```python
df = pd.read_csv('AirQualityUCI.csv', sep=';', decimal=',')

# حذف ستون‌های خالی
df = df.drop(['Unnamed: 15', 'Unnamed: 16'], axis=1)

# 200- تبدیل ستون‌های عددی و جایگزین کردن
for col in df.columns:
    if col not in ['Date', 'Time']:
        df[col] = pd.to_numeric(df[col], errors='coerce')
        df[col] = df[col].replace(-200, np.nan)

df = df.dropna()

print(f"-تعداد نمونه ها: {len(df)}")
print(f"- تعداد ستون ها: {df.shape[1]}")
print(df.describe())
```

در این کد داده های مربوط به کیفیت هوا از فایل **AirQualityUCI.csv** خوانده شد. ستون های اضافی حذف شدند و مقادیر عددی به صورت مناسب تبدیل شدند. داده های گمشده که با مقدار (200-) مشخص شده بودند به Nan تغییر یافتند و سپس ردیف های ناقص حذف شدند. در پایان، تعداد نمونه ها و ستون ها چاپ شد و خلاصه ای آماری از داده ها ارائه گردید.

```python
# انتخاب ویژگی‌های ورودی و هدف
feature_cols = ['CO(GT)', 'PT08.S1(CO)', 'C6H6(GT)', 'T', 'RH', 'AH']
target_col = 'NOx(GT)'

X = df[feature_cols].values
y = df[target_col].values.reshape(-1, 1)

print(f"  - میانگین X: {X.mean(axis=0).round(2)}")
print(f"  - میانگین y: {y.mean():.2f}")
print(f"  - بیشینه y: {y.max():.2f}، کمینه y: {y.min():.2f}")
```

در این مرحله ابتدا مجموعه ای از ویژگی های ورودی شامل غلظت گازCO ، سنسورهای مرتبط، دما، رطوبت نسبی و رطوبت مطلق انتخاب شد. ستون هدف نیز مقدار **NOx(GT)** در نظر گرفته شد. سپس داده های مربوط به ویژگی‌ها در متغیر **X** و داده های هدف در متغیر **y** ذخیره شدند.

برای بررسی اولیه داده ها، میانگین هر ویژگی ورودی محاسبه و چاپ شد. همچنین میانگین، بیشینه و کمینه مقدار هدف (NOx) نمایش داده شد تا دید کلی نسبت به توزیع داده ها به دست آید.

```python
# نرمال‌سازی داده‌ها
scaler_X = StandardScaler()
scaler_y = StandardScaler()

X_scaled = scaler_X.fit_transform(X)
y_scaled = scaler_y.fit_transform(y)

# تقسیم به آموزشی و آزمایشی
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y_scaled, test_size=0.2, random_state=42)

print(f"  - Number of Training_set: {X_train.shape[0]}")
print(f"  - Number of Testing_set: {X_test.shape[0]}")
```

در این مرحله داده ها پیش از مدل سازی نرمال سازی شدند تا همه ی ویژگی ها در یک مقیاس مشابه قرار گیرند و اثر اختلاف واحدها کاهش یابد. برای این کار از StandardScaler استفاده شد که داده ها را با **میانگین صفر** و **انحراف معیار یک** تبدیل می‌کند.

```python
class SimpleNeuralNetwork:

    def __init__(self, input_size, hidden_size=16, output_size=1,
                 activation='linear', learning_rate=0.1, adaptive=False):

        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.activation = activation
        self.learning_rate = learning_rate
        self.adaptive = adaptive

        # مقدارهای اولیه وزن‌ها و بایاس‌ها
        self.W1 = np.random.randn(input_size, hidden_size) * 0.1 # وزن‌های بین لایه ورودی و پنهان
        self.b1 = np.zeros((1, hidden_size)) # بایاس لایه پنهان
        self.W2 = np.random.randn(hidden_size, output_size) * 0.1 # وزن‌های بین لایه پنهان و خروجی
        self.b2 = np.zeros((1, output_size)) # بایاس لایه خروجی

        # برای نرخ یادگیری وفقی (Momentum)
        self.v_W1 = np.zeros_like(self.W1)
        self.v_b1 = np.zeros_like(self.b1)
        self.v_W2 = np.zeros_like(self.W2)
        self.v_b2 = np.zeros_like(self.b2)

        self.losses = []
        self.train_accuracies = []
        self.test_accuracies = []
```

سپس داده های ورودی و خروجی به دو بخش آموزشی و آزمایشی تقسیم شدند؛ ۸۰٪ داده ها برای آموزش مدل و ۲۰٪ برای ارزیابی عملکرد آن در نظر گرفته شد. در پایان، تعداد نمونه های هر بخش چاپ گردید تا مشخص شود چه تعداد داده در مجموعه ی آموزشی و آزمایشی وجود دارد.

در این قسمت یک کلاس ساده برای شبکه ی عصبی تعریف شده است. در سازنده‌ی کلاس () اندازه ی ورودی، تعداد نورون های لایه ی پنهان، اندازه ی خروجی، نوع تابع فعال سازی، نرخ یادگیری و امکان استفاده از روش وفقی (Momentum) مشخص می شوند.

وزن ها و بایاس ها برای لایه های ورودی–پنهان و پنهان–خروجی به صورت **تصادفی** مقداردهی اولیه شده اند تا شبکه بتواند فرآیند یادگیری را آغاز کند. همچنین متغیرهایی برای ذخیره ی مقادیر سرعت (Velocity) در روش Momentum تعریف شده اند تا در صورت فعال بودن، به‌روزرسانی وزن ها با شتاب و پایداری بیشتری انجام شود.

در پایان، لیست‌هایی برای ثبت مقادیر خطا (Loss) و دقت مدل در داده های آموزشی و آزمایشی ایجاد شده اند تا روند یادگیری شبکه در طول زمان قابل پیگیری باشد.

```python
#تابع خطی
    def linear(self, x):
        return x
#مشتق تابع خطی
    def linear_derivative(self, x):
        return np.ones_like(x)
#تابع سیگموئید
    def sigmoid(self, x):
        x = np.clip(x, -500, 500)
        return 1 / (1 + np.exp(-x))
#مشتق تابع سیگموئید
    def sigmoid_derivative(self, x):
        sig = self.sigmoid(x)
        return sig * (1 - sig)

    def forward(self, X):
        self.z1 = np.dot(X, self.W1) + self.b1
        # لایه اول: z1 = X·W1 + b1
        # اعمال تابع فعالسازی
        if self.activation == 'linear':
            self.a1 = self.linear(self.z1)
        else:  # sigmoid
            self.a1 = self.sigmoid(self.z1)

        # لایه خروجی
        #لایه دوم: z2 = a1·W2 + b2
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = self.z2  # خروجی بدون فعالسازی تابع فعالسازی
        return self.a2
```

در این قسمت توابع **فعال سازی** و بخش **پیش رو (Forward Pass)** شبکه ی عصبی پیاده سازی شده است. دو تابع فعال سازی در نظر گرفته شده‌اند:

- **تابع خطی (Linear)** که خروجی را بدون تغییر برمیگرداند و مشتق آن همیشه برابر با یک است.
- **تابع سیگموئید (Sigmoid)** که خروجی را **بین ۰ و ۱** نگه می دارد و مشتق آن بر اساس رابطه‌ی

Sig(x) * (1 – sig(x)) محاسبه میشود. برای جلوگیری از خطای محاسباتی، ورودی در بازه‌ی [-500,500] محدود شده است.

در بخش **forward**، ابتدا خروجی لایه ی اول محاسبه میشود ($Z_1 = X_1.W_1+b$) و سپس با توجه به نوع تابع فعال سازی انتخاب شده، مقدار a1 به دست می‌آید. در ادامه، خروجی لایه ی دوم محاسبه میشود ($Z_2=a_1 W_2+b_2$) و به‌عنوان خروجی نهایی شبکه (a2) بازگردانده میشود.

```
def backward(self, X, y):
    m = X.shape[0] #تعداد نمونه ها
    dz2 = self.a2 - y    # خطای خروجی

    # گرادیان‌های لایه دوم
    dW2 = np.dot(self.a1.T, dz2) / m
    db2 = np.sum(dz2, axis=0, keepdims=True) / m

    # خطای لایه پنهان
    da1 = np.dot(dz2, self.W2.T)

    if self.activation == 'linear':# خطی
        dz1 = da1 * self.linear_derivative(self.z1)
    else:  # sigmoid
        dz1 = da1 * self.sigmoid_derivative(self.z1)

    # گرادیان‌های لایه اول
    dW1 = np.dot(X.T, dz1) / m
    db1 = np.sum(dz1, axis=0, keepdims=True) / m

    return dW1, db1, dW2, db2
```

در این بخش **الگوریتم پس انتشار خطا** برای محاسبه ی گرادیان ها پیاده سازی شده است. ابتدا خطای خروجی شبکه محاسبه می‌شود ($dZ_2=a_2 - y$) و سپس گرادیان های وزن ها و بایاس های لایه ی دوم به دست می آیند.

در ادامه، خطا به لایه ی پنهان منتقل میشود و با توجه به نوع تابع فعال سازی (خطی یا سیگموئید) مشتق مناسب اعمال میگردد تا مقدار dZ_1 محاسبه شود. سپس گرادیان های وزن ها و بایاس های لایه ی اول محاسبه میشوند.

در نهایت، مقادیر گرادیان های هر دو لایه بازگردانده میشوند تا در مرحله ی بروزرسانی وزن ها مورد استفاده قرار گیرند. این فرآیند امکان یادگیری شبکه از طریق کاهش خطا را فراهم میکند.

```
def update_weights(self, dW1, db1, dW2, db2, epoch):
    if self.adaptive:
        # Momentum
        beta = 0.9
        self.v_W1 = beta * self.v_W1 - (1 - beta) * self.learning_rate * dW1
        self.v_b1 = beta * self.v_b1 - (1 - beta) * self.learning_rate * db1
        self.v_W2 = beta * self.v_W2 - (1 - beta) * self.learning_rate * dW2
        self.v_b2 = beta * self.v_b2 - (1 - beta) * self.learning_rate * db2

        self.W1 += self.v_W1
        self.b1 += self.v_b1
        self.W2 += self.v_W2
        self.b2 += self.v_b2
    else:
        # Gradient Descent
        self.W1 -= self.learning_rate * dW1
        self.b1 -= self.learning_rate * db1
        self.W2 -= self.learning_rate * dW2
        self.b2 -= self.learning_rate * db2
```

در این بخش روش بروزرسانی وزن ها و بایاس های شبکه ی عصبی پیاده سازی شده است. دو حالت در نظر گرفته شده:

- **گرادیان نزولی (Gradient Descent) :** وزن ها و بایاس ها مستقیماً با استفاده از گرادیان ها و نرخ یادگیری بروزرسانی میشوند. این روش ساده ترین شکل یادگیری است که در هر گام، وزن ها در جهت کاهش خطا تغییر میکنند.

- **روش مومنتوم (Momentum)** : علاوه بر گرادیان فعلی، بخشی از تغییرات گذشته نیز در نظر گرفته میشود. این کار باعث میشود حرکت وزن ها نرم تر و پایدارتر باشد و از نوسان شدید یا گیر افتادن در مینیمم های محلی جلوگیری شود. در این حالت متغیرهای سرعت (velocity) برای هر وزن و بایاس تعریف شده اند و بروزرسانی نهایی با استفاده از آنها انجام میشود.

```python
#Mean Squared Error
def mse_loss(self, y_pred, y_true):
    return np.mean((y_pred - y_true) ** 2)

def train(self, X_train, y_train, X_test, y_test, epochs=200):
    self.losses = []
    self.train_accuracies = []
    self.test_accuracies = []

    for epoch in range(epochs):
        # Forward
        y_pred_train = self.forward(X_train)
        loss = self.mse_loss(y_pred_train, y_train)
        self.losses.append(loss)

        # Backward
        dW1, db1, dW2, db2 = self.backward(X_train, y_train)

        # Update
        self.update_weights(dW1, db1, dW2, db2, epoch)

        # Accuracy
        train_mae = np.mean(np.abs(y_pred_train - y_train))
        y_pred_test = self.forward(X_test)
        test_mae = np.mean(np.abs(y_pred_test - y_test))

        self.train_accuracies.append(train_mae)
        self.test_accuracies.append(test_mae)

        if (epoch + 1) % 20 == 0:
            print(f"Epoch {epoch+1:3d}/{epochs} | Loss: {loss:.6f} | Train MAE: {train_mae:.6f} | Test MAE: {test_mae:.6f}")

def predict(self, X):
    return self.forward(X)
```

در این قسمت معیار خطا و روند آموزش شبکه ی عصبی پیادهسازی شده است.

- **تابع خطا (Loss Function):** از معیار **میانگین مربعات خطا (MSE)** استفاده شده است که اختلاف بین خروجی پیشبینی شده و مقدار واقعی را به صورت مربعی محاسبه و سپس میانگین می گیرد.
- **تابع آموزش (train)** :
  - در هر تکرار(epoch) ، ابتدا مرحلهی پیش رو (Forward) اجرا و خروجی شبکه محاسبه میشود.
  - سپس مقدار خطا با استفاده از MSE محاسبه و ذخیره میگردد.
  - در ادامه، الگوریتم پس انتشار خطا (Backward) اجرا شده و گرادیان ها به دست می آیند.
  - وزن ها و بایاس ها با توجه به گرادیان ها و نرخ یادگیری بروزرسانی میشوند.
  - برای ارزیابی عملکرد، معیار **MAE(میانگین قدر مطلق خطا)** روی داده های آموزشی و آزمایشی محاسبه و ثبت میشود.
  - هر ۲۰ تکرار، مقادیر خطا و دقت روی داده های آموزشی و آزمایشی چاپ میشوند تا روند یادگیری قابل مشاهده باشد.
- **تابع پیشبینی (predict)** : تنها مرحله ی پیشرو را اجرا کرده و خروجی شبکه را برای داده های جدید برمیگرداند.

```python
# آموزش مدل خطی
model_linear = SimpleNeuralNetwork(
    input_size=6,
    hidden_size=16,
    output_size=1,
    activation='linear',
    learning_rate=0.1,
    adaptive=False
)

model_linear.train(X_train, y_train, X_test, y_test, epochs=200)

# ارزیابی
y_pred_linear = model_linear.predict(X_test)
test_loss_linear = model_linear.mse_loss(y_pred_linear, y_test)
print(f"\n MSE: {test_loss_linear:.6f}")
print(f" MAE: {np.mean(np.abs(y_pred_linear - y_test)):.6f}")
```

در این مرحله یک شبکه‌ی عصبی ساده با **تابع فعال‌سازی خطی** طراحی و آموزش داده شد. (بدون استفاده از مومنتوم)

مدل به مدت ۲۰۰ تکرار (epoch) روی داده های آموزشی تمرین داده شد و سپس روی داده های آزمایشی مورد ارزیابی قرار گرفت. برای سنجش عملکرد، دو معیار خطا محاسبه شدند:

- **MSE (میانگین مربعات خطا) :**نشان دهنده‌ی میزان پراکندگی خطاها.
- **MAE (میانگین قدر مطلق خطا) :**بیانگر میانگین فاصله‌ی پیش بینی‌ها از مقادیر واقعی.

در پایان، مقادیر این معیارها چاپ شدند تا کیفیت پیش بینی مدل روی داده های آزمایشی مشخص شود.

```python
model_sigmoid = SimpleNeuralNetwork(
    input_size=6,
    hidden_size=16,
    output_size=1,
    activation='sigmoid',
    learning_rate=0.1,
    adaptive=False
)


model_sigmoid.train(X_train, y_train, X_test, y_test, epochs=200)

# ارزیابی
y_pred_sigmoid = model_sigmoid.predict(X_test)
test_loss_sigmoid = model_sigmoid.mse_loss(y_pred_sigmoid, y_test)
print(f"\n MSE: {test_loss_sigmoid:.6f}")
print(f" MAE: {np.mean(np.abs(y_pred_sigmoid - y_test)):.6f}")
```

در این مرحله یک شبکه عصبی ساده با **تابع فعال سازی سیگموئید** طراحی و آموزش داده شد. (بدون استفاده از مومنتوم)

مدل به مدت ۲۰۰ تکرار (epoch) روی داده های آموزشی تمرین داده شد. پس از آموزش، عملکرد مدل روی داده های آزمایشی ارزیابی گردید. برای سنجش کیفیت پیش بینی‌ها، دو معیار خطا محاسبه شدند:

- **MSE (میانگین مربعات خطا) :**نشان دهنده‌ی میزان پراکندگی خطاها.
- **MAE (میانگین قدر مطلق خطا) :**بیانگر میانگین فاصله‌ی پیش بینی‌ها از مقادیر واقعی.

در پایان، مقادیر این معیارها چاپ شدند تا عملکرد مدل با تابع فعال‌سازی سیگموئید قابل مقایسه با مدل خطی باشد.

جواب قسمت ب) در این آزمایش، دو مدل شبکه عصبی با ساختار مشابه اما با توابع فعال سازی متفاوت (سیگموئید و خطی) آموزش داده شدند. هدف، مقایسه روند یادگیری و کیفیت پیش بینی‌ها بود.

- مدل سیگموئید:(Sigmoid)
  - در ابتدای آموزش، خطا بسیار بالا بود.

- با ادامه آموزش، خطا به طور پیوسته کاهش یافت و در پایان به MSE = 0.0796 وMAE = 0.2123 رسید.
- روند کاهش خطا نشان دهنده همگرایی پایدار و توانایی مدل در یادگیری روابط غیرخطی داده هاست.

- مدل خطی:(Linear)
- از همان ابتدا خطا نسبتاً پایین‌تر بود.
- کاهش خطا سریع‌تر تثبیت شد و در پایان به MSE = 0.0827 وMAE = 0.2180رسید.
- اگرچه مدل خطی عملکرد مناسبی داشت، اما در مقایسه با مدل سیگموئید، خطای نهایی کمی بیشتر باقی ماند.

```
# مقایسه تصویری
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# نمودار 1: تابع هزینه
axes[0, 0].plot(model_linear.losses, label='Linear', linewidth=2, alpha=0.7)
axes[0, 0].plot(model_sigmoid.losses, label='Sigmoid', linewidth=2, alpha=0.7)
axes[0, 0].set_xlabel('Epoch')
axes[0, 0].set_ylabel('Loss (MSE)')
axes[0, 0].set_title('Cost function comparison')
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)

# نمودار 2: دقت آموزشی
axes[0, 1].plot(model_linear.train_accuracies, label='Linear (Train)', linewidth=2, alpha=0.7)
axes[0, 1].plot(model_sigmoid.train_accuracies, label='Sigmoid (Train)', linewidth=2, alpha=0.7)
axes[0, 1].set_xlabel('Epoch')
axes[0, 1].set_ylabel('Mean Absolute Error')
axes[0, 1].set_title('train accuracies')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# نمودار 3: دقت آزمایشی
axes[1, 0].plot(model_linear.test_accuracies, label='Linear (Test)', linewidth=2, alpha=0.7)
axes[1, 0].plot(model_sigmoid.test_accuracies, label='Sigmoid (Test)', linewidth=2, alpha=0.7)
axes[1, 0].set_xlabel('Epoch')
axes[1, 0].set_ylabel('Mean Absolute Error')
axes[1, 0].set_title('test accuracies')
axes[1, 0].legend()
axes[1, 0].grid(True, alpha=0.3)

# نمودار 4: مقایسه پیش‌بینی‌ها
axes[1, 1].scatter(y_test, y_pred_linear, alpha=0.6, label='Linear', s=50)
axes[1, 1].scatter(y_test, y_pred_sigmoid, alpha=0.6, label='Sigmoid', s=50)
axes[1, 1].plot([-2, 2], [-2, 2], 'r--', linewidth=2, label='Perfect Prediction')
axes[1, 1].set_xlabel('y_true')
axes[1, 1].set_ylabel('y_pred')
axes[1, 1].set_title('prediction comparison')
axes[1, 1].legend()
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

در این بخش، نتایج آموزش دو مدل مختلف به صورت تصویری مقایسه شده اند و چهار نمودار در یک پنجره‌ی ۲×۲ رسم شده است:

۱. **نمودار تابع هزینه(Loss Function Comparison) :**

- محور افقی: تعداد تکرارها(Epoch) .
- محور عمودی: مقدار خطا بر اساس معیارMSE .
- دو منحنی نشان دهنده روند کاهش خطا در مدل خطی و مدل سیگموئید هستند.
- این نمودار کمک میکند ببینیم کدام مدل سریع تر یا پایدارتر به خطای کمتر رسیده است.

۲. **نمودار دقت آموزشی (Train Accuracies) :**

- محور افقی: تعداد تکرارها.
- محور عمودی: مقدار MAE روی داده های آموزشی.
- این نمودار نشان میدهد مدل ها در طول آموزش چقدر توانسته اند خروجی‌های درست نزدیک به مقادیر واقعی تولید کنند.

۳. **نمودار دقت آزمایشی:(Test Accuracies)**

- مشابه نمودار قبلی، اما روی داده های آزمایشی محاسبه شده است.
- این نمودار برای بررسی **توان تعمیم (Generalization)** مدل ها اهمیت دارد. اگر خطا روی داده های آزمایشی خیلی بیشتر از آموزشی باشد، احتمالاً مدل دچار بیش برازش (Overfitting) شده است.

۴. **نمودار مقایسه‌ی پیش‌بینی‌ها(Prediction Comparison) :**

- محور افقی: مقادیر واقعی.{y_{true}
- محور عمودی: مقادیر پیش‌بینی‌شده.{y_{pred}
- نقاط آبی و نارنجی مربوط به پیش‌بینی‌های مدل خطی و سیگموئید هستند.
- خط قرمز نقطه چین نشان دهنده‌ی پیش بینی کامل (Perfect Prediction) است؛ یعنی جایی که خروجی دقیقاً برابر مقدار واقعی باشد.
- هرچه نقاط نزدیکتر به این خط باشند، عملکرد مدل بهتر است.

```
comparison_df = pd.DataFrame({
    'metric': [
        'Final Train Error',
        'MAE',
        'MSE',
        'train accuracies',
        'test accuracies'
    ],
    'Linear': [
        f"{test_loss_linear:.6f}",
        f"{np.mean(np.abs(y_pred_linear - y_test)):.6f}",
        f"{min(model_linear.losses):.6f}",
        f"{min(model_linear.train_accuracies):.6f}",
        f"{min(model_linear.test_accuracies):.6f}"
    ],
    'Sigmoid': [
        f"{test_loss_sigmoid:.6f}",
        f"{np.mean(np.abs(y_pred_sigmoid - y_test)):.6f}",
        f"{min(model_sigmoid.losses):.6f}",
        f"{min(model_sigmoid.train_accuracies):.6f}",
        f"{min(model_sigmoid.test_accuracies):.6f}"
    ]
})

print("\n" + comparison_df.to_string(index=False))
if test_loss_sigmoid < test_loss_linear:

    print(f" sigmoid function is better ")
else:
    print(f"\nlinear function is better")
```

در این قسمت نتایج دو مدل شبکه عصبی (با تابع فعال سازی خطی و سیگموئید) به صورت جدولی مقایسه شدند. جدول شامل معیارهای کلیدی عملکرد مانند خطای نهایی آموزش، MAE، MSE، بهترین دقت آموزشی و بهترین دقت آزمایشی است. مقادیر هر معیار برای هر دو مدل در ستون‌های جداگانه نمایش داده شدند تا امکان مقایسه مستقیم فراهم شود.

پس از ساخت جدول، مقدار خطای MSE روی داده های آزمایشی بررسی شد. بر اساس این معیار، مدلی که **مقدار MSE کمتری** داشته باشد به‌عنوان مدل **بهتر** انتخاب و نتیجه مقایسه به صورت متنی چاپ شد. این کار علاوه بر نمایش جزئیات عملکرد، یک جمع بندی ساده و قابل فهم از برتری یکی از مدل ها ارائه می‌دهد.

جواب قسمت ب) براساس نتایج به دست آمده، تابع فعالسازی Sigmoid عملکرد بهتری نسبت به تابع خطی داشته است .مهم ترین معیار، خطای آزمایشی (Test MAE) است که در آن Sigmoid با مقدار 0.212272 نسبت به Linear با مقدار 0.217958 برتری 2.6% داشته است. این نشان می دهد که مدل Sigmoid تعمیم دهی بهتری روی داده های جدید دارد . همچنین خطای نهایی آموزشی (Final Train Error) در Sigmoid کمتر است (0.079609 در مقابل 0.082653)، که نشان دهنده یادگیری بهتر الگوهای داده است .تنها در معیار MSE ، تابع Linear عملکرد بهتری داشت، اما این به دلیل حساسیت بیش از حد MSE به خطاهای بزرگ است . MAE معیار معتبرتری برای ارزیابی است.

نتیجه: تابع Sigmoid به دلیل ماهیت غیرخطی، توانایی بیشتری در مدل کردن روابط پیچیده بین ویژگی ها دارد و برای این دیتاست مناسب تر است.

```python
# خطای آموزشی
train_mae = min(model.train_accuracies)
# خطای آزمایشی
test_mae = min(model.test_accuracies)

print(f"\nBest Train MAE: {train_mae:.6f}")
print(f"Best Test MAE:  {test_mae:.6f}")
print(f"Gap (overfitting indicator): {abs(train_mae - test_mae):.6f}")

if abs(train_mae - test_mae) > 0.1:
    print("\n⚠️  WARNING: Large gap! Possible overfitting")
    print("   → Consider reducing learning rate")
else:
    print("\n✅ Gap is reasonable")
```

در این بخش عملکرد مدل از نظر خطای آموزشی و آزمایشی مورد بررسی قرار گرفت. برای این منظور، بهترین مقدار **MAE** در طول آموزش و بهترین مقدار **MAE** روی داده‌های آزمایشی محاسبه شد. سپس اختلاف بین این دو مقدار به‌عنوان شاخصی برای تشخیص **بیش برازش** در نظر گرفته شد.

اگر این اختلاف (Gap) بزرگ‌تر از ۰.۱ باشد، به عنوان نشانه ای از بیش برازش تلقی میشود. در چنین حالتی مدل روی داده های آموزشی عملکرد بسیار خوبی دارد اما روی داده‌های آزمایشی ضعیفتر عمل میکند. این وضعیت میتواند نشان دهنده‌ی یادگیری بیش از حد جزئیات داده های آموزشی باشد.

در صورتی که اختلاف کمتر یا مساوی ۰.۱ باشد، شکاف بین خطای آموزشی و آزمایشی معقول است و مدل توانسته تعمیم مناسبی روی داده های جدید داشته باشد.

```
# تأثیر تعداد نورون‌های لایه پنهان
hidden_sizes = [2, 4, 8, 16]
results_hidden = []

print("\n The Effect of N Neurons:")
for hs in hidden_sizes:
    model = SimpleNeuralNetwork(
        input_size=6, hidden_size=hs, output_size=1,
        activation='sigmoid', learning_rate=0.1
    )
    model.train(X_train, y_train, X_test, y_test, epochs=200)
    y_pred = model.predict(X_test)
    loss = model.mse_loss(y_pred, y_test)
    results_hidden.append(loss)
    print(f" Number of Neurons : {hs:2d} →  MSE: {loss:.6f}")

# تأثیر نرخ یادگیری
learning_rates = [0.001, 0.01, 0.05, 0.1]
results_lr = []

print("\n effect of learning rate:")
for lr in learning_rates:
    model = SimpleNeuralNetwork(
        input_size=6, hidden_size=16, output_size=1,
        activation='sigmoid', learning_rate=lr
    )
    model.train(X_train, y_train, X_test, y_test, epochs=200)
    y_pred = model.predict(X_test)
    loss = model.mse_loss(y_pred, y_test)
    results_lr.append(loss)
    print(f"learning rate: {lr:0.003f} → loss MSE: {loss:.6f}")
```

جواب قسمت پ) در این بخش، دو پارامتر مهم شبکه عصبی یعنی **تعداد نورون های لایه پنهان** و **نرخ یادگیری** مورد بررسی قرار گرفتند. هدف این آزمایش ها، تحلیل تأثیر این پارامترها بر کیفیت پیش بینی مدل و میزان خطا بود.

۱. **تأثیر تعداد نورون های لایه پنهان:**
   - مدل با تعداد نورون‌های مختلف (۲، ۴، ۸ و ۱۶) در لایه پنهان آموزش داده شد.
   - پس از ۲۰۰ تکرار، مقدار خطای MSE روی داده های آزمایشی محاسبه و ثبت گردید.
   - نتایج نشان میدهند که افزایش تعداد نورون ها معمولاً باعث افزایش ظرفیت مدل و توانایی یادگیری روابط پیچیده تر میشود. با این حال، تعداد زیاد نورون ها میتواند منجر به بیش برازش شود و عملکرد روی داده های آزمایشی را کاهش دهد.

۲. **تأثیر نرخ یادگیری:**

- مدل با نرخ های یادگیری مختلف (۰.۰۰۱، ۰.۰۱، ۰.۰۵ و ۰.۱) آموزش داده شد.

  پس از آموزش، مقدار خطای MSE روی داده های آزمایشی محاسبه شد.

- محلی میشود، در حالی که نرخ های بزرگتر (مانند ۰.۱) میتوانند سرعت یادگیری را افزایش دهند اما خطر نوسان و عدم همگرایی را نیز به همراه دارند. انتخاب نرخ یادگیری مناسب برای دستیابی به تعادل بین سرعت و پایداری اهمیت زیادی دارد.

```python
# کلاس شبکه با چند لایه پنهان
class MultiLayerNN:
    def __init__(self, layer_sizes, learning_rate=0.1):

        self.layer_sizes = layer_sizes
        self.learning_rate = learning_rate
        self.num_layers = len(layer_sizes) - 1

        # مقداردهی اولیه وزن‌ها و بایاس‌ها
        self.weights = []
        self.biases = []

        for i in range(self.num_layers):
            w = np.random.randn(layer_sizes[i], layer_sizes[i+1]) * 0.1
            b = np.zeros((1, layer_sizes[i+1]))
            self.weights.append(w)
            self.biases.append(b)

        # برای ذخیره نتایج
        self.losses = []
        self.test_accuracies = []

    def sigmoid(self, x):
        x = np.clip(x, -500, 500)
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        sig = self.sigmoid(x)
        return sig * (1 - sig)

    def forward(self, X):
        self.activations = [X]  # a0 = X
        self.z_values = []
```

این ادامه هم دارد که به صورت کامل در فایل اجرایی هست اما در اینجا فقط تیکه اول کد آمده است :

ادامه جواب قسمت پ)

در این آزمایش، سه ساختار مختلف شبکه عصبی چندلایه با تعداد متفاوت لایه های پنهان مورد بررسی قرار گرفتند. هدف، تحلیل تأثیر افزایش عمق شبکه بر کیفیت یادگیری و دقت پیش‌بینی ها بود.

- **مدل تک‌لایه (1 layer: [6, 8, 1])** :
این مدل با یک لایه پنهان توانست در طول آموزش به سرعت خطا را کاهش دهد. مقدار نهایی خطا برابر با **MSE = 0.0878** و **MAE = 0.2214** بود که نشان دهنده‌ی عملکرد مناسب و توانایی تعمیم قابل قبول روی داده های آزمایشی است.
- **مدل دولایه (2 layer: [6, 8, 4, 1])** :
این مدل با دو لایه پنهان نتوانست بهبود قابل توجهی نسبت به مدل ساده تر ایجاد کند. خطای نهایی آن برابر با **MSE = 1.0174** و **MAE = 0.8041** بود. مقادیر بالا نشان میدهند که مدل در همگرایی موفق نبوده و دچار ضعف در یادگیری روابط داده ها شده است.
- **مدل سه‌لایه (3 layer: [6, 8, 6, 4, 1])** :
افزایش تعداد لایه ها به سه لایه نیز نتیجه مشابهی داشت. خطای نهایی برابر با **MSE = 1.0229** و **MAE = 0.8057** بود که تقریباً مشابه مدل دولایه است و نشان دهنده عدم بهبود عملکرد با افزایش عمق شبکه در این مسئله می‌باشد.

```python
# نمودار تأثیر پارامترها
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# نمودار تعداد نورون‌ها
axes[0].bar(range(len(hidden_sizes)), results_hidden, alpha=0.7, color='steelblue')
axes[0].set_xticks(range(len(hidden_sizes)))
axes[0].set_xticklabels([str(hs) for hs in hidden_sizes])
axes[0].set_xlabel('Number of hidden layer neurons')
axes[0].set_ylabel('(MSE)')
axes[0].set_title('The Effect of N Neurons')
axes[0].grid(True, alpha=0.3, axis='y')

# نمودار نرخ یادگیری
axes[1].bar(range(len(learning_rates)), results_lr, alpha=0.7, color='coral')
axes[1].set_xticks(range(len(learning_rates)))
axes[1].set_xticklabels([str(lr) for lr in learning_rates])
axes[1].set_xlabel('Learning Rate')
axes[1].set_ylabel('(MSE)')
axes[1].set_title('Effect of Learning Rate')
axes[1].grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.show()
```

در این بخش، نتایج آزمایش های مربوط به **تعداد نورون های لایه پنهان** و **نرخ یادگیری** به صورت تصویری ارائه شده‌اند.
هدف از این نمودارها، نمایش مستقیم رابطه این پارامترها با میزان خطای مدل است.

۱. **نمودار تعداد نورون‌های لایه پنهان(The Effect of N Neurons) :**
   - محور افقی تعداد نورون های لایه پنهان را نشان میدهد (۲، ۴، ۸، ۱۶) .
   - محور عمودی مقدار خطای MSE روی داده های آزمایشی است.
   - هر ستون بیانگر عملکرد مدل با تعداد مشخصی نورون است.
   - این نمودار کمک میکند تا مشخص شود افزایش تعداد نورون ها چه تأثیری بر کاهش یا افزایش خطا دارد. معمولاً تعداد بیشتر نورون‌ها ظرفیت مدل را بالا میبرد، اما ممکن است باعث بیش برازش شود.

۲. **نمودار نرخ یادگیری(Effect of Learning Rate) :**

- محور افقی نرخ های یادگیری مختلف (۰.۰۰۱، ۰.۰۱، ۰.۰۵، ۰.۱) را نشان میدهد.
- محور عمودی مقدار خطای MSE روی داده های آزمایشی است.
- هر ستون بیانگر عملکرد مدل با نرخ یادگیری مشخص است.
- این نمودار نشان میدهد نرخ‌های خیلی کوچک باعث یادگیری کند و نرخ های خیلی بزرگ باعث نوسان یا عدم همگرایی میشوند. انتخاب نرخ یادگیری مناسب برای دستیابی به تعادل بین سرعت و پایداری اهمیت دارد.

```python
# تحلیل نتایج
best_hidden_idx = np.argmin(results_hidden)
best_lr_idx = np.argmin(results_lr)

print(f"The Best N of Neurons: {hidden_sizes[best_hidden_idx]} (Error: {results_hidden[best_hidden_idx]:.6f})")
print(f"The Best Learning Rate: {learning_rates[best_lr_idx]:.3f} (Error: {results_lr[best_lr_idx]:.6f})")
```

```
The Best N of Neurons: 16 (Error: 0.078949)
The Best Learning Rate: 0.100 (Error: 0.080356)
```

با استفاده از تابع np.argmin، کمترین مقدار خطا و شاخص متناظر آن استخراج میشود.

```python
# مدل با نرخ ثابت
print("\nModel with Fixed learning rate:")
model_fixed = SimpleNeuralNetwork(
    input_size=6, hidden_size=16, output_size=1,
    activation='sigmoid', learning_rate=0.1, adaptive=False
)
model_fixed.train(X_train, y_train, X_test, y_test, epochs=200)
y_pred_fixed = model_fixed.predict(X_test)
loss_fixed = model_fixed.mse_loss(y_pred_fixed, y_test)

# مدل با نرخ وفقی
print("\nModel with adaptive learning rate (Momentum):")
model_adaptive = SimpleNeuralNetwork(
    input_size=6, hidden_size=16, output_size=1,
    activation='sigmoid', learning_rate=0.1, adaptive=True
)
model_adaptive.train(X_train, y_train, X_test, y_test, epochs=200)
y_pred_adaptive = model_adaptive.predict(X_test)
loss_adaptive = model_adaptive.mse_loss(y_pred_adaptive, y_test)

print(f"- Fixed rate - MSE: {loss_fixed:.6f}")
print(f"- Adapted rate - MSE: {loss_adaptive:.6f}")
```

جواب قسمت ت) در این بخش، هدف بررسی تأثیر **روش بروزرسانی وزن ها** بر عملکرد شبکه عصبی است. برای این منظور، دو مدل با ساختار یکسان اما با روش های متفاوت آموزش داده شدند:

۱. **مدل با نرخ یادگیری ثابت(Fixed Learning Rate) :**
   - در این حالت، وزن ها و بایاس ها تنها بر اساس گرادیان فعلی و نرخ یادگیری ثابت (۰.۱) بروزرسانی میشوند.
   - این روش ساده ترین شکل گرادیان نزولی است و هر گام آموزشی مستقل از تاریخچه ی تغییرات قبلی انجام میشود.
   - پس از ۲۰۰ تکرار، مقدار خطای MSE روی داده های آزمایشی محاسبه شد.

۲. **مدل با نرخ یادگیری وفقی(Momentum) :**
   - در این حالت علاوه بر گرادیان فعلی، بخشی از تغییرات گذشته نیز در نظر گرفته میشود.
   - ضریب مومنتوم باعث میشود وزن ها با نوعی شتاب حرکت کنند و از نوسان شدید یا گیر افتادن در مینیمم های محلی جلوگیری شود.
   - پس از ۲۰۰ تکرار، مقدار خطای MSE روی داده های آزمایشی محاسبه شد.

برای این مسئله مدل با نرخ یادگیری ثابت عملکرد بهتری را نشان میدهد.

```
# مقایسه نمودار
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# نمودار 1: تابع هزینه
axes[0, 0].plot(model_fixed.losses, label='Fixed LR', linewidth=2, alpha=0.7)
axes[0, 0].plot(model_adaptive.losses, label='Adaptive LR (Momentum)', linewidth=2, alpha=0.7)
axes[0, 0].set_xlabel('Epoch')
axes[0, 0].set_ylabel('Loss (MSE)')
axes[0, 0].set_title('Cost function MSE comparison ')
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)

# نمودار 2: دقت آموزشی
axes[0, 1].plot(model_fixed.train_accuracies, label='Fixed LR', linewidth=2, alpha=0.7)
axes[0, 1].plot(model_adaptive.train_accuracies, label='Adaptive LR', linewidth=2, alpha=0.7)
axes[0, 1].set_xlabel('Epoch')
axes[0, 1].set_ylabel('MAE')
axes[0, 1].set_title('Training Error (MAE)')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# نمودار 3: دقت آزمایشی
axes[1, 0].plot(model_fixed.test_accuracies, label='Fixed LR', linewidth=2, alpha=0.7)
axes[1, 0].plot(model_adaptive.test_accuracies, label='Adaptive LR', linewidth=2, alpha=0.7)
axes[1, 0].set_xlabel('Epoch')
axes[1, 0].set_ylabel('MAE')
axes[1, 0].set_title('Test Error (MAE)')
axes[1, 0].legend()
axes[1, 0].grid(True, alpha=0.3)

# نمودار 4: مقایسه خطاها
methods = ['Fixed LR', 'Adaptive LR']
final_losses = [loss_fixed, loss_adaptive]
axes[1, 1].bar(methods, final_losses, alpha=0.7, color=['steelblue', 'coral'])
axes[1, 1].set_ylabel('(MSE)')
axes[1, 1].set_title('Final error comparison')
axes[1, 1].grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.show()
```

در این بخش، عملکرد دو مدل شبکه عصبی با **نرخ یادگیری ثابت** و **نرخ یادگیری وفقی (Momentum)** به صورت تصویری مقایسه شده است. چهار نمودار مختلف رسم شده‌اند:

۱. **نمودار تابع هزینه (MSE)** : روند کاهش خطا در طول تکرارها نشان داده شده است.
۲. **نمودار دقت آموزشی (MAE)** :تغییرات خطا آموزشی در طول زمان مقایسه شده است.
۳. **نمودار دقت آزمایشی (MAE)** :عملکرد مدل ها روی داده های آزمایشی بررسی شده است.
۴. **نمودار میله‌ای مقایسه نهایی خطاها** :مقادیر نهایی خطای MSE برای هر دو مدل نمایش داده شده است. این نمودار به طور مستقیم نشان میدهد کدام روش در پایان آموزش عملکرد بهتری داشته است که **اینجا مدل با نرخ یادگیری ثابت** عملکرد بهتری دارد.

بخش های اضافه شده یا تغییر داده شده مدل :

```
class SimpleNeuralNetwork:
    def __init__(self, input_size, hidden_size=4, output_size=1,
                 activation='linear', learning_rate=0.1,
                 adaptive=False, adaptive_lr=False, decay=0.01):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.activation = activation
        self.learning_rate = learning_rate
        self.adaptive = adaptive
        self.adaptive_lr = adaptive_lr
        self.decay = decay
```

این بخش از کد وظیفه مقداردهی اولیه کلاس شبکه عصبی را بر عهده دارد و پارامترهای کلیدی مانند تعداد ویژگی های ورودی، تعداد نورون های پنهان، نرخ یادگیری، و فعال بودن حالت‌های مومنتوم یا یادگیری وفقی (adaptive) را دریافت میکند.
با تعریف این پارامترها، انعطاف پذیری مدل برای تست سناریوهای مختلف مثل نرخ یادگیری ثابت یا متغیر و استفاده یا عدم استفاده از Momentumفراهم میشود.

```
  def update_weights(self, dW1, db1, dW2, db2, epoch, lr):
      if self.adaptive:
          beta = 0.9
          self.v_W1 = beta * self.v_W1 - (1 - beta) * lr * dW1
          self.v_b1 = beta * self.v_b1 - (1 - beta) * lr * db1
          self.v_W2 = beta * self.v_W2 - (1 - beta) * lr * dW2
          self.v_b2 = beta * self.v_b2 - (1 - beta) * lr * db2
          self.W1 += self.v_W1
          self.b1 += self.v_b1
          self.W2 += self.v_W2
          self.b2 += self.v_b2
      else:
          self.W1 -= lr * dW1
          self.b1 -= lr * db1
          self.W2 -= lr * dW2
          self.b2 -= lr * db2
```

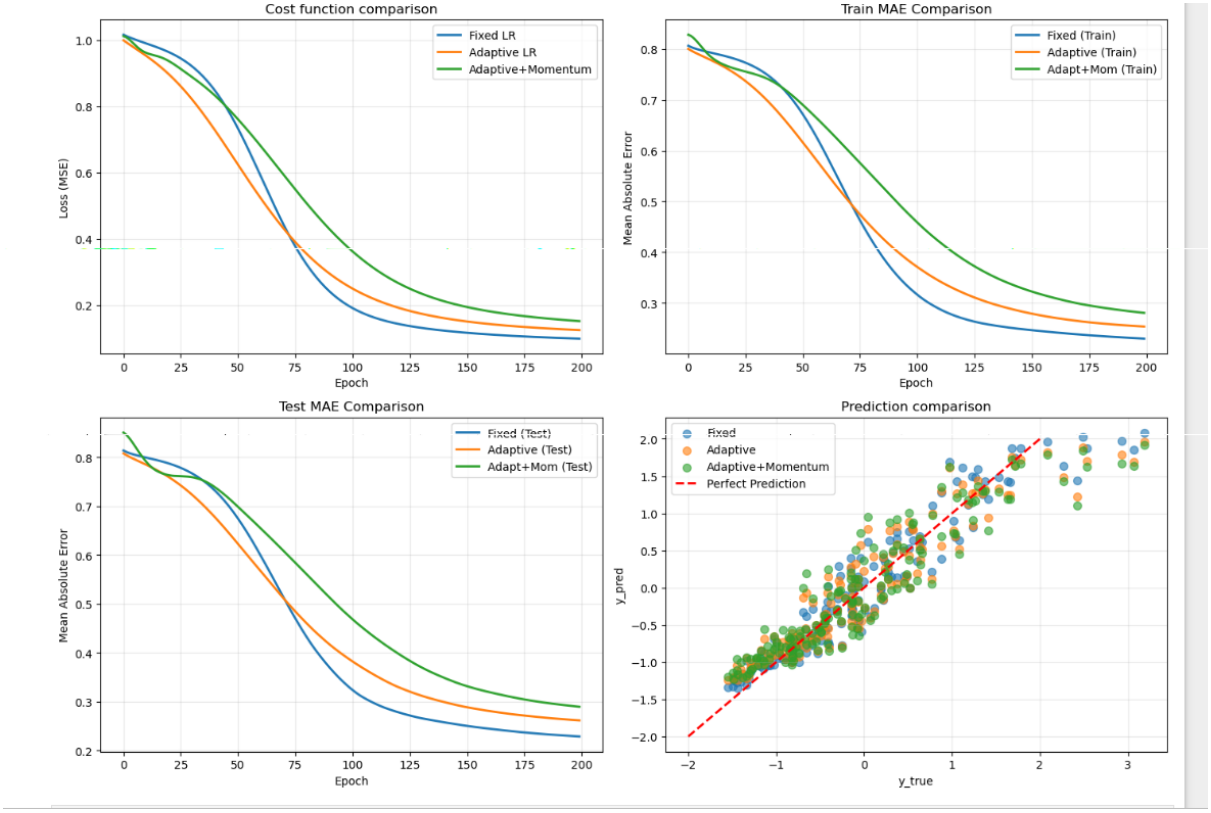
این تابع مسئول بروزرسانی وزن ها و بایاس های شبکه است.
در حالت adaptive=True، الگوریتم Momentum اجرا میشود که با استفاده از متغیرهای سرعت (v_W, v_b) و نرخ یادگیری فعلی، وزن ها را با شتاب بیشتری نسبت به گرادیان های قبلی اصلاح میکند.
در حالت معمولی(else) ، روش گرادیان نزولی ساده با نرخ یادگیری فعلی روی وزن ها اعمال میشود.

```
# آموزش مدل با حالت نرخ ثابت
model_fixed = SimpleNeuralNetwork(
    input_size=6, hidden_size=8, output_size=1,
    activation='sigmoid', learning_rate=0.1,
    adaptive=False, adaptive_lr=False
)
model_fixed.train(X_train, y_train, X_test, y_test, epochs=200)

# آموزش مدل با حالت فقط Adaptive LR
model_adaptive = SimpleNeuralNetwork(
    input_size=6, hidden_size=8, output_size=1,
    activation='sigmoid', learning_rate=0.1,
    adaptive=False, adaptive_lr=True, decay=0.01
)
model_adaptive.train(X_train, y_train, X_test, y_test, epochs=200)

# آموزش مدل با حالتی که هم Adaptive LR هم Momentum فعال است
model_adaptive_momentum = SimpleNeuralNetwork(
    input_size=6, hidden_size=8, output_size=1,
    activation='sigmoid', learning_rate=0.1,
    adaptive=True, adaptive_lr=True, decay=0.01
)
model_adaptive_momentum.train(X_train, y_train, X_test, y_test, epochs=200)
```

در این بخش از کد، سه حالت مختلف یادگیری برای شبکه عصبی تعریف و آموزش داده شده است:

- حالت اول با نرخ یادگیری ثابت (بدون adaptive یاmomentum)
- حالت دوم فقط با نرخ یادگیری وفقی (adaptive lr)
- حالت سوم با هردو: نرخ یادگیری وفقی و Momentum

این ساختار امکان مقایسه عملکرد مدل در شرایط مختلف بهینه سازی را فراهم میکند و نشان میدهد هرکدام چه اثری روی همگرایی و دقت شبکه دارند.



در نمودار Cost Function و MAE (Test/Train)، خط مدل Fixed LR (آبی) در همه مراحل به خصوص انتهای آموزش، پایین تر از مدل های دیگر قرار دارد و نشان میدهد همگرایی نهایی و دقت در این حالت بهتر بوده است.

مدل Adaptive LR ابتدا سریع تر کاهش پیدا میکند اما در ادامه میزان خطا کمی بالاتر از مدل ثابت باقی میماند و اضافه شدن Momentum هم این اختلاف را جبران نکرده، بلکه در انتها کمی بدتر شده است.

در نمودار پراکندگی پیش بینی ها، با وجود اینکه همه مدل ها روند کلی را خوب گرفته اند، نقاط مدل ثابت (آبی) پراکندگی کمتری نسبت به دو مدل دیگر دارد و بیشتر حول خط ایده‌آل y=x متمرکز هستند.