MachineLearning _ Prerequisites_and_Scikit-learn

☕ برای اینکه انرژی بگیرم، میتونی یه کافی بهم بدی؟

خریدن یه فنجون قهوه

این فایل، خلاصه‌ای ساختاریافته و کاربردی از مفاهیم کلیدی یادگیری ماشین است که با تمرکز ویژه بر نحوه پیاده‌سازی با کتابخانه‌ی Scikit-learn نگاشته شده است.

🔍 هدف این جزوه، فراهم‌کردن یک مرجع سریع و جمع‌وجور در حین کدنویسی است؛ به‌ویژه برای هنگام ساخت مدل‌های یادگیری ماشین در پروژه‌های واقعی.

✅ سرفصل‌های پوشش‌داده‌شده:

- مراحل پیش‌پردازش داده‌ها، اعتبارسنجی، ساخت Pipeline و ارزیابی مدل‌ها
- نکات کلیدی در مواجهه با داده‌های دنیای واقعی (مثل کلاس‌های نامتوازن، Scaling، Encoding و ...)
- دسته‌بندی جامع مدل‌های یادگیری:
  - مدل‌های نظارت‌شده (Classification, Regression)
  - مدل‌های بدون نظارت (Clustering, Dimensionality Reduction)
  - مدل‌های تقویتی و نیمه‌نظارتی
  - مدل‌های ترکیبی
- ارزیابی مدل
- تنظیم هایپرپارامترها و بهینه‌سازی عملکرد مدل

📎 **توجه:**
این فایل در کنار نوت‌بوک‌های تمرینی پروژه محور ماشین لرنیگ در گیت هاب در آدرس زیر
https://github.com/mahdianfe/Machine_Learning/tree/master/ML_Booklets
قرار گرفته؛ اما **مستقل** از آن‌ها می‌باشد و به‌عنوان یک مرجع مفهومی و کدنویسی طراحی شده است.

📘 فصل 1: پیش‌پردازش، بررسی توزیع داده، و آماده‌سازی مدل و ابزارهای Pipeline یادگیری ماشین

✅ بخش 1: پاک‌سازی داده‌ها با Scikit-learn (`SimpleImputer`)

📌 هدف: جایگزینی مقادیر گمشده (NaN) با مقادیر مناسب

استراتژی‌ها	توضیح
`"mean"`	جایگزینی با میانگین
`"median"`	جایگزینی با میانه
`"most_frequent"`	جایگزینی با پرتکرارترین مقدار
`"constant"`	مقدار ثابت مشخص‌شده

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
df_num = df.select_dtypes(include=[np.number])
df_clean = imputer.fit_transform(df_num)
```

✅ بخش 2: بررسی توزیع داده و تشخیص مشکلات آماری

✅ بخش 2.1 چولگی (Skewness)

📌چولگی یعنی **عدم تقارن توزیع داده**:

نوع	توضیح	راه‌حل پیشنهادی
چولگی به راست (/ Right Skew Positive Skew)	بیشتر داده‌ها سمت چپ جمع شده‌اند، دنباله به سمت راست کشیده شده	√ log, √ sqrt, √ Box-Cox برای نرمال‌سازی
چولگی به چپ (/ Left Skew Negative Skew)	بیشتر داده‌ها سمت راست جمع شده‌اند	√ تبدیل عکس معکوس (`x/1`)
چولگی = 0	توزیع نرمال	نیازی به اصلاح نیست

✅ بخش 2.2 داده‌های پرت (Outliers)

نوع	توضیح	راه‌حل پیشنهادی
داده پرت ساده	داده‌هایی که خیلی متفاوت‌اند (مثلاً حقوق = 10 میلیارد)	√ حذف یا اصلاح با IQR یا Z-score
دم سنگین (Heavy Tail)	دنباله‌ی طولانی در چپ یا راست (مثل درآمد)	√ استفاده از مدل‌های مقاوم یا log-transform

نوع	توضیح	راه‌حل پیشنهادی
چندبخشی بودن (Multimodal)	داده دارای چند قله (مانند چند گروه سنی)	√ تقسیم داده به گروه‌ها یا استفاده از کلاسترینگ قبل از مدل‌سازی

✅ بخش 2.3: ساخت ویژگی‌های ترکیبی (Attribute Combination)

📌 بهترین جا: بعد از Cleaning و قبل از Scaling یا انتخاب ویژگی‌ها
عبارت Attribute Combinations یعنی ساخت ویژگی‌های جدید (New Features) از ترکیب ویژگی‌های موجود.

📌 ترکیب ویژگی‌های موجود برای استخراج اطلاعات پنهان

```
df["rooms_per_household"] = df["total_rooms"] / df["households"]
df["bedrooms_per_room"] = df["total_bedrooms"] / df["total_rooms"]
```

📌هدف: کمک به مدل برای یادگیری بهتر الگوهای پنهان در داده

✅ بخش 2.4: تحلیل همبستگی (Correlation Analysis)

📌 برای شناسایی ویژگی‌های تکراری یا بی‌ربط به هدف

```
corr_matrix = df.corr(numeric_only=True)
corr_matrix["target_variable"].sort_values(ascending=False)
```

✅ بخش 2.5: پردازش ویژگی‌های دسته‌ای (Categorical Encoding)

📌 ویژگی‌هایی که مقدارشان متنی (string) است و باید به عدد تبدیل شوند تا مدل یادگیری ماشین بتواند مستقیم از آن‌ها استفاده کند

روش	توضیح	مناسب برای
`LabelEncoder`	هر کلاس عدد یکتا می‌گیرد	فقط برای داده‌های **Ordinal**
`OneHotEncoder`	تبدیل به ستون‌های باینری	برای **Nominal** (بدون ترتیب)
`OrdinalEncoder`	رمزگذاری با ترتیب دلخواه	برای ویژگی‌های با ترتیب قابل‌فهم
`TargetEncoder` (جدا از libs نیاز به)	میانگین target برای هر کلاس	در مدل‌های پیچیده (مثلاً XGBoost)

```
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder()
encoded = encoder.fit_transform(df[["feature_name"]])
```

📌 عبارت **Categorical Features** = ویژگی‌های دسته‌ای
که اشاره به نوع ستون‌ها دارد: مثل `"gender"`, `"color"`, `"city"` که **دسته‌ای / گسسته / غیرعددی** هستند

📌 عبارت **Categorical Encoding** = پردازش این ستون‌ها
⟶ به فرآیند تبدیل ویژگی‌های دسته‌ای به اعداد قابل‌فهم برای مدل می‌گویند (مثل: OneHotEncoding, LabelEncoding)

✅ بخش 2.6: نرمال‌سازی (Scaling)

📌 هدف: برای اینکه بخاطر بزرگ بودن بعضی از اعداد ستونها نسبت به بقیه ستونها مدل تفاوتی قائل نشود همه اعداد را بین صفر و یک می آوریم

🎁 انواع Scaler ها:

Scaler	توضیح	مناسب برای
`StandardScaler`	نرمال با میانگین = 0 و انحراف معیار = 1	الگوریتم‌های مبتنی بر فاصله مثل KNN, SVM
`MinMaxScaler`	مقیاس‌دهی بین 0 تا 1	شبکه‌های عصبی، وقتی ویژگی‌ها مثبت‌اند
`RobustScaler`	مقاوم به داده‌های پرت (استفاده از IQR)	داده‌های با outlier زیاد
`Normalizer`	نرمال‌سازی بردار (نه ستون به ستون)	داده‌های sparse یا برای مدل‌های distance-based مثل KNN

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler

scaler=StandardScaler()
X=scaler.fit_transform(X)
```

✅ بخش 2.7: پردازش همزمان چند نوع ستون با `ColumnTransformer`

📌 وقتی بعضی ستون‌ها عددی‌اند و بعضی دسته‌ای، باید به‌صورت جدا پیش‌پردازش شوند.

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler

num_attribs = ["age", "income"]
cat_attribs = ["gender", "region"]

preprocessor = ColumnTransformer([
    ("num", StandardScaler(), num_attribs),
    ("cat", OneHotEncoder(), cat_attribs)
])
```

درواقع ColumnTransformer یک ابزار پیش‌پردازش است که به شما امکان می‌دهد تبدیلات مختلفی را به ستون‌های مختلف داده‌ها اعمال کنید.

تصور کنید یک جدول داده دارید که شامل ستون‌هایی با انواع مختلف داده‌ها است (مثلاً اعداد، متن، دسته‌بندی). قبل از اینکه بتوانید این داده‌ها را به یک مدل یادگیری ماشین وارد کنید، باید آنها را به فرمتی تبدیل کنید که مدل بتواند آن را درک کند.

و ColumnTransformer به شما امکان می‌دهد تا این کار را به صورت سازماندهی شده و کارآمد انجام دهید. به طور خاص:

```
تبدیلات جداگانه: می‌توانید مشخص کنید که هر ستون باید با چه نوع تبدیلی پردازش
مقیاس‌بندی کنید و StandardScaler شود. برای مثال، می‌توانید ستون‌های عددی را با
ستون‌های متنی را با OneHotEncoder تبدیل کنید.
مدیریت ستون‌های ناهمگن: به جای نوشتن کد پیچیده برای مدیریت هر ستون به صورت
برای تعریف ColumnTransformer جداگانه، می‌توانید از
```

✅ بخش 3: عدم تعادل در کلاس‌ها - مقابله با نامتوازنی در کلاس‌ها (Imbalanced Classes)

📌 زمانی که یک کلاس خیلی بیشتر از کلاس دیگر است (مثلاً کلاس اسپم 5٪ و نرمال 95٪)

📌 وقتی یک کلاس (مثلاً اسپم) خیلی کمتر از بقیه است، مدل به آن توجه نمی‌کند.

🎯 راه‌حل‌ها:

1. Resampling

روش	توضیح
`RandomOverSampler`	زیاد کردن داده کلاس اقلیت
`RandomUnderSampler`	کم کردن داده کلاس غالب

روش	توضیح
`SMOTE`	ساخت داده‌ی جدید مصنوعی از کلاس اقلیت

```
from imblearn.over_sampling import SMOTE

smote = SMOTE()
X_res, y_res = smote.fit_resample(X, y)
```

```
# فرض کنید داده‌ها نامتوازن هستند
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

# جدا کردن داده
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
test_size=0.2)

# اعمال SMOTE روی داده‌های آموزشی
smote = SMOTE()
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

# آموزش مدل
model = LogisticRegression()
model.fit(X_resampled, y_resampled)

# ارزیابی
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))
```

---

3. استفاده از متریک مناسب (به‌جای accuracy)

متریک	مناسب برای
`f1-score`	توازن precision و recall
`ROC-AUC`	مدل‌های دودویی
`Precision/Recall`	مخصوص کلاس‌های نادر

---

2. استفاده از 'class_weight='balanced

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(class_weight='balanced')
```

✅ بخش 4: نمونه‌برداری طبقه‌بندی‌شده (Stratified Sampling) و تقسیم‌بندی داده‌ها (Train-Test Split)

📌 نکته مهم: جلوگیری از سوگیری در نمونه‌برداری

📌 این مورد باید در بخش **Train-Test Split** گنجانده شود.
نمونه‌برداری طبقه‌بندی شده (Stratified Sampling):
ایا این همانجلوگیری از سمپلینگ بایاس یا سوگیری نمونه گیری

اگر داده دارای توزیع نابرابر در گروه‌هایی خاص (مثلاً جنسیت یا ناحیه جغرافیایی) است:

مثلا دو نمونه اقا و خانوم داریم که درصد وجود اقا و خانم 50 50 نیست و یکی 80 است خب این نمونه نتیجه درستی را به ما نمیده
بنابراین می اییم برای اسپلیت کردن تست و ترین از هم میگوییم که 20 درصد از 80 درصد خانم را بردار و 20 درصد از 20 درصد اقا را بردار که سوگیری اتفاق نیفته

گاهی لازمه که:
الف. روی نمونه خودماان طبقه بندی انجام دهیم
و ب. سپس تست و ترین را از هم جدا کنیم
و ج. حذف ستونی که برای طبقه بندی ایجاد کرده بودیم

✓ راه‌حل: **طبقه‌بندی مصنوعی داده قبل از Train-Test**

```
# 1. ایجاد ستون طبقه‌بندی
df["income_cat"] = pd.cut(df["median_house_value"],
                          bins=[0, 100000, 150000, 200000, 300000, np.inf],
                          labels=[1, 2, 3, 4, 5])

# 2. اعمال stratified sampling
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)

for train_idx, test_idx in split.split(df, df["income_cat"]):
    strat_train_set = df.loc[train_idx]
    strat_test_set = df.loc[test_idx]

# 3. حذف ستون کمکی
```

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

📌 تقسیم داده به آموزش و آزمون با حفظ توزیع کلاس‌ها

```
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)
```

📌 عبارت `stratify=y` باعث می‌شود نسبت کلاس‌ها حفظ شود.

تکرارپذیری: اگر random_state را با یک مقدار ثابت تنظیم کنید، هر بار که کد را اجرا می‌کنید، همان تقسیم داده‌ها را دریافت خواهید کرد.

✅ بخش 5: ساخت Pipeline در Scikit-learn

📌 **هدف:** زنجیره‌ای‌کردن مراحل مختلف پیش‌پردازش و مدل‌سازی (مثل Encoding → Scaling → مدل‌سازی) در یک ساختار یکپارچه و قابل استفاده در هر مرحله از پروژه.

🎯 چرا از Pipeline استفاده می‌کنیم؟

مزیت	توضیح
✅ کد تمیز و ساختارمند	تمام مراحل آموزش در یک شیء واحد جمع می‌شوند.
✅ قابل ترکیب با GridSearchCV و Cross-Validation	بدون نیاز به پردازش جداگانه
✅ جلوگیری از نشت داده (Data Leakage)	پیش‌پردازش فقط روی داده‌های آموزش اعمال می‌شود
✅ قابلیت ذخیره‌سازی و بارگذاری آسان	قابل استفاده در محیط‌های عملیاتی و تولیدی

بخش 5.1: روش‌های ساخت Pipeline:

1. `Pipeline` : ساخت مرحله‌به‌مرحله با نام دلخواه

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
```

```
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', LogisticRegression())
])
pipe.fit(X_train, y_train)
```

. `make_pipeline` : نسخه سریع و اتوماتیک‌تر

```
from sklearn.pipeline import make_pipeline

pipe = make_pipeline(StandardScaler(), LogisticRegression())
pipe.fit(X_train, y_train)
```

📌 نیازی به نام‌گذاری دستی مراحل نیست. نام‌ها به‌طور خودکار از کلاس‌ها گرفته می‌شود (مثلاً `standardscaler` ، `logisticregression`).

تفاوت	`Pipeline`	`make_pipeline`
نام‌گذاری مراحل	دستی (`scaler','`) `''clf`	خودکار (`standardscaler','`) `''logisticregression`
مناسب برای	پروژه‌های بزرگ یا قابل کنترل	کدهای سریع و تستی

✅ بخش 5.2: ترکیب Pipeline با GridSearchCV

```
from sklearn.model_selection import GridSearchCV

param_grid = {'clf__C': [0.1, 1, 10]}
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
```

📌 توجه: برای دسترسی به پارامترهای داخلی Pipeline از علامت `__` (دابل آندرلاین) استفاده می‌کنیم.
مثلاً: `'clf__C'` یعنی پارامتر C در مرحله‌ی `'clf'` .

✅ بخش 5.3: ابزارهای مکمل برای ساخت Pipeline

🎁 ابزارهای مکمل:

ابزار	کاربرد
`ColumnTransformer`	اعمال متفاوت روی ستون‌های مختلف
`FunctionTransformer`	تعریف تبدیل دلخواه سفارشی
`make_pipeline()`	نسخه کوتاه‌تر ساخت Pipeline

1. تکنیک `ColumnTransformer` : اعمال پیش‌پردازش‌های متفاوت روی ستون‌های مختلف

📌نکته: `ColumnTransformer` به ما اجازه می‌دهد **برای هر ستون یا گروهی از ستون‌ها، پیش‌پردازش جداگانه** تعریف کنیم.

🎯 چرا مهم است؟

در دنیای واقعی، داده‌ها مخلوط‌اند:

- بعضی عددی‌اند → `StandardScaler`
- بعضی طبقه‌ای‌اند → `OneHotEncoder`
- بعضی شاید نیاز به هیچ تبدیلی ندارند

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

# ستون‌های عددی و طبقه‌ای
num_features = ['age', 'income']
cat_features = ['gender', 'city']

# مرحله 1: ساخت ColumnTransformer
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), num_features),
    ('cat', OneHotEncoder(), cat_features)
])

# مرحله 2: ترکیب در Pipeline
pipe = Pipeline([
    ('pre', preprocessor),
    ('clf', LogisticRegression())
])

pipe.fit(X_train, y_train)
```

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), ['age', 'income']),           # روی ستون‌های عددی
        ('cat', OneHotEncoder(), ['gender', 'city'])            # روی ستون‌های طبقه‌ای
    ]
)
```

📌 بسیار مناسب برای داده‌های ترکیبی (عددی + متنی).

---

2. استفاده از `FunctionTransformer` در Pipeline

📌 این ابزار برای مواقعی‌ست که نیاز داریم **یک تابع دلخواه (custom)** روی داده‌ها اجرا کنیم.
📎 کاربرد اصلی `FunctionTransformer` :
اضافه کردن عملیات **سفارشی ریاضی یا منطقی** داخل
مثلاً: اعمال log روی ویژگی‌ها، یا تبدیل درصد به عدد بین ۰ تا ۱.

مثال ساده: اعمال log روی کل داده‌ها

```
from sklearn.preprocessing import FunctionTransformer
import numpy as np

log_transformer = FunctionTransformer(np.log1p)

X_log = log_transformer.fit_transform(X)
```

✅ مثال در یک Pipeline:

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import FunctionTransformer, StandardScaler
from sklearn.linear_model import Ridge
import numpy as np

pipe = make_pipeline(
    FunctionTransformer(np.log1p),  # log(x+1)
    StandardScaler(),
    Ridge()
)
```

```
pipe.fit(X_train, y_train)
```

---

3. تبدیل‌های سفارشی روی داده : `FunctionTransformer`

```
from sklearn.preprocessing import FunctionTransformer
import numpy as np

log_transformer = FunctionTransformer(np.log1p)

pipe = Pipeline([
    ('log', log_transformer),
    ('clf', LogisticRegression())
])
```

📌 مناسب برای اعمال تبدیل‌های خاص مثل log، sqrt، یا ساخت توابع شخصی.

✅ بخش 5.4: کوتاه برای دسترسی به پارامترهای داخل Pipeline با `__` (دابل آندرلاین)

فرض کن یک `Pipeline` ساده داریم:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', LogisticRegression())
])
```

اینجا مرحله دوم پایپ‌لاین `clf` است که یک **مدل** **LogisticRegression** است.
این مدل، در داخل خودش یک **ابرپارامتر به نام** `C` دارد که میزان **رگولاریزاسیون** را کنترل می‌کند.

اکنون می‌خواهیم مقدار `C` (که یک ابرپارامتر در `LogisticRegression` است) را در `GridSearchCV` تنظیم کنیم:
✅ منظور از `'clf__C'` یعنی:

به مرحله `'clf'` در Pipeline برو (اشاره به مرحله دوم پایپ‌لاین (که `LogisticRegression` است)) و پارامتر `C` را تنظیم کن.

```
param_grid = {
    'clf__C': [0.01, 0.1, 1, 10]  # دسترسی به پارامتر C در مرحله 'clf'
}

grid = GridSearchCV(pipe, param_grid, cv=3)
grid.fit(X_train, y_train)
```

📌 **پس تو داری به GridSearch می‌گی:** مقدار `C` مدل `clf` را بین این گزینه‌ها امتحان کن.
(مقدار `C` خودش داخل `LogisticRegression()` هست ولی با `GridSearchCV` قراره تست و تنظیم بشه)

یا مثال دیگر: اگر در مرحله `scaler` پارامتری مثل `with_mean=False` را بخواهی تغییر دهی:

```
param_grid = {'scaler__with_mean': [True, False]}
```

✅ بخش 5.5: چرا Pipeline جلوی نشت اطلاعات (Data Leakage) را می‌گیرد؟

🔶 تعریف نشت اطلاعات:

نشت اطلاعات یعنی بخشی از داده‌های تست **ناخواسته در آموزش** مدل یا مراحل پیش‌پردازش استفاده شوند.
و این اتفاق باعث می‌شود که مدل اطلاعاتی از داده‌های تست داشته باشد و در نتیجه، **عملکرد واقعی مدل روی داده‌های جدید بیش‌برآورد شود (Overestimated)** و **مدل دچار بیش‌برازش (Overfitting)** گردد.

📌 مشکل وقتی پیش می‌آید که:

مثلاً قبل از تقسیم داده به train/test، کل داده را Standardize کنیم: که باعث نشت اطلاعات میشه

یا

❌ اگر قبل از تقسیم داده‌ها، پیش‌پردازش انجام دهی (مثل `fit_transform` روی کل `X`)، اطلاعات تست وارد فرآیند آموزش می‌شود.

🔶 مثال نشت:

```
# ❌ اشتباه: اعمال Scaler روی کل داده
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)  # از نشت داده‌ها به train به test
```

یعنی:
تو فقط باید روی `X_train` ، عمل `()fit` انجام بدی. یعنی:
✓ راه حل درست:

```
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

- عمل `fit` فقط روی `X_train` باید انجام شود
- عمل `transform` روی هر دو انجام شده (اما با پارامترهایی که از `X_train` یاد گرفته شده)

📌 نکته تکمیلی:
عمل `()fit` فقط میانگین و انحراف معیار را از `X_train` محاسبه می‌کند.
سپس `()transform` همین مقادیر را برای نرمال‌سازی `X_test` به کار می‌برد، بدون اینکه آن را دیده باشد.

> ✓ اما روش دیگر که نشت اتفاق نمی افتد:
> از Pipeline استفاده کنی:

```
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('clf', LogisticRegression())
])
```

در این حالت:

- عملیات `StandardScaler` فقط روی `X_train` **fit** می‌شود.
- سپس همان Scaler روی `X_test` **transform** می‌کند، بدون دیدن آن.
- 📌 پس هیچ اطلاعی از `X_test` در مرحله آموزش (fit) به مدل نرسیده ← **جلوگیری از نشت**

چرا وقتی از `Pipeline` و `train_test_split` استفاده می‌کنیم، `StandardScaler` فقط روی `X_train` `()fit` میشه؟
✓ جواب: چون **Scikit-learn به‌صورت خودکار تضمین می‌کنه** که هر چیزی داخل `Pipeline` هست، فقط روی `X_train` آموزش ببینه (`()fit` بشه) و بعد فقط روی `X_test` اجرا بشه (`()transform` یا `()predict` بشه).

و سپس:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y)
pipe.fit(X_train, y_train)      # فقط روی train fit می‌شود
```

```
pipe.score(X_test, y_test)      # تست روی داده‌های نادیده
```

📌 نتیجه: Scaler فقط روی `X_train` آموزش می‌بیند و سپس روی `X_test` فقط transform انجام می‌دهد.

📌 نکته نهایی:

1. هر مرحله در `Pipeline` باید یکی از این دو باشد:

نوع مرحله	متد لازم	مثال
مرحله پیش‌پردازش	`fit` , `transform`	`StandardScaler` , `OneHotEncoder` , `PCA`
مرحله نهایی (مدل)	`fit` , `predict`	`LogisticRegression` , `RandomForest`

فقط مرحله‌ی آخر می‌تواند `predict` داشته باشد، چون خروجی Pipeline باید **قابل پیش‌بینی** باشد.

2. ساخت Pipeline نه‌تنها فرآیند آموزش را ساده می‌کند، بلکه امن‌تر، قابل تنظیم‌تر، و قابل بازتولید است.

3. ترکیب آن با GridSearchCV یا Cross-Validation باعث می‌شود مدل نهایی دقیق‌تر و مقاوم‌تر به overfitting باشد.

✅ بخش 6: اعتبارسنجی مدل (Validation)

📌 هدف: ارزیابی عملکرد مدل روی داده‌هایی که در آموزش استفاده نشده‌اند.

🔸 در آموزش حرفه‌ای، معمولاً سه قسمت داریم:

- **Train:** برای آموزش
- **Validation:** برای تنظیم مدل (با Hold-out یا Cross-validation)
- **Test:** برای ارزیابی نهایی (فقط یک‌بار)

🎁 انواع روش‌های Validation:

روش	توضیح	ایمپورت لازم	کد نمونه
Hold-out Validation	تقسیم داده به train/test - تست یا ولیدیشن	`from sklearn.model_selection import train_test_split`	`train_test_split(X, y)`
K-Fold Cross Validation	داده به K بخش تقسیم می‌شود	`from sklearn.model_selection import KFold`	`KFold(n_splits=5)`
Stratified K-Fold	مشابه K-Fold با حفظ نسبت کلاس‌ها	`from sklearn.model_selection import StratifiedKFold`	`StratifiedKFold(n_splits=5)`
Leave-One-Out (LOO)	هر بار یک داده برای تست	`from sklearn.model_selection import LeaveOneOut`	`LeaveOneOut()`
ShuffleSplit	چند تقسیم تصادفی و مستقل	`from sklearn.model_selection import ShuffleSplit`	`ShuffleSplit(n_splits=5, test_size=0.2)`

```python
from sklearn.model_selection import cross_val_score, StratifiedKFold

kfold = StratifiedKFold(n_splits=5)
scores = cross_val_score(model, X, y, cv=kfold, scoring='accuracy')
print(scores.mean())
```

Validation (اعتبارسنجی)
├── Hold-Out Validation
│ └── train_test_split
│
└── Cross-Validation (اعتبارسنجی متقاطع)
├── K-Fold
├── Stratified K-Fold
├── Leave-One-Out (LOO)
├── ShuffleSplit
└── Group K-Fold / Group ShuffleSplit (برای داده‌های گروه‌بندی‌شده)

✅ بخش 6.1: Cross-validation در Scikit-learn

📌 روش رایج برای ارزیابی مدل در چند مرحله

🎁 انواع روش‌ها:

روش	توضیح	مناسب برای
Hold-Out	Train/Validation split ساده	ساده‌ترین
K-Fold	داده به k بخش تقسیم می‌شود	رایج‌ترین
Stratified K-Fold	با حفظ نسبت کلاس‌ها	داده‌های نامتوازن
Leave-One-Out	هر بار فقط یک نمونه test است	داده‌های بسیار کم
ShuffleSplit	نمونه‌گیری تصادفی چندباره	منعطف

```
from sklearn.model_selection import cross_val_score, StratifiedKFold

kfold = StratifiedKFold(n_splits=5)
scores = cross_val_score(model, X, y, cv=kfold, scoring='accuracy')
```

```
from sklearn.model_selection import cross_val_score
#%%
dt_mae = -cross_val_score(dt_regression_pipeline,
                          train_features,
                          train_target,
                          scoring="neg_mean_absolute_error",
                          cv=5)
# اولین عضوش مدل را میگذاریم
# دومین عضوش ایکس و بعدش وای را وارد میکنیم
# scoring یعنی به تعداد باری که تعیین cv یعنی با چه معیاری کار کن. # سی وی
میکنی که داده را مثلا 5 قسمت مساوی میکند و یه یک پنجم را به عنوان ولیدیشن
برمیداره
```

✅ بخش 7: کلون کردن مدل‌ها (Cloning Models)

📌 چرا لازم است؟
برای استفاده مجدد از یک مدل بدون تاثیر تغییرات قبلی.

🔍 کاربرد:

- جلوگیری از خراب شدن مدل اصلی
- ساخت مدل جدید در loop ها
- استفاده در Pipeline یا ensemble

🎁 ابزار:

```
from sklearn.base import clone

new_model = clone(existing_model)
```

🎯 کاربرد سریع و واضح:

فرض کن یه مدل `LogisticRegression(C=1.0)` داری و می‌خوای **همون مدل با همون تنظیمات** ولی بدون اثرات یادگیری قبلی، یه جای دیگه استفاده کنی:

```
from sklearn.linear_model import LogisticRegression
from sklearn.base import clone

model = LogisticRegression(C=1.0)
model.fit(X_train, y_train)  # مدل آموزش می‌بیند

# حالا می‌خوای یه مدل جدید با همون تنظیمات بسازی:
new_model = clone(model)
new_model.fit(X_new, y_new)  # کاملاً از صفر آموزش می‌بیند
```

📘 فصل 2: مدل‌های یادگیری ماشین

🔷 دسته‌بندی کلی مدل‌ها:

1. **مدل‌های نظارت‌شده (Supervised Learning)**
   ⟵ داده‌های دارای برچسب (Label) ⟶ مدل پیش‌بینی‌گر
   📌 شامل: Classification و Regression
2. **مدل‌های غیر نظارت‌شده (Unsupervised Learning)**
   ⟵ داده‌های بدون برچسب ⟶ مدل‌های خوشه‌بندی یا کاهش ابعاد
3. **مدل‌های یادگیری تقویتی (Reinforcement Learning)**
   ⟵ عامل (Agent) با محیط تعامل می‌کند و پاداش می‌گیرد
4. **مدل‌های یادگیری نیمه‌نظارتی (Semi-Supervised Learning)**
   ⟵این مدل‌ها از ترکیب داده‌های برچسب‌دار و بدون برچسب برای آموزش استفاده می‌کنند که در بسیاری از سناریوهای واقعی کاربردی است.
5. **مدل‌های یادگیری تقویتی و ترکیبی (Hybrid Learning Models)**
   ⟵ این مدل‌ها رویکردهای مختلف یادگیری ماشین (مانند نظارت‌شده، غیرنظارت‌شده، و تقویتی) را با هم ترکیب می‌کنند تا

بهترین استفاده را از مزایای هر کدام ببرند.

📘 فصل 3: مدل‌های نظارت‌شده (Supervised Learning)

✅ بخش 1: دسته‌بندی کلی مدل‌ها

یادگیری با نظارت به دو گروه اصلی تقسیم می‌شود:

نوع مدل	تعریف کوتاه	هدف اصلی	خروجی y
Classification	مدل‌بندی دسته‌ها	پیش‌بینی دسته/کلاس	گسسته (Discrete)
Regression	مدل‌بندی مقادیر	پیش‌بینی مقدار عددی	پیوسته (Continuous)

به عبارت دیگر:

ویژگی	توضیح
نوع داده‌ها	دارای **ورودی (X)** و **خروجی (y)** یا برچسب پاسخ هستند.
هدف اصلی	مدل یاد بگیرد چگونه از X، مقدار y را پیش‌بینی کند.
دو نوع کلی	- **Classification**: پیش‌بینی دسته/برچسب (کلاس) - **Regression**: پیش‌بینی مقدار عددی پیوسته

✅ بخش 2: مدل‌های طبقه‌بندی (Classification Models) پرکاربرد در Scikit-learn

مدل	توضیح کوتاه	کد نمونه Scikit-learn
Logistic Regression	مدل خطی برای پیش‌بینی برچسب دودویی ساده و تفسیرپذیر، سریع، داده‌های خطی	`from sklearn.linear_model import LogisticRegression`
K-Nearest Neighbors (KNN)	بر اساس همسایه‌های نزدیک ساده، داده کم، بدون نیاز به آموزش، حساس به مقیاس	`from sklearn.neighbors import KNeighborsClassifier`

مدل	توضیح کوتاه	کد نمونه Scikit-learn
Naive Bayes	مدل احتمال شرطی با فرض استقلال متنی، مستقل بودن ویژگی‌ها، سریع و ساده	`from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB`
Decision Tree	درخت تصمیم‌گیری منطقی تفسیرپذیر، سریع، تمایل به overfitting	`from sklearn.tree import DecisionTreeClassifier`
Random Forest	مجموعه‌ای از درخت‌های تصمیم دقت بالا، مقاوم، مناسب برای داده‌های پیچیده	`from sklearn.ensemble import RandomForestClassifier`
SVM (Support Vector)	یافتن مرز تصمیم بهینه داده‌های با ابعاد بالا، دسته‌بندی دقیق	`from sklearn.svm import SVC`
Gradient Boosting	مدل تقویتی تدریجی دقت بالا، مدل‌های ضعیف را تقویت می‌کند	`from sklearn.ensemble import GradientBoostingClassifier`
MLP / Neural Net	شبکه عصبی ساده	`from sklearn.neural_network import MLPClassifier`
XGBoost	بسیار قدرتمند، سرعت و دقت بالا	`from xgboost import XGBClassifier` (کتابخانه جدا)
SGD Classifier	داده‌های حجیم، بهینه‌سازی سریع با گرادیان تصادفی	`from sklearn.linear_model import SGDClassifier`

✓ بخش 3: مدل‌های رگرسیون (Regression Models) پرکاربرد در Scikit-learn

مدل	توضیح کوتاه	کد نمونه Scikit-learn
Linear Regression	مدل خطی برای مقدار پیوسته ساده، خطی، تفسیرپذیر	`from sklearn.linear_model import LinearRegression`

مدل	توضیح کوتاه	کد نمونه Scikit-learn
KNN Regressor	مشابه طبقه‌بندی اما برای مقدار عددی داده کم، ساده، بدون نیاز به آموزش	`KNeighborsRegressor`
Decision Tree Regressor	درخت برای رگرسیون الگوهای غیرخطی، تفسیرپذیر	`DecisionTreeRegressor`
Random Forest Regressor	مدل ترکیبی درخت‌ها الگوهای پیچیده، پایدار در برابر overfitting	`RandomForestRegressor`
SVR	رگرسیون با SVM داده‌های با ابعاد بالا، پیچیده	`SVR`
Gradient Boosting Regressor	مدل قوی با ترکیب تدریجی دقت بالا، برای داده‌های پیچیده	`GradientBoostingRegressor`
MLP Regressor	شبکه عصبی برای رگرسیون	`MLPRegressor`
SGD Regressor	مناسب داده‌های حجیم، سرعت بالا	`from sklearn.linear_model import SGDRegressor`
Ridge / Lasso	رگرسیون خطی با regularization	`from sklearn.linear_model import Ridge, Lasso`
Ridge Regression	جلوگیری از overfitting، تنظیم L2	`from sklearn.linear_model import Ridge`
Lasso Regression	کاهش ویژگی‌ها، تنظیم L1	`from sklearn.linear_model import Lasso`

✅ بخش 4: انتخاب مدل مناسب

- با توجه به:
  - نوع داده (کمی یا کیفی)
  - حجم داده
  - نیاز به تفسیر یا دقت بالا
  - سرعت اجرا و مقیاس‌پذیری

معیار انتخاب	توضیح کاربردی
نوع خروجی (y)	- اگر عدد گسسته: Classification - اگر عدد پیوسته: Regression
نوع ویژگی‌ها (X)	اگر متنی/دسته‌ای زیاد: استفاده از مدل‌هایی با encoder یا شبکه عصبی مناسب
حجم داده‌ها	- حجم کم: مدل‌های ساده مثل Logistic، Tree - حجم زیاد: SGD، RandomForest، SVM
تفسیرپذیری مدل	نه SVM مدل‌های تفسیرپذیر هستند، اما شبکه‌های عصبی یا Tree و Logistic
دقت مدل موردنیاز	برای دقت بالا مناسب‌تر هستند RandomForest یا Boosting
سرعت آموزش/پیش‌بینی	مدل‌های ساده مثل Linear Regression، Logistic، Naive Bayes سریع‌تر هستند

وضعیت داده	مدل پیشنهادی
داده ساده، تفسیرپذیر	Logistic (Classification) / Linear (Regression)
داده پیچیده و غیرخطی	Tree, RandomForest, Boosting
حجم داده زیاد	SGD, Naive Bayes, XGBoost
سرعت مهم باشد	Naive Bayes, Logistic, Linear
مدل با قابلیت احتمال‌دهی	Logistic, Naive Bayes, SGD
ویژگی‌ها زیاد و sparse	SVM, SGD

✓ بخش 5: ابزارهای مشترک برای هر دو دسته

- متد `.score()`, `.predict()`, `.fit()`
- متد `.predict_proba()` برای مدل‌های احتمالاتی
- استفاده در pipeline
- ترکیب با `cross_val_score`, `GridSearchCV` و غیره

ابزار/متد	کاربرد
`.fit(X, y)`	آموزش مدل با داده‌های ورودی و خروجی
`.predict(X)`	پیش‌بینی خروجی برای داده جدید
`.score(X, y)`	ارزیابی سریع عملکرد مدل (معمولاً دقت برای classification یا R^2 برای regression)
`.predict_proba(X)`	احتمال تعلق به کلاس‌های مختلف (در مدل‌های احتمالاتی مثل Logistic یا Naive Bayes)
Pipeline	زنجیره کردن مراحل پیش‌پردازش و مدل‌سازی با `Pipeline`
GridSearchCV	جستجوی ترکیب بهینه‌های هایپرپارامترها با استفاده از ولیدیشن متقابل

ابزار/متد	کاربرد
cross_val_score	ارزیابی عملکرد مدل با اعتبارسنجی متقاطع (K-fold و غیره)

✅ بخش 6: مثال ساده پیاده‌سازی مدل

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

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

model = LogisticRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred))
```

✅ 📌 بخش 7: توضیحات تفصیلی هر مدل

✅ 1. Naive Bayes (بیز ساده)

📌 ایده: مدل آماری که با فرض استقلال ویژگی‌ها، احتمال تعلق نمونه به کلاس‌ها را محاسبه می‌کند.

🎁 کلاس‌های مهم:

کلاس	کاربرد	توضیح
`GaussianNB`	طبقه‌بندی	ویژگی‌ها پیوسته باشند (مثل قد، وزن) – توزیع نرمال
`MultinomialNB`	طبقه‌بندی	داده‌های شمارشی (مثل تعداد کلمات در متن)
`BernoulliNB`	طبقه‌بندی	داده‌های صفر و یک (باینری)

```
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB

model = GaussianNB()
model.fit(X_train, y_train)
```

✅ 2. K-Nearest Neighbors (KNN)

📌 ایده: پیش‌بینی بر اساس نزدیک‌ترین همسایه‌ها

🎁 کلاس‌های مهم:

کلاس	کاربرد	توضیح
`KNeighborsClassifier`	طبقه‌بندی	با استفاده از K همسایه نزدیک
`KNeighborsRegressor`	رگرسیون	با میانگین خروجی K همسایه
`RadiusNeighborsClassifier`	طبقه‌بندی	همسایه‌هایی در شعاع مشخص (نه K مشخص)
`RadiusNeighborsRegressor`	رگرسیون	میانگین خروجی همسایه‌های شعاع مشخص
`NearestNeighbors`	همسایه‌یابی	پیدا کردن نزدیک‌ترین داده‌ها – بدون برچسب (برای خوشه‌بندی و جستجو)

```
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.neighbors import RadiusNeighborsClassifier,
RadiusNeighborsRegressor
from sklearn.neighbors import NearestNeighbors
```

✅ 3. Decision Tree (درخت تصمیم)

📌 ایده: درختی از تصمیمات با تقسیم ویژگی‌ها

🎁 کلاس‌های مهم:

کلاس	کاربرد	توضیح
`DecisionTreeClassifier`	طبقه‌بندی	هر گره تقسیم بر اساس بیشترین اطلاعات
`DecisionTreeRegressor`	رگرسیون	تقسیم بر اساس کاهش MSE یا MAE

```
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor

model = DecisionTreeClassifier()
model.fit(X_train, y_train)
```

✅ 4. Random Forest (جنگل تصادفی)

📌 ایده: ترکیب چند درخت برای کاهش overfitting

🎁 کلاس‌ها:

کلاس	کاربرد	توضیح
`RandomForestClassifier`	طبقه‌بندی	میانگین رأی‌های چند درخت
`RandomForestRegressor`	رگرسیون	میانگین خروجی درخت‌ها

```
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
```

✅ 5. Support Vector Machine (SVM)

📌 ایده: مرز تصمیم بهینه بین کلاس‌ها

🎁 کلاس‌ها:

کلاس	کاربرد	توضیح
`SVC`	طبقه‌بندی	SVM با kernel قابل انتخاب (linear, rbf, poly...)
`SVR`	رگرسیون	نسخه رگرسیون SVM
`LinearSVC`	طبقه‌بندی	نسخه سریع و ساده‌ی خطی
`LinearSVR`	رگرسیون	نسخه رگرسیون سریع خطی

```
from sklearn.svm import SVC, SVR, LinearSVC, LinearSVR
```

✅ 6. Logistic Regression (رگرسیون لجستیک)

📌 ایده: مدل آماری برای پیش‌بینی احتمال تعلق به کلاس

🎁 کلاس:

کلاس	کاربرد	توضیح
`LogisticRegression`	طبقه‌بندی	مدل ساده و سریع – پایه بسیاری از مدل‌های پیچیده‌تر

```
from sklearn.linear_model import LogisticRegression
```

✅ 7. Multi-layer Perceptron (MLP)

📌 ایده: شبکه عصبی چند لایه با آموزش backpropagation

🎁 کلاس‌ها:

کلاس	کاربرد	توضیح
`MLPClassifier`	طبقه‌بندی	شبکه با لایه‌های مخفی قابل تنظیم
`MLPRegressor`	رگرسیون	مشابه بالا برای مقادیر پیوسته

```
from sklearn.neural_network import MLPClassifier, MLPRegressor
```

📘 فصل 4: یادگیری بدون نظارت (Unsupervised Learning)

> در این فصل به مدل‌هایی می‌پردازیم که بدون داشتن برچسب (Label) روی داده‌ها، ساختار پنهان یا گروه‌بندی آن‌ها را یاد می‌گیرند.

📌 هدف: کشف الگوها در داده‌هایی که برچسب (label) ندارند.

شامل:

✅ 1. خوشه‌بندی (Clustering):

📌 هدف: **گروه‌بندی نقاط داده مشابه** در دسته‌های (خوشه‌های) مختلف. مثال: K-Means، Hierarchical Clustering، DBSCAN. ویژگی‌ها: معمولاً خوشه‌های واضح و جداگانه ایجاد می‌کنند.

✅ 2. کاهش ابعاد (Dimensionality Reduction):

📌 هدف: **کاهش تعداد ویژگی‌ها (ابعاد)** در داده‌ها، در حالی که اطلاعات مهم حفظ شود. مثال: UMAP، t-SNE، PCA. ویژگی‌ها: داده‌ها را به فضای کم‌بعدتر نگاشت می‌کنند، اغلب برای تجسم یا بهبود عملکرد مدل‌های دیگر.

✅ 3. تشخیص ناهنجاری (Anomaly Detection / Outlier Detection):

📌 هدف: **شناسایی نقاط داده‌ای که به طور قابل توجهی از الگوی اصلی داده‌ها منحرف هستند.** این نقاط "ناهنجار" یا "دورافتاده" نامیده می‌شوند. مثال: Isolation Forest، One-Class SVM. ویژگی‌ها: به جای گروه‌بندی همه داده‌ها، تمرکز بر یافتن "موارد عجیب" است.

✅ 4. مدل‌های آماری/احتمالاتی بدون نظارت (مانند Gaussian Mixture Model):

این دسته هم یک نوع خوشه‌بندی است، اما با رویکردی متفاوت و مبتنی بر احتمال.

📌 **هدف:** خوشه‌بندی، اما با این فرض که نقاط داده از **توزیع‌های گوسی (Gaussian distributions) مختلفی (یعنی خوشه‌های مختلف)** تولید شده‌اند.

📌 **تفاوت با K-Means:**
در واقع **K-Means:** هر نقطه را به **نزدیک‌ترین مرکز خوشه** اختصاص می‌دهد (خوشه‌های سخت و دایره‌ای).
اما **GMM:** برای هر نقطه، **احتمال تعلق آن به هر خوشه** را محاسبه می‌کند (خوشه‌های نرم و بیضوی). به عبارت دیگر، یک نقطه می‌تواند با درصدهای مختلف به چندین خوشه تعلق داشته باشد (خوشه‌بندی "نرم" یا "احتمالاتی").
در GMM می‌تواند خوشه‌هایی با اشکال بیضوی و اندازه‌های متفاوت را به خوبی مدل کند، در حالی که K-Means بیشتر برای خوشه‌های کروی و هم‌اندازه مناسب است.

🎁 **ابزار:** `GaussianMixture` در `sklearn.mixture`

✅ بخش 1: خوشه‌بندی (Clustering)

📌 هدف: گروه‌بندی داده‌های مشابه بدون برچسب.

🎁 الگوریتم‌های scikit-learn برای خوشه‌بندی:

کلاس	کاربرد	توضیح کوتاه
`KMeans`	خوشه‌بندی	داده‌ها را به K خوشه تقسیم می‌کند (روش پایه)
`MiniBatchKMeans`	خوشه‌بندی سریع	نسخه سریع‌تر KMeans برای دیتاست بزرگ
`AgglomerativeClustering`	خوشه‌بندی سلسله‌مراتبی	از پایین به بالا، ادغام خوشه‌ها
`DBSCAN`	خوشه‌بندی چگالی‌محور	خوشه‌بندی بر اساس تراکم نقاط
`MeanShift`	خوشه‌بندی چگالی‌محور	نیازی به تعیین تعداد خوشه‌ها ندارد
`SpectralClustering`	خوشه‌بندی طیفی	با تحلیل طیفی گراف مشابهت بین داده‌ها
`AffinityPropagation`	خوشه‌بندی با پیام‌رسانی	خوشه‌ها را بدون تعیین K مشخص می‌کند
`Birch`	خوشه‌بندی مقیاس‌پذیر	مناسب برای دیتاست‌های بسیار بزرگ
`OPTICS`	شبیه به DBSCAN	بهتر در کشف خوشه‌های با تراکم‌های متفاوت

✓ مثال از KMeans:

```
from sklearn.cluster import KMeans

model = KMeans(n_clusters=3)
model.fit(X)
labels = model.labels_
```

✅ بخش 1.1: انتخاب تعداد خوشه مناسب

📌برای انتخاب `n_clusters` مناسب می‌توان از:

- **روش Elbow** (زانو):
روش Elbow بر اساس مفهوم **درون‌خوشه‌ای جمع مربعات (Within-Cluster Sum of Squares - WCSS)** کار می‌کند. WCSS مجموع مربعات فاصله نقاط داده از مرکز خوشه خودشان است. به عبارت دیگر، هرچه WCSS کمتر باشد، نقاط درون خوشه‌ها به مرکز خوشه خود نزدیک‌تر هستند و خوشه‌ها فشرده‌ترند.
- روش Elbow بیشتر یک "معیار بصری" یا "روش اکتشافی" است تا یک شاخص کمی، چون شما در آن به دنبال یک الگوی بصری (نقطه زانو) هستید.
- **شاخص سیلوئت (Silhouette Score)**:
شاخص سیلوئت (Silhouette Score) معیاری برای سنجش کیفیت خوشه‌بندی است که نشان می‌دهد هر نقطه داده تا چه حد به خوشه خود شباهت دارد و از خوشه‌های دیگر متمایز است. این شاخص یک روش اندازه‌گیری عددی (کمی) برای سنجش کیفیت خوشه‌بندی است.. این شاخص برای هر نقطه داده محاسبه می‌شود و سپس میانگین آن برای کل خوشه‌بندی گزارش می‌گردد.

✓ برای **داده‌های بسیار بزرگ (Big Data)**، **روش Elbow (بر اساس WCSS/Inertia)** به دلیل پیچیدگی محاسباتی کمتر، معمولاً **ترجیح داده می‌شود**.

- اگرچه ممکن است کمی ذهنی باشد(به معنای فرایند تفسیر نمودار توسط انسان است، نه محاسبات)، اما اجرای آن برای داده‌های حجیم سریع‌تر است و می‌توانید یک تخمین اولیه از K بهینه را به دست آورید.
- شاخص سیلوئت، به دلیل نیاز به محاسبه فواصل جفتی، می‌تواند برای داده‌های بزرگ **بسیار کند** و حتی غیرعملی باشد

✓ اما **پیشنهاد عملی:**
برای داده‌های بزرگ، می‌توانید از ترکیب این دو روش به این صورت استفاده کنید:

1. **با روش Elbow شروع کنید:** یک بازه منطقی از K را با استفاده از نمودار Elbow پیدا کنید. این کار به سرعت شما را به یک دامنه کوچکتر از Kهای احتمالی محدود می‌کند.
2. **نمونه‌برداری (Sampling):** اگر داده‌ها واقعاً بسیار بزرگ هستند و حتی Elbow هم کند است، می‌توانید **بخشی از داده‌ها (Sample)** را انتخاب کرده و هر دو روش (Elbow و Silhouette) را روی این نمونه اجرا کنید. سپس K بهینه را که از نمونه به دست آمده، روی کل داده‌ها اعمال کنید. البته این روش همیشه بهترین نتیجه را تضمین نمی‌کند، اما می‌تواند یک رویکرد عملی باشد.

✓ مثال از silhouette_score:

```
from sklearn.metrics import silhouette_score

score = silhouette_score(X, labels)
print(score)
```

با توجه به خروجی score را به عنوان مقداری برای n_clusters در نظر میگیریم

✅ بخش 2: کاهش ابعاد (Dimensionality Reduction)

📌 هدف: کاهش تعداد ویژگی‌ها بدون از دست رفتن اطلاعات مهم

- ساده‌تر شدن مدل
- افزایش سرعت پردازش
- تجسم بهتر داده‌ها

📌 مناسب برای پیش‌پردازش، تجسم داده، کاهش نویز

🎁 کلاس‌های مهم:

کلاس	کاربرد	توضیح کوتاه	نحوه ایمپورت (مثال)
`PCA`	کاهش بعد خطی	تحلیل مؤلفه‌های اصلی – بیشترین واریانس = فشرده‌سازی داده با حفظ بیشترین واریانس	`from sklearn.decomposition import PCA`
`TruncatedSVD`	مشابه PCA	مخصوص ماتریس‌های sparse (مثل داده‌های متنی) یا زمانی که داده‌ها مرکزگذاری نشده‌اند.	`from sklearn.decomposition import TruncatedSVD`
`NMF`	فاکتورگیری ماتریس	همه مقادیر مثبت – مناسب متن و تصویر (برای یافتن مؤلفه‌های مثبت و قابل تفسیر)	`from sklearn.decomposition import NMF`
`KernelPCA`	کاهش بعد غیرخطی	از Kernel Trick برای مدل‌سازی روابط پیچیده‌تر و غیرخطی استفاده می‌کند.	`from sklearn.decomposition import KernelPCA`
`Isomap`	حفظ فواصل ژئودزیک	برای داده‌های با ساختار منیفولد (manifold) یا غیرخطی – حفظ فواصل واقعی در فضای با ابعاد بالا	`from sklearn.manifold import Isomap`
`TSNE` (یا `t-SNE`)	تجسم 2D یا 3D	مخصوص نمایش بصری داده‌های با ابعاد بالا در فضای کمتر، با حفظ خوشه‌های محلی	`from sklearn.manifold import TSNE`

کلاس	کاربرد	توضیح کوتاه	نحوه ایمپورت (مثال)
`UMAP`	کاهش بعد سریع	کاهش بعد سریع و مؤثر، اغلب بهتر از t-SNE در حفظ ساختار کلی و سرعت.	`import umap` (نیاز به نصب جداگانه `pip install umap-learn`)
`FactorAnalysis`	مشابه PCA	یک مدل احتمالاتی که به دنبال عوامل پنهان (latent factors) در داده‌ها می‌گردد.	`from sklearn.decomposition import FactorAnalysis`

✓ مثال از PCA:

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X)
```

✓ بخش 3: تشخیص ناهنجاری (Anomaly Detection)

📌 هدف: تشخیص نقاط غیرعادی (outliers) در داده‌های بدون برچسب
📌 کاربردها: تشخیص تقلب، ناهنجاری شبکه، سنسور خراب و...

🎁 کلاس‌ها:

کلاس	کاربرد	توضیح کوتاه
`IsolationForest`	تشخیص ناهنجاری	با ساختن درخت‌های تصادفی، نقاط جداافتاده را شناسایی می‌کند
`OneClassSVM`	ناهنجاری	نسخه خاص SVM برای داده‌های بدون برچسب
`LocalOutlierFactor`	ناهنجاری	مقایسه چگالی هر نقطه با همسایه‌ها
`EllipticEnvelope`	ناهنجاری	فرض داده‌ها از توزیع نرمال – داده‌های خارج از بیضی ناهنجارند

```
from sklearn.ensemble import IsolationForest

model = IsolationForest()
model.fit(X)
outliers = model.predict(X)  # مقدار -1 یعنی ناهنجار
```

✓ بخش4: مدل‌های آماری بدون نظارت

مدل	توضیح	ابزار
Gaussian Mixture Model (GMM)	مدل احتمالی برای داده‌های خوشه‌ای با شکل بیضوی	`GaussianMixture`
Isolation Forest	تشخیص ناهنجاری در داده	`IsolationForest`
One-Class SVM	مدل SVM برای تشخیص ناهنجاری	`OneClassSVM`

✓ مثال از GMM:

```
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=3, random_state=42)
gmm.fit(X)
labels = gmm.predict(X)
```

✅ خلاصه فصل 3:

دسته	روش‌ها	کاربرد
خوشه‌بندی	KMeans, DBSCAN, Agglomerative	گروه‌بندی داده‌ها
کاهش ابعاد	PCA, t-SNE, TruncatedSVD	ساده‌سازی و تجسم
تشخیص ناهنجاری	IsolationForest, GMM, One-Class SVM	شناسایی داده‌های پرت یا غیرطبیعی

📘 فصل 5: یادگیری تقویتی (Reinforcement Learning)

📌 در یادگیری تقویتی، یک عامل (Agent) در یک محیط (Environment) با انجام **اقدام (Action)** و گرفتن **پاداش (Reward)**، یاد می‌گیرد چه تصمیم‌هایی بگیرد تا بیشترین پاداش ممکن را کسب کند.

✅ تعریف اجزای اصلی:

مفهوم	توضیح کوتاه
Agent	موجود تصمیم‌گیر (مثل ربات، مدل)
Environment	محیطی که Agent در آن تعامل دارد

مفهوم	توضیح کوتاه
State	وضعیت فعلی Agent در محیط
Action	انتخابی که Agent انجام می‌دهد
Reward	امتیازی که بعد از Action دریافت می‌شود
Policy	استراتژی انتخاب عمل در هر وضعیت
Value Function	ارزش وضعیت‌ها (با توجه به پاداش‌های آینده)

✅ الگوریتم‌های رایج (مبتنی بر Python):

🔸 در `scikit-learn` الگوریتم RL وجود ندارد
اما می‌توان با کتابخانه‌های زیر استفاده کرد:

🎁 کتابخانه‌های مخصوص RL:

کتابخانه	توضیح
`stable-baselines3`	کتابخانه حرفه‌ای برای پیاده‌سازی RL در پایتون
`gymnasium` (یا `gym`)	شبیه‌ساز محیط‌های استاندارد RL
`RLlib`	فریم‌ورک توزیع‌شده برای آموزش مدل‌های RL
`Keras-RL`	RL با TensorFlow/Keras

✅ الگوریتم‌های مهم یادگیری تقویتی:

الگوریتم	نوع	توضیح کوتاه
Q-Learning	Value-Based	نگهداری Q-جدول برای تصمیم‌گیری
SARSA	Value-Based	مشابه Q-Learning ولی با سیاست فعلی
Deep Q-Network (DQN)	Value-Based	استفاده از شبکه عصبی به‌جای Q-table
Policy Gradient	Policy-Based	بهبود مستقیم سیاست تصمیم‌گیری
Actor-Critic	Combined	ترکیب سیاست و تابع ارزش
PPO (Proximal Policy Optimization)	Advanced	الگوریتم پایدار و مدرن برای یادگیری سیاست
A3C/A2C	Multi-agent	یادگیری موازی با چند عامل

✅ مثال ساده با `stable-baselines3` :

```
pip install stable-baselines3[extra] gymnasium
import gymnasium as gym
from stable_baselines3 import DQN

env = gym.make("CartPole-v1")
model = DQN("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=10000)
obs, _ = env.reset()

for _ in range(1000):
    action, _ = model.predict(obs)
    obs, reward, terminated, truncated, _ = env.step(action)
    if terminated or truncated:
        obs, _ = env.reset()
```

✅ کاربردهای RL:

- بازی‌ها (مثل شطرنج، Atari، Go)
- کنترل ربات‌ها
- مدیریت منابع (مثلاً CPU، حافظه، شبکه)
- معاملات مالی الگوریتمی

📘 فصل 6 : مدل‌های یادگیری تقویتی و ترکیبی

(Ensemble Learning & Boosting Methods)

> این فصل درباره مدل‌هایی است که **ترکیبی از چند مدل ضعیف‌تر** را برای ساختن یک مدل قدرتمند استفاده می‌کنند. این مدل‌ها معمولاً دقت بالاتری نسبت به مدل‌های تکی دارند.

✅ بخش 6.1: یادگیری ترکیبی (Ensemble Learning)

📌 ایده اصلی: به جای استفاده از یک مدل، چند مدل را با هم ترکیب می‌کنیم تا تصمیم نهایی بهتر شود.

✔ روش‌های رایج ترکیبی:

روش	توضیح	ابزار در `sklearn`
Bagging	ترکیب چند مدل روی نمونه‌های مختلف داده	`BaggingClassifier`, `BaggingRegressor`
Boosting	مدل‌ها به‌صورت زنجیره‌ای ساخته می‌شوند، هر مدل جدید خطاهای قبلی را اصلاح می‌کند	`AdaBoostClassifier`, `GradientBoostingClassifier`
Stacking	چند مدل مختلف آموزش می‌بینند و خروجی آن‌ها به یک مدل نهایی داده می‌شود	`StackingClassifier`, `StackingRegressor`
Voting	چند مدل آموزش می‌بینند و رأی‌گیری نهایی انجام می‌شود	`VotingClassifier`

✓ مثال ساده از VotingClassifier:

```
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC

model1 = LogisticRegression()
model2 = DecisionTreeClassifier()
model3 = SVC(probability=True)

voting_model = VotingClassifier(estimators=[
    ('lr', model1), ('dt', model2), ('svc', model3)],
    voting='soft')

voting_model.fit(X_train, y_train)
```

✓ بخش 6.2 : Bagging (Bootstrap Aggregation)

📌 مفهوم: مدل‌ها روی نمونه‌های متفاوتی از داده آموزش می‌بینند (با جایگذاری). سپس پیش‌بینی‌ها میانگین یا رأی‌گیری می‌شوند.

✓ معروف‌ترین پیاده‌سازی: **Random Forest**

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=100, max_depth=5,
random_state=42)
```

```
model.fit(X_train, y_train)
```

✅ بخش 6.3: Boosting – تقویت مدل‌ها

📌 مدل‌های جدید **به‌صورت ترتیبی** ساخته می‌شوند، هر مدل جدید روی **خطاهای مدل قبلی** تمرکز می‌کند.

🎁 روش‌های Boosting در Scikit-learn:

روش	توضیح	ابزار
AdaBoost	مدل پایه با وزن‌دهی به نمونه‌های مشکل‌دار	`AdaBoostClassifier`
Gradient Boosting	مدل پایه با کاهش گرادیانی خطا	`GradientBoostingClassifier`
HistGradientBoosting	نسخه سریع‌تر و دقیق‌تر برای داده‌های بزرگ	`HistGradientBoostingClassifier` (جدیدتر)

✓ مثال از Gradient Boosting:

```
from sklearn.ensemble import GradientBoostingClassifier

gb_model = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1,
max_depth=3)
gb_model.fit(X_train, y_train)
```

✅ بخش 6.4 : Stacking – مدل‌سازی چندمرحله‌ای

📌 چند مدل به عنوان Base-Models آموزش می‌بینند.
📌 خروجی آن‌ها به یک **مدل نهایی (Meta-Model)** داده می‌شود.

✓ مفید وقتی مدل‌ها رفتار متفاوت دارند.

```
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier

base_models = [
```

```
    ('rf', RandomForestClassifier(n_estimators=10)),
    ('gb', GradientBoostingClassifier(n_estimators=10))
]

meta_model = LogisticRegression()

stacking_model = StackingClassifier(estimators=base_models,
final_estimator=meta_model)
stacking_model.fit(X_train, y_train)
```

✅ خلاصه فصل 6:

روش	مناسب برای	مزایا	معایب
Bagging	کاهش واریانس	سریع و پایدار	مدل‌های مستقل لازم
Boosting	کاهش بایاس	دقت بالا	زمان‌بر، حساس به نویز
Stacking	ترکیب مدل‌های متنوع	انعطاف‌پذیر	پیاده‌سازی پیچیده‌تر
Voting	ساده‌ترین ترکیب	آسان، سریع	معمولاً دقیق‌تر از تک مدل نیست مگر مدل‌های قوی انتخاب شوند

📘 فصل 7: یادگیری نیمه‌نظارتی، برچسب‌زنی، و دیگر تکنیک‌های خاص

✅ بخش 1: یادگیری نیمه‌نظارتی (Semi-Supervised Learning)

🔍 تعریف:

یادگیری با ترکیب داده‌های برچسب‌دار (labeled) و بدون برچسب (unlabeled) برای آموزش بهتر مدل.

ویژگی	توضیح
کاربرد	زمانی که برچسب‌گذاری کل داده بسیار هزینه‌بر یا زمان‌بر باشد
هدف	استفاده از ساختار داده‌ی unlabeled برای بهبود یادگیری

ویژگی	توضیح
مثال‌های کاربردی	تحلیل متن، تشخیص اسپم، تشخیص تقلب، پردازش تصویر

🎁 ابزار در Scikit-learn:

Scikit-learn مستقیماً Semi-supervised را پشتیبانی می‌کند:

🔹 `LabelPropagation`

```
from sklearn.semi_supervised import LabelPropagation

model = LabelPropagation()
model.fit(X, y)  # y باید شامل مقادیر -1 برای داده‌های بدون برچسب باشد
```

🔹 `LabelSpreading`

```
from sklearn.semi_supervised import LabelSpreading

model = LabelSpreading(kernel='knn', n_neighbors=7)
model.fit(X, y)
```

✅ نکته: در این مدل‌ها `y_unlabeled = -1` است و مدل خودش مقادیر گمشده را حدس می‌زند.

✅ بخش 2: یادگیری فعال (Active Learning)

🔍 تعریف:

مدل به‌جای برچسب زدن همه داده‌ها، **خودش انتخاب می‌کند** که کدام نمونه‌ها را باید برچسب زد.

| کاربرد | زمانی که برچسب زدن بسیار پرهزینه است و می‌خواهیم فقط بهترین داده‌ها را برچسب بزنیم |

🔧 این روش بیشتر با کتابخانه‌هایی مانند `libact` , `modAL` استفاده می‌شود (در scikit-learn نیست).

✅ بخش 3: یادگیری با داده‌های ناقص یا نویزی (Weak Supervision)

🔍 یادگیری از برچسب‌های غیرقطعی، قوانین تقریبی یا منابع ضعیف (مانند کرادسورسینگ یا Heuristics).

🛠 ابزارهای معروف:

- `Snorkel`
- `Data Programming`

✅ بخش 4: یادگیری با داده‌های برچسب‌نخورده (Unlabeled Data) با کمک خوشه‌بندی

✅ گاهی ابتدا با **خوشه‌بندی (Clustering)** داده‌ها را گروه‌بندی می‌کنیم
و سپس از نتایج آن برای ایجاد برچسب اولیه استفاده می‌کنیم (Pseudo-Labeling)

✅ بخش 5: کاربرد ترکیبی در پروژه‌ها

موقعیت پروژه واقعی	راهکار پیشنهاد شده
۱۰٪ داده برچسب‌دار، ۹۰٪ بدون برچسب	Semi-Supervised (`LabelPropagation`)
۵۰۰هزار نمونه بدون برچسب، هزینه بالا	با انتخاب بهترین نمونه‌ها Active Learning
داده‌های کم + خوشه‌بندی اولیه	استفاده از Clustering برای تولید pseudo-label

---

📘 فصل 8: ارزیابی عملکرد مدل‌ها (Model Evaluation)

├── ارزیابی در Classification
│ ├── Confusion Matrix
│ ├── Accuracy, Precision, Recall, F1
│ ├── ROC, AUC
│ ├── Threshold (آستانه تصمیم‌گیری)
│ └── مثال تغییر آستانه در LogisticRegression
│
├── ارزیابی در Regression
├── ارزیابی Multi-class
├── ارزیابی Multi-label
├── ارزیابی با Cross-validation
├── فاصله اطمینان (Confidence Interval)

✅ بخش 1: ارزیابی مدل‌های طبقه‌بندی دودویی (Binary Classification)

📌 مدل‌هایی مثل: اسپم/نه اسپم، بله/خیر، بیمار/سالم

متریک	توضیح	ابزار Scikit-learn
Accuracy	نسبت نمونه‌هایی که به‌درستی پیش‌بینی شده‌اند	`accuracy_score`
Precision	درصد پیش‌بینی‌های مثبت که واقعاً مثبت بودند فرمول: Precision = `TP / (TP + FP)`	`precision_score`
Recall	درصد نمونه‌های مثبت واقعی که به درستی پیش‌بینی شده‌اند فرمول: Recall = `TP / (TP + FN)`	`recall_score`
F1-Score	میانگین هارمونیک precision و recall فرمول: F1 = `2 * (P * R) / (P + R)`	`f1_score`
Confusion Matrix	جدولی شامل: True Positives, False Positives, False Negatives, True Negatives	`confusion_matrix`
ROC Curve	منحنی نرخ مثبت صحیح (TPR) در برابر نرخ مثبت کاذب (FPR) مقدار محور Y: `TP / (TP + FN)` محور X: `FP / (FP + TN)`	`roc_curve`
AUC (ROC-AUC)	مساحت زیر منحنی ROC. Receiver Operating Characteristic Area Under the Curve معیاری از توان تمایز بین کلاس‌ها. عددی بین 0.5 تا 1.0. هرچه بیشتر بهتر	`roc_auc_score`
PR Curve	منحنی‌ای بین Precision و Recall. مفید در داده‌های Imbalanced	`precision_recall_curve`

🎁 ابزارهای Scikit-learn:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, confusion_matrix
```

✅ بخش 2: ارزیابی مدل‌های رگرسیون

📌 برای پیش‌بینی مقادیر عددی (مثلاً قیمت، دما، درآمد)

متریک	توضیح	ابزار sklearn
MAE	میانگین قدرمطلق خطاها	`mean_absolute_error`
MSE	میانگین توان دوم خطاها	`mean_squared_error`
RMSE	ریشه دوم MSE	دستی با `np.sqrt()`
R² (R-squared)	درصد واریانس قابل توضیح	`r2_score`

🎁 ابزارهای Scikit-learn:

```
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score
```

✅ بخش 3: ارزیابی مدل‌های چندکلاسه (Multiclass)

📌 مثل: پیش‌بینی عدد 0 تا 9، یا کلاس‌های گربه، سگ، پرنده

🔹 متریک‌ها مثل binary هستند اما با **استراتژی macro, micro, weighted** ترکیب می‌شوند:

```
f1_score(y_true, y_pred, average='macro')
```

✅ بخش 4: ارزیابی مدل‌های Multi-Label و Multi-Output

اصطلاح	تعریف
Multi-Label	هر نمونه می‌تواند چند کلاس داشته باشد (مثلاً ایمیل هم اسپم، هم تبلیغاتی)
Multi-Output	مدل چند خروجی عددی یا دسته‌ای دارد (مثلاً پیش‌بینی دما و رطوبت)

🎁 ابزار sklearn:

```
from sklearn.metrics import classification_report

print(classification_report(y_true, y_pred, target_names=...))
```

✅ بخش 5: ارزیابی با اعتبارسنجی متقابل (Cross Validation Evaluation)

📌 با استفاده از `cross_val_score` می‌توان ارزیابی را با تکرار روی foldهای مختلف انجام داد:

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(model, X, y, scoring='f1_macro', cv=5)
print(scores.mean())
```

✅ بخش 6: فاصله اطمینان (Confidence Interval) در ارزیابی مدل

فاصله اطمینان یعنی:

بازه‌ای از مقادیر که **با درصد اطمینان مشخصی** انتظار داریم مقدار واقعی (مثلاً دقت واقعی مدل یا میانگین جامعه) داخل اون بازه باشه.

✅ مثال عددی ساده:

فرض کن دقت (Accuracy) مدل شما در 10 بار cross-validation اینه:

```
accuracies = [0.83, 0.81, 0.85, 0.82, 0.84, 0.86, 0.80, 0.83, 0.82, 0.84]
```

می‌خوای بدونی دقت واقعی مدل چقدره، نه فقط میانگین!

🔢 محاسبه دستی فاصله اطمینان 95٪:

می‌گیم با 95٪ اطمینان، دقت واقعی مدل در این بازه هست:

```
import numpy as np
import scipy.stats as stats

mean = np.mean(accuracies)
std_err = stats.sem(accuracies)  # انحراف معیار / sqrt(n)
conf_interval = stats.t.interval(0.95, len(accuracies)-1, loc=mean,
```

```
scale=std_err)

print(f"📊 دقت میانگین: {mean:.3f}")
print(f"✅ فاصله اطمینان ۹۵٪: {conf_interval}")
```

📤 خروجی:

```
📊 دقت میانگین: 0.832
✅ فاصله اطمینان ۹۵٪: (0.818, 0.846)
```

➕ یعنی می‌تونیم با ۹۵٪ اطمینان بگیم دقت واقعی بین **۸۱.۸٪ تا ۸۴.۶٪** هست.

✅ چه فایده‌ای داره؟

- فقط گفتن "میانگین دقت 83٪" **کافی نیست**
- ولی گفتن: "با ۹۵٪ اطمینان بین 81٪ تا 84٪ است" ⟵ دید **آماری و دقیق‌تر** می‌ده
- برای مقایسه مدل‌ها:
  اگر فاصله‌های اطمینان دوتا مدل هم‌پوشانی نداشته باشند ⟵ مدل‌ها تفاوت معنی‌دار دارند

✅ کجاها از فاصله اطمینان استفاده کنیم؟

موقعیت	چرا؟
🔹 مقایسه چند مدل با هم	ببینی کدوم عملکرد واقعاً بهتره، نه فقط از روی میانگین
🔹 گزارش نتایج پروژه	مثلاً تو گزارش بنویسی: دقت مدل 83٪ ± 2٪ (فاصله اطمینان)
🔹 تحلیل آماری ویژگی‌ها یا مقادیر پیش‌بینی	مثلاً بگی میانگین حقوق در دیتاست بین 42k تا 48k دلار با 95٪ اطمینان

✅📍 فاصله اطمینان (Confidence Interval) کجا باید بیاد؟

کاربرد فاصله اطمینان	فصل مناسب در جزوه	چرا؟
ارزیابی دقت مدل	✅ فصل **ارزیابی مدل‌ها** (Validation)	برای نشان دادن "عدم قطعیت" در تخمین دقت
تحلیل آماری روی داده‌ها	✅ فصل **تحلیل داده‌های آماری**	مثل برآورد میانگین ویژگی‌ها
مقایسه بین چند مدل	✅ فصل **بهینه‌سازی مدل‌ها**	برای دیدن تفاوت معنی‌دار یا نه

📘 فصل 9: تنظیم و بهینه‌سازی مدل‌ها

✅ پیش نیاز

فرق پارامتر (Parameter) با ابرپارامتر (Hyperparameter)

مورد	پارامتر (Parameter)	ابرپارامتر (Hyperparameter)
🔍 تعریف	مقادیری هستند که **مدل در طول آموزش یاد می‌گیرد**	مقادیری هستند که **قبل از آموزش توسط کاربر تنظیم می‌شوند**
📌 مثال در مدل خطی	وزن‌ها (Weights) و عرض از مبدأ (bias/intercept)	نرخ یادگیری (learning rate)، تعداد تکرارها (epochs)
🧠 چه کسی آن را تعیین می‌کند؟	مدل خودش در فرآیند یادگیری	شما (یا با استفاده از ابزارهایی مثل GridSearchCV)
🎯 نقش اصلی	کنترل مستقیم خروجی مدل	تأثیر غیرمستقیم بر نحوه آموزش مدل
💡 محل استفاده	داخل مدل (مثل w و b در خط رگرسیون)	خارج مدل: نحوه آموزش، پیچیدگی مدل، تنظیمات ساختاری

✅ چند مثال عینی:

مدل	پارامترها	ابرپارامترها
Linear Regression	وزن‌ها و بایاس	-
SVM	-	`C`, `kernel`, `gamma`
KNN	-	`n_neighbors`
Decision Tree	-	`max_depth`, `min_samples_split`
Neural Networks (MLP)	وزن‌ها، بایاس	`learning_rate`, `hidden_layer_sizes`

✅ بخش 1: تفاوت Model Tuning (تنظیم مدل) و Model Optimization (بهینه‌سازی مدل)

هرچند مفاهیم **Model Tuning (تنظیم مدل)** و **Model Optimization (بهینه‌سازی مدل)** در یادگیری ماشین اغلب به جای یکدیگر به کار می‌روند و همپوشانی زیادی دارند، اما تفاوت‌های ظریفی نیز بین آن‌ها وجود دارد که درک آن‌ها می‌تواند مفید باشد:

مورد	Hyperparameter Tuning	Model Optimization
تعریف	جستجو برای بهترین تنظیمات	بهبود عملکرد کلی مدل
ابزار	GridSearchCV, RandomizedSearchCV	استفاده از الگوریتم‌ها، تنظیم نرخ یادگیری، انتخاب ویژگی‌ها
هدف	پیدا کردن بهترین پارامتر	افزایش دقت، سرعت، پایداری مدل
مکان	فصل 7	فصل 8

1. عبارت Model Tuning (تنظیم مدل) یا Hyperparameter Tuning (تنظیم ابرپارامتر)

تعریف: Model Tuning به فرآیند پیدا کردن بهترین مجموعه از **ابرپارامترها (Hyperparameters)** برای یک مدل یادگیری ماشین گفته می‌شود تا عملکرد آن را به حداکثر برساند. ابرپارامترها، متغیرهای پیکربندی مدل هستند که **نمی‌توانند** از طریق داده‌های آموزشی یاد گرفته شوند. این متغیرها رفتار مدل را در طول آموزش و ساختار خود مدل را تعیین می‌کنند.

مثال‌هایی از ابرپارامترها:

- **نرخ یادگیری (Learning Rate):** در شبکه‌های عصبی، سرعت تغییر وزن‌ها در هر مرحله از آموزش را تعیین می‌کند.
- **اندازه دسته (Batch Size):** تعداد نمونه‌هایی که در هر مرحله آموزش برای به‌روزرسانی وزن‌ها استفاده می‌شوند.
- **تعداد لایه‌های پنهان (Number of Hidden Layers):** در یک شبکه عصبی.
- **نوع تابع فعال‌سازی (Activation Function):** در شبکه‌های عصبی (مانند ReLU, Sigmoid, Tanh).
- **ضریب رگولاریزاسیون (Regularization Parameter):** برای کنترل بیش‌برازش (Overfitting).
- **تعداد درختان (Number of Trees):** در مدل‌های جنگل تصادفی (Random Forest) یا گرادیان بوستینگ (Gradient Boosting).

هدف: یافتن ترکیب بهینه‌ای از ابرپارامترها که مدل بهترین عملکرد را بر روی داده‌های جدید (داده‌های تست یا اعتبارسنجی) داشته باشد، یعنی بهترین تعمیم‌پذیری (Generalization) را از خود نشان دهد.

روش‌ها:

- **جستجوی شبکه‌ای (Grid Search):** امتحان کردن تمام ترکیبات ممکن از مقادیر مشخص شده برای هر ابرپارامتر.
- **جستجوی تصادفی (Random Search):** نمونه‌برداری تصادفی از فضای ابرپارامترها.
- **بهینه‌سازی بیزی (Bayesian Optimization):** استفاده از مدل‌های احتمالی برای راهنمایی جستجو به سمت مناطق promisingتر.
- **بهینه‌سازی تکاملی (Evolutionary Optimization):** الهام گرفته از تکامل طبیعی.

۲. Model Optimization (بهینه‌سازی مدل)

تعریف کلی: Model Optimization یک مفهوم گسترده‌تر است که به هر فرآیندی اشاره دارد که با هدف بهبود کارایی، عملکرد، یا کارایی یک مدل یادگیری ماشین انجام می‌شود. این می‌تواند شامل جنبه‌های مختلفی باشد، نه فقط تنظیم ابرپارامترها.

جنبه‌های مختلف Model Optimization:

- **بهینه‌سازی پارامترهای داخلی مدل (Parameter Optimization):** این مورد معمولاً در طول فرآیند آموزش مدل اتفاق می‌افتد. الگوریتم‌های بهینه‌سازی (مانند Gradient Descent و انواع آن) برای یافتن بهترین وزن‌ها و بایاس‌ها (Parameters) که تابع هزینه (Loss Function) را حداقل می‌کنند، استفاده می‌شوند. این پارامترها برخلاف ابرپارامترها، از داده‌های آموزشی **یاد گرفته می‌شوند**.
  - **مثال:** استفاده از بهینه‌ساز SGD، RMSprop، Adam و ... برای به‌روزرسانی وزن‌های یک شبکه عصبی.
- **تنظیم ابرپارامترها (Hyperparameter Tuning):** همان Model Tuning که در بالا توضیح داده شد، زیرمجموعه‌ای از Model Optimization محسوب می‌شود، زیرا هدف آن بهبود عملکرد مدل است.
- **بهینه‌سازی ساختار مدل (Model Architecture Optimization):** انتخاب یا طراحی بهترین ساختار برای مدل (مثلاً تعداد لایه‌ها و نورون‌ها در یک شبکه عصبی، یا نوع و تعداد لایه‌های کانولوشن در CNN). این مورد می‌تواند شامل Automated Machine Learning (AutoML) نیز باشد که به صورت خودکار به دنبال بهترین معماری و ابرپارامترها می‌گردد.
- **بهینه‌سازی برای استقرار (Deployment Optimization):** پس از آموزش و تنظیم مدل، ممکن است نیاز باشد مدل برای استقرار در محیط‌های عملیاتی بهینه شود. این شامل:
  - **کوانتیزاسیون (Quantization):** کاهش دقت عددی وزن‌ها و فعال‌سازی‌ها (مثلاً از Float32 به Float16 یا Int8) برای کاهش حجم مدل و افزایش سرعت inference.
  - **هرس کردن (Pruning):** حذف وزن‌ها یا اتصالات کم‌اهمیت در شبکه برای کاهش پیچیدگی و اندازه مدل.
  - **تبدیل مدل (Model Conversion):** تبدیل مدل به فرمت‌های بهینه‌سازی شده برای پلتفرم‌های خاص (مثلاً ONNX, TensorFlow Lite).
  - **تقطیر مدل (Model Distillation):** آموزش یک مدل کوچک‌تر (دانشجو) برای تقلید از رفتار یک مدل بزرگ‌تر و پیچیده‌تر (معلم).
- **بهینه‌سازی مجموعه داده (Data Optimization):** هرچند مستقیماً به خود مدل مربوط نیست، اما کیفیت و ویژگی‌های داده‌ها نیز بر عملکرد مدل تأثیر بسزایی دارند. پیش‌پردازش داده، انتخاب ویژگی (Feature Selection)، مهندسی ویژگی (Feature Engineering) و تعادل کلاس‌ها همگی می‌توانند به بهبود عملکرد مدل کمک کنند و بخشی از یک رویکرد جامع بهینه‌سازی هستند.

✓ بخش 2: تنظیم هایپرپارامترها (Hyperparameter Tuning)

📌 توجه: Tuning فقط به انتخاب بهترین پارامترها مربوط میشه.
📌 مدل‌ها دارای پارامترهایی هستند که باید دستی تنظیم شوند مثل `C` در SVM یا `n_estimators` در RandomForest

---

🎁 ابزارهای Scikit-learn:

این ابزار **نوعی بهینه‌سازی غیرگرادیانی** محسوب می‌شوند.
در واقع فضای جستجو را اسکن می‌کنند تا بهترین مقدار برای پارامترها پیدا شود.

1. `GridSearchCV` : جستجوی کامل در بین تمام ترکیب‌های پارامتر

```
from sklearn.model_selection import GridSearchCV

param_grid = {'clf__C': [0.1, 1, 10]}
grid = GridSearchCV(pipe, param_grid, cv=5, scoring='accuracy')
grid.fit(X_train, y_train)
```

✓ این روش دقیق است ولی برای شبکه پارامترهای بزرگ، زمان‌بر است.

---

2. `RandomizedSearchCV` : جستجوی تصادفی در ترکیب‌ها

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform

param_dist = {'clf__C': uniform(0.01, 10)}
search = RandomizedSearchCV(pipe, param_distributions=param_dist, n_iter=20,
cv=5)
search.fit(X_train, y_train)
```

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_dist = {
    'n_estimators': randint(10, 100),
    'max_depth': randint(3, 10)
}

random_search = RandomizedSearchCV(RandomForestClassifier(),
param_distributions=param_dist, n_iter=10, cv=5)
random_search.fit(X_train, y_train)
```

✓ سریع‌تر از GridSearchCV است، مخصوصاً وقتی پارامتر زیاد داریم.

✅ بخش 2.2: ترکیب با Cross Validation

📌 معمولاً GridSearchCV و RandomizedSearchCV همراه با cross-validation انجام می‌شوند تا **مدل روی داده‌های مختلف تست شود.**

توجه که روش جدیدی نیست بلکه باعث می‌شود ارزیابی بهینه‌تر انجام شود.
الگوریتم همان است، فقط دقیق‌تر و مقاوم‌تر به overfitting می‌شود.
✓ این ارزیابی دقیق‌تر از Hold-out است.

📌 عبارت**Hold-Out Validation چیه؟**

- ساده‌ترین روش اعتبارسنجی است که فقط یک بار داده را به train/test تقسیم می‌کنیم (معمولاً با `train_test_split`)
- در مقایسه با Cross-Validation دقت کمتری دارد چون مدل فقط یک‌بار ارزیابی می‌شود.

```
from sklearn.model_selection import StratifiedKFold, RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from scipy.stats import randint

cv_strategy = StratifiedKFold(n_splits=5)

param_dist = {
    'n_estimators': randint(10, 100),
    'max_depth': randint(3, 10)
}

random_search = RandomizedSearchCV(
    estimator=RandomForestClassifier(),
    param_distributions=param_dist,
    n_iter=10,
    cv=cv_strategy,  # اینجا به‌صراحت نشان می‌دهیم که با CV ترکیب شده
    scoring='accuracy',
    random_state=42
)

random_search.fit(X_train, y_train)
```

🔍 در اینجا `StratifiedKFold` یک استراتژی CV است که به صورت مشخص با `RandomizedSearchCV` ترکیب شده.

✅ بخش 2.3: استخراج بهترین مدل

```
# بهترین پارامترها
print(grid.best_params_)

# بهترین مدل آموزش‌دیده
best_model = grid.best_estimator_
```

✅ بخش 3: بهینه‌سازی مدل‌ها (Model Optimization)

بخش	عنوان	توضیح
✅ بخش 10.1	الگوریتم SGD (Stochastic Gradient Descent)	بهینه‌سازی با داده‌های بزرگ
✅ بخش 10.2	پارامترها و سالورهای مهم در Scikit-learn	مثل `loss` , `penalty` , `learning_rate` و ...
✅ بخش 10.3	معرفی ابزارهای مهم `sklearn.linear_model` و `sklearn.preprocessing`	کدهای ایمپورت و پارامترهای متداول

✅ این فصل **مکمل فصل 7** است که در آن به **تنظیم مدل** از طریق جستجوی پارامتر (مثل GridSearchCV) پرداختیم.
در این فصل، بیشتر به **مباحث الگوریتمی، تنظیمات کلی مدل و پارامترهای یادگیری** می‌پردازیم که بر دقت و سرعت یادگیری مدل‌ها اثر می‌گذارند.

✅ بخش 3.1: الگوریتم SGD – گرادیان نزولی تصادفی

📌 عبارت **SGD** مخفف **Stochastic Gradient Descent** است.

✓ مناسب برای داده‌های بزرگ، مدل‌هایی که نیاز به بروزرسانی سریع دارند و مشکلاتی با فضای جستجوی بزرگ.

🎁 مدل‌ها در Scikit-learn:

مدل	کاربرد
`SGDClassifier`	برای دسته‌بندی
`SGDRegressor`	برای رگرسیون

```
from sklearn.linear_model import SGDClassifier, SGDRegressor
```

📌 ویژگی‌های کلیدی + مثال کاربردی:

پارامتر	توضیح	مثال ساده کد در Scikit-learn
`loss`	نوع تابع هزینه (مثل `'hinge'` برای SVM، یا `'log'` برای Logistic Regression)	`SGDClassifier(loss='log')` ← مشابه Logistic Regression
`penalty`	نوع منظم‌سازی برای جلوگیری از بیش‌برازش (`'l2'`, `'l1'`, `'elasticnet'`)	`SGDClassifier(penalty='l1')` ← استفاده از L1 برای سادگی مدل
`alpha`	ضریب رگولاریزاسیون که شدت جریمه را کنترل می‌کند	`SGDClassifier(alpha=0.0001)` ← مقدار پیش‌فرض
`learning_rate`	نحوه تغییر نرخ یادگیری در طول زمان (`'constant'`, `'optimal'`, `'adaptive'`)	`SGDClassifier(learning_rate='adaptive')` ← کاهش نرخ در صورت توقف بهبود
`max_iter`	تعداد تکرار برای آموزش مدل	`SGDClassifier(max_iter=1000)` ← آموزش حداکثر ۱۰۰۰ تکرار

✅ مثال کامل:

```
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split

# داده ساده
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

# مدل با تنظیمات خاص
clf = SGDClassifier(loss='log', penalty='elasticnet', alpha=0.001,
                    learning_rate='optimal', max_iter=1000, random_state=42)

clf.fit(X_train, y_train)

print("دقت مدل:", clf.score(X_test, y_test))
```

📌 در این مثال:

- تابع هزینه `'log'` انتخاب شده تا رفتار مدل شبیه Logistic Regression شود.

- از ترکیب دو نوع رگولاریزاسیون با `elasticnet` استفاده شده.
- عبارت `'learning_rate='optimal` یعنی از نرخ یادگیری تطبیقی خودکار استفاده شود.
- عبارت `max_iter=1000` حداکثر تکرار برای آموزش است.

✅ بخش 3.2: ماژول‌های کلیدی Scikit-learn برای بهینه‌سازی

مدل	کاربرد
`LogisticRegression`	طبقه‌بندی باینری یا مالتی‌کلاس
`LinearRegression`	رگرسیون خطی ساده
`Ridge` , `Lasso`	مدل‌های منظم‌شده برای رگرسیون

🎁 در `sklearn.linear_model`

```
from sklearn.linear_model import LogisticRegression, LinearRegression,
Ridge, Lasso, SGDClassifier, SGDRegressor
```

🎁 در `sklearn.preprocessing`

ابزار	کاربرد	
`StandardScaler`	نرمال‌سازی داده با میانگین صفر و انحراف معیار یک	
`MinMaxScaler`	مقیاس‌بندی داده‌ها در بازه [0, 1]	
`RobustScaler`	مقاوم به داده‌های پرت (با صدک‌ها)	
`OneHotEncoder`	تبدیل داده‌های طبقه‌ای به بردار باینری	
`LabelEncoder`	تبدیل لیبل‌ها به اعداد صحیح (فقط برای y)	

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler,
RobustScaler, OneHotEncoder, LabelEncoder
```

✅ بخش 3.3: تکنیک‌های پیشرفته در Optimization (در سطح بالاتر)

در حالی که تکنیک‌های پایه بهینه‌سازی برای شروع کار با مدل‌ها کافی هستند، در پروژه‌های پیچیده‌تر و با مدل‌های بزرگ‌تر (مانند شبکه‌های عصبی عمیق)، نیاز به ابزارهای پیشرفته‌تری برای **افزایش عملکرد، پایداری، و کارایی فرآیند آموزش** داریم. این تکنیک‌ها به شما کمک می‌کنند تا از مشکلات رایج جلوگیری کرده و مدل‌های قوی‌تری بسازید.

📌 این بخش اختیاری و برای کسانی است که مدل‌های پیچیده‌تر می‌خواهند:

- تکنیک**EarlyStopping** در Keras یا PyTorch برای جلوگیری از overfitting
- تکنیک **Bayesian Optimization** (مثلاً با `bayes_opt` , `optuna`)
- تکنیک **Learning Rate Scheduler**
- تکنیک **Gradient Clipping** برای پایداری در شبکه‌های عصبی

> ✓ این ابزارها فعلاً در Scikit-learn نیستند اما برای پروژه‌های حرفه‌ای‌تر استفاده می‌شوند.

تکنیک 1. Early Stopping (توقف زودهنگام)

تصور کنید در حال آموزش یک مدل هستید و مدل شما بر روی داده‌های آموزشی مدام بهتر می‌شود. اما آیا این بهبود به معنی عملکرد بهتر بر روی داده‌های جدید (داده‌هایی که مدل قبلاً ندیده است) نیز هست؟ نه لزوماً! **Early Stopping** یک تکنیک قدرتمند برای جلوگیری از **بیش‌برازش (Overfitting)** است. این روش، آموزش مدل را زمانی متوقف می‌کند که عملکرد مدل بر روی **داده‌های اعتبارسنجی (Validation Data)** - که داده‌های جدیدی هستند و مدل برای آموزش از آن‌ها استفاده نکرده - شروع به بدتر شدن کند. با این کار، از ادامه آموزش بی‌هدف که فقط باعث می‌شود مدل الگوهای نویز در داده‌های آموزشی را یاد بگیرد، جلوگیری می‌شود. این تکنیک به‌خصوص در فریم‌ورک‌هایی مانند Keras و PyTorch به راحتی قابل پیاده‌سازی است.

تکنیک 2. Bayesian Optimization (بهینه‌سازی بیزی)

همانطور که می‌دانید، انتخاب **ابرپارامترهای (Hyperparameters)** مناسب (مثل نرخ یادگیری یا تعداد لایه‌ها) تأثیر زیادی بر عملکرد مدل دارد. **Bayesian Optimization** یک روش هوشمندانه‌تر برای یافتن بهترین ترکیب ابرپارامترها نسبت به روش‌های سنتی مانند Grid Search (جستجوی شبکه‌ای) یا Random Search (جستجوی تصادفی) است. این تکنیک از آمار و احتمال برای "یادگیری" فضای ابرپارامترها استفاده می‌کند. به جای امتحان کردن تصادفی یا تمام مقادیر ممکن، Bayesian Optimization به طور هوشمندانه نقاط جدیدی را برای ارزیابی پیشنهاد می‌دهد که بر اساس نتایج قبلی، احتمال بهبود عملکرد مدل در آن‌ها بیشتر است. این باعث می‌شود فرآیند تنظیم ابرپارامترها **سریع‌تر و کارآمدتر** باشد و زمان کمتری برای رسیدن به بهترین مدل صرف شود. ابزارهایی مانند `optuna` و `bayes_opt` این قابلیت را فراهم می‌کنند.

تکنیک 3. Learning Rate Scheduler (برنامه‌ریز نرخ یادگیری)

نرخ یادگیری (Learning Rate) یکی از مهم‌ترین ابرپارامترها در آموزش مدل‌های یادگیری عمیق است. یک نرخ یادگیری بالا می‌تواند آموزش را سریع کند اما ممکن است باعث شود مدل از نقطه بهینه پرش کند؛ در حالی که نرخ یادگیری پایین آموزش را کند می‌کند. **Learning Rate Scheduler** یک مکانیزم برای **تغییر پویا (داینامیک) نرخ یادگیری** در طول فرآیند آموزش است. به جای استفاده از یک نرخ ثابت، این برنامه‌ریز می‌تواند نرخ یادگیری را با گذشت زمان کاهش دهد (مثلاً

پس از هر تعداد مشخصی از دوره‌های آموزشی یا وقتی عملکرد مدل ثابت می‌ماند). این کار به مدل کمک می‌کند تا در ابتدا سریع‌تر همگرا شود و سپس با نرخ کوچک‌تر، با دقت بیشتری به نقطه بهینه برسد و آموزش پایدارتری داشته باشد.

تکنیک 4. Gradient Clipping (محدود کردن گرادیان)

در شبکه‌های عصبی عمیق، به خصوص در مدل‌هایی مانند شبکه‌های بازگشتی (RNNs)، ممکن است با پدیده‌ای به نام **"انفجار گرادیان" (Exploding Gradients)** مواجه شوید. این اتفاق زمانی می‌افتد که گرادیان‌ها (شیب‌های تابع خطا که برای به‌روزرسانی وزن‌ها استفاده می‌شوند) به طور غیرعادی بزرگ می‌شوند. نتیجه انفجار گرادیان، به‌روزرسانی‌های بسیار بزرگ و ناپایدار وزن‌ها است که می‌تواند منجر به مقادیر "NaN" (Not a Number) در مدل شده و فرآیند آموزش را مختل یا حتی متوقف کند. **Gradient Clipping** با **محدود کردن حداکثر اندازه گرادیان‌ها** از این مشکل جلوگیری می‌کند. این کار تضمین می‌کند که گرادیان‌ها در محدوده قابل کنترل باقی بمانند و آموزش مدل به صورت پایدار ادامه پیدا کند.

نکته مهم: این ابزارها عمدتاً در فریم‌ورک‌های یادگیری عمیق مانند Keras و PyTorch رایج هستند و در کتابخانه‌هایی مانند Scikit-learn که بیشتر بر روی مدل‌های سنتی‌تر تمرکز دارند، کمتر دیده می‌شوند. استفاده از آن‌ها می‌تواند تفاوت بزرگی در عملکرد و پایداری مدل‌های شما در پروژه‌های حرفه‌ای‌تر ایجاد کند.