POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Credit card fraud detection

NUMERICAL ANALYSIS FOR MACHINE LEARNING

COMPUTER SCIENCE AND ENGINEERING (T2I - ARTIFICIAL INTELLIGENCE)

First Name: Davide
Family Name: Gesualdi

Student Number: 101761 - Personal Code: 10885255

Professor:
Edie MIGLIO

Course credits:
10

Academic year:
2023-2024

Abstract: The project aims to implement a method for credit card fraud detection, building upon the research presented in "Efficient credit card fraud detection using evolutionary hybrid feature selection and random weight networks" [1]. The proliferation of digital transactions has concurrently increased the vulnerability of credit card systems to fraudulent activities. Effective fraud detection is crucial to safeguard financial assets. The paper introduces an efficient method for credit card fraud detection that relies on Competitive Swarm Optimization (CSO) and Random Weight Network (RWN). Additionally, the system includes an automated hybrid feature selection capability to identify the most pertinent features during the detection process. The experimental outcomes validate that this system can attain outstanding results in G-Mean, RUC, and Recall values.

1. Introduction

Credit card fraud detection has become a critical challenge in modern financial transactions, driven by the rising prevalence of online transactions and digital payment methods. The unauthorized use of credit cards for fraudulent activities presents considerable financial risks for individuals, businesses and financial institutions. Machine learning algorithms have revolutionized the field of fraud detection, providing advanced capabilities for automated data analysis and pattern recognition [2] [3]. These algorithms can learn from historical transaction data and identify anomalies that may signal fraudulent behavior. The problem of credit card fraud detection is particularly complex from a machine learning perspective. Notably, the dataset is inherently imbalanced, with legitimate transactions vastly outnumbering fraudulent ones. Moreover, the concept of fraud evolves over time as consumer habits and fraudulent tactics undergo changes, resulting in concept drift within the data. The selection of relevant features from transaction data is a cornerstone of constructing effective fraud detection models. Not all attributes contribute equally to the discrimination between legitimate and fraudulent transactions [4]. Feature selection is the process of identifying and retaining those attributes that are most influential in detecting fraudulent patterns, while discarding irrelevant or redundant information. Methods like filter and wrapper approaches are commonly employed to carry out feature selection, allowing models

to achieve enhanced performance, characterized by improved accuracy, reduced false positive rates and better generalization across diverse fraud scenarios [5]. The proposed efficient credit card fraud detection model, named HybridIG-CSO, utilizes an automatic feature selection mechanism that identifies significant features through the combined use of Information Gain (IG) and Competitive Swarm Optimization (CSO) techniques, with a Random Weight Network (RWN) serving as the foundational classifier. The objective of this project is to implement the HybridIG-CSO model and subsequently compare its performance against several fundamental classifiers, including Naïve Bayes (NB), Random Forest (RF) and Support Vector Machine (SVM).

2. Dataset Analysis

The models were evaluated using four distinct datasets, as outlined in the table below, where "Abb." denotes the assigned dataset code and "#PS" signifies the positive samples in each dataset:

Dataset	Abb.	#Samples	#Features	#PS	Link
Loan Prediction	D_1	614	11	192 (31%)	https://github.com/Paliking/ML_examples/blob/master/LoanPrediction/train_u6lujuX_CVtuZ9i.csv
Creditcardcsvpresent	D_2	3075	10	448 (14%)	https://github.com/gksj7/creditcardcsvpresent/blob/main/creditcardcsvpresent.csv
Default ofCreditCardClients	D_3	30000	23	6636 (22%)	https://archive.ics.uci.edu/dataset/350/default+of+credit+card+clients
European cardholders	D_4	284807	30	482 (0.17%)	https://kaggle.com/mlg-ulb/creditcardfraud

Table 1: The dataset characteristics.

2.1. Skewness

Skewness is a statistical measure that quantifies the asymmetry of the distribution of values in a dataset. It indicates whether the data points are skewed to the left (negative skew) or the right (positive skew) relative to the mean.

- Positive Skewness (Right Skew): the majority of data points are concentrated on the left side of the distribution, with a few extreme values on the right side.
$$\text{Mean} > \text{Median} > \text{Mode}$$
- Negative Skewness (Left Skew): most of the data points are concentrated on the right side of the distribution, while a few extreme values on the left side.
$$\text{Mean} < \text{Median} < \text{Mode}$$
- Zero Skewness (Symmetrical Distribution): perfectly symmetrical distribution, where the mean, median, and mode are equal. In such a distribution, the data points are evenly distributed around the central point.

Skewness was computed only for numerical attributes containing no missing values.

```
skewed_feats = dataset[numerical_cols].apply(lambda x: skew(x.dropna()))
skewness = pd.DataFrame({"Variable":skewed_feats.index, "Skewness":skewed_feats.values})
```

Variable	Skewness
ApplicantIncome	6.523526
CoapplicantIncome	7.473217
Credit_History	-1.877351
LoanAmount	2.670763
Loan_Amount_Term	-2.356504

Table 2: Skewness dataset D_1.

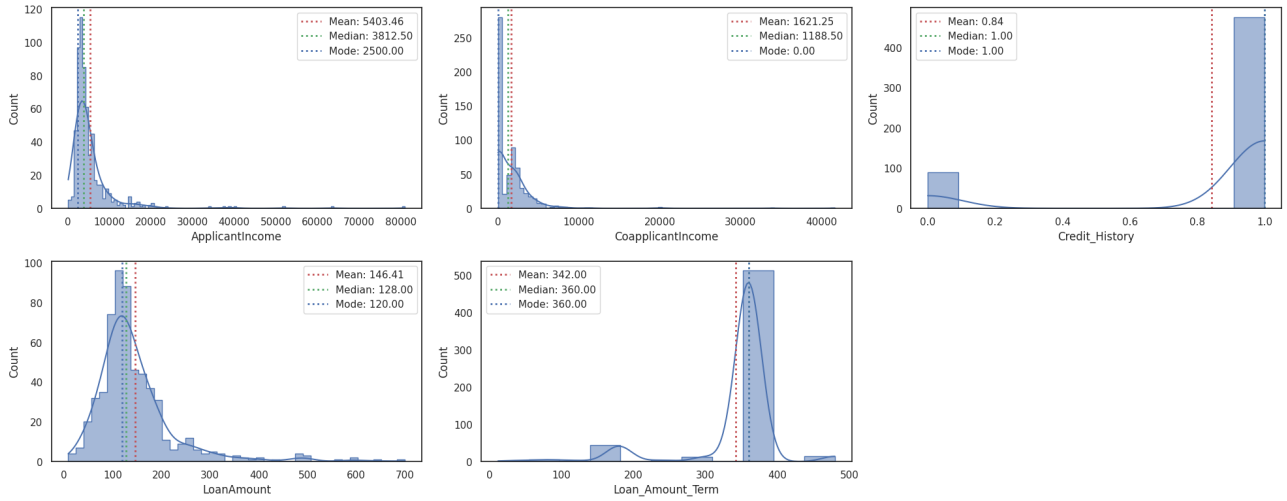


Figure 1: Histograms to show distributions of dataset D_1 with a kernel density estimate to smooth the distribution and vertical lines denoting the mean (red), median (green), mode (blue).

2.2. Correlation Analysis

The computation of pairwise attribute correlations using the Pearson correlation coefficient (excluding missing values) revealed that, across all four datasets, the attributes are predominantly weakly linear correlated, either positively or negatively. Only a small number of attributes exhibit moderate positive linear correlations. The exception is the third dataset, which exhibits a higher number of strong linear correlations compared to the other datasets. Nevertheless, these strong correlations are still relatively modest in comparison to the overall prevalence of very weak correlations.

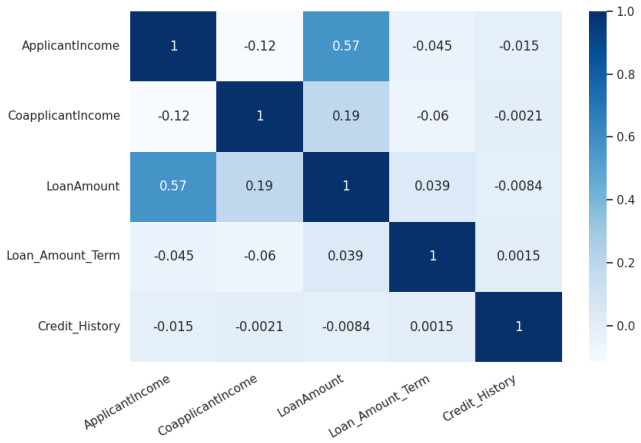


Figure 2: Correlation Analysis of dataset D_1.

2.3. Dataset splitting

The dataset was shuffled and split into training, validation and test sets, with proportions of 70%, 15% and 15%, respectively. During the testing phase, model performance was assessed both on a separate test set and using a 10-fold cross-validation approach.

2.4. Handling missing values

Missing values were exclusively present in the first dataset. Categorical features underwent label encoding prior to handling missing data.

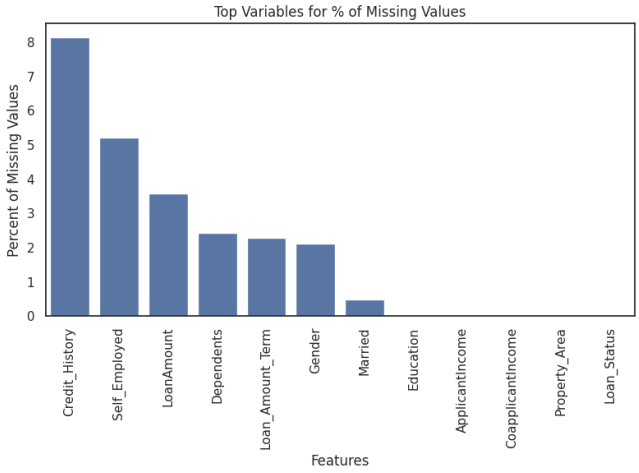


Figure 3: Top Variables by percentage of Missing Values in the dataset D_1.

To address missing values, a K-Nearest Neighbors (KNN) imputation approach was employed. For numerical features, KNN mean imputation was implemented using the KNNImputer class from the scikit-learn library, with the number of neighbors set to 3. Consequently, missing values in numerical features for each sample were imputed with the mean value of its K nearest neighbors in the training dataset.

```
X_train_imputed = X_train.dropna(subset=categorical_cols)
imputer = KNNImputer(n_neighbors=n_neighbors).fit(X_train_imputed)
X_train_imputed = pd.DataFrame(
    imputer.transform(X_train_imputed),
    columns=X_train.columns,
    index=X_train_imputed.index)
```

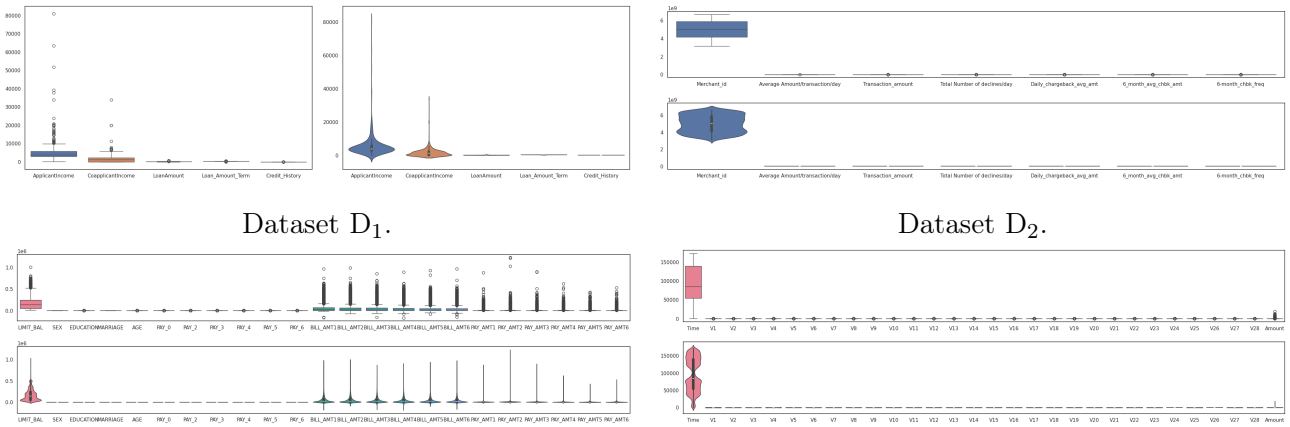
Categorical features were imputed using KNN majority vote imputation through the scikit-learn KNeighborsClassifier class.

```
def majority_vote_imputation(df, df_imputed, classifier, categorical_cols, n_neighbors):
  for col in categorical_cols:
    to_impute = df[df[col].isna() & df.drop(columns=col).notna().all(axis=1)].copy()
    for i in range(len(to_impute)):
      sample = to_impute.drop(columns=col).iloc[i]
      to_impute[col].iloc[i] = classifier[col].predict([sample])[0]
    df_imputed = pd.concat([df_imputed, to_impute])
  return df_imputed
```

Samples containing missing values in multiple categorical features were removed from the dataset, resulting in a reduction of dataset size of only 13 samples (2%).

2.5. Data normalization

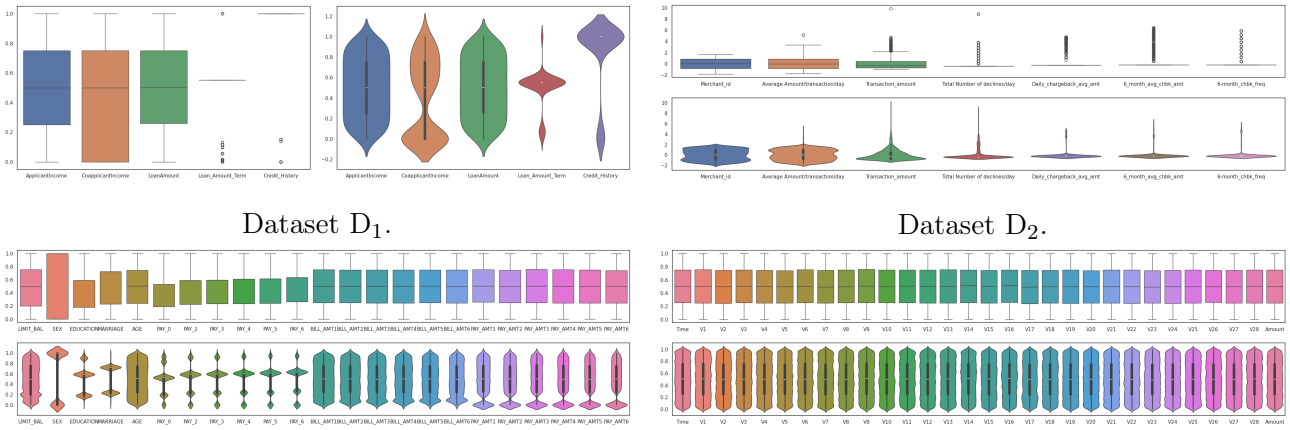
Box plots and violin plots generated from the original datasets revealed a clear need for data normalization, due to the pervasive presence of outliers and the disparate scales across variables.
After evaluating the StandardScaler, the QuantileTransformer was selected for data normalization.
StandardScaler is sensitive to outliers, as it relies on the mean and standard deviation for scaling, which can be significantly influenced by extreme values.
In contrast, QuantileTransformer mitigates the impact of outliers by using quantiles information. QuantileTransformer applies a non-linear transformation such that the probability density function of each feature will be mapped to a uniform or Gaussian distribution. In this case, all the data, including outliers, will be mapped to a uniform distribution within the range [0, 1], making outliers indistinguishable from inliers.



Dataset D_1. Dataset D_2.

Dataset D_3. Dataset D_4.

Figure 4: Box plots and violin plots of the original data in the datasets.



Dataset D_1. Dataset D_2.

Dataset D_3. Dataset D_4.

Figure 5: Box plots and violin plots of the normalized data in the datasets.

2.6. Imbalanced Data

In order to address the class imbalance present in the training data, the Synthetic Minority Over-sampling Technique (SMOTE) was employed.
This method surpasses traditional oversampling techniques, which merely duplicate existing minority class instances, by generating synthetic instances in the neighborhood of existing minority class samples.
SMOTE accomplishes this by identifying the k-nearest neighbors of every existing minority class samples x. Then, it generates a synthetic instance at some intermediate point along the line segment joining x to one of its randomly chosen k-nearest neighbor, until the desired number of synthetic instances is reached. This approach effectively augments the minority class without altering the number of majority class instances, resulting in a more balanced dataset while preserving data diversity.

3. HybridIG-CSO

The key elements of the HybridIG-CSO model are feature selection and classification using Random Weight Network (RWN).

3.1. Feature selection

3.1.1. Information Gain (IG)

In the filter approach, features are evaluated independently based on statistical measures such as information gain. The information gain technique is a mathematical method, which measures the reduction in uncertainty, or entropy, about a target variable when a specific feature is known.
Mathematically, the information gain (IG) for a feature X with respect to a target variable Y can be calculated as:

$$IG(Y|X) = H(Y) - H(Y|X) \tag{1}$$

where $H(Y)$ represents the entropy of the target variable Y before considering feature X, and $H(Y|X)$ represents the conditional entropy of Y given the values of feature X.

```
def InformationGain(attribute, target):
  original_entropy = DistributionEntropy(target.value_counts())
  attribute_entropy = AttributeEntropy(attribute, target)
  return original_entropy - attribute_entropy
```

Additionally, the calculations for entropy $H(Y)$ and conditional entropy $H(Y|X)$ are outlined as follows, where $P(x)$ is the proportion of instances with value x for feature X and $H(Y|X = x)$ is the entropy of Y for instances where feature X has value x:

$$H(Y) = -\sum_{y \in Y} P(y) log_2(P(y)) \tag{2}$$

```
def DistributionEntropy(fc):
  d = fc/sum(fc)
  return sum([-x * np.log2(x) for x in d[d!=0]])
```

$$H(Y|X) = -\sum_{x \in X} P(x) \sum_{y \in Y} P(y|x) log_2(P(y|x)) \tag{3}$$

```
def AttributeEntropy(attribute, target):
  # Attribute entropy for numerical attributes
  if attribute.dtype != 'object':
    values = np.sort(attribute.unique())
    entropy_values = []
    for i, value in enumerate(values):
      left_mask = attribute<=value
      right_mask = ~left_mask
      entropy = left_mask.sum()/len(attribute) * DistributionEntropy(target[left_mask].
    value_counts())
      entropy += right_mask.sum()/len(attribute) * DistributionEntropy(target[right_mask].
    value_counts())
      entropy_values.append(entropy)
    return np.min(entropy_values)
  # Attribute entropy for categorical attributes
  weights = attribute.value_counts()/sum(attribute.value_counts())
  attribute_values = attribute.value_counts().index
  entropy = 0
  for i, value in enumerate(attribute_values):
    mask = attribute==value
    entropy += weights[i] * DistributionEntropy(target[mask].value_counts())
  return entropy
```

The attribute entropy for continuous numerical attributes was determined by initially sorting numerical values, including class labels, followed by an exhaustive evaluation of potential cut-points to identify the optimal split maximizing information gain, equivalently minimizing attribute entropy.

3.1.2. Competitive Swarm Optimization (CSO)

The wrapper-based technique employed is the CSO, chosen for its ability to explore the complex search space of potential feature subsets. CSO is an algorithm rooted in the original PSO technique, devised to tackle the issue of premature convergence that often arises when applying PSO to complex search spaces containing numerous local optima[6], by removing the influence of *gbest* and *pbest* associated with each particle. Consequently, CSO solely relies on pairwise competition between randomly selected particles to guide the search process. Hence, with the assumption of having k particles within the swarm (population), the CSO procedure initiates with

a population consisting of randomly initialized particles denoted as $P(t)$, where 't' signifies the generation. Every potential solution is depicted by one of the swarm's particles. Each particle can be considered as a point represented by a position X and a velocity V. During each iteration, the swarm $P(t)$ is divided into two equal and randomly selected groups. Subsequently, CSO selects two particles, one from each group, and initiates a competition between them. The winning particle is directly transferred to the next generation. The losing particle is updated based on the information derived from the winner and then included in the next generation. The positions and velocities of the particles that emerged as winners and losers in the i^{th} pairwise competition, relative to generation t, can be expressed as follows, where the winning particle's position is denoted as $X_{wi}(t)$ and its velocity $V_{wi}(t)$ and, similarly, the losing particle's position is $X_{li}(t)$ and its velocity $V_{li}(t)$:

$$V_{li}(t+1) = R_1(i,t)V_{li}(t) + R_2(i,t)(X_{wi}(t) - X_{li}(t)) + \varphi R_3(i,t)(\overline{X}_i(t) - X_{li}(t)) \tag{4}$$

$$X_{li}(t+1) = X_{li}(t) + V_{li}(t+1) \tag{5}$$

$R_1(i,t)$, $R_2(i,t)$ and $R_3(i,t)$ represent three vectors of randomly generated numbers sampled from the range [0,1). $\overline{X}_i(t)$ denotes the average position of the pertinent particles. These pertinent particles could encompass the entire swarm of particles or a predefined group of neighboring particles. The parameter φ governs the extent to which $\overline{X}_i(t)$ influences the process.

3.2. Random Weight Network (RWN)

The fundamental architecture of the RWN architecture follows a fully connected architecture with a single hidden layer. Unlike conventional gradient descent methods that necessitate the adjustment of multiple parameters, such as learning rates and the number of training epochs, RWN simplifies the training process by focusing solely on the number of hidden neurons.

RWN begins by randomly initializing the input weights and hidden layer biases. Subsequently, it utilizes a dataset of N training samples to construct the hidden layer output matrix. The output weights are then determined through the application of the Moore-Penrose (MP) generalized inverse.

Given a dataset of N training samples, where each sample is represented as (x_i, t_i), with $x_i \in \mathbb{R}^n$ and $t_i \in \mathbb{R}^m$, and considering the activation function $g(x)$, the weight vector $w_j \in \mathbb{R}^n$ connecting the j^{th} hidden neuron to the n input nodes, and the set of output weights values $\beta_j \in \mathbb{R}^m$ connecting the j^{th} hidden neurons to the m output nodes, compute the output weight matrix β[7], such that:

$$H\beta = T \tag{6}$$

where H is the hidden layer output matrix and T is the target matrix:

$$H = \begin{Bmatrix} g(w_1 \cdot x_1 + b_1) & \cdots & g(w_L \cdot x_1 + b_L) \\ \vdots & \cdots & \vdots \\ g(w_1 \cdot x_N + b_1) & \cdots & g(w_L \cdot x_N + b_L) \end{Bmatrix}_{N \times L} \quad \beta = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_L^T \end{bmatrix}_{L \times m} \quad T = \begin{bmatrix} t_1^T \\ \vdots \\ t_N^T \end{bmatrix}_{N \times m} \tag{7}$$

The developed implementation utilized the sigmoid activation function in conjunction with a prediction threshold of 0.5. Predictions were classified as positive if they exceeded this threshold, and negative otherwise.

```
def sigmoid(x):
  return 1 / (1 + np.exp(-x))

def compute_H(dataset, particle):
  n_neurons = int(particle['n_neurons'], 2)
  params = particle['params']
  # Select only the weights and biases corresponding to the selected features
  mask = np.array(list(particle['features'].values()))==1
  weights = params[0][mask, :n_neurons]
  bias = params[1][:n_neurons].flatten()
  H = sigmoid(dataset @ weights + bias)
  return np.array(H)

def RWN_model(particle, X, y):
  X_train, y_train = prepare_data(particle, X, y)
  # Calculate the hidden layer output matrix H
  H = compute_H(X_train, particle)
  # Output weights determined using the Moore-Penrose generalized inverse
  return np.linalg.pinv(H) @ y_train
```

3.3. Methodology

The initial phase employs IG as a filter-based method to rank the features within the credit card dataset. Only the top-ranked features are retained and subsequently passed to the wrapper-based algorithm, CSO. The RWN algorithm serves as the learning model within this hybrid framework.

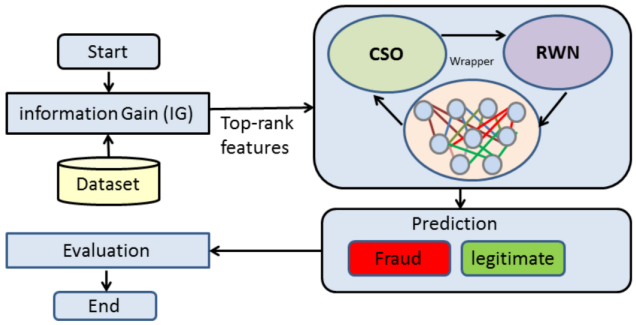


Figure 6: Flowchart of proposed method.

The CSO particle is encoded as a real vector encompassing the subsequent components:

- A set of binary flags indicating the inclusion or exclusion of corresponding features.
- A set of binary flags dictating the number of neurons in the hidden layer of the RWN.
- The RWN parameters, which encapsulate the values of input weights and hidden biases.

```
def create_particles(population, ig_attributes):
  particles = []
  for subset in population:
    features = {}
    for col in ig_attributes:
      features[col] = 1 if col in subset else 0
    n_neurons = ''.join(map(str, features.values()))
    params = initialize_RWN_params(
        np.array(list(features.values())), int(n_neurons, 2))
    particles.append({
        'features':features, 'n_neurons':n_neurons, 'params':params})
  return particles
```

The fitness of each particle is evaluated to determine its effectiveness in contributing to the desired outcomes. This fitness calculation guides the optimization process. The fitness function is calculated as follows:

$$Fit = \alpha CLErr + \beta \frac{ft}{FT} + \gamma \frac{hd}{HD} \tag{8}$$

where $CLErr$ represents the error rate in classifying the RWN network, ft indicates the number of features selected, FT represents the overall count of features in the dataset, hd denotes the count of hidden neurons set by the optimizer and HD is the maximum allowable number of neurons in the RWN. The parameters α, β and γ manage the impact of weights, aiming to enhance the reduction rate of features, curtail RWN complexity, and diminish the quantity of chosen features.

```
def compute_fitness(particle, X_train, y_train, X, y, dataset_features, alpha, beta, gamma,
    threshold):
  # Compute the classification error rate
  pred = compute_pred(particle, X_train, y_train, X, threshold)
  error_rate = compute_error_rate(pred, y)
  # Number of features identified
  ft = np.sum(list(particle['features'].values()))
  # Count of hidden neurons set by the optimizer
  hd = int(particle['n_neurons'], 2)
  # Maximum allowable number of neurons in the RWN
  HD = 2**(len(particle['features'])+1) - 1
  # Compute fitness
  return alpha * error_rate + beta * (ft / dataset_features) + gamma * (hd / HD)
```

3.3.1. Attribute Ranking using IG Technique

In order to establish a suitable threshold, the standard deviation of IG values is calculated, a common practice for precise threshold determination [8] [9]. Attributes exceeding or meeting this threshold are retained, while those falling below it are discarded. This step ensures that only the most impactful attributes move forward.

```
ig_values = {}
for col in X_train.columns:
  ig_values[col] = InformationGain(X_train[col], y_train)
  print(f"{col}: {ig_values[col]}")
# Compute the standard deviation of the Information Gain (IG) values
threshold = np.std(list(ig_values.values()))
print(f"\nThreshold: {threshold}")
# Select attributes exceeding or meeting the threshold
ig_attributes = [col for col, ig in ig_values.items() if ig >= threshold]
print(f"Attributes exceeding or meeting the threshold: {ig_attributes}")
```

3.3.2. Wrapper CSO Algorithm with RWN Learning

The top-ranked attributes, identified through IG, are integrated into the wrapper CSO algorithm, with the RWN serving as the learning algorithm. The CSO's primary objective is to iteratively discover optimal subsets of features, achieved through a sequence of generations.
In the developed implementation, the RWN employs a pre-allocated maximum-sized zero weight matrix to enhance flexibility and computational efficiency. This strategic decision offers several advantages. Firstly, it accommodates the dynamic addition of new features. Secondly, by avoiding frequent memory reallocations, it significantly improves the algorithm's computational performance. Lastly, it provides a clear and intuitive representation of the feature-neuron relationship. During the update process of the CSO particle positions, the subset of features can change, necessitating a weight matrix with dimensions that can accommodate any possible feature subset. By initializing a matrix of dimensions equal to the maximum possible size and initially setting only weights corresponding to selected features using the uniform Xavier initialization, the RWN can dynamically adapt to any feature subset encountered by the CSO algorithm. Furthermore, the velocity update formula for the RWN weights within the CSO algorithm requires weight matrices of consistent dimensions for the velocity calculation. This is crucial as the update formula incorporates the current particle's weight matrix, the competitor's weight matrix and the mean weight matrix of the population. By defining a weight matrix of the maximum size and initializing only the relevant rows, we maintain compatibility with the CSO algorithm's velocity update requirements. This approach facilitates seamless updates and ensures that the optimization process remains consistent and effective, regardless of the changing feature subsets during the iterations.

```
def uniform_Xavier_initialization(input_size, output_size):
  limit = np.sqrt(6 / (input_size + output_size))
  return np.random.uniform(-limit, limit, (input_size, output_size))

def initialize_RWN_params(features, n_neurons):
  params = list()
  # Create a zero matrix of the max size
  max_n_neurons = 2**(len(features)) - 1
  W = np.zeros((len(features), max_n_neurons))
  # Initialize only the selected portion with random weights
  mask = features==1
  W[mask, :n_neurons] = uniform_Xavier_initialization(np.sum(features), n_neurons)
  # Create a zero vector of the max size
  b = np.zeros((max_n_neurons, 1))
  # Initialize only the selected portion with random weights
  b[:n_neurons] = np.random.randn(n_neurons, 1)
  params.append(W)
  params.append(b)
  return params
```

3.3.3. CSO Particle Initialization and Competition

The CSO Randomly initializes the number of individuals for the population, where candidate feature subsets are encoded as particles. Consequently, population is defined by enumerating all possible subsets of top-ranked features (combinations without repetition), setting a maximum size of 50, as suggested in the paper.

```
def population_generator(ig_attributes, max_size=50):
  population = list(
      chain.from_iterable(
```

```
        combinations(ig_attributes, r) for r in range(1, len(ig_attributes)+1)
        ))
  if len(population) > max_size:
    # Randomly select particles from the population within the max size limit
    idxs = np.random.choice(len(population), max_size, False)
    return [population[i] for i in idxs]
  return population
```

The population is divided into two equal parts, each comprising k/2 particles. The ensuing pairwise competition determines winners and losers among particles. Winners advance to the next generation, while losers undergo updates before moved to the next generation. The subsequent stage entails training diverse RWNs using each particle, followed by the computation of the fitness value for each feature subset. Aims to refine the particle population iteratively.
The CSO algorithm's concepts of position and velocity were implemented by defining three positions and velocities for each particle: one for feature subset selection and two for the RWN configuration (one for weights and one for biases). These were initialized as follows:

```
velocities = [
    [
        np.random.randn(len(ig_attributes)),
        np.random.randn(*particle['params'][0].shape),
        np.random.randn(*particle['params'][1].shape)
    ] for particle in particles
]
```

The evaluation and update of particles were executed in parallel using Joblib, leveraging multiple processors to enhance computational efficiency and reduce execution time.

```
def competition(ig_attributes, particles, velocities, X_train, y_train, X_valid, y_valid,
    dataset_features, alpha, beta, gamma, phi, threshold):
  # Randomly shuffle the particles and velocities
  particles, velocities = shuffle(particles, velocities)
  majority_features, mean_params = compute_mean_particle(particles)
  next_gen_particles = []
  next_gen_vel = []
  # Execute the evaluation and update of particles in parallel
  results = Parallel(n_jobs=-1)(
      delayed(evaluate_particles)(
          ig_attributes,
          particles[i], particles[i+1], velocities[i], velocities[i+1],
          X_train, y_train, X_valid, y_valid,
          dataset_features, alpha, beta, gamma, phi,
          majority_features, mean_params,
          threshold
        ) for i in range(0, len(particles)-1, 2))
  # Extract the results
  for result in results:
    next_gen_particles.extend(result[0])
    next_gen_vel.extend(result[1])
  # If there is an odd number of particles, add the last one to the winners list
  if len(particles) % 2 != 0:
    next_gen_particles.append(particles[-1])
    next_gen_vel.append(velocities[-1])
  return next_gen_particles, next_gen_vel
```

The update of the losing particle is carried out in accordance with the previously outlined CSO algorithm formulas:

```
def update_loser(ig_attributes, velocity, winner, loser, majority_features, mean_params, phi,
    threshold):
  # Matrix of vectors with randomly generated numbers sampled from the range [0,1)
  R = np.random.rand(3)
  # Compute the velocity update for feature selection in the loser particle
  X_wi = np.array(list(winner['features'].values()))
  X_li = np.array(list(loser['features'].values()))
  vel_feature = R[0]*velocity[0] + R[1]*(X_wi - X_li) + R[2]*phi*(majority_features-X_li)
  X_li_updated = (sigmoid(X_li + vel_feature) > threshold).astype(int)
  features = {}
  for i, col in enumerate(ig_attributes):
    features[col] = X_li_updated[i]
  n_neurons = ''.join(map(str, features.values()))
  # Compute the velocity update for the weights in the RWN configuration
  X_wi = winner['params'][0]
  X_li = loser['params'][0]
  vel_weights = R[0]*velocity[1] + R[1]*(X_wi - X_li) + R[2]*phi*(mean_params[0]-X_li)
  weights = X_li + vel_weights
```

```
  # Compute the velocity update for the biases in the RWN configuration
  X_wi = winner['params'][1]
  X_li = loser['params'][1]
  vel_biases = R[0]*velocity[2] + R[1]*(X_wi - X_li) + R[2]*phi*(mean_params[1]-X_li)
  biases = X_li + vel_biases
  particle = {'features':features, 'n_neurons':n_neurons, 'params':[weights, biases]}
  return particle, [vel_feature, vel_weights, vel_biases]
```

where the average position of the population's particles $\overline{X}_i(t)$ is determined by computing the majority vote for the selected subset of features and by calculating the mean value for each parameter within the weights and biases matrices.

```
def compute_mean_particle(particles):
  # Compute the majority vote for selected features
  data = np.array([list(particle['features'].values()) for particle in particles])
  # Compare counts to half the number of rows to determine majority
  majority_features = (np.sum(data, axis=0) > data.shape[0] / 2).astype(int)
  # Compute the mean of the parameters for the particles
  mean_weights = np.mean([particle['params'][0] for particle in particles], axis=0)
  mean_biases = np.mean([particle['params'][1] for particle in particles], axis=0)
  mean_params = [mean_weights, mean_biases]
  return majority_features, mean_params
```

3.3.4. Iterative Refinement

The methodology persists iteratively until a predefined maximum iteration count is reached. Throughout this process, the wrapper CSO algorithm continues its pursuit of the best feature subset and corresponding RWN configuration.

3.4. Hyperparameters tuning

The hyperparameters α, β, γ and φ were optimized through a 10-fold cross-validation approach, by determining the combination that yields the highest average accuracy score over the 10 folds.

```
def hyperparameters_tuning(ig_attributes, X, y, dataset_features, categorical_cols, max_iter,
    threshold):
  # SMOTE
  X_train, y_train = perform_SMOTE(X, y, categorical_cols)
  values = [1e-3, 0.1, 1, 10, 100]
  # Provides train/test indices to split data in train/test sets
  kf = KFold(n_splits=10, shuffle=True, random_state=seed)
  # Define alpha search space
  param_grid = [(alpha, 0.5, 0.5, 0.5) for alpha in values]
  print('Tuning alpha ...')
  best_score, best_params = perform_tuning(
      kf, param_grid, -np.inf, None,
      ig_attributes,
      X, y, dataset_features, categorical_cols, max_iter, threshold)
  # Define beta search space
  param_grid = [(best_params[0], beta, 0.5, 0.5) for beta in values]
  print('Tuning beta ...')
  best_score, best_params = perform_tuning(
      kf, param_grid, best_score, best_params,
      ig_attributes,
      X, y, dataset_features, categorical_cols, max_iter, threshold)
  # Define gamma search space
  param_grid = [(best_params[0], best_params[1], gamma, 0.5) for gamma in values]
  print('Tuning gamma ...')
  best_score, best_params = perform_tuning(
      kf, param_grid, best_score, best_params,
      ig_attributes,
      X, y, dataset_features, categorical_cols, max_iter, threshold)
  # Define phi search space
  param_grid = [(best_params[0], best_params[1], best_params[2], phi) for phi in values]
  print('Tuning phi ...')
  best_score, best_params = perform_tuning(
      kf, param_grid, best_score, best_params,
      ig_attributes,
      X, y, dataset_features, categorical_cols, max_iter, threshold)
  print("Best parameters: ", best_params)
  print("Best fitness score: ", best_score)
  return best_params
```

4. Performance Evaluation

4.1. Performance Metrics

The evaluation of the involved models' performance is based on a series of metrics derived from the confusion matrix. These metrics utilize parameters such as true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN):

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{9}$$

$$Sensitivity = \frac{TP}{TP + FN} \tag{10}$$

$$Specificity = \frac{TN}{TN + FP} \tag{11}$$

$$G - mean = \sqrt{Sensitivity \times Specificity} \tag{12}$$

$$Precision = \frac{TP}{TP + FP} \tag{13}$$

$$Recall = \frac{TP}{TP + FN} \tag{14}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{15}$$

Finally, the AUC (Area Under the Curve), assessing the models' differentiation capability via the ROC curve.

4.2. Experiment I: Comparisons performance between HybridIG-CSO, RWN with filter approach, and RWN with CSO

In this experiment, the HybridIG-CSO method was evaluated by comparing it with three distinct techniques: the classical RWN, RWN with a filter-based approach (IG-RWN) and manually tuned CSO-RWN. The performance of the HybridIG-CSO method was assessed against the other approaches using six different metrics: Accuracy, Precision, Recall, AUC, F1 and G-mean.

Dataset	Algorithm	Accuracy	Precision	Recall	AUC	F1	G-Mean
D_1	Classic-RWN	0.5393	0.5094	0.6429	0.5464	0.5684	0.5451
	IG-RWN	0.7528	0.9623	0.7183	0.7034	0.8226	0.6540
	CSO-RWN	0.6854	0.8302	0.6984	0.6512	0.7586	0.6261
	HybridIG-CSO	**0.7528**	**0.9623**	**0.7183**	**0.7034**	**0.8226**	**0.6540**
D_2	Classic-RWN	0.9567	0.8765	0.8765	0.9251	0.8765	0.9239
	IG-RWN	0.9567	0.9383	0.8352	0.9495	0.8837	0.9494
	CSO-RWN	**0.9827**	0.9259	**0.9740**	**0.9603**	**0.9494**	**0.9597**
	HybridIG-CSO	0.9610	**0.9506**	0.8462	0.9569	0.8953	0.9569
D_3	IG-RWN	**0.7831**	0.5494	**0.5126**	0.6998	0.5303	0.6834
	HybridIG-CSO	0.7793	**0.5753**	0.5044	**0.7066**	**0.5375**	**0.6943**
D_4	IG-RWN	**0.9784**	**0.8941**	**0.7525**	**0.9387**	**0.8172**	**0.9376**
	HybridIG-CSO	0.9701	0.8824	0.6696	0.9288	0.7614	0.9276

Table 3: Performance experiment I on a separate test set.

Due to the limitations of the free usage plans on Colab and Kaggle, it was necessary to significantly reduce the number of features, as the initialization of the RWN exhausted all the available RAM. Consequently, it was not feasible to implement the classical RWN and the manually tuned CSO-RWN for the last two datasets, as these

require the use of the full dataset's features. Regarding Dataset 4, downsampling was required. Specifically, the majority class was downsampled.

Dataset	Algorithm	Accuracy	Precision	Recall	AUC	F1	G-Mean
D_1	Classic-RWN	0.4904	0.4984	0.4933	0.4912	0.4915	0.4857
	IG-RWN	0.7332	**0.9540**	0.6633	0.7350	0.7809	0.7004
	CSO-RWN	**0.7790**	0.8398	**0.7513**	**0.7792**	**0.7913**	**0.7754**
	HybridIG-CSO	0.7018	0.8960	0.6450	0.7023	0.7474	0.6713
D_2	Classic-RWN	0.9720	0.9722	0.9721	0.9722	0.9721	0.9722
	IG-RWN	0.9730	0.9781	0.9683	0.9731	0.9732	0.9730
	CSO-RWN	**0.9863**	**0.9892**	**0.9837**	**0.9865**	**0.9864**	**0.9865**
	HybridIG-CSO	0.9707	0.9755	0.9665	0.9708	0.9709	0.9707
D_3	IG-RWN	0.6980	**0.5582**	**0.7749**	**0.6980**	**0.6490**	**0.6839**
	HybridIG-CSO	**0.6897**	0.5490	0.7640	0.6897	0.6389	0.6752
D_4	IG-RWN	**0.9651**	**0.9502**	**0.9794**	**0.9651**	**0.9645**	**0.9649**
	HybridIG-CSO	0.9212	0.9253	0.9339	0.9211	0.9268	0.9170

Table 4: Performance experiment I using 10-fold cross-validation.

4.3. Experiment II: Comparison with other classifiers

In this experimental evaluation, the effectiveness of the HybridIG-CSO is assessed in the context of fraud classification, by comparing its performance against other widely employed algorithms typically used as induction techniques in feature selection wrapper-based methods, namely Naïve Bayes (NB), Random Forest (RF) and Support Vector Machine (SVM).

Dataset	Classifier	Accuracy	Precision	Recall	AUC	F1	G-Mean
D_1	NB	0.7528	0.9623	0.7183	0.7034	0.8226	0.6540
	RF	0.7528	0.9057	**0.7385**	**0.7167**	0.8136	**0.6914**
	SVM	0.7528	0.9623	0.7183	0.7034	0.8226	0.6540
	HybridIG-CSO	**0.7528**	**0.9623**	0.7183	0.7034	**0.8226**	0.6540
D_2	NB	0.9113	0.7284	0.7564	0.8393	0.7421	0.8319
	RF	**0.9762**	0.9753	**0.8977**	**0.9758**	**0.9349**	**0.9758**
	SVM	0.9524	**0.9877**	0.7921	0.9663	0.8791	0.9660
	HybridIG-CSO	0.9610	0.9506	0.8462	0.9569	0.8953	0.9569
D_3	NB	0.6769	**0.6630**	0.3734	0.6719	0.4777	0.6719
	RF	0.8047	0.4576	**0.5781**	0.6809	0.5109	0.6433
	SVM	0.7556	0.6022	0.4628	0.7009	0.5234	0.6939
	HybridIG-CSO	**0.7793**	0.5753	0.5044	**0.7066**	**0.5375**	**0.6943**
D_4	NB	0.9886	0.8353	**0.9467**	0.9163	0.8875	0.9127
	RF	**0.9886**	0.8706	0.9136	0.9329	0.8916	0.9309
	SVM	0.9746	**0.8941**	0.7103	**0.9366**	0.7917	**0.9357**
	HybridIG-CSO	0.9701	0.8824	0.6696	0.9288	0.7614	0.9276

Table 5: Performance experiment II on a separate test set.

Dataset	Classifier	Accuracy	Precision	Recall	AUC	F1	G-Mean
D_1	NB	0.7248	0.9492	0.6566	0.7267	0.7746	0.6907
	RF	**0.8044**	0.8334	**0.7908**	**0.8050**	**0.8106**	**0.8040**
	SVM	0.7248	**0.9590**	0.6550	0.7269	0.7765	0.6871
	HybridIG-CSO	0.7018	0.8960	0.6450	0.7023	0.7474	0.6713
D_2	NB	0.8413	0.7125	0.9588	0.8411	0.8173	0.8311
	RF	**0.9897**	**0.9955**	**0.9841**	**0.9898**	**0.9898**	**0.9898**
	SVM	0.9673	0.9724	0.9628	0.9673	0.9675	0.9672
	HybridIG-CSO	0.9707	0.9755	0.9665	0.9708	0.9709	0.9707
D_3	NB	0.6724	0.6657	0.6748	0.6724	0.6702	0.6723
	RF	**0.8768**	**0.8525**	**0.8960**	**0.8768**	**0.8737**	**0.8764**
	SVM	0.7017	0.5990	0.7538	0.7017	0.6675	0.6941
	HybridIG-CSO	0.6876	0.5455	0.7621	0.6876	0.6358	0.6727
D_4	NB	0.9160	0.8358	0.9955	0.9160	0.9086	0.9124
	RF	**0.9938**	**0.9906**	**0.9969**	**0.9938**	**0.9937**	**0.9937**
	SVM	0.9619	0.9458	0.9773	0.9619	0.9613	0.9618
	HybridIG-CSO	0.9212	0.9253	0.9339	0.9211	0.9268	0.9170

Table 6: Performance experiment II using 10-fold cross-validation.

5. Conclusions

Before presenting the obtained results, it is important to acknowledge that accuracy, while commonly used metrics in classification tasks, is not a reliable index of the quality of the trained model in case of imbalanced datasets or when the importance of wrongly predicting positive-class samples is different from wrongly predicting negative-class samples. It can give a misleading picture of the model's performance, often favoring the majority class.
More suitable metrics in such scenarios are the AUC, F1 score and G-Mean, which provide a more balanced assessment by considering both precision and recall or by emphasizing the balance between sensitivity and specificity, thereby offering a more comprehensive evaluation of the model's performance on imbalanced data.
The performance achieved in these more robust metrics closely aligns with the results reported in the referenced paper, except for the Classic-RWN model in the first dataset in Experiment I, which exhibited notably worse performance.

The performance metrics obtained on a separate test set indicate that the HybridIG-CSO model outperforms the other models across all metrics and datasets, except for the second and fourth datasets. In these cases, the CSO-RWN and IG-RWN model, respectively, perform better, although the HybridIG-CSO demonstrates very close performance and superior Precision.
However, due to the limitations of the free usage plans on Colab and Kaggle, it was not feasible to implement the classical RWN and the manually tuned CSO-RWN for the last two datasets.
In the evaluation of performance using 10-fold cross-validation, which is more characterized by independence from the choice of samples, the CSO-RWN outperforms the HybridIG-CSO across all metrics. In contrast, for the last two datasets, IG-RWN emerges as the best-performing model among the two tested.
In the second experiment, the HybridIG-CSO outperforms the other classifiers in almost all metrics on the first and third datasets on a separate test set. In all other cases, including using 10-fold cross-validation, the ensemble learning method, Random Forest, which combines the output of multiple decision trees to reach a single result, emerges as the best classifier.
The hyperparameters α, β, γ and φ were optimized through a 10-fold cross-validation approach, by determining the combination that yields the highest average score over the 10 folds.
Obviously, the performance of HybridIG-CSO could be further optimized by expanding the search space for hyperparameters or optimizing with respect to other metrics such as F1, G-mean or AUC, increasing the number of iterations, testing different thresholds for filter-based feature selection using Information Gain, or testing a higher prediction threshold for the sigmoid function.
Ensemble learning with HybridIG-CSO models, using different configurations, could further enhance classification performance.
However, the most noteworthy aspect of this project is the efficiency of the method proposed in the paper.

With a significantly simplified training process compared to conventional gradient descent methods, it achieves performance, on a complex machine learning task as credit card fraud detection, that is almost comparable to an ensemble learning method like Random Forest, and in some cases, on a separate test set, even surpasses it. Particularly remarkable is the use of the hybrid approach for feature selection, combining Information Gain and Competitive Swarm Optimization, which enabled the model to be tested on large datasets, overcoming resource limitations, and outperforming the other models on a smaller dataset like the first one.

References

[1] Enas Faisal, Hadeel Ahmad, and Rawan Zaghloul. Efficient credit card fraud detection using evolutionary hybrid feature selection and random weight networks. *International Journal of Data and Network Science*, 09 2024. doi: 10.5267/j.ijdns.2023.9.009.

[2] Rejwan Bin Sulaiman, Vitaly Schetinin, and Paul Sant. Review of machine learning approach on credit card fraud detection. *Human-Centric Intelligent Systems*, 2, 05 2022. doi: 10.1007/s44230-022-00004-0.

[3] Mohammad Masoud, Yousef Jaradat, Esraa Rababa, and Ahmad Manasrah. Turnover prediction using machine learning: Empirical study. *International Journal of Advances in Soft Computing & Its Applications*, 13(1), 2021.

[4] Rafael Lima and Adriano Pereira. Feature selection approaches to fraud detection in e-payment systems. volume 278, pages 111–126, 02 2017. ISBN 978-3-319-53675-0. doi: 10.1007/978-3-319-53676-7_9.

[5] Naoufal Rtayli and Nourddine Enneya. Enhanced credit card fraud detection based on svm-recursive feature elimination and hyper-parameters optimization. *Journal of Information Security and Applications*, 55: 102596, 12 2020. doi: 10.1016/j.jisa.2020.102596.

[6] Ran Cheng and Yaochu Jin. A competitive swarm optimizer for large scale optimization. *IEEE transactions on cybernetics*, 45, 05 2014. doi: 10.1109/TCYB.2014.2322602.

[7] Guang-Bin Huang, Qin-Yu Zhu, and Chee Siew. Extreme learning machine: A new learning scheme of feedforward neural networks. volume 2, pages 985 – 990 vol.2, 08 2004. ISBN 0-7803-8359-1. doi: 10.1109/IJCNN.2004.1380068.

[8] J Roseline, GBSR Naidu, Samuthira Pandi Velusamy, S Rajasree, and Dr.N. Mageswari. Autonomous credit card fraud detection using machine learning approach. *Computers and Electrical Engineering*, 102:108132, 09 2022. doi: 10.1016/j.compeleceng.2022.108132.

[9] Maria Prasetiyowati, Nur Maulidevi, and Kridanto Surendro. Determining threshold value on information gain feature selection to increase speed and prediction accuracy of random forest. *Journal of Big Data*, 8, 06 2021. doi: 10.1186/s40537-021-00472-4.