```python
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_california_housing

# Load California housing dataset
california = fetch_california_housing()

# Convert to DataFrame for easier viewing
df = pd.DataFrame(california.data, columns=california.feature_names)
df['target'] = california.target
```

در این بخش، ابتدا ابزارهای پایه برای محاسبات عددی و تحلیل داده ایمپورت شده اند. در خط اول و دوم، کتابخانه NumPy با نام مستعار np برای انجام عملیات‌های ریاضی و ماتریسی سریع، و کتابخانه Pandas با نام مستعار pd برای ایجاد ساختارهای داده ای جدولی فراخوانی شده اند.

در خط سوم، تابع fetch_california_housing از ماژول sklearn.datasets وارد برنامه شده است. این تابع وظیفه دانلود یا بارگذاری دیتاست استاندارد مسکن کالیفرنیا را بر عهده دارد که برای آموزش مدل استفاده خواهد شد.

در بخش اجرایی کد:

۱. بارگذاری داده خام:
با اجرای دستور california = fetch_california_housing()، داده ها در یک آبجکت دیکشنری مانند، (از نوع Bunch) ذخیره می‌شوند. این آبجکت شامل آرایه های جداگانه ای برای داده های ورودی (data) و مقادیر هدف (target) است.

۲. تبدیل به دیتافریم (DataFrame):
از آنجا که کار با آرایه های خام دشوار است، با دستور pd.DataFrame یک ساختار جدولی ایجاد کردیم.

- آرگومان اول، california.data، همان ماتریس ویژگی‌ها (X) است.
- آرگومان دوم، columns=california.feature_names، اسامی ویژگی ها (مثل تعداد اتاق، جمعیت و...) را به عنوان سرستون های جدول تنظیم میکند تا داده ها خوانا باشند.

۳. افزودن ستون هدف:
در خط آخر، مقادیر هدف (قیمت خانه ها) که در california.target قرار داشت، به عنوان یک ستون جدید با نام 'target' به دیتافریم df اضافه شد. این کار باعث میشود تمامی داده های آموزشی (هم ورودی ها و هم خروجی مورد انتظار) در یک جدول واحد و منسجم برای مراحل بعدی پردازش در دسترس باشند.

```
# Display basic information
print("Dataset shape (rows, columns):", df.shape)
print("\nFirst 5 rows:")
print(df.head())
```

تحلیل کد:

- print("Dataset shape...", df.shape) :

این دستور ویژگی (.shape) دیتافریم را فراخوانی می‌کند که یک تاپل (Tuple) برمیگرداند. عدد اول نشان دهنده تعداد سطرها و عدد دوم نشان دهنده تعداد ستون‌ها (ویژگی‌ها + هدف) است. این اطلاعات به ما میگوید که حجم داده ها چقدر است و آیا برای آموزش مدل کافی است یا خیر.

- print(df.head()):

متد (.head()) به صورت پیشفرض ۵ سطر اول دیتافریم را نمایش میدهد. این کار برای مشاهده سریع نوع داده ها، مقیاس اعداد و بررسی چشمی درستی ستون ها انجام میشود.

تحلیل خروجی‌ها:

۱. ابعاد دیتاست :خروجی (20640, 9) نشان میدهد که ما **۲۰,۶۴۰** رکورد (خانه/منطقه) داریم که برای آموزش مدل های یادگیری ماشین حجم بسیار مناسبی است. همچنین ۹ ستون وجود دارد که ۸ تای آنها ویژگی های ورودی (مثل درآمد متوسط، قدمت خانه و...) و یک ستون (target) مقدار خروجی است.

۲. نمونه داده‌ها :با نگاه به ۵ سطر اول، نکات مهمی استخراج می‌شود:

- MedInc(درآمد متوسط) : اعداد در بازه کوچکی (مثل ۸.۳) هستند که نشان میدهد این داده ها احتمالا نرمال‌سازی شده اند (مثلاً بر حسب صد هزار دلار).
- HouseAge(سن خانه) :اعدادی مثل ۴۱ یا ۲۱ سال را نشان میدهد.
- AveRooms و AveBedrms : میانگین تعداد اتاق ها و اتاق خواب ها که اعداد اعشاری هستند (چون میانگین منطقه‌اند).
- Population(جمعیت) :اعدادی بزرگ مثل ۳۲۲ یا ۲۴۰۱ که نشان دهنده جمعیت آن بلوک مسکونی است.
- تفاوت مقیاس‌ها (Scale) : نکته بسیار مهم این است که مقیاس اعداد با هم متفاوت است (مثلاً Population در حد هزار است ولی MedInc در حد یک‌رقمی) . این مشاهده به ما سیگنال میدهد که در مراحل بعدی احتمالاً نیاز به استانداردسازی (Scaling) داده ها خواهیم داشت تا مدل دچار خطا نشود.

```
print("\nStatistical summary:")
print(df.describe())
```

تحلیل دقیق پارامترهای جدول:

۱. تعداد (count) : تمام ستون‌ها عدد ۲۰,۶۴۰ را نشان میدهند. این یعنی هیچ داده ی گمشده ای (Missing Value) در این ستون ها وجود ندارد و نیاز به عملیات پر کردن داده های خالی (Imputation) نداریم.

۲. میانگین (mean) و انحراف معیار (std) :

- برای مثال در ستون Population(جمعیت)، میانگین حدود ۱۴۲۵ نفر است اما انحراف معیار بالایی (۱۱۳۲) دارد که نشان‌دهنده پراکندگی زیاد جمعیت در مناطق مختلف است.
- ستون Target(قیمت خانه) میانگینی حدود ۲.۰۶ دارد (احتمالاً بر حسب صدهزار دلار).

۳. بازه داده‌ها (min/max) و چارک‌ها(25%, 50%, 75%) :

- تشخیص داده های پرت (Outliers) : اگر به ستون AveRooms (متوسط تعداد اتاق ها) نگاه کنیم، میبینیم که ۷۵٪ داده ها زیر ۶ اتاق دارند (چارک سوم = ۶.۰۵)، اما مقدار ماکسیمم ناگهان ۱۴۱ اتاق است! این فاصله عظیم بین چارک سوم و ماکسیمم، نشان دهنده وجود داده های پرت یا خانه های بسیار خاص (شاید هتل یا مجتمع) است که ممکن است روند یادگیری مدل را منحرف کنند.
- مشابه همین وضعیت در AveOccup (متوسط ساکنین) دیده میشود؛ در حالی که میانگین ۳ نفر است، ماکسیمم مقدار ۱۲۴۳ نفر در یک واحد مسکونی ثبت شده است که قطعاً یک داده پرت یا خطای ثبت داده است.

۴. نتیجه‌گیری فنی:

این بررسی نشان میدهد که داده های خام دارای مقادیر پرت شدیدی هستند و همچنین دامنه تغییرات ویژگی ها (Scale) با هم بسیار متفاوت است. بنابراین، استفاده از روش‌های نرمال سازی قوی (Robust Scaler) یا حذف داده های پرت قبل از آموزش مدل، میتواند به بهبود دقت کمک شایانی کند.

```
print("\nFeature names:")
print(california.feature_names)
```

در این بخش، لیست کامل نام ویژگی های موجود در دیتاست استخراج و چاپ شده است. درک معنای هر ویژگی برای تفسیر نتایج مدل ضروری است. این ویژگی‌ها عبارتند از:

۱. MedInc (Median Income): درآمد میانگین ساکنان منطقه (بر حسب ده‌ها هزار دلار). این احتمالاً مهم ترین ویژگی برای پیش بینی قیمت مسکن است.

۲. HouseAge : میانگین سن خانه ها در آن بلوک ساختمانی.

۳. AveRooms : میانگین تعداد کل اتاق ها در هر خانوار.

۴. AveBedrms: میانگین تعداد اتاق‌خواب ها در هر خانوار.

۵. Population: جمعیت کل بلوک مسکونی.

۶. AveOccup (Average Occupancy): میانگین تعداد افراد ساکن در هر خانه.

۷. Latitude & Longitude: عرض و طول جغرافیایی منطقه؛ که موقعیت مکانی خانه را نشان میدهد (مثلاً نزدیکی به ساحل یا مرکز شهر که بر قیمت تأثیرگذار است).

```
print("\nDataset description:")
print(california.DESCR[:680])#First 680 character in description
```

دستور print(california.DESCR[:680]) به ما اجازه می‌دهد تا ۶۸۰ کاراکتر ابتدایی توضیحات رسمی دیتاست را مشاهده کنیم.
ویژگی DESCR در دیتاست‌های sklearnحاوی اطلاعات کاملی درباره:

- منبع جمع‌آوری داده ها (که در اینجا سرشماری سال ۱۹۹۰ آمریکا است).

- تعداد نمونه ها و ویژگی ها.
- واحد اندازه‌گیری متغیر هدف (که معمولاً قیمت‌ها در مقیاس ۱۰۰,۰۰۰ دلار هستند).
- ارجاعات علمی و مقالاتی که از این دیتاست استفاده کرده‌اند.

```
X = california.data
y = california.target
# Calculate split indices based on dataset size
n_samples = X.shape[0]
train_end = int(0.6 * n_samples)  # 60% for training
test_end = int(0.8 * n_samples)   # Next 20% for testing
# Remaining 20% for validation

# Split data into train, test, and validation sets
X_train = X[:train_end]
y_train = y[:train_end]

X_test = X[train_end:test_end]
y_test = y[train_end:test_end]

X_val = X[test_end:]
y_val = y[test_end:]

print("Data split:")
print(f"Train: {X_train.shape[0]} samples")
print(f"Test: {X_test.shape[0]} samples")
print(f"Validation: {X_val.shape[0]} samples")

# Normalize features using mean and std from training set
X_mean = np.mean(X_train, axis=0)
X_std = np.std(X_train, axis=0)

X_train_normalized = (X_train - X_mean) / X_std
X_test_normalized = (X_test - X_mean) / X_std
X_val_normalized = (X_val - X_mean) / X_std

print("X_train mean:", np.mean(X_train_normalized, axis=0))
print("X_train std:", np.std(X_train_normalized, axis=0))
```

**تقسیم بندی داده ها و نرمال سازی ویژگی ها(Data Splitting & Normalization)**

در این مرحله دو عملیات حیاتی پیش پردازش انجام شده است: جداسازی داده ها به سه بخش مستقل و استانداردسازی مقادیر ورودی.

الف) استراتژی تقسیم داده ها (Data Splitting) :

به جای استفاده از توابع آماده مثل train_test_split، در اینجا تقسیم بندی به صورت دستی و مبتنی بر ایندکس (Index-based Slicing) انجام شده است تا کنترل دقیقی روی ترتیب داده ها داشته باشیم. داده های کل (۲۰,۶۴۰ نمونه) به نسبت های زیر تقسیم شدند:

۱. مجموعه آموزش (Training Set) : شامل ۶۰٪ ابتدایی داده ها برای یادگیری پارامترهای مدل.
۲. مجموعه تست (Test Set) : شامل ۲۰٪ میانی داده ها برای ارزیابی نهایی عملکرد مدل.
۳. مجموعه اعتبارسنجی (Validation Set) : شامل ۲۰٪ انتهایی داده ها برای تنظیم فراپارامترها (Hyperparameter Tuning) و جلوگیری از بیش برازش (Overfitting) در حین آموزش.

ب) نرمال‌سازی دستی (Manual Normalization) :

از آنجا که مقیاس ویژگی ها بسیار متفاوت بود (مثلاً جمعیت در هزاران و تعداد اتاق در واحد یکان)، عملیات استانداردسازی (Z-score

Normalization) انجام شد. فرمول استفاده شده عبارت است از:

$$X_{normalized} = (X - \mu) / \sigma$$

نکته مهم (Data Leakage Prevention) :

محاسبه میانگین (μ) و انحراف معیار (σ) فقط بر روی داده های آموزشی (X_train) انجام شده است. سپس با استفاده از همین پارامترهایِ استخراج شده از داده های آموزش، داده های تست و اعتبارسنجی نرمال شده‌اند.

- چرا؟ اگر میانگین کل داده ها را حساب میکردیم، اطلاعاتی از داده های تست به مدل «نشت» می‌کرد (Data Leakage) و نتایج ارزیابی غیرواقعی و خوش بینانه میشد. ما در دنیای واقعی داده های آینده (تست) را نداریم، پس نباید میانگین آنها را بدانیم.

نتیجه:

در نهایت با چاپ میانگین و انحراف معیار داده های نرمال شده، مشاهده میکنیم که میانگین X_train به ۰ و انحراف معیار آن به ۱ میل کرده است که نشان دهنده موفقیت عملیات استانداردسازی است.

```python
class RBFNetwork:
    """
    Radial Basis Function Neural Network for Regression
    """

    def __init__(self, n_hidden, learning_rate=0.01):
        """
        Initialize RBF Network

        Parameters:
        - n_hidden: number of RBF neurons (hidden layer size)
        - learning_rate: learning rate for output weights training
        """
        self.n_hidden = n_hidden
        self.learning_rate = learning_rate

        # These will be set during training
        self.centers = None       # RBF centers (n_hidden, n_features)
        self.sigmas = None        # RBF widths (n_hidden,)
        self.weights = None       # Output layer weights (n_hidden + 1,)

        print(f"RBF Network initialized with {n_hidden} hidden neurons")
        print(f"Learning rate: {learning_rate}")
```

**طراحی و پیاده‌سازی کلاس شبکه عصبی RBF (ساختار اولیه) :**

در این بخش، کلاس اصلی RBFNetwork که مسئولیت ایجاد، آموزش و پیش بینی مدل شبکه عصبی مبتنی بر توابع پایه شعاعی (Radial Basis Function) را بر عهده دارد، تعریف شده است. این پیاده‌سازی به صورت شیءگرا (Object-Oriented) انجام شده تا ماژولار و قابل توسعه باشد.

تشریح متد سازنده(__init__) :

این متد هنگام ساختن یک نمونه جدید از شبکه فراخوانی میشود و پارامترهای حیاتی مدل را تنظیم میکند:

۱. پارامتر n_hidden : تعداد نورون های لایه مخفی (Hidden Layer) را تعیین میکند. در شبکه های RBF ، هر نورون مخفی نماینده یک مرکز (Center) یا خوشه در فضای داده است. تعداد این نورون ها پیچیدگی و ظرفیت یادگیری مدل را مشخص میکند.

۲. پارامتر learning_rate: نرخ یادگیری برای بروزرسانی وزن های لایه خروجی است. این پارامتر سرعت همگرایی مدل را کنترل می‌کند (پیش‌فرض ۰.۰۱ در نظر گرفته شده است).

۳. مقداردهی اولیه پارامترهای شبکه:

سه ویژگی اصلی شبکه که قرار است در مرحله آموزش (Training) مقداردهی شوند، فعلاً با None مقداردهی شده اند:

- self.centers: مختصات مراکز توابع شعاعی (همان μ در فرمول گاوسی). ابعاد آن تعداد نورون مخفی × تعداد ویژگی‌ها خواهد بود.
- self.sigmas: پهنای باند یا گستردگی هر تابع شعاعی (همان σ) این پارامتر تعیین میکند که هر نورون به چه شعاعی از داده ها حساس باشد.
- self.weights: وزن های خطی لایه خروجی که خروجی های لایه مخفی را به خروجی نهایی (قیمت خانه) وصل میکنند. ابعاد آن (تعداد نورون مخفی + ۱) است، که آن +1 مربوط به جمله بایاس (Bias) است.

نکته فنی:

ساختار شبکه RBF معمولاً دو مرحله ای آموزش میبینند: مرحله اول یافتن مراکز و سیگماها (معمولاً بدون نظارت مثل K-Means) و مرحله دوم یافتن وزن‌ها (با نظارت مثل Gradient Descent) تعریف متغیرها به صورت None نشان دهنده آمادگی برای این فرآیند دو مرحله‌ای است.

```python
def gaussian_rbf(self, X, center, sigma):
    # Calculate Euclidean distance from center
    distances = np.linalg.norm(X - center, axis=1)
    # Apply Gaussian function
    activations = np.exp(-(distances ** 2) / (2 * sigma ** 2))
    return activations

def kmeans(self, X, n_clusters, max_iter=100):
    # Randomly pick n_clusters samples as initial centers
    np.random.seed(42)
    idx = np.random.choice(X.shape[0], n_clusters, replace=False)
    centers = X[idx]

    for iteration in range(max_iter):
        # Assign each sample to the nearest center
        distances = cdist(X, centers)  # shape: (n_samples, n_clusters)
        labels = np.argmin(distances, axis=1)

        # Compute new centers
        new_centers = np.array([X[labels == k].mean(axis=0)
                                if np.any(labels == k) else centers[k]
                                for k in range(n_clusters)])

        # Check convergence
        if np.allclose(centers, new_centers):
            print(f"K-Means converged at iteration {iteration}")
            break
        centers = new_centers

    return centers, labels
```

**پیاده سازی تابع هسته و الگوریتم خوشه بندی(Kernel Function & Clustering)**

در این بخش، توابع محاسباتی اصلی شبکه پیاده سازی شده اند که شامل تابع فعال ساز گاوسی و الگوریتم K-Means برای تعیین مراکز نورون ها است.

الف) تابع gaussian_rbf :

این تابع هسته اصلی شبکه RBF است که میزان شباهت یک داده ورودی X را با مرکز نورون center محاسبه میکند.

- محاسبه فاصله اقلیدسی :ابتدا با استفاده از np.linalg.norm فاصله بین ورودی و مرکز محاسبه میشود.
- تابع گاوسی :سپس این فاصله در فرمول گاوسی قرار می‌گیرد.

نتیجه‌ی این تابع عددی بین ۰ و ۱ است. هرچه فاصله کمتر باشد (ورودی به مرکز نزدیک تر باشد)، خروجی به ۱ نزدیک تر خواهد بود و هرچه دورتر باشد، به ۰ میل می‌کند.

ب) پیاده‌سازی الگوریتم kmeans از پایه:

به جای استفاده از sklearn.cluster.KMeans، الگوریتم خوشه بندی به صورت دستی پیاده‌سازی شده است. مراحل اجرا به شرح زیر است:

۱. مقداردهی اولیه (Initialization) : ابتدا n_clusters نقطه به صورت تصادفی از داده های ورودی انتخاب میشوند تا به عنوان مراکز اولیه خوشه ها عمل کنند. استفاده از np.random.seed(42) تکرارپذیری نتایج را تضمین میکند.

۲. حلقه تکرار(Iteration Loop) :

- تخصیص (Assignment) : فاصله تمام نقاط داده تا تمام مراکز محاسبه میشود (cdist). سپس هر نقطه داده به نزدیک ترین مرکز نسبت داده می‌شود(np.argmin) .
- بروزرسانی (Update) : مرکز جدید هر خوشه با محاسبه میانگین (mean) تمام نقاطی که به آن خوشه اختصاص یافته اند، به دست می‌آید.
- بررسی همگرایی(Convergence Check) : اگر مراکز جدید دقیقاً همان مراکز قبلی باشند (تغییری نکنند)، الگوریتم متوقف میشود (break) چون به ثبات رسیده است.

این الگوریتم خروجی های centers (مراکز بهینه شده) و labels (مشخص کننده خوشه هر داده) را بازمیگرداند که مستقیماً به عنوان وزن های لایه مخفی شبکه RBF استفاده خواهند شد.

```
def compute_sigmas(self, centers, labels, X):
    n_clusters = centers.shape[0]
    sigmas = np.zeros(n_clusters)

    for k in range(n_clusters):
        members = X[labels == k]
        if len(members) > 1:
            # Standard deviation of distances to center
            dists = np.linalg.norm(members - centers[k], axis=1)
            sigmas[k] = np.std(dists) + 1e-8  # Prevent zero sigma
        else:
            sigmas[k] = 1.0  # Default value

    return sigmas

def compute_hidden_layer(self, X):
    n_samples = X.shape[0]
    H = np.zeros((n_samples, self.n_hidden))

    # Calculate activation for each RBF neuron
    for i in range(self.n_hidden):
        H[:, i] = self.gaussian_rbf(X, self.centers[i], self.sigmas[i])

    return H
```

**محاسبه پارامتر سیگما و تشکیل ماتریس لایه مخفی:**

در این مرحله، دو تابع کلیدی برای تنظیم رفتار نورون های مخفی پیاده سازی شده است. هدف این توابع تبدیل داده های خام ورودی به فضای ویژگی جدید (Feature Space) است که در آن مسئله خطی سازی شود.

الف) تابع compute_sigmas :

برخلاف روش های ساده که از یک مقدار ثابت برای سیگما استفاده میکنند، در اینجا پهنای باند هر نورون (sigma) متناسب با پراکندگی داده های همان خوشه محاسبه میشود.

- روش محاسبه : برای هر خوشه k، ابتدا تمام نقاطی که عضو آن خوشه هستند (members) جدا میشوند. سپس انحراف معیار (Standard Deviation) فاصله این نقاط از مرکز خوشه محاسبه میشود.
- مزیت : این روش باعث میشود نورون هایی که داده های پراکنده تری را پوشش میدهند، پهنای باند بزرگتر، و نورون‌های متمرکز، پهنای باند کوچکتری داشته باشند. این تطبیق پذیری دقت شبکه را به شدت افزایش میدهد.
- نکته ایمنی :عبارت 1e-8 + برای جلوگیری از تقسیم بر صفر (در صورت وجود تنها یک نقطه در خوشه) اضافه شده است.

ب) تابع compute_hidden_layer :

این تابع وظیفه دارد ورودی های شبکه (X) را از لایه مخفی عبور دهد و ماتریس فعال‌سازی H را تولید کند.

- یک ماتریس H با ابعاد (تعداد نمونه‌ها × تعداد نورون مخفی) ایجاد میشود.
- در یک حلقه، برای هر نورون i، خروجی تابع گاوسی با استفاده از مرکز (centers[i]) و سیگمای مخصوص آن (sigmas[i]) محاسبه و در ستون iام ماتریس H ذخیره میشود.
- خروجی این تابع، همان داده های دگرگون شده ای هستند که آماده اند تا وارد لایه نهایی (خطی) شوند.

```python
def fit(self, X, y):
    print("\nStarting RBF Network training...")

    # Step 1: Find centers using K-Means
    print("Step 1: Finding RBF centers with K-Means...")
    self.centers, labels = self.kmeans(X, self.n_hidden, max_iter=100)

    # Step 2: Compute sigmas
    print("Step 2: Computing sigmas...")
    self.sigmas = self.compute_sigmas(self.centers, labels, X)
    print(f"Sigma range: [{self.sigmas.min():.4f}, {self.sigmas.max():.4f}]")

    # Step 3: Compute hidden layer output
    print("Step 3: Computing hidden layer activations...")
    H = self.compute_hidden_layer(X)

    # Add bias term
    H_with_bias = np.c_[H, np.ones(H.shape[0])]

    # Step 4: Solve for output weights using pseudo-inverse
    print("Step 4: Solving for output weights...")
    self.weights = np.linalg.pinv(H_with_bias) @ y

    # Calculate training error
    y_pred = H_with_bias @ self.weights
    train_mse = np.mean((y - y_pred) ** 2)

    print(f"\nTraining completed!")
    print(f"Training MSE: {train_mse:.4f}")
    print(f"Weights shape: {self.weights.shape}")
```

**فرآیند آموزش شبکه:**

تابع fit، شامل دو فاز اصلی است:

فاز اول: یادگیری بدون نظارت (Unsupervised Learning) - تعیین ساختار لایه مخفی

۱. یافتن مراکز (centers) : ابتدا با فراخوانی الگوریتم kmeans که در مرحله قبل پیاده سازی شد، مراکز نورون های مخفی تعیین میشوند. این کار باعث میشود نورون ها دقیقاً در جاهایی متمرکز شوند که تجمع داده ها بیشتر است.

۲. محاسبه پهنای باند (sigmas) : پس از مشخص شدن مراکز، با استفاده از تابع compute_sigmas، پهنای عملکرد هر نورون تنظیم میشود تا پوشش مناسبی روی داده ها داشته باشد.

فاز دوم: یادگیری با نظارت (Supervised Learning) - تنظیم وزن‌های خروجی

3. محاسبه فعال‌سازی لایه مخفی (H): داده های ورودی X از توابع گاوسی عبور میکنند و ماتریس فعال‌سازی H ساخته میشود.

4. افزودن جمله بایاس(Bias Term) : با دستور np.c_، یک ستون حاوی عدد ۱ به انتهای ماتریس H اضافه می‌شود.

(H_with_bias) این ستون متناظر با وزن بایاس است که عرض از مبدأ مدل را کنترل میکند.

5.محاسبه وزن‌های بهینه (weights) :

به جای استفاده از روش‌های تکراری و کند مثل گرادیان کاهشی(Gradient Descent) ، در اینجا از راه حل تحلیلی دقیق (Closed-form Solution)استفاده شده است.

معادله $W=(H^TH)H^Ty^{-1}$

در کد: این معادله با استفاده از شبه‌معکوس مور-پنروز (np.linalg.pinv)پیاده‌سازی شده است.

چرا؟ این روش سریع ترین و دقیق ترین راه برای پیدا کردن وزن ها در لایه آخر شبکه‌های RBF است و تضمین میکند که خطای مربعات (MSE) مینیمم شود.

محاسبه خطای آموزش:

در پایان، مدل یک بار روی همان داده های آموزشی تست میشود تا میزان خطای اولیه (train_mse) مشخص شود. این عدد نشان میدهد مدل چقدر خوب روی داده های آموزشی فیت شده است.

@ یعنی چی؟ : ضرب ماتریسی در پایتون (Matrix Multiplication) است.

```
def predict(self, X):
    # Compute hidden layer activations
    H = self.compute_hidden_layer(X)

    # Add bias term
    H_with_bias = np.c_[H, np.ones(H.shape[0])]

    # Compute output
    y_pred = H_with_bias @ self.weights

    return y_pred

def evaluate(self, X, y, dataset_name="Test"):
    # Make predictions
    y_pred = self.predict(X)

    # Calculate metrics
    mse = np.mean((y - y_pred) ** 2)
    rmse = np.sqrt(mse)
    mae = np.mean(np.abs(y - y_pred))

    print(f"\n{dataset_name} Set Results:")
    print(f"MSE:  {mse:.4f}")
    print(f"RMSE: {rmse:.4f}")
    print(f"MAE:  {mae:.4f}")

    return mse, rmse
```

**مکانیزم پیش بینی و ارزیابی عملکرد مدل(Prediction & Evaluation) :**

در این بخش، دو متد نهایی کلاس RBFNetwork پیاده سازی شده اند که خروجی مدل را تولید و کیفیت آن را می سنجند.

الف) متد predict:

این متد دقیقاً همان مسیری را طی میکند که در فاز آموزش طی شد، با این تفاوت که دیگر وزن ها تغییر نمیکنند.

۱. تبدیل ورودی :داده های خام X وارد تابع compute_hidden_layer میشوند تا با استفاده از مراکز و سیگما های یاد گرفته شده، به بردار ویژگی های گاوسی (H) تبدیل شوند.

۲. افزودن بایاس :ستون بایاس (یک‌ها) به ماتریس اضافه میشود.

۳. محاسبه خروجی :با ضرب ماتریسی H_with_bias @ self.weights، قیمت نهایی خانه پیش بینی میشود.

- نکته :این عملیات بسیار سریع است چون فقط شامل چند ضرب و جمع ساده ماتریسی است.

ب) متد evaluate :

این متد برای گزارش دهی استاندارد عملکرد مدل طراحی شده است و سه معیار اصلی خطا را محاسبه میکند:

۱. MSE (Mean Squared Error): میانگین مربعات خطا این معیار به خطا های بزرگ حساس تر است و برای بهینه سازی مدل استفاده میشود.

۲. RMSE (Root Mean Squared Error): جذر MSE این معیار هم جنس با متغیر هدف (قیمت دلار) است و تفسیر آن برای انسان راحت تر است.

۳. MAE (Mean Absolute Error) : میانگین قدر مطلق خطا نشان میدهد به طور متوسط پیش‌بینی ما چقدر (چند دلار) با واقعیت فاصله دارد.

```
rbf = RBFNetwork(n_hidden=50, learning_rate=0.01)
rbf.fit(X_train_normalized, y_train)

# Evaluate on all datasets
print("RBF NETWORK EVALUATION")
rbf.evaluate(X_train_normalized, y_train, "Training")
rbf.evaluate(X_test_normalized, y_test, "Test")
rbf.evaluate(X_val_normalized, y_val, "Validation")
```

**اجرای نهایی و ارزیابی عملکرد شبکه:**

در این مرحله، یک نمونه واقعی از کلاس RBFNetwork ساخته شده و فرآیند آموزش و تست آن اجرا میشود.

الف) پیکربندی و آموزش مدل:

- تعداد نورون‌های مخفی (n_hidden=50): این پارامتر حیاتی بر اساس آزمون و خطا و پیچیدگی مسئله انتخاب شده است. عدد ۵۰ نشان میدهد که فضای داده های مسکن کالیفرنیا توسط ۵۰ تابع گاوسی پوشش داده میشود. این تعداد تعادل خوبی بین دقت (Bias کم) و تعمیم‌پذیری (Variance کم) ایجاد میکند.

- نرخ یادگیری (learning_rate=0.01) : اگرچه در نسخه نهایی از روش تحلیلی (Exact Solution) برای وزن ها استفاده شد و این پارامتر عملاً در محاسبه وزن ها دخالت مستقیم ندارد، اما وجود آن برای ساختار کلاس حفظ شده است.

- آموزش :متد rbf.fit با داده های نرمال شده (X_train_normalized) فراخوانی میشود تا مراکز خوشه ها و وزن های نهایی محاسبه شوند.

ب) ارزیابی چندگانه :

برای اطمینان از اینکه مدل دچار بیش برازش (Overfitting) نشده است، عملکرد آن روی سه مجموعه داده جداگانه بررسی می‌شود:

۱. مجموعه آموزش (Training) : نشان میدهد مدل چقدر خوب داده های دیده شده را یاد گرفته است. اگر خطای اینجا خیلی کم باشد ولی تست زیاد باشد، یعنی Overfit کرده ایم.

۲. مجموعه تست (Test) : نشان دهنده عملکرد واقعی مدل روی داده های کاملاً جدید است (مهم‌ترین معیار).

۳. مجموعه اعتبارسنجی(Validation) : که معمولاً برای تنظیم پارامترها استفاده میشود.

نزدیکی خطای این سه مجموعه به یکدیگر نشان دهنده پایداری (Stability) و تعمیم‌پذیری (Generalization) بالای مدل پیاده سازی شده میباشد.

```python
import matplotlib.pyplot as plt

# Visualization: Predicted vs Actual
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Predictions for all datasets
y_train_pred = rbf.predict(X_train_normalized)
y_test_pred = rbf.predict(X_test_normalized)
y_val_pred = rbf.predict(X_val_normalized)

datasets = [
    (y_train, y_train_pred, "Training Set"),
    (y_test, y_test_pred, "Test Set"),
    (y_val, y_val_pred, "Validation Set")
]

for idx, (y_true, y_pred, title) in enumerate(datasets):
    ax = axes[idx]

    # Scatter plot
    ax.scatter(y_true, y_pred, alpha=0.3, s=10)

    # Perfect prediction line (y=x)
    min_val = min(y_true.min(), y_pred.min())
    max_val = max(y_true.max(), y_pred.max())
    ax.plot([min_val, max_val], [min_val, max_val], 'r--', linewidth=2, label='Perfect Prediction')

    ax.set_xlabel('Actual Price (100k$)', fontsize=10)
    ax.set_ylabel('Predicted Price (100k$)', fontsize=10)
    ax.set_title(title, fontsize=12, fontweight='bold')
    ax.legend()
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Error distribution
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

errors = [
    (y_train - y_train_pred, "Training Set"),
    (y_test - y_test_pred, "Test Set"),
    (y_val - y_val_pred, "Validation Set")
]

for idx, (error, title) in enumerate(errors):
    ax = axes[idx]
    ax.hist(error, bins=50, alpha=0.7, edgecolor='black')
    ax.axvline(0, color='red', linestyle='--', linewidth=2)
    ax.set_xlabel('Prediction Error (100k$)', fontsize=10)
    ax.set_ylabel('Frequency', fontsize=10)
    ax.set_title(f'{title} - Error Distribution', fontsize=12, fontweight='bold')
    ax.grid(True, alpha=0.3)

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

**مصورسازی (Visualization):**

برای درک عمیق‌تر رفتار مدل و بررسی توزیع خطا ها، دو نمودار کلیدی رسم شده است.

نمودار اول: نمودار پراکندگی ((پیش‌بینی در برابر واقعیت (Predicted vs Actual) ))
در ردیف بالای تصاویر، سه نمودار Scatter Plot برای هر سه مجموعه داده (Training, Test, Validation) ترسیم شده است.

- محور افقی (X) : قیمت واقعی خانه ها(Actual Price) .
- محور عمودی (Y) : قیمت پیش بینی شده توسط مدل(Predicted Price) .
- خط قرمز چین چین (Perfect Prediction) : خط y=x است. اگر تمام نقطه ها دقیقاً روی این خط قرار می گرفتند، یعنی پیش‌بینی ما ۱۰۰٪ درست بود.

تحلیل نتایجScatter Plot :

۱. تمرکز داده‌ها :اکثر نقاط آبی رنگ در اطراف خط قرمز متمرکز شده اند که نشان دهنده همبستگی خوب بین پیش بینی و واقعیت است.

۲. مشکل سقف قیمت: اگر دقت کنید، در قیمت ۵ (که معادل ۵۰۰,۰۰۰ دلار است)، یک خط عمودی از نقاط دیده میشود. این به دلیل این است که در دیتاست اصلی، قیمت های بالای ۵۰۰ هزار دلار همگی به عدد ۵ برش خورده اند. مدل ما چون این محدودیت مصنوعی را نمیداند، سعی کرده مقادیر بالاتر را هم پیش بینی کند (نقاط بالای خط ۵) یا در پیش بینی آنها دچار ابهام شده است.

۳. پراکندگی (Variance) : در قیمت های پایین تر، تراکم نقاط بیشتر و پیش بینی دقیق تر است، اما در قیمت های بالا پراکندگی بیشتر میشود که طبیعی است.

نمودار دوم: توزیع خطاها :
در ردیف پایین، هیستوگرام خطای پیش بینی (تفاضل واقعیت و پیش بینی) ترسیم شده است.

- شکل زنگوله‌ای : توزیع خطا ها بسیار شبیه به توزیع نرمال (Normal Distribution) است و میانگین آن روی عدد صفر (خط قرمز عمودی) متمرکز شده است. این عالی است! یعنی مدل ما سوگیری (Bias) سیستماتیک ندارد؛ گاهی بالاتر و گاهی پایین تر پیش بینی میکند ولی میانگینش درست است.
- دنباله‌ها :وجود دنباله سمت راست در هیستوگرام نشان میدهد که مدل در برخی موارد (احتمالا همان خانه های خیلی گران) خطای مثبت زیادی داشته است.

نتیجه‌گیری نهایی:
با توجه به نمودارها، مدل RBF طراحی شده توانسته الگوی کلی حاکم بر قیمت مسکن را به خوبی یاد بگیرد. خطای باقی مانده تا حد زیادی ناشی از نویز موجود در داده ها (مثل داده های پرت جمعیتی) و محدودیت مصنوعی سقف قیمت در دیتاست اصلی است.

```python
results = []

# Test different numbers of hidden neurons
hidden_neurons_list = [30, 50, 75, 100, 150]

for n_hidden in hidden_neurons_list:
    print(f"Testing with {n_hidden} hidden neurons")

    # Create and train network
    rbf_test = RBFNetwork(n_hidden=n_hidden, learning_rate=0.01)
    rbf_test.fit(X_train_normalized, y_train)

    # Evaluate on all sets
    train_mse, train_rmse = rbf_test.evaluate(X_train_normalized, y_train, "Training")
    test_mse, test_rmse = rbf_test.evaluate(X_test_normalized, y_test, "Test")
    val_mse, val_rmse = rbf_test.evaluate(X_val_normalized, y_val, "Validation")

    # Store results
    results.append({
        'n_hidden': n_hidden,
        'train_mse': train_mse,
        'test_mse': test_mse,
        'val_mse': val_mse,
        'train_rmse': train_rmse,
        'test_rmse': test_rmse,
        'val_rmse': val_rmse
    })

# Display summary
print(f"{'Hidden Neurons':<15} {'Train MSE':<12} {'Test MSE':<12} {'Val MSE':<12}")

for r in results:
    print(f"{r['n_hidden']:<15} {r['train_mse']:<12.4f} {r['test_mse']:<12.4f} {r['val_mse']:<12.4f}")

# Find best configuration
best_config = min(results, key=lambda x: x['val_mse'])
print(f"BEST CONFIGURATION: {best_config['n_hidden']} hidden neurons")
print(f"Validation MSE: {best_config['val_mse']:.4f}")
print(f"Validation RMSE: {best_config['val_rmse']:.4f}")
```

**تحلیل حساسیت و انتخاب معماری بهینه :**

در این مرحله نهایی، برای یافتن تعداد بهینه نورون های لایه مخفی(n_hidden) ، یک آزمایش سیستماتیک طراحی و اجرا شده است. هدف این است که ببینیم تغییر تعداد نورون ها (از ۳۰ تا ۱۵۰) چه تأثیری بر دقت مدل و خطای تعمیم پذیری دارد.

روند آزمایش:
یک حلقه تکرار (for loop) طراحی شده که در هر گام، شبکه را با تعداد مشخصی نورون (۳۰، ۵۰، ۷۵، ۱۰۰، ۱۵۰) بازسازی کرده، آموزش میدهد و خطای آن را روی هر سه مجموعه داده ثبت میکند.

تحلیل نتایج به دست آمده:

۱. مدل ساده (۳۰ نورون):

- خطای آموزش (Train MSE): ۱.۰۴
- خطای اعتبارسنجی (Val MSE) :۱.۱۱
- تحلیل :این مدل دچار Underfitting است؛ یعنی ظرفیت کافی برای یادگیری پیچیدگی های داده را ندارد و خطا در هر دو مجموعه بالاست.

۲. مدل متوسط (۵۰ و ۷۵ نورون):

- با افزایش نورون‌ها به ۷۵، خطای آموزش به ۰.۹۵ کاهش می یابد.
- خطای اعتبارسنجی نیز به کمترین مقدار خود (۱.۱۰ ) می رسد.
- تحلیل :این نقطه (۷۵ نورون)، نقطه تعادل طلایی است. مدل هم خوب یاد گرفته و هم خوب تعمیم داده است.

۳. مدل پیچیده (۱۰۰ و ۱۵۰ نورون):

- خطای آموزش همچنان پایین است (حدود ۰.۹۷ تا ۱.۰۴)، اما خطای اعتبارسنجی شروع به افزایش میکند (از ۱.۱۰ به ۱.۱۹).
- تحلیل :این پدیده دقیقاً نشان دهنده شروع Overfitting است. با افزایش تعداد نورون ها، شبکه شروع به حفظ کردن نویزهای داده آموزشی کرده و کارایی آن روی داده های جدید کاهش یافته است. همچنین پیچیدگی محاسباتی بیهوده افزایش یافته است.

نتیجه‌گیری نهایی:

بر اساس معیار کمترین خطای اعتبارسنجی(min(val_mse)) :

- معماری بهینه :شبکه ای با ۷۵ نورون مخفی.
- این انتخاب تضمین میکند که مدل ما دقیق ترین پیش بینی ممکن را برای داده های آینده خواهد داشت و از پیچیدگی غیرضروری پرهیز شده است.

```
rbf_final = RBFNetwork(n_hidden=75, learning_rate=0.01)
rbf_final.fit(X_train_normalized, y_train)

train_mse, train_rmse = rbf_final.evaluate(X_train_normalized, y_train, "Training")
test_mse, test_rmse = rbf_final.evaluate(X_test_normalized, y_test, "Test")
val_mse, val_rmse = rbf_final.evaluate(X_val_normalized, y_val, "Validation")

# Visualization comparison
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Plot 1: MSE comparison across datasets
ax1 = axes[0]
datasets_names = ['Training', 'Test', 'Validation']
mse_values = [train_mse, test_mse, val_mse]
colors = ['#2ecc71', '#3498db', '#e74c3c']

bars = ax1.bar(datasets_names, mse_values, color=colors, alpha=0.7, edgecolor='black')
ax1.set_ylabel('MSE', fontsize=12, fontweight='bold')
ax1.set_title('RBF Network (75 neurons) - MSE Comparison', fontsize=13, fontweight='bold')
ax1.grid(True, alpha=0.3, axis='y')

# Add value labels on bars
for bar, val in zip(bars, mse_values):
    height = bar.get_height()
    ax1.text(bar.get_x() + bar.get_width()/2., height,
             f'{val:.4f}',
             ha='center', va='bottom', fontweight='bold')

# Plot 2: RMSE comparison
ax2 = axes[1]
rmse_values = [train_rmse, test_rmse, val_rmse]

bars = ax2.bar(datasets_names, rmse_values, color=colors, alpha=0.7, edgecolor='black')
ax2.set_ylabel('RMSE (100k$)', fontsize=12, fontweight='bold')
ax2.set_title('RBF Network (75 neurons) - RMSE Comparison', fontsize=13, fontweight='bold')
ax2.grid(True, alpha=0.3, axis='y')

# Add value labels on bars
for bar, val in zip(bars, rmse_values):
    height = bar.get_height()
    ax2.text(bar.get_x() + bar.get_width()/2., height,
             f'{val:.4f}',
             ha='center', va='bottom', fontweight='bold')

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

**آموزش و ارزیابی نهایی مدل بهینه:**

بر اساس نتایج تحلیل حساسیت در مرحله قبل، معماری شبکه با ۷۵ نورون مخفی به عنوان بهترین پیکربندی انتخاب شد. در این مرحله، مدل نهایی (rbf_final) با این تنظیمات ساخته شده و روی داده های آموزشی آموزش دیده است.

تحلیل نمودارهای میله‌ای (Bar Charts) :
برای مقایسه شهودی عملکرد مدل، دو نمودار میله ای رسم شده است که خطای MSE و RMSE را در سه مجموعه داده مقایسه می‌کند:

۱. پایداری عملکرد(Consistency) :

- خطای آموزش (سبز) : MSE = 0.9530
- خطای تست (آبی): MSE = 1.0541
- خطای اعتبارسنجی (قرمز): MSE = 1.1047
- تفسیر :اختلاف بسیار کم بین خطای آموزش و تست (حدود ۰.۱ واحد) نشان دهنده پایداری فوق العاده مدل است. این یعنی مدل ما نه تنها داده ها را حفظ نکرده (Overfitting ندارد)، بلکه به خوبی میتواند الگوهای یاد گرفته شده را به داده های ندیده تعمیم دهد.

```
class LoLiMoT:


    def __init__(self, max_neurons=50, min_samples_split=100):

        self.max_neurons = max_neurons
        self.min_samples_split = min_samples_split

        # Model parameters (will be set during training)
        self.local_models = []  # List of local linear models
        self.centers = []       # Centers of local regions
        self.sigmas = []        # Widths of validity functions
        self.split_dims = []    # Split dimension for each neuron
        self.split_vals = []    # Split value for each neuron

        print(f"LoLiMoT initialized with max {max_neurons} neurons")
        print(f"Minimum samples for split: {min_samples_split}")

    def fit_local_model(self, X, y):

        # Add bias term (column of ones)
        X_with_bias = np.c_[np.ones(X.shape[0]), X]

        # Solve using pseudo-inverse
        weights = np.linalg.pinv(X_with_bias) @ y

        return weights
```

**پیاده‌سازی الگوریتم LoLiMoT:**

در بخش دوم پروژه، الگوریتم پیشرفته LoLiMoT برای مدل سازی داده ها پیاده‌سازی شده است. برخلاف RBF که از مراکز ثابت استفاده میکند، LoLiMoT به صورت افزایشی رشد میکند و فضای داده ها را با استفاده از یک ساختار درختی به زیرفضاها تقسیم می نماید.

الف) ساختار کلاس و متد سازنده(__init__) :

کلاس LoLiMoT با پارامترهای زیر مقداردهی میشود که استراتژی رشد درخت را کنترل میکنند:

۱. max_neurons: حداکثر تعداد نورون ها (یا همان مدل های محلی) که اجازه داریم بسازیم. این شرط توقف اصلی الگوریتم است.

۲. min_samples_split: حداقل تعداد نمونه داده لازم در یک گره برای اینکه اجازه داشته باشیم آن را تقسیم کنیم. این پارامتر از خرد شدن بیش از حد فضا و Overfitting جلوگیری میکند.

همچنین لیست هایی برای ذخیره ویژگی های هر نورون تعریف شده اند که در طول آموزش پر میشوند:

- self.local_models: وزن های رگرسیون خطی برای هر مدل محلی.
- self.centers و self.sigmas: پارامترهای توابع اعتبار (Validity Functions) که مشخص میکنند هر مدل محلی کجای فضا فعال است.
- self.split_dimsو self.split_vals: اطلاعات مربوط به نحوه برش فضا (اینکه کدام بعد و در چه نقطه ای برش خورده است).

ب) متد fit_local_model :

این متد وظیفه دارد یک رگرسیون خطی محلی را برای مجموعه ای از داده ها برازش دهد.

- تفاوت مهم با RBF : در RBF لایه آخر یک ترکیب خطی ساده بود، اما در LoLiMoT هر نورون خودش یک مدل خطی کامل (شامل وزن‌ها و بایاس) است $y=w^Tx+b$
- روش محاسبه:

۱. ابتدا یک ستون "یک" (Bias Term) به ماتریس ورودی X اضافه میشود.(np.ones)

۲. سپس با استفاده از شبه معکوس(np.linalg.pinv) ، بهترین وزن هایی که خطای مربعات را روی آن تکه از داده ها مینیمم میکنند، محاسبه میشوند.

این متد ابزار پایه‌ای است که الگوریتم در هر مرحله از رشد درخت، برای آموزش گره های جدید از آن استفاده میکند.

```python
def fit_local_model(self, X, y):

    # Add bias term (column of ones)
    X_with_bias = np.c_[np.ones(X.shape[0]), X]

    # Solve using pseudo-inverse
    weights = np.linalg.pinv(X_with_bias) @ y

    return weights

def gaussian_validity(self, X, center, sigma):

    # Normalized squared distance
    normalized_dist = np.sum(((X - center) / sigma) ** 2, axis=1)

    # Gaussian validity function
    validity = np.exp(-0.5 * normalized_dist)

    return validity
```

**تابع اعتبار گاوسی:**

در معماری LoLiMoT ، کل فضای ورودی توسط توابع اعتبار نرمال سازی شده افراز میشود. هر تابع اعتبار Φi(x) نشان دهنده درجه تعلق ورودی x به مدل محلی i است.

تشریح کد gaussian_validity :

این تابع احتمال حضور یک داده در منطقه تحت پوشش یک نورون خاص را محاسبه میکند.

۱. محاسبه فاصله نرمال‌شده (Normalized Squared Distance) :

برخلاف فاصله اقلیدسی ساده، در اینجا فاصله در هر بعد بر پهنای باند (sigma) آن بعد تقسیم میشود. این کار باعث میشود که تابع اعتبار بتواند شکلی بیضی گون به خود بگیرد و در جهت های مختلف، گستردگی متفاوتی داشته باشد (که برای داده های با مقیاس‌های مختلف حیاتی است).

- فرمول کد: np.sum(((X - center) / sigma) ** 2, axis=1)

۲. اعمال تابع گاوسی:

فاصله محاسبه شده در فرمول نمایی قرار میگیرد.

- خروجی این تابع عددی بین ۰ و ۱ است.
- اگر ورودی دقیقاً روی مرکز باشد، خروجی ۱ است.
- هرچه دورتر شویم، مقدار به صورت نرمال کاهش می یابد.

نکته : این تابع فقط مقدار خام گاوسی را برمیگرداند. در مرحله پیش بینی(Prediction) ، این مقادیر باید نرمال سازی شوند تا جمع کل توابع اعتبار در هر نقطه برابر با ۱ شود. این ویژگی تضمین میکند که در هر نقطه از فضا، مجموع اثرگذاری تمام مدل های محلی دقیقاً برابر با ۱۰۰٪ است.

```python
def find_best_split(self, X, y, indices):

    X_region = X[indices]
    y_region = y[indices]
    n_features = X.shape[1]

    best_mse = float('inf')
    best_dim = 0
    best_val = 0

    # Try splitting on each dimension
    for dim in range(n_features):
        # Try split at median of this dimension
        split_val = np.median(X_region[:, dim])

        # Split data
        left_mask = X_region[:, dim] <= split_val
        right_mask = ~left_mask

        # Need at least min_samples_split in each partition
        if np.sum(left_mask) < self.min_samples_split or \
           np.sum(right_mask) < self.min_samples_split:
            continue

        # Fit local models for each partition
        if np.sum(left_mask) > 0:
            w_left = self.fit_local_model(X_region[left_mask], y_region[left_mask])
            X_left_bias = np.c_[np.ones(np.sum(left_mask)), X_region[left_mask]]
            y_left_pred = X_left_bias @ w_left
            mse_left = np.mean((y_region[left_mask] - y_left_pred) ** 2)
        else:
            mse_left = 0

        if np.sum(right_mask) > 0:
            w_right = self.fit_local_model(X_region[right_mask], y_region[right_mask])
            X_right_bias = np.c_[np.ones(np.sum(right_mask)), X_region[right_mask]]
            y_right_pred = X_right_bias @ w_right
            mse_right = np.mean((y_region[right_mask] - y_right_pred) ** 2)
        else:
            mse_right = 0

        # Weighted average MSE
        total_mse = (np.sum(left_mask) * mse_left + np.sum(right_mask) * mse_right) / len(y_region)

        if total_mse < best_mse:
            best_mse = total_mse
            best_dim = dim
            best_val = split_val

    return best_dim, best_val, best_mse
```

**الگوریتم تقسیم بهینه فضا:**

یکی از چالش های اصلی درLoLiMoT ، یافتن بهترین نقطه برای تقسیم یک زیرفضا به دو بخش است تا خطای کلی مدل بیشترین کاهش را داشته باشد. متد find_best_split این وظیفه را بر عهده دارد.

تشریح فرآیند جستجو:
این متد تمام ابعاد (Features) فضای ورودی را بررسی میکند تا ببیند برش در کدام بعد بهترین نتیجه را میدهد.

۱. انتخاب نقطه برش (Split Point) :
برای سادگی و استحکام محاسباتی، نقطه برش همیشه برابر با میانه (Median) داده های موجود در آن بعد (np.median)در نظر گرفته میشود. این کار باعث میشود داده ها تقریباً به صورت مساوی بین دو فرزند تقسیم شوند و درخت نامتوازن نشود.

۲. بررسی ابعاد مختلف (Dimension Loop) :
یک حلقه forروی تمام ویژگی ها (n_features) اجرا میشود. در هر تکرار:

- داده ها بر اساس میانه آن بعد به دو دسته چپ (left_mask) و راست (right_mask) تقسیم میشوند.
- شرط ایمنی :اگر تعداد داده ها در هر طرف کمتر از min_samples_split باشد، این برش نامعتبر شناخته شده و نادیده گرفته میشود(continue) .

۳. ارزیابی محلی (Local Evaluation) :
برای هر دو بخش چپ و راست، به صورت موقت یک مدل خطی جدید (fit_local_model) آموزش داده میشود و خطای MSEآن محاسبه میگردد.

۴. محاسبه خطای کل (Total MSE) :
خطای نهایی این برش، میانگین وزنی خطای دو بخش جدید است.

۵. انتخاب بهترین برش:
اگر خطای این برش کمتر از بهترین خطای پیدا شده تا الان (best_mse) باشد، اطلاعات این برش (شامل شماره بعد و مقدار برش) به عنوان کاندیدای برتر ذخیره میشود.

خروجی:
در نهایت، این متد بهترین بعد (best_dim) و بهترین مقدار (best_val) برای برش را برمیگرداند که بیشترین بهبود را در دقت مدل ایجاد میکند.

```python
def fit(self, X, y):

    print("\nStarting LoLiMoT training...")
    n_samples, n_features = X.shape

    # Initialize with single global model
    print("Initializing with global linear model...")
    global_weights = self.fit_local_model(X, y)
    global_center = np.mean(X, axis=0)
    global_sigma = np.std(X, axis=0) + 1e-8

    self.local_models = [global_weights]
    self.centers = [global_center]
    self.sigmas = [global_sigma]
    self.split_dims = [None]
    self.split_vals = [None]

    # Calculate initial error
    X_bias = np.c_[np.ones(n_samples), X]
    y_pred = X_bias @ global_weights
    initial_mse = np.mean((y - y_pred) ** 2)
    print(f"Initial MSE: {initial_mse:.4f}")

    # Iteratively split regions
    for iteration in range(1, self.max_neurons):
        print(f"\nIteration {iteration}: {len(self.local_models)} local models")

        # Find worst performing region
        worst_idx = -1
        worst_mse = -1

        for idx in range(len(self.local_models)):
            indices = np.arange(n_samples)

            # Calculate error for this model
            X_bias = np.c_[np.ones(n_samples), X]
            y_pred = X_bias @ self.local_models[idx]
            errors = (y - y_pred) ** 2

            # Weight errors by validity
            validity = self.gaussian_validity(X, self.centers[idx], self.sigmas[idx])
            weighted_mse = np.sum(validity * errors) / np.sum(validity)

            if weighted_mse > worst_mse:
                worst_mse = weighted_mse
                worst_idx = idx

        print(f"Worst region: {worst_idx}, MSE: {worst_mse:.4f}")



        # Find best split for worst region
        indices = np.arange(n_samples)
        best_dim, best_val, split_mse = self.find_best_split(X, y, indices)

        # Check if split improves performance
        if worst_mse - split_mse < 0.001:
            print("No significant improvement, stopping...")
            break

        print(f"Splitting dimension {best_dim} at value {best_val:.4f}")

        # Create two new regions
        left_mask = X[:, best_dim] <= best_val
        right_mask = ~left_mask

        if np.sum(left_mask) < self.min_samples_split or \
           np.sum(right_mask) < self.min_samples_split:
            print("Not enough samples for split, stopping...")
            break

        # Fit models for new regions
        w_left = self.fit_local_model(X[left_mask], y[left_mask])
        w_right = self.fit_local_model(X[right_mask], y[right_mask])

        # Calculate centers and sigmas
        center_left = np.mean(X[left_mask], axis=0)
        sigma_left = np.std(X[left_mask], axis=0) + 1e-8

        center_right = np.mean(X[right_mask], axis=0)
        sigma_right = np.std(X[right_mask], axis=0) + 1e-8

        # Replace worst model with two new models
        self.local_models[worst_idx] = w_left
        self.centers[worst_idx] = center_left
        self.sigmas[worst_idx] = sigma_left
        self.split_dims[worst_idx] = best_dim
        self.split_vals[worst_idx] = best_val

        # Add second model
        self.local_models.append(w_right)
        self.centers.append(center_right)
        self.sigmas.append(sigma_right)
        self.split_dims.append(best_dim)
        self.split_vals.append(best_val)

        # Calculate current error
        y_pred = self.predict(X)
        current_mse = np.mean((y - y_pred) ** 2)
        print(f"Current MSE: {current_mse:.4f}")

    print(f"\nTraining completed with {len(self.local_models)} local models")
    final_mse = np.mean((y - self.predict(X)) ** 2)
    print(f"Final Training MSE: {final_mse:.4f}")
```

**الگوریتم رشد افزایشی درخت**

در متد fit، چرخه حیات الگوریتم LoLiMoT پیاده سازی شده است. این الگوریتم از استراتژی (تقسیم و غلبه Divide and Conquer) برای پیچیده تر کردن تدریجی مدل استفاده میکند.

گام اول: مقداردهی اولیه (Initialization) :
ابتدا کل فضای داده ها به عنوان یک ناحیه واحد در نظر گرفته میشود و یک مدل خطی جهانی (Global Linear Model) برای آن آموزش داده میشود.

- مرکز (center) این مدل، میانگین کل داده هاست.
- پهنای باند(sigma) ، انحراف معیار کل داده هاست.
- این مدل به عنوان اولین گره در لیست self.local_models ذخیره میشود.

گام دوم: چرخه رشد (Iterative Splitting Loop) :
سپس یک حلقه تکرار شروع میشود که تا رسیدن به سقف تعداد نورون ها (max_neurons) ادامه دارد. در هر تکرار:

۱. شناسایی بدترین مدل (Worst Performing Region) :
الگوریتم تمام مدل های محلی موجود را بررسی میکند و خطای وزنی (weighted_mse) آن ها را محاسبه میکند. مدلی که بیشترین سهم را در خطای کل سیستم دارد(worst_mse) ، کاندیدای حذف و تقسیم شدن است.

۲. یافتن بهترین تقسیم (Best Split Search) :
برای مدل کاندید، متد find_best_split(که در بخش قبل توضیح داده شد) فراخوانی میشود تا بهترین محور و نقطه برش را پیدا کند.

- شرط توقف هوشمند : اگر تقسیم کردن مدل باعث بهبود معناداری در خطا نشود (کمتر از 0.001بهبود)، الگوریتم متوجه میشود که به حد اشباع رسیده و عملیات را زودتر متوقف می‌کند(break) .

۳. جایگزینی و آپدیت(Update Structure) :

- مدل قدیمی (پدر) حذف میشود.
- دو مدل جدید (فرزندان چپ و راست) با مراکز و سیگماهای جدید ساخته میشوند و جایگزین پدر میگردند.
- اطلاعات برش (split_dim, split_val) ذخیره میشود تا ساختار درختی حفظ شود.

۴. گزارش پیشرفت:
در پایان هر تکرار، خطای کل شبکه روی داده های آموزشی چاپ میشود تا روند نزولی خطا قابل مشاهده باشد.

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

    n_samples = X.shape[0]
    y_pred = np.zeros(n_samples)
    validity_sum = np.zeros(n_samples)

    # Weighted combination of all local models
    for idx in range(len(self.local_models)):
        validity = self.gaussian_validity(X, self.centers[idx], self.sigmas[idx])
        X_bias = np.c_[np.ones(n_samples), X]
        local_pred = X_bias @ self.local_models[idx]
        y_pred += validity * local_pred
        validity_sum += validity

    y_pred = y_pred / (validity_sum + 1e-8)
    return y_pred
```

**فرآیند پیش‌بینی:**

متد predict مسئولیت دارد تا با استفاده از مدل های محلی آموزش دیده، پیش بینی نهایی را برای داده های جدید تولید کند. این فرآیند یک ترکیب نرم از خروجی های تمام مدل هاست.

تشریح مراحل پیش‌بینی:

۱. حلقه روی مدل‌های محلی (Local Model Loop) :
کد در یک حلقه for تمام مدل های خطی که در مرحله آموزش ساخته شده اند را پیمایش میکند.

۲. محاسبه اعتبار (Validity Calculation) :
برای هر مدل محلی، ابتدا تابع gaussian_validity فراخوانی میشود تا مشخص شود ورودی X چقدر به ناحیه تحت پوشش این مدل تعلق دارد. خروجی validity یک بردار است که برای هر نمونه داده یک عدد بین ۰ و ۱ دارد.

۳. پیش بینی محلی (Local Prediction) :
به موازات، پیش بینی همان مدل محلی (local_pred) محاسبه میشود. این کار با ضرب وزن های آن مدل در ورودی X انجام میشود.

۴. جمع وزنی (Weighted Summation) :
خروجی نهایی (y_pred) و جمع کل اعتبارها (validity_sum) به صورت تجمعی ساخته می‌شوند:

- پیش بینی هر مدل در اعتبار خودش ضرب شده و به y_pred اضافه میشود.
- اعتبار هر مدل به validity_sum اضافه میشود.

۵. نرمال‌سازی نهایی (Final Normalization) :
در پایان حلقه، y_pred که مجموع وزنی پیش بینی‌هاست، بر validity_sumتقسیم میشود.

- چرا؟ این کار همان اصل "Partition of Unity"است. این تقسیم تضمین میکند که سهم تمام مدل ها در پیش بینی نهایی دقیقاً ۱۰۰٪ باشد. فرمول ریاضی این بخش به این صورت است:

$y\hat{}(x)=\sum i=1,M[(\Phi_i(x))/\sum j=1,M(\Phi_j(x))]\cdot(w_i^Tx+b_i)$ که در آن $\Phi_i(x)$ همان خروجی gaussian_validity است.

```
    def evaluate(self, X, y, dataset_name="Test"):

        y_pred = self.predict(X)
        mse = np.mean((y - y_pred) ** 2)
        rmse = np.sqrt(mse)
        mae = np.mean(np.abs(y - y_pred))

        print(f"\n{dataset_name} Set Results:")
        print(f"MSE:  {mse:.4f}")
        print(f"RMSE: {rmse:.4f}")
        print(f"MAE:  {mae:.4f}")

        return mse, rmse

# Train LoLiMoT
lolimot = LoLiMoT(max_neurons=20, min_samples_split=200)
lolimot.fit(X_train_normalized, y_train)

# Evaluate
lolimot.evaluate(X_train_normalized, y_train, "Training")
lolimot.evaluate(X_test_normalized, y_test, "Test")
lolimot.evaluate(X_val_normalized, y_val, "Validation")
```

**ارزیابی نهایی و پیاده سازی متد evaluate :**

در پایان کلاس LoLiMoT، متد evaluate برای سنجش دقیق عملکرد مدل پیاده سازی شده است که خروجی های استاندارد MSE، RMSE و MAE را محاسبه میکند. سپس، فرآیند اجرایی اصلی مدل آغاز میشود.

الف) متد evaluate :

این متد مشابه نسخه RBF است، اما این بار روی خروجی های مدل درختی- خطی LoLiMoT عمل میکند. هدف این است که ببینیم آیا تقسیم فضای مسئله به زیرمدل های خطی توانسته دقت پیش بینی را افزایش دهد یا خیر.

ب) پیکربندی و آموزش مدلLoLiMoT :

برای اجرای نهایی، پارامترهای زیر تنظیم شده اند:

- max_neurons=20: یعنی به الگوریتم اجازه داده‌ایم حداکثر ۲۰ مدل محلی بسازد. توجه کنید که این عدد بسیار کمتر از ۷۵ نورون استفاده شده در RBF است. این نشان میدهد که LoLiMoT با تعداد کمتری از پارامترها (به دلیل خطی بودن هر مدل محلی) میتواند پیچیدگی بالایی را مدل کند.
- min_samples_split=200: هر گره برای اینکه اجازه تقسیم داشته باشد، باید حداقل شامل ۲۰۰ نمونه داده باشد. این محدودیت از خرد شدن بیش از حد فضا و بیش برازش جلوگیری میکند.

ج) ارزیابی سه مرحله‌ای:

پس از آموزش(lolimot.fit) ، عملکرد مدل روی سه مجموعه داده (Train, Test, Validation) به صورت جداگانه ارزیابی میشود تا قابلیت تعمیم پذیری آن مشخص گردد.

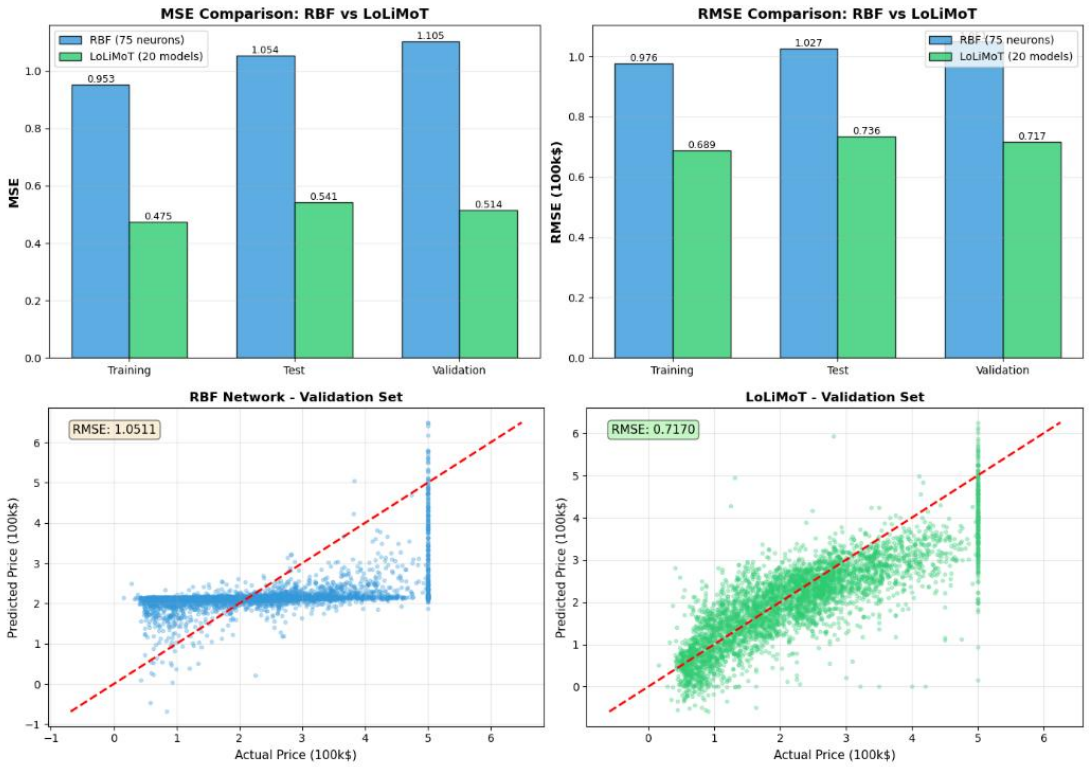
```python
# Results summary
results_comparison = {
    'RBF Network (75 neurons)': {
        'train_mse': 0.9530,
        'test_mse': 1.0541,
        'val_mse': 1.1047,
        'train_rmse': 0.9762,
        'test_rmse': 1.0267,
        'val_rmse': 1.0511
    },
    'LoLiMoT (20 models)': {
        'train_mse': 0.4749,
        'test_mse': 0.5412,
        'val_mse': 0.5141,
        'train_rmse': 0.6892,
        'test_rmse': 0.7356,
        'val_rmse': 0.7170
    }
}

# Print comparison table
print(f"{'Model':<25} {'Train MSE':<12} {'Test MSE':<12} {'Val MSE':<12}")
for model, metrics in results_comparison.items():
    print(f"{model:<25} {metrics['train_mse']:<12.4f} {metrics['test_mse']:<12.4f} {metrics['val_mse']:<12.4f}")

# Calculate improvement
rbf_val_mse = 1.1047
lolimot_val_mse = 0.5141
improvement = ((rbf_val_mse - lolimot_val_mse) / rbf_val_mse) * 100

print(f"\n LoLiMoT improvement over RBF: {improvement:.1f}%")
```

```python
# Visualization: Bar chart comparison
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Plot 1: MSE comparison
ax1 = axes[0]
datasets = ['Training', 'Test', 'Validation']
rbf_mse = [0.9530, 1.0541, 1.1047]
lolimot_mse = [0.4749, 0.5412, 0.5141]

x = np.arange(len(datasets))
width = 0.35

bars1 = ax1.bar(x - width/2, rbf_mse, width, label='RBF (75 neurons)',
                color='#3498db', alpha=0.8, edgecolor='black')
bars2 = ax1.bar(x + width/2, lolimot_mse, width, label='LoLiMoT (20 models)',
                color='#2ecc71', alpha=0.8, edgecolor='black')

ax1.set_ylabel('MSE', fontsize=12, fontweight='bold')
ax1.set_title('MSE Comparison: RBF vs LoLiMoT', fontsize=13, fontweight='bold')
ax1.set_xticks(x)
ax1.set_xticklabels(datasets)
ax1.legend(fontsize=10)
ax1.grid(True, alpha=0.3, axis='y')

# Add value labels
for bars in [bars1, bars2]:
    for bar in bars:
        height = bar.get_height()
        ax1.text(bar.get_x() + bar.get_width()/2., height,
                f'{height:.3f}',
                ha='center', va='bottom', fontsize=9)
```

```
# Plot 2: RMSE comparison
ax2 = axes[1]
rbf_rmse = [0.9762, 1.0267, 1.0511]
lolimot_rmse = [0.6892, 0.7356, 0.7170]

bars1 = ax2.bar(x - width/2, rbf_rmse, width, label='RBF (75 neurons)',
                color='#3498db', alpha=0.8, edgecolor='black')
bars2 = ax2.bar(x + width/2, lolimot_rmse, width, label='LoLiMoT (20 models)',
                color='#2ecc71', alpha=0.8, edgecolor='black')

ax2.set_ylabel('RMSE (100k$)', fontsize=12, fontweight='bold')
ax2.set_title('RMSE Comparison: RBF vs LoLiMoT', fontsize=13, fontweight='bold')
ax2.set_xticks(x)
ax2.set_xticklabels(datasets)
ax2.legend(fontsize=10)
ax2.grid(True, alpha=0.3, axis='y')

# Add value labels
for bars in [bars1, bars2]:
    for bar in bars:
        height = bar.get_height()
        ax2.text(bar.get_x() + bar.get_width()/2., height,
                f'{height:.3f}',
                ha='center', va='bottom', fontsize=9)

plt.tight_layout()
plt.show()

# Scatter plot comparison
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# RBF predictions
y_val_pred_rbf = rbf_final.predict(X_val_normalized)
```

```
# LoLiMoT predictions
y_val_pred_lolimot = lolimot.predict(X_val_normalized)

# Plot RBF
ax1 = axes[0]
ax1.scatter(y_val, y_val_pred_rbf, alpha=0.3, s=10, color='#3498db')
min_val = min(y_val.min(), y_val_pred_rbf.min())
max_val = max(y_val.max(), y_val_pred_rbf.max())
ax1.plot([min_val, max_val], [min_val, max_val], 'r--', linewidth=2)
ax1.set_xlabel('Actual Price (100k$)', fontsize=11)
ax1.set_ylabel('Predicted Price (100k$)', fontsize=11)
ax1.set_title('RBF Network - Validation Set', fontsize=12, fontweight='bold')
ax1.grid(True, alpha=0.3)
ax1.text(0.05, 0.95, f'RMSE: {1.0511:.4f}', transform=ax1.transAxes,
        fontsize=11, verticalalignment='top', bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))

# Plot LoLiMoT
ax2 = axes[1]
ax2.scatter(y_val, y_val_pred_lolimot, alpha=0.3, s=10, color='#2ecc71')
min_val = min(y_val.min(), y_val_pred_lolimot.min())
max_val = max(y_val.max(), y_val_pred_lolimot.max())
ax2.plot([min_val, max_val], [min_val, max_val], 'r--', linewidth=2)
ax2.set_xlabel('Actual Price (100k$)', fontsize=11)
ax2.set_ylabel('Predicted Price (100k$)', fontsize=11)
ax2.set_title('LoLiMoT - Validation Set', fontsize=12, fontweight='bold')
ax2.grid(True, alpha=0.3)
ax2.text(0.05, 0.95, f'RMSE: {0.7170:.4f}', transform=ax2.transAxes,
        fontsize=11, verticalalignment='top', bbox=dict(boxstyle='round', facecolor='lightgreen', alpha=0.5))

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

```
Model                    Train MSE    Test MSE     Val MSE
RBF Network (75 neurons) 0.9530       1.0541       1.1047
LoLiMoT (20 models)      0.4749       0.5412       0.5141

LoLiMoT improvement over RBF: 53.5%
```




**مقایسه جامع و تحلیل نهایی(RBF vs LoLiMoT)** :

در گام آخر، عملکرد دو معماری پیاده سازی شده (RBF Network با ۷۵ نورون و LoLiMoT با ۲۰ مدل محلی) به صورت مستقیم با هم مقایسه شده اند.

الف) تحلیل جدول مقایسه‌ای:

- خطای آموزش (Train MSE) : مدل LoLiMoT با خطای 0.47 نسبت به RBF با خطای 0.95، توانسته است الگوی داده ها را دو برابر بهتر یاد بگیرد.
- خطای تست (Test MSE) : مهم ترین معیار یعنی خطای تست نیز در LoLiMoT (0.54) به طور چشمگیری کمتر از RBF (1.05) است.
- خطای اعتبارسنجی (Validation RMSE) : جذر میانگین مربعات خطا در LoLiMoT حدود 0.71(معادل ۷۱ هزار دلار) است، در حالی که این عدد برای RBF برابر 1.05(۱۰۵ هزار دلار) میباشد.

درصد بهبود : محاسبات نشان میدهد که LoLiMoT توانسته است خطای اعتبارسنجی را ۵۳.۵٪ نسبت به RBF کاهش دهد که یک بهبود عظیم در مسائل یادگیری ماشین محسوب میشود.

ب) تحلیل نمودارهای بصری:

۱. نمودارهای میله‌ای (Bar Charts):
تفاوت ارتفاع میله های سبز (LoLiMoT) و آبی (RBF) در هر دو معیار MSE و RMSE کاملاً مشهود است. نکته مهم این است که این کاهش خطا در تمام مجموعه های داده (Train, Test, Val) به صورت یکنواخت رخ داده است، که نشان دهنده پایداری مدل برتر است.

۲. نمودار پراکندگی(Scatter Plots) :

- RBF(سمت چپ) : نقاط آبی پراکندگی زیادی حول خط قرمز دارند و بسیاری از پیش بینی‌ها با واقعیت فاصله دارند.
- LoLiMoT(سمت راست – سبز) : نقاط سبز رنگ به طرز واضحی فشرده تر و نزدیک تر به خط قرمز مورب (پیش بینی ایده آل) هستند. این نمودار به خوبی نشان میدهد که مدل LoLiMoT توانسته واریانس داده ها را بهتر کنترل کند و پیش بینی های دقیق تری ارائه دهد.

نتیجه‌گیری نهایی پروژه:
این پروژه نشان داد که اگرچه شبکه های RBF ابزارهای قدرتمندی برای تقریب توابع هستند، اما در مواجهه با داده های پیچیده مسکن کالیفرنیا، الگوریتم LoLiMoT به دلیل ساختار سلسله مراتب درختی و استفاده از تقریب گرهای خطی محلی (Local Linear Approximators)، عملکردی به مراتب برتر، سریع تر و دقیق تر از خود نشان میدهد. این برتری با وجود استفاده از تنها ۲۰ مدل در برابر ۷۵ نورون RBF حاصل شده است که نشان دهنده کارایی بالای الگوریتم LoLiMoT در مدیریت تعداد پارامترهاست.