XGBoost Algorithm

Atıl Samancıoğlu

1 Giriş

XGBoost (Extreme Gradient Boosting), Gradient Boosting algoritmasının geliştirilmiş bir versiyonudur. Özellikle büyük veri setlerinde ve karmaşık modellerde üstün performansı ile bilinir.

Gradient Boosting ile başlıca farkları:

- **Pruning (Budama):** XGBoost, ağaçları *post-pruning* ile optimize eder.
- **Regularization:** L1/L2 cezaları sayesinde overfitting riskini azaltır.
- **Hız:** Paralelleştirme ve hafıza optimizasyonları içerir.
- **Advanced Gain Hesabı:** Gain ve similarity kavramları ile split kalitesi ölçülür.
- **Sigmoid + Log-Odds:** Sınıflandırmada residual hesaplaması log-odds ile yapılır.

Bu dökümanda XGBoost'un temel çalışma mantığını **örneklerle ve tüm hesaplamalarıyla** açıklıyoruz.

2 Örnek Veri Seti

Aşağıdaki veri seti, Salary ve Credit Score kullanarak kredi kartı onayı tahmini yapmaktadır:

ID	Salary (X1)	Credit Score (X2)	Approval (Y)
1	30K	Low	0
2	40K	Low	0
3	50K	Medium	1
4	60K	High	1
5	70K	Medium	1
6	80K	High	1
7	90K	High	1

Table 1: Örnek Veri Seti (Sınıflandırma Problemi)

3 Adım 1: Base Model (Log-Odds Hesabı)

Gradient Boosting'de olduğu gibi XGBoost'ta da ilk adım olarak tüm veri seti için bir **base model** kurulur. Ancak burada, sınıflandırma problemi olduğu için **log-odds** kavramı devreye girer.

Log-Odds Nedir?

Log-odds, sınıflandırmada olasılıkları logaritmik ölçekte ifade etmek için kullanılan bir kavramdır. Normalde sınıflandırmada her gözlem için bir olasılık (örneğin %70 onay alma şansı gibi) tahmin ederiz. Bu olasılığı bir logaritmik dönüşüme sokarsak **log-odds** elde ederiz. Formülü şöyledir:

$$\text{Log-Odds} = \log\left(\frac{\text{Pozitif Oranı}}{\text{Negatif Oranı}}\right)$$

Bizim veri setimizde:

- Pozitif sınıf (Y=1): 5 gözlem
- Negatif sınıf (Y=0): 2 gözlem

Bu durumda:

$$\text{Log-Odds} = \log\left(\frac{5/7}{2/7}\right) = \log(2.5) \approx 0.916$$

Bu değer tüm gözlemler için başlangıç **F(0)** değeri olarak atanır. Yani model henüz hiçbir öğrenme yapmadan önce tüm gözlemler için başlangıç skoru aynıdır.

Sigmoid Fonksiyonu Neden Kullanılır?

Log-odds değeri doğrudan bir olasılığı temsil etmez. Log-odds değerini **0 ile 1 arasına sıkıştırmak** ve bir olasılığa çevirmek için **sigmoid (lojistik) fonksiyonunu** kullanırız. Sigmoid'in formülü şöyledir:

$$p = \frac{1}{1 + e^{-F(x)}}$$

Burada:

- $F(x)$ bizim log-odds değerimizdir.
- Çıkan p değeri, onay alma olasılığını gösterir.

Örneğimizde:

$$p = \frac{1}{1 + e^{-0.916}} \approx 0.714$$

Bu demek oluyor ki, model henüz hiç öğrenme yapmadan tüm gözlemler için **%71.4** olasılıkla onay verir.

Bu Adımın Amacı

- Model henüz **hiçbir özellik** kullanmadan genel bir tahminde bulunuyor.
- Bu başlangıç, modelin üzerine yeni karar ağaçları inşa edeceği **ilk yapı taşıdır**.
- Sigmoid ile olasılığa dönüştürerek gerçek hayatta *"bu müşteri onay alır mı?"* sorusuna cevap üretiriz.

Özet: XGBoost sınıflandırma problemlerinde tahminler **log-odds** formatında başlatılır, sonra her adımda **sigmoid** ile gerçek olasılık elde edilir. Bu, Gradient Boosting'den en büyük farklardan biridir çünkü klasik GB doğrudan residual ile çalışırken burada **log-odds** dünyasında işlem yapılır.

Neden Gradient Boosting'den Farklı?

Gradient Boosting algoritması, sınıflandırma problemlerinde seçilen kayıp fonksiyonuna göre davranır. Eğer **log-loss** kullanılırsa, klasik Gradient Boosting de log-odds ve sigmoid dönüşümleri kullanmak zorundadır. Yani bu özellikler sadece XGBoost'a özel değildir.

Ancak **XGBoost**, bu süreci çok daha kararlı ve optimize edilmiş şekilde uygular:

- Log-loss fonksiyonu temel alınır.
- İlk tahminler log-odds formatında yapılır.
- Sigmoid fonksiyonu ile olasılıklar hesaplanır.
- Her iterasyonda yalnızca gradyan değil, ikinci türev (**hessian**) de hesaplanarak daha hassas güncellemeler yapılır.

Önemli Not

Klasik Gradient Boosting, log-loss kullandığında log-odds ve sigmoid hesaplamaları yapar. XGBoost'un farkı, bu işlemleri sistematik hale getirmesi, regularization ve hessian gibi ek mekanizmalarla daha güçlü ve dengeli bir optimizasyon sunmasıdır.

Örnek: Sigmoid ve Log-Odds Olmasaydı?

Klasik Gradient Boosting kullansaydık, base model olarak örneğin ortalama sınıf değeri ile başlardık:

$$\bar{Y} = \frac{1+1+1+1+1+0+0}{7} = \frac{5}{7} \approx 0.714$$

Bu durumda tüm gözlemler için başlangıç tahmini:

$$p = 0.714$$

Residual:

$$R_i = Y_i - 0.714$$

Bu aslında XGBoost'ta da ortaya çıkan residual'la benzer gözükse de, bu yaklaşım optimizasyon sürecinde **ikinci türev bilgisi** (Hessian) barındırmaz. Ayrıca 0.714 tahmini, log-loss fonksiyonuna göre optimal değildir.

Oysa XGBoost bu tahmini log-odds cinsinden yapar:

$$F_0 = \log\left(\frac{0.714}{1 - 0.714}\right) = \log\left(\frac{5}{2}\right) \approx 0.916$$

Bu log-odds değeri:

- Daha stabil gradyanlar üretir,
- Sigmoid sayesinde düzgün olasılık skalasına çevrilebilir,
- Loss fonksiyonu log-loss olduğu için model doğru yönde optimize olur.

Gradyan ve Hessian Nedir?

Gradyan (1. türev): Hatanın ne kadar büyük olduğunu gösterir. **Hessian** (2. türev): Bu hatanın değişim hızını gösterir.
XGBoost bu ikisini birlikte kullanır:

$$\text{Güncelleme} = -\frac{g}{h + \lambda}$$

Bu sayede:

- Tahminler çok yanlışsa: Büyük düzeltme yapılır,
- Tahminler zaten iyiyse: Düzeltme az yapılır,
- Model daha dengeli ve kararlı öğrenir.

Örnek:
Gerçek $Y = 1$, model tahmini $p = 0.9$ Gradyan: $g = 0.9 - 1 = -0.1$
Hessian: $h = 0.9 \cdot 0.1 = 0.09$
Yani hata küçük, düzeltme de küçük olur.

Kullandığımız Loss Fonksiyonu: Log-Loss

XGBoost sınıflandırma problemlerinde log-loss fonksiyonunu optimize eder:

$$L = -[Y \log(p) + (1 - Y) \log(1 - p)]$$

Bu loss fonksiyonunun gradyanı:

$$g = p - Y$$

İkinci türevi (hessian):

$$h = p(1 - p)$$

Bu sayede her adımda daha hassas güncellemeler yapılabilir.

4 Adım 2: İlk Residual (Negatif Gradyan) Hesaplama

İlk adımda tüm gözlemler için **base model (log-odds)** ile bir başlangıç tahmini yapmıştık. Şimdi sırada, bu tahminlerin **ne kadar hatalı olduğunu** belirlemek var. XGBoost sınıflandırma problemlerinde bu hatayı hesaplamak için **negatif gradyan** kullanır.

Negatif Gradyan (Residual) Nedir?

Negatif gradyan, modelimizin yaptığı hatayı gösterir. Formül şu şekildedir:

$$R_i = Y_i - p_i$$

Burada:

- Y_i = Gerçek etiket (0 veya 1)
- p_i = Mevcut tahmin edilen olasılık (sigmoid ile hesaplanan değer)

Bu formülün amacı şudur:

- Eğer R_i negatifse: Model **çok yüksek tahmin** yapmış demektir (örneğin gerçek sınıf 0 ama model 0.71 tahmin etmiş).
- Eğer R_i pozitifse: Model **yetersiz tahmin** yapmış demektir (örneğin gerçek sınıf 1 ama model 0.71 tahmin etmiş).

Hesaplama

İlk adımdaki tüm tahminlerimiz yaklaşık olarak 0.714 idi (çünkü başlangıç modelimiz tüm gözlemler için aynı tahmini yapmıştı). Hesaplamalar:

ID	Y (Gerçek)	p (Tahmin)	Residual (R_1)
1	0	0.714	-0.714
2	0	0.714	-0.714
3	1	0.714	+0.286
4	1	0.714	+0.286
5	1	0.714	+0.286
6	1	0.714	+0.286
7	1	0.714	+0.286

Table 2: İlk Residual (Negatif Gradyan) Hesaplamaları

Bu Sonuçlar Ne Anlatıyor?

Şimdi dikkat edelim:

- ID 1 ve 2 için: Gerçek değer **0**, model ise **0.714** tahmin etmiş. Yani model **çok iyimser** davranmış, hatası büyük ve negatif olmuş (-0.714).
- Diğer ID'ler için: Gerçek değer **1**, model ise sadece **0.714** tahmin etmiş. Yani model **yetersiz kalmış**, hatası **+0.286**.

Bu residual değerleri bize şunu söylüyor:
- Negatif residual'lar: *"Bu veri noktalarında modelin tahminleri fazla yüksek. Yeni decision tree burada **azaltıcı** bir etkide bulunmalı."* - Pozitif residual'lar: *"Bu veri noktalarında modelin tahmini yetersiz. Yeni decision tree burada **artırıcı** bir etkide bulunmalı."*

Karar Ağacı Ne Öğrenecek?

Bu residual değerleri bizim "etiketimiz" haline gelir. Yani bir sonraki adımda inşa edeceğimiz **decision tree**, artık gerçek sınıfları (0 veya 1) değil, bu **residual (hata)** değerlerini tahmin etmeye çalışacak.

Örneğin:

- ID 1 ve 2 için: Tree yaklaşık **-0.714** tahmin etmeye çalışacak. - Diğerleri için: Tree yaklaşık **+0.286** tahmin etmeye çalışacak.

Bu Ne İşe Yarıyor?

- İlk decision tree'miz **hataları öğrenerek** mevcut tahminlerin daha iyi hale gelmesini sağlar.
- Residual'ların bu şekilde ayrılması sayesinde tree, *"hangi gözlemler için model çok fazla tahmin yaptı, hangileri için yetersiz kaldı?"* bilgisini öğrenir.
- Bu süreç, modelin adım adım hatalarını azaltarak daha iyi tahminler yapmasını sağlar.

Not: Henüz decision tree'yi **kurmadık**, sadece **etiketlerimizi (residual'ları)** hazırladık. Bu etiketler, bir sonraki adımda kurulacak olan tree'nin **ne öğreneceğini** belirler.

5 Adım 3: Split, Similarity ve Gain Hesaplama

Önceki adımda tüm gözlemler için residual (negatif gradyan) hesaplamıştık. Şimdi bu residual'ları kullanarak **hangi split'in en iyi sonucu vereceğini** belirlemek istiyoruz.

Ne Yapıyoruz?

Burada henüz decision tree'yi kurmadık. XGBoost'ta karar ağacı oluştururken her seferinde **hangi özellikten ve hangi eşikten bölünme yapılırsa en yüksek bilgi kazancı (gain) sağlanır** bunu hesaplarız. Bu adımın amacı:

- Farklı split'ler için **similarity** ve **gain** hesaplamak,
- En yüksek gain sağlayan split'i belirlemek,
- En iyi split belirlendikten sonra tree'yi inşa etmeye başlamak.

Örnek Split: Salary $\leq 50K$?

Bu adımda örnek olarak **Salary** özelliğine göre bir split öneriyoruz:

- **Left node:** ID 1, 2, 3 (Salary $\leq 50K$)
- **Right node:** ID 4, 5, 6, 7 (Salary $> 50K$)

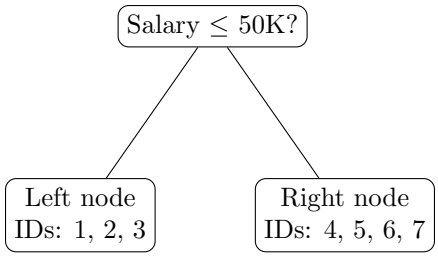

Figure 1: Önerilen Split: Salary $\leq 50K$

Not: Bu sadece bir öneri. Normalde tüm potansiyel split'ler (örneğin Salary ve Credit Score için farklı eşikler) tek tek denenir ve hepsinin gain değeri hesaplanır. En yüksek gain sağlayan split seçilir.

Hangi Formülleri Kullanıyoruz?

Similarity:

$$S = \frac{(\sum R_i)^2}{\sum p_i(1-p_i) + \lambda}$$

Burada:

- R_i = residual değeri
- p_i = mevcut tahmin (sigmoid sonucu)
- λ = regularization katsayısı (varsayım: $\lambda = 1$)

Gain:

$$\text{Gain} = S_{\text{left}} + S_{\text{right}} - S_{\text{root}}$$

Gain, split'in faydasını ölçer. Büyük gain = daha iyi ayrım demektir.

Hesaplamalar:

- **Left node (ID 1, 2, 3):**

$$\sum R = -0.714 - 0.714 + 0.286 = -1.142$$

$$\sum p(1-p) = 3 \times 0.714 \times 0.286 \approx 0.612$$

$$S_{\text{left}} = \frac{(-1.142)^2}{0.612 + 1} \approx 0.809$$

- **Right node (ID 4, 5, 6, 7):**

$$\sum R = 4 \times 0.286 = 1.144$$

$$\sum p(1-p) = 4 \times 0.714 \times 0.286 \approx 0.816$$

$$S_{\text{right}} = \frac{(1.144)^2}{0.816 + 1} \approx 0.702$$

- **Root node:**

$$\sum R = 0$$

$$S_{\text{root}} = 0 \quad \text{(çünkü kök düğümde tek bir grup var)}$$

- **Gain:**

$$\text{Gain} = 0.809 + 0.702 - 0 = 1.511$$

Peki Neden Bu Hesaplar?

Karar ağacı split yaparken **hedefimiz:** veri setini öyle bölmek ki her alt grup (**leaf node**) daha saf (pure) hale gelsin, yani residual'lar aynı yönde olsun (hepsi + veya hepsi - gibi).

Bu yüzden:

- **Similarity:** Bir node'un kendi içinde ne kadar güçlü tahmin yaptığına bakar.
- **Gain:** Bir bölmenin toplam bilgi kazancını ölçer.

Eğer bu Salary split'i dışında başka split'ler de denenseydi, onların da gain değerleri hesaplanırdı. **En yüksek gain sağlayan split seçilir** ve ancak o zaman decision tree'nin **ilk split'i** kesinleşir.

Özet

- Şu an **decision tree'yi kurmadık**, sadece **hangi split'in daha iyi olduğu** hesaplanıyor. - Split önerisi: Salary $\leq 50K$ - Hesaplanan gain yüksek (**1.511**), bu nedenle bu split büyük ihtimalle seçilecektir.

Bu hesaplardan sonra artık gerçekten tree'yi kurma aşamasına geçeceğiz (bir sonraki adımda).

6 Adım 4: İlk Decision Tree ve Tahmin Güncelleme

Bir önceki adımda en yüksek **gain** değerine sahip split olarak **Salary** $\leq 50K$ seçildi. Bu split ile decision tree'nin yapısı belirlendi ancak henüz bu node'ların **leaf değerleri** (yani her dalın çıktısı) hesaplanmamıştı. Şimdi, bu ağacı tamamlayarak her leaf'in hangi değeri üreteceğini hesaplıyoruz.

Leaf Değerleri Nedir?

Her leaf değeri, o leaf'e düşen gözlemlerin residual hatalarını (negatif gradyanlarını) minimize edecek şekilde hesaplanır. XGBoost'ta bu değer şu formülle bulunur:

$$V = \frac{\sum R_i}{\sum p_i(1-p_i) + \lambda}$$

- $\sum R_i$: Leaf'e düşen tüm gözlemlerin residual toplamı,
- $\sum p_i(1-p_i)$: Leaf'e düşen tüm gözlemlerin olasılık varyansı,
- λ: Regularization katsayısı (örneğimizde $\lambda = 1$).

Bu formül sayesinde, her leaf modeli kendi içindeki hataları en iyi şekilde düzeltecek çıktıyı öğrenir.

Split Sonrası Leaf Grupları

Bir önceki split sonucunda oluşan gruplar:

- **Left node:** ID 1, 2, 3 (Salary $\leq 50K$),
- **Right node:** ID 4, 5, 6, 7 (Salary $> 50K$).

Hesaplamalar

- **Left leaf:**

$$V = \frac{-1.142}{0.612 + 1} \approx -0.709$$

- **Right leaf:**

$$V = \frac{1.144}{0.816 + 1} \approx +0.631$$

Bu değerler, modelimizin **ilk decision tree ($\mathbf{H}_1$)** için hesapladığı leaf değerleridir.

İlk Decision Tree Görseli

Aşağıda, kurduğumuz ilk decision tree'yi görebilirsiniz. Her leaf'in içinde hesapladığımız değerler yer alıyor; bu değerler modelin yeni tahminlerine katkıda bulunacak.

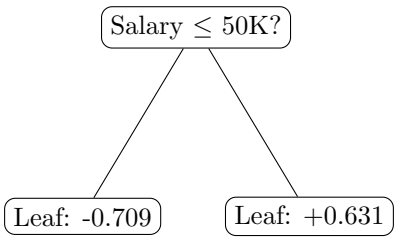

Figure 2: İlk Decision Tree (H_1)

Tahmin Güncelleme

Şimdi modelimizi güncelleme zamanı: her gözlem için başlangıçtaki $\mathbf{F}_0$ değerine, kendi düştüğü leaf'in değerini **learning rate** ($\alpha = 0.1$) ile çarparak ekliyoruz.

Örneğin ID 1 için:

$$F_1 = 0.916 + 0.1 \times (-0.709) = 0.916 - 0.071 = 0.845$$

Bu yeni tahmin hâlâ **log-odds** cinsindedir. Olasılık tahmini için sigmoid fonksiyonunu uygularız:

$$p = \frac{1}{1 + e^{-0.845}} \approx 0.699$$

Güncellenmiş Tahminler

Aşağıdaki tabloda tüm gözlemler için **H_1 sonrası güncellenmiş tahminler** yer alıyor. Sol leaf'teki gözlemler negatif bir düzeltme alırken, sağ leaf'teki gözlemler pozitif bir iyileştirme alıyor.

ID	F_0	Leaf	F_1	Yeni p
1	0.916	-0.709	0.845	0.699
2	0.916	-0.709	0.845	0.699
3	0.916	-0.709	0.845	0.699
4	0.916	+0.631	0.979	0.727
5	0.916	+0.631	0.979	0.727
6	0.916	+0.631	0.979	0.727
7	0.916	+0.631	0.979	0.727

Table 3: Güncellenmiş Tahminler (H_1 Sonrası)

Neden Bu Adımı Yaptık?

Bu güncelleme ile model artık daha **hassas** hale geliyor. İlk tahmin (F_0) tüm gözlemler için aynıydı; H_1 sayesinde artık gözlemler Salary değerine göre farklılaştırılıyor. Bu, modelin adım adım öğrenmesini ve daha iyi sonuçlar üretmesini sağlıyor.

7 Adım 5: Yeni Residual (Negatif Gradyan) Hesaplama

Bir önceki adımda, modelimiz ilk decision tree'nin yardımıyla tahminlerini güncelledi. Artık tahminlerimiz **ilk halinden farklı** ve daha iyi hale geldi. Şimdi sırada yeni residual'ları (negatif gradyanları) yeniden hesaplamak var.

Neden Bu Adımı Yapıyoruz?

Gradient Boosting'in temel felsefesi, her iterasyonda modelin yaptığı hataları gidermeye çalışmaktır. İlk ağaç hataları bir miktar düzeltti; ancak model hâlâ mükemmel değil. Yeni decision tree'lerin öğrenebilmesi için **güncellenmiş tahminler** üzerinden yeni hatalar hesaplamamız gerekiyor.

Bu residual'lar **bir sonraki decision tree'nin öğrenmesi gereken şeyi gösterir.**

Formül

Her gözlem için negatif gradyan (residual):

$$R_2 = Y - p$$

Burada:

- Y: Gerçek sınıf etiketi (0 veya 1),
- p: Güncellenmiş tahmin (sigmoid ile hesaplanan olasılık),
- R_2: Yeni residual (hata) değeri.

Hesaplamalar

Örneğin **ID 1** için:

$$R_2 = 0 - 0.699 = -0.699$$

Tüm gözlemler için hesaplayalım:

ID	Gerçek Y	Yeni p	$\mathbf{R_2}$
1	0	0.699	-0.699
2	0	0.699	-0.699
3	1	0.699	+0.301
4	1	0.727	+0.273
5	1	0.727	+0.273
6	1	0.727	+0.273
7	1	0.727	+0.273

Table 4: Yeni Residual (R_2) Hesaplamaları

Bu Değerler Ne Anlama Geliyor?

- Negatif residual'lar (örneğin ID 1 ve 2) demek ki model **fazla tahmin yapıyor** (yanlış şekilde pozitif sınıfa yakın tahminler üretmiş).
- Pozitif residual'lar (örneğin ID 3–7) modelin **eksik tahmin yaptığını** ve bu gözlemler için daha yüksek bir tahmin üretmesi gerektiğini gösteriyor.

Bu residual değerleri, yeni kurulacak decision tree'ye hangi örneklerde hata olduğunu ve nasıl düzeltmesi gerektiğini öğretir. Örneğin:

- Sol node (ID 1, 2, 3) yine düşük residual'lar içeriyor, bu da bu grubun hâlâ zorlandığını gösterir.
- Sağ node (ID 4–7) ise daha küçük hatalarla daha doğru tahminler yapıyor ama hâlâ iyileştirilebilir.

Sonraki Adım: Yeni Decision Tree Kurulumu

Bu yeni residual'lar R_2 artık bir **yeni decision tree ($\mathbf{H_2}$)** tarafından öğrenilecek. Süreç tamamen aynıdır:

- Yeni bir split önerilir (örneğin yine Salary $\leq 50K$),
- Similarity ve gain hesapları yapılır,
- Yeni leaf değerleri hesaplanır,
- Ana model yeniden güncellenir.

Bu süreç **M** kadar iterasyon boyunca devam eder ve model her yeni adımda biraz daha iyi hale gelir.

Neden Bu Tekrarlama?

XGBoost'un başarısı, bu iteratif **düzeltme** yaklaşımından gelir:

- İlk başta çok genel bir model ile başlıyoruz,
- Sonra hataları hedef alan karar ağaçları oluşturuyoruz,
- Her adımda biraz daha **iyileşiyoruz**,
- Overfitting riskini azaltmak için küçük learning rate (α) ve regularization (λ) kullanıyoruz.

Bu yüzden yeni residual hesaplaması **modelin motorudur;** hatalar belirlenir, öğrenilir ve düzeltilir.

8 Ek: Cover ve Regularization

XGBoost'u Gradient Boosting'ten ayıran en önemli güçlü yanlardan biri **regularization** (düzenleme) ve **pruning** (budama) özellikleridir. Bu, modelin aşırı öğrenmesini (overfitting) engeller ve daha sade, güçlü modeller oluşturur.

- **Lambda (λ):** Gain ve leaf değerlerini hesaplarken λ terimi eklenir. Leaf değerleri:
$$V = \frac{\sum R}{\sum p(1-p) + \lambda}$$
λ ne kadar büyükse leaf değerleri o kadar **baskılanır**, yani model aşırı uca kaçmaz. Örneğin $\lambda = 0$ olursa klasik Gradient Boosting gibi çalışır; $\lambda > 0$ ise regularization sağlar.
- **Gamma (γ):** Bir split yapılmadan önce **minimum gain** şartı koyar. Eğer bir split'in gain'i γ'dan küçükse, XGBoost o split'i **reddeder**. Bu, çok küçük katkı yapan bölünmeleri engeller. Örneğin $\gamma = 0.1$ ise, gain'i 0.05 olan bir split yapılmaz.

- **Cover:**

$$\text{Cover} = \sum p(1-p)$$

  Cover, bir node'un **ne kadar bilgi taşıdığını** gösterir. Eğer cover çok küçükse (örneğin tüm tahminler %0 veya %100'e çok yakınsa), o node artık fazla bilgi taşımaz ve split yapılması önerilmez. Bu mekanizma otomatik budama sağlar.

Örnek: Bir leaf için $\sum R = 2$, $\sum p(1-p) = 1$, $\lambda = 1$ olsun.
Leaf değeri:

$$V = \frac{2}{1+1} = 1$$

$\lambda = 10$ olsaydı:

$$V = \frac{2}{1+10} \approx 0.18$$

Yani λ arttıkça leaf değeri küçülür ve model daha temkinli olur.

Neden? Bu ek parametreler sayesinde model, gereksiz yere karmaşıklaşmaz; sade ama güçlü kalır. XGBoost'un Gradient Boosting'den en büyük farkı budur: yalnızca hataları değil, aynı zamanda **model karmaşıklığını** da optimize eder.

9 Final XGBoost Formülü

XGBoost'un genel formülü şu şekildedir:

$$F(x) = F_0 + \sum_{m=1}^{M} \alpha_m H_m(x)$$

ve nihai tahmin olasılığı:

$$p(x) = \frac{1}{1 + e^{-F(x)}}$$

Burada:

- F_0: Base model (log-odds),
- $H_m(x)$: Her adımda kurulan decision tree,
- α_m: Learning rate (α) çarpanı,
- M: Iterasyon sayısı.

Her yeni tree, önceki hataları biraz daha düzeltir ve sigmoid ile nihai olasılık tahmini yapılır.

10 XGBoost Regresyon: Mantık ve Adımlar

XGBoost yalnızca sınıflandırma için değil, aynı zamanda **regresyon problemleri** için de güçlü bir algoritmadır. Sınıflandırmada log-odds ve sigmoid ile olasılıklar hesaplanırken, regresyonda doğrudan **sürekli değerler** tahmin edilir. Bu yüzden formüller daha basittir ama süreç aynıdır.

Örnek Veri Seti (Regresyon Problemi)

Aşağıdaki veri seti, Salary ve Credit Score kullanarak kişinin **harcama miktarını (Spending)** tahmin etmeye çalışır:

ID	Salary (X1)	Credit Score (X2)	Spending (Y)
1	30K	Low	2000
2	40K	Low	3000
3	50K	Medium	4000
4	60K	High	5000
5	70K	Medium	6000
6	80K	High	7000
7	90K	High	8000

Table 5: Örnek Veri Seti (Regresyon Problemi)

11 Adım 1: Base Model (Ortalama Hesabı)

Regresyon problemlerinde XGBoost (ve klasik Gradient Boosting) ilk adımda tüm veri setinin **ortalamasını** tahmin ederek başlar.

$$F_0 = \frac{\sum Y}{N} = \frac{2000 + 3000 + 4000 + 5000 + 6000 + 7000 + 8000}{7} = 5000$$

Neden? Başlangıçta model hiçbir şeyi bilmiyor; bu yüzden en iyi tahmin **ortalama değeri** kullanmaktır. Bu, tüm gözlemler için başlangıç tahminidir.

12 Adım 2: İlk Residual (Hata) Hesaplama

Regresyonda residual hesaplaması çok basittir:

$$R_i = Y_i - F_0$$

Bu, her gözlemin **ne kadar sapma yaptığını** gösterir.
Ne Anlatıyor?

- ID 1–3: Model fazla tahmin yapmış (negatif residual),
- ID 5–7: Model az tahmin yapmış (pozitif residual),
- ID 4: Tam 5000 olduğu için hata yok.

ID	Gerçek (Y)	Tahmin ($\mathbf{F}_0$)	Residual ($\mathbf{R}_1$)
1	2000	5000	-3000
2	3000	5000	-2000
3	4000	5000	-1000
4	5000	5000	0
5	6000	5000	+1000
6	7000	5000	+2000
7	8000	5000	+3000

Table 6: İlk Residual (Hata) Hesaplamaları

13 Adım 3: Split ve Gain Hesaplama

Karar ağacını kurmak için yine **hangi split'in en iyi sonucu vereceğini** hesaplarız. Örneğin, **Salary** $\leq 60K$? diye bir split önerelim:

- **Left node:** ID 1, 2, 3, 4
- **Right node:** ID 5, 6, 7

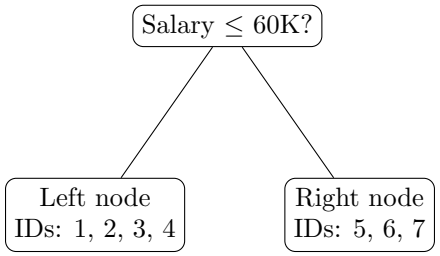

Figure 3: Önerilen Split (Regresyon)

Gain Hesabı

Regresyonda gain hesaplamak için en basit haliyle şu formül kullanılır (MSE bazlı):

$$\text{Gain} = \frac{(\sum R_{\text{left}})^2}{N_{\text{left}}} + \frac{(\sum R_{\text{right}})^2}{N_{\text{right}}} - \frac{(\sum R_{\text{total}})^2}{N_{\text{total}}}$$

Hesaplamalar:
- **Left node:**

$$\sum R = -3000 - 2000 - 1000 + 0 = -6000, \quad N = 4$$

- **Right node:**

$$\sum R = +1000 + 2000 + 3000 = +6000, \quad N = 3$$

- **Root node:**

$$\sum R = 0, \quad N = 7$$

$$\text{Gain} = \frac{(-6000)^2}{4} + \frac{(6000)^2}{3} - \frac{(0)^2}{7} = \frac{36,000,000}{4} + \frac{36,000,000}{3} = 9,000,000 + 12,000,000 = 21,000,000$$

Yorum: Bu oldukça yüksek bir gain değeri! Diğer split'lerle karşılaştırılıp en iyisi seçilir.

14 Adım 4: İlk Decision Tree ve Tahmin Güncelleme

Bu split'in leaf değerleri şöyle hesaplanır:

- **Left leaf:**

$$V = \frac{\sum R}{N} = \frac{-6000}{4} = -1500$$

- **Right leaf:**

$$V = \frac{+6000}{3} = +2000$$

Ağaç görseli:

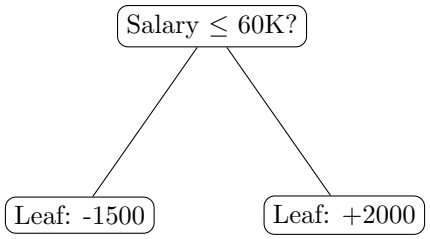

Figure 4: İlk Decision Tree (Regresyon)

Güncelleme: $\alpha = 0.1$
Örneğin ID 1 için:

$$F_1 = 5000 + 0.1 \times (-1500) = 5000 - 150 = 4850$$

Tüm tahminler:

15 Adım 5: Yeni Residual Hesaplama

$$R_2 = Y - F_1$$

Örneğin ID 1 için:

$$2000 - 4850 = -2850$$

Tam tablo:

ID	$\mathbf{F}_0$	Leaf	$\mathbf{F}_1$
1	5000	-1500	4850
2	5000	-1500	4850
3	5000	-1500	4850
4	5000	-1500	4850
5	5000	+2000	5200
6	5000	+2000	5200
7	5000	+2000	5200

Table 7: Güncellenmiş Tahminler (H_1 Sonrası, Regresyon)

ID	Gerçek Y	$\mathbf{F}_1$	$\mathbf{R}_2$
1	2000	4850	-2850
2	3000	4850	-1850
3	4000	4850	-850
4	5000	4850	+150
5	6000	5200	+800
6	7000	5200	+1800
7	8000	5200	+2800

Table 8: Yeni Residual Hesaplamaları (Regresyon)

16 Regularization ve Final Formül

Regresyonda da λ (L2 regularization) leaf değerlerinin hesaplanmasında kullanılır:

$$V = \frac{\sum R}{N + \lambda}$$

Diğer regularization parametreleri (γ, min split gain vs.) aynıdır.
Final formül:

$$F(x) = F_0 + \sum_{m=1}^{M} \alpha_m H_m(x)$$

Burada sigmoid yoktur çünkü direkt sayı tahmini yapılır.

17 Regresyon ve Sınıflandırma Farkları

- Sınıflandırmada log-odds + sigmoid, regresyonda doğrudan sayı tahmini yapılır.
- Gain ve similarity hesapları sınıflandırmada probabilistik, regresyonda MSE bazlıdır.

- Güncelleme adımları neredeyse aynıdır ama yorumlamalar farklıdır.

Sonuç: XGBoost regresyonda da sınıflandırmadaki gibi residual öğrenerek iteratif biçimde daha güçlü modeller kurar.

18 Sonuç

XGBoost:

- Gradient Boosting'in optimize edilmiş halidir.
- Gain, similarity ve regularization ile daha güçlüdür.
- Sigmoid ve log-odds yapısı ile sınıflandırmada üstün sonuç verir.
- Python'da `xgboost.XGBClassifier` ile kolayca uygulanır.