
بسم الله الرحمن الرحيم

دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

پروژه مبانی هوش مصنوعی

گزارش کتبی فاز اول

اعضای گروه:

ارشیا شفیعی

رضا چراخ

فرناز موحدی

آبان ۱۴۰۳

فهرست مطالب

عنوان **صفحه**

فهرست مطالب

عنوان **صفحه**

فصل اول
جستجو

۱-۱- مقدمه

در مباحث هوش مصنوعی، الگوریتم‌های جستجو برای یافتن مسیر بین دو نقطه، حل مسائل در فضای حالت و پیدا کردن راه‌حل‌های بهینه اهمیت دارند. سه الگوریتمDFS ، UCSو *A از رایج‌ترین الگوریتم‌های جستجو هستند که به‌طور متفاوتی از پشته، صف اولویت‌دار و هیوریستیک‌ها برای پیمایش فضای حالت استفاده می‌کنند. هدف هر یک از این الگوریتم‌ها یافتن مسیری است که از حالت شروع به حالت هدف می‌رسد، و تفاوت‌ها در چگونگی پیدا کردن این مسیر و توجه به بهینگی آن است.

۱-۲- الگوریتم DFS

۱-۲-۱- پیاده‌سازی

الگوریتم با استفاده از یک پشته به نام frontier آغاز می‌شود که حالت اولیه در آن قرار دارد. این پشته به عنوان مکانی برای نگهداری مسیرهایی که باید بررسی شوند استفاده می‌شود. به همین صورت یک مجموعه به‌نام explored ساخته می‌شود برای حالت‌های تکراری که در مسیر دیده می‌شوند.

هر بار که حالتی از پشته خارج می‌شود، بررسی می‌شود که آیا این حالت، حالت هدف است یا خیر.

اگر حالت فعلی، حالت هدف نبود، فرزندان این حالت را نگاه کرده و آن‌ها به پشته اضافه می‌شود تا در جستجوی بعدی بررسی شوند.

اگر دوباره به حالتی برسیم که قبلاً بررسی شده، آن را نادیده می‌گیریم تا از تکرار جلوگیری کنیم.

```python
def depthFirstSearch(problem):
    # Use a stack for DFS
    frontier = util.Stack()
    start_state = problem.getStartState()
    frontier.push((start_state, []))  # (state, actions)
    explored = set()

    while not frontier.isEmpty():
        state, actions = frontier.pop()

        if problem.isGoalState(state):
            return actions

        if state not in explored:
            explored.add(state)

            for successor, action, _ in problem.getSuccessors(state):
                if successor not in explored:
                    new_actions = actions + [action]
                    frontier.push((successor, new_actions))

    return []
```

در تصاویر زیر گزارش اجرای الگوریتم DFS در سه محیط simple corner, hard corner, big corner قرار گرفته است.

```
└─$ python pacman.py -l simpleCorner -p SearchAgent -a fn=dfs,prob=CornersProblem
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 44 in 0.0 seconds
Search nodes expanded: 202
Pacman emerges victorious! Score: 466
Average Score: 466.0
Scores:        466.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
└─$ python pacman.py -l HardCorner -p SearchAgent -a fn=dfs,prob=CornersProblem
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 221 in 0.0 seconds
Search nodes expanded: 371
Pacman emerges victorious! Score: 319
Average Score: 319.0
Scores:        319.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
└─$ python pacman.py -l BigCorner -p SearchAgent -a fn=dfs,prob=CornersProblem
[SearchAgent] using function dfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 316 in 0.0 seconds
Search nodes expanded: 974
Pacman emerges victorious! Score: 234
Average Score: 234.0
Scores:        234.0
Win Rate:      1/1 (1.00)
Record:        Win
```

۱-۳- الگوریتم UCS

۱-۳-۱- پیاده‌سازی

UCS نوعی از جستجوی گراف است که همواره به دنبال مسیری با کمترین هزینه می‌گردد و این هدف را با استفاده از یک صف اولویت‌دار دنبال می‌کند.

در ابتدا حالت شروع در صف اولویت‌دار قرار می‌گیرد، با هزینه‌ای برابر با صفر.

هر بار، حالتی که کمترین هزینه را دارد از صف خارج و بررسی می‌شود. اگر این حالت، حالت هدف باشد، الگوریتم پایان می‌یابد.

اگر حالت هدف نبود، تمام جانشین‌های آن با هزینه‌های جدید در صف قرار می‌گیرد. هر جانشین بر اساس هزینه‌اش مرتب شده و اولویت داده می‌شود. همچنین کنش انجام‌شده و و هزینه آن به کنش‌ها و هزینه تا حالا انجام شده آن اضافه می‌شود و به حالت بعدی داده می‌شوند.

هر حالت به محض این که بررسی شد، در مجموعه‌ای به نام explored قرار می‌گیرد تا از بررسی مجدد آن جلوگیری شود.

```python
# UCS implementation
def uniformCostSearch(problem):
    frontier = util.PriorityQueue()
    start_state = problem.getStartState()
    frontier.push((start_state, [], 0), 0)  # (state, actions, cost), priority
    explored = set()

    while not frontier.isEmpty():
        state, actions, cost = frontier.pop()

        if problem.isGoalState(state):
            return actions

        if state not in explored:
            explored.add(state)

            for successor, action, step_cost in problem.getSuccessors(state):
                if successor not in explored:
                    new_actions = actions + [action]
                    new_cost = cost + step_cost
                    frontier.push((successor, new_actions, new_cost), new_cost)

    return []
```

در تصاویر زیر گزارش اجرای الگوریتم DFS در سه محیط simple corner, hard corner, big corner قرار گرفته است.

```
└─$ python pacman.py -l simpleCorner -p SearchAgent -a fn=ucs,prob=CornersProblem
[SearchAgent] using function ucs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 20 in 0.0 seconds
Search nodes expanded: 219
Pacman emerges victorious! Score: 490
Average Score: 490.0
Scores:        490.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
└─$ python pacman.py -l HardCorner -p SearchAgent -a fn=ucs,prob=CornersProblem
[SearchAgent] using function ucs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 1908
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
└$ python pacman.py -l BigCorner -p SearchAgent -a fn=ucs,prob=CornersProblem
[SearchAgent] using function ucs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 11128
Pacman emerges victorious! Score: 340
Average Score: 340.0
Scores:        340.0
Win Rate:      1/1 (1.00)
Record:        Win
```

۱-۴- الگوریتم *A

یک الگوریتم جستجوی بهینه است که هم هزینه مسیر را در نظر می‌گیرد و هم از یک تابع هیوریستیک برای تخمین هزینه باقیمانده استفاده می‌کند تا سریع‌تر به پاسخ برسد.

پیاده سازی الگوریتم:

صف اولویت frontier ایجاد می‌شود، و حالت اولیه‌ی مسئله به صف اضافه می‌گردد.

در هر تکرار، حالت با کمترین هزینه از صف خارج می‌شود. اگر آن حالت وضعیت هدف باشد، مسیر پیدا شده به عنوان جواب برگردانده می‌شود.

اگر حالت فعلی در مجموعه حالت‌های explored نباشد، به آن مجموعه اضافه می‌شود و سپس فرزندان حالت فعلی پردازش می‌شوند.

برای هر فرزند، هزینه‌ی جدید محاسبه می‌شود و با توجه به تابع هیوریستیک، حالت جدید به صف اولویت اضافه می‌گردد.

```
# A* implementation
def aStarSearch(problem, heuristic=nullHeuristic):
    frontier = util.PriorityQueue()
    start_state = problem.getStartState()
    frontier.push((start_state, [], 0), 0)
    explored = set()

    while not frontier.isEmpty():
        state, actions, cost = frontier.pop()

        if problem.isGoalState(state):
            return actions

        if state not in explored:
            explored.add(state)

            for successor, action, step_cost in problem.getSuccessors(state):
                new_actions = actions + [action]
                new_cost = cost + step_cost
                heuristic_cost = new_cost + heuristic(successor, problem)
                frontier.push((successor, new_actions, new_cost), heuristic_cost)

    return []
```

۱-۴-۱- تابع هیوریستیک

هیوریستیک تابعی است که هزینه تخمینی مسیر باقی‌مانده از یک حالت به حالت هدف را ارزیابی می‌کند. تابع هیوریستیکی که ما طراحی کردیم شامل چند بخش مختلف است که همگی در کنار هم با ایفا کردن نقش خود، تشکیل یک تابع هیوریستیک مناسب را داده‌اند.

بخش‌های تابع:

ابتدا گوشه‌های خورده نشده تعیین می‌شوند. اگر هیچ گوشه‌ای باقی نمانده باشد، تابع مقدار ۰ را

```
def cornersHeuristic(state, problem):
    pacman_pos, eaten_corners = state
    remaining_corners = [corner for corner, eaten in zip(problem.corners, eaten_corners) if not eaten]

    if not remaining_corners:
        return 0

    num_remaining_corners = len(remaining_corners)
```

برمی‌گرداند.

Greedy Nearest Corner:

این بخش یک رویکرد حریصانه را شبیه‌سازی می‌کند که در آن Pacman ابتدا به نزدیک‌ترین گوشه می‌رود، سپس به نزدیک‌ترین گوشه بعدی می‌رود و به همین ترتیب تا رسیدن به تمام گوشه‌ها ادامه می‌دهد.

مجموع مسافت محاسبه شده در اینجا (total_greedy_cost) یک تقریب حداقلی برای مسیر Pacman است تا از تمام گوشه های باقی مانده بازدید کند، اما همیشه کوتاه ترین مسیر ممکن نیست.

```
# --- 1. Greedy Nearest Corner ---
current_position = pacman_pos
total_greedy_cost = 0
greedy_remaining = remaining_corners[:]

while greedy_remaining:
    distances_to_corners = [
        (util.manhattanDistance(current_position, corner), corner) for corner in greedy_remaining
        ]
    min_distance, closest_corner = min(distances_to_corners)
    total_greedy_cost += min_distance
    current_position = closest_corner
    greedy_remaining.remove(closest_corner)
```

Manhattan Distance to Farthest Corner:

این فاصله به عنوان یک تخمین تقریبی از مسافتی است که Pacman باید طی کند تا به دورترین گوشه برسد، با این فرض که ممکن است بخشی از مسیر بهینه باشد.

```
# --- 2. Manhattan Distance to Farthest Corner ---
farthest_distance = max([util.manhattanDistance(pacman_pos, corner) for corner in remaining_corners])
```

Minimum Spanning Tree:

استفاده از MST به این معنی است که هزینه دسترسی به تمام گوشه‌ها از یک مسیر بهینه محاسبه می‌شود. این بخش پیچیده‌ترین قسمت تابع است.

MST حداقل مسیر را برای اتصال تمام گوشه های باقی مانده بدون در نظر گرفتن ترتیب دقیق تخمین می زند. هر گوشه را به عنوان یک گره در یک گراف در نظر می گیرد و حداقل اتصالات مورد نیاز برای پوشش تمام گوشه ها را پیدا می کند.

این بخش زمانی موثر است که چندین گوشه باقی بماند، زیرا MST نشان دهنده کوتاه ترین راه ممکن برای اتصال همه آن‌ها است، با این فرض که Pacman در نهایت از هر یک عبور می کند.

```
# --- 3. MST (Minimum Spanning Tree) Heuristic ---
from itertools import combinations
corner_graph = []
for corner1, corner2 in combinations(remaining_corners, 2):
    distance = util.manhattanDistance(corner1, corner2)
    corner_graph.append((distance, corner1, corner2))
corner_graph.sort()

mst_cost = 0
corner_sets = {corner: corner for corner in remaining_corners}

def find(corner):
    if corner_sets[corner] != corner:
        corner_sets[corner] = find(corner_sets[corner])
    return corner_sets[corner]

def union(corner1, corner2):
    root1, root2 = find(corner1), find(corner2)
    if root1 != root2:
        corner_sets[root1] = root2

edges_used = 0
for distance, corner1, corner2 in corner_graph:
    if find(corner1) != find(corner2):
        union(corner1, corner2)
        mst_cost += distance
        edges_used += 1
        if edges_used == len(remaining_corners) - 1:
            break
```

Penalizing Unexplored Corners:

یک پنالتی اضافی (unexplored_corner_penalty) بر اساس تعداد کرنرهای باقی مانده معرفی می شود. این پنالتی Pacman را تشویق می کند تا رسیدن به کرنرهای بیشتری را در اولویت قرار دهد، حتی اگر آنها دورتر باشند.

Dynamic Weighting:

تابع اکتشافی به صورت پویا وزن ها را برای اجزای مختلف بسته به تعداد گوشه های باقی مانده تنظیم می کند:

گوشه‌های بیشتری باقی‌مانده‌اند: وزن بیشتری به هیوریستیک MST اضافه می‌کند، زیرا برای مجموعه‌های

بزرگ‌تری از اهداف بازدید نشده دقیق‌تر است.

گوشه‌های کمتری باقی‌مانده‌اند: وزن بیشتری به اجزای الگوریتم حریصانه نزدیکترین گوشه و دورترین فاصله منهتن می‌افزاید، که وقتی اهداف کمی باقی می‌مانند مؤثرتر هستند

```
# --- 4. Penalizing the number of unexplored corners ---
unexplored_corner_penalty = num_remaining_corners * 5

# --- 5. Dynamic weighting based on the number of remaining corners ---
# Weights dynamically adjusted based on the number of remaining corners
# More remaining corners -> higher MST weight; fewer corners -> higher greedy/farthest weights
weight_factor = num_remaining_corners / max(len(problem.corners), 1)  # Proportion of corners left

# More weight to MST when many corners are left
weighted_mst = mst_cost * (1 + weight_factor)
# More weight to Greedy for fewer corners
weighted_greedy = total_greedy_cost * (1 + (1 - weight_factor))
# More weight to Farthest when corners reduce
weighted_farthest = farthest_distance * (1 + (1 - weight_factor))

# --- 6. Final heuristic: combining all parts with dynamic weights and penalties ---
heuristic_value = max(weighted_greedy, weighted_mst, weighted_farthest) + unexplored_corner_penalty

return heuristic_value
```

در نهایت، تابع حداکثر مقدار را در بین اجزای هیوریستیک وزن دار برمی گرداند، و اطمینان حاصل می کند که همیشه از بالاترین برآورد استفاده می کند. این رویکرد تخمین های خوش بینانه و بدبینانه را متعادل می کند و بازتاب دقیق تری از هزینه باقی مانده را ارائه می دهد.

در تصاویر زیر گزارش اجرای الگوریتم A* در سه محیط simple corner, hard corner, big corner قرار گرفته است.

```
[SearchAgent] using function astar and heuristic cornersHeuristic
[SearchAgent] using problem type CornersProblem
Path found with total cost of 20 in 0.0 seconds
Search nodes expanded: 68
Pacman emerges victorious! Score: 490
Average Score: 490.0
Scores:        490.0
Win Rate:      1/1 (1.00)
Record:        Win
```

.

```
[SearchAgent] using function astar and heuristic cornersHeuristic
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 263
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
[SearchAgent] using function astar and heuristic cornersHeuristic
[SearchAgent] using problem type CornersProblem
Path found with total cost of 214 in 0.1 seconds
Search nodes expanded: 827
Pacman emerges victorious! Score: 336
Average Score: 336.0
Scores:        336.0
Win Rate:      1/1 (1.00)
Record:        Win
```

فصل دوم
رگرسیون خطی

۲-۱- مقدمه

در این بخش الگوریتم آموزش یک مدل رگرسیون خطی از روی داده‌های داده شده درمورد وقوع سیل پیاده‌سازی شده است. مدل آموزش داده شده می‌تواند با دریافت ورودی، پیش‌بینی کند که چقدر احتمال دارد سیل رخ دهد.

۲-۲- مراحل پیاده‌سازی

۲-۲-۱- خواندن مجموعه‌داده‌های آموزش و آزمایش

ابتدا داده‌های آموزش و آزمایش را خوانده و سپس ستون id را از داده‌ها حذف می‌کنیم. برای ساخت ماتریس ویژگی در داده‌های آموزش و آزمایش، مقدار flood probability را از داده‌ها حذف کرده و بردار برچسب‌ها را هم می‌سازیم.

```
# Load the train and test datasets
train_df = pd.read_csv('train.csv')
test_df = pd.read_csv('test.csv')

# Drop the 'id' column in both train and test sets
train_df = train_df.drop(columns=['id'])
test_df = test_df.drop(columns=['id'])

# Separate features (X) and label (y) in train and test datasets
X_train = train_df.drop(columns=['FloodProbability'])
y_train = train_df['FloodProbability']

X_test = test_df.drop(columns=['FloodProbability'])
y_test = test_df['FloodProbability']
```

۲-۲-۲- بررسی وجود مقادیر ناموجود

در این بخش وجود یا عدم وجود مقادیر ناموجود در داده‌ها بررسی شده است.

```
# Check the missing values in both train and test sets
train_df_missing_values = train_df.isnull().sum()
test_df_missing_values = test_df.isnull().sum()
print('train missing values: \n', train_df_missing_values, '\n-----------------------')
print('test missing values: \n', test_df_missing_values)
```

همانطور که در شکل زیر مشخص است، تعداد مقادیر ناموجود در هر ستون از داده‌های آموزش و آزمایش صفر است. بنابراین نیازی به راهکاری برای پرکردن مقادیر ناموجود نیست.

```
train missing values:
 MonsoonIntensity                 0
TopographyDrainage                0
RiverManagement                   0
Deforestation                     0
Urbanization                      0
ClimateChange                     0
DamsQuality                       0
Siltation                         0
AgriculturalPractices             0
Encroachments                     0
IneffectiveDisasterPreparedness   0
DrainageSystems                   0
CoastalVulnerability              0
Landslides                        0
Watersheds                        0
DeterioratingInfrastructure       0
PopulationScore                   0
WetlandLoss                       0
InadequatePlanning                0
PoliticalFactors                  0
FloodProbability                  0
dtype: int64
```

```
test missing values:
 MonsoonIntensity                 0
TopographyDrainage                0
RiverManagement                   0
Deforestation                     0
Urbanization                      0
ClimateChange                     0
DamsQuality                       0
Siltation                         0
AgriculturalPractices             0
Encroachments                     0
IneffectiveDisasterPreparedness   0
DrainageSystems                   0
CoastalVulnerability              0
Landslides                        0
Watersheds                        0
DeterioratingInfrastructure       0
PopulationScore                   0
WetlandLoss                       0
InadequatePlanning                0
PoliticalFactors                  0
FloodProbability                  0
dtype: int64
```

۲-۲-۳- تعیین ابرپارامترهای آموزش از قبیل Learning Rate و Max Epochs

مقدار learning rate = 0.001 و مقدار max epochs = 1000 انتخاب شده است. در ادا

۲-۲-۴- پیاده سازی الگوریتم بهینه سازی SGD برای آموش مدل رگرسیون خطی

در مسیر linear_regression/sgd/sgd.py فایل مربوط به پیاده‌سازی کلاس SGD وجود دارد.

۲-۲-۴-۱- تابع init

در اولین تابع این کلاس، مقادیر اولیه مربوط به ابرپارامترها و ویژگی‌ها به متغیرها نسبت داده می‌شوند.

- نرخ یادگیری (Learning Rate): این ابرپارامتر سرعت به‌روزرسانی وزن‌ها را در هر مرحله از آموزش مشخص می‌کند. اگر نرخ یادگیری خیلی بالا باشد، ممکن است به همگرایی نرسد و از ناحیه مطلوب خارج شود. از طرف دیگر، اگر خیلی پایین باشد، فرآیند یادگیری ممکن است بسیار کند شده و زمان زیادی ببرد تا به نتیجه برسد.
- تعداد اپوک‌ها (Epochs): مشخص می‌کند که کل مجموعه داده چند بار توسط مدل مورد استفاده قرار می‌گیرد. به بیان دیگر، تعداد دفعاتی است که مدل بر روی داده‌های آموزشی آموزش می‌بیند. هر چه تعداد اپوک‌ها بیشتر باشد، مدل فرصت بیشتری برای یادگیری از داده‌ها خواهد داشت، اما این می‌تواند منجر به Overfitting شود که در آن مدل به‌جای یادگیری الگوهای عمومی، به جزئیات نویز در داده‌ها توجه می‌کند.
- اندازه بچ (Batch Size): تعداد نمونه‌هایی را تعیین می‌کند که در هر مرحله از به‌روزرسانی گرادیان مورد استفاده قرار می‌گیرند. استفاده از batch های کوچک می‌تواند به همگرایی سریع‌تر و ایجاد نوسانات مفید در به‌روزرسانی‌ها کمک کند، اما ممکن است باعث افزایش زمان محاسبات شود. از سوی دیگر، batch های بزرگ می‌توانند سرعت محاسبات را افزایش دهند، اما ممکن است به یادگیری بهتر منجر نشوند.
- تحمل (Tolerance): معیاری برای تعیین همگرایی مدل است. به عبارتی، اگر اندازه گرادیان وزن‌ها کمتر از tolerance باشد، مدل متوقف خواهد شد. این ویژگی می‌تواند به جلوگیری از اجرای بی‌پایان الگوریتم کمک کند و زمانی که به حد کافی بهینه‌سازی انجام شده باشد، فرآیند آموزش را متوقف کند.
- مومنتوم (Momentum): یک تکنیک برای بهبود فرآیند یادگیری است که به کمک آن، به‌روزرسانی‌ها به نرمی انجام می‌شوند. این پارامتر مشخص می‌کند که چقدر از به‌روزرسانی‌های قبلی (یعنی گرادیان‌های قبلی) در به‌روزرسانی‌های فعلی وزن‌ها استفاده شود. استفاده از momentum می‌تواند به شتاب بخشیدن به یادگیری در مسیرهای هموار و کاهش نوسانات در نواحی پیچیده کمک کند.
- Self.weights: وزن‌ها پارامترهایی هستند که ورودی‌ها را به خروجی‌ها تبدیل می‌کنند و توسط

مدل یاد گرفته می‌شوند. وزن‌ها در طی فرآیند آموزش به‌روزرسانی می‌شوند تا پیش‌بینی‌ها به بهترین نحو ممکن با مقادیر واقعی تطابق داشته باشند.

- Self.bias: بایاس (Bias) یک پارامتر اضافی است که به مدل اجازه می‌دهد تا حتی زمانی که ورودی‌ها صفر هستند، به یک مقدار غیر صفر نگاشت شود.
- Self.velocity_bias , Self.velocity_weights: این متغیرها نشان‌دهنده سرعت به‌روزرسانی وزن‌ها و بایاس‌ها در طی فرآیند یادگیری هستند. با حفظ یک مقدار برای momentum، این متغیرها کمک می‌کنند تا سرعت به‌روزرسانی‌ها هموارتر و پایدارتر شود.

```
class SGD:
    Farnaz Movahedi *
    def __init__(self, lr=0.001, epochs=1000, batch_size=1024, tol=1e-3, momentum=0.9):
        self.learning_rate = lr
        self.epochs = epochs
        self.batch_size = batch_size
        self.tolerance = tol
        self.momentum = momentum
        self.weights = None
        self.bias = None
        self.velocity_weights = None
        self.velocity_bias = None
```

۲-۲-۴-۲- تابع predict

این تابع از وزن‌ها و بایاس‌های جاری برای پیش‌بینی خروجی استفاده می‌کند. در اینجا، از ضرب داخلی بین X و self.weights استفاده شده و سپس self.bias به نتیجه اضافه می‌شود.

```
Farnaz Movahedi
def predict(self, X):
    return np.dot(X, self.weights) + self.bias
```

۲-۲-۴-۳- تابع gradient

این تابع گرادیان‌های تابع هزینه را نسبت به وزن‌ها و بایاس مدل محاسبه می‌کند. این گرادیان‌ها در به‌روزرسانی پارامترهای مدل در حین آموزش با استفاده از الگوریتم SGD استفاده می‌شوند.

خط اول تابع predict را فراخوانی می‌کند و X_batch جاری را به آن می‌دهد. سپس مقادیر پیش‌بینی‌شده یا همان y_pred را با استفاده از وزن‌ها و بایاس‌های جاری محاسبه می‌کند. error با کم کردن مقادیر واقعی هدف (y_batch) از مقادیر پیش‌بینی‌شده (y_pred) محاسبه می‌شود. این تفاوت نشان می‌دهد

که پیش‌بینی‌ها چقدر از مقادیر واقعی فاصله دارند.

در این مرحله، گرادیان وزن‌ها و بایاس محاسبه می‌شود. ابتدا ضرب داخلی بین ترانهاده ورودی و خطا محاسبه می‌شود. این مقدار یک بردار به ما می‌دهد که نشان‌دهنده این است که هر وزن باید به چه میزان بر اساس خطاهای پیش‌بینی شده به‌روز شود. تقسیم بر X_batch.shape[0] (تعداد نمونه‌ها در batch) میانگین گرادیان را می‌دهد. گرادیان بایاس هم با محاسبه میانگین خطاها به دست می‌آید. این نشان می‌دهد که بایاس بر اساس میانگین خطای پیش‌بینی باید به چه میزان در سراسر batch به‌روز شود.

در نهایت، این تابع گرادیان‌های محاسبه‌شده برای وزن‌ها و بایاس را بر می‌گرداند. این گرادیان‌ها در مرحله به‌روزرسانی بعدی استفاده خواهند شد.

```
Farnaz Movahedi
def gradient(self, X_batch, y_batch):
    y_pred = self.predict(X_batch)
    error = y_pred - y_batch
    gradient_weights = np.dot(X_batch.T, error) / X_batch.shape[0]
    gradient_bias = np.mean(error)
    return gradient_weights, gradient_bias
```

۲-۲-۴-۴- تابع fit

عملیات آموزش به صورت دقیق در این تابع اتفاق می‌افتد. n_samples تعداد نمونه‌های داده و n_features تعداد ویژگی‌های هر نمونه است. در ابتدا وزن‌های مدل به‌صورت تصادفی و با مقادیرکوچک مقداردهی می‌شوند تا تاثیر یک ویژگی خاص در ابتدا زیاد نباشد. مقدار بایاس هم صفر تنظیم می‌شود.

Self.velocity_weights یک بردار برای ذخیره سرعت به‌روزرسانی‌های مربوط به وزن‌ها در مدل است. مقدار اولیه این بردار تماما صفر بوده و به این معنا است که در ابتدای فرآیند آموزش، سرعت اولیه‌ای وجود ندارد. این متغیر مقادیر قبلی به‌روزرسانی‌های وزن‌ها را ذخیره و در فرایند آموزش از آنها استفاده می‌کند تا از نوسانات شدید جلوگیری کرده و به تسریع روند همگرایی کمک کند. Self.velocity_bias هم مشابه همین توضیحات را برای بایاس عملی می‌کند.

Self.losses یک لیست برای ذخیره مقادیر خطای هر epoch است تا برای تحلیل و نمایش از آن استفاده شود.

```
def fit(self, X, y):
    # Ensure that X and y are numpy arrays
    X = np.array(X)
    y = np.array(y)

    n_samples, n_features = X.shape
    self.weights = np.random.randn(n_features) * 0.01  # Small random weights
    self.bias = 0  # Initialize bias with 0

    # Initialize velocities
    self.velocity_weights = np.zeros(n_features)
    self.velocity_bias = 0

    # List to store loss for each epoch
    self.losses = []
```

فرآیند آموزش برای تعداد مشخصی از epochs اجرا می‌شود. هر epoch یک بار مرور کامل بر داده‌های آموزشی است. در ادامه مشاهده می‌شود که داده‌ها در ابتدای هر دوره به‌صورت تصادفی درهم‌ریخته می‌شوند تا مدل از ترتیب ثابت داده‌ها الگو نگیرد و بایاس ایجاد نشود. در ادامه یک حلقه دیگر وجود دارد که در آن به اندازه batch_size داده از مجموعه کل داده‌ها برداشته می‌شود.

در شروع روند آموزش، گرادیان نسبت به وزن‌ها و بایاس با استفاده از batch فعلی محاسبه می‌شود. در ادامه سرعت به‌روزرسانی وزن‌ها و بایاس با استفاده از momentum و lr محاسبه شده و در نهایت این مقادیر به‌روزرسانی می‌شوند.

در انتهای هر epoch، مدل روی کل داده‌ها پیش‌بینی را انجام می‌دهد و خطای میانگین مربعات محاسبه می‌شود. مقدار خطا در لیست self.losses ذخیره می‌شود تا عملکرد مدل در طول زمان پیگیری شود.

```
for epoch in range(self.epochs):
    indices = np.random.permutation(n_samples)
    X_shuffled = X[indices]
    y_shuffled = y[indices]

    for i in range(0, n_samples, self.batch_size):
        X_batch = X_shuffled[i:i + self.batch_size]
        y_batch = y_shuffled[i:i + self.batch_size]

        gradient_weights, gradient_bias = self.gradient(X_batch, y_batch)

        # Update velocities
        self.velocity_weights = self.momentum * self.velocity_weights - self.learning_rate * gradient_weights
        self.velocity_bias = self.momentum * self.velocity_bias - self.learning_rate * gradient_bias

        # Update weights and bias
        self.weights += self.velocity_weights
        self.bias += self.velocity_bias

    # Calculate and store loss at the end of each epoch
    y_pred = self.predict(X)
    loss = mean_squared_error(y, y_pred)
    self.losses.append(loss)  # Store each epoch's loss
```

پس از گذشت مقدار مناسبی از epoch ها، مدل مقادیرloss و R^2 را چاپ می‌کند.

حلقه آموزش بررسی می‌کند که آیا اندازه گرادیان کمتر از یک مقدار مشخص (self.tolerance) شده است یا خیر. اگر این‌طور باشد، آموزش متوقف می‌شود، به این معنا که به‌روزرسانی‌های بیشتر احتمالاً بهبود قابل‌توجهی ایجاد نمی‌کنند.

بعد از اتمام آموزش، خطا در طول دوره‌ها رسم می‌شود تا همگرایی مدل به‌صورت تصویری بررسی شود.

```
    if epoch % 100 == 0:
        r2 = r2_score(y, y_pred)
        print(f"Epoch {epoch}: Loss {loss}, R² {r2}")

    if np.linalg.norm(gradient_weights) < self.tolerance:
        print("Convergence reached.")
        break

# Plot the loss over epochs
plt.plot( *args: range(len(self.losses)), self.losses, "b-", linewidth=1)
plt.xlabel("Epoch")
plt.ylabel("Loss (MSE)")
plt.title("Loss function over epochs")
plt.grid(True)
plt.show()

return self.weights, self.bias
```

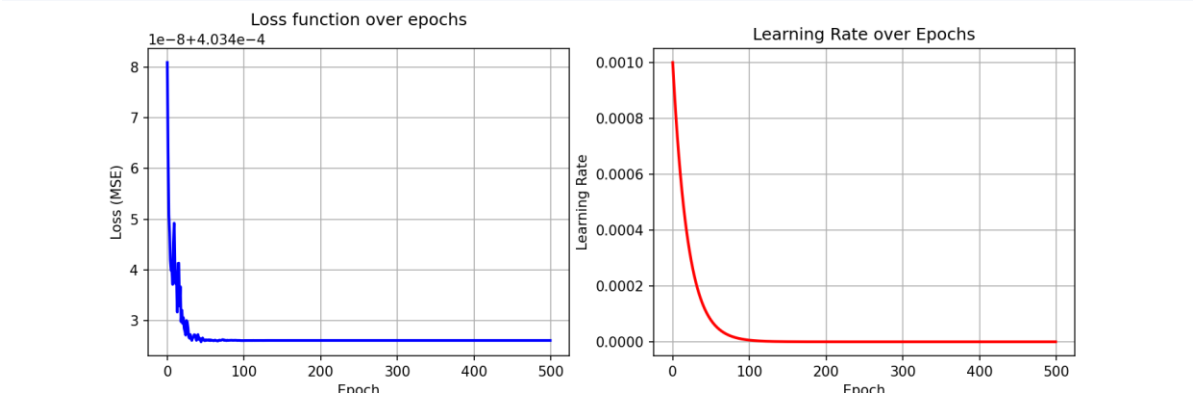
۲-۲-۵- آموزش مدل با مجموعه داده آموزش

در مسیر linear_regression/main.py فایل مربوط به پیاده‌سازی این بخش وجود دارد.

ابتدا داده‌ها از فایل‌های train.csv و test.csv خوانده شده و سپس با استفاده از تابع preprocess_data() پیش‌پردازش می‌شوند تا ویژگی‌ها و متغیر هدف جدا شوند. پس از آن، ویژگی‌ها با تابع normalize_features() نرمال‌سازی می‌شوند تا مقادیر آن‌ها در محدوده مشابهی قرار گیرند و کارایی مدل بهبود یابد.

در مرحله بعد، یک نمونه از کلاس SGD با تنظیمات موردنظر ایجاد می‌شود و مدل با داده‌های آموزشی آموزش می‌بیند. پس از آموزش، با استفاده از داده‌های آزمایش، پیش‌بینی‌ها انجام می‌شود و عملکرد مدل با معیارهای مختلفی نظیر خطای میانگین مربعات، خطای مطلق میانگین و ضریب تعیین ارزیابی می‌شود.

۲-۲-۶- رسم نمودار تغییر مقادیر تابع زیان و نرخ یادگیری در هنگام آموزش برحسب epoch



۲-۲-۷- ارزیابی مدل با مجموعه داده آزمایش

```
Epoch 0: Loss 0.00040349711683330047, R² 0.8451278136819529
Epoch 10: Loss 0.00040351534784699775, R² 0.8451208161673925
Epoch 20: Loss 0.00040345219871167971, R² 0.8451450543694518
Epoch 30: Loss 0.00040343514834868346, R² 0.8451515987063968
Epoch 40: Loss 0.00040343134150349014, R² 0.8451530598825043
Epoch 50: Loss 0.00040342738162454125, R² 0.8451545797821141
Epoch 60: Loss 0.00040342613685424445, R² 0.8451550575557815
Epoch 70: Loss 0.00040342643864866225, R² 0.8451549417120637
Epoch 80: Loss 0.00040342607727195564, R² 0.8451550804450516
Epoch 90: Loss 0.00040342608317231617, R² 0.845155078160235
Test Loss: 0.00040432877901050008, R² Score: 0.8444483091750770
PS D:\Git Projects\search-and-machine-learning-fakesmart>
```

۲-۲-۶- بخش‌های امتیازی

ما در ادامه علاوه بر بخش‌های مومنتم و استفاده از توقف زودهنگام در بهینه سازی، بخش تغییر نرخ یادگیری و استفاده از Regularization را نیز اضافه کردیم.

۲-۲-۶-۱- تغییر نرخ یادگیری

برای تغییر نرخ یادگیری از Decay rate استفاده شده. این استراتژی برای جلوگیری از افزایش بیش از

حد نرخ یادگیری و بهبود همگرایی مدل استفاده می‌شود. در اینجا، با هر اپک، نرخ یادگیری اولیه با ضرب شدن در decay_rate کاهش می‌یابد.

```
for epoch in range(self.epochs):
    # Update learning rate with decay
    self.learning_rate = self.initial_lr * (self.decay_rate ** epoch)
```

Regularization -۲-۶-۲-۲

ما از L2 regularization استفاده کردیم. L2 regularization (همچنین به نام regularization Ridge) به جلوگیری از افزایش بیش از حد وزن‌ها و جلوگیری از بیش‌برازش کمک می‌کند. این کار با اضافه کردن جمله‌ای به گرادیان که متناسب با وزن‌ها باشد انجام می‌شود.

```
def gradient(self, X_batch, y_batch):
    y_pred = self.predict(X_batch)
    error = y_pred - y_batch
    gradient_weights = (np.dot(X_batch.T, error) / X_batch.shape[0]) + (self.l2_lambda * self.weights)
    gradient_bias = np.mean(error)
    return gradient_weights, gradient_bias
```

منابع

[1] "andreaperlato," [Online]. Available: https://www.andreaperlato.com/theorypost/the-learning-rate/#:~:text=The%20range%20of%20values%20to,starting%20point%20on%20your%20problem.

[2] "Medium," [Online]. Available: https://chandhana520.medium.com/implementing-sgd-stochastic-gradient-descent-for-linear-regression-1a82cddbb36b.

[3] "geeksforgeeks," [Online]. Available: https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/.

[4] "realpython," [Online]. Available: https://realpython.com/gradient-descent-algorithm-python/.

[5] "youtube," [Online]. Available: https://www.youtube.com/watch?v=6TsL96NAZCo&t=373s.

[6] "stanford," [Online]. Available: https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.

[7] "finxter," [Online]. Available: https://blog.finxter.com/python-a-the-simple-guide-to-the-a-star-search-algorithm/.

[8] "brilliant," [Online]. Available: https://brilliant.org/wiki/a-star-search/.