, scikit-learn'de döndürmek istediğimiz küme sayısını seçmemize izin veren bir AgglomerativeClustering uygulaması da vardır. Bu, hiyerarşik küme ağacını budamak istediğimizde kullanışlıdır.

alinpikbarbell@gmail.com

n_cluster parametresini 3 olarak ayarlayarak, şimdi giriş örneklerini daha önce olduğu gibi Öklid mesafesi metriğine dayalı aynı tam bağlantı yaklaşımını kullanarak üç grupta kümeleyeceğiz:

```
>> from sklearn.cluster import AgglomerativeClustering
>>  ac = AgglomerativeClustering(n_clusters=3,
...                                 affinity='euclidean',
...                                 linkage='complete')
>>> etiketler = ac.fit_predict(X)
>> print(f'Küme etiketleri: {etiketler}')
Küme etiketleri: [1 0 0 2 1]
```

Tahmin edilen küme etiketlerine baktığımızda, birinci ve beşinci örneklerin (ID_0 ve ID_4) bir kümeye (etiket 1), ID_1 ve ID_2 örneklerinin ise ikinci bir kümeye (etiket 0) atandığını görebiliriz. ID_3 örneği kendi kümesine (etiket 2) yerleştirilmiştir. Genel olarak sonuçlar, dendrogramda gözlemlediğimiz sonuçlarla tutarlıdır. Bununla birlikte, ID_3'ün ID_1 ve ID_2'ye kıyasla ID_4 ve ID_0'a daha çok benzediğini belirtmeliyiz; önceki dendrogram şeklinde gösterildiği gibi, bu scikit-learn'in kümeleme sonuçlarından anlaşılmamaktadır. Şimdi aşağıdaki kod parçacığında n_cluster=2 kullanarak AgglomerativeClustering'i yeniden çalıştıralım:

```
>>  ac = AgglomerativeClustering(n_clusters=2,
...                                 affinity='euclidean',
...                                 linkage='complete')
>>> etiketler = ac.fit_predict(X)
>> print(f'Küme etiketleri: {etiketler}')
Küme etiketleri: [0 1 1 0 0 ]
```

Gördüğünüz gibi, bu *budanmış* kümeleme hiyerarşisinde ID_3 etiketi ID_0 ile aynı kümeye atanmıştır ve ID_4, beklendiği gibi.

# DBSCAN aracılığıyla yüksek yoğunluklu bölgelerin bulunması

Bu bölümde çok sayıda farklı kümeleme algoritmasını ele alamasak **da**, en azından bir kümeleme yaklaşımına daha yer verelim: k-ortalamalar gibi küresel kümeler hakkında varsayımlarda bulunmayan veya veri kümesini manuel bir kesme noktası gerektiren hiyerarşilere ayırmayan **gürültülü uygulamaların yoğunluk tabanlı uzamsal kümelenmesi** (**DBSCAN**). Adından da anlaşılacağı üzere, yoğunluk tabanlı kümeleme, noktaların yoğun olduğu bölgelere dayalı olarak küme etiketleri atar. DBSCAN'de yoğunluk kavramı, belirli bir yarıçap, $\varepsilon\varepsilon$ içindeki noktaların sayısı olarak tanımlanır.

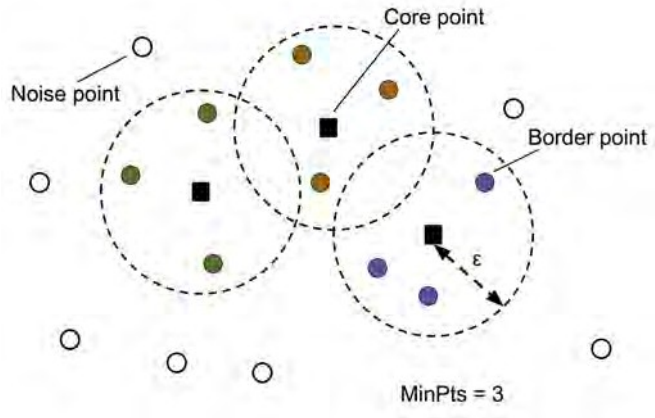
DBSCAN algoritmasına göre, aşağıdaki kriterler kullanılarak her örneğe (veri noktası) özel bir etiket atanır:

- Bir nokta, komşu noktaların en az belirli bir sayısı (MinPts) belirtilen yarıçap, $\varepsilon\varepsilon$ içinde kalıyorsa **çekirdek nokta olarak** kabul edilir.
- Bir **sınır noktası,** $\varepsilon\varepsilon$ içinde MinPts'den daha az komşusu olan, ancak $\varepsilon\varepsilon$ içinde yer alan bir noktadır. bir çekirdek noktasının yarıçapı
- Ne çekirdek ne de sınır noktası olan diğer tüm noktalar **gürültü noktası olarak** kabul edilir

Noktaları çekirdek, sınır veya gürültü olarak etiketledikten sonra, DBSCAN algoritması iki basit adımda özetlenebilir:

1. Her bir çekirdek nokta veya bağlantılı çekirdek nokta grubu için ayrı bir küme oluşturun. (Çekirdek noktalar $\varepsilon$'den daha uzak değillerse bağlantılıdırlar).
2. Her sınır noktasını karşılık gelen çekirdek noktasının kümesine atayın.

Uygulamaya geçmeden önce DBSCAN sonucunun neye benzeyebileceğini daha iyi anlamak için *Şekil 10.13*'te çekirdek noktaları, sınır noktaları ve gürültü noktaları hakkında öğrendiklerimizi özetleyelim:




*Şekil 10.13: DBSCAN için çekirdek, gürültü ve sınır noktaları*

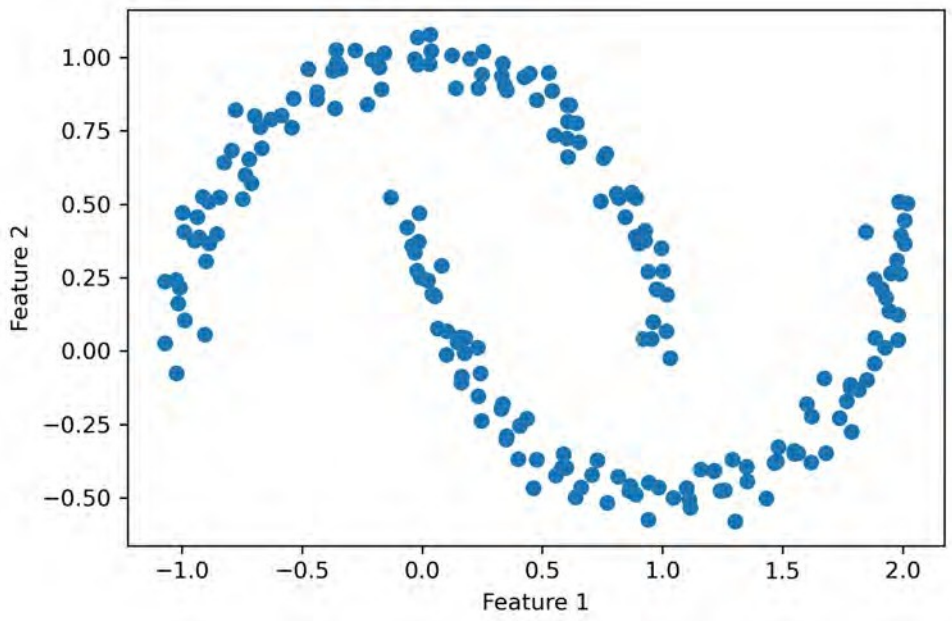
DBSCAN kullanmanın temel avantajlarından biri, kümelerin k-ortalamalarındaki gibi küresel bir şekle sahip olduğunu varsaymamasıdır. Ayrıca DBSCAN, k-ortalamalar ve hiyerarşik kümelemeden farklı olarak her noktayı bir kümeye atamak zorunda değildir, ancak gürültü noktalarını kaldırabilir.

Daha açıklayıcı bir örnek için, k-ortalamalar kümelemesi, hiyerarşik kümeleme ve DBSCAN'ı karşılaştırmak üzere yarım ay şeklindeki yapılardan oluşan yeni bir veri kümesi oluşturalım:

```
>> from sklearn.datasets import make_moons
>> X, y = make_moons(n_samples=200,
...                   gürültü=0.05,
...                   random_state=0)
>> plt.scatter(X[:, 0], X[:, 1])
>> plt.xlabel('Özellik 1')
>> plt.ylabel('Özellik 2')
>> plt.tight_layout()
>> plt.show()
```

 olimpikbarbell@gmail.com 

Ortaya çıkan grafikte görebileceğiniz gibi, her biri 100 örnekten (veri noktası) oluşan iki görünür, yarım ay şeklinde grup vardır:




*Şekil 10.14: İki özellikli yarım ay şeklinde bir veri kümesi*

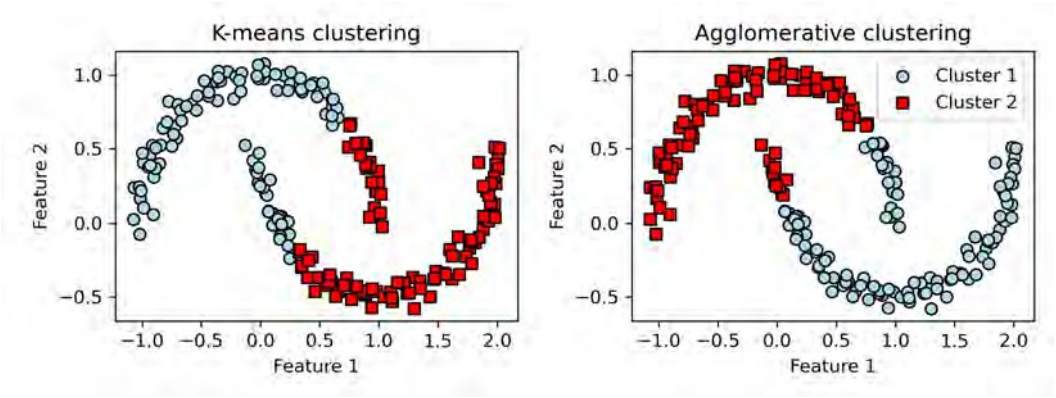
Daha önce tartışılan kümeleme algoritmalarından birinin yarım ay şekillerini ayrı kümeler olarak başarılı bir şekilde tanımlayıp tanımlayamayacağını görmek için k-ortalamalar algoritmasını ve tam bağlantı kümelemesini kullanarak başlayacağız. Kod aşağıdaki gibidir:

```
>> f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
>> km = KMeans(n_clusters=2,
...            random_state=0)
>>> y_km = km.fit_predict(X)
>> ax1.scatter(X[y_km == 0, 0],
...            X[y_km == 0, 1],
...            c='lightblue',
...            edgecolor='siyah',
...            marker='o',
...            s=40,
...            label='küme 1')
```

```
>> ax1.scatter(X[y_km == 1, 0],
...                 X[y_km == 1, 1],
...                 c='kırmızı',
...                 edgecolor='siyah',
...                 marker='s',
...                 s=40,
...                 label='küme 2')
>> ax1.set_title('K-ortalamalar kümeleme')
>> ax1.set_xlabel('Özellik 1')
>> ax1.set_ylabel('Özellik 2')

>>  ac = AgglomerativeClustering(n_clusters=2,
...                                 affinity='euclidean',
...                                 linkage='complete')
>> y_ac = ac.fit_predict(X)
>> ax2.scatter(X[y_ac == 0, 0],
...                 X[y_ac == 0, 1],
...                 c='lightblue',
...                 edgecolor='siyah',
...                 marker='o',
...                 s=40,
...                 label='Küme 1')
>> ax2.scatter(X[y_ac == 1, 0],
...                 X[y_ac == 1, 1],
...                 c='kırmızı',
...                 edgecolor='siyah',
...                 marker='s',
...                 s=40,
...                 label='Küme 2')
>> ax2.set_title('Agglomerative clustering')
>> ax2.set_xlabel('Özellik 1')
>> ax2.set_ylabel('Özellik 2')
>> plt.legend()
>> plt.tight_layout()
>> plt.show()
```

alinpikbarbell@gmail.com

Görselleştirilmiş kümeleme sonuçlarına dayanarak, k-ortalamalar algoritmasının iki kümeyi ayıramadığını ve ayrıca hiyerarşik kümeleme algoritmasının bu karmaşık şekiller tarafından zorlandığını görebiliriz:

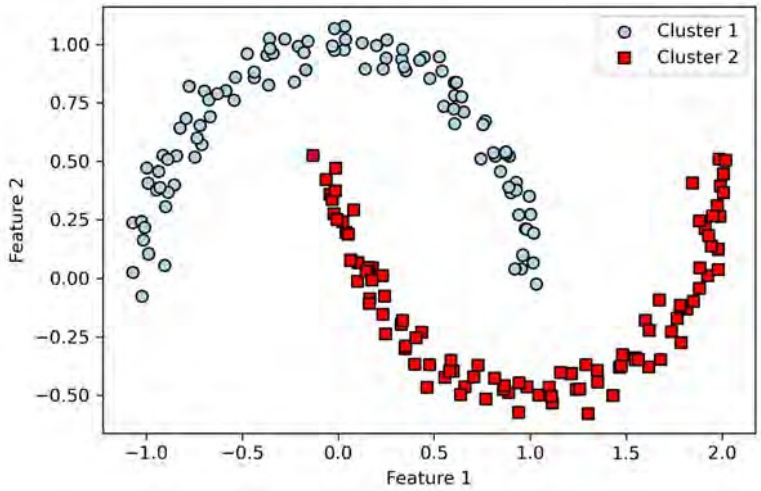



*Şekil 10.15: Yarım ay şeklindeki veri kümesinde k-ortalamalar ve aglomeratif kümeleme*

Son olarak, yoğunluk tabanlı bir yaklaşım kullanarak iki yarım ay şeklindeki kümeyi bulup bulamayacağını görmek için DBSCAN algoritmasını bu veri kümesi üzerinde deneyelim:

```
>> from sklearn.cluster import DBSCAN
>> db = DBSCAN(eps=0.2,
...             min_samples=5,
...             metric='euclidean')
>> y_db = db.fit_predict(X)
>> plt.scatter(X[y_db == 0, 0],
...            X[y_db == 0, 1],
...            c='lightblue',
...            edgecolor='siyah',
...            marker='o',
...            s=40,
...            label='Küme 1')
>> plt.scatter(X[y_db == 1, 0],
...            X[y_db == 1, 1],
...            c='kırmızı',
...            edgecolor='siyah',
...            marker='s',
...            s=40,
...            label='Küme 2')
>> plt.xlabel('Özellik 1')
>> plt.ylabel('Özellik 2')
>> plt.legend()
>> plt.tight_layout()
>> plt.show()
```

DBSCAN algoritması yarım ay şekillerini başarılı bir şekilde tespit edebilir, bu da DBSCAN'ın güçlü yönlerinden birini vurgular - rastgele şekillerdeki verileri kümeleme:




*Şekil 10.16: Yarım ay şeklindeki veri kümesi üzerinde DBSCAN kümeleme*

Bununla birlikte, DBSCAN'ın bazı dezavantajlarına da dikkat etmeliyiz. Veri setimizdeki özellik sayısı arttıkça -sabit sayıda eğitim örneği **olduğunu** varsayarsak- **boyutluluk lanetinin** olumsuz etkisi artar. Bu, özellikle Öklid uzaklık metriğini kullanıyorsak bir sorundur. Bununla birlikte, boyutluluk laneti sorunu DBSCAN'e özgü değildir: Öklid mesafe metriğini kullanan diğer kümeleme algoritmalarını da etkiler, örneğin k-ortalamalar ve hiyerarşik kümeleme algoritmaları. Buna ek olarak, DBSCAN'da iyi kümeleme sonuçları elde etmek için optimize edilmesi gereken iki hiper parametre (MinPts ve $\varepsilon\varepsilon$ ) vardır. MinPts ve $\varepsilon\varepsilon$'nin iyi bir kombinasyonunu bulmak

$\varepsilon\varepsilon$, veri kümesindeki yoğunluk farklılıkları nispeten büyükse sorunlu olabilir.

**Grafik tabanlı kümeleme**

Şimdiye kadar, kümeleme algoritmalarının en temel üç kategorisini gördük: k-ortalamalar ile prototip tabanlı kümeleme, aglomeratif hiyerarşik kümeleme ve DBSCAN aracılığıyla den- sity tabanlı kümeleme. Bununla birlikte, bu bölümde ele almadığımız daha gelişmiş kümeleme algoritmalarının dördüncü bir sınıfı da vardır: çizge tabanlı kümeleme. Çizge tabanlı kümeleme ailesinin muhtemelen en önde gelen üyeleri özel kümeleme algoritmalarıdır.

Spektral kümelemenin birçok farklı uygulaması olmasına rağmen, hepsinin ortak noktası, küme ilişkilerini türetmek için bir benzerlik veya uzaklık matrisinin özvektörlerini kullanmalarıdır. Spektral kümeleme bu kitabın kapsamı dışında olduğundan, bu konu hakkında daha fazla bilgi edinmek için Ulrike von Luxburg'un mükemmel eğitimini okuyabilirsiniz (*A tutorial on spectral clustering*, *Statistics and Computing,* 17(4): 395-416, 2007). Ücretsiz olarak http://arxiv.org/pdf/0711.0189v1.pdf adresindeki arXiv'den temin edilebilir.

 olimpikbarbell@gmail.com 

Uygulamada, belirli bir veri kümesinde hangi kümeleme algoritmasının en iyi performansı göstereceğinin her zaman açık olmadığını unutmayın, özellikle de veriler görselleştirmeyi zorlaştıran veya imkansız kılan birden fazla boyutta geliyorsa. Ayrıca, başarılı bir kümelemenin yalnızca algoritmaya ve hiperparametrelerine bağlı olmadığını vurgulamak önemlidir; daha ziyade, uygun bir mesafe metriğinin seçimi ve deneysel kurulumu yönlendirmeye yardımcı olabilecek alan bilgisinin kullanımı daha da önemli olabilir.

Bu nedenle, boyutluluk laneti bağlamında, kümeleme yapmadan önce boyutluluk azaltma tekniklerini uygulamak yaygın bir uygulamadır. Denetimsiz veri kümeleri için bu tür boyutluluk azaltma teknikleri, *Bölüm 5, Boyutluluk Azaltma Yoluyla Verileri Sıkıştırma'*da ele aldığımız temel bileşen analizi ve t-SNE'yi içerir. Ayrıca, veri kümelerini iki boyutlu alt uzaylara sıkıştırmak özellikle yaygındır, bu da sonuçları değerlendirmek için özellikle yararlı olan iki boyutlu dağılım grafikleri kullanarak kümeleri ve atanan etiketleri görselleştirmemize olanak tanır.

# Özet

Bu bölümde, verilerdeki gizli yapıların veya bilgilerin keşfedilmesinde bize yardımcı olabilecek üç farklı kümeleme algoritması hakkında bilgi edindiniz. Prototip tabanlı bir yaklaşım olan k-ortalamalar ile başladık, bu yaklaşım örnekleri belirli sayıda küme merkezini temel alarak küresel şekillerde kümeler. Kümeleme denetimsiz bir yöntem olduğundan, bir modelin performansını değerlendirmek için temel gerçek etiketlerinin lüksüne sahip değiliz. Bu nedenle, kümeleme kalitesini ölçmek için dirsek yöntemi veya siluet analizi gibi içsel performans ölçütlerini kullandık.

Daha sonra kümelemeye yönelik farklı bir yaklaşımı inceledik: aglomeratif hiyerarşik kümeleme. Hiyerarşik kümeleme, küme sayısının önceden belirtilmesini gerektirmez ve sonuç, sonuçların yorumlanmasına yardımcı olabilecek bir dendrogram gösteriminde görselleştirilebilir. Bu bölümde ele aldığımız son kümeleme algoritması, noktaları yerel yoğunluklara göre gruplayan ve aykırı değerleri işleyebilen ve küresel olmayan şekilleri tanımlayabilen bir algoritma olan DBSCAN idi.

Denetimsiz öğrenme alanındaki bu gezintiden sonra, şimdi denetimli öğrenme için en heyecan verici makine öğrenimi algoritmalarından bazılarını tanıtmanın zamanı geldi: çok katmanlı yapay sinir ağları. Son zamanlarda yeniden canlanan sinir ağları, makine öğrenimi araştırmalarında bir kez daha en sıcak konu haline geldi. Yakın zamanda geliştirilen derin öğrenme algoritmaları sayesinde, sinir ağları görüntü sınıflandırma, doğal dil işleme ve konuşma tanıma gibi birçok karmaşık görev için son teknoloji olarak kabul edilmektedir. *Bölüm 11, Sıfırdan Çok Katmanlı Yapay Sinir Ağı* Uygulaması'nda kendi çok katmanlı sinir ağımızı oluşturacağız. *Bölüm 12'*de, *PyTorch ile Sinir Ağı Eğitimini Paralelleştirme, sinir ağı* eğitimi konusunda uzmanlaşmış PyTorch kütüphanesi ile çalışacağız.
grafik işlem birimlerini kullanarak çok katmanlı modelleri çok verimli bir şekilde kullanabilir.

# Kitabımızın Discord alanına katılın

Benzer düşünen insanlarla tanışmak ve 2000'den fazla üyemizle birlikte öğrenmek için Discord topluluğumuza katılın:
https://packt.link/MLwPyTorch

6587191CA584E
olimpikbarbell@gmail.com


# 11

# Sıfırdan Çok Katmanlı Yapay Sinir Ağının Uygulanması

Bildiğiniz gibi, derin öğrenme basının büyük ilgisini çekiyor ve şüphesiz makine öğrenimi alanındaki en sıcak konu. Derin öğrenme, çok katmanlı yapay **sinir ağlarını** (**NN**) verimli bir şekilde eğitmekle ilgilenen makine öğrenmesinin bir alt alanı olarak anlaşılabilir. Bu bölümde, yapay NN'lerin temel kavramlarını öğreneceksiniz, böylece Python tabanlı gelişmiş derin öğrenme kütüphanelerini ve özellikle görüntü ve metin analizleri için çok uygun olan **derin sinir ağı** (**DNN**) mimarilerini tanıtacak olan sonraki bölümler için iyi bir donanıma sahip olacaksınız.

Bu bölümde ele alacağımız konular aşağıdaki gibidir:

- Çok katmanlı NN'ler hakkında kavramsal bir anlayış kazanma
- NN eğitimi için temel geri yayılım algoritmasının sıfırdan uygulanması
- Görüntü sınıflandırması için temel bir çok katmanlı NN eğitimi

## Yapay sinir ağları ile karmaşık fonksiyonların modellenmesi

Bu kitabın başında, *Bölüm 2, Sınıflandırma için Basit Makine Öğrenimi Algoritmalarının Eğitimi'nde* yapay nöronlarla makine öğrenimi algoritmaları yolculuğumuza başladık. Yapay nöronlar, bu bölümde tartışacağımız çok katmanlı yapay NN'lerin yapı taşlarını temsil etmektedir.

Yapay NN'lerin arkasındaki temel konsept, insan beyninin karmaşık problem görevlerini çözmek için nasıl çalıştığına dair hipotezler ve modeller üzerine inşa edilmiştir. Yapay NN'ler son yıllarda oldukça popülerlik kazanmış olsa da, NN'lerle ilgili ilk çalışmalar Warren McCulloch ve Walter Pitts'in nöronların nasıl çalışabileceğini ilk kez tanımladıkları 1940'lara kadar uzanmaktadır. (*A logical calculus of the ideas immanent in nervous activity*, by
*W. S. McCulloch* ve *W. Pitts*, *The Bulletin of Mathematical Biophysics*, 5(4):115-133, 1943.)

Bununla birlikte, **McCulloch-Pitts nöron** modelinin ilk uygulamasını takip eden on yıllarda - 1950'lerde Roseblatt'ın perceptron'u - birçok araştırmacı ve makine öğrenimi uygulayıcısı yavaş yavaş NN'lere olan ilgisini kaybetmeye başladı çünkü hiç kimse çok katmanlı bir NN'yi eğitmek için iyi bir çözüme sahip değildi. Sonunda, NN'lere olan ilgi 1986 yılında D.E. Rumelhart, G.E. Hinton ve R.J. Williams'ın NN'leri daha verimli bir şekilde eğitmek için geri yayılım algoritmasının (yeniden) keşfedilmesi ve popüler hale getirilmesiyle yeniden canlandı ve bu bölümde daha sonra daha ayrıntılı olarak tartışacağız (*Learning rep- resentations by backpropagating errors,* by *D.E. Rumelhart, G.E. Hinton*, and *R.J. Williams*, *Nature,* 323 (6088): 533-536, 1986). **Yapay zeka** (**YZ**), makine öğrenimi ve NN'lerin tarihiyle ilgilenen okuyucuların, araştırma topluluğunun büyük bir kısmının NN'lerin çalışılmasına olan ilgisini kaybettiği dönemler olan sözde *YZ kışları* hakkındaki Wikipedia makalesini okumaları da tavsiye edilir **(** https://en.wikipedia.org/wiki/AI_winter).

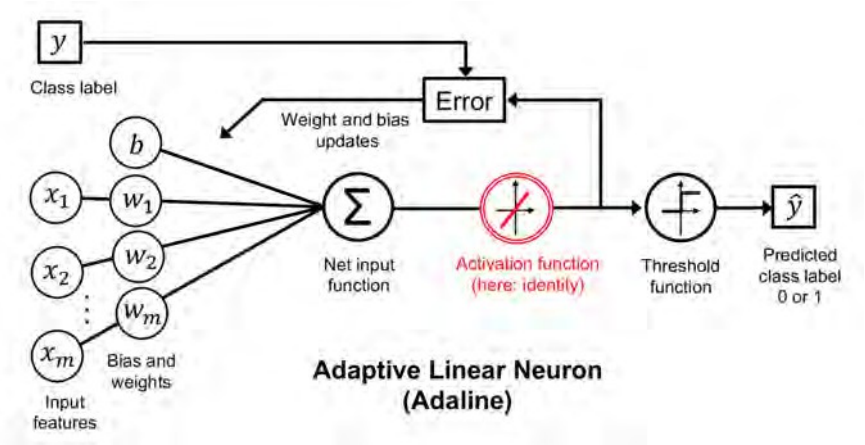
Ancak NN'ler, geçtiğimiz on yılda yapılan ve artık derin öğrenme algoritmaları ve mimarileri dediğimiz, birçok katmandan oluşan NN'lerle sonuçlanan birçok atılım sayesinde bugün her zamankinden daha popülerdir. YSA'lar sadece akademik araştırmalarda değil, aynı zamanda yapay YSA'lara ve derin öğrenme araştırmalarına büyük yatırım yapan Facebook, Microsoft, Amazon, Uber, Google ve daha birçok büyük teknoloji şirketinde de sıcak bir konudur.

Bugün itibariyle, derin öğrenme algoritmalarıyla desteklenen karmaşık NN'ler, görüntü ve ses tanıma gibi karmaşık problemlerin çözümü için son teknoloji çözümler olarak kabul edilmektedir. Son uygulamalardan bazıları şunlardır:

- Bir dizi X-ışınından COVID-19 kaynak ihtiyaçlarının tahmin edilmesi (https://arxiv.org/ abs/2101.04909)
- Virüs mutasyonlarının modellenmesi (https://science.sciencemag.org/content/371/6526/284)
- Aşırı hava olaylarını yönetmek için sosyal medya platformlarındaki verilerden yararlanma (https:// onlinelibrary.wiley.com/doi/abs/10.1111/1468-5973.12311)
- Görme engelli kişiler için fotoğraf açıklamalarının iyileştirilmesi (https://tech. fb.com/how-facebook-is-using-ai-to-improve-photo-descriptions-for-people-who-are-blind-or-visually-impaired/)

# Tek katmanlı sinir ağı özeti

Bu bölüm tamamen çok katmanlı NN'ler, nasıl çalıştıkları ve karmaşık problemleri çözmek için nasıl eğitilecekleri hakkındadır. Ancak, belirli bir çok katmanlı NN mimarisini derinlemesine incelemeden önce, *Bölüm 2'*de tanıttığımız tek katmanlı NN'lerle ilgili bazı kavramları, yani *Şekil 11.1'*de gösterilen **ADAp- tive LInear NEuron** (**Adaline**) algoritmasını kısaca tekrarlayalım:




*Şekil 11.1: Adaline algoritması*

*Bölüm 2*'de, ikili sınıflandırma gerçekleştirmek için Adaline algoritmasını uyguladık ve modelin ağırlık katsayılarını öğrenmek için gradyan inişi optimizasyon algoritmasını kullandık. Her epokta (eğitim veri kümesi üzerinden geçiş), aşağıdaki güncelleme kuralını kullanarak ağırlık vektörünü $\boldsymbol{w}$ ve bias birimini *b güncelledik:*

$$ww\ w\ ww + wwww\ b\ w\ b + wb$$

burada $\Delta w_j = -\eta \frac{\partial L}{\partial \partial \partial j}$ ve $\Delta b\ b\ b\eta \frac{\partial L}{\partial b}$ önyargı birimi için ve her ağırlık $w_j$ ağırlık vektöründe $\boldsymbol{w}$.

Başka bir deyişle, tüm eğitim veri setine dayalı olarak gradyanı hesapladık ve $\nabla L(ww)$ kayıp gradyanının ters yönünde bir adım atarak modelin ağırlıklarını güncelledik. (Basitlik açısından, aşağıdaki paragraflarda ağırlıklara odaklanacağız ve önyargı birimini ihmal edeceğiz; ancak, *Bölüm 2'*den hatırlayacağınız gibi, aynı kavramlar geçerlidir). Modelin optimum **ağırlıklarını** bulmak için, **karesel hataların ortalaması** (**MSE**) kayıp fonksiyonu olarak tanımladığımız bir amaç fonksiyonunu optimize ettik
$L(\boldsymbol{w})$. Ayrıca, gradyanı, **öğrenme** hızını kayıp fonksiyonunun küresel minimumunu aşma riskine karşı dengelemek için dikkatle seçmemiz gereken bir faktör olan $\boldsymbol{\eta}$ öğrenme **oranı** ile çarptık.

Gradyan iniş optimizasyonunda, her epoktan sonra tüm ağırlıkları eş zamanlı olarak güncelledik ve ağırlık vektöründeki her ağırlık $w_j$ için kısmi türevi aşağıdaki gibi tanımladık:

$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{n} \sum_i \left(y^{(i)} - a^{(i)}\right)^2 = \frac{2}{n} \sum_i \left(y^{(i)} - a^{(i)}\right) x_j^{(i)}$$

Burada, $y^{(i)}$ belirli bir $x^{(i)}$ örneğinin hedef sınıf etiketidir ve $a^{(i)}$ Adaline'in özel durumunda doğrusal bir fonksiyon olan nöronun aktivasyonudur.

Ayrıca, aktivasyon fonksiyonunu $\sigma(\cdot)$ aşağıdaki gibi tanımladık:

$$\sigma(\cdot) = z = a$$

Burada, net girdi, $z$, girdi katmanını çıktı katmanına bağlayan ağırlıkların doğrusal bir kombinasyonudur:

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_m x_m = \boldsymbol{w}^T \boldsymbol{x} + b$$

Gradyan güncellemesini hesaplamak için $\sigma(\cdot)$ aktivasyonunu kullanırken, sürekli değerli çıktıyı tahmin için ikili sınıf etiketlerine sıkıştırmak için bir eşik fonksiyonu uyguladık:

$$\hat{y} = \begin{cases} 1 & \text{ise } z \geq \theta \\ -1 & \text{aksi takdirde} \end{cases}$$

**Tek katmanlı adlandırma kuralı**

Adaline'in bir girdi katmanı ve bir çıktı katmanı olmak üzere iki katmandan oluşmasına rağmen, girdi ve çıktı katmanları arasındaki tek bağlantı nedeniyle tek katmanlı bir ağ olarak adlandırıldığını unutmayın.

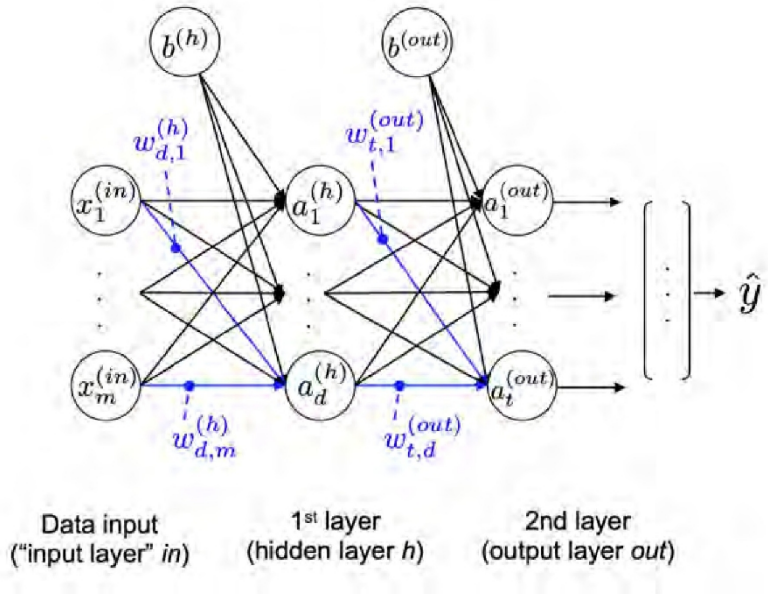
Ayrıca, model öğrenimini hızlandırmak için **stokastik gra- dient descent** (**SGD**) optimizasyonu olarak adlandırılan belirli bir *hile* öğrendik. SGD, tek bir eğitim örneğinden (çevrimiçi öğrenme) veya küçük bir eğitim örneği alt kümesinden (mini toplu öğrenme) elde edilen kayba yaklaşır. Bu bölümün ilerleyen kısımlarında **çok katmanlı** bir **algılayıcıyı** (**MLP**) uygularken ve eğitirken bu kavramdan faydalanacağız. Gradyan inişine kıyasla daha sık ağırlık güncellemeleri nedeniyle daha hızlı öğrenmenin yanı sıra, dışbükey bir kayıp fonksiyonuna sahip olmayan doğrusal olmayan aktivasyon fonksiyonlarına sahip çok katmanlı NN'leri eğitirken gürültülü doğasının da faydalı olduğu düşünülmektedir. Burada, eklenen gürültü yerel kayıp minimumlarından kaçmaya yardımcı olabilir, ancak bu konuyu bu bölümün ilerleyen kısımlarında daha ayrıntılı olarak tartışacağız.

## Çok katmanlı sinir ağı mimarisinin tanıtılması

Bu bölümde, çok katmanlı ileri beslemeli bir NN'ye birden fazla tek nöronun nasıl bağlanacağını öğreneceksiniz; bu özel *tam bağlantılı* ağ türü **MLP** olarak da adlandırılır.

ilinpikbarbell@gmail.com

*Şekil 11.2'de* iki katmandan oluşan bir MLP kavramı gösterilmektedir:




*Şekil 11.2: İki katmanlı bir MLP*

Veri girişinin yanında, *Şekil 11.2'*de gösterilen MLP bir gizli katmana ve bir çıkış katmanına sahiptir. Gizli katmandaki birimler giriş özelliklerine tam olarak bağlıdır ve çıkış katmanı gizli katmana tam olarak bağlıdır. Böyle bir ağ birden fazla gizli katmana sahipse, buna **derin NN** de diyoruz. (Bazı bağlamlarda, girdilerin de bir katman olarak kabul edildiğini unutmayın. Ancak bu durumda, tek katmanlı bir sinir ağı olan Adaline modelini iki katmanlı bir sinir ağı haline getirecektir ki bu da mantığa aykırı olabilir).

**Ek gizli katmanlar ekleme**

Daha derin ağ mimarileri oluşturmak için MLP'ye istediğimiz sayıda gizli katman ekleyebiliriz. Pratik olarak, bir NN'deki katman ve birim sayısını, *Bölüm 6, Model Değerlendirme ve Hiperparametre Ayarlama için En İyi Uygulamaları Öğrenme'd*e tartıştığımız çapraz doğrulama tekniğini kullanarak belirli bir problem görevi için optimize etmek istediğimiz ek *hiper* parametreler olarak düşünebiliriz.

Ancak, daha sonra geri yayılım yoluyla hesaplayacağımız ağ parametrelerini güncellemek için kayıp gradyanları, bir ağa daha fazla katman eklendikçe giderek daha küçük hale gelecektir. Bu kaybolan gradyan sorunu model öğrenmeyi daha zor hale getirir. Bu nedenle, bu tür DNN yapılarının eğitilmesine yardımcı olmak için özel algoritmalar geliştirilmiştir; bu, sonraki bölümlerde daha ayrıntılı olarak tartışacağımız **derin öğrenme** olarak bilinir.

*Şekil 11.2*'de gösterildiği gibi, l'inci katmandaki *l*'inci aktivasyon birimini $a_i^{(l)}$ olarak gösteriyoruz. Matematik ve kod uygulamalarını biraz daha sezgisel hale getirmek için, katmanlara atıfta bulunmak için sayısal indisler kullanmayacağız, ancak girdi özellikleri için *in* üst simgesini, gizli katman için *h* üst simgesini ve
çıktı katmanı için üst simge. Örneğin, $x_i^{(in)}$ birinci giriş özellik değerini, $a_i^{(h)}$ gizli katmandaki birinci birimi ve $a_i^{(out)}$ çıkış katmanındaki birinci birimi ifade eder. *Şekil 11.2*'deki ***b***'lerin önyargı birimlerini gösterdiğine dikkat edin. Aslında, $\boldsymbol{b}^{(h)}$ ve $\boldsymbol{b}^{(out)}$ , eleman sayısı karşılık geldikleri katmandaki düğüm sayısına eşit olan vektörlerdir. Örneğin, $\boldsymbol{b}^{(h)}$ *d* önyargı birimini depolar, burada *d* gizli katmandaki düğüm sayısıdır. Bu kulağa kafa karıştırıcı geliyorsa endişelenmeyin. Ağırlık matrislerini ve bias birimi vektörlerini başlattığımız kod uygulamasına daha sonra bakmak, bu kavramları netleştirmeye yardımcı olacaktır.

Katman *l*'deki her düğüm, katman *l* + 1'deki tüm düğümlere bir ağırlık katsayısı aracılığıyla bağlanır. Örneğin, *l* katmanındaki k. birim ile *l* + 1 katmanındaki j. birim arasındaki bağlantı $w_{ij}^{(l,l+1)}$ şeklinde yazılacaktır. Referans
*Şekil 11.2*'ye geri dönersek, girişi gizli katmana bağlayan ağırlık matrisini $\boldsymbol{W}^{(h)}$ olarak belirtiriz ve gizli katmanı çıkış katmanına bağlayan matrisi $\boldsymbol{W}^{(out)}$ olarak yazarız.

Çıkış katmanındaki bir birim ikili sınıflandırma görevi için yeterli olsa da, önceki şekilde bir NN'nin daha genel bir biçimini gördük, bu da **bire karşı hepsi** (**OvA**) tekniğinin genelleştirilmesi yoluyla çok sınıflı sınıflandırma yapmamızı sağlar. Bunun nasıl çalıştığını daha iyi anlamak için, *Bölüm 4, İyi Eğitim Veri Kümeleri Oluşturma - Veri Önişleme*'de tanıttığımız kategorik değişkenlerin **tek vuruşlu** temsilini hatırlayın.

Örneğin, bilinen Iris veri kümesindeki üç sınıf etiketini (*0=Setosa,* 1=Versicol- *or, 2=Virginica*) aşağıdaki gibi kodlayabiliriz:

$$0 = \begin{bmatrix}1\\0\\0\end{bmatrix}, 1 = \begin{bmatrix}0\\1\\0\end{bmatrix}, 2 = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

Bu tek vuruşlu vektör temsili, eğitim veri kümesinde bulunan rastgele sayıda benzersiz sınıf etiketi ile sınıflandırma görevlerinin üstesinden gelmemizi sağlar.

NN gösterimlerinde yeniyseniz, indeksleme gösterimi (alt simgeler ve üst simgeler) ilk başta biraz kafa karıştırıcı görünebilir. İlk başta aşırı karmaşık görünen bu durum, ilerleyen bölümlerde NN gösterimini vektörleştirdiğimizde çok daha anlamlı hale gelecektir. Daha önce tanıtıldığı gibi, girdi ve gizli katmanları birbirine bağlayan ağırlıkları $d \times m$ boyutlu bir $\boldsymbol{W}^{(h)}$ matrisi ile özetliyoruz; burada *d* gizli birimlerin sayısı ve *m* girdi birimlerinin sayısıdır.

# İleri yayılım yoluyla bir sinir ağının etkinleştirilmesi

Bu bölümde, bir MLP modelinin çıktısını hesaplamak için **ileri yayılma** sürecini açıklayacağız. Bir MLP modeli öğrenme bağlamına nasıl uyduğunu anlamak için, MLP öğrenme prosedürünü üç basit adımda özetleyelim:

1. Giriş katmanından başlayarak, bir çıktı oluşturmak için eğitim verilerinin kalıplarını ağ boyunca ileriye doğru yayarız.
2. Ağın çıktısına dayanarak, daha sonra açıklayacağımız bir kayıp fonksiyonu kullanarak

olimpikbarbell@gmail.com


en aza indirmek istediğimiz kaybı hesaplarız.

3. Kaybı geriye doğru ilerletir, ağdaki her ağırlık ve önyargı birimine göre türevini bulur ve modeli güncelleriz.

Son olarak, bu üç adımı birden fazla epok için tekrarladıktan ve MLP'nin ağırlık ve sapma parametrelerini öğrendikten sonra, ağ çıktısını hesaplamak için ileri yayılımı kullanırız ve önceki bölümde açıkladığımız tek-sıcak gösterimde tahmin edilen sınıf etiketlerini elde etmek için bir eşik fonksiyonu uygularız.

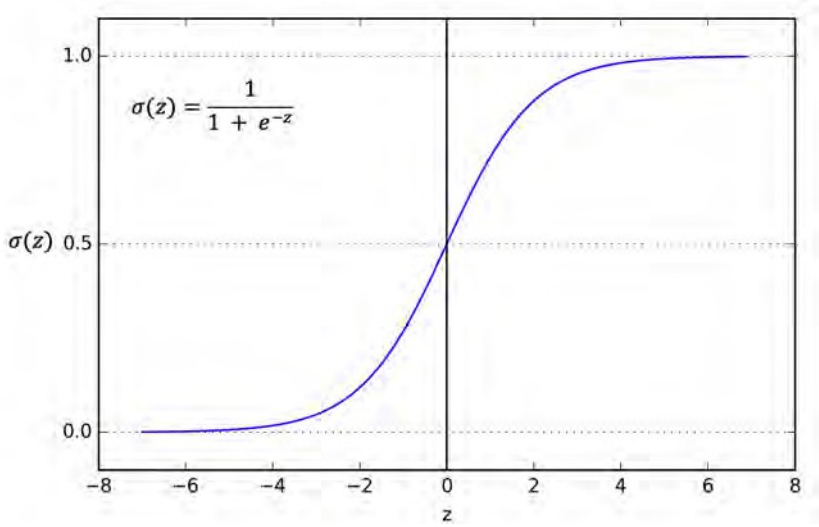
Şimdi, eğitim verilerindeki örüntülerden bir çıktı oluşturmak için ileri yayılımın bireysel adımlarını inceleyelim. Gizli katmandaki her birim giriş katmanlarındaki tüm birimlere bağlı olduğundan, ilk olarak gizli katmanın aktivasyon birimini $a^{(h)}_1$aşağıdaki gibi hesaplarız:

$$z^{(h)}_1 = x^{(in)}_1 w^{(h)}_{1,1} + x^{(in)}_2 w^{(h)}_{1,2} + \cdots + x^{(in)}_m w^{(h)}_{1,m}$$
$$a^{(h)}_1 = \sigma(z^{(h)}_1)$$

Burada$_1$ $z^{(h)}$ net girdidir ve $\sigma(\cdot)$, gradyan tabanlı bir yaklaşım kullanarak nöronları birbirine bağlayan ağırlıkları öğrenmek için türevlenebilir olması gereken aktivasyon fonksiyonudur. Görüntü sınıflandırma gibi karmaşık problemleri çözebilmek için MLP modelimizde doğrusal olmayan aktivasyon fonksiyonlarına ihtiyacımız vardır, örneğin *Bölüm 3*, *Scikit-Learn Kullanarak Makine Öğrenimi Sınıflandırıcıları Turu*'nda lojistik regresyonla ilgili bölümden hatırladığımız sigmoid (lojistik) aktivasyon fonksiyonu gibi:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Hatırlayacağınız üzere sigmoid fonksiyonu, *Şekil 11.3'te* gösterildiği gibi, net girdi *z*'yi 0 ila 1 aralığında lojistik bir dağılıma eşleyen ve *y* eksenini *z* = 0'da kesen S *şeklinde bir* eğridir:




*Şekil 11.3: Sigmoid aktivasyon fonksiyonu*

olimpikbarbell@gmail.com

MLP tipik bir ileri beslemeli yapay NN örneğidir. **İleri besleme** terimi, bu bölümün ilerleyen kısımlarında ele alacağımız ve *Bölüm 15, Tekrarlayan Sinir Ağları Kullanarak Sıralı Verilerin Modellenmesi'nde* daha ayrıntılı olarak tartışacağımız bir mimari olan tekrarlayan NN'lerin aksine, her katmanın döngüler olmadan bir sonraki katmana girdi olarak hizmet ettiği gerçeğini ifade eder. *Çok katmanlı* algılayıcı terimi biraz kafa karıştırıcı gelebilir çünkü bu ağ mimarisindeki yapay nöronlar tipik olarak algılayıcı değil sigmoid birimlerdir. MLP'deki nöronları, 0 ile 1 arasındaki sürekli aralıkta değerler döndüren lojistik regresyon birimleri olarak düşünebiliriz.

Kod verimliliği ve okunabilirliği amacıyla, şimdi aktivasyonu temel doğrusal cebir kavramlarını kullanarak daha kompakt bir biçimde yazacağız, bu da kod uygulamamızı birden fazla iç içe geçmiş ve hesaplama açısından pahalı Python `for` döngüsü yazmak yerine NumPy aracılığıyla vektörleştirmemizi sağlayacaktır:

$$z^{(h)} = \mathbf{x}^{(in)}\mathbf{W}^{(h)T} + \mathbf{b}^{(h)}$$

$$\mathbf{a}^{(h)} = \sigma(\mathbf{z}^{(h)})$$

Burada, $\mathbf{x}^{(in)}$ $1 \times m$ boyutlu özellik vektörümüzdür. $\mathbf{W}^{(h)}$ $d \times m$ boyutlu bir ağırlık matrisidir, burada $d$ gizli katmandaki birim sayısıdır; sonuç olarak, transpoze matris $\mathbf{W}^{(h)T}$ $m \times d$ boyutludur. Yanlılık vektörü $\mathbf{b}^{(h)}$ $d$ yanlılık biriminden oluşur (gizli düğüm başına bir yanlılık birimi).

Matris-vektör çarpımından sonra, aktivasyonu hesaplamak için 1×d boyutlu net girdi vektörü $\mathbf{z}^{(h)}$ elde ederiz $\mathbf{a}^{(h)}$ (burada $a^{(h)} \in \mathbb{R}^{1\times d}$).

Ayrıca, bu hesaplamayı eğitim veri kümesindeki tüm $n$ örneklere genelleştirebiliriz:

$$\mathbf{Z}^{(h)} = \mathbf{X}^{(in)}\mathbf{W}^{(h)T} + \mathbf{b}^{(h)}$$

Burada, $\mathbf{X}^{(in)}$ artık bir n×m *matrisidir* ve matris çarpımı n×d boyutlu net giriş matrisi $\mathbf{Z}^{(h)}$ ile sonuçlanacaktır. Son olarak, bir sonraki katmanda (burada, çıkış katmanı) n×d aktivasyon matrisini elde etmek için net giriş matrisindeki her değere $\sigma(\cdot)$ *aktivasyon* fonksiyonunu uygularız:

$$A^{(h)} = \sigma(Z^{(h)})$$

Benzer şekilde, çıktı katmanının aktivasyonunu birden fazla örnek için vektörleştirilmiş biçimde yazabiliriz:

$$\mathbf{Z}^{(dışarı)} = \mathbf{A}^{(h)}\mathbf{W}^{(dışarı)T} + \mathbf{b}^{(dışarı)}$$

Burada, $t \times d$ matrisinin $\mathbf{W}^{(out)}$ ($t$ çıktı birimlerinin sayısıdır) transpozesini n×d boyutlu matris $\mathbf{A}^{(h)}$ ile çarparız ve $n \times t$ *boyutlu matris* $\mathbf{Z}^{(out)}$ 'yi elde etmek için $t$ *boyutlu* sapma vektörünü $\mathbf{b}^{(out)}$ ekleriz. (Bu matristeki satırlar her örnek için çıktıları temsil eder).

Son olarak, ağımızın sürekli değerli çıktısını elde etmek için sigmoid aktivasyon fonksiyonunu uyguluyoruz:

$$A^{(out)} = \sigma(Z^{(out)})$$

$\mathbf{Z}^{(out)}$ 'ye benzer şekilde, $\mathbf{A}^{(out)}$ $n \times t$ boyutlu bir matristir.

# El yazısı rakamlarının sınıflandırılması

Bir önceki bölümde, NN'lerle ilgili birçok teoriyi ele aldık, bu konuya yeniyseniz biraz fazla yorucu olabilir. MLP modelinin ağırlıklarını öğrenmek için kullanılan algoritma olan geriye yayılımı tartışmaya devam etmeden önce, teoriye kısa bir ara verelim ve bir NN'yi çalışırken görelim.

**Geriye yayılım hakkında ek kaynaklar**

NN teorisi oldukça karmaşık olabilir; bu nedenle, okuyuculara bu bölümde tartıştığımız bazı konuları daha ayrıntılı olarak veya farklı bir bakış açısıyla ele alan ek kaynaklar sunmak istiyoruz:

- *Bölüm 6*, *Derin İleri Beslemeli Ağlar*, *Derin Öğrenme*, *I. Goodfellow, Y. Bengio* ve *A. Courville*, MIT Press, 2016 (makalelere http://www. deeplearningbook.org adresinden ücretsiz olarak erişilebilir).
- *Pattern Recognition and Machine Learning*, *C. M. Bishop*, Springer New York, 2006.
- Sebastian Raschka'nın derin öğrenme kursundan ders video slaytları:
  https://sebastianraschka.com/blog/2021/dl-course.html#l08-multinomial-logistic-regression--softmax-regression
  https://sebastianraschka.com/blog/2021/dl-course.html#l09-multilayer-perceptrons-and-backpropration

Bu bölümde, Yann LeCun ve diğerleri tarafından oluşturulan ve makine öğrenimi algoritmaları için popüler bir kıyaslama veri kümesi olarak hizmet veren popüler **Karma Ulusal Standartlar ve Teknoloji Enstitüsü** (**MNIST**) veri kümesinden el yazısı rakamları sınıflandırmak için ilk çok katmanlı NN'**mizi** uygulayacak ve eğiteceğiz (*Gradient-Based Learning Applied to Document Recognition* by *Y. LeCun*, *L. Bottou,*
*Y. Bengio* ve *P. Haffner*, *Proceedings of the IEEE,* 86(11): 2278-2324, 1998).

# MNIST veri setinin elde edilmesi ve hazırlanması

MNIST veri seti http://yann.lecun.com/exdb/mnist/ adresinde halka açıktır ve aşağıdaki dört bölümden oluşmaktadır:

1. **Eğitim veri kümesi görüntüleri**: train-images-idx3-ubyte.gz (9,9 MB, 47 MB sıkıştırılmamış ve 60.000 örnek)
2. **Eğitim veri kümesi etiketleri**: train-labels-idx1-ubyte.gz (29 KB, 60 KB sıkıştırılmamış ve 60.000 etiket)
3. **Test veri kümesi görüntüleri**: t10k-images-idx3-ubyte.gz (1,6 MB, 7,8 MB sıkıştırılmamış ve 10.000 örnek)
4. **Test veri kümesi etiketleri**: t10k-labels-idx1-ubyte.gz (5 KB, 10 KB sıkıştırılmamış ve 10.000 etiket)

 olimpikbarbell@gmail.com 

MNIST veri kümesi, ABD **Ulusal Standartlar ve Teknoloji Enstitüsü'nün (NIST)** iki veri kümesinden oluşturulmuştur. Eğitim veri kümesi, yüzde 50'si lise öğrencisi ve yüzde 50'si Nüfus Sayım Bürosu çalışanı olmak üzere 250 farklı kişinin el yazısı rakamlarından oluşmaktadır. Test veri kümesinin, aynı bölünmeyi takip eden farklı kişilerin el yazısı rakamlarını içerdiğini unutmayın.

Yukarıda bahsedilen veri kümesi dosyalarını indirmek ve bunları NumPy dizilerine önceden işlemek yerine, scikit-learn'in MNIST veri kümesini daha rahat yüklememizi sağlayan yeni fetch_openml işlevini kullanacağız:

```
>> from sklearn.datasets import fetch_openml
>> X, y = fetch_openml('mnist_784', version=1,
...                     return_X_y=True)
>> X = X.values
>> y = y.astype(int).values
```

scikit-learn'de fetch_openml fonksiyonu MNIST veri setini OpenML'den (https://www.openml.org/d/554) pandas DataFrame ve Series nesneleri olarak indirir, bu yüzden altta yatan NumPy dizilerini elde etmek için .values niteliğini kullanırız. (1.0'dan daha eski bir scikit-learn sürümü kullanıyorsanız, fetch_openml NumPy dizilerini doğrudan indirir, böylece .values niteliğini kullanmayabilirsiniz). $n \times m$ boyutlu X dizisi, her biri 784 piksel olan 70.000 görüntüden oluşur ve y dizisi, karşılık gelen 70.000 sınıf etiketini depolar; dizilerin boyutlarını aşağıdaki gibi kontrol ederek bunu doğrulayabiliriz:

```
>> print(X.shape)
(70000, 784)
>> print(y.shape)
(70000,)
```

MNIST veri kümesindeki görüntüler 28×28 pikselden oluşur ve her piksel gri tonlamalı bir yoğunluk değeriyle temsil edilir. Burada, fetch_openml 28×28 pikseli tek boyutlu satır vektörlerine dönüştürmüştür; bu vektörler yukarıdaki X dizimizdeki satırları (satır veya görüntü başına 784) temsil etmektedir. fetch_openml fonksiyonu tarafından döndürülen ikinci dizi (y) ilgili hedef değişkeni, yani el yazısı rakamların sınıf etiketlerini (0-9 tam sayıları) içerir.

Ardından, aşağıdaki kod satırını kullanarak MNIST'teki piksel değerlerini -1 ila 1 aralığına (orijinal olarak 0 ila 255) normalleştirelim:

```
>> X = ((X / 255.) - .5) * 2
```

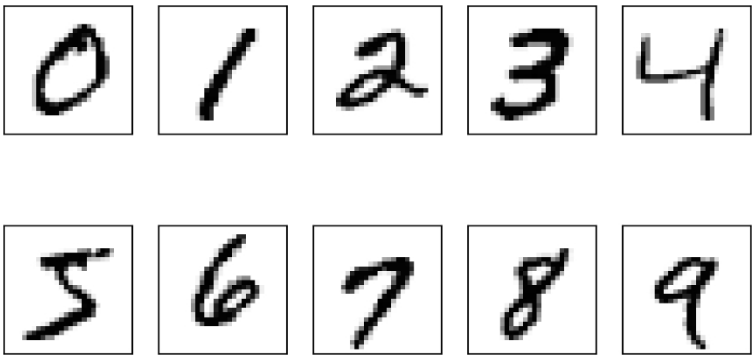
Bunun nedeni, *Bölüm 2*'de tartışıldığı gibi, gradyan tabanlı optimizasyonun bu koşullar altında çok daha kararlı olmasıdır. Görüntüleri piksel piksel ölçeklendirdiğimizi unutmayın; bu, önceki bölümlerde kullandığımız özellik ölçeklendirme yaklaşımından farklıdır.

Daha önce, eğitim veri kümesinden ölçeklendirme parametreleri türettik ve bunları eğitim veri kümesindeki ve test veri kümesindeki her sütunu ölçeklendirmek için kullandık. Bununla birlikte, görüntü pikselleriyle çalışırken, bunları sıfırda ortalamak ve [-1, 1] aralığına yeniden ölçeklendirmek de yaygındır ve genellikle pratikte iyi çalışır.

MNIST'teki bu görüntülerin nasıl göründüğüne dair bir fikir edinmek için, özellik matrisimizden 784 piksellik vektörleri Matplotlib'in imshow işlevi aracılığıyla çizebileceğimiz orijinal 28×28 görüntüye yeniden şekillendirdikten sonra 0-9 rakamlarının örneklerini görselleştirelim:

```
>> import matplotlib.pyplot as plt
>> fig, ax = plt.subplots(nrows=2, ncols=5,
...                        sharex=True, sharey=True)
>> ax = ax.flatten()
>> for i in range(10):
...     img = X[y == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>> ax[0].set_xticks([])
>> ax[0].set_yticks([])
>> plt.tight_layout()
>> plt.show()
```

Şimdi her bir benzersiz rakamın temsili bir görüntüsünü gösteren 2×5 alt şekillerin bir grafiğini görmeliyiz:



*Şekil 11.4: Her sınıftan rastgele seçilen bir el yazısı rakamını gösteren bir çizim*

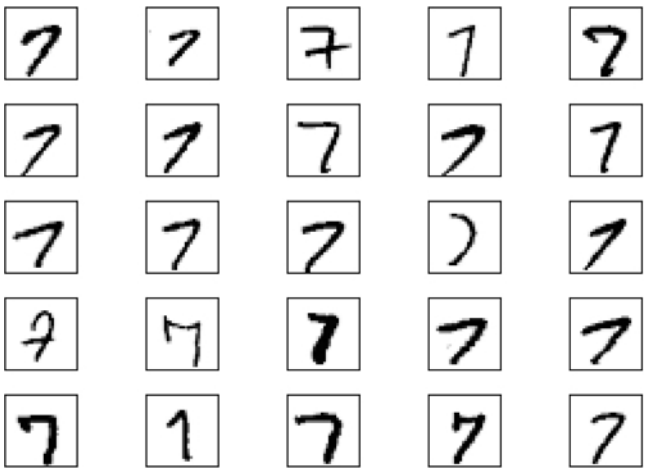
Buna ek olarak, her birinin el yazısının gerçekten ne kadar farklı olduğunu görmek için aynı rakamın birden fazla örneğini de çizelim:

```
>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True)
>> ax = ax.flatten()
```

 ilinpikbarbell@gmail.com 

```
>> for i in range(25):
...         img = X[y == 7][i].reshape(28, 28)
...         ax[i].imshow(img, cmap='Greys')
>> ax[0].set_xticks([])
>> ax[0].set_yticks([])
>> plt.tight_layout()
>> plt.show()
```

Kodu çalıştırdıktan sonra, şimdi 7 rakamının ilk 25 varyantını görmeliyiz:



*Şekil 11.5: El yazısı 7 rakamının farklı varyantları*

Son olarak, veri kümesini eğitim, doğrulama ve test alt kümelerine ayıralım. Aşağıdaki kod, veri kümesini eğitim için 55.000 görüntü, doğrulama için 5.000 görüntü ve test için 10.000 görüntü kullanılacak şekilde bölecektir:

```
>> from sklearn.model_selection import train_test_split
>> X_temp, X_test, y_temp, y_test = train_test_split(
...         X, y, test_size=10000, random_state=123, stratify=y
... )
>> X_train, X_valid, y_train, y_valid = train_test_split(
...         X_temp, y_temp, test_size=5000,
...         random_state=123, stratify=y_temp
... )
```

## Çok katmanlı bir algılayıcının uygulanması

Bu alt bölümde, şimdi MNIST veri kümesindeki görüntüleri sınıflandırmak için sıfırdan bir MLP uygulayacağız. İşleri basit tutmak için, yalnızca bir gizli katmana sahip bir MLP uygulayacağız. Uygulama ilk başta biraz karmaşık görünebileceğinden, bu bölüm için örnek kodu Packt Publishing web sitesinden veya GitHub'dan (https://github.com/rasbt/ machine-learning-book) indirmeniz önerilir, böylece bu MLP uygulamasını daha iyi okunabilirlik için yorumlarla ve sözdizimi vurgulamalarıyla açıklanmış olarak görebilirsiniz.

Kodu beraberindeki Jupyter Notebook dosyasından çalıştırmıyorsanız veya internete erişiminiz yoksa, bu bölümdeki NeuralNetMLP kodunu geçerli çalışma dizininizdeki bir Python betik dosyasına kopyalayın (örneğin, neuralnet.py), daha sonra aşağıdaki komutla geçerli Python oturumunuza aktarabilirsiniz:

```
from neuralnet import NeuralNetMLP
```

Kod, geriye yayılım algoritması gibi henüz bahsetmediğimiz kısımlar içerecektir. Kodun tamamı size hemen anlamlı gelmiyorsa endişelenmeyin; bu bölümün ilerleyen kısımlarında belirli kısımları takip edeceğiz. Ancak, bu aşamada kodun üzerinden geçmek daha sonra teoriyi takip etmeyi kolaylaştırabilir.

Şimdi, lojistik sigmoid aktivasyonunu hesaplamak ve tamsayı sınıf etiketi dizilerini tek vuruşlu kodlanmış etiketlere dönüştürmek için iki yardımcı fonksiyonla başlayarak bir MLP'nin aşağıdaki uygulamasına bakalım:

```
import numpy as np

def sigmoid(z):
    1 döndürür. / (1. + np.exp(-z))


def int_to_onehot(y, num_labels):

    ary = np.zeros((y.shape[0], num_labels))
    for i, val in enumerate(y):
        ary[i, val] = 1

    ary'yi döndür
```

ilinpikbarbell@gmail.com

Aşağıda, MLP'miz için NeuralNetMLP adını verdiğimiz ana sınıfı uyguluyoruz. init (), .forward() ve .backward() olmak üzere üç sınıf yöntemi vardır ve init yapıcısından başlayarak bunları tek tek ele alacağız:

```
sınıf NeuralNetMLP:

    def init (self, num_features, num_hidden,
                num_classes, random_seed=123):
        super(). init () self.num_classes

        = num_classes

        # gizli
        rng = np.random.RandomState(random_seed)

        self.weight_h = rng.normal(
            loc=0.0, scale=0.1, size=(num_hidden, num_features))
        self.bias_h = np.zeros(num_hidden)

        # çıktı
        self.weight_out = rng.normal(
            loc=0.0, scale=0.1, size=(num_classes, num_hidden))
        self.bias_out = np.zeros(num_classes)
```

init kurucusu, gizli ve çıktı katmanı için ağırlık matrislerini ve yanlılık vektörlerini örneklendirir. Daha sonra, bunların tahminler yapmak için ileri yöntemde nasıl kullanıldığını görelim:

```
    def ileri(self, x):
      # Gizli katman

        # girdi dim: [n_örnekler, n_özellikler] #
                dot [n_hidden, n_features].T #
        çıktı dim: [n_examples, n_hidden]
        z_h = np.dot(x, self.weight_h.T) + self.bias_h a_h
        = sigmoid(z_h)

        # Çıktı katmanı
        # giriş dim: [n_examples, n_hidden] #
                nokta [n_classes,
        n_hidden].T
        # çıktı dim: [n_examples, n_classes]
        z_out = np.dot(a_h, self.weight_out.T) + self.bias_out
        a_out = sigmoid(z_out)
        a_h, a_out döndür
```

İleri yöntemi bir veya daha fazla eğitim örneği alır ve tahminleri döndürür. Aslında, hem gizli katmandan hem de çıktı katmanından aktivasyon değerlerini, a_h ve a_out'u döndürür. a_out, önemsediğimiz sınıf etiketlerine dönüştürebileceğimiz sınıf üyeliği olasılıklarını temsil ederken, model parametrelerini optimize etmek için gizli katmandan gelen aktivasyon değerlerine, yani gizli ve çıktı katmanlarının ağırlık ve yanlılık birimlerine de ihtiyacımız vardır.

Son olarak, sinir ağının ağırlık ve bias parametrelerini güncelleyen geriye dönük yöntemden bahsedelim:

```
    def geri(self, x, a_h, a_out, y):

        #########################
        ### Çıktı katmanı ağırlıkları
        #########################

        # tek seferlik kodlama
        y_onehot = int_to_onehot(y, self.num_classes)

        # Bölüm 1: dLoss/dOutWeights
        ## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight ##
        burada DeltaOut = dLoss/dOutAct * dOutAct/dOutNet
        ## uygun yeniden kullanım için

        # girdi/çıktı dim: [n_examples, n_classes]
        d_loss d_a_out = 2.*(a_out - y_onehot) / y.shape[0]

        # girdi/çıktı dim: [n_examples, n_classes]
        d_a_out d_z_out = a_out * (1. - a_out) # sigmoid türev

        # çıktı dim: [n_examples, n_classes]
        delta_out = d_loss d_a_out * d_a_out d_z_out

        # çıktı ağırlıkları için gradyan #

        [n_examples, n_hidden]
        d_z_out dw_out = a_h

        # girdi dim: [n_classes, n_examples]
        #                    dot [n_examples, n_hidden]
        # çıktı dim: [n_classes, n_hidden]
        d_loss dw_out = np.dot(delta_out.T, d_z_out dw_out) d_loss
        db_out = np.sum(delta_out, axis=0)
```

 olimpikbarbell@gmail.com 

```
        ##################################
        # Bölüm 2: dLoss/dHiddenWeights
        ## = DeltaOut * dOutNet/dHiddenAct * dHiddenAct/dHiddenNet #
              * dHiddenNet/dWeight

        # [n_classes, n_hidden]
        d_z_out a_h = self.weight_out

        # çıktı dim: [n_examples, n_hidden]
        d_kayıp a_h = np.dot(delta_out, d_z_out a_h)

        # [n_examples, n_hidden]
        d_a_h d_z_h = a_h * (1. - a_h) # sigmoid türev

        # [n_examples, n_features]
        d_z_h d_w_h = x

        # çıktı dim: [n_hidden, n_features]
        d_loss d_w_h = np.dot((d_loss a_h * d_a_h d_z_h).T,
                                    d_z_h d_w_h)
        d_loss d_b_h = np.sum((d_loss a_h * d_a_h d_z_h), axis=0)

        return (d_loss dw_out, d_loss db_out,
                d_loss d_w_h, d_loss d_b_h)
```

Geriye doğru yöntem, ağırlık ve önyargı parametrelerine göre kaybın gradyanlarını hesaplayan geriye *yayılma* algoritmasını uygular. Adaline'e benzer şekilde, bu gradyanlar daha sonra bu parametreleri gradyan inişi yoluyla güncellemek için kullanılır. Çok katmanlı NN'lerin tek katmanlı kardeşlerinden daha karmaşık olduğunu ve kodu tartıştıktan sonraki bir bölümde gradyanların nasıl hesaplanacağına ilişkin matematiksel kavramların üzerinden geçeceğimizi unutmayın. Şimdilik, geriye doğru yöntemi gradyan iniş güncellemeleri için kullanılan gradyanları hesaplamanın bir yolu olarak düşünün. Basit olması açısından, bu türetmenin dayandığı kayıp fonksiyonu Adaline'de kullandığımız MSE kaybı ile aynıdır. İlerleyen bölümlerde, ikili lojistik regresyon kaybının birden fazla sınıfa genelleştirilmesi olan çok kategorili çapraz entropi kaybı gibi alternatif kayıp fonksiyonlarını inceleyeceğiz.

NeuralNetMLP sınıfının bu kod uygulamasına baktığınızda, bu nesne yönelimli uygulamanın **.fit()** ve .predict() yöntemleri etrafında odaklanan tanıdık scikit-learn API'sinden farklı olduğunu fark etmiş olabilirsiniz. Bunun yerine, NeuralNetMLP sınıfının ana yöntemleri şunlardır
.forward() ve .backward() yöntemleri. Bunun arkasındaki nedenlerden biri, karmaşık bir nöro- ral ağı, bilginin ağlar arasında nasıl aktığı açısından biraz daha kolay anlaşılır hale getirmesidir.

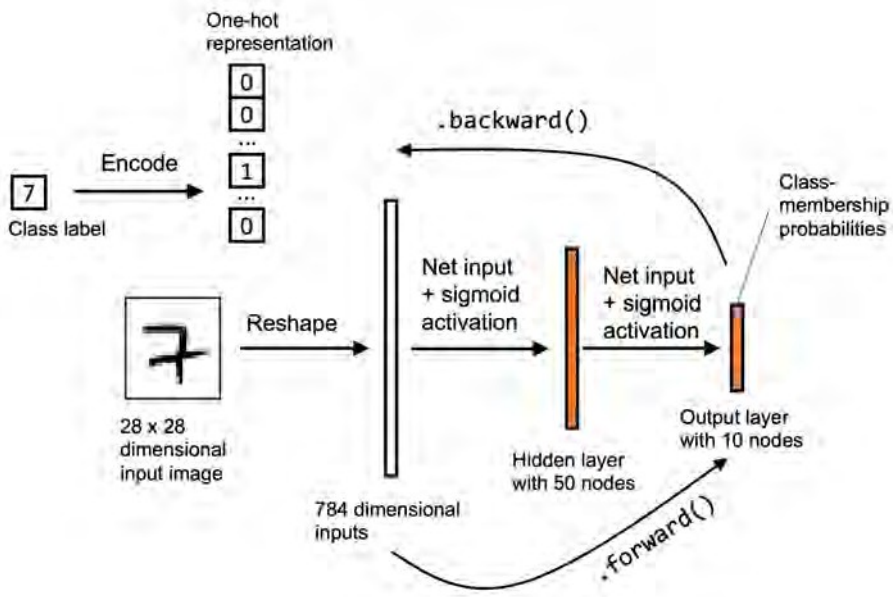
Bir diğer neden ise bu uygulamanın, ilerleyen bölümlerde daha karmaşık sinir ağları kurmak için tanıtacağımız ve kullanacağımız PyTorch gibi daha gelişmiş derin öğrenme kütüphanelerinin çalışma şekline nispeten benzer olmasıdır.

NeuralNetMLP sınıfını uyguladıktan sonra, yeni bir NeuralNetMLP sınıfı oluşturmak için aşağıdaki kodu kullanırız
NeuralNetMLP nesnesi:

```
>>> model = NeuralNetMLP(num_features=28*28,
...                      num_hidden=50,
...                      num_classes=10)
```

Model, 10 tamsayı sınıfı (0-9 rakamları) için 784 boyutlu vektörlere (daha önce tanımladığımız X_train, X_valid veya X_test biçiminde) yeniden şekillendirilmiş MNIST görüntülerini kabul eder. Gizli katman 50 düğümden oluşur. Ayrıca, daha önce tanımlanmış olan .forward() yönteminde, işleri basit tutmak için ilk gizli katman ve çıktı katmanından sonra bir sigmoid aktivasyon fonksiyonu kullanıyoruz. Daha sonraki bölümlerde, hem gizli hem de çıktı katmanları için alternatif aktivasyon fonksiyonlarını öğreneceğiz.

*Şekil 11.6* yukarıda örneklediğimiz sinir ağı mimarisini özetlemektedir:




*Şekil 11.6: El yazısı rakamları etiketlemek için NN mimarisi*

Bir sonraki alt bölümde, ağı geri yayılım yoluyla mini veri yığınları üzerinde eğitmek için kullanabileceğimiz eğitim fonksiyonunu uygulayacağız.

 olimpikbarbell@gmail.com 

## Sinir ağı eğitim döngüsünün kodlanması

Önceki alt bölümde NeuralNetMLP sınıfını uyguladığımıza ve bir model başlattığımıza göre, bir sonraki adım modeli eğitmektir. Bunu birden fazla adımda ele alacağız. İlk olarak, veri yükleme için bazı yardımcı fonksiyonlar tanımlayacağız. İkinci olarak, bu fonksiyonları veri kümesi üzerinde birden fazla epokta yineleme yapan eğitim döngüsüne yerleştireceğiz.

Tanımlayacağımız ilk fonksiyon, veri setimizi alan ve stokastik gradyan inişi eğitimi için istenen boyutta mini partilere bölen bir mini parti oluşturucudur. Kod aşağıdaki gibidir:

```
>> import numpy as np
>> num_epochs = 50
>> minibatch_size = 100

>> def minibatch_generator(X, y, minibatch_size):
...         indices = np.arange(X.shape[0])
...         np.random.shuffle(indices)
...         for start_idx in range(0, indices.shape[0] - minibatch_size
...                                 + 1, minibatch_size):
...             batch_idx = indices[start_idx:start_idx + minibatch_size]
...             yield X[batch_idx], y[batch_idx]
```

Sonraki fonksiyonlara geçmeden önce, mini parti oluşturucunun amaçlandığı gibi çalıştığını ve istenen boyutta mini partiler ürettiğini doğrulayalım. Aşağıdaki kod veri kümesi boyunca yineleme yapmaya çalışacak ve ardından mini partilerin boyutunu yazdıracağız. Aşağıdaki kod örneklerinde break ifadelerini kaldıracağımızı unutmayın. Kod aşağıdaki gibidir:

```
>> # eğitim epokları üzerinde yineleme
>> for i in range(num_epochs):
...         # minibatchler üzerinde yinele
...         minibatch_gen = minibatch_generator(
...             X_train, y_train, minibatch_size)
...         for X_train_mini, y_train_mini in minibatch_gen:
...             Mola
...         Mola
>> print(X_train_mini.shape)
(100, 784)
>> print(y_train_mini.shape)
(100,)
```

Gördüğümüz gibi, ağ amaçlandığı gibi 100 boyutlu mini partiler döndürmektedir.

Daha sonra, eğitim sürecini izlemek ve modeli değerlendirmek için kullanabileceğimiz kayıp fonksiyonumuzu ve performans metriğimizi tanımlamamız gerekir. MSE kayıp ve doğruluk fonksiyonu aşağıdaki gibi uygulanabilir:

```
>> def mse_loss(targets, probas, num_labels=10):
...     onehot_targets = int_to_onehot(
...         targets, num_labels=num_labels
...     )
...     return np.mean((onehot_targets - probas)**2)

>> def accuracy(targets, predicted_labels):
...     return np.mean(predicted_labels == targets)
```

Önceki fonksiyonu test edelim ve önceki bölümde örneklediğimiz modelin ilk doğrulama seti MSE'sini ve doğruluğunu hesaplayalım:

```
>> _, probas = model.forward(X_valid)
>>> mse = mse_loss(y_valid, probas)
>> print(f'İlk doğrulama MSE: {mse:.1f}') İlk
doğrulama MSE: 0,3

>>> tahmini_etiketler = np.argmax(probas, axis=1)
>>> acc = accuracy(y_valid, predicted_labels)
>> print(f'İlk doğrulama doğruluğu: {acc*100:.1f}%') İlk
doğrulama doğruluğu: %9,4
```

Bu kod örneğinde, `model.forward()` işlevinin gizli ve çıktı katmanı aktivasyonlarını döndürdüğüne dikkat edin. 10 çıkış düğümümüz olduğunu unutmayın (her bir benzersiz sınıf etiketine karşılık gelen bir tane). Bu nedenle, MSE'yi hesaplarken, ilk olarak sınıf etiketlerini `mse_loss()` işlevinde bir-sıcak kodlanmış sınıf etiketlerine dönüştürdük. Pratikte, önce karesel fark matrisinin satırlarının mı yoksa sütunlarının mı ortalamasını alacağımız bir fark yaratmaz, bu nedenle `np.mean()` işlevini herhangi bir eksen belirtmeden çağırırız, böylece bir skaler döndürür.

Lojistik sigmoid fonksiyonunu kullandığımız için çıktı katmanı aktivasyonları [0, 1] aralığındaki değerlerdir. Her girdi için, çıktı katmanı [0, 1] aralığında 10 değer üretir, bu nedenle tahmin edilen sınıf etiketini veren en büyük değerin indeks konumunu seçmek için `np.argmax()` işlevini kullandık. Daha sonra tanımladığımız `accuracy()` fonksiyonu aracılığıyla doğruluğu hesaplamak için gerçek etiketleri tahmin edilen sınıf etiketleriyle karşılaştırdık. Önceki çıktıdan da görebileceğimiz gibi, doğruluk çok yüksek değildir. Bununla birlikte, 10 sınıflı dengeli bir veri kümemiz olduğu göz önüne alındığında, yaklaşık yüzde 10'luk bir tahmin doğruluğu, rastgele tahminler üreten eğitimsiz bir model için beklediğimiz şeydir.

ilinpikbarbell@gmail.com

Önceki kodu kullanarak, hedeflere girdi olarak y_train ve modeli X_train ile besleyerek tahmin edilen etiketleri sağlarsak, örneğin tüm eğitim seti üzerindeki performansı hesaplayabiliriz. Ancak pratikte, bilgisayar belleğimiz genellikle modelin bir ileri geçişte ne kadar veri alabileceği konusunda sınırlayıcı bir faktördür (büyük matris çarpımları nedeniyle). Bu nedenle, MSE ve doğruluk hesaplamamızı önceki mini parti oluşturucumuza göre tanımlıyoruz. Aşağıdaki işlev, bellek açısından daha verimli olması için veri kümesi üzerinde her seferinde bir mini yığın yineleyerek MSE ve doğruluğu aşamalı olarak hesaplayacaktır:

```
>> def compute_mse_and_acc(nnet, X, y, num_labels=10,
...                                 minibatch_size=100):
...         mse, correct_pred, num_examples = 0., 0, 0
...         minibatch_gen = minibatch_generator(X, y, minibatch_size)
...         for i, (özellikler, hedefler) in enumerate(minibatch_gen):
...             _, probas = nnet.forward(özellikler)
...             öngörülen_etiketler = np.argmax(probas, eksen=1)
...             onehot_targets = int_to_onehot(
...                 targets, num_labels=num_labels
...             )
...             kayıp = np.mean((onehot_targets - probas)**2)
...             correct_pred += (predicted_labels == targets).sum()
...             num_examples += targets.shape[0]
...             mse += kayıp
...         mse = mse/i
...         acc = correct_pred/num_examples
...         dönüş mse, acc
```

Eğitim döngüsünü uygulamadan önce, işlevi test edelim ve önceki bölümde örneklediğimiz modelin ilk eğitim seti MSE'sini ve doğruluğunu hesaplayalım ve amaçlandığı gibi çalıştığından emin olalım:

```
>>> mse, acc = compute_mse_and_acc(model, X_valid, y_valid)
>> print(f'İlk geçerli MSE: {mse:.1f}') İlk
geçerli MSE: 0,3
>> print(f'İlk geçerli doğruluk: {acc*100:.1f}%') İlk
geçerli doğruluk: %9,4
```

Sonuçlardan da görebileceğimiz gibi, üreteç yaklaşımımız, MSE'deki küçük bir yuvarlama hatası (0,27'ye karşı 0,28) dışında, daha önce tanımlanan MSE ve doğruluk fonksiyonlarıyla aynı sonuçları üretmektedir.

Şimdi ana kısma geçelim ve modelimizi eğitmek için kodu uygulayalım:

```
>> def train(model, X_train, y_train, X_valid, y_valid, num_epochs,
...                 learning_rate=0.1):
...         epoch_loss = []
```

```
...     epoch_train_acc = []
...     epoch_valid_acc = []
...
...     for e in range(num_epochs):
...         # minibatchler üzerinde yinele
...         minibatch_gen = minibatch_generator(
...             X_train, y_train, minibatch_size)
...         for X_train_mini, y_train_mini in minibatch_gen:
...             #### Çıktıları hesapla ####
...             a_h, a_out = model.forward(X_train_mini)

...             #### Gradyanları hesaplayın ####
...             d_loss d_w_out, d_loss d_b_out, \
...             d_loss d_w_h, d_loss d_b_h = \
...                 model.backward(X_train_mini, a_h, a_out,
...                                y_train_mini)
...
...             #### Güncelleme ağırlıkları ####
...             model.weight_h -= learning_rate * d_loss d_w_h
...             model.bias_h -= learning_rate * d_loss d_b_h
...             model.weight_out -= learning_rate * d_loss d_w_out
...             model.bias_out -= learning_rate * d_loss d_b_out
...
...         #### Epoch Günlüğü ####
...         train_mse, train_acc = compute_mse_and_acc(
...             model, X_train, y_train
...         )
...         valid_mse, valid_acc = compute_mse_and_acc(
...             model, X_valid, y_valid
...         )
...         train_acc, valid_acc = train_acc*100, valid_acc*100
...         epoch_train_acc.append(train_acc)
...         epoch_valid_acc.append(valid_acc)
...         epoch_loss.append(train_mse)
...         print(f'Epoch: {e+1:03d}/{num_epochs:03d} '
...               f'| Train MSE: {train_mse:.2f} '
...               f'| Tren Acc: {train_acc:.2f}% '
...               f'| Geçerli Acc: {valid_acc:.2f}%')
...
...     epoch_loss, epoch_train_acc, epoch_valid_acc döndür
```