МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА” ІНСТИТУТ

КОМП’ЮТЕРНИХ НАУК ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ КАФЕДРА СИСТЕМ ШТУЧНОГО ІНТЕЛЕКТУ



**ЗВІТ**

про виконання лабораторної роботи №3 з курсу «Машинне навчання»

**Виконав:**

ст. групи КНСШ-12 Служала Л.Т. **Перевірила**:

Бойко Н.І.

ЛЬВІВ - 2023

У сттаті використані клдасичні методи машинного навчання, а саме Логістична регресія (Logistic regression), Метод опорних векторів (Support vector machine), а також ймовірнісна нейронна мережа (Probabilistic neural network), і її різновиди поліпшення з використання SVM та поліпшення.

Для дослідження даних моделей було використано власний датасет, для класифікації якості сплавів 4 класи (від поганого до чудового). Стаття була опублікована без коду і датасету, проте автори радо поділилися зі мною другим.

У даному юпітер ноутбуці буде виконанна наступна робота:

1. Застосування додаткового методу (на вибір)

Робота була виконання студентом Бойчуком Олеслав-Іваном Михайловичем у рамках курсу "Машинне навчання".

# 0. Підвантаження бібліотек

Пдівантажуємо бібліотеки для подальшого використання у даній роботі.

from future import print\_function import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler, MinMaxScaler from sklearn.preprocessing import LabelEncoder

from tensorflow.keras.models import Sequential from tensorflow.keras.layers import Dense

from tensorflow.keras.utils import to\_categorical import time

import numpy as np

from sklearn.decomposition import PCA from sklearn.manifold import TSNE from sklearn import svm

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import KFold

from sklearn.model\_selection import cross\_validate import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import seaborn as sns from sklearn import svm import numpy as np

import matplotlib.pyplot as plt import pandas as pd

%matplotlib inline

Підключаємося до гугл диску і завантажуємо дані за допомогою pandas бібліотеки.

from google.colab import drive drive.mount('/content/drive')

*# Load data from CSV using pandas*

train\_data = pd.read\_csv('/content/drive/MyDrive/data/MTrainData.txt', header=None)

test\_data = pd.read\_csv('/content/drive/MyDrive/data/MTestData.txt', header=None)

*# The target column is named 20* X\_train = train\_data.drop(20, axis=1) y\_train = train\_data[20]

X\_test = test\_data.drop(20, axis=1) y\_test = test\_data[20]

*# Encode labels to numerical values*

label\_encoder = LabelEncoder()

y\_train = label\_encoder.fit\_transform(y\_train) y\_test = label\_encoder.transform(y\_test)

*# Standardize features by removing the mean and scaling to unit variance*

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train) X\_test = scaler.transform(X\_test)

*# y\_train = to\_categorical(y\_train) # y\_test = to\_categorical(y\_test)*

Mounted at /content/drive

# Застосування додаткового методу (на вибір)

Оскліьки у даній роботі потрібно використати (запропонувати) додатковий метод на вибір якийне був використаний статті. Я вирішив використати різні методи і спробувати порівняти між собою їх кінцеву точнітсь.

Перед початком роботи проводимо обєднання тестових даних із навчальними для кросвалідації і пошуку гіперпараметрів.

train\_data = pd.read\_csv('/content/drive/MyDrive/data/MTrainData.txt', header=None)

test\_data = pd.read\_csv('/content/drive/MyDrive/data/MTestData.txt', header=None)

train\_data.columns = train\_data.columns.astype(str) test\_data.columns = test\_data.columns.astype(str)

X\_train = train\_data.drop('20', axis=1) y\_train = train\_data['20']

X\_test = test\_data.drop('20', axis=1) y\_test = test\_data['20']

X = pd.concat([X\_train, X\_test]) y = pd.concat([y\_train, y\_test])

## Гребневий класифікатор Ridge classifier

* + Найвища точність 68.9%

from sklearn.model\_selection import RepeatedStratifiedKFold from sklearn.model\_selection import GridSearchCV

from sklearn.linear\_model import RidgeClassifier from sklearn.model\_selection import KFold

model = RidgeClassifier()

alpha = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

*# define grid search*

grid = dict(alpha=alpha)

cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=10, random\_state=1)

grid\_search = GridSearchCV(estimator=model, param\_grid=grid, n\_jobs=- 1, cv=cv, scoring='accuracy',error\_score=0)

grid\_result = grid\_search.fit(X, y)

*# summarize results*

print("Best: %f using %s" % (grid\_result.best\_score\_, grid\_result.best\_params\_))

means = grid\_result.cv\_results\_['mean\_test\_score']

stds = grid\_result.cv\_results\_['std\_test\_score'] params = grid\_result.cv\_results\_['params']

for mean, stdev, param in zip(means, stds, params): print("%f (%f) with: %r" % (mean, stdev, param))

Best: 0.688963 using {'alpha': 0.2}

0.688754 (0.047681) with: {'alpha': 0.1}

0.688963 (0.047263) with: {'alpha': 0.2}

0.688963 (0.047263) with: {'alpha': 0.3}

0.688963 (0.047355) with: {'alpha': 0.4}

0.688958 (0.047391) with: {'alpha': 0.5}

0.688958 (0.047391) with: {'alpha': 0.6}

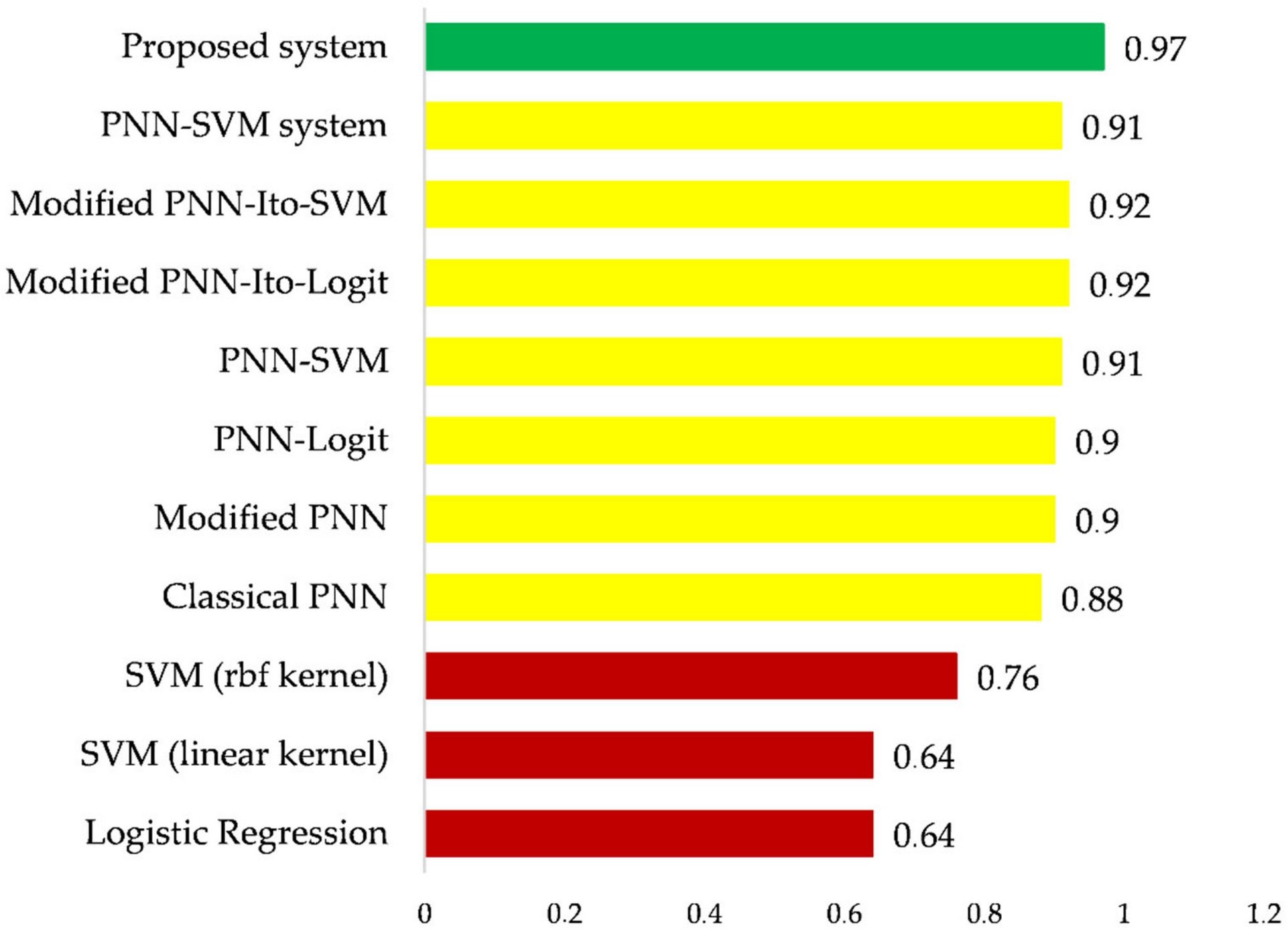
0.688958 (0.047391) with: {'alpha': 0.7}

0.688542 (0.047219) with: {'alpha': 0.8}

0.688542 (0.047767) with: {'alpha': 0.9}

0.688542 (0.047767) with: {'alpha': 1.0}

Як бачимо із результатів використанні RidgeClassifier отримали найвищу точність у 68.896% що є недостатньо хорошою точнітсь для даного датасету



## Випадковий ліс Random Forest Classifier

* + Найвища точність 98.61%

from sklearn.model\_selection import RepeatedStratifiedKFold from sklearn.model\_selection import GridSearchCV

from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier() n\_estimators = [10, 100, 1000] max\_features = ['sqrt', 'log2'] *# define grid search*

grid = dict(n\_estimators=n\_estimators,max\_features=max\_features) cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=10, random\_state=1)

grid\_search = GridSearchCV(estimator=model, param\_grid=grid, n\_jobs=- 1, cv=cv, scoring='accuracy',error\_score=0)

grid\_result = grid\_search.fit(X, y)

*# summarize results*

print("Best: %f using %s" % (grid\_result.best\_score\_, grid\_result.best\_params\_))

means = grid\_result.cv\_results\_['mean\_test\_score'] stds = grid\_result.cv\_results\_['std\_test\_score'] params = grid\_result.cv\_results\_['params']

for mean, stdev, param in zip(means, stds, params): print("%f (%f) with: %r" % (mean, stdev, param))

Best: 0.986082 using {'max\_features': 'log2', 'n\_estimators': 1000} 0.933274 (0.043799) with: {'max\_features': 'sqrt', 'n\_estimators': 10}

0.981191 (0.028158) with: {'max\_features': 'sqrt', 'n\_estimators':

100}

0.985358 (0.025920) with: {'max\_features': 'sqrt', 'n\_estimators':

1000}

0.932447 (0.048714) with: {'max\_features': 'log2', 'n\_estimators': 10}

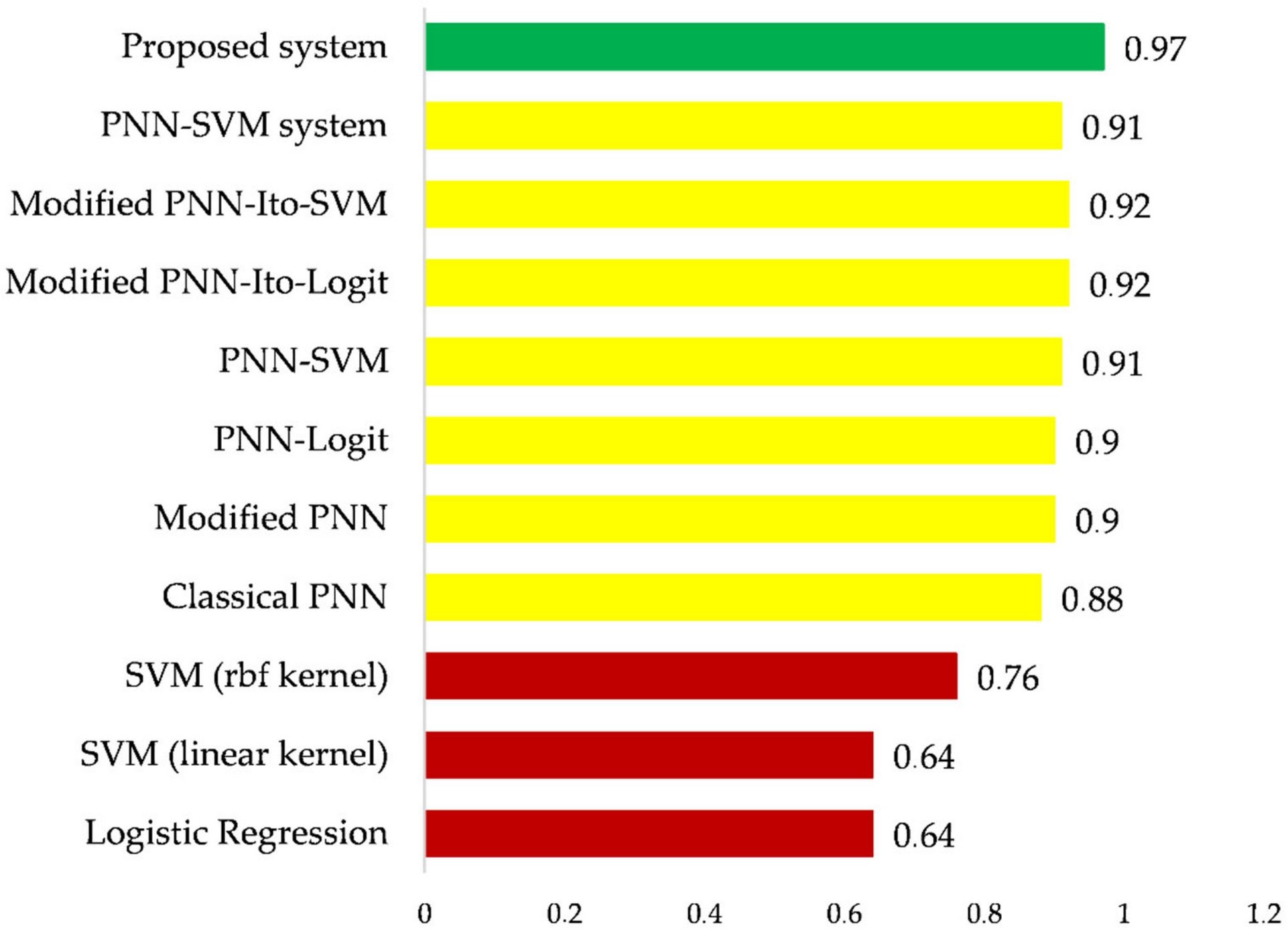
0.980482 (0.028521) with: {'max\_features': 'log2', 'n\_estimators':

100}

0.986082 (0.027105) with: {'max\_features': 'log2', 'n\_estimators':

1000}

Випадковий ліс чудово із поставленою задачею і показав вищий і точніший результат, ніж найкращий у статті.



## Багатошаровий перцептрон Multilayer perceptron

* + Найвища точність 97.72%

from sklearn.model\_selection import RepeatedStratifiedKFold from sklearn.model\_selection import GridSearchCV

from sklearn.neural\_network import MLPClassifier

model = MLPClassifier()

solvers = ['lbfgs', 'adam', 'sgd'] max\_iters=[1000] hidden\_layer\_sizess=[(100, 20)] alphas=[1e-5]

*# define grid search*

grid = dict(solver=solvers, max\_iter=max\_iters, hidden\_layer\_sizes=hidden\_layer\_sizess, alpha=alphas) cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=10, random\_state=1)

grid\_search = GridSearchCV(estimator=model, param\_grid=grid, n\_jobs=- 1, cv=cv, scoring='accuracy', error\_score=0)

grid\_result = grid\_search.fit(X, y)

*# summarize results*

print("Best: %f using %s" % (grid\_result.best\_score\_,

grid\_result.best\_params\_))

means = grid\_result.cv\_results\_['mean\_test\_score'] stds = grid\_result.cv\_results\_['std\_test\_score'] params = grid\_result.cv\_results\_['params']

for mean, stdev, param in zip(means, stds, params): print("%f (%f) with: %r" % (mean, stdev, param))

Best: 0.977225 using {'alpha': 1e-05, 'hidden\_layer\_sizes': (100, 20), 'max\_iter': 1000, 'solver': 'lbfgs'}

0.977225 (0.017779) with: {'alpha': 1e-05, 'hidden\_layer\_sizes': (100,

20), 'max\_iter': 1000, 'solver': 'lbfgs'}

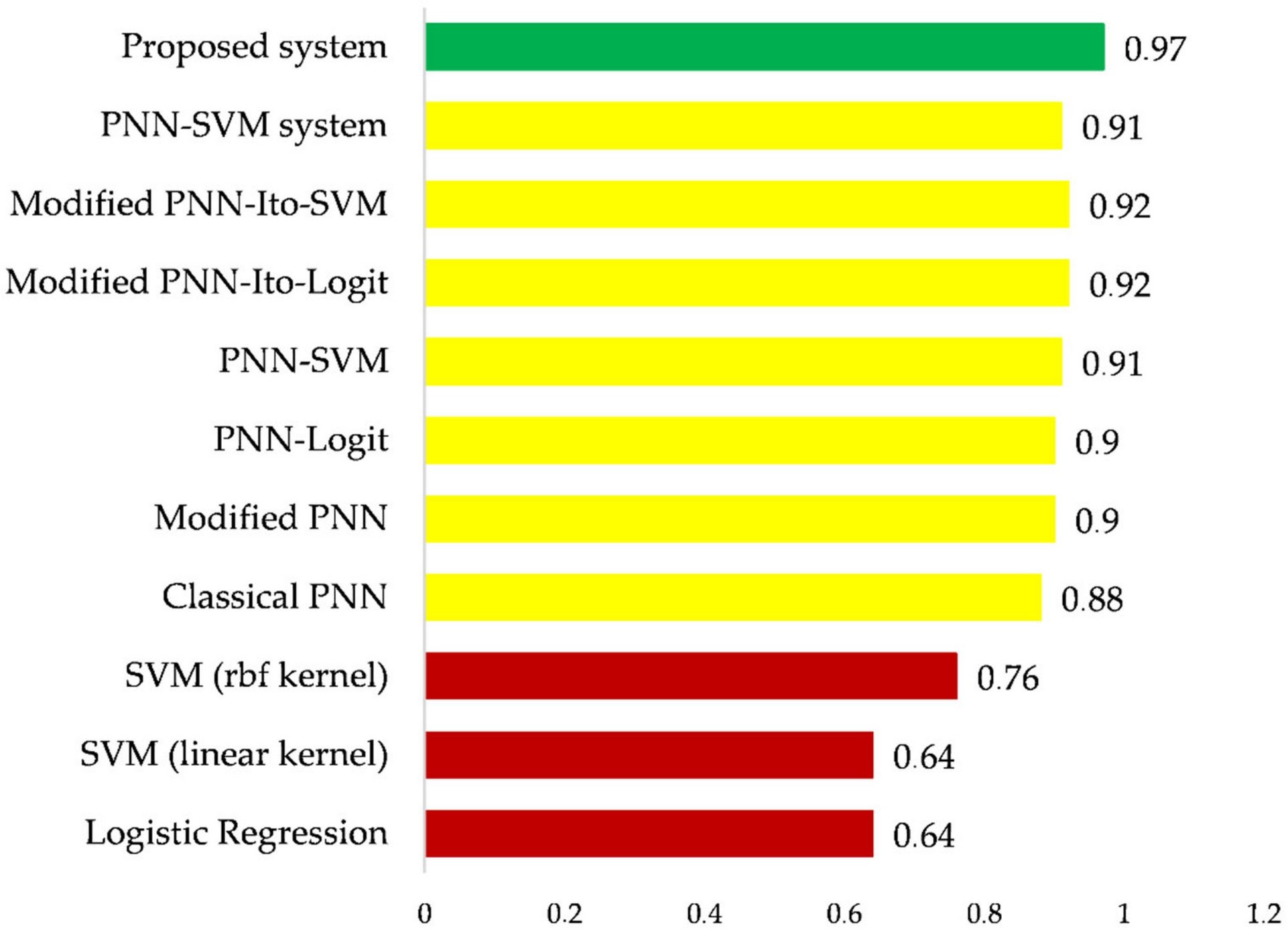
0.977017 (0.017632) with: {'alpha': 1e-05, 'hidden\_layer\_sizes': (100,

20), 'max\_iter': 1000, 'solver': 'adam'}

0.977017 (0.017632) with: {'alpha': 1e-05, 'hidden\_layer\_sizes': (100,

20), 'max\_iter': 1000, 'solver': 'sgd'}

Так MLP показав гірший результат ніж попередній (RF), проте точнітсь є на рівні із найкращим методом зі статті. Що свідчуть про хорошу точність моделі.



## Глибоке навчання Deep Learning

* + Найвища точність 100%

У даному прикладі буде використано, стандартний підхід глибокого навчання з вчителем на табличних даних. Спершу створимо функцію, яка буде малювати метрики.

def plot\_metrics(history, ignore\_epoch = 0):

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,5)) fig.suptitle('Horizontally stacked subplots') ax1.plot(history.history['loss'][ignore\_epoch:], label= "loss") ax1.plot(history.history['val\_loss'][ignore\_epoch:],

label="validation loss")

min\_los = (history.history['val\_loss'] [np.argmin(history.history['val\_loss'])])

ax1.title.set\_text(f"Min Val Loss: {min\_los:.5f}") ax1.set\_xlabel("Number of Epochs") ax1.set\_ylabel("Loss")

ax1.legend() ax2.plot(history.history['accuracy'][ignore\_epoch:],

label="accuracy") ax2.plot(history.history['val\_accuracy'][ignore\_epoch:],

label="validation accuracy")

max\_acc = (history.history['val\_accuracy'] [np.argmax(history.history['val\_accuracy'])])

ax2.title.set\_text(f"Max Val Accuracy: {max\_acc:.5f}") ax2.set\_xlabel("Number of Epochs") ax2.set\_ylabel("Acc")

ax2.legend() plt.show()

Завантажуємо датасет і виконуємо "препроцессінг" даних для кращих результаті класифікації. Створюємо звичайну модель для класифікації багатьох класів. У результаті отримуємо 3 шарову нейронну мережу без дропаутів, вчимо за допомгою MSE+ADAM.

*# Load data from CSV using pandas*

train\_data = pd.read\_csv('/content/drive/MyDrive/data/MTrainData.txt', header=None)

test\_data = pd.read\_csv('/content/drive/MyDrive/data/MTestData.txt', header=None)

*# Assume the target column is named 'target'* X\_train = train\_data.drop(20, axis=1) y\_train = train\_data[20]

X\_test = test\_data.drop(20, axis=1) y\_test = test\_data[20]

*# Encode labels to numerical values*

label\_encoder = LabelEncoder()

y\_train = label\_encoder.fit\_transform(y\_train) y\_test = label\_encoder.transform(y\_test)

*# Standardize features by removing the mean and scaling to unit*

*variance*

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train) X\_test = scaler.transform(X\_test)

*# Build the neural network model*

model = Sequential([

Dense(128, activation='relu', input\_shape=(X\_train.shape[1],)), Dense(64, activation='relu'),

Dense(32, activation='relu'), Dense(4, activation='sigmoid')

])

y\_train = to\_categorical(y\_train) y\_test = to\_categorical(y\_test)

model.compile(optimizer='adam', loss='mse', metrics=['accuracy']) model.summary()

Model: "sequential\_2"

Layer (type) Output Shape Param #

=================================================================

|  |  |  |  |
| --- | --- | --- | --- |
| dense\_8 (Dense) | (None, | 128) | 2688 |
|  |  |  |  |
| dense\_9 (Dense) | (None, | 64) | 8256 |
|  |  |  |  |
| dense\_10 (Dense) | (None, | 32) | 2080 |
|  |  |  |  |
| dense\_11 (Dense) | (None, | 4) | 132 |

=================================================================

Total params: 13156 (51.39 KB)

Trainable params: 13156 (51.39 KB) Non-trainable params: 0 (0.00 Byte)

*# Train the model*

history = model.fit(X\_train, y\_train, epochs=100, batch\_size=32, validation\_split=0.1)

*# Evaluate the model on test data*

model.evaluate(X\_test, y\_test)

Epoch 1/100

12/12 [==============================] - 3s 53ms/step - loss: 0.2344 -

accuracy: 0.2820 - val\_loss: 0.2022 - val\_accuracy: 0.3542 Epoch 2/100

12/12 [==============================] - 0s 8ms/step - loss: 0.1807 -

accuracy: 0.4987 - val\_loss: 0.1660 - val\_accuracy: 0.5521 Epoch 3/100

12/12 [==============================] - 0s 7ms/step - loss: 0.1497 -

accuracy: 0.6371 - val\_loss: 0.1447 - val\_accuracy: 0.5938 Epoch 4/100

12/12 [==============================] - 0s 9ms/step - loss: 0.1247 -

accuracy: 0.7232 - val\_loss: 0.1215 - val\_accuracy: 0.7083 Epoch 5/100

12/12 [==============================] - 0s 8ms/step - loss: 0.1032 -

accuracy: 0.7963 - val\_loss: 0.1031 - val\_accuracy: 0.8125 Epoch 6/100

12/12 [==============================] - 0s 9ms/step - loss: 0.0860 -

accuracy: 0.8198 - val\_loss: 0.0920 - val\_accuracy: 0.7812 Epoch 7/100

12/12 [==============================] - 0s 8ms/step - loss: 0.0728 -

accuracy: 0.8460 - val\_loss: 0.0809 - val\_accuracy: 0.8438 Epoch 8/100

12/12 [==============================] - 0s 8ms/step - loss: 0.0617 -

accuracy: 0.8721 - val\_loss: 0.0738 - val\_accuracy: 0.8438 Epoch 9/100

12/12 [==============================] - 0s 5ms/step - loss: 0.0527 -

accuracy: 0.9060 - val\_loss: 0.0647 - val\_accuracy: 0.8750 Epoch 10/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0448 -

accuracy: 0.9321 - val\_loss: 0.0579 - val\_accuracy: 0.8646 Epoch 11/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0380 -

accuracy: 0.9452 - val\_loss: 0.0521 - val\_accuracy: 0.9062 Epoch 12/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0323 -

accuracy: 0.9556 - val\_loss: 0.0451 - val\_accuracy: 0.9375 Epoch 13/100

12/12 [==============================] - 0s 5ms/step - loss: 0.0271 -

accuracy: 0.9661 - val\_loss: 0.0405 - val\_accuracy: 0.9271 Epoch 14/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0226 -

accuracy: 0.9713 - val\_loss: 0.0344 - val\_accuracy: 0.9583 Epoch 15/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0188 -

accuracy: 0.9765 - val\_loss: 0.0327 - val\_accuracy: 0.9688 Epoch 16/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0163 -

accuracy: 0.9765 - val\_loss: 0.0276 - val\_accuracy: 0.9583 Epoch 17/100

12/12 [==============================] - 0s 5ms/step - loss: 0.0135 -

accuracy: 0.9791 - val\_loss: 0.0263 - val\_accuracy: 0.9688 Epoch 18/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0113 -

accuracy: 0.9817 - val\_loss: 0.0226 - val\_accuracy: 0.9792

Epoch 19/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0096 -

accuracy: 0.9817 - val\_loss: 0.0201 - val\_accuracy: 0.9896 Epoch 20/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0083 -

accuracy: 0.9896 - val\_loss: 0.0188 - val\_accuracy: 0.9896 Epoch 21/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0071 -

accuracy: 0.9922 - val\_loss: 0.0166 - val\_accuracy: 0.9896 Epoch 22/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0062 -

accuracy: 0.9922 - val\_loss: 0.0162 - val\_accuracy: 0.9896 Epoch 23/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0053 -

accuracy: 0.9922 - val\_loss: 0.0146 - val\_accuracy: 0.9896 Epoch 24/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0047 -

accuracy: 0.9922 - val\_loss: 0.0137 - val\_accuracy: 0.9896 Epoch 25/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0042 -

accuracy: 0.9922 - val\_loss: 0.0127 - val\_accuracy: 0.9896 Epoch 26/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0039 -

accuracy: 0.9922 - val\_loss: 0.0121 - val\_accuracy: 0.9896 Epoch 27/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0036 -

accuracy: 0.9922 - val\_loss: 0.0114 - val\_accuracy: 0.9896 Epoch 28/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0034 -

accuracy: 0.9922 - val\_loss: 0.0109 - val\_accuracy: 0.9896 Epoch 29/100

12/12 [==============================] - 0s 5ms/step - loss: 0.0032 -

accuracy: 0.9922 - val\_loss: 0.0102 - val\_accuracy: 0.9896 Epoch 30/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0031 -

accuracy: 0.9922 - val\_loss: 0.0099 - val\_accuracy: 0.9896 Epoch 31/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0030 -

accuracy: 0.9922 - val\_loss: 0.0095 - val\_accuracy: 0.9896 Epoch 32/100

12/12 [==============================] - 0s 5ms/step - loss: 0.0028 -

accuracy: 0.9922 - val\_loss: 0.0093 - val\_accuracy: 0.9896 Epoch 33/100

12/12 [==============================] - 0s 5ms/step - loss: 0.0028 -

accuracy: 0.9922 - val\_loss: 0.0088 - val\_accuracy: 0.9896 Epoch 34/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0027 -

accuracy: 0.9922 - val\_loss: 0.0084 - val\_accuracy: 0.9896 Epoch 35/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0026 -

accuracy: 0.9922 - val\_loss: 0.0083 - val\_accuracy: 0.9896 Epoch 36/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0026 -

accuracy: 0.9922 - val\_loss: 0.0079 - val\_accuracy: 0.9896 Epoch 37/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0025 -

accuracy: 0.9922 - val\_loss: 0.0076 - val\_accuracy: 0.9896 Epoch 38/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0025 -

accuracy: 0.9922 - val\_loss: 0.0075 - val\_accuracy: 0.9896 Epoch 39/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0024 -

accuracy: 0.9922 - val\_loss: 0.0073 - val\_accuracy: 0.9896 Epoch 40/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0024 -

accuracy: 0.9922 - val\_loss: 0.0071 - val\_accuracy: 0.9896 Epoch 41/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0023 -

accuracy: 0.9948 - val\_loss: 0.0069 - val\_accuracy: 0.9896 Epoch 42/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0022 -

accuracy: 0.9948 - val\_loss: 0.0067 - val\_accuracy: 0.9896 Epoch 43/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0020 -

accuracy: 1.0000 - val\_loss: 0.0065 - val\_accuracy: 0.9896 Epoch 44/100

12/12 [==============================] - 0s 4ms/step - loss: 0.0013 -

accuracy: 1.0000 - val\_loss: 0.0059 - val\_accuracy: 1.0000 Epoch 45/100

12/12 [==============================] - 0s 6ms/step - loss: 0.0012 -

accuracy: 1.0000 - val\_loss: 0.0044 - val\_accuracy: 1.0000 Epoch 46/100

12/12 [==============================] - 0s 4ms/step - loss: 6.4355e-

04 - accuracy: 1.0000 - val\_loss: 0.0044 - val\_accuracy: 1.0000 Epoch 47/100

12/12 [==============================] - 0s 4ms/step - loss: 4.3757e-

04 - accuracy: 1.0000 - val\_loss: 0.0045 - val\_accuracy: 1.0000 Epoch 48/100

12/12 [==============================] - 0s 5ms/step - loss: 3.9517e-

04 - accuracy: 1.0000 - val\_loss: 0.0042 - val\_accuracy: 1.0000 Epoch 49/100

12/12 [==============================] - 0s 6ms/step - loss: 3.3005e-

04 - accuracy: 1.0000 - val\_loss: 0.0037 - val\_accuracy: 1.0000 Epoch 50/100

12/12 [==============================] - 0s 6ms/step - loss: 3.0183e-

04 - accuracy: 1.0000 - val\_loss: 0.0036 - val\_accuracy: 1.0000 Epoch 51/100

12/12 [==============================] - 0s 6ms/step - loss: 2.7658e-

04 - accuracy: 1.0000 - val\_loss: 0.0035 - val\_accuracy: 1.0000 Epoch 52/100

12/12 [==============================] - 0s 6ms/step - loss: 2.5664e-

04 - accuracy: 1.0000 - val\_loss: 0.0033 - val\_accuracy: 1.0000 Epoch 53/100

12/12 [==============================] - 0s 6ms/step - loss: 2.4121e-

04 - accuracy: 1.0000 - val\_loss: 0.0033 - val\_accuracy: 1.0000 Epoch 54/100

12/12 [==============================] - 0s 4ms/step - loss: 2.2873e-

04 - accuracy: 1.0000 - val\_loss: 0.0032 - val\_accuracy: 1.0000 Epoch 55/100

12/12 [==============================] - 0s 4ms/step - loss: 2.1428e-

04 - accuracy: 1.0000 - val\_loss: 0.0031 - val\_accuracy: 1.0000 Epoch 56/100

12/12 [==============================] - 0s 6ms/step - loss: 2.0387e-

04 - accuracy: 1.0000 - val\_loss: 0.0030 - val\_accuracy: 1.0000 Epoch 57/100

12/12 [==============================] - 0s 6ms/step - loss: 1.9234e-

04 - accuracy: 1.0000 - val\_loss: 0.0029 - val\_accuracy: 1.0000 Epoch 58/100

12/12 [==============================] - 0s 4ms/step - loss: 1.8348e-

04 - accuracy: 1.0000 - val\_loss: 0.0029 - val\_accuracy: 1.0000 Epoch 59/100

12/12 [==============================] - 0s 6ms/step - loss: 1.7458e-

04 - accuracy: 1.0000 - val\_loss: 0.0028 - val\_accuracy: 1.0000 Epoch 60/100

12/12 [==============================] - 0s 4ms/step - loss: 1.6637e-

04 - accuracy: 1.0000 - val\_loss: 0.0027 - val\_accuracy: 1.0000 Epoch 61/100

12/12 [==============================] - 0s 5ms/step - loss: 1.5905e-

04 - accuracy: 1.0000 - val\_loss: 0.0026 - val\_accuracy: 1.0000 Epoch 62/100

12/12 [==============================] - 0s 5ms/step - loss: 1.5173e-

04 - accuracy: 1.0000 - val\_loss: 0.0026 - val\_accuracy: 1.0000 Epoch 63/100

12/12 [==============================] - 0s 5ms/step - loss: 1.4567e-

04 - accuracy: 1.0000 - val\_loss: 0.0025 - val\_accuracy: 1.0000 Epoch 64/100

12/12 [==============================] - 0s 6ms/step - loss: 1.3952e-

04 - accuracy: 1.0000 - val\_loss: 0.0024 - val\_accuracy: 1.0000 Epoch 65/100

12/12 [==============================] - 0s 6ms/step - loss: 1.3387e-

04 - accuracy: 1.0000 - val\_loss: 0.0024 - val\_accuracy: 1.0000 Epoch 66/100

12/12 [==============================] - 0s 6ms/step - loss: 1.2823e-

04 - accuracy: 1.0000 - val\_loss: 0.0023 - val\_accuracy: 1.0000 Epoch 67/100

12/12 [==============================] - 0s 6ms/step - loss: 1.2324e-

04 - accuracy: 1.0000 - val\_loss: 0.0023 - val\_accuracy: 1.0000

Epoch 68/100

12/12 [==============================] - 0s 5ms/step - loss: 1.1858e-

04 - accuracy: 1.0000 - val\_loss: 0.0022 - val\_accuracy: 1.0000 Epoch 69/100

12/12 [==============================] - 0s 4ms/step - loss: 1.1409e-

04 - accuracy: 1.0000 - val\_loss: 0.0022 - val\_accuracy: 1.0000 Epoch 70/100

12/12 [==============================] - 0s 6ms/step - loss: 1.0974e-

04 - accuracy: 1.0000 - val\_loss: 0.0022 - val\_accuracy: 1.0000 Epoch 71/100

12/12 [==============================] - 0s 4ms/step - loss: 1.0613e-

04 - accuracy: 1.0000 - val\_loss: 0.0021 - val\_accuracy: 1.0000 Epoch 72/100

12/12 [==============================] - 0s 6ms/step - loss: 1.0237e-

1. - accuracy: 1.0000 - val\_loss: 0.0021 - val\_accuracy: 1.0000 Epoch 73/100

12/12 [==============================] - 0s 6ms/step - loss: 9.8859e-

1. - accuracy: 1.0000 - val\_loss: 0.0021 - val\_accuracy: 1.0000 Epoch 74/100

12/12 [==============================] - 0s 6ms/step - loss: 9.5354e-

05 - accuracy: 1.0000 - val\_loss: 0.0020 - val\_accuracy: 1.0000 Epoch 75/100

12/12 [==============================] - 0s 4ms/step - loss: 9.2087e-

05 - accuracy: 1.0000 - val\_loss: 0.0020 - val\_accuracy: 1.0000 Epoch 76/100

12/12 [==============================] - 0s 5ms/step - loss: 8.9134e-

05 - accuracy: 1.0000 - val\_loss: 0.0019 - val\_accuracy: 1.0000 Epoch 77/100

12/12 [==============================] - 0s 5ms/step - loss: 8.6366e-

05 - accuracy: 1.0000 - val\_loss: 0.0019 - val\_accuracy: 1.0000 Epoch 78/100

12/12 [==============================] - 0s 6ms/step - loss: 8.3641e-

05 - accuracy: 1.0000 - val\_loss: 0.0019 - val\_accuracy: 1.0000 Epoch 79/100

12/12 [==============================] - 0s 6ms/step - loss: 8.1072e-

05 - accuracy: 1.0000 - val\_loss: 0.0018 - val\_accuracy: 1.0000 Epoch 80/100

12/12 [==============================] - 0s 6ms/step - loss: 7.8508e-

05 - accuracy: 1.0000 - val\_loss: 0.0018 - val\_accuracy: 1.0000 Epoch 81/100

12/12 [==============================] - 0s 4ms/step - loss: 7.6142e-

05 - accuracy: 1.0000 - val\_loss: 0.0018 - val\_accuracy: 1.0000 Epoch 82/100

12/12 [==============================] - 0s 6ms/step - loss: 7.3809e-

05 - accuracy: 1.0000 - val\_loss: 0.0017 - val\_accuracy: 1.0000 Epoch 83/100

12/12 [==============================] - 0s 6ms/step - loss: 7.1732e-

05 - accuracy: 1.0000 - val\_loss: 0.0017 - val\_accuracy: 1.0000 Epoch 84/100

12/12 [==============================] - 0s 6ms/step - loss: 6.9853e-

05 - accuracy: 1.0000 - val\_loss: 0.0017 - val\_accuracy: 1.0000 Epoch 85/100

12/12 [==============================] - 0s 5ms/step - loss: 6.7953e-

05 - accuracy: 1.0000 - val\_loss: 0.0017 - val\_accuracy: 1.0000 Epoch 86/100

12/12 [==============================] - 0s 4ms/step - loss: 6.5831e-

05 - accuracy: 1.0000 - val\_loss: 0.0016 - val\_accuracy: 1.0000 Epoch 87/100

12/12 [==============================] - 0s 6ms/step - loss: 6.3912e-

05 - accuracy: 1.0000 - val\_loss: 0.0016 - val\_accuracy: 1.0000 Epoch 88/100

12/12 [==============================] - 0s 6ms/step - loss: 6.2270e-

05 - accuracy: 1.0000 - val\_loss: 0.0016 - val\_accuracy: 1.0000 Epoch 89/100

12/12 [==============================] - 0s 6ms/step - loss: 6.0542e-

05 - accuracy: 1.0000 - val\_loss: 0.0016 - val\_accuracy: 1.0000 Epoch 90/100

12/12 [==============================] - 0s 6ms/step - loss: 5.8902e-

05 - accuracy: 1.0000 - val\_loss: 0.0015 - val\_accuracy: 1.0000 Epoch 91/100

12/12 [==============================] - 0s 5ms/step - loss: 5.7463e-

05 - accuracy: 1.0000 - val\_loss: 0.0015 - val\_accuracy: 1.0000 Epoch 92/100

12/12 [==============================] - 0s 6ms/step - loss: 5.6055e-

05 - accuracy: 1.0000 - val\_loss: 0.0015 - val\_accuracy: 1.0000 Epoch 93/100

12/12 [==============================] - 0s 6ms/step - loss: 5.4601e-

05 - accuracy: 1.0000 - val\_loss: 0.0015 - val\_accuracy: 1.0000 Epoch 94/100

12/12 [==============================] - 0s 6ms/step - loss: 5.3261e-

05 - accuracy: 1.0000 - val\_loss: 0.0015 - val\_accuracy: 1.0000 Epoch 95/100

12/12 [==============================] - 0s 4ms/step - loss: 5.1815e-

05 - accuracy: 1.0000 - val\_loss: 0.0014 - val\_accuracy: 1.0000 Epoch 96/100

12/12 [==============================] - 0s 6ms/step - loss: 5.0684e-

05 - accuracy: 1.0000 - val\_loss: 0.0014 - val\_accuracy: 1.0000 Epoch 97/100

12/12 [==============================] - 0s 4ms/step - loss: 4.9355e-

05 - accuracy: 1.0000 - val\_loss: 0.0014 - val\_accuracy: 1.0000 Epoch 98/100

12/12 [==============================] - 0s 5ms/step - loss: 4.8270e-

05 - accuracy: 1.0000 - val\_loss: 0.0014 - val\_accuracy: 1.0000 Epoch 99/100

12/12 [==============================] - 0s 4ms/step - loss: 4.7001e-

05 - accuracy: 1.0000 - val\_loss: 0.0014 - val\_accuracy: 1.0000 Epoch 100/100

12/12 [==============================] - 0s 6ms/step - loss: 4.5966e-

05 - accuracy: 1.0000 - val\_loss: 0.0013 - val\_accuracy: 1.0000

3/3 [==============================] - 0s 7ms/step - loss: 0.0013 -

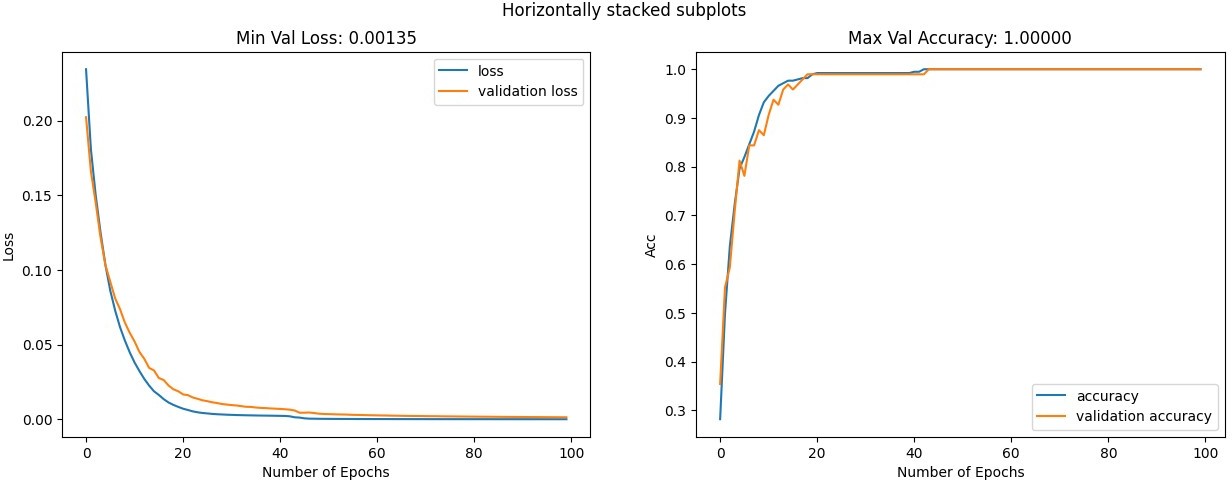
accuracy: 1.0000

[0.0013490788405761123, 1.0]

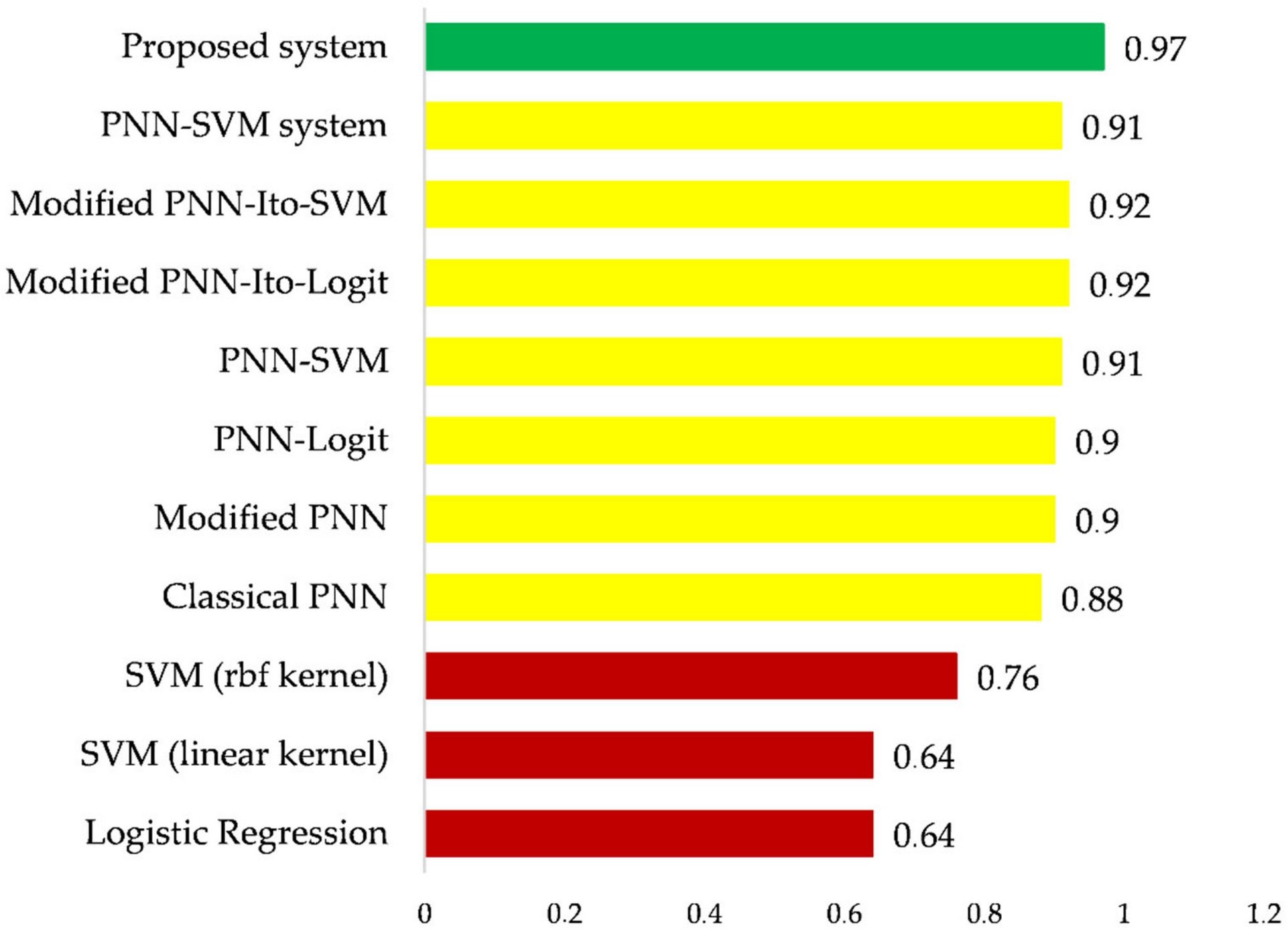
model.evaluate(X\_test, y\_test) plot\_metrics(history)

3/3 [==============================] - 0s 7ms/step - loss: 0.0013 -

accuracy: 1.0000



Як бачимо із результатів за 100 епох модель навчилася неперевершено класифікувати результати і отримала точнітсь у 100%. Що є найкращим результатом для даного датасету.



# Додаткове (за власним бажання): Використання різних методів проектування у двох та трьох вимірний простір

У данній частині роботи я спробува вспроектувати дані із багато вимірного. простору у манші вимір, щоб побачити як між собою відносяться дані. Дл яцього я використав PCA, t- SNE та UMAP

import pandas as pd

df = pd.read\_csv('/content/drive/MyDrive/data/MTestData.txt', header=None)

df = df.astype('int') df[20] = df[20].astype(str) df[20] = "class" + df[20]

df.columns = df.rename(columns=str).columns

import pandas as pd import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

*# Load data from CSV using pandas*

data = pd.read\_csv('/content/drive/MyDrive/data/MTrainData.txt', header=None)

data.columns = data.rename(columns=str).columns

*# Assume the target column is named 'target'*

X = data.drop('20', axis=1) y = data['20']

*# Assuming X contains your feature data # Perform PCA for 3 components*

pca = PCA(n\_components=3)

principal\_components = pca.fit\_transform(X)

*# Create a DataFrame with the principal components*

pc\_df = pd.DataFrame(data=principal\_components, columns=['PC1', 'PC2', 'PC3'])

*# Concatenate the principal components with the target variable if available*

*# For example, if your target variable is in a Series called 'target' # pc\_df = pd.concat([pc\_df, data['target']], axis=1)*

*# Create a 3D scatter plot*

fig = plt.figure(figsize=(8, 6))

ax = fig.add\_subplot(111, projection='3d') ax.set\_xlabel('Principal Component 1')

ax.set\_ylabel('Principal Component 2')

ax.set\_zlabel('Principal Component 3') ax.set\_title('3D PCA')

*# Replace 'target' with your actual target variable if available*

if '20' in data.columns:

targets = data['20'].unique()

colors = sns.color\_palette("husl", len(targets))

for target, color in zip(targets, colors): indices\_to\_keep = data['20'] == target ax.scatter(pc\_df.loc[indices\_to\_keep, 'PC1'],

pc\_df.loc[indices\_to\_keep, 'PC2'], pc\_df.loc[indices\_to\_keep, 'PC3'], color=color,

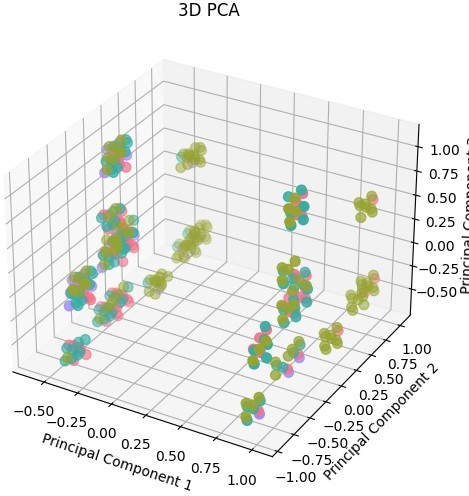
s=50)

else:

ax.scatter(pc\_df['PC1'], pc\_df['PC2'], pc\_df['PC3'], c='b',

marker='o', s=50)

plt.show()



Як бачимо із результатів PCA розбив дані на багато мальньких кластерів.

import pandas as pd import seaborn as sns

import matplotlib.pyplot as plt from sklearn.manifold import TSNE

*# Load data from CSV using pandas*

data = pd.read\_csv('/content/drive/MyDrive/data/MTrainData.txt', header=None)

data.columns = data.rename(columns=str).columns

*# Assume the target column is named 'target'*

X = data.drop('20', axis=1) y = data['20']

*# Assuming X contains your feature data*

*# Perform t-SNE for 3 components*

tsne = TSNE(n\_components=3, random\_state=42, perplexity=5, n\_iter=5000, n\_iter\_without\_progress=1000)

tsne\_components = tsne.fit\_transform(X)

*# Create a DataFrame with t-SNE components*

tsne\_df = pd.DataFrame(data=tsne\_components, columns=['t-SNE1', 't- SNE2', 't-SNE3'])

*# Concatenate the t-SNE components with the target variable if available*

*# For example, if your target variable is in a Series called 'target' # tsne\_df = pd.concat([tsne\_df, data['target']], axis=1)*

*# Create a 3D scatter plot*

fig = plt.figure(figsize=(8, 6))

ax = fig.add\_subplot(111, projection='3d') ax.set\_xlabel('t-SNE Component 1')

ax.set\_ylabel('t-SNE Component 2')

ax.set\_zlabel('t-SNE Component 3') ax.set\_title('3D t-SNE')

*# Replace 'target' with your actual target variable if available*

if '20' in data.columns:

targets = data['20'].unique()

colors = sns.color\_palette("husl", len(targets))

for target, color in zip(targets, colors): indices\_to\_keep = data['20'] == target ax.scatter(tsne\_df.loc[indices\_to\_keep, 't-SNE1'],

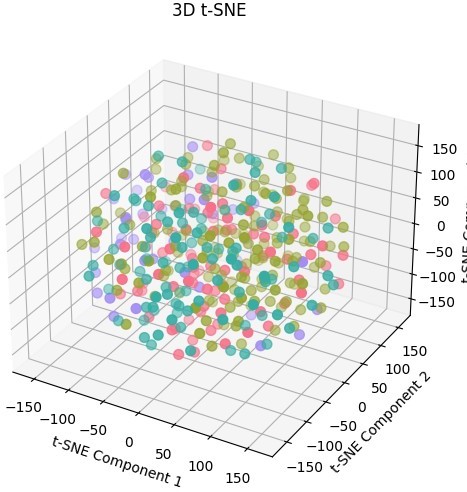
tsne\_df.loc[indices\_to\_keep, 't-SNE2'], tsne\_df.loc[indices\_to\_keep, 't-SNE3'], color=color,

s=50)

else:

ax.scatter(tsne\_df['t-SNE1'], tsne\_df['t-SNE2'], tsne\_df['t- SNE3'], c='b', marker='o', s=50)

plt.show()



У даному випадку t-SNEне вдалося "логічно" розбити дані у кластери для кожного класу

import pandas as pd

import plotly.express as px

from sklearn.manifold import TSNE

*# Load data from CSV using pandas*

data = pd.read\_csv('/content/drive/MyDrive/data/MTrainData.txt', header=None)

data.columns = data.rename(columns=str).columns

*# Assume the target column is named 'target'*

X = data.drop('20', axis=1)

y = data['20'].astype(int).astype(str)

*# Assuming X contains your feature data # Perform t-SNE for 3 components*

tsne = TSNE(n\_components=3, random\_state=42, perplexity=50, n\_iter=5000)

tsne\_components = tsne.fit\_transform(X)

*# Create a DataFrame with t-SNE components*

tsne\_df = pd.DataFrame(data=tsne\_components, columns=['t-SNE1', 't- SNE2', 't-SNE3'])

*# Concatenate the t-SNE components with the target variable if available*

*# For example, if your target variable is in a Series called 'target' # tsne\_df = pd.concat([tsne\_df, data['target']], axis=1)*

*# Create an interactive 3D scatter plot using Plotly Express*

fig = px.scatter\_3d(tsne\_df, x='t-SNE1', y='t-SNE2', z='t-SNE3',

labels={'t-SNE1': 't-SNE Component 1', 't-SNE2': 't-SNE Component 2', 't-SNE3': 't-SNE Component 3'},

color=y, *# addd from my self*

title='3D t-SNE Visualization')

*# Replace 'target' with your actual target variable if available*

if '20' in data.columns:

*# Assuming 'target' is categorical, you can use color='target' for coloring*

fig.update\_traces(marker=dict(size=5),

selector=dict(mode='markers+text'))

else:

*# If no target variable, use a single color for all data points*

fig.update\_traces(marker=dict(size=5, color='blue'),

selector=dict(mode='markers+text'))

*# Show the interactive plot*

fig.show()

*# from sklearn.manifold import TSNE # import plotly.express as px*

*# df = px.data.iris()*

*# features = df.loc[:, :'petal\_width']*

*# tsne = TSNE(n\_components=3, random\_state=0) # projections = tsne.fit\_transform(features, )*

*# fig = px.scatter\_3d(*

*# projections, x=0, y=1, z=2,*

*# color=df.species, labels={'color': 'species'} # )*

*# fig.update\_traces(marker\_size=8) # fig.show()*

Перепрогнав дані з іншими параметрами і використавши іншу візуалізацію. Можна побачити, що метод добре справився для класу 1. Більшість обєктів знаходяться поруч, що свідчить про близький звязок між обєктами одного класу.

!pip install umap-learn

import pandas as pd from umap import UMAP

import plotly.express as px

*# Load data from CSV using pandas*

data = pd.read\_csv('/content/drive/MyDrive/data/MTrainData.txt', header=None)

data.columns = data.rename(columns=str).columns

*# Assume the target column is named 'target'*

X = data.drop('20', axis=1)

y = data['20'].astype(int).astype(str)

*# Assuming X contains your feature data # Perform UMAP for 3 components*

umap\_model = UMAP(n\_components=3, random\_state=42)

umap\_components = umap\_model.fit\_transform(X)

*# Create a DataFrame with UMAP components*

umap\_df = pd.DataFrame(data=umap\_components, columns=['UMAP1', 'UMAP2', 'UMAP3'])

*# Concatenate the UMAP components with the target variable if available*

*# For example, if your target variable is in a Series called 'target' # umap\_df = pd.concat([umap\_df, data['target']], axis=1)*

*# Create an interactive 3D scatter plot using Plotly Express*

fig = px.scatter\_3d(umap\_df, x='UMAP1', y='UMAP2', z='UMAP3',

labels={'UMAP1': 'UMAP Component 1', 'UMAP2': 'UMAP Component 2', 'UMAP3': 'UMAP Component 3'},

color=y,

title='3D UMAP Visualization')

*# Replace 'target' with your actual target variable if available*

if 'target' in data.columns:

*# Assuming 'target' is categorical, you can use color='target' for coloring*

fig.update\_traces(marker=dict(size=5),

selector=dict(mode='markers+text'))

else:

*# If no target variable, use a single color for all data points*

fig.update\_traces(marker=dict(size=5, color='blue'),

selector=dict(mode='markers+text'))

*# Show the interactive plot*

fig.show()

/usr/local/lib/python3.10/dist-packages/umap/umap\_.py:1943: UserWarning:

n\_jobs value -1 overridden to 1 by setting random\_state. Use no seed for parallelism.

У даному випадку був використаний метод UMAP, який розбив дані на багато під кластерів у кожному із яких добре видно, що дані одного класу між собою знаходяться ближче.

# Висновки

import matplotlib.pyplot as plt

def create\_rating\_bar(rating):

*# Define the rating categories and corresponding colors*

categories = ['Ridge',

'MLP',

'RF', 'DL'

]

*# Create a bar plot*

plt.bar(categories, rating)

*# Set labels and title* plt.xlabel('Моделі') plt.ylabel('Точність') plt.title('Рейтинг моделей')

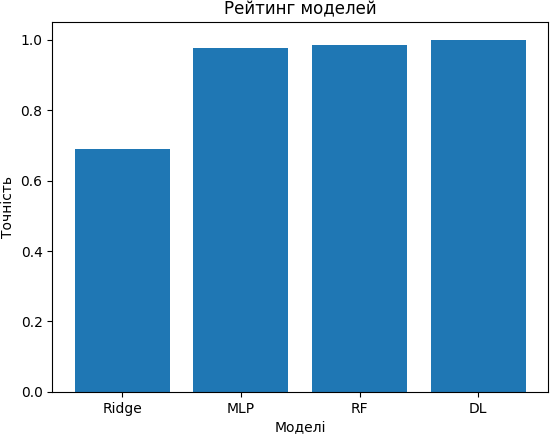
*# Show the plot*

plt.show()

*# Example usage*

rating\_data = [0.688963, 0.977225, 0.986082, 1.] *# Replace with your own data*

create\_rating\_bar(rating\_data)



Висновок: якщо порівнювати результапти наших моделей та моделей статті як бачимо 3 із 4 модель показали точність state-of-the-art, або краще ніж у статті.

