```
Name: total_bill, dtype: int64
```

Linear Regression

Mathematics

```
# I've created the content of the cell below in a Word document in a much more readable format
# like I could use proper formatting for equations
# If you're interested please reach out to me and I'll send the .docx file form of the content below
```

```
# Notation
Bold variables are vectors and/or matrices, non-bold variables are scalars


'The Residual Error'
Typically: We assume ϵ is Gaussian Distributed (i.e. Gaussian noise)
ϵ ~ 𝒩(μ, σ2)
p(ϵ) = 𝒩(ϵ|0, σ2)
But other distributions are also possible


'The distribution of y'
the error is gaussian, thus y is gaussian too - a distribution derivated from a gaussian distribution is also gaussian
conditional probability density for the target variable y:
p (y|x,w,σ) = 𝒩(y|wT*x, σ2)
Notation: p(y|x,w,σ) = p(y|x,θ)


'General Linear Model'
f(x) = wT*x + ϵ = y,
where x0 = 1 and w0: intercept (w0 = b (bias) when y = mx + b, m (slope/coefficient) = w (weight))
# Reminder: bold variables are vectors and/or matrices, non-bold variables are scalars

Note: It is assumed that the noise is Gaussian distributed! i.e. ϵ ~ 𝒩(μ, σ2)
        -> We assume μ = 0: That means we only have to estimate σ!

The model parameters are:
θ = { w0, w1, … , wn, σ } = {w, σ}


'Optimal parameters'
We search over all possible θ and select the θ which most likely generated our training dataset 𝒟
θ* = argmax p(𝒟|θ), θ: mean and standard deviation of the data
        θ
This process is called Maximum Likelihood Estimation (MLE)

Data has a certain mean value and a sigma value, as well as parameters have a certain mean value and sigma value
Model: 𝒟 ~ 𝒩(μ, σ2), parameters: θ = {μ, σ}

p(𝒟|θ) = l(θ) and is called Likelihood: the likelihood of observing our data given a certain theta certain parametrization


'The Likelihood Function'
We assume each sample (each individual (x, y) combination like a vector) is independent, identically distributed (i.i.d)!
ℒ(θ) =  p(𝒟|θ)
= p(s0, s1, …, sn|θ)
= p(s0|θ) * p(s1|θ) * p(sn|θ)
= ∏(i=1)^N [[p(si|θ)]]


'The Log-Likelihood'
The product of small numbers in a computer is unstable
We transform the likelihood into the quasi-equivalent Log-Likelihood:
ℒ(θ) = ∏(i=1)^N [[p(si|θ)]]  ⇔  ∑(i=1)^N [[log[p(si|θ)]]  = l(θ)

The logarithm „just“ scales the problem. Likelihood and Log-Likelihood both have to be maximized!
```

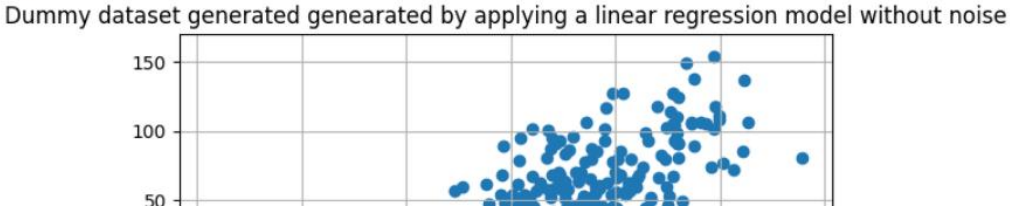
```
The logarithm „just“ scales the problem. Likelihood and Log-Likelihood both have to be maximized!


'The conditional distribution of p(si|θ)'
The conditional is the Gaussian distribution of y
p(s_i|θ) = p(y_i|x_i, w, σ) = 𝒩(y_i|w^T * x_i, σ2)        ,where w0 = intercept/bias and x0_i = 1

After inserting this into the log-likelihood,
simplifying and removing the constant parts like for example sigma, we get the Loss:
(sigma is variable of the input, but in the sake of modelling it is fixed (constant))
l(θ) = ∑(i=1)^N [[log[p(si|θ)]]
     = ∑(i=1)^N [[log[𝒩(y_i|w^T*x_i, σ2)]]
     = simplifying and omitting (removing) the constant parts like for example sigma
     = -∑(i=1)^N [[(y_i - w^T*x_i)^2]]  ,where w0 = intercept/bias and x0_i = 1

this (without the minus sign) is the Residual Sum of Squares
we only have to minimize this


'The Residual Sum of Squares'
We minimize the Loss term:
ℒ(θ)  =  ∑(i=1)^N [[(y_i - w^T * x_i)^2]] , where w0 = intercept/bias and x0_i = 1

For linear regression: The loss is shaped like a bowl with a unique minimum
Multiple ways to find the minimum:
- Analytical Solution - In practice too time consuming to compute
- Gradient Descend
- Newtons Method
- Particle Swarm Optimization
- Genetic Algorithms
- Many more methods

'MSE'
MSE = J(w,b) = 1/N * ∑(i=1)^N [[(y_i - (w*x_i + b))^2]]  (common notation)
             = 1/N * ∑(i=1)^N [[(y_i - w^T*x_i)^2]]  (vector notation)
                                  where w0 = b = intercept = bias
                                  x_1 = (1,   x_2 = (1,     x0_i = 1 always for the intercept,
                                         5)          10)    x1_1 = 5 and x2_1 = 10 are just examples


'Quality of the fit: R^2
A measure for the percentage of variability of the dependent variable (y) in the data explained by the model

Total sum of squares: SS_tot = ∑(i=1) [[(y_i - y_mean)^2]]
Residual sum of squares (unexplained variability): SS_res = ∑(i=1) [[(y_i - f(x_i)^2]]
Sum of squares regression (explained variability): SS_regr = ∑(i=1) [[(f(x_i) - y_mean)^2]]

Coefficient of determination: R^2 = 1-SS_res/SS_tot = SS_regr/SS_tot


'Basis Function expansion'
Polynomial regression fits a nonlinear relationship between input of x and the corresponding conditional mean of y
Example of a polynomial model:
y=w0 + w1*x + w2*x^2 + wm*x^m + ϵ
model is still linear in weights, quadratic or polynomic in x

How can we use polynomials in linear regression?
With Basis function expansion: apply a function before we predict our “linear line”:
f(x)= ∑_(j=1)^D [[wj * Φj(x) + ϵ]] = y
So we transform our Input-Space by using the function Φ

Example for the Basis Function:
Φ(x) = (1  x  x^2  x^3  x^4  x^5  x^6)^T

The model parameters are:
θ = {w0, w1, w2, w3, w4, w5, w6, σ}
σ: The standard deviation of the gaussian noise
```

Mathematics behind the sklearn.datasets.make_regression() function

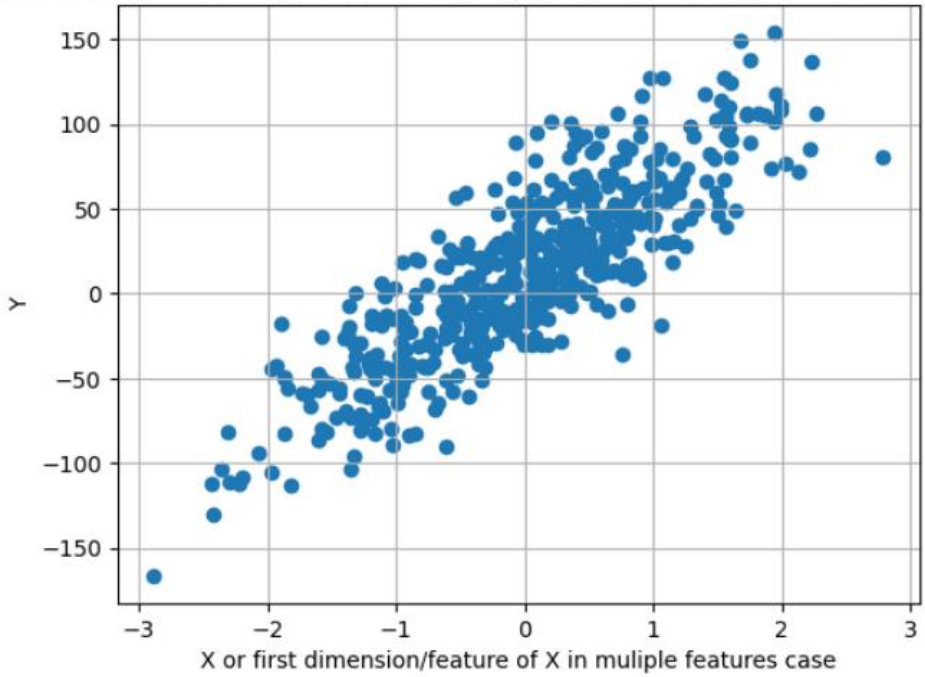
Mathematics behind the sklearn.datasets.make_regression() function

```
# Investigating how the underlying mathematics of Linar Regression work within
# sklearn's built-in make_regression() function

biasOrIntercept = 10
StDevOfNoise = 0

from sklearn import datasets
X, y, coefficients = datasets.make_regression(
        n_samples=500, n_features=2, bias = biasOrIntercept, noise=StDevOfNoise, random_state=4, coef = True)
# make.regression:
# generates random linear regression datasets, with added Gaussian noise to add some randomness.
#   The input set (X) can either be well conditioned ('kinda Gaus Distr') (by default) or have a low rank-fat tail singular profile
#   The output (Y) is generated by applying a (potentially biased) random linear regression model with
#    n_informative nonzero regressors to the previously generated input and some gaussian centered noise with some adjustable scale

# n_features: single/multiple features/variables for X |
#   Note: to display more than one feature the plot is not properly set
# noise: float, default=0.0 The standard deviation of the gaussian noise applied to the output
                  # In other words: apply some (gaussian centered) noise to the linear model
# coef: ndarray of shape (n_features,) The coefficient (the weights) of the underlying linear model

print(X.shape)
print(y.shape)

plt.grid()
plt.scatter(X[:, 0], y)
plt.title('Dummy dataset generated genearated by applying a linear regression model without noise')
plt.xlabel('X or first dimension/feature of X in muliple features case')
plt.ylabel('Y')
plt.show()

# The underlying linear model behind the generated data:
# f(x) = wT*x + ε = y, x0=1
    # ε ~ N(μ, σ2) ,where we assume μ = 0: that means we only have to estimate σ
    # The parameters of the model are:
    # θ = {w0, w1, … , wm, σ} = {w, σ}

print("w0 =",biasOrIntercept) # =b (bias) when y = mx + b, w0 = b
print("w1 =", coefficients[0].astype(float))
print("w2 =", coefficients[1].astype(float))
print("x0 =",1)
print("σ =",StDevOfNoise) # => ε = 0 because ε ~ N(μ, σ2), assumption μ = 0
print("ε = 0 <= As ε ~ N(μ, σ2) and σ = 0, μ = 0")
print()

for i in range(0,2):
    print(f"x1 = {X[i, 0]}, x2 = {X[i, 1]}, y = {y[i]}")

# With calculator
# x1 = 1.0910373925705656, x2 = -1.1104262019784206, y = 30.10364701683811
# f(x) = wT*x + ε = w0*x0 + w1*x1 + w2*x2 + 0(ε) = 10*1 + 48.59557438453652*1.0910373925705656 + 29.642619826054194*-1.1104262019784206 + 0 = 30.1036 = y

# x_1 = 0.010833796049367062, x_2 = -0.6803566054338689, y = -9.641077659237016
# f(x) = wT*x + ε = w0*x0 + w1*x1 + w2*x2 + 0(ε) = 10*1 + 48.59557438453652*0.010833796049367062 + 29.642619826054194*-0.6803566054338689 + 0 = -9.6410 = y
```

```
(500, 2)
(500,)
```



Dummy dataset generated genearated by applying a linear regression model without noise




```
w0 = 10
w1 = 48.59557438453652
w2 = 29.642619826054194
x0 = 1
σ = 0
ϵ = 0 <= As ϵ ~ 𝒩(μ, σ2) and σ = 0, μ = 0

x1 = 1.0910373925705656, x2 = -1.1104262019784206, y = 30.10364701683811
x1 = 0.010833796049367062, x2 = -0.6803566054338689, y = -9.641077659237016
```

Creating model from scratch using NumPy (with mathematics in comments)

```
# Introduction

# Even though Linear Regression only works with simple straight-line patterns (in the default case)
# understanding the math behind it helps us understand Gradient Descent and Loss Minimization methods,
# which are important for models used in machine learning and deep learning tasks


# General Linear Model

# f(x) = wT*x + ϵ = y, where x0 = 1 and w0: intercept (w0 = b (bias) when y = mx + b, m (slope/coefficients) = w (weights))
# Bold variables are vectors and/or matrices, non-bold variables are scalars

# It is assumed that the noise is Gaussian distributed i.e. ϵ ~ 𝒩(μ, σ2)
#    -> We assume μ = 0: That means we only have to estimate σ
# The model parameters are:
# θ = { w0,w1, … , wn, σ } = {w, σ}

# Note: sigma is variable of the input, but in the sake of modelling and optimization it is fixed (constant),
# therefore when we computed the Residual Sum of Squares (in the above Mathematics cell) it got removed,
# thus for further calculations like MSE (loss) we don't have to compute it

class LinRegNumPy:

# Class initialization
 # Setting the the learning rate and number of iterations hyperparameters to a default value for the Gradient Descent
 # The number of weight parameters depends on the input features within the data,
 # and as we do not have access to data yet, initializing them to None for now
    def __init__(self,
                 learning_rate = 0.1,
                 n_iters = 1000):
```

```python
class LinRegNumPy:

# Class initialization
 # Setting the the learning rate and number of iterations hyperparameters to a default value for the Gradient Descent
 # The number of weight parameters depends on the input features within the data,
 # and as we do not have access to data yet, initializing them to None for now
    def __init__(self,
                 learning_rate = 0.1,
                 n_iters = 1000):

        self.learning_rate = learning_rate
        self.n_iters = n_iters
        self.weights = None
        self.bias = None  # =intercept

    # Normalization of the independent variables
     # As the implementation is general, the model can deal with any number of features
     # Actually normalization is not needed for linear regression, as the
     # calculated coefficients account for the appropriate scaling
     # However, when using gradient descent-based optimization algorithms, feature scaling has some advantages,
     # for example it can help speed up convergence, help improve model performance, and
     # ensures that the coefficients are on the same scale, maaking it possible to compare
     # the impact of different predictors on the dependent variable
     # So I decided to include normalization
     # (it did actually improve performance as I first didn't include normalization and then compared results using it)
     # The other reason why I decided to include besides advantages is that other models do require normalization
     # (like Lasso, Ridge, Elastic Net regressions) because of the penalty coefficients,
     # and I want my model implementation to be as general as possible
    def _normalizeX(self, X):
        for i in range(X.shape[1]):
            X[:, i] = ((X[:, i] - np.mean(X[:,i]))/np.std(X[:, i]))
            # different method for averaging over columns of X:
            # X = ((X - np.mean(X, axis=0)) / np.std(X, axis=0)
            # computes less accurate results, please see the cell below for comparison

        return X

   # Cost/loss function
    # Let's take the Mean Squared Error (MSE) as the cost/loss function
    # (MSE is the mean of the Residual Sum of Squares, for further information please see the Mathematics cell above)
    # (another common cost function is the half MSE - division by two is for the sake of cleaner MSE derivation calculus)
    # We will find its minimum with Gradient Descent
    # MSE = J(w,b) = 1/N * ∑(i=1)^N ⟦(y_i - (w*x_i + b))^2⟧  (common notation)
    def _loss_function(self, X, y_pred, y):
        num_samples = X.shape[0]
        return (1 / num_samples) * np.sum((y_pred-y)**2)
        # same: return np.mean(np.square(y_pred-y))
        # same: return mean_squared_error(y_pred, y)


   # Gradient Descent
    # It's the training process to find optimal weights and bias
    # We aim to find optimal values of the parameters which generate a line
    # that minimizes the loss (converging towards minimum of the loss function),
    # so the squared difference between predicted and actual y values (Residual Sum of Squares)

    # There are multiple ways to find the minimum of the loss function (now MSE), for example Gradient Descent:
    # We take the partial derivates (gradients) of the MSE with respect to weights and bias
    # Note: When we have two or more derivatives of the same function, they are called a Gradient
        # J(w,b) = MSE = 1/N * ∑(i=1)^N ⟦(y_i - (w*x_i + b))^2⟧

    def _GradDesc_computeWeightDeriv(self, X, y_pred, y):
        num_samples = X.shape[0]
    # J'(w,b) = dJ/dw =
    # = dw (weight derivative) =
    # = 1/N * ∑(i=1)^N ⟦-2x_i * (y_i - (w*x_i + b))⟧ = 1/N * ∑(i=1)^N ⟦-2x_i * (y_i - y_i_pred))⟧ =
    # = 1/N * ∑(i=1)^N ⟦ 2x_i * (y_i_pred - y_i))⟧
    # We omit the constant 2
        return (1 / num_samples) * np.dot(X.T, y_pred - y)
         # X.T (num_features, num_samples) multiply by (y_pred - y) (num_samples, ) = dw (num_features, )
         # So we have a separate derivate value for each weight, and we optimize them separately
```

```
    # We omit the constant 2
        return (1 / num_samples) * np.dot(X.T, y_pred - y)
         # X.T (num_features, num_samples) multiply by (y_pred - y) (num_samples, ) = dw (num_features, )
         # So we have a separate derivate value for each weight, and we optimize them separately

    def _GradDesc_computeBiasDeriv(self, X, y_pred, y):
        num_samples = X.shape[0]
    # J'(w,b) = dJ/db =
    # = db (bias derivative) =
    # = 1/N * ∑(i=1)^N [[-2 * (y_i - (w*x_i + b))]] = 1/N * ∑(i=1)^N [[-2 * (y_i - y_i_pred))]] =
    # = 1/N * ∑(i=1)^N [[ 2 * (y_i_pred - y_i))]]
    # We omit the constant 2
        return (1 / num_samples) * np.sum(y_pred - y)
        # (y_pred - y).shape: (num_samples, ), thus db.shape: (num_samples, )
        # So the derivate value for the bias is a single value just as the bias is


    # We took the gradient of the loss with respect to each weight
    # We need to push J(w) towards its minimum (converge towards minimum)
    # The gradient of MSE is usually initially positive (but depends on the initial weights),
    # so we decrease the weight to move towards the minimum
    # (moving backwards on the x-axis or 'moving the values in the opposite direction of the gradient, because the gradient of MSE usually points in the direction of increase not decrease, but this depends on the initial weights'):
    # weights = weights - alpha (learning rate) * dw (weights partial derivative of the loss)
    # bias = bias - alpha (learning rate) * db (bias partial derivative of the loss)
    # The learning rate (or alpha) controls (scales) the size of the steps
    # We start with random weight values and change them slightly for multiple steps
    # We only make a small change in the value for each step for stable movement towards the minimum
    def _GradDesc_updateWeightValues(self, dw):
        return self.weights - self.learning_rate * dw

    def _GradDesc_updateBiasValue(self, db):
        return self.bias - self.learning_rate * db

   # Limitations of the Gradient Descent
    # There is no guarantee that the algorithm converges to the global optimum, for example:
    # The learning rate can lead to slow convergence if not properly configured
    # Or it will never converge because the update value is too large (just jumping around minimum value)
    # The starting parameters can lead to a solution stuck in a local minimum (see
    # there are ways to adapt gradually during learning the learning rate:
    # dynamic learning rate update, adaptive learning rate


    # Plotting the loss/cost in the function of num_iters
    def _plot_cost(self, MSELossFuncValues):
        plt.grid()
        plt.xlabel('Epochs (Number of iterations)')
        plt.ylabel('MSE (Loss/Cost function)')
        plt.plot(range(self.n_iters), MSELossFuncValues, 'm', linewidth = "5")
        plt.xticks(np.linspace(0, self.n_iters, 11))
        plt.yticks(np.linspace(0, int(round(MSELossFuncValues.max(),-3)), 20))
        plt.show()


    def fit(self, X, y, GradDescPlot = False):

       # Training initialization
        # Normalizing data of independent variables
        X = self._normalizeX(X)

        # Reading data of independent variables shape
        num_samples, num_features = X.shape  # X.shape: (num_samples, num_features)

        # Assigning random initial values for weights and 0 for bias,
        self.weights = np.random.rand(num_features)  # weights shape:  (num_features, )
          # np.random.rand: Create an array of the given shape and populate it with random samples
          # from a uniform distribution over [0, 1)
        self.bias = 0

        # Creating collector for MSE values of each iteration
        MSELossFuncValues = np.empty(0)
```

```
        # from a uniform distribution over [0, 1)
        self.bias = 0

        # Creating collector for MSE values of each iteration
        MSELossFuncValues = np.empty(0)


       # Tranining process
        # Incrementally performing the steps to find optimal values for the parameters
        # by converging towards minimum of the loss function
        for i in range(self.n_iters):

            # 1. step: Get predictions with current weights
                # so that in the next step we are able to compute the partial derivatives (gradient) of the Loss function
             # X (num_samples, num_features) multiply by weights (num_features, ) = y_pred (num_samples, )
             # The bias value is added at the end of each prediction
            y_pred = np.dot(X, self.weights) + self.bias # could create a '_predict' function instead and
                                                         # use here and for the 'predict' function,
                                                         # but this approach seems to be worse for me here... ?

            # ----------------- just for investigating the Gradient Descent in action -----------
            # If GradDescPlot is set to True,
            # then compute and collect Loss (MSE) values in each iteration during the training process
            if GradDescPlot == True:
                MSELossFunc = self._loss_function(X, y_pred, y)
                MSELossFuncValues = np.append(MSELossFuncValues, MSELossFunc)

                if i % 100 == 0:
                        print(f"n_iters: {i}, MSE LinRegNumPy: {MSELossFunc}")
                        print(f"Current value(s) for weight(s) in this iteration: {np.float64(self.weights)}\n")
                        # for checking: print(f"n_iters: {i} || MSE BuiltIn: {mean_squared_error(y_pred,y)}")
            # ----------------- just for investigating the Gradient Descent in action -----------


            # 2. step: Take the weights and bias partial derivatives (gradient) of the Loss function (MSE)
                # so that in the next step we are able to update the parameters
            # Partial derivative values in the current iteration of GradDesc:
            weightDeriv = self._GradDesc_computeWeightDeriv(X, y_pred, y)
            biasDeriv = self._GradDesc_computeBiasDeriv(X, y_pred, y)


            # 3. step: Update weights and bias
                # so that with the updated parameters predictions have (produce) a smaller Loss
            self.weights = self._GradDesc_updateWeightValues(weightDeriv)
            self.bias = self._GradDesc_updateBiasValue(biasDeriv)


        # ----------------- just for investigating the Gradient Descent in action -----------
        # Plotting the loss/cost in the function of num_iters
        if GradDescPlot == True:
            print(f"Final value(s) for weight(s) set by {self.n_iters} training interations (iterations of minimizing the Loss): {np.float64(self.weights)}")
            self._plot_cost(MSELossFuncValues)
        # ----------------- just for investigating the Gradient Descent in action -----------

        return self


    # Predicting with trained model
     # After training (updating the parameters in incremental steps) now we have the optimal set of weights and bias
     # The predicted values should now be close to the original values
     # X (num_samples, num_features) multiply by weights (num_features, ) = y_pred (num_samples, )
     # The bias value is added at the end of each prediction
     def predict(self, X):
         X = self._normalizeX(X)
         return np.dot(X, self.weights) + self.bias
```

```
# Comparing different column averaging methods for normalization:
 # np.mean/std(X, axis=0) vs. np.mean/std(X[:, columnIndex])
```

```
# Comparing different column averaging methods for normalization:
 # np.mean/std(X, axis=0) vs. np.mean/std(X[:, columnIndex])

X, y = datasets.make_regression(n_samples=600, bias = 30, n_features=2, noise = 15, random_state=4)
X2 = X.copy()

X2 = ((X2 - np.mean(X2, axis=0)) / np.std(X2, axis=0))

X3 = X.copy()
for i in range(0, X3.shape[1]):
          X3[:, i] = ((X3[:, i] - np.mean(X3[:,i]))/np.std(X3[:, i]))


comparison = {
  'Mean of X columns using axis': [np.mean(X, axis=0), np.mean(X2, axis=0), np.mean(X3, axis=0)],
  'Mean of X columns with indexing': [[np.mean(X[:, 0]), np.mean(X[:, 1])], [np.mean(X2[:, 0]), np.mean(X2[:, 1])], [np.mean(X3[:, 0]), np.mean(X3[:, 1])]],
  'Mean of whole X': [X.mean(), X2.mean(), X3.mean()],
   ' ': ['','',''],
  'StD of X columns using axis': [np.std(X, axis=0), np.std(X2, axis=0), np.std(X3, axis=0)],
  'StD of X columns with indexing': [[np.std(X[:, 0]), np.std(X[:, 1])], [np.std(X2[:, 0]), np.std(X2[:, 1])], [np.std(X3[:, 0]), np.std(X3[:, 1])]],
  'StD of whole X': [X.std(), X2.std(), X3.std()]
}

df = pd.DataFrame.from_dict(comparison, orient='index', columns=['Original X', 'X normalized using axis', 'X normalized with indexing'])
df.index.name = 'Measure on (normalized) X'
pd.set_option("display.precision", 18)
display(df)
```

Measure on (normalized) X	Original X	X normalized using axis	X normalized with indexing
Mean of X columns using axis	[0.0007967313319007003, 0.09053283843981025]	[4.838722015657974e-17, -2.312964634635743e-17]	[3.839521293495333e-17, 5.736152293896642e-18]
Mean of X columns with indexing	[0.0007967313319007137, 0.0905328384398102]	[2.960594732333751e-17, -6.513308411134252e-17]	[2.8125649957170635e-17, -2.0724163126336256e-17]
Mean of whole X	0.045664784885855458	-0.000000000000000015	0.000000000000000006
StD of X columns using axis	[0.9757849010980479, 0.9540903315309516]	[0.9999999999999998, 1.0]	[0.9999999999999994, 1.0000000000000002]
StD of X columns with indexing	[0.9757849010980482, 0.9540903315309516]	[1.0000000000000002, 1.0]	[1.0, 1.0]
StD of whole X	0.96604110119373543	1.000000000000000222	1.0

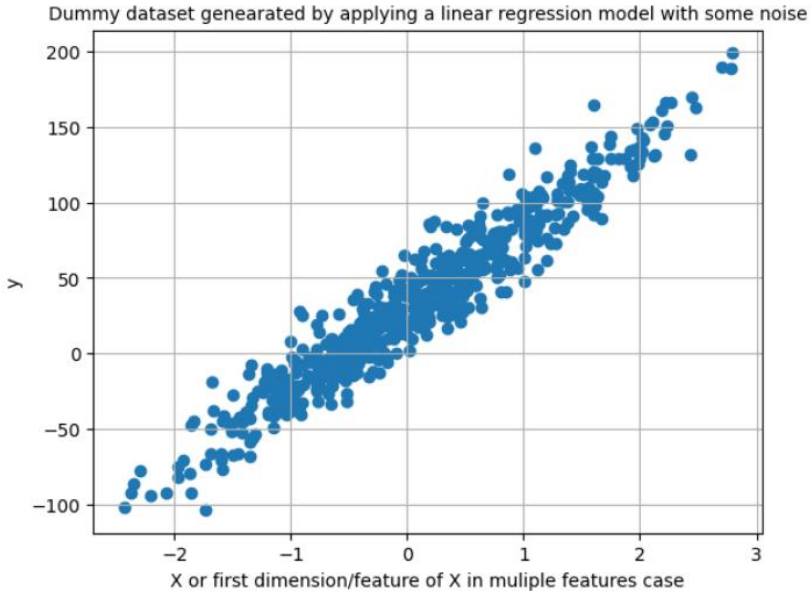
Applying created model on data, visualizing Gradient Descent in action and performance benchmarking

```
# Using the created model on data


# Dataset (dummy) generating method with Scikit-Learn by applying a linear regression model with some noise to X
# X: well conditioned data ('kinda' Gaus Distr)
# Y: generated by applying a (potentially biased) linear regression model
# For the interpretation of this sklearn function and for more information please see the
# '### Mathematics behind the sklearn.datasets.make_regression() function' Notebook heading
X, y = datasets.make_regression(n_samples=600, bias = 30, n_features=1, noise=15, random_state=4)
# Note: to display multiple features the plots below are not properly set

num_samples, num_features = X.shape
print("Number of generated samples:", num_samples)
print("Number of generated features:", num_features)

# Dataset plotting
plt.figure(figsize=(6.6,5))
plt.grid()
plt.title('Dummy dataset genearated by applying a linear regression model with some noise', fontsize = 10)
plt.xlabel('X or first dimension/feature of X in muliple features case')
plt.ylabel('y')
```

```
plt.xlabel('X or first dimension/feature of X in muliple features case')
plt.ylabel('y')
plt.scatter(X[:, 0], y)
plt.show()


# Dataset splitting
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1235)
print("Train shape:", X_train.shape)
print("Test shape:", X_test.shape)


# Performance of NumPy model without training (random values for weights and 0 for bias)
regressorNoTrain = LinRegNumPy(n_iters=0)
y_pred_noTrain = regressorNoTrain.fit(X_train, y_train).predict(X_test)

MSE_ModelsOwnFunc = regressorNoTrain._loss_function(X_test, y_pred_noTrain, y_test)
MSE_NumPy = np.mean(np.square(y_pred_noTrain - y_test))
MSE_scikitlearn = mean_squared_error(y_pred_noTrain, y_test)

plt.figure(figsize=(6.6,5))
plt.grid()
plt.title('LinRegNumPy without training (random weights, bias=30)')
plt.xlabel('X or first dimension/feature of X in muliple features case')
plt.ylabel('y')
plt.scatter(X_train[:, 0], y_train, label = 'Training set')
plt.scatter(X_test[:, 0], y_test, color = 'green', label = 'Test set')
#plt.scatter(X[:, ], y, label='Whole dataset')
plt.plot(X_test[:, 0], y_pred_noTrain, linewidth=0.8, color='black', label='Predictions on test set without training')
plt.legend(loc='upper left')
plt.text(1.1, 0, f'\nMSE ModelsFunc: {round(MSE_ModelsOwnFunc,5)}\nMSE NumPy: {round(MSE_NumPy,5)}\nMSE sklearn: {round(MSE_scikitlearn,5)}', fontsize = '9', ha='left', va='top')
plt.show()



# Performance of NumPy model with training
regressor = LinRegNumPy(n_iters=1001)
y_pred = regressor.fit(X_train, y_train).predict(X_test)

MSE_ModelsOwnFunc = regressor._loss_function(X_test, y_pred, y_test)
MSE_NumPy = np.mean(np.square(y_pred - y_test))
MSE_scikitlearn = mean_squared_error(y_pred, y_test)

plt.figure(figsize=(6.6,5))
plt.grid()
plt.title('LinRegNumPy with training')
plt.xlabel('X or first dimension/feature of X in muliple features case')
plt.ylabel('y')
plt.scatter(X_train[:, 0], y_train, label = 'Training set')
plt.scatter(X_test[:, 0], y_test, color = 'green', label = 'Test set')
#plt.scatter(X[:, ], y, label='Whole dataset')
plt.plot(X_test[:, 0], y_pred, linewidth=0.8, color='black', label='Predictions on test set after training')
plt.legend(loc='upper left')
plt.text(1.2, 0, f'\nMSE ModelsFunc: {round(MSE_ModelsOwnFunc,5)}\nMSE NumPy: {round(MSE_NumPy,5)}\nMSE sklearn: {round(MSE_scikitlearn,5)}', fontsize = '9', ha='left', va='top')
plt.show()



# Performance of Scikit-Learn's built-in LinearRegression model (with training)
regressorSklearn = sklearn.linear_model.LinearRegression()
y_pred = regressorSklearn.fit(X_train, y_train).predict(X_test)

sklearnModel_MSE_NumPy = np.mean(np.square(y_pred - y_test))
sklearnModel_MSE_scikitlearn = mean_squared_error(y_pred, y_test)

plt.figure(figsize=(6.6,5))
plt.grid()
plt.title('LinearRegression() from sklearn with training')
plt.xlabel('X or first dimension/feature of X in muliple features case')
plt.ylabel('y')
plt.scatter(X_train[:, 0], y_train, label = 'Training set')
```

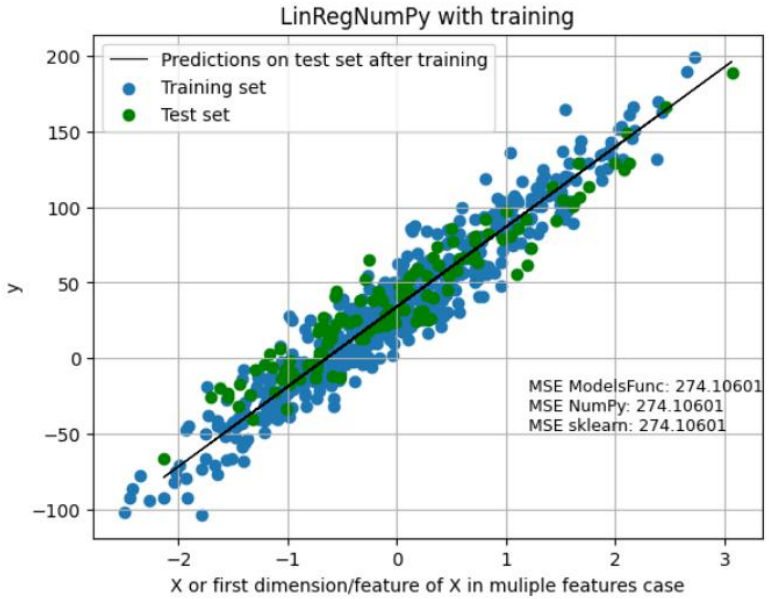
```
plt.grid()
plt.title('LinearRegression() from sklearn with training')
plt.xlabel('X or first dimension/feature of X in muliple features case')
plt.ylabel('y')
plt.scatter(X_train[:, 0], y_train, label = 'Training set')
plt.scatter(X_test[:, 0], y_test, color = 'green', label = 'Test set')
#plt.scatter(X[:, ], y, label='Whole dataset')
plt.plot(X_test[:, 0], y_pred, linewidth=0.8, color='black', label='Predictions on test set after training')
plt.legend(loc='upper left')
plt.text(1.5, 0, f'\nMSE NumPy: {round(sklearnModel_MSE_NumPy,5)}\nMSE sklearn: {round(sklearnModel_MSE_scikitlearn,5)}', fontsize = '9', ha='left', va='top')
plt.show()
```

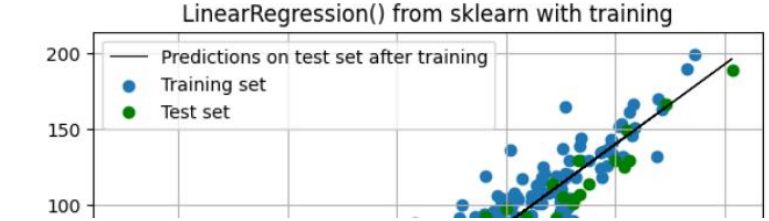
```
Number of generated samples: 600
Number of generated features: 1
```

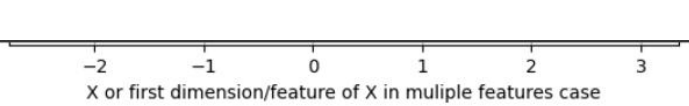


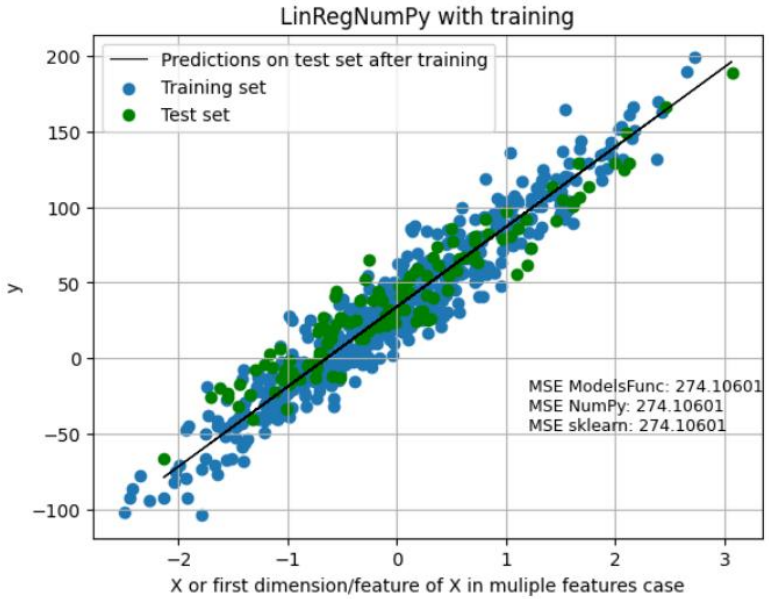

```
Train shape: (480, 1)
Test shape: (120, 1)
```

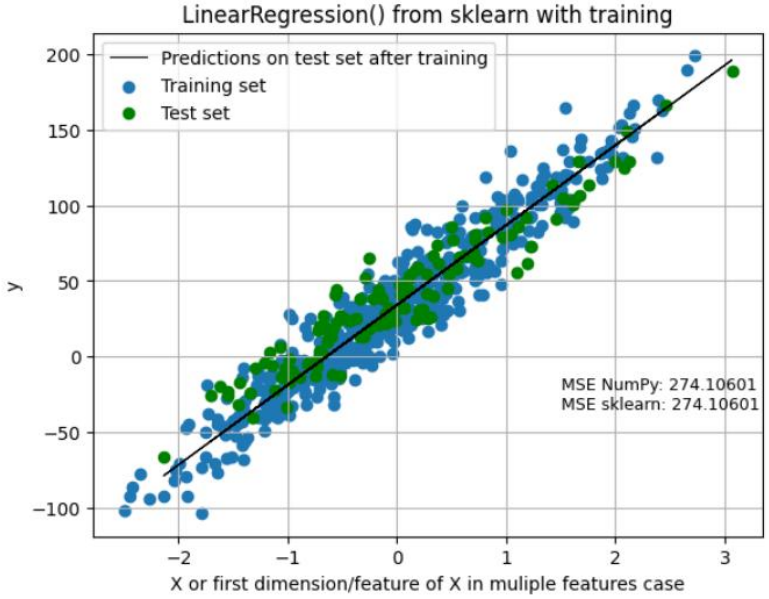
```
Train shape: (480, 1)
Test shape: (120, 1)
```







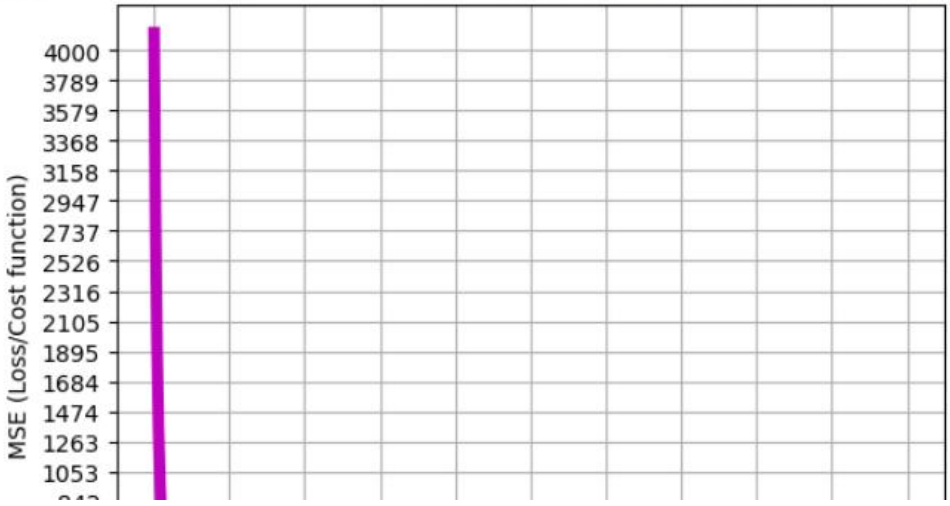




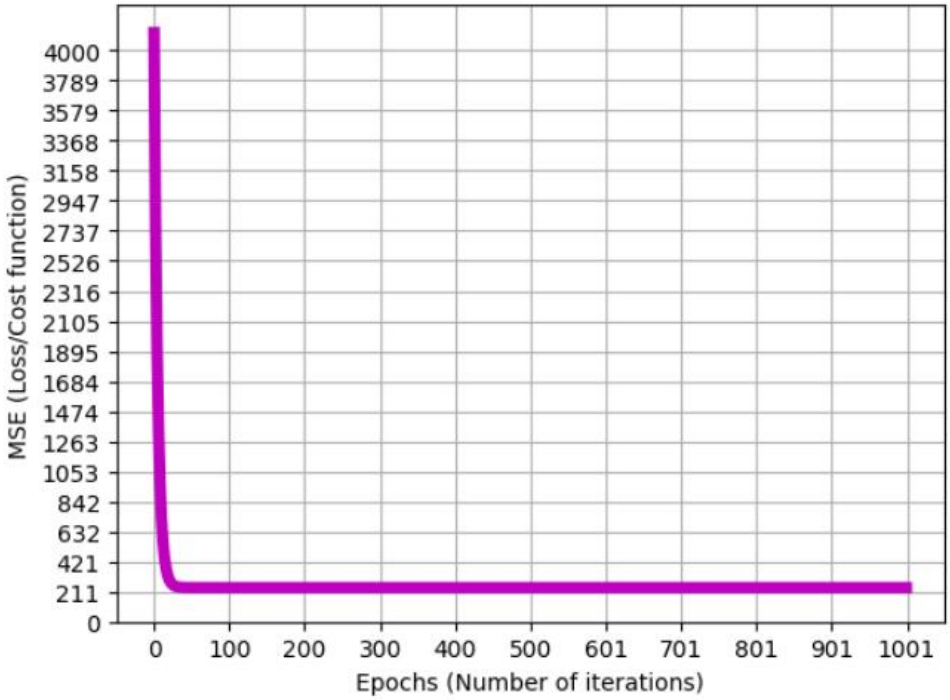

```
# Visualizing Gradient Descent in work during the training process:
 # Plot of the loss/cost in the function of num_iters
 # Two different learning rates for comparison

# In the above cell we already trained, predicted with and calculated MSE with the
# learning_rate = 0.1 model (default setting for the LinRegNumPy class)
# Here we only call it (and not assign it to a variable) just for visualization (GradDescplot = True)
```

```
# Visualizing Gradient Descent in work during the training process:
 # Plot of the loss/cost in the function of num_iters
 # Two different learning rates for comparison

# In the above cell we already trained, predicted with and calculated MSE with the
# learning_rate = 0.1 model (default setting for the LinRegNumPy class)
# Here we only call it (and not assign it to a variable) just for visualization (GradDescplot = True)
LinRegNumPy(n_iters=1001, learning_rate = 0.1).fit(X_train, y_train, GradDescPlot = True)


# Different learning rate to do comparison in the next cell
lowerLRregressor = LinRegNumPy(n_iters=1001, learning_rate=0.01)
# So far we don't have MSE score for the learning_rate = 0.01 model, therefore we predict with it and calculate MSE
y_pred = lowerLRregressor.fit(X_train, y_train, GradDescPlot=True).predict(X_test)
lowerLR_MSE_NumPy = np.mean(np.square(y_pred - y_test))
```

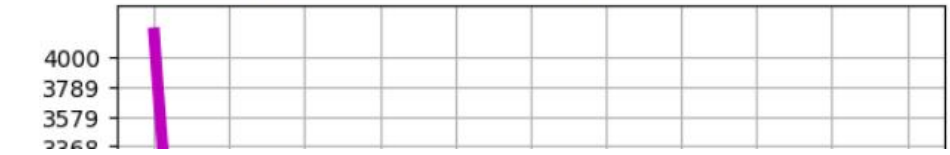
```
n_iters: 0, MSE LinRegNumPy: 4124.079402046711
Current value(s) for weight(s) in this iteration: 0.6356580358223799

n_iters: 100, MSE LinRegNumPy: 239.13626897670375
Current value(s) for weight(s) in this iteration: 52.79759768708586

n_iters: 200, MSE LinRegNumPy: 239.13626623584562
Current value(s) for weight(s) in this iteration: 52.79898318117169

n_iters: 300, MSE LinRegNumPy: 239.13626623584562
Current value(s) for weight(s) in this iteration: 52.79898321797235

n_iters: 400, MSE LinRegNumPy: 239.13626623584562
Current value(s) for weight(s) in this iteration: 52.798983217973294

n_iters: 500, MSE LinRegNumPy: 239.13626623584562
Current value(s) for weight(s) in this iteration: 52.798983217973294

n_iters: 600, MSE LinRegNumPy: 239.13626623584562
Current value(s) for weight(s) in this iteration: 52.798983217973294

n_iters: 700, MSE LinRegNumPy: 239.13626623584562
Current value(s) for weight(s) in this iteration: 52.798983217973294

n_iters: 800, MSE LinRegNumPy: 239.13626623584562
Current value(s) for weight(s) in this iteration: 52.798983217973294

n_iters: 900, MSE LinRegNumPy: 239.13626623584562
Current value(s) for weight(s) in this iteration: 52.798983217973294

n_iters: 1000, MSE LinRegNumPy: 239.13626623584562
Current value(s) for weight(s) in this iteration: 52.798983217973294

Final value(s) for weight(s) set by 1001 training interations (iterations of minimizing the Loss): 52.798983217973294
```

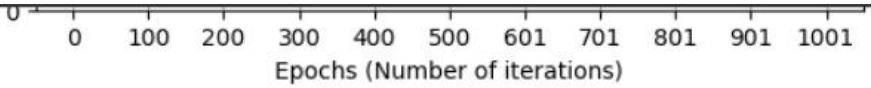


```
n_iters: 1000, MSE LinRegNumPy: 239.13626623584562
Current value(s) for weight(s) in this iteration: 52.798983217973294

Final value(s) for weight(s) set by 1001 training interations (iterations of minimizing the Loss): 52.798983217973294
```

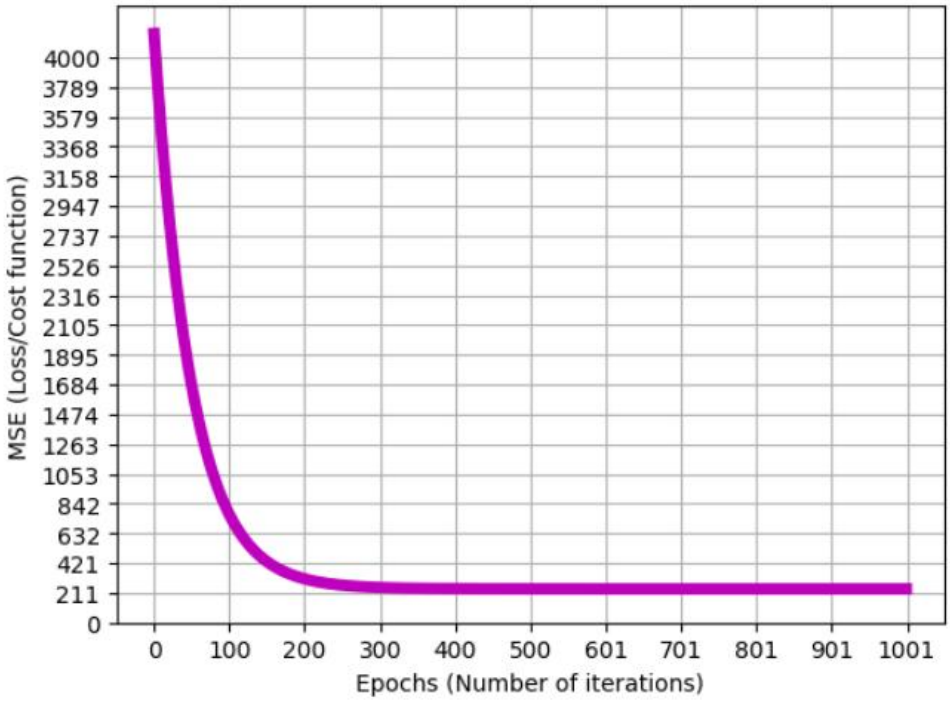



```
n_iters: 0, MSE LinRegNumPy: 4166.358333635428
Current value(s) for weight(s) in this iteration: 0.23196479722265828

n_iters: 100, MSE LinRegNumPy: 765.304201921055
Current value(s) for weight(s) in this iteration: 33.55775439167297

n_iters: 200, MSE LinRegNumPy: 309.6320751796347
Current value(s) for weight(s) in this iteration: 45.75607118170867

n_iters: 300, MSE LinRegNumPy: 248.5812717969835
Current value(s) for weight(s) in this iteration: 50.22104963595796

n_iters: 400, MSE LinRegNumPy: 240.40170500995845
Current value(s) for weight(s) in this iteration: 51.85537615330138

n_iters: 500, MSE LinRegNumPy: 239.3058093113539
Current value(s) for weight(s) in this iteration: 52.45359251484948

n_iters: 600, MSE LinRegNumPy: 239.15898156197667
Current value(s) for weight(s) in this iteration: 52.672559050254904

n_iters: 700, MSE LinRegNumPy: 239.13930962785494
Current value(s) for weight(s) in this iteration: 52.752707883869846

n_iters: 800, MSE LinRegNumPy: 239.13667398851751
Current value(s) for weight(s) in this iteration: 52.78204494908824

n_iters: 900, MSE LinRegNumPy: 239.13632086641604
Current value(s) for weight(s) in this iteration: 52.79278326375621

n_iters: 1000, MSE LinRegNumPy: 239.13627355523167
Current value(s) for weight(s) in this iteration: 52.79671383421545

Final value(s) for weight(s) set by 1001 training interations (iterations of minimizing the Loss): 52.79673652805303
```

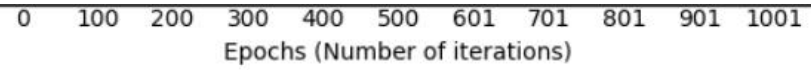





```
n_iters: 0, MSE LinRegNumPy: 4166.358333635428
Current value(s) for weight(s) in this iteration: 0.23196479722265828

n_iters: 100, MSE LinRegNumPy: 765.304201921055
Current value(s) for weight(s) in this iteration: 33.55775439167297

n_iters: 200, MSE LinRegNumPy: 309.6320751796347
Current value(s) for weight(s) in this iteration: 45.75607118170867

n_iters: 300, MSE LinRegNumPy: 248.5812717969835
Current value(s) for weight(s) in this iteration: 50.22104963595796

n_iters: 400, MSE LinRegNumPy: 240.40170500995845
Current value(s) for weight(s) in this iteration: 51.85537615330138

n_iters: 500, MSE LinRegNumPy: 239.3058093113539
Current value(s) for weight(s) in this iteration: 52.45359251484948

n_iters: 600, MSE LinRegNumPy: 239.15898156197667
Current value(s) for weight(s) in this iteration: 52.672559050254904

n_iters: 700, MSE LinRegNumPy: 239.13930962785494
Current value(s) for weight(s) in this iteration: 52.752707883869846

n_iters: 800, MSE LinRegNumPy: 239.13667398851751
Current value(s) for weight(s) in this iteration: 52.78204494908824

n_iters: 900, MSE LinRegNumPy: 239.13632086641604
Current value(s) for weight(s) in this iteration: 52.79278326375621

n_iters: 1000, MSE LinRegNumPy: 239.13627355523167
Current value(s) for weight(s) in this iteration: 52.79671383421545

Final value(s) for weight(s) set by 1001 training interations (iterations of minimizing the Loss): 52.79673652805303
```




```
# Comparing paramater values of NumPy from scratch model vs Scikit-Learn's model after training:
#   Comparison of parameter values set by training and the MSE scored by predictions on the Test set

comparison_11Decimals = {
  'LinearRegression() sklearn ': [regressorSklearn.coef_.round(11), regressorSklearn.intercept_, sklearnModel_MSE_NumPy],
```




```
# Comparing paramater values of NumPy from scratch model vs Scikit-Learn's model after training:
#   Comparison of parameter values set by training and the MSE scored by predictions on the Test set

comparison_11Decimals = {
  'LinearRegression() sklearn ': [regressorSklearn.coef_.round(11), regressorSklearn.intercept_, sklearnModel_MSE_NumPy],
  'LinRegNumPy lr=0.1 ': [regressor.weights.round(11), regressor.bias, MSE_NumPy],
  'LinRegNumPy lr=0.01': [lowerLRregressor.weights.round(11), lowerLRregressor.bias, lowerLR_MSE_NumPy]
   }

df = pd.DataFrame.from_dict(comparison_11Decimals, orient='index', columns=['Weight(s)/Coefficient(s) (param)', 'Bias/Interc
pd.set_option("display.precision", 11)
display(df)

comparison_15Decimals = {
  'LinearRegression() sklearn ': [[np.float64(regressorSklearn.coef_)], regressorSklearn.intercept_, sklearnModel_MSE_NumPy]
  'LinRegNumPy lr=0.1 ': [[np.float64(regressor.weights)], regressor.bias, MSE_NumPy],
  'LinRegNumPy lr=0.01': [[np.float64(lowerLRregressor.weights)], lowerLRregressor.bias, lowerLR_MSE_NumPy]
   }

df = pd.DataFrame.from_dict(comparison_15Decimals, orient='index', columns=['Weight(s)/Coefficient(s) (param)', 'Bias/Interc
pd.set_option("display.precision", 15)
display(df)

# Interpretation of results
 # Note: after the current Kernel has been restarted the values I pasted here won't be exactly the same,
 # but the relation (smaller/bigger) between them will be the same, so the interpretation is the same

 # Comparing the two dataframes:
 # The first df suggests that LinRegNumPy lr=0.1 model's parameter values are equal (or at least very close)
 # to the values of the imported sklearn model, which validates the created NumPy class.
 # On the second more precise df we can see that although the  values are very close to each other,
 # they're not exactly the same. This might be because sklearn uses not exactly the same methods
 # for creating their LinearRegression() class

 # Comparing different learning rates:
 # With a lower learning rate we get a better (smaller) Mean Squared Error score on the predictions,
 # whereas with this lower learning rate we reached a worse loss function value at the 1000.th iteration of
 # Gradient Descent: 239.13627355841672. With a higher learning rate already just at the 100.th ireation GradDesc
 # reached a smaller loss function value (239.1362689603656)
 # This could mean that with a higher learning rate we overfitted (just a little bit, but still did) the model
 # IN COMPARISON to training with lower learning rate, that might be why we get better prediction MSE score with lower LR
```

	Weight(s)/Coefficient(s) (param)	Bias/Intercept (param)	MSE (metric of preds on Test set)
LinearRegression() sklearn	[52.79898321797]	34.11642774020	274.10601178077
LinRegNumPy lr=0.1	[52.79898321797]	34.11642774020	274.10601178077
LinRegNumPy lr=0.01	[52.79673652805]	34.11496961994	274.08152930247

	Weight(s)/Coefficient(s) (param)	Bias/Intercept (param)	MSE (metric of preds on Test set)
LinearRegression() sklearn	[52.79898321797334]	34.116427740196400	274.106011780774736
LinRegNumPy lr=0.1	[52.798983217973294]	34.116427740196372	274.106011780774281
LinRegNumPy lr=0.01	[52.79673652805303]	34.114969619941085	274.081529302467459

LinReg presentacio kerdesek docxbol (fájlok között) a matekos reszeket bemasolni a cellakba
further considerations cella a class létrehozás alá közvetlenül
reviewed notebookokbol a static fv-ek es egyeb dolgok (pl átnvezeések) áthozása az eles notebookba

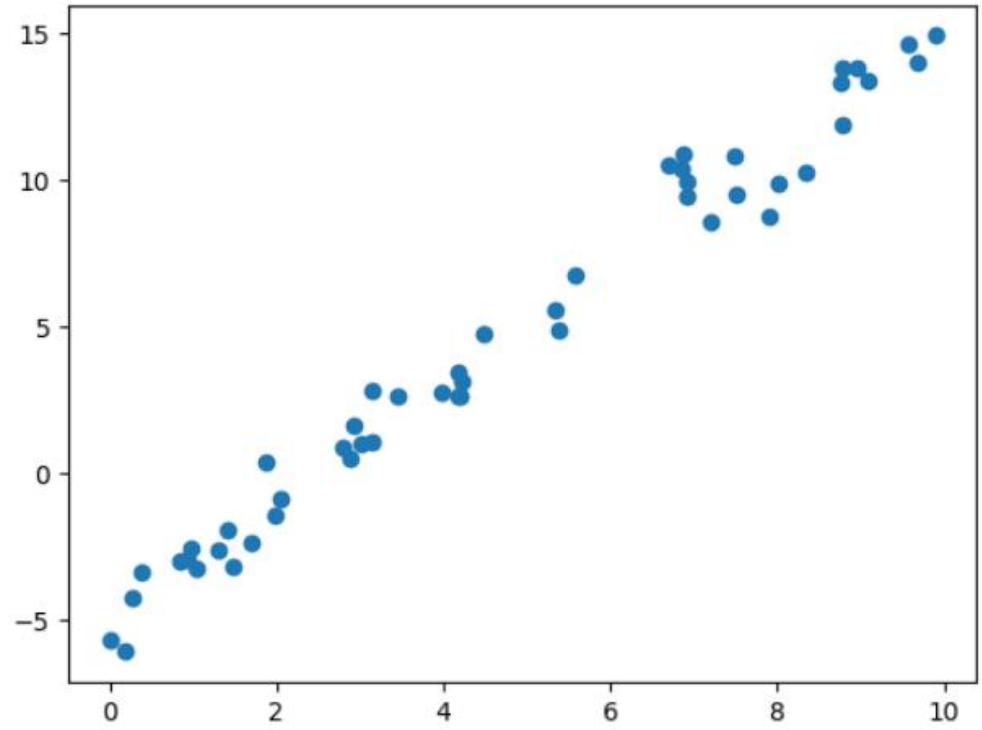
```
# Further considerations
```

```
# Further considerations

    # Rename normalization to standardization and use mean and std of train set to standardize test set

    # Use num_features inside 'fit' function for calling other _functions instead of calculating num_features inside the other _functions - didn't work?

    # Proper plot for multipe features

    # Create '_predict' func and use inside 'predict' and 'fit' instead of current approach - would be better or not?

    # Interpretation of the results of the cell above - correct?

    # for the LinRegNumPy class use static methods instead where possible, for example for def _GradDesc_updateWeightValues

    # Put mathematical formulas into LaTeX form: $\sigma(x)\$

    # Add R2 to the metrics used

# Notes for myself
        # Amikor lr 0.1 es modell MSE (metric of preds on Test set) kisebb, ezért lehet:
        # a training alatt az 1000. iteracioban a 0.1 es modell MSEje (graddesces) kisebb mint a 0.01esé,
        # tehát a 0.1-esnek optimalisabb paramtereket sikerült találnia a training alatt, igy a test seten optimalisabbak a predikciok
        # ha a jobb training ellenére a predikcios MSE magasabb lenne (legutobbi output esetben), az azt jelentené sztem h már tul optimalis/minimális értékét találtuk meg a lossnak (MSEnek) a training alatt, tehát olyan paraméter értékeket amik a training setre márt túl jók (túltanítás)
```

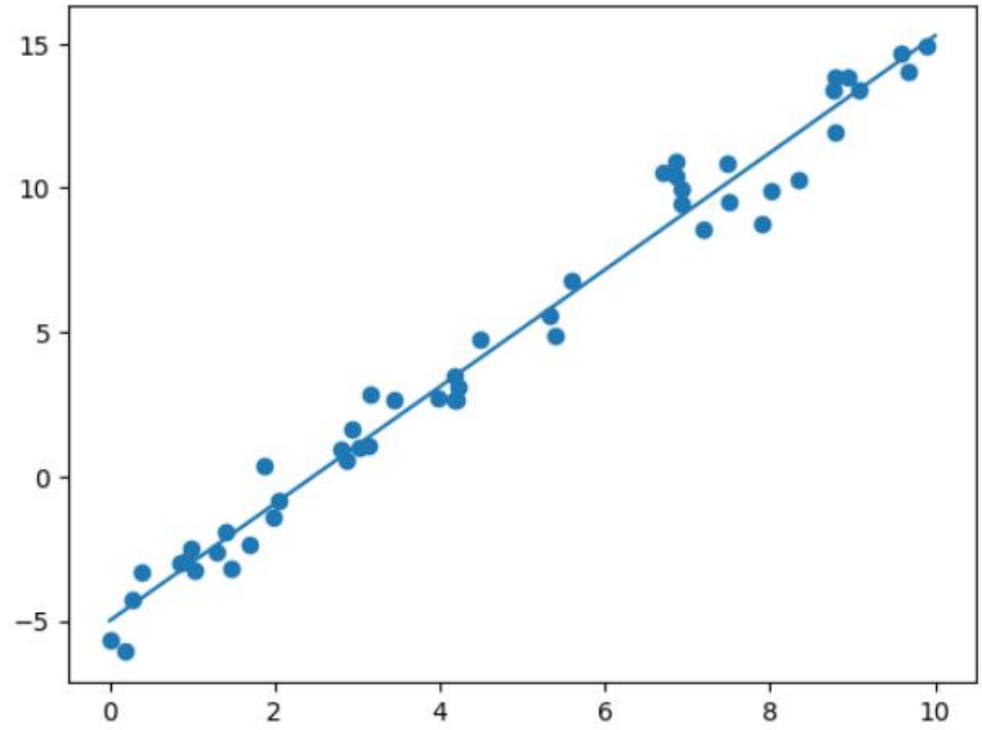
Model extensions (Polynomial and Gaussian basis functions, Regularization techniques)

```
# Linear Regression with single feature data and with multidimensional data

# Data which is scattered about a line with a slope of 2 and an intercept of -5

rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = 2 * x - 5 + rng.randn(50)

plt.scatter(x, y)
plt.show()

print(x.shape, x.reshape(-1, 1).shape)

# Estimator to fit data and construct the best-fit line
model = LinearRegression(fit_intercept=True)
model.fit(x.reshape(-1, 1), y)


xfit = np.linspace(0, 10, 1000)
yfit = model.predict(xfit.reshape(-1, 1))

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.show()

# Parameteres of the model
print("Model slope:    ", model.coef_[0])
print("Model intercept:", model.intercept_)
# they're not perfectly 2 and -5 because of the noise that we added to y: + rng.randn(50)


# Handling multidimensional data
rng = np.random.RandomState(1)
X = 10 * rng.rand(100, 3)
y = 0.5 + np.dot(X, [1.5, -2., 1.])
print()
print(X.shape, y.shape)
print(X[:2])
print(y[:2])

model.fit(X, y)
 # Geometrically, this is akin to fitting a plane to points in three dimensions, or fitting a
```

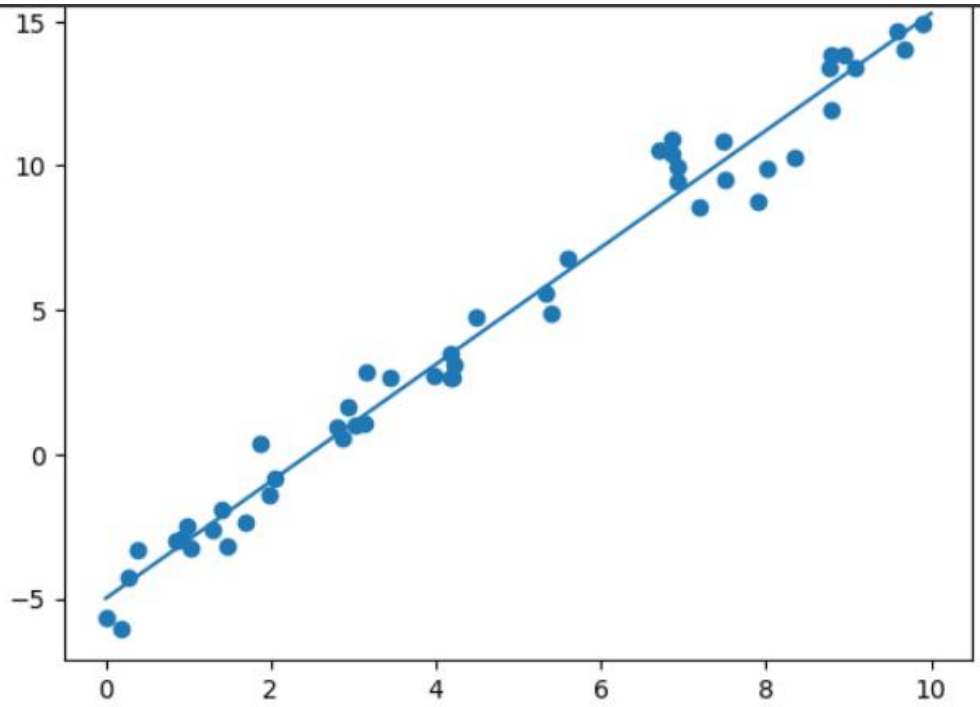
```
print(X.shape, y.shape)
print(X[:2])
print(y[:2])

model.fit(X, y)
 # Geometrically, this is akin to fitting a plane to points in three dimensions, or fitting a
 # hyper-plane to points in higher dimensions
print()
print("Model weights/coefficients:    ", model.coef_)
print("Model intercept:", model.intercept_)

# In this way, we can use the single LinearRegression estimator to fit lines, planes, or hyperplanes to our data
```




```
(50,) (50, 1)
```




```
Model slope:     2.0272088103606953
Model intercept: -4.998577085553204

(100, 3) (100,)
[[4.17022005e+00 7.20324493e+00 1.14374817e-03]
 [3.02332573e+00 1.46755891e+00 9.23385948e-01]]
```



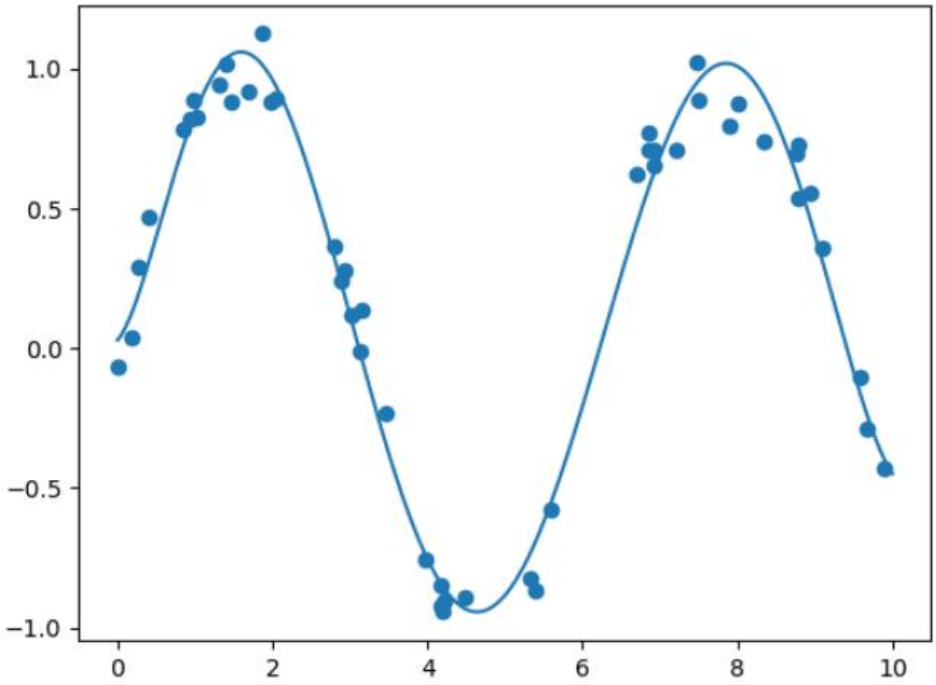

```
Model slope:     2.0272088103606953
Model intercept: -4.998577085553204

(100, 3) (100,)
[[4.17022005e+00 7.20324493e+00 1.14374817e-03]
 [3.02332573e+00 1.46755891e+00 9.23385948e-01]]
[-7.65001605  3.02325672]

Model weights/coefficients:     [ 1.5 -2.   1. ]
Model intercept: 0.5000000000000033
```

```
# Basis Function Regression

# One trick you can use to adapt linear regression to nonlinear relationships between variables is
# to transform the data according to basis functions

# With a single-dimensional input x:
# We let xn=fn(x), where fn() is some function that transforms our data
```

```
# Polynomial basis functions


# if fn(x)=x^n, our model becomes a polynomial regression:
# y = a0 + a1x + a2x^2 + a3x^3 + ...
# This is still a linear model—the linearity refers to the fact that the coefficients an never multiply/divide ea
# We have projected our one-dimensional x values into a higher dimension,
# so that a linear fit can fit more complicated relationships between x and y


from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])

# Higher-dimensional data representation
poly = PolynomialFeatures(degree = 3, include_bias=False)
print(poly.fit_transform(x.reshape(-1,1)))


poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())
# With this transform in place, we can use the linear model to fit much more complicated relationships between x
# E.g. sine wave with noise:
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)
```

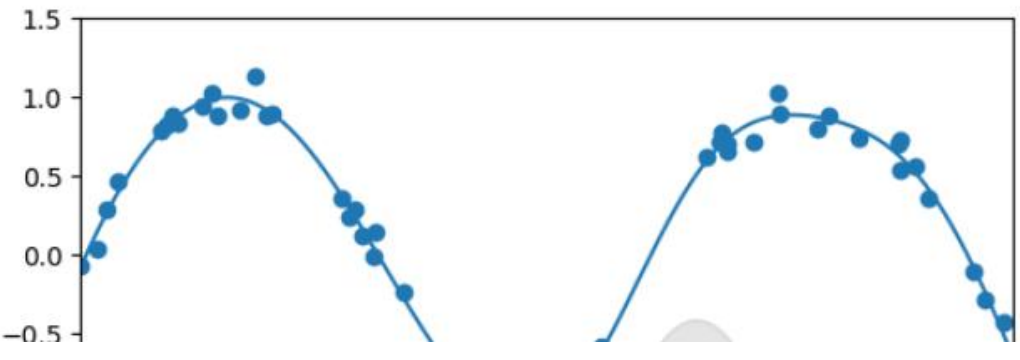
```
poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())
# With this transform in place, we can use the linear model to fit much more complicated relationships between x and y
# E.g. sine wave with noise:
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)

poly_model.fit(x.reshape(-1,1), y)

xfit = np.linspace(0, 10, 1000)
yfit = poly_model.predict(xfit.reshape(-1,1))

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.show()
```

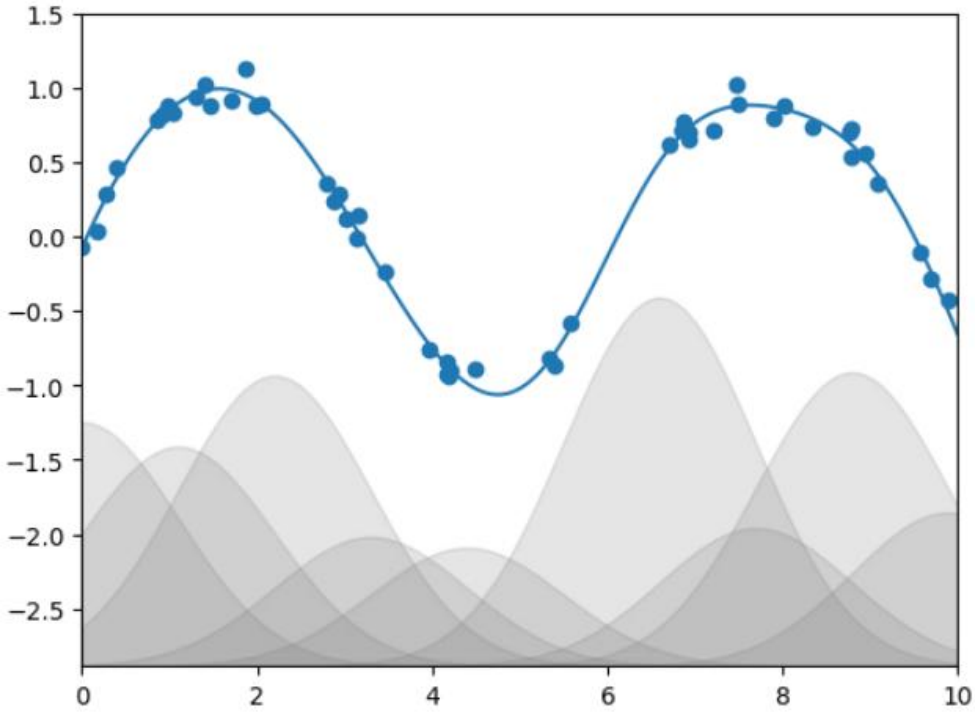
```
[[ 2.  4.  8.]
 [ 3.  9. 27.]
 [ 4. 16. 64.]]
```




```
# Gaussian basis functions

# Another basis function is for example to fit a model that is not a sum of polynomial bases,
# but a sum of Gaussian bases. The shaded regions in the plot are the scaled basis functions,
# and when added together they reproduce the smooth curve through the data

from sklearn.base import BaseEstimator, TransformerMixin

# BaseEstimator is a base class in the scikit-learn machine learning library that defines the basic
# interface for all estimators in scikit-learn. It provides a set of methods and attributes that
# any custom estimator needs to implement or inherit to work properly with the scikit-learn framework.
#  This base class enables to set and get parameters of the estimator
# implements get_params and set_params for you, These two methods allow your custom transformer to be cloned
# and hyperparameter-tuned using GridSearchCV or RandomizedSearchCV, t also contains some common
# input validation methods, and more recently, feature name handling.

# TransformerMixin: mixin class for all transformers in scikit-learn

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""
```

```
class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""


    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor


    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))


    def fit(self, X, y=None):
        # creating N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] - self.centers_[0])
        return self


    def transform(self, X):
        return __class__._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                      self.width_, axis=1)


rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)
xfit = np.linspace(0, 10, 1000)

gauss_model = make_pipeline(GaussianFeatures(N = 10, width_factor = 1.0),
                            LinearRegression())

gauss_model.fit(x.reshape(-1,1), y)
yfit = gauss_model.predict(xfit.reshape(-1,1))

gf = gauss_model.named_steps['gaussianfeatures']
lm = gauss_model.named_steps['linearregression']

fig, ax = plt.subplots()

for i in range(10):
    selector = np.zeros(10)
    selector[i] = 1
    Xfit = gf.transform(xfit.reshape(-1,1)) * selector
    yfit = lm.predict(Xfit)
    ax.fill_between(xfit, yfit.min(), yfit, color='gray', alpha=0.2)

ax.scatter(x, y)
ax.plot(xfit, gauss_model.predict(xfit.reshape(-1,1)))
ax.set_xlim(0, 10)
ax.set_ylim(yfit.min(), 1.5)
plt.show()

print(lm.coef_)
```

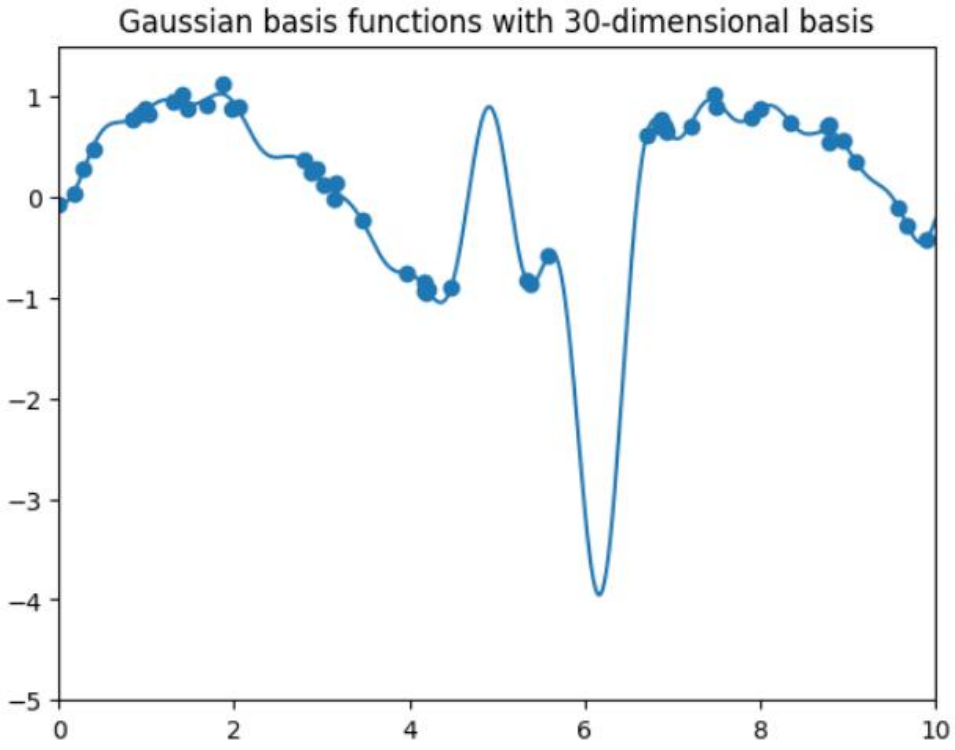


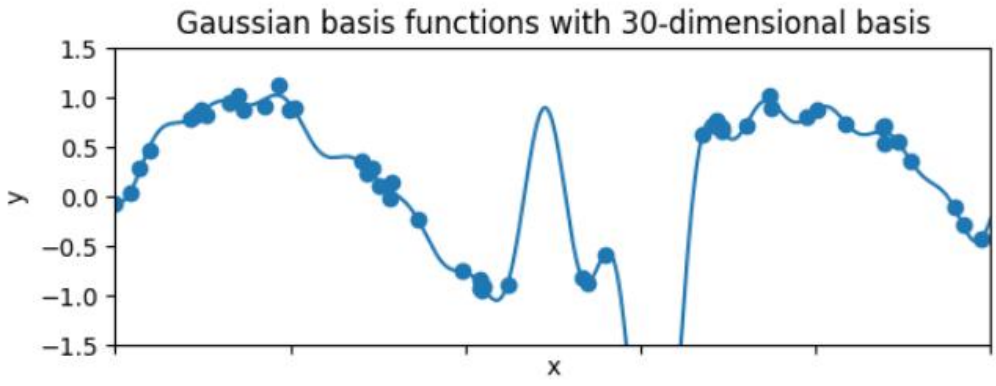


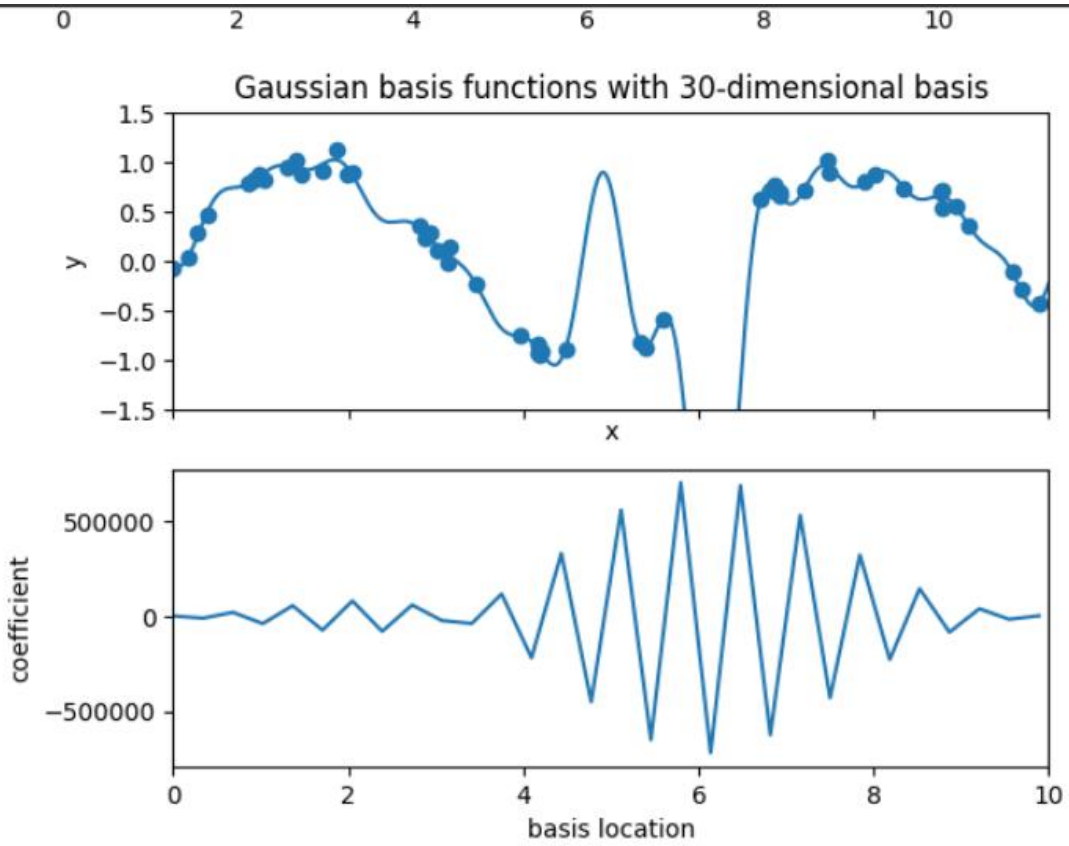

```
[ 1.64110388  1.47667984  1.9534691   0.8702509   0.79767371 -0.08567809
  2.47809194  0.92998259  1.9757953   1.03547107]
```

```
# Regularization


# Increasing complexity of models can quickly lead to overfitting,
# for example too many Gaussian basis functions:
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)
xfit = np.linspace(0, 10, 1000)

model = make_pipeline(GaussianFeatures(30),
                      LinearRegression())


model.fit(x.reshape(-1,1), y)

plt.scatter(x, y)
plt.plot(xfit, model.predict(xfit.reshape(-1,1)))

plt.xlim(0, 10)
plt.ylim(-5, 1.5)
plt.title("Gaussian basis functions with 30-dimensional basis")
plt.show()

# With the data projected to the 30-dimensional basis, the model has far too much flexibility
# We can see the reason for this if we plot the coefficients of the Gaussian bases with respect to their locations
def basis_plot(model, title=None):
    fig, ax = plt.subplots(2, sharex=True)
    model.fit(x[:, np.newaxis], y)
    ax[0].scatter(x, y)
    ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
    ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

    if title:
        ax[0].set_title(title)

    ax[1].plot(model.steps[0][1].centers_,
               model.steps[1][1].coef_)
    ax[1].set(xlabel='basis location',
              ylabel='coefficient',
              xlim=(0, 10))
```

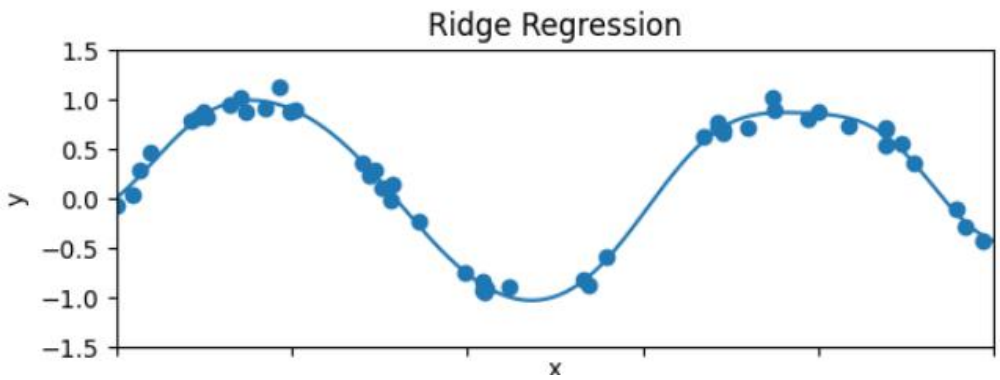
```
# We can see the reason for this if we plot the coefficients of the Gaussian bases with respect to their locations
def basis_plot(model, title=None):
    fig, ax = plt.subplots(2, sharex=True)
    model.fit(x[:, np.newaxis], y)
    ax[0].scatter(x, y)
    ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
    ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

    if title:
        ax[0].set_title(title)

    ax[1].plot(model.steps[0][1].centers_,
               model.steps[1][1].coef_)
    ax[1].set(xlabel='basis location',
              ylabel='coefficient',
              xlim=(0, 10))
    #ax[1].set_yticks(np.linspace(round(model.steps[1][1].coef_.min(), 1), round(model.steps[1][1].coef_.max(), 1), 8


    plt.show()

model = make_pipeline(GaussianFeatures(30, width_factor = 2.0), LinearRegression())
basis_plot(model, title="Gaussian basis functions with 30-dimensional basis")

# On the plot we see a typical overfitting behavior: the basis functions overlap, the coefficients of adjacent
# basis functions blow up and cancel each other out.
# It would be nice if we could limit such spikes explitiсly in the model by penalizing large values of
# the model parameters: Regularization
```



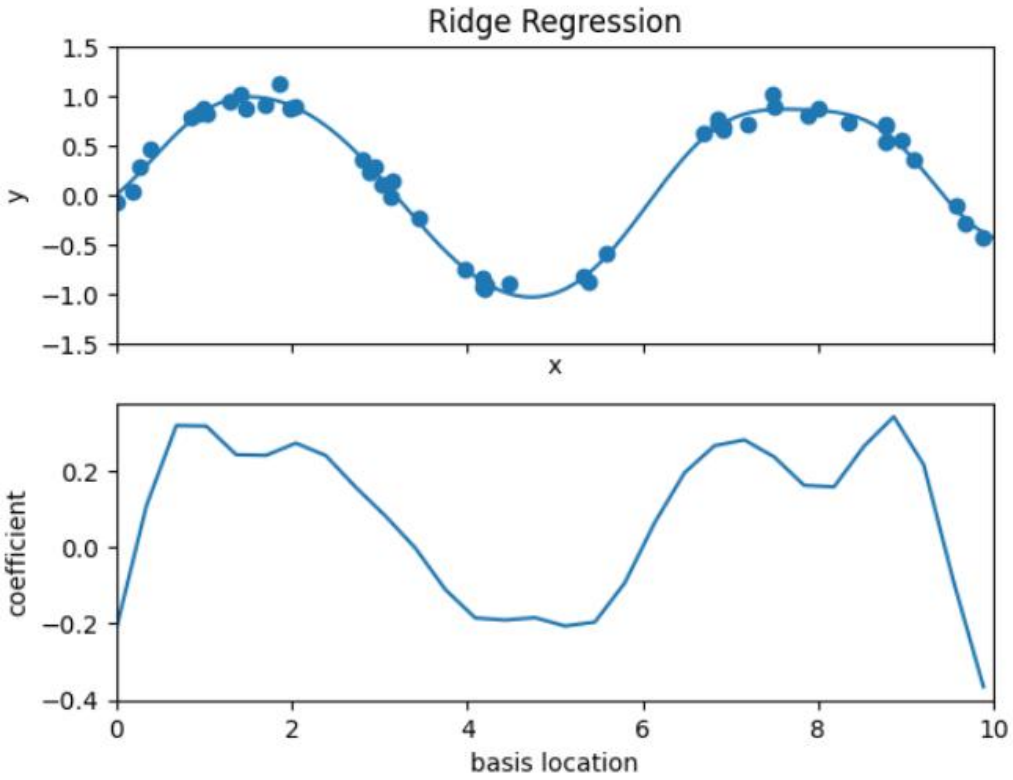




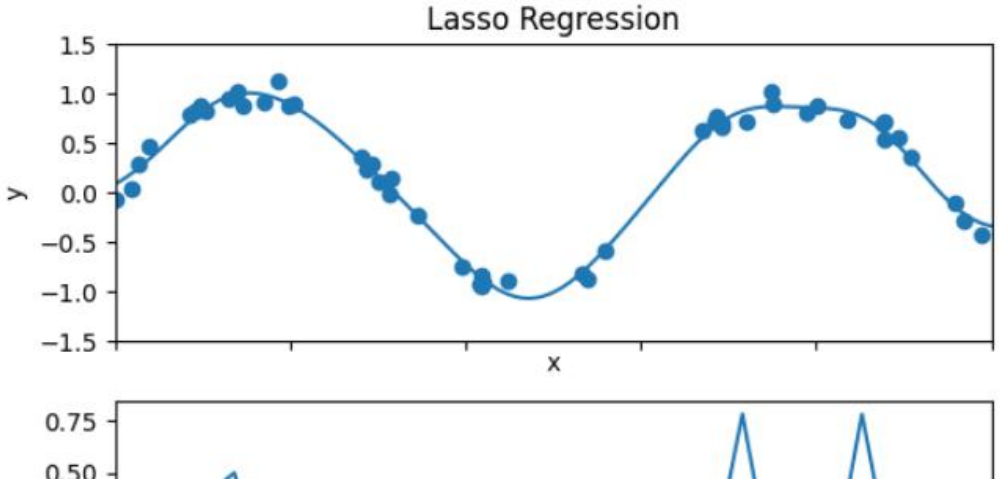

```
# Ridge regression (L2 Regularization)

# Sometimes also called Tikhonov regularization
# Penalizes the sum of squares (2-norms) of the model coefficients/weights

# Residual Sum of Squares in this case:
# ∑(i=1)^N [[(y_i - w^T * x_i)^2]] + α * ∑(j=1)^D [[wj^2]] ,
 # where α is a free parameter that controls the strength of the penalty,
 # D: number of dimensions,
 # w0 = intercept/bias and x0_i = 1
# In this formula the first sum is to fit the training data well,
# and the second is to keep weights small (so to prevent overfitting)
# Thus there's a trade-of between these 2 goals:
# as we incresae weights (increase overfitting) the prediction errors (on the training set) decrease, and vica-versa

from sklearn.linear_model import Ridge
RidgeModel = make_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))
basis_plot(RidgeModel, title='Ridge Regression')

# The α parameter is essentially a knob controlling the complexity of the resulting model
# In the limit α→0, we recover the standard linear regression result (the weights won't be penalized for any size)
# in the limit α→∞ all model responses will be suppressed (the penalty is so strong that weights→0)

# One advantage of Ridge regression is that it can be computed very efficiently,
# at hardly more computational cost than the original linear regression model
```



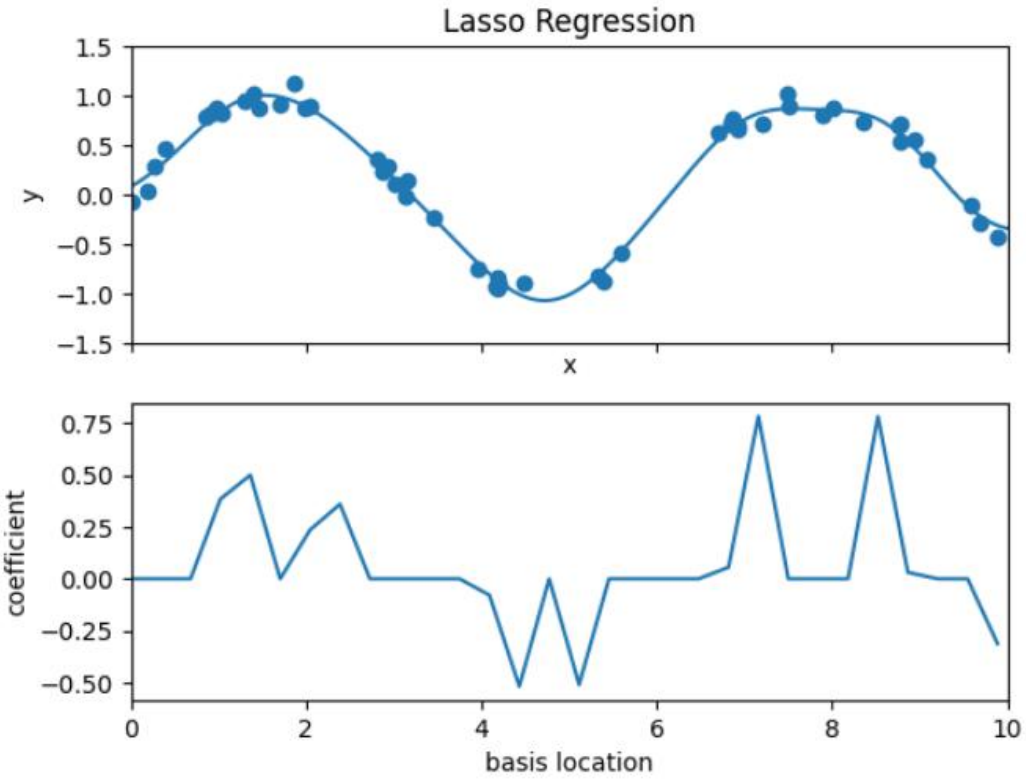
```
# One advantage of Ridge regression is that it can be computed very efficiently,
# at hardly more computational cost than the original linear regression model
```




```
# Lasso regression (L1 regularization, Least absolute shrinkage and selection operator)

# Penalizes the sum of absolute values (1-norms) of the model coefficients/weights

# Residual Sum of Squares in this case:
# ∑(i=1)^N [[(y_i - w^T * x_i)^2]] + α * ∑(j=1)^D [[|wj|]] ,
 # where α is a free parameter that controls the strength of the penalty,
 # D: number of dimensions,
 # w0 = intercept/bias and x0_i = 1
# In this formula the first sum is to fit the training data well,
# and the second is to keep weights small (so to prevent overfitting)
# Thus there's a trade-of between these 2 goals:
# as we incresae weights (increase overfitting) the prediction errors (on the training set) decrease, and vica-versa

from sklearn.linear_model import Lasso
LassoModel = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.002))
basis_plot(LassoModel, title='Lasso Regression')

# One advantage of Lasso regression is that it is able to perform 'embedded feature selection'
# by setting coefficients to exactly zero (Ridge can only set them close to zero)
```



```
# by setting coefficients to exactly zero (Ridge can only set them close to zero)
```

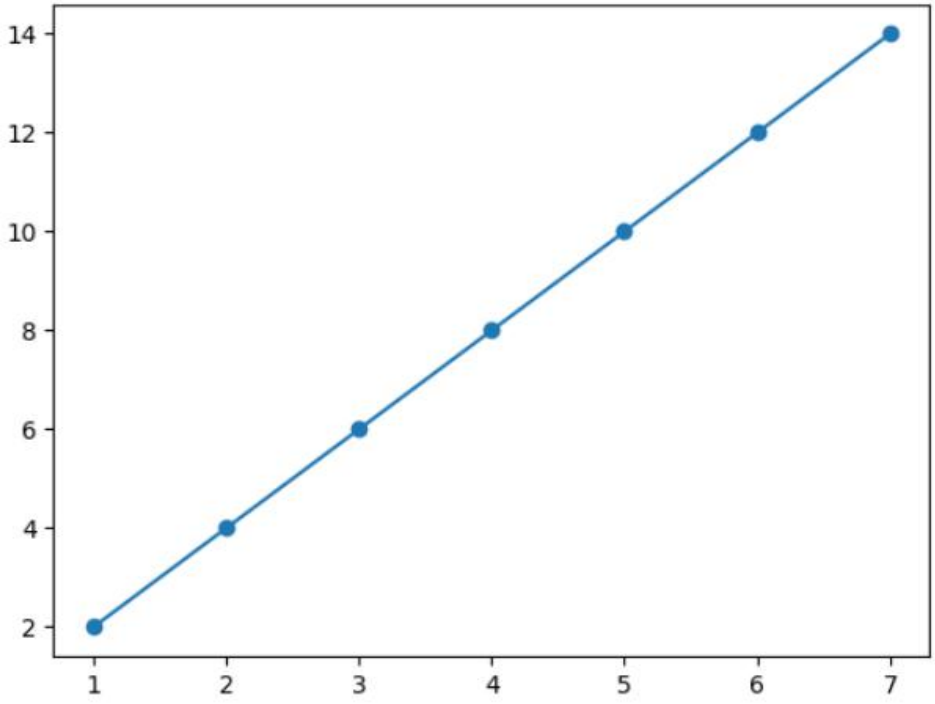


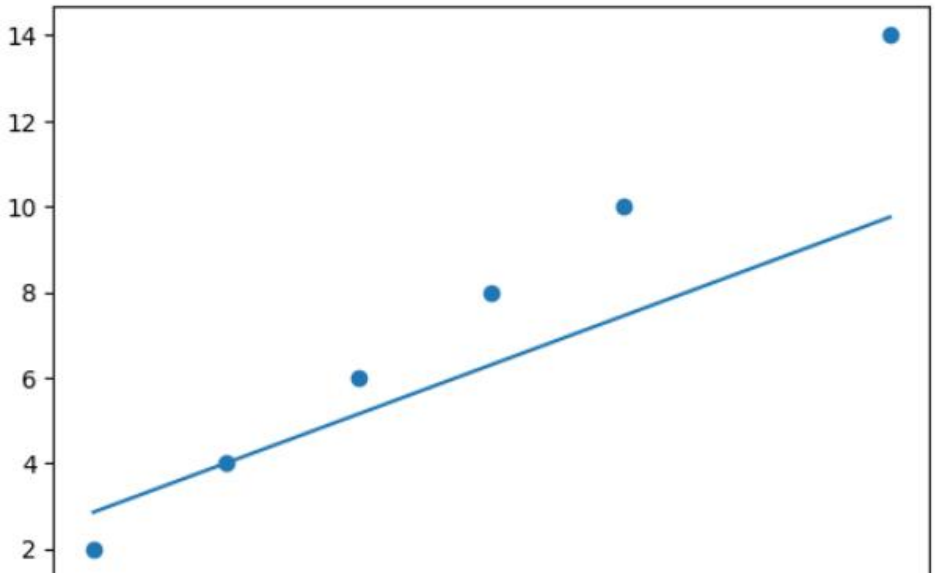

```
# Lasso vs. Ridge Regression

# Unlike Ridge, due to geometric reasons (diamond vs circle problem) Lasso is able to perform
# 'embedded feature selection' by setting coefficients to exactly zero, while Ridge can only
# shrink them close to zero, but never equal to zero. So Lasso tends to favor sparse models where possible.

# On the plot above we can see that the majority of the coefficients are exactly zero,
# so only a small subset of the basis functions are 'available'/'active'
print(LassoModel.steps[1][1].coef_)
print()
print(RidgeModel.steps[1][1].coef_)

# Problem with the 'automated feature selection' of Lasso:
# When we have correlated features/variables, it usually retains only one of feature/variable
# which is found to be the most important, and sets other correlated variables to zero,
# so sometimes it cancels too many features and thus increases bias too much (error by simplifying assumptions of the

# One 'solution' to this problem:
# Elastic Net Regression, which combines both Ridge and Lasso Regression techniques
# by learning from their shortcomings to improve regularization
# So the penalty term is a combination of the l1-norm (absolute value) and the l2-norm (square) of the coefficients,
 # scaled by a parameter called alpha.
# Like lasso, Elastic Net can also generate reduced models by zero-valued coefficients,
# but it can outperform Lasso on data with highly correlated predictors/variables/features
# Can be be found here: sklearn.linear_model.ElasticNet
```

```
[-0.         -0.          0.          0.38496444  0.500335    0.
  0.23609036  0.35994884  0.          0.         -0.         -0.
 -0.07887826 -0.5184722  -0.         -0.51073554 -0.         -0.
  0.          0.          0.05560327  0.78410977  0.          0.
  0.          0.78318771  0.0308607   0.         -0.         -0.31307227]

[-0.21105794  0.10862732  0.31824548  0.31676161  0.2418113   0.24003021
  0.27210133  0.23947825  0.15675484  0.08094844 -0.00307864 -0.11134067
 -0.18560261 -0.19090305 -0.18482812 -0.20657632 -0.19638464 -0.09387141
  0.06369053  0.1945448   0.26527119  0.28056559  0.23593141  0.16159984
  0.15753808  0.26372747  0.34134295  0.21541902 -0.09008851 -0.36548691]
```

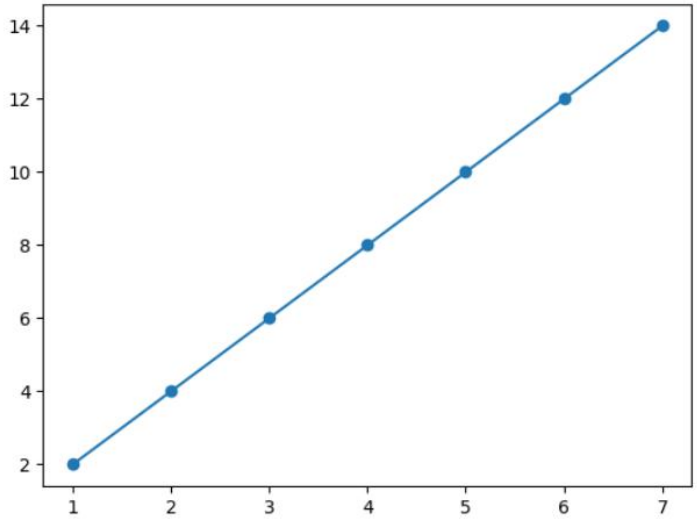
```
# A bit off-topic:
```

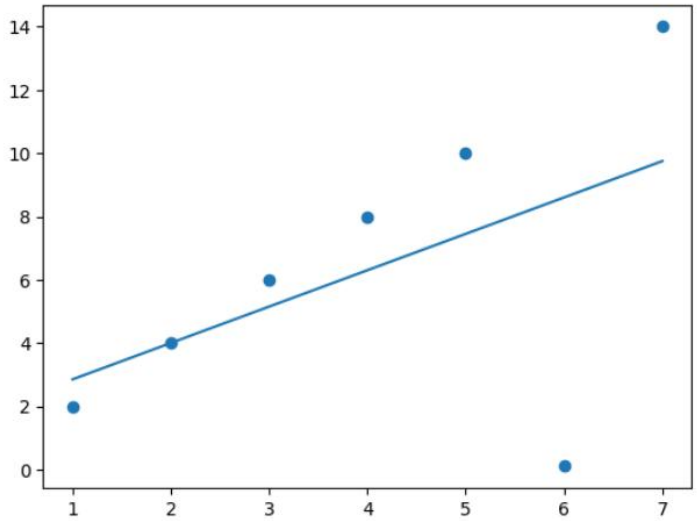
```
# A bit off-topic:
 #Cell to show how prone to outliers Linear Regression is

x = np.array([1, 2, 3, 4, 5, 6, 7])
y = np.array([2, 4, 6, 8, 10, 12, 14])

model = LinearRegression().fit(x.reshape(-1,1), y)
yfit = model.predict(x.reshape(-1,1))

plt.scatter(x, y)
plt.plot(x, yfit)
plt.show()

x = np.array([1, 2, 3, 4, 5, 6, 7])
y = np.array([2, 4, 6, 8, 10, 0.1, 14])

model = LinearRegression().fit(x.reshape(-1,1), y)
yfit = model.predict(x.reshape(-1,1))

plt.scatter(x, y)
plt.plot(x, yfit)
plt.show()
```





```
plt.scatter(x, y)
plt.plot(x, yfit)
plt.show()
```





Logistic Regression

Mathematics

Introduction

Logistic Regression is a widely-used method in Machine Learning for binary classification, where the goal is to categorize data into one of two classes. It falls under the umbrella of Supervised Learning, requiring labeled training data.

The Logistic Model

Logistic Regression

Mathematics

Introduction

Logistic Regression is a widely-used method in Machine Learning for binary classification, where the goal is to categorize data into one of two classes. It falls under the umbrella of Supervised Learning, requiring labeled training data.

The Logistic Model

1. Linear equation: Calculating the distance

$$f(x) = \sum_{j=1}^{D} w_j x_j$$

Explanation:

- Calculates a signed distance between the input and the linear model (a linear line as the decision boundary between the 2 classes)
- Returns positive value if input x belongs to the class on the right side of the linear decision boundary, and negative if input belongs to the class on the left side of the linear decision boundary
- The larger the distance of the input x to the decision boundary, the more certain it belongs to either class 1 or class 2

2. Sigmoid Function

The key component in Logistic Regression is the Sigmoid function, denoted as $\sigma(x)$, mapping any real number to a value between 0 and 1. It is expressed as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Explanation:

- The Sigmoid function maps any real number x to a value between 0 and 1.
- It is crucial in logistic regression as it transforms the output of the linear equation into probabilities.

Why Sigmoid?

The Sigmoid function is chosen for logistic regression due to its properties:

- Maps to values between 0 and 1, ideal for binary classification probabilities.
- Smooth and differentiable, facilitating optimization using methods like gradient descent.
- Simple mathematical form for efficient computation

Why Sigmoid?

The Sigmoid function is chosen for logistic regression due to its properties:

- Maps to values between 0 and 1, ideal for binary classification probabilities.
- Smooth and differentiable, facilitating optimization using methods like gradient descent.
- Simple mathematical form for efficient computation.
- Symmetric around the origin, ensuring model stability and interpretability.

Alternative Functions

For logistic regression, other functions can replace the Sigmoid:

- **Softmax Function:** Generalization for multi-class classification.
- **Probit Function:** Based on the cumulative distribution function of the standard normal distribution.
- **Hyperbolic Tangent Function (tanh):** Maps to values between -1 and 1.

3. From distance to probability: Inserting the function f(x) into Sigmoid

$$\mu(x, w) = \frac{1}{1 + e^{-w^T x}}$$

Explanation:

- **This is the model now**, calculates probabilities instead of distances
- Returns the probability of input $\mathbf{x}$ belonging **only** to the **positive class**.

Modelling both events: Bernoulli distribution

$$p(y|x, w) = Ber(y|x, w) = \mu(x, w)^y (1 - \mu(x, w))^{1-y}$$

Explanation:

- Returns the probability of input $\mathbf{x}$ belonging to **either** the **positive class** or **negative class** (based on y).

Optimization

Likelihood function

The objective function: Maximum Likelihood Estimation
Logistic Regression employs Maximum Likelihood Estimation (MLE) to find the optimal weights. MLE is a statistical method used to estimate the parameters of a model that maximize the likelihood of observing our data given a certain parametrization (w)/of the observed data under the given model.

$$\underline{N}$$

Optimization

Likelihood function

The objective function: Maximum Likelihood Estimation
Logistic Regression employs Maximum Likelihood Estimation (MLE) to find the optimal weights. MLE is a statistical method used to estimate the parameters of a model that maximize the likelihood of observing our data given a certain parametrization (w)/of the observed data under the given model.

$$L(w|X,y) = \prod_{i=1}^{N} p(y_i = 1|x_i, w)^{y_i}(1 - p(y_i = 1|x_i, w))^{1-y_i}$$

$$With\ Bernoulli$$

$$L(\theta) = \prod_{i=1}^{N} \mu(x_i, w)^{y_i}(1 - \mu(x_i, w))^{1-y_i}$$

Explanation:

- $L(w|X,y)$ is the likelihood function representing the joint probability of observed labels y given input data X and model parameters w.
- It is used in Maximum Likelihood Estimation (MLE) to find the set of parameters that maximize the likelihood of observing our data given a certain parametrization(w) /of the observed data under the given model.

Logarithmic Likelihood/Log-Likelihood

$$\log L(w|X,y) = \sum_{i=1}^{N} y_i \log(p(y_i = 1|x_i, w)) + (1 - y_i)\log(1 - p(y_i = 1|x_i, w))$$

$$With\ Bernoulli$$

$$\ell(\theta) = \sum_{i=1}^{N} \log[p(y_i|\mathbf{x}_i, \theta]$$

$$= \sum_{i=1}^{N} \log(\mu(x_i, w)^{y_i}(1 - \mu(x_i, w)^{1-y_i})$$

$$= \sum_{i=1}^{N} y_i \log(\mu(x_i, w)) + (1 - y_i)\log(1 - \mu(x_i, w))$$

Why to take the logarithm of the Likelihood function?

Some of the main reasons:

- The product of small numbers in a computer is unstable, for example when $w^T x \to -\infty$ => $e^{-w^T x} \to +\infty$ => $\frac{1}{1+e^{-w^T x}} \to 0$, which means

Why to take the logarithm of the Likelihood function?

Some of the main reasons:

- The product of small numbers in a computer is unstable, for example when $w^T x \to -\infty$ => $e^{-w^T x} \to +\infty$ => $\frac{1}{1+e^{-w^T x}} \to 0$, which means $p(y=1|x,w)^y \to 0$ and then this factor (factor for this (x_i, y_i)) in the product might turn to exactly 0 (instead of very close to zero), which results in the whole product turning to 0 (compare `print(1e-323)` vs. `print(1e-324)`)
- Often times the logarithm of a non-concave Likelihood function is concave (concave down, and monotonically increasing in x), which is a change from multiple local maximums to one unique (global) maximum (after this it's much easier to optimize i.e. find the global optimum)
- Easier to take the derivative of the log of the function than the function itself as the log of a product is the sum of the logs of its factors

Loss function (Binary Cross Entropy)

Instead of Maximizing the Log-Likelihood, we minimize the Negative Log-Likelihood: Binary Cross Entropy Loss

$$J(w) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p(x_i, w)) + (1 - y_i)\log(1 - p(x_i, w))$$

With Bernoulli

$$J(\theta) = -\sum_{i=1}^{N} y_i \log(\mu(x_i, w)) + (1 - y_i)\log(1 - \mu(x_i, w))$$

Explanation:

- $J(w) = J(\theta)$ is the objective function (Loss function) to be minimized during training
- N is the number of training samples, y_i is the actual label (0 or 1), $p(x_i, w)$ is the predicted probability of x_i belonging to the positive class, and w is the vector of weights
- Measures the difference between values for predicted probability and true values (true labels): the logarithmic terms penalize a lot more when the predicted probability is far away from the true value compared to when it's closer (e.g. compare the Loss value difference ($y_i = 0$ or 1) between $p(x_i, w) = \mu(x_i, w) = 0.8$ and $p(x_i, w) = \mu(x_i, w) = 0.9$ with the difference between $p(x_i, w) = \mu(x_i, w) = 0.1$ and $p(x_i, w) = \mu(x_i, w) = 0.2$)
- The model's objective is to find optimal weights w that minimize $J(w) = J(\theta)$.
- It has a unique minimum, thus can be optimized with for example Gradient Descent

Training Process

Repeat for each training iteration:

1. Get predictions with current weights

   So that we are able to compute the next step
   (the derivative formula needs predictions as input)

   $$\hat{y_i} = p(x_i, w)$$

Training Process

Repeat for each training iteration:

1. Get predictions with current weights

   So that we are able to compute the next step
   (the derivative formula needs predictions as input)

$$\hat{y_i} = p(x_i, w)$$

2. Compute gradient (derivative) of the Loss Function

   So that in the next step we are able to update the weights
   We use Gradient Descent as the optimization method to minimize the Loss Function
   The gradient of the Loss is computed as:

$$\frac{\partial J(w)}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y_i} - y_i) x_{ij}$$

   Explanation:

   - Calculates the gradient (derivative) of the Loss Function with respect to the weights.
   - Visually the Loss looks like a mountainous landscape, we find the minimum by "walking down" the slope of the mountain
   - The derivative (gradient) represents the direction and magnitude of the steepest ascent/descent (depends on initial weights) of the corresponding function (now the Loss Function).

3. Update weights

   So that with the updated parameters predictions produce a smaller Loss

$$w_j = w_j - \alpha \frac{\partial J(w)}{\partial w_j}$$

   Explanation:

   - Updates the weights in the direction of the minimum of the Loss Function/in the direction that reduces the Loss
   - The learning rate α controls (scales) the size of the update

Regularization

Large values of the weights (parameters) make the model follow small changes in the input too closely in the ouput (overfitting)
We can control overfitting explitiely in the model by penalizing these large weights: Regularization

L1 Regularization

Regularization

Large values of the weights (parameters) make the model follow small changes in the input too closely in the ouput (overfitting)
We can control overfitting explitiely in the model by penalizing these large weights: Regularization

L1 Regularization

Penalizes the sum of absolute values (1-norms) of the model weights.
Modifies the Loss Function (more precisely the Likelihood Function, from which the Loss is derived) by adding the penalty term $\lambda \sum_{j=1}^{n} |w_j|$

Loss Function with L1 Regularization:

$$J(w) = -\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log(p(x_i, w)) + (1 - y_i) \log(1 - p(x_i, w)) \right] + \lambda \sum_{j=1}^{n} |w_j|$$

Explanation:

- λ is a free parameter that controls the strength of the regularization (of the penalty)
- The first $\sum$ is to fit the training data well, and the second is to keep weights small (to prevent overfitting)
- So there's a trade-of between these 2 goals: as we incresae weights (i.e. increase overfitting, so the right $\sum$ increase), the prediction errors on the training set decrease (so the negative of the left $\sum$ decrease), and vica-versa
- In the limit $\lambda \rightarrow$ we recover the standard Logistic Regression without penalty (weights won't be penalized for any size)
- In the limit $\lambda \rightarrow \infty$ all model responses will be suppressed (the penalty is so strong that $w \rightarrow 0$)

L2 Regularization

Penalizes the sum of squares (2-norms) of the model weights. Modifies the Loss Function (more precisely the Likelihood Function, from which the Loss is derived) by adding the penalty term $\lambda \sum_{j=1}^{n} w_j^2$

Loss Function with L2 Regularization:

$$J(w) = -\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log(p(x_i, w)) + (1 - y_i) \log(1 - p(x_i, w)) \right] + \lambda \sum_{j=1}^{n} w_j^2$$

Comapring L1 vs. L2 techniques for regularization

Due to geometric reasons (diamond vs circle problem), L1 can perform 'embedded feature selection' by being able to drive some weights to exactly zero, while L2 can only shrink them close to zero, but never equal to zero.

- Thus L1 penalty term encourages sparsity in the model and is often used when feature selection is desirable

Problem with the 'automated feature selection' of L1:

Problem with the 'automated feature selection' of L1:
When we have correlated features/variables, it usually retains only one of feature/variable which is found to be the most important, and sets other correlated variables to zero

- Thus sometimes L1 cancels too many features and therefore increases bias too much (error induced by simplifying assumptions of the model)

One suggestion to this problem:

- L2 technique discourages the model from relying too much on any particular feature

It can not perform feature selection because of not being able to set weights to exactly zero (only to shrink close to zero), due to the same geometric reasons

Another suggestion:

- Elastic Net Regression, which combines both L1 and L2 techniques by learning from their shortcomings to improve regularization

For Elastic Net the penalty term is a combination of the l1-norm (absolute value) and the l2-norm (square) of the coefficients scaled by a parameter.
Like L1, it can also generate reduced models by zero-valued weights, but empirical studies have suggested that it can outperform L1 on data with highly correlated features.

Creating model from scratch using NumPy (code structure in line with Mathematics) [...]

Applying created model, analyzing the training process and comparing performance to sklearn's model [...]

Evaluation metrics

Creating and applying class to a classification task

Creating model from scratch using NumPy (code structure in line with Mathematics)

```
# Get sklearn model's default parameter values
regressorSklearn = sklearn.linear_model.LogisticRegression()
DEFAULTSKPARAMS = regressorSklearn.__dict__
```

```
class LogRegNumPy:

    def __init__(self,
                 learning_rate = 0.1,
                 penalty = DEFAULTSKPARAMS['penalty'], #='l2', the technique to use for regularization
                 lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0, the λ (lambda) a free parameter that controls the
                                                            #strength of the regularization (of the penalty)
                                        # the 'C' parameter in sklearn's LogReg class is the inverse of this parameter)
                 max_iter = DEFAULTSKPARAMS['max_iter'], #=100, number of iterations for the training process (for Gradient Descent)
                 tol = DEFAULTSKPARAMS['tol'], #=0.0001=1e-4, when the Loss Function decreases so slightly that
                                    # the difference between the Loss in the current iteration and in the previous
                                    # iteration is below this tolerance boundary, then we can say the Loss Fnction
                                    # has converged into its minimum and there's no need to take more gradient steps
                 patience=20, # Stop training iterations after this number (patience) of such iterations in which the
                              # Loss was not lower than the lowest Loss that Gradient Descent achieved so far in this training
                 treshold=0.5, # Cut-off value for the predicted probabilities (of x_i belonging to the positive class)
                               # based on which value x_i classified into the positive or negative class
                 initialweights_lowerlimit = 0, # Initialize weights for the first training iteration from a uniformly
                 initialweights_upperlimit = 1): # distributed interval [initialweights_lowerlimit, initialweights_upperlimit)
                                    # Useful for fixed results (not different results when cell rerun) for comparison,
                                    # or to check performance of the model without training


        self.learning_rate = learning_rate
        self.penalty = penalty
        self.lambda_or_inverseC = lambda_or_inverseC
        self.max_iter = max_iter
        self.tol = tol
        self.patience = patience
        self.treshold = treshold
        self.initialweights_lowerlimit = initialweights_lowerlimit
        self.initialweights_upperlimit = initialweights_upperlimit
        self.loss_history = [] # Collector for the Loss value in each iteration
        self.weights_history = [] # Collector for the weight(s) in each iteration
        self.weights = None # Initializing values to weights later during fit
        self.X_train_means = [] # For saving during fit() the mean and standard deviation of the train set
        self.X_train_stds = [] # with respect to each feature in order to be able to standardize the test set  with them during predict()

    # ---------------------- Standardize the input features  ----------------------
    @staticmethod
    def _standardizeX(X, trainX_means, trainX_stds):

        for feature in range(X.shape[1]):
            X[:, feature] = ((X[:, feature] - trainX_means[feature])/trainX_stds[feature])

        return X


    # ---------------------- The Logistic Model ----------------------
    # 1. Calculate distance
    @staticmethod
    def _calculate_distance(weights, X):
        return np.dot(weights, X.T) # transpose of X instead of weights as in the Mathematics

    @staticmethod
    # 2. Sigmoid Function
    def _sigmoid_function(z):
        return 1 / (1 + np.exp(-z))


    @staticmethod
```

```
    @staticmethod
    # 3. From distance to probability
    def _predict_probability(wTx_distance):
        return __class__._sigmoid_function(wTx_distance)


    # Computing Loss is not part of the core training process (3 steps) and is not required for training,
    # here it is computed only to report (print) and visualize Loss values during training
    @staticmethod
    def _compute_loss(num_samples, y_pred, y_true, weights, regularization_technique, regularization_strength):

        # Binary Cross Entropy as the Loss Function
        cross_entropy = - (1 / num_samples) * np.sum(y_true*np.log(y_pred) + (1-y_true)*np.log(1-y_pred))
        # Same results:
        # epsilon = 1e-5
        # cross_entropy = (-y_true * np.log(y_pred + epsilon) - (1 - y_true) * np.log(1 - y_pred + epsilon)).mean()


        # The penalty term depends on the Regularization technique
        if regularization_technique == 'l1':
            penalty_term = regularization_strength * np.sum(np.abs(weights))

        elif regularization_technique == 'l2':
            penalty_term = regularization_strength * np.sum(np.square(weights))

        return cross_entropy + penalty_term


    @staticmethod
    def _visualize_graddesc(loss_values, weight_values, lowest_loss, number_of_dimensions):

        if number_of_dimensions == 1:
            fig, ax = plt.subplots(nrows=1, ncols=2, sharey=True)
            fig.set_figheight(4)
            fig.set_figwidth(15)
            ax[0].grid()
            ax[0].set(xlabel='Epochs (Number of iterations)', ylabel='Loss/Cost function')
            ax[0].plot(range(1, len(loss_values)+1), loss_values, 'm', linewidth = "3")
            ax[0].axhline(lowest_loss, color = 'green', linestyle = '-', linewidth = "1.5")
            ax[0].set_xticks = np.linspace(0, len(loss_values), 11)

            ax[1].grid()
            ax[1].set(xlabel='Weight')
            ax[1].axhline(lowest_loss, color = 'green', linestyle = '-', linewidth = "1.5")
            ax[1].set_xticks = np.linspace(weight_values.min(), weight_values.max(), 15)

            ax[1].scatter(weight_values, loss_values)
            for iterator in range(len(weight_values[:10])-1):
                ax[1].arrow(weight_values[iterator],
                            loss_values[iterator],
                            weight_values[iterator+1]-weight_values[iterator],
                            loss_values[iterator+1]-loss_values[iterator],
                            color = 'm', shape='full', width = 0.0003, head_width = 0.003, length_includes_head=True)

#           ax[1].plot(range(1,len(weight_values)+1), weight_values, 'm', linewidth = "3")
#           loss_values = np.array(loss_values)
#           weight_values = np.array(weight_values)
#           SortedIndexes = np.argsort(weight_values)
#           ax[1].plot(weight_values[SortedIndexes], loss_values[SortedIndexes], 'm', linewidth = "3")

            plt.show()

        else:
            plt.grid()
            plt.xlabel('Epochs (Number of iterations)')
            plt.ylabel('Loss/Cost function')
            plt.plot(range(1, len(loss_values)+1), loss_values, 'm', linewidth = "3")
            plt.axhline(lowest_loss, color = 'green', linestyle = '-', linewidth = "2")
            plt.xticks(np.linspace(0, len(loss_values), 11))
            plt.show()
```

```
        else:
            plt.grid()
            plt.xlabel('Epochs (Number of iterations)')
            plt.ylabel('Loss/Cost function')
            plt.plot(range(1, len(loss_values)+1), loss_values, 'm', linewidth = "3")
            plt.axhline(lowest_loss, color = 'green', linestyle = '-', linewidth = "2")
            plt.xticks(np.linspace(0, len(loss_values), 11))
            plt.show()

    # ---------------------- Functions for the training process ----------------------
    @staticmethod
    def _compute_gradient(X, y_pred, y_true, weights, regularization_technique, regularization_strength):

        num_samples = X.shape[0]

        # Derivative of the Cross Entropy Loss Function
        loss_derivative = 1 / num_samples * np.dot(X.T, y_pred - y_true)

         # Derivative of the penalty term
        if regularization_technique == 'l1':
            penalty_derivative = regularization_strength * np.sign(weights)
            # np.sign: returns -1 if x < 0, 0 if x==0, 1 if x > 0 as the derivative of |x| = x/|x| = sign(x)

        elif regularization_technique == 'l2':
            penalty_derivative = 2 * regularization_strength * weights

        return loss_derivative + penalty_derivative

    @staticmethod
    def _update_weights(gradient, weights, regularization_technique, learning_rate):

        if regularization_technique== 'l1':
            updated_weights = weights - learning_rate * gradient

        elif regularization_technique == 'l2':
            updated_weights = weights - learning_rate * gradient

        return updated_weights



    # ---------------------- Main functions of the class: fit and predict ----------------------
    def fit(self, X, y, training_report = False):

        num_samples, num_features = X.shape

        for feature in range(num_features):
            self.X_train_means.append(np.mean(X[:, feature]))
            self.X_train_stds.append(np.std(X[:, feature]))
        X = __class__._standardizeX(X, self.X_train_means, self.X_train_stds)
        # using np.mean(X, axis=0) and np.std(X, axis=0) to get the mean and the std for each feature
        # computes less accurate results, for details please see the last cell about comparing
        # the two methods in this heading: Linear Regression - Creating from scratch

        self.weights =  np.random.uniform(self.initialweights_lowerlimit, self.initialweights_upperlimit, num_features)

        weights_of_lowest_loss = None
        lowest_loss = float('inf')
        loss_not_lower_than_lowest_counter = 0
        previous_loss = None
        number_of_iteration = 0


        # -------------------------------------  Training process  -------------------------------------
        for number_of_iteration in range(1, self.max_iter+1):

            #  1. Get predictions with current weights
            distances = __class__._calculate_distance(self.weights, X)
            y_pred = __class__._predict_probability(distances)
```

```
for number_of_iteration in range(1, self.max_iter+1):

    #  1. Get predictions with current weights
    distances = __class__._calculate_distance(self.weights, X)
    y_pred = __class__._predict_probability(distances)



# -------------------  Functionalities outside of the core training (3 steps) process  -------------------

    current_loss = __class__._compute_loss(num_samples, y_pred, y, self.weights,
                                           self.penalty, self.lambda_or_inverseC)
    gradient_for_printing = __class__._compute_gradient(X, y_pred, y, self.weights,
                                                        self.penalty, self.lambda_or_inverseC)


    if number_of_iteration == 1:
        lowest_loss = current_loss
        weights_of_lowest_loss = self.weights.copy()
        iteration_of_lowest_loss = 1
        loss_not_lower_than_lowest_counter = 0

        from decimal import Decimal
        decimalplaces_for_rounding = -Decimal(str(self.tol)).as_tuple().exponent + 2
        # for example when tol = 1e-13, then decimalplaces_for_rounding = -(-13) + 2 = 15
        # this is for when investigating the printed out Losses:
           # to be able to compare current vs. previous Loss using the computed difference

        if training_report == True:
            print(f"Training initialized with the following hyperparameter settings:\nLearning rate: {self.learning_rate}, "
              f"inverseC: {self.lambda_or_inverseC}, Maximum iterations: {self.max_iter}, "
              f"Tolerance for convergence: {self.tol}, Patience for early stop: {self.patience},\n"
              f"Initial weight(s) randomly from the interval: [{self.initialweights_lowerlimit}, "
              f"{self.initialweights_upperlimit}), Initial Loss: {current_loss.round(decimalplaces_for_rounding)}"
              f" with the {self.weights} initial weight(s)\n")


            print(f"It: {number_of_iteration}. Loss: {current_loss.round(9)}, Weight(s): {self.weights}, "
                  f"Gradient: {gradient_for_printing.round(7)}")
        else:
            print(f"Training initialized with {self.max_iter} maximum possible iterations")


    if number_of_iteration > 1:
        # Compare current iteration's Loss to the previous iteration's
        loss_difference_from_prevloss = (previous_loss - current_loss)
        difference_in_science_format = "{:.1e}".format(np.abs(loss_difference_from_prevloss))

        if current_loss < previous_loss:
            loss_status = "lower"
        elif current_loss > previous_loss:
            loss_status = "higher"
            # when the Loss has not decreased compared to the previous iteration but increased
            # then probably the learning rate is too high and the Loss function is increasing
        else:
            loss_status = "equal"

        if training_report == True:
            # Print information of current iteration
            if number_of_iteration <= 5 or number_of_iteration % np.round(self.max_iter*0.1, -1) == 0:

                print(f"It: {number_of_iteration}. Loss: {current_loss.round(9)} {loss_status} "
                f"by {difference_in_science_format} than in previous iteration, Weight(s): {self.weights}, "
                f"Gradient: {gradient_for_printing.round(7)}")

                if number_of_iteration == 5:
                    print()

        # Compare current iteration's Loss to the lowest Loss achieved so far
        if current_loss < lowest_loss:
```

```
            # Compare current iteration's Loss to the lowest Loss achieved so far
            if current_loss < lowest_loss:
                lowest_loss = current_loss
                weights_of_lowest_loss = self.weights.copy()
                iteration_of_lowest_loss = number_of_iteration
                loss_not_lower_than_lowest_counter = 0
            else:
                loss_not_lower_than_lowest_counter += 1


            # Check for convergence (current iteration's Loss vs previous iteration's)
            # when the Loss Function decreases (or when too high lr increases) so slightly that
            # the difference between the Loss in current iteration and in the previous iteration is below this
            # tolerance treshold then we can say the Loss Fnction has converged into its minimum,
            # (potentially sometimes it can be a local minimum instead of the global optimum)
            # and there's no need to take more gradient iterations
            # Note: the loss_difference_from_prevloss difference is lower than the tolerance but still positive
            # as the current iteration's Loss is smaller than the previous iteration's, except for when the
            # lr is too high and the Loss function is increasing - that's why taking the absolute of the difference
            if np.abs(loss_difference_from_prevloss) < self.tol:
                tol_in_science_format = "{:.1e}".format(self.tol)

                if training_report == True:
                    print(f"\nConverged at iteration {number_of_iteration}., "
                          f"where the Loss ({current_loss.round(decimalplaces_for_rounding)}) was {loss_status} "
                          f"by {difference_in_science_format} than\nin the prev it"
                          f"({previous_loss.round(decimalplaces_for_rounding)}), which is below the tolerance "
                          f"for convergence: {tol_in_science_format}")

                else:
                    print(f"\nConverged at iteration {number_of_iteration}, where the drop in Loss was below "
                          f"the tolerance set for convergence")

                break


            # Check for early stopping (current iteration's Loss vs the lowest Loss achieved so far)
            # Stop taking more training iterations when the Loss has not been getting
            # lower than the lowest achieved Loss for a given number (patience) of iterations
            # (learning rate is too high, thus the Loss function is increasing)
            if loss_not_lower_than_lowest_counter >= self.patience:

                print(f"\nEarly stopped at iteration {number_of_iteration}., where the patience limit "
                f"({self.patience}) was reached")

                if training_report == True:
                    print(f"The lowest Loss {lowest_loss.round(decimalplaces_for_rounding)}) "
                    f"was achieved in the {iteration_of_lowest_loss}. iteration with the weight(s) "
                    f"{weights_of_lowest_loss}, and after that there were {loss_not_lower_than_lowest_counter} "
                    f"iterations where the Loss was not lower than that")

                break


        # Collect current iteration's Loss and weights for visualizing Gradient Descent in work
        if training_report == True:
            self.loss_history.append(current_loss)
            for weight in self.weights:
                self.weights_history.append(weight)


        # Before updating weights save the current iteration's Loss (Loss with the current weights)
        previous_loss = current_loss

    # ------------------  End of Functionalities outside of the core training (3 steps) process ------------------


        # 2. Compute gradient (derivative) of the Loss Function
        current_gradient = __class__._compute_gradient(X, y_pred, y, self.weights, self.penalty,
                                                       self.lambda_or_inverseC)
```

```
            previous_loss = current_loss

        # --------------------  End of Functionalities outside of the core training (3 steps) process --------------------


            # 2. Compute gradient (derivative) of the Loss Function
            current_gradient = __class__._compute_gradient(X, y_pred, y, self.weights, self.penalty,
                                                           self.lambda_or_inverseC)

            # 3. Update weights
            self.weights = __class__._update_weights(current_gradient, self.weights, self.penalty,
                                                     self.learning_rate)



        # Visualize Gradient Descent in work
        if training_report == True:
            print(f"\nTraining finished with the following results:\nLast ireation: {number_of_iteration}., "
                  f"Weight(s): {self.weights}, Loss: {current_loss} which was {loss_status} than in previous iteration\n"
                  f"The lowest Loss {lowest_loss} was achieved in the {iteration_of_lowest_loss}. iteration"
                  f"with the weight(s) {weights_of_lowest_loss}\n")

            __class__._visualize_graddesc(np.array(self.loss_history), np.array(self.weights_history),
                                          lowest_loss, num_features)

        else:
            if self.max_iter == 0: # same: number_of_iteration == 0:
                print("There was no training performed as maximum iterations for the training are set to 0\n")

            else:
                print(f"\nTraining finished after {number_of_iteration} iterations\n")


        # In case the training process was either stopped with early stopping by reaching the patience limit,
        # or wasn't stopped before the maximum iterations, set the weight(s)
        # to those weight(s) with whom the lowest loss was achieved (if training was stopped with convergence, then
        # the last iteration's weights produce the lowest loss and thus this below is useless)
        if number_of_iteration > 1:
            self.weights = weights_of_lowest_loss


        return self


    def predict_proba(self, X):
        X = __class__._standardizeX(X, self.X_train_means, self.X_train_stds)

        distances = __class__._calculate_distance(self.weights, X)

        return __class__._predict_probability(distances)

    def predict(self, X):
        y_pred_proba = self.predict_proba(X)

        return y_pred_proba >= self.treshold
```

Applying created model, analyzing the training process and comparing performance to sklearn's model

```
X, y = datasets.make_classification(n_samples=10000, n_features=1, n_clusters_per_class= 1, n_informative = 1,
                                    n_redundant = 0, n_classes = 2, random_state=11)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)

num_samples, num_features = X.shape
print("Number of generated samples:", num_samples, "Number of generated features:", num_features, "\n\n")

print("----------------- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight "
      "-----------------\n\n")
```

```
        return y_pred_proba >= self.threshold
```

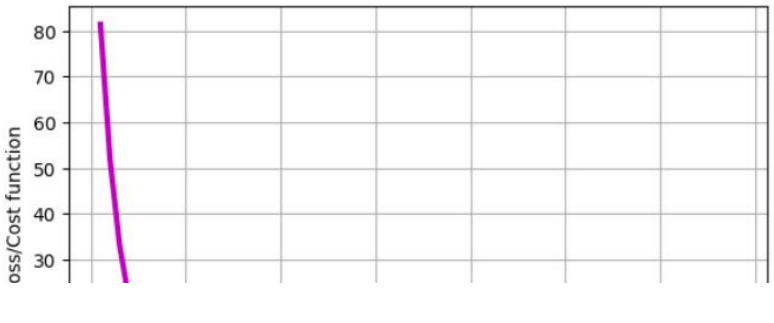
Applying created model, analyzing the training process and comparing performance to sklearn's model

```
X, y = datasets.make_classification(n_samples=10000, n_features=1, n_clusters_per_class= 1, n_informative = 1,
                                    n_redundant = 0, n_classes = 2, random_state=11)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)

num_samples, num_features = X.shape
print("Number of generated samples:", num_samples, "Number of generated features:", num_features, "\n\n")

print("----------------- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight "
      "-----------------\n\n")

model1 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
            initialweights_lowerlimit = 2, # on this particular data the optimal weight is around 0.2
            initialweights_upperlimit = 10 # that's why these limits are given)
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model1.predict(X_train)
y_pred = model1.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model2 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
            initialweights_lowerlimit = 0.2, # on this particular data the optimal weight is around 0.2
            initialweights_upperlimit = 2 # that's why these limits are given
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model2.predict(X_train)
y_pred = model2.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model3 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
            initialweights_lowerlimit = 0, # on this particular data the optimal weight is around 0.2
            initialweights_upperlimit = 0.2 # that's why these limits are given
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model3.predict(X_train)
y_pred = model3.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


print("----------------- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate "
```

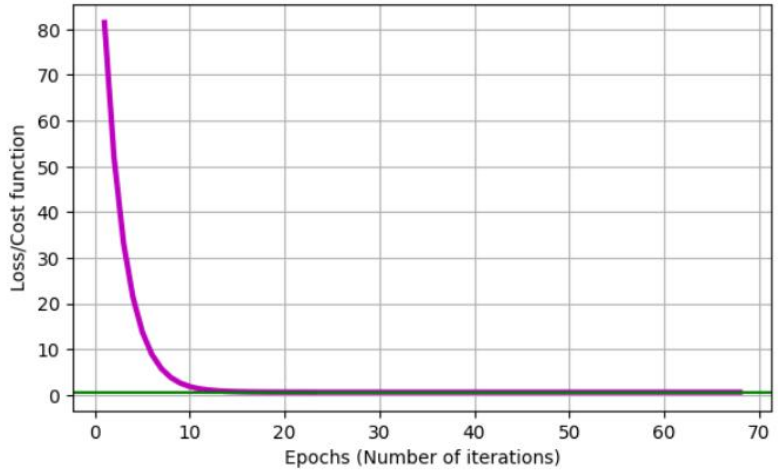
```
print("----------------- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate "
      "-----------------\n\n")


model4 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.1,
            initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
            initialweights_upperlimit = 0
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model4.predict(X_train)
y_pred = model4.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model5 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.4,
            initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
            initialweights_upperlimit = 0
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model5.predict(X_train)
y_pred = model5.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model6 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.5,
            initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
            initialweights_upperlimit = 0
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model6.predict(X_train)
y_pred = model6.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model7 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.8,
            initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
            initialweights_upperlimit = 0
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model7.predict(X_train)
```

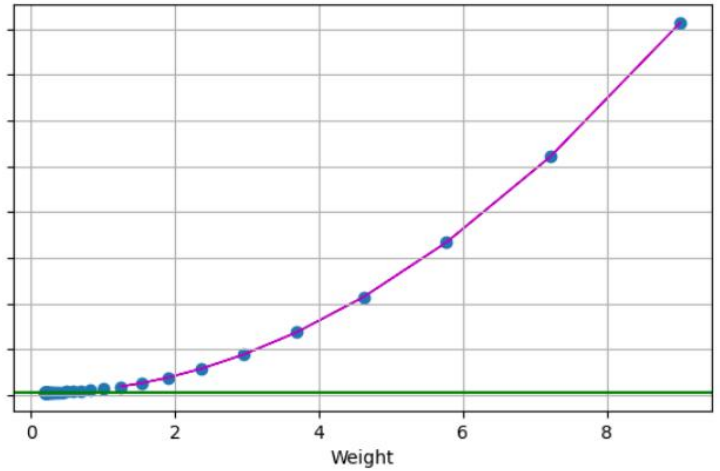
```
y_train_pred = model7.predict(X_train)
y_pred = model7.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model8 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.9,
            initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
            initialweights_upperlimit = 0
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model8.predict(X_train)
y_pred = model8.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")
```

```
Number of generated samples: 10000 Number of generated features: 1


----------------- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight -----------------


Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [2, 10), Initial Loss: 81.45833969862673 with the [9.01916148] initial weight(s)

It: 1. Loss: 81.458339699, Weight(s): [9.01916148], Gradient: [18.0461857]
It: 2. Loss: 52.150078823 lower by 2.9e+01 than in previous iteration, Weight(s): [7.2145429], Gradient: [14.4348743]
It: 3. Loss: 33.399821632 lower by 1.9e+01 than in previous iteration, Weight(s): [5.77105548], Gradient: [11.5435534]
It: 4. Loss: 21.411227002 lower by 1.2e+01 than in previous iteration, Weight(s): [4.61670014], Gradient: [9.2263133]
It: 5. Loss: 13.756035351 lower by 7.7e+00 than in previous iteration, Weight(s): [3.69406881], Gradient: [7.366173]

It: 10. Loss: 1.852708364 lower by 7.6e-01 than in previous iteration, Weight(s): [1.24248932], Gradient: [2.2945471]
It: 20. Loss: 0.655843041 lower by 5.2e-03 than in previous iteration, Weight(s): [0.28396894], Gradient: [0.1874847]
It: 30. Loss: 0.64805589 lower by 3.2e-05 than in previous iteration, Weight(s): [0.20698242], Gradient: [0.0147641]
It: 40. Loss: 0.648007653 lower by 2.0e-07 than in previous iteration, Weight(s): [0.2009244], Gradient: [0.0011607]
It: 50. Loss: 0.648007355 lower by 1.2e-09 than in previous iteration, Weight(s): [0.20044817], Gradient: [9.12e-05]
It: 60. Loss: 0.648007353 lower by 7.6e-12 than in previous iteration, Weight(s): [0.20041074], Gradient: [7.2e-06]

Converged at iteration 69., where the Loss (0.648007353207076) was lower by 7.8e-14 than
in the prev it(0.648007353207154), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 69., Weight(s): [0.20040787], Loss: 0.6480073532070763 which was lower than in previous iteration
The lowest Loss 0.6480073532070763 was achieved in the 69. iterationwith the weight(s) [0.20040787]
```



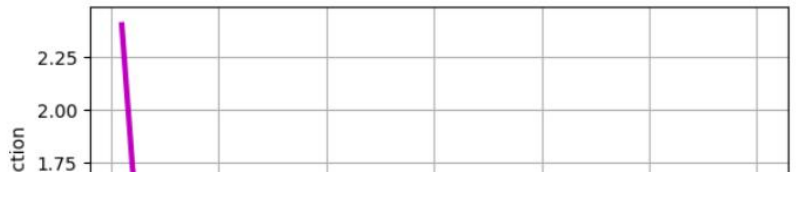
```
It: 60. Loss: 0.64800733 lower by 7.6e-12 than in previous iteration, Weight(s): [0.20041074], Gradient: [7.2e-06]

Converged at iteration 69., where the Loss (0.648007353207076) was lower by 7.8e-14 than
in the prev it(0.648007353207154), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 69., Weight(s): [0.20040787], Loss: 0.6480073532070763 which was lower than in previous iteration
The lowest Loss 0.6480073532070763 was achieved in the 69. iterationwith the weight(s) [0.20040787]
```

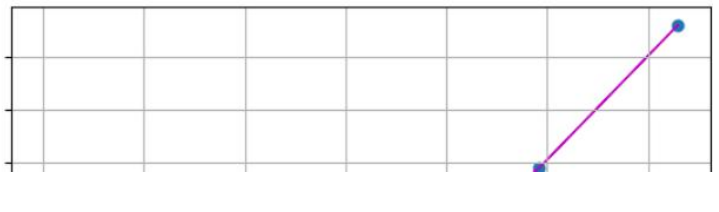




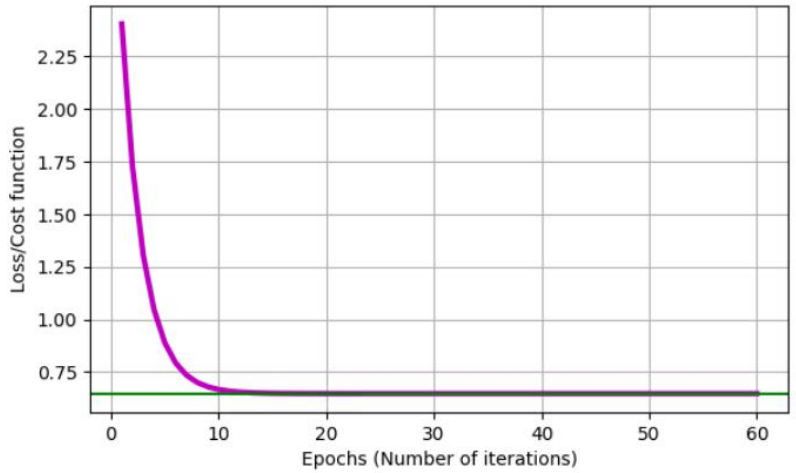

```
Accuracy on Train set: 0.969875
Accuracy on Test set: 0.963
```

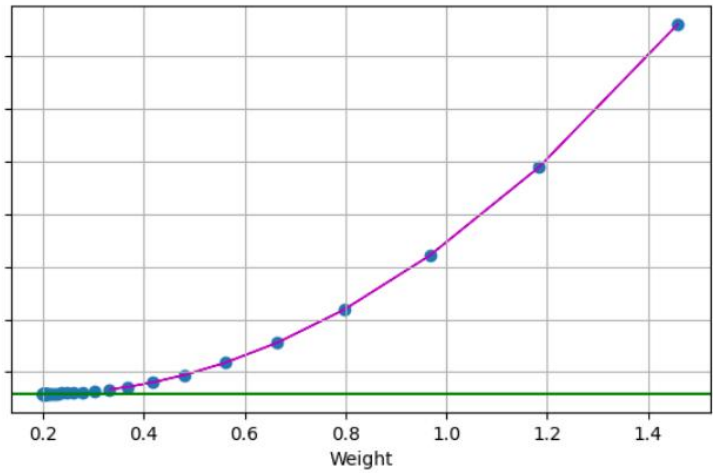
```
Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0.2, 2), Initial Loss: 2.402061731524572 with the [1.4598525] initial weight(s)

It: 1. Loss: 2.402061732, Weight(s): [1.4598525], Gradient: [2.7594064]
It: 2. Loss: 1.721990221 lower by 6.8e-01 than in previous iteration, Weight(s): [1.18391186], Gradient: [2.1684821]
It: 3. Loss: 1.302603773 lower by 4.2e-01 than in previous iteration, Weight(s): [0.96706365], Gradient: [1.6987145]
It: 4. Loss: 1.045549692 lower by 2.6e-01 than in previous iteration, Weight(s): [0.7971922], Gradient: [1.3272254]
It: 5. Loss: 0.88877907 lower by 1.6e-01 than in previous iteration, Weight(s): [0.66446967], Gradient: [1.0348506]

It: 10. Loss: 0.667221669 lower by 1.3e-02 than in previous iteration, Weight(s): [0.33128343], Gradient: [0.2934708]
It: 20. Loss: 0.648126565 lower by 7.9e-05 than in previous iteration, Weight(s): [0.21071179], Gradient: [0.0231376]
It: 30. Loss: 0.64800809 lower by 4.9e-07 than in previous iteration, Weight(s): [0.20121759], Gradient: [0.0018191]
It: 40. Loss: 0.648007358 lower by 3.0e-09 than in previous iteration, Weight(s): [0.20047122], Gradient: [0.000143]
It: 50. Loss: 0.648007353 lower by 1.9e-11 than in previous iteration, Weight(s): [0.20041255], Gradient: [1.12e-05]
It: 60. Loss: 0.648007353 lower by 1.2e-13 than in previous iteration, Weight(s): [0.20040794], Gradient: [9.e-07]

Converged at iteration 61., where the Loss (0.648007353207063) was lower by 6.9e-14 than
in the prev it(0.648007353207132), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 61., Weight(s): [0.20040785], Loss: 0.6480073532070632 which was lower than in previous iteration
The lowest Loss 0.6480073532070632 was achieved in the 61. iterationwith the weight(s) [0.20040785]
```





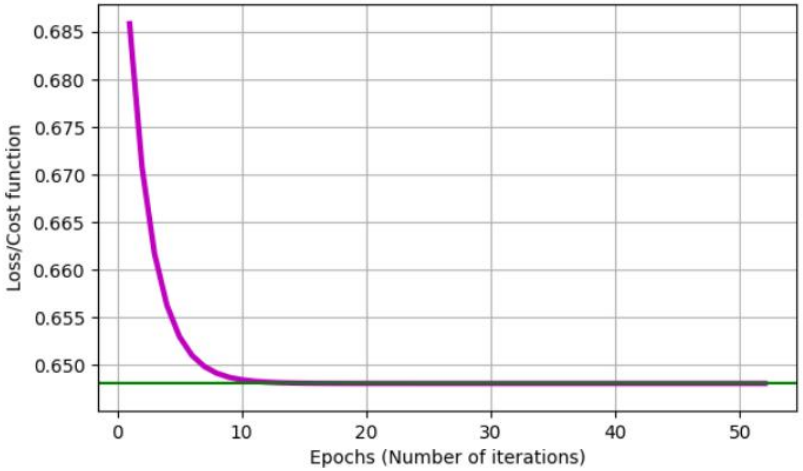
```
It: 1. Loss: 2.402061732, Weight(s): [1.4598525], Gradient: [2.7594064]
It: 2. Loss: 1.721990221 lower by 6.8e-01 than in previous iteration, Weight(s): [1.18391186], Gradient: [2.1684821]
It: 3. Loss: 1.302603773 lower by 4.2e-01 than in previous iteration, Weight(s): [0.96706365], Gradient: [1.6987145]
It: 4. Loss: 1.045549692 lower by 2.6e-01 than in previous iteration, Weight(s): [0.7971922], Gradient: [1.3272254]
It: 5. Loss: 0.88877907 lower by 1.6e-01 than in previous iteration, Weight(s): [0.66446967], Gradient: [1.0348506]

It: 10. Loss: 0.667221669 lower by 1.3e-02 than in previous iteration, Weight(s): [0.33128343], Gradient: [0.2934708]
It: 20. Loss: 0.648126565 lower by 7.9e-05 than in previous iteration, Weight(s): [0.21071179], Gradient: [0.0231376]
It: 30. Loss: 0.64800809 lower by 4.9e-07 than in previous iteration, Weight(s): [0.20121759], Gradient: [0.0018191]
It: 40. Loss: 0.648007358 lower by 3.0e-09 than in previous iteration, Weight(s): [0.20047122], Gradient: [0.000143]
It: 50. Loss: 0.648007353 lower by 1.9e-11 than in previous iteration, Weight(s): [0.20041255], Gradient: [1.12e-05]
It: 60. Loss: 0.648007353 lower by 1.2e-13 than in previous iteration, Weight(s): [0.20040794], Gradient: [9.e-07]

Converged at iteration 61., where the Loss (0.648007353207063) was lower by 6.9e-14 than
in the prev it(0.648007353207132), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 61., Weight(s): [0.20040785], Loss: 0.6480073532070632 which was lower than in previous iteration
The lowest Loss 0.6480073532070632 was achieved in the 61. iterationwith the weight(s) [0.20040785]
```

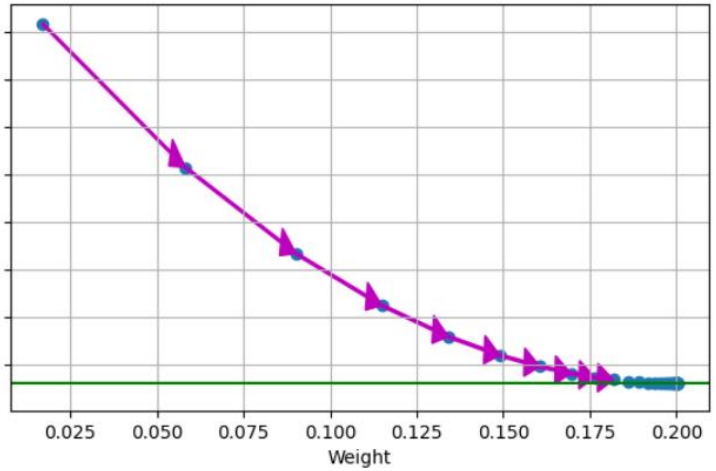





```
Accuracy on Train set: 0.96825
Accuracy on Test set: 0.963


Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0.2), Initial Loss: 0.685828398991805 with the [0.01695944] initial weight(s)

It: 1. Loss: 0.685828399, Weight(s): [0.01695944], Gradient: [-0.4124669]
It: 2. Loss: 0.670729356 lower by 1.5e-02 than in previous iteration, Weight(s): [0.05820613], Gradient: [-0.3196689]
It: 3. Loss: 0.661659883 lower by 9.1e-03 than in previous iteration, Weight(s): [0.09017303], Gradient: [-0.247763]
It: 4. Loss: 0.656211505 lower by 5.4e-03 than in previous iteration, Weight(s): [0.11494932], Gradient: [-0.1920449]
It: 5. Loss: 0.652937996 lower by 3.3e-03 than in previous iteration, Weight(s): [0.13415382], Gradient: [-0.1488674]

It: 10. Loss: 0.648394399 lower by 2.6e-04 than in previous iteration, Weight(s): [0.18184239], Gradient: [-0.0416983]
It: 20. Loss: 0.648009743 lower by 1.6e-06 than in previous iteration, Weight(s): [0.19894856], Gradient: [-0.0032765]
It: 30. Loss: 0.648007368 lower by 9.8e-09 than in previous iteration, Weight(s): [0.20029287], Gradient: [-0.0002575]
It: 40. Loss: 0.648007353 lower by 6.0e-11 than in previous iteration, Weight(s): [0.20039853], Gradient: [-2.02e-05]
It: 50. Loss: 0.648007353 lower by 3.7e-13 than in previous iteration, Weight(s): [0.20040684], Gradient: [-1.6e-06]

Converged at iteration 53., where the Loss (0.648007353207081) was lower by 8.1e-14 than
in the prev it(0.648007353207163), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
```

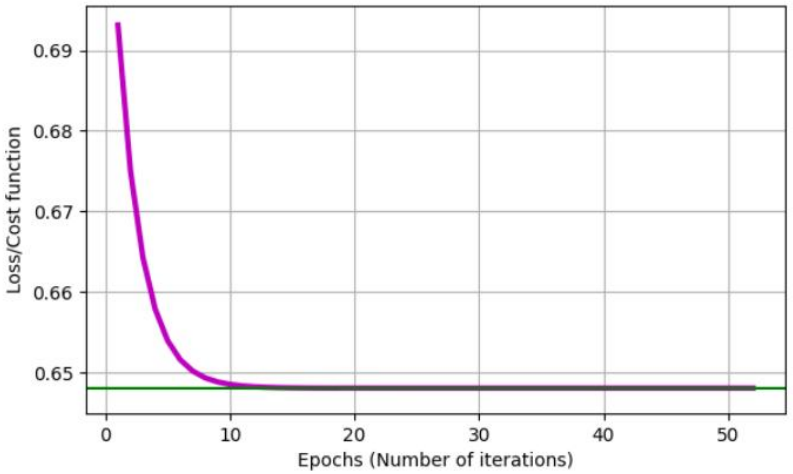
```
Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0.2), Initial Loss: 0.685828398991805 with the [0.01695944] initial weight(s)

It: 1. Loss: 0.685828399, Weight(s): [0.01695944], Gradient: [-0.4124669]
It: 2. Loss: 0.670729356 lower by 1.5e-02 than in previous iteration, Weight(s): [0.05820613], Gradient: [-0.3196689]
It: 3. Loss: 0.661659883 lower by 9.1e-03 than in previous iteration, Weight(s): [0.09017303], Gradient: [-0.247763]
It: 4. Loss: 0.656211505 lower by 5.4e-03 than in previous iteration, Weight(s): [0.11494932], Gradient: [-0.1920449]
It: 5. Loss: 0.652937996 lower by 3.3e-03 than in previous iteration, Weight(s): [0.13415382], Gradient: [-0.1488674]

It: 10. Loss: 0.648394399 lower by 2.6e-04 than in previous iteration, Weight(s): [0.18184239], Gradient: [-0.0416983]
It: 20. Loss: 0.648009743 lower by 1.6e-06 than in previous iteration, Weight(s): [0.19894856], Gradient: [-0.0032765]
It: 30. Loss: 0.648007368 lower by 9.8e-09 than in previous iteration, Weight(s): [0.20029287], Gradient: [-0.0002575]
It: 40. Loss: 0.648007353 lower by 6.0e-11 than in previous iteration, Weight(s): [0.20039853], Gradient: [-2.02e-05]
It: 50. Loss: 0.648007353 lower by 3.7e-13 than in previous iteration, Weight(s): [0.20040684], Gradient: [-1.6e-06]

Converged at iteration 53., where the Loss (0.648007353207081) was lower by 8.1e-14 than
in the prev it(0.648007353207163), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 53., Weight(s): [0.20040722], Loss: 0.6480073532070812 which was lower than in previous iteration
The lowest Loss 0.6480073532070812 was achieved in the 53. iterationwith the weight(s) [0.20040722]
```

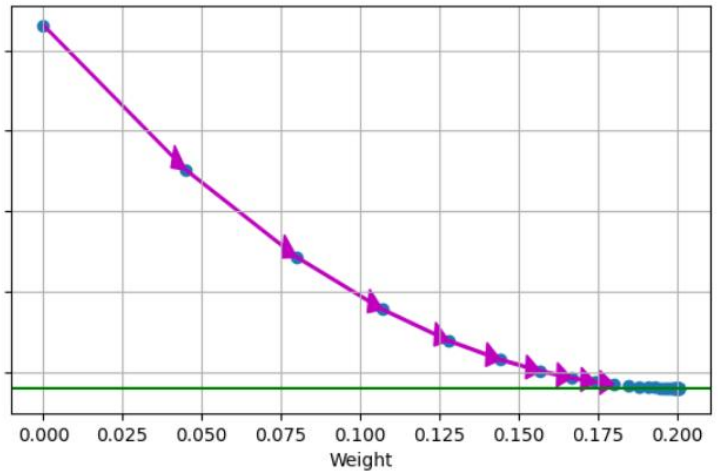





```
Accuracy on Train set: 0.969875
Accuracy on Test set: 0.963

----------------- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate -----------------

Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.675125271 lower by 1.8e-02 than in previous iteration, Weight(s): [0.04506255], Gradient: [-0.3492381]
It: 3. Loss: 0.664300458 lower by 1.1e-02 than in previous iteration, Weight(s): [0.07998636], Gradient: [-0.2706749]
It: 4. Loss: 0.657797878 lower by 6.5e-03 than in previous iteration, Weight(s): [0.10705385], Gradient: [-0.2097991]
It: 5. Loss: 0.653891174 lower by 3.9e-03 than in previous iteration, Weight(s): [0.12803376], Gradient: [-0.1626261]

It: 10. Loss: 0.64846917 lower by 3.1e-04 than in previous iteration, Weight(s): [0.18012839], Gradient: [-0.0455487]
It: 20. Loss: 0.648010205 lower by 1.9e-06 than in previous iteration, Weight(s): [0.19881389], Gradient: [-0.0035789]
It: 30. Loss: 0.648007371 lower by 1.2e-08 than in previous iteration, Weight(s): [0.20028228], Gradient: [-0.0002813]
```

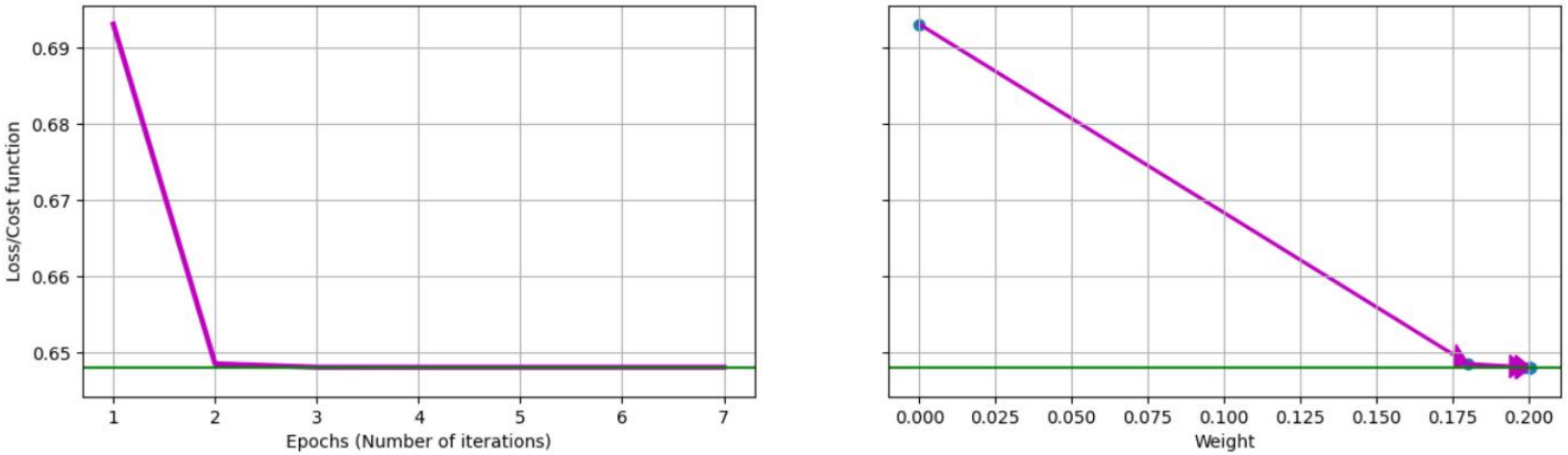
```
----------------- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate -----------------


Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.675125271 lower by 1.8e-02 than in previous iteration, Weight(s): [0.04506255], Gradient: [-0.3492381]
It: 3. Loss: 0.664300458 lower by 1.1e-02 than in previous iteration, Weight(s): [0.07998636], Gradient: [-0.2706749]
It: 4. Loss: 0.657797878 lower by 6.5e-03 than in previous iteration, Weight(s): [0.10705385], Gradient: [-0.2097991]
It: 5. Loss: 0.653891174 lower by 3.9e-03 than in previous iteration, Weight(s): [0.12803376], Gradient: [-0.1626261]

It: 10. Loss: 0.64846917 lower by 3.1e-04 than in previous iteration, Weight(s): [0.18012839], Gradient: [-0.0455487]
It: 20. Loss: 0.648010205 lower by 1.9e-06 than in previous iteration, Weight(s): [0.19881389], Gradient: [-0.0035789]
It: 30. Loss: 0.648007371 lower by 1.2e-08 than in previous iteration, Weight(s): [0.20028228], Gradient: [-0.0002813]
It: 40. Loss: 0.648007353 lower by 7.2e-11 than in previous iteration, Weight(s): [0.2003977], Gradient: [-2.21e-05]
It: 50. Loss: 0.648007353 lower by 4.5e-13 than in previous iteration, Weight(s): [0.20040677], Gradient: [-1.7e-06]

Converged at iteration 53., where the Loss (0.648007353207105) was lower by 9.7e-14 than
in the prev it(0.648007353207202), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 53., Weight(s): [0.20040719], Loss: 0.6480073532071049 which was lower than in previous iteration
The lowest Loss 0.6480073532071049 was achieved in the 53. iterationwith the weight(s) [0.20040719]
```





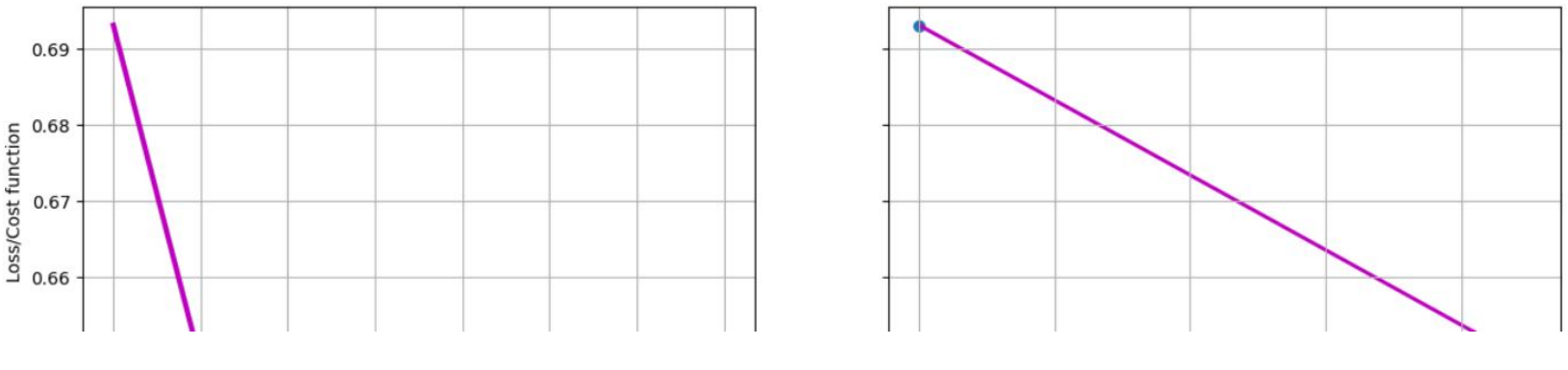

```
Accuracy on Train set: 0.96825
Accuracy on Test set: 0.963


Training initialized with the following hyperparameter settings:
Learning rate: 0.4, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.648463639 lower by 4.5e-02 than in previous iteration, Weight(s): [0.1802502], Gradient: [-0.0452751]
It: 3. Loss: 0.64801206 lower by 4.5e-04 than in previous iteration, Weight(s): [0.19836024], Gradient: [-0.0045977]
It: 4. Loss: 0.648007402 lower by 4.7e-06 than in previous iteration, Weight(s): [0.20019931], Gradient: [-0.0004676]
It: 5. Loss: 0.648007354 lower by 4.8e-08 than in previous iteration, Weight(s): [0.20038636], Gradient: [-4.76e-05]


Converged at iteration 8., where the Loss (0.648007353206959) was lower by 5.3e-14 than
in the prev it(0.648007353207012), which is below the tolerance for convergence: 1.0e-13
```

```
Converged at iteration 8., where the Loss (0.6480073532069959) was lower by 5.3e-14 than
in the prev it(0.648007353207012), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 8., Weight(s): [0.20040753], Loss: 0.6480073532069592 which was lower than in previous iteration
The lowest Loss 0.6480073532069592 was achieved in the 8. iterationwith the weight(s) [0.20040753]
```

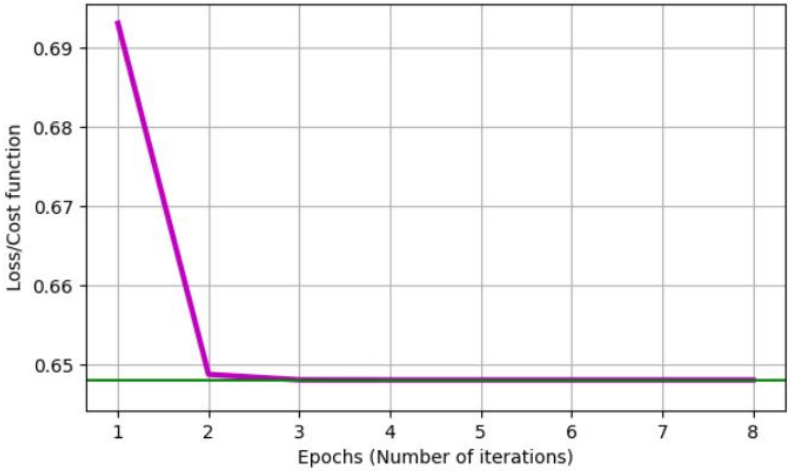


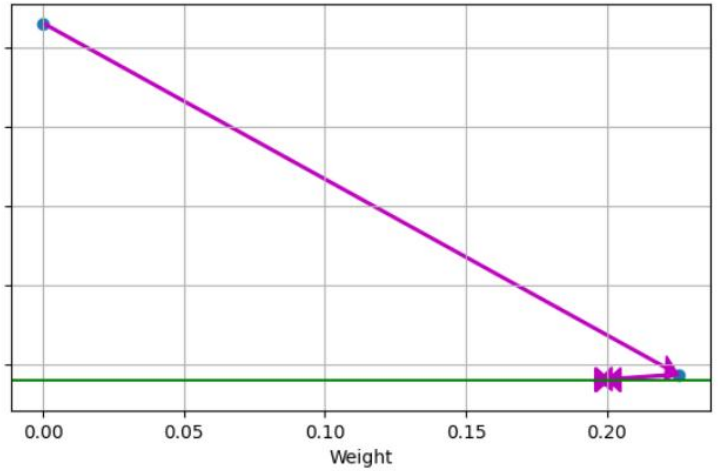

```
Accuracy on Train set: 0.969875
Accuracy on Test set: 0.963
```

```
Training initialized with the following hyperparameter settings:
Learning rate: 0.5, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.648703701 lower by 4.4e-02 than in previous iteration, Weight(s): [0.22531275], Gradient: [0.0559152]
It: 3. Loss: 0.648017815 lower by 6.9e-04 than in previous iteration, Weight(s): [0.19735515], Gradient: [-0.0068549]
It: 4. Loss: 0.648007511 lower by 1.0e-05 than in previous iteration, Weight(s): [0.20078259], Gradient: [0.0008422]
It: 5. Loss: 0.648007356 lower by 1.6e-07 than in previous iteration, Weight(s): [0.20036148], Gradient: [-0.0001034]

Converged at iteration 9., where the Loss (0.648007353206959) was lower by 8.0e-15 than
in the prev it(0.648007353206967), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 9., Weight(s): [0.20040754], Loss: 0.6480073532069588 which was lower than in previous iteration
The lowest Loss 0.6480073532069588 was achieved in the 9. iterationwith the weight(s) [0.20040754]
```



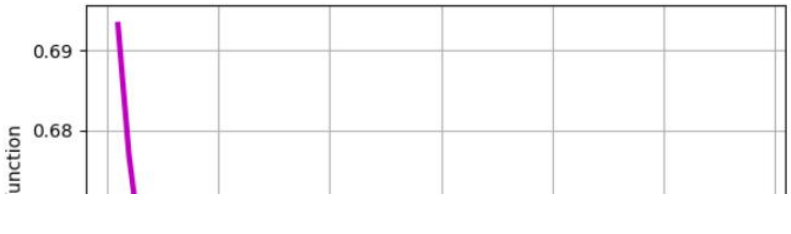
```
Converged at iteration 9., where the Loss (0.648007353206959) was lower by 8.0e-15 than
in the prev it(0.648007353206967), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 9., Weight(s): [0.20040754], Loss: 0.6480073532069588 which was lower than in previous iteration
The lowest Loss 0.6480073532069588 was achieved in the 9. iterationwith the weight(s) [0.20040754]
```

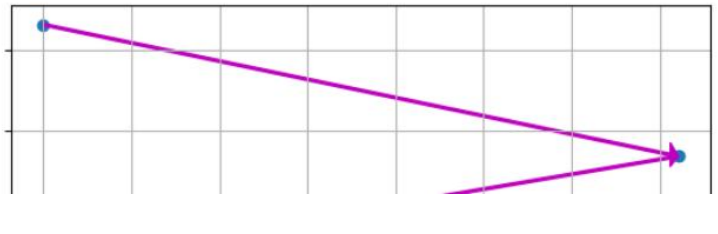




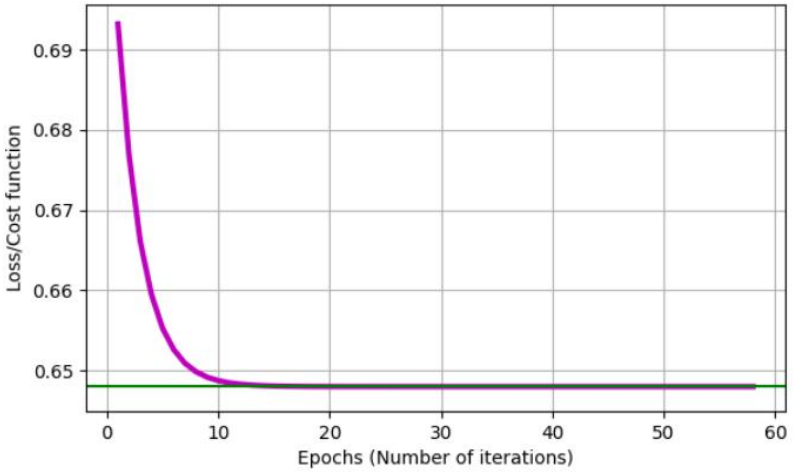

```
Accuracy on Train set: 0.96825
Accuracy on Test set: 0.963
```

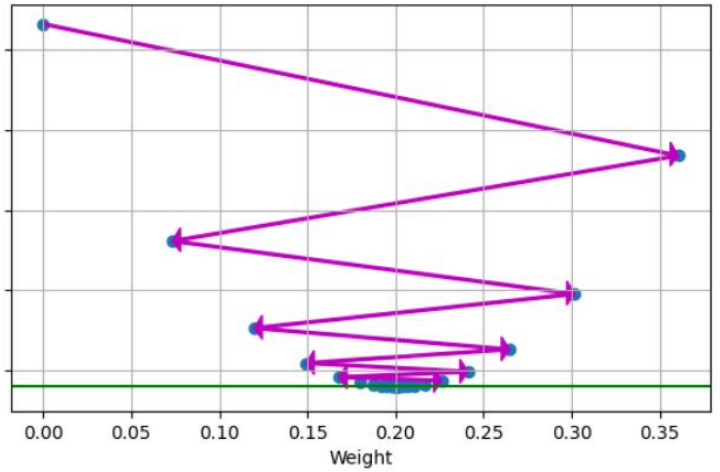
```
Training initialized with the following hyperparameter settings:
Learning rate: 0.8, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.676751161 lower by 1.6e-02 than in previous iteration, Weight(s): [0.3605004], Gradient: [0.3588454]
It: 3. Loss: 0.666125128 lower by 1.1e-02 than in previous iteration, Weight(s): [0.0734241], Gradient: [-0.2854357]
It: 4. Loss: 0.659536162 lower by 6.6e-03 than in previous iteration, Weight(s): [0.3017727], Gradient: [0.2273819]
It: 5. Loss: 0.655294248 lower by 4.2e-03 than in previous iteration, Weight(s): [0.11986717], Gradient: [-0.1809872]

It: 10. Loss: 0.648757173 lower by 4.3e-04 than in previous iteration, Weight(s): [0.22625137], Gradient: [0.058022]
It: 20. Loss: 0.648015275 lower by 4.6e-06 than in previous iteration, Weight(s): [0.20306377], Gradient: [0.0059648]
It: 30. Loss: 0.648007437 lower by 4.8e-08 than in previous iteration, Weight(s): [0.20068064], Gradient: [0.0006133]
It: 40. Loss: 0.648007354 lower by 5.1e-10 than in previous iteration, Weight(s): [0.20043563], Gradient: [6.31e-05]
It: 50. Loss: 0.648007353 lower by 5.4e-12 than in previous iteration, Weight(s): [0.20041044], Gradient: [6.5e-06]

Converged at iteration 59., where the Loss (0.648007353207114) was lower by 9.0e-14 than
in the prev it(0.648007353207204), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 59., Weight(s): [0.20040718], Loss: 0.6480073532071146 which was lower than in previous iteration
The lowest Loss 0.6480073532071146 was achieved in the 59. iterationwith the weight(s) [0.20040718]
```



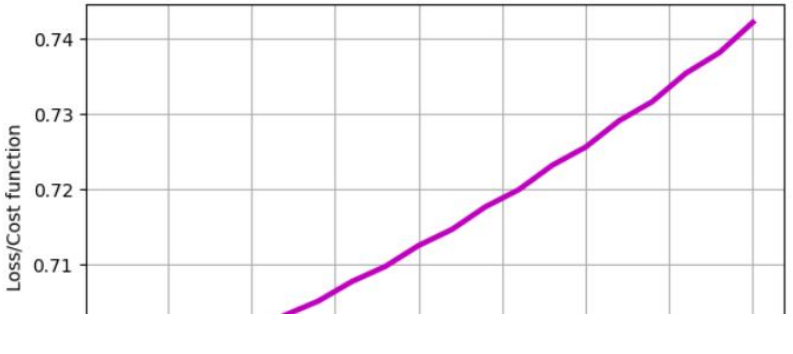


```
in the prev it(0.648007353207204), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 59., Weight(s): [0.20040718], Loss: 0.6480073532071146 which was lower than in previous iteration
The lowest Loss 0.6480073532071146 was achieved in the 59. iterationwith the weight(s) [0.20040718]
```

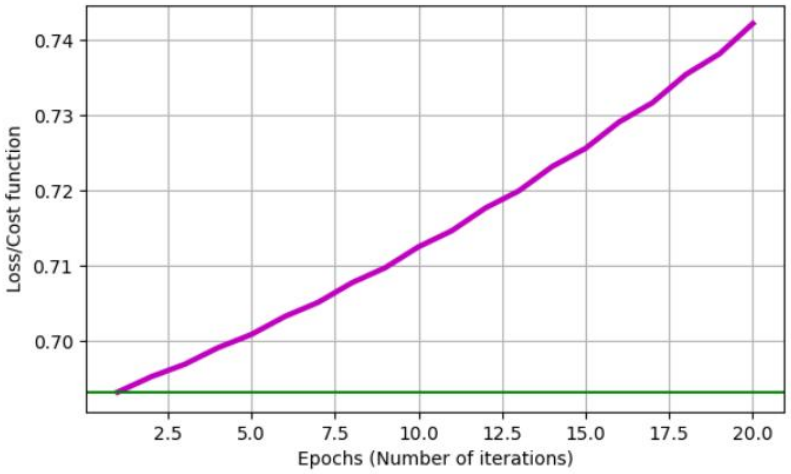


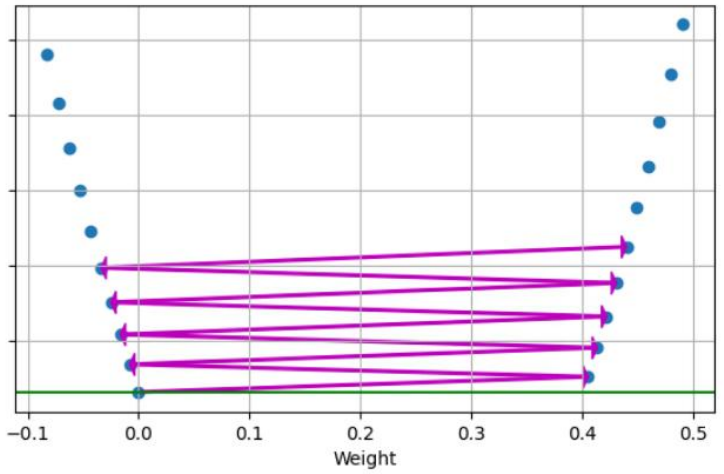



```
Accuracy on Train set: 0.969875
Accuracy on Test set: 0.963


Training initialized with the following hyperparameter settings:
Learning rate: 0.9, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0.], Gradient: [-0.4506255]
It: 2. Loss: 0.695191357 higher by 2.0e-03 than in previous iteration, Weight(s): [0.40556295], Gradient: [0.4595564]
It: 3. Loss: 0.696841896 higher by 1.7e-03 than in previous iteration, Weight(s): [-0.00803779], Gradient: [-0.4687105]
It: 4. Loss: 0.699053298 higher by 2.2e-03 than in previous iteration, Weight(s): [0.41380167], Gradient: [0.4779527]
It: 5. Loss: 0.700818471 higher by 1.8e-03 than in previous iteration, Weight(s): [-0.0163558], Gradient: [-0.4874259]

It: 10. Loss: 0.712479494 higher by 2.8e-03 than in previous iteration, Weight(s): [0.44025842], Gradient: [0.5369923]
It: 20. Loss: 0.742173499 higher by 4.0e-03 than in previous iteration, Weight(s): [0.49034785], Gradient: [0.6486118]

Early stopped at iteration 21., where the patience limit (20) was reached
The lowest Loss 0.693147180559945) was achieved in the 1. iteration with the weight(s) [0.], and after that there were 20 iterations where the Loss was not lower than that

Training finished with the following results:
Last ireation: 21., Weight(s): [-0.09340276], Loss: 0.7450507371802867 which was higher than in previous iteration
The lowest Loss 0.6931471805599454 was achieved in the 1. iterationwith the weight(s) [0.]
```



```
Early stopped at iteration 21., where the patience limit (20) was reached
The lowest Loss 0.693147180559945) was achieved in the 1. iteration with the weight(s) [0.], and after that there were 20 iterations where the Loss was not lower than that

Training finished with the following results:
Last ireation: 21., Weight(s): [-0.09340276], Loss: 0.7450507371802867 which was higher than in previous iteration
The lowest Loss 0.6931471805599454 was achieved in the 1. iterationwith the weight(s) [0.]
```






```
Accuracy on Train set: 0.494875
Accuracy on Test set: 0.5245
```

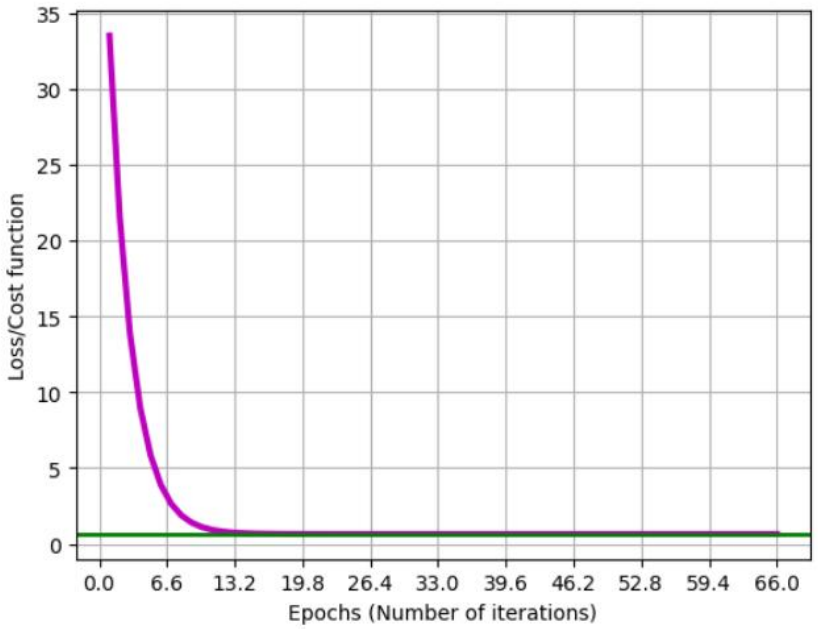
```python
X, y = datasets.make_classification(n_samples=10000, n_features=2, n_clusters_per_class= 1, n_informative = 1,
                                    n_redundant = 0, n_classes = 2, random_state=11)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)

num_samples, num_features = X.shape
print("Number of generated samples:", num_samples, "Number of generated features:", num_features, "\n\n")

print("----------------- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight "
      "-----------------\n\n")

model1 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
            initialweights_lowerlimit = 2,
            initialweights_upperlimit = 6
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model1.predict(X_train)
y_pred = model1.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model2 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
```

```
model2 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
            initialweights_lowerlimit = 0.2,
            initialweights_upperlimit = 2
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model2.predict(X_train)
y_pred = model2.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model3 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.1, # fixed learning rate to compare different initial weight(s)
            initialweights_lowerlimit = 0,
            initialweights_upperlimit = 0.2
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model3.predict(X_train)
y_pred = model3.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


print("----------------- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate "
      "-----------------\n\n")


model4 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.1,
            initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
            initialweights_upperlimit = 0
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model4.predict(X_train)
y_pred = model4.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model5 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.4,
            initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
            initialweights_upperlimit = 0
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model5.predict(X_train)
y_pred = model5.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
```

```
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model5.predict(X_train)
y_pred = model5.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model6 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.5,
            initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
            initialweights_upperlimit = 0
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model6.predict(X_train)
y_pred = model6.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model7 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.8,
            initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
            initialweights_upperlimit = 0
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model7.predict(X_train)
y_pred = model7.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")


model8 = LogRegNumPy(penalty = DEFAULTSKPARAMS['penalty'], #='l2'
            lambda_or_inverseC = DEFAULTSKPARAMS['C'], #=1.0
            max_iter = DEFAULTSKPARAMS['max_iter'], #=100
            tol = 1e-13, #DEFAULTSKPARAMS['tol'], #=0.0001=1e-4,
            learning_rate = 0.9,
            initialweights_lowerlimit = 0, # fixed zero initial weight(s) to compare different learning rates
            initialweights_upperlimit = 0
            ).fit(X_train, y_train, training_report = True)

y_train_pred = model8.predict(X_train)
y_pred = model8.predict(X_test)
train_accuracy = accuracy_score(y_train_pred, y_train)
test_accuracy = accuracy_score(y_pred, y_test)
print("Accuracy on Train set:", train_accuracy)
print("Accuracy on Test set:", test_accuracy)
print("\n\n\n")
```

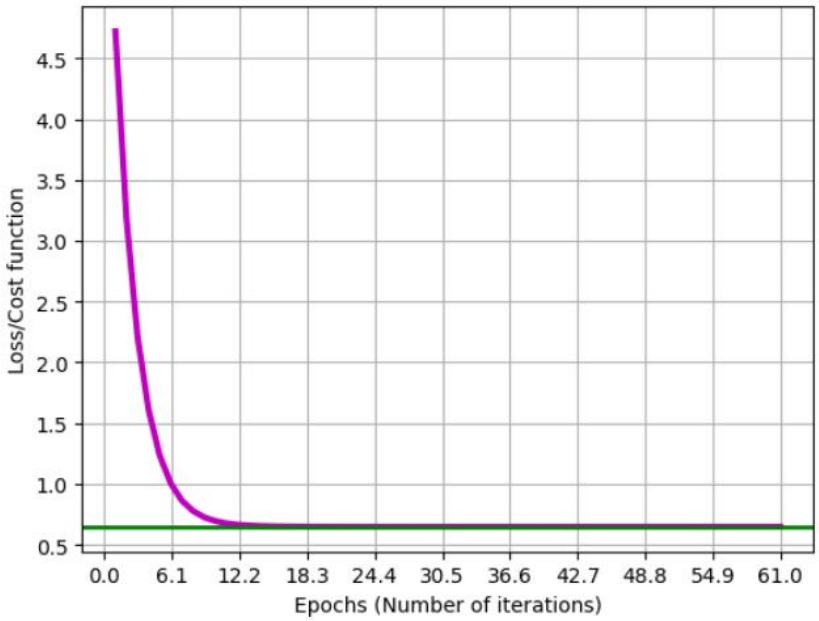
```
Number of generated samples: 10000 Number of generated features: 2


----------------- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight -----------------
```

```
Number of generated samples: 10000 Number of generated features: 2


----------------- Comparing effect of initial weights: Fixed learning rate (0.1), decreasing initial weight -----------------


Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [2, 6), Initial Loss: 33.50612445884154 with the [2.89818808 4.98351743] initial weight(s)

It: 1. Loss: 33.506124459, Weight(s): [2.89818808 4.98351743], Gradient: [5.9355506 9.9071946]
It: 2. Loss: 21.505991554 lower by 1.2e+01 than in previous iteration, Weight(s): [2.30463302 3.99279798], Gradient: [4.7491078 7.9154874]
It: 3. Loss: 13.842442023 lower by 7.7e+00 than in previous iteration, Weight(s): [1.82972224 3.20124924], Gradient: [3.7988291 6.3178954]
It: 4. Loss: 8.957239129 lower by 4.9e+00 than in previous iteration, Weight(s): [1.44983933 2.56945969], Gradient: [3.0364183 5.0349542]
It: 5. Loss: 5.852332528 lower by 3.1e+00 than in previous iteration, Weight(s): [1.1461975  2.06596427], Gradient: [2.4232886 4.0036814]

It: 10. Loss: 1.105823629 lower by 2.9e-01 than in previous iteration, Weight(s): [0.34197211 0.7447842 ], Gradient: [0.7449147 1.2123816]
It: 20. Loss: 0.651182719 lower by 1.9e-03 than in previous iteration, Weight(s): [0.03357508 0.24264454], Gradient: [0.0591315 0.0969486]
It: 30. Loss: 0.648336759 lower by 1.2e-05 than in previous iteration, Weight(s): [0.00942578 0.2029147 ], Gradient: [0.0045614 0.0075794]
It: 40. Loss: 0.648319501 lower by 7.0e-08 than in previous iteration, Weight(s): [0.00756396 0.19981003], Gradient: [0.0003514 0.000592 ]
It: 50. Loss: 0.648319397 lower by 4.2e-10 than in previous iteration, Weight(s): [0.00742056 0.19956755], Gradient: [2.71e-05 4.62e-05]
It: 60. Loss: 0.648319396 lower by 2.6e-12 than in previous iteration, Weight(s): [0.00740952 0.19954862], Gradient: [2.1e-06 3.6e-06]

Converged at iteration 67., where the Loss (0.648319396111103) was lower by 7.2e-14 than
in the prev it(0.648319396111175), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 67., Weight(s): [0.00740875 0.19954728], Loss: 0.6483193961111027 which was lower than in previous iteration
The lowest Loss 0.6483193961111027 was achieved in the 67. iterationwith the weight(s) [0.00740875 0.19954728]
```



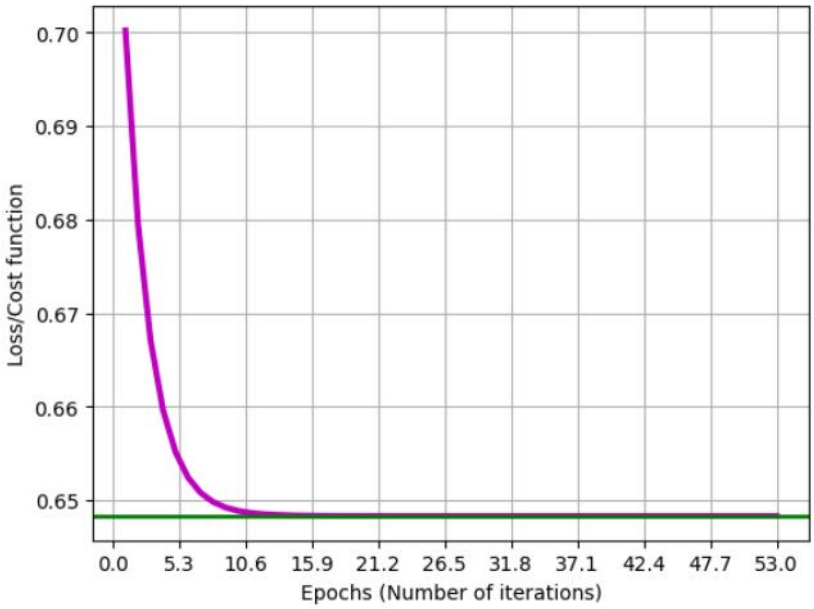

```
Accuracy on Train set: 0.96625
Accuracy on Test set: 0.962


Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0.2, 2), Initial Loss: 4.72373869589876 with the [1.6994989  1.13235245] initial weight(s)

It: 1. Loss: 4.723738696, Weight(s): [1.6994989  1.13235245], Gradient: [3.6353942 2.0006157]
It: 2. Loss: 3.17983031 lower by 1.5e+00 than in previous iteration, Weight(s): [1.33595948 0.93229089], Gradient: [2.8818782 1.5872726]
```

```
Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0.2, 2), Initial Loss: 4.72373869589876 with the [1.6994989  1.13235245] initial weight(s)

It: 1. Loss: 4.723738696, Weight(s): [1.6994989  1.13235245], Gradient: [3.6353942 2.0006157]
It: 2. Loss: 3.17983031 lower by 1.5e+00 than in previous iteration, Weight(s): [1.33595948 0.93229089], Gradient: [2.8818782 1.5872726]
It: 3. Loss: 2.210685714 lower by 9.7e-01 than in previous iteration, Weight(s): [1.04777166 0.77356363], Gradient: [2.276655 1.254969]
It: 4. Loss: 1.606615752 lower by 6.0e-01 than in previous iteration, Weight(s): [0.82010616 0.64806673], Gradient: [1.7918589 0.9884953]
It: 5. Loss: 1.232889744 lower by 3.7e-01 than in previous iteration, Weight(s): [0.64092027 0.5492172 ], Gradient: [1.4052178 0.7757172]

It: 10. Loss: 0.69533007 lower by 3.1e-02 than in previous iteration, Weight(s): [0.18643102 0.29819371], Gradient: [0.4022017 0.2221722]
It: 20. Loss: 0.648605632 lower by 1.9e-04 than in previous iteration, Weight(s): [0.02139807 0.20718176], Gradient: [0.0315012 0.0172579]
It: 30. Loss: 0.648321127 lower by 1.2e-06 than in previous iteration, Weight(s): [0.00849941 0.2001355 ], Gradient: [0.0024565 0.0013306]
It: 40. Loss: 0.648319407 lower by 7.0e-09 than in previous iteration, Weight(s): [0.00749365 0.19959235], Gradient: [0.0001915 0.0001025]
It: 50. Loss: 0.648319396 lower by 4.2e-11 than in previous iteration, Weight(s): [0.00741523 0.1995505 ], Gradient: [1.49e-05 7.90e-06]
It: 60. Loss: 0.648319396 lower by 2.6e-13 than in previous iteration, Weight(s): [0.00740911 0.19954728], Gradient: [1.2e-06 6.0e-07]

Converged at iteration 62., where the Loss (0.648319396111132) was lower by 9.2e-14 than
in the prev it(0.648319396111224), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 62., Weight(s): [0.00740891 0.19954717], Loss: 0.6483193961111323 which was lower than in previous iteration
The lowest Loss 0.6483193961111323 was achieved in the 62. iterationwith the weight(s) [0.00740891 0.19954717]
```



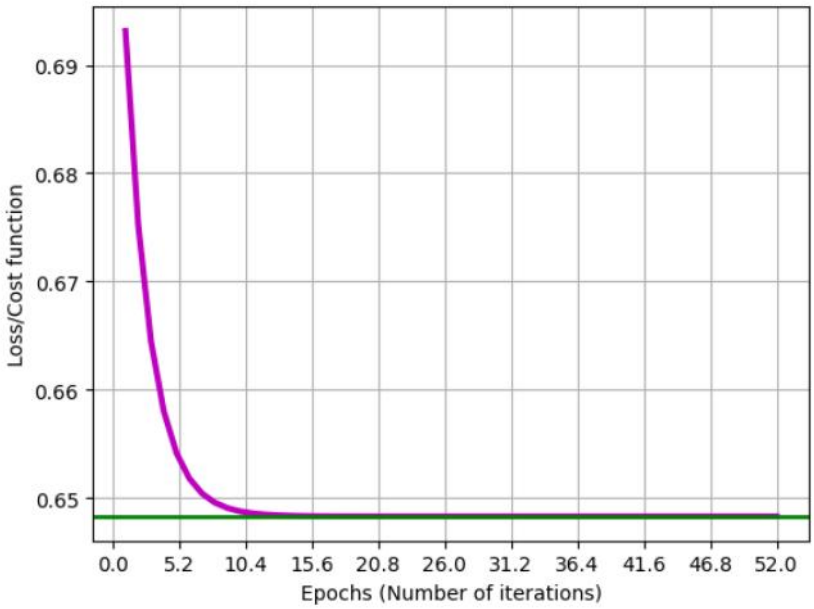

```
Accuracy on Train set: 0.9665
Accuracy on Test set: 0.9615

Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0.2), Initial Loss: 0.700201731952336 with the [0.13328449 0.02497659] initial weight(s)

It: 1. Loss: 0.700201732, Weight(s): [0.13328449 0.02497659], Gradient: [ 0.2815861 -0.3914447]
It: 2. Loss: 0.679554759 lower by 2.1e-02 than in previous iteration, Weight(s): [0.10512588 0.06412106], Gradient: [ 0.2186186 -0.3036367]
It: 3. Loss: 0.667124523 lower by 1.2e-02 than in previous iteration, Weight(s): [0.08326402 0.09448472], Gradient: [ 0.1697081 -0.2355242]
It: 4. Loss: 0.65964161 lower by 7.5e-03 than in previous iteration, Weight(s): [0.0662932  0.11803714], Gradient: [ 0.1317335 -0.1826979]
It: 5. Loss: 0.655136819 lower by 4.5e-03 than in previous iteration, Weight(s): [0.05311986 0.13630693], Gradient: [ 0.1022557 -0.1417279]

It: 10. Loss: 0.648859546 lower by 3.6e-04 than in previous iteration, Weight(s): [0.02029863 0.18175931], Gradient: [ 0.0288268 -0.039847 ]
It: 20. Loss: 0.648322794 lower by 2.2e-06 than in previous iteration, Weight(s): [0.00843426 0.19813834], Gradient: [ 0.0022935 -0.003155 ]
It: 30. Loss: 0.648319418 lower by 1.4e-08 than in previous iteration, Weight(s): [0.00749022 0.19943542], Gradient: [ 0.0001825 -0.0002499]
It: 40. Loss: 0.648319396 lower by 8.9e-11 than in previous iteration, Weight(s): [0.00741509 0.19953817], Gradient: [ 1.45e-05 -1.98e-05]
```

```
It: 30. Loss: 0.648319418 lower by 1.4e-08 than in previous iteration, Weight(s): [0.00749022 0.19943542], Gradient: [ 0.0001825 -0.0002499]
It: 40. Loss: 0.648319396 lower by 8.9e-11 than in previous iteration, Weight(s): [0.00741509 0.19953817], Gradient: [ 1.45e-05 -1.98e-05]
It: 50. Loss: 0.648319396 lower by 5.6e-13 than in previous iteration, Weight(s): [0.00740911 0.19954631], Gradient: [ 1.2e-06 -1.6e-06]

Converged at iteration 54., where the Loss (0.648319396111106) was lower by 7.4e-14 than
in the prev it(0.64831939611118), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 54., Weight(s): [0.00740878 0.19954676], Loss: 0.6483193961111063 which was lower than in previous iteration
The lowest Loss 0.6483193961111063 was achieved in the 54. iterationwith the weight(s) [0.00740878 0.19954676]
```



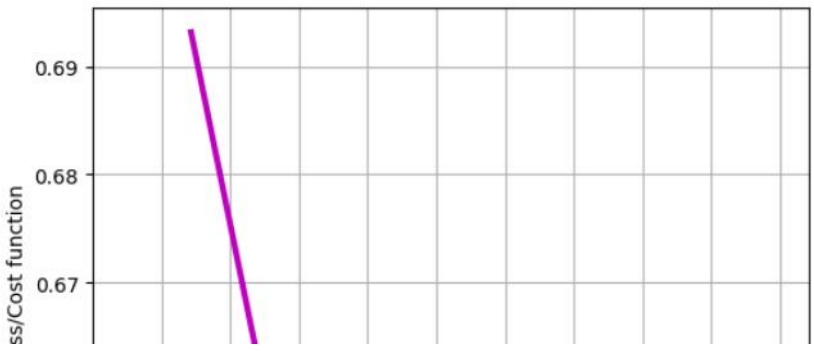

```
Accuracy on Train set: 0.96625
Accuracy on Test set: 0.962


----------------- Comparing effect of learning rate: Fixed initial weight (0), increasing learning rate -----------------


Training initialized with the following hyperparameter settings:
Learning rate: 0.1, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]
It: 2. Loss: 0.675245175 lower by 1.8e-02 than in previous iteration, Weight(s): [0.00183623 0.04487573], Gradient: [-0.0138445 -0.3477744]
It: 3. Loss: 0.664494313 lower by 1.1e-02 than in previous iteration, Weight(s): [0.00322068 0.07965317], Gradient: [-0.0104313 -0.2695284]
It: 4. Loss: 0.658037291 lower by 6.5e-03 than in previous iteration, Weight(s): [0.00426381 0.10660601], Gradient: [-0.0078545 -0.2089012]
It: 5. Loss: 0.654158622 lower by 3.9e-03 than in previous iteration, Weight(s): [0.00504925 0.12749614], Gradient: [-0.0059102 -0.161923 ]

It: 10. Loss: 0.64877737 lower by 3.0e-04 than in previous iteration, Weight(s): [0.00685711 0.17936223], Gradient: [-0.001408  -0.0453424]
It: 20. Loss: 0.648322221 lower by 1.9e-06 than in previous iteration, Weight(s): [0.00738237 0.19796126], Gradient: [-7.2100e-05 -3.5614e-03]
It: 30. Loss: 0.648319414 lower by 1.2e-08 than in previous iteration, Weight(s): [0.00740788 0.19942239], Gradient: [-2.700e-06 -2.799e-04]
It: 40. Loss: 0.648319396 lower by 7.1e-11 than in previous iteration, Weight(s): [0.00740865 0.19953722], Gradient: [ 0.0e+00 -2.2e-05]
It: 50. Loss: 0.648319396 lower by 4.4e-13 than in previous iteration, Weight(s): [0.00740861 0.19954624], Gradient: [ 0.0e+00 -1.7e-06]

Converged at iteration 53., where the Loss (0.648319396111139) was lower by 9.6e-14 than
in the prev it(0.648319396111235), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 53., Weight(s): [0.0074086  0.19954665], Loss: 0.6483193961111394 which was lower than in previous iteration
The lowest Loss 0.6483193961111394 was achieved in the 53. iterationwith the weight(s) [0.0074086  0.19954665]
```

```
Training finished with the following results:
Last ireation: 53., Weight(s): [0.0074086  0.19954665], Loss: 0.6483193961111394 which was lower than in previous iteration
The lowest Loss 0.6483193961111394 was achieved in the 53. iterationwith the weight(s) [0.0074086  0.19954665]
```

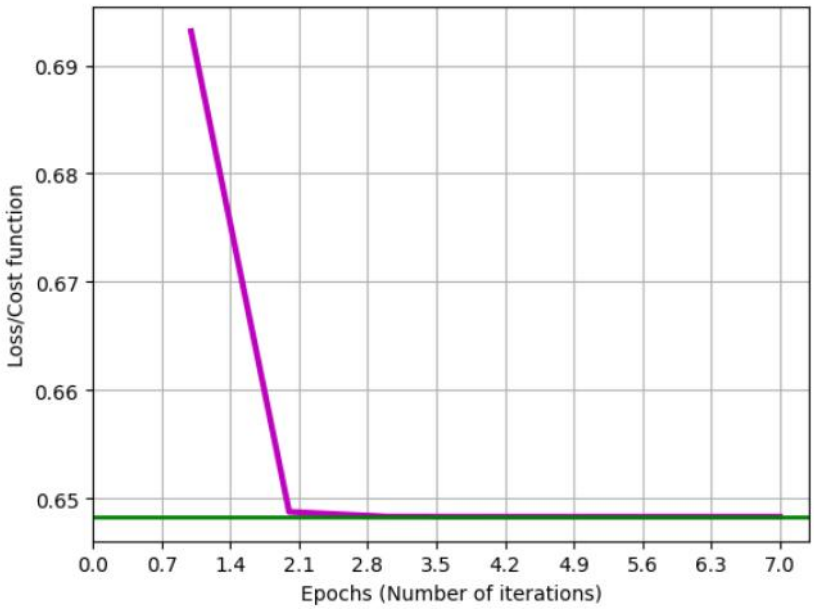



```
Accuracy on Train set: 0.9665
Accuracy on Test set: 0.9615
```

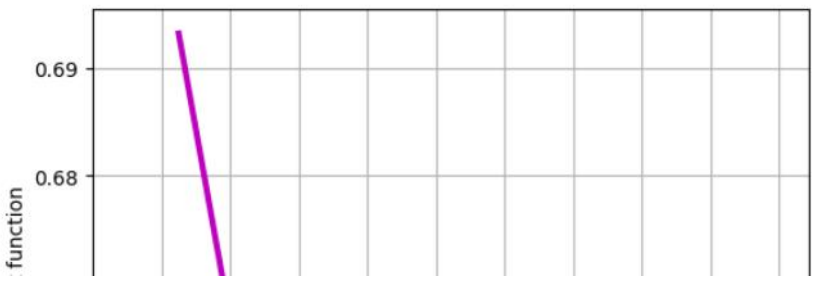
```
Training initialized with the following hyperparameter settings:
Learning rate: 0.4, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]
It: 2. Loss: 0.648770593 lower by 4.4e-02 than in previous iteration, Weight(s): [0.00734493 0.17950293], Gradient: [-0.0003102 -0.0450222]
It: 3. Loss: 0.64832405 lower by 4.5e-04 than in previous iteration, Weight(s): [0.00746901 0.1975118 ], Gradient: [ 0.0001189 -0.0045701]
It: 4. Loss: 0.648319444 lower by 4.6e-06 than in previous iteration, Weight(s): [0.00742147 0.19933984], Gradient: [ 2.720e-05 -4.651e-04]
It: 5. Loss: 0.648319397 lower by 4.8e-08 than in previous iteration, Weight(s): [0.00741059 0.1995259 ], Gradient: [ 4.30e-06 -4.74e-05]


Converged at iteration 8., where the Loss (0.648319396110995) was lower by 5.5e-14 than
in the prev it(0.64831939611105), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 8., Weight(s): [0.0074086  0.19954699], Loss: 0.6483193961109951 which was lower than in previous iteration
The lowest Loss 0.6483193961109951 was achieved in the 8. iterationwith the weight(s) [0.0074086  0.19954699]
```



```
Converged at iteration 8., where the Loss (0.648319396110995) was lower by 5.5e-14 than
in the prev it(0.64831939611105), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 8., Weight(s): [0.0074086  0.19954699], Loss: 0.6483193961109951 which was lower than in previous iteration
The lowest Loss 0.6483193961109951 was achieved in the 8. iterationwith the weight(s) [0.0074086  0.19954699]
```



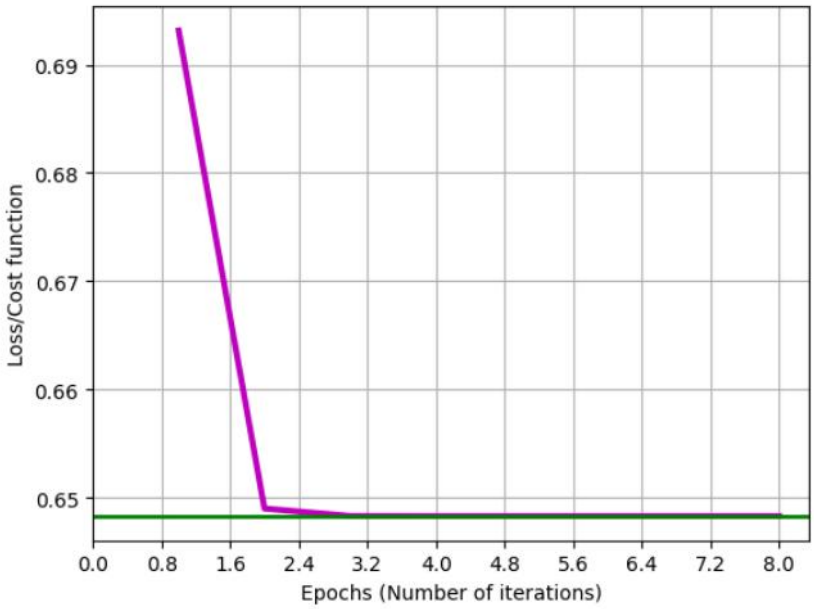

```
Accuracy on Train set: 0.96625
Accuracy on Test set: 0.962


Training initialized with the following hyperparameter settings:
Learning rate: 0.5, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]
It: 2. Loss: 0.64901555 lower by 4.4e-02 than in previous iteration, Weight(s): [0.00918117 0.22437866], Gradient: [0.0041885 0.0557662]
It: 3. Loss: 0.648329976 lower by 6.9e-04 than in previous iteration, Weight(s): [0.00708691 0.19649556], Gradient: [-0.0007484 -0.0068556]
It: 4. Loss: 0.648319558 lower by 1.0e-05 than in previous iteration, Weight(s): [0.00746111 0.19992336], Gradient: [0.0001211 0.0008456]
It: 5. Loss: 0.648319399 lower by 1.6e-07 than in previous iteration, Weight(s): [0.00740053 0.19950056], Gradient: [-1.850e-05 -1.044e-04]


Converged at iteration 9., where the Loss (0.648319396110995) was lower by 9.1e-15 than
in the prev it(0.648319396111004), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 9., Weight(s): [0.00740859 0.199547  ], Loss: 0.6483193961109949 which was lower than in previous iteration
The lowest Loss 0.6483193961109949 was achieved in the 9. iterationwith the weight(s) [0.00740859 0.199547  ]
```



It: 5. Loss: 0.648319399 lower by 1.6e-07 than in previous iteration, Weight(s): [0.00740053 0.19950056], Gradient: [-1.850e-05 -1.044e-04]

Converged at iteration 9., where the Loss (0.648319396110995) was lower by 9.1e-15 than
in the prev it(0.648319396111004), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 9., Weight(s): [0.00740859 0.199547], Loss: 0.6483193961109949 which was lower than in previous iteration
The lowest Loss 0.6483193961109949 was achieved in the 9. iterationwith the weight(s) [0.00740859 0.199547]



Accuracy on Train set: 0.9665
Accuracy on Test set: 0.9615

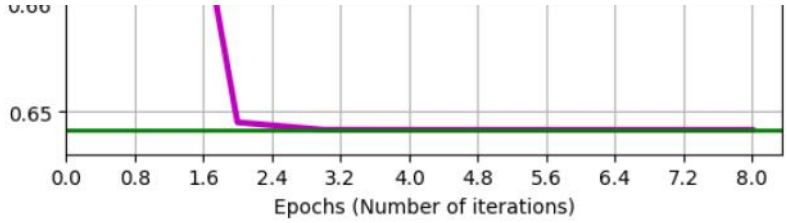
Training initialized with the following hyperparameter settings:
Learning rate: 0.8, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]
It: 2. Loss: 0.676906019 lower by 1.6e-02 than in previous iteration, Weight(s): [0.01468987 0.35900586], Gradient: [0.0176221 0.3574985]
It: 3. Loss: 0.666370645 lower by 1.1e-02 than in previous iteration, Weight(s): [0.00059219 0.07300707], Gradient: [-0.0164014 -0.2845005]
It: 4. Loss: 0.659829209 lower by 6.5e-03 than in previous iteration, Weight(s): [0.01371333 0.30060744], Gradient: [0.0149793 0.2267572]
It: 5. Loss: 0.655611274 lower by 4.2e-03 than in previous iteration, Weight(s): [0.00172986 0.11920165], Gradient: [-0.0134463 -0.1806002]

It: 10. Loss: 0.649081701 lower by 4.3e-04 than in previous iteration, Weight(s): [0.0103301 0.22542959], Gradient: [0.0067786 0.0581345]
It: 20. Loss: 0.648327882 lower by 4.8e-06 than in previous iteration, Weight(s): [0.00793952 0.20224237], Gradient: [0.0012156 0.0060573]
It: 30. Loss: 0.648319493 lower by 5.5e-08 than in previous iteration, Weight(s): [0.00748772 0.19982975], Gradient: [0.0001802 0.0006356]
It: 40. Loss: 0.648319397 lower by 6.3e-10 than in previous iteration, Weight(s): [0.00741936 0.19957687], Gradient: [2.44e-05 6.72e-05]
It: 50. Loss: 0.648319396 lower by 7.5e-12 than in previous iteration, Weight(s): [0.00740999 0.19955019], Gradient: [3.1e-06 7.1e-06]
It: 60. Loss: 0.648319396 lower by 9.1e-14 than in previous iteration, Weight(s): [0.00740877 0.19954735], Gradient: [4.e-07 8.e-07]

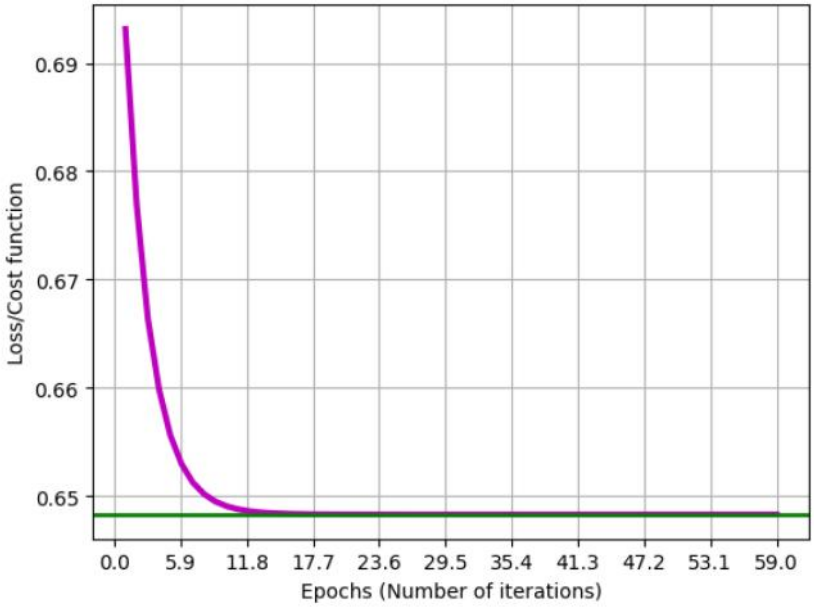
Converged at iteration 60., where the Loss (0.648319396111158) was lower by 9.1e-14 than
in the prev it(0.648319396111249), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 60., Weight(s): [0.00740877 0.19954735], Loss: 0.6483193961111584 which was lower than in previous iteration
The lowest Loss 0.6483193961111584 was achieved in the 60. iterationwith the weight(s) [0.00740877 0.19954735]



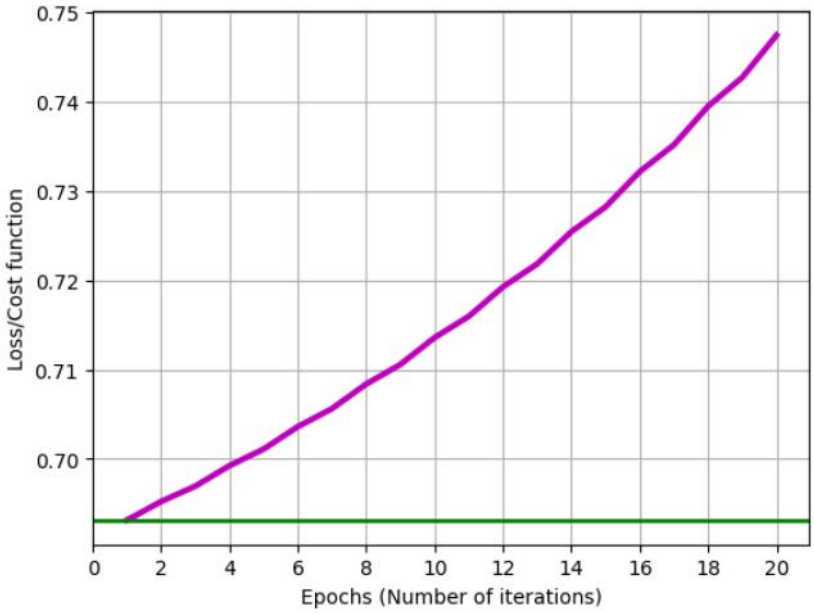

```
Accuracy on Train set: 0.9665
Accuracy on Test set: 0.9615
```

```
Training initialized with the following hyperparameter settings:
Learning rate: 0.8, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]
It: 2. Loss: 0.676906019 lower by 1.6e-02 than in previous iteration, Weight(s): [0.01468987 0.35900586], Gradient: [0.0176221 0.3574985]
It: 3. Loss: 0.666370645 lower by 1.1e-02 than in previous iteration, Weight(s): [0.00059219 0.07300707], Gradient: [-0.0164014 -0.2845005]
It: 4. Loss: 0.659829209 lower by 6.5e-03 than in previous iteration, Weight(s): [0.01371333 0.30060744], Gradient: [0.0149793 0.2267572]
It: 5. Loss: 0.655611274 lower by 4.2e-03 than in previous iteration, Weight(s): [0.00172986 0.11920165], Gradient: [-0.0134463 -0.1806002]

It: 10. Loss: 0.649081701 lower by 4.3e-04 than in previous iteration, Weight(s): [0.0103301  0.22542959], Gradient: [0.0067786 0.0581345]
It: 20. Loss: 0.648327882 lower by 4.8e-06 than in previous iteration, Weight(s): [0.00793952 0.20224237], Gradient: [0.0012156 0.0060573]
It: 30. Loss: 0.648319493 lower by 5.5e-08 than in previous iteration, Weight(s): [0.00748772 0.19982975], Gradient: [0.0001802 0.0006356]
It: 40. Loss: 0.648319397 lower by 6.3e-10 than in previous iteration, Weight(s): [0.00741936 0.19957687], Gradient: [2.44e-05 6.72e-05]
It: 50. Loss: 0.648319396 lower by 7.5e-12 than in previous iteration, Weight(s): [0.00740999 0.19955019], Gradient: [3.1e-06 7.1e-06]
It: 60. Loss: 0.648319396 lower by 9.1e-14 than in previous iteration, Weight(s): [0.00740877 0.19954735], Gradient: [4.e-07 8.e-07]

Converged at iteration 60., where the Loss (0.648319396111158) was lower by 9.1e-14 than
in the prev it(0.648319396111249), which is below the tolerance for convergence: 1.0e-13

Training finished with the following results:
Last ireation: 60., Weight(s): [0.00740877 0.19954735], Loss: 0.6483193961111584 which was lower than in previous iteration
The lowest Loss 0.6483193961111584 was achieved in the 60. iterationwith the weight(s) [0.00740877 0.19954735]
```




```
Accuracy on Train set: 0.96625
Accuracy on Test set: 0.962
```

```
Training initialized with the following hyperparameter settings:
Learning rate: 0.9, inverseC: 1.0, Maximum iterations: 100, Tolerance for convergence: 1e-13, Patience for early stop: 20,
Initial weight(s) randomly from the interval: [0, 0), Initial Loss: 0.693147180559945 with the [0. 0.] initial weight(s)

It: 1. Loss: 0.693147181, Weight(s): [0. 0.], Gradient: [-0.0183623 -0.4487573]
It: 2. Loss: 0.695236876 higher by 2.1e-03 than in previous iteration, Weight(s): [0.0165261  0.40388159], Gradient: [0.022074  0.4578133]
It: 3. Loss: 0.696953572 higher by 1.7e-03 than in previous iteration, Weight(s): [-0.00334049 -0.00815036], Gradient: [-0.0259487 -0.4671244]
It: 4. Loss: 0.699242572 higher by 2.3e-03 than in previous iteration, Weight(s): [0.0200133  0.41226159], Gradient: [0.0299457 0.476552 ]
It: 5. Loss: 0.701106195 higher by 1.9e-03 than in previous iteration, Weight(s): [-0.00693784 -0.01663517], Gradient: [-0.0341157 -0.486246 ]

It: 10. Loss: 0.713583516 higher by 3.0e-03 than in previous iteration, Weight(s): [0.03211921 0.43950658], Gradient: [0.0572353 0.5374363]
It: 20. Loss: 0.747414831 higher by 4.7e-03 than in previous iteration, Weight(s): [0.05848048 0.49238784], Gradient: [0.1164904 0.6554187]

Early stopped at iteration 21., where the patience limit (20) was reached
The lowest Loss 0.693147180559945) was achieved in the 1. iteration with the weight(s) [0. 0.], and after that there were 20 iterations where th

Training finished with the following results:
Last ireation: 21., Weight(s): [-0.04636087 -0.09748896], Loss: 0.7508948125722171 which was higher than in previous iteration
The lowest Loss 0.6931471805599454 was achieved in the 1. iterationwith the weight(s) [0. 0.]
```




```
Accuracy on Train set: 0.499875
Accuracy on Test set: 0.504
```

```python
# Comparing performance to sklearn's model

X, y = datasets.make_classification(n_samples=350, n_features=2, n_redundant = 0, n_classes = 2, random_state=11)
# Note: to display multiple features the plots below are not properly set

num_samples, num_features = X.shape
print("Number of generated samples:", num_samples, "Number of generated features:", num_features, "\n")


# Dataset plotting
plt.figure(figsize=(6,5))
```

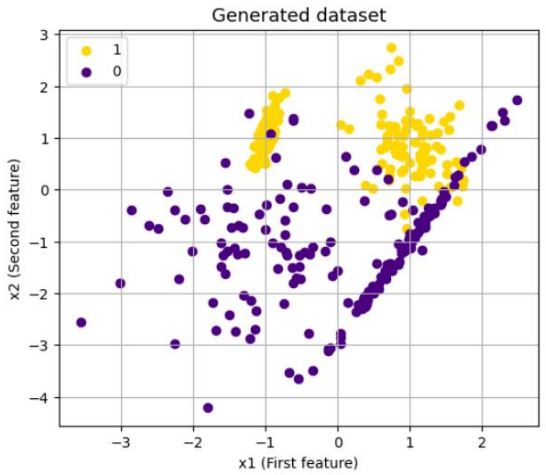
```
# Comparing performance to sklearn's model

X, y = datasets.make_classification(n_samples=350, n_features=2, n_redundant = 0, n_classes = 2, random_state=11)
# Note: to display multiple features the plots below are not properly set

num_samples, num_features = X.shape
print("Number of generated samples:", num_samples, "Number of generated features:", num_features, "\n")


# Dataset plotting
plt.figure(figsize=(6,5))
plt.grid()
plt.title('Generated dataset', fontsize = 13)
plt.xlabel('x1 (First feature)')
plt.ylabel('x2 (Second feature)')
plt.scatter(X[y==1][:, 0], X[y==1][:, 1], c = "gold", label="1")
plt.scatter(X[y==0][:, 0], X[y==0][:, 1], c = "indigo", label="0")
plt.legend(loc="upper left")
plt.show()

# Dataset splitting
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)
print("After splitting the dataset:")
print("Shape of x_train:", X_train.shape, "Shape of y_train:", y_train.shape, "Shape of x_test:", X_test.shape, "Shape of y_test:", y_test.shape



print("\n\n\n\n")
print("---------------------------------- Created LogRegNumPy without training "
      "----------------------------------\n")
# Performance of NumPy model without training (0 as initial=final value(s) for the weight(s))

regressorNoTrain = LogRegNumPy(max_iter=0,
                               initialweights_lowerlimit = 0,
                               initialweights_upperlimit = 0)

fittedRegressorNoTrain = regressorNoTrain.fit(X_train, y_train)
y_pred = fittedRegressorNoTrain.predict(X_test)
y_TrainPred = fittedRegressorNoTrain.predict(X_train)

TrainAccuracy_LogRegNumPyNoTrain = accuracy_score(y_TrainPred, y_train)
TestAccuracy_LogRegNumPyNoTrain = accuracy_score(y_pred, y_test)
print("Number of wrong predictions on the test set:", y_pred[y_pred!=y_test].shape[0])
print()
print("Ratio of y=1 in y_train:", len(y_train[y_train==1])/(len(y_train)))
print("Ratio of y=1 in y_test:", len(y_test[y_test==1])/len(y_test))

print("\nAs the weight(s) are initialized to zero(s) and there is no training, weight(s) = 0 is/are used to make predictions")
print("With zero weights(s) the Sigmoid function returns exactly 0.5, thus all predictions are y = 1")

fig, ax = plt.subplots()
fig.set_figheight(4)
fig.set_figwidth(15)
ax.set_facecolor("lightgray")

ax.scatter(X_test[y_pred==y_test][:,0], y_pred[y_pred==y_test], c = 'green', label = 'Correct predictions on the test set')
ax.scatter(X_test[y_pred!=y_test][:,0], y_pred[y_pred!=y_test], c = 'red', label = 'Wrong predictions on the test set')

ax.set_title('LogRegNumPy without training', fontsize = 15)
ax.legend(loc = 'center left', fontsize = '11')
ax.set_yticks(np.linspace(0, 1, 2))
ax.set_xlabel('x1 (First feature)', fontsize = 13)
ax.set_ylabel('Predictions on the test set', fontsize = 13)
ax.text(1.43, 0.5, f'Train accuracy: {round(TrainAccuracy_LogRegNumPyNoTrain,5)}\n Test accuracy: {round(TestAccuracy_LogRegNumPyNoTrain,5)}', 
plt.show()

# pred_0 = X_test[y_pred==0]
# pred_0_set = set([tuple(x) for x in A])
```

```
ax.set_ylabel('Predictions on the test set', fontsize = 13)
ax.text(1.43, 0.5, f'Train accuracy: {round(TrainAccuracy_LogRegNumPyNoTrain,5)}\n Test accuracy: {round(TestAccuracy_LogRegNumPyNoTrain,5)}', fontsize = '11.5', ha='left', va='center')
plt.show()

# pred_0 = X_test[y_pred==0]
# pred_0_set = set([tuple(x) for x in A])

# correct_pred = X_test[y_pred == y_test]
# correct_pred_set = set([tuple(x) for x in correct_pred ])

# pred_0_correct = np.array([x for x in pred_0_set & correct_pred_set])
# ....

plt.figure(figsize=(6,5))
plt.grid()
plt.title('LogRegNumPy without training', fontsize = 13)
plt.xlabel('x1 (First feature)')
plt.ylabel('x2 (Second feature)')
plt.scatter(X_test[y_pred==y_test][:,0], X_test[y_pred==y_test][:,1], c = 'green', label = 'Correct predictions on test set')
plt.scatter(X_test[y_pred!=y_test][:,0], X_test[y_pred!=y_test][:,1], c = 'red', label = 'Wrong predictions on test set')
plt.legend(loc="lower right", fontsize = "small")
plt.show()

print("\n\n\n\n")
print("---------------------------------- Created LogRegNumPy with training "
      "-----------------------------------\n")
# Performance of LogRegNumPy with training
# Set LogRegNumPy regressor's parameters to sklearn regressor's default values for the parameters
regressorSklearn = sklearn.linear_model.LogisticRegression()
defaultSkParams = regressorSklearn.__dict__

# For a proper comparison with sklearn's built-in model the weights are initialized to 0 here (for the created) class
# as well
# With weights initialized between 0 and 1 sometimes (depends on the final weights set at the end of the training)
# we get better results: 0.92857 Test Accuracy
regressor = LogRegNumPy(penalty = defaultSkParams['penalty'],
                        lambda_or_inverseC = defaultSkParams['C'],
                        max_iter = defaultSkParams['max_iter'],
                        tol = defaultSkParams['tol'],
                        initialweights_lowerlimit = 0,
                        initialweights_upperlimit = 0)


fittedRegressor = regressor.fit(X_train, y_train)
y_pred = fittedRegressor.predict(X_test)
y_TrainPred = fittedRegressor.predict(X_train)

TrainAccuracy_LogRegNumPy = accuracy_score(y_TrainPred, y_train)
TestAccuracy_LogRegNumPy = accuracy_score(y_pred, y_test)
print("Number of wrong predictions on the test set:", y_pred[y_pred!=y_test].shape[0])

fig, ax = plt.subplots()
fig.set_figheight(4)
fig.set_figwidth(15)
ax.set_facecolor("lightgray")

ax.scatter(X_test[y_pred==y_test][:,0], y_pred[y_pred==y_test], c = 'green', label = 'Correct predictions on the test set')
ax.scatter(X_test[y_pred!=y_test][:,0], y_pred[y_pred!=y_test], c = 'red', label = 'Wrong predictions on the test set')

# Why the approach below  (mean of features on the X-axis) can sometimes be problematic:
# when for two (or more) samples the mean of the features are close to each other, then these samples overlap
# so much on the X-axis that we see both of them as one point in the scatter plot:
# print(X_test[:,0].std())
# # vs
# print(np.mean(X_test, axis = 1).std())
# Normalization of X_test happened when the regressor was fitted, so we can take the mean of different features
# ax.scatter(np.mean(X_test[y_pred==y_test], axis = 1), y_pred[y_pred==y_test], c = 'green', label = "Correct")
# ax.scatter(np.mean(X_test[y_pred!=y_test], axis = 1), y_pred[y_pred!=y_test], c = 'red', label = "Wrong")
# print(np.sort(np.mean(X_test[y_pred!=y_test], axis = 1)))
```

```
# so much on the X-axis that we see both of them as one point in the scatter plot:
# print(X_test[:,0].std())
# # vs
# print(np.mean(X_test, axis = 1).std())
# Normalization of X_test happened when the regressor was fitted, so we can take the mean of different features
# ax.scatter(np.mean(X_test[y_pred==y_test], axis = 1), y_pred[y_pred==y_test], c = 'green', label = "Correct")
# ax.scatter(np.mean(X_test[y_pred!=y_test], axis = 1), y_pred[y_pred!=y_test], c = 'red', label = "Wrong")
# print(np.sort(np.mean(X_test[y_pred!=y_test], axis = 1)))

# Problem with the approach below: As both correctly and incorrectly predicted samples are plotted
# at the same time, incorrect predictions won't be in the foreground and thus sometimes they can't be
# seen because of the much more correctly predicted green data points hiding the red points
# from matplotlib.colors import ListedColormap
# ax.scatter(X_test[:,0], y_pred, c = (y_pred == y_test), cmap=listedColormap(['red','green']))
# import matplotlib.patches as mpatches
# NotCorrectPatch = mpatches.Patch(color='red', label='Wrong test set predictions')
# CorrectPatch = mpatches.Patch(color='green', label='Correct test set predictions')
# ax.legend(handles=[CorrectPatch, NotCorrectPatch], loc='center left', frameon=False)
# print(np.sort(X_test[y_pred!=y_test][:,0]))

ax.set_title('LogRegNumPy with training', fontsize = 15)
ax.legend(loc = 'center left', fontsize = '11')
ax.set_yticks(np.linspace(0, 1, 2))
ax.set_xlabel('x1 (First feature)', fontsize = 13)
ax.set_ylabel('Predictions on the test set', fontsize = 13)
ax.text(1.43, 0.5, f'Train accuracy: {round(TrainAccuracy_LogRegNumPy,5)}\n Test accuracy: {round(TestAccuracy_LogRegNumPy,5)}', fontsize = '11.5', ha='left', va='center')
plt.show()

plt.figure(figsize=(6,5))
plt.grid()
plt.title('LogRegNumPy with training', fontsize = 13)
plt.xlabel('x1 (First feature)')
plt.ylabel('x2 (Second feature)')
plt.scatter(X_test[y_pred==y_test][:,0], X_test[y_pred==y_test][:,1], c = 'green', label = 'Correct predictions on test set')
plt.scatter(X_test[y_pred!=y_test][:,0], X_test[y_pred!=y_test][:,1], c = 'red', label = 'Wrong predictions on test set')
plt.legend(loc="lower right", fontsize = "small")
plt.show()


print("\n\n\n\n")
print("---------------------------------- Scikit-Learn's built-in model "
      "----------------------------------\n")
# Performance of Scikit-Learn's built-in model (with training)
    # Changing the solver (optimization method of the loss function) to Stochastic Average Gradient (SAG), as
    # this is the most similar method to Gradient Descent implemented in the LogRegNumPy class
    # Stochastic Gradient Descent (SGD) is a variation of GD that updates the model parameters based on the gradient of the
    # cost function for a single training example, rather than the average gradient over the entire training set
    # Stochastic naming: in each training step the gradient is only an approximation of the actual gradient as it is
    # calculated based on only one training sample
    # Stochastic Average Gradient (SAG) updates the parameters based on the average of the gradients for each data point

    # Based on ChatGPT (nowhere else I could find information about this) the initial weight(s) for
    # the SAG solver are either zero(s) or (a) close to zero value(s)
    # This leads to proper comparison as for the created class zero weights were initialized as well

    # fit_intercept set to False as with the created class no intercept is used either
regressorSklearn = sklearn.linear_model.LogisticRegression(solver = 'sag',
                                                           fit_intercept = False,
                                                           n_jobs = 1,
                                                           verbose = True)

y_pred = regressorSklearn.fit(X_train, y_train).predict(X_test)
y_TrainPred = regressorSklearn.fit(X_train, y_train).predict(X_train)

TrainAccuracy_LogRegSklearn = accuracy_score(y_TrainPred, y_train)
TestAccuracy_LogRegSklearn = accuracy_score(y_pred, y_test)
print("\nNumber of wrong predictions on the test set:", y_pred[y_pred!=y_test].shape[0])

fig, ax = plt.subplots()
fig.set_figheight(4)
```

```
y_TrainPred = regressorSklearn.fit(X_train, y_train).predict(X_train)

TrainAccuracy_LogRegSklearn = accuracy_score(y_TrainPred, y_train)
TestAccuracy_LogRegSklearn = accuracy_score(y_pred, y_test)
print("\nNumber of wrong predictions on the test set:", y_pred[y_pred!=y_test].shape[0])

fig, ax = plt.subplots()
fig.set_figheight(4)
fig.set_figwidth(15)
ax.set_facecolor("lightgray")

ax.scatter(X_test[y_pred==y_test][:,0], y_pred[y_pred==y_test], c = 'green', label = 'Correct predictions on the test set')
ax.scatter(X_test[y_pred!=y_test][:,0], y_pred[y_pred!=y_test], c = 'red', label = 'Wrong predictions on the test set')

ax.set_title('LogisticRegression from sklearn', fontsize = 15)
ax.legend(loc = 'center left', fontsize = '11')
ax.set_yticks(np.linspace(0, 1, 2))
ax.set_xlabel('x1 (First feature)', fontsize = 13)
ax.set_ylabel('Predictions on the test set', fontsize = 13)
ax.text(1.18, 0.5, f'            Train accuracy: {round(TrainAccuracy_LogRegSklearn,5)}\nTrain built-in accuracy: {round(regressorSklearn.score(X_train, y_train), 5)}\n            Test accuracy: {round(TestAccuracy_LogRegSklearn,5)}\n  Test built-in accuracy: {round(regressorSklearn.scor
plt.show

plt.figure(figsize=(6,5))
plt.grid()
plt.title('LogisticRegression from sklearn', fontsize = 13)
plt.xlabel('x1 (First feature)')
plt.ylabel('x2 (Second feature)')
plt.scatter(X_test[y_pred==y_test][:,0], X_test[y_pred==y_test][:,1], c = 'green', label = 'Correct predictions on test set')
plt.scatter(X_test[y_pred!=y_test][:,0], X_test[y_pred!=y_test][:,1], c = 'red', label = 'Wrong predictions on test set')
plt.legend(loc="lower right", fontsize = "small")
plt.show()
```

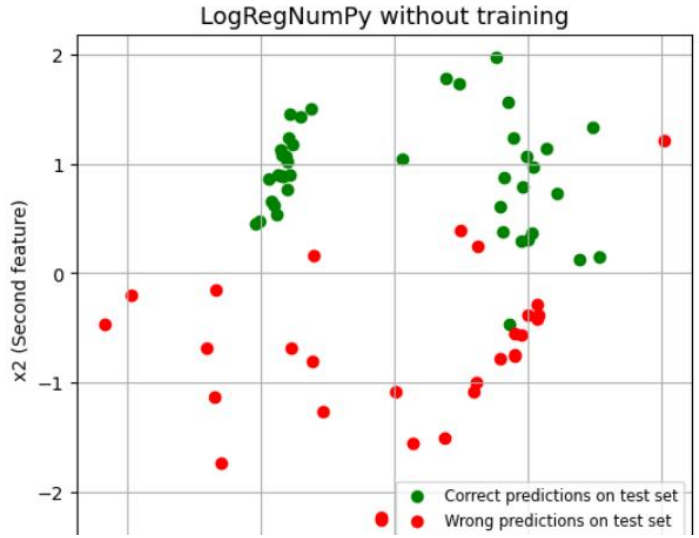
```
Number of generated samples: 350 Number of generated features: 2
```

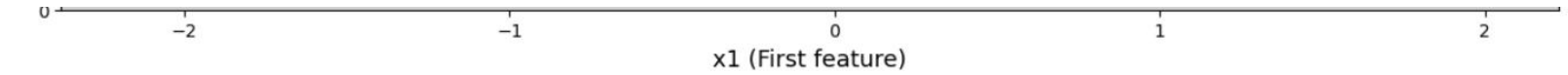


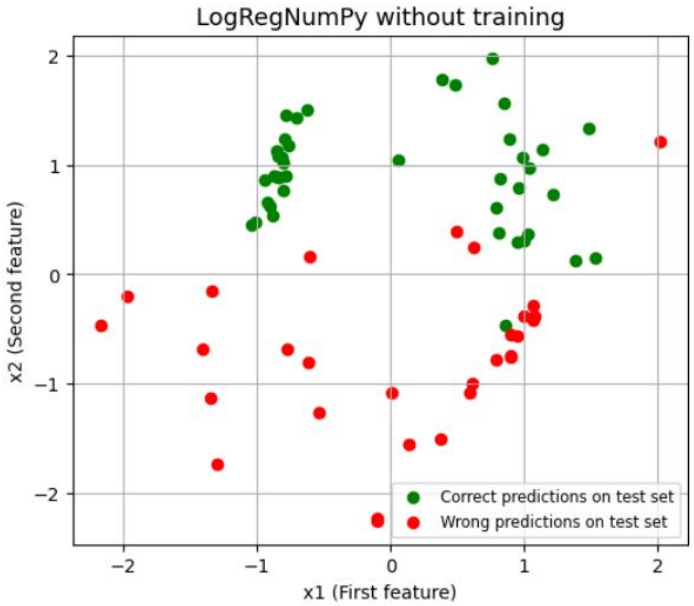

```
After splitting the dataset:
Shape of x_train: (280, 2) Shape of y_train: (280,) Shape of x_test: (70, 2) Shape of y_test: (70,)
```

x1 (First feature)

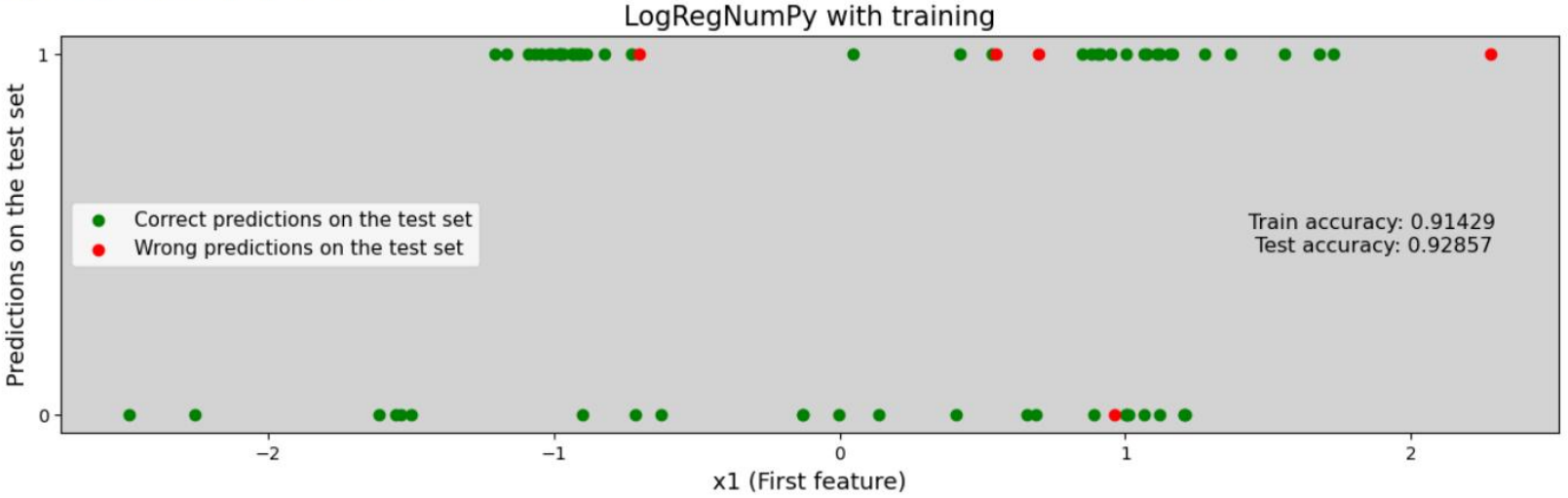
```
After splitting the dataset:
Shape of x_train: (280, 2) Shape of y_train: (280,) Shape of x_test: (70, 2) Shape of y_test: (70,)



---------------------------------- Created LogRegNumPy without training ----------------------------------

There was no training performed as maximum iterations for the training are set to 0

Number of wrong predictions on the test set: 29

Ratio of y=1 in y_train: 0.4785714285714286
Ratio of y=1 in y_test: 0.5857142857142857

As the weight(s) are initialized to zero(s) and there is no training, weight(s) = 0 is/are used to make predictions
With zero weights(s) the Sigmoid function returns exactly 0.5, thus all predictions are y = 1
```



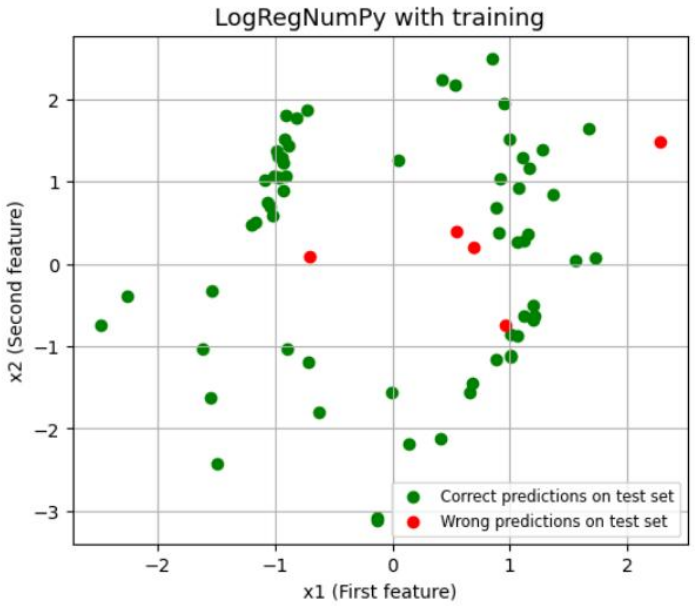




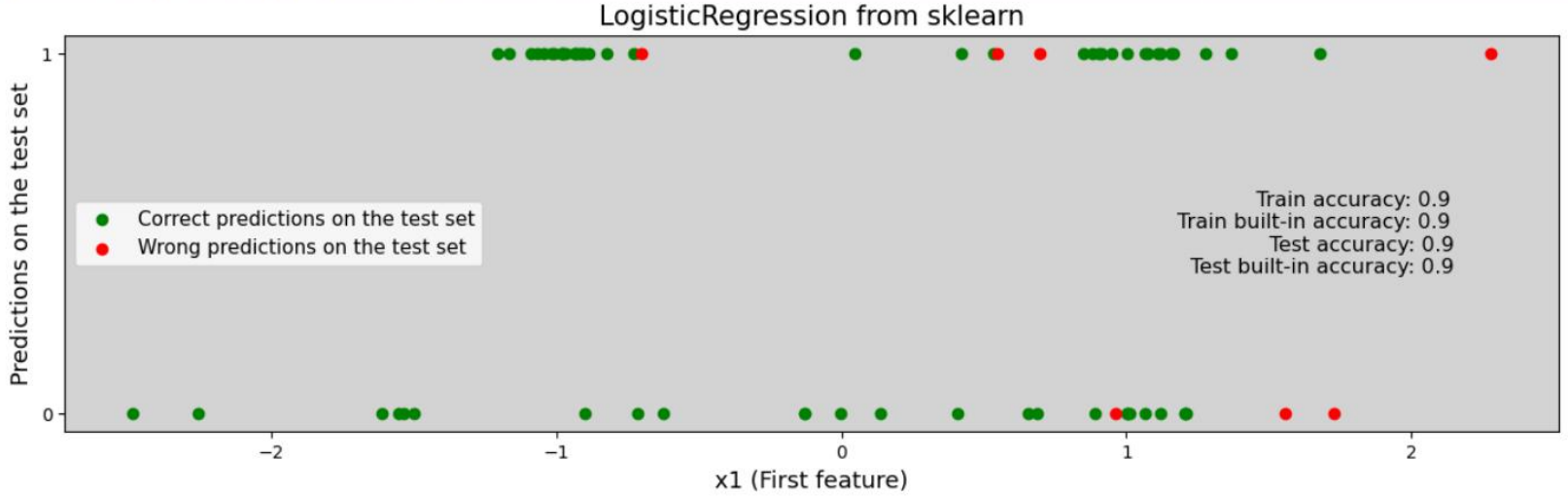

```
---------------------------------- Created LogRegNumPy with training ----------------------------------

Training initialized with 100 maximum possible iterations

Converged at iteration 12, where the drop in Loss was below the tolerance set for convergence

Training finished after 12 iterations

Number of wrong predictions on the test set: 5
```

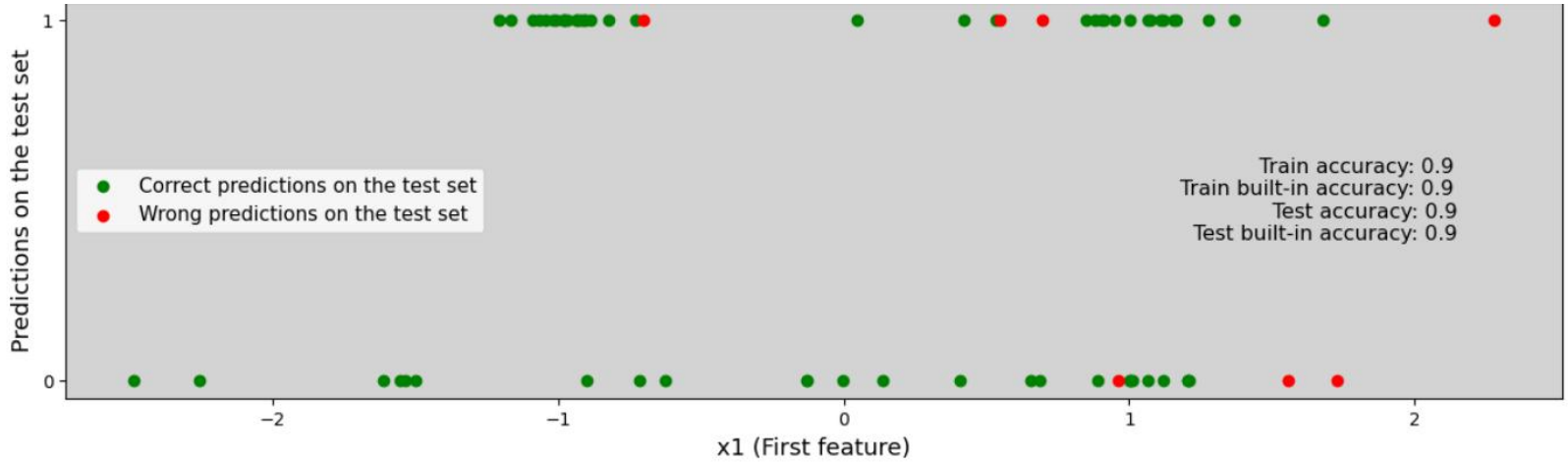


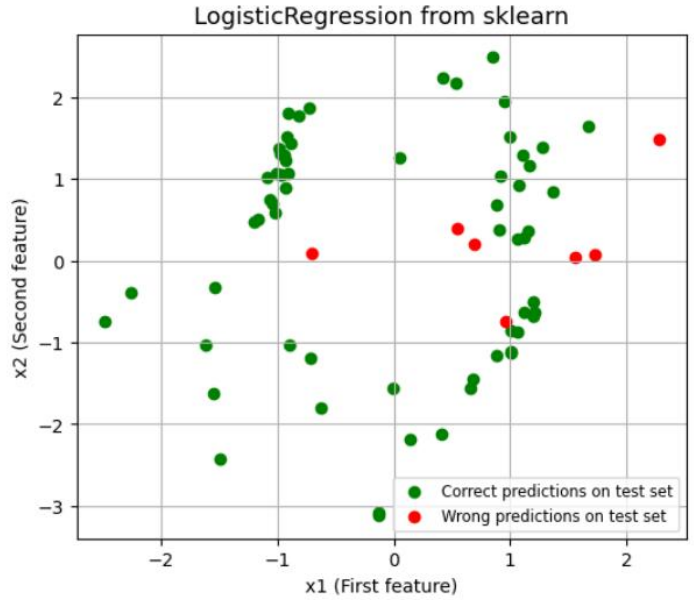
x1 (First feature)




```
---------------------------------- Scikit-Learn's built-in model ----------------------------------

convergence after 19 epochs took 0 seconds
convergence after 22 epochs took 0 seconds

Number of wrong predictions on the test set: 7
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s finished
```








```
# Further considerations and todos

# Use functions instead of many duplicated lines of code in cells where possible

# local minimumba érést összehozni (nagyon nagy kezdő súlyok -> sok és optimálistól messze kezdődő iteracio ami alatt egyszer lehet local minimumba ér)
# és ez a local minimumba érés jelenlen meg ploton is, lehet h sok feature (suly) kell hozza hogy elég komplex
# legyen a loss function hogy legyen local minimuma is

# nyilak javítása a plotokon

# Mindegyik reviewed notebookbol mindent ami más mint itt átrakni
```

Evaluation metrics

Creating and applying class to a classification task