KNN (K-nearest neighbours)

Teori

K-En Yakın Komşu (K-NN), hem sınıflandırma hem de regresyon problemlerinde kullanılan, **parametrik olmayan**, **gözetimli bir makine öğrenimi algoritmasıdır**. "Yönetim" terimi, daha çok K-NN algoritmasının nasıl çalıştığı, veriyi nasıl kullandığı ve karar mekanizmasının arkasındaki prensipleri ifade eder. Temelde, K-NN'in arkasındaki teori şuna dayanır: **Benzer örnekler, özellik uzayında birbirine yakındır ve bu nedenle aynı sınıfa (sınıflandırmada) veya benzer değere (regresyonda) sahip olma eğilimindedirler.**

1. K-NN'in Temel Felsefesi: "Yakın Çevre" Prensibi

K-NN'in temel çalışma prensibi oldukça basittir ve insan sezgisine dayalıdır: "Bana arkadaşını söyle, sana kim olduğunu söyleyeyim." Yani, bir veri noktasının özellikleri, onun en yakınındaki komşularının özelliklerine benzer olacaktır.

- **Tembel Öğrenci (Lazy Learner):** K-NN, "tembel öğrenci" (lazy learner) olarak adlandırılır çünkü **eğitim aşamasında aslında hiçbir şey öğrenmez veya bir model oluşturmaz.** Tüm hesaplamalar ve kararlar, yalnızca bir tahminde bulunma ihtiyacı doğduğunda (yani test aşamasında) yapılır. Bu, diğer modellerin (örneğin regresyon, destek vektör makineleri) eğitim sırasında karmaşık bir matematiksel fonksiyon veya karar sınırları öğrenmesinin aksinedir.

2. Algoritmanın Çalışma Adımları (Sınıflandırma İçin)

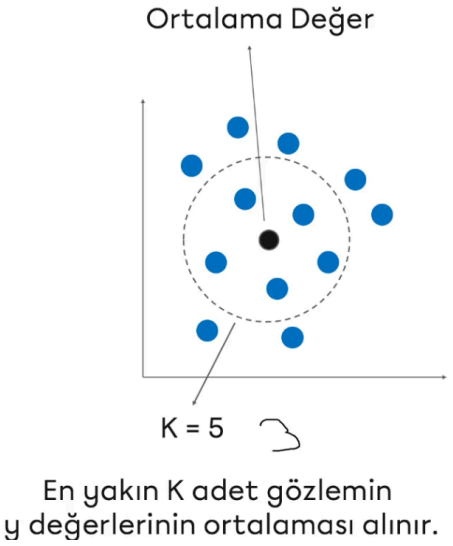
Bir test verisi noktası geldiğinde, K-NN aşağıdaki adımları izler:

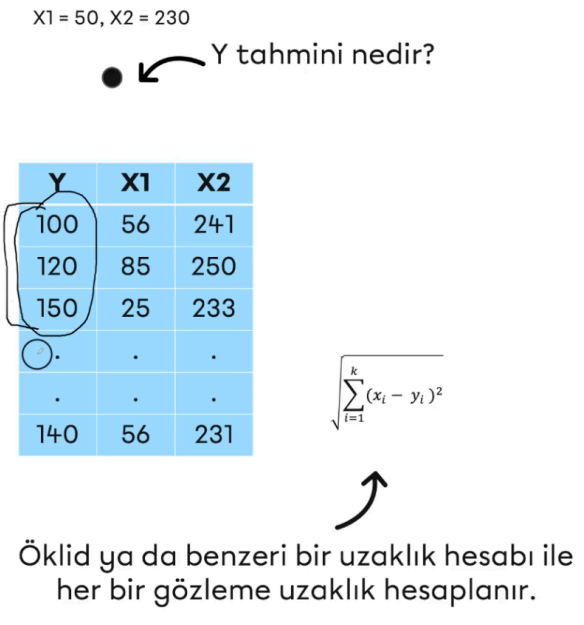
1. **Mesafe Hesaplama:** Yeni, bilinmeyen veri noktası ile eğitim setindeki **tüm** diğer veri noktaları arasındaki uzaklıklar hesaplanır. En yaygın kullanılan uzaklık ölçüsü **Öklid Mesafesi**'dir (Euclidean Distance), ancak Manhattan Mesafesi, Minkowski Mesafesi gibi başka ölçütler de kullanılabilir.
    - **Öklid Mesafesi:** İki nokta (x1,y1) ve (x2,y2) arasındaki mesafe: (x2−x1)2+(y2−y1)2

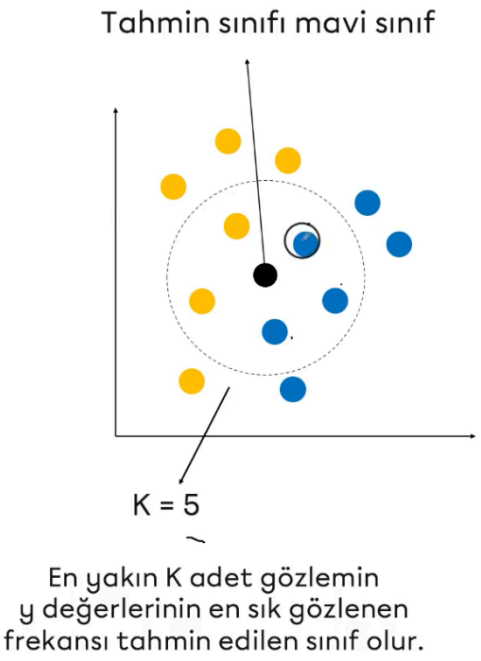
- Daha yüksek boyutlar için genelleştirilir: $\sum_{i=1}^{n}(p_i - q_i)^2$

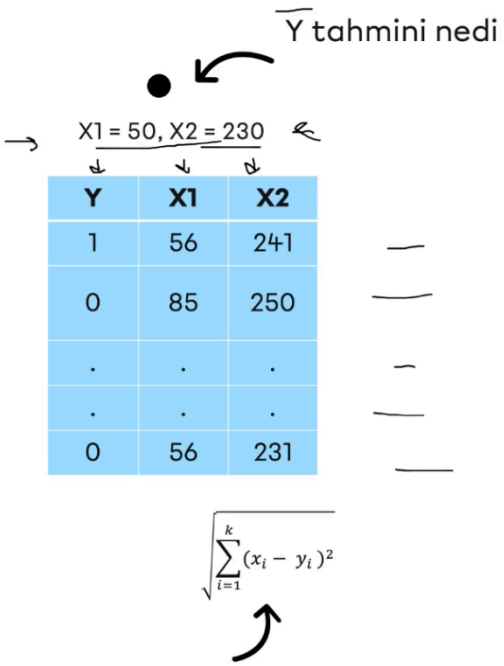
---

2. **En Yakın K Komşuyu Seçme:** Hesaplanan mesafelere göre, yeni veri noktasına en yakın olan **K** adet eğitim veri noktası belirlenir. Buradaki **K**, algoritmanın ana hiperparametresidir ve modelin davranışını büyük ölçüde etkiler.
3. **Çoğunluk Oyu (Sınıflandırma):** Seçilen K komşunun sınıflarına bakılır. Yeni veri noktası, bu K komşu arasında **en çok oy alan sınıfa** atanır. Örneğin, K=5 ise ve 3 komşu A sınıfı, 2 komşu B sınıfı ise, yeni nokta A sınıfına atanır.
    - **Ağırlıklı Oylama (Opsiyonel):** Bazı durumlarda, daha yakın komşuların oyları daha uzak komşuların oylarından daha fazla ağırlık taşıyacak şekilde ağırlıklı oylama yapılabilir. Bu, yakın komşuların karar üzerindeki etkisini artırır.









Keşifçi Veri Analizi (EDA)

```
#####################################################
# 1. Exploratory Data Analysis
#####################################################

df = pd.read_csv("datasets/diabetes.csv")

df.head()
df.shape
df.describe().T
df["Outcome"].value_counts()
```

Veri Ön İşleme (Data Pre-Processing)

- Uzaklık ve GD problemlerinde feature scaling lazım olur.

```
# 2. Data Preprocessing & Feature Engineering
#####################################################

y = df["Outcome"]
X = df.drop(["Outcome"], axis=1)

X_scaled = StandardScaler().fit_transform(X)

X = pd.DataFrame(X_scaled, columns=X.columns)
```

Modelleme

```
#####################################################
# 3. Modeling & Prediction
#####################################################

knn_model = KNeighborsClassifier().fit(X, y)

random_user = X.sample(1, random_state=45)

knn_model.predict(random_user)
```

Model Başarı Değerlendirme

- **Accuracy (Doğruluk):** Modelin tüm tahminlerinin ne kadarının **genel olarak doğru** olduğunu gösterir. Yani, hem doğru diyabetli hem de doğru diyabetli olmayan teşhislerin toplamıdır.
- **Precision (Kesinlik):** Model **"Diyabetli"** dediğinde, bu tahminlerin ne kadarının **gerçekten doğru** olduğunu gösterir. Yani, modelin pozitif olarak işaretlediği (diyabetli dediği) vakalarda ne kadar **güvenilir** olduğu.
- **Recall (Duyarlılık): Gerçekte diyabetli olan** tüm hastaların (yani TP ve FN'lerin toplamı) ne kadarının model tarafından **doğru bir şekilde yakalandığını** gösterir. Yani, modelin gerçekten diyabetli olanları **kaçırmama** yeteneği

Accuracy: Doğru sınıflandırma oranıdır: (TP+TN) / (TP+TN+FP+FN)

Precision: Pozitif sınıf (1) tahminlerinin basari oranıdır: TP / (TP+FP)

Recall: Pozitif sınıfın (1) doğru tahmin edilme oranıdır: TP / (TP + FN)

F1 SCORE: 2 * (Precision * Recall) / (Precision + Recall)

		Tahmin Edilen Sınıf	
		Sınıf = 1	Sınıf = 0
Gerçek Sınıf	Sınıf = 1	True Pozitif (TP)	False Negatif (FN)
	Sınıf = 0	False Pozitif (FP)	True Negatif (TN)

```
##################################################
# 4. Model Evaluation
##################################################

# Confusion matrix için y_pred:
y_pred = knn_model.predict(X)

# AUC için y_prob:
y_prob = knn_model.predict_proba(X)[:, 1]

print(classification_report(y, y_pred))
# acc 0.83
# f1 0.74
# AUC
roc_auc_score(y, y_prob)
# 0.90

cv_results = cross_validate(knn_model, X, y, cv=5, scoring=["accuracy", "f
1", "roc_auc"])

cv_results['test_accuracy'].mean()
cv_results['test_f1'].mean()
cv_results['test_roc_auc'].mean()
```

```
# 0.73
# 0.59
# 0.78

#Başarı sonuçları nasıl arttırılabilir:
# 1. Örnek boyutu arttıralabilir.
# 2. Veri ön işleme
# 3. Özellik mühendisliği
# 4. İlgili algoritma için optimizasyonlar yapılabilir.

#Hiperparametreler
knn_model.get_params()
```

Hiperparametre Optimizasyonu

```
##################################################
# 5. Hyperparameter Optimization
##################################################

knn_model = KNeighborsClassifier()
knn_model.get_params()
#{'algorithm': 'auto', 'leaf_size': 30, 'metric': 'minkowski', 'metric_params':
None, 'n_jobs': None,
# 'n_neighbors': 5, 'p': 2, 'weights': 'uniform'}

#  Hiperparametre Arama Alanını Tanımlama
# Optimize etmek istediğimiz hiperparametreleri ve denemek istediğimiz de
ğer aralıklarını belirtiyoruz.
# Burada sadece 'n_neighbors' (K değeri) için 2'den 49'a kadar (50 dahil d
eğil) tüm tam sayıları denemek istiyoruz.
knn_params = {"n_neighbors": range(2, 50)}



#  GridSearchCV Kullanarak Hiperparametre Optimizasyonu Yapma
# GridSearchCV, belirttiğimiz parametre aralığındaki tüm kombinasyonları si
stematik olarak dener.
# Her bir kombinasyon için çapraz doğrulama (cross-validation) yaparak m
```

```
odelin performansını ölçer.

knn_gs_best = GridSearchCV(knn_model,      # Optimize edilecek modelimi
z (K-NN)
                           knn_params,     # Denenecek hiperparametre aralıklarımız
                           cv=5,           # 5 katlı çapraz doğrulama kullanacağız.
                                           # Bu, veri setini 5 parçaya böler, 4'ünü eğitim, 1'in
i test için kullanır
                                           # ve bu işlemi 5 kez farklı parçalarla tekrarlar.
                                           # Ortalama performans alınır.
                           n_jobs=-1,      # Mevcut tüm CPU çekirdeklerini kullanarak
paralel hesaplama yapar.
                                           # Bu, arama sürecini hızlandırır.
                           verbose=1       # İşlem devam ederken detaylı çıktılar verir (i
lerleme çubuğu, denenen parametreler vb.).
                           ).fit(X, y)     # GridSearchCV'yi başlatıp, tüm veri seti (X ve
y) üzerinde eğitiriz.
                                           # GridSearchCV, her bir n_neighbors değeri için
modelin performansını ölçer.

# 4. En İyi Hiperparametreleri Bulma
# GridSearchCV tamamlandıktan sonra, çapraz doğrulama sonuçlarına göre
en iyi performansı veren
# hiperparametre kombinasyonunu 'best_params_' özelliği ile elde edebiliri
z.
# {'n_neighbors': 17}
knn_gs_best.best_params_
```

Final Model

```
#################################################
# 6. Final Model
#################################################
#birden fazla argüman alınıyor.
knn_final = knn_model.set_params(**knn_gs_best.best_params_).fit(X, y)

cv_results = cross_validate(knn_final,
```

```
                    X,
                    y,
                    cv=5,
                    scoring=["accuracy", "f1", "roc_auc"])

cv_results['test_accuracy'].mean()
cv_results['test_f1'].mean()
cv_results['test_roc_auc'].mean()

random_user = X.sample(1)

knn_final.predict(random_user)
```

Train-Test Split (stratify)

stratify=y neden önemli? Kullanımı da:

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

🔍 Neden Gereklidir?

Çünkü sınıflandırma problemlerinde veri setinde genellikle **dengesizlik** olabilir.
Örneğin:

Sınıf (y)	Gözlem Sayısı	Oran
0	900	%90
1	100	%10

Eğer bu veri setini **rastgele** bölersen:

- Test setine **hiç 1'li örnek düşmeyebilir.**
- Bu durumda model o sınıfı görmez → **öğrenemez, tahmin edemez.**

📦 Stratify Ne Yapıyor?

Diyor ki:

> "Test ve eğitim setlerine **aynı oranla sınıfları dağıtayım.**"

Örnek:

- %20 test ayırıyorsun
- Stratify olmadan:
  - Eğitim: 800 sınıf 0, 20 sınıf 1
  - Test: 100 sınıf 0, **80 sınıf 1** ← dengesizlik oldu!
- Stratify ile:
  - Eğitim: 720 sınıf 0, 80 sınıf 1
  - Test: 180 sınıf 0, 20 sınıf 1
    → **Aynı oranlar korundu (%90–%10)**