**Theory and Conceptual Understanding**

**1. What is Artificial Immune Pattern Recognition?**

* **Artificial Immune Systems (AIS)** are computational models inspired by the principles and processes of the biological immune system.
* In **Pattern Recognition tasks** like classification, an **Artificial Immune System** simulates how natural immune cells (like antibodies) recognize pathogens (antigens) to **detect patterns or anomalies**.
* A special method under AIS is **Negative Selection Algorithm (NSA)**:
  + It **generates "detectors"** that can distinguish between **self** (normal behavior) and **non-self** (anomalies or different classes).
  + These detectors can recognize when a pattern **deviates from the normal** and classify it.

**2. Structural Damage Classification Problem**

In engineering, especially **civil and mechanical engineering**, monitoring structures (like bridges, buildings) for **damage** is important.

Here, we assume:

* **Classes**:
  1. **No Damage** (Healthy structure)
  2. **Minor Damage** (Early signs of structural weakness)
  3. **Major Damage** (Severe structural issues)

Our goal:

* Based on **sensor readings or features**, classify a structure into one of these 3 categories using an immune-inspired algorithm.

**3. How this Practical Simulates it?**

**a) Synthetic Data Generation**

Since real structural damage data is not available here:

* We **simulate** feature data using random numbers to **mimic real-world noisy measurements**.
* 3 different clusters are created corresponding to:
  + No damage (centered around 0)
  + Minor damage (centered around 2)
  + Major damage (centered around 4)
* Some **noise and label flips** are intentionally added to make the problem harder and realistic (just like real-world data can be noisy).

**b) NSA Classifier**

We create a **custom classifier** inspired by:

* **Negative Selection Algorithm (NSA)**
* **Affinity (Distance-based) Matching**, specifically using **Mahalanobis Distance**.

**Main Ideas:**

* For each class (damage category):
  + Create a number of **detectors** (similar to immune cells).
  + Each detector is a point in feature space that "recognizes" examples of its class.
* When a new sample is given:
  + Check **which detector is closest** (using Mahalanobis Distance).
  + Assign the label of the **closest detector** to the sample.

**Mahalanobis Distance** is used because:

* It considers **correlation** between features.
* It **adapts** based on how spread out data is in different directions.
* Much better than plain Euclidean distance when data is correlated.

**c) Evaluation**

Once the NSA Classifier is trained:

* It is tested on unseen data.
* Evaluation metrics like:
  + **Classification Report** (Precision, Recall, F1-Score)
  + **Confusion Matrix**
  + **PCA Visualization**
  + **Bar Plot of Metrics** are plotted to **analyze** the model performance visually and numerically.

**4. Key Concepts in the Code**

| **Concept** | **Description** |
| --- | --- |
| **Synthetic Data** | Generated with noise and class separation. |
| **Detectors** | Randomly mutated points from class samples, used to recognize future inputs. |
| **Mahalanobis Distance** | A distance metric considering feature variance and correlation. |
| **Negative Selection** | Training phase where detectors are generated by mutating real samples. |
| **Classification** | Done by finding the detector with minimum Mahalanobis distance to the test sample. |
| **Confusion Matrix** | Visual matrix showing where model is correct and where it confuses classes. |
| **PCA** | Reduces high-dimensional test data to 2D for visualization. |
| **Performance Metrics** | Precision, Recall, F1-score plotted for easy interpretation. |

**5. Summary of Theory**

* **Artificial Immune Systems (AIS)** mimic biological immunity to solve computational problems.
* **Negative Selection Algorithm (NSA)** is a major technique for **anomaly detection** and **classification**.
* In this practical:
  + Synthetic structural damage data was generated.
  + A **custom NSA classifier** was built using detectors and Mahalanobis distance.
  + It was evaluated using proper **ML evaluation metrics** and visualizations.
* This approach is suitable for problems where:
  + **Data is noisy**
  + **Multiple classes** exist
  + **Adaptive, biologically-inspired** pattern recognition is helpful.

**🎯 Practical Flowchart**

plaintext

CopyEdit

Generate Synthetic Data (Simulate Damage Levels)

↓

Split into Training and Test Sets

↓

Train NSA Classifier

- Generate detectors

- Store covariance information

↓

Predict on Test Set

↓

Evaluate Performance

- Confusion Matrix

- Classification Report

- PCA Visualization

- Precision/Recall/F1 Bar Plot

**Full, Extremely Detailed Code Explanation**

**📂 Importing Required Libraries**

python

CopyEdit

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

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

from sklearn.metrics import classification\_report, confusion\_matrix

from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler

import pandas as pd

from scipy.spatial import distance

from numpy.linalg import inv, LinAlgError

✅ **What happens here?**

* **numpy**: for handling numerical arrays and computations.
* **matplotlib.pyplot**: for plotting graphs (line plots, scatter plots, etc.).
* **seaborn**: built on matplotlib, gives prettier graphs (like heatmaps).
* **sklearn.model\_selection.train\_test\_split**: for splitting data into training and testing.
* **sklearn.metrics**: to evaluate model performance (classification report, confusion matrix).
* **sklearn.decomposition.PCA**: for dimensionality reduction (for visualization in 2D).
* **sklearn.preprocessing.StandardScaler**: standardize data (though in this code it's imported but unused).
* **pandas**: for easy handling of tabular data (used for report formatting).
* **scipy.spatial.distance**: used to compute Mahalanobis distance.
* **numpy.linalg.inv, LinAlgError**:
  + inv() calculates the inverse of a matrix.
  + LinAlgError handles exceptions if a matrix is non-invertible.

**📂 Data Generation Function**

python

CopyEdit

def generate\_harder\_data(samples=300, features=6, noise=1.5, label\_flip\_rate=0.05):

X = np.zeros((samples, features))

y = np.zeros(samples)

✅ **Purpose:**  
Create **synthetic dataset** for 3 damage classes:

* 300 samples
* 6 features per sample
* With noise
* And intentional label flipping (to simulate real-world noisy labels)

✅ **X**:

* Feature matrix of zeros initially (300 x 6).

✅ **y**:

* Label array of zeros initially (300, 1 per sample).

**Inside the Loop:**

python

CopyEdit

for i in range(samples):

cls = i % 3

y[i] = cls

* For each sample i:
  + Assign it a class (cls):
    - 0, 1, or 2 (using modulo 3 → cycles between 0, 1, 2, 0, 1, 2...)
  + Store this class in the label array y.

✅ This ensures an **even number of samples** for each class.

**Create Features based on Class:**

python

CopyEdit

if cls == 0:

X[i] = np.random.normal(loc=0, scale=noise, size=features)

elif cls == 1:

X[i] = np.random.normal(loc=2, scale=noise, size=features)

else:

X[i] = np.random.normal(loc=4, scale=noise, size=features)

* Based on the assigned class:
  + Generate **random feature values** from a **Gaussian Distribution**.
  + Different **means (centers)**:
    - Class 0 (No damage): Mean = 0
    - Class 1 (Minor damage): Mean = 2
    - Class 2 (Major damage): Mean = 4
* **Noise** adds variability (standard deviation = 1.5).

**Add Label Noise:**

python

CopyEdit

flip\_indices = np.random.choice(samples, int(samples \* label\_flip\_rate), replace=False)

y[flip\_indices] = np.random.choice([0, 1, 2], size=len(flip\_indices))

✅ Randomly flip ~5% of labels:

* flip\_indices: randomly choose indices to flip.
* y[flip\_indices]: randomly assign a new label from {0, 1, 2}.

✅ Simulates **human error** or **sensor noise** in real-world datasets.

**Return Statement:**

python

CopyEdit

return X, y.astype(int)

✅ Return the feature matrix X and label array y (converted to integer type).

**📂 NSAClassifier Class (Negative Selection Algorithm)**

python

CopyEdit

class NSAClassifier:

def \_\_init\_\_(self, num\_detectors=150, mutation\_rate=0.1):

self.num\_detectors = num\_detectors

self.mutation\_rate = mutation\_rate

✅ **This defines a new type of model (classifier).**

* num\_detectors: total number of detectors to create.
* mutation\_rate: randomness when creating detectors (for exploration).

**📂 Training the Classifier**

python

CopyEdit

def train(self, X, y):

self.detectors = []

self.class\_cov\_inv = {}

✅ **Purpose:**

* Create detectors for each class based on training data.
* Calculate and store the **inverse covariance matrix** for Mahalanobis distance.

✅ **detectors**:

* List of all created detectors.

✅ **class\_cov\_inv**:

* Dictionary mapping each class label → inverse covariance matrix.

**For each unique class:**

python

CopyEdit

for label in np.unique(y):

print(f"Generating detectors for class {label}")

class\_samples = X[y == label]

✅ Loop over all unique labels (0, 1, 2):

* Get **samples belonging to that class**.

**Try to compute Covariance Inverse:**

python

CopyEdit

try:

cov\_inv = inv(np.cov(class\_samples.T) + 1e-6 \* np.eye(X.shape[1]))

self.class\_cov\_inv[label] = cov\_inv

✅ Calculate the **covariance matrix** of features for this class. ✅ Add 1e-6 \* Identity Matrix to make sure matrix is invertible. ✅ Then compute **inverse covariance** → Needed for Mahalanobis distance.

**Handle Matrix Errors:**

python

CopyEdit

except LinAlgError:

print(f"Covariance matrix for class {label} is singular. Using identity matrix.")

self.class\_cov\_inv[label] = np.eye(X.shape[1])

✅ If matrix is **singular (non-invertible)**:

* Fallback: use an **identity matrix**.

**Create Detectors for this Class:**

python

CopyEdit

n\_detectors = self.num\_detectors // len(np.unique(y))

for \_ in range(n\_detectors):

idx = np.random.randint(0, len(class\_samples))

center = class\_samples[idx] + np.random.normal(0, self.mutation\_rate, size=X.shape[1])

self.detectors.append({'center': center, 'label': label})

✅ For each class:

* Equally divide detectors among classes.
* Randomly pick a sample.
* Slightly mutate it (add small noise) to form a new detector.
* Store detector as a dictionary → {'center': mutated\_sample, 'label': class\_label}

**📂 Predict Function**

python

CopyEdit

def predict(self, X):

preds = []

✅ For a given test dataset X, predict class labels.

**For each test sample:**

python

CopyEdit

for x in X:

best\_label = None

best\_distance = float('inf')

✅ Initialize:

* best\_distance: very large (positive infinity).
* best\_label: none yet.

**Compare with Each Detector:**

python

CopyEdit

for det in self.detectors:

cov\_inv = self.class\_cov\_inv.get(det['label'], np.eye(len(x)))

dist = distance.mahalanobis(x, det['center'], cov\_inv)

✅ For each detector:

* Get the correct covariance inverse for the detector's class.
* Compute **Mahalanobis distance** between the test sample and the detector's center.

**Select Best Match:**

python

CopyEdit

if dist < best\_distance:

best\_distance = dist

best\_label = det['label']

preds.append(best\_label)

✅ Find the detector with the **smallest distance**. ✅ Assign its label as the prediction for the sample.

✅ After looping through all detectors, append the best label to preds.

**Return All Predictions:**

python

CopyEdit

return np.array(preds)

✅ Return the array of predicted labels.

**📂 Running the Test**

python

CopyEdit

X, y = generate\_harder\_data(samples=300, features=6, noise=1.5, label\_flip\_rate=0.05)

✅ Generate synthetic dataset.

python

CopyEdit

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, stratify=y, random\_state=42)

✅ Split into training (70%) and testing (30%) sets. ✅ Use stratify so that class distribution is preserved.

python

CopyEdit

clf = NSAClassifier(num\_detectors=150, mutation\_rate=0.1)

clf.train(X\_train, y\_train)

✅ Initialize classifier. ✅ Train it on training data.

python

CopyEdit

y\_pred = clf.predict(X\_test)

✅ Predict on the test set.

**📂 Evaluate the Model**

python

CopyEdit

print("Classification Report:")

print(classification\_report(y\_test, y\_pred, target\_names=["No Damage", "Minor Damage", "Major Damage"]))

✅ Show Precision, Recall, F1-score for each class.

**Confusion Matrix Heatmap**

**Code:**

python

CopyEdit

cm = confusion\_matrix(y\_test, y\_pred)

plt.figure(figsize=(6,5))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',

xticklabels=["No Damage", "Minor Damage", "Major Damage"],

yticklabels=["No Damage", "Minor Damage", "Major Damage"])

plt.title("Confusion Matrix")

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.tight\_layout()

plt.show()

**✅ What is it?**

* A **confusion matrix** shows how well your model predicted each class.
* **Rows** → Actual (true) labels.
* **Columns** → Predicted labels.
* Each cell (i, j) → Number of samples where:
  + True class = i
  + Predicted class = j

**✅ How to read it?**

Example confusion matrix layout:

|  | **Predicted No Damage** | **Predicted Minor Damage** | **Predicted Major Damage** |
| --- | --- | --- | --- |
| **Actual No Damage** | 35 | 5 | 2 |
| **Actual Minor Damage** | 3 | 30 | 7 |
| **Actual Major Damage** | 2 | 4 | 36 |

Interpretations:

* 35 samples of true No Damage were **correctly** predicted as No Damage.
* 5 samples of true No Damage were **incorrectly** predicted as Minor Damage.
* 2 samples of true No Damage were **wrongly** predicted as Major Damage.
* and so on.

✅ **Diagonal elements** (Top-left to Bottom-right) = **Correct predictions** ✅  
✅ **Off-diagonal elements** = **Misclassifications** ❌

**✅ Visual Aspects:**

* **Blues colormap** → Darker blue = More samples.
* **Numbers** shown (annot=True) inside each box.
* **Axes labels**:
  + X-axis = Predicted
  + Y-axis = Actual

**✅ Why is it useful?**

* Immediately spot **which classes are confused with each other**.
* Check if the model is **biased** towards one class.
* Identify **major weaknesses** in predictions.

**📊 2. PCA 2D Scatter Plot**

**Code:**

python

CopyEdit

pca = PCA(n\_components=2)

X\_test\_2D = pca.fit\_transform(X\_test)

plt.figure(figsize=(7,5))

scatter = plt.scatter(X\_test\_2D[:, 0], X\_test\_2D[:, 1], c=y\_pred, cmap='viridis', edgecolors='k', alpha=0.8)

plt.legend(handles=scatter.legend\_elements()[0], labels=["No Damage", "Minor", "Major"])

plt.title("NSA Predicted Classes in PCA-Reduced 2D Space")

plt.xlabel("PCA 1")

plt.ylabel("PCA 2")

plt.grid(True)

plt.tight\_layout()

plt.show()

**✅ What is it?**

* **PCA (Principal Component Analysis)** reduces high-dimensional data (6 features) into just **2 features**.
* Then, it plots the test points in **2D space**, coloring each point by the **predicted class**.

**✅ How to read it?**

* Each **dot** = one sample.
* **Color of dot** = class predicted by NSA:
  + Example:
    - Yellow = Major Damage
    - Green = Minor Damage
    - Purple = No Damage  
      (depends on viridis colormap).
* **Clustered points**:
  + If dots of the same color group together = Model is consistent.
* **Mixed clusters**:
  + If different colors overlap = Model is confused between classes.

✅ If you see **clear separate regions** (different colors), it means NSA is predicting well.

**✅ Visual Aspects:**

* **X-axis = PCA Component 1**, **Y-axis = PCA Component 2** (abstract directions capturing most variance).
* **Edge colors** = black outline around each point (for better visibility).
* **Legend** shows what each color means.
* **Alpha=0.8** = points are slightly transparent.

**✅ Why is it useful?**

* Visualize how well the **NSA classifier separates different classes**.
* Spot if there’s **overlap/confusion** between classes in feature space.
* Helps **debug** if certain classes are hard to separate.

**📊 3. Bar Plot of Performance Metrics (Precision, Recall, F1-score)**

**Code:**

python

CopyEdit

report\_dict = classification\_report(y\_test, y\_pred, output\_dict=True)

df = pd.DataFrame(report\_dict).transpose().iloc[:3][["precision", "recall", "f1-score"]]

df.plot(kind='bar', figsize=(8,5), ylim=(0,1), colormap="Set2", rot=0)

plt.title("Performance Metrics per Class")

plt.ylabel("Score")

plt.tight\_layout()

plt.show()

**✅ What is it?**

* A **bar chart** showing:
  + Precision
  + Recall
  + F1-score
* For **each class** separately.

**✅ How to read it?**

For each class (No Damage, Minor Damage, Major Damage):

* There are 3 bars:
  + **Precision**: How many predicted positives were actually correct?
  + **Recall**: How many actual positives were captured?
  + **F1-score**: Balance between precision and recall.

✅ Height of the bar (0 to 1 scale) = value of the metric.

| **Metric** | **Meaning** |
| --- | --- |
| Precision | Of all predicted 'Minor Damage', how many were correct? |
| Recall | Of all actual 'Minor Damage', how many were detected? |
| F1-score | Trade-off (harmonic mean) between precision and recall |

**✅ Visual Aspects:**

* **Different colors** for different metrics (due to Set2 colormap).
* **X-axis** = classes.
* **Y-axis** = metric value (between 0 and 1).

**✅ Why is it useful?**

* Quickly compare **model performance across classes**.
* Identify if the model is:
  + **Better at one class** (high F1).
  + **Worse at another class** (low precision or recall).
* Helps decide if you need to **collect more data** for certain classes.

**📢 Summary Table:**

| **Graph** | **What It Shows** | **Why Important** |
| --- | --- | --- |
| Confusion Matrix Heatmap | Correct vs incorrect predictions | Find which classes are confused |
| PCA 2D Scatter Plot | Spatial separation of predicted classes | Visualize class separability |
| Bar Plot | Precision, Recall, F1-score per class | Measure model's balance and strengths |

**🚀 Visual Flow in Your Code:**

1. **Train model** on synthetic noisy data.
2. **Predict** on test data.
3. **Confusion Matrix** → Where the model got it right/wrong.
4. **PCA Plot** → How predictions cluster spatially.
5. **Bar Chart** → How strong/weak the classifier is for each damage type.