Logistic Regression Classifier from scratch

October 2, 2021

Abstract

A logistic regression classifier has been implemented in two versions: one based on the method of Loss Minimization through Gradient Descent; the other based on Likelihood Maximization through Gradient Ascent. The algorithms have been applied to two datasets: Hepatitis and Bankruptcy. Different techniques of data analysis have been applied to the data. Experiments have been conducted to determine the balance between accuracy and running time by changing the relative tolerance of the algorithms and the learning rate. Further feature engineering has been applied to the data in order to increase the obtained accuracy of the models.

1 Introduction

In this report, we describe an end-to-end project on which a logistic regression classifier is implemented from scratch. The method proposed includes the classical steps of a Machine Learning project: data acquisition and analysis, feature engineering, model training, comparison between models, and data visualization. We aim at getting acquainted with each of these steps while looking for a model that predicts the proposed data with the highest accuracy possible.

Two different datasets are used in the project: one characterizing patients with (or without) hepatitis, the other characterizing features that weather lead to the bankruptcy of individuals or not. Histograms and correlation maps are plotted on each of them to further understand the features distribution and their relation with the target variable.

We propose two different versions of the logistic regression classifier. The first minimizes the cross-entropy loss function for the data through the technique of gradient descent. The second is based on the maximization of the likelihood of the model. To test the models proposed, we implement a cross-validation function (see Appendix 5.3) that returns the average accuracy of the input model against a specific dataset. Analyses of accuracy, running times, and learning rates are conducted to arrive at optimal versions of the engineered data and the models suggested in this report.

1.1 The Logistic Regression Classifier

In this project we implement two different versions of the Logistic Regression Classifier (see [3, pp. 119]). The corresponding implementations can be observed in Appendix 5.2. The first is based in minimizing the loss function

$$\text{loss}(\mathbf{w}) = \sum_i \left(\frac{1}{m} \left[-\mathbf{y}^T \log(h) - (1-\mathbf{y}^T) \log(1-h) \right] \right)_i \tag{1}$$

were $\mathbf{w}$ is the weight vector of the model, m is the number of features, $\mathbf{y}$ is the target vector, $h = 1/(1 + exp(-\mathbf{w}^T\mathbf{X}))$, and $\mathbf{X}$ is the feature matrix.

The other version is based in maximizing the likelihood estimation. The ML function is

$$ll(D) = \sum_i \left(y(\mathbf{w}^T\mathbf{X}) - \mathbf{log}(1 + \exp \mathbf{w}^T\mathbf{X}) \right)_i . \tag{2}$$

Both optimization processes are conducted by computing the gradient of functions 1 and 2 relative to $\mathbf{w}$, and updating the value of $\mathbf{w}$ using the gradient descent (or ascent) rule with predefined learning rate α. In consequence we will call these methods Gradient Descent – Loss Minimization and Gradient Ascent – Likelihood Maximization for the rest of this report.

2 Datasets

In this project we are required to perform two–class classification on two datasets:

Dataset Hepatitis: consists of various features for hepatitis patients. The main aim of the Hepatitis dataset is to discriminate two classes: survivors and patients for whom the hepatitis proved terminal.

Dataset Bankruptcy: contains various econometric attributes for bankruptcy status prediction.

In this section we describe the pre–analysis performed on each of these datasets. Only selected examples will be given. We refer the reader to the files cited in Appendix 5.1 for a more thorough discussion of the data analysis conducted.

2.1 Dataset Hepatitis

The data has 142 entries, each with 20 columns: 19 features and 1 target class called `ClassLabel`. The features were divided into continuous and discrete features, according to whether they have more or less than 25 unique different values respectively. The corresponding sets are:

```
discrete_features = ['sex',  'steroid', 'antivirals', 'fatigue', 'malaise',
'anorexia', 'liver_big', 'liver_firm', 'spleen_palable', 'spiders',
'ascites','varices', 'histology']

continuous_features = ['age', 'bilirubin', 'alk_phosphate', 'sgot', 'albumin',
'protime']
```

Box plots were created for each of the discrete features, so as to observe how their different values averaged with the target `ClassLabel`. The features `'ascites'` and `'varices'` resulted the most correlated with `ClassLabel`, since their values 1 and 2 reported the higher differences between them for the averaged `ClassLabel`.

The continuous features were analyzed via histograms. For the case of `'age'`, a Gaussian-like histogram was obtained (see figure 1a). The other features, present strongly-skewed histograms, like the one shown in 1b.

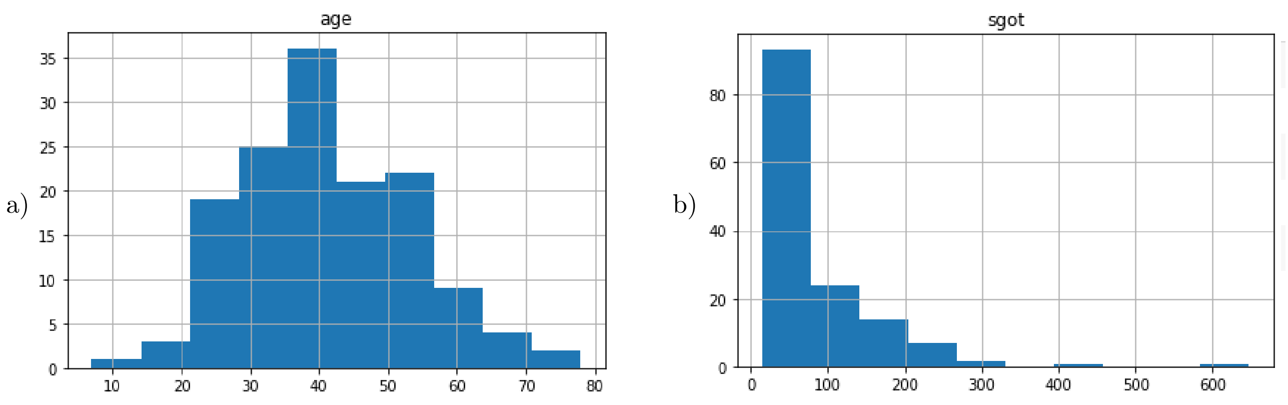


Figure 1: Examples of histograms plotted on the continuous figures: (a) Gaussian–like histogram of `age`, (b) skewed histogram of `sgot`.

Also, some of the figures present a large quantity of outliers, as can be observed in figure 2 for one of the box plots obtained for the continuous features.

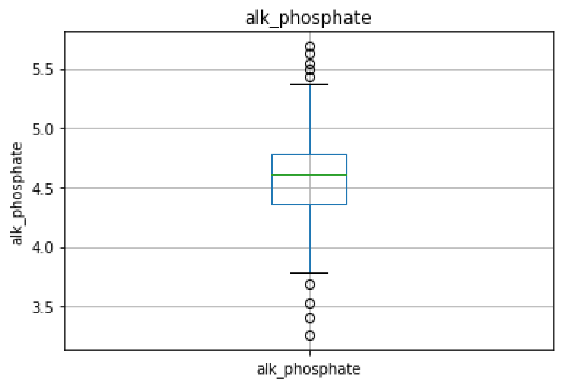
Finally, a heatmap of the correlations between the different continous variables and with the target variable was obtained (see figure 2). The feature `albumin` is positively correlated with `protime`, while negatively correlated with `alk_phosphate` and `bilirubin`. The target class `ClassLabel` was found to be highly correlated with `albumin` and `bilirubin`.

2.2 Dataset Bankruptcy

The dataset Bankruptcy was analyzed with similar techniques to those applied for the dataset Hepatitis. In this case, the data contain 64 features, all of them are continuous (in the sense described in the previous section). Again, tail–heavy distributions were found for many of the 64 features. The higher correlations found between the features and the target `fig:outliers` was of around 0.3. Since similar behaviour was found for the two datasets, the same models will be tested on both. For more insights on this data set we refer the reader to Appendix 5.1.

3 Results

The results discussed in this report are only a fraction of what is shown in the attached `mini_project_1_team_2`. We refer the reader to this notebook for the complete discussion.

a) 

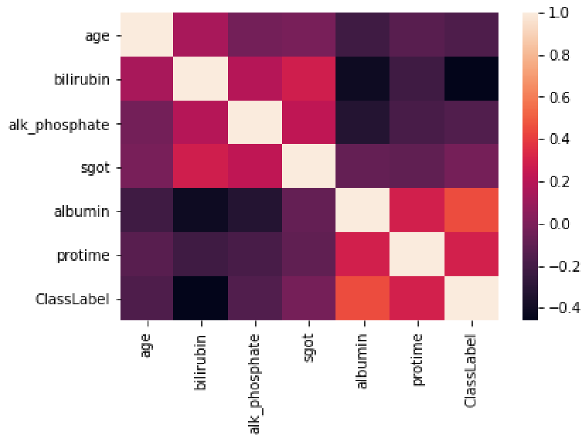
b) 

Figure 2: (a) Box plot of the feature `'alk_phosphate'`. (b): Heatmap with the correlation between the different continuous features of the dataset and between these features and the target label `ClassLabel`.

3.1 Analysis of accuracy, running time and learning rate

In this section we will explore how the running time of the training/validation process behaves as the different input parameters of the code are changed.

3.1.1 Stopping Criteria

For the two versions of the logistic regression classifier implemented, three simultaneous stopping criteria were used in the **w**–update cycles:

1. Maximum iterations: A maximum number of iterations was defined. The default value is 100000.
2. Relative tolerance: When a user–defined target value for $\text{abs}\left((\mathbf{w}_{n+1} - \mathbf{w}_n)/\mathbf{w_n}\right)$ is reached, the calculation stops. The default value is $-\infty$.
3. Relative tolerance: When a user–defined target value for $\text{abs}\left(\mathbf{w}_{n+1} - \mathbf{w}_n\right)$ is reached, the calculation stops. The default value is $-\infty$.

To determine which are the best combinations for the values of these different stopping criteria, together with the value of the learning rate, several experiments were conducted with different sets of input variables. A balance must be made between the accuracy obtained and the running time of the code (which, for very small tolerances could be prohibitive). The calculations shown in this section are based in the following inputs for the models:

1. Maximum iterations is 100000
2. Relative tolerance is spaced evenly on a log scale in $[10e-5, 10e-1]$
3. Absolute tolerance is $-\infty$
4. Learning rate is spaced evenly on a log scale in $(0, 0.9)$, which is suggested to be small number by [2, pp. 62 - 64]
5. Training attribute is exactly the given data (feature engineering to be delivered in the next section).

3.1.2 Running time Analysis

Figure 3 show the training/validation running time of several experiments which the relative tolerance was changed, for the method of Gradient Descent–Minimum Loss applied on the Hepatitis dataset.

It can be observed that the training time decreased exponentially from 35 seconds to $4e-2$ seconds when the relative tolerance was changed in the range $[1e-4, 1e-1]$. However, the time in $1e-4$ and $1e-5$ is almost identical. Therefore, it can be concluded that the stopping criterion being used by the algorithm in this case is the maximum number of iterations. Thus, the training model has reached it possible minimum tolerance for the maximum iterations of 100000.

The same kind of analysis was conducted on the algorithm Gradient Ascent – Maximum Likelihood for both the Hepatitis and Bankruptcy sets, and again the Gradient Descent – Minimum Loss for the Bankruptcy test. In all cases identical behaviours was observed for the running time. The corresponding plots can be observed in the attached Notebook. These plots can be used to evaluate the relative computational cost of using different values for the stopping criteria.

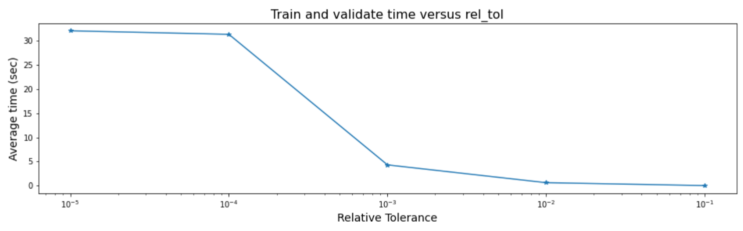


Figure 3: Training time versus Relative tolerance, Hepatitis dataset, Gradient descent

3.1.3 Accuracy analysis

The most important figure of merit of an ML algorithm is the accuracy with which it predicts the validation data. In figure 4 we show the results of accuracy values (averaged percentage of correct predictions in the validation sets) of the algorithm Gradient Descent–Loss Minimization applied on the Hepatitis dataset for several experiments using different learning rates and relative tolerances.

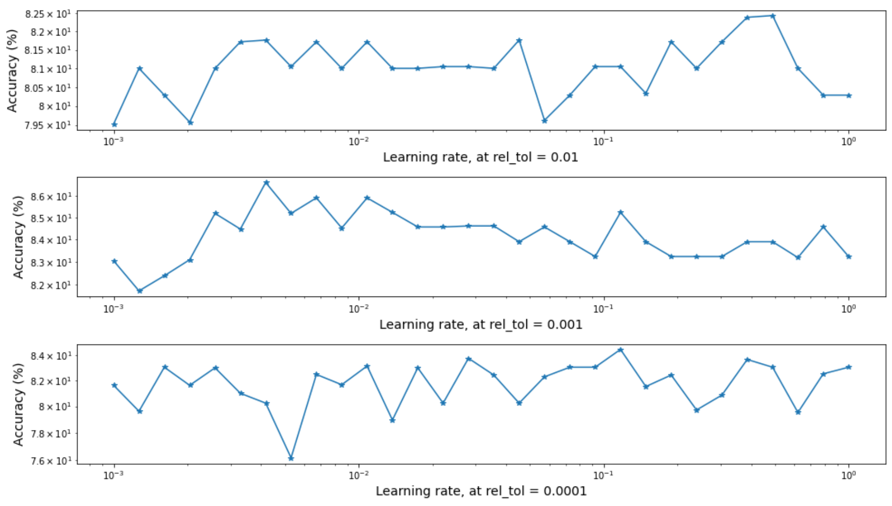


Figure 4: Accuracy versus Learning rate, Hepatitis dataset, Gradient descent

It can be observed that in general, decreasing the relative tolerance increases the accuracy. In fact, with a stricter stopping criteria of relative tolerance, the error function can reach the minimum value closer before stopping training; which results in a generally better trained model. However, diminishing the relative tolerance leads to a trade-off between accuracy and system training time, as was shown in the previous section. For instance, setting the relative tolerance from $1 \cdot 10^{-2}$ to $1 \cdot 10^{-4}$ make the peak system performance from 82.4% to 84.4%; in contrast degrading training time exponentially from 0.6 seconds to 31.2 seconds.

From the simulation for Hepatitis dataset, we conclude that the optimum parameter set that yields peak performance at 86% by learning rate $4 \cdot 10^{-3}$, relative tolerance $1 \cdot 10^{-3}$ while applying Gradient descent model. Repeating similar test applying Maximum likelihood model yields almost identical result, except a small degradation in accuracy.

3.1.4 Dataset Bankruptcy

In general, the trend discussed in the previous subsection hold for the Bankruptcy test. Higher optimal learning rates while applying Gradient descent were observed for this dataset when compared with the Hepatitis dataset. For the algorithm of Maximum Likelihood, the simulations conducted on the Bankruptcy dataset give an optimum parameter with learning rate $1.2e-3$, relative tolerance $1e-3$, which yields peak performance at 79.4%.

3.2 Optimization of the model

After the previous analysis, we are going to discuss in this subsection experiments with a learning rate of 0.1 and a relative tolerance of 0.01, since these values give a good balance between model accuracy and running time for the practical purposes of this project.

For each of the two versions of the logistic regression classifier four different feature engineering trials were tested on each of the two data sets (see for example [1, pp. 66]):

Normalization of the data: A logarithm was applied to each of the continuous features of the dataset in order to distribute them in a Gaussian like shape.

Standardization of the data: The average was subtracted from each continuous feature and the result was divided by the standard deviation, in order to reduce the weight of outliers and standardize the data.

Normalization AND standardization of the data: The above two processes were applied one after the other.

Squared features: This technique consists on incorporating new features to the dataset that are the square of the original features. The process is done by incorporating one squared feature at a time and testing whether the accuracy increases. If it does increases, the squared feature is now kept as a new feature in the dataset.

The corresponding implementations can be seen in Appendix 5.4. The resulting accuracies from utilizing cross validation with k = 10 folds are shown in figure 5. In this figure, columns of different colors correspond to different feature engineering processes as discussed above. Also included is the accuracy corresponding to applying the model to the raw original dataset (blue columns).

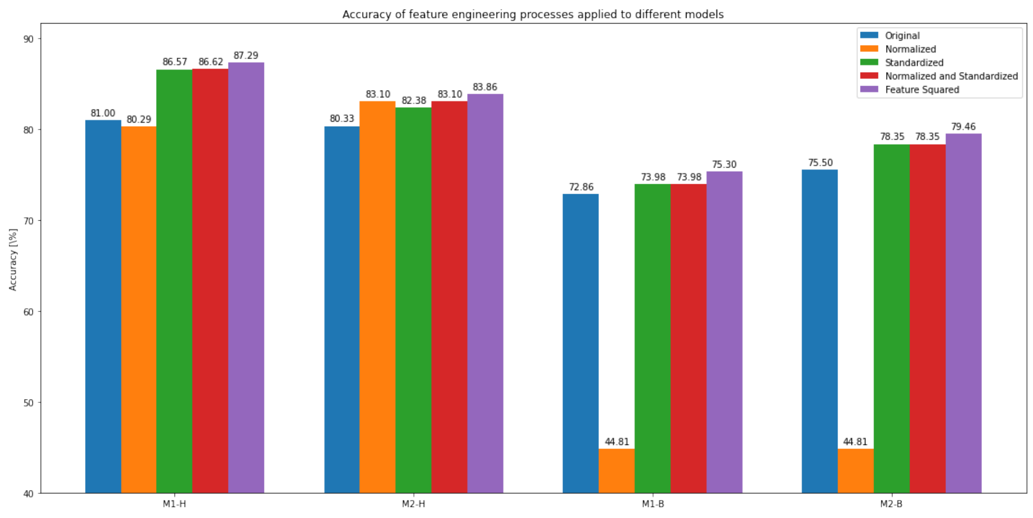


Figure 5: Accuracies obtained for the two versions of the model Logistic Regression (M1: Loss minimization through gradient descent, M2: Likelihood maximization through gradient ascent) applied on the two data sets (H: hepatitis, B: bankruptcy). Different feature engineering processes are applied to the data before the accuracy calculation, as shown in the legend. In all the cases a learning rate of $\alpha = 0.1$ and a relative tolerance of 0.01 have been used.

It can be seen that the incremental application of the feature engineering techniques as described above increase the accuracy of the models applied on the hepatitis dataset. For the case of bankruptcy analysis however, we learn that the normalization procedure in fact reduces the accuracy to an inadmissible level. Thus, we selected not to include normalization in the process of detecting squared features to improve the accuracy.

4 Discussion and Conclusions

In this project, we implemented 2 optimization approaches for the Logistic Regression Classifier. The two approaches were tested against two different datasets. Data analysis on each dataset was conducted, and we could conclude that some feature engineering could help in increasing the corresponding accuracies of the models. We studied how the learning rate and the relative tolerance of the algorithms can influence the proposed models on each of the datasets in terms of running time and accuracy. It was concluded that a smaller relative tolerance renders higher accuracy but larger running times. After choosing a good balance between accuracy and running time, we proceeded to implement feature engineering methods to improve the obtained accuracy. In this way, optimal data pre-processing methods were obtained for each dataset and each proposed method.

It was concluded that for the Hepatitis dataset, the Gradient descent – Loss minimization method gave the most accurate results when a learning rate of $4 \cdot 10^{-3}$ was used, a relative tolerance of $1 \cdot 10^{-3}$ was used.

The optimal feature engineering processes to apply in this case were the normalization and standardization techniques. Also, incorporating quadratic features to the dataset showed an increase in accuracy.

For the case of the Bankruptcy dataset, the maximum accuracy was obtained when the data was normalized and when squared features were added to the data set. The best model for this dataset was the Gradient Ascent – Likelihood maximization, for a learning rate is $1.2 \cdot 10^{-3}$, with relative tolerance of $1 \cdot 10^{-3}$. In this case the process of Normalization strongly decreased the accuracy, so it is not recommended.

References

[1] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[2] Goodfellow I. et al. *Deep Learning*. Cambridge: MIT Press, 2016.

[3] Friedman J., Hastie T., and Tibshirani R. *The Elements of Statistical Learning*. Series in Statistics. New York: Springer, 2001.

5 Appendix

5.1 Data analysis

The attached files `hepatitis_initial_data_analysis` and `bankrupcy_initial_data_analysis` contain notebooks with plots and tables corresponding to the data analysis conducted on the data.

5.2 Classes definition

```python
# imports
import numpy as np
import pandas as pd
from scipy import special



# Auxiliary functions

def sigmoid(X, w):
  """
  Sigmoid function

  Parameters:
  x: {vector} of shape (n_features)
    Column vector corresponding to n_features of one sample
  w: {vector} of shape (n_features)
    Column vector corresponding to the weights
  """
  z = np.dot(X, w)
  return special.expit(z)

def cross_entropy_loss(h,y):
  """
  Returns the cross entropy loss

  Parameters
  h = scalar, result of using the sigmoid function: h = sigmoid (w' X)
  y: scalar with the target class
  """
  return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()


def gradient_descent(X,h,y):
  return np.dot(X.T, (h-y)) / y.shape[0]
def update_weight_loss(w,learning_rate,gradient):
  return w - learning_rate *gradient

def log_likelihood(x,y, w):
  z=np.dot(x,w)
  ll = np.sum(y*z - np.log(1+np.exp(z)))
  return ll

def gradient_ascent(X, h, y):
    return np.dot(X.T, y - h)
def update_weight_mle(weight, learning_rate, gradient):
    return weight + learning_rate * gradient


# Logistic Regression by minimization of Cross Entropy Loss through gradient descent

class LogisticRegression_gradient_descent:
  """
  This class implements logistic regression.
  Parameters:

  Attributes:

  """
  def __init__(self, learning_rate=0.1, max_iter= 100000, rel_tol = -np.inf, abs_tol = -np.inf
    , print_time=False):
    self.max_iter = max_iter
    self.learning_rate = learning_rate
    self.rel_tol = rel_tol
    self.abs_tol = abs_tol
    self.print_time = print_time

  def fit (self, X, y):
    """
    Fit the model according to the given training data

    Parameters
    ----------
    X:{array} of shape (n_samples, n_features)
      Training vector, where n_samples is the number of samples and
      n_features is the number of features
    y:{array} of shape (n_samples), default = None
      Target vector relative to X.

    Returns
    -------
    self
      Fitted estimator
    """
    if self.print_time:
      start_time = time.time()

    self.w = np.zeros(X.shape[1])
    for i in range(self.max_iter):
      self.h= sigmoid(X,self.w)
      self.gradient = gradient_descent(X,self.h,y)

      new_w = update_weight_loss(self.w, self.learning_rate, self.gradient)


      rel_error = np.linalg.norm(new_w - self.w) /  np.linalg.norm(new_w)
      abs_error = np.linalg.norm(new_w - self.w)

      if (rel_error < self.rel_tol) or (abs_error < self.abs_tol):
        break

      self.w = new_w

    if self.print_time:
      print("Fitting time:" + str(time.time() - start_time) + " seconds")

  def predict(self, X):
    """
    Extimated prediction.

    Parameters
    ---------
    X: array of shape (n_samples, n_features)

    Returns
    -------
    y: array of shape(n_samples)
    Each row has the probability of y being of class 1, therefore we need to further process
    the results to
    compare them with 0.5 so as to classify the output as 0 or 1
```

```python
    """
    result  =   sigmoid(X,self.w)
    result = np.where(result<0.5, 0 ,1)
    return result


# Logistic Regression by maximization of Log Likelihood through gradient ascent

class LogisticRegression_maximum_likelihood:
  """
  This class implements logistic regression.
  Parameters:

  Attributes:

  """
  def __init__(self, learning_rate=0.1, max_iter= 100000, rel_tol = -np.inf, abs_tol = -np.inf
    , print_time = False):
    self.max_iter = max_iter
    self.learning_rate = learning_rate
    self.rel_tol = rel_tol
    self.abs_tol = abs_tol
    self.print_time = print_time


  def fit (self, X, y):
    """
    Fit the model according to the given training data

    Parameters
    ----------
    X:{array} of shape (n_samples, n_features)
      Training vector, where n_samples is the number of samples and
      n_features is the number of features
    y:{array} of shape (n_samples), default = None
      Target vector relative to X.

    Returns
    -------
    self
      Fitted estimator
    """

    if self.print_time:
      start_time = time.time()

    self.w = np.zeros(X.shape[1])
    for i in range(self.max_iter):
      self.h= sigmoid(X,self.w)
      self.gradient = gradient_ascent(X,self.h,y)

      new_w = update_weight_mle(self.w, self.learning_rate, self.gradient)


      rel_error = np.linalg.norm(new_w - self.w) /  np.linalg.norm(new_w)
      abs_error = np.linalg.norm(new_w - self.w)

      if (rel_error < self.rel_tol) or (abs_error < self.abs_tol):
        break

      self.w = new_w

    if self.print_time:
      print("Fitting time:" + str(time.time() - start_time) + " seconds")

  def predict(self, X):
    """
    Extimated prediction.

    Parameters
    ---------
    X: array of shape (n_samples, n_features)

    Returns
    -------
    y: array of shape(n_samples)
    Each row has the probability of y being of class 1, therefore we need to further process
    the results to
    compare them with 0.5 so as to classify the output as 0 or 1
    """
    result  =   sigmoid(X,self.w)
    result = np.where(result<0.5, 0 ,1)
    return result
```

5.3 Model evaluation

```
# imports
import numpy as np
import pandas as pd
import time

def Acc_eval( y_hat, y):
  """
  Computes the accuracy of prediction y_hat with respect to known target y

  Parameters:
  ----------
  y, y_hat: matrixes of shape (n_features) containing known target and prediction

  Returns:
  -------
  accuracy: percentage of accuracy
  """
  return np.count_nonzero([y == y_hat]) / len(y) *100



def cross_val( estimator, X, y, n_folds):
  """
  Implements cross validation to obtain true error of a given model

  Parameters:
  -----------
  estimator: object of an estimator class with fit and predict methods
  X:{array} of shape (n_samples, n_features)
      Training vector, where n_samples is the number of samples and
      n_features is the number of features
  y:{array} of shape (n_samples), target vector relative to X
  n_folds: integer that indicates the number of sets to divide the data to
    make the cross validation

  Returns:
  -------
  accuracy: average accuracy through the n_folds of the cross validation. uses the function
  Acc_eval.
  """
  X_split = np.array_split(X,n_folds)
  y_split = np.array_split(y,n_folds)

  accuracies = []

  start_time = time.time()

  for fold in np.arange(n_folds):
    X_test = np.concatenate([X_split[i] for i in np.arange(n_folds) if i != fold])
    X_val = X_split[fold]
    y_test = np.concatenate([y_split[i] for i in np.arange(n_folds) if i != fold])
    y_val = y_split[fold]

    estimator.fit(X_test, y_test)
    y_hat = estimator.predict(X_val)
    accuracy_fold = Acc_eval(y_hat, y_val)
    accuracies += [accuracy_fold]

  # print("Cross Validation time:" + str(time.time() - start_time) + " seconds")
  return np.mean(accuracies)
```

5.4 Feature engineering functions

```
import numpy as np
import pandas as pd
from sklearn.utils import shuffle
```

```python
from model_evaluation import cross_val
import time

def re_sample(df_data, target_column_label = 'ClassLabel', random_state=42):
  df_data = shuffle(df_data, random_state = 42)
  X = df_data.drop(columns=target_column_label).copy()
  intercept = np.ones((X.shape[0], 1))
  X = np.concatenate((intercept, X), axis=1)
  y= df_data[target_column_label]
  return X, y;


def logarithm_transformer (data_frame):
  data= data_frame.copy()
  continuous_features = [ feature for feature in data if len(data[feature].unique()) > 25]
  for feature in continuous_features:
    if 0 not in data[feature].unique():
      data[feature] = np.log(data[feature])
  return data

def standard_scaler(data_frame):
  data= data_frame.copy()
  continuous_features = [ feature for feature in data if len(data[feature].unique()) > 25]
  for feature in continuous_features:
    data[feature] = data[feature] - np.mean(data[feature])
    data[feature] = data[feature] / np.std(data[feature])
  return data

def quadratic_feature_tester(data_frame, features):
  data= data_frame.copy()
  for feature in features:
    data[feature+'_2'] = data[feature] ** 2
  return data

def hola():
  print('hola')
```