دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

گزارش پروژه مبانی هوش_فاز ۱

مسائل جست و جو و رگرسیون خطی

استاد:

دکتر حسین کارشناس

دستیار آموزشی:

پوریا صامتی

اعضا گروه:

سپهر فاطمی

پوریا اردستانی

شیما مغزی

مهر ۱۴۰۳

فهرست مطالب

عنوان صفحه

۱- بازی پک‌من

در این قسمت از پروژه با استفاده از الگوریتم‌های مختلف جست و جو مسیر پک‌من برای رسیدن به غذاها در مازهای مختلف را پیدا می‌کنیم

۱-۱- جست و جو عمق اول[1]

در این الگوریتم برای پیاده‌سازی DFS (depth first search)، از یک لیست برای ذخیره سازی حالات دیده شده برای جلوگیری از ایجاد حلقه در حین جست و جو و همچنین از یک لیست به عنوان frontier استفاده شده. این الگوریتم یک الگوریتم کامل اما غیربهینه است.

```
def depthFirstSearch(problem):
    currentState = problem.getStartState()
    Path = []
    if problem.isGoalState(currentState):
        return Path

    frontier = []
    visited = []
    frontier.append((currentState, Path))
    while not len(frontier) == 0:
        currentState, Path = frontier.pop()
        if problem.isGoalState(currentState):
            return Path
        visited.append(currentState)
        for successors in problem.getSuccessors(currentState):
            if successors[0] not in visited:
                frontier.append((successors[0], Path + [successors[1]]))
```

نتایج این جست و جو برای محیط‌های hardCorner و bigCorner در ادامه آمده است.

[SearchAgent] using function dfs

Hard Corner Problem

Path found with total cost of 221 in 0.0 seconds

Search nodes expanded: 378

Pacman emerges victorious! Score: 319

Record: Win

[SearchAgent] using function dfs

Big Corner Problem

Path found with total cost of 316 in 0.0 seconds

Search nodes expanded: 1015

Pacman emerges victorious! Score: 234

Record: Win

۱-۲- جست و جو با هزینه یکنواخت[2]

در این الگوریتم مسیریابی با در نظر گرفتن هزینه مسیر تا حالت فعلی انجام می‌شود، به طوریکه در انتخاب حالت بعدی حالتی که باعث ایجاد کمترین هزینه می‌شود انتخاب خواهد شد. به همین سبب برای پیاده‌سازی frontier در این الگوریتم از صف اولویت دار استفاده می‌شود و برای پیدا کردن بهترین هزینه برای هر نود از یک دیکشنری جهت

[1] Depth First Search
[2] Uniform Cost Search

ذخیره‌ی حالات دیده شده استفاده شده است تا در صورت بر خوردن به حالت تکراری زمانی که هزینه حالت با مسیر فعلی کمتر است، حالت تکراری دوباره به frontier اضافه شود. این جست و جو کامل و بهینه است.

```
visited = {}
frontier = util.PriorityQueue()
current_state = problem.getStartState()
frontier.push((current_state, [],0 ), 0)
visited[current_state] = 1

while not frontier.isEmpty():
    current_state, path, current_cost = frontier.pop()
    if problem.isGoalState(current_state):
        return path
    if visited[current_state] < current_cost:
        continue

    for state, action, new_cost in problem.getSuccessors(current_state):
        state_cost = new_cost + current_cost
        if state not in visited.keys() or visited[state] > state_cost:
            visited[state] = state_cost
            frontier.push((state, path+[action],state_cost), state_cost)
return path
```

نتایج این جست و جو برای محیط‌های hardCorner و bigCorner در ادامه آمده است.

[SearchAgent] using function ucs

Hard Corner problem

Path found with total cost of 106 in 0.0 seconds

Search nodes expanded: 1908

Pacman emerges victorious! Score: 434

Record: Win

[SearchAgent] using function ucs

Big Corner Problem

Path found with total cost of 210 in 0.1 seconds

Search nodes expanded: 11128

Pacman emerges victorious! Score: 340

Record: Win

۱-۳- جست و جو A*

در این الگوریتم علاوه بر در نظر گرفتن هزینه مسیر تا حالت فعلی از یک تابع اکتشافی ک هزینه مسیر تا حالت هدف را به طور حدودی مشخص می‌کند نیز استفاده می‌شود. ساختار frontier و ذخیره‌سازی حالات دیده شده همانند الگوریتم UCS(uniform cost search) است. برای ذخیره‌سازی حالات دیده‌شده از یک دیکشنری استفاده شده که در آن کلیدهای دیکشنری حالات و مقادیر هزینه آنها در زمانی که دیده شده اند است، این ساختار کمک می‌کند که در صورتی که یک حالت با هزینه کمتر از هزینه فعلی قبلا دیده شده بود دیگر بررسی نشود و وارد frontier نشود.این الگوریتم یک الگوریتم کامل و بهینه است.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    startState = problem.getStartState()
    frontier = util.PriorityQueue()
    visited = {}
    frontier.push((startState, [], 0), heuristic(startState, problem))
    visited[startState] = 0
    while not frontier.isEmpty():
        currentState, path, current_cost = frontier.pop()
        if problem.isGoalState(currentState):
            return path
```

```
        if visited[currentState] < current_cost:
            continue
        for successor, action, step_cost in
problem.getSuccessors(currentState):
            new_cost = current_cost + step_cost
            if successor not in visited or new_cost < visited[successor]:
                visited[successor] = new_cost
                priority = new_cost + heuristic(successor, problem)
                frontier.push((successor, path + [action], new_cost),
priority)
    return []
```

برای پیاده‌سازی تابع اکتشافی ابتدا مختصات‌های اهداف خورده نشده در حالت فعلی را پیدا می‌کنیم، سپس فاصله منهتن حالت فعلی تا اهداف خورده نشده را محاسبه می‌کنیم و نزدیک‌ترین هدف خورده نشده را پیدا می‌کنیم. سپس یک درخت پوشای کمینه با ریشه اولین هدف در لیست اهداف خورده نشده می‌سازیم و هزینه این درخت پوشا را بر حسب فاصله منهتن بین اهداف در این درخت محاسبه می‌کنیم، فرایند ساخت این درخت پوشا به این صورت است که ابتدا هر ترکیب دوتایی ممکن از اهداف خورده نشده را پیدا می‌کنیم و سپس فاصله هر منهتن هر دوتایی از این ترکیبات را محاسبه می‌کنیم، با مرتب کردن این فواصل یک لیست connected می‌سازیم که شامل یک حال هدف اولیه است و بقیه حالات از لیست فواصل به ترتیب به connected اضافه می‌شوند. مقدار نهایی تابع اکتشافی هزینه رسیدن از حالت فعلی به نزدیک‌ترین هدف خورده نشده به علاوه هزینه این درخت پوشا است.

```
def cornersHeuristic(state, problem):
    x, y = state[0]
    corner_with_coordinates = zip(problem.corners, state[1])
    uneaten_corners = [corner for corner, eaten in corner_with_coordinates
if not eaten]

    if not uneaten_corners:
        return 0

    distances = [util.manhattanDistance((x, y), corner) for corner in
uneaten_corners]
    nearest_corner_distance = min(distances)
    if len(uneaten_corners)==1 : return len(uneaten_corners)
    all_combinations = combinations(uneaten_corners, 2)
    edges = [(util.manhattanDistance(c1, c2), c1, c2) for c1, c2 in
all_combinations]
    edges.sort()

    mst_cost = 0
    connected =[uneaten_corners[0]]

    while len(connected) < len(uneaten_corners):
        for distance, corner1, corner2 in edges:
            if (connected[-1]==corner1 or  connected[-1]==corner2 ) :
                if(corner2 in connected )^( corner2 in connected ):
                    mst_cost += distance
                    connected.update([corner1, corner2])
                    break

    heuristic = nearest_corner_distance + mst_cost
    return heuristic
```

نتایج این جست و جو برای محیط‌های hardCorner و bigCorner در ادامه آمده است.

[SearchAgent] using function astar and heuristic cornersHeuristic

Hard Corner Problem

Path found with total cost of 106 in 0.0 seconds

Search nodes expanded: 731

Pacman emerges victorious! Score: 434

Record: Win

[SearchAgent] using function astar and heuristic cornersHeuristic

Big Corner Problem

Path found with total cost of 210 in 0.0 seconds

Search nodes expanded: 2507

Pacman emerges victorious! Score: 340

Record: Win

۲- رگرسیون خطی

برای پیش‌بینی احتمال وقوع سیل، یک SGD را پیاده سازی می‌کنیم و با استفاده از آن یک مدل رگرسیون خطی را با داده‌های train، آموزش می‌دهیم.

۲-۱- ماژول های استفاده شده

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error, mean_absolute_error
```

۲-۲- پیاده سازی تابع SGD

ابتدا وزن‌ها و بایاس با مقدار صفر مقداردهی اولیه می‌شوند. در هر epoch، یک نمونه تصادفی از داده‌ها به اندازه‌ی Batch size انتخاب می‌شود و گرادیان‌ها محاسبه می‌شوند. گرادیان‌های محاسبه شده به وزن‌ها و بایاس اعمال می‌شوند تا مدل بهینه‌سازی شود. پس از هر تکرار،میتوان مقدار learning_rate را کاهش داد (با پارامتر divideby) و خطا برای آن epoch ذخیره می‌شود.

```
def SGD(train_data, learning_rate, epoch, k, divideby):
    w = np.zeros(shape=(1, train_data.shape[1] - 2))
    b = 0
    current_iter = 1
    losses = []
```

سپس وارد حلقه اصلی میشویم؛ در هر epoch یک نمونه تصادفی از داده‌های train به اندازه k که همان Batch size است انتخاب میکنیم. سپس به جز FloodProbability و id ، باقی فیچرها را در x میریزیم و فیچر تارگت یا همان FloodProbability را در y قرار می‌دهیم.

```
while current_iter <= epoch:
    temp = train_data.sample(k)
    y = np.array(temp['FloodProbability'])
    x = np.array(temp.drop(['FloodProbability', 'id'], axis=1))

    w_gradient = np.zeros_like(w)
    b_gradient = 0
    loss = 0
```

برای پیشبینی، محاسبه گرادیان و loss function از فرمول های زیر استفاده می‌کنیم:

مقدار خطا:

$$\min_{w,b} \sum_{i=1}^{n} (\underbrace{\text{Actual Value}}_{(y_i)} - \underbrace{\text{Predicted Value}}_{(\bar{y}_i)})^2$$

محاسبه مقدار Prediction:

$$Prediction\ (\hat{y}_i) = w^T x_i + b$$

function Loss:

$$\mathcal{L}(w,b) \longrightarrow \min_{w,b} \sum_{i=1}^{n} \left(y_i - (w^T x_i + b)\right)^2$$

```
    for i in range(k):
        Prediction = np.dot(w, x[i]) + b
        error = y[i] - Prediction
        w_gradient += (-2) * x[i] * error
        b_gradient += (-2) * error
        loss += error ** 2

    losses.append(loss / k)

    w -= learning_rate * (w_gradient / k)
    b -= learning_rate * (b_gradient / k)

    learning_rate /= divideby
    current_iter += 1

return w, b, losses
```

با توجه به فرمول‌های زیر، مقدار گرادیان را برای بهینه سازی وزن و بایاس، بروزرسانی می‌کنیم:

$$\frac{\partial \mathcal{L}(w)}{\partial w} = \sum_{i=1}^{n} (-2x_i)\left(y_i - (w^T x_i + b)\right)$$

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^{n} (-2)\left(y_i - (w^T x_i + b)\right)$$

سپس وزن و بایاس را با استفاده از مقدار گرادیان و نرخ یادگیری، آپدیت می‌کنیم. همچنین میتوان در پایان هر epoch، نرخ یادگیری را با مقداری معینی کاهش داد تا همگرایی بهتری داشته باشیم.

۲-۳- پیاده سازی تابع Predict

این تابع برای پیش‌بینی خروجی‌های مدل روی دیتاهای test استفاده می‌شود. برای هر نمونه در x، مقدار پیش‌بینی‌شده را محاسبه و در y_pred ذخیره می‌کند.

```
def Predict(x, w, b):
    y_pred = []
    for i in range(len(x)):
        y = (np.dot(w, x[i]) + b).item()
        y_pred.append(y)
    return np.array(y_pred)
```

۲-۴- پیاده سازی Feature Scaling

تابع Scale داده‌ها را به بازه [0, 1] مقیاس‌بندی می‌کند. Scale را روی همه ستون ها به جز ستون اول و آخر داده‌ها (که id و FloodProbability می‌باشند) اعمال می‌کنیم.

```
def Scale(data):
    scaled_data = data.copy()
    for column in data.columns[1:-1]:
        col_min = data[column].min()
        col_max = data[column].max()
        scaled_data[column] = (data[column] - col_min) / (col_max -
col_min)
    return scaled_data
```

۲-۵- پیش پردازش داده‌ها[3]

برای پیش پردازش داده‌ها ابتدا داده‌های train و test از فایل‌های CSV خوانده می‌شوند. سپس عملیات زیر بر روی داده‌ها انجام می‌شود:

- **جای‌گذاری مقادیر ناموج:** ستون‌هایی که دارای مقدار نیستند، با میانگین مقدار ستون جایگزین می‌شوند تا مشکل عدم وجود برخی از مقادیر حل شود. برای این کار از تابع fillna ماژول pandas استفاده می‌کنیم.
- **Feature Scaling:** مقادیر فیچرها به بازه‌ی [0, 1] مقیاس‌بندی می‌شوند تا مدل نسبت به تغییرات مقادیر دقت بیشتری داشته باشد. این کار با تابع Scale انجام می‌شود.

```
train_data = pd.read_csv("train.csv")
test_data = pd.read_csv("test.csv")

for column in train_data.columns:
    if train_data[column].isnull().sum() > 0:
        train_data[column].fillna(train_data[column].mean(), inplace=True)

for column in test_data.columns:
    if test_data[column].isnull().sum() > 0:
        test_data[column].fillna(test_data[column].mean(), inplace=True)

train_data = Scale(train_data)
test_data = Scale(test_data)
```

[3]Data PreProcessing

۲-۶- حذف ویژگی ها

فیچرهای موردنظر مثل id و FloodProbability را از داده‌های train و test حذف‌کرده و FloodProbability را به عنوان هدف قرار می‌دهیم.

```
x_train = train_data.drop(['FloodProbability', 'id'], axis=1).values
y_train = train_data['FloodProbability'].values
x_test = test_data.drop(['FloodProbability', 'id'], axis=1).values
y_test = test_data['FloodProbability'].values

train_data['FloodProbability'] = y_train
```

۲-۷- آموزش مدل

پارامتر های learning rate ، epoch ، batch size و divideby را به صورت زیر تعریف می‌کنیم و مدل را با داده های train آموزش می‌دهیم. قرار دادن پارامترها به صورت زیر منجر به دستیابی به دقت موردنظر مدل شد.

```
learning_rate = 0.06
epoch = 1500
k = 32
divideby = 1

w, b, losses = SGD(train_data, learning_rate, epoch, k, divideby)
```

۲-۸- نمودار Loss Function

با استفاده از مقادیر losses که در طول آموزش ذخیره کردیم، میتوانیم تغییر مقادیر تابع زیان در طول آموزش برحسب epoch را رسم کنیم.

```
plt.plot(range(len(losses)), losses, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss Function')
plt.legend()
plt.show()
```

۲-۹- ارزیابی مدل

برای ارزیابی مدل، ابتدا مقادیر prediction مدل را روی داده های test و train به دست می‌آوریم؛ و سپس با استفاده از توابع ماژول scikit learn، score های MSE و MAE را برای داده‌های آموزش و تست محاسبه می‌کنیم.

```
y_train_pred = Predict(x_train, w, b)
y_test_pred = Predict(x_test, w, b)

train_mse = mean_squared_error(y_train, y_train_pred)
train_mae = mean_absolute_error(y_train, y_train_pred)

test_mse = mean_squared_error(y_test, y_test_pred)
test_mae = mean_absolute_error(y_test, y_test_pred)

print(f"Train MSE: {train_mse}")
print(f"Train MAE: {train_mae}")
print(f"Test MSE: {test_mse}")
print(f"Test MAE: {test_mae}")
```

برای محاسبه امتیاز R^2 نیز با توجه به فرمول زیر، این score را با تابع پیاده‌سازی شده r2_score به دست می‌آوریم.

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

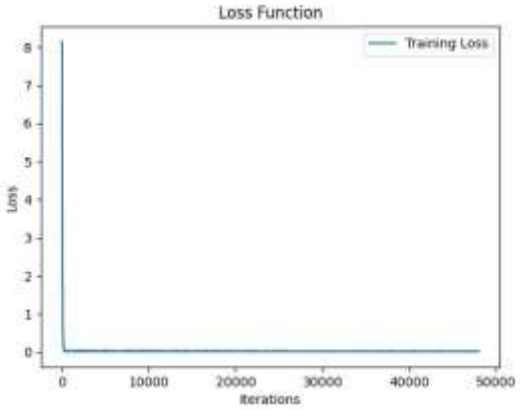
```
def r2_score(y_true, y_pred):
    y_mean = np.mean(y_true)

    ss_res = np.sum((y_true - y_pred) ** 2)
    ss_tot = np.sum((y_true - y_mean) ** 2)

    r2 = 1 - (ss_res / ss_tot)
    return r2

train_r2 = r2_score(y_train, y_train_pred)
test_r2 = r2_score(y_test, y_test_pred)

print(f"Train R^2 Score: {train_r2}")
print(f"Test R^2 Score: {test_r2}")
```

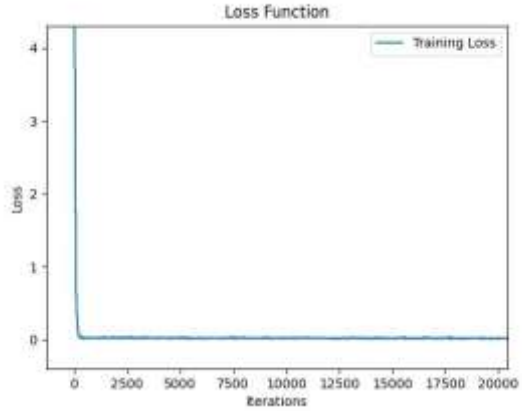
۲-۱۰- نتایج ارزیابی مدل

با قرار دادن هایپرپارامترها به صورت زیر:

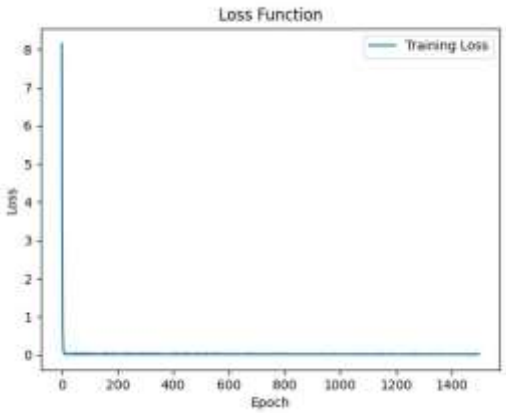
```
learning_rate = 0.06
epoch = 1500
k = 32
divideby = 1
```

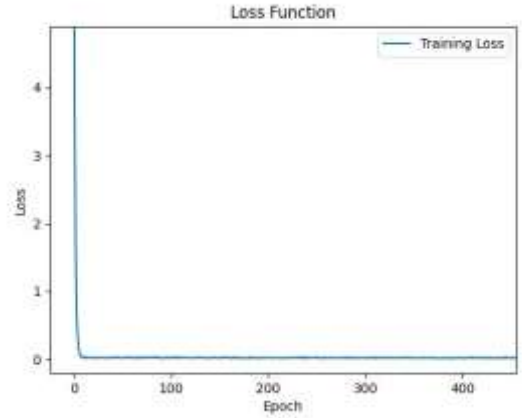
نمودار loss function بر حسب iteration به صورت زیر می‌باشد:





و این نمودار بر حسب epoch به صورت زیر است.





همچنین score های مدل به صورت زیر می‌باشند:

Train MSE: 0.0004805239530324092

Train MAE: 0.01791193371296469

Test MSE: 0.0004748998488591834

Test MAE: 0.018078536284652624

Train R^2 Score: 0.8155630063273405

Test R^2 Score: 0.8124548749787326

مراجع

https://chatgpt.com :)

https://medium.com/@freskoinnovationlabs/data-preprocessing-with-numpy-36bc21fc0fa7#:~:text=NumPy's%20max%20and%20min%20functions,max()

https://medium.com/@nikhilparmar9/simple-sgd-implementation-in-python-for-linear-regression-on-boston-housing-data-f63fcaaecfb1