Gradient Boosting Algoritması

Atıl Samancıoğlu

1 Giriş

Makine öğrenmesinde tek bir model bazen yeterli olmayabilir. Özellikle karar ağaçları gibi "zayıf" modeller, karmaşık verileri iyi öğrenemez. İşte bu noktada **Boosting** adı verilen bir fikir devreye girer:

Boosting Nedir?

Birden fazla zayıf modeli art arda eğiterek, her birinin hatasını bir sonrakine öğretmek ve böylece güçlü bir model oluşturmaktır.

AdaBoost ile Farkı

AdaBoost'ta yanlış sınıflara verilen ağırlıklar arttırılırken, Gradient Boosting doğrudan tahmin hatalarını (residual) hedef alır. Ayrıca AdaBoost genellikle çok basit (tek seviyeli) ağaçlar kullanır, Gradient Boosting ise daha derin ağaçlarla çalışabilir.

—

2 Bölüm 1: Regresyon Problemi ile Gradient Boosting

Senaryo:

Bir şirkette çalışanların tecrübesi ve sertifika durumuna göre maaşlarını tahmin etmeye çalışıyoruz. Hedefimiz sayısal bir değeri (maaş) tahmin etmek, bu nedenle bu bir regresyon problemidir.

Örnek Veri Seti

Adım 1: Başlangıç Tahmini (Base Model)

Hiçbir model henüz kurulmamışken, veriye bakmadan yapılabilecek en iyi şey herkes için aynı değeri tahmin etmektir. Bu değer de genellikle hedef değişkenin ortalamasıdır:

ID	Tecrübe (Yıl)	Sertifika (Evet/Hayır)	Maaş (Y)
1	1	Hayır	40
2	3	Evet	60
3	5	Evet	80
4	7	Hayır	100

Table 1: Regresyon için örnek veri seti

$$F_0 = \frac{40 + 60 + 80 + 100}{4} = 70$$

Neden ortalama ile başladık?

Hiçbir şey bilmiyorsak, herkese ortalama maaşı tahmin etmek en mantıklı ve adil seçenektir.

Adım 2: İlk Hataların (Residual) Hesaplanması

İlk modelimiz herkes için 70 tahmin etti. Ama verideki gerçek maaşlar farklı. Bu farklılıklar bizim ilk hatalarımızdır.

$$R_1 = Y_i - F_0$$

ID	Gerçek Maaş	Tahmin (F_0)	Residual R_1
1	40	70	-30
2	60	70	-10
3	80	70	+10
4	100	70	+30

Table 2: İlk residual hesaplamaları

Bu adım ne işe yarıyor?

Hataları (residual) hesaplayarak, modelin hangi noktalarda başarısız olduğunu öğreniyoruz.

Adım 3: İlk Ağaç – Hataları Öğren

Bu residual'ları tahmin etmeye çalışan ilk karar ağacımızı kuruyoruz. Diyelim ki ağaç şöyle bir kural buldu:

$$\text{Tecrübe} < 4 \Rightarrow -20, \quad \text{Aksi halde} \Rightarrow +20$$

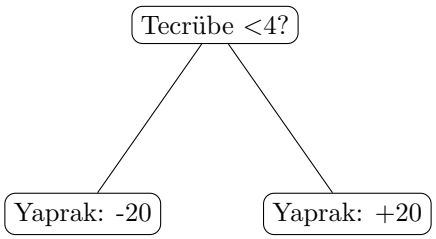

Figure 1: İlk karar ağacı (Residual tahmini)

Neden residual'ları öğreniyoruz?

Model, hatalardan öğrenir. Bu ağaç sadece hataları tahmin etmeye çalışıyor.

Adım 4: Tahminleri Güncelle

Şimdi bu ağacın çıktısını kullanarak ilk modelimizi güncelliyoruz. Ancak çıktıları direkt eklemiyoruz, önce bir öğrenme oranı ($\alpha = 0.1$) ile çarpıyoruz:

$$F_1 = F_0 + \alpha \cdot H_1$$

ID	F_0	Tree Output	Yeni Tahmin
1	70	-20	68
2	70	-20	68
3	70	+20	72
4	70	+20	72

Table 3: Güncellenmiş tahminler (F_1)

Öğrenme oranı (α) ne işe yarar?

Modelin küçük adımlarla öğrenmesini sağlar. Böylece aşırı öğrenmenin (overfitting) önüne geçilir.

Adım 5: Yeni Hatalar ve Yeni Ağaçlar

Modelimiz güncellendi. Ancak hâlâ tam isabetli tahmin yapamıyor. Şimdi tekrar hataları hesaplıyoruz:

$$R_2 = Y - F_1$$

Ve yeni residual'lara göre ikinci bir karar ağacı kuruyoruz. Bu süreç:

- Hata hesapla,
- Ağacı kur,
- Güncelle,

şeklinde devam eder.

Model neden tekrar tekrar ağaç kuruyor?

Çünkü her ağaç, bir öncekinin düzeltemediği hataları düzeltmeye çalışıyor. Bu zincirleme düzeltme ile model giderek daha isabetli hale geliyor.

3 Bölüm 2: Sınıflandırma Problemi ile Gradient Boosting

Senaryo:

Bir bankada müşterilere kredi kartı verilmeli mi? Karar verirken müşterinin:

- Geliri (bin TL),
- Kredi skoru (Düşük = 0, Orta = 1, Yüksek = 2)

bilgileri kullanılıyor.

Örnek Veri Seti

ID	Gelir	Kredi Skoru	Kart Onayı (Y)
1	30	0	0
2	40	0	0
3	50	1	1
4	60	2	1

Table 4: Sınıflandırma için örnek veri

Adım 1: Başlangıç Tahmini (Log-Odds)

Modelin başlangıçta hiçbir özelliğe (gelir, kredi skoru vs.) bakmadan, sadece verideki sınıf oranına göre genel bir tahminde bulunması gerekir.

Veri setimizde:

- 2 kişi kredi kartı almış ($Y = 1$),

- 2 kişi almamış ($Y = 0$).

Bu durumda pozitif sınıf oranı: $p = \frac{2}{4} = 0.5$

Ancak Gradient Boosting, sınıflandırmada doğrudan bu oranı kullanmaz. Çünkü modelin iç optimizasyonu log-odds (logaritmik oran) üzerinden çalışır. Bu yüzden bu oranı logaritmik forma dönüştürürüz:

$$F_0 = \log\left(\frac{p}{1-p}\right) = \log\left(\frac{0.5}{0.5}\right) = \log(1) = 0$$

Neden log-odds?

Modelin tahminlerini logaritmik ölçekte başlatmak, hem daha dengeli gradyanlar üretir hem de sigmoid ile düzgün bir olasılık elde etmemizi sağlar.

Adım 2: Sigmoid ile Olasılık

Şimdi elimizde her gözlem için bir başlangıç değeri var: $F_0 = 0$

Ama bu log-odds değeri bizim gerçek hayatta işimize yaramaz. Bizim ihtiyacımız olan şey: bu kişinin kredi kartı alma olasılığı kaç?

Bunun için log-odds'u sigmoid fonksiyonu ile %0–100 arası bir olasılığa çeviriyoruz:

$$p = \frac{1}{1 + e^{-F_0}} = \frac{1}{1 + e^0} = 0.5$$

Yani modelimiz başlangıçta her kişi için "bu kişi %50 ihtimalle kart alır" diyor.

Neden sigmoid?

Sigmoid, negatif sonsuzdan pozitif sonsuza kadar olan log-odds değerlerini, 0 ile 1 arasına sıkıştırır. Bu da onu olasılık üretmek için ideal yapar.

Adım 3: İlk Gradyan (Residual) Hesabı

Artık elimizde ilk tahminler var. Herkes için model $p = 0.5$ tahmin etti. Şimdi bu tahminlerin ne kadar hatalı olduğunu hesaplayalım.

Kredi kartı almayanlar için model çok iyimser davranmış, çünkü %50 ihtimal vermiş. Alanlar içinse tam tersi: model biraz daha kötümser kalmış.

İşte bu hatayı ölçmek için kullandığımız formül:

$$R_i = Y_i - p_i$$

- Gerçek değer $Y_i = 1$, tahmin $p_i = 0.5 \rightarrow$ hata $= +0.5$

- Gerçek değer $Y_i = 0$, tahmin $p_i = 0.5 \rightarrow$ hata = -0.5

Bu değer neden önemli?

Bu residual (gradyan), yeni kurulacak decision tree'nin öğrenmeye çalışacağı hedef haline gelir. Yani model hatalardan öğrenir.

Adım 4: İlk Ağaç (Split: Gelir <50?)

Bu adımda, bir karar ağacı kuruyoruz. Ancak bu ağaç doğrudan sınıf (0 veya 1) tahmini yapmıyor — az önce hesapladığımız **gradyan (residual)** değerlerini tahmin etmeye çalışıyor.

Split önerisi: Gelir <50?

Bu kuralla grupladığımızda:

- ID 1 ve 2 $\rightarrow$ Ortalama residual: -0.5
- ID 3 ve 4 $\rightarrow$ Ortalama residual: $+0.5$

Bu sayede ağaç, "kimler için tahminim çok fazlaydı, kimler için çok azdı?" sorusuna cevap vermeyi öğrenir.

Neden ağaç kurduk?

Ağaç, residual'ları öğrenerek tahmin hatalarının yönünü bulur. Sonraki adımda bu tahminlerle modelimizi güncelleyeceğiz.

Adım 5: Güncellenmiş Log-Odds Değerleri

Ağacın çıktısını doğrudan modele eklemiyoruz — önce küçük bir oran ($\alpha = 0.1$) ile çarpıyoruz. Bu oran, öğrenme oranıdır.

$$F_1 = F_0 + \alpha \cdot H_1$$

- ID 1,2 $\rightarrow$ F = 0 + 0.1 * (-0.5) = -0.05
- ID 3,4 $\rightarrow$ F = 0 + 0.1 * (+0.5) = +0.05

Neden küçük oranla ekliyoruz?

Modelin çok hızlı öğrenmesini istemiyoruz. Öğrenme oranı sayesinde model adım adım gelişir ve aşırı öğrenme (overfitting) riski azalır.

Adım 6: Yeni Olasılıklar (Sigmoid Uygula)

Yeni log-odds değerlerimiz var. Bunları tekrar sigmoid fonksiyonuna sokarak güncel olasılık tahminlerini elde ediyoruz:

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

- F = -0.05 → p = 0.487
- F = +0.05 → p = 0.512

Model artık küçük ama anlamlı bir adım attı: yüksek gelirli kişilere biraz daha yüksek, düşük gelirli kişilere biraz daha düşük olasılık vermeye başladı.

Neden tekrar sigmoid uyguladık?

Log-odds tahminlerini tekrar anlaşılır olasılıklara çevirmek için.

4 Sonuç

- Gradient Boosting, hataları adım adım düzelten güçlü bir tekniktir.
- Regresyon problemlerinde sayısal değer tahmini yapar.
- Sınıflandırmada log-odds + sigmoid ile olasılık tahmini yapılır.
- XGBoost, bu algoritmanın daha hızlı, düzenlemeli ve gelişmiş halidir.