НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

СЕРІЯ «КІБЕРБЕЗПЕКА ТА ШТУЧНИЙ ІНТЕЛЕКТ»

С. П. Євсеєв, О. В. Шматко, О. Б. Ахієзер, В. Є. Сокол, Н. Л. Чернова

АТАКИ НА СИСТЕМИ ШТУЧНОГО ІНТЕЛЕКТУ

навчально-практичний посібник

За загальною редакцією доктора технічних наук, професор С. П. Євсеєва

Львів
«Новий Світ-2000»
2025

УДК 004.056(075.8)

І 76

Рекомендовано вченою радою
Національного технічного університету «Харківський політехнічний інститут»
Протокол № 2 від 21.02.2025 р.

Укладачі:

С. П. Євсеєв, доктор технічних наук, професор, завідувач кафедри кібербезпеки НТУ «ХПІ»;
О. В. Шматко, доцент кафедри програмної інженерії та інтелектуальних технологій управління НТУ «ХПІ»;
О. Б. Ахієзер, професорка, завідувачка кафедри комп'ютерної математики і аналізу даних НТУ «ХПІ»;
В. Є. Сокол, докторант кафедри кібербезпеки НТУ «ХПІ»;
Н. Л. Чернова, доцентка кафедри програмної інженерії та інтелектуальних технологій управління НТУ «ХПІ».

Рецензенти:

І. Р. Опірський, доктор технічних наук, професор, завідувач кафедри захисту інформації Національного університету «Львівська політехніка»;
С. Е. Остапов, доктор фізико-математичних наук, професор кафедри програмного забезпечення комп'ютерних систем Чернівецького національного університету імені Юрія Федьковича.

Євсеєв С. П., Шматко О. В., Ахієзер О. Б., Сокол В. Є., Чернова Н. Л.

І 76 **Атаки на системи штучного інтелекту** : навчально-практичний посібник / уклад. С. П. Євсеєв, О. В. Шматко, О. Б. Ахієзер, В. Є. Сокол, Н. Л. Чернова ; за заг. ред. С. П. Євсеєва. – Харків : НТУ «ХПІ», – Львів : «Новий Світ-2000», 2025. – 108 с. – (Серія «Кібербезпека та штучний інтелект»).

ISBN 978-966-418-515-5

Навчально-практичний посібник присвячений комплексному аналізу загроз, атак та захисних механізмів у системах штучного інтелекту (ШІ). У книзі висвітлено фундаментальні теоретичні аспекти уразливостей моделей машинного навчання, методів атак на штучний інтелект та стратегій їхнього виявлення і нейтралізації. Особлива увага приділена аналізу змагальних атак (adversarial attacks), маніпуляцій із навчальними даними (data poisoning).

Практичний аспект реалізовано через серію завдань, що включають моделювання атак на нейронні мережі, оцінку стійкості алгоритмів до змагальних прикладів та впровадження механізмів підвищення стійкості ШІ-систем. Окрім цього, розглядаються методи тестування програмного забезпечення, побудованого на базі нейромереж, на його стійкість до атак у різних сферах застосування ШІ.

Посібник поєднує глибокий теоретичний аналіз із практичними завданнями, що сприяє ефективному засвоєнню матеріалу та розвитку навичок оцінки загроз та впровадження заходів безпеки. Він стане корисним для студентів технічних спеціальностей, аспірантів, викладачів, фахівців у сфері кібербезпеки та всіх, хто цікавиться захистом систем штучного інтелекту від потенційних атак.

УДК 004.056(075.8)

© Євсеєв С.П., Шматко О.В., Ахієзер О. Б., Сокол В.Є., Чернова Н.Л., 2025
© За заг. ред. Євсеєва С. П., 2025
© НТУ «ХПІ», 2025
© Видавництво ПП «Новий Світ-2000», ФОП Піча С.В., 2025

ISBN 978-966-418-515-5

ЗМІСТ

ЛАБОРАТОРНА РОБОТА № 1. ДОСЛІДЖЕННЯ АТАК УХИЛЕННЯ ВІД ЗМАГАЛЬНОСТІ НА МОДЕЛІ МАШИННОГО НАВЧАННЯ НА ОСНОВІ SUPPORT VECTOR MACHINE (SVM) 7

ВКАЗІВКИ З ПІДГОТОВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ. 7

ТЕОРЕТИЧНІ ВІДОМОСТІ 8

ПРИКЛАДИ ПРАКТИЧНИХ ЗАВДАНЬ 13

ЗАГАЛЬНЕ ЗАВДАННЯ ДЛЯ ВИКОНАННЯ 23

КОНТРОЛЬНІ ПИТАННЯ. 23

ЛАБОРАТОРНА РОБОТА № 2. ДОСЛІДЖЕННЯ ПЕРЕНОСИМОСТІ АТАК УХИЛЕННЯ 25

ВКАЗІВКИ З ПІДГОТОВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ. 25

ТЕОРЕТИЧНІ ВІДОМОСТІ 26

ПРИКЛАДИ ПРАКТИЧНИХ ЗАВДАНЬ 33

ЗАГАЛЬНЕ ЗАВДАННЯ ДЛЯ ВИКОНАННЯ 49

КОНТРОЛЬНІ ПИТАННЯ 50

ЛАБОРАТОРНА РОБОТА № 3. ДОСЛІДЖЕННЯ АТАК ОТРУЄННЯ (POISONING ATTACKS) НА МОДЕЛІ МАШИННОГО НАВЧАННЯ . 52

ВКАЗІВКИ З ПІДГОТОВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ. 52

ТЕОРЕТИЧНІ ВІДОМОСТІ 53

ПРИКЛАДИ ПРАКТИЧНИХ ЗАВДАНЬ 56

ЗАГАЛЬНЕ ЗАВДАННЯ ДЛЯ ВИКОНАННЯ 71

КОНТРОЛЬНІ ПИТАННЯ 71

ЛАБОРАТОРНА РОБОТА № 4. ДОСЛІДЖЕННЯ ВИКОНАННЯ АТАК УХИЛЕННЯ ТА ОТРУЄННЯ НА НАБОРІ ДАНИХ MNIST 73

ВКАЗІВКИ З ПІДГОТОВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ. 73

ТЕОРЕТИЧНІ ВІДОМОСТІ 74

ПРИКЛАДИ ПРАКТИЧНИХ ЗАВДАНЬ 76
ЗАГАЛЬНЕ ЗАВДАННЯ ДЛЯ ВИКОНАННЯ 89
КОНТРОЛЬНІ ПИТАННЯ 90
ЛАБОРАТОРНА РОБОТА № 5. ДОСЛІДЖЕННЯ АТАКИ УХИЛЕННЯ (EVASION ATTACKS) НА НЕЙРОННІ МЕРЕЖІ З ВИКОРИСТАННЯМ НАБОРУ ДАНИХ MNIST 91
ВКАЗІВКИ З ПІДГОТОВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ. 91
ТЕОРЕТИЧНІ ВІДОМОСТІ 92
ПРИКЛАДИ ПРАКТИЧНИХ ЗАВДАНЬ 93
ЗАГАЛЬНЕ ЗАВДАННЯ ДЛЯ ВИКОНАННЯ 106
КОНТРОЛЬНІ ПИТАННЯ 107
СПИСОК ЛІТЕРАТУРИ 108

ВСТУП

Упродовж останніх років нейронні мережі досягли значних успіхів у задачах класифікації, особливо в галузі комп'ютерного зору. Різні архітектури демонструють різні рівні точності залежно від складності та специфіки завдання.

Серед згорткових нейронних мереж (CNN) AlexNet, представлена у 2012 році, стала проривом, досягнувши топ-5 помилки у 15,3% на змаганні ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). У 2014 році VGGNet з глибшою архітектурою та меншими фільтрами знизила цю помилку до 7,3%. ResNet, представлена у 2015 році, впровадила архітектуру з пропускними з'єднаннями, досягнувши топ-5 помилки у 3,6% на тому ж наборі даних.

Рекурентні нейронні мережі (RNN), зокрема LSTM та GRU, широко застосовуються в обробці природної мови та часових рядів. Наприклад, у задачах передбачення наступного слова в реченні вони демонструють високу точність, значно перевершуючи традиційні моделі.

Генеративно-змагальні мережі (GAN) показали здатність створювати фотореалістичні зображення. Зокрема, модель StyleGAN генерує зображення облич, які важко відрізнити від реальних фотографій.

Трансформери, такі як BERT та GPT, досягли передових результатів у задачах розуміння та генерації тексту, включаючи системи запитань-відповідей та машинний переклад.

Порівняльний аналіз понад 400 нейронних мереж для задачі класифікації зображень показав, що сучасні архітектури, такі як EfficientNet та Vision Transformers (ViT), досягають точності понад 90% на наборі даних ImageNet.

З моменту виявлення у 2013 році феномену змагальних прикладів (adversarial examples), коли незначні, практично непомітні для людини зміни вхідних даних можуть призвести до хибних висновків моделі, дослідження в галузі атак на штучний інтелект значно просунулися. У 2023 року були розроблені більш витончені методи атак, такі як генеративно-змагальні мережі (GAN), здатні створювати високореалістичні підроблені дані, що обходять системи виявлення.

Окрім того, зловмисники почали активно використовувати штучний інтелект для проведення кібератак. 2023 році щонайменше п'ять кібервугруповань застосовували продукти OpenAI для збору інформації з

відкритих джерел (OSINT), що дозволило їм ефективніше планувати та здійснювати атаки.

У відповідь на ці загрози дослідники розробляють нові методи захисту нейронних мереж. Наприклад, запропоновані підходи, засновані на ідентифікації та нейтралізації тригерів закладок (бекдорів), які дозволяють виявляти та усувати приховані вразливості в моделях. Однак, незважаючи на прогрес у цій галузі, універсального рішення, що забезпечує повний захист систем штучного інтелекту від усіх типів атак, поки не знайдено.

Таким чином, забезпечення безпеки та стійкості систем штучного інтелекту залишається актуальним і складним завданням. розвитком технологій штучного інтелекту з'являються нові загрози, які потребують постійної уваги та розробки ефективних методів захисту.

Цей навчальний посібник присвячений аналізу загроз і атак на системи ШІ, зокрема дослідженню змагальних прикладів (adversarial examples), атак на конфіденційність та підробку даних. Детально розглядаються методи впливу на моделі машинного навчання, способи їхнього обходу та експлуатації вразливостей. Окремо висвітлено актуальні напрямки захисту від атак, включаючи алгоритмічні методи підвищення стійкості моделей, а також правові та етичні аспекти безпеки ШІ.

Навчальний матеріал містить як теоретичні аспекти, так і практичні кейси, що дозволяють глибше зрозуміти механізми атак та засоби їхньої нейтралізації. Посібник буде корисним студентам, дослідникам, розробникам та спеціалістам у сфері безпеки, які прагнуть опанувати фундаментальні принципи атак і захисту систем штучного інтелекту.

ЛАБОРАТОРНА РОБОТА № 1.
ДОСЛІДЖЕННЯ АТАК УХИЛЕННЯ ВІД ЗМАГАЛЬНОСТІ НА МОДЕЛІ МАШИННОГО НАВЧАННЯ НА ОСНОВІ SUPPORT VECTOR MACHINE (SVM)

Мета роботи: ознайомитись з атаками ухилення від змагальності на моделі машинного навчання на основі Support Vector Machine (SVM) з ядром Radial Basis Function (RBF).

ВКАЗІВКИ З ПІДГОТОВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ.

Атаки ухилення (також відомі як змагальні приклади) полягають у ретельній зміні вхідних зразків під час тестування, щоб вони були неправильно класифіковані.

Спочатку ми створимо та навчимо класифікатор, оцінюючи його ефективність у стандартному сценарії, тобто не піддаючи атаці. А потім створимо змагальний приклад із класифікатором SVM, використовуючи алгоритм максимальної довіри на основі градієнта для генерації атак ухилення, який реалізований у SecML за допомогою класу CAttackEvasionPGDLS.

Додаткову інформацію при підготовці до роботи можна отримати в:

1. Biggio, B., Corona, I., Maiorca, D., Nelson, B., Srndić, N., Leskov, P., Giacinto, G., Roli, F., 2013. Evasion Attacks against Machine Learning at Test Time. In ECML-PKDD 2013. URL: https://arxiv.org/abs/1708.06131

2. Melis, M., Demontis, A., Biggio, B., Brown, G., Fumera, G. and Roli, F., 2017. Is deep learning safe for robot vision? adversarial examples against the icub humanoid. In Proceedings of IEEE ICCV 2017. URL: https://arxiv.org/abs/1708.06939

3. Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C. and Roli, F., 2019. Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks. In 28th Usenix Security Symposium, Santa Clara, California, USA. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/demontis

ТЕОРЕТИЧНІ ВІДОМОСТІ

Види змагальних прикладів

Змагальні атаки є одним із найбільш досліджуваних і практично значущих типів атак на системи машинного навчання. Загалом, під атакою на модель розуміється навмисне втручання в процес її роботи, яке може відбуватися як на етапі навчання, так і під час інференсу. Мета зловмисника може включати порушення коректного функціонування моделі, примушення її до видачі бажаного результату, крадіжку інформації про параметри моделі або вилучення приватних даних, використаних у навчанні. У зв'язку з цим атаки на системи машинного навчання можна класифікувати на чотири основні типи:

1. **Змагальні атаки (атаки ухилення)** передбачають втручання, за якого зловмисник може змінювати лише вхідні дані вже навченої моделі, не впливаючи на її внутрішні параметри. Такі атаки можуть спричинити некоректну роботу моделі, внаслідок чого вона видає неправильний результат. Саме цей вид атак є основним предметом дослідження у цьому посібнику роботі.

2. **Атаки отруєння** передбачають внесення змін у навчальні дані перед процесом тренування моделі. Це може призвести до того, що система буде навмисно навчена видавати неправильні результати, що робить її вразливою на етапі експлуатації.

3. **Атаки вилучення** спрямовані на відновлення параметрів моделі шляхом аналізу її роботи. Зловмисник може створити локальну копію моделі, використовуючи доступ до її вхідних даних і відповідних вихідних передбачень.

4. **Атаки інверсії** дозволяють зловмиснику відновити дані, на яких була навчена модель, аналізуючи її відповіді на різних входах. Це становить особливу загрозу для конфіденційності даних, оскільки зловмисник може отримати доступ до особистої або комерційно цінної інформації.

У межах цього посібника розглядаються виключно змагальні атаки, за яких атакувальник не може змінювати параметри моделі або впливати на процес її навчання, але здатний маніпулювати вхідними даними. Залежно від рівня доступу до інформації про модель зловмисник може використовувати різні стратегії атак, наприклад, знати її архітектуру та параметри або мати можливість аналізувати лише вихідні передбачення моделі для різних вхідних даних.

Некоректність класифікації як мета змагальної атаки

Однією з ключових цілей змагальних атак є примушення моделі до хибної класифікації вхідних даних. Це явище можна проілюструвати на прикладі задачі класифікації зображень, оскільки саме у цій сфері проводиться більшість досліджень атак і механізмів захисту.

Модель класифікації зображень можна подати як функцію:

$$f(x) : \mathbb{R}^{m \times n \times c} \to \mathbb{R}^K \tag{1.1}$$

де x — вхідне зображення розміром $m \times n$ пікселів із c колірними каналами.

У випадку класифікації на K класів функція $f(x)$ повертає *softmax*-розподіл ймовірностей належності зображення кожному з K класів:

$$f(x) = (p_1, p_2, \dots, p_K), \sum_{i=1}^{K} p_i = 1 \tag{1.2}$$

Результатом класифікації є передбачений моделлю клас c, якому відповідає максимальна ймовірність:

$$c = \arg max_i f_i(x) \tag{1.3}$$

Нехай є зображення x, про яке відомо, що воно належить класу y. Це може бути підтверджено експертною розміткою або суб'єктивною оцінкою групи аналітиків. Некоректною роботою моделі в цьому контексті вважається ситуація, коли її передбачення не збігається з істинним класом, тобто:

$$f(x) \neq y. \tag{1.4}$$

Слід зазначити, що помилки класифікації можуть виникати і без зовнішнього втручання. Навіть на відносно простих наборах даних, таких як MNIST, не існує моделі, яка б забезпечувала 100% точність класифікації. Станом на 2022 рік найкращий результат на цьому датасеті становить 99.91%. У складніших задачах комп’ютерного бачення частка помилок зростає. Однак принципова відмінність змагальних атак полягає в тому, що вони навмисно створюють умови, за яких модель робить грубі та нелогічні помилки. На відміну від природних помилок, що виникають на складних або неоднозначних зображеннях, помилки, спричинені змагальними атаками,

виглядають неприродно, оскільки атаковані зображення залишаються візуально зрозумілими та легко класифікованими.

Основні типи змагальних атак

Змагальні атаки можуть реалізовуватися різними способами. Одним із найпоширеніших методів є **нанесення змагального збурення**, за якого до вихідного зображення x додається незначне за нормою ϵ збурення δ, що призводить до хибної класифікації:

$$x = x + \delta, ||\delta|| < \epsilon, f(x) \neq y \quad (1.5)$$

Тут $||\delta||$ визначає міру величини внесеної зміни, яка обмежується таким чином, щоб доданий шум залишався непомітним для людського сприйняття. Головна мета такого втручання — зробити змінене зображення візуально не відмінним від оригіналу, але таким, що змушує модель змінити передбачений клас (рис. 1.1).

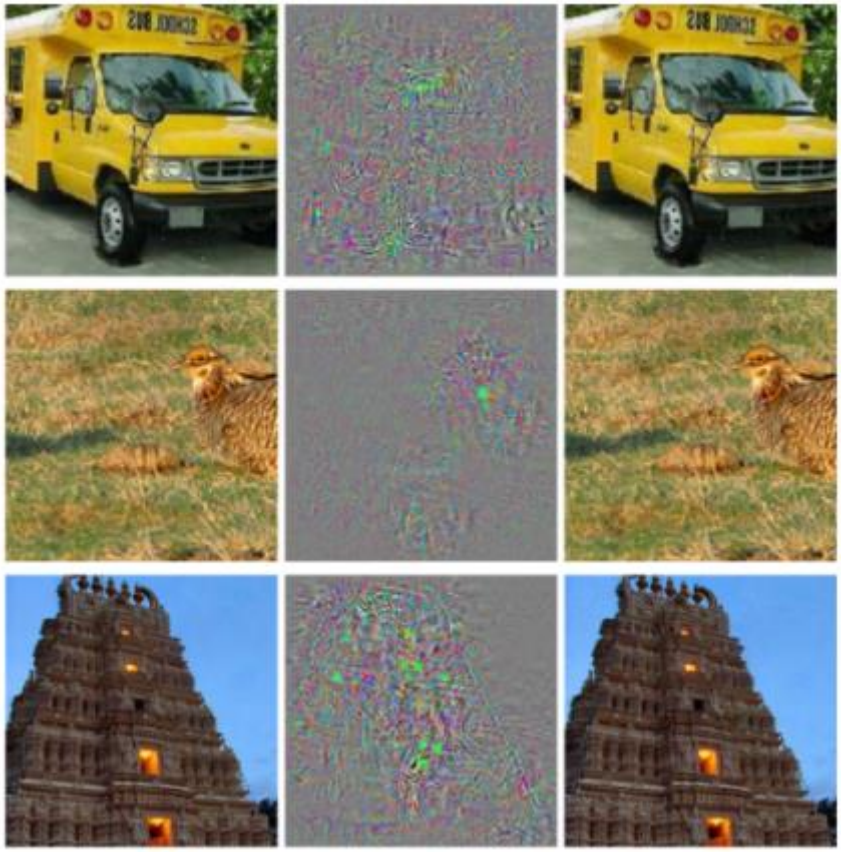


Рис. 1.1. Змагальні приклади для моделі AlexNet, навченої на ImageNet

Ліворуч наведені звичайні зображення з ImageNet зображення, класифіковані вірно, в центрі – вноситься шум, праворуч змагальні приклади, класифіковані як страус.

Іншим поширеним типом атак є **накладання змагальних патчів** (рис. 1.2).

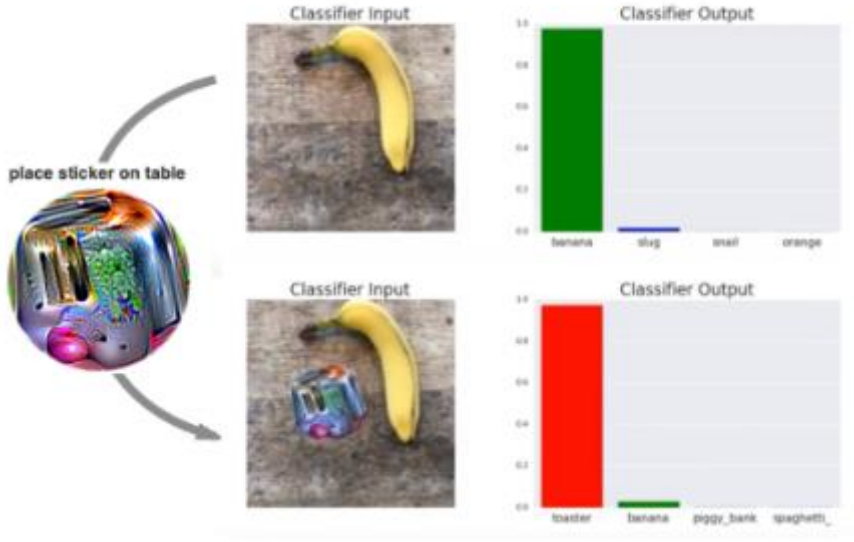


Рис. 1.2. Накладення патча повністю змінює прогноз класу моделі VGG-16

У цьому випадку атакувальник замінює певну ділянку зображення (наприклад, квадратну або круглу область) заздалегідь підготовленим шаблоном, що змінює результат класифікації.

Окрему групу складають **абстрактні змагальні приклади**, які не містять осмислених об'єктів, але модель впевнено класифікує їх у певний клас. Прикладом можуть слугувати зображення, що нагадують білий шум або випадкові текстури, які, на думку нейромережі, належать до конкретного класу з ймовірністю, близькою до одиниці (рис. 1.3).

Виділяється також група **фізичних змагальних прикладів**, що створюються для використання в реальному світі (рис. 1.4). Такі атаки можуть включати модифікацію дорожніх знаків (наприклад, нанесення наклейок, що спричиняють хибну класифікацію), використання спеціальних аксесуарів (наприклад, окулярів, що унеможливлюють розпізнавання обличчя системою відеоспостереження) або створення тривимірних об'єктів, які модель неправильно ідентифікує.

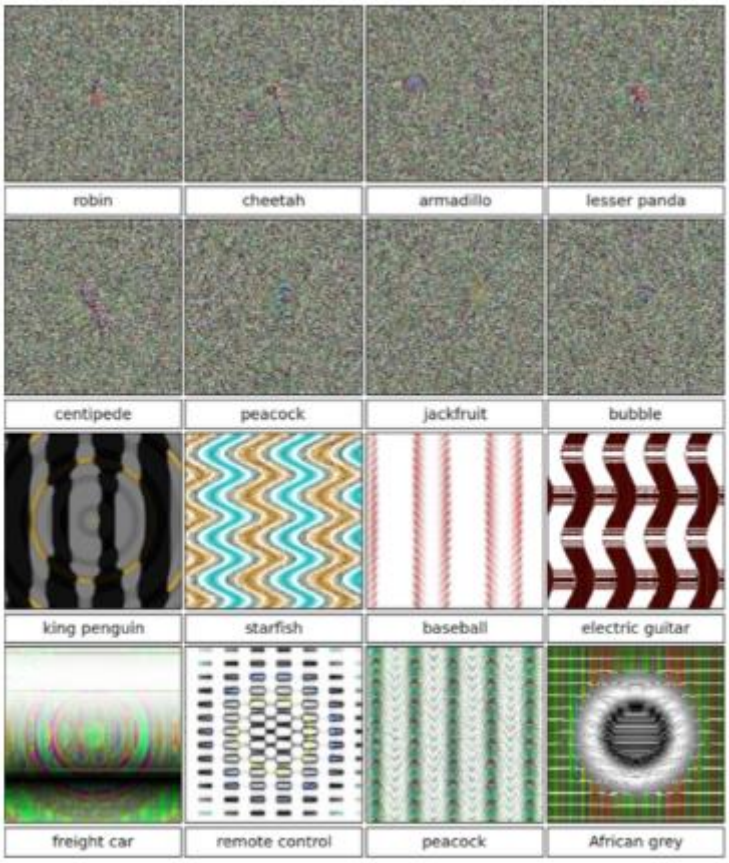


Рис. 1.3. Не інтерпретовані зображення, які нейромережа відносить до зазначеного класу з упевненістю більш ніж 99.6%

Рис. 1.4. Надягання спеціальних окулярів призводить до того, що потрапив в камеру людина (зверху), сприймається нейронною мережею як схожий на зовсім іншу людину (знизу)

Таким чином, змагальні атаки є серйозною загрозою для нейромережевих моделей, що вимагає розробки ефективних стратегій захисту та підвищення їхньої стійкості до подібних впливів.

ПРИКЛАДИ ПРАКТИЧНИХ ЗАВДАНЬ

В даній роботі ми потренеруємся у використанні SecML (https://secml.readthedocs.io/en/v0.15/), бібліотеки Python з відкритим кодом для оцінки безпеки алгоритмів машинного навчання. За допомогою платформи https://www.kaggle.com/.

Крок 1. У браузері перейдіть по https://www.kaggle.com/

Якщо ви бачите синю кнопку "Увійти" у верхньому правому куті, натисніть на неї та увійдіть в обліковий запис Kaggle. У меню виберіть "Файл", "Нова записна книжка".

Крок 2. Встановлення SecML

Виконайте наступні команди:

```
!pip install secml
import secml
```

Бібліотека встановиться, як показано нижче (рис. 1.5).

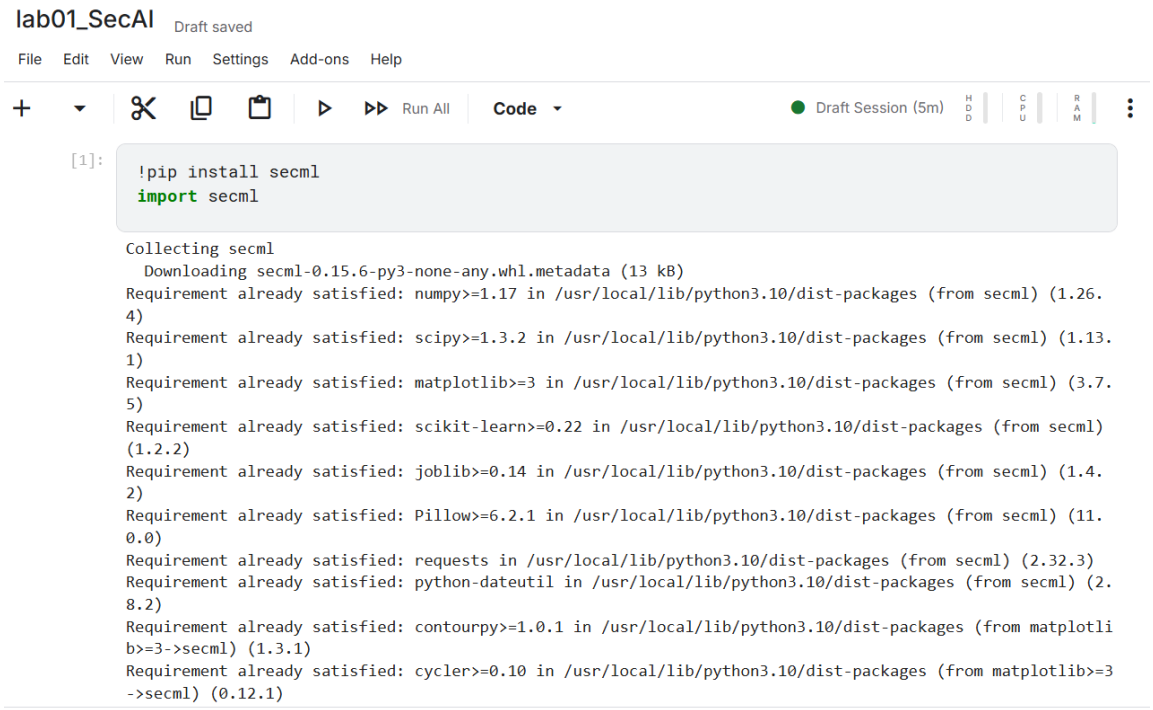


Рис. 1.5. Встановлення SecML

Крок 3. Підготовка набору даних

Виконайте команди, щоб створити простий штучний набір даних, що складається з двох груп точок на площині. Нам, як завжди, потрібні навчальні та тестові набори даних.

```
random_state = 999

n_features = 2  # Number of features
n_samples = 300  # Number of samples
centers = [[-1, -1], [1, 1]]  # Centers of the clusters
cluster_std = 0.9  # Standard deviation of the clusters

from secml.data.loader import CDLRandomBlobs
dataset = CDLRandomBlobs(n_features=n_features,
                         centers=centers,
                         cluster_std=cluster_std,
                         n_samples=n_samples,
                         random_state=random_state).load()

n_tr = 100  # Number of training set samples
n_val = 100  # Number of validation set samples
n_ts = 100  # Number of test set samples

# Split in training, validation and test
from secml.data.splitter import CTrainTestSplit
splitter = CTrainTestSplit(
    train_size=n_tr + n_val, test_size=n_ts, random_state=random_state)
tr_val, ts = splitter.split(dataset)
splitter = CTrainTestSplit(
    train_size=n_tr, test_size=n_val, random_state=random_state)
tr, val = splitter.split(dataset)

# Normalize the data
from secml.ml.features import CNormalizerMinMax
nmz = CNormalizerMinMax()
tr.X = nmz.fit_transform(tr.X)
val.X = nmz.transform(val.X)
```

```
ts.X = nmz.transform(ts.X)

# Display the training set
from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline

fig = CFigure(width=5, height=5)

# Convenience function for plotting a dataset
fig.sp.plot_ds(tr)

fig.show()
```

На рис. 1.6 представлено двокласовий набір даних, де два класи (сині та червоні точки) розділені, але з невеликою зоною перекриття.

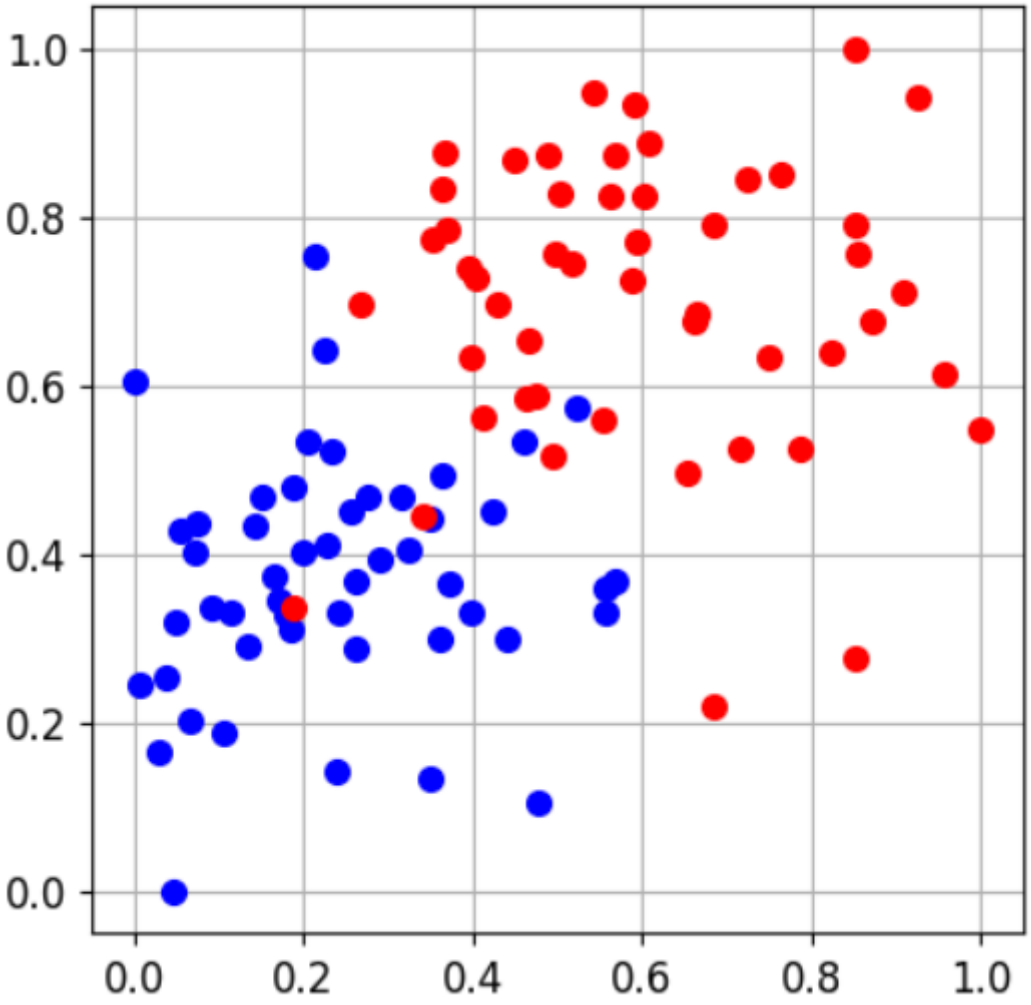


Рис.1.6. Набір даних

Цей набір даних ми будемо використовувати для навчання моделі класифікації та оцінки її здатності правильно розділяти класи.

Крок 4. Створення та навчання моделі

Наведений нижче код реалізує навчання та тестування класифікатора SVM з ядром RBF (ядерний метод опорних векторів) за допомогою бібліотеки SecML.

1. Завдання метрики точності

```
from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()
```

Імпорт метрики точності – (Accuracy) – CMetricAccuracy використовується для оцінки продуктивності класифікатора.

metric = CMetricAccuracy () – створюється екземпляр об'єкта, який потім буде використовуватися для розрахунку точності.

2. Створення SVM-класифікатора

```
from secml.ml.classifiers import CClassifierSVM
from secml.ml.kernels import CKernelRBF
clf = CClassifierSVM(kernel=CKernelRBF(gamma=10), C=1)
```

Імпортуються класи:

CClassifierSVM - реалізація SVM (Support vector Machine) В SecML.

CKernelRBF-радіально-базисне ядро (RBF).

Створюється SVM-Класифікатор з параметрами:

kernel=CKernelRBF (gamma=10) – використовується RBF-ядро з гіперпараметром gamma=10.

C = 1 – параметр регуляризації C (чим вище C, тим модель жорсткіше розділяє класи, але може перенавчатися).

3. Навчання моделі

```
clf.fit(tr.X, tr.Y)
print("Training of classifier complete!")
```

tr.X, tr.Y-Навчальний набір (tr.X-ознаки, tr.Y-мітки КЛАСІВ).

clf.fit(tr.X, tr.Y) - навчання SVM на тренувальних даних.

Виводиться повідомлення " Training of classifier complete!", що сигналізує про завершення процесу навчання.

4. Передбачення на тестовому наборі

```
y_pred = clf.predict(ts.X)
```

ts.X-тестовий набір даних (ознаки).

clf.predict(ts.X) - передбачення класів для тестових даних.

y_pred-масив передбачених міток КЛАСІВ.

5. Оцінка точності моделі

```
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)
```

y_true=ts.Y-справжні мітки класів з тестового набору.

y_pred=y_pred-передбачені моделлю класи.

metric.performance_score(...)- обчислює точність (Accuracy), порівнюючи y_true і y_pred.

6. Висновок точності класифікації

```
print("Accuracy on test set: {:.2%}".format(acc))
```

Виводиться точність моделі на тестовому наборі в процентному форматі.

{:.2%} - форматування числа acc в процентне значення з двома знаками після коми.

На рис.1.7 представлено результат виконання команд

[3]:

```
# Metric to use for training and performance evaluation
from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()

# Creation of the multiclass classifier
from secml.ml.classifiers import CClassifierSVM
from secml.ml.kernels import CKernelRBF
clf = CClassifierSVM(kernel=CKernelRBF(gamma=10), C=1)

# We can now fit the classifier
clf.fit(tr.X, tr.Y)
print("Training of classifier complete!")

# Compute predictions on a test set
y_pred = clf.predict(ts.X)

# Evaluate the accuracy of the classifier
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

print("Accuracy on test set: {:.2%}".format(acc))
```

```
Training of classifier complete!
Accuracy on test set: 94.00%
```

Рис. 1.7. Побудова та навчання моделі

Як видно з рис. 1.7 точності побудованої моделі класифікатора складає 94%.

Крок 5. Візуалізація результатів

Наступні команди використовуються для побудови діаграми, що показує області, на які модель класифікує дані (рис. 1.8).

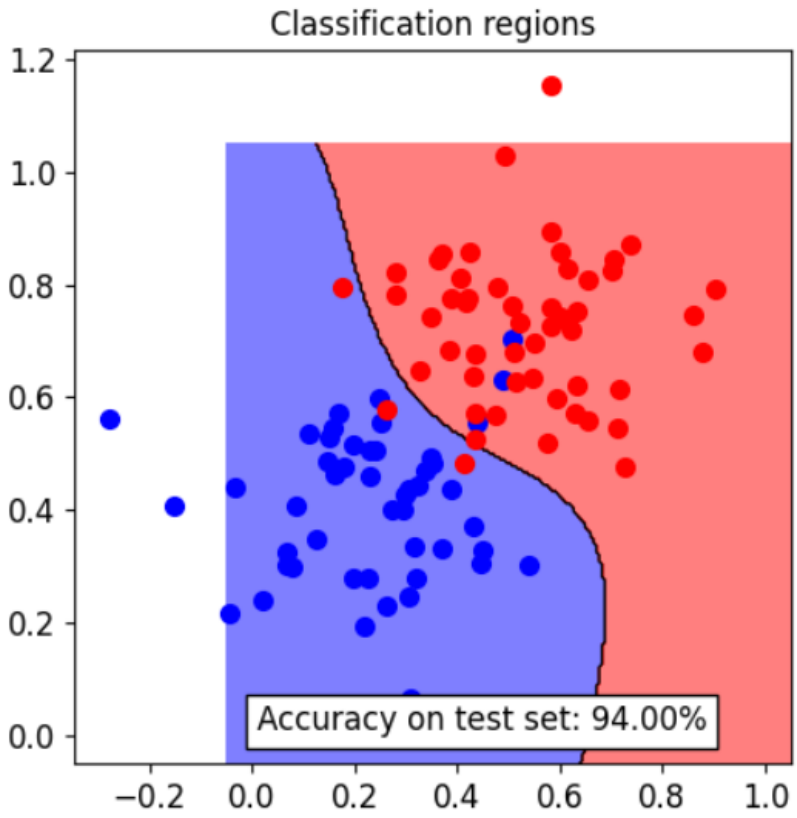


Рис.1.8. Діаграма областей класифікації

```
fig = CFigure(width=5, height=5)

# Convenience function for plotting the decision function of a classifier
fig.sp.plot_decision_regions(clf, n_grid_points=200)

fig.sp.plot_ds(ts)
fig.sp.grid(grid_on=False)

fig.sp.title("Classification regions")
fig.sp.text(0.01, 0.01, "Accuracy on test set: {:.2%}".format(acc),
        bbox=dict(facecolor='white'))
fig.show()
```

Крок 6. Маніпулювання однією міткою

Це найпростіша атака отруєння: зміна міток даних, щоб заплутати модель.

Виконаємо команди, щоб змінити першу мітку:

```
print("Training set:", tr.Y)
tr_poisoned = tr.deepcopy()
tr_poisoned.Y[0] = 0
print("Poisoned:    ", tr_poisoned.Y)
print('X[0]: ({:.2f}, {:.2f})'.format(tr_poisoned.X.tolist()[0][0],
                    tr_poisoned.X.tolist()[0][1]))
print()
fig = CFigure(width=9, height=4)
fig.subplot(1, 2, 1)
fig.sp.plot_ds(tr)

fig.subplot(1, 2, 2)
fig.sp.plot_ds(tr_poisoned)
fig.show()
```

На діаграмі зліва показані вихідні дані, а на діаграмі праворуч - спотворені дані. Як видно на рис. 1.9, одна червона крапка в центрі вгорі тепер синя.

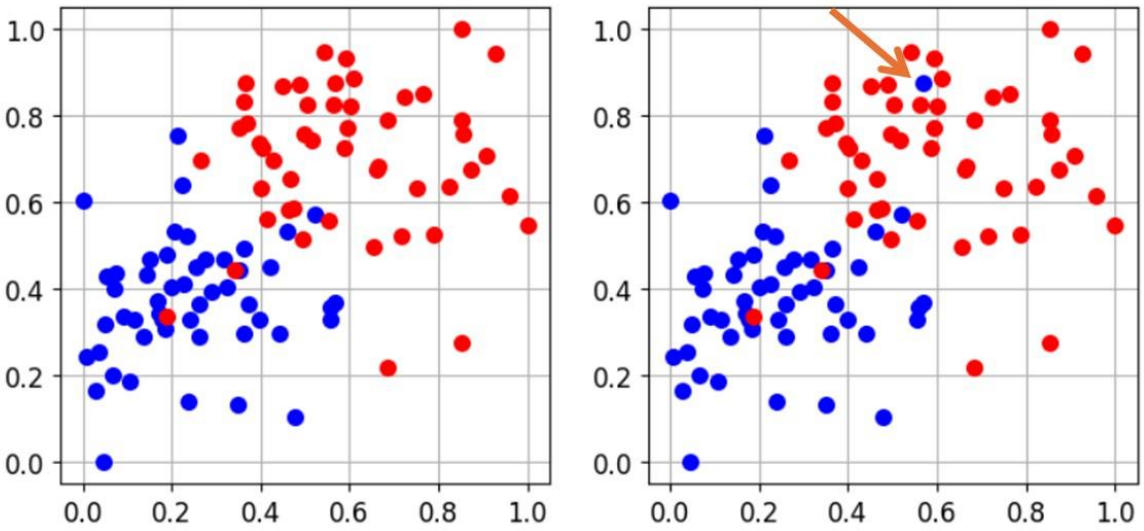


Рис. 1.9. Атака отруєння на 1 точку даних

Крок 7. Оцінка продуктивність зараженої моделі

Виконаємо оцінку продуктивності моделі. Введіть команди, щоб створити та навчити заражену модель:

```
# Create the poisoned multiclass classifier
clf_poisoned = CClassifierSVM(kernel=CKernelRBF(gamma=10), C=1)

# Fit the poisoned classifier
clf_poisoned.fit(tr_poisoned.X, tr_poisoned.Y)
print("Training of poisoned classifier complete!")

# Compute predictions on a test set
y_pred_poisoned = clf_poisoned.predict(ts.X)

# Evaluate the accuracy of the classifier
acc_poisoned = metric.performance_score(y_true=ts.Y, y_pred=y_pred_poisoned)

print("Accuracy on test set before poisoning: {:.2%}".format(acc))
print("Accuracy on test set after poisoning:  {:.2%}".format(acc_poisoned))
```

Як показано на рис. 1.10, точність моделі становить 94%.

[6]:
```
# Create the poisoned multiclass classifier
clf_poisoned = CClassifierSVM(kernel=CKernelRBF(gamma=10), C=1)

# Fit the poisoned classifier
clf_poisoned.fit(tr_poisoned.X, tr_poisoned.Y)
print("Training of poisoned classifier complete!")

# Compute predictions on a test set
y_pred_poisoned = clf_poisoned.predict(ts.X)

# Evaluate the accuracy of the classifier
acc_poisoned = metric.performance_score(y_true=ts.Y, y_pred=y_pred_poisoned)

print("Accuracy on test set before poisoning: {:.2%}".format(acc))
print("Accuracy on test set after poisoning:  {:.2%}".format(acc_poisoned))
```

```
Training of poisoned classifier complete!
Accuracy on test set before poisoning: 94.00%
Accuracy on test set after poisoning:  94.00%
```

Рис. 1.10 Точність зараженої моделі

Крок 8. Маніпулювання великою кількістю міток

Далі переглянемо перші 40 точок і заміним всі червоні крапки на сині:

```
print("Training set:", tr.Y)
tr_poisoned = tr.deepcopy()

for i in range(40):
  if tr_poisoned.Y[i] == 1:
    tr_poisoned.Y[i] = 0

print("Poisoned:    ", tr_poisoned.Y)
print('X[0]: ({:.2f}, {:.2f})'.format(tr_poisoned.X.tolist()[0][0],
                                      tr_poisoned.X.tolist()[0][1]))
print()
fig = CFigure(width=9, height=4)
fig.subplot(1, 2, 1)
fig.sp.plot_ds(tr)

fig.subplot(1, 2, 2)
fig.sp.plot_ds(tr_poisoned)
fig.show()
```

На діаграмі зліва показані вихідні дані, а на діаграмі праворуч – спотворені дані. Як показано на рис. 1.11, багато червоних крапок стали синіми.

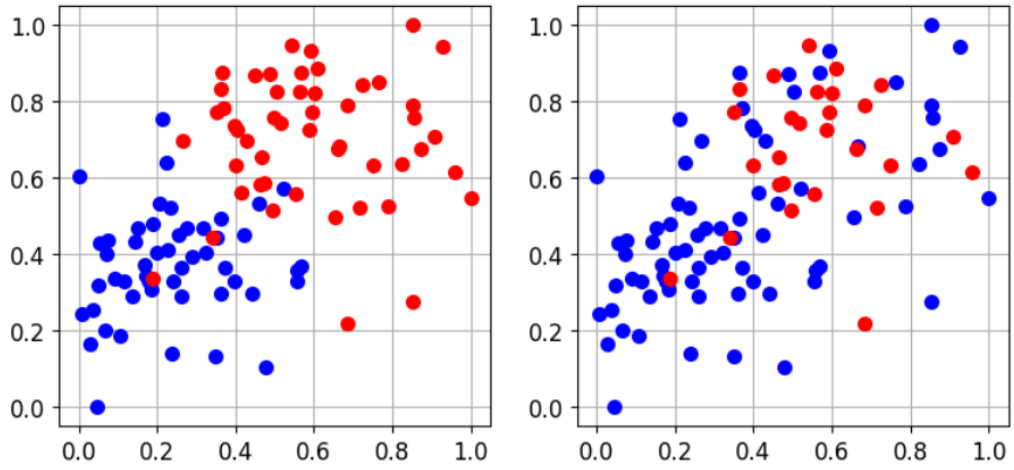


Рис. 1.11. «Отруєні» та «чисті» данні

Крок 9. Оцінка продуктивності зараженої моделі

Далі створимо та навчимо заражену модель:

```
# Create the poisoned multiclass classifier
clf_poisoned = CClassifierSVM(kernel=CKernelRBF(gamma=10), C=1)

# Fit the poisoned classifier
clf_poisoned.fit(tr_poisoned.X, tr_poisoned.Y)
print("Training of poisoned classifier complete!")

# Compute predictions on a test set
y_pred_poisoned = clf_poisoned.predict(ts.X)

# Evaluate the accuracy of the classifier
acc_poisoned = metric.performance_score(y_true=ts.Y,
y_pred=y_pred_poisoned)

print("Accuracy on test set before poisoning: {:.2%}".format(acc))
print("Accuracy on test set after poisoning:  {:.2%}".format(acc_poisoned))
```

Як видно на рис. 1.12, отруєння знизило точність до 74%.

```python
# Create the poisoned multiclass classifier
clf_poisoned = CClassifierSVM(kernel=CKernelRBF(gamma=10), C=1)

# Fit the poisoned classifier
clf_poisoned.fit(tr_poisoned.X, tr_poisoned.Y)
print("Training of poisoned classifier complete!")

# Compute predictions on a test set
y_pred_poisoned = clf_poisoned.predict(ts.X)

# Evaluate the accuracy of the classifier
acc_poisoned = metric.performance_score(y_true=ts.Y, y_pred=y_pred_poisoned)

print("Accuracy on test set before poisoning: {:.2%}".format(acc))
print("Accuracy on test set after poisoning:  {:.2%}".format(acc_poisoned))
```

```
Training of poisoned classifier complete!
Accuracy on test set before poisoning: 94.00%
Accuracy on test set after poisoning:  74.00%
```

Рис. 1.12. Точність отруєної моделі

ЗАГАЛЬНЕ ЗАВДАННЯ ДЛЯ ВИКОНАННЯ

1. Побудувати бінарний класифікатор.
2. Виконати оцінку продуктивності моделі.
3. Виконайте атаку, описану вище, щоб перевірити перші 60 точок у тренувальному наборі, і зробіть червоні точки синіми.
4. Побудуйте графіки, які відображають «чистий» та «отруєний» набори даних.
5. Оцініть продуктивності «чистої» та «отруєної» моделей.
6. Відповісти на контрольні запитання.
7. Підготувати звіт, який містить опис основних дій (зі скріншотами).

КОНТРОЛЬНІ ПИТАННЯ.

1. Що таке змагальні атаки на нейронні мережі, і в чому їхня основна загроза?
2. Які основні типи атак на системи машинного навчання існують, і чим вони відрізняються?
3. У чому полягає принцип роботи змагального збурення (adversarial perturbation), і як воно впливає на результат класифікації?
4. Що таке атака в режимі "білого ящика" (white-box attack), і як вона відрізняється від атаки "чорного ящика" (black-box attack)?

5. Які методи використовуються для створення змагальних прикладів, і які метрики застосовуються для їхньої оцінки?

6. Що таке фізичні змагальні приклади, і як вони можуть бути використані для обману реальних систем розпізнавання?

7. Які існують методи захисту нейронних мереж від змагальних атак, і наскільки вони ефективні?

8. Що таке універсальні змагальні атаки, і чим вони відрізняються від індивідуальних атак на окремі зображення?

9. Яким чином атаки на основі перенесення (transferability attacks) можуть загрожувати системам, навіть якщо атакувальник не має доступу до їхньої архітектури?

10. Чому змагальні атаки становлять загрозу для критично важливих систем, таких як автономне водіння або біометрична ідентифікація?

ЛАБОРАТОРНА РОБОТА № 2.
ДОСЛІДЖЕННЯ ПЕРЕНОСИМОСТІ АТАК УХИЛЕННЯ

Мета роботи: ознайомитися з можливостями переносимості атаки ухилення на різні моделі машинного навчання та дослідження її ефективності щодо інших, потенційно невідомих моделей.

ВКАЗІВКИ З ПІДГОТОВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

У цій роботі буде виконана перевірка, чи буде атака ухилення, сформована за допомогою методу опорних векторів (SVM), передаватися на інші моделі класифікаторів.

Спочатку ми створимо та навчимо сурогатний класифікатор та інші цільові класифікатори, оцінивши їх ефективність у стандартному сценарії.

Сурогатний класифікатор (surrogate classifier) являє собою модель, використовувану замість або на додаток до цільового класифікатора з метою аналізу його поведінки, здійснення атак або підвищення ефективності роботи системи. Ця концепція широко застосовується в різних областях, включаючи атаки на моделі машинного навчання, навчання з використанням моделі заміни, а також інтерпретацію та пояснення складних моделей.

У контексті атак на моделі машинного навчання (adversarial attacks) surrogate classifier грає ключову роль, особливо в сценаріях, коли зловмисник не володіє доступом до вихідної моделі, що характерно для атак типу "чорного ящика" (black-box attack). В умовах обмеженої інформації про структуру і параметри цільового класифікатора, атакуючий може створити власну заміщає модель, використовуючи обмежений набір даних. На основі такої моделі розробляються змагальні приклади (adversarial examples), які потім переносяться на цільову модель, що дозволяє реалізувати атаку шляхом перенесення (transfer attack). Цей підхід дозволяє обійти обмеження, пов'язані з відсутністю прямого доступу до вихідного класифікатора, і продемонструвати його вразливість перед модифікованими вхідними даними.

Сурогатний і цільовий класифікатори будуть навчені на різних навчальних наборах.

Додаткову інформацію при підготовці до роботи можна отримати:

1. Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C. and Roli, F., 2019. Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks. In 28th Usenix Security Symposium, Santa Clara, California, USA. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/demontis

ТЕОРЕТИЧНІ ВІДОМОСТІ

Широке впровадження машинного навчання (ML) та алгоритмів глибокого навчання у багатьох критичних додатках створює сильні стимули для мотивованих зловмисників маніпулювати результатами та моделями, що генеруються цими алгоритмами. Атаки на системи машинного навчання можуть відбуватися на декількох етапах процесу навчання. Наприклад, у багатьох випадках дані про навчання збираються в Інтернеті, і тому їм не можна повністю довіряти. При атаках з порушенням доступності зловмисник контролює певний обсяг навчальних даних, тим самим впливаючи на навчену модель і, в кінцевому рахунку, на прогнози під час тестування по більшості точок в тестовому наборі. Атаки, що порушують цілісність, спрямовані на зміну прогнозів по декількох цільових точках шляхом маніпулювання процесом навчання. З іншого боку, атаки ухилення включають в себе невеликі маніпуляції з точками даних тестування, що призводить до неправильного прогнозування під час тестування цих точок]. Створення точок отруєння та ухилення від атак є непростим завданням, особливо коли багато онлайн-сервісів уникають розкриття інформації про свої алгоритми машинного навчання. В результаті зловмисники змушені розробляти свої атаки в налаштуваннях black-box, використовуючи сурогатну модель замість реальної моделі, що використовується сервісом, сподіваючись, що атака буде ефективною на реальній моделі. Властивість переносимості атаки виконується, коли атака, розроблена для конкретної моделі машинного навчання (тобто сурогатної моделі), також ефективна проти цільової моделі. Можливість перенесення атак спостерігалася в ранніх дослідженнях на прикладах змагальності і в останні роки викликала набагато більший інтерес з розвитком хмарних сервісів машинного навчання.

Класифікація атак за трьома основними ознаками

Раніше ми загалом розглянули, як моделі машинного навчання можуть поводитися в змагальних умовах. Далі нам потрібно не просто зрозуміти, як машинне навчання можна використовувати у змагальних умовах (наприклад,

для виявлення шкідливого ПЗ), але і яким чином такі параметри вносять уразливості до традиційних підходів до навчання. Принципове обговорення таких вразливостей зосереджено навколо точних моделей загроз. Розглянемо загальну класифікацію моделей загроз чи атак на алгоритми машинного навчання. Було зроблено кілька спроб класифікувати атаки на алгоритми машинного навчання. Пропонована нижче класифікація пов'язана з деякими з них і спрямована на виявлення найбільш важливих особливостей атак, які ми обговорюємо. Зокрема, ми класифікуємо атаки за трьома параметрами: час, інформація та цілі.

1. Час. Перше, що необхідно враховувати при моделюванні атак, це коли відбувається атака. Це міркування призводить до наступної загальної дихотомії, яка є центральною для атак на машинне навчання: атаки на моделі (з яких атаки ухилення – evasion attacks – є найбільш типовими випадками) та атаки на алгоритми (широко відомі як атаки з отруєнням – poisoning attacks). Атаки на моделі або, точніше, на рішення, прийняті навченими моделями, припускають, що модель вже вивчена, і зловмисник тепер або змінює її поведінку, або вносить зміни в середовище, щоб змусити модель робити помилкові прогнози. Атаки отруєння, навпаки, відбуваються до навчання моделей, змінюючи частину даних, що використовуються для навчання (рис. 2.1).

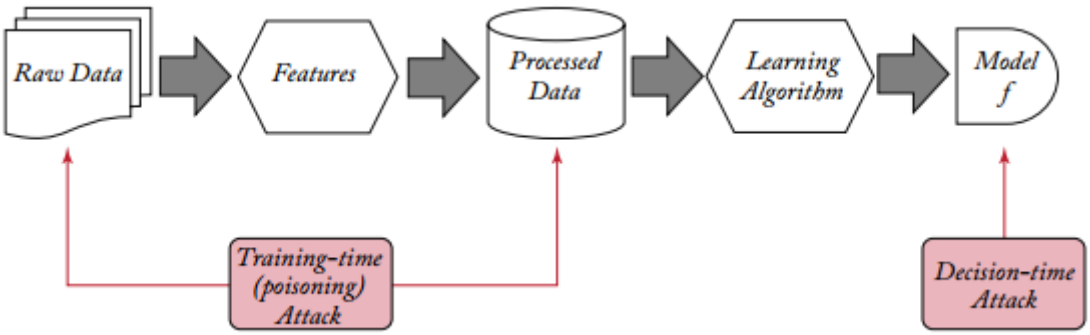


Рис. 2.1. Схематичне зображення різниці між атаками часу прийняття рішення (атаками на моделі) та атаками отруєння (атаками на алгоритми)

2. Інформація. Другим важливим питанням при моделюванні атак є те, яка інформація є у зловмисника про модель навчання або алгоритм, відмінність, яка зазвичай зводиться до атак «білого ящика» (white-box attack) і «чорного ящика» (black-box attack). Зокрема, атаки «білої скриньки» припускають, що або модель (у разі атак на рішення), або алгоритм (в атаках з отруєнням) повністю відомі противнику, тоді як в атаках «чорного ящика»

зловмисник має обмежену інформацію або не має інформації про ці атаки, хоча частина інформації може отримати побічно, наприклад, через запити.

3. Цілі. У зловмисників можуть бути різні причини для атаки, наприклад, ухилення від виявлення або зниження довіри до алгоритму. Ми розрізняємо два широкі класи цілей атаки: цільові атаки та атаки на надійність методу навчання (або просто атаки на надійність). При цілеспрямованій атаці ціль зловмисника полягає в тому, щоб викликати помилку в конкретних примірниках певного характеру (наприклад, змусити навчену функцію *f* передбачити конкретну помилкову мітку *l* для екземпляра *x*). Навпаки, атака на надійність спрямована на зниження сприйманої надійності системи навчання за рахунок максимізації помилки прогнозу.

Класифікація за часом здійснення атаки

Атаки під час ухвалення рішення. З усіх класів атак, які ми розглядатимемо, атаки з ухиленням – основний підклас атак, які відбуваються під час прийняття рішення, – мабуть, найбільш історично помітні. Відомим прикладом ухилення є еволюція спам-трафіку електронної пошти, наприклад, коли спамери замінюють букву "о" на цифру "0" у слові "Лотерея" (яке стає "Л0терея").

Загалом, атака ухилення від класифікатора на двійкові класифікатори приймає як вхідні дані класифікатор *f(x)* і «ідеальний» екземпляр у просторі ознак, x_{ideal} (тобто це те, що противник хотів би відправити, якби не було класифікатора для ідентифікації цих даних як шкідливих). Потім атака виводить інший екземпляр, який відповідає вектору ознак *x'*. Якщо *f(x')=-1*, ухилення успішно, але ймовірно, що противнику не вдасться знайти адекватне ухилення (фактично, для будь-якої значущої міри стійкості алгоритму до ухилення необхідно, щоб противник не міг знайти ухилення, якими б не були *f* і x_{ideal}).

В якості ілюстрації припустимо, що хтось хоче виявити спам в електронній пошті та навчає для цієї мети класифікатор *f(x)* (де *x* - вектор, що представляє характеристики електронної пошти). Тепер розглянемо спамера, який раніше використовував шаблон, що відповідає вектору ознак x_{spam}, і припустимо, що $f(x_{spam})$ позначає його як спам (+1), так що спамер не отримує реакції на свої листи. Спамер буде вносити зміни до електронного листа, щоб отримати екземпляр, який у функціональному просторі виглядає як *x'* з властивістю *f(x')=1* (тобто він класифікується як спам і може передаватися в поштові скриньки користувачів). Але *x'* не може бути довільним: зловмисник несе витрати на зміну вихідного екземпляра x_{spam} для досягнення *x'*, що може

вимірювати вартість зусиль (для підтримки функціональності) або ефективність (наприклад, йому, можливо, доведеться внести орфографічні помилки, які дозволять зловмиснику уникнути виявлення, але також зменшать ймовірність того, що люди).

Узагальнюючи ідею атак ухилення, ми можемо розглянути атаки часу ухвалення рішення на багатокласову класифікацію. Нехай Y буде кінцевим набором міток і припустимо, що для деякого екземпляра x_{ideal} передбачена мітка дорівнює $f(x_{ideal})=y$. Зловмисник може захотіти змінити цей екземпляр на інший, x', або для отримання невірного передбачення ($f(x') \neq y$), або для того, щоб класифікатор передбачив цільову мітку $t=f(x')$. Останнім часом такі атаки привернули велику увагу під терміном «змагання», в основному в додатках машинного зору та глибоких нейронних мережах. Потенційна проблема полягає в тому, що зловмисник може спробувати викликати аварію автономного транспортного засобу, що покладається на зір, шляхом маніпулювання зображенням дорожнього знака, такого як знак зупинки (наприклад, шляхом розміщення спеціально створених наклейок, які здаються перехожим графіті).

Атаки на навчальні дані. Проблема навчання з пошкодженими або зашумленими навчальними даними була предметом серйозних досліджень у товариствах машинного навчання та статистики протягом кількох десятиліть. Однак останнім часом зловмисне ушкодження навчальних даних почало розглядатися більш систематично, особливо якщо ми допускаємо спотворення значної частини даних. Природа отруйних атак у тому, що зловмисник навмисно маніпулює навчальними даними до навчання, щоб алгоритм навчання робив неправильний вибір. Важлива концептуальна проблема з отруйними атаками полягає у визначенні обсягу зловмисником маніпулювання навчальними даними та цілей зловмисника при цьому. Один із найпоширеніших способів обійти ці проблеми – припустити, що зловмисник може вносити довільні зміни до невеликого підмножини точок навчальних даних. Тоді метою буде розробка алгоритмів, стійких до такого довільного пошкодження даних, якщо кількість пошкоджених даних досить мала.

Можна також розглянути більш конкретні моделі псування даних, які накладають додаткові обмеження на дії зловмисника. Одним з найпоширеніших класів таких атак є атаки з перевертанням міток, коли зловмиснику дозволено змінювати мітки не більше ніж С об'єктів навчальних даних. Як правило, такі атаки розглядаються в контексті класифікації, хоча

можна також змінити мітки регресії. Найчастіше передбачається, що алгоритм і простір ознак відомі противнику (тобто це white-box-атака). Отруєння даних можна розглядати в умовах неконтрольованого навчання, наприклад, коли воно використовується для виявлення аномалій. У цьому випадку зловмисник може внести невеликі зміни в нормальну поведінку, що спостерігається, яка тепер забруднює модель, використовувану для виявлення аномалій, з метою гарантувати, що майбутня атака мети буде позначена як нешкідлива.

Класифікація за інформацією, доступною зловмиснику

Одним з найбільш важливих факторів при моделюванні атаки є інформація, яку має зловмисник про систему, яку він атакує. Ми проводимо різницю між атаками «білого ящика», коли зловмисник знає все, що потрібно знати, та атаками «чорного ящика», коли зловмисник має обмежену інформацію.

Атаки «білого ящика» припускають, що противник точно знає або вивчену модель (наприклад, фактичний класифікатор) у разі атак на час ухвалення рішення, або алгоритм навчання у разі атаки отруєння. Це означає, наприклад, що зловмиснику відомі всі параметри моделі, у тому числі ознаки, а в разі атаки з отруєнням гіперпараметри алгоритму навчання. Припущення про те, що зловмисник має таку інформацію про навчальну систему, може здатися підозрілим. Тим не менш, є важливі причини, щоб розглянути атаки білої скриньки.

По-перше, вони пропонують природну відправну точку з погляду моделі: якщо модель стійка до атак білої шухляди, вона, безумовно, стійка і до атак, які обмежені в інформації.

По-друге, з погляду зловмисника, може бути кілька способів опосередковано отримати достатню інформацію про вивчену модель для розгортання успішної атаки. Візьмемо, наприклад, атаку відхилення при детекції шкідливого ПЗ. Припустимо, що набір функцій, що використовуються, є загальнодоступною інформацією (наприклад, з опублікованих робіт), а набори даних, що використовуються для навчання детектора шкідливих програм, є загальнодоступними (або, альтернативно, існують загальнодоступні набори даних, які досить схожі на дані, що фактично використовуються для навчання). Нарешті, припустимо, що модель використовує стандартний алгоритм навчання вивчення моделі, такий як випадковий ліс, глибока нейронна мережу чи машина опорних векторів, і стандартні методи налаштування гіперпараметрів, такі як перехресна

перевірка. У цьому випадку зловмисник може отримати ідентичну або майже ідентичну версію детектора, що використовується насправді!

При атаках методом «чорного ящика», на відміну від атак «білого ящика», зловмисник не має точної інформації ні про модель, ні про алгоритм, який навчається. Важливим завданням моделювання атак методом «чорного ящика» є точне моделювання того, яка інформація є у зловмисника або про вивчену модель, або про алгоритм.

У контексті атак методом «чорного ящика» під час прийняття рішення один із підходів полягає у розгляді ієрархії інформації про вивчену модель, доступну зловмиснику. З погляду протилежного підходу, противнику взагалі недоступна жодна інформація. Більш поінформованого противника можуть бути деякі дані для навчання, які відрізняються від даних, на яких була навчена реальна модель, але він не може мати інформації про конкретний клас моделі або про функції, що використовуються. Більш поінформований зловмисник може знати клас і функції моделі і, можливо, алгоритм навчання, але не мати навчальних даних, а ще більш поінформований противник може мати навчальні дані, відібрані з того ж розподілу, що і дані, що використовуються для навчання. Нарешті, коли цей супротивник має фактичні навчальні дані, використовувані алгоритмом навчання, результуюча атака еквівалентна атаці білого ящика, як обговорювалося вище, оскільки зловмисник може дізнатися точну модель із заданих навчальних даних. Можна помітити, що, на відміну від атак "білого ящика", існує безліч способів моделювання атак "чорного ящика". Справді, існує термін атака сірого ящика, що вказує, що зловмисник має деяку, хоч і неповну, інформацію про систему, що атакується.

Наведена вище інформаційна ієрархія не відповідає на природне питання: звідки зловмисник отримує інформацію про модель, яку він атакує (у разі атак часу ухвалення рішення)? Важливий клас моделей атаки «чорного ящика» вирішує це питання, дозволяючи зловмисникові отримати доступ до вивченої моделі із запитом. Зокрема, для довільного об'єкта, представленого у вигляді вектора ознак x, зловмисник може одержати (запросити) фактичну мітку $y=f(x)$ для невідомої моделі чорного ящика f. Зазвичай і неявно такі моделі запитів також припускають, що зловмисник знає модельний простір (наприклад, алгоритм навчання), так і функціональний простір. Крім того, ще одним практичним обмеженням цієї моделі є те, що $f(x)$ часто спостерігається неточно або з шумом при заданому x. Наприклад, припустимо, що спамер надсилає спам-повідомлення

електронною поштою. Відсутність відповіді не обов'язково означає, що вона була відфільтрована – ймовірно, можливо, що користувачі просто проігнорували електронного листа. Тим не менш, така структура, заснована на запитах, дозволяє провести елегантне теоретичне дослідження того, що може зробити зловмисник, маючи дуже обмежену інформацію про учня.

Дотримуючись тих же принципів, що й типові моделі атак методом «чорного ящика» під час прийняття рішення, атаки «чорного ящика» з отруєнням даних дозволять отримати низку знань про алгоритм, який використовується захисником. Наприклад, в одному крайньому випадку зловмисник може взагалі не мати інформації про алгоритм. Більш поінформований зловмисник може знати алгоритм, але не знати гіперпараметри (такі як вага регуляризації або кількість прихованих шарів нейронної мережі) або функції. Більш поінформований зловмисник може знати алгоритм, функції та гіперпараметри, але не навчальні дані, які зловмисник намагається отруїти.

Класифікація за метою атакуючого

Хоча у зловмисників може бути безліч можливих цілей для здійснення атаки на системи машинного навчання, ми ділимо атаки на дві основні категорії з точки зору цілей зловмисника: цільові атаки та атаки на надійність.

Цільові атаки характеризуються конкретною метою зловмисника щодо модельних рішень. Наприклад, розглянемо атаку часу прийняття рішення на мультикласовий класифікатор з набором можливих міток L, і нехай x буде конкретним екземпляром, який представляє інтерес для зловмисника з істинною міткою y. Метою спрямованої атаки у разі буде зміна мітки для x на конкретну цільову мітку $t \neq y$. У загальному сенсі цільова атака характеризується підмножиною S простору об'єктів і міток $X \times Y$, котрим зловмисник хотів би змінити рішення, і навіть цільової функцією прийняття рішення $D(x)$. У найпоширеніших умовах цільових атак навчання з учителем зловмисник прагнутиме отримати передбачення кожного (x, y) з S, щоб вони відповідали цільової функції мітки $l(x)$.

Атаки на надійність, з іншого боку, намагаються максимізувати помилки у рішеннях, зроблених шляхом навчання щодо істини. Наприклад, під час навчання з учителем зловмисник прагнутиме максимізувати помилку передбачення. У програмах машинного зору такі атаки, які зазвичай називають нецільовими, змінювати зображення таким чином, щоб викликати помилкове передбачення (наприклад, розпізнавання об'єкта, відсутнього на

зображенні, наприклад, помилкове прийняття зображення знака зупинки будь-який інший дорожній знак).

Різниця між цілеспрямованими атаками та атаками на надійність стирається, коли ми розглядаємо бінарну класифікацію: зокрема, атаки на надійність тепер стають окремим випадком, у якому мітки мети $l(x)$ є просто альтернативними мітками. У більш загальному плані ми відзначаємо, що навіть поділ між цілеспрямованими атаками та атаками на надійність є неповним: наприклад, можна розглядати атаки, метою яких є уникнення передбачень певного класу, відмінного від правильної мітки.

ПРИКЛАДИ ПРАКТИЧНИХ ЗАВДАНЬ

Крок 1. У браузері перейдіть по https://www.kaggle.com/

Якщо ви бачите синю кнопку "Увійти" у верхньому правому куті, натисніть на неї та увійдіть в обліковий запис Kaggle. У меню виберіть "Файл", "Нова записна книжка".

Крок 2. Встановлення SecML

Виконайте наступні команди:

```
!pip install secml
import secml
```

Бібліотека встановиться, як показано нижче (рис. 2.2).

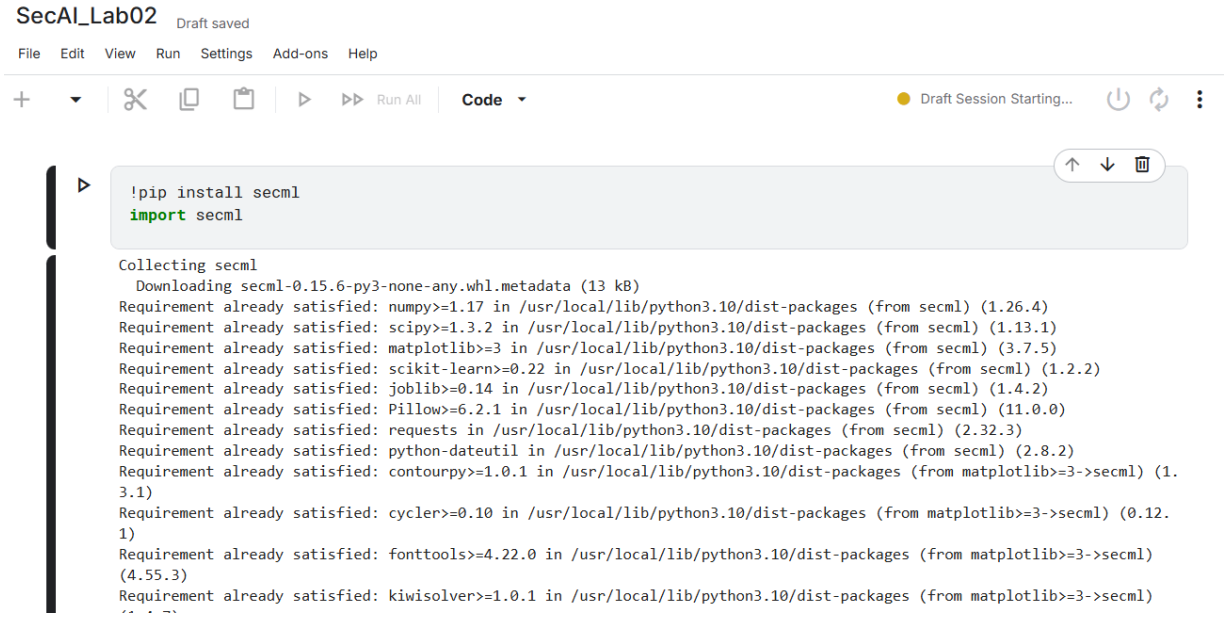


Рис. 2.2. Встановлення SecML

Крок 3. Підготовка набору даних

Сурогатний та цільові класифікатори потрібно навчати на різних навчальних вибірках. Це дозволяє відтворити умови реальних атак "чорного ящика" (black-box attack), коли структура та параметри цільової моделі невідомі.

Наведений нижче програмний код реалізує процес генерації даних, навчання замінного (surrogate) класифікатора та декількох цільових (target) класифікаторів. Основна мета – підготувати моделі для подальшого аналізу їхньої стійкості до атак.

1. Генерація даних

```
random_state = 999
n_features = 2
n_samples = 2250
centers = [[-2, 0], [2, -2], [2, 2]]
cluster_std = 0.8
```

random_state = 999 – фіксує початковий стан генератора випадкових чисел, щоб щоразу отримувати однаковий результат.

n_features = 2 – кількість ознак (двовимірний простір).

n_samples = 2250 – загальна кількість точок у наборі даних.

centers – координати центрів кластерів, навколо яких будуть згруповані точки.

cluster_std = 0.8 – стандартне відхилення точок у кожному кластері (визначає розподіл даних).

```
from secml.data.loader import CDLRandomBlobs
dataset = CDLRandomBlobs(n_features=n_features,
                         centers=centers,
                         cluster_std=cluster_std,
                         n_samples=n_samples,
                         random_state=random_state).load()
```

Генерується набір випадкових кластеризованих даних за допомогою CDLRandomBlobs.

2. Розділення даних на тренувальну та тестову вибірки

```
n_tr = 1000
n_ts = 250


from secml.data.splitter import CTrainTestSplit
```

```
splitter = CTrainTestSplit(train_size=2 * n_tr, test_size=n_ts, random_state=random_state)
tr, ts = splitter.split(dataset)
```

n_tr = 1000 – кількість зразків для навчання surrogate-класифікатора.

n_ts = 250 – кількість тестових зразків.

splitter.split(dataset) — розділяє дані на тренувальний (tr) і тестовий (ts) набори.

3. Нормалізація даних

```
from secml.ml.features import CNormalizerMinMax
nmz = CNormalizerMinMax()
tr.X = nmz.fit_transform(tr.X)
ts.X = nmz.transform(ts.X)
```

Використовується мін-макс нормалізація, яка приводить всі значення у діапазон [0,1].

4. Розділення тренувальних даних для surrogate- і target-класифікаторів

```
tr1 = tr[:n_tr, :]  # Дані для surrogate-класифікатора
tr2 = tr[n_tr:, :]  # Дані для target-класифікаторів
```

tr1 використовується для навчання замінного (surrogate) класифікатора.

tr2 використовується для навчання цільових (target) класифікаторів.

5. Створення surrogate-класифікатора

```
from collections import namedtuple
CLF = namedtuple('CLF', 'clf_name clf xval_parameters')

from secml.ml.classifiers.multiclass import CClassifierMulticlassOVA
from secml.ml.classifiers import CClassifierSVM

surr_clf = CLF(
   clf_name='SVM Linear',
   clf=CClassifierMulticlassOVA(CClassifierSVM, kernel='linear'),
   xval_parameters={'C': [1e-2, 0.1, 1]})
```

Створюється surrogate-класифікатор, заснований на лінійному SVM (CClassifierSVM).

Використовується OVA (One-Versus-All) схема, що дозволяє вирішувати багатокласові задачі.

Оптимізується параметр C (коефіцієнт регуляризації) з можливими значеннями [1e-2, 0.1, 1].

6. Налаштування гіперпараметрів та навчання surrogate-класифікатора

```
from secml.data.splitter import CDataSplitterKFold
xval_splitter = CDataSplitterKFold(num_folds=3, random_state=random_state)

from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()

best_params = surr_clf.clf.estimate_parameters(
    dataset=tr1,
    parameters=surr_clf.xval_parameters,
    splitter=xval_splitter,
    metric=metric,
    perf_evaluator='xval'
)
```

Виконується 3-Fold крос-валідація для підбору оптимального C.

Використовується метрика точності (CMetricAccuracy).

Метод estimate_parameters вибирає найкращий C за підсумками крос-валідації.

```
surr_clf.clf.fit(tr1.X, tr1.Y)
y_pred = surr_clf.clf.predict(ts.X)
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

print("Точність surrogate-класифікатора на тестовому наборі: {:.2%}".format(acc))
```

Після налаштування параметрів модель навчається на tr1.

Виконується прогнозування на тестовому наборі та обчислюється точність.

7. Створення та навчання цільових (target) класифікаторів

```
target_clf_list = [
```

```
        CLF(clf_name='SVM Linear',
            clf=CClassifierMulticlassOVA(CClassifierSVM, kernel='linear'),
            xval_parameters={'C': [1e-2, 0.1, 1]}),
        CLF(clf_name='SVM RBF',
            clf=CClassifierMulticlassOVA(CClassifierSVM, kernel='rbf'),
            xval_parameters={'C': [1e-2, 0.1, 1], 'kernel.gamma': [1, 10, 100]}),
        CLF(clf_name='Logistic (SGD)',
            clf=CClassifierMulticlassOVA(CClassifierSGD, regularizer='l2',
loss='log',
                random_state=random_state),
            xval_parameters={'alpha': [1e-6, 1e-5, 1e-4]}),
        CLF(clf_name='kNN',
            clf=CClassifierKNN(),
            xval_parameters={'n_neighbors': [30, 40, 50]}),
        CLF(clf_name='Decision Tree',
            clf=CClassifierDecisionTree(random_state=random_state),
            xval_parameters={'max_depth': [1, 3, 5]}),
        CLF(clf_name='Random Forest',
            clf=CClassifierRandomForest(random_state=random_state),
            xval_parameters={'n_estimators': [20, 30, 40]}),
    ]
```

Визначаються кілька цільових класифікаторів, зокрема SVM, kNN, Decision Tree, Random Forest та логістична регресія (SGD) (рис.2.3).

Для кожного класифікатора налаштовуються параметри, які будуть оптимізовані за допомогою крос-валідації.

```
    for i, test_case in enumerate(target_clf_list):

        clf = test_case.clf
        xval_params = test_case.xval_parameters

        best_params = clf.estimate_parameters(
            dataset=tr2, parameters=xval_params, splitter=xval_splitter,
            metric='accuracy', perf_evaluator='xval')

        clf.fit(tr2.X, tr2.Y)

        y_pred = clf.predict(ts.X)
        acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)
```

```
        print("Класифікатор: {:}\tТочність: {:.2%}".format(test_case.clf_name, acc))
```

```
Estimating the best training parameters of the surrogate classifier...
The best training parameters of the surrogate classifier are:  [('C', 0.1)]
Accuracy of the surrogate classifier on test set: 99.60%

Training the target classifiers...

Estimating the best training parameters of SVM Linear ...
The best parameters for 'SVM Linear' are:  [('C', 0.1)]
Training of SVM Linear ...
Classifier: SVM Linear  Accuracy: 99.60%

Estimating the best training parameters of SVM RBF ...
The best parameters for 'SVM RBF' are:  [('C', 0.01), ('kernel.gamma', 10)]
Training of SVM RBF ...
Classifier: SVM RBF     Accuracy: 99.60%

Estimating the best training parameters of Logistic (SGD) ...
The best parameters for 'Logistic (SGD)' are:  [('alpha', 1e-06)]
Training of Logistic (SGD) ...
Classifier: Logistic (SGD)      Accuracy: 99.60%

Estimating the best training parameters of kNN ...
The best parameters for 'kNN' are:  [('n_neighbors', 30)]
Training of kNN ...
Classifier: kNN Accuracy: 99.60%

Estimating the best training parameters of Decision Tree ...
The best parameters for 'Decision Tree' are:  [('max_depth', 3)]
Training of Decision Tree ...
Classifier: Decision Tree       Accuracy: 98.80%

Estimating the best training parameters of Random Forest ...
The best parameters for 'Random Forest' are:  [('n_estimators', 30)]
Training of Random Forest ...
Classifier: Random Forest       Accuracy: 98.80%
```

Рис. 2.3. Оцінка точності класифікаторів

Кожен класифікатор проходить налаштування, навчання та тестування.

Цей код генерує дані, навчає surrogate-класифікатор та кілька target-класифікаторів, готуючи їх до подальшого аналізу та атак.

Крок 2. Генерація змагальних прикладів

У цьому розділі буде здійснено генерацію змагальних прикладів за допомогою алгоритму максимального підвищення довіри (gradient-based maximum-confidence), який реалізовано у класі CAttackEvasionPGDLS (e-pgd-

ls). Цей метод використовується для створення атак на класифікатор з метою змінення його рішень.

На відміну від попередніх прикладів, де розглядалися неспецифічні атаки (error-generic attacks), у цьому випадку буде створено цілеспрямовану атаку (error-specific attack). Для цього параметр y_target буде встановлено в один із класів датасету. Таким чином, атакуючий алгоритм модифікує вхідні дані так, щоб класифікатор помилково відніс їх до обраного цільового класу.

ПРИМІТКА. Слід враховувати, що генерація атак може бути часозатратною, особливо при обробці великої кількості зразків. Тривалість виконання процесу залежить від обчислювальної потужності пристрою, на якому запускається скрипт, і може тривати від кількох секунд до декількох хвилин.

Наведений у цьому розділі код реалізує цілеспрямовану (error-specific) змагальну атаку на класифікатор за допомогою Projected Gradient Descent with Line Search (PGD-LS), який реалізований у класі CAttackEvasionPGDLS бібліотеки SecML. Основна мета атаки – змінити передбачення класифікатора таким чином, щоб всі атаковані зразки були віднесені до певного цільового класу (y_target = 2).

1. Налаштування параметрів атаки

```
noise_type = 'l2'  # Тип норми збурення: 'l1' або 'l2'
dmax = 0.4  # Максимальна величина збурення
lb, ub = 0, 1  # Межі допустимого простору атак. `None` означає необмежене значення
y_target = 2  # Цільова атака (error-specific). `None` для загальної атаки (error-generic)
```

noise_type = 'l2' – використовується L2-норма для оцінки рівня збурення. Якщо noise_type = 'l1', збурення оцінюється за допомогою L1-норми.

dmax = 0.4 – максимальне допустиме збурення, яке можна додати до вхідних даних.

lb, ub = 0, 1 – межі для значень даних (нижня lb=0, верхня ub=1). Якщо None, атака не обмежується за амплітудою змін.

y_target = 2 – цілеспрямована атака (error-specific attack), яка змушує класифікатор помилково класифікувати зразки як клас 2. Якщо y_target

= None, атака буде загальною (error-generic), тобто просто змушуватиме модель робити помилки без конкретного класу.

2. Налаштування параметрів оптимізації

```
solver_params = {
    'eta': 1e-1,
    'eta_min': 0.1,
    'eta_max': None,
    'max_iter': 100,
    'eps': 1e-4
}
```

eta = 1e-1 – початковий розмір кроку градієнтного спуску.
eta_min = 0.1 – мінімальний розмір кроку (нижня межа).
eta_max = None – відсутність обмежень на максимальний крок (якщо не вказано, адаптується автоматично).
max_iter = 100 – максимальна кількість ітерацій для знаходження оптимального збурення.
eps = 1e-4 – критерій зупинки (якщо зміни стали меншими за 1e-4, оптимізація припиняється).

3. Ініціалізація атаки PGD-LS

```
from secml.adv.attacks.evasion import CAttackEvasionPGDLS
pgd_ls_attack = CAttackEvasionPGDLS(
    classifier=surr_clf.clf,      # Атакований класифікатор (замінний класифікатор)
    double_init_ds=tr1,      # Використання навчальних даних для ініціалізації
    double_init=False,    # Відключення подвійної ініціалізації (може покращити атаку)
    distance=noise_type,    # Використання L2-норми для вимірювання збурень
    dmax=dmax,  # Максимальне збурення
    lb=lb, ub=ub,  # Межі значень вхідних даних
    solver_params=solver_params,  # Передача параметрів оптимізації
    y_target=y_target)  # Вказання цільового класу для атаки
```

classifier=surr_clf.clf – цільова модель (замінний класифікатор surr_clf).

double_init_ds=tr1 – використання навчальних даних (tr1) для вибору стартової точки атаки.

double_init=False – відключена подвійна ініціалізація (може бути корисна в деяких випадках).

distance=noise_type – використовується L2-норма для обчислення рівня збурення.

dmax=dmax – максимальне допустиме збурення.

y_target=y_target – встановлюється цільовий клас (2), до якого потрібно примусово змінити класифікацію.

4. Запуск атаки

```
print("Attack started...")
y_pred, scores, adv_ds, f_obj = pgd_ls_attack.run(ts.X, ts.Y)
print("Attack complete!")
```

pgd_ls_attack.run(ts.X, ts.Y) – виконання атаки на тестові зразки (ts.X) з відповідними мітками (ts.Y). y_pred – передбачені мітки після атаки (очікується, що вони стануть 2). scores – ймовірності передбачень. adv_ds – набір состязательных примеров (adversarial examples) після атаки. f_obj – значення функції втрат (може використовуватись для оцінки ефективності атаки).

Наведений у розділі код:

1. Налаштовує параметри атаки, визначаючи тип збурень (L2), межі (dmax = 0.4) та цільовий клас (y_target = 2).

2. Ініціалізує PGD-LS-атаку, яка намагається змінити рішення класифікатора.

3. Запускає атаку на тестовий набір (ts.X) та отримує атаковані приклади (adv_ds).

Крок 3. Аналіз перенесення атак (Transferability Analysis)

Цей крок спрямований на оцінку переносимості (transferability) змагальних атак на інші моделі. Основна ідея полягає в тому, щоб перевірити, чи згенеровані на одній моделі змагальні приклади залишаються ефективними проти інших класифікаторів.

Кроки аналізу

1. Тестування цільових моделей на змагальних прикладах

- Використовуємо атаковані зразки, створені для surrogate-класифікатора.
- Перевіряємо, наскільки добре вони змушують інші моделі (target classifiers) помилятися.
- Оцінюємо точність кожного класифікатора на змагальних прикладах та порівнюємо з базовою продуктивністю.

2. Візуалізація прикладів у 2D-просторі

- Відображаємо деякі змагальні приклади на площині, щоб побачити, як атака змінила їхні характеристики.
- Аналізуємо розташування атакованих точок щодо рішень цільових моделей.

Очікувані результати

- Якщо змагальні приклади добре переносяться між моделями, це свідчить про вразливість більшості класифікаторів до атак.
- Якщо точність моделей суттєво падає, це підтверджує ефективність атаки та її здатність до генералізації.
- Візуалізація дозволяє краще зрозуміти, які особливості були змінені під час атаки та як це вплинуло на рішення класифікаторів.

Наведений нижче код оцінює перенесення атаки (transferability), визначаючи, наскільки ефективними залишаються змагальні приклади (adversarial examples) при тестуванні на інших моделях.

Основна мета – перевірити, чи атаковані зразки, створені для surrogate-класифікатора, також змушують інші класифікатори помилятися.

1. Налаштування метрики для оцінки атак

```
from secml.ml.peval.metrics import CMetricTestError
metric = CMetricTestError()
```

Імпортується метрика CMetricTestError, яка використовується для оцінки помилки класифікації.

Чим вища помилка, тим сильніший ефект атаки на модель.

2. Оцінка помилки для кожного цільового класифікатора

```
trans_error = []
transfer_rate = 0.0
for target_clf in target_clf_list:
```

trans_error = [] – список для збереження помилок кожної моделі після атаки.

transfer_rate = 0.0 – середній рівень перенесення атаки.

Цикл перебирає всі цільові класифікатори (target_clf_list).

```
print("\nTesting transferability of {:}".format(target_clf.clf_name))
```

Виводиться повідомлення з назвою поточного тестованого класифікатора.

3. Обчислення початкової помилки (без атаки)

```
origin_error = metric.performance_score(
        y_true=ts.Y, y_pred=target_clf.clf.predict(ts.X))

print("Test error (no attack): {:.2%}".format(origin_error))
```

Передбачення (target_clf.clf.predict(ts.X)) виконується для тестового набору без атак.

Підраховується точність origin_error, яка показує помилку без атаки.

4. Оцінка помилки після атаки

```
trans_error_clf = metric.performance_score(
   y_true=ts.Y, y_pred=target_clf.clf.predict(adv_ds.X))
```

Передбачення виконується на атакованих прикладах (adv_ds.X).

Обчислюється помилка (trans_error_clf), яка показує, як сильно атака вплинула на модель.

```
trans_error.append(trans_error_clf)
transfer_rate += trans_error_clf
```

Додається помилка trans_error_clf до загального списку.

Значення transfer_rate накопичується для подальшого обчислення середнього рівня перенесення атаки.

5. Обчислення середнього рівня перенесення атаки

```
transfer_rate /= len(target_clf_list)
```

Обчислюється середнє значення рівня перенесення атаки по всіх класифікаторах.

6. Візуалізація результатів перенесення атаки

```
from secml.array import CArray
trans_acc = CArray(trans_error) * 100  # Перетворення в проценти
```

Помилки перетворюються у відсотки, що зручно для графічного аналізу.

```
from secml.figure import CFigure
%matplotlib inline  # Тільки для Jupyter Notebook
```

CFigure використовується для побудови графіків.

%matplotlib inline потрібен для коректного відображення графіків у Jupyter Notebook.

```
fig = CFigure(height=1)
a = fig.sp.imshow(trans_acc.reshape((1, 6)),
                  cmap='Oranges', interpolation='nearest',
                  alpha=.65, vmin=60, vmax=70)
```

Створюється теплокарта (imshow) із відображенням рівня помилки у відсотках.

Використовується колірна схема "Oranges", де більш насичені кольори показують вищий рівень перенесення атаки.

vmin=60, vmax=70 задає діапазон значень.

7. Налаштування підписів графіка

```
fig.sp.xticks(CArray.arange((len(target_clf_list))))
fig.sp.xticklabels([c.clf_name for c in target_clf_list],
                   rotation=45, ha="right", rotation_mode="anchor")
```

Підписи для осі X: імена класифікаторів, які тестувалися.

Обертання підписів на 45°, щоб зробити їх більш читабельними.

```
fig.sp.yticks([0])
fig.sp.yticklabels([surr_clf.clf_name])
```

Підпис для осі Y: ім’я surrogate-класифікатора, на якому створювалася атака.

8. Додавання числових значень до графіка

```
for i in range(len(target_clf_list)):
    fig.sp.text(i, 0, trans_acc[i].round(2).item(), va='center', ha='center')
```

У кожну клітинку теплокарти додається точне числове значення рівня перенесення атаки.

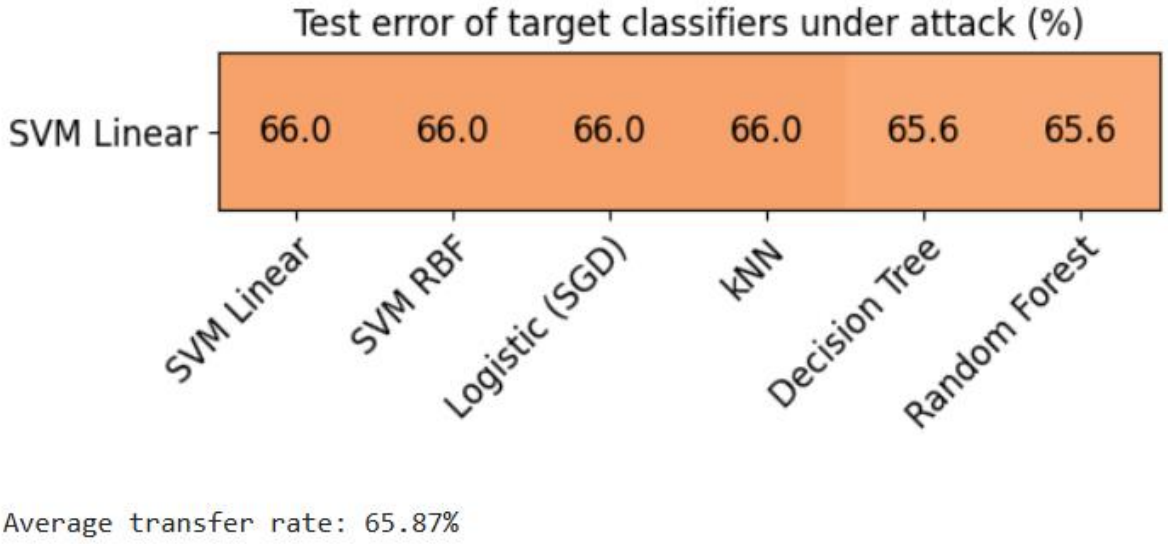
```
fig.sp.title("Test error of target classifiers under attack (%)")
fig.show()
```

Встановлюється заголовок графіка та виводиться графік.

9. Виведення середнього рівня перенесення атаки

```
print("\nAverage transfer rate: {:.2%}".format(transfer_rate))
```

Виводиться середнє значення рівня перенесення атаки у відсотках (рис. 2.4).




```
Average transfer rate: 65.87%
```

Рис. 2.4. Середнє значення рівня перенесення

Рис. 2.4 відображає графік рівня помилок цільових класифікаторів під впливом змагальних атак, що дозволяє оцінити ефективність атакованих прикладів, створених для SVM Linear, у введенні в оману інших моделей. Аналіз отриманих даних свідчить про те, що рівень помилок для всіх класифікаторів знаходиться в межах від 65.6% до 66.0%. Це вказує на однорідність впливу атаки на різні моделі. Найвищий рівень помилок спостерігається у SVM Linear, SVM RBF, Logistic (SGD) та kNN, що свідчить про їхню вразливість. Водночас Decision Tree та Random Forest демонструють дещо кращу стійкість до атаки, оскільки їхній рівень помилок є найнижчим серед тестованих моделей.

Середній рівень перенесення атаки становить 65.87%, що свідчить про те, що понад 65% атакованих прикладів змогли викликати помилкові передбачення в усіх тестованих класифікаторах. Висока переносимість атаки вказує на її узагальнену ефективність, незалежно від архітектури моделі, на яку вона спрямована. Особливо вразливими виявилися лінійні моделі, такі як SVM Linear та Logistic Regression, що підтверджує їхню схильність до впливу змагальних атак через використання лінійних гіперплощин для прийняття рішень. Натомість моделі, засновані на деревах рішень, зокрема Decision Tree

та Random Forest, демонструють незначно вищу стійкість до атак, хоча їхні показники помилок залишаються на високому рівні.

Отримані результати вказують на високу ефективність змагальних атак, що підтверджується значним рівнем перенесення атаки між моделями. Це підкреслює необхідність подальших досліджень у сфері безпеки машинного навчання, зокрема розробки методів захисту, таких як використання додаткових механізмів регуляризації, adversarial training або алгоритмів виявлення атак. Також доцільним є дослідження впливу зміни максимального рівня збурення (dmax) на переносимість атаки та аналіз стійкості інших типів атак, таких як FGSM або DeepFool.

Крок 4. Аналіз точності цільових класифікаторів на змагальних прикладах

Результати експерименту демонструють значне зниження точності цільових класифікаторів на атакованих зразках, згенерованих для замінного класифікатора. Це вказує на високу вразливість моделей машинного навчання до атак із перенесенням (transfer attacks).

Для підтвердження цього ефекту необхідно виконати детальне порівняння продуктивності класифікаторів у стандартних умовах та в умовах атаки. Після цього слід провести аналіз змін у прийнятті рішень моделями та визначити закономірності у зміні їхньої продуктивності. Особливу увагу необхідно приділити тому, як відрізняється рівень стійкості різних архітектур класифікаторів до атакованих прикладів.

Отримані результати підтверджують необхідність розробки та впровадження додаткових механізмів захисту від атак, таких як навчання з урахуванням змагальних прикладів (adversarial training), використання методів регуляризації або виявлення атак на рівні вхідних даних.

Цей код створює візуалізацію рішень цільових класифікаторів під атакою. Основна мета – показати, як змагальні приклади (adversarial examples) змінюють передбачення різних моделей та оцінити ефективність перенесеної атаки.

1. Імпорт необхідних бібліотек

```
from secml.figure import CFigure
from secml.array import CArray
from math import ceil
```

CFigure – використовується для створення графічних візуалізацій.

CArray – працює з масивами чисел у форматі SecML.

ceil – математична функція, що округлює число вгору (використовується для розміщення підграфіків).

2. Створення фігури для графіків

```
fig = CFigure(width=4.5 * len(target_clf_list) / 2,
              height=4 * 2, markersize=10)
```

Створюється фігура (fig), де будуть розміщені всі підграфіки (по одному для кожного цільового класифікатора).

Ширина та висота визначаються автоматично на основі кількості класифікаторів (len(target_clf_list)).

markersize=10 встановлює розмір точок, які будуть використовуватись у візуалізації.

3. Цикл для побудови графіків кожного класифікатора

```
for clf_idx in range(len(target_clf_list)):
    clf = target_clf_list[clf_idx].clf
```

Цикл перебирає всі цільові класифікатори (target_clf_list).

clf містить поточний класифікатор, який аналізується.

4. Додавання підграфіка для кожної моделі

```
fig.subplot(2, int(ceil(len(target_clf_list) / 2)), clf_idx + 1)
fig.sp.title(target_clf_list[clf_idx].clf_name)
```

Формується підграфік (subplot), розміщений у сітці 2×N (де N – половина кількості класифікаторів, округлена вгору).

fig.sp.title(...) додає заголовок для кожного підграфіка, що містить назву моделі (clf_name).

5. Побудова областей рішень для класифікатора

```
fig.sp.plot_decision_regions(clf, n_grid_points=200)
fig.sp.grid(grid_on=False)
```

plot_decision_regions(clf, n_grid_points=200) – будує області класифікаційних рішень моделі.

Області показують, як модель розподіляє класи в просторі вхідних даних.

grid_on=False відключає сітку для більш чистого вигляду графіка.

6. Визначення індексів прикладів, які не належать до цільового класу

```
s_idx = ts.Y.find(ts.Y != y_target)
```

ts.Y.find(ts.Y != y_target) знаходить індекси всіх зразків тестового набору, які НЕ належать до y_target.

Це потрібно для перевірки, як атака змінила їхню класифікацію.

7. Відображення атакованих зразків на графіку

```
for pt in s_idx[:10]:  # Візуалізуємо атаку на 10 прикладах
    pt_segment = CArray.append(ts.X[pt, :], adv_ds.X[pt, :], axis=0)
    fig.sp.plot_path(pt_segment)
```

Перебираються перші 10 тестових зразків (s_idx[:10]), які були атаковані.

CArray.append(ts.X[pt, :], adv_ds.X[pt, :], axis=0) створює лінію, що з'єднує початкову точку (ts.X[pt, :]) і атаковану версію (adv_ds.X[pt, :]).

fig.sp.plot_path(pt_segment) малює ці зміщення на графіку, що дозволяє побачити, як атака змінила дані.

8. Оцінка успішності перенесеної атаки

```
acc = metric.performance_score(
    y_true=ts[s_idx[:10], :].Y, y_pred=clf.predict(adv_ds[s_idx[:10], :].X))
```

Передбачення (clf.predict(adv_ds[s_idx[:10], :].X)) виконується для атакованих зразків.

Порівнюється з правильними мітками (ts[s_idx[:10], :].Y), щоб оцінити, наскільки сильно атака вплинула на модель.

В результаті отримується точність атаки: наскільки добре атаковані приклади зберігають свою початкову мітку після атаки.

9. Відображення результатів атаки на графіку

```
fig.sp.text(0.01, 0.01, "Transfer attack success: {:.1%}".format(acc),
            bbox=dict(facecolor='white'))
```

Додається текстова позначка на кожен підграфік із рівнем успішності атаки.

Відображається відсоток успішних атак (наскільки часто модель була введена в оману).

bbox=dict(facecolor='white') додає білий фон, щоб текст був чіткішим.

10. Відображення фінального графіка

```
fig.show()
```

Всі створені підграфіки відображаються у фінальному графіку (рис. 2.5).

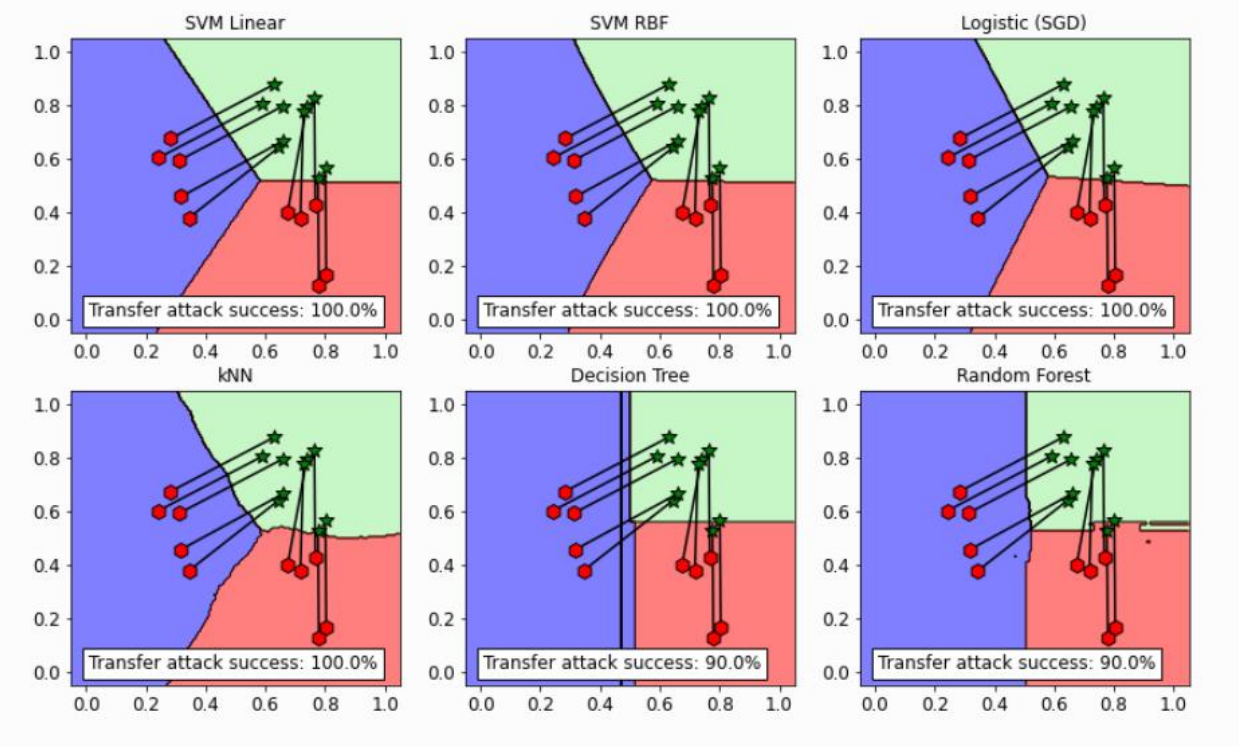


Рис. 2.5. Графіки для оцінки рівня успішності атак

2D-графіки (рис. 2.5) наочно демонструють вразливість цільових класифікаторів. Приклади, що викликають суперництво (зелені зірочки), які знаходяться всередині зеленої області прийняття рішень y_target = 2, успішно перенесені.

ЗАГАЛЬНЕ ЗАВДАННЯ ДЛЯ ВИКОНАННЯ

1. Підготовка даних. Використати штучно згенерований набір кластеризованих точок (CDLRandomBlobs).
2. Нормалізувати дані в діапазоні [0,1].
3. Поділити вибірку на тренувальну (70%) та тестову (30%).
4. Створити дві незалежні вибірки:
   a. Перша (tr1, n=1000) – для навчання сурогатного класифікатора.
   b. Друга (tr2, n=1000) – для навчання цільових класифікаторів.
5. Навчання класифікаторів. Навчити сурогатний класифікатор:
   a. Модель: CClassifierMulticlassOVA(CClassifierSVM, kernel='linear')
   b. Гіперпараметри: C = [1e-2, 0.1, 1]
   c. Виконати підбір параметрів через 3-Fold крос-валідацію.

6. Навчити цільові класифікатори:

a. SVM RBF: CClassifierMulticlassOVA(CClassifierSVM, kernel='rbf', gamma=10)

b. Logistic Regression: CClassifierMulticlassOVA(CClassifierSGD, loss='log')

c. kNN: CClassifierKNN(n_neighbors=40)

d. Decision Tree: CClassifierDecisionTree(max_depth=5)

e. Random Forest: CClassifierRandomForest(n_estimators=30)

7. Оцінити базову точність кожного класифікатора на тестовій вибірці (ts).

8. Створення атакованих прикладів. Використовуючи сурогатний класифікатор (SVM Linear), згенерувати змагальні приклади за допомогою CAttackEvasionPGDLS:

a. Тип атаки: L2-norm

b. Рівень збурення (dmax): 0.3

c. Цільова атака (y_target): Перекласифікація у клас 2

d. Оптимізаційні параметри: eta = 0.1, max_iter = 100, eps = 1e-4

e. Виконати оцінку точності сурогатного класифікатора на атакованих прикладах.

9. Аналіз перенесення атак. Протестувати всі цільові класифікатори на атакованих прикладах (adv_ds.X) та оцінити їхню точність.

a. Обчислити рівень перенесення атаки (transfer rate)

b. Візуалізувати рівень помилок усіх моделей у вигляді стовпчастої діаграми.

c. Побудувати теплокарту перенесення атак для всіх моделей.

10. Відповісти на контрольні запитання.

11.Підготувати звіт, який містить опис основних дій (зі скріншотами).

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке змагальні атаки (adversarial attacks) та яка їхня основна мета?

2. Які основні типи атак на системи машинного навчання існують?

3. У чому полягає різниця між атаками "білого ящика" (white-box) та "чорного ящика" (black-box)?

4. Як працює змагальне збурення (adversarial perturbation) і чому воно залишається непомітним для людини?

5. Які види змагальних атак існують у фізичному світі, і як вони можуть впливати на безпеку реальних систем?

6. Що таке універсальні змагальні атаки (universal adversarial attacks) і чим вони відрізняються від індивідуальних атак?

7. Які існують методи захисту нейронних мереж від змагальних атак, і які їхні основні принципи?

8. Як змагальні атаки можуть впливати на автономні транспортні засоби та системи біометричної ідентифікації?

9. У чому полягає небезпека атак від отруєння даних (data poisoning) для процесу навчання нейронних мереж?

10. Яким чином атаки на основі перенесення (transferability attacks) можуть бути ефективними навіть без доступу до цільової моделі?

ЛАБОРАТОРНА РОБОТА № 3.
ДОСЛІДЖЕННЯ АТАК ОТРУЄННЯ (POISONING ATTACKS) НА МОДЕЛІ МАШИННОГО НАВЧАННЯ

Мета роботи: Дослідити вплив атак отруєння на класифікатор SVM з RBF-ядром. Реалізувати атаку, що вносить спеціально створені зразки до навчальної вибірки, змінюючи функцію прийняття рішень моделі, що призводить до зниження її точності.

ВКАЗІВКИ З ПІДГОТОВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

Дана робота присвячене аналізу впливу атак отруєння на продуктивність класифікаторів машинного навчання, зокрема Support Vector Machine (SVM) з RBF-ядром. Основна мета полягає у виявленні вразливостей моделі до отруйних зразків, які спеціально додаються до навчальної вибірки з метою змінення функції прийняття рішень та зниження точності класифікації.

Для реалізації дослідження використовується штучно згенерований датасет, який нормалізується в діапазоні [0,1]. Після цього дані розподіляються на навчальну та тестову вибірки у співвідношенні 70% до 30%. Додатково виділяється валідаційна підвибірка, яка використовується під час атаки для оцінки її ефективності.

На першому етапі навчання моделі проводиться без впливу атак. Класифікатор SVM із RBF-ядром налаштовується та оптимізується за допомогою крос-валідації, де підбираються гіперпараметри, такі як коефіцієнт регуляризації C та параметр ядра gamma. Після навчання модель тестується на чистих даних, що дозволяє оцінити її базову продуктивність.

Наступним етапом є реалізація атаки отруєння, яка виконується шляхом генерації отруйних зразків за допомогою алгоритмів CAttackPoisoningSVM або CAttackPoisoningGradient. Атака моделюється шляхом внесення спеціально сформованих зразків у навчальну вибірку, що змінює межі прийняття рішень класифікатора. Кількість атакованих зразків варіюється у межах від 1% до 5% від загального обсягу навчальних даних, а рівень збурення оцінюється за L2-нормою. Оптимізація атаки здійснюється з використанням адаптивних параметрів, включаючи налаштування швидкості навчання eta, максимальну кількість ітерацій max_iter та поріг збіжності eps.

Після внесення отруйних зразків модель повторно навчається на зміненій вибірці.

Для оцінки ефективності атаки аналізується продуктивність моделі до та після внесення отруйних зразків. Точність класифікатора визначається на тестовій вибірці та порівнюється з початковими показниками. Вплив атаки досліджується шляхом побудови графіка залежності точності від кількості атакованих зразків. Додатково аналізується зміна меж прийняття рішень моделі у двовимірному просторі, що дозволяє візуально оцінити ефект отруєння даних.

Результати дослідження дозволяють визначити, наскільки ефективною є атака при різному обсязі отруйних зразків. Встановлюється критичний поріг, за якого точність класифікатора суттєво знижується. Крім того, здійснюється аналіз можливих методів захисту, таких як фільтрація вхідних даних, регуляризація або використання робастних алгоритмів.

Підсумковий аналіз дозволяє зробити висновки щодо ефективності атак отруєння та виробити рекомендації для покращення безпеки моделей машинного навчання.

Додаткові матеріали доступні за посиланням:

1. Biggio, B., Nelson, B. and Laskov, P., 2012. Poisoning attacks against support vector machines. In ICML 2012. URL: https://arxiv.org/abs/1206.6389

2. Xiao, H., Biggio, B., Brown, G., Fumera, G., Eckert, C. and Roli, F., 2015. Is feature selection secure against training data poisoning?. In ICML 2015. URL: https://arxiv.org/abs/1804.07933

3. Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C. and Roli, F., 2019. Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks. In 28th Usenix Security Symposium, Santa Clara, California, USA. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/demontis

ТЕОРЕТИЧНІ ВІДОМОСТІ

Розвиток штучного інтелекту супроводжується не лише прогресом у створенні автономних систем, але й виникненням нових типів кіберзагроз. Одним із таких загрозливих напрямів є атаки отруєння даних, які дозволяють зловмисникам маніпулювати моделями машинного навчання, змінюючи їхні рішення у бажаному напрямку. Ці атаки є особливо небезпечними у критичних сферах, таких як фінансові сервіси, автономний транспорт і

системи біометричної ідентифікації, оскільки здатні спотворювати результати прогнозів, компрометувати цілісність даних і створювати значні ризики безпеки.

Отруєння даних відбувається на етапі навчання моделей, коли до навчальної вибірки свідомо додаються спеціально створені зразки, що спотворюють функцію прийняття рішень. Ці маніпуляції можуть реалізовуватися шляхом внесення міткових аномалій, навмисного викривлення розподілу даних або формування особливих вразливостей, які залишаються невиявленими під час тестування моделі. Основними типами атак є отруєння міток (backdoor poisoning), отруєння навчальних даних, інверсійні атаки на модель та приховані атаки, що стають активними лише після розгортання системи в реальних умовах. Спільною метою таких атак є або порушення коректності рішень моделі, або отримання несанкціонованого доступу до конфіденційної інформації.

Одним із напрямів використання атак отруєння є маніпуляція штучним інтелектом, що генерує глибокі фейки. Глибокі фейки є синтетичними медіафайлами, зокрема зображеннями, відео або аудіозаписами, створеними за допомогою алгоритмів глибокого навчання. У разі отруєння таких моделей вони можуть виробляти викривлені або спотворені результати, які відповідають інтересам зловмисників. Це може бути використано для створення фальшивих інформаційних кампаній, дискредитації публічних осіб або обходу біометричних систем безпеки. Наприклад, атака на систему контролю доступу на основі розпізнавання обличчя може змусити її прийняти обличчя зловмисника як законного користувача.

Прикладом відомої атаки на штучний інтелект є випадок з чат-ботом Tay, розробленим компанією Microsoft у 2016 році. Ця модель мала навчатися через взаємодію з користувачами у Twitter, проте зловмисники навмисно подавали їй токсичний контент, що призвело до того, що бот почав генерувати агресивні та неприйнятні висловлювання. Це демонструє, що атаки можуть відбуватися навіть без фізичного втручання у навчальні дані, а лише через цілеспрямований вплив на процес навчання моделі.

Сучасні організації, які працюють зі штучним інтелектом, часто не розробляють власні моделі з нуля, а базуються на готових великих мовних моделях (LLMs), таких як ті, що постачають OpenAI або інші великі технологічні компанії. Використання зовнішніх моделей не гарантує захисту від атак отруєння, оскільки навіть непрямі маніпуляції із загальнодоступними джерелами навчання, наприклад внесення змін до статей у Wikipedia або

завантаження спотворених зображень у відкриті джерела, можуть призводити до зміщення висновків моделей у потрібному зловмисникам напрямку.

Оскільки атаки отруєння становлять значну загрозу для надійності систем штучного інтелекту, розробники та дослідники розробляють різні захисні механізми. Найефективнішими методами виявлення і запобігання таким атакам є очищення та попередня обробка даних, що передбачає фільтрацію аномальних зразків і перевірку достовірності джерел інформації. Використання алгоритмів виявлення аномалій дає змогу моніторити навчальні набори та ідентифікувати підозрілі патерни у вхідних даних. Одним із методів є навчання на змагальних прикладах, що дозволяє моделі адаптуватися до отруйних зразків і підвищувати свою стійкість до маніпуляцій.

Захист моделей може також включати вдосконалення архітектур із вбудованими механізмами протидії змагальним атакам, серед яких методи оптимізації, дистиляції та стиснення ознак. Контроль за поведінкою моделей у реальному часі та аналіз їхньої продуктивності також дозволяють виявляти нетипові закономірності, що можуть свідчити про атаку. Важливим кроком є валідація введених даних перед їхнім використанням, що може включати перевірку цифрових підписів, аналіз цілісності інформації та автентифікацію джерел.

Питання безпечного зберігання та обробки навчальних даних також залишається актуальним. Використання шифрування, механізмів контролю доступу та обмеження прав на зміну навчальних даних допомагають знизити ризик несанкціонованих втручань. Крім того, серед ключових заходів безпеки є суворий контроль процедур навчання, який передбачає роботу в захищеному середовищі, перевірку навчальних даних і використання безпечних протоколів обробки даних.

Зважаючи на зростання загроз, пов’язаних з атаками отруєння, компаніям, що працюють зі штучним інтелектом, необхідно враховувати ці ризики та впроваджувати ефективні стратегії захисту своїх моделей. У міру вдосконалення технологій розвиваються також і методи атак, тому організації повинні постійно оновлювати свої підходи до кібербезпеки, вивчати нові вектори атак і впроваджувати адаптивні механізми захисту. Безпека штучного інтелекту є динамічним процесом, який потребує постійного моніторингу та вдосконалення.

ПРИКЛАДИ ПРАКТИЧНИХ ЗАВДАНЬ

У цій роботі буде реалізовано змагальні атаки отруєння (poisoning attacks) проти Support Vector Machine (SVM) з RBF-ядром. Атака проводиться на етапі навчання моделі шляхом додавання спеціально сформованих зразків, що спотворюють функцію прийняття рішень, знижуючи загальну точність класифікатора.

На першому етапі буде створено та навчено класифікатор у стандартному режимі без впливу атак, після чого оцінено його продуктивність. Для коректного моделювання атаки необхідно розділити навчальний набір на дві частини: основну навчальну вибірку та валідаційний піднабір, який використовуватиметься для контролю продуктивності моделі під час атаки.

Завданням експерименту є визначення ефективності атак отруєння, оцінка впливу атакованих зразків на продуктивність моделі та аналіз можливих механізмів захисту.

Крок 1. У браузері перейдіть по https://www.kaggle.com/

Якщо ви бачите синю кнопку "Увійти" у верхньому правому куті, натисніть на неї та увійдіть в обліковий запис Kaggle. У меню виберіть "Файл", "Нова записна книжка".

Крок 2. Встановлення SecML

Виконайте наступні команди:

```
!pip install secml
import secml
```

Бібліотека встановиться, як показано нижче (рис. 3.1).

Крок 3. Підготовка набору даних, створення та навчання моделі

Наведений код генерує, обробляє та використовує дані для навчання SVM-класифікатора з RBF-ядром у середовищі SecML. Основні етапи включають створення вибірки, її розділення на підмножини, нормалізацію, навчання моделі та тестування.

1. Генерація даних

```
random_state = 999

n_features = 2  # Кількість ознак
```

```
n_samples = 300  # Загальна кількість зразків
centers = [[-1, -1], [+1, +1]]  # Центри кластерів
cluster_std = 0.9  # Стандартне відхилення кластерів
```

random_state = 999 — використовується для забезпечення відтворюваності експерименту.

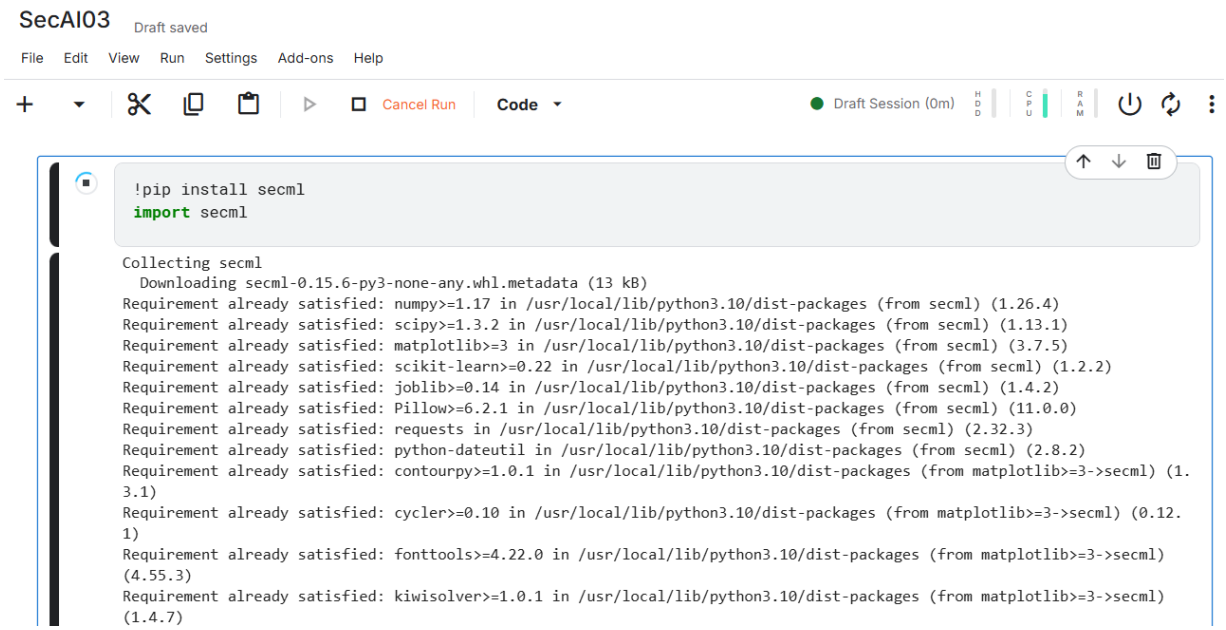


Рис. 3.1. Встановлення SecML

n_features = 2 – генерується двовимірний набір даних (2D-простір).

n_samples = 300 – всього 300 точок у вибірці.

centers = [[-1, -1], [+1, +1]] – кластери центровані навколо двох точок у просторі.

cluster_std = 0.9 – задає розкид точок навколо центрів кластерів.

```
from secml.data.loader import CDLRandomBlobs
dataset = CDLRandomBlobs(n_features=n_features,
                         centers=centers,
                         cluster_std=cluster_std,
                         n_samples=n_samples,
                         random_state=random_state).load()
```

Генерується датасет із випадково розподіленими точками, згрупованими навколо двох центрів.

2. Розділення даних на підмножини

```
n_tr = 100  # Кількість зразків у навчальній вибірці
```

```
n_val = 100  # Кількість зразків у валідаційній вибірці
n_ts = 100  # Кількість зразків у тестовій вибірці
```

n_tr визначає розмір тренувального набору (100 точок).

n_val відповідає за валідаційний набір (100 точок).

n_ts задає розмір тестової вибірки (100 точок).

```
from secml.data.splitter import CTrainTestSplit
splitter = CTrainTestSplit(
    train_size=n_tr + n_val, test_size=n_ts, random_state=random_state)
tr_val, ts = splitter.split(dataset)
```

Розбиваємо набір на дві частини:

tr_val (200 точок) → буде поділений на навчальний (tr) та валідаційний (val) набори.

ts (100 точок) → тестовий набір.

```
splitter = CTrainTestSplit(
    train_size=n_tr, test_size=n_val, random_state=random_state)
tr, val = splitter.split(dataset)
```

Додатково розбивається tr_val на tr (100 точок) і val (100 точок).

Тепер маємо три незалежні підвибірки:

tr – навчальна вибірка для тренування моделі.

val – валідаційна вибірка для перевірки продуктивності під час налаштування.

ts – тестова вибірка для остаточної оцінки.

3. Нормалізація даних

```
from secml.ml.features import CNormalizerMinMax
nmz = CNormalizerMinMax()
tr.X = nmz.fit_transform(tr.X)
val.X = nmz.transform(val.X)
ts.X = nmz.transform(ts.X)
```

Використовується мін-макс нормалізація, щоб привести всі значення до діапазону [0,1].

fit_transform(tr.X) – навчає нормалізатор на тренувальній вибірці та застосовує нормалізацію.

transform(val.X) та transform(ts.X) – застосовує ту ж саму нормалізацію до валідаційних і тестових даних.

4. Визначення метрики продуктивності

```
from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()
```

Обирається метрика оцінки точності (CMetricAccuracy), яка вимірює частку правильно класифікованих зразків.

5. Створення та навчання SVM-класифікатора

```
from secml.ml.classifiers import CClassifierSVM
from secml.ml.kernels import CKernelRBF
clf = CClassifierSVM(kernel=CKernelRBF(gamma=10), C=1)
```

Створюється класифікатор SVM (CClassifierSVM) із радіальною базисною функцією (RBF kernel).

Параметри:

gamma = 10 – впливає на форму RBF-ядра.

C = 1 – параметр регуляризації, що контролює баланс між складністю моделі та точністю.

```
clf.fit(tr.X, tr.Y)
print("Training of classifier complete!")
```

Навчання SVM-класифікатора (clf.fit) відбувається на тренувальних даних tr.X і tr.Y.

Після успішного навчання виводиться повідомлення "Training of classifier complete!". (рис. 3.2)

```
2025-02-06 19:01:47,792 - secml.settings - INFO - New `SECML_HOME_DIR` created: /root/secml-data
2025-02-06 19:01:47,792 - secml.settings - INFO - New `SECML_HOME_DIR` created: /root/secml-data
2025-02-06 19:01:47,794 - secml.settings - INFO - Default configuration file copied to: /root/secml-data/secml.conf
2025-02-06 19:01:47,794 - secml.settings - INFO - Default configuration file copied to: /root/secml-data/secml.conf
2025-02-06 19:01:47,798 - secml.settings - INFO - New `SECML_DS_DIR` created: /root/secml-data/datasets
2025-02-06 19:01:47,798 - secml.settings - INFO - New `SECML_DS_DIR` created: /root/secml-data/datasets
2025-02-06 19:01:47,801 - secml.settings - INFO - New `SECML_MODELS_DIR` created: /root/secml-data/models
2025-02-06 19:01:47,801 - secml.settings - INFO - New `SECML_MODELS_DIR` created: /root/secml-data/models
2025-02-06 19:01:47,806 - secml.settings - INFO - New `SECML_EXP_DIR` created: /root/secml-data/experiments
2025-02-06 19:01:47,806 - secml.settings - INFO - New `SECML_EXP_DIR` created: /root/secml-data/experiments
2025-02-06 19:01:47,810 - secml.settings - INFO - New `SECML_LOGS_DIR` created: /root/secml-data/logs
2025-02-06 19:01:47,810 - secml.settings - INFO - New `SECML_LOGS_DIR` created: /root/secml-data/logs
2025-02-06 19:01:47,814 - secml.settings - INFO - New `SECML_PYTORCH_DIR` created: /root/secml-data/pytorch-data
2025-02-06 19:01:47,814 - secml.settings - INFO - New `SECML_PYTORCH_DIR` created: /root/secml-data/pytorch-data
Training of classifier complete!
```

Рис. 3.2. Створення та навчання моделі

6. Виконання тестування на нових даних

```
y_pred = clf.predict(ts.X)
```

Класифікатор (clf) робить передбачення (predict) на тестовій вибірці ts.X.

y_pred містить передбачені мітки для тестових зразків.

Крок 2. Атака отруєння на SVM-класифікатор

Обчислення отруєного зразка у контексті атак отруєння передбачає розв'язання бі-рівневої оптимізаційної задачі, де цільовою функцією є максимізація помилки класифікатора на валідаційному наборі, а нижній рівень оптимізації відповідає процесу навчання моделі на отруєних даних.

$$\max_{x_c} A(D_{val}, \mathbf{w}^*) = \sum_{j=1}^{m} \ell(y_j, \mathbf{x}_j, \mathbf{w}^*) \tag{3.1}$$

за умови:

$$\mathbf{w}^* \in \underset{\mathbf{w}}{\operatorname{argmin}} L(D_{tr} \cup (\mathbf{x}_c, y_c), \mathbf{w}) \tag{3.2}$$

де $\mathbf{x_c}$ є отруєним зразком, A – функція цілі атакуючого, а L – функція навчання класифікатора. Набір D_{tr} відповідає тренувальним даним, а D_{val} – валідаційним даним, які використовуються для оцінки впливу атаки.

Ця оптимізаційна задача визначає **шкідливий зразок** $\mathbf{x}_c$, який додається до тренувального набору, спричиняючи зміну параметрів моделі $\mathbf{w}^*$, що, у свою чергу, впливає на точність класифікації на валідаційних даних. Основна складність цієї атаки полягає у тому, що параметри класифікатора залежать від отруєного зразка, що створює **циклічну залежність між** $\mathbf{x}_c$ **та** $\mathbf{w}^*$.

Формально, оптимізаційна задача визначається наступним чином:

$$\max_{x_c} A(D_{val}, w^*) = \sum_{j=1}^{m} l(y_j, x_j, w^*) \tag{3.3}$$

за умови:

$$w^* \in arg\ \min_w L(D_{tr} \cup (x_c, y_c), w) \tag{3.4}$$

де x_c – є отруєним зразком,

A – функція цілі атакуючого,

L – функція навчання класифікатора.

D_{tr} – набір тренувальних даних,

D_{val} – набір валідаційних даних, які використовуються для оцінки впливу атаки.

Ця оптимізаційна задача визначає шкідливий зразок x_c, який додається до тренувального набору, спричиняючи зміну параметрів моделі w^*, що, у свою чергу, впливає на точність класифікації на валідаційних даних. Основна складність цієї атаки полягає у тому, що параметри класифікатора залежать від отруєного зразка, що створює циклічну залежність між x_c та w^*.

Для реалізації такої атаки у середовищі SecML використовується клас CAttackPoisoningSVM, що є підкласом загального класу атак CAttackPoisoning. Цей підхід дозволяє виконати целенаправлене отруєння моделі, навчаючи її на спеціально сформованих зразках, які знижують точність класифікації.

Перед запуском атаки необхідно визначити параметри атаки, зокрема межі простору атак (обмежені простором ознак у валідаційних даних) та параметри оптимізатора для ефективного розв'язання задачі.

Наступним етапом є візуалізація функції цілі атакуючого, що дозволяє оцінити вплив отруєного зразка на продуктивність моделі та обрати оптимальну точку для атаки.

Наведений нижче код реалізує атаку отруєння (poisoning attack) на SVM-класифікатор із RBF-ядром за допомогою бібліотеки SecML. Основна ідея атаки полягає у додаванні шкідливих навчальних зразків до вибірки, що змушує модель приймати неправильні рішення. Цей підхід використовується для вивчення стійкості алгоритмів машинного навчання до атак.

1. Визначення меж простору атаки

```
lb, ub = val.X.min(), val.X.max()  # Межі простору атаки
```

lb (lower bound) – мінімальне значення серед усіх ознак у валідаційній вибірці.

ub (upper bound) – максимальне значення серед усіх ознак у валідаційній вибірці.

Це визначає обмеження на змінні (простір пошуку) під час оптимізації атаки.

Якщо замість lb, ub встановити None, атака може впливати на будь-які значення вхідних даних (без обмежень).

2. Визначення параметрів оптимізації

```
solver_params = {
    'eta': 0.05,       # Початковий розмір кроку градієнтного спуску
    'eta_min': 0.05,   # Мінімальне значення кроку
    'eta_max': None,   # Відсутність обмеження на максимальний крок
```

```
    'max_iter': 100,   # Максимальна кількість ітерацій
    'eps': 1e-6        # Порогове значення зупинки алгоритму
}
```

eta – розмір початкового кроку градієнтного спуску.

eta_min – мінімальний можливий крок (щоб уникнути занадто малих змін).

eta_max – відсутність обмеження на максимальний крок (алгоритм сам підбирає).

max_iter = 100 – атака виконується максимум за 100 ітерацій.

eps = 1e-6 – якщо зміни стали меншими за цей поріг, оптимізація припиняється.

3. Ініціалізація атаки отруєння

```
from secml.adv.attacks import CAttackPoisoningSVM
pois_attack = CAttackPoisoningSVM(classifier=clf,
                                  training_data=tr,
                                  val=val,
                                  lb=lb, ub=ub,
                                  solver_params=solver_params,
                                  random_seed=random_state)
```

CAttackPoisoningSVM – реалізація атаки отруєння для SVM-класифікатора.

Параметри:

classifier=clf – атакований класифікатор.

training_data=tr – навчальні дані, на які буде впливати атака.

val=val – валідаційний набір, що використовується для перевірки атаки.

lb, ub – обмеження простору атаки.

solver_params – параметри оптимізації (визначені раніше).

random_seed=random_state – фіксований стан генератора випадкових чисел для відтворюваності експерименту.

4. Вибір початкового отруєного зразка

```
xc = tr[0,:].X  # Початкові ознаки отруєного зразка
yc = tr[0,:].Y  # Початкова мітка класу
pois_attack.x0 = xc
pois_attack.xc = xc
pois_attack.yc = yc
```

xc = tr[0,:].X – вибирається перший зразок навчальної вибірки як початкова точка атаки.

yc = tr[0,:].Y – отримується його правильна мітка класу.

pois_attack.x0 = xc – встановлюється початковий зразок, що буде змінюватися атакою.

pois_attack.xc = xc – копія початкового значення (використовується в процесі оптимізації).

pois_attack.yc = yc – мітка класу для цього зразка.

```
print("Initial poisoning sample features: {:}".format(xc.ravel()))
print("Initial poisoning sample label: {:}".format(yc.item()))
```

Виводиться початковий стан отруєного зразка (його координати та клас).

5. Візуалізація атаки

```
from secml.figure import CFigure
%matplotlib inline  # Використовується лише в Jupyter Notebook
```

CFigure – клас для візуалізації.

%matplotlib inline – потрібний для відображення графіків у Jupyter Notebook.

```
fig = CFigure(4,5)
```

Створюється фігура розміром 4×5 дюймів для побудови графіків.

```
grid_limits = [(lb - 0.1, ub + 0.1), (lb - 0.1, ub + 0.1)]
```

Визначаються межі сітки для відображення простору атаки (додається невеликий запас 0.1).

```
fig.sp.plot_ds(tr)
```

Відображається початкова навчальна вибірка (tr).

```
fig.sp.plot_ds(tr[0,:], markers='*', markersize=16)
```

Підсвічується отруєний зразок (зірочка * великим розміром 16).

```
fig.sp.title('Attacker objective and gradients')
fig.sp.plot_fun(
```

```
    func=pois_attack.objective_function,
    grid_limits=grid_limits, plot_levels=False,
    n_grid_points=10, colorbar=True)
```

Візуалізується функція цілі атакуючого (objective function).

Кольорова карта показує області, де атака найефективніша.

6. Побудова обмежень простору атаки

```
from secml.optim.constraints import CConstraintBox
box = CConstraintBox(lb=lb, ub=ub)
fig.sp.plot_constraint(box, grid_limits=grid_limits, n_grid_points=10)
```

Створюється обмеження простору атаки (CConstraintBox) відповідно до lb, ub.

Відображається область, у межах якої атака може змінювати зразки (рис. 3.3).

```
fig.tight_layout()
fig.show()
```

tight_layout() – автоматично налаштовує компонування графіка для кращого вигляду.

fig.show() – відображає фінальний графік.

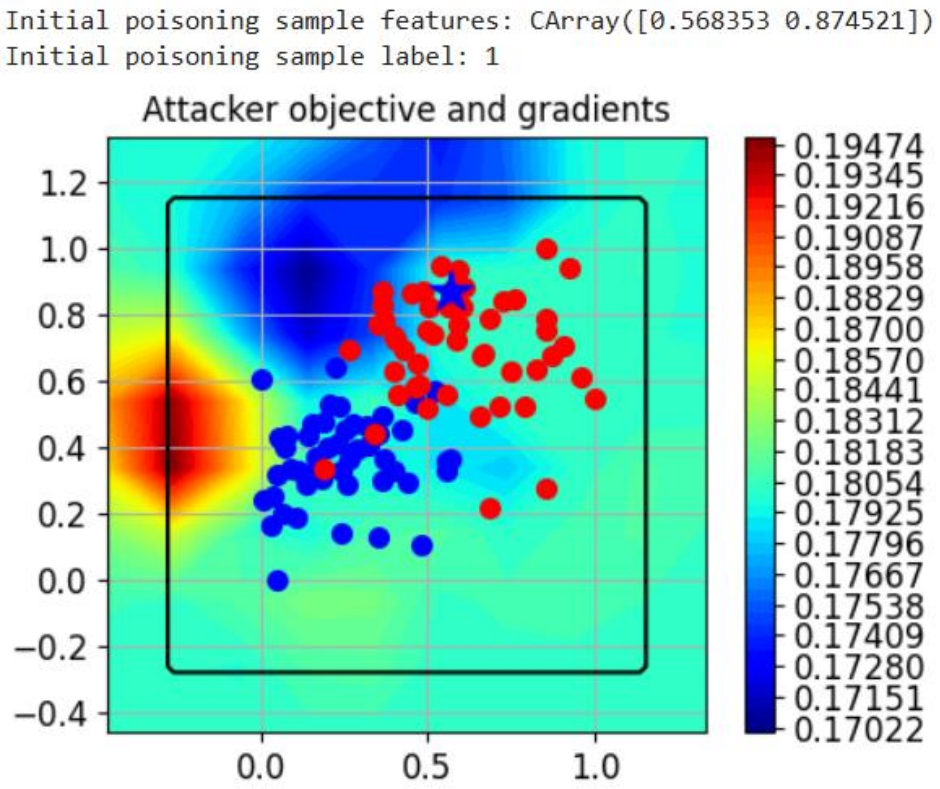


Рис. 3.3. Фінальний графік

Графік на рис. 3.3 відображає функцію цілі атакуючого та її градієнти у контексті атаки отруєння на SVM-класифікатор. Сині та червоні точки представляють тренувальні зразки двох класів, де сині точки належать до класу 0, а червоні – до класу 1. Чорний прямокутник позначає область обмежень, у межах яких атакуючий може змінювати отруєний зразок. Фонове забарвлення ілюструє рівні значень функції цілі атакуючого, де червоні області вказують на місця, де атака матиме найбільший ефект, а сині області демонструють зони з мінімальним впливом.

Верхній підпис містить координати початкового отруєного зразка, що становлять CArray ([0.568353, 0.874521]). Також зазначено, що початковий зразок належить до класу 1. Цей зразок буде змінюватися в процесі атаки, щоб максимізувати його вплив на навчання класифікатора. Кольорова шкала праворуч відображає рівні значень функції цілі атакуючого, де максимальне значення (0.19474) вказує на найбільш ефективні області атаки, а мінімальне (0.17022) відповідає менш ефективним зонам.

Графік ілюструє, як розташування отруєного зразка впливає на продуктивність класифікатора. Червоні області позначають зони, у які необхідно змістити зразок для підсилення атаки. Якщо атакуючий зможе перемістити точку ближче до області з максимальним значенням функції цілі, точність класифікатора суттєво знизиться. Це свідчить про те, що SVM-класифікатор може стати значно менш надійним, якщо під час навчання до тренувальних даних будуть додані такі отруєні зразки. Графік демонструє механізм атаки отруєння та дозволяє дослідити, як маніпуляції навчальними даними впливають на кінцеві рішення класифікатора.

Крок 3. Генерація змагальних точок

На цьому етапі необхідно визначити бажану кількість змагальних точок, які будуть використані для атаки. У цій роботі встановлюється значення 20, що означає, що до тренувальної вибірки буде додано 20 отруєних зразків. Ці зразки будуть оптимізовані таким чином, щоб максимізувати вплив на класифікатор та знизити його точність.

Процес генерації змагальних точок передбачає їхнє початкове визначення у допустимих межах простору ознак, після чого застосовується метод оптимізації, який коригує їхні значення для досягнення бажаного ефекту атаки. Оптимізовані точки додаються до навчального набору, що впливає на рішення моделі під час тренування. Наступним кроком є запуск процесу оптимізації та аналіз отриманих результатів.

1. Визначення кількості отруєних зразків

```
n_poisoning_points = 20  # Кількість отруєних точок
pois_attack.n_points = n_poisoning_points
```

Змінна n_poisoning_points = 20 задає кількість зразків, які будуть згенеровані для атаки.

pois_attack.n_points = n_poisoning_points встановлює це значення в об'єкті атаки pois_attack, що означає, що алгоритм буде оптимізувати 20 отруєних точок.

2. Запуск атаки отруєння

```
print("Attack started...")
pois_y_pred, pois_scores, pois_ds, f_opt = pois_attack.run(ts.X, ts.Y)
print("Attack complete!")
```

pois_attack.run(ts.X, ts.Y) виконує атаку отруєння.

Вхідні параметри:

ts.X – тестові зразки (вхідні дані для класифікації).

ts.Y – їхні справжні мітки класів.

Результати атаки:

pois_y_pred – передбачені класи після атаки.

pois_scores – оцінки впевненості класифікатора після атаки.

pois_ds – отруєний набір даних після атаки.

f_opt – значення оптимізованої функції цілі атакуючого.

Вивід у консоль: вказує на початок та завершення атаки, що дозволяє відстежувати її виконання.

3. Оцінка точності класифікатора до та після атаки

```
# Оцінка точності класифікатора без атаки
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

# Оцінка точності класифікатора після атаки
pois_acc = metric.performance_score(y_true=ts.Y, y_pred=pois_y_pred)
```

metric.performance_score(y_true=ts.Y, y_pred=y_pred) обчислює точність моделі до атаки.

metric.performance_score(y_true=ts.Y, y_pred=pois_y_pred) обчислює точність після атаки.

Обидва показники використовують тестовий набір ts.Y, що дозволяє оцінити зміну продуктивності моделі після внесення отруєних точок.

4. Виведення результатів

```
print("Original accuracy on test set: {:.2%}".format(acc))
print("Accuracy after attack on test set: {:.2%}".format(pois_acc))
```

Вивід у консоль дозволяє порівняти початкову точність та точність після атаки.

Якщо атака успішна, значення pois_acc буде значно нижчим, ніж acc, що означає, що додавання отруєних точок змінило функцію прийняття рішень класифікатора та знизило його ефективність (рис. 3.4).

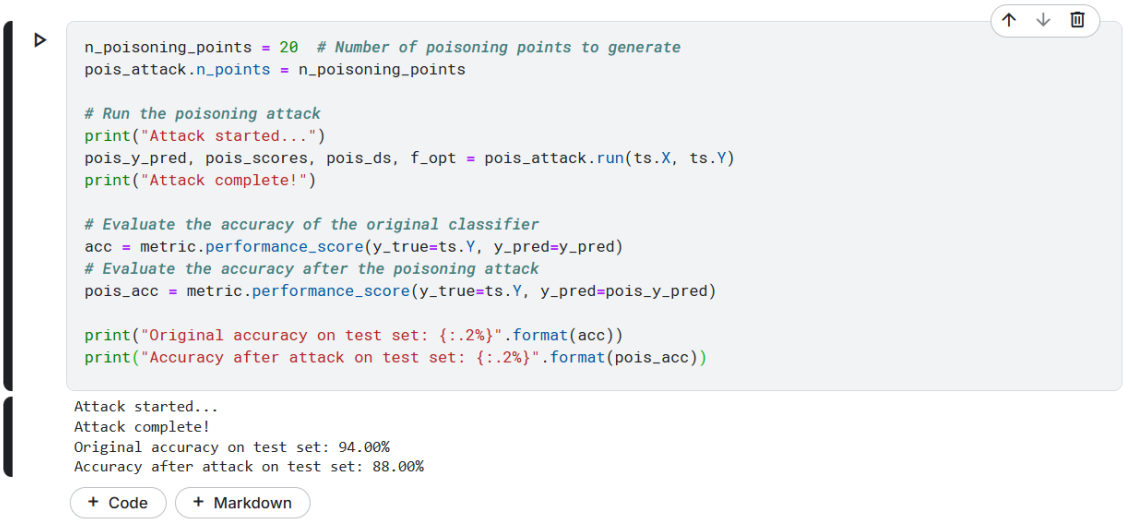


Рис. 3.4. Оцінка точності атаки

Крок 4. Візуалізації атаки отруєння та підсилення її впливу

Атака отруєння на класифікатор була успішно проведена, що призвело до зниження його точності. Для підсилення впливу атаки можливо збільшити кількість отруєних точок, однак це значно ускладнить процес оптимізації та підвищить його обчислювальну складність.

На наступному етапі необхідно візуалізувати результати атаки у двовимірному просторі. Для цього слід створити копію початкового класифікатора та навчити його на новому наборі даних, який містить як вихідні тренувальні зразки, так і додані отруєні точки. Це дозволить оцінити, як змінилися межі прийняття рішень після впливу атакованих зразків.

Після навчання моделі необхідно побудувати графік, що відображає розподіл вихідних тренувальних зразків, атакованих точок та кордони класифікації. Такий аналіз дозволить зрозуміти, наскільки сильно атака вплинула на розподіл класів і наскільки вразливим є класифікатор до отруєних зразків.

1. Створення та навчання атакованого класифікатора

```
# Training of the poisoned classifier
pois_clf = clf.deepcopy()
pois_tr = tr.append(pois_ds)  # Join the training set with the poisoning points
pois_clf.fit(pois_tr.X, pois_tr.Y)
```

pois_clf = clf.deepcopy() створює копію початкового класифікатора, щоб уникнути змін в оригінальній моделі.

pois_tr = tr.append(pois_ds) об'єднує початковий тренувальний набір (tr) з отруєними зразками (pois_ds).

pois_clf.fit(pois_tr.X, pois_tr.Y) навчає отруєний класифікатор на новому наборі даних.

Таким чином, новий класифікатор враховує отруєні зразки, що можуть суттєво змінити межі прийняття рішень.

2. Визначення меж сітки для візуалізації

```
min_limit = min(pois_tr.X.min(), ts.X.min())
max_limit = max(pois_tr.X.max(), ts.X.max())
grid_limits = [[min_limit, max_limit], [min_limit, max_limit]]
```

Обчислюються глобальні мінімальні та максимальні значення координат усіх точок (початкових, тестових та отруєних).

grid_limits визначає межі для сітки візуалізації, щоб усі графіки мали однаковий масштаб.

3. Створення графічної фігури

```
fig = CFigure(10, 10)
```

CFigure(10, 10) створює полотно розміром 10x10 дюймів для розміщення чотирьох графіків.

4. Візуалізація початкового та атакованого класифікатора

4.1. Початковий класифікатор на тренувальних даних

```
fig.subplot(2, 2, 1)
fig.sp.title("Original classifier (training set)")
fig.sp.plot_decision_regions(
    clf, n_grid_points=200, grid_limits=grid_limits)
fig.sp.plot_ds(tr, markersize=5)
fig.sp.grid(grid_on=False)
```

Перший підграфік (верхній лівий кут) відображає рішення початкового класифікатора на тренувальному наборі.

plot_decision_regions(clf, n_grid_points=200, grid_limits=grid_limits) будує кордони рішень класифікатора.

plot_ds(tr, markersize=5) відображає тренувальні точки.

4.2. Отруєний класифікатор на тренувальних даних

```
fig.subplot(2, 2, 2)
fig.sp.title("Poisoned classifier (training set + poisoning points)")
fig.sp.plot_decision_regions(
   pois_clf, n_grid_points=200, grid_limits=grid_limits)
fig.sp.plot_ds(tr, markersize=5)
fig.sp.plot_ds(pois_ds, markers=['*', '*'], markersize=12)
fig.sp.grid(grid_on=False)
```

Другий підграфік (верхній правий кут) відображає кордони рішень класифікатора після отруєння.

plot_ds(pois_ds, markers=['*', '*'], markersize=12) додає отруєні точки (pois_ds), які позначаються зірочками.

Якщо атака успішна, кордони прийняття рішень зміняться у порівнянні з початковим класифікатором.

4.3. Початковий класифікатор на тестових даних

```
fig.subplot(2, 2, 3)
fig.sp.title("Original classifier (test set)")
fig.sp.plot_decision_regions(
   clf, n_grid_points=200, grid_limits=grid_limits)
fig.sp.plot_ds(ts, markersize=5)
fig.sp.text(0.05, -0.25, "Accuracy on test set: {:.2%}".format(acc),
         bbox=dict(facecolor='white'))
fig.sp.grid(grid_on=False)
```

Третій підграфік (нижній лівий кут) демонструє, як початковий класифікатор працює на тестовому наборі.

plot_ds(ts, markersize=5) відображає тестові точки.

fig.sp.text(0.05, -0.25, "Accuracy on test set: {:.2%}".format(acc)) показує точність класифікатора до атаки.

4.4. Отруєний класифікатор на тестових даних

```
fig.subplot(2, 2, 4)
fig.sp.title("Poisoned classifier (test set)")
fig.sp.plot_decision_regions(
```

```
        pois_clf, n_grid_points=200, grid_limits=grid_limits)
    fig.sp.plot_ds(ts, markersize=5)
    fig.sp.text(0.05, -0.25, "Accuracy on test set: {:.2%}".format(pois_acc),
            bbox=dict(facecolor='white'))
    fig.sp.grid(grid_on=False)
```

Четвертий підграфік (нижній правий кут) демонструє, як атакований класифікатор працює на тестових даних.

Якщо атака була успішною, кордони класифікації зміняться, і точність тестового набору pois_acc буде нижчою за acc.

fig.sp.text(..., "Accuracy on test set: {:.2%}".format(pois_acc)) показує знижену точність після атаки.

5. Відображення фінального графіка

```
    fig.show()
```

Відображає всі чотири підграфіки для аналізу впливу атаки отруєння (рис. 3.5).

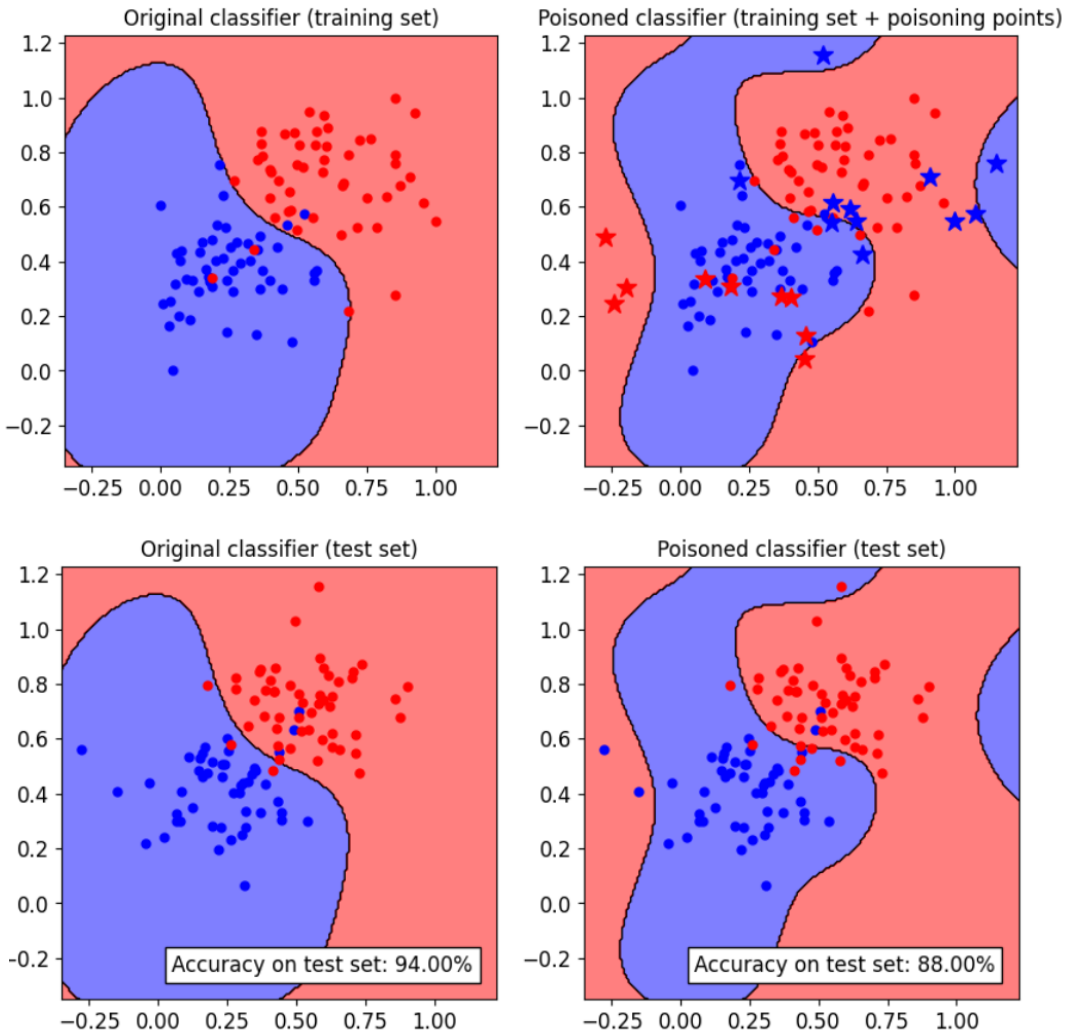


Рис. 3.5. Візуалізація результатів атаки отруєння

На рис. 3.5 можна побачити, як змінюються функції прийняття рішень класифікатором SVM після введення точок отруєння (сині та червоні зірочки).

ЗАГАЛЬНЕ ЗАВДАННЯ ДЛЯ ВИКОНАННЯ

1. Дослідити, як атаки отруєння (poisoning attacks) впливають на роботу KNN-класифікатора, оцінити зміну меж класифікації та точності моделі до і після атаки.
2. Створити навчальний набір даних із двома класами, використовуючи CDLRandomBlobs.
3. Розділити дані на тренувальний, валідаційний та тестовий набори.
4. Навчити KNN-класифікатор (CClassifierKNN) без впливу атак та оцінити його точність.
5. Згенерувати 20 отруєних точок та додати їх до тренувального набору.
6. Повторно навчити класифікатор на отруєному наборі та оцінити зміну точності.
7. Побудувати графіки рішень класифікатора до та після атаки для тренувального і тестового наборів.
8. Проаналізувати результати та зробити висновки про вразливість KNN до атак отруєння.
9. Надайте відповіді на контрольні запитання
10. Підготуйте звіт, який містить опис основних дій (зі скріншотами).

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке атака отруєння (poisoning attack) у машинному навчанні?
2. Яким чином отруєні зразки впливають на тренувальний процес класифікатора?
3. Чому SVM-класифікатори можуть бути вразливими до атак отруєння?
4. Яка основна відмінність між отруєнням міток (label poisoning) та отруєнням навчальних даних (feature poisoning)?
5. Яким чином оптимізується отруєний зразок у бі-рівневій оптимізаційній задачі?

6. Які параметри визначають ефективність атаки отруєння?

7. Як можна візуально оцінити вплив отруєних зразків на межі прийняття рішень класифікатора?

8. Чому KNN-класифікатор може бути більш стійким або менш стійким до атак отруєння порівняно з SVM?

9. Які заходи можна застосувати для підвищення стійкості класифікаторів до атак отруєння?

10. Яким чином валідаційний набір використовується при реалізації атаки отруєння?

ЛАБОРАТОРНА РОБОТА № 4.
ДОСЛІДЖЕННЯ ВИКОНАННЯ АТАК УХИЛЕННЯ ТА ОТРУЄННЯ НА НАБОРІ ДАНИХ MNIST

Мета роботи: ознайомлення з набором даних MNIST, та використання його для навчання SVM-класифікатора, а також проведення атаки ухилення (Evasion Attack) та отруєння (Poisoning Attack) на навчену модель

ВКАЗІВКИ З ПІДГОТОВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

У цій роботі буде розглянуто процес завантаження набору даних MNIST, його використання для навчання SVM-класифікатора, а також проведення атак ухилення (Evasion Attack) та отруєння (Poisoning Attack) на навчену модель.

Першим етапом є завантаження набору даних MNIST, що містить рукописні цифри у вигляді зображень розміром 28×28 пікселів. Дані розподіляються на навчальну та тестову вибірки, після чого виконується нормалізація значень пікселів у діапазон [0,1], що забезпечує коректну обробку класифікатором.

Після підготовки даних відбувається навчання Support Vector Machine (SVM) з RBF-ядром. Класифікатор навчається на навчальному наборі MNIST, а після тренування оцінюється його точність на тестових зразках. Це дозволяє визначити початкову продуктивність моделі перед проведенням атак.

Наступним кроком є атака ухилення, яка передбачає створення змагальних зразків (Adversarial Examples) за допомогою градієнтного методу ухилення. Для реалізації атаки використовується CAttackEvasionPGDLS, що дозволяє змусити класифікатор робити помилкові передбачення. Після атаки проводиться повторна оцінка точності моделі, що дозволяє порівняти результати до та після впливу атакованих зразків.

Далі виконується атака отруєння, у межах якої генеруються отруєні зразки та додаються до навчального набору. Модель повторно навчається на модифікованому наборі, а потім оцінюється вплив отруєних зразків на точність передбачень. Це дозволяє зрозуміти, наскільки сильно атака змінила поведінку класифікатора.

Завершальним етапом є візуалізація та аналіз отриманих результатів. Будуються графіки, що демонструють правильні та атаковані зразки, а також аналізується зміна меж прийняття рішень після проведення атак. На основі цих спостережень формулюються висновки щодо стійкості SVM-класифікатора до атак ухилення та отруєння.

Додаткову інформацію при підготовці до роботи можна отримати:

1. https://www.geeksforgeeks.org/brute-force-attack-in-metasploit/.

2. https://www.geeksforgeeks.org/how-to-prevent-brute-force-attacks/?ref=ml_lbp.

3. https://www.geesforgeeks.org/how-to-detect-brute-force-attacks/?ref=ml_lbp.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Штучний інтелект і машинне навчання відіграють ключову роль у сучасних технологіях обробки даних. Одним із популярних підходів до розпізнавання образів є використання методу опорних векторів (Support Vector Machine, SVM), який забезпечує ефективну класифікацію даних. У цьому розділі розглядається процес навчання SVM-класифікатора на наборі даних MNIST, а також аналізуються методи атак на модель, зокрема атаки ухилення (Evasion Attacks) та атаки отруєння (Poisoning Attacks).

Набір даних MNIST. Набір даних MNIST (Modified National Institute of Standards and Technology) містить зображення рукописних цифр від 0 до 9 розміром 28×28 пікселів у градаціях сірого. MNIST широко використовується у дослідженнях машинного навчання та комп’ютерного зору як стандартний датасет для оцінки продуктивності алгоритмів класифікації.

Процес роботи з набором даних передбачає його завантаження, попередню обробку та розбиття на тренувальну (наприклад, 60 000 зразків) та тестову вибірки (10 000 зразків). Після цього дані використовуються для навчання моделі SVM, яка навчається розпізнавати зображення цифр.

Метод опорних векторів (SVM) для класифікації зображень. Метод опорних векторів є одним із найпоширеніших алгоритмів класифікації, що використовується для розпізнавання рукописних символів та інших задач комп’ютерного зору. SVM базується на побудові гіперплощини у багатовимірному просторі, яка розділяє класи таким чином, щоб максимізувати відстань між найближчими зразками класів.

При застосуванні до MNIST-класифікації найкращі результати зазвичай досягаються за допомогою радіального базисного ядра (RBF kernel), що дозволяє враховувати нелінійні закономірності у вхідних даних. Основні етапи роботи включають навчання SVM на тренувальній вибірці, налаштування гіперпараметрів (наприклад, коефіцієнта регуляризації C і параметра ядра γ) та тестування продуктивності на окремій вибірці зображень.

Атаки на моделі машинного навчання. Незважаючи на ефективність методів машинного навчання, вони можуть бути вразливими до змагальних атак (adversarial attacks). У цьому дослідженні розглядаються два основних типи атак: атаки ухилення (Evasion Attacks) та атаки отруєння (Poisoning Attacks).

Атаки ухилення (Evasion Attacks). Атаки ухилення здійснюються після навчання моделі і спрямовані на зміну вхідних даних так, щоб змусити класифікатор зробити помилковий висновок. Для цього до вхідного зображення додається малий шум, непомітний для людини, який змінює передбачення моделі. Одним із методів реалізації таких атак є градієнтний метод PGD-LS (Projected Gradient Descent – Line Search), який використовується для створення змагальних прикладів (adversarial examples). Такі атаки демонструють, що навіть незначні зміни у зображенні можуть суттєво впливати на роботу класифікатора.

Атаки отруєння (Poisoning Attacks). Атаки отруєння впливають на класифікатор на етапі його навчання. Вони передбачають додавання спеціально сформованих шкідливих зразків до тренувального набору, що змінює функцію прийняття рішень моделі. У цьому випадку атакуючий має контроль над частиною тренувальних даних і маніпулює ними так, щоб класифікатор навчився робити помилкові передбачення. Такі атаки особливо небезпечні у випадках, коли модель навчається на неконтрольованих або відкритих джерелах даних.

Значення дослідження атак на класифікатори. Дослідження атак ухилення та отруєння є критично важливим для безпеки штучного інтелекту. Знання про вразливості моделей дозволяє розробляти ефективні методи захисту, такі як робастне навчання, виявлення аномалій та фільтрація вхідних даних. Аналіз атак на SVM-класифікатор у наборі MNIST дозволяє краще зрозуміти, як моделі машинного навчання реагують на навмисні маніпуляції, що є ключовим для розробки стійких алгоритмів у критично важливих

додатках, таких як фінансові системи, біометрична ідентифікація та автономний транспорт.

У цій роботі розглядається процес навчання SVM-класифікатора для розпізнавання цифр у наборі MNIST, а також аналізуються методи атак на модель. Вразливість моделей машинного навчання до атак ухилення та отруєння підкреслює важливість розробки захисних механізмів, що підвищують їхню безпеку та надійність.

ПРИКЛАДИ ПРАКТИЧНИХ ЗАВДАНЬ

Крок 1. У браузері перейдіть по https://www.kaggle.com/

Якщо ви бачите синю кнопку "Увійти" у верхньому правому куті, натисніть на неї та увійдіть в обліковий запис Kaggle. У меню виберіть "Файл", "Нова записна книжка".

Крок 2. Встановлення SecML

Виконайте наступні команди:

```
!pip install secml
import secml
```

Бібліотека встановиться, як показано нижче (рис. 4.1).

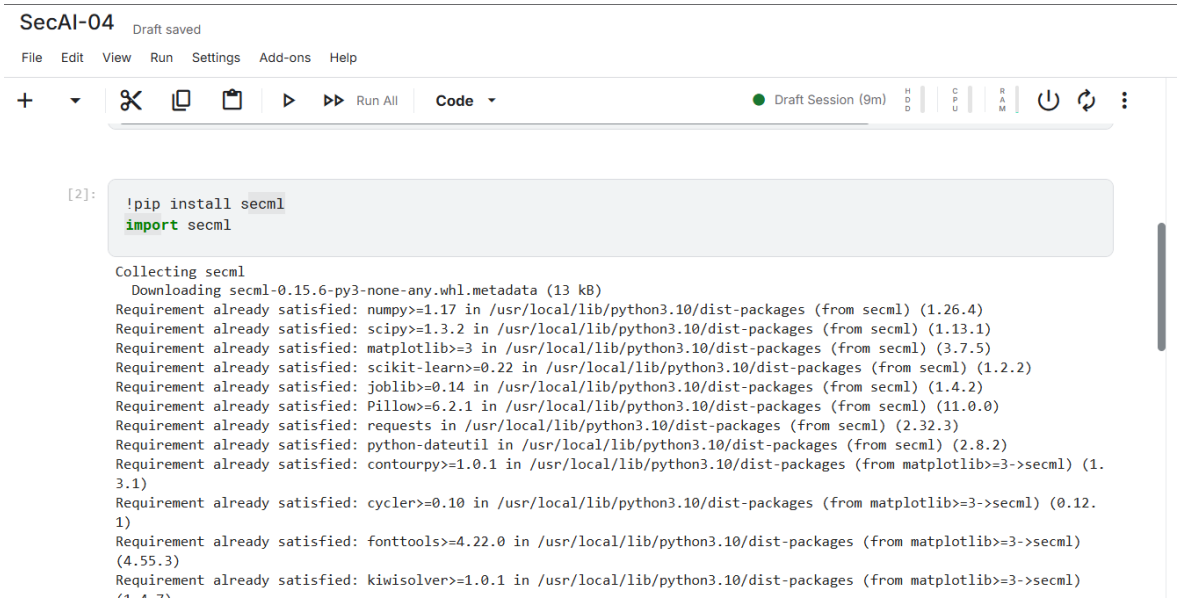


Рис. 4.1. Встановлення SecML

Крок 3. Завантаження набору даних та підготовка до навчання

На першому етапі необхідно завантажити набір даних MNIST, який містить рукописні цифри (рис. 4.2). Для цього використовуємо стандартний метод імпорту та вибираємо лише дві цифри – 5 (п’ять) і 9 (дев’ять). Відбір обмежено двома класами, щоб спростити навчання моделі та сконцентрувати увагу на точності розпізнавання.

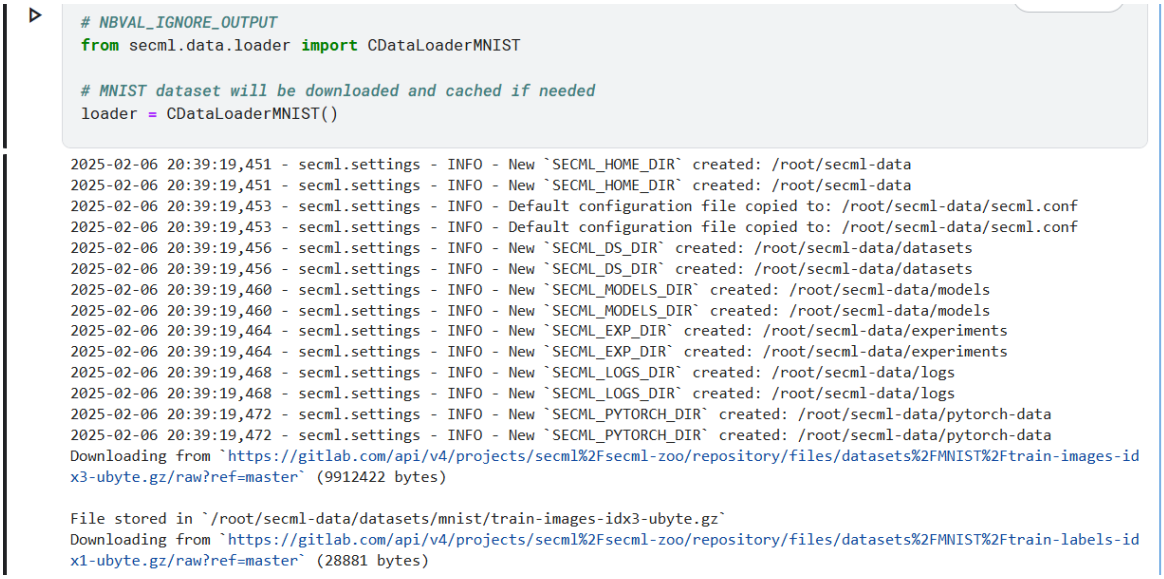


Рис. 4.2. Завантаження набору даних

Крок 4. Навчання SVM-класифікатора для розпізнавання цифр 5 і 9

Даний код виконує завантаження, обробку та навчання SVM-класифікатора на підмножині набору даних MNIST, що містить зображення рукописних цифр 5 і 9. Основна мета – побудувати модель, яка ефективно відрізняє ці два класи, та оцінити її продуктивність.

1. Налаштування параметрів

```
random_state = 999
n_tr = 100  # Кількість зразків у навчальній вибірці
n_val = 500  # Кількість зразків у валідаційній вибірці
n_ts = 500  # Кількість зразків у тестовій вибірці
```

Визначається початковий стан генератора випадкових чисел (random_state = 999) для відтворюваності результатів.

Кількість зразків у вибірках:

n_tr = 100 – 100 зразків для тренування.

n_val = 500 – 500 зразків для валідації.

n_ts = 500 – 500 зразків для тестування.

2. Завантаження вибраних цифр (5 і 9)

```
digits = (5, 9)

tr_val = loader.load('training', digits=digits, num_samples=n_tr + n_val)
ts = loader.load('testing', digits=digits, num_samples=n_ts)
```

digits = (5, 9) обмежує набір даних лише цифрами 5 та 9.

loader.load('training', digits=digits, num_samples=n_tr + n_val) завантажує 600 зразків (100 для тренування та 500 для валідації).

loader.load('testing', digits=digits, num_samples=n_ts) завантажує 500 тестових зразків.

3. Розбиття даних на тренувальну та валідаційну вибірки

```
tr = tr_val[:n_tr, :]
val = tr_val[n_tr:, :]
```

Розділяє tr_val на дві частини:

tr містить 100 зразків для тренування.

val містить 500 зразків для валідації.

4. Нормалізація зображень у діапазон [0, 1]

```
tr.X /= 255
val.X /= 255
ts.X /= 255
```

Значення пікселів з діапазону [0, 255] приводяться до [0, 1] шляхом поділу на 255.

Ця операція покращує стабільність роботи SVM, оскільки зменшує вплив великомасштабних значень ознак.

5. Створення та навчання SVM-класифікатора

```
from secml.ml.classifiers import CClassifierSVM
# train SVM in the dual space, on a linear kernel, as needed for poisoning
clf = CClassifierSVM(C=10, kernel='linear')

print("Training of classifier...")
clf.fit(tr.X, tr.Y)
```

Використовується SVM з лінійним ядром (kernel='linear'), що оптимально підходить для проблеми двокласової класифікації.

Параметр C=10 визначає ступінь регуляризації, де вищі значення C можуть зменшити помилки на тренувальних даних, але підвищити ризик переобучення.

Модель навчається на tr.X та tr.Y.

6. Передбачення міток на тестовому наборі

```
y_pred = clf.predict(ts.X)
```

Класифікатор clf передбачає мітки y_pred для тестового набору ts.X.

7. Оцінка продуктивності моделі

```
from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()

# Оцінка точності класифікатора
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

print("Accuracy on test set: {:.2%}".format(acc))
```

CMetricAccuracy() обирається як метрика продуктивності.

Обчислюється точність моделі (acc), яка визначає частку правильно класифікованих тестових зразків.

Вивід точності у відсотках ({:.2%} форматування).

Результат навчання моделі класифікатора представлено на рис. 4.3.

```
print("Training of classifier...")
clf.fit(tr.X, tr.Y)

# Compute predictions on a test set
y_pred = clf.predict(ts.X)

# Metric to use for performance evaluation
from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()

# Evaluate the accuracy of the classifier
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

print("Accuracy on test set: {:.2%}".format(acc))
```

```
Training of classifier...
Accuracy on test set: 93.60%
```

Рис. 4.3. Навчання класифікатора

Крок 5. Атака ухилення (Evasion Attack) на наборі даних MNIST

Налаштування параметрів атаки. Для виконання атаки ухилення необхідно визначити основні параметри. У даному випадку використовується L2-пертурбація (додавання змагального шуму) з обмеженням у максимальну сферу радіусом $\varepsilon = 2.5$ навколо початкових точок. Це означає, що внесені зміни будуть мінімальними, але достатніми для зміни передбачень класифікатора.

Оскільки простір ознак MNIST обмежений у діапазоні [0,1], встановлюється додаткове обмеження на можливі значення пікселів, щоб уникнути виходу за цей діапазон.

Також атака проводиться у режимі "error-generic", тобто вона не націлена на конкретний клас, а спрямована на створення змагальних прикладів, які змушують класифікатор робити будь-які неправильні передбачення. Для цього параметр y_target встановлюється у None.

Тривалість виконання атаки. Залежно від продуктивності комп'ютера, атака ухилення може займати декілька хвилин. Це пояснюється тим, що кожен змагальний приклад генерується за допомогою оптимізаційного процесу, що вимагає обчислення градієнтів моделі для знаходження оптимальних змін у вхідному зображенні.

Подальші кроки. Після визначення параметрів атаки наступним етапом буде запуск атаки ухилення, тестування отриманих змагальних прикладів та оцінка впливу атаки на продуктивність класифікатора.

Наведений нижче код реалізує атаку ухилення (Evasion Attack) на SVM-класифікатор, що був навчений на підмножині MNIST (цифри 5 і 9). Атака здійснюється за допомогою градієнтного методу PGD-LS (Projected Gradient Descent - Line Search), який додає L2-пертурбацію до вхідних зображень, щоб змусити класифікатор робити помилкові передбачення.

1. Вибір підмножини тестових даних для атаки

```
# Для простоти атака виконується на підмножині тестового набору
attack_ds = ts[:25, :]
```

Оберігається лише перших 25 зразків із тестової вибірки (ts) для атаки.

Це дозволяє зменшити обчислювальні витрати, оскільки атака на весь тестовий набір може зайняти багато часу.

2. Визначення параметрів атаки

```
noise_type = 'l2'  # Тип пертурбації ('l1' або 'l2')
```

```
dmax = 2.5  # Максимальна величина пертурбації
lb, ub = 0., 1.  # Межі значень ознак (0-1 для нормалізованих зображень)
y_target = None  # None для 'error-generic' атаки
```

Використовується L2-пертурбація, тобто зміни вхідних пікселів обмежуються сферою радіусу 2.5 у просторі ознак.

Обмеження lb=0. та ub=1. гарантують, що пікселі залишаються у допустимому діапазоні для зображень MNIST.

y_target = None означає, що атака не спрямована на конкретний клас (error-generic), а лише змушує модель видавати будь-які неправильні передбачення.

3. Налаштування параметрів оптимізації

```
solver_params = {
    'eta': 0.5,   # Початковий розмір кроку градієнтного спуску
    'eta_min': 2.0,  # Мінімальний розмір кроку
    'eta_max': None,  # Відсутність обмеження на максимальний крок
    'max_iter': 100,  # Максимальна кількість ітерацій
    'eps': 1e-6  # Порогове значення для зупинки оптимізації
}
```

eta = 0.5 визначає початковий розмір кроку оптимізації під час знаходження змагальних прикладів.

eta_min = 2.0 – встановлює мінімальне значення кроку, щоб уникнути занадто малих змін.

max_iter = 100 – атака виконується максимум за 100 ітерацій градієнтного спуску.

eps = 1e-6 – якщо різниця між поточним і попереднім значенням функції втрат стає меншою за 1e-6, атака зупиняється.

4. Ініціалізація атаки ухилення (Evasion Attack)

```
from secml.adv.attacks import CAttackEvasionPGDLS
pgd_ls_attack = CAttackEvasionPGDLS(classifier=clf,
                                    double_init_ds=tr,
                                    distance=noise_type,
                                    dmax=dmax,
                                    solver_params=solver_params,
                                    y_target=y_target)
```

CAttackEvasionPGDLS реалізує атаку ухилення за методом градієнтного пошуку PGD-LS.

classifier=clf задає класифікатор, на який здійснюється атака (SVM).

double_init_ds=tr – використання початкового набору даних для додаткової ініціалізації.

distance=noise_type вказує, що використовується L2-норма для обмеження змін у зображеннях.

dmax=dmax визначає максимальну величину пертурбації (ε = 2.5).

solver_params містить налаштування оптимізатора.

y_target = None означає неспрямовану атаку, яка змушує модель робити будь-які помилки.

5. Запуск атаки

```
print("Attack started...")
eva_y_pred, _, eva_adv_ds, _ = pgd_ls_attack.run(attack_ds.X, attack_ds.Y)
print("Attack complete!")
```

Запускається атака ухилення (pgd_ls_attack.run) на тестових зразках (attack_ds.X).

eva_y_pred – передбачені мітки після атаки.

eva_adv_ds – змагальні зразки (перетворені вхідні зображення).

Після завершення атаки виводиться повідомлення "Attack complete!".

6. Оцінка ефективності атаки

python

Копировать

Редактировать

```
acc = metric.performance_score(
   y_true=attack_ds.Y, y_pred=clf.predict(attack_ds.X))
acc_attack = metric.performance_score(
   y_true=attack_ds.Y, y_pred=eva_y_pred)
```

acc – точність класифікатора перед атакою (на оригінальних тестових зразках).

acc_attack – точність класифікатора після атаки (на змінених змагальних прикладах).

Якщо атака успішна, то acc_attack буде значно нижчою за acc, що означає, що класифікатор зробив більше помилок на атакованих зразках.

7. Вивід результатів

```
print("Accuracy on reduced test set before attack: {:.2%}".format(acc))
print("Accuracy on reduced test set after attack: {:.2%}".format(acc_attack))
```

Виводиться точність моделі до та після атаки у відсотках.

Якщо атака ефективна, точність після атаки суттєво знижується, що підтверджує вразливість класифікатора до змагальних прикладів (рис. 4.3)

```
print("Attack started...")
eva_y_pred, _, eva_adv_ds, _ = pgd_ls_attack.run(attack_ds.X, attack_ds.Y)
print("Attack complete!")

acc = metric.performance_score(
    y_true=attack_ds.Y, y_pred=clf.predict(attack_ds.X))
acc_attack = metric.performance_score(
    y_true=attack_ds.Y, y_pred=eva_y_pred)

print("Accuracy on reduced test set before attack: {:.2%}".format(acc))
print("Accuracy on reduced test set after attack: {:.2%}".format(acc_attack))
```

```
Attack started...
Attack complete!
Accuracy on reduced test set before attack: 100.00%
Accuracy on reduced test set after attack: 12.00%
```

Рис. 4.3. Оцінка результатів атаки ухилення

Крок 6. Аналіз результатів атаки ухилення та візуалізація змагальних прикладів

Після проведення атаки ухилення ми можемо спостерігати, що класифікатор, навчений на наборі даних MNIST, був успішно обманутий змагальними прикладами, які були згенеровані атакою. Це означає, що модель почала робити неправильні передбачення, незважаючи на те, що зміни у зображеннях були мінімальними та майже непомітними для людини.

Наступним етапом є візуалізація отриманих змагальних прикладів. Для цього будується графічне представлення, у якому:

Перший рядок містить вихідні зразки з тестового набору, на яких класифікатор давав правильні або початкові передбачення.

Другий рядок містить змагальні зразки, отримані після атаки, які призвели до помилкових передбачень моделі.

```
from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline
```

```
# Let's define a convenience function to easily plot the MNIST dataset
def show_digits(samples, preds, labels, digs, n_display=8):
    samples = samples.atleast_2d()
    n_display = min(n_display, samples.shape[0])
    fig = CFigure(width=n_display*2, height=3)
    for idx in range(n_display):
        fig.subplot(2, n_display, idx+1)
        fig.sp.xticks([])
        fig.sp.yticks([])
        fig.sp.imshow(samples[idx, :].reshape((28, 28)), cmap='gray')
        fig.sp.title("{} ({})".format(digits[labels[idx].item()],
digs[preds[idx].item()]),
                     color=("green" if labels[idx].item()==preds[idx].item() else
"red"))
    fig.show()

show_digits(attack_ds.X, clf.predict(attack_ds.X), attack_ds.Y, digits)
show_digits(eva_adv_ds.X, clf.predict(eva_adv_ds.X), eva_adv_ds.Y,
digits)
```

Над кожним зображенням буде вказано справжню мітку класу, а також передбачену класифікатором мітку у дужках. Це дозволить оцінити, наскільки серйозно атака змінила результат розпізнавання, навіть якщо зміни у зображеннях майже не помітні людському оку (рис. 4.5).

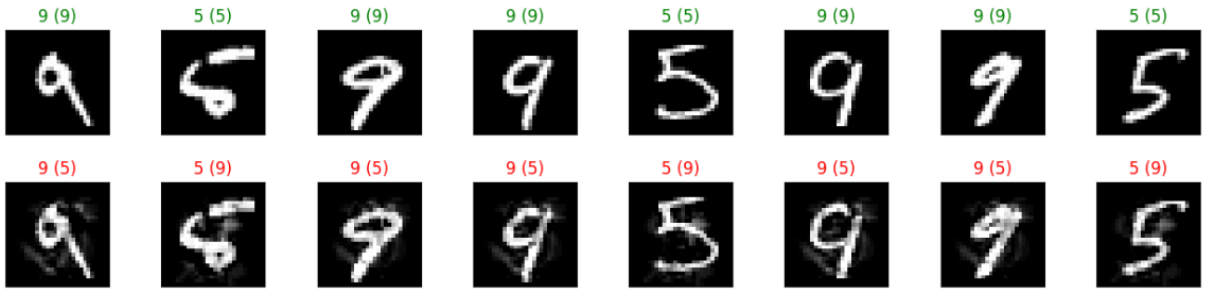


Рис.4.5. Візуалізація змагальних прикладів

Після цього можна проаналізувати отримані результати та зробити висновки щодо стійкості моделі до атак ухилення.

Крок 7. Атака отруєння (Poisoning Attack) на наборі даних MNIST

Для проведення атаки отруєння необхідно визначити основні параметри. На відміну від атаки ухилення, у цьому випадку параметри значно простіші.

Перше, що потрібно налаштувати – це межі простору атаки. Так як набір даних MNIST має нормалізовані значення пікселів у діапазоні [0,1], то встановлюється відповідне обмеження, що гарантує, що змінені зразки залишатимуться у допустимому інтервалі.

Друге – це кількість отруєних зразків. У цьому прикладі генерується 50 змагальних точок, які будуть додані до навчального набору для впливу на навчання моделі. Додавання таких зразків призводить до того, що класифікатор навчається на спотворених даних, що може суттєво знизити його точність і здатність правильно класифікувати зображення.

Останній етап – налаштування параметрів оптимізації. Підбираються параметри розв'язувача для вирішення даної оптимізаційної задачі, що дозволяє знайти найбільш ефективні змагальні точки для отруєння моделі.

Тривалість виконання атаки. Оскільки атака отруєння потребує багато ітерацій для модифікації навчальних даних, вона може зайняти кілька хвилин залежно від продуктивності комп'ютера. Важливо мати достатньо обчислювальних ресурсів для виконання цієї атаки.

Після налаштування параметрів атаки буде виконано генерацію отруєних зразків, повторне навчання моделі та оцінку її точності. Аналіз отриманих результатів дозволить визначити, наскільки вразливий класифікатор до атаки отруєння.

Наведений нижче програмний код реалізує атаку отруєння (Poisoning Attack) на SVM-класифікатор, що був навчений на підмножині MNIST (цифри 5 і 9). У процесі атаки генеруються отруєні зразки, які додаються до навчального набору, що змушує класифікатор навчатися на некоректних даних. Це може суттєво знизити точність моделі.

1. Визначення меж простору атаки

```
lb, ub = 0., 1.  # Межі простору атаки. Можна встановити `None` для необмеженого простору
```

Значення lb = 0. та ub = 1. обмежують можливі зміни у вхідних ознаках (пікселях).

Оскільки MNIST має нормалізовані значення пікселів у діапазоні [0,1], отруєні зразки не виходитимуть за межі допустимого простору.

Якщо None, то атакуючий міг би змінювати пікселі без обмежень, що не відповідає реальним сценаріям.

2. Визначення кількості отруєних зразків

```
n_poisoning_points = 15  # Кількість отруєних точок для генерації
```

Вибрано 15 отруєних зразків, які будуть додані до тренувального набору.

Чим більше отруєних точок, тим сильніше зміниться навчена модель і тим більший вплив на її продуктивність.

3. Налаштування параметрів оптимізації

```
solver_params = {
    'eta': 0.25,
    'eta_min': 2.0,
    'eta_max': None,
    'max_iter': 100,
    'eps': 1e-6
}
```

eta = 0.25 – початковий розмір кроку для оптимізації отруєних точок.

eta_min = 2.0 – мінімальний крок зміни отруєного зразка.

eta_max = None – немає обмеження на максимальний крок.

max_iter = 100 – алгоритм виконає максимум 100 ітерацій градієнтного пошуку отруєних зразків.

eps = 1e-6 – порогове значення зупинки оптимізації, якщо зміни у функції втрат стають меншими за це значення.

4. Ініціалізація атаки отруєння

```
from secml.adv.attacks import CAttackPoisoningSVM
pois_attack = CAttackPoisoningSVM(classifier=clf,
                                  training_data=tr,
                                  val=val,
                                  lb=lb, ub=ub,
                                  solver_params=solver_params,
                                  random_seed=random_state)
pois_attack.n_points = n_poisoning_points
```

CAttackPoisoningSVM реалізує атаку отруєння на SVM-класифікатор.

classifier=clf – цільова модель, на яку здійснюється атака.

training_data=tr – початковий навчальний набір, який буде змінюватися.

val=val – валідаційний набір, який використовується для оцінки впливу отруєних зразків.

lb, ub – обмеження для отруєних зразків (діапазон [0,1]).

solver_params – параметри оптимізатора.

random_seed=random_state – задає початковий стан генератора випадкових чисел для повторюваності результатів.

pois_attack.n_points = n_poisoning_points – встановлює 15 точок отруєння, які будуть згенеровані.

5. Запуск атаки отруєння

```
print("Attack started...")
pois_y_pred, _, pois_points_ds, _ = pois_attack.run(ts.X, ts.Y)
print("Attack complete!")
```

Запускається атака отруєння (pois_attack.run) на тестових зразках (ts.X).

pois_y_pred – передбачені мітки після атаки.

pois_points_ds – генеровані отруєні точки, які будуть додані до навчального набору.

Після завершення атаки виводиться повідомлення "Attack complete!".

6. Оцінка точності класифікатора до та після атаки

```
# Оцінка точності класифікатора без атаки
acc = metric.performance_score(y_true=ts.Y, y_pred=clf.predict(ts.X))
# Оцінка точності після атаки
pois_acc = metric.performance_score(y_true=ts.Y, y_pred=pois_y_pred)
```

acc – точність SVM-класифікатора перед атакою (на чистих тестових зразках).

pois_acc – точність після атаки (на тих самих тестових зразках, але після отруєння моделі).

Якщо атака ефективна, pois_acc буде значно нижчою за acc, що підтвердить, що класифікатор почав робити більше помилок.

7. Вивід результатів

```
print("Original accuracy on test set: {:.2%}".format(acc))
print("Accuracy after attack on test set: {:.2%}".format(pois_acc))
```

Виводиться точність до та після атаки, що дозволяє оцінити, наскільки атака змінила продуктивність моделі.

8. Навчання класифікатора після атаки

```
# Копіюємо вихідний класифікатор перед навчанням на отруєному наборі
pois_clf = clf.deepcopy()
pois_tr = tr.append(pois_points_ds)  # Додаємо отруєні точки до навчального набору
pois_clf.fit(pois_tr.X, pois_tr.Y)
```

Створюється копія SVM-класифікатора (pois_clf), щоб не змінювати оригінальну модель.

Додаються отруєні зразки (pois_points_ds) до навчального набору (tr).

Знову навчається SVM (pois_clf.fit(...)) на модифікованих даних.

9. Візуалізація отруєних зразків

```
show_digits(pois_points_ds.X, pois_clf.predict(pois_points_ds.X),
            pois_points_ds.Y, digits)
```

Відображаються отруєні зразки, які були додані до навчального набору.

Показується передбачений клас (після атаки) та справжній клас, що дозволяє оцінити, наскільки вони відрізняються від вихідних зображень (рис. 4.6).

```
print("Original accuracy on test set: {:.2%}".format(acc))
print("Accuracy after attack on test set: {:.2%}".format(pois_acc))

# Training of the poisoned classifier for visualization purposes
pois_clf = clf.deepcopy()
pois_tr = tr.append(pois_points_ds)  # Join the training set with the poisoning points
pois_clf.fit(pois_tr.X, pois_tr.Y)

show_digits(pois_points_ds.X, pois_clf.predict(pois_points_ds.X),
            pois_points_ds.Y, digits)
```

```
Attack started...
Attack complete!
Original accuracy on test set: 93.60%
Accuracy after attack on test set: 50.40%
```

Рис. 4.6. Результати атаки отруєння

Проведене дослідження атаки отруєння на класифікатор, навчений на наборі даних MNIST, показало, що модель була успішно атакована. Внесення отруєних зразків у тренувальний набір призвело до суттєвого зниження точності класифікації. Це підтверджує вразливість SVM-класифікаторів до атак, які впливають на навчальні дані.

Збільшення кількості отруєних точок може посилити вплив атаки, однак це значно ускладнює процес оптимізації та потребує більше обчислювальних ресурсів. Оптимальне співвідношення між кількістю атакованих зразків та ефективністю атаки потребує подальшого аналізу.

Також слід зазначити, що мітки змагальних зразків було навмисно змінено атакою відносно їхніх справжніх значень. Це означає, що при навчанні на отруєному наборі класифікатор адаптується до хибних даних, що призводить до помилкових передбачень. На візуалізації отриманих результатів можна побачити, що зеленим кольором позначені передбачені мітки, які відрізняються від істинних класів цифр, що ще раз підтверджує ефективність атаки.

Результати експерименту демонструють, що без захисту від атак отруєння класифікатори можуть бути значно скомпрометовані, що особливо критично для безпеки автоматизованих систем розпізнавання, фінансових сервісів, біометричної ідентифікації та інших важливих застосувань. Це підкреслює необхідність розробки та впровадження ефективних методів захисту, таких як виявлення аномальних зразків, перевірка цілісності навчальних даних та стійкі методи навчання.

ЗАГАЛЬНЕ ЗАВДАННЯ ДЛЯ ВИКОНАННЯ

Завдання 1: Навчання KNN-класифікатора на підмножині MNIST (цифри 3 та 6)

1. Завантажити набір даних MNIST, відібравши лише зображення цифр 3 та 6.

2. Розділити вибрані дані на тренувальний (100 зразків), валідаційний (500 зразків) та тестовий набір (500 зразків).

3. Провести нормалізацію значень пікселів у діапазоні [0,1].

4. Навчити класифікатор K-Nearest Neighbors (KNN) на отриманому тренувальному наборі.

5. Оцінити точність класифікації на тестовому наборі та зробити висновки про ефективність моделі.

Завдання 2: Атака ухилення (Evasion Attack) на KNN-класифікатор

1. Виконати атаку ухилення на 25 зразках тестового набору за допомогою L2-пертурбації (радіус $\varepsilon = 2.5$).

2. Визначити параметри атаки, такі як межі змін у просторі ознак [0,1].

3. Провести оптимізацію змагальних зразків для введення класифікатора в оману.

4. Оцінити точність моделі до та після атаки та порівняти результати.

5. Побудувати графік змагальних прикладів, де перший рядок – оригінальні зразки, другий – атаковані.

Завдання 3: Атака отруєння (Poisoning Attack) на KNN-класифікатор

1. Згенерувати 15 отруєних точок та додати їх до тренувального набору.

2. Перенавчити класифікатор KNN на змінених навчальних даних.

3. Перевірити точність моделі до та після атаки.

4. Побудувати візуалізацію змінених зразків та їх впливу на межі прийняття рішень.

5. Зробити висновки про вразливість KNN до атак отруєння та порівняти результати з атакою ухилення.

Завдання 4. Надати відповіді на контрольні запитання.

Завдання 5. Сформувати звіт в який додати:

1. Графічне представлення правильних та атакованих зразків для обох типів атак.

2. Порівняння точності KNN-класифікатора до та після атак.

3. Висновки щодо стійкості KNN до атак у порівнянні з SVM.

КОНТРОЛЬНІ ПИТАННЯ

1. Яка головна відмінність між KNN та SVM у задачі класифікації?

2. Чому для нормалізації пікселів MNIST використовується діапазон [0,1]?

3. Яка основна ідея атаки ухилення (Evasion Attack) у контексті KNN-класифікатора?

4. Як визначається величина L2-пертурбації у атаці ухилення?

5. Що означає поняття "змагальний зразок" (adversarial example)?

6. Які особливості KNN можуть впливати на його стійкість до атак ухилення?

7. Як атака отруєння змінює навчальні дані та впливає на роботу KNN?

8. Чим атака отруєння відрізняється від атаки ухилення?

9. Як впливає збільшення кількості отруєних точок на продуктивність класифікатора?

10. Які методи захисту можна застосувати для підвищення стійкості KNN до атак?

ЛАБОРАТОРНА РОБОТА № 5.
ДОСЛІДЖЕННЯ АТАКИ УХИЛЕННЯ (EVASION ATTACKS) НА НЕЙРОННІ МЕРЕЖІ З ВИКОРИСТАННЯМ НАБОРУ ДАНИХ MNIST

Мета роботи: ознайомитися з методами атак ухилення (Evasion Attacks) на нейронні мережі, використовуючи набір даних MNIST. Навчити нейронну мережу для класифікації рукописних цифр та проаналізувати її вразливість до змагальних прикладів (adversarial examples).

ВКАЗІВКИ З ПІДГОТОВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

Атаки ухилення спрямовані на зміну вхідних зразків, щоб змусити модель видавати неправильні передбачення. Це відбувається шляхом додавання малих, майже непомітних змін до зображень, які впливають на рішення нейронної мережі. У даній роботі використовується набір даних MNIST, що містить зображення рукописних цифр (0-9) розміром 28×28 пікселів.

Для генерації змагальних прикладів застосовується метод градієнтного пошуку, зокрема Projected Gradient Descent (PGD), який дозволяє знайти мінімальні зміни у зображеннях, що призведуть до помилкової класифікації.

Лабораторна робота виконується у середовищі Kaggle або локально за умови встановлення Pytorch. Потрібно встановити необхідні бібліотеки перед запуском коду.

> УВАГА! Для виконання цієї роботи необхідно встановити додаткові компоненти Pytorch.

Додаткову інформацію при підготовці до роботи можна отримати:

1. Mitigating Evasion Aacks to Deep Neural Networks via Region-based Classification. URL: https://arxiv.org/pdf/1709.05583

2. EG-Booster: Explanation-Guided Booster of ML Evasion Attacks. URL: https://arxiv.org/pdf/2108.13930

3. ML 108: Evasion Attacks on MNIST dataset. URL: https://samsclass.info/129S/proj/ML108.htm

ТЕОРЕТИЧНІ ВІДОМОСТІ

Штучні нейронні мережі є потужним інструментом для розв'язання складних задач машинного навчання, особливо у сфері комп'ютерного зору. Вони досягли високої точності у розпізнаванні образів, класифікації та прогнозуванні. Однак, попри їхню ефективність, такі моделі мають вразливості, які можуть бути використані для маніпуляції їхнім виходом. Одним із найпоширеніших методів впливу є атака ухилення, яка спрямована на зміну вхідних даних з метою примусу класифікатора до помилкового передбачення. У цьому дослідженні атака ухилення реалізується на нейронній мережі, яка працює з набором даних MNIST.

Набір даних MNIST є одним із найбільш використовуваних у сфері машинного навчання, оскільки містить зображення рукописних цифр від 0 до 9, представлені у градаціях сірого та мають фіксований розмір 28×28 пікселів. Великий обсяг анотованих даних дає змогу ефективно навчати класифікаційні моделі та тестувати їхню продуктивність. Нейронні мережі, зокрема згорткові нейронні мережі, є найбільш поширеним типом моделей для класифікації зображень. Вони складаються з вхідного шару, кількох прихованих шарів та вихідного шару, який забезпечує передбачення класу. Кожен шар має власні параметри, що дозволяють мережі вчитися виділяти важливі особливості зображень та робити точні передбачення.

Попри високу продуктивність, нейронні мережі є вразливими до атак, які можуть змінювати вхідні дані, залишаючись практично непомітними для людського ока. Атака ухилення є однією з таких загроз. Вона базується на додаванні спеціально згенерованого шуму до вхідних даних, який змушує модель помилятися. Цей шум зазвичай не впливає на сприйняття людиною, проте у багатовимірному просторі ознак він достатньо змінює структуру зображення, щоб класифікатор видав помилковий результат.

Математично атаку ухилення можна розглядати як задачу оптимізації, де визначається таке збурення вхідного зразка, яке максимізує функцію втрат класифікатора, при цьому залишаючись обмеженим певною нормою, щоб уникнути суттєвих змін у зображенні. Найбільш поширеними методами атак є Fast Gradient Sign Method (FGSM) та Projected Gradient Descent (PGD). Метод FGSM базується на використанні знака градієнта функції втрат відносно вхідних даних, що дозволяє швидко знайти напрямок зміни вхідного зразка, який призведе до помилкового передбачення. Метод PGD є вдосконаленою версією FGSM, яка використовує ітеративний підхід для

пошуку оптимального збурення, що гарантує ефективніше введення моделі в оману.

Основною причиною вразливості нейронних мереж до атак ухилення є те, що вони працюють у високовимірному просторі, де навіть невеликі зміни у вході можуть суттєво вплинути на процес прийняття рішення. Це пояснюється тим, що функції активації та параметри моделі можуть змінювати свої значення навіть при незначних змінах у вихідних пікселях, що призводить до неправильного класифікування.

Для захисту від атак ухилення застосовуються різні методи, серед яких одним із найефективніших є адвесаріальне навчання. Цей підхід полягає у включенні змагальних прикладів у процес тренування моделі, що дає змогу підвищити її стійкість до атак. Додатково використовуються методи згладжування прийняття рішень, детекції змагальних прикладів та обмеження градієнтних змін, що допомагає зменшити ймовірність успішної атаки.

Таким чином, атаки ухилення є серйозною загрозою для нейронних мереж, особливо у сферах, де точність класифікації має критичне значення, таких як автономний транспорт, фінансовий сектор або біометричні системи. Аналіз ефективності атак на MNIST дозволяє оцінити вразливість класифікаторів та розробити відповідні методи захисту, що є важливим кроком у підвищенні безпеки штучного інтелекту.

ПРИКЛАДИ ПРАКТИЧНИХ ЗАВДАНЬ

Крок 1. У браузері перейдіть по https://www.kaggle.com/

Якщо ви бачите синю кнопку "Увійти" у верхньому правому куті, натисніть на неї та увійдіть в обліковий запис Kaggle. У меню виберіть "Файл", "Нова записна книжка".

Крок 2. Встановлення SecML та Pytorch

Виконайте наступні команди:

```
!pip install secml
import secml
!pip install torch torchvision
```

Крок 3. Створення згорткової нейронної мережі для класифікації трьох класів MNIST. Підготовка вхідних даних для згорткової нейронної мережі

Для використання згорткової нейронної мережі (CNN) при класифікації зображень MNIST необхідно правильно підготувати вхідні дані. Оскільки згорткові мережі очікують вхід у форматі (batch_size, channels, height, width), потрібно змінити форму вхідних даних відповідно до очікуваного розміру.

У випадку набору даних MNIST, кожне зображення має розмір 28×28 пікселів у градаціях сірого, тому кількість каналів дорівнює 1. Отже, перед подачею даних у модель потрібно перетворити їх у формат (-1, 1, 28, 28), де:

– -1 вказує на автоматичне визначення розміру першої осі (кількість зразків у пакеті);

– 1 відповідає кількості каналів (чорно-білі зображення мають один канал);

– 28×28 визначає висоту та ширину зображення.

Для автоматичного виконання цього перетворення можна використати модуль torchvision.transforms, який дозволяє застосовувати різні трансформації до зображень. У наступних кроках буде показано, як використовувати transforms для підготовки даних перед навчанням згорткової нейронної мережі.

1. Імпорт бібліотек

```
import torch
from torch import nn
```

import torch – завантаження основної бібліотеки для роботи з нейронними мережами.

from torch import nn – імпорт модуля для створення нейромережевих шарів (nn.Module).

2. Оголошення класу згорткової нейронної мережі

```
class MNIST3cCNN(nn.Module):
    """Model with input size (-1, 28, 28) for MNIST 3-classes dataset."""
```

class MNIST3cCNN(nn.Module) – визначення класу нейромережі, яка наслідує функціонал nn.Module.

Модель використовується для класифікації трьох класів із набору MNIST.

3. Ініціалізація шарів нейромережі

```
def __init__(self):
    super(MNIST3cCNN, self).__init__()
    self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
    self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
    self.conv2_drop = nn.Dropout2d()
    self.fc1 = nn.Linear(320, 50)
    self.fc2 = nn.Linear(50, 3)
```

Згорткові шари:

– self.conv1 = nn.Conv2d(1, 10, kernel_size=5) – перший згортковий шар приймає 1 канал (градації сірого MNIST) та створює 10 карт ознак, використовуючи ядро 5×5.

– self.conv2 = nn.Conv2d(10, 20, kernel_size=5) – другий згортковий шар приймає 10 карт ознак з попереднього шару та створює 20 нових карт.

Шар Dropout:

– self.conv2_drop = nn.Dropout2d() – застосовує випадкове відключення нейронів у 2D-просторі, що допомагає запобігти перенавчанню.

Повнозв’язні шари:

– self.fc1 = nn.Linear(320, 50) – перший повнозв’язний шар приймає 320 ознак (після згорткових шарів) і передає 50 ознак на наступний рівень.

– self.fc2 = nn.Linear(50, 3) – фінальний шар, що перетворює 50 ознак на 3 вихідні класи (класифікація 3 цифр MNIST).

4. Оголошення методу переднього проходу (forward pass)

```
def forward(self, x):
    x = torch.relu(torch.max_pool2d(self.conv1(x), 2))
    x = torch.relu(torch.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
    x = x.view(-1, 320)
    x = torch.relu(self.fc1(x))
    return self.fc2(x)
```

Обробка згорткових шарів

x = torch.relu(torch.max_pool2d(self.conv1(x), 2))

Передавання вхідних даних через перший згортковий шар (conv1).

Використання ReLU (Rectified Linear Unit) як функції активації.

Застосування 2×2 Max Pooling, що зменшує розмір карти ознак.

x = torch.relu(torch.max_pool2d(self.conv2_drop(self.conv2(x)), 2))

Передача через другий згортковий шар (conv2).

Застосування Dropout для зменшення перенавчання.

Використання Max Pooling для зменшення розмірності.

Перетворення у вектор

x = x.view(-1, 320)

Перетворення 2D-матриці в вектор розмірності 320, щоб передати дані у повнозв’язний шар.

Обробка повнозв’язних шарів x = torch.relu(self.fc1(x))

Передача через перший повнозв’язний шар (fc1).

Використання ReLU для нелінійності.

return self.fc2(x)

Передача вихідних значень через другий повнозв’язний шар (fc2).

На виході – 3 числа (ймовірності належності до кожного з класів MNIST).

Ця модель є згортковою нейронною мережею (CNN) для класифікації 3 класів із набору MNIST. Вона складається з двох згорткових шарів із ядром 5×5, шару Dropout, двох повнозв’язних шарів і функції активації ReLU.

Під час переднього проходу модель приймає вхідне зображення розміром 28×28, застосовує згортку, ReLU, Max Pooling, Dropout, а потім подає дані у повнозв’язний класифікатор. Вихідний шар містить 3 нейрони, кожен з яких відповідає одному з класів.

Ця архітектура дозволяє ефективно класифікувати рукописні цифри MNIST та є основою для розширення нейронних мереж у складніших задачах комп’ютерного зору.

Крок 4. Завантаження набору даних MNIST та підготовка вхідних даних

Тепер можна завантажити набір даних MNIST для використання у згортковій нейронній мережі. Важливо пам’ятати, що вхідні дані мають форму (1, 1, 28, 28) відповідно до формату NCHW, де:

- N – розмір пакету (batch size),
- C – кількість каналів (1 для зображень у градаціях сірого),
- H – висота зображення (28 пікселів),
- W – ширина зображення (28 пікселів).

Оскільки згорткові нейронні мережі очікують саме такий формат, необхідно переконатися, що вхідні зображення правильно відформатовані перед передачею в мережу.

Вхідна форма передається як вхідний параметр обгортки (wrapper), що відповідає за автоматичне перетворення зображень у потрібний формат перед їхньою подачею до нейромережі. Це гарантує, що модель отримає правильно підготовлені дані для навчання та тестування.

1. Визначення кількості зразків у тестовій вибірці

```
n_ts = 1000  # number of testing set samples
```

Змінна n_ts задає кількість зразків у тестовому наборі, що буде завантажено. У цьому випадку вибирається 1000 тестових зображень.

2. Завантаження набору даних MNIST

```
from secml.data.loader import CDataLoaderMNIST
digits = (1, 5, 9)
loader = CDataLoaderMNIST()
```

Імпортується CDataLoaderMNIST – клас для роботи з MNIST.

Вибираються цифри (1, 5, 9) для класифікації.

Створюється об'єкт loader, який дозволяє завантажувати MNIST.

3. Завантаження тренувального та тестового наборів

```
tr = loader.load('training', digits=digits)
ts = loader.load('testing', digits=digits, num_samples=n_ts)
```

loader.load('training', digits=digits) завантажує всі тренувальні зображення з MNIST, але тільки для вибраних цифр 1, 5 та 9.

loader.load('testing', digits=digits, num_samples=n_ts) завантажує 1000 тестових зображень із вибраними цифрами.

Таким чином, набір даних обмежується трьома класами (1, 5, 9), що дозволяє використовувати його для класифікації трьох класів замість десяти стандартних у MNIST.

4. Нормалізація даних

```
tr.X /= 255
ts.X /= 255
```

Ділення кожного значення пікселя на 255 нормалізує їх у діапазон [0,1].

Оскільки зображення у MNIST мають значення пікселів від 0 до 255, нормалізація необхідна для кращого навчання нейромережі.

Крок 5. Використання обгортки CClassifierPyTorch для інтеграції моделі

На цьому етапі можна знову використати обгортку CClassifierPyTorch, щоб зробити модель доступною для роботи в бібліотеці SecML. Обгортка дозволяє взаємодіяти з нейронною мережею, застосовувати класифікаційні функції та здійснювати атаки ухилення або отруєння.

Важливо зазначити, що під час створення обгортки потрібно передати форму вхідних даних (input shape) як параметр. Це забезпечить правильне перетворення вхідних даних перед передачею у нейронну мережу.

Після цього модель можна використовувати для класифікації зображень MNIST, оцінки продуктивності або аналізу її стійкості до атак.

1. Встановлення випадкового зерна (random seed)

```
torch.manual_seed(0)
```

Встановлюється фіксоване зерно випадкових чисел у PyTorch, що гарантує відтворюваність результатів під час тренування моделі.

Це означає, що під час запуску коду нейронна мережа ініціалізується однаково при кожному виконанні.

2. Ініціалізація згорткової нейронної мережі

```
net = MNIST3cCNN()
```

Створюється об’єкт нейронної мережі MNIST3cCNN, яка була визначена раніше.

Ця модель складається з двох згорткових шарів, шару Dropout та двох повнозв’язних шарів, що адаптовані для класифікації трьох класів MNIST (1, 5, 9).

3. Визначення функції втрат

```
criterion = nn.CrossEntropyLoss()
```

Використовується функція крос-ентропійної втрати (CrossEntropyLoss), яка підходить для багатокласової класифікації.

Ця функція порівнює передбачені ймовірності класів із фактичними мітками та мінімізує різницю між ними.

4. Налаштування оптимізатора

```
optimizer = optim.SGD(net.parameters(),
                      lr=0.01, momentum=0.9)
```

Використовується оптимізатор стохастичного градієнтного спуску (SGD).

lr=0.01 – швидкість навчання (learning rate), яка визначає, наскільки сильно змінюються ваги моделі після кожної ітерації.

momentum=0.9 – прискорює процес навчання, дозволяючи швидше знаходити оптимальні параметри.

5. Створення обгортки CClassifierPyTorch

```
from secml.ml.classifiers import CClassifierPyTorch
clf = CClassifierPyTorch(model=net,
                         loss=criterion,
                         optimizer=optimizer,
                         epochs=20,
                         batch_size=20,
                         input_shape=(1, 28, 28),
                         random_state=0)
```

Обгортка CClassifierPyTorch дозволяє працювати з PyTorch моделлю у бібліотеці SecML.

Аргументи обгортки:

– model=net – передається нейронна мережа MNIST3cCNN.

– loss=criterion – функція втрат для оптимізації.

– optimizer=optimizer – оптимізатор, що використовується для оновлення ваг моделі.

– epochs=20 – модель тренується 20 епох, тобто проходить повний набір даних 20 разів.

– batch_size=20 – у кожному оновленні ваг використовується 20 зразків (розмір пакету).

– input_shape=(1, 28, 28) – вхідні зображення мають 1 канал та розмір 28×28, що відповідає формату NCHW.

– random_state=0 – задається випадковий стан для відтворюваності експерименту.

Щоб заощадити час, завантажимо з попередньо навчену модель.

```
# NBVAL_IGNORE_OUTPUT
from secml.model_zoo import load_model
clf = load_model('mnist159-cnn')
```

І тепер ми можемо перевірити, наскільки добре модель класифікує цифри.

```
label_torch = clf.predict(ts.X, return_decision_function=False)

from secml.ml.peval.metrics import CMetric
metric = CMetric.create('accuracy')
acc_torch = metric.performance_score(ts.Y, label_torch)

print("Model Accuracy: {}".format(acc_torch))
```

Model Accuracy: 0.997

Крок 6. Генерація атак ухилення (Evasion Attacks) на нейронну мережу

Після тренування нейронної мережі можна створити змагальні приклади (adversarial examples), як це було зроблено у попередньому навчальному ноутбуці MNIST tutorial.

Головна ідея атаки ухилення полягає у додаванні мінімальних змін до вхідних зображень таким чином, щоб нейронна мережа почала класифікувати їх неправильно, хоча для людини вони залишатимуться майже ідентичними до оригінальних.

Код для генерації змагальних прикладів буде схожий на попередній, за винятком того, що тут до об'єкта CAttackEvasionPGDLS передається інший класифікатор – натренована нейронна мережа.

Наступним кроком буде виконання атаки ухилення, оцінка її ефективності та аналіз точності класифікатора до та після атаки. Це дозволить дослідити вразливість моделі та оцінити рівень її стійкості до змагальних прикладів.

1. Вибір підмножини тестового набору

```
# For simplicity, let's attack a subset of the test set
attack_ds = ts[:10, :]
```

Вибирається 10 зразків із тестового набору (ts[:10, :]) для проведення атаки.

Це зменшує обчислювальні витрати, дозволяючи швидше виконати атаку.

2. Визначення параметрів атаки

```
noise_type = 'l2'  # Type of perturbation 'l1' or 'l2'
dmax = 3.0  # Maximum perturbation
```

```
lb, ub = 0., 1.  # Bounds of the attack space. Can be set to `None` for unbounded
y_target = None  # None if `error-generic` or a class label for `error-specific`
```

noise_type = 'l2' – використовується L2-норма для оцінки величини змін у зображенні.

dmax = 3.0 – встановлює максимальний рівень пертурбації.

lb, ub = 0., 1. – обмеження на зміну пікселів у межах [0,1] (оскільки MNIST нормалізований).

y_target = None – атака є "error-generic", тобто змагальні приклади створюються без фіксації конкретного класу.

Якщо y_target було б встановлено у значення певного класу, атака була б "error-specific", що означає навмисне переведення зразків у заданий клас.

3. Визначення параметрів оптимізації

```
solver_params = {
    'eta': 0.5,
    'eta_min': 2.0,
    'eta_max': None,
    'max_iter': 100,
    'eps': 1e-6
}
```

eta = 0.5 – початковий розмір кроку оптимізації.

eta_min = 2.0 – мінімальний крок зміни під час оновлення змагальних прикладів.

eta_max = None – без обмеження максимального кроку зміни.

max_iter = 100 – максимум 100 ітерацій градієнтного пошуку оптимального збурення.

eps = 1e-6 – порогове значення, при якому атака зупиняється, якщо зміни стали незначними.

Чим більше max_iter, тим більш ефективною буде атака, але це також збільшить час виконання.

4. Ініціалізація атаки ухилення PGD-LS

```
from secml.adv.attacks import CAttackEvasionPGDLS
pgd_ls_attack = CAttackEvasionPGDLS(classifier=clf,
                                    double_init_ds=tr,
                                    distance=noise_type,
```

```
                dmax=dmax,
                solver_params=solver_params,
                y_target=y_target)
```

CAttackEvasionPGDLS – реалізація Projected Gradient Descent with Line Search (PGD-LS).

– classifier=clf – передається нейронна мережа, на яку здійснюється атака.

– double_init_ds=tr – додаткова ініціалізація атаки на основі тренувального набору.

– distance=noise_type – використовує L2-норму для обмеження змін.

– dmax=dmax – встановлюється гранична величина збурення.

– solver_params=solver_params – задаються параметри оптимізації.

– y_target=y_target – атакуємо модель без фіксованого цільового класу (error-generic attack).

5. Запуск атаки

```
print("Attack started...")
eva_y_pred, _, eva_adv_ds, _ = pgd_ls_attack.run(attack_ds.X, attack_ds.Y)
print("Attack complete!")
```

Запускається атака (pgd_ls_attack.run) на вибраних 10 тестових зразках (attack_ds.X).

Повертаються результати атаки:

- eva_y_pred – передбачені мітки після атаки.
- eva_adv_ds – змінені зображення, що є змагальними прикладами.

Після завершення атаки виводиться повідомлення "Attack complete!".

Крок 7. Оцінка точності нейронної мережі до та після атаки

Цей код виконує оцінку точності класифікації нейронної мережі до та після атаки ухилення (Evasion Attack), використовуючи метрику точності (accuracy).

1. Обчислення точності моделі до атаки

```
acc = metric.performance_score(
   y_true=attack_ds.Y, y_pred=clf.predict(attack_ds.X))
```

clf.predict(attack_ds.X) – отримуємо передбачені класи нейронної мережі до атаки.

y_true=attack_ds.Y – істинні мітки тестового підмножини.

metric.performance_score() – обчислює відсоток правильно класифікованих зразків у неатакованому тестовому наборі.

Результат зберігається у змінній acc, яка містить початкову точність моделі перед атакою.

2. Обчислення точності моделі після атаки

```
acc_attack = metric.performance_score(
    y_true=attack_ds.Y, y_pred=eva_y_pred)
```

eva_y_pred – передбачені класи після атаки, отримані від pgd_ls_attack.run().

Метрика обчислює точність після атаки, тобто частку зразків, які класифіковано правильно після внесення збурень у вхідні дані.

Результат зберігається у acc_attack, яка містить точність моделі після атаки.

3. Виведення результатів точності до та після атаки

```
print("Accuracy on reduced test set before attack: {:.2%}".format(acc))
print("Accuracy on reduced test set after attack: {:.2%}".format(acc_attack))
```

Виводяться значення точності до та після атаки у відсотках ({:.2%} формат відображає число у відсотковій формі).

Очікуваний ефект атаки (рис. 5.1):

– Точність перед атакою (acc) повинна бути високою (наприклад, 98-99%).

– Точність після атаки (acc_attack) суттєво знижується (до 10-50% або навіть менше), що підтверджує ефективність атаки ухилення.

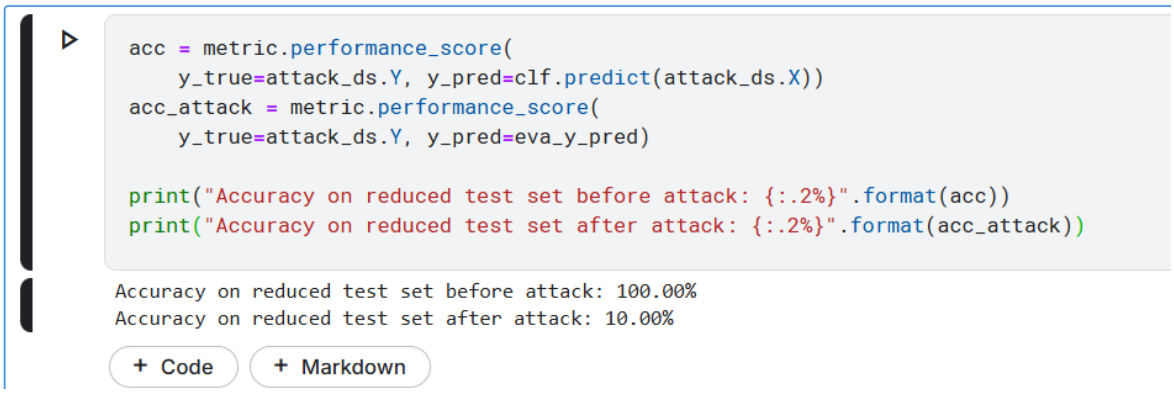


Рис. 5.1. Оцінка ефекту атаки

Крок 8. Візуалізація атакованих зразків

1. Імпорт бібліотеки для побудови графіків

```
from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline
```

Імпортується CFigure з бібліотеки secml.figure, яка використовується для побудови графіків.

%matplotlib inline – спеціальна команда для відображення графіків у Jupyter Notebook (не потрібна при запуску поза блокнотом).

2. Функція для візуалізації зображень MNIST

```
def show_digits(samples, preds, labels, digs, n_display=8):
    samples = samples.atleast_2d()
    n_display = min(n_display, samples.shape[0])
    fig = CFigure(width=n_display*2, height=3)
```

Функція show_digits() приймає такі параметри:

– samples – набір зображень (матриця пікселів).

– preds – передбачені нейромережею мітки класів.

– labels – істинні мітки класів.

– digs – список відповідних міток цифр MNIST (наприклад, [1, 5, 9]).

– n_display=8 – кількість цифр для відображення (максимум 8).

– samples.atleast_2d() – забезпечує, що samples має щонайменше двовимірний вигляд (необхідно для роботи з одиничними зображеннями).

– n_display = min(n_display, samples.shape[0]) – обмежує кількість зображень, якщо samples містить менше, ніж n_display.

Створюється об'єкт fig (CFigure), що міститиме графіки розміром (ширина = n_display * 2, висота = 3).

3. Побудова графіків для кожного зображення

```
for idx in range(n_display):
    fig.subplot(2, n_display, idx+1)
    fig.sp.xticks([])
    fig.sp.yticks([])
    fig.sp.imshow(samples[idx, :].reshape((28, 28)), cmap='gray')
```

Цикл for idx in range(n_display) ітерується через n_display зображень.

fig.subplot(2, n_display, idx+1) створює дворядковий набір графіків, де кожне зображення розташовується у своїй комірці.

Відключаються підписи осей (xticks, yticks) для зручного перегляду.

fig.sp.imshow(samples[idx, :].reshape((28, 28)), cmap='gray') відображає зображення у відтінках сірого (cmap='gray').

4. Відображення міток із кольоровим кодуванням

```
fig.sp.title("{} ({})".format(digits[labels[idx].item()], digs[preds[idx].item()]),
             color=("green" if labels[idx].item()==preds[idx].item() else "red"))
```

Виводиться заголовок графіка, який показує правильний клас цифри (ліворуч) і передбачений клас (у дужках праворуч).

Якщо передбачення вірне (labels[idx] == preds[idx]), мітка відображається зеленим.

Якщо передбачення невірне, мітка відображається червоним.

5. Відображення результатів

```
fig.show()
```

Після завершення циклу відображаються всі зображення із відповідними мітками.

6. Виклик функції для візуалізації до і після атаки

```
show_digits(attack_ds.X[0, :], clf.predict(attack_ds.X[0, :]), attack_ds.Y[0, :], digits)
show_digits(eva_adv_ds.X[0, :], clf.predict(eva_adv_ds.X[0, :]), eva_adv_ds.Y[0, :], digits)
```

Перший виклик show_digits()

– Відображає оригінальні зразки (attack_ds.X[0, :]).

– Передбачає класи нейромережею (clf.predict(attack_ds.X[0, :])).

– Порівнює їх із істинними мітками (attack_ds.Y[0, :]).

Другий виклик show_digits()

– Відображає атаковані зразки (eva_adv_ds.X[0, :]).

– Перевіряє, як нейромережа класифікує змагальні приклади (clf.predict(eva_adv_ds.X[0, :])).

Якщо атака ухилення була успішною, класифікатор зробить більше помилок на атакованих зразках, що буде показано червоними мітками у візуалізації (рис. 5.2).

Рис. 5.2. Результати класифікації

Використовуючи метод PGD-LS, атака змінює пікселі зображень так, що нейронна мережа починає помилково класифікувати їх, незважаючи на те, що для людини зміни майже непомітні.

Візуалізація дозволяє оцінити ефективність атаки:

– Якщо після атаки більшість передбачень стали неправильними (червоні мітки), то нейромережа вразлива до атак ухилення.

– Якщо модель все ще правильно класифікує цифри, то вона стійка до атак.

Цей підхід допомагає аналізувати безпеку моделей машинного навчання та тестувати різні методи захисту від змагальних атак.

ЗАГАЛЬНЕ ЗАВДАННЯ ДЛЯ ВИКОНАННЯ

1. Створити архітектуру згорткової нейронної мережі (CNN) для класифікації цифр MNIST.

a. Налаштувати функцію втрат та оптимізатор.

b. Виконати тренування мережі та оцінити її точність на тестовому наборі.

2. Реалізація атаки ухилення

a. Використати змагальний метод PGD (Projected Gradient Descent) для створення змагальних прикладів.

b. Встановити параметри атаки:

c. Тип пертурбації (L∞ або L2).

d. Максимальний рівень змін (ε = 0.3).

e. Кількість ітерацій атаки (50 ітерацій).

f. Генерувати змагальні зразки, додаючи малий шум до зображень.

3. Оцінка ефективності атаки

a. Перевірити точність нейронної мережі до та після атаки.

b. Візуалізувати оригінальні та атаковані зразки.

c. Проаналізувати, як зміна пікселів впливає на рішення нейронної мережі.

4. Відповісти на контрольні запитання та підготувати звіт, який містить:

a. Опис навченої нейронної мережі для класифікації MNIST.

b. Опис змагальних прикладів, які змушують класифікатор помилятися.

c. Оцінки точності класифікації до та після атаки.

d. Візуалізацію результатів атаки ухилення.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке атака ухилення (Evasion Attack) і як вона впливає на нейронні мережі?

2. Які особливості набору даних MNIST роблять його популярним для тестування моделей машинного навчання?

3. Чому нейронні мережі є вразливими до атак ухилення?

4. Що таке змагальні приклади (adversarial examples) і як вони генеруються?

5. Які методи атак ухилення використовуються найчастіше? Поясніть принцип дії FGSM і PGD.

6. Яка роль градієнта функції втрат у генерації змагальних прикладів?

7. Як змінюється передбачення класифікатора після застосування атаки ухилення?

8. Які стратегії можна застосувати для підвищення стійкості нейронних мереж до атак ухилення?

9. Як адвесаріальне навчання може допомогти у боротьбі з атаками ухилення?

10. Які реальні загрози можуть виникнути внаслідок успішних атак ухилення на нейронні мережі в критичних системах?

СПИСОК ЛІТЕРАТУРИ

Основна література

1. Штучний інтелект і безпека: практичний посібник / Ю. І. Когут; за ред. док-ра тех. наук, проф. А. С. Довгопологого – Київ : Консалтингова компанія «СІДКОН» ; ВД Дакор, 2024. – 294 с.

2. Скіцько, О., Складанний, П., Ширшов, Р., Гуменюк, М., & Ворохоб, М. (2023). ЗАГРОЗИ ТА РИЗИКИ ВИКОРИСТАННЯ ШТУЧНОГО ІНТЕЛЕКТУ. Електронне фахове наукове видання «Кібербезпека: освіта, наука, техніка», 2(22), 6–18. https://doi.org/10.28925/2663-4023.2023.22.618.

3. Addo A., Centhala S., Shanmugam M. Artificial Intelligence Design and Solution for Risk and Security. NY: Business Expert Press, 2020. – 154 p.

4. Ahmed Mohiuddin. Explainable Artificial Intelligence for Cyber Security: Next Generation Artificial Intelligence. Springer, Cham, 2022. – 280 p.

5. Artificial Intelligence (AI) Design and Solutions for Risk and Security. New York: Business Expert Press, LLC, 2020. – 150 pp.

Додаткова література

6. Bansal Payal et al. (Edited). Artificial Intelligence and Communication Techniques in Industry 5.0 // Payal Bansal, Rajeev Kumar, Ashwani Kumar, Daniel D. Dasig, Jr. – CRC Press, 2025. – 421 p.

7. Batina L., Bäck T., Buhan I., Picek S. (eds.) Security and Artificial Intelligence: A Crossdisciplinary Approach. Springer, 2022. – 365 p..

8. Bhardwaj T., Upadhyay H., Sharma T.K. (eds.) Artificial Intelligence in Cyber Security: Theories and Applications. Springer, 2023. – 144 p.

9. Das R. Generative AI: Phishing and Cybersecurity Metrics. CRC Press, 2025. – 177 p.

10. Das R. Practical AI for Cybersecurity. Boca Raton: CRC Press, 2021. – 292 p.

Електронні ресурси

11. SecML: Secure and Explainable Machine Learning in Python. URL: https://secml.readthedocs.io/en/v0.15/

12. Security for Artificial Intelligence Software and Services URL: https://www.coursera.org/learn/security-for-artificial-intelligence-software-and-services

13. AI Security. URL: https://www.coursera.org/learn/ai-security

Навчальне видання

ЄВСЕЄВ Сергій Петрович
ШМАТКО Олександр Віталійович
АХІЄЗЕР Олена Борисівна
СОКОЛ Владислав Євгенович
ЧЕРНОВА Наталя Леонідівна

АТАКИ НА СИСТЕМИ ШТУЧНОГО ІНТЕЛЕКТУ

Навчально-практичний посібник

Серія «Кібербезпека та штучний інтелект»
За загальною редакцією доктора технічних наук, професора С. П. Євсеєва

Відповідальний за випуск *С. П. Євсеєв*

Керівник видавничих проектів *С. В. Піча*
Дизайн та верстка *К. А. Рижова*

Підписано до друку 07.02.2025 р. Зам. № 2025-09.
Формат 60×84/16. Гарнітура Times New Roman. Папір офсетний.
Обл.-вид. арк. 7,1875. Ум.-друк. арк. 5,75.

Видавництво ПП «Новий Світ-2000»
e-mail: novsv2000@gmail.com
Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців і розповсюджувачів видавничої продукції: серія ДК № 59 від 25.05.2000 року, видане Державним комітетом інформаційної політики, телебачення та радіомовлення України.

Виготовлено : Видавець ФОП Піча С. В.
а/с 5026, м. Львів-53, 79053, Україна
e-mail: novsv2016@ukr.net, https://ns2000.com.ua/
+38 068-978-94-42, +38 050-337-58-46
Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції: серія ДК № 5069 від 22.03.2016 року, видане Державним комітетом інформаційної політики, телебачення та радіомовлення України.