ЛЕКЦІЯ
ТИПИ РЕГРЕСІЙ. МАШИННЕ НАВЧАННЯ. РЕГУЛЯРИЗАЦІЯ. ПРАКТИЧНА РЕАЛІЗАЦІЯ

Лінійні моделі для класифікації:

Логістична регресія.

Завдання класифікації є одним із найважливіших і поширених завдань. Її основна мета полягає у поділі даних на класи відповідно до заданих ознак. Лінійні моделі є одним із найбільш популярних підходів до вирішення задачі класифікації.

В даній лекції ми розглянемо, як використовувати лінійні моделі Python для вирішення задачі класифікації.

Лінійна модель - це математична модель, яка є лінійною комбінацією вхідних ознак. У задачі класифікації лінійна модель використовується для поділу даних на два або більше класів.

Python надає багато бібліотек для роботи з лінійними моделями. Одна з найпопулярніших бібліотек для роботи з лінійними моделями в Python це **scikit-learn**. Scikit-learn надає безліч алгоритмів машинного навчання, у тому числі й лінійні моделі для класифікації. Однією з таких моделей є логістична регресія.

Логістична регресія - це алгоритм машинного навчання, який використовується для вирішення задачі бінарної класифікації, тобто поділу даних на два класи. Вона отримала свою назву завдяки тому, що використовує логістичну функцію для прогнозування ймовірності приналежності об'єкта одного з класів.

Логістична регресія використовує лінійну комбінацію вхідних ознак та відповідних ваг, яка описує лінійну гіперплощину у просторі ознак. Потім цей результат проходить через логістичну функцію, яка переводить лінійну комбінацію на ймовірність приналежності об'єкта до одного з класів.

По суті логістична регресія просто використовує рівняння лінійної регресії і застовує його як параметр сигмовидної функції. Математично це виражається наступним чином:

$$P(Y=1|X)=\frac{1}{1+e^{-\beta_0-\beta_1X_1-\beta_2X_2-\ldots-\beta_pX_p}} \quad (1)$$

де:

Y - бінарний вихідний результат (0 або 1);

X - вектор ознак, який використовується для прогнозування *Y*;

P(Y=1|X) — ймовірність того, що *Y* дорівнює 1 при заданому *X*;

beta0, *beta1*, *beta2*, ..., *betap* - коефіцієнти моделі, які потрібно визначити в ході навчання, щоб досягти найкращої відповідності даних

e - число Ейлера.

Логістична регресія також може бути використана для багатокласової класифікації, коли необхідно розділити дані на більш ніж два класи. Для цього навчають K моделей, кожна з яких відрізняється лише цільовим класом. По суті, завдання бінарної класифікації вирішується кілька разів і видається сукупне рішення декількох моделей.

В цілому, логістична регресія – це потужний інструмент для вирішення завдань бінарної та багатокласової класифікації у Python.

Вона проста у використанні та надає безліч метрик для оцінки якості роботи моделі.

Реалізація циклу навчання логістичної регресії у Python.

Реалізація циклу навчання логістичної регресії за допомогою Python і бібліотеки PyTorch.

Імпортуємо всі необхідні бібліотеки:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from sklearn.datasets import
make_classification
from sklearn.metrics import
classification_report
```

Далі напишемо клас, який реалізує логістичну регресію. Варто звернути увагу, що від лінійної регресії, відрізняється лише застосуванням сигмоїди та новим методом **predict** (оскільки тепер ми вирішуємо завдання класифікації).

```
class LogisticRegression(nn.Module):
    def __init__(self, input_size):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(input_size, 1))
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = x @ self.weights
        x = self.sigmoid(x)
        return x

    def fit(self, X, y, lr=0.01, num_iterations=1000):
        X = torch.from_numpy(X).float()
        y = torch.from_numpy(y).float().view(-1, 1)
# Ініціалізуємо функцію втрат та оптимізатор
            criterion = nn.BCELoss()
        optimizer = optim.SGD(self.parameters(), lr=lr)
```

```
for epoch in range(num_iterations):
# Занулюємо градієнти
        optimizer.zero_grad()
# Отримуємо передбачення моделі та обчислюємо функцію втрат
        y_pred = self(X)
        loss = criterion(y_pred, y)
       # Оновлюємо ваги
         loss.backward()
         optimizer.step()
    def predict(self, X):
         X = torch.from_numpy(X).float()

# Отримуємо передбачення моделі та присвоюємо мітки класів на основі
ймовірності
         y_pred = self(X)
         y_pred_labels = [1 if i > 0.5 else 0 for i in
y_pred.detach().numpy().flatten()]

         return y_pred_labels
```

Згенеруємо вибірку для класифікації самостійно, використовуючи **make_classification** із бібліотеки **scikit-learn**. А далі навчимо нашу модель і оцінимо її якість:

```
# Генеруємо дані
X, y = make_classification(n_samples=1000,
n_features=2, n_redundant=0,
n_informative=2,random_state=1,
n_clusters_per_class=1)
# Створюємо екземпляр класу та навчаємо на навчальній
вибірці
model = LogisticRegression(X.shape[1])
model.fit(X, y, lr=0.1, num_iterations=100)
# Прогнозуємо мітки класів на тестовій вибірці
y_pred = model.predict(X)
print(classification_report(y, y_pred))
```

OUT:

	precision	recall	f1-score	support
0	0.85	0.96	0.90	500
1	0.95	0.83	0.88	500
accuracy			0.89	1000
macro avg	0.90	0.89	0.89	1000
weighted avg	0.90	0.89	0.89	1000

Для «чистоти експерименту» навчимо ***логістичну регресію*** з бібліотеки scikit-learn і побачимо, що якість отриманих моделей приблизно однакова:

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X, y)
y_pred = model.predict(X)

print(classification_report(y, y_pred))

OUT:
 precision    recall  f1-score   support

           0       0.90      0.90      0.90       500
           1       0.90      0.90      0.90       500

    accuracy                           0.90      1000
   macro avg       0.90      0.90      0.90      1000
weighted avg       0.90      0.90      0.90      1000
```

Оцінимо візуально як модель приймає своє рішення Рис.1:

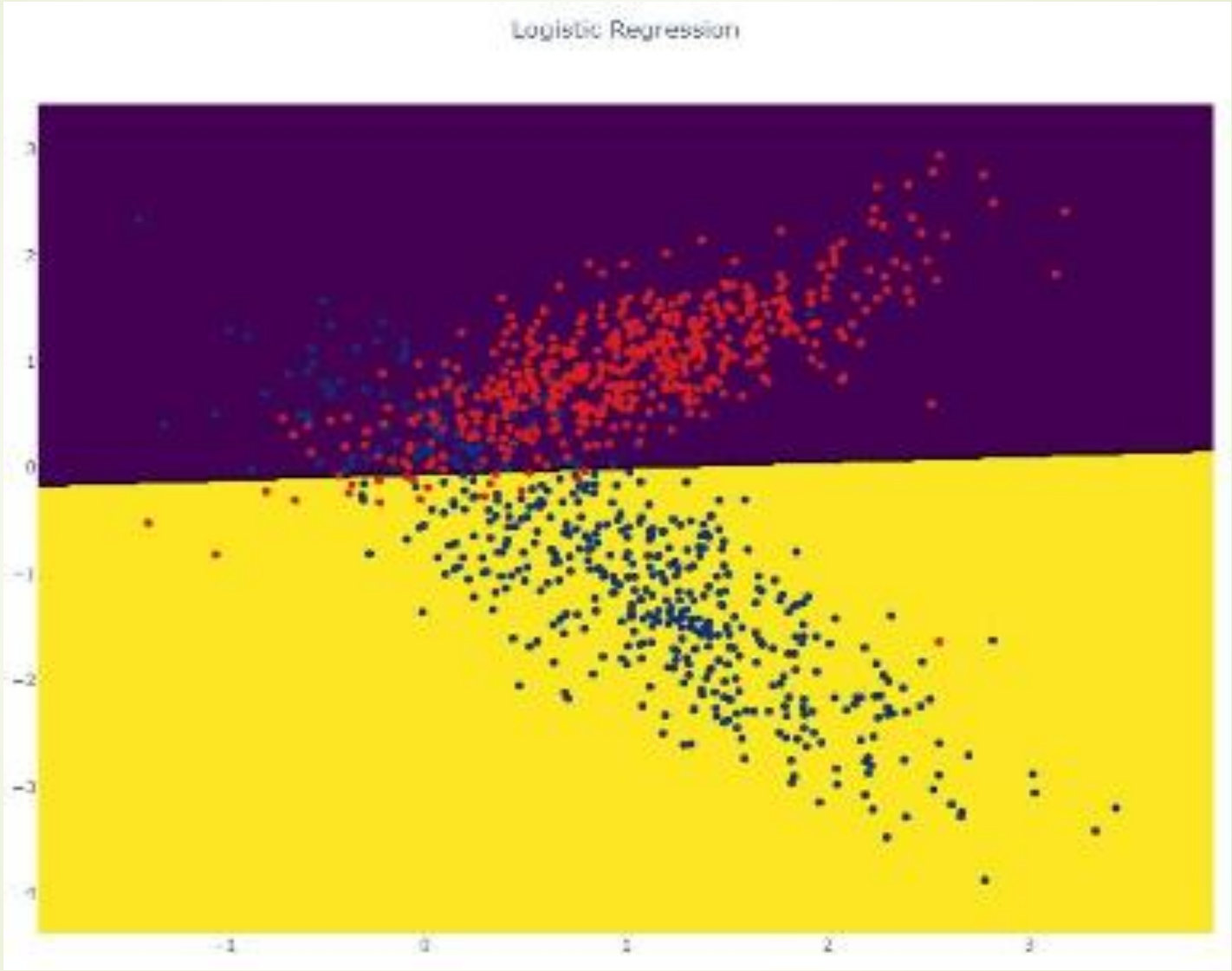


Рис.1. Рішення логістичної регресії.

Як бачимо, результатом роботи алгоритму виступає лінія, що розділяє класи. Якби ми візуалізували модель у процесі навчання **градієнтним спуском**, то побачили б, як ця лінія підбирається у процесі оптимізації Рис.2:

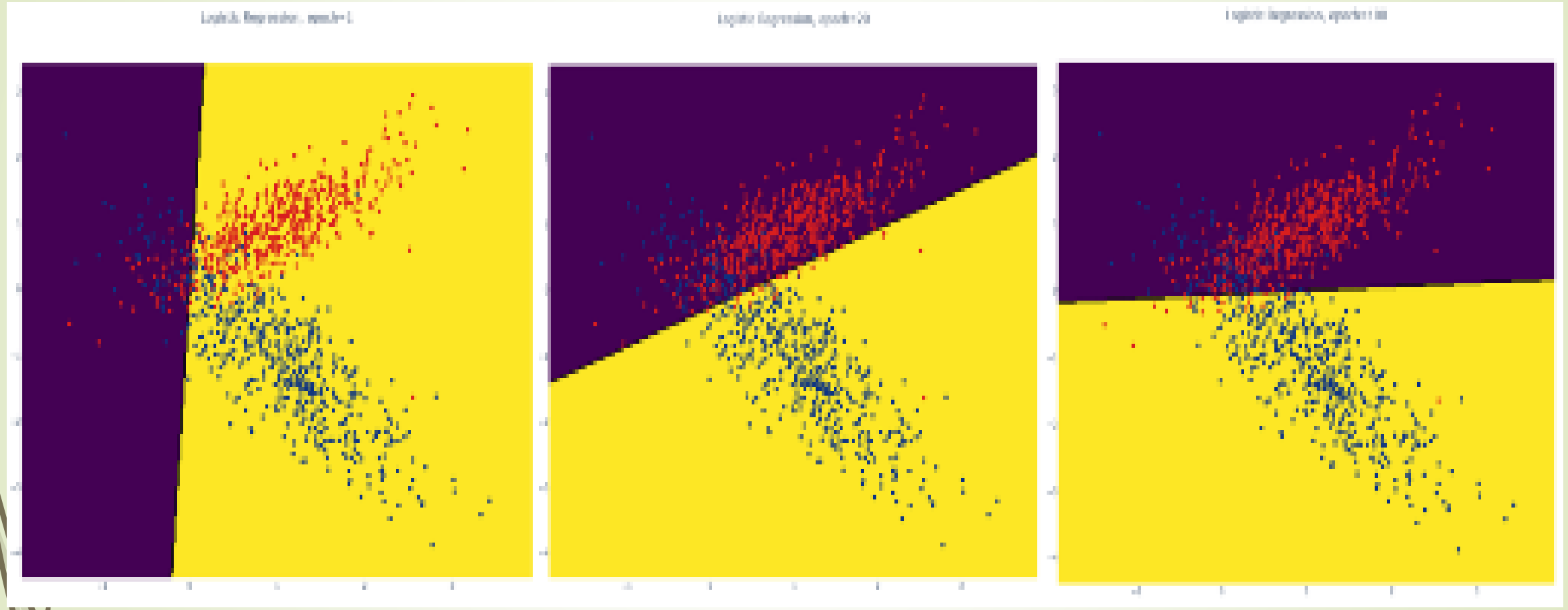


Рис.2. Візуалізація навчання градієнтним спуском.

Плюси логістичної регресії:

Це відносно простий алгоритм, який вимагає невеликої кількості обчислювальних ресурсів і може бути ефективно використаний для вирішення великої кількості класифікаційних завдань.

Інтерпретованість: логістична регресія дозволяє розуміти, які змінні впливають класифікацію і як.

Працює добре на невеликих наборах даних: логістична регресія показує добрі результати на невеликих наборах даних.

Невелика ймовірність перенавчання: логістична регресія схильна до менш перенавчання, оскільки вона не має безліч параметрів, які потрібно оптимізувати.

Мінуси логістичної регресії:

Потрібна нормалізація ознак: логістична регресія вимагає нормалізації ознак, щоб гарантувати, що ознаки роблять однаковий внесок у модель.

Працює погано на складних завданнях: може працювати погано на задачах із великою кількістю ознак чи складною структурою даних.

Лінійність: логістична регресія працює лише з лінійними межами рішень, що обмежує її здатність вирішувати складні завдання класифікації.

Низька точність: логістична регресія може показувати низьку точність, якщо класи є лінійно роздільними.

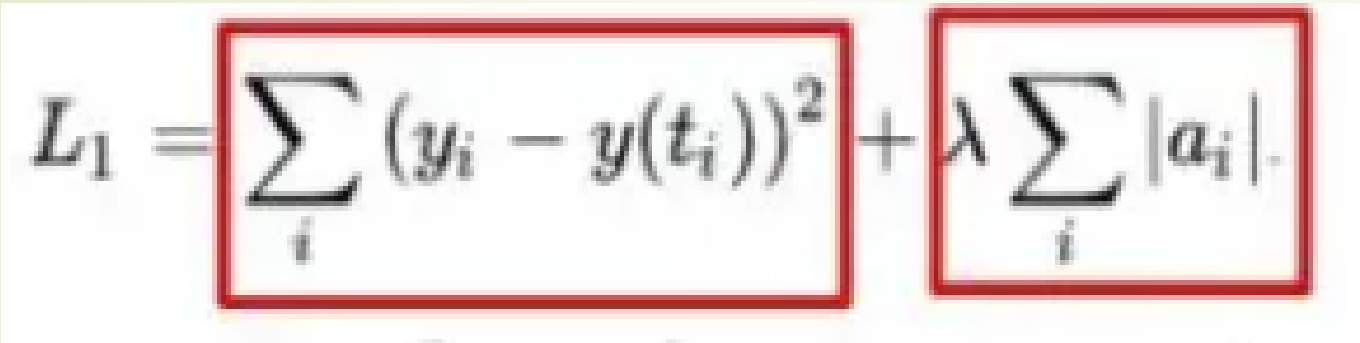
L1 ТА L2 РЕГУЛЯРИЗАЦІЯ.

У машинному навчанні та Data science, регуляризація є важливою технікою для управління перенавчанням моделі. Вона допомагає уникнути надто складної моделі, яка може добре підлаштуватися під навчальні дані, але погано працюватиме на нових даних.

Ми розглянемо два основні типи регуляризації: L1 та L2. Більш конкретно розглянемо як вони працюють, і як їх можна використовувати в Python для створення більш надійних моделей у data science.

L1 регуляризація також відома як Lasso (Least Absolute Shrinkage and Selection Operator) регуляризація Вона заснована на додаванні штрафу, що дорівнює абсолютному значенню коефіцієнтів моделі.

Формально, L1 регуляризація додає в функцію втрат додатковий компонент, що накладає штраф за складність моделі, тобто високі ваги:



L1 регуляризація схильна до відбору ознак, оскільки може зменшити ваги ознак до нуля. Це дозволяє прибрати неінформативні ознаки з моделі, що може зменшити складність моделі та покращити її узагальнюючу здатність.

У бібліотеці Python scikit-learn, можна використовувати регуляризацію L1 при навчанні лінійної регресії:

```
from sklearn import linear_model
reg = linear_model.Lasso(alpha=0.1)
```

Тут параметр **alpha** це гіперпараметр, який управляє «загальною силою» регуляризації. Великі значення alpha відповідають сильнішій регуляризації.

L1 регуляризація є ефективним методом боротьби з перенавчанням моделі у машинному навчанні. Однак, при використанні методу градієнтного спуску, який є одним із найпопулярніших алгоритмів оптимізації моделі, регуляризація L1 може призвести до деяких проблем.

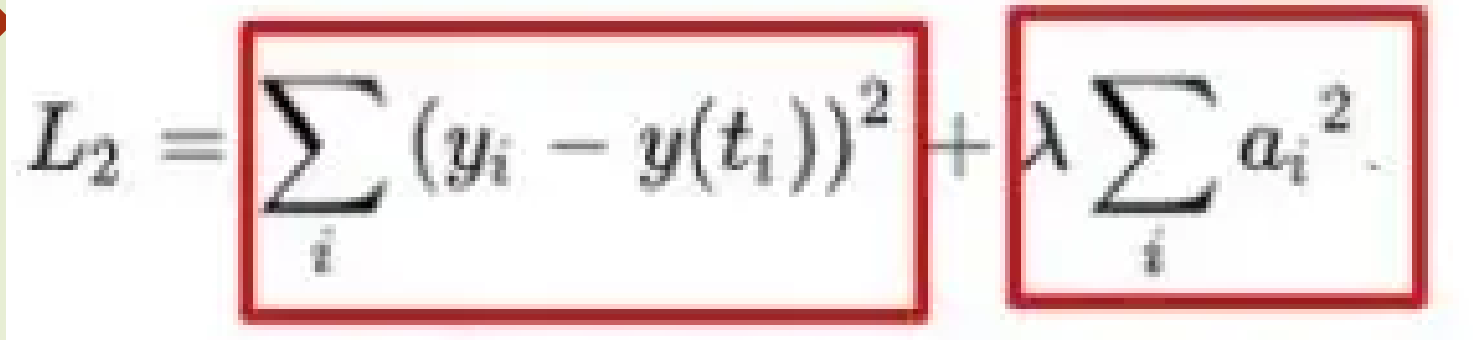
Зокрема, L1 регуляризація має кілька «гострих» кутів (розривів) на околиці нуля, де похідна не визначена. Це ускладнює обчислення градієнта функції втрат, коли використовується регуляризація L1. Метод градієнтного спуску вимагає, щоб градієнт був гладким і безперервним, щоб правильно працювати, і тому регуляризація L1 може бути менш ефективна при використанні градієнтного спуску.

Замість L1 регуляризації в методі градієнтного спуску часто використовується L2 регуляризація, так як вона має більш гладку похідну і краще працювати з градієнтним спуском. Однак, в деяких випадках L1 регуляризація може все ж таки використовуватися в методі градієнтного спуску з використанням різних технік оптимізації, таких як координатний спуск або L-BFGS, які можуть краще обробляти розриви функції втрат.

L2 РЕГУЛЯРИЗАЦІЯ.

Крім L1 регуляризації, існує також L2 регуляризація (іноді звана Ridge регуляризацією), яка також застосовується в лінійній регресії та багатьох інших моделях.

L2 регуляризація також додає до оптимізаційної функції моделі штрафну функцію:



Ця штрафна функція є сумою квадратів ваги моделі, помножених на гіперпараметр регуляризації. Це означає, що регуляризація L2 штрафує великі значення ваг, змушуючи їх наближатися до нуля, але на відміну від регуляризації L1 не зануляє їх повністю. Натомість L2 регуляризація штрафує великі значення ваг гладкіше і безперервно, що дозволяє більш впевнено керувати компромісом між точністю та складністю моделі.

Крім того, L2 регуляризація може допомогти у запобіганні перенавчання та поліпшенні узагальнюючої здатності моделі, а також у зменшенні впливу шуму даних на модель.

У бібліотеці Python scikit-learn, можна використовувати регуляризацію L2 при навчанні лінійної регресії:

```
from sklearn import linear_model
reg = linear_model.Ridge(alpha=0.1)
```

Тут параметр **alpha** це гіперпараметр, який управляє загальною силою регуляризації. Великі значення alpha відповідають сильнішій регуляризації.

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import load_diabetes
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

# отримаємо датасет з бібліотеки sklearn
diabetes = load_diabetes()
scaler = MinMaxScaler()
inputs = scaler.fit_transform(diabetes.data)
targets = diabetes.target

X_train, X_test, y_train, y_test = train_test_split(inputs,
targets, test_size=0.3, random_state=42)
X_train, X_test = torch.from_numpy(X_train).float(),
torch.from_numpy(X_test).float()
y_train, y_test = torch.from_numpy(y_train).float().view(-1, 1),
torch.from_numpy(y_test).float().view(-1, 1)
```

Наш клас, що реалізує лінійну регресію, доповниться двома новими методами, що відповідають за L1 і L2 відповідно:

```
class LinearRegression(nn.Module):
    def __init__(self, input_size, output_size, lambda_):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(input_size,
output_size))
        self.bias = nn.Parameter(torch.randn(output_size))
        self.lambda_ = lambda_

    def forward(self, x):
        return x @ self.weights + self.bias

    def l1_reg(self):
        return self.lambda_*torch.sum(torch.abs(self.weights))

    def l2_reg(self):
        return self.lambda_*torch.sum(torch.pow(self.weights, 2))
```

```
# Ініціалізація моделі
input_size = X_train.shape[1]
output_size = 1
lambda_ = 0.01
model = LinearRegression(input_size,
output_size, lambda_)
```

Далі визначимо функцію втрат та оптимізаційний алгоритм. На цей раз використовуємо алгоритм L-BFGS. L-BFGS є методом оптимізації, який використовує інформацію про градієнт функції втрат, але не є прямим градієнтним методом оптимізації. Одну епоху навчання визначимо функцією fitness_step().

Тут же і додаватимемо нашу регуляризацію у вигляді штрафу до функції втрат.

```
# Ініціалізуємо функцію втрат і оптимізатор
criterion = nn.MSELoss()
#optimizer = optim.SGD(model.parameters(), lr=0.1)
optimizer = optim.LBFGS(model.parameters(), lr=1.0)

# епоха навчання
def fitness_step():
    outputs = model(X_train) # Отримаємо прогноз
    loss = criterion(outputs, y_train) # Обчислюємо функцію втрат
    if reg == 'l1':
        loss += model.l1_reg() # Добавляемо L1 регуляризацію
    elif reg == 'l2':
        loss += model.l2_reg() # Добавляемо L2 регуляризацію
    # Виконуємо оптимізацію параметрів моделі
    optimizer.zero_grad()
    loss.backward()
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
    return loss
```

Далі будемо додавати штраф до функції втрат безпосередньо в циклі навчання (у коді застосовується L1, для використання L2 слід перевизначити змінну reg на 'l2'):

```
# Запускаємо навчання
reg = 'l1'
num_epochs = 500
for epoch in range(num_epochs):
    optimizer.step(fitness_step)

print(f'MSE модели на обучающей выборке
{criterion(model(X_train), y_train)}')
print(f'MSE модели на тестовой выборке
{criterion(model(X_test), y_test)}')
```

Як L1, так і L2 регуляризації застосовуються для обмеження ваги моделі з метою уникнути перенавчання і досягти найкращої узагальнюючої здатності моделі. Застосування L1-регуляризації іноді може давати корисний побічний ефект, що викликає прагнення одного або більше вагових значень 0, а це означає, що відповідний ознака не важливий. Це з форм того, що називають селекцією ознак (feature selection). Однак, використання L1-регуляризації не завжди можливе, тому що вона може не підходити для деяких алгоритмів машинного навчання, особливо для тих, які використовують чисельні методи обчислення градієнта. На відміну від L1, L2-регуляризація працює з усіма алгоритмами машинного навчання, але не видаляє важливі ознаки. Зрештою, вибір між L1 і L2 регуляризацією залежить від конкретної задачі та методів навчання, тому визначення найбільш підходящої регуляризації може вимагати тестування методами проб та помилок.