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 №1 та №2 з курсу «Машинне навчання»

**Виконав:**

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

Бойко Н.І.

ЛЬВІВ - 2024

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:

1-2. Відтворення експерименту статті, а також його покращення

# 0. Підвантаження бібліотек

Підвантажуємо бібліотеки, які будуть вкористовуватися на даному етапі роботи.

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 from sklearn import svm

import numpy as np

import matplotlib.pyplot as plt import pandas as pd

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

%matplotlib inline

# 1 - 2. Відтворення експерименту статті та покращення експерименту

Підключаємося до гугл диску, і завантажуємо тренувальну та тестову частину датасету за допомогою пандас (pandas) бібліотеки. Одразу після цього розбиваємо дані на X (вхідні дані) та y (вихідні).

Об'єднання тестових даних із навчальними для проведення кросвалідації на цьому датасеті.

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])

## Логістична регресія (Logistic Regression)

* Найвища точність у статті 0.64
* Найвища точність у покращенні 0.7105

Підвантажуємо кілька бібліотек, які використовуються лише для логічстиної регресії і інші які потрібні для проведення кросвалідації та пошуку найкращих гіпер-параметрів моделі.

*# example of grid searching key hyperparametres for logistic regression*

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

from sklearn.linear\_model import LogisticRegression

Створюємо модель, визначаємо які гіперпараметри будуть використовуватися, для кросвалідації 'solvers'(метод) & 'C' (обернена сила регуляризації). Для кросвалідації використовуємо 10 різних сплітів (розбиття) даних. повторюємо 10 разів алгоритм для точності результатів. І виводимо результат у консоль.

model = LogisticRegression()

solvers = ['lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga']

penalty = ['l2']

c\_values = [100, 10, 1.0, 0.1, 0.01]

*# define grid search*

grid = dict(solver=solvers,penalty=penalty,C=c\_values) 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: 0.710465 using {'C': 10, 'penalty': 'l2', 'solver': 'sag'}

0.701871 (0.051047) with: {'C': 100, 'penalty': 'l2', 'solver': 'lbfgs'}

0.696875 (0.048601) with: {'C': 100, 'penalty': 'l2', 'solver': 'liblinear'}

0.701658 (0.051090) with: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}

0.696875 (0.049045) with: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cholesky'}

0.694783 (0.053686) with: {'C': 100, 'penalty': 'l2', 'solver': 'sag'}

0.701871 (0.051301) with: {'C': 100, 'penalty': 'l2', 'solver': 'saga'}

0.704171 (0.051040) with: {'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}

0.696250 (0.048044) with: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}

0.704379 (0.050587) with: {'C': 10, 'penalty': 'l2', 'solver': 'newton-cg'}

0.696042 (0.048307) with: {'C': 10, 'penalty': 'l2', 'solver': 'newton-cholesky'}

0.710465 (0.045650) with: {'C': 10, 'penalty': 'l2', 'solver': 'sag'}

0.704171 (0.051040) with: {'C': 10, 'penalty': 'l2', 'solver': 'saga'}

0.700403 (0.048094) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs'}

0.695824 (0.048658) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'liblinear'}

0.700403 (0.048094) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cg'}

0.695199 (0.047998) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cholesky'}

0.700403 (0.048094) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'sag'}

0.700403 (0.048094) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'saga'}

0.683258 (0.045053) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}

0.679078 (0.046405) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}

0.683258 (0.045053) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}

0.675102 (0.046520) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cholesky'}

0.683258 (0.045053) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'sag'}

0.683258 (0.045053) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'saga'}

0.543240 (0.043462) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}

0.577677 (0.044347) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'liblinear'}

0.543240 (0.043462) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}

0.431733 (0.022845) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cholesky'}

0.543032 (0.043017) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'sag'}

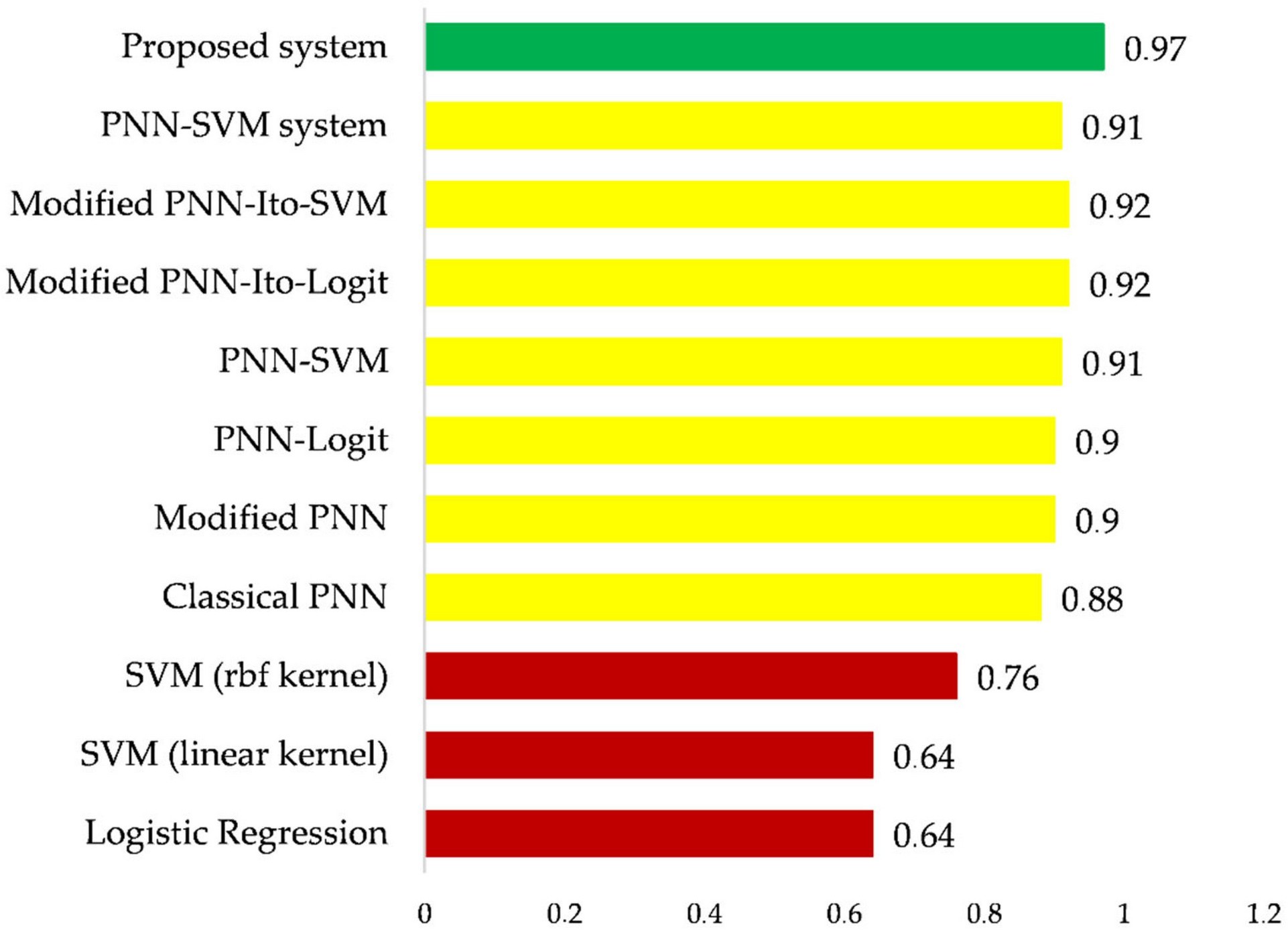
0.543240 (0.043462) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'saga'}

/usr/local/lib/python3.10/dist-packages/sklearn/linear\_model/

\_sag.py:350: ConvergenceWarning: The max\_iter was reached which means the coef\_ did not converge

warnings.warn(

Як бачимо найкраща точність є 71.05 % при використання наступних гіпер параметрів: Обернена сила регуляризації: 10, штраф (penalty) = l2, та алгоритм для використання в задачі оптимізації (solver) = sag. У статі зазначена куди менша точність що свідчить про не достатню кількість експериментів з боку науковців у цій тематиці.



## Метод опорних векторів (Support vector machine)

* Найвища точність у статті 0.76
* Найвища точність у покращенні 0.99875

Підвантажуємо кілька бібліотек, які використовуються лише для методу опорних векторів регресії і інші які потрібні для проведення кросвалідації та пошуку найкращих гіпер- параметрів моделі.

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

from sklearn.svm import SVC

Створюємо модель, визначаємо які гіперпараметри будуть використовуватися, для кросвалідації 'kernels'я (ядро, для вирівнювання не лінійних даних) 'solvers'(метод) & 'C' (обернена сила регуляризації). Для кросвалідації використовуємо 10 різних сплітів (розбиття) даних. повторюємо 10 разів алгоритм для точності результатів. І виводимо результат у консоль.

model = SVC()

kernel = ['poly', 'rbf', 'sigmoid', 'linear']

C = [100, 50, 10, 1.0, 0.1, 0.01]

gamma = ['scale', 'auto']

*# define grid search*

grid = dict(kernel=kernel,C=C,gamma=gamma)

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: 0.998750 using {'C': 100, 'gamma': 'scale', 'kernel': 'poly'}

0.998750 (0.009233) with: {'C': 100, 'gamma': 'scale', 'kernel': 'poly'}

0.998542 (0.010314) with: {'C': 100, 'gamma': 'scale', 'kernel': 'rbf'}

0.555962 (0.062377) with: {'C': 100, 'gamma': 'scale', 'kernel': 'sigmoid'}

0.799738 (0.049211) with: {'C': 100, 'gamma': 'scale', 'kernel': 'linear'}

0.930235 (0.040305) with: {'C': 100, 'gamma': 'auto', 'kernel': 'poly'}

0.998542 (0.010314) with: {'C': 100, 'gamma': 'auto', 'kernel': 'rbf'}

0.776600 (0.048916) with: {'C': 100, 'gamma': 'auto', 'kernel': 'sigmoid'}

0.799738 (0.049211) with: {'C': 100, 'gamma': 'auto', 'kernel': 'linear'}

0.998750 (0.009233) with: {'C': 50, 'gamma': 'scale', 'kernel': 'poly'}

0.998542 (0.010314) with: {'C': 50, 'gamma': 'scale', 'kernel': 'rbf'}

0.554060 (0.061158) with: {'C': 50, 'gamma': 'scale', 'kernel': 'sigmoid'}

0.799113 (0.049178) with: {'C': 50, 'gamma': 'scale', 'kernel': 'linear'}

0.845284 (0.047565) with: {'C': 50, 'gamma': 'auto', 'kernel': 'poly'}

0.990607 (0.020511) with: {'C': 50, 'gamma': 'auto', 'kernel': 'rbf'}

0.783497 (0.050765) with: {'C': 50, 'gamma': 'auto', 'kernel': 'sigmoid'}

0.799113 (0.049178) with: {'C': 50, 'gamma': 'auto', 'kernel': 'linear'}

0.998750 (0.009233) with: {'C': 10, 'gamma': 'scale', 'kernel': 'poly'}

0.998542 (0.010314) with: {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}

0.587274 (0.063630) with: {'C': 10, 'gamma': 'scale', 'kernel':

'sigmoid'}

0.798905 (0.049779) with: {'C': 10, 'gamma': 'scale', 'kernel': 'linear'}

0.574973 (0.042383) with: {'C': 10, 'gamma': 'auto', 'kernel': 'poly'}

0.872832 (0.041583) with: {'C': 10, 'gamma': 'auto', 'kernel': 'rbf'}

0.753214 (0.053590) with: {'C': 10, 'gamma': 'auto', 'kernel': 'sigmoid'}

0.798905 (0.049779) with: {'C': 10, 'gamma': 'auto', 'kernel': 'linear'}

0.984313 (0.028428) with: {'C': 1.0, 'gamma': 'scale', 'kernel': 'poly'}

0.868245 (0.042195) with: {'C': 1.0, 'gamma': 'scale', 'kernel': 'rbf'}

0.679987 (0.054231) with: {'C': 1.0, 'gamma': 'scale', 'kernel': 'sigmoid'}

0.762824 (0.050019) with: {'C': 1.0, 'gamma': 'scale', 'kernel': 'linear'}

0.392509 (0.009102) with: {'C': 1.0, 'gamma': 'auto', 'kernel': 'poly'}

0.739428 (0.051350) with: {'C': 1.0, 'gamma': 'auto', 'kernel': 'rbf'}

0.686893 (0.045280) with: {'C': 1.0, 'gamma': 'auto', 'kernel': 'sigmoid'}

0.762824 (0.050019) with: {'C': 1.0, 'gamma': 'auto', 'kernel': 'linear'}

0.724029 (0.042676) with: {'C': 0.1, 'gamma': 'scale', 'kernel': 'poly'}

0.428688 (0.025689) with: {'C': 0.1, 'gamma': 'scale', 'kernel': 'rbf'}

0.596946 (0.031869) with: {'C': 0.1, 'gamma': 'scale', 'kernel': 'sigmoid'}

0.737141 (0.044163) with: {'C': 0.1, 'gamma': 'scale', 'kernel': 'linear'}

0.392509 (0.009102) with: {'C': 0.1, 'gamma': 'auto', 'kernel': 'poly'}

0.392509 (0.009102) with: {'C': 0.1, 'gamma': 'auto', 'kernel': 'rbf'}

0.392509 (0.009102) with: {'C': 0.1, 'gamma': 'auto', 'kernel': 'sigmoid'}

0.737141 (0.044163) with: {'C': 0.1, 'gamma': 'auto', 'kernel': 'linear'}

0.392509 (0.009102) with: {'C': 0.01, 'gamma': 'scale', 'kernel': 'poly'}

0.392509 (0.009102) with: {'C': 0.01, 'gamma': 'scale', 'kernel': 'rbf'}

0.392509 (0.009102) with: {'C': 0.01, 'gamma': 'scale', 'kernel': 'sigmoid'}

0.392509 (0.009102) with: {'C': 0.01, 'gamma': 'scale', 'kernel': 'linear'}

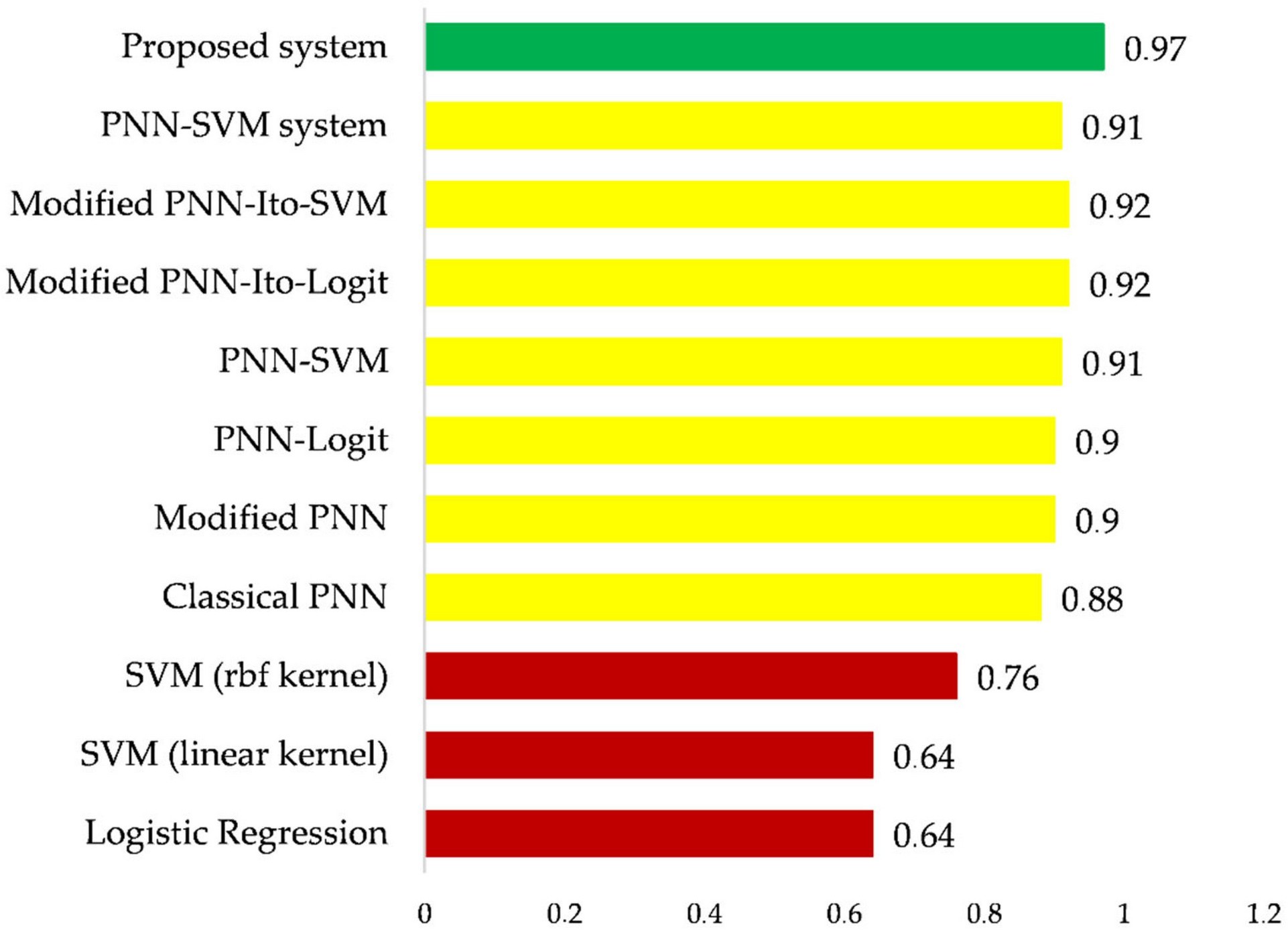
0.392509 (0.009102) with: {'C': 0.01, 'gamma': 'auto', 'kernel': 'poly'}

0.392509 (0.009102) with: {'C': 0.01, 'gamma': 'auto', 'kernel': 'rbf'}

0.392509 (0.009102) with: {'C': 0.01, 'gamma': 'auto', 'kernel': 'sigmoid'}

0.392509 (0.009102) with: {'C': 0.01, 'gamma': 'auto', 'kernel': 'linear'}

Як бачимо найкраща точність є 99.88% при використання наступних гіпер параметрів: Обернена сила регуляризації: 100 та ядро poly (підходить для пошуку складних поліноміних залежностей, не лінійні дані) У статі зазначена куди менша точність що свідчить про не достатню кількість експериментів з боку науковців у цій тематиці. А також про те, що метод опорних векторів показує вищу точність ніж та яка зазначена найвищою у статті.

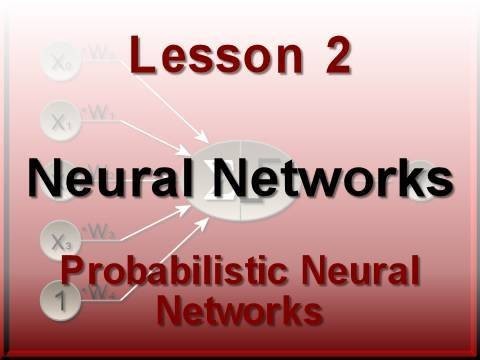


## Ймовірнісна нейронна мережа (Probabilistic neural network)

* Найвища точність у статті 0.88
* Найвища точність під час покращення 0.885

Ймовірнісна нейронна мережа — вид штучних нейронних мереж, який використовує баєсову статистику для виконання певних завдань. Ймовірнісна нейронна мережа була розроблена Дональдом Спехтом. Нижче є наведено [відео](https://youtu.be/uAKu4g7lBxU?si=WS19JaPf_yjfWmBq), яке чудово ілюструє принцип роботи даної моделі. А також пояснює коефіціент згладження і принцип роботи у кожному шарі.

from IPython.display import YouTubeVideo YouTubeVideo('uAKu4g7lBxU')



Підвантажуємо данні і перетворюємо їх у numpy масив, оскільки данна мережа сприймає лише данні у такому форматі.

*# 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)

X\_train = train\_data.drop(20, axis=1).astype(int).to\_numpy() y\_train = train\_data[20].astype(int).to\_numpy() - 1

X\_test = test\_data.drop(20, axis=1).astype(int).to\_numpy() y\_test = test\_data[20].astype(int).to\_numpy() - 1

Код моделі було взято із даного [джерела](https://github.com/verowulf/PNN/blob/master/PNN.ipynb)

from sklearn.model\_selection import KFold

from sklearn.model\_selection import cross\_validate import numpy as np

import math

import numpy as np import math

*# Probabilistic Neural Network with 4 layers*

class PNN(object):

def init (self):

self.L2 = [] *# Layer 2 that holds the patterns*

print('Empty PNN created.')

def train(self, X, y, p=2):

self.n\_ = X.shape[1] *# num of features*

self.p\_ = p *# num of classes*

*# Layer 2 (Pattern): Set up empty lists for each class*

for k in range(self.p\_):

self.L2.append([]) *# Using Python's basic lists because ndarray cannot append empty arrays*

*# Also perhaps we might have to*

*input different data types*

*# Enter patterns into Layer 2*

for i in range(X.shape[0]): self.L2[y[i]].append(X[i])

self.L2 = np.array(self.L2) *# Change to ndarray for speed (Is this faster?)*

print('PNN with %d classes trained.' % self.p\_) def crossValidate(self, X, y, sigma=0.5):

result = self.predict(X, sigma)

num\_correct = sum(result[:, 0] == y)

print('Cross validation accuracy with sigma %.2f: %.1f%%' % (sigma, num\_correct/len(y) \* 100))

def predict(self, X, sigma=0.5): m = X.shape[0]

accL3 = np.zeros((m, self.p\_)) accL4 = np.zeros(m)

self.sigma\_ = sigma *# smoothing parameter, not standard deviation*

self.C1\_ = 2 \* self.sigma\_\*\*2

C2\_ = (math.sqrt(2\*math.pi) \* self.sigma\_) \*\* (- self.n\_)

*# Layer 1 (Input): x*

for i in range(m): x = X[i]

*# Layer 3 (Averaging): for each class*

self.L3\_ = np.zeros(self.p\_) for k in range(self.p\_):

for ki in range(len(self.L2[k])):

self.L3\_[k] += self.\_activation(x, self.L2[k][ki]) self.L3\_[k] /= len(self.L2[k])

*# Multiply constant* self.L3\_[k] \*= C2\_ accL3[i][k] = self.L3\_[k]

*# Layer 4 (Output/Decision): Maxing* self.L4\_ = self.L3\_.argmax() accL4[i] = self.L4\_

return np.column\_stack((accL4, accL3)) def \_activation(self, x, w):

diff = x - w

return math.exp( - np.dot(diff, diff) / self.C1\_ )

pnn = PNN()

pnn.train(X\_train, y\_train, p=4)

begin = 0.01

end = 1.00

step = 0.01

s = begin

while s < end+step:

pnn.crossValidate(X\_test, y\_test, sigma=s) s += step

Empty PNN created.

PNN with 4 classes trained.

<ipython-input-53-e9414b75d14d>:27: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or- tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.

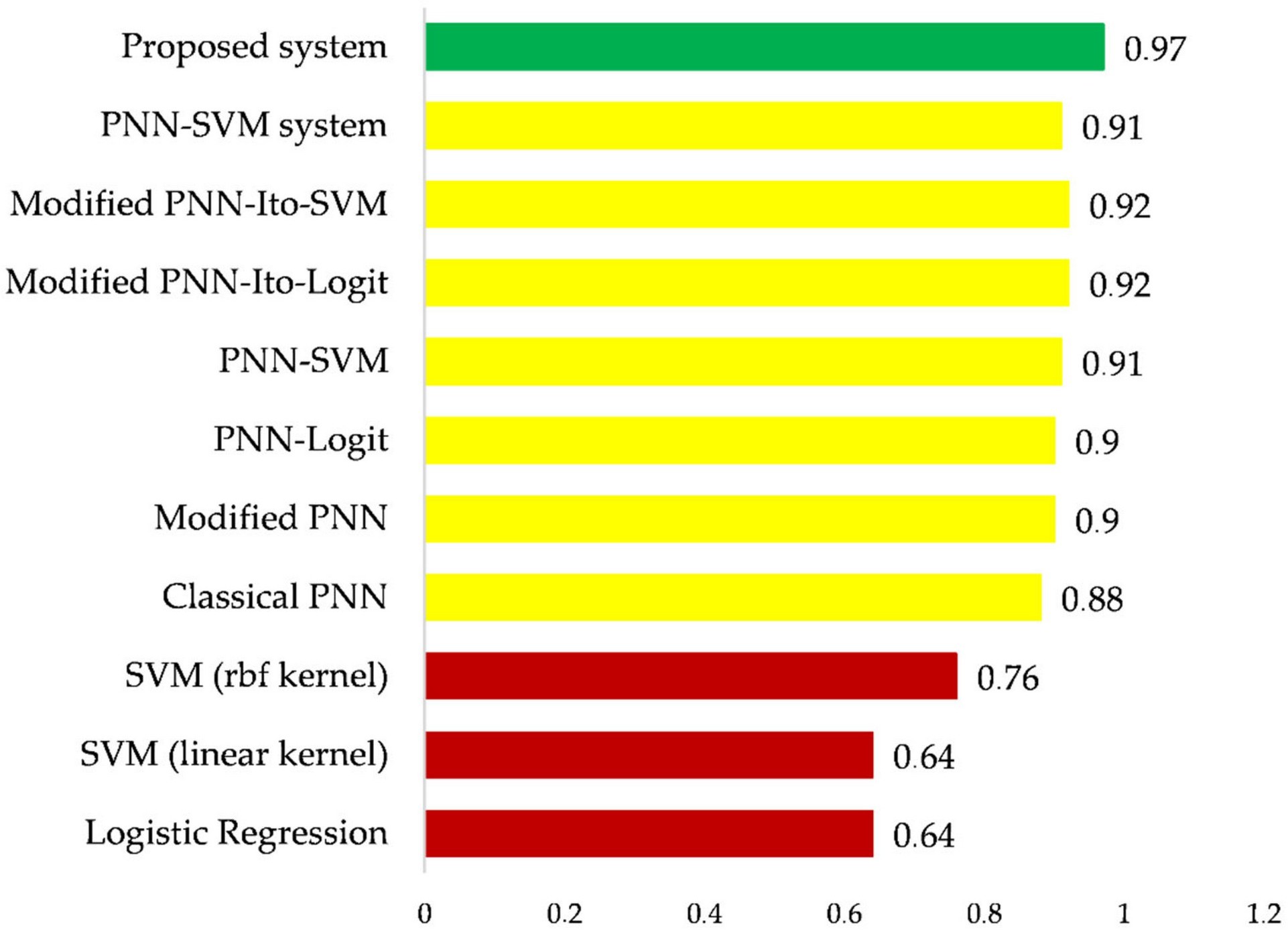
self.L2 = np.array(self.L2) # Change to ndarray for speed (Is this faster?)

Cross validation accuracy with sigma 0.01: 34.4% Cross validation accuracy with sigma 0.02: 34.4% Cross validation accuracy with sigma 0.03: 34.4% Cross validation accuracy with sigma 0.04: 88.5%

|  |
| --- |
| Cross validation accuracy with sigma 0.05: 88.5% |
| Cross validation accuracy with sigma 0.06: 88.5% |
| Cross validation accuracy with sigma 0.07: 88.5% |
| Cross validation accuracy with sigma 0.08: 88.5% |
| Cross validation accuracy with sigma 0.09: 88.5% |
| Cross validation accuracy with sigma 0.10: 88.5% |
| Cross validation accuracy with sigma 0.11: 88.5% |
| Cross validation accuracy with sigma 0.12: 88.5% |
| Cross validation accuracy with sigma 0.13: 88.5% |
| Cross validation accuracy with sigma 0.14: 88.5% |
| Cross validation accuracy with sigma 0.15: 88.5% |
| Cross validation accuracy with sigma 0.16: 88.5% |
| Cross validation accuracy with sigma 0.17: 88.5% |
| Cross validation accuracy with sigma 0.18: 88.5% |
| Cross validation accuracy with sigma 0.19: 88.5% |
| Cross validation accuracy with sigma 0.20: 88.5% |
| Cross validation accuracy with sigma 0.21: 88.5% |
| Cross validation accuracy with sigma 0.22: 88.5% |
| Cross validation accuracy with sigma 0.23: 88.5% |
| Cross validation accuracy with sigma 0.24: 88.5% |
| Cross validation accuracy with sigma 0.25: 88.5% |
| Cross validation accuracy with sigma 0.26: 88.5% |
| Cross validation accuracy with sigma 0.27: 88.5% |
| Cross validation accuracy with sigma 0.28: 88.5% |
| Cross validation accuracy with sigma 0.29: 88.5% |
| Cross validation accuracy with sigma 0.30: 88.5% |
| Cross validation accuracy with sigma 0.31: 88.5% |
| Cross validation accuracy with sigma 0.32: 88.5% |
| Cross validation accuracy with sigma 0.33: 88.5% |
| Cross validation accuracy with sigma 0.34: 88.5% |
| Cross validation accuracy with sigma 0.35: 88.5% |
| Cross validation accuracy with sigma 0.36: 88.5% |
| Cross validation accuracy with sigma 0.37: 88.5% |
| Cross validation accuracy with sigma 0.38: 88.5% |
| Cross validation accuracy with sigma 0.39: 88.5% |
| Cross validation accuracy with sigma 0.40: 88.5% |
| Cross validation accuracy with sigma 0.41: 88.5% |
| Cross validation accuracy with sigma 0.42: 88.5% |
| Cross validation accuracy with sigma 0.43: 88.5% |
| Cross validation accuracy with sigma 0.44: 88.5% |
| Cross validation accuracy with sigma 0.45: 88.5% |
| Cross validation accuracy with sigma 0.46: 88.5% |
| Cross validation accuracy with sigma 0.47: 88.5% |
| Cross validation accuracy with sigma 0.48: 88.5% |
| Cross validation accuracy with sigma 0.49: 88.5% |
| Cross validation accuracy with sigma 0.50: 88.5% |
| Cross validation accuracy with sigma 0.51: 88.5% |
| Cross validation accuracy with sigma 0.52: 88.5% |
| Cross validation accuracy with sigma 0.53: 88.5% |

|  |
| --- |
| Cross validation accuracy with sigma 0.54: 88.5% |
| Cross validation accuracy with sigma 0.55: 88.5% |
| Cross validation accuracy with sigma 0.56: 88.5% |
| Cross validation accuracy with sigma 0.57: 88.5% |
| Cross validation accuracy with sigma 0.58: 88.5% |
| Cross validation accuracy with sigma 0.59: 88.5% |
| Cross validation accuracy with sigma 0.60: 87.5% |
| Cross validation accuracy with sigma 0.61: 87.5% |
| Cross validation accuracy with sigma 0.62: 87.5% |
| Cross validation accuracy with sigma 0.63: 87.5% |
| Cross validation accuracy with sigma 0.64: 87.5% |
| Cross validation accuracy with sigma 0.65: 87.5% |
| Cross validation accuracy with sigma 0.66: 87.5% |
| Cross validation accuracy with sigma 0.67: 86.5% |
| Cross validation accuracy with sigma 0.68: 86.5% |
| Cross validation accuracy with sigma 0.69: 86.5% |
| Cross validation accuracy with sigma 0.70: 85.4% |
| Cross validation accuracy with sigma 0.71: 85.4% |
| Cross validation accuracy with sigma 0.72: 85.4% |
| Cross validation accuracy with sigma 0.73: 85.4% |
| Cross validation accuracy with sigma 0.74: 85.4% |
| Cross validation accuracy with sigma 0.75: 85.4% |
| Cross validation accuracy with sigma 0.76: 85.4% |
| Cross validation accuracy with sigma 0.77: 85.4% |
| Cross validation accuracy with sigma 0.78: 85.4% |
| Cross validation accuracy with sigma 0.79: 85.4% |
| Cross validation accuracy with sigma 0.80: 85.4% |
| Cross validation accuracy with sigma 0.81: 85.4% |
| Cross validation accuracy with sigma 0.82: 85.4% |
| Cross validation accuracy with sigma 0.83: 85.4% |
| Cross validation accuracy with sigma 0.84: 85.4% |
| Cross validation accuracy with sigma 0.85: 85.4% |
| Cross validation accuracy with sigma 0.86: 85.4% |
| Cross validation accuracy with sigma 0.87: 84.4% |
| Cross validation accuracy with sigma 0.88: 84.4% |
| Cross validation accuracy with sigma 0.89: 84.4% |
| Cross validation accuracy with sigma 0.90: 82.3% |
| Cross validation accuracy with sigma 0.91: 82.3% |
| Cross validation accuracy with sigma 0.92: 82.3% |
| Cross validation accuracy with sigma 0.93: 81.2% |
| Cross validation accuracy with sigma 0.94: 81.2% |
| Cross validation accuracy with sigma 0.95: 81.2% |
| Cross validation accuracy with sigma 0.96: 81.2% |
| Cross validation accuracy with sigma 0.97: 81.2% |
| Cross validation accuracy with sigma 0.98: 80.2% |
| Cross validation accuracy with sigma 0.99: 77.1% |
| Cross validation accuracy with sigma 1.00: 77.1% |

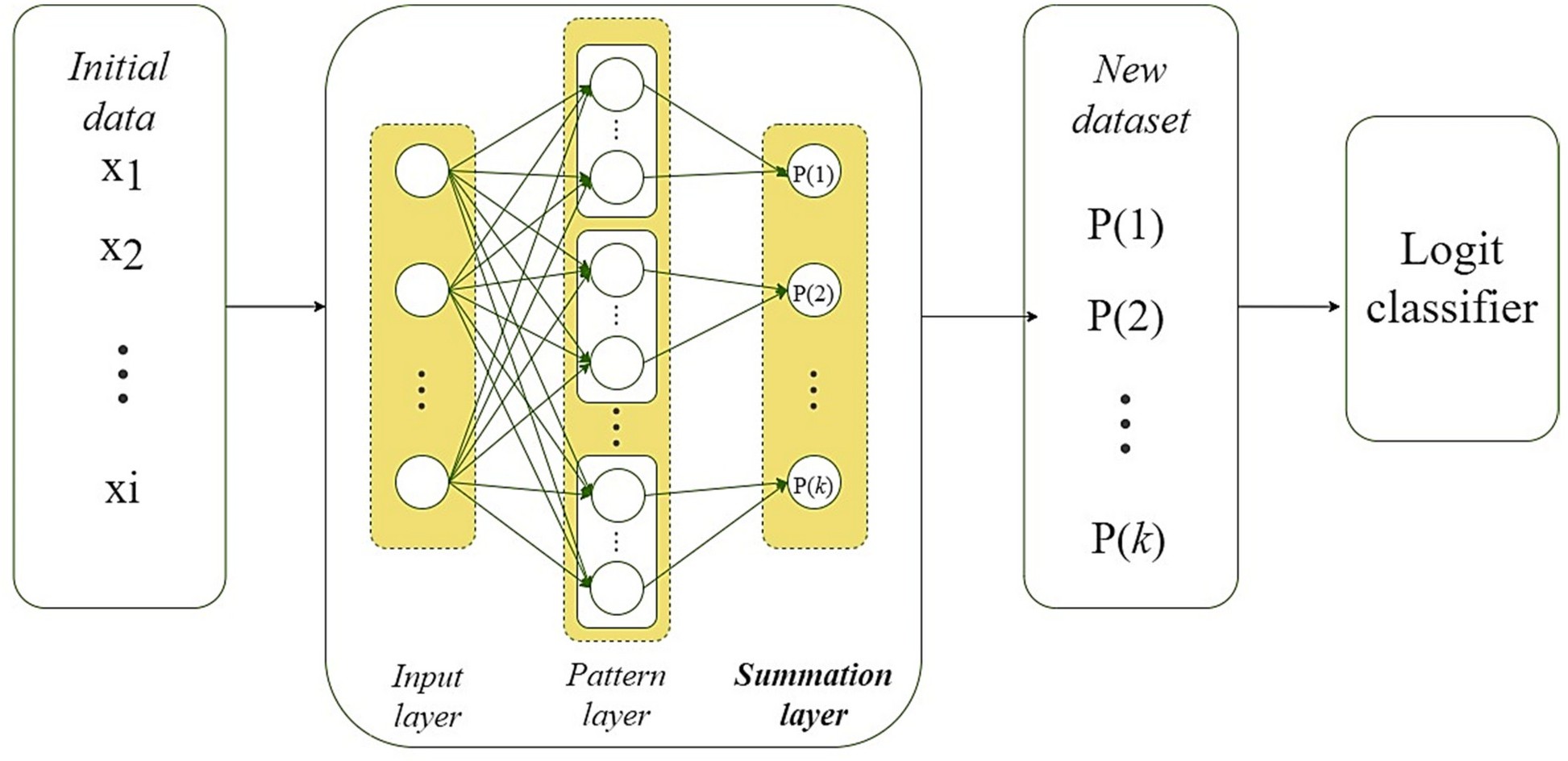
Як бачимо із крос валідації моделі у проміжку від [0.04, 0.59] коефіціент згладжування сігма показує найвищий результат 88.5%, наведенний у статті результат має дещо нищу точність 88%, але це напевно помилка округлення даних зі сторони науковців.



## Ймовірнісна нейронна мережа із використанням логістичної регресії

* Найвища точність у сттаті 90%
* Найвища точність підчас покращення результатів 64.58%

Данна модель буде складатися із ймовірнісна нейронної мережі(PNN) та використанням логістичної регресії для покращення наявних результатів.



1. Спершу буде використано PNN(ймовірнісну нейронну мережу) для навчання і формування нового датасету.
2. Другим кроком буде натренувати логістичну регресію на наявних результатах і порівняти поліпшення точності.

Повторюємо всі кори які були виконанні у попередньому методі із начання PNN.

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

Mounted at /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)

X\_train = train\_data.drop(20, axis=1).astype(int).to\_numpy() y\_train = train\_data[20].astype(int).to\_numpy() - 1

X\_test = test\_data.drop(20, axis=1).astype(int).to\_numpy() y\_test = test\_data[20].astype(int).to\_numpy() - 1

import numpy as np import math

*# Probabilistic Neural Network with 4 layers*

class PNN(object):

def init (self):

self.L2 = [] *# Layer 2 that holds the patterns*

print('Empty PNN created.')

def train(self, X, y, p=2):

self.n\_ = X.shape[1] *# num of features*

self.p\_ = p *# num of classes*

*# Layer 2 (Pattern): Set up empty lists for each class*

for k in range(self.p\_):

self.L2.append([]) *# Using Python's basic lists because ndarray cannot append empty arrays*

*# Also perhaps we might have to*

*input different data types*

*# Enter patterns into Layer 2*

for i in range(X.shape[0]): self.L2[y[i]].append(X[i])

self.L2 = np.array(self.L2) *# Change to ndarray for speed (Is this faster?)*

print('PNN with %d classes trained.' % self.p\_) def crossValidate(self, X, y, sigma=0.5):

result = self.predict(X, sigma)

num\_correct = sum(result[:, 0] == y)

print('Cross validation accuracy with sigma %.2f: %.1f%%' % (sigma, num\_correct/len(y) \* 100))

return result

def predict(self, X, sigma=0.5): m = X.shape[0]

accL3 = np.zeros((m, self.p\_)) accL4 = np.zeros(m)

self.sigma\_ = sigma *# smoothing parameter, not standard deviation*

self.C1\_ = 2 \* self.sigma\_\*\*2

C2\_ = (math.sqrt(2\*math.pi) \* self.sigma\_) \*\* (- self.n\_)

*# Layer 1 (Input): x*

for i in range(m): x = X[i]

*# Layer 3 (Averaging): for each class*

self.L3\_ = np.zeros(self.p\_) for k in range(self.p\_):

for ki in range(len(self.L2[k])):

self.L3\_[k] += self.\_activation(x, self.L2[k][ki]) self.L3\_[k] /= len(self.L2[k])

*# Multiply constant* self.L3\_[k] \*= C2\_ accL3[i][k] = self.L3\_[k]

*# Layer 4 (Output/Decision): Maxing* self.L4\_ = self.L3\_.argmax() accL4[i] = self.L4\_

return np.column\_stack((accL4, accL3)) def \_activation(self, x, w):

diff = x - w

return math.exp( - np.dot(diff, diff) / self.C1\_ )

*# Normalize to unit length: [0, 1] # X must be ndarray*

def Normalize(X):

x\_max = X.max(axis=0) x\_min = X.min(axis=0)

return (X - x\_min) / (x\_max - x\_min)

pnn = PNN()

pnn.train(X\_train, y\_train, p = 4)

begin = 0.01

end = 1.00

step = 0.01

s = begin

while s < end+step:

pnn.crossValidate(X\_test, y\_test, sigma=s) s += step

Empty PNN created.

PNN with 4 classes trained.

Cross validation accuracy with sigma 0.01: 34.4%

<ipython-input-8-179a21a4f331>:23: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.

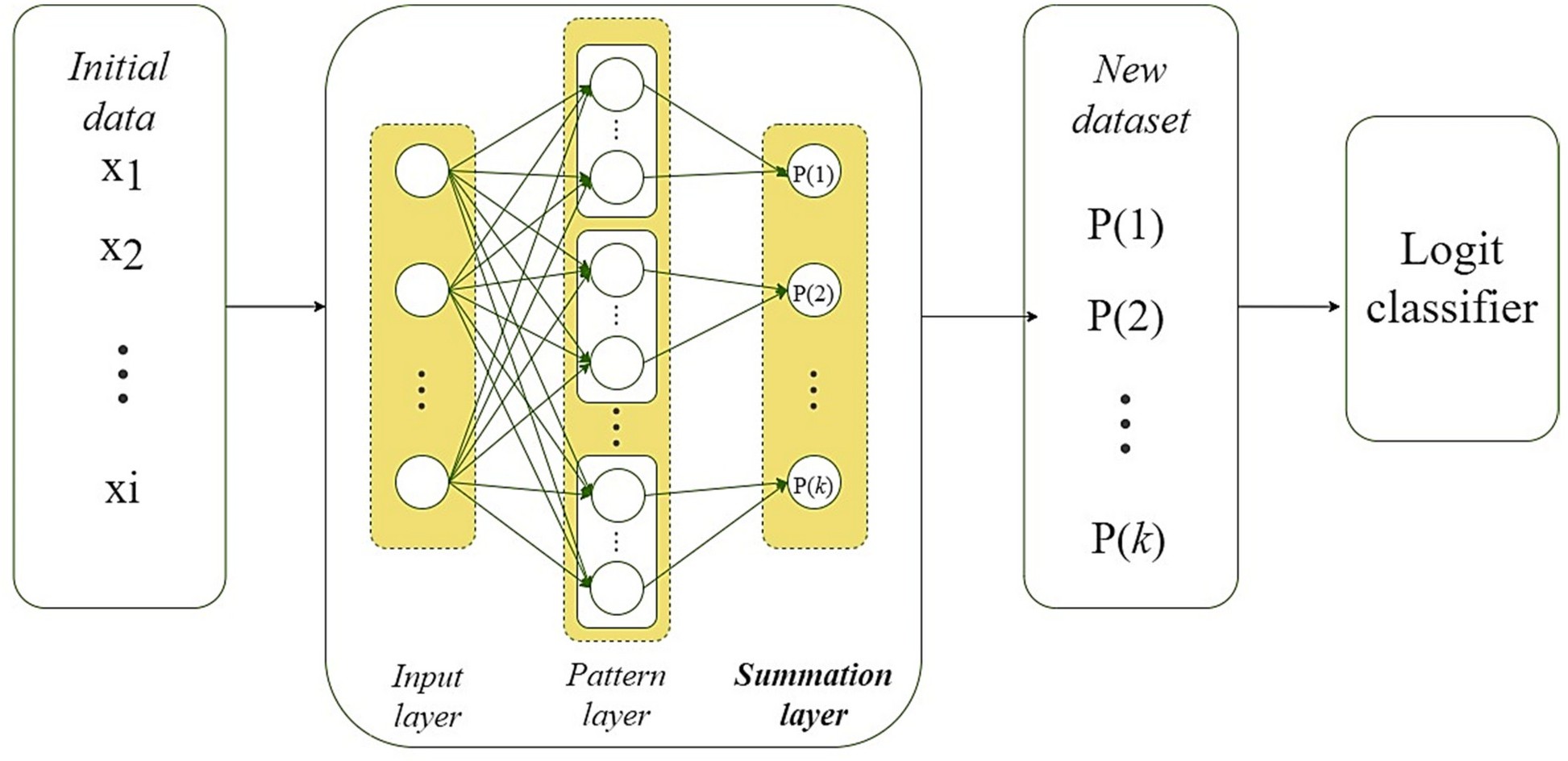
self.L2 = np.array(self.L2) # Change to ndarray for speed (Is this faster?)

Cross validation accuracy with sigma 0.02: 34.4% Cross validation accuracy with sigma 0.03: 34.4% Cross validation accuracy with sigma 0.04: 88.5% Cross validation accuracy with sigma 0.05: 88.5% Cross validation accuracy with sigma 0.06: 88.5%

|  |
| --- |
| Cross validation accuracy with sigma 0.07: 88.5% |
| Cross validation accuracy with sigma 0.08: 88.5% |
| Cross validation accuracy with sigma 0.09: 88.5% |
| Cross validation accuracy with sigma 0.10: 88.5% |
| Cross validation accuracy with sigma 0.11: 88.5% |
| Cross validation accuracy with sigma 0.12: 88.5% |
| Cross validation accuracy with sigma 0.13: 88.5% |
| Cross validation accuracy with sigma 0.14: 88.5% |
| Cross validation accuracy with sigma 0.15: 88.5% |
| Cross validation accuracy with sigma 0.16: 88.5% |
| Cross validation accuracy with sigma 0.17: 88.5% |
| Cross validation accuracy with sigma 0.18: 88.5% |
| Cross validation accuracy with sigma 0.19: 88.5% |
| Cross validation accuracy with sigma 0.20: 88.5% |
| Cross validation accuracy with sigma 0.21: 88.5% |
| Cross validation accuracy with sigma 0.22: 88.5% |
| Cross validation accuracy with sigma 0.23: 88.5% |
| Cross validation accuracy with sigma 0.24: 88.5% |
| Cross validation accuracy with sigma 0.25: 88.5% |
| Cross validation accuracy with sigma 0.26: 88.5% |
| Cross validation accuracy with sigma 0.27: 88.5% |
| Cross validation accuracy with sigma 0.28: 88.5% |
| Cross validation accuracy with sigma 0.29: 88.5% |
| Cross validation accuracy with sigma 0.30: 88.5% |
| Cross validation accuracy with sigma 0.31: 88.5% |
| Cross validation accuracy with sigma 0.32: 88.5% |
| Cross validation accuracy with sigma 0.33: 88.5% |
| Cross validation accuracy with sigma 0.34: 88.5% |
| Cross validation accuracy with sigma 0.35: 88.5% |
| Cross validation accuracy with sigma 0.36: 88.5% |
| Cross validation accuracy with sigma 0.37: 88.5% |
| Cross validation accuracy with sigma 0.38: 88.5% |
| Cross validation accuracy with sigma 0.39: 88.5% |
| Cross validation accuracy with sigma 0.40: 88.5% |
| Cross validation accuracy with sigma 0.41: 88.5% |
| Cross validation accuracy with sigma 0.42: 88.5% |
| Cross validation accuracy with sigma 0.43: 88.5% |
| Cross validation accuracy with sigma 0.44: 88.5% |
| Cross validation accuracy with sigma 0.45: 88.5% |
| Cross validation accuracy with sigma 0.46: 88.5% |
| Cross validation accuracy with sigma 0.47: 88.5% |
| Cross validation accuracy with sigma 0.48: 88.5% |
| Cross validation accuracy with sigma 0.49: 88.5% |
| Cross validation accuracy with sigma 0.50: 88.5% |
| Cross validation accuracy with sigma 0.51: 88.5% |
| Cross validation accuracy with sigma 0.52: 88.5% |
| Cross validation accuracy with sigma 0.53: 88.5% |
| Cross validation accuracy with sigma 0.54: 88.5% |
| Cross validation accuracy with sigma 0.55: 88.5% |

|  |
| --- |
| Cross validation accuracy with sigma 0.56: 88.5% |
| Cross validation accuracy with sigma 0.57: 88.5% |
| Cross validation accuracy with sigma 0.58: 88.5% |
| Cross validation accuracy with sigma 0.59: 88.5% |
| Cross validation accuracy with sigma 0.60: 87.5% |
| Cross validation accuracy with sigma 0.61: 87.5% |
| Cross validation accuracy with sigma 0.62: 87.5% |
| Cross validation accuracy with sigma 0.63: 87.5% |
| Cross validation accuracy with sigma 0.64: 87.5% |
| Cross validation accuracy with sigma 0.65: 87.5% |
| Cross validation accuracy with sigma 0.66: 87.5% |
| Cross validation accuracy with sigma 0.67: 86.5% |
| Cross validation accuracy with sigma 0.68: 86.5% |
| Cross validation accuracy with sigma 0.69: 86.5% |
| Cross validation accuracy with sigma 0.70: 85.4% |
| Cross validation accuracy with sigma 0.71: 85.4% |
| Cross validation accuracy with sigma 0.72: 85.4% |
| Cross validation accuracy with sigma 0.73: 85.4% |
| Cross validation accuracy with sigma 0.74: 85.4% |
| Cross validation accuracy with sigma 0.75: 85.4% |
| Cross validation accuracy with sigma 0.76: 85.4% |
| Cross validation accuracy with sigma 0.77: 85.4% |
| Cross validation accuracy with sigma 0.78: 85.4% |
| Cross validation accuracy with sigma 0.79: 85.4% |
| Cross validation accuracy with sigma 0.80: 85.4% |
| Cross validation accuracy with sigma 0.81: 85.4% |
| Cross validation accuracy with sigma 0.82: 85.4% |
| Cross validation accuracy with sigma 0.83: 85.4% |
| Cross validation accuracy with sigma 0.84: 85.4% |
| Cross validation accuracy with sigma 0.85: 85.4% |
| Cross validation accuracy with sigma 0.86: 85.4% |
| Cross validation accuracy with sigma 0.87: 84.4% |
| Cross validation accuracy with sigma 0.88: 84.4% |
| Cross validation accuracy with sigma 0.89: 84.4% |
| Cross validation accuracy with sigma 0.90: 82.3% |
| Cross validation accuracy with sigma 0.91: 82.3% |
| Cross validation accuracy with sigma 0.92: 82.3% |
| Cross validation accuracy with sigma 0.93: 81.2% |
| Cross validation accuracy with sigma 0.94: 81.2% |
| Cross validation accuracy with sigma 0.95: 81.2% |
| Cross validation accuracy with sigma 0.96: 81.2% |
| Cross validation accuracy with sigma 0.97: 81.2% |
| Cross validation accuracy with sigma 0.98: 80.2% |
| Cross validation accuracy with sigma 0.99: 77.1% |
| Cross validation accuracy with sigma 1.00: 77.1% |

Дістаємо ймовірності для кожного класу із PNN і формуємо новий датасет для навчання логістичної регресії. Використовуємо ймовірності із нейронної мережі, як новий датасет для лінійної регресії:



X\_logit\_train = pnn.crossValidate(X\_train, y\_train, sigma=0.32) X\_logit\_test = pnn.crossValidate(X\_test, y\_test, sigma=0.32)

X\_logit\_train = X\_logit\_train[:,:] X\_logit\_test = X\_logit\_test[:,:]

Cross validation accuracy with sigma 0.32: 100.0% Cross validation accuracy with sigma 0.32: 88.5%

Виконуємо повний перебір параметрів для пошуку найкращого гіперпараметра для моделі

from sklearn.datasets import load\_iris

from sklearn.linear\_model import LogisticRegression

solvers = ['lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga']

penalty = ['l2']

c\_values = [100, 10, 1.0, 0.1, 0.01]

for solver in solvers:

for c\_value in c\_values:

clf = LogisticRegression(solver = solver, C = c\_value, max\_iter=10000)

print(clf.fit(X\_logit\_train, y\_train)) print(clf.score(X\_logit\_test, y\_test))

LogisticRegression(C=100, max\_iter=10000) 0.6041666666666666

LogisticRegression(C=10, max\_iter=10000) 0.6041666666666666

LogisticRegression(max\_iter=10000) 0.6041666666666666

LogisticRegression(C=0.1, max\_iter=10000) 0.6041666666666666

LogisticRegression(C=0.01, max\_iter=10000) 0.6041666666666666

LogisticRegression(C=100, max\_iter=10000, solver='liblinear') 0.6458333333333334

LogisticRegression(C=10, max\_iter=10000, solver='liblinear') 0.6458333333333334

LogisticRegression(max\_iter=10000, solver='liblinear') 0.6458333333333334

LogisticRegression(C=0.1, max\_iter=10000, solver='liblinear') 0.6458333333333334

LogisticRegression(C=0.01, max\_iter=10000, solver='liblinear') 0.6458333333333334

LogisticRegression(C=100, max\_iter=10000, solver='newton-cg') 0.6041666666666666

LogisticRegression(C=10, max\_iter=10000, solver='newton-cg') 0.6041666666666666

LogisticRegression(max\_iter=10000, solver='newton-cg') 0.6041666666666666

LogisticRegression(C=0.1, max\_iter=10000, solver='newton-cg') 0.6041666666666666

LogisticRegression(C=0.01, max\_iter=10000, solver='newton-cg') 0.6041666666666666

LogisticRegression(C=100, max\_iter=10000, solver='newton-cholesky') 0.6041666666666666

LogisticRegression(C=10, max\_iter=10000, solver='newton-cholesky') 0.6041666666666666

LogisticRegression(max\_iter=10000, solver='newton-cholesky') 0.6041666666666666

LogisticRegression(C=0.1, max\_iter=10000, solver='newton-cholesky') 0.6041666666666666

LogisticRegression(C=0.01, max\_iter=10000, solver='newton-cholesky') 0.6041666666666666

LogisticRegression(C=100, max\_iter=10000, solver='sag') 0.6041666666666666

LogisticRegression(C=10, max\_iter=10000, solver='sag') 0.6041666666666666

LogisticRegression(max\_iter=10000, solver='sag') 0.6041666666666666

LogisticRegression(C=0.1, max\_iter=10000, solver='sag') 0.6041666666666666

LogisticRegression(C=0.01, max\_iter=10000, solver='sag') 0.6041666666666666

LogisticRegression(C=100, max\_iter=10000, solver='saga') 0.6041666666666666

LogisticRegression(C=10, max\_iter=10000, solver='saga')

0.6041666666666666

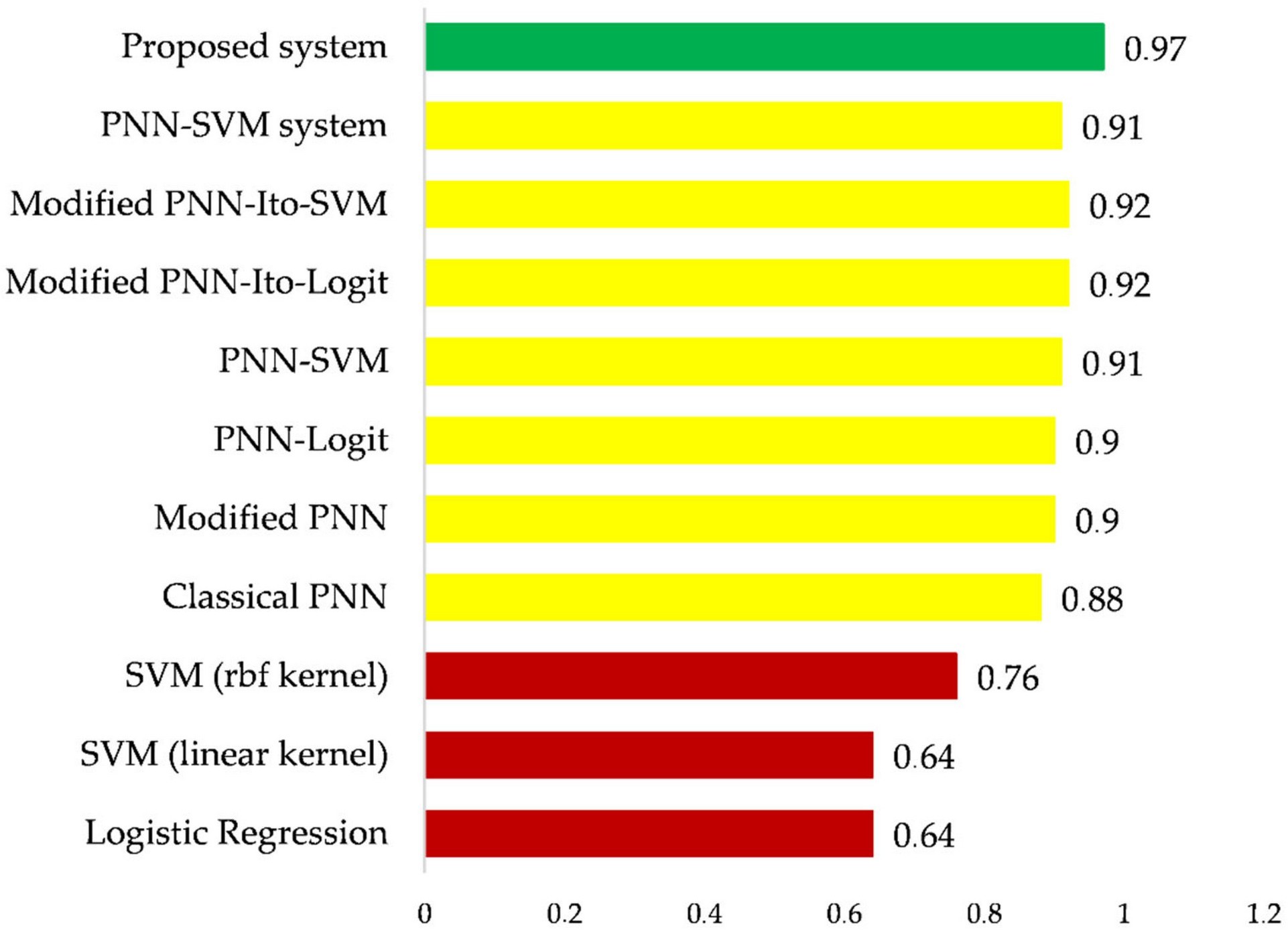
LogisticRegression(max\_iter=10000, solver='saga') 0.6041666666666666

LogisticRegression(C=0.1, max\_iter=10000, solver='saga') 0.6041666666666666

LogisticRegression(C=0.01, max\_iter=10000, solver='saga') 0.6041666666666666

Найвища точність отримана у даному експерименті є 64.58%, що є менше ніж наведенно у статті. На жаль, або використанні мною моделі є іншими ніж у дані статті, або автори могли помилитися при наведенні результатів або проведення експериментів (наприклад об'єднати тестову вибірку із тренувальною, що сильно може підвищити точнітсь моделі).

Що є доречі, наведено нижче (точінсть 97.7%)



*# example of grid searching key hyperparametres for logistic regression*

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

from sklearn.linear\_model import LogisticRegression

X = np.concatenate((X\_logit\_train, X\_logit\_test)) y = np.concatenate((y\_train, y\_test))

model = LogisticRegression()

solvers = ['lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga']

penalty = ['l2']

c\_values = [100, 10, 1.0, 0.1, 0.01]

*# define grid search*

grid = dict(solver=solvers,penalty=penalty,C=c\_values) 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: 0.977017 using {'C': 100, 'penalty': 'l2', 'solver': 'lbfgs'}

0.977017 (0.017632) with: {'C': 100, 'penalty': 'l2', 'solver': 'lbfgs'}

0.977017 (0.017632) with: {'C': 100, 'penalty': 'l2', 'solver': 'liblinear'}

0.977017 (0.017632) with: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}

0.977017 (0.017632) with: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cholesky'}

0.977017 (0.017632) with: {'C': 100, 'penalty': 'l2', 'solver': 'sag'}

0.977017 (0.017632) with: {'C': 100, 'penalty': 'l2', 'solver': 'saga'}

0.977017 (0.017632) with: {'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}

0.977017 (0.017632) with: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}

0.977017 (0.017632) with: {'C': 10, 'penalty': 'l2', 'solver': 'newton-cg'}

0.977017 (0.017632) with: {'C': 10, 'penalty': 'l2', 'solver': 'newton-cholesky'}

0.977017 (0.017632) with: {'C': 10, 'penalty': 'l2', 'solver': 'sag'}

0.977017 (0.017632) with: {'C': 10, 'penalty': 'l2', 'solver': 'saga'}

0.977017 (0.017632) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs'}

0.885208 (0.017400) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'liblinear'}

0.977017 (0.017632) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cg'}

0.977017 (0.017632) with: {'C': 1.0, 'penalty': 'l2', 'solver':

'newton-cholesky'}

0.977017 (0.017632) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'sag'}

0.977017 (0.017632) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'saga'}

0.894065 (0.030696) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}

0.661822 (0.010879) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}

0.894065 (0.030696) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}

0.653480 (0.015579) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cholesky'}

0.896565 (0.035459) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'sag'}

0.899065 (0.038607) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'saga'}

0.653480 (0.015579) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}

0.661822 (0.010879) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'liblinear'}

0.653480 (0.015579) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}

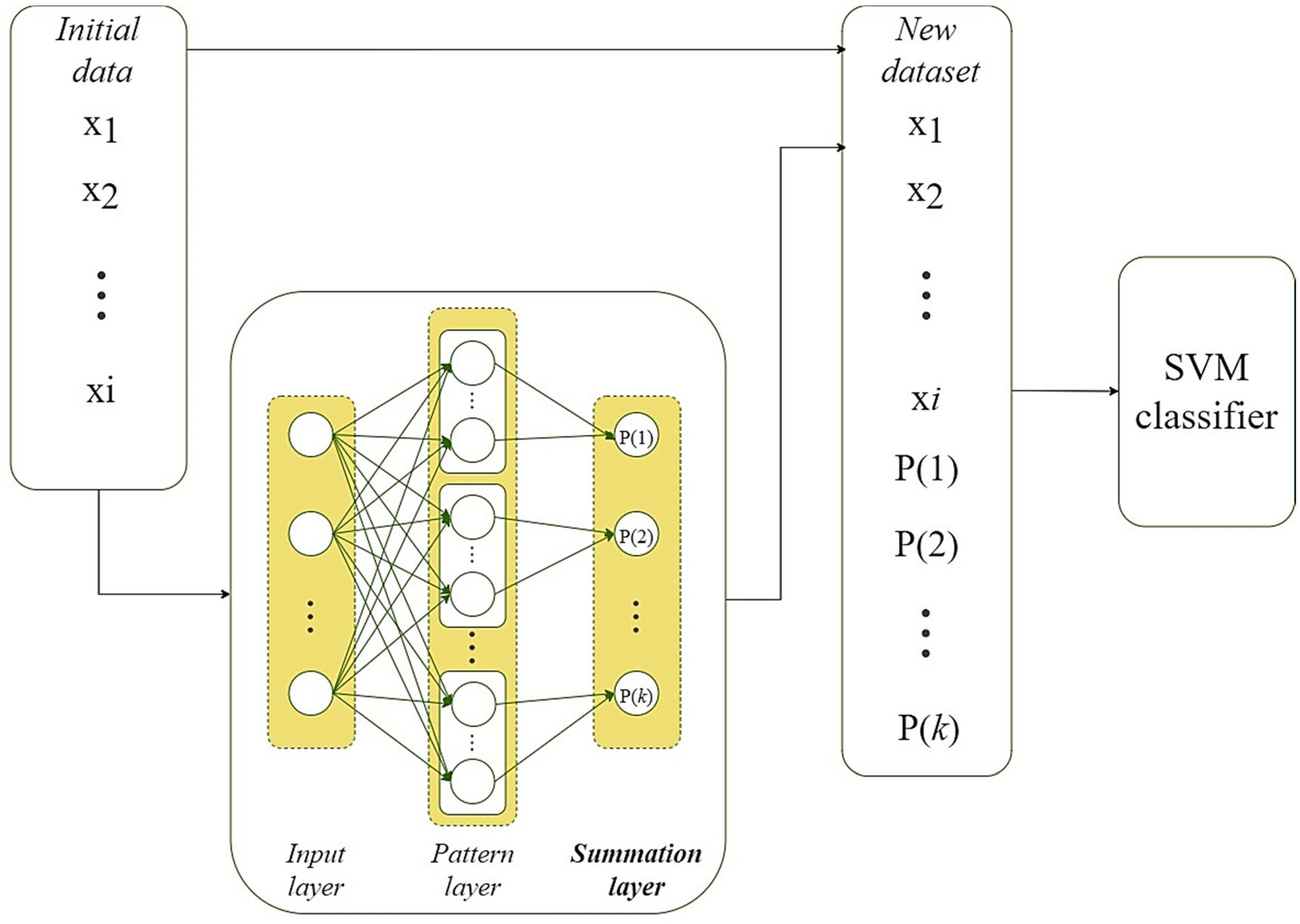
0.653480 (0.015579) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cholesky'}

0.653480 (0.015579) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'sag'}

0.653480 (0.015579) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'saga'}

## PNN-SVM Classifier

Наступним експериментом є об'єднання датасет із ймовірностями із PNN для створення нового датасету і подачі його до SVM. Як це зображенно на картинці.



Об'єднуємо датасет із ймовірностями

X\_svm\_train = pnn.crossValidate(X\_train, y\_train, sigma=0.5) X\_svm\_test = pnn.crossValidate(X\_test, y\_test, sigma=0.5)

*# Take only probability* X\_svm\_train = X\_svm\_train[:,1:] X\_svm\_test = X\_svm\_test[:,1:]

X\_svm\_train = np.hstack((X\_train, X\_svm\_train)) X\_svm\_test = np.hstack((X\_test, X\_svm\_test))

Cross validation accuracy with sigma 0.50: 100.0% Cross validation accuracy with sigma 0.50: 88.5%

Перевіряємо розмірнітсь даних

X\_svm\_train.shape (383, 24)

X\_svm\_test.shape

(96, 24)

Тренуємо для прикладу з початковими параметрами SVM.

from sklearn import svm

regr = svm.SVR(kernel='linear') print(regr.fit(X\_svm\_train, y\_train)) print(regr.score(X\_svm\_test, y\_test))

SVR(kernel='linear') 0.4827743549244463

У результаті отримємо не високу точність(меншу ніж у статті). Проте якщо виконати пошук гіперпараметрів моделі. То можна отримати кращі результати. Що було зроблене нижче:

kernels = ['poly', 'rbf', 'sigmoid', 'linear']

c\_values = [100, 50, 10, 1.0, 0.1, 0.01]

gammas = ['scale', 'auto']

for kernel in kernels:

for c\_value in c\_values: for gamma in gammas:

clf = svm.SVC(kernel = kernel, C = c\_value, gamma=gamma) print(clf.fit(X\_svm\_train, y\_train)) print(clf.score(X\_svm\_test, y\_test))

SVC(C=100, kernel='poly') 1.0

SVC(C=100, gamma='auto', kernel='poly') 0.9270833333333334

SVC(C=50, kernel='poly') 1.0

SVC(C=50, gamma='auto', kernel='poly') 0.7708333333333334

SVC(C=10, kernel='poly') 1.0

SVC(C=10, gamma='auto', kernel='poly') 0.34375

SVC(kernel='poly') 0.9791666666666666

SVC(gamma='auto', kernel='poly') 0.34375

SVC(C=0.1, kernel='poly') 0.6666666666666666

SVC(C=0.1, gamma='auto', kernel='poly') 0.34375

SVC(C=0.01, kernel='poly') 0.34375

SVC(C=0.01, gamma='auto', kernel='poly') 0.34375

SVC(C=100) 1.0

SVC(C=100, gamma='auto') 0.9791666666666666 SVC(C=50)

1.0

SVC(C=50, gamma='auto') 0.96875

SVC(C=10) 1.0

SVC(C=10, gamma='auto') 0.875

SVC() 0.8958333333333334

SVC(gamma='auto') 0.6875

SVC(C=0.1) 0.34375

SVC(C=0.1, gamma='auto') 0.34375

SVC(C=0.01) 0.34375

SVC(C=0.01, gamma='auto') 0.34375

SVC(C=100, kernel='sigmoid') 0.53125

SVC(C=100, gamma='auto', kernel='sigmoid') 0.8125

SVC(C=50, kernel='sigmoid') 0.5

SVC(C=50, gamma='auto', kernel='sigmoid') 0.7708333333333334

SVC(C=10, kernel='sigmoid') 0.5520833333333334

SVC(C=10, gamma='auto', kernel='sigmoid') 0.7708333333333334

SVC(kernel='sigmoid') 0.6145833333333334

SVC(gamma='auto', kernel='sigmoid') 0.6770833333333334

SVC(C=0.1, kernel='sigmoid') 0.5

SVC(C=0.1, gamma='auto', kernel='sigmoid') 0.34375

SVC(C=0.01, kernel='sigmoid') 0.34375

SVC(C=0.01, gamma='auto', kernel='sigmoid')

0.34375

SVC(C=100, kernel='linear') 0.8229166666666666

SVC(C=100, gamma='auto', kernel='linear') 0.8229166666666666

SVC(C=50, kernel='linear') 0.8229166666666666

SVC(C=50, gamma='auto', kernel='linear') 0.8229166666666666

SVC(C=10, kernel='linear') 0.8229166666666666

SVC(C=10, gamma='auto', kernel='linear') 0.8229166666666666

SVC(kernel='linear') 0.78125

SVC(gamma='auto', kernel='linear') 0.78125

SVC(C=0.1, kernel='linear') 0.71875

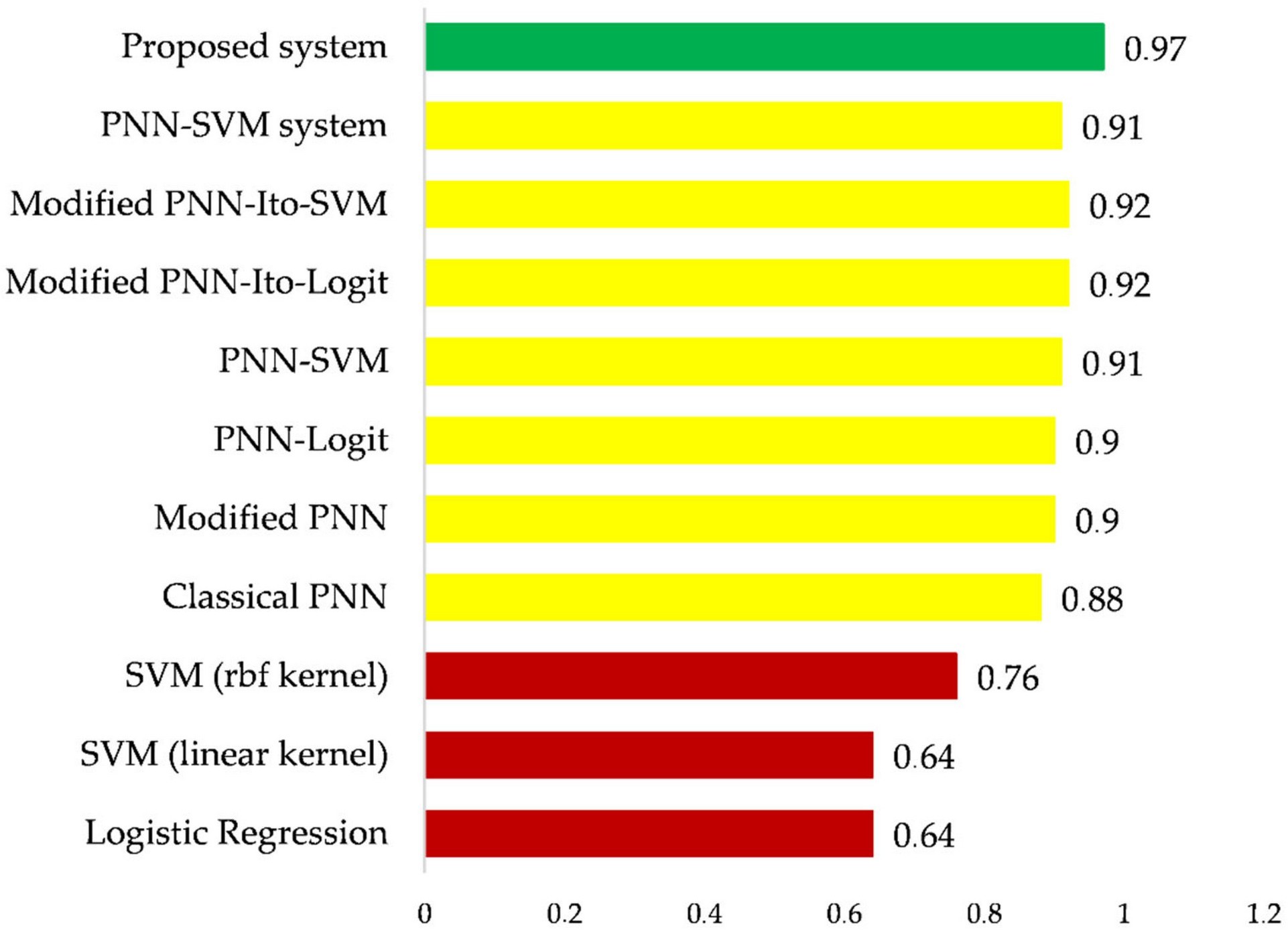
SVC(C=0.1, gamma='auto', kernel='linear') 0.71875

SVC(C=0.01, kernel='linear') 0.34375

SVC(C=0.01, gamma='auto', kernel='linear') 0.34375

У порівнянні із результатами статті у нас вийшла набагато краща система оскільким ми перебрали гіперпараметри моделі. У статі PNN-SVM та їхня запропонована модель мають точності 91% та 97% відповідно. У нашому ж випадку вдалося підняти цю точнітсь до 100% одразу у 3 різних наборах паремтрів, а саме:

SVC(C=100, kernel='poly') 1.0 SVC(C=50, kernel='poly') 1.0 SVC(C=10, kernel='poly') 1.0



# Висновок:

Данні саме у цьому датасеті ідеально підходять для вирівнювання задопомогою Polynomial Kernel. У результаті дослідження і пошуку гіпер параметрів мені вдалося підвищити точність багатьох методів зі статті.