
توضیح مرحله به مرحله کد KNN:

اولین قدم، همونطور که قبلاً گفتم، **وارد کردن کتابخانه‌ها (Libraries)** هست.

قسمت اول: وارد کردن کتابخانه‌ها

```
Python

!pip install ipywidgets
!jupyter nbextension enable --py widgetsnbextension

import numpy as np
import matplotlib.pyplot as plt
import ipywidgets as widgets
from IPython.display import display
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay
import matplotlib.cm as cm
```

توضیح:

- **!pip install ipywidgets و !jupyter nbextension enable --py widgetsnbextension:**
  - این دو خط برای نصب و فعال کردن چیزی به اسم **ipywidgets** هستن. ipywidgets به ما کمک می‌کنه تا بتونیم دکمه، نوار لغزنده (slider) و اینجور چیزها رو توی محیط Colab بسازیم تا کاربر بتونه با کدمون تعامل داشته باشه. (علامت ! اول خط یعنی این دستور رو توی ترمینال سیستم اجرا کن، نه به عنوان کد پایتون).
  - فعلاً نگران جزئیاتش نباش، فقط بدون که برای ساختن یه رابط کاربری گرافیکی ساده استفاده میشن.
- **import numpy as np:**
  - numpy (نام‌پای) یه کتابخونه خیلی قدرتمند توی پایتونه که برای کار با **آرایه‌ها (Arrays)** و عملیات ریاضی سنگین استفاده میشه. آرایه رو مثل یه جدول بزرگ پر از عدد در نظر بگیر.
  - as np یعنی به جای اینکه هر بار اسم کامل numpy رو بنویسیم، میتونیم از مخفف np استفاده کنیم. این یه قرارداد رایجه.
- **import matplotlib.pyplot as plt:**
  - matplotlib یه کتابخونه عالی برای **کشیدن نمودار و تصویر (Plotting)** هست.
  - pyplot قسمت خاصی از این کتابخونه‌ست که دستوراتش خیلی شبیه متلب (MATLAB) هستن.
  - as plt هم مثل numpy برای راحت‌تر شدن اسمش استفاده میشه.
- **import ipywidgets as widgets:**
  - این خط ipywidgets رو وارد می‌کنه تا بتونیم از اون دکمه‌ها و نوارها استفاده کنیم. as widgets هم برای کوتاهی اسمشه.
- **from IPython.display import display:**
  - IPython.display به ما کمک می‌کنه تا چیزهایی مثل widgets رو توی خروجی Colab نشون بدیم.
- **from sklearn.model_selection import train_test_split:**

- sklearn (سایکت‌لرن) یه کتابخونه بسیار معروف و پرکاربرد برای **یادگیری ماشین (Machine Learning)** هست.
- train_test_split یه ابزار از این کتابخونه‌ست که به ما کمک می‌کنه داده‌هامون رو به دو بخش **آموزش (Training)** و **آزمایش (Testing)** تقسیم کنیم. (بعداً توضیح میدم چرا این کار رو می‌کنیم).

- **from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay:**
  - اینا هم ابزارهایی از sklearn.metrics (معیارهای ارزیابی) هستن که برای **سنجش عملکرد مدل** ما (KNN) استفاده میشن:
    - accuracy_score: برای محاسبه دقت مدل (چند درصد پیش‌بینی‌ها درست بودن).
    - confusion_matrix: برای ساختن ماتریس درهم‌ریختگی (جدولی که نشون میده مدل ما چقدر خوب دسته‌بندی کرده).
    - ConfusionMatrixDisplay: برای نمایش گرافیکی confusion_matrix.
- **import matplotlib.cm as cm:**
  - cm (Colormap) به matplotlib کمک می‌کنه تا رنگ‌های مختلفی رو توی نمودارها و نقشه‌ها (مثل نمودار تصمیم‌گیری) استفاده کنه.

خلاصه قسمت اول:

ما در این بخش ابزارهایی (کتابخانه‌هایی) رو وارد کردیم که در مراحل بعدی به ما کمک می‌کنن تا:

- با اعداد و آرایه‌ها راحت‌تر کار کنیم (numpy).
- نمودار و شکل بکشیم (matplotlib).
- رابط کاربری ساده بسازیم (ipywidgets).
- کارهای مربوط به یادگیری ماشین رو انجام بدیم (sklearn).

قسمت دوم: ساختار کلی یک Class در پایتون

حالا به بخش مهم کد می‌رسیم:

```
Python

class KNNClassifier:
    def __init__(self, k=3, distance_func=None):
        self.k = k
        if distance_func is None:
            self.distance_func = self._euclidean_distance
        else:
            self.distance_func = distance_func
```

توضیح: class چیه؟

قبل از هر چیز، باید مفهوم کلاس (Class) رو توضیح بدم.

تصور کن می‌خوای یه قالب (blueprint) برای ساختن خونه‌های مختلف درست کنی. این قالب بهت میگه که هر خونه باید چند تا اتاق، آشپزخونه، دستشویی و ... داشته باشه. وقتی از این قالب استفاده می‌کنی و یه خونه واقعی می‌سازی، به اون خونه می‌گن "شیء" (Object).

در پایتون، class دقیقاً همین کار رو می‌کنه. یه class مثل یه نقشه یا قالب برای ساختن "اشیاء" هست. هر شیء می‌تونه خصوصیات (variables/attributes) و رفتارها (functions/methods) خاص خودش رو داشته باشه.

- **class KNNClassifier:**: این خط میگه که ما داریم یه قالب (کلاس) جدید به اسم KNNClassifier می‌سازیم. این کلاس قراره منطق مدل KNN رو در خودش نگه داره.
- **def __init__(self, k=3, distance_func=None):**
  - این یک تابع (Function) خاص داخل کلاس هست به نام **"سازنده" (Constructor)**.
  - هر وقت ما یک "شیء" جدید از روی کلاس KNNClassifier می‌سازیم، این تابع __init__ به صورت **خودکار** اجرا میشه.
  - **self**: همیشه اولین پارامتر (ورودی) توابع داخل یک کلاس هست. self به شیء جاری (همون خونه‌ای که الان داریم می‌سازیم) اشاره می‌کنه.
  - **k=3**: این یک ورودی به اسم k هست که عدد پیش‌فرضش ۳ گذاشته شده. در KNN، k به تعداد "نزدیک‌ترین همسایه‌ها" اشاره داره که قراره برای پیش‌بینی استفاده بشن. (مثلاً اگه k=3 باشه، مدل ۳ تا نزدیک‌ترین نقطه رو در نظر می‌گیره.)
  - **distance_func=None**: این هم یه ورودی دیگه هست که برای تعیین نوع تابع محاسبه فاصله استفاده میشه. اگر چیزی بهش ندیم (None)، تابع فاصله پیش‌فرض (euclidean_distance_) استفاده میشه.
- **self.k = k**: این خط مقدار k رو که به تابع __init__ داده شده، توی یک متغیر به اسم k داخل خود شیء ذخیره می‌کنه. یعنی هر شیء KNNClassifier که می‌سازیم، مقدار k مخصوص خودش رو داره. (علامت . بین self و k به این معنیه که k متعلق به همین شیء هست).
- **if distance_func is None:**: این یک **شرط (Conditional Statement)** هست. میگه: "اگر distance_func هیچ مقداری نداشت (یعنی None بود)..."
  - **self.distance_func = self._euclidean_distance**: "...اونوقت تابع محاسبه فاصله رو به تابع پیش‌فرض euclidean_distance_ تنظیم کن." (ما خودمون این تابع رو پایین‌تر می‌نویسیم).
- **else:** "در غیر این صورت (یعنی اگر distance_func مقداری داشت)..."
  - **self.distance_func = distance_func**: "...اون مقدار رو به عنوان تابع محاسبه فاصله تنظیم کن."

خلاصه قسمت دوم:

این بخش در واقع **اسکلت اصلی** مدل KNN ما رو تعریف می‌کنه. وقتی ما می‌گیم my_knn_model = KNNClassifier(k=5), پایتون یه شیء جدید از روی این قالب می‌سازه. این شیء جدید یه k (مثلاً 5) و یه تابع برای محاسبه فاصله (مثلاً فاصله اقلیدسی) رو در خودش ذخیره می‌کنه.

قسمت سوم: تابع fit (آموزش مدل)

```
Python

  def fit(self, X, y):
      self.X_train = X
      self.y_train = y
```

توضیح:

- **def fit(self, X, y)**::
  - این تابع **fit** (به معنی "آموزش دادن" یا "برازش دادن") هست.
  - در یادگیری ماشین، مرحله fit جاییه که مدل ما **"یاد می‌گیره"**.
  - **X**: این پارامتر (ورودی) داده‌های **ویژگی‌ها (Features)** رو شامل میشه. مثلاً اگه داری خونه‌ها رو بر اساس متراژ و تعداد اتاق دسته‌بندی می‌کنی، X میشه متراژ و تعداد اتاق هر خونه.
  - **y**: این پارامتر **برچسب‌ها (Labels)** یا **خروجی‌های صحیح (Target Values)** رو شامل میشه. مثلاً اگه X متراژ و تعداد اتاق خونه‌ست، y میشه "لوکس"، "متوسط" یا "اقتصادی" بودن اون خونه.
  - در مدل KNN، مرحله آموزش خیلی ساده‌ست!
- **self.X_train = X**:
  - این خط داده‌های ویژگی‌ها (X) رو که به تابع fit داده شده، در متغیر X_train (X_آموزش) داخل شیء ذخیره می‌کنه.
  - train_ به معنی "داده‌های آموزشی" هست.
- **self.y_train = y**:
  - این خط برچسب‌های صحیح (y) رو در متغیر y_train (y_آموزش) داخل شیء ذخیره می‌کنه.

نکته مهم در KNN:

KNN یک مدل Lazy Learner (یادگیرنده تنبل) هست. این یعنی در مرحله fit، مدل هیچ کار پیچیده‌ای انجام نمیده. فقط داده‌های آموزشی رو "به خاطر می‌سپره". تمام کار پیچیده (محاسبه فاصله و پیدا کردن همسایه‌ها) در مرحله پیش‌بینی (predict) انجام میشه.

خلاصه قسمت سوم:

تابع fit داده‌های آموزشی (ویژگی‌ها و برچسب‌هایشون) رو به مدل KNN ما میده تا اون‌ها رو ذخیره کنه و برای پیش‌بینی‌های آینده آماده باشه.

قسمت چهارم: توابع محاسبه فاصله

Python

```
  def _euclidean_distance(self, x1, x2):
      return np.sqrt(np.sum((x1 - x2) ** 2))

  def _manhattan_distance(self, x1, x2):
      return np.sum(np.abs(x1 - x2))

  def _minkowski_distance(self, x1, x2, p=3):
      return np.sum(np.abs(x1 - x2) ** p) ** (1 / p)
```

توضیح:

این سه تابع برای محاسبه "فاصله" بین دو نقطه (یا دو داده) استفاده میشن. KNN برای پیدا کردن "نزدیک‌ترین همسایه‌ها" نیاز داره بدونه که هر نقطه چقدر از بقیه نقاط فاصله داره.

- **def _euclidean_distance(self, x1, x2)::**
  - این تابع **فاصله اقلیدسی (Euclidean Distance)** رو محاسبه می‌کنه.
  - x1 و x2 دو نقطه (داده) هستن.
  - **فرمولش:** $\sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2}$
    - x1 - x2: اختلاف بین هر ویژگی دو نقطه رو حساب می‌کنه.
    - ** 2: این اختلاف‌ها رو به توان ۲ می‌رسونه.
    - np.sum(...): تمام این اختلاف‌های به توان ۲ رسیده رو با هم جمع می‌کنه.
    - np.sqrt(...): از کل جمع ریشه دوم (جذر) می‌گیره.
  - این همون "فاصله خط راست" هست که مثلاً روی نقشه با خطکش اندازه می‌گیریم. رایج‌ترین فاصله در KNN.
- **def _manhattan_distance(self, x1, x2)::**
  - این تابع **فاصله منهتن (Manhattan Distance)** رو محاسبه می‌کنه. بهش "فاصله تاکسی" هم میگن، چون مثل مسیر حرکت یه تاکسی تو خیابون‌های شبکه ای شهر می‌مونه.
  - **فرمولش:** $\sum_{i=1}^{n} |x_{1i} - x_{2i}|$
    - np.abs(x1 - x2): قدر مطلق اختلاف هر ویژگی رو حساب می‌کنه (همیشه مثبت).
    - np.sum(...): تمام این قدر مطلق‌ها رو با هم جمع می‌کنه.
- **def _minkowski_distance(self, x1, x2, p=3)::**
  - این تابع **فاصله مینکوفسکی (Minkowski Distance)** هست.
  - این یک فرمول کلی‌تره که هم فاصله اقلیدسی و هم منهتن رو پوشش میده.
  - **فرمولش:** $(\sum_{i=1}^{n} |x_{1i} - x_{2i}|^p)^{(1/p)}$
    - اگر p=1 باشه، میشه فاصله منهتن.
    - اگر p=2 باشه، میشه فاصله اقلیدسی.
  - اینجا p=3 به عنوان مقدار پیش‌فرض در نظر گرفته شده.

نکته: علامت _ در ابتدای اسم توابع (مثل euclidean_distance_) به این معنیه که این توابع قرار نیست مستقیماً از بیرون کلاس صدا زده بشن و بیشتر برای استفاده داخلی خود کلاس هستن.

خلاصه قسمت چهارم:

این توابع به مدل KNN ما می‌گن که چطور "نزدیکی" یا "دوری" بین نقاط داده رو اندازه بگیره. بسته به مسئله‌ای که داریم، ممکنه یکی از این روش‌های محاسبه فاصله بهتر از بقیه عمل کنه.

قسمت پنجم: تابع predict (پیش‌بینی)

این مهم‌ترین قسمت مدل KNN هست که در نهایت خروجی نهایی رو تولید می‌کنه.

```Python
  def predict(self, X):
```

```
    predictions = []
    for index, x in enumerate(X):
        distances = [self.distance_func(x, x_train) for x_train in self.X_train]

        # indices of the k nearest neighbors
        k_indices = np.argsort(distances)[:self.k]
        # labels of the k nearest neighbors
        k_neighbor_labels = self.y_train[k_indices]
        # majority vote
        counts = np.bincount(k_neighbor_labels.astype(int))
        predicted_label = np.argmax(counts)
        predictions.append(predicted_label)

    return np.array(predictions)
```

توضیح:

- **def predict(self, X)::**
  - این تابع predict (به معنی "پیش‌بینی کردن") هست.
  - **X**: این پارامتر (ورودی) شامل داده‌های جدیدی هست که می‌خوایم برچسب (label) اون‌ها رو پیش‌بینی کنیم. این‌ها داده‌هایی هستند که مدل ما قبلاً در مرحله fit ندیده.
- **predictions = []:**
  - یک لیست خالی به اسم predictions (پیش‌بینی‌ها) تعریف می‌کنیم. قرار هست نتیجه پیش‌بینی هر نقطه جدید رو داخل این لیست بریزیم.
- **for index, x in enumerate(X)::**
  - این یک **حلقه (Loop)** هست. یعنی قراره یه سری عملیات رو برای **هر کدوم از نقاط جدید** داخل X تکرار کنیم.
  - enumerate(X) به ما هم index (شماره ترتیب) و هم x (خود داده) رو در هر بار تکرار حلقه میده. x در اینجا یک "نقطه" یا "داده" جدید هست که می‌خوایم براش پیش‌بینی کنیم.
- **distances = [self.distance_func(x, x_train) for x_train in self.X_train]:**
  - این یک خط کد پیشرفته‌تر پایتون به نام **"لیست کامپرهنشن" (List Comprehension)** هست.
  - معنیش اینه: "برای هر x_train (یعنی هر داده آموزشی) که توی self.X_train (داده‌های آموزشی که در مرحله fit ذخیره کردیم) هست، بیا و فاصله x (نقطه جدید) رو از x_train (نقطه آموزشی) با استفاده از تابع self.distance_func (که قبلاً تعیین کردیم، مثلاً اقلیدسی) حساب کن و همه‌ی این فاصله‌ها رو در یک لیست به اسم distances بریز."
  - نتیجه: لیستی از فاصله‌های نقطه جدید x از **همه** نقاط آموزشی که مدل دیده بود.
- **k_indices = np.argsort(distances)[:self.k]:**
  - **np.argsort(distances)**: این تابع از numpy این کار رو می‌کنه: "لیست distances رو بگیر، و اندیس‌های (جایگاه) عناصری که اگر مرتب بشن، از کوچک به بزرگ (فاصله‌های کمتر به بیشتر) قرار می‌گیرند رو بهت بده." مثلاً اگه فاصله‌ها [5, 1, 8, 2] باشن، argsort میگه [1, 3, 0, 2] (یعنی عنصر با اندیس 1 کوچیکترینه، بعد عنصر با اندیس 3، بعد 0 و بعد 2).
  - **[:self.k]**: این یه برش (Slice) هست. یعنی "فقط k تا از اولین اندیس‌ها رو بهم بده". این همون k نزدیک‌ترین همسایه رو به ما میده.
  - نتیجه: k_indices لیستی از اندیس‌های (جایگاه) k نزدیک‌ترین نقاط آموزشی به x هست.
- **k_neighbor_labels = self.y_train[k_indices]:**

- حالا که اندیس k نزدیک‌ترین همسایه رو داریم، از این اندیس‌ها استفاده می‌کنیم تا **برچسب‌های (Labels)** این همسایه‌ها رو از self.y_train (برچسب‌های داده‌های آموزشی) پیدا کنیم.
- نتیجه: k_neighbor_labels لیستی از برچسب‌های k نزدیک‌ترین همسایه به x هست.

- **(counts = np.bincount(k_neighbor_labels.astype(int))**:
  - **(k_neighbor_labels.astype(int**: اطمینان حاصل می‌کنه که برچسب‌ها از نوع عددی (integer) باشن.
  - **np.bincount(...)**: این تابع از numpy شمارش می‌کنه که هر برچسب (عدد) چند بار در k_neighbor_labels تکرار شده. مثلاً اگه k_neighbor_labels باشه [0, 1, 0, 0, 1] (برای k=5، bincount بهت میگه: [3, 2] (یعنی برچسب 0 سه بار و برچسب 1 دو بار تکرار شده). به این میگن **"رای اکثریت" (Majority Vote)**.
- **(predicted_label = np.argmax(counts)**:
  - **(np.argmax(counts)**: این تابع از numpy بهت میگه که کدوم عنصر در لیست counts (شمارش‌ها) **بیشترین مقدار** رو داره. (یعنی کدوم برچسب بیشترین تکرار رو داشته).
  - نتیجه: predicted_label میشه برچسبی که **بیشترین رای** رو از k نزدیک‌ترین همسایه گرفته. این همون پیش‌بینی مدل برای نقطه جدید x هست.
- **(predictions.append(predicted_label)**:
  - این خط predicted_label (برچسب پیش‌بینی شده برای نقطه x) رو به لیست predictions اضافه می‌کنه.
- **(return np.array(predictions)**:
  - بعد از اینکه حلقه برای تمام نقاط در X تموم شد، لیست predictions رو به یک آرایه numpy تبدیل می‌کنه و به عنوان خروجی تابع predict برمی‌گردونه.

خلاصه قسمت پنجم:

تابع predict قلب مدل KNN ماست. برای هر نقطه جدید، فاصله اون رو از تمام نقاط آموزشی حساب می‌کنه، k تا نزدیک‌ترینشون رو پیدا می‌کنه، و بعد برچسبی رو پیش‌بینی می‌کنه که بیشترین تکرار رو بین اون k همسایه داشته باشه.

جمع‌بندی کلی کد KNNClassifier:

این کلاس KNNClassifier در واقع یه مدل ساده و کاربردی از الگوریتم K-Nearest Neighbors رو پیاده‌سازی می‌کنه.

1. **__init__**: کلاس رو با تنظیم k (تعداد همسایه‌ها) و نوع تابع فاصله (اقلیدسی، منهتن یا مینکوفسکی) آماده می‌کنه.
2. **fit**: داده‌های آموزشی رو فقط "به خاطر می‌سپره".
3. **توابع فاصله**: راه‌های مختلفی برای اندازه‌گیری "نزدیکی" بین نقاط داده ارائه میدن.
4. **predict**: برای هر نقطه جدید، نزدیک‌ترین همسایه‌ها رو پیدا می‌کنه و بر اساس رای اکثریت اون‌ها، برچسب نقطه جدید رو پیش‌بینی می‌کنه.

خیلی خوبه! حالا که مفهوم کد KNN رو درک کردیم، وقتشه که ازش استفاده کنیم. برای این کار، اول نیاز به **داده** داریم. این بخش از کد دقیقاً برای **تولید داده‌های نمونه مصنوعی** نوشته شده تا بتونیم مدل KNN خودمون رو روش آموزش بدیم و امتحان کنیم.

تولید داده‌های مصنوعی (generate_synthetic_data):

بیا این قسمت از کد رو خط به خط بررسی کنیم:

```
Python

def generate_synthetic_data(m=3, num_points_per_class=100, cluster_std=1.0):
    np.random.seed(40)
    X = []
    y = []
    means = []
    for i in range(m):
        angle = 2 * np.pi * i / m
        radius = 3
        mean = [radius * np.cos(angle), radius * np.sin(angle)]
        means.append(mean)
        cov = [[cluster_std, 0], [0, cluster_std]]
        class_data = np.random.multivariate_normal(mean, cov, num_points_per_class)
        X.append(class_data)
        y += [i] * num_points_per_class

    X = np.vstack(X)
    y = np.array(y)
    return X, y, means
```

هدف این تابع: تولید داده‌های دو بعدی (یعنی هر نقطه فقط یک مختصات x و یک مختصات y داره) که به صورت **خوشه (Cluster)** در اطراف یک سری نقاط مرکزی قرار گرفتن.

توضیح پارامترها (ورودی‌ها):

- m=3: این تعداد **کلاس‌ها (Classes)** یا گروه‌هایی رو که می‌خوایم داده تولید کنیم، مشخص می‌کنه. (پیش‌فرض 3).
- num_points_per_class=100: تعداد **نقاط (Points)** در هر کلاس رو تعیین می‌کنه. (پیش‌فرض 100).
- cluster_std=1.0: این پارامتر **میزان پراکندگی (Standard Deviation)** نقاط رو در اطراف مرکز هر خوشه نشون میده. هرچی این عدد بیشتر باشه، نقاط از مرکزشون دورتر و پراکنده‌تر هستن. (پیش‌فرض 1.0).

توضیح خط به خط تابع:

1. np.random.seed(40): این خط یه "دانه" (Seed) برای تولید اعداد تصادفی تنظیم می‌کنه. یعنی چی؟ یعنی هر بار که این کد رو اجرا کنی، دقیقاً **همون داده‌های تصادفی قبلی** تولید میشن. این کار برای اینه که نتایج قابل تکرار باشن و بتونیم آزمایشاتمون رو با همون داده‌ها انجام بدیم. اگه این خط رو ننویسیم، هر بار داده‌های کاملاً متفاوتی ساخته میشن.
2. means ،y = [] ،X = [] = []: سه تا **لیست خالی** تعریف می‌کنه.
   - X: قراره **ویژگی‌های (Features)** نقاط رو نگهداری کنه (مختصات x و y هر نقطه).
   - y: قراره **برچسب (Label)** یا کلاس مربوط به هر نقطه رو نگهداری کنه.
   - means: قراره **مراکز (Means)** هر خوشه رو نگهداری کنه.

3. for i in range(m):: این یک **حلقه for** هست که به تعداد m (تعداد کلاس‌ها) بار تکرار میشه. در هر بار تکرار، یک کلاس جدید از داده‌ها رو تولید می‌کنیم. i هم نماینده شماره کلاس هست (مثلاً 0، 1، 2 و...).
   - angle = 2 * np.pi * i / m: این خط برای هر کلاس، یک **زاویه** منحصر به فرد حساب می‌کنه. هدف اینه که مراکز خوشه‌ها رو به صورت دایره‌ای شکل در اطراف مبدأ (0,0) قرار بدیم.
     - 2 * np.pi: یک دایره کامل (360 درجه) رو نشون میده.
     - i / m: این دایره رو به m قسمت مساوی تقسیم می‌کنه.
   - radius = 3: یک **شعاع ثابت** برای قرارگیری مراکز خوشه‌ها تعیین می‌کنه. یعنی همه مراکز خوشه‌ها در فاصله 3 واحد از مبدأ قرار می‌گیرن.
   - mean = [radius * np.cos(angle), radius * np.sin(angle)]: این خط **مختصات مرکز (میانگین)** هر خوشه رو با استفاده از شعاع و زاویه حساب می‌کنه. این همون فرمول تبدیل مختصات قطبی به دکارتیه (x = r cos theta, y = r sin theta).
     - np.cos(angle) و np.sin(angle): توابع مثلثاتی کسینوس و سینوس از کتابخانه numpy.
   - means.append(mean): مرکز حساب شده برای کلاس فعلی رو به لیست means اضافه می‌کنه.
   - cov = [[cluster_std, 0], [0, cluster_std]]: این **ماتریس کوواریانس (Covariance Matrix)** رو تعریف می‌کنه.
     - برای سادگی، این ماتریس به ما میگه که پراکندگی نقاط در جهت x و y چقدره. cluster_std همون پراکندگی رو کنترل می‌کنه. (فعلاً زیاد درگیر جزئیاتش نشو، فقط بدون برای تولید داده‌های خوشه‌ای استفاده میشه.)
   - class_data = np.random.multivariate_normal(mean, cov, num_points_per_class): این مهم‌ترین خط در تولید داده‌هاست.
     - np.random.multivariate_normal: تابعی از numpy که برای تولید داده‌های تصادفی به صورت **نرمال چند متغیره** استفاده میشه. (یعنی نقاط اطراف یک مرکز خاص با یک پراکندگی مشخص پخش میشن).
     - mean: مرکز خوشه‌ای که الان حساب کردیم.
     - cov: ماتریس کوواریانس (پراکندگی).
     - num_points_per_class: تعداد نقاطی که برای این خوشه می‌خوایم تولید کنیم.
     - نتیجه: class_data یک آرایه شامل num_points_per_class نقطه تصادفیه که در اطراف mean با پراکندگی cluster_std توزیع شدن.
   - X.append(class_data): این نقاط تولید شده برای کلاس فعلی رو به لیست X اضافه می‌کنه.
   - y += [i] * num_points_per_class: این خط **برچسب (Label)** مربوط به نقاط کلاس فعلی رو به لیست y اضافه می‌کنه.
     - [i] * num_points_per_class: یعنی i (شماره کلاس، مثلاً 0 یا 1 یا 2) رو num_points_per_class بار تکرار کن. مثلاً اگه i=0 و num_points_per_class=100 باشه، 100 تا صفر به y اضافه می‌کنه. این تضمین می‌کنه که هر نقطه در X برچسب صحیح خودش رو در y داشته باشه.
4. X = np.vstack(X): بعد از اینکه حلقه تموم شد و داده‌های تمام کلاس‌ها رو در لیست X جمع کردیم، این خط تمام آرایه‌های داخل لیست X رو به صورت **عمودی (vertically)** روی هم قرار میده و یک آرایه numpy بزرگ و یکپارچه از تمام ویژگی‌های داده (X) تولید می‌کنه.
5. y = np.array(y): لیست y (شامل برچسب‌ها) رو به یک آرایه numpy تبدیل می‌کنه.
6. return X, y, means: در نهایت، تابع سه چیز رو برمی‌گردونه:
   - X: آرایه شامل تمام نقاط داده (ویژگی‌ها).
   - y: آرایه شامل برچسب‌های مربوط به هر نقطه.
   - means: لیست مراکز واقعی خوشه‌ها (برای اینکه بتونیم ببینیم داده‌ها چطور چیده شدن).

استفاده از تابع `generate_synthetic_data` و نمایش داده‌ها:

```
Python

m = 5  # number of classes
num_points_per_class = 50
cluster_std = 1.0

X, y, class_means = generate_synthetic_data(m=m, num_points_per_class=num_points_per_class,
cluster_std=cluster_std)

plt.figure(figsize=(8,6))
colors = plt.colormaps['tab10'].colors
for i in range(m):
    plt.scatter(X[y == i, 0], X[y == i, 1], color=colors[i], label=f'Class {i}')
plt.title('Synthetic Training Data for Multiple Classes')
plt.legend()
plt.show()
```

توضیح این بخش:

1. `m = 5`، `num_points_per_class = 50`، `cluster_std = 1.0`: اینجا مقادیر دلخواه رو برای پارامترهای تابع `generate_synthetic_data` تعریف می‌کنیم.
    - یعنی 5 کلاس داریم.
    - هر کلاس 50 نقطه داره.
    - پراکندگی نقاط در هر خوشه هم 1.0 هست.
2. `X, y, class_means = generate_synthetic_data(...)`: تابع رو با این پارامترها فراخوانی می‌کنه و خروجی‌هاش رو (یعنی داده‌ها، برچسب‌ها و مراکز واقعی) در متغیرهای `X`, `y`, `class_means` ذخیره می‌کنه.
3. `plt.figure(figsize=(8,6))`: یک پنجره جدید برای رسم نمودار ایجاد می‌کنه و اندازه‌اش رو تعیین می‌کنه (8 اینچ عرض، 6 اینچ ارتفاع).
4. `colors = plt.colormaps['tab10'].colors`: یک مجموعه از رنگ‌ها رو از `matplotlib` برای نمایش کلاس‌های مختلف انتخاب می‌کنه. `tab10` یک پالت رنگی رایج هست.
5. `for i in range(m)`:: دوباره یک حلقه `for` که به تعداد کلاس‌ها (5 بار) تکرار میشه. در هر تکرار، نقاط یک کلاس رو روی نمودار رسم می‌کنه.
    - `plt.scatter(X[y == i, 0], X[y == i, 1], color=colors[i], label=f'Class {i}')`: این خط **نقاط** رو روی نمودار پخش (scatter plot) می‌کنه.
        - `X[y == i, 0]`: مختصات `x` (ستون اول) تمام نقاطی رو انتخاب می‌کنه که برچسبشون (`y`) برابر با `i` (شماره کلاس فعلی) باشه.
        - `X[y == i, 1]`: مختصات `y` (ستون دوم) تمام نقاطی رو انتخاب می‌کنه که برچسبشون (`y`) برابر با `i` باشه.
        - `color=colors[i]`: رنگ نقاط رو بر اساس رنگ مربوط به کلاس `i` تنظیم می‌کنه.
        - `label=f'Class {i}'`: یک برچسب برای این مجموعه از نقاط در راهنمای نمودار (Legend) ایجاد می‌کنه.
6. `plt.title('Synthetic Training Data for Multiple Classes')`: عنوان نمودار رو تنظیم می‌کنه.

7. plt.legend(): راهنمای نمودار رو نمایش میده (که نشون میده هر رنگ مربوط به کدوم کلاس هست).
8. plt.show(): در نهایت نمودار رو نمایش میده.

نتیجه‌ای که می‌بینی:

وقتی این کد رو اجرا کنی، یک نمودار میبینی که توش 5 تا "خوشه" از نقاط رنگی وجود داره. هر خوشه نماینده یک کلاس هست و نقاط داخل اون خوشه کمی از هم پراکنده شدن. این دقیقاً همون داده‌های آموزشی هستن که قراره مدل KNN ما روی اون‌ها کار کنه!

حالا نوبت توئه!

1. آیا می‌تونی این بخش از کد رو اجرا کنی و نمودار رو ببینی؟
2. اگه مقادیر num_points_per_class ،m یا cluster_std رو تغییر بدی، فکر می‌کنی چه اتفاقی برای نمودار میفته؟ امتحانش کن!
   - مثلاً m رو بذار 2.
   - یا cluster_std رو بذار 0.2 (خیلی متمرکز) یا 3.0 (خیلی پراکنده).

کوواریانس (Covariance) به زبان ساده

تصور کن داری دو تا چیز رو اندازه می‌گیری:

1. **میزان ساعت مطالعه یک دانش‌آموز** (متغیر اول)
2. **نمره‌ای که در امتحان می‌گیره** (متغیر دوم)

حالا می‌خوای بدونی که آیا این دوتا متغیر به هم ربط دارن یا نه، و اگر ربط دارن، چطور به هم ربط دارن.

کوواریانس به ما میگه که دو تا متغیر عددی، چطور با هم "تغییر" می‌کنن.

- **اگر کوواریانس مثبت باشه:**
  - یعنی وقتی یکی از متغیرها زیاد میشه، اون یکی هم **تمایل داره زیاد بشه**.
  - و وقتی یکی کم میشه، اون یکی هم **تمایل داره کم بشه**.
  - مثال ما: اگر دانش‌آموز بیشتر درس بخونه، نمره‌اش هم بیشتر میشه. (رابطه مستقیم)
- **اگر کوواریانس منفی باشه:**
  - یعنی وقتی یکی از متغیرها زیاد میشه، اون یکی **تمایل داره کم بشه**.
  - و وقتی یکی کم میشه، اون یکی **تمایل داره زیاد بشه**.
  - مثال: فرض کن "تعداد ساعت خواب" و "میزان خستگی" رو اندازه بگیریم. هرچی بیشتر بخوابی، کمتر خسته میشی. (رابطه معکوس)
- **اگر کوواریانس نزدیک به صفر باشه:**
  - یعنی دو تا متغیر **ارتباط خطی کمی با هم دارن** یا اصلاً ارتباط خطی ندارن.
  - مثال: "اندازه کفش یک دانش‌آموز" و "نمره‌ای که در امتحان می‌گیره". هیچ رابطه‌ای منطقی بین این دوتا نیست.

تفاوت کوواریانس با واریانس (Variance)

- **واریانس (Variance):**
  - واریانس فقط برای **یک متغیر** استفاده میشه.
  - به ما میگه که داده‌های یک متغیر، چقدر از میانگین خودشون **پراکنده** هستن.
  - مثال: "واریانس نمرات امتحانی" نشون میده که نمرات دانش‌آموزان چقدر از میانگین نمرات (مثلاً نمره 15) دور هستن؛ یعنی نمرات نزدیک به همن یا خیلی پخش و پلا هستن.
- **کوواریانس (Covariance):**
  - کوواریانس برای **دو یا چند متغیر** استفاده میشه.
  - به ما میگه که این دو متغیر **چطور با هم حرکت می‌کنن** یا به هم مرتبط هستن. (هم‌جهت هستن یا خلاف جهت هم).

بنابراین:

- **واریانس:** پراکندگی **یک** متغیر رو نشون میده.
- **کوواریانس:** رابطه **دو** متغیر با هم رو نشون میده (اینکه با هم بالا میرن یا یکی بالا میره و یکی پایین میاد).

کاربرد کوواریانس در کد KNN ما (و در کل در یادگیری ماشین)

در کدی که دیدیم:

```
Python

cov = [[cluster_std, 0], [0, cluster_std]]
class_data = np.random.multivariate_normal(mean, cov, num_points_per_class)
```

اینجا `cov` یک **ماتریس کوواریانس (Covariance Matrix)** هست.

- وقتی ما فقط دو ویژگی (مثلاً مختصات x و y) داریم، ماتریس کوواریانس به ما میگه که:
  - پراکندگی در جهت x چقدره (همون `cluster_std` در قطر اصلی ماتریس).
  - پراکندگی در جهت y چقدره (همون `cluster_std` در قطر اصلی ماتریس).
  - و **ارتباط بین تغییرات x و y چقدره**. (اون `0`هایی که در خارج از قطر اصلی ماتریس هستن).
- در کد ما، چون کوواریانس بین x و y رو صفر گذاشتیم (`0` در خارج قطر اصلی)، این یعنی فرض کردیم که **تغییرات در محور x هیچ ارتباطی با تغییرات در محور y نداره**. یعنی اگر یک نقطه در محور x حرکت کنه، هیچ تاثیری روی حرکتش در محور y نداره و برعکس. به همین دلیل خوشه‌ها به شکل دایره‌ای پخش میشن.

اگر اون `0`ها رو به یک عدد مثبت یا منفی تغییر میدادیم، خوشه‌های ما دیگه دایره‌ای نبودن و مثلاً بیضوی میشدن، چون نشون می‌داد که تغییرات در x و y به هم وابسته هستن.

باشه، هیچ مشکلی نیست. بیا قسمت **تولید داده‌های مصنوعی** رو از اول و با جزئیات بیشتر توضیح بدیم، جوری که کامل برات جا بیفته.

فرض کن می‌خوایم یه مدل هوش مصنوعی بسازیم که بتونه گربه‌ها رو از سگ‌ها تشخیص بده. برای آموزش این مدل، نیاز به عکس گربه و سگ داریم. اما اگه عکس کافی نداشته باشیم چی؟ یا اگه بخوایم برای آموزش‌های اولیه خودمون یه محیط کنترل‌شده داشته باشیم؟ اینجاست که **"تولید داده‌های مصنوعی"** به کار میاد.

تولید داده مصنوعی یعنی چی؟

یعنی به جای اینکه بریم دنبال داده‌های واقعی (مثل عکس گربه و سگ تو اینترنت)، خودمون با استفاده از ریاضیات و کدنویسی، داده‌های شبیه به واقعیت رو **می‌سازیم**. این داده‌ها "مصنوعی" هستن چون توسط ما تولید شدن، نه از دنیای واقعی جمع‌آوری شده باشن.

چرا داده مصنوعی تولید می‌کنیم؟

1. **نداشتن داده کافی:** گاهی اوقات جمع‌آوری داده‌های واقعی سخته یا زمان‌بره.
2. **کنترل‌پذیری:** وقتی خودمون داده رو می‌سازیم، دقیقاً می‌دونیم که چطور توزیع شده، چند تا کلاس داره، و چقدر نویز (اختلال) داره. این برای یادگیری و آزمایش الگوریتم‌ها خیلی مفیده.
3. **حریم خصوصی:** در برخی موارد، استفاده از داده‌های واقعی ممکنه مسائل حریم خصوصی رو ایجاد کنه.

سناریوی ما: تشخیص الگوهای دایره‌ای

در کدی که داری، ما نمی‌خوایم گربه و سگ تشخیص بدیم. می‌خوایم یه مدل بسازیم که بتونه **نقطه‌ها رو بر اساس جایی که تو فضای دو بعدی قرار گرفتن، دسته‌بندی کنه**.

تصور کن یه عالمه نقطه داریم که باید تشخیص بدیم هر کدوم به کدوم گروه (کلاس) تعلق دارن. مثلاً نقاط آبی، قرمز، سبز و ...

کد `generate_synthetic_data` دقیقاً برای تولید این نوع نقاط طراحی شده، جوری که این نقاط به صورت **"خوشه" (Cluster)** در اطراف یک سری مراکز (مانند نقاط مرکزی یک دایره) قرار بگیرن.

توضیح کد `generate_synthetic_data` (باز هم با مثال!)

بیا تابع رو دوباره ببینیم:

```
Python

def generate_synthetic_data(m=3, num_points_per_class=100, cluster_std=1.0):
    np.random.seed(40) # 1
    X = [] # 2 برای نگهداری مختصات نقاط
    y = [] # 3 برای نگهداری برچسب (کلاس) نقاط
    means = [] # 4 برای نگهداری مراکز واقعی خوشه ها

    for i in range(m): # 5 به تعداد کلاس ها (m) تکرار کن. مثلاً اگه m=3 باشه، i میشه 0، 1، 2
        # --- تعیین مرکز هر خوشه به صورت دایره‌ای ---
        angle = 2 * np.pi * i / m # 6 زاویه برای قرار دادن مرکز خوشه روی دایره
```

```
                radius = 3                  # 7 فاصله مرکز خوشه از مبدأ (شعاع دایره):
mean = [radius * np.cos(angle), radius * np.sin(angle)] # 8 مختصات x و y مرکز خوشه:
means.append(mean)                          # 9 مرکز رو به لیست means اضافه کن:

# --- تعیین پراکندگی نقاط اطراف مرکز ---
cov = [[cluster_std, 0], [0, cluster_std]] # 10 ماتریس کوواریانس (میزان پخش شدن نقاط):

# --- تولید خود نقاط برای این خوشه ---
class_data = np.random.multivariate_normal(mean, cov, num_points_per_class) # 11 تولید نقاط تصادفی اطراف مرکز:
X.append(class_data)                        # 12 نقاط رو به X اضافه کن:
y += [i] * num_points_per_class             # 13 برچسب (i) رو به تعداد نقاط به y اضافه کن:

X = np.vstack(X) # 14 همه نقاط X رو تبدیل به یه آرایه بزرگ کن:
y = np.array(y)  # 15 برچسب ها رو تبدیل به یه آرایه کن:
return X, y, means # 16 نقاط، برچسب ها و مراکز رو برگردون:
```

حالا با جزئیات بیشتر:

1. **np.random.seed(40):** فرض کن میخوای چند بار یه تاس بندازی و هر بار هم دقیقاً همون اعداد قبلی رو به دست بیاری. این خط دقیقاً همین کارو برای اعداد تصادفی کامپیوتری می‌کنه. یعنی اگه این خط رو بذاری، هر بار که کد رو اجرا کنی، **همون مجموعه نقاط رو تولید می‌کنه**. اگه حذفش کنی، هر بار نقاط جدیدی ساخته میشن. (این برای آزمایش کد و مطمئن شدن از نتایج مفیده).
2. **X = [] و y = [] و means = []:**
   - **X**: قراره لیست "ویژگی‌های" هر نقطه رو نگه داره. مثلاً اگه یه نقطه در مختصات (2.5, 1.8) باشه، این [2.5, 1.8] میشه ویژگی‌های اون.
   - **y**: قراره "برچسب" یا "کلاس" هر نقطه رو نگه داره. مثلاً اگه نقطه آبی باشه، برچسبش 0، اگه قرمز باشه، برچسبش 1 و...
   - **means**: این لیست فقط برای دیدن اینه که مراکز اصلی خوشه‌ها کجا بودن. مدل ما ازش استفاده نمی‌کنه، فقط برای ما مفیده.
3. **for i in range(m): (شروع حلقه برای تولید کلاس‌ها):**
   - فرض کن m = 3 (می‌خوایم 3 تا کلاس یا گروه نقطه داشته باشیم).
   - این حلقه 3 بار اجرا میشه: یک بار برای i=0 (کلاس 0)، یک بار برای i=1 (کلاس 1)، و یک بار برای i=2 (کلاس 2).
   - **angle = 2 * np.pi * i / m و radius = 3 و mean = [radius * np.cos(angle), radius * np.sin(angle)]:**
     - هدف این 3 خط اینه که برای هر کلاس، یک **"نقطه مرکزی" (mean)** روی محیط یک دایره فرضی با شعاع 3 پیدا کنیم.
     - **np.pi** عدد پی (3.14159...) هست. 2 * np.pi یعنی 360 درجه.
     - i / m: مثلاً برای m=3 و i=0 میشه 0 (زاویه 0 درجه)، برای i=1 میشه 1/3 (زاویه 120 درجه)، برای i=2 میشه 2/3 (زاویه 240 درجه).
     - با این کار، مراکز خوشه‌ها به صورت مساوی روی یه دایره چیده میشن.
     - np.cos(angle) و np.sin(angle): اینا توابع ریاضی هستن که با استفاده از زاویه و شعاع، مختصات x و y مرکز رو حساب می‌کنن.
   - **cov = [[cluster_std, 0], [0, cluster_std]]:**

- اینجا **ماتریس کوواریانس (Covariance Matrix)** رو می‌سازیم.
- همونطور که قبلاً گفتم، این ماتریس به ما میگه که نقاط یک خوشه چقدر از مرکز اون خوشه "پخش" میشن.
- cluster_std: این مقداریه که کنترل می‌کنه نقاط چقدر از مرکز پخش بشن. اگه cluster_std کوچیک باشه، نقاط خیلی نزدیک به مرکز قرار می‌گیرن (خوشه فشرده). اگه بزرگ باشه، نقاط خیلی پخش میشن (خوشه پراکنده).
- اون 0هایی که در خارج از قطر اصلی ماتریس هستن، یعنی فرض می‌کنیم که پراکندگی در جهت افقی (x) با پراکندگی در جهت عمودی (y) **هیچ ارتباطی نداره**. به همین خاطر خوشه‌ها گرد (دایره‌ای) در میان. اگه به جای 0 عدد دیگه ای میگذاشتیم، خوشه‌ها بیضوی می‌شدند.

  - **class_data = np.random.multivariate_normal(mean, cov, num_points_per_class):**
    - این دستور جادویی numpy هست!
    - بهش میگیم: "بر اساس این mean (مرکز) و این cov (پراکندگی)، num_points_per_class (مثلاً 100) تا نقطه **تصادفی** تولید کن."
    - این تابع نقاط رو جوری تولید می‌کنه که بیشترشون نزدیک mean باشن و هرچی از mean دورتر میشن، تعدادشون کمتر میشه (مثل زنگوله).
    - نتیجه: یه عالمه نقطه که دور مرکز این خوشه خاص پخش شدن.
  - **X.append(class_data):** این نقاطی که الان برای یک کلاس خاص تولید شدن رو به لیست بزرگ X اضافه می‌کنه.
  - **y += [i] * num_points_per_class:**
    - این خط، برچسب (شماره کلاس) i رو به تعداد num_points_per_class بار تکرار می‌کنه و به لیست y اضافه می‌کنه.
    - مثلاً اگه i=0 و num_points_per_class=100 باشه، 100 تا صفر به y اضافه میشه. این یعنی 100 تا نقطه اولی که تو X هستن، برچسبشون 0 هست.

4. **X = np.vstack(X) و y = np.array(y):**
   - بعد از اینکه حلقه تموم شد و نقاط و برچسب‌های همه کلاس‌ها رو جدا جدا تو لیست X و y جمع کردیم، این دو خط اون‌ها رو به یک **آرایه numpy یکپارچه و بزرگ** تبدیل می‌کنن. اینطوری کار کردن باهاشون راحت‌تره.
5. **return X, y, means:** در نهایت، تابع این سه تا چیز رو به ما برمی‌گردونه تا بتونیم ازشون استفاده کنیم.

تصویر ذهنی نهایی:

این تابع رو مثل یه کارخونه کوچک در نظر بگیر. به این کارخونه میگی:

- "چند نوع محصول (m کلاس) می‌خوام؟"
- "هر نوع محصول چند تا باشه (num_points_per_class)؟"
- "محصولات چقدر از هم دور باشن (cluster_std)؟"

بعد این کارخونه، طبق اون مشخصات، برات یه عالمه نقطه تولید می‌کنه که هر گروهشون یه رنگ و یه مرکز دارن و یه برچسب. این دقیقاً همون چیزیه که برای آموزش و تست مدل KNN نیاز داریم.

حالا نوبت تونه!

خیلی عالیه! حالا که داده‌های مصنوعی رو تولید کردیم و مفهومش رو فهمیدیم، وقتشه که اون‌ها رو برای آموزش و آزمایش مدل **KNN** خودمون آماده کنیم. این بخش از کد دقیقاً همین کار رو می‌کنه.

توابع فاصله (دوباره مرور کنیم!)

قبل از هر چیز، دوباره اون توابع محاسبه فاصله رو یادآوری کرده:

```
Python

:(def euclidean_distance(x1, x2
((return np.sqrt(np.sum((x1 - x2) ** 2

:(def manhattan_distance(x1, x2
((return np.sum(np.abs(x1 - x2
```

- **(euclidean_distance(x1, x2**: این تابع **فاصله اقلیدسی** رو محاسبه می‌کنه. اگه x1 و x2 دو نقطه تو فضای دو بعدی باشن، این فاصله همون خط راستیه که این دو نقطه رو به هم وصل می‌کنه (مثل فاصله روی نقشه). این رایج‌ترین نوع فاصله در KNN هست.
- **(manhattan_distance(x1, x2**: این تابع **فاصله منهتن** رو محاسبه می‌کنه. این فاصله مثل اینه که توی یک شهر با خیابان‌های شبکه‌ای بخوای از یه نقطه به یه نقطه دیگه بری و فقط مجاز باشی افقی یا عمودی حرکت کنی (مثل مسیر حرکت تاکسی).

این دو تابع رو قبلاً توی کلاس KNNClassifier تعریف کرده بودیم، اینجا فقط برای یادآوری آورده شده‌اند.

تقسیم داده‌ها به بخش‌های آموزش و آزمایش (train_test_split)

این یکی از **مهم‌ترین مراحل** در هر پروژه یادگیری ماشینه.

```
Python

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMatrixDisplay

test_size = 0.2  # 20% for testing

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

("{[print(f"Total samples: {X.shape[0
("{[print(f"Training samples: {X_train.shape[0
("{[print(f"Testing samples: {X_test.shape[0
```

چرا داده‌ها رو تقسیم می‌کنیم؟

تصور کن یه دانش‌آموز داری که می‌خوای ببینی چقدر ریاضی بلده. اگه تمام سوالاتی که ازش می‌پرسی، دقیقاً همون‌هایی باشه که قبلاً بهش یاد دادی، چطور می‌فهمی واقعاً یاد گرفته یا فقط حفظ کرده؟

در یادگیری ماشین هم همینطوره. اگه مدلمون رو با **تمام** داده‌هایی که داریم آموزش بدیم و بعد با **همون داده‌ها** امتحان کنیم، بهمون یه دقت (accuracy) خیلی بالا نشون میده. اما این دقت واقعی نیست! ممکنه مدل فقط داده‌های آموزشی رو **"حفظ"** کرده باشه و نتونه روی داده‌های جدید و ندیده شده خوب عمل کنه. به این مشکل میگن **"Overfitting" (بیش‌برازش)**.

برای جلوگیری از این مشکل، داده‌ها رو به دو بخش تقسیم می‌کنیم:

1. **داده‌های آموزشی (Training Data):** این بخشی از داده‌هاست که مدل از اون‌ها **یاد می‌گیره**. (مثل درس‌هایی که به دانش‌آموز میدی).
2. **داده‌های آزمایش (Testing Data):** این بخشی از داده‌هاست که مدل **هیچوقت** اون‌ها رو در مرحله آموزش ندیده و فقط برای **امتحان کردن عملکرد مدل** روی داده‌های جدید استفاده میشه. (مثل سوالات امتحانی که دانش‌آموز تا حالا ندیده).

توضیح خط به خط کد تقسیم داده:

1. `from sklearn.model_selection import train_test_split`: این خط ابزار `train_test_split` رو از کتابخانه `sklearn` وارد می‌کنه.
2. `test_size = 0.2`: این متغیر تعیین می‌کنه که چه **نسبتی از کل داده‌ها** برای **آزمایش (Testing)** کنار گذاشته بشه. `0.2` یعنی **20 درصد** از کل داده‌ها برای آزمایش و 80 درصد باقی‌مانده برای آموزش استفاده میشن.
3. `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, stratify=y, random_state=42)`: این دستور اصلی تقسیم داده‌هاست.
   - **`X`**: کل داده‌های ویژگی (نقاط).
   - **`y`**: کل برچسب‌های (کلاس‌های) مربوط به نقاط.
   - **`test_size=test_size`**: نسبت داده‌های آزمایش رو مشخص می‌کنه (همون 0.2 که تعریف کردیم).
   - **`stratify=y`**: این پارامتر خیلی مهمه! اگه ازش استفاده نکنیم، ممکنه تقسیم داده‌ها به صورت تصادفی باعث بشه که تعداد نقاط یک کلاس در بخش آموزش یا آزمایش نامتعادل بشه. مثلاً اگه 100 تا گربه و 1000 تا سگ داریم، بدون `stratify` ممکنه تو بخش آموزش 100 گربه و 100 سگ بیفته و تو بخش آزمایش 900 سگ! `stratify=y` تضمین می‌کنه که **نسبت کلاس‌ها** (تعداد گربه‌ها به سگ‌ها) در بخش‌های آموزش و آزمایش **مشابه کل داده‌ها** باقی بمونه. این باعث میشه که مدل روی نمونه‌های هر کلاس به اندازه کافی آموزش ببینه و تست بشه.
   - **`random_state=42`**: این هم مثل `np.random.seed(40)` در تابع تولید داده‌هامون عمل می‌کنه. اگه این رو نذاریم، هر بار که کد رو اجرا کنی، داده‌ها به شکل متفاوتی تقسیم میشن. با قرار دادن یک عدد (مثلاً 42)، تضمین می‌کنه که تقسیم‌بندی داده‌ها هر بار **دقیقاً یکسان** باشه. این برای قابلیت تکرار نتایج و مقایسه آزمایشات خیلی مهمه.
   - **خروجی این تابع:**
     - `X_train`: ویژگی‌های (نقاط) بخش آموزش.
     - `X_test`: ویژگی‌های (نقاط) بخش آزمایش.
     - `y_train`: برچسب‌های (کلاس‌های) بخش آموزش.
     - `y_test`: برچسب‌های (کلاس‌های) بخش آزمایش.
4. **`print(f"Total samples: {X.shape[0]}")` و بقیه خطوط `print`:**
   - این خطوط فقط برای اینه که اطلاعاتی در مورد تعداد نقاط در هر بخش نمایش داده بشه.
   - `X.shape[0]` تعداد سطرها (یعنی تعداد کل نقاط) در آرایه `X` رو نشون میده.

خلاصه این بخش:

ما در این بخش:

- یادآوری کردیم که مدل KNN برای کارش نیاز به محاسبه **فاصله** بین نقاط داره.
- مهمتر از اون، یاد گرفتیم که چطور داده‌هایمون رو به دو بخش **آموزش (Train)** و **آزمایش (Test)** تقسیم کنیم. این کار برای ارزیابی **واقعی** عملکرد مدل و جلوگیری از "حفظ کردن" داده‌ها توسط مدل، حیاتیه.

حالا که داده‌ها رو داریم و آماده‌سازی‌ها انجام شده، در مرحله بعد می‌تونیم مدل KNNClassifier رو که ساختیم، روی این داده‌ها آموزش بدیم و عملکردش رو ارزیابی کنیم!

بسیار خب! حالا که داده‌ها آماده و تقسیم‌بندی شدن، و مدل KNN رو هم ساختیم، چطور می‌تونیم ببینیم که این مدل واقعاً چطور کار می‌کنه؟ یعنی چطور نقاط رو دسته‌بندی می‌کنه؟

اینجاست که تابع plot_decision_boundaries وارد عمل میشه. این تابع یه کار بسیار جالب و بصری رو انجام میده: **مرزهای تصمیم‌گیری (Decision Boundaries)** مدل رو رسم می‌کنه.

مرز تصمیم‌گیری (Decision Boundary) چیست؟

فرض کن یه دیوار کشیدی و گربه‌ها رو یه طرف دیوار و سگ‌ها رو طرف دیگه گذاشتی. اون دیوار، مرز تصمیم‌گیری توئه!

در مدل‌های دسته‌بندی (مثل KNN)، مرز تصمیم‌گیری خط یا سطحی هست که فضاهای مربوط به کلاس‌های مختلف رو از هم جدا می‌کنه. هر نقطه‌ای که در یک سمت این مرز قرار بگیره، به یک کلاس و هر نقطه‌ای که در سمت دیگه قرار بگیره، به کلاس دیگه اختصاص داده میشه.

در **KNN**، این مرزها همیشه صاف و مستقیم نیستن؛ می‌تونن نامنظم و منحنی باشن، چون بر اساس نزدیک‌ترین همسایه‌ها تصمیم‌گیری می‌کنن.

توضیح کد plot_decision_boundaries

```Python
def plot_decision_boundaries(classifier, X, y, means, title='Decision Boundaries'):
    # 1. تعیین محدوده نمودار
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    # 2. ایجاد شبکه ای از نقاط
    h = 0.1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
```

```
# 3. پیش‌بینی برای هر نقطه در شبکه
grid_points = np.c_[xx.ravel(), yy.ravel()]
Z = classifier.predict(grid_points)
Z = Z.reshape(xx.shape)

# 4. رسم مرزها و نقاط اصلی
plt.figure(figsize=(10,8))
colors = plt.get_cmap('tab10', np.unique(y).size + 1)

plt.contourf(xx, yy, Z, alpha=0.5, cmap=colors) # رسم مناطق رنگی (مرزها)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=colors, edgecolor='k', s=40) # رسم نقاط داده اصلی

# 5. اضافه کردن عنوان، برچسب و راهنما
plt.title(title)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(handles=[
    plt.Line2D([0], [0], marker='o', color='w', label=f'Class {i}',
               markerfacecolor=colors(i), markersize=10) for i in range(np.unique(y).size)
], loc='upper right')
plt.show()
```

توضیح پارامترها (ورودی‌ها):

- `classifier`: این همون شیء مدل `KNNClassifier` هست که ما ساختیم (یا هر مدل دسته‌بندی دیگری). این تابع از متد `predict` این مدل استفاده می‌کنه.
- `X`: تمام داده‌های ویژگی (نقاط).
- `y`: تمام برچسب‌های (کلاس‌های) مربوط به نقاط.
- `means`: مراکز خوشه‌های واقعی (در اینجا استفاده نمیشه، اما در تابع اصلی هست).
- `title='Decision Boundaries'`: عنوانی برای نمودار.

توضیح خط به خط تابع:

1. **تعیین محدوده نمودار (`x_min, x_max, y_min, y_max`):**
   - `X[:, 0].min() - 1, X[:, 0].max() + 1`: این خطوط کوچکترین و بزرگترین مقدار در ستون اول (مختصات x) از همه نقاط `X` رو پیدا می‌کنن و یه واحد هم به طرفین (برای حاشیه) اضافه/کم می‌کنن.
   - هدف: تعیین مرزهای افقی و عمودی (کمترین و بیشترین x و y) برای کل نمودار، تا همه نقاط و فضای اطرافشون رو پوشش بده.
2. **ایجاد شبکه‌ای از نقاط (`xx, yy` و `h` و `np.meshgrid`):**
   - **`h = 0.1`**: این **گام (step size)** شبکه هست. یعنی هر 0.1 واحد، یک نقطه جدید روی شبکه ایجاد می‌کنیم. اگه این عدد کوچکتر باشه، شبکه متراکم‌تر و دقیق‌تر میشه، اما زمان محاسبات بیشتر میشه.
   - **`np.arange(x_min, x_max, h)`**: یک آرایه از اعداد ایجاد می‌کنه که از `x_min` شروع میشه و با گام `h` تا `x_max` ادامه پیدا می‌کنه.
   - **`np.meshgrid(...,...)`**: این تابع از `numpy` یک **شبکه (Grid)** از نقاط ایجاد می‌کنه.

- تصور کن یه صفحه شطرنجی داری. np.meshgrid مختصات x (سطرهای xx) و y (ستون‌های yy) تمام تقاطع‌های این شطرنج رو بهت میده.
- هدف: ما می‌خوایم برای **هر نقطه ممکن** در فضای نمودار، پیش‌بینی کنیم که مدل اون نقطه رو به کدوم کلاس اختصاص میده. این شبکه به ما کمک می‌کنه تا تمام این نقاط رو پوشش بدیم.

3. **پیش‌بینی برای هر نقطه در شبکه (grid_points, Z = classifier.predict(...), Z.reshape):**
   - **grid_points = np.c_[xx.ravel(), yy.ravel()]:**
     - xx.ravel() و yy.ravel(): این‌ها آرایه‌های xx و yy رو به یک آرایه تک‌بعدی (فلت) تبدیل می‌کنن.
     - np.c_[]: این تابع numpy دو تا آرایه رو **ستون به ستون** کنار هم می‌ذاره.
     - نتیجه: grid_points یک آرایه بزرگ از **تمام نقاط** روی شبکه (مثلاً [[x1,y1], [x2,y2], ...]) هست.
   - **Z = classifier.predict(grid_points)**: این خط مهم‌ترین بخش این مرحله هست. ما مدل classifier (یعنی همون KNNClassifier خودمون) رو صدا می‌زنیم و بهش می‌گیم: "برای **تمام این نقاط شبکه (grid_points)**، پیش‌بینی کن که هر کدوم به کدوم کلاس تعلق دارن."
     - خروجی Z یک آرایه از برچسب‌های پیش‌بینی شده برای هر نقطه در شبکه هست (مثلاً [0, 0, 1, 2, 0, ...]).
   - predict: **Z = Z.reshape(xx.shape)** خروجی رو به صورت یک لیست یک‌بعدی برمی‌گردونه. ما باید این لیست رو دوباره به شکل **شبکه اصلی (xx.shape)** در بیاریم تا بتونیم اون رو روی نمودار رسم کنیم.
4. **رسم مرزها و نقاط اصلی (plt.figure, plt.contourf, plt.scatter):**
   - **plt.figure(figsize=(10,8))**: یک پنجره جدید برای نمودار ایجاد می‌کنه با اندازه مشخص.
   - **colors = plt.get_cmap('tab10', np.unique(y).size + 1)**: یک پالت رنگی مناسب انتخاب می‌کنه. np.unique(y).size تعداد کلاس‌های منحصر به فرد در داده‌های ماست.
   - **plt.contourf(xx, yy, Z, alpha=0.5, cmap=colors)**: این دستور جادویی **رنگ‌آمیزی مناطق** روی نمودار رو انجام میده!
     - xx, yy: مختصات شبکه رو مشخص می‌کنن.
     - Z: برچسب‌های پیش‌بینی شده برای هر نقطه از شبکه هستن.
     - alpha=0.5: شفافیت رنگ‌ها رو تعیین می‌کنه (تا بتونیم نقاط اصلی رو هم زیرش ببینیم).
     - cmap=colors: پالت رنگی رو مشخص می‌کنه.
     - **نتیجه:** این خط باعث میشه که هر ناحیه‌ای از نمودار که مدل پیش‌بینی می‌کنه متعلق به یک کلاس خاصه، با رنگ اون کلاس پر بشه. **مرزهای بین این مناطق رنگی، همون "مرزهای تصمیم‌گیری" هستن.**
   - **plt.scatter(X[:, 0], X[:, 1], c=y, cmap=colors, edgecolor='k', s=40)**: این خط همون نقاط **داده اصلی** ما رو روی نمودار رسم می‌کنه.
     - X[:, 0] و X[:, 1]: مختصات x و y تمام نقاط داده ما.
     - c=y: رنگ هر نقطه رو بر اساس برچسب اصلی خودش (y) تنظیم می‌کنه.
     - edgecolor='k' و s=40: نقاط رو با حاشیه سیاه و اندازه مشخصی رسم می‌کنه.
     - **هدف:** این نقاط نشون میدن که داده‌های اصلی ما کجا هستن و آیا مرزهای تصمیم‌گیری به درستی اون‌ها رو جدا کردن یا نه.
5. **اضافه کردن عنوان، برچسب و راهنما (plt.title, plt.xlabel, plt.ylabel, plt.legend):**
   - این خطوط فقط برای زیباتر شدن نمودار و قابل فهم‌تر شدنش هستن.
   - plt.legend(...): یک راهنمای سفارشی برای نمودار ایجاد می‌کنه که نشون میده هر رنگ مربوط به کدوم کلاس هست.

خروجی این تابع چیست؟

وقتی این تابع رو صدا می‌زنی (در مراحل بعدی این کار رو خواهیم کرد)، یه نمودار دو بعدی می‌بینی. روی این نمودار:

- **مناطق رنگی:** نشون‌دهنده "مناطق نفوذ" هر کلاس هستن. یعنی اگر یک نقطه جدید در هر کجای این مناطق قرار بگیره، مدل KNN اون رو به رنگ اون منطقه دسته‌بندی می‌کنه.
- **خطوط بین مناطق رنگی:** همین‌ها **مرزهای تصمیم‌گیری** هستن.
- **نقاط سیاه با رنگ‌های داخلی:** این‌ها همون نقاط داده‌های آموزشی واقعی ما هستن که با رنگ کلاس اصلی خودشون مشخص شدن.

با دیدن این نمودار، می‌تونی بفهمی که مدل KNN چطور فضای رو تقسیم کرده و نقاط جدید رو چطور دسته‌بندی می‌کنه.

خیلی عالیه! حالا که همه‌چیز رو آماده کردیم (کلاس KNN، داده‌های مصنوعی، تقسیم‌بندی داده‌ها و تابع رسم مرز تصمیم)، وقتشه که مدل رو آموزش بدیم و عملکردش رو **ارزیابی** کنیم. این بخش از کد دقیقاً همین کار رو انجام میده و به ما کمک می‌کنه بهترین مقدار k رو برای مدل KNN پیدا کنیم.

یافتن بهترین K برای KNN

همانطور که قبلاً صحبت کردیم، k در KNN تعداد "نزدیک‌ترین همسایه‌ها" رو مشخص می‌کنه. انتخاب k مناسب خیلی مهمه؛ اگه k خیلی کوچیک باشه، مدل ممکنه به نویز (Noise) حساس بشه و "بیش‌برازش" (Overfitting) پیدا کنه. اگه k خیلی بزرگ باشه، ممکنه مدل "کم‌برازش" (Underfitting) پیدا کنه و جزئیات رو از دست بده.

این کد تلاش می‌کنه تا با آزمایش مقادیر مختلف k، ببینه کدوم k بهترین **دقت (Accuracy)** رو به ما میده.

توضیح کد خط به خط

```
Python

k_values = range(1, 36) # 1: برای آزمایش k مقادیر مختلف

accuracies = [] # 2: k لیستی برای ذخیره دقت های هر

for k in k_values: # 3: k حلقه ای برای تست هر
    knn = KNNClassifier(k=k, distance_func=euclidean_distance) # 4: فعلی و فاصله k با KNN ساخت یک مدل
اقلیدسی
    knn.fit(X_train, y_train) # 5: آموزش مدل روی داده های آموزشی
    y_pred = knn.predict(X_test) # 6: پیش بینی برچسب ها برای داده های آزمایشی
    acc = accuracy_score(y_test, y_pred) # 7: محاسبه دقت مدل
    accuracies.append(acc) # 8: اضافه کردن دقت به لیست accuracies

# plotting Accuracy for different values of k
plt.figure(figsize=(10,6)) # 9: ایجاد نمودار
```

```
plt.plot(k_values, accuracies, marker='o', linestyle='-', color='b') # 10: رسم خط دقت ها
plt.title('kNN Classification Accuracy for Different k Values') # 11: عنوان نمودار
plt.xlabel('Number of Neighbors (k)') # 12: برچسب محور افقی
plt.ylabel('Accuracy') # 13: برچسب محور عمودی
plt.xticks(k_values) # 14: روی محور افقی k نمایش همه مقادیر
plt.grid(True) # 15: نمایش خطوط شبکه ای
plt.show() # 16: نمایش نمودار
```

1. **k_values = range(1, 36):**
   - این یک **بازه از اعداد صحیح** رو تعریف می‌کنه. range(1, 36) یعنی اعداد از 1 شروع میشن و تا 35 (شامل 35) ادامه پیدا می‌کنن. پس ما می‌خوایم مدل KNN رو با kهای 1، 2، 3 و... تا 35 امتحان کنیم.
2. **accuracies = []:**
   - یک لیست خالی به اسم accuracies (دقت‌ها) ایجاد می‌کنه. ما قراره دقت مدل رو برای هر k حساب کنیم و نتیجه رو داخل این لیست ذخیره کنیم.
3. **for k in k_values::**
   - یک **حلقه for** که به ازای هر مقدار k در لیست k_values یک بار اجرا میشه. یعنی این عملیات 35 بار تکرار میشه، یک بار برای k=1، یک بار برای k=2 و...
4. **knn = KNNClassifier(k=k, distance_func=euclidean_distance):**
   - در هر بار تکرار حلقه، یک **شیء جدید** از کلاس KNNClassifier می‌سازیم.
   - این شیء k رو برابر با مقدار فعلی k در حلقه (مثلاً 1، بعد 2، بعد 3 و...) قرار میده.
   - distance_func=euclidean_distance هم بهش میگه که برای محاسبه فاصله از **فاصله اقلیدسی** استفاده کنه.
5. **knn.fit(X_train, y_train):**
   - متد fit مدل knn رو فراخوانی می‌کنه.
   - در این مرحله، مدل داده‌های **آموزشی (X_train, y_train)** رو "به خاطر می‌سپره" تا بتونه بر اساس اون‌ها پیش‌بینی کنه. (همانطور که قبلاً گفتیم، در KNN مرحله آموزش خیلی ساده است و فقط شامل ذخیره‌سازی داده‌ها میشه).
6. **y_pred = knn.predict(X_test):**
   - متد predict مدل knn رو فراخوانی می‌کنه.
   - این متد، برچسب‌های پیش‌بینی شده رو برای **داده‌های آزمایشی (X_test)** محاسبه می‌کنه.
   - نتیجه (برچسب‌های پیش‌بینی شده) در متغیر y_predicted) y_pred) ذخیره میشه.
7. **acc = accuracy_score(y_test, y_pred):**
   - accuracy_score تابعی از sklearn.metrics هست که برای محاسبه **دقت (Accuracy)** استفاده میشه.
   - این تابع دو لیست رو با هم مقایسه می‌کنه:
     - y_test: برچسب‌های **واقعی** داده‌های آزمایشی.
     - y_pred: برچسب‌هایی که **مدل پیش‌بینی کرده**.
   - دقت مدل برابر است با "تعداد پیش‌بینی‌های صحیح" تقسیم بر "تعداد کل پیش‌بینی‌ها". نتیجه یک عدد بین 0 تا 1 است (مثلاً 0.9 به معنی 90% دقت).
8. **accuracies.append(acc):**
   - مقدار دقت acc که در مرحله قبل محاسبه شد رو به لیست accuracies اضافه می‌کنه.

رسم نمودار دقت (Accuracy Plotting)

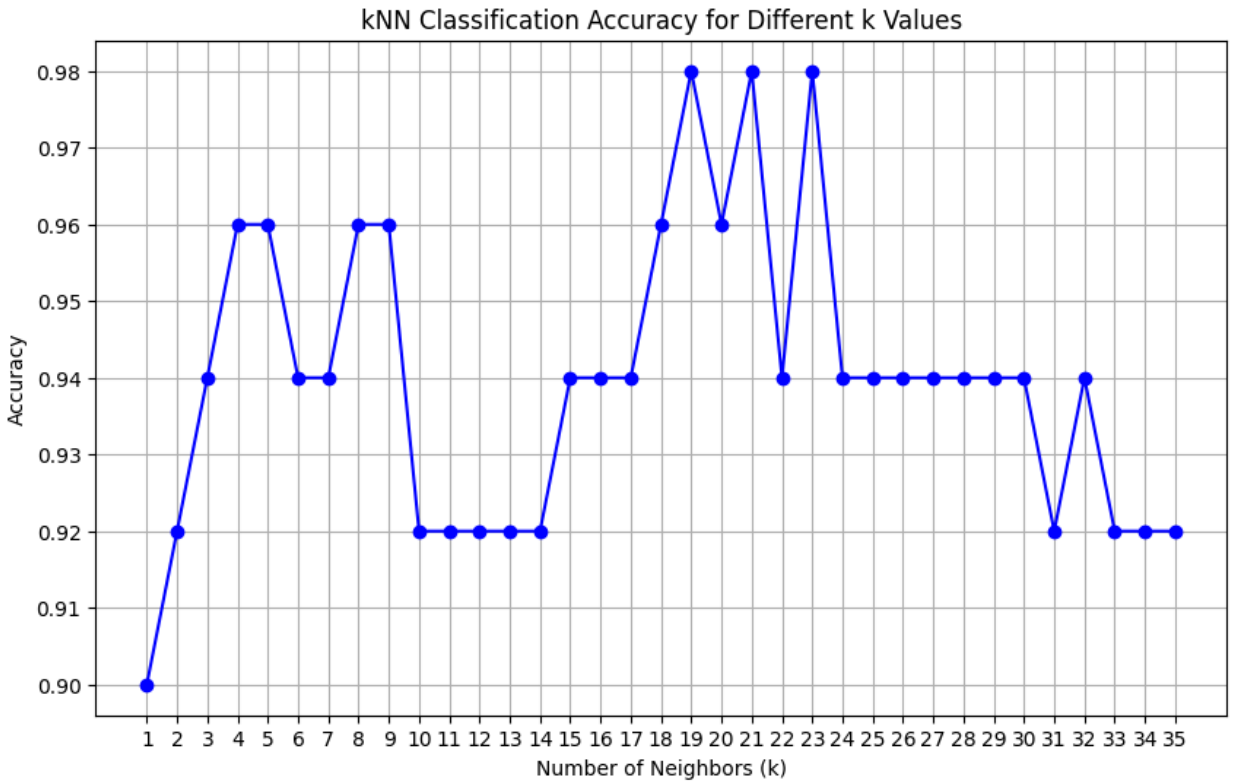
بعد از اینکه حلقه تموم شد و دقت‌ها رو برای تمام مقادیر k جمع‌آوری کردیم، حالا می‌تونیم یک نمودار رسم کنیم تا ببینیم دقت چطور با تغییر k تغییر می‌کنه.

9. **((plt.figure(figsize=(10,6**:
   - یک پنجره نمودار جدید با اندازه مشخص ایجاد می‌کنه.
10. **('plt.plot(k_values, accuracies, marker='o', linestyle='-', color='b**:
    - این خط اصلی برای رسم نمودار خطی هست.
      - k_values: مقادیر روی محور افقی (X-axis) که تعداد همسایه‌ها (k) رو نشون میده.
      - accuracies: مقادیر روی محور عمودی (Y-axis) که دقت مدل رو نشون میده.
      - 'marker='o: نقاط داده رو به شکل دایره نشون میده.
      - '-'=linestyle: نقاط رو با خط ممتد به هم وصل می‌کنه.
      - 'color='b: رنگ خط رو آبی (blue) قرار میده.
11. **('plt.title('kNN Classification Accuracy for Different k Values**:
    - عنوان نمودار رو تنظیم می‌کنه.
12. **('plt.xlabel('Number of Neighbors (k**:
    - برچسب محور افقی (X-axis) رو تعیین می‌کنه.
13. **('plt.ylabel('Accuracy**:
    - برچسب محور عمودی (Y-axis) رو تعیین می‌کنه.
14. **(plt.xticks(k_values**:
    - تضمین می‌کنه که تمام مقادیر k (از 1 تا 35) روی محور افقی نمایش داده بشن، نه فقط چندتا از اون‌ها.
15. **(plt.grid(True**:
    - خطوط شبکه‌ای رو روی نمودار نمایش میده که خوندن مقادیر رو راحت‌تر می‌کنه.
16. **()plt.show**:
    - در نهایت نمودار رو نمایش میده.

نتیجه‌ای که می‌بینی

با اجرای این کد، یک نمودار میبینی که محور افقی اون "k" (تعداد همسایه‌ها) و محور عمودی اون "دقت" (Accuracy) هست. با نگاه کردن به این نمودار، می‌تونی نقطه‌ای رو پیدا کنی که در اون دقت مدل به بیشترین مقدار خودش رسیده. این k، احتمالاً **بهترین مقدار k** برای داده‌های ما در این سناریو هست.

با این آزمایش، ما داریم مدل رو "بهینه" می‌کنیم تا بهترین عملکرد رو داشته باشه.



بسیار عالی! این دقیقاً همون نموداری هست که انتظار داشتیم. حالا بیا این نمودار رو با هم تحلیل کنیم تا بفهمیم بهمون چی میگه و چطور می‌تونیم ازش برای انتخاب بهترین k استفاده کنیم.

تحلیل نمودار kNN Classification Accuracy for Different k Values

این نمودار به ما نشون میده که **دقت (Accuracy)** مدل **KNN** چطور با تغییر مقدار k (تعداد همسایه‌ها) تغییر می‌کنه.

محورها:

- **محور افقی (X-axis): Number of Neighbors (k)**
  - این محور مقادیر مختلف k رو نشون میده که ما از 1 تا 35 (همون k_values = range(1, 36)) آزمایش کردیم.
- **محور عمودی (Y-axis): Accuracy**
  - این محور دقت مدل رو نشون میده که بین 0 (0%) و 1 (100%) متغیره. هرچی عدد به 1 نزدیک‌تر باشه، دقت مدل بیشتره.

چه چیزهایی رو از نمودار می‌تونیم بفهمیم؟

1. **رفتار دقت با kهای کوچک (مثلاً k=1 تا k=8):**

- برای k=1، دقت حدود 0.90 (90%) هست.
- وقتی k به 2، 3 یا 4 افزایش پیدا می‌کنه، دقت به طور قابل توجهی بالا میره و به 0.96 (96%) می‌رسه. این نشون میده که استفاده از فقط 1 همسایه ممکنه مدل رو به نویزها حساس کنه (Overfitting).
- برای k=5 تا k=8، دقت کمی نوسان داره اما همچنان در محدوده خوب 0.94 تا 0.96 باقی می‌مونه.

2. **افت دقت برای kهای میانی (مثلاً k=9 تا k=13):**
   - می‌بینیم که از k=9 تا k=13، دقت به شدت افت می‌کنه و به 0.92 (92%) می‌رسه و در این محدوده ثابت می‌مونه. این نشون میده که برای این داده‌های خاص ما، انتخاب k در این بازه باعث میشه مدل ضعیف‌تر عمل کنه.
3. **بهبود و اوج دقت در kهای بالاتر (مثلاً k=14 تا k=22):**
   - از k=14 دقت دوباره شروع به افزایش می‌کنه و در k=15 به 0.94 می‌رسه.
   - نکته جالب اینجاست که برای k=21، k=19 و k=23، دقت به **بالاترین مقدار خودش یعنی 0.98 (98%)** میرسه! اینا "نقاط شیرین" یا "sweet spot" ما هستن.
4. **کاهش و ثبات نسبی در kهای خیلی بزرگ (مثلاً k=24 تا k=35):**
   - بعد از k=23، دقت دوباره کمی افت می‌کنه و در 0.94 ثابت میشه.
   - نزدیک k=32، دقت به 0.92 کاهش پیدا می‌کنه و تا k=35 در همین محدوده می‌مونه.

چطور بهترین k را انتخاب کنیم؟

هدف ما پیدا کردن **بیشترین دقت ممکن** است. با توجه به این نمودار، بهترین دقت (0.98) در **k=21، k=19 و k=23** اتفاق افتاده.

- معمولاً بین مقادیر k که بهترین دقت رو میدن، مقدار **کوچکتر** رو انتخاب می‌کنیم. چرا؟
  - چون هرچه k بزرگتر باشه، محاسبات بیشتری باید انجام بشه و مدل کندتر میشه.
  - همچنین، kهای خیلی بزرگ ممکنه باعث بشن مدل بیش از حد "عمومی" (Underfit) عمل کنه و جزئیات مهم رو نادیده بگیره.
  - در این حالت، **k=19** می‌تونه یک انتخاب عالی باشه، چون اولین مقداری است که به بالاترین دقت می‌رسد. k=21 و k=23 هم به همین اندازه خوب هستند.

نتیجه‌گیری از این آزمایش:

این نمودار به ما می‌گه که برای داده‌های مصنوعی که تولید کردیم، مدل **KNN** در حالت k=19 (یا 21 یا 23) با استفاده از فاصله اقلیدسی، **98% دقت** روی داده‌های آزمایشی (داده‌های ندیده شده) داره. این یک عملکرد بسیار عالیه!

این آزمایش به ما کمک می‌کنه تا پارامتر مهم k در مدل KNN رو بهینه کنیم.

آیا این تحلیل نمودار برات مفید بود و بهت کمک کرد که نتایج رو درک کنی؟

این سوال بسیار اساسی و مهمیه! تشخیص **Overfitting (بیش‌برازش)** و **Underfitting (کم‌برازش)** یکی از چالش‌های کلیدی در یادگیری ماشینه. اگه این دو مفهوم رو خوب درک کنی، می‌تونی مدل‌های خیلی بهتری بسازی.

بیا با مثال و تصویر ذهنی توضیح بدم:

تصویر ذهنی:

فرض کن یه معلم (مدل هوش مصنوعی) داری که می‌خوای ازش بخوای برای امتحان ریاضی (یک مسئله) آماده‌ات کنه.

1. Overfitting (بیش‌برازش) - "حفظ کردن" به جای "فهمیدن"

- **معلم Overfit:** فرض کن معلم به جای اینکه مفاهیم ریاضی رو بهت یاد بده، **دقیقاً همون سوالاتی که قراره تو امتحان بیاد رو با جواباش بهت میده و ازت می‌خواد حفظ کنی.**
  - **نتیجه:** تو سوالات رو حفظ می‌کنی و در امتحان **قبلی (داده آموزشی)** نمره 20 می‌گیری.
  - **مشکل:** وقتی سوالات جدید (شبیه به همون سوالات قبلی ولی با اعداد یا جزئیات کمی متفاوت) تو امتحان **نهایی (داده واقعی/تست)** میاد، تو نمی‌تونی جواب بدی چون فقط حفظ کردی و مفهوم رو نفهمیدی. نمره‌ات تو امتحان نهایی خیلی پایین میشه.
- **در مدل هوش مصنوعی:**
  - مدل روی **داده‌های آموزشی (Training Data)** عملکرد **بسیار عالی** (دقت خیلی بالا) داره. یعنی "حفظشون کرده".
  - اما وقتی با **داده‌های جدید و ندیده شده (Testing Data)** روبرو میشه، عملکردش **به شدت افت می‌کنه**.
  - **علامت تشخیص: اختلاف زیاد بین دقت روی داده‌های آموزشی و دقت روی داده‌های آزمایشی.** (دقت آموزش بالا، دقت تست پایین).
  - **نمودار kNN Classification Accuracy ما:**
    - اگه برای k=1 (که معمولاً مستعد Overfitting هست) دقت روی داده‌های آموزشی مثلاً 99% باشه ولی دقت روی داده‌های آزمایشی (که ما رسم کردیم) 90% باشه، این یه نشونه از Overfitting‌ه. مدل برای k=1 فقط به نزدیک‌ترین همسایه نگاه می‌کنه و جزئیات و نویزهای داده‌های آموزشی رو هم یاد میگیره.

2. Underfitting (کم‌برازش) - "یاد نگرفتن کافی"

- **معلم Underfit:** فرض کن معلم به جای اینکه درس بده، فقط بهت بگه "درس بخون". هیچ توضیح یا تمرینی بهت نده.
  - **نتیجه:** تو هیچ چیزی یاد نمی‌گیری.
  - **مشکل:** نه تو امتحان **قبلی (داده آموزشی)** نمره خوبی می‌گیری و نه تو امتحان **نهایی (داده واقعی/تست)**. عملکردت روی هر دو بد هست.
- **در مدل هوش مصنوعی:**
  - مدل روی **داده‌های آموزشی (Training Data)** عملکرد **ضعیفی** (دقت پایین) داره. یعنی نتونسته مفاهیم و الگوها رو به اندازه کافی یاد بگیره.
  - به تبع، روی **داده‌های جدید و ندیده شده (Testing Data)** هم عملکردش **ضعیفه**.
  - **علامت تشخیص: دقت پایین (یا نسبتاً پایین) روی هر دو مجموعه داده‌های آموزشی و آزمایشی.**
  - **نمودار kNN Classification Accuracy ما:**
    - اگه دقت‌ها روی نمودار ما مثلاً کلاً زیر 70% بودن و دیگه بالاتر نمیرفتن، این می‌تونست یه نشونه از Underfitting باشه. مثلاً اگه k رو خیلی خیلی بزرگ انتخاب کنیم (مثلاً k=100 برای 150 تا نقطه)، مدل میانگین کل کلاس‌ها رو می‌گیره و دیگه نمیتونه بین کلاس‌ها تمایز قائل بشه، که باعث Underfitting میشه.

خلاصه و مقایسه در یک نگاه:

مشکل	دقت آموزش (Training Accuracy)	دقت تست (Testing Accuracy)	توضیحات
Overfitting	**خیلی بالا**	**پایین**	مدل داده‌های آموزشی را حفظ کرده، اما الگوی کلی را یاد نگرفته.
Underfitting	**پایین**	**پایین**	مدل حتی داده‌های آموزشی را هم خوب یاد نگرفته و الگوها را درک نکرده است.
Good Fit	**بالا**	**بالا و نزدیک به آموزش**	مدل الگوها را خوب یاد گرفته و می‌تواند روی داده‌های جدید هم خوب عمل کند.

چطور در عمل بفهمیم؟

برای تشخیص دقیق Overfitting و Underfitting، معمولاً در طول آموزش مدل (یا بعد از آن)، **هم دقت روی داده‌های آموزشی و هم دقت روی داده‌های آزمایشی** را محاسبه و مقایسه می‌کنند.

در کد ما (KNN):

ما در حلقه‌ای که Accuracy را برای kهای مختلف محاسبه کردیم، فقط دقت روی X_test (داده‌های آزمایشی) را ذخیره کردیم.
برای تشخیص دقیق‌تر، می‌توانستیم دقت روی X_train را هم محاسبه کنیم و یک خط دوم در نمودار برای آن رسم کنیم.

مثلاً:

```
Python

# ... (کدهای قبلی) ...

train_accuracies = [] # لیست جدید برای دقت های آموزش
test_accuracies = []  # لیست برای دقت های تست (همان accuracies قبلی)

for k in k_values:
    knn = KNNClassifier(k=k, distance_func=euclidean_distance)
    knn.fit(X_train, y_train)
```

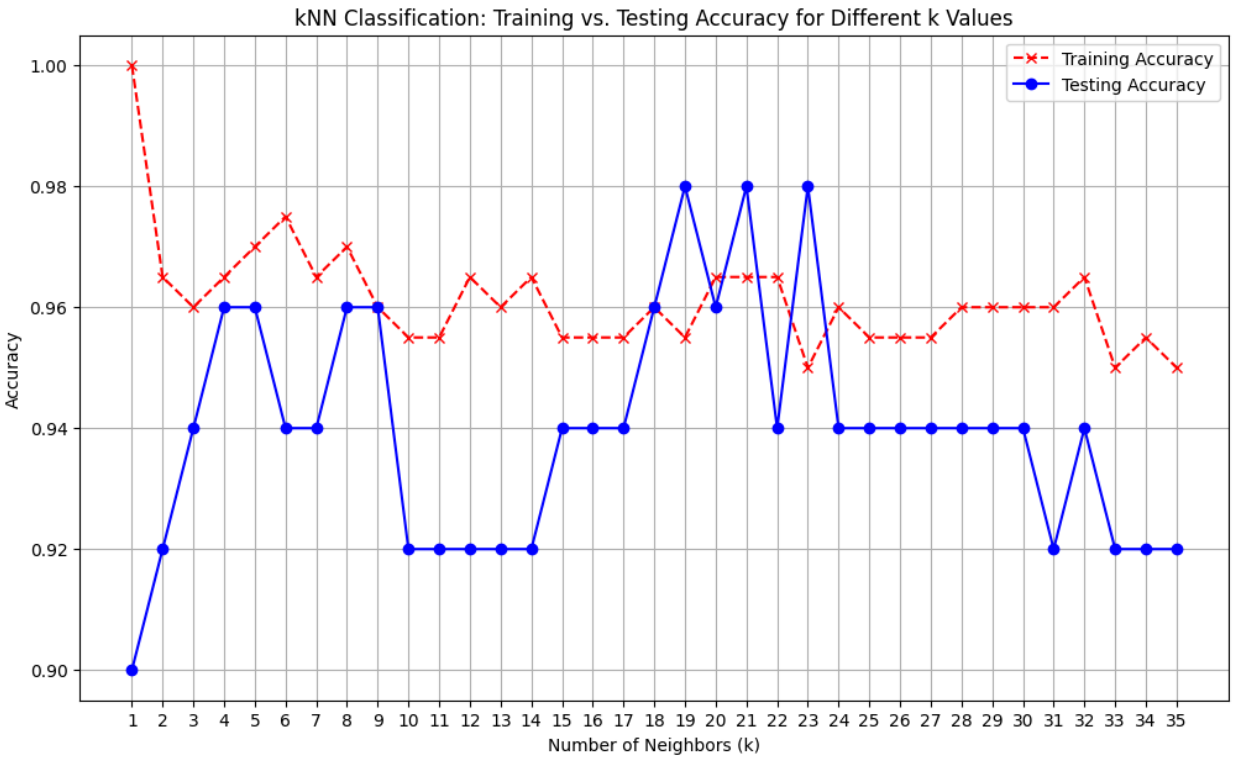
```
(y_pred_train = knn.predict(X_train) # پیش بینی روی داده های آموزش
(train_acc = accuracy_score(y_train, y_pred_train
(train_accuracies.append(train_acc

(y_pred_test = knn.predict(X_test) # پیش بینی روی داده های تست
(test_acc = accuracy_score(y_test, y_pred_test
(test_accuracies.append(test_acc

# ... (کدهای رسم نمودار) ...

('plt.plot(k_values, train_accuracies, marker='x', linestyle='--', color='r', label='Training Accuracy') # خط برای دقت آموزش
('plt.plot(k_values, test_accuracies, marker='o', linestyle='-', color='b', label='Testing Accuracy') # خط برای دقت تست
plt.legend() # برای نمایش برچسب خطوط

# ... (ادامه کد نمودار) ...
```

با رسم دو خط (یکی برای دقت آموزش و یکی برای دقت تست)، می‌توانی به وضوح Overfitting یا Underfitting را تشخیص دهی:

- **اگر خط آبی (تست) خیلی پایین‌تر از خط قرمز (آموزش) باشد:** Overfitting
- **اگر هر دو خط (تست و آموزش) همزمان پایین باشند:** Underfitting
- **اگر هر دو خط (تست و آموزش) همزمان بالا و نزدیک به هم باشند:** Good Fit (مدل خوب برازش شده)

در نموداری که تو فرستادی، خط دقت (که مربوط به داده‌های تست بود) برای k=1، از 98% (که در k=19 دیدیم) به 90% افت کرده. این یک **نشانه‌ی خفیف از Overfitting** برای k=1 هست. یعنی مدل برای k=1 روی داده‌های آموزش احتمالاً عملکرد بالاتری (مثلاً 100%) داشته، ولی روی داده‌های تست افت کرده.

امیدوارم این توضیح مفهوم Overfitting و Underfitting رو کاملاً برات روشن کرده باشه. این دو مفهوم ستون فقرات ارزیابی مدل‌های یادگیری ماشین هستند. سوالی داری؟



بسیار عالی! این نمودار جدید که شامل **هم دقت آموزش و هم دقت تست** هست، دقیقا چیزیه که برای تحلیل Overfitting و Underfitting نیاز داشتیم. حالا بیایید این نمودار رو با دقت بیشتری بررسی کنیم.

تحلیل نمودار kNN Classification: Training vs. Testing Accuracy for Different k Values

این نمودار دو خط رو نشون میده:

- **خط قرمز (با ضربدر و خطچین): Training Accuracy (دقت آموزش)**
  - این خط نشون میده که مدل روی **داده‌هایی که باهاشون آموزش دیده**، چقدر خوب عمل می‌کنه.
- **خط آبی (با دایره و خط ممتد): Testing Accuracy (دقت تست)**
  - این خط نشون میده که مدل روی **داده‌های جدید و ندیده شده**، چقدر خوب عمل می‌کنه.

تحلیل مقادیر مختلف k:

1. **k = 1 (گوشه سمت چپ نمودار):**
   - **دقت آموزش (خط قرمز):** دقیقاً 1.00 (100%) هست.
   - **دقت تست (خط آبی):** حدود 0.90 (90%) هست.
   - **نتیجه:** در اینجا ما یک مورد واضح از **Overfitting (بیش‌برازش)** داریم! مدل برای k=1، داده‌های آموزشی رو "کامل حفظ" کرده و به همه نویزها و جزئیات اون‌ها توجه کرده. به همین دلیل، وقتی با داده‌های جدید (تست) روبرو میشه، نمی‌تونه خوب تعمیم بده و دقتش افت می‌کنه. این تفاوت 10% (100% آموزش در مقابل 90% تست) یک نشانه قوی از Overfitting هست.
2. **k = 2 تا k = 3:**

- دقت آموزش کمی افت می‌کنه (از 100% به حدود 97% و بعد 96%)، که نشون میده مدل داره کمتر حفظ می‌کنه.
- دقت تست شروع به افزایش می‌کنه (به حدود 0.94).
- این یعنی مدل داره به سمت برازش بهتر حرکت می‌کنه.

3. **k = 4 تا k = 5:**
   - دقت آموزش دوباره کمی افزایش پیدا می‌کنه.
   - **دقت تست به 0.96 می‌رسه.** در اینجا، فاصله بین دقت آموزش و تست کمتر شده و مدل عملکرد خوبی داره.
4. **k = 9 تا k = 13:**
   - **دقت تست به 0.92 افت می‌کنه.**
   - دقت آموزش هم نوسان داره اما همچنان بالاتر از دقت تست باقی می‌مونه. این نشون میده که این kها مناسب نیستن.
5. **k = 19, k = 21, k = 23 (اوج‌های دقت تست):**
   - **دقت تست به 0.98 می‌رسه.** این بالاترین دقتیه که مدل ما روی داده‌های ندیده شده به دست آورده.
   - در این نقاط، دقت آموزش هم بالا (حدود 0.96 یا 0.97) و **نسبتاً نزدیک به دقت تست** هست. این نشون‌دهنده یک **"Good Fit" (برازش خوب)** هست. مدل الگوها رو به خوبی یاد گرفته و توانایی تعمیم خوبی روی داده‌های جدید داره.
6. **k = 24 تا k = 35 (مقادیر بزرگتر k):**
   - دقت تست دوباره افت می‌کنه (به حدود 0.94 یا 0.92).
   - فاصله بین دقت آموزش و تست دوباره کمی بیشتر میشه.
   - اگر k رو خیلی بزرگ کنیم (مثلاً k برابر با تعداد کل نقاط آموزشی یا نزدیک به اون باشه)، هر نقطه جدید با توجه به بخش بزرگی از کل داده‌های آموزشی پیش‌بینی میشه و این باعث میشه مدل بیش از حد "عمومی" عمل کنه و نتونه جزئیات بین کلاس‌ها رو تشخیص بده. این می‌تونه منجر به **Underfitting (کم‌برازش)** بشه (هرچند در این نمودار، دقت تا این حد پایین نیومده که Underfitting واضحی ببینیم، اما به سمت اون حرکت می‌کنه).

نتیجه‌گیری برای انتخاب k:

با توجه به این نمودار، بهترین عملکرد مدل (بالاترین دقت تست) در k=21، k=19 و k=23 حاصل شده است.

- بهترین انتخاب معمولاً **کوچکترین k** است که بالاترین دقت تست را می‌دهد، زیرا مدل ساده‌تر است و محاسبات کمتری دارد. پس **k=19** انتخاب بسیار خوبی است.
- همچنین، دقت آموزش و تست در این نقاط نزدیک به هم هستند که نشان‌دهنده یک مدل **خوب برازش شده** است.

این نمودار واقعاً ابزار قدرتمندی برای فهم و بهینه‌سازی مدل‌های یادگیری ماشین است. آیا این تحلیل برای شما واضح و قابل فهم بود؟