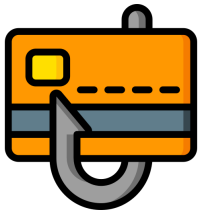
POLITECNICO
MILANO 1863

Numerical Analysis for Machine Learning

Credit card fraud detection



Filippo Lazzati - 10629918
Margherita Musumeci - 10600069

Professor
Edie Miglio

January 31, 2022

Contents

1 Introduction **2**

2 Dataset overview **3**

3 Preprocessing **4**

4 Algorithms **6**
4.1 Logistic regression 6
4.2 K-Nearest neighbours 10
4.3 Naive Bayes 12

5 Performances evaluation **16**

6 Additional analysis **21**
6.1 Neural networks 21
6.2 Cross-validation 25
6.3 Logistic Regression with sklearn.linear_model library 26

7 Conclusion **27**

8 References **29**
8.1 Papers and books 29
8.2 Tools 29
8.3 Websites 29

1 Introduction

Payment card fraud is a major challenge for business owners, payment card issuers, and transactional services companies, causing every year substantial and growing financial losses. Credit card fraud detection, a branch of Internet security, increases its importance in the last years because of the spreading of transactions than in the past.

The goal of the project is to apply different machine learning algorithms to the dataset containg transactions made by credit cards in September 2013 by European cardholders. The dataset is available at this link.

The information has to be analyzed exploiting the methods adopted in the paper[1] provided to us:

- Logistic Regression;
- KNN (k-nearest neighbours);
- Naive Bayes.

However, we have decided to try an additional algorithm. Using an artificial neural network, we attempted to improve the predictions.

The report is organized as follows:

1. **Section 2:** describes the dataset and includes some Python code for analyzing its main features;
2. **Section 3:** shows the use of random under sampling (RUS) to preprocess the dataset in order to cope with unbalanced data;
3. **Section 4:** demonstrates how we implemented the algorithms adopted in the paper;
4. **Section 5:** provides analysis of the results obtained by applying the algorithms of section 4;
5. **Section 6:** provides an additional analysis of the problem which is not present in the paper; in particular it covers the adoption of the neural networks to solve this problem and the use of K-fold cross-validation to better estimate the performances of the algorithms;
6. **Section 7:** sums up the results of this project;
7. **Section 8:** links to the references and the material used to solve the project.

[1] Itoo, F., Meenakshi Singh, S. Comparison and analysis of logistic regression, Naive Bayes and KNN machine learning algorithms for credit card fraud detection. Int. j. inf. tecnol. 13, 1503–1511 (2021). https://doi.org/10.1007/s41870-020-00430-y.

2 Dataset overview

The dataset provided (https://www.kaggle.com/mlg-ulb/creditcardfraud) contains data about 284807 credit card transactions. Such transactions date back to September 2013 and were performed by European cardholders. Each transaction is labeled as fraudulent (1) or genuine (0). Unfortunately, due to confidentiality issues, the original features of the transactions (like credit limit, gender, marital status, credit amount, ... etc.) could not be provided. This is why all the features of the samples are the result of a principal component transformation, and thus they are called $V1, V2, ..., V28$. The structure of the dataset is shown in table 1:

Time	V1	V2	...	V28	Amount	Class
...	...	...	...	...	...	...

Table 1: Overview of the dataset.

Feature `Time` contains the seconds elapsed between each transaction and the first transaction in the dataset, while `Amount` is the transaction amount and the last column, `Class` takes on value 0 if it is a genuine transaction and 1 if it is fraudulent.
We have exploited Panda Python library for an overview of the data:

1. we have imported the library and the dataset:

```
import pandas as pd
dataset = pd.read_csv('creditcard.csv')
```

2. we have removed records with missing entries:

```
dataset = dataset.dropna()
```

3. we have looked at an overview of the dataset and at some statistics (see the notebook for more details):

```
dataset.head()
dataset.info()
dataset.describe()
dataset.corr()
fig, ax = plt.subplots(figsize=(40,40))
sns.heatmap(dataset.corr(),ax = ax,annot = True,
            cmap='vlag_r',vmin=-1,vmax=1)
```

3 Preprocessing

After getting an idea of the dataset, we can begin working on the data. The first step (according also to the reference paper) is pre-processing of the data. This involves four different tasks:

1. **normalize**, which aims to avoid that differences in the ranges of values that features can take on can influence the predictions;
2. **shuffle it**, to avoid that the order with which data has been stored can influence the prediction;
3. **divide into training set/test set**, such that we have also the chance to validate our predictor;
4. **random under sampling**, to manage the skewed (or unbalanced data). As a matter of fact, only 0.172% of the transactions are fraudulent, and it is clear that the dataset is unbalanced; therefore, the application of random under sampling helps to equalize this disparity.

The code is:

1. to normalize the dataset, we subtract the mean and then divide by the standard deviation of the whole dataset (except for the last column, which represents the output):

```
temp = data.iloc[:,:-1]
mean = temp.mean()
std = temp.std()
data_normalized = (temp - mean) / std
data_normalized['Class'] = data.iloc[:,-1]
```

2. to shuffle the dataset, we exploit `Numpy.random`:

```
np.random.seed(0) # set seed to 0 for reproducibility
np.random.shuffle(data_normalized_np)
```

3. to divide into training set and test set, we use (the paper suggests to use 70% for the training set and 30% for the test set):

```
fraction_validation = 0.3
num_train = int(data_normalized.shape[0] * (1 - fraction_validation))
train_set = data_normalized[:num_train,:]
test_set = data_normalized[num_train:,:]
```

4. to apply random under sampling, we have written this function, which receives in input the desired ratio (defined as: $ratio := fraud/non\_fraud$) between fraudulent and non-fraudulent transactions and the training set on which applying it. Such function randomly pops samples from the training set which represent non-fraudulent transactions until the ratio is reached (see the notebook for comments and explanation of each row):

```
def RUS(ratio, training_set):
train_fraud = list()
train_not_fraud = list()
for sample in training_set:
    if sample[30] == 1:
        train_fraud.append(sample)
    else:
        train_not_fraud.append(sample)
new_length_train_not_fraud = int(len(train_fraud) / ratio)
np.random.seed(0)
np.random.shuffle(train_not_fraud)
train_not_fraud_reduced = train_not_fraud[0:new_length_train_not_fraud]
train_fraud.extend(train_not_fraud_reduced)
np.random.seed(0)
np.random.shuffle(train_fraud)
return train_fraud
```

The paper adopts three training sets with different ratios, and so we do:

```
ratio1 = 1 # 50:50
ratio2 = 34/66 # 34:66
ratio3 = 1/3 # 25:75
training_set1 = RUS(ratio1, train_set)
training_set2 = RUS(ratio2, train_set)
training_set3 = RUS(ratio3, train_set)
```

Now we have the dataset ready to be analysed through the algorithms of the next section.

4 Algorithms

The algorithms adopted in the paper are logistic regression, k-nearest neighbors and naive Bayes. This is a binary classification problem and all of them can be used to solve it. The main goal is to establish which of them is best suited for this application.

4.1 Logistic regression

Logistic regression, despite its name, is one of the most used and powerful machine learning algorithm for linear and binary classification problems. It can be seen as a neuron with multiple inputs and one output, that undergoes a threshold function before being considered as the prediction:

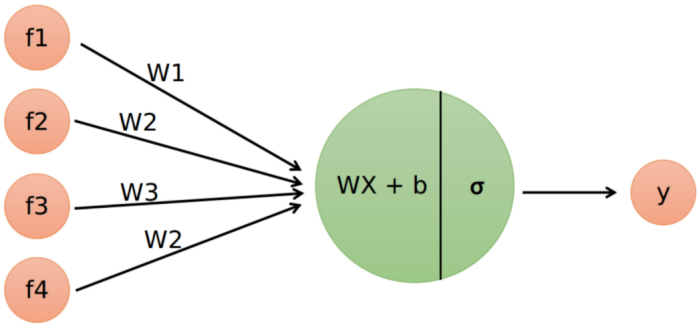


Figure 1: Illustration of logistic regression neuron.

So, we have some inputs $x_1, x_2, ..., x_n$ to which we apply weights $\mathbf{w}$ and bias b; next, we apply the activation function σ, which is the sigmoid: $\sigma(z) = 1/(1+e^{-z})$. Since the output of the sigmoid function belongs to $[0, 1]$, it can be interpreted as a likelihood, thus the final prediction can be retrieved through a threshold function:

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The loss function can be found by applying the maximum likelihood method to $p(\mathbf{y}|\mathbf{x}, \mathbf{w})$, and then negate the obtained log-likelihood function in order to have a loss function to minimize:

$$J(\mathbf{w}) = \sum_{i=1}^{n} \Big[-y^{(i)} \log(\sigma(z^i)) - (1 - y^{(i)}) \log(1 - \sigma(z^i)) \Big] \tag{2}$$

It can be minimized through the stochastic gradient descent (SGD) to obtain the parameters to use for making the predictions. Gradient descent is an iterative algorithm, that starts from a random point on a function and travels down its

slope in steps until it reaches the lowest point of that function. In order to find the slope of the objective function, we compute the gradient of the function with respect to parameters. SGD randomly picks some data (mini-batch) from the whole dataset and evaluates the gradient only on that subsets. This method not only reduces the computations but concur to avoid overfitting.

The "learning rate" is a flexible parameter applied in all Gradient Descent methods to determine the step size at each iteration while moving toward a minimum of a function. The learning rate heavily influences the convergence of the algorithm. Larger learning rates make the algorithm take huge steps down the slope and it might jump across the minimum point thereby missing it. So, it is always good to stick to a low learning rate.

N.B.: It should be remarked that overfitting has to be avoided, because it makes the algorithm loosing the ability to generalize (this problem is enhanced because of the adoption of random under sampling, which shrinks the training set). The following code (see the notebook for a more detailed explanation):

1. defines the function used to initialize the weights. We decided to initialize the parameters with random real numbers taken from a Gaussian distribution centered in zero with standard deviation equals to 0.1:

```
def initialize_params(weight_size):
  rgen = np.random.RandomState(seed = 1)
  return rgen.normal(loc=0.0, scale=0.1, size=weight_size )
```

2. defines the sigmoid function:

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

3. defines the loss function. We introduced a *clip* function to limit the argument of the logarithm in order to avoid *log(0) = -inf* :

```
def loss(x,y,params,bias):
    return -np.sum( np.log(
        y * np.clip(sigmoid(np.dot(x, params) + bias),1e-15,1-1e-15)
        + (1-y) * np.clip(1-sigmoid(np.dot(x, params) + bias),1e-15,1-1e-15)))
```

4. implements the stochastic gradient descent method for training the model. We used batches of 100 samples each and a decadent learning rate starting from 2e-4:

```
def fit(x,y):
  num_epochs = 500
  learning_rate_min = 5e-5
  learning_rate_max = 2e-4
  learning_rate_decay = 2000

  batch_size = 100
  num_train = x.shape[0]
```

```
    params = initialize_params(x.shape[1])
    gradients = list()
    bias = 0

    train_history = [loss(x, y, params,bias)]

    t0 = time.time()
    for epoch in tqdm(range(num_epochs)):

      minibatch = np.random.choice(num_train, batch_size)
      x_mini = x[minibatch,:]
      y_mini = y[minibatch]

      value = np.dot(x_mini, params) + bias
      y_pred = sigmoid(value)
      gradients =  np.dot(x_mini.T, (y_pred - y_mini))
      bias_gradients = np.sum(y_pred - y_mini)

      learning_rate = max(learning_rate_min, learning_rate_max *
                          (1 - epoch/learning_rate_decay))
      for i in range(len(params)):

        params[i] -= learning_rate * gradients[i]

      bias -= learning_rate * bias_gradients
      train_history.append(loss(x, y, params,bias))

    print('train loss: %1.3e' % train_history[-1])
    plt.loglog(train_history, label = 'train loss')

    return [params,bias]
```

5. defines the function that makes the predictions:

```
def prediction(x,params):
    w = params[0]
    bias = params[1]

    z = np.dot(x, w) + bias
    phi = sigmoid(z)

    pred = list()

    for i in range(phi.shape[0]):
        pred.append(1 if phi[i] >= 0.5 else 0)

    return pred
```

6. defines the function that given the training and the test set applies the algorithm:

```
def logistic_regression(train_set,valid_set):
    x = train_set[:,:-1]
    y = train_set[:,-1]

    params = fit(x,y)

    x_valid = valid_set[:,:-1]
    pred = prediction(x_valid,params)

    return pred
```

7. Applies the algorithm:

```
predictionsLR1 = logistic_regression(training_set1,valid_set)
predictionsLR2 = logistic_regression(training_set2,valid_set)
predictionsLR3 = logistic_regression(training_set3,valid_set)
```

4.2 K-Nearest neighbours

K-nearest neighbours (KNN) is a machine learning classifier rather different from the others seen. As a matter of fact, it is a lazy learner that tries to memorize the training dataset instead of learning it.
The idea behind KNN is quite simple: to provide a prediction for a new sample, we just have to compute the distance between the new sample and all the samples in the training set; next, we draw k samples from the training set that are the nearest to the new sample. At the end we count how many times each label is present in the k samples and the most present one will be our prediction for the new sample:

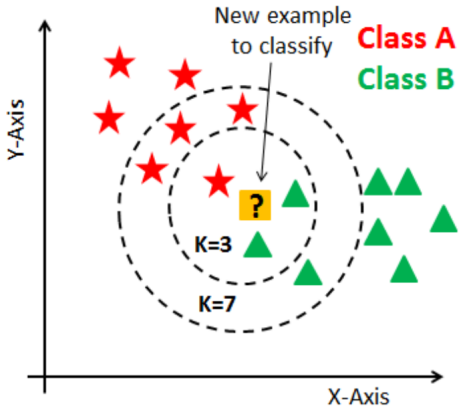


Figure 2: Illustration of KNN algorithm.

This approach has the advantage of immediately adapting as we collect new training data. Unfortunately, the main downside is its computational complexity, which grows linearly in the number of samples in the training set. A possible solution can be the adoption of efficient data structures, like k-d trees, but in case the training set is very big it is better to choose another technique. In this case we have decided to manually implement it without k-d trees, since the training set is not really big. It is important to select a proper K parameter and a proper distance (Euclidean, Manhattan, Minkowski, etc.) because they influence a lot the accuracy of the predictions.
Our implementation exploits Euclidean distance and looks for the neighbours linearly in the dimension of the training set; therefore, the complexity is linear in the number of elements of the training set.
The code is the following:

1. define the distance function between two samples (rows):

```
def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
```

```
        distance += (row1[i] - row2[i])**2
    return sqrt(distance)
```

2. define the neighbors function; it simply computes the distances between the sample and all the samples of the dataset and extracts the closest ones:

```
def get_neighbors(train_set, sample, k):
  distances = list()
  for train_row in train_set:
    dist = euclidean_distance(sample, train_row)
    distances.append((train_row, dist))

  distances.sort(key=lambda tup: tup[1])

  neighbors = list()
  for i in range(k):
    neighbors.append(distances[i][0])

  return neighbors
```

3. define the function that makes the prediction:

```
def predict_classification(training_set, sample, k):
  neighbors = get_neighbors(training_set, sample, k)
  output_values = [row[-1] for row in neighbors]
  prediction = max(set(output_values), key=output_values.count)

  return prediction
```

4. define the function that applies the algorithm; it receives in input both the training set and the test set and the hyperparameter `k`:

```
def k_nearest_neighbors(training_set, test_set, k):
        predictions = list()
        for row in test_set:
                output = predict_classification(training_set, row, k)
                predictions.append(output)

        return(predictions)
```

5. apply the algorithm over the three test sets suggested by the paper (after having random under sampled the test set for speed reasons):

```
k = 10
predictionsKNN1 = k_nearest_neighbors(training_set1, validation_set, k)
predictionsKNN2 = k_nearest_neighbors(training_set2, validation_set, k)
predictionsKNN3 = k_nearest_neighbors(training_set3, validation_set, k)
```

4.3 Naive Bayes

Naive Bayes is an algorithm based on the Bayes theorem $p(y_i|\mathbf{x}) = p(\mathbf{x}|y_i)p(y_i)/p(\mathbf{x})$, where y_i is the class sample i belongs to and $\mathbf{x} = [x_1, x_2, ..., x_n]$ is the vector containing the feature values for the sample. The theorem states that, in order to compute the probability that a certain sample $\mathbf{x}$ is of class y_i, we have to compute:

- $p(y_i)$, the probability of belonging to class y_i;
- $p(\mathbf{x})$, the probability of obtaining a certain sample $\mathbf{x}$;
- $p(\mathbf{x}|y_i)$, the probability that the sample, that we know that belongs to class y_i, has values $\mathbf{x}$ for its features.

The first probability ($p(y_i)$) is easy to compute: we just need to take the number of samples of a certain class and dividing it by the total number of samples in the training dataset. Unfortunately, the other two probabilities cannot be easily computed, but we have to make some assumptions:

1. the features $x_1, x_2, ..., x_n$ are independent of each other; in such a way, the denominator becomes a useless (normalization) constant $p(\mathbf{x} = p(x_1) * p(x_2) * ... * p(x_n)$ that can be omitted (we want to maximize $p(y_i|\mathbf{x})$, if we divide by a constant the maximum does not change). Moreover, it entails that: $p(\mathbf{x}|y_i)p(y_i) = p(x_1|y_i) * p(x_2|y_i) * ... * p(x_n|y_i)$;
2. the distribution of each $p(x_j|y_i)$ is known; in our case, we can assume that this distribution is Gaussian, therefore we can compute each of the $p(x_j|y_i)$ simply by using the PDF of a Gaussian whose parameters (mean and variance) are determined by computing the mean and the variance of the columns (features) after having divided the training set in sets characterized by having the same label (class).

The code is:

1. create a partition of all the samples by class (either fraud or non fraud) and store the partition in a new data structure, namely a dictionary where each key is the class and the value is the list of samples of that class:

```
def separate_by_class(dataset):
  separated = dict()

  for i in range(len(dataset)):
    vector = dataset[i]
    class_value = vector[-1] # the class is in the last column

    if (class_value not in separated):
      separated[class_value] = list()
```

```
    separated[class_value].append(vector)

  return separated
```

2. define the mean function:

```
def mean(numbers):

  return sum(numbers)/float(len(numbers))
```

3. define the standard deviation function:

```
def stdev(numbers):

  avg = mean(numbers)
  variance = sum([(x-avg)**2 for x in numbers]) / float(len(numbers)-1)
  return sqrt(variance)
```

4. define a function that computes mean, standard deviation and cardinality for each column of the dataset:

```
def summarize_dataset(dataset):
  summaries = [(mean(column), stdev(column), len(column))
                    for column in zip(*dataset)]

  del(summaries[-1])

  return summaries
```

5. define a function that computes mean, standard deviation and cardinality for each column of each set of the partition; now the dictionary returned has the same keys but, as values, the lists of triples mean-standard deviation-cardinality:

```
def summarize_by_class(dataset):
  separated = separate_by_class(dataset)

  summaries = dict()

  for class_value, rows in separated.items():
    summaries[class_value] = summarize_dataset(rows)

  return summaries
```

6. define a function that computes the Gaussian probability density function given mean and standard deviation:

```
def calculate_probability(x, mean, stdev):
```

```
  exponent = exp(-((x-mean)**2 / (2 * stdev**2 )))
  return (1 / (sqrt(2 * pi) * stdev)) * exponent
```

7. define a function for computing the class probabilities according to the rule:
$p(y_i|x_1, x_2, ..., x_n) = p(x_1|y_i)p(x_2|y_i)...p(x_n|y_i)p(y_i)$

```
def calculate_class_probabilities(summaries, sample):
  total_rows = sum([summaries[class_value][0][2] for class_value in summaries])

  probabilities = dict()

  for class_value, class_summaries in summaries.items():

    probabilities[class_value] = summaries[class_value][0][2]/float(total_rows)

    for i in range(len(class_summaries)):
      mean, stdev, count = class_summaries[i]
p(yi|x1,x2,...,xn)=p(yi)*p(x1|yi)*p(x2|yi)*...*p(xn|yi)
      probabilities[class_value] *= calculate_probability(sample[i], mean, stdev)

  return probabilities
```

8. define a function that makes the prediction for a sample given the summary (mean, standard deviation, cardinality) of each column:

```
def predict(summaries, sample):
        probabilities = calculate_class_probabilities(summaries, sample)
        best_label, best_prob = None, -1

        for class_value, probability in probabilities.items():
                if best_label is None or probability > best_prob:
                        best_prob = probability
                        best_label = class_value

        return best_label
```

9. define a function to apply the algorithm given the training set and the test set:

```
def naive_bayes(training_set, test_set):

  summarize = summarize_by_class(training_set)

  predictions = list()

  for row in test_set:
    output = predict(summarize, row)
```

```
        predictions.append(output)

    return(predictions)
```

10. apply the algorithm:

```
predictionsNB1 = naive_bayes(training_set1, valid_set)
predictionsNB2 = naive_bayes(training_set2, valid_set)
predictionsNB3 = naive_bayes(training_set3, valid_set)
```

5 Performances evaluation

According to the paper, the dataset is divided such that the 70% of the samples are used as training set and the remaining 30% as the validation set. Moreover, the training set has underwent RUS with three different ratios, namely 50:50, 34:66 and 25:75. The algorithms presented in section 4 have then be applied to all these three training under sampled set, and some metrics have been used to compare the results.
The metrics[2] adopted are:

- *Accuracy*, defined as $Accuracy := \frac{TP+TN}{TP+TF+FP+FN}$;
- *Sensitivity*, defined as $Sensitivity := \frac{TP}{TP+FN}$;
- *Specificity*, defined as $Specificity := \frac{TN}{FP+TN}$;
- *Precision*, defined as $Precision := \frac{TP}{TP+FP}$;
- *F-measure*, defined as $Fmeasure := \frac{TP}{TP+0.5(FP+FN)}$;
- *Area under curve (AUC)*, defined as $AUC := 0.5(Sensitivity+Specificity)$.

Our results are:

1. **Logistic Regression**

Parameters	**Ratio 50:50**	**Ratio 34:66**	**Ratio 25:75**
TP	128	124	124
TN	85125	85191	85252
FP	163	97	36
FN	27	31	31

Table 2: Predictions

Metrics	**Ratio 50:50**	**Ratio 34:66**	**Ratio 25:75**
Accuracy	99.8%	99.9%	99.9%
Sensitivity	82.6 %	80.0%	80.0%
Specificity	99.8%	99.9%	100.0%
Precision	44.0%	56.1%	77.5%
F_measure	57.4%	66.0%	78.7%
AUC	91.2%	89.9%	90.0%

Table 3: Metrics values

[2]The following definitions have been used: $TP := TruePositives$, $FP := FalsePositives$, $TN := TrueNegatives$, $FN := FalseNegatives$

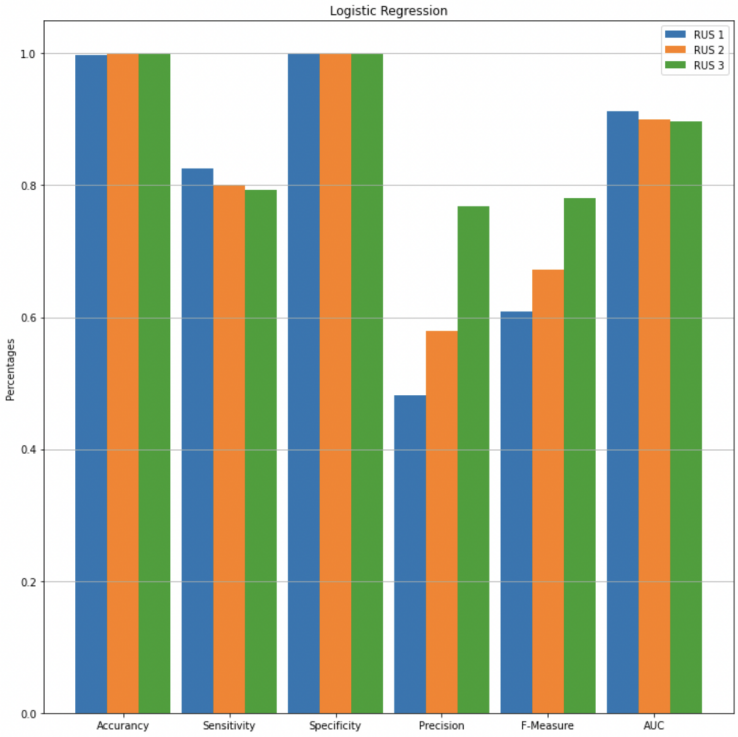


Figure 3: Metrics plots

2. **K-Nearest neighbours**

Metrics	**Ratio 50:50**	**Ratio 34:66**	**Ratio 25:75**
TP	138	134	132
TN	84309	85017	85110
FP	979	271	178
FN	17	21	23

Table 4: Predictions

Metrics	Ratio 50:50	Ratio 34:66	Ratio 25:75
Accuracy	98.8%	99.7%	99.8%
Sensitivity	89.0%	86.5%	85.2%
Specificity	98.9%	99.7%	99.8%
Precision	12.4%	33.1%	42.6%
F_measure	21.7%	47.9%	56.8%
AUC	93.9%	93.1%	92.5%

Table 5: Metrics values

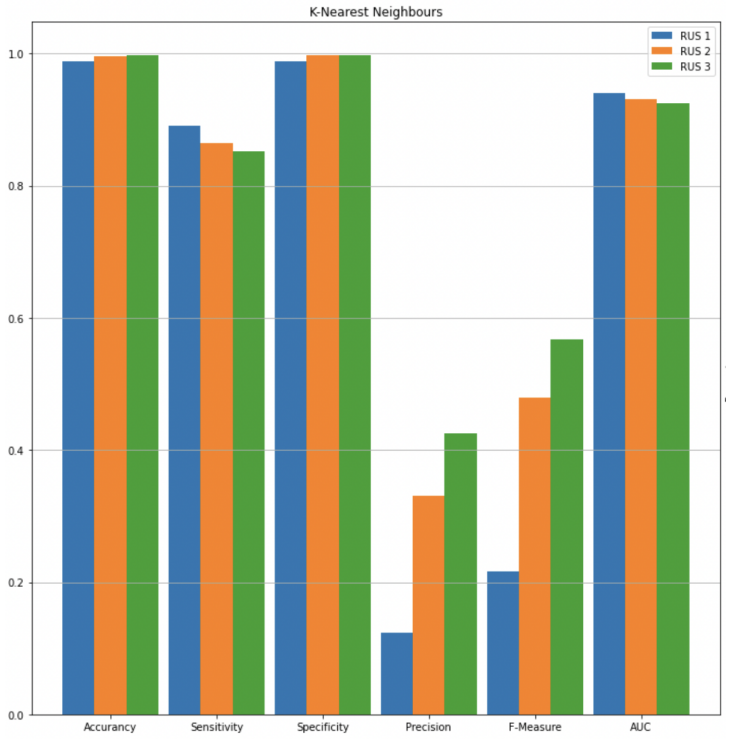


Figure 4: Metrics plots

3. **Naive Bayes**

Metrics	Ratio 50:50	Ratio 34:66	Ratio 25:75
TP	135	134	136
TN	82960	82781	82801
FP	2328	2507	2487
FN	20	21	19

Table 6: Predictions

Metrics	Ratio 50:50	Ratio 34:66	Ratio 25:75
Accuracy	97.3%	97.0%	97.1%
Sensitivity	87.1%	86.5%	87.7%
Specificity	97.3%	97.1%	97.1%
Precision	5.5%	5.1%	5.2%
F_measure	10.3%	9.6%	9.8%
AUC	92.2%	91.8%	92.4%

Table 7: Metrics values

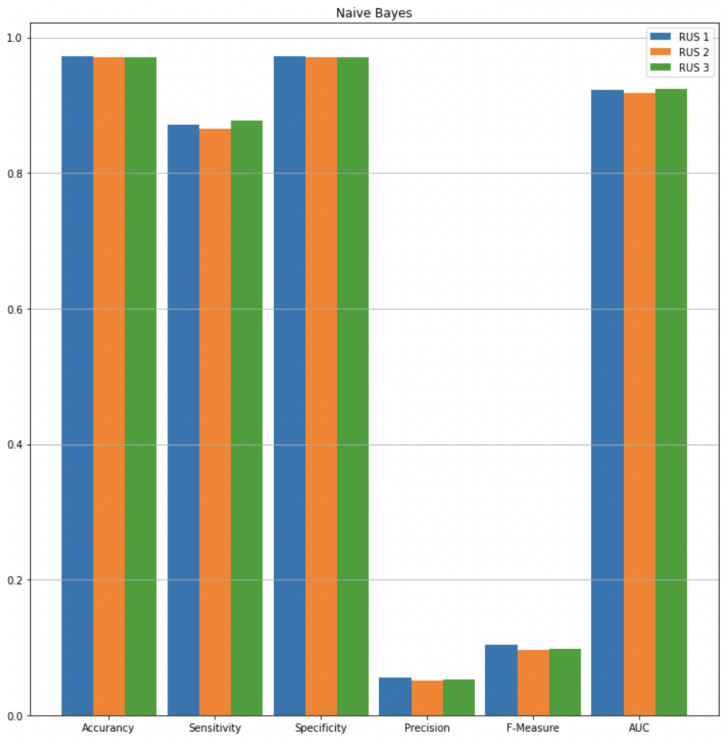


Figure 5: Metrics plots

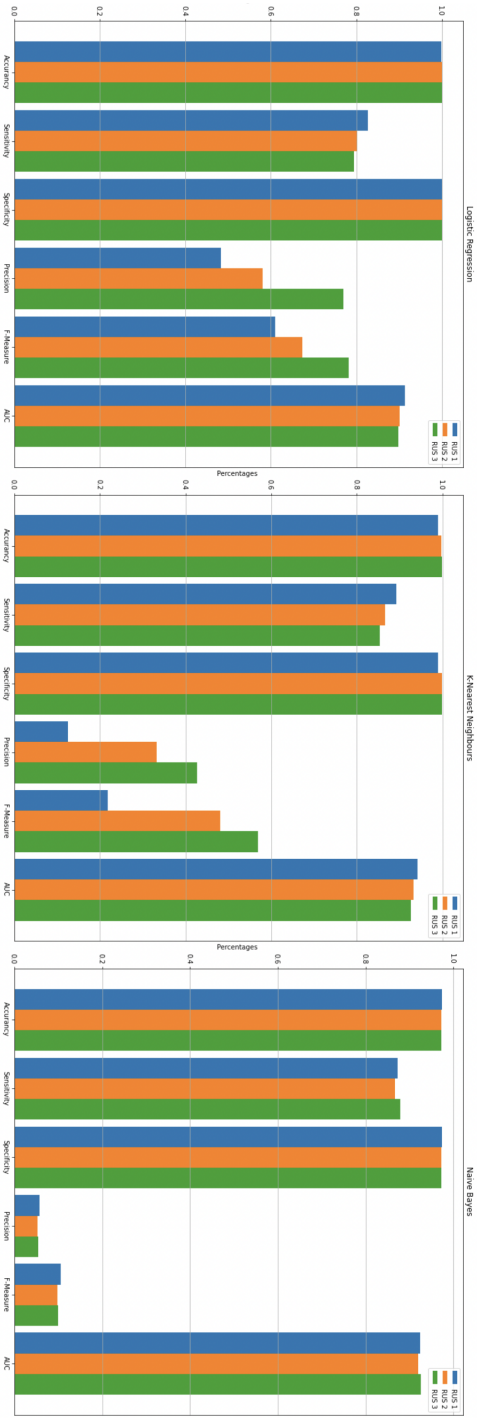


Figure 6: Metrics plots summary

6 Additional analysis

We have decided to go beyond the analysis inside the reference paper by applying the neural networks for trying to solve this binary classification problem. Moreover, we have decided to apply cross-validation in order to obtain a more consistent indicator of the validity of our algorithms. We have also decided to exploit the logistic regression implementation of scikit-learn to test the goodness of our implementation.

6.1 Neural networks

Neural networks are probably the most widespread and famous machine learning algorithm. The idea is similar to that of logistic regression: a neural networks is a function (meant as a computer science function, namely a function that a Touring machine can implement) that aims to return meaningful values about the situation at issue. Such function is initialized more or less randomly, therefore its starting behaviour has nothing to do with the goal behaviour. However, similarly to the logistic regression, an optimization algorithm, like stochastic gradient descent or one of its variants (for example, RMSPros), is adopted in order to calibrate the randomly initialized parameters in a way that makes the neural network emulate the goal function. More precisely, in our case, the function to implement takes in input data about credit card transactions and returns in output a binary value, which says whether the occurred transaction was fraudulent or not. What makes the fortune of neural networks is the complexity of the function to learn. In other words, the number of degrees of freedom (parameters) that can be tuned. As a matter of fact, an algorithm like logistic regression is faster and simpler, but it cannot be able to learn complex functions as neural networks do (the universality theorem holds for neural networks). This is why we have decided to provide an implementation of neural networks.
It should be noticed that we have opted for a 4-layers architecture with, respectively, 30, 20, 10 and 2 neurons. Such values have not been tuned through a trial-and-error procedure, but they have simply been taken from the literature. As aforementioned, we use Root Mean Squared Propagation, or RMSProp, which is a variant of stochastic gradient descent, as optimization algorithm and cross entropy as loss function.
The code is the following:

1. first of all, prepare the randomly undersampled dataset by dividing sample features from labels:

```
labels_train1 = train_set1[:,30]
x_data1 = train_set1[:,:30].transpose()
labels_train1.shape, x_data1.shape

labels_train2 = train_set2[:,30]
```

```
x_data2 = train_set2[:,:30].transpose()
labels_train2.shape, x_data2.shape

labels_train3 = train_set3[:,30]
x_data3 = train_set3[:,:30].transpose()
labels_train3.shape, x_data3.shape

labels_valid = valid_set[:,30]
x_data_valid = valid_set[:,:30].transpose()
labels_valid.shape, x_data_valid.shape
```

2. next, create a **one-hot** representation of the labels, which will be useful later:

```
y_data1 = np.zeros((2, labels_train1.shape[0]))
for i in range(2):
  y_data1[i, labels_train1==i] = 1

y_data2 = np.zeros((2, labels_train2.shape[0]))
for i in range(2):
  y_data2[i, labels_train2==i] = 1

y_data3 = np.zeros((2, labels_train3.shape[0]))
for i in range(2):
  y_data3[i, labels_train3==i] = 1

y_data_valid = np.zeros((2, labels_valid.shape[0]))
for i in range(2):
  y_data_valid[i, labels_valid==i] = 1
```

3. then we can start with the out-and-out implementation of the neural network; define a function that initialize the parameters according to the Xavier Uniform initialization:

```
def initialize_params_ann(layers_size):
  np.random.seed(0)
  params = list()
  for i in range(len(layers_size) - 1):
    params.append(np.random.uniform(
    - np.sqrt(6 / (layers_size[i] + layers_size[i+1])),
    np.sqrt(6 / (layers_size[i] + layers_size[i+1])),
    (layers_size[i+1], layers_size[i])))

    params.append(np.zeros((layers_size[i+1],1)))
  return params
```

4. define a function that computes the neural network; we have opted for the tanh as activation function. Moreover, we apply **softmax** to the last

layer, namely $\hat{z}_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$, so that the output results can be interpreted as probabilities. We applied **softmax** instead of **sigmoid**, which is most suitable for binary problems, mainly because we have structured the neural network as a multi-class classification problem:

```
def ANN(x, params):
  W = params[0::2]
  b = params[1::2]
  layer = x
  for i in range(len(W)):
    layer = W[i] @ layer - b[i]
    if i < len(W) - 1:
      layer = jnp.tanh(layer)
  layer_exp = jnp.exp(layer)
  return layer_exp / jnp.sum(layer_exp, axis = 0)
```

5. define the (cross-entropy) loss function:

```
def cross_entropy(x, y, params):
  y_appr = ANN(x, params)
  return - jnp.mean(jnp.sum(y*jnp.log(y_appr), axis=0))
```

6. train the network exploiting all the tools previously mentioned (for details about the hyperparameters see the notebook):

```
def training_loop(x_train,y_train):

  num_epochs = 1000
  batch_size = 100
  learning_rate = 5e-4
  decay_rate = 0.9
  delta = 1e-7

  history_train_Xen = list()

  grad_jit = jax.jit(jax.grad(cross_entropy, argnums = 2))
  loss_Xen_jit = jax.jit(cross_entropy)

  params = initialize_params(layers_size)

  history_train_Xen.append(loss_Xen_jit(x_train, y_train, params))
  cumulate_grad = [0.0 for i in range(len(params))]

  t0 = time.time()
  for epoch in tqdm(range(num_epochs)):
    idxs = np.random.choice(x_train.shape[1], batch_size)
    gradients = grad_jit(x_train[:,idxs], y_train[:,idxs], params)
```

```
    for i in range(len(params)):
      cumulate_grad[i] = decay_rate*cumulate_grad[i]+
                  (1 - decay_rate)*gradients[i]**2
      params[i] -= learning_rate/(delta + jnp.sqrt(cumulate_grad[i])) *
                  gradients[i]

    history_train_Xen.append(loss_Xen_jit(x_train, y_train, params))

  print('train loss: %1.3e' % history_train_Xen[-1])
  plt.loglog(history_train_Xen, label = 'train loss')
  return params
```

7. finally, we apply the algorithm giving three different training sets (one for each ratio above mentioned) and we evaluate the results through the metrics used above:

```
def evaluations(x, y, params):
  digit_appr = jnp.argmax(ANN(x, params), axis = 0)
  digit      = jnp.argmax(y, axis = 0)

  TP = jnp.sum(digit_appr * digit )
  TN = jnp.sum((1- digit_appr)* (1-digit ))
  FP = jnp.sum((1 - digit) * (digit_appr))
  FN = jnp.sum((digit ) * (1- digit_appr))

  return [TP,TN,FP,FN]

params1 = training_loop(x_data1,y_data1)
params2 = training_loop(x_data1,y_data1)
params3 = training_loop(x_data1,y_data1)

predictionsNN1 = evaluations(x_data_valid, y_data_valid, params1)
predictionsNN2 = evaluations(x_data_valid, y_data_valid, params2)
predictionsNN3 = evaluations(x_data_valid, y_data_valid, params3)
```

Our results are:
Initial Cross-Entropy: 0.746139

Metrics	Ratio 50:50	Ratio 34:66	Ratio 25:75
Cross-Entropy	0.144865	0.230297	0.313053
Accuracy	95.2%	91.5%	88.2%
Sensitivity	93.5 %	94.2%	95.5%
Specificity	95.2%	91.5%	88.2%
Precision	3.4%	2.0%	1.5%
F_measure	6.6%	3.9%	2.9%
AUC	94.4%	92.8%	91.9%

Table 8: Metrics values

6.2 Cross-validation

Cross-validation is any of various model validation techniques for assessing how the results of a statistical analysis will generalize to an independent data set. Cross-validation is a resampling method that uses different portions of the data to test and train a model on different iterations. It is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice.
We have decided to apply **k-fold cross-validation**, in which the original dataset is randomly partitioned into k equal-sized folds. We compute the performance metrics by considering, one at a time, each of the folds as test set, and all the others together as training set. Finally we compute the average value of each metric and we are done.

Our results with 3 folds and a RUS ratio of 0.5 are:

```
Mean Accuracy: 63.279%
Mean sensitivity: 76.609%
Mean specificity: 37.204%
Mean precision: 70.911%
Mean fmeas: 73.495%
Mean auc: 56.907%
```

Figure 7: KNN metrics evaluated through cross-validation

```
Mean Accuracy: 89.160%
Mean sensitivity: 86.030%
Mean specificity: 95.575%
Mean precision: 97.465%
Mean fmeas: 91.375%
Mean auc: 90.803%
```

Figure 8: NB metrics evaluated through cross-validation

6.3 Logistic Regression with sklearn.linear_model library

Scikit-learn provides a class that implements regularized logistic regression. We decided to use and evaluete the performances of the models trained with this classifier.

Firstly we import the Logistic Regression module and create a Logistic Regression classifier object using the LogisticRegression function.

```
from sklearn.linear_model import LogisticRegression
logisticRegr = LogisticRegression()
```

Then, we fit our model on the train set using the fit function.

```
logisticRegr.fit(train_set[:,:-1], train_set[:,-1])
```

Finally, we perform prediction on the test set using the predict function.

```
predictions = logisticRegr.predict(valid_set[:,:-1])
```

The following picture summarizes the metrics used to evaluates the goodness of the algorithm.

```
Mean Accuracy:    91.734%
Mean Sensitivity: 92.294%
Mean Specificity: 90.819%
Mean Precision:   95.156%
Mean Fmeas:       93.687%
Mean AUC:         91.556%
```

Figure 9: Metrics evaluated through cross-validation

7 Conclusion

Our scope, like the one presented in the paper[3], was to analyze how accurately three machine learning algorithms are able to determine if a credit card transaction is a fraud or not with the Random Under Sampling (RUS) method. As opposed to the paper, this report includes the code that we used for self-implementing three machine learning algorithms: Logistic Regression, K-Nearest Neighbors, and Naive Bayes.

As we can see in section 5, all three algorithms with all three ratios have very high accuracy, with a maximum of 99.9% for the Logistic Regression - 25:75 ratio.

Regarding sensitivity, all of the evaluations are above 80.0%, with the best result being 89.0% in K-Nearest Neighbors - 50:50 ratio. This parameter, jointly with specificity is very significant for estimating the goodness of our predictions. The former is the percentage of fraudulent transactions that have been acknowledged among all those present. The latter represents the dual: the percentage of non-fraudulent transactions that have been properly cataloged. All the values associated with specificity are elevated with a maximum in Logistic Regression - 25:75 ratio rounded to 100.0%.

Despite different attempts at improvement, precision and F-Measure are two metrics that remain low. The most significant results are obtained again with Logistic Regression - 25:75 where precision and f-measure have reached values just below 80%. It is important to analyse precision compared to sensitivity. Sensitivity metric tells us how many fraud transactions are hurt down while precision metric indicates how many times our algorithm alerted us because the transaction considered could be fraudulent.

As a final metric, we evaluated AUC, which has better values in K-Nearest Neighbors - 50:50 ratio.

Now we will compare our results, obtained with the three self-implemented algorithms, with those provided in the paper. Except for precision and f-measure, Logistic Regression metrics are comparable, our Naive Bayes method improves sensitivity and accuracy and our K-Nearest Neighbors improve significantly all the measures. Despite that, if we compare the results that came out from the paper with those obtained through k-fold cross-validation also Naive Bayes and K-Nearest Neighbors' values will align.

More efficient resampling methods could potentially improve our results since the overfitting issue is aggravated by the use of random undersampling, which shrinks significantly the training set.

One other improvement that could be made is substituting a linear decision boundary to more complex, nonlinear decision boundaries to prevent underfitting as well. An example is the Kernel Regression algorithm evaluated during

[3]Itoo, F., Meenakshi Singh, S. Comparison and analysis of logistic regression, Naive Bayes and KNN machine learning algorithms for credit card fraud detection. Int. j. inf. tecnol. 13, 1503–1511 (2021). https://doi.org/10.1007/s41870-020-00430-y.

the lectures.

Discussing K-Nearest Neighbors in particular, as we said that is important to select a proper K parameter and a proper distance because they influence a lot the accuracy of the predictions. We exploited Euclidean distance, as the reference paper said, but you could try with Manhattan or Minkowski ones. Likewise, the K parameter can be enhance.

Concerning the Artificial Neural Network, we have obtained a fairly high accuracy (however lower than logistic regression) but a greater sensitivity compared to all other algorithms implemented.

To summarize our results, Logistic Regression (in particular the one implemented with the advantages of Scikit-learn's class) seems the best of the three algorithms for Credit Card Fraud Detection. Furthermore, the overall best results meet the best efficiency and speed.

The K-Nearest Neighbors classifier offers an alternative approach for Credit Card Fraud classification via lazy learning that allows us to make predictions without any model training, but with a more computational expensive prediction step and, moreover, with decreasing sensitivity and specificity as the imbalance of the data increases.

Finally, Naive Bayes classifiers are easy to implement, computationally efficient, and tend to perform particularly well on relatively small datasets compared to the other algorithms. But the assumption that all predictors are independent does not hold in real life.

8 References

All the contents covered in this report are available at the link below:
https://github.com/MargheritaMusumeci/Credit-card-fraud-detection

8.1 Papers and books

- Itoo, F., Meenakshi Singh, S. Comparison and analysis of logistic regression, Naïve Bayes and KNN machine learning algorithms for credit card fraud detection. Int. j. inf. tecnol. 13, 1503–1511 (2021). https://doi.org/10.1007/s41870-020-00430-y
- Python machine learning, Sebastian Raschka, Vahid Mirjalili, ISBN 978-1-78995-575-0
- Naive Bayes and Text Classification I – Introduction and Theory, S. Raschka, Computing Research Repository (CoRR), abs/1410.5329, 2014, http://arxiv.org/pdf/1410.5329v3.pdf

8.2 Tools

- Panda
- Numpy
- Jax
- Google colab
- Scikit-learn

8.3 Websites

- machinelearningmastery
- towardsdatascience.com
- kaggle.com