**MAKİNE ÖĞRENİMİ**

**MAKİNE ÖĞRENMESİ SİSTEMİNİN TÜRLERİ**

**ÇEVRİMİÇİ ÖĞRENME**

Verilerin küçük gruplar halinde, sistemi beslemesiyle öğrenimin arttırılması amaçlanır. Sistem sürekli yeni verilerle beslenir. Örneğin, kur tahmini yapan sistemi besleyen döviz kuru verisi buna örnektir.

Sistemin değişen verilere hızlı bir şekilde adapte olabilmesi, oldukça önemlidir. Sistemin öğrenmesine bağlı olarak **öğrenme oranı** oluşur. Oran fazla olursa, sistem verileri çabuk unutur. Az olursa da sistemin eylemsizliği artar.

Çevrimiçi öğrenme sürecinde, sistem hatalı verilerle beslenebilir. Akabinde bu sorunlara yol açacaktır. Böyle bir durumda sistemin durdurularak, doğru çalıştığı son noktadan tekrar başlatılması gerekmektedir.

**ÖRNEK TABANLI ÖĞRENME**

Sistem örnek veriyi ezberler. Ardından yeni veriler ile benzerlik karşılaştırması yaparak öğrenmeye çalışır.. Verimli olmadığından dolayı, pek kullanılmaz.

**MODEL TABANLI ÖĞRENME**

Mevcut verilerimiz ile veriye uygun bir model oluşturulur. Uygun model seçimi yapılmazsa, modelin hatalı tahmin yapması kaçınılmazdır.

Örneğin elimizde bir ülkenin kişi başı milli gelir oranı olsun. Bu oran arttıkça kişilerin geliri de artacağından, kişilerin alım gücünün de artması beklenir. Bu durumda doğrusal bir artışın olacağını görebiliriz. Doğrusal artışın olduğu veriler için doğrusal fonksiyon ile model oluşturulur. İzlediğimiz bu yönteme de **modelin seçilmesi** denir.

Seçtiğimiz modelin performansını ölçmek isteyebiliriz. Bunun için iki adet fonksiyonumuz mevcuttur.

* **Yardımcı Fonksiyon**, iyi performansı
* **Maliyet Fonskiyonu**, kötü performansını ölçer.

Modelimizi oluşturduktan sonra, eğitim verileriyle modeli besleyerek en uygun parametreleri bulmaya çalışırız. Bu işleme **modelin eğitilmesi** denir.

Yapacaklarımızı özetleyecek olursak sırasıyla,

* Verimizi inceledik
* Verimize uygun modeli seçtik
* Performansını ölçmek için maliyet fonksiyonunu kullanarak eğittik.
* Modele yeni örnekler göstererek genelleştirilmesini sağladık.

**KARŞILAŞILAN TEMEL ZORLUKLAR**

**EĞİTİM VERİLERİNİN YETERSİZ OLUŞU**

Makine öğrenmesinde algoritmanın düzgün çalışması için çok fazla veriye ihtiyacımız vardır.

**TEMSİL EDEMEYEN EĞİTİM VERİLERİ**

Modelimizin genelleştirilmesi için, temsil ettiği değerlere ait tüm verilerle eğitilmesi gerekmektedir. Verinin eksik olması durumunda hatalı sonuçlar alabiliriz. Örnek verilerimizin az olması, **veri gürültüsüne** yol açacak ve verimliliği düşürecektir.

Örnek verimizin büyük olmasına karşın, verileri uygun bir şekilde karıştırmadan seçim yaparsak **örneklem yanlılığı** oluşacaktır.

Örneğin yapılan bir araştırmaya ait verilerin içinden en hevesli alıcılara ait ilk 200 veriyi seçersek modelimiz diğer kesimleri (cevap vermeyenler, hevesli olmayanlar) temsil etmeyeceğinden güvenilir olmayacaktır.

**DÜŞÜK KALİTELİ VERİLER**

Eğitim verilerimizde bulunan hatalar ve aykırı değerler sistemin öğrenimini zorlaştıracak, performansını düşürecektir. Eğitim verimizi temizlemek için harcamamız gereken zaman, eğitim verimizin boyutu ile orantılı olarak artacaktır. Eğitim verimizi temizlemek için izlememiz gereken bazı yöntemler ise:

* Bazı örneklerin aykırı değere sahip olduğu açıkca görünüyorsa, elle düzeltilmeli
* Örneklerde eksik nitelik varsa göz ardı edebilir ya da değerler ile doldurabiliriz.

**ALAKASIZ ÖZNİTELİKLER**

Öznitelik, bir veri seti içerisinde bulunan ve hedeflenen model çıktısını oluşturmamızı sağlayacak olan her bir kolon/sütundur. Bunlar yaş, cinsiyet, alışveriş yapma sıklığı gibi verilerden oluşur.

Sistem fazla öznitelik içerdiği gibi eksik öznitelikler de içerebilir. Bu süreçte yapmamız gerekenler ise:

* Öznitelik, mevcut veriler arasından en kullanışlı olanlardan seçilir.
* Daha kullanışlı olması için, öznitelik çıkartma, birleşitmr veya boyut azaltma algoritmaları uygulanır.
* Yeni nitelikler toplayarak, yeni öznitelik oluşturma yolunu seçebiliriz.

**EĞİTİM VERİLERİNE AŞIRI UYDURMA**

Modelimiz, eğitim için kullandığımız veri üzerinde fazla çalışarak genelleme yapabilir buna **aşırı uydurma**

denir.

Derin sinir ağları gibi modeller veride bulunan küçük örüntüleri algılayabilir. Fakat eğitim verisi gürültülü ya da çok küçükse bu algılama zorlaşacaktır. Bunun sonucunda alakasız gruplamalar ve örneklemeler ortaya çıkacaktır.

Örneğin içinde H harfi olan ülkeler çok gelişmiştir tarzında bir gruplama hatalı olacaktır. Çünkü H gruplama için bir koşul belirtmemektedir. Bunları çözmek için,

* Fazla eğitim veri seti toplanmalı
* Eğitim verilerinde bulunan gürültü azaltılmalı
* Öznitelik sayısı azaltılmalı ya da modeli sınırlandırıp daha az parametre barındıran farklı bir model seçilmelidir.

Modeli daha basit hale getirip, aşırı uydurma riskini azaltmak için bazı kısıtlamalar yaparız. Buna

**düzenlileştirme (regularization)** denir. Öğrenme sırasında uygulanacak düzenleştirme miktarını bir adet **hiperparametre** ile kontrol ederiz. Bu öğrenme algoritmasının bir parametresidir, modelin değil. Bu nedenle öğrenme algoritması kendinden etkilenmemiş olur. Hiperparametre eğitimin başlamadan ayarlanır ve sonuna kadar sabit kalır.

**EĞİTİM VERİLERİNE YETERSİZ UYDURMA**

Bir model yetersiz öğrenmeye sahipse, model eğitim verilerine uymaz ve verilerde bulunan yapılara öğrenmede yetersiz kalır. Bunun önüne geçmek için,

* Daha fazla parametreye sahip daha güçlü bir model seçmeliyiz
* Öğrenme algoritmasını, daha iyi öznitelikler ile besleyebiliriz
* Modeldeki kısıtlamaları (hiperparametreyi) azaltmalıyız

**TEST VE DOĞRULAMA**

Veri seti eğitim ve test verileri olmak üzere iki bölüme ayrılmalıdır. Genel de bu oran veri setinin boyutuna göre değişebilmekle birlikte %80 - %20’ dir.

Modelimizin eğitim hatası düşük fakat genelleştirme hatası yüksek ise modelin eğitim veri setine aşırı uydurduğu anlamı çıkmaktadır.

**HİPERPARAMETRE AYARI VE MODEL SEÇİMİ**

Test veri setinde birden çok kez ölçüm yapmamız ve model mimarisi ile hiperparametreleri belirli bir veri seti için en iyi modeli üretecek şekilde adapte etmemiz, performans düşüklüğü oluşturacaktır. Bu sorunun çözümünün en yaygın olanı **Hold-out doğrulamasıdır**. Bu yöntem veri setini eğitim ve test olmak üzere ikiye ayırma yöntemidir.

Farklı aday modellerimizi, eğitim aşamasında farklı alt veri setiyle değerlendirip en iyi olanını seçeriz. Bu alt veri setine **doğrulama veri seti** denir. Eğer doğrulama veri setimiz küçükse değerlendirme doğru olmaz. Çok büyük olması durumunda, çok hızlı giden bir arabadan çevreyi anlamak gibi veriyi yeterli derecede öğrenememiş olacaktır. Bunun çözümü ise **Çapraz Doğrulama (Cross Validation)** işlemidir. Eğitim verisi birden çok alt veri setine ayrılır. Tek alt kümeyi eğitim için kullanıp diğer kalan kümeleri doğrulama işlemi için kullanılırız. Bu işlem çapraz bir şekilde tüm alt kümeler için tekrarlanır.

Doğrulama veri setinin performansı kötüyse sorun veri uyumsuzluğudur. Tersi bir durum ise eğitim veri setine aşırı uyduruyor olmasıdır. Bu durumda önceden belirttiğimiz üç maddeye göre hareket etmeliyiz.

**UÇTAN UCA MAKİNE ÖĞRENMESİ PROJESİ**

Öğrendiklerimizi daha iyi geliştirmek amacıyla, ilk projemizi yapalım. Amacımız herhangi bir bölgede bulunan konut fiyatlarını tahmin eden bir model oluşturmak.

Projeme başlamadan önce, iyi bir veri bilimcinin yapması gereken ilk şey, süreç listesi oluşturmaktır.

* Problemin çerçevesini oluştur
* Verileri topla ve fikir sahibi olmak adına veriyi keşfet
* Verilerin içerdiği ürüntüleri ortaya çıkartmak için uygun öğrenme algoritmasını hazırla
* Modellere ince ayar yaparak, genelleştirilmiş mükemmel bir çözümde birleştir
* Sistemi çalıştırıp, gerekli bakımlarını yap

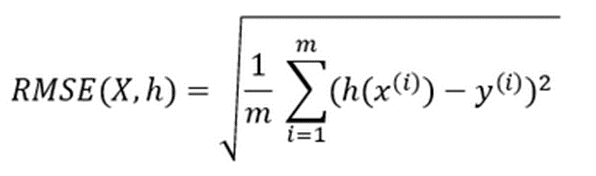
**PROBLEMİ TANIMAK**

Modelin hangi süreçte kullanılacağı önemlidir. Modelin çıktısı başka bir sistem için sinyal girdisi olarak da kullanılabilir.

Elimizde bulunan verilerden yola çıkarak, hangi öğrenme türlerine göre ilerleyeceğimizi belirleyelim. Elimizde bulunan konut fiyatlarını içeren veri seti, etiketlere sahip olduğundan denetimli öğrenme görevini kullanacağımız açıktır. Bir değer tahmininde bulunacağımız için de bağlanım görevine de örnektir. Sisteme sürekli bir veri akışını gelmediğinden uyum sağlama problemimiz bulunmuyor. Yığın öğrenmeye de gayet uygundur. Eğer veri setimiz büyükse, yığın öğrenmeyi birden fazla sunucuya bölmemiz gerekir.

**PERFORMANS ÖLÇÜTÜNÜN SEÇİLMESİ**

**Kök Ortalama Kare Hata (Root Mean Square Error)**, tahminleyicinin tahmin ettiği değer ile gerçek değer arasındaki uzaklığın bulunmasında kullanılan ölçütlerden biridir. RMSE, verilere uyan en iyi çizgiyi bize söyler. Negatif değerleri de işleme katar. Büyük hatalara daha fazla ağırlık ve ceza vererek ne kadar hata yapıldığı ile ilgili bilgi verir.



* **M**, veri setindeki örnek sayısıdır.
* **x(i)**, veri setindeki i. örneğin tüm niteliklerinin (etiketi hariç) değerlerini içeren bir vektördür.
* **y(i**), ise üstteki örneğin etiketidir. (Yani örnek için istenen çıktı değeri)
* **X**, veri setindeki tüm örneklerin tüm niteliklerini (etiketleri hariç) içeren bir matristir. Her satır bir örneğe aittir. ve **i.** satır ve **x(i)** vektörünün transpozuna **(x(i))T** eşittir.
* **h**, sistemimizin tahmin fonksiyonudur. Sisteme bir örneğe ait nitelik vektörü **x(i)** verildiğin çıktısı o örneğe ait tahmin **y~(i) = h(x(i))** edilen edilen bir değerdir. Tahmin hatası ise **y~(i) - y(i)** dir.
* **RMSE(X,h)** h hipotezinin bir örnekler seti için ölçülen maliyet fonksiyonudur.

**VARSAYIMLARI KONTROL EDİN**

Projemizin çıktısının ne için kullanılacağı önemlidir. Örneğin çıktımız kategori olarak kullanılacaksa, uyumsuzluk ortaya çıkacaktır. Bu yüzden sonucun da ne için kullanılacağını bilip buna göre hareket etmeliyiz.

**VERİLERİ TOPLA VE TANI**

Projemizin stabil çalışması ve kütüphane sürüm sorunlarıyla karşılaşmaması için sanal alan oluşturarak ilerlememiz gereklidir. Bknz: **Conda Virtual Environment**

Projemizi besleyecek olan veri setinin de sürekli güncel olması gerekir. Kullanım anında bir kere indirmek yerine, sunucu üzerinden sürekli beslenmesini sağlamak daha sağlıklı olacaktır.

Şimdi uygulamamızı geliştirmeye, kütüphaneler tanımlayarak başlayalım.

**import** os

**import** tarfile

**import** urllib

Veri setimizi indireceğimiz siteyi tanımlayalım.

DOWNLOAD\_ROOT = “ https://github.com/ageron/handson-ml2“

HOUSING\_PATH = **os.path.join**(“datasets”,"housing”)

HOUSING\_URL = DOWNLOAD\_ROOT + “datasets/housing/housing.tgz”

Veri setimiz “**.tgz**” uzantılı bir dosya olarak inecektir. Bunu klasöre çıkartmak için,

**def fetch\_housing\_data**(housing\_url = HOUSING\_URL, housing\_path = HOUSING\_PATH):

**os.makedirs**(housing\_path, **exist\_ok=True**)

tgz\_path = **os.path.join**(housing\_path, "housing.tgz")

**urllib.request.urlretrieve**(housing\_url, tgz\_path)

housing\_tgz = **tarfile.open**(tgz\_path)

housing\_tgz.**extractall**(**path** = housing\_path)

housing\_tgz.**close**()

Veri setimizi kullanıma hazırlamak için  nesnesine dönüştürmemiz gerekir.

**import** pandas **as** pd

**def load\_housing\_data**(housing\_path = HOUSING\_PATH)

csv\_path = **os.path.join**(housing\_path, “housing.csv)

**return** **pd.read\_csv**(csv\_path)

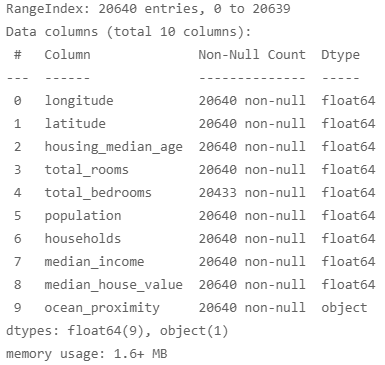
Verilerimizi **DataFrame** nesnesine dönüştürdüğümüze göre içeriği hakkında bilgi edinelim. İlk beş satırı görmek için **.head()** metodunu kullanırız.

housing = load\_housing\_data()

housing.**head()**

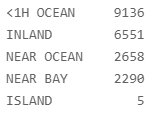
İçeriği hakkında ayrıntılı bilgileri görmek için **.info()**

housing.**info()**



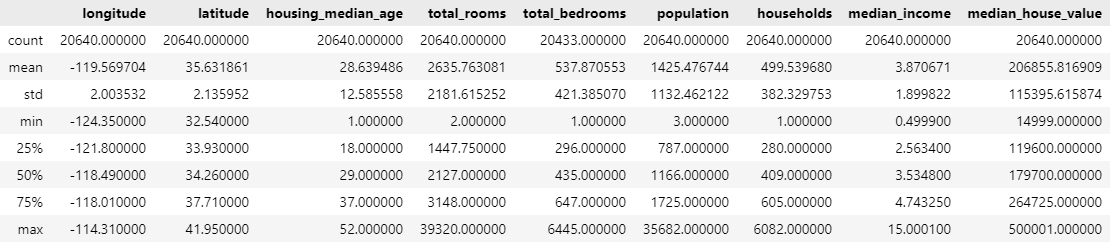
Verilerimizin **float64** ve **object** tiplerinden oluştuğunu görüyoruz. Object veri türü, kategorik verilerin olduğunu belirtir. Kategorik verinin içeriğini görmek için **.value\_counts()** metodunu kullanalım.

housing[“ocean\_proximity”].**value\_counts()**



**.describe()** metodu, **NaN** değerleri göz ardı ederek, numerik içeriklerle ilgili bize bilgiler verecektir.

housing.**describe**()

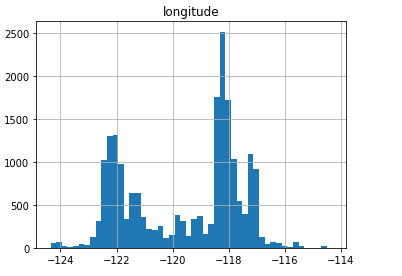


Verilerimizi daha iyi inceleyebilmek için histograma aktaralım. Çizim için **matplotlib.pyplot** kütüphanesine ait **.hist**, göstermek için de **.show** metotlarını kullanalım.

**import** matplotlib.pyplot **as** plt

housing.**hist**(**bins** = 50, **figsize** = (20,15))

plt**.show**()



**TEST VERİ SETİNİ OLUŞTURMA**

Algoritmamızı eğitmek için elimizde bulunan verilerin bir kısmını test verisi olarak ayırırız. Genelde bu oran %80 - %20 şeklinde olup, veri setinin büyüklüğüne göre değişmektedir.

**Scikit-Learn** kütüphanesini kullanarak veri setimizin %20’sini eğitim için ayıralım.

**from** sklearn.model\_selection **import** train\_test\_split

train\_set, test\_set = train\_test\_split(housing, **test\_size** = 0.2, **random\_state** = 42)

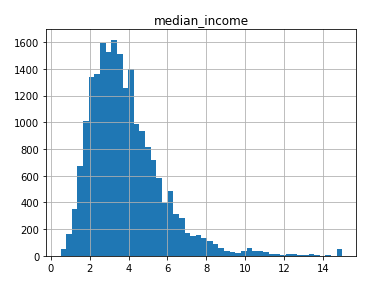
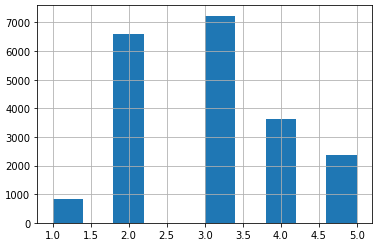
**train\_test\_split** metonu kolay yoldan veri setimizi bölmemizi sağlar. **random\_state** üretilen eğitim ve test endekslerinin rastgeleliğini kontrol eder. 42 yazmasının bir anlamı yoktur.

Verilerimizin tabakada bulunan her değere karşılık örnek bulundurması, tahmin açısından önemlidir. Yoksa sonuçlar yanlı olacaktır. Tabaka dağılımına ait veriler yetersiz ise mevcutu **pd.cut()** metodunu ile genişleterek dengeli bir dağılım sağlayabiliriz.

housing[“income\_cat”] = **pd.cut**(housing[“median\_income”], **bins**  = [0., 1.5, 3.0, 4.5, 6., **np.inf**],

**labels** = [1, 2 , 3, 4, 5])

Hologramda gördüğümüz üzere, yanlı olacağını düşündüğümüz **median\_income** verisinin dağılımını değiştirerek **income\_cat** verisini oluşturduk. **Labels** ile 1’den 5’ e kadar kategorileri oluşturup, **bins** ile kategorlerin sınırlarını belirledik.

İki türlü veri örnekleme yöntemi vardır.

1. **Rastgele Örnekleme** : Sınıflarda bulunan verilerin alt kümeye seçilme olasılığı eşittir.
2. **Tabakalı Örnekleme** : Sınıflardaki verilerin, veri kümesinin boyutu göz önüne alındığında genel popülasyonun yüzdesine dayalı olarak gruplara seçilmesidir.

Oluşturduğumuz income\_cat kategorisini kullanarak tabakalı örnekleme yapalım. Bunun için **StratifiedShuffleSplit** metodunu kullanacağız.Test için verimizin %20’sini ayıracağız.

**from** sklearn.model\_selection **import** StratifiedShuffleSplit

split = **StratifiedShuffleSplit**(**n\_splits** = 1, **test\_size** = 0.2, **random\_state** = 42)

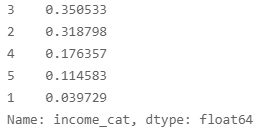
**for** train\_index, test\_index **in** split.split(housing, housing[“income\_cat”]):

strat\_train\_set = housing.**loc**[train\_index]

strat\_test\_set = housing.**loc**[test\_index]

Çalıştığını test etmek için elemanların toplam sayısını, test eleman sayısına bölerek, oranlarını kontrol edelim.

strat\_test\_set[“income\_cat”].**value\_counts()** / **len**(strat\_test\_set)



Şimdi **income\_cat** I kaldırıp, veri setimizi ilk durumuna getirelim.

**for** set\_ **in** (strat\_train\_set, strat\_test\_set):

set\_**.drop**(“income\_cat”, **axis** = 1, **inplace** = **TRUE**)

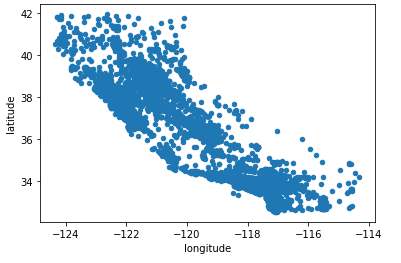
**FİKİR SAHİBİ OLMAK İÇİN VERİYİ GÖRSELLEŞTİR VE KEŞFET**

Keşif işlemini her zaman eğitim verisi üzerinden yapmamız gerekmektedir. Veri boyutumuz fazla büyükse, küçük bir parçası üzerinde de çalışabiliriz.

Şimdi enlem ve boylam bilgilerinin bulunduğu veri setimizin bir kopyasını alıp, görselleştirelim.

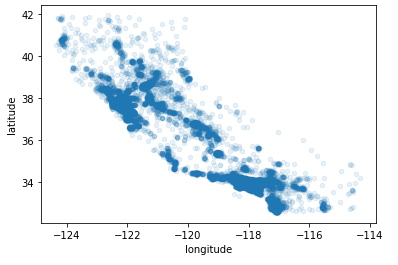
hosuing = strat\_train\_set.**copy()**

housing.**plot**(**kind** = “scatter”, **x** = “longitude”,  **y** = “latitude”)



Grafimizin anlatmak istediklerini anlamak için üzerinde biraz oynama yapalım. Önce **alpha** değeri ile yoğunluğunu ayarlayacağız. Sonra daire **population** a göre, **median\_house\_value** bilgisini renklendireceğiz.

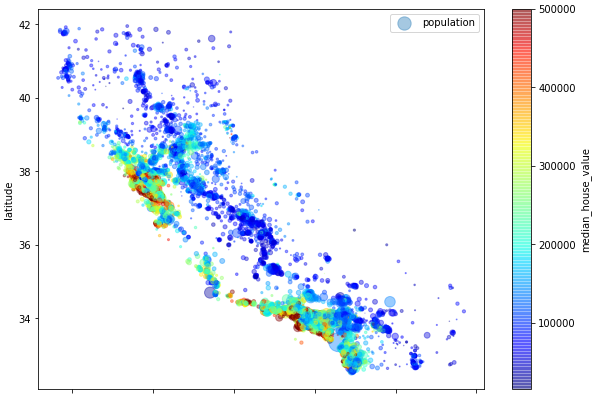
housing.**plot**(**kind** = “scatter”, **x** = “longitude”,  **y** = “latitude”, **alpha** = 0.1)



housing.**plot**(**kind** = “scatter”, **x** = “longitude”,  **y** = “latitude”, **alpha** = 0.1, **s** = housing[“population”] / 100,

**label** = “population”, **figsize** = (10,7), **c** = “median\_house\_value”, **cmap** = **plt.get\_cmap**(“jet”),

**colorbar** = **TRUE**)



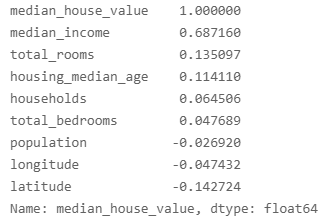
**s = housing["population"] / 100**  , population dairelerinin boyutunu ayarlar. 100 yerine 10 olsaydı daireler daha büyük olurdu.

**KORERASYONLARIN BULUNMASI**

Korerasyon, verilerin birbirileriyle olan bağlantının oranını verir. -1 ve 1 arasında hesaplanır, mutlak değer ne kadar büyükse bu iki veri o kadar birbiriyle ilişkilidir.1’e yaklaşırken güçlü ve pozitif korelasyon, -1’ e yaklaştıkça güçlü ve negatif korelasyon, 0’a yaklaşırken doğru bir korelasyon olmadığı anlamına gelir. **corr()** metodu ile **median\_house\_valueb** verisinin standart korelasyon katsayısını hesaplayalım.

corr\_matrix = housing**.corr**()

corr\_matrix[“median\_house\_value”].**sort\_values**(**ascending** = **FALSE**)

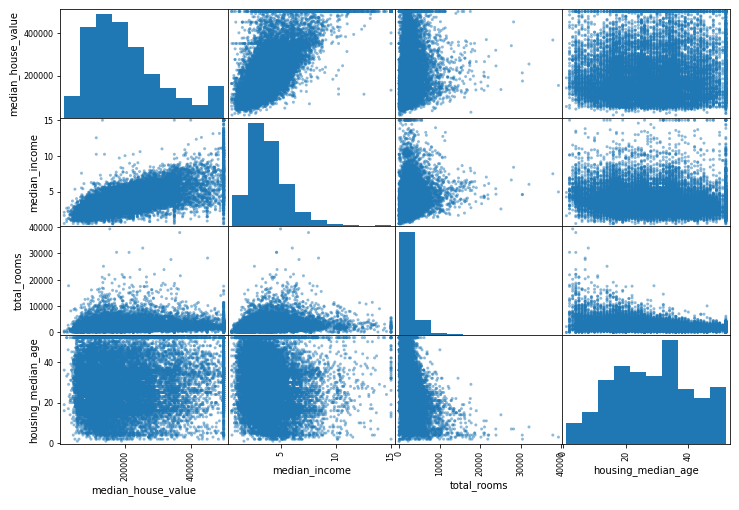


Korerasyon sadece doğrusallıkları ölçer. Doğrusal olmayanları göz ardı eder. Nitelikler arası korelasyonun olup olmadığını **scatter\_matrix()** metoduyla öğreniriz. Bir sayısal niteliğin diğer nitelikler ile olan ilişkisini histogram yardımı ile gösterir. Elimizde 11 adet nitelik olduğundan 112 = 121 çizim bekleriz.

**from** pandas.plotting **import** scatter\_matrix

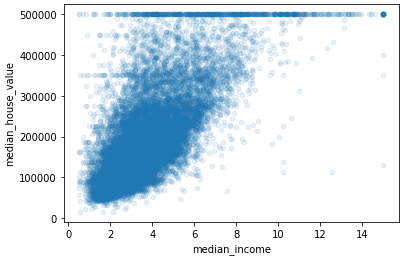
attributes = [“median\_house\_value”, “median\_income”, “total\_rooms”, “housing\_median\_age”]

**scatter\_matrix**(housing[attributes], **figsize** = (12,8))



Daha yakından bakmak için sadece **median\_house\_value** ile **median\_income** arasındaki korelasyonu çizelim.

housing.**plot**(**kind** = “scatter, **x**=”median\_income”, **y**=”median\_house\_value”, **alpha** = 0.1)



Histograma göre korelasyon çok güçlü. Noktaların üst kısım toplandığı görülmektedir. Bunlar yeniden öğrenmeyi engelleyeceğinden, bu bölgeleri kaldırmalıyız.

**NİTELİK KOMBİNASYONLARININ DENENMESİ**

Korerasyon grafiğinde gördüğümüz sorunları dönüştürerek çözmeye çalışacağız. Makine öğreniminde bize verilen bilgileri kullanarak farklı kombinasyonlar oluşturabiliriz. Örneğin elimizdeki **total\_rooms** ve **houseolds** bilgileriyle **rooms\_per\_housebold** verisini bulabiliriz.

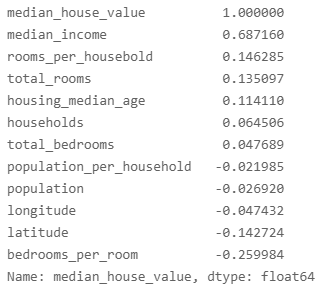
housing[“**rooms\_per\_housebold**”] = housing[“**total\_rooms**”] / housing[“**households**”]

housing[“**bedrooms\_per\_room**”] = housing[“**total\_bedrooms**”] / housing[“**total\_rooms**”]

housing[“**population\_per\_household**”] = housing[“**population**”] / housing[“**households**”]

corr\_matrix = housing.**corr()**

corr\_matrix[“median\_house\_value”].**sort\_values**(**ascending** = **FALSE**)



Yeni Korerasyon da **median\_house\_value** ve **bedrooms\_per\_room**’ un **total\_bedrooms** ve **total\_rooms**’ a oranla daha ilişkili olduğunu görüyoruz. Yani oran düşükse evler daha pahalı olmaktadır.

**MAKİNE ÖĞRENİM ALGORİTMASI İÇİN VERİLERİ HAZIRLAMAK**

Verileri kullanıma hazırlamak için sonra da kullanabileceğimiz fonksiyonlar yazmak bize kolaylık sağlayacaktır. İşleme başlamadan önce eğitim setimizin bir kopyasını alalım.

housing = strat\_train\_set.**drop**(“median\_house\_value”, **axis** = 1)

housing\_labels = strat\_train\_set[“median\_house\_value”].**copy()**

**VERİ TEMİZLEME**

Verilerimizde farklı ya da kayıp değerler olabilir. Örneğin **total\_bedrooms** verimiz de 20640 değer olmasını beklerken, 20433 değer bulunmaktadır. (Bknz: **housing.info()** ) Bunu çözmek, eksik değerleri tamamlamak için Scikit-Learn kütüphanesinin **SimpleImputer** sınıfını kullanacağız.

**from** sklearn.impute **import** SimpleImputer

imputer = **SimpleImputer**(**strategy** = “median”)

Sadece sayısal niteliklerde çalışabildiğinden, “**ocean\_proximity**” sayısal niteliğini sildiğimiz bir kopya oluşturacağız.

housing\_num = housing.**drop**(“ocean\_proximity”, **axis** = 1)

Sadece sayısal değerlerin bulunduğu verimize, imputer fonksiyonunu uygulayalım.

imputer.**fit**(housing\_num)

**ımputer**, her niteliğin medyan değerini hesaplayarak, **statistics\_** somut değişkeninde tutmaktadır. Devasa veriler ile çalıştığımızda, hangisinde eksik veri var görmemiz zorlaşır. O yüzden **imputer** fonksiyonunu tüm sayısal verilere uygulamak, daha sağlıklı olacaktır.

Şimdi eğitmiş olduğumuz **ımputer**’I kullanarak eğitim veri setimizde bulunan eksik değerleri, öğrenilen medyan değerleriyle değiştirelim.

X = imputer.**transform**(housing\_num)

Dönüştürdüğümüz nitelikler bir **Numpy** dizisi olacaktır. Tekrar Pandas - **DataFrame**’ e dönüştürmek istersek:

housing\_tr = pd.**DataFrame**(X, **columns** = housing\_num.**columns**, **index** = housing\_num**.index**)

**SCIKIT-LEARN TASARIMI**

* **Kestiriciler**

Veri setinden yola çıkarak, bazı parametreleri kestirebilen nesnelerdir. Örneğin **ımputer** bir kestiricidir. Kestirme fonksiyonu **.fit()** metodu ile uygulanır ve sadece bir veri setini parametre olarak alır. İhtiyaç duyulan diğer tüm parametrelere **hiperparametre** denir.

* **Dönüştürücüler**

Aynı zamanda veri setini dönüştürebilen kestiricilere, dönüştürücü denir. Dönüşüm **transform()** metodu ile uygulanır. Dönüşüm, **ımputer** da olduğu gibi öğrenilen parametrelere dayanır. Tüm dönüştürücüler, **fit\_transform()** metodunasahiptir. Bu metod, **fit()** ve **transform()** metotlarının yaptığı işi tek seferde yapar.

* **Tahminciler**

Tahmin yapabilen bazı kestiricilere tahminci denir. **LinearRegression()** modeli bir tahmincidir. Tahmincilerde **predict()** metodu bulunur ve yeni örnekleri alarak, ilişkili tahminlerle geri döndürür. Test veri setinin tahmin kalitesini ölçmek için de **scrore()** metodu kullanılır.

* **Kontrol Edilebilirlik**

Değişkenlere ait hiperparametrelere ve somut değişkenlere dışarıdan ulaşabiliriz.(Bknz: **imputer.-strategy**) Kestiricilerin öğrenilen parametrelerine “\_” ile dışarıdan ulaşabiliriz. (Bknz: **imputer.statistics\_**)

* **Kompozisyon**

Tekrar tekrar kullanabiliriz.

**METİNSEL VE KATEGORİK NİTELİKLERLE UĞRAŞMAK**

Şimdi **ocean\_proximity** verimizi inceleyelim. Sınırlı sayıda olası değeri olup, her biri bir kategoriyi temsil

etmektedir. Kategorik verimize yakından bakalım.

housing\_cat = housing[[“**ocean\_proximity**”]]

housing\_cat.**head**(10)



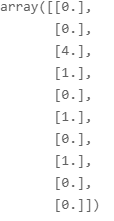
Çoğu makine öğrenimi algoritması sayısal verilerle çalışmayı tercih eder. Bu yüzden metinsel kategorilerimizi sayısal veriye dönüştürreceğiz. Bunun için Scikit-Learn kütüphanesinden **OrdinalEncoder**’I kullanacağız.

**from** sklearn.preprocessing **import** OrdinalEncoder

ordinal\_encoder = **OrdinalEncoder()**

housing\_cat\_encoded = ordinal\_encoder.**fit\_transform**(housing\_cat)

housing\_cat\_encoded[:10]



Sayısal sonuçlardan bir şey anlamayabiliriz. İlk halinde olduğu gibi listelemek istersek **categories\_** dönüş değişkenini kullanmamız gerekir. Bu her kategorik nitelik için bir boyutlu kategori dizisi içeren liste döndürecektir.

ordinal\_encoder.**categories\_**



Bu gösterimde ile ilgili önemli bir sorunumuz bulunmaktadır. Algoritma birbirine yakın iki değerin, birbirine uzak iki değere göre daha benzer olduğunu varsaymaktadır. Fakat bizde olan veriye göre **1H OCEAN** ile **NEAR OCEAN** sıralamada uzak olmalarına rağmen birbirilerine daha çok benzemektedir.

Bunu çözümü ise her kategori için ikili bir nitelik oluşturmaktır. Kategori değeri **1H OCEAN** olduğunda niteliğin 1 (değilse 0) olması, kategori **INLAND** olduğunda başka bir niteliğin 1 (değilse 0) olması ve diğeri için de böyle devam etmesidir. Buna **bir-elemanı-bir-kodlama** denir. Bu dönümüm için **OneHotEncoder** sınıfını kullanacağız.

**from** sklearn.preprocessing **import** OneHotEncoder

cat\_encoder = **OneHotEncoder()**

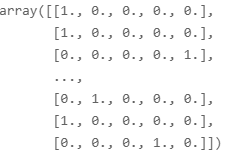
housing\_cat\_1hot = cat\_encoder.**fit\_transform**(housing\_cat)

housing\_cat\_1hot

Çıktımız **Numpy** dizisi yerine, **SciPy** **seyrek matrisi**’dir. Seyrek matris kullanımının yararı, 0 olan değerlerin yer tutmasını engellemesidir. Binlerce matristen oluşan bir işlemde çok işimize yaramaktadır.

2 boyutlu dizileri kullanabileceğimiz gibi, istersek **toarray**() metodu ile NumPy dizisine çevirebiliriz.

housing\_cat\_1hot.**toarray**()



Kategorik bir nitelik, içerisinde çok sayıda kategori sahipse, bir-elemanı-bir-kodlama fazla sayıda girdi özniteliği oluşturacaktır. Bu da performansta düşüklüğe yol açabilir. Böyle durumlarda kategorik girdi ile kullanışlı sayısal niteliklerin yerlerini değiştirebiliriz. Alternatif olarak, her kategoriyi öğrenebilir **gömülme** olarak adlandırılan düşük boyutlu bir vektör ile değiştirebiliriz. Her kategorinin gösterimi eğitim sırasında öğrenilmedir ve bu **gösterim öğrenmenin** bir örneğidir.

**ÖZEL DÖNÜŞTÜRÜCÜLER**

Scikit-Learn dönüştücüleri kullanışlı olasada, bazı durumlarda kendi özel dönüştürücümüzü yazmamız gerekebilir. **Sklearn.base** kütüphanesinden, **BaseEstimator** ve **TransformerMixin** sınıflarını aktardıktan sonra, bir sınıf oluşturup, **fit()**, **transform()**, **fit\_transform()** metotlarını oluşturmalıyız.

**ÖZNİTELİKLERİN ÖLÇEKLENMESİ**

Sayısal girdilerin farklı ölçeklerde olması, bazı durumlar dışında verimliliği düşürmektedir. Tüm nitelikleri aynı ölçeğe getirmek için iki yöntem vardır. **Küçük-en büyük ölçekleme (max - min scaling)** ve **standardizasyon**.

Max-min scaling, en çok kullanılan yöntem olup, değerleri 0-1 arasında yeniden ölçeklendirir. Bunun için Scikit-Learn kütüphanesinde **MixMaxScalar** sınıfını kullanırız. **feature\_range** isimli bir hiperparametreye de sahip olup, 0-1 dışında aralık belirlememizi sağlar.

Standardizasyon, değerleri belli bir aralıkta olmaya zorlamaz. Bu örneğin 0-1 arası çalışan sinir ağları için sorun olmaktadır. Fakat aykırı değerlerden de çok az etkilenir.

Ölçeklendirme tüm veri setine değil, sadece eğitim setine uygulanmalıdır.

**DÖNÜŞÜM İLETİM HATLARI**

Veriyi modeller için hazırlamak adına birkaç ön-işlemden geçirmemiz gerekebilir. Bunu Scikit-Learn, **Pipeline** metodunu kullanarak yaparız. İlgili metod verinin peşpeşe seri biçimde bir takım işleme girmesini sağlar.

**from** sklearn.pipeline **import** Pipeline

**from** sklearn.preprocessing **import** StandardScaler

num\_pipeline = **Pipeline**([

("impurt", **SimpleImputer**(**strategy**="median")),

("attribs\_adder", **CombineAttributesAdder**()),

("std\_scaler", **StandardScaler**()),

])

housing\_num\_tr = num\_pipeline.fit\_transform(housing\_num)

Yapılacak adımlarda, son kestirici dışındaki diğer tüm kestiriciler aynı anda dönüştürücü olmalıdır. Yani **fit\_transform()** metoduna da sahip olmalılar. Her çıktı bir sonrakine parametre olarak gönderilir.

housing\_num\_tr = num\_pipeline.**fit\_transform**(housing\_num)

Şu ana kadar kategorik ve sayısal sütunları ayrı ayrı ele aldık. Şimdi tüm sütunları bir kerede alıp her sütuna uygun dönüşümü yapan bir dönüştürücü yazalım. Bunun için **ColumnTransformer**’I kullanacağız.

**from** sklearn.compose **import** ColumnTransformer

num\_attribs = **list**(housing\_num)

cat\_attribs = ["ocean\_proximity"]

full\_pipeline = **ColumnTransformer**([

("num", num\_pipeline, num\_attribs),

("cat", **OneHotEncoder**(), cat\_attribs),

])

housing\_prepared = full\_pipeline.**fit\_transform**(housing)

Dönüştürücü kullanırken alternatif olarak kullanılmayan sütunları kaldırmak istersek **drop**, dokunulmamasını istiyorsak **passthrough** dizisini belirtebiliriz. Varsayılan olarak, kullanılmayan sütunlar kaldırılacaktır. Bu sütunları farklı bir şekilde ele almak istersek **reminder** hiperparametresini atlarız.

**MODEL SEÇİMİ ve EĞİTİMİ**

**LinearRegression**, bağımlı ve bağımsız değerler arasındaki ilişkiyi ifade eden doğrusal bir fonksiyondur. Şimdi bu fonksiyon ile modelimizi eğitelim.

**from** sklearn.linear\_model **import** LinearRegression

lin\_reg = **LinearRegression()**

lin\_reg.**fit**(housing\_prepared, housing\_labels)

Eğitimden sonra modelimizi birkaç örnek ile deneyelim.

some\_data = housing.**iloc**[:5]

some\_labels = housing\_labels.**iloc**[:5]

some\_data\_prepared = full\_pipeline.**transform**(some\_data)

Eğitim veri setinde bulunan modelin **RMSE (Kök Ortalama Kare Hata)** skorunu Scikit-Learn kütüphanesinde bulunan **mean\_squared\_error()** fonksiyonu yardımıyla ölçelim.

from sklearn.metrics import mean\_squared\_error

housing\_predictions = lin\_reg.predict(housing\_prepared)

lin\_mse = mean\_squared\_error(housing\_labels, housing\_predictions)

lin\_rmse = np.sqrt(lin\_mse)

lin\_rmse



Bu durum eğitim verisinin yetersiz uydurmanın bir sonucudur. İyi tahminler yapabilmek için yeterli bilgiyi sağlayamadığı ya da modelin yeterince güçlü olmadığı anlamına gelir. Yetersiz uydurmanın çözümlerinden biri olan daha güçlü model seçimi yeterli olacaktır. Fakat öncesinde karmaşık bir model kullanarak nasıl bir sonuç alacağımızı görelim.

**DecisionTreeRegressor**, Karar ağacı regresyonu özetle şu işi yapıyor: Bağımsız değişkenleri bilgi kazancına göre aralıklara ayırıyor. Tahmin esnasında bu aralıktan bir değer sorulduğunda cevap olarak bu aralıktaki (eğitim esnasında öğrendiği) ortalamayı söyleyiveriyor. Bu sebeple karar ağacı regresyonu diğer regresyon modelleri gibi sürekli değil, kesiklidir. Yani belli bir aralıkta istenen tahminler için aynı sonuçları üretir.

from sklearn.tree import DecisionTreeRegressor

tree\_reg = DecisionTreeRegressor(random\_state=42)

tree\_reg.fit(housing\_prepared, housing\_labels)

Modelimizi eğittiğimize göre, değerlendirebiliriz.

housing\_predictions = tree\_reg.predict(housing\_prepared)

tree\_mse = mean\_squared\_error(housing\_labels, housing\_predictions)

tree\_rmse = np.sqrt(tree\_mse)

tree\_rmse



Bu hiç hata olmadığı anlamına gelmektedir. Bu durumda model aşırı uydurma gösteriyor diyebiliriz. Emin olmadan test veri setine dokunmak istemediğimizden, model doğrulaması için eğitim veri setinin bir bölümünü kullanacağız.

ÇAPRAZ DOĞRULAMA KULLANARAK DAHA İYİ DEĞERLENDİRME

Karar ağacı modelimizi değerlendirmenin bir diğer yolu da **train\_test\_split()** fonksiyonudur. Veri setini karıştırarak, eğitim ve test verisi olacak şekilde iki parçaya ayırır.

Çok daha iyi bir alternatif ise, Scikit-Learn’ nün **K- parça çapraz doğrulama** (**Cross-Validation**) özelliğidir. K değeri, eğitim veri setinin rastgele kaç ayrı alt kümeye ayırıp, kaç kere eğitilip değerlendirileceğini belirtir. Bu yöntem küçük ila orta boyutlu veri kümelerinde harika çalışır.

Bir maliyet fonksiyonundan ziyade, fayda fonksiyonuna ihtiyaç duymaktadır. Dolayısıyla skor fonkiyonu MSE’nin tam tersi şekildedavranır. Bu yüzden, aşağıda bulunan kod öbeği karekökü hesaplamadan önce **-scores** hesaplar.

**from** sklearn.model\_selection **import** cross\_val\_score

scores = **cross\_val\_score**(tree\_reg, housing\_prepared, housing\_labels,

scoring="neg\_mean\_squared\_error", cv=10)

tree\_rmse\_scores = np.**sqrt**(-scores)

Sonuçlara bakacak olursak:

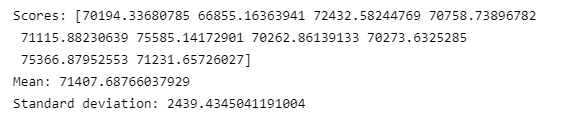
def display\_scores(scores):

print("Scores:", scores)

print("Mean:", scores.mean())

print("Standard deviation:", scores.std())

display\_scores(tree\_rmse\_scores)



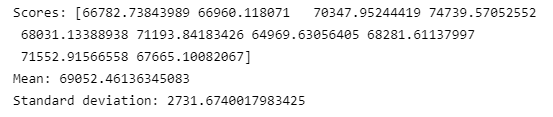
Sonuç doğrusal bağlanım modelinden daha kötü durumdadır. Çapraz doğrulama modelin performansının yanında modelin kesinliğini de ölçer (Standart Sapma). Emin olmak için aynı skorları doğrusal bağlanım modeli için de hesaplayalım.

lin\_scores = cross\_val\_score(lin\_reg, housing\_prepared, housing\_labels,

scoring="neg\_mean\_squared\_error", cv=10)

lin\_rmse\_scores = np.sqrt(-lin\_scores)

display\_scores(lin\_rmse\_scores)



Gördüğümüz gibi karar ağacı aşırı uydurduğundan doğrusal bağlanım modelinden daha kötü bir performans göstermiştir.

Son bir model daha deneyelim: **RandomForestRegressor**. Rastgele ormanlar algoritması, birden fazla karar ağacını kullanarak daha uyumlu modeller üreterek isabetli tahminlerde bulunmaya yarayan bir regresyon modelidir. Karar ağaçlarını kullandığı için kesiklidir. Yani belli bir aralıkta istenen tahminler için aynı sonuçları üretir. Birçok alt modelin üzerine model inşa etmek **topluluk öğrenme** olarak adlandırılır ve modelleri bir adım öteye taşımanın çok güzel bir yoludur.

**from** sklearn.ensemble **import** RandomForestRegressor

forest\_reg = **RandomForestRegressor**(n\_estimators=100, random\_state=42)

forest\_reg.**fit**(housing\_prepared, housing\_labels)

housing\_predictions = forest\_reg.**predict**(housing\_prepared)

forest\_mse = **mean\_squared\_error**(housing\_labels, housing\_predictions)

forest\_rmse = np.**sqrt**(forest\_mse)

forest\_rmse



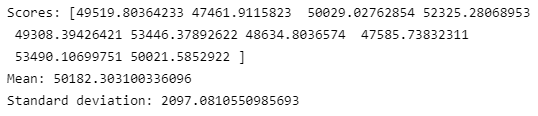
**from** sklearn.model\_selection **import** cross\_val\_score

forest\_scores = **cross\_val\_score**(forest\_reg, housing\_prepared, housing\_labels,

scoring="neg\_mean\_squared\_error", cv=10)

forest\_rmse\_scores = np.**sqrt**(-forest\_scores)

display\_scores(forest\_rmse\_scores)



Sonuçlar çok iyi olsada, eğitim veri setine ait skorlar, doğrulama veri setine ait olan skorlardan oldukça düşük ve bu durumda modelin halen aşırı uydurduğunu işaret etmektedir. Bunu ortadan kaldırmak için olası çözümler modeli basitleştirmek, kısıtlamak ya da daha fazla eğitim verisi bulmaktır.

Denediğimiz kaydederek hangisine daha çok ihtiyacımız olduğunu görmemizde bize fayda sağlayacaktır. Böylece modeller arası skorları da karşılaştırabileceğiz. Scikit-Learn modellerini **Python Pickle** modülü ya da **NumPy** dizilerini etkin şekilde serileştiren **Joblib** kütüphanesini kullanarak kolayca kayıt edebiliriz.

my\_model = forest\_reg

**import** joblib

joblib.**dump**(my\_model, "my\_model.pkl")

#...

my\_model\_loaded = joblib.**load**("my\_model.pkl")

**MODELE İNCE AYAR**

**HIPERPARAMETRE OPTİMİZASYONU**

Bir makine öğrenmesi algoritması için belirlenen başarı metriğine göre en uygun hiperparametre kombinasyonunu bulma işlemidir. hiperparametre optimizasyonu için farklı yöntemler geliştirilmiştir. **GridSearcCV** ve **RandomizedSearchCV** bu yöntemlerdendir.

**GridSearchCV**, modelde denenmesi istenen hiperparametreler ve değerleri için bütün kombinasyonlar ile ayrı ayrı model kurulur ve belirtilen metriğe göre en başarılı hiperparametre seti belirlenir. Küçük veri setlerinde ve sadece birkaç tane hiperparametre denenmek istendiğinde çok iyi çalışır. Tek yapmamız gereken, deneyeceğimiz hiperparametreleri ve alacağı değerleri belirtmektir. Alacağı değer “**n\_estimators**” konusunda bilginiz yoksa 10 ve katlarını girmek hassas aramalar için faydalı olacaktır. Örneğin örnek kod öbeği, RandomForestRegressor modeli için en iyi hiperparametre değerleri kombinasyonunu arar.

**from** sklearn.model\_selection **import** GridSearchCV

param\_grid = [

{'n\_estimators': [3, 10, 30], 'max\_features': [2, 4, 6, 8]},

{'bootstrap': [False], 'n\_estimators': [3, 10], 'max\_features': [2, 3, 4]},

]

forest\_reg = **RandomForestRegressor**(random\_state=42)

grid\_search = **GridSearchCV**(forest\_reg, param\_grid, cv=5,

scoring='neg\_mean\_squared\_error',

return\_train\_score=True)

grid\_search.fit(housing\_prepared, housing\_labels)

**RandomizedSearchCV**, rastgele olarak bir hiperparametre seti seçilir ve **cross-validation** ile model kurularak test edilir. Belirlenen hesaplama süresi limitine ya da iterasyon sayısına ulaşıncaya kadar bu adımlar devam eder.

**from** sklearn.model\_selection **import** RandomizedSearchCV

from scipy.stats import randint

param\_distribs = {

'n\_estimators': randint(low=1, high=200),

'max\_features': randint(low=1, high=8),

}

forest\_reg = **RandomForestRegressor**(random\_state=42)

rnd\_search = **RandomizedSearchCV**(forest\_reg, param\_distributions=param\_distribs,

n\_iter=10, cv=5, scoring='neg\_mean\_squared\_error', random\_state=42)

rnd\_search.fit(housing\_prepared, housing\_labels)

Maaliyet açısından düşünüldüğünde gerçek hayatta kendine **GridSearchCV** den daha fazla yer bulur.