Linear Regression Algorithm

Assignment 1: Linear Regression

ECGR-4105-001

Robert Thomas

801371500

Feb 25, 2025

Project GitHub Repository

Table of Contents

Instructions for Running Program

This script can run both part 1 and part 2 for the project. Parameters can be specified using command-line arguments.

Command-Line Arguments

> -a <alpha> : Sets the learning rate (default: 0.05).
> -i <iterations> : Sets the number of iterations for gradient descent (default: 1000).
> -f <fileName> : Specifies the input CSV file (default: "D3.csv").
> -o <outputFile> : Specifies the output file name for plots (default: auto-generated).
> -p <projectPart> : Chooses the project to run (1 for Project 1, 2 for Project 2).
> --help : Displays help information.

Example Usage

Run Project 1 with default settings:

```
python3 linear_regression_main.py -p 1
```

Run Project 2 with a custom learning rate and iterations:

```
python3 linear_regression_main.py -p 2 -a 0.01 -i 5000
```

Display help information:

```
python3 linear_regression_main.py --help
```

Theory and Background Information

Linear regression is a mathematical technique that finds a relationship between variables to predict the value of an unknown variable. It is often used in machine learning to make predictions based on past data.

There are four parts of the linear regression formula: the hypothesis function, the cost function, the gradient of the cost function, and updating the gradient descent parameters.

The Hypothesis Function:

The hypothesis function, or the linear model, is the prediction function for linear regression. The goal is to find the best possible coefficients (θ) given the data input arrays *x* that causes the function to best fit the model.

$$h_\Theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Equation 1: The Hypothesis Function

As each value for θ is adjusted, the function shifts and scales such that it better fits the data points. This is determined by the cost function.

The Cost Function:

The cost function, or mean-squared function, is used to determine the error between the hypothesis function and the actual data points. The goal is to keep the error low while also not collecting the noise in the function, also called overfitting. We want the trend, not the noise.

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

Equation 2: The Cost Function

```
error = h_theta - y  # distance between h and y for error correction
```

First we'll get the error for the current value of theta to be used in the cost function.

```
cost = (1/(2*len(y))) * np.sum((h_theta - y)**2)
cost_history.append(cost)
```

Then we'll add the cost to an array. This will enable us to calculate the gradient.

Gradient of the Cost Function:

The gradient of the cost function is the derivative for each parameter (θ_j):

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

Equation 3: Cost Function Gradient

The gradient indicates how the cost function changes with (θ_j).

Updating Gradient Descent Parameters:

This rule is used for updating each parameter (θ_j). It is influenced by the learning rate (α) which determines how much the value will increase or decreas with each iteration.

$$\theta_{j^{new}} = \theta_{j^{old}} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

Equation 4: Gradient Descent Update Function

We'll calculate the gradient with the array we made for the cost function:

```
theta = theta - 
(learning_rate/len(y)) * 
np.dot(x.T, error)
```

We can then store the cost function history for debugging later:

```
cost_df = 
pd.DataFrame({"Iteration": 
np.arange(1, iterations + 1), 
"Cost": cost_history})
cost_df.to_csv('cost_history.csv', 
index=False)
```

Therefore the total function will be:

```
def gradient_descent(x, y, theta, 
learning_rate, iterations):
    cost_history=[]

    for i in range(iterations):
        h_theta = np.dot(x, theta)  
# dot product of x and theta
        error = h_theta - y  # 
distance between h and y for error 
correction

        cost = (1/(2*len(y))) * 
np.sum((h_theta - y)**2)
        cost_history.append(cost)
        #get new theta
        theta = theta - 
(learning_rate/len(y)) * 
np.dot(x.T, error)

    #storing cost history to a 
.txt file for debugging
    cost_df = 
pd.DataFrame({"Iteration": 
np.arange(1, iterations + 1), 
"Cost": cost_history})

cost_df.to_csv('cost_history.csv', 
index=False)

    return theta, cost_history
```

Note that two values are returned, the new final value for (θ) and an array for the cost history.

Making Predictions:

The new values for (θ) are entered into the hypothesis function (eq. 1) which can then be used to make predictions about future values when plugging in new data for the x values.

The function to predict new values based on inputs for x values is given as follows:

```
def predict(theta, X_new):
    X_new = np.insert(X_new, 0, 1)
    y_pred = np.dot(theta, X_new)
    return y_pred
```

Full Regression Algorithm:

The full regression algorithm can be placed into its own python script:

```
import pandas as pd
import numpy as np

def gradient_descent(x, y, theta, 
learning_rate, iterations):
    cost_history=[]

    for i in range(iterations):
        h_theta = np.dot(x, theta)  
# dot product of x and theta
```

```
        error = h_theta - y  #
distance between h and y for error
correction

        cost = (1/(2*len(y))) *
np.sum((h_theta - y)**2)
        cost_history.append(cost)
        #get new theta
        theta = theta -
(learning_rate/len(y)) *
np.dot(x.T, error)

    #storing cost history to a
.txt file for debugging
    cost_df =
pd.DataFrame({"Iteration":
np.arange(1, iterations + 1),
"Cost": cost_history})

cost_df.to_csv('cost_history.csv',
index=False)

    return theta, cost_history

def predict(theta, X_new):
    X_new = np.insert(X_new, 0, 1)
    y_pred = np.dot(theta, X_new)
    return y_pred
```

Cleaning Data

Before running the data through the linear regression algorithm, it must be cleaned to ensure that it doesn't have issues being processed. In the given dataset, there are empty cells that the algorithm cannot process. Thus, the empty cells should be replaced with values that accurately reflect the rest of the dataset. Empty cells can be replaced with the median values of their columns to avoid altering the data too much.

We can add a Python script to clean the data:

```
import pandas as pd
import numpy as np

def clean_empty(df: pd.DataFrame)
-> pd.DataFrame:
    # Replaces empty cells with
the average value of the column
    df = df.apply(pd.to_numeric,
errors='coerce')
    df.fillna(df.mean(),
inplace=True)
    return df
```

Part 1

Project description:

Develop a code that runs linear regression with a gradient decent algorithm for each explanatory variable in isolation. In this case, you assume that in each iteration, only one explanatory variable (either x1, x2, or x3) is explaining the output. You need to do three different training, one per each explanatory variable. For the learning rate, explore different values between 0.1 and 0.01 (your choice). Initialize your parameters to zero (theta to zero).

1. *Report the linear model you found for each explanatory variable.*
2. *Plot the final regression model and loss over the iteration per each explanatory variable.*
3. *Which explanatory variable has the lower loss (cost) for explaining the output (Y)?*
4. *Based on your training observations, describe the impact of the different learning rates on the final loss and number of training iterations.*

For this we will iterate through each x value, run the regression algorithm, and use that to make the hypothesis function.

Report the linear model you found for each explanatory variable.

```
theta_final, cost_history =
gradient_descent(X, y,
theta_initial, alpha, iterations)
#for each
```

```
print("final theta values:",
theta_final) #at the end
```

The code will optimize (θ) for each explanatory variable (x).

Plot the final regression model and loss over the iteration per each explanatory variable.

We can use Matplot for this:

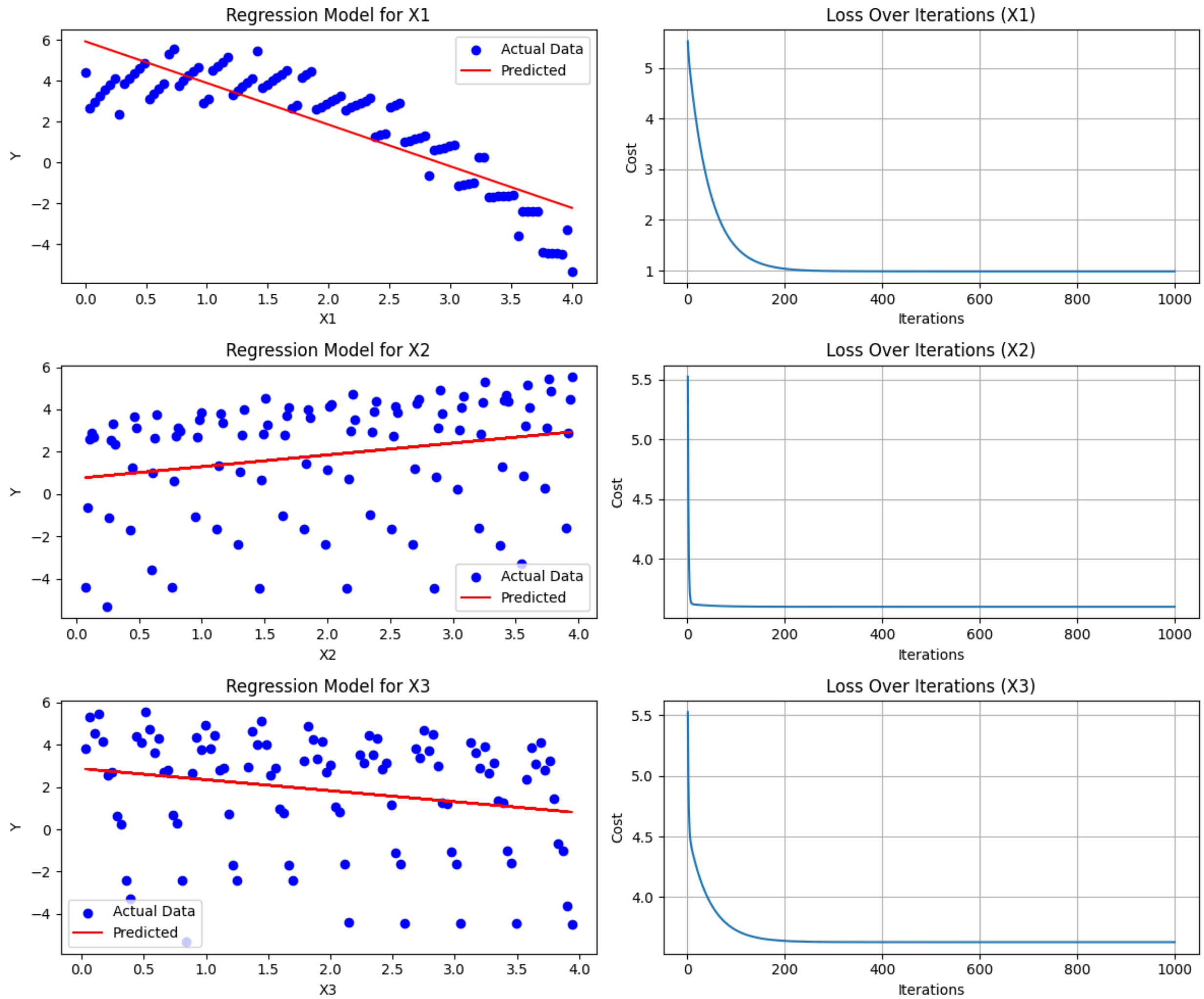
```
plt.subplot(3, 2, 2 * i + 1)
plt.scatter(df[var], y,
color='blue', label="Actual Data")
plt.plot(df[var], np.dot(X,
theta_final), color='red',
label="Predicted")
plt.xlabel(var)
plt.ylabel("Y")
plt.title(f"Regression Model for
{var}")
plt.legend()
```

This will give us the regression model for each variable.

```
plt.subplot(3, 2, 2 * i + 2)
plt.plot(range(1, iterations + 1),
cost_history, linestyle='-')
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.title(f"Loss Over Iterations
({var})")
plt.grid(True)
```

This will give us the loss vs. iterations plot.

The output, when α is 0.05, is plotted as follows:



Plot 1: Project part 1

Which explanatory variable has the lower loss (cost) for explaining the output (Y)?

We can find the best value by iterating over all the gradients and finding the lowest x value:

```
for i, var in enumerate(X_vars):
    theta_initial =
np.zeros(X.shape[1])
    theta_final, cost_history =
gradient_descent(X, y,
theta_initial, alpha, iterations)

    final_losses[var] =
cost_history[-1]  #store final
loss as array
```

```
best_var = min(final_losses,
key=final_losses.get)
```

The best_var will be the lowest value of x. It ends up being a value for X1 which is 3.6.

Based on your training observations, describe the impact of the different learning rates on the final loss and number of training iterations.

A very high learning rate will cause the algorithm to converge faster but it may oscillate too much. A very low learning rate will be more stable but may be too slow and take up too much processing power. A learning rate of around 0.05 or so seems to be the best option.

Part 2

Project description:

This time, run linear regression with gradient descent algorithm using all three explanatory variables. For the learning rate, explore different values between 0.1 and 0.01 (your choice). Initialize your parameters (theta to zero).

1. *Report the final linear model you found the best.*
2. *Plot loss over the iteration.*
3. *Based on your training observations, describe the impact of the different learning rates on the final loss and number of training iterations.*
4. *Predict the value of y for new (x1, x2, x3) values: (1, 1, 1), (2, 0, 4), and (3, 2, 1)*

For the second part of the project the linear regression algorithm is applied to all three explanatory values at the same time. The model will also explore different learning rates and try to find the best one.

```
results = {}
cost_values = []
cost_histories = {}

for lr in learning_rates:
    theta_final, cost_history =
gradient_descent(X, y,
theta_initial, lr, iterations)
    results[lr] = theta_final

cost_values.append(cost_history[-1
])
    cost_histories[lr] =
cost_history
```

```
best_lr =
learning_rates[np.argmin(cost_valu
es)]
best_cost_history =
cost_histories[best_lr]
```

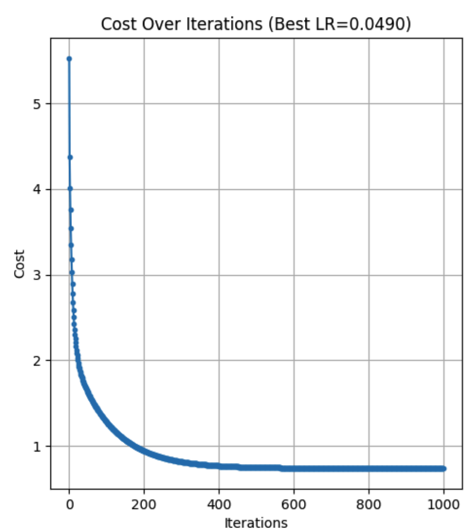
The final values for (θ) represent the best-fit linear model. The best learning rate is determined by the lowest cost.

Plot Loss Over Iterations

Using Matplot:

```
plt.subplot(1, 2, 2)
plt.plot(range(1, iterations + 1),
best_cost_history, marker='.',
linestyle='-')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title(f'Cost Over Iterations
(Best LR={best_lr:.4f})')
plt.grid(True)
```

The script plots the relationship between the cost and the number of iterations. It shows that the loss decreases as the gradient descent optimizes the model.



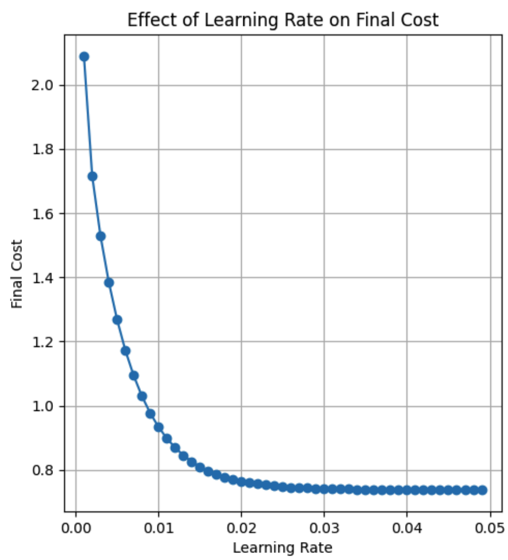
Plot 2: Project Part 2, Cost vs Iterations

Impact of Learning Rates on Final Loss and Iterations

Using Matplot:

```
plt.subplot(1, 2, 1)
plt.plot(learning_rates,
cost_values, marker='o',
linestyle='-')
plt.xlabel('Learning Rate')
plt.ylabel('Final Cost')
plt.title('Effect of Learning Rate
on Final Cost')
plt.grid(True)
```

Higher learning rates may lead to instability (higher loss or failure to converge). Lower learning rates may converge more reliably but take more iterations. The best learning rate is the one with the lowest final cost.



Plot 3: Project Part 2, Cost vs Learning Rate

Predicting Y for New (X1, X2, X3) Values

Given the values: (1, 1, 1), (2, 0, 4), and (3, 2, 1), I made a prediction function that accepts new explanatory values and gives the predicted output using the hypothesis function.

```
def predict(theta, X_new):
    X_new = np.insert(X_new, 0, 1)
    y_pred = np.dot(theta, X_new)
    return y_pred
```

Conclusion

Linear regression can be used to make predictions of the future based on past data.

This project implemented linear regression using gradient descent to train models on a dataset with multiple explanatory variables. In Part 1, individual models were trained for each variable, revealing that X1 had the lowest final loss, making it the strongest predictor of Y. In Part 2, all explanatory variables were used together, improving the model's predictive accuracy.

Experimenting with different learning rates showed that a moderate value (around 0.05) provided the best balance between convergence speed and stability. Overall, this project reinforced the importance of selecting appropriate features and hyperparameters in machine learning mode

All Code Sources

linear_regression_main.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sys

from cleaning import clean_empty
from regression import gradient_descent
from proj1 import runProj1
from proj2 import runProj2

fileName="D3.csv"
alpha=0.05
iterations = 1000
outputFileName = ''
whichProj=0
def help():
    with open("help.txt", "r") as file:
        print(file.read())
    return

if len(sys.argv) > 1:
    for i in range(1, len(sys.argv)):
        if sys.argv[i] == "-a":
            if i + 1 < len(sys.argv):
                alpha=float(sys.argv[i+1])
        elif sys.argv[i] == "-i":
            if i + 1 < len(sys.argv):
                iterations = int(sys.argv[i+1])
        elif sys.argv[i] == "-f":
            if i + 1 < len(sys.argv):
                fileName=sys.argv[i+1]
        elif sys.argv[i]=="-o":
            if i + 1 < len(sys.argv):
                outputFileName = sys.argv[i+1]
        elif sys.argv[i]=="-p":
            if i + 1 < len(sys.argv):
                whichProj = int(sys.argv[i+1])
        elif sys.argv[i] == "--help":
            help()

if whichProj==1:
    print("Project 1: ")
    if outputFileName=='':
```

```
        outputFileName='project_1_plot'
    runProj1(fileName, outputFileName, alpha, iterations)
elif whichProj==2:
    print("Project 2: ")
    if outputFileName=='':
        outputFileName='project_2_plot'
    runProj2(fileName, outputFileName, alpha, iterations)
else:
    help()
```

regression.py

```
import pandas as pd
import numpy as np

def gradient_descent(x, y, theta, learning_rate, iterations):
    cost_history=[]

    for i in range(iterations):
        h_theta = np.dot(x, theta)  # dot product of x and theta
        error = h_theta - y  # distance between h and y for error
correction

        cost = (1/(2*len(y))) * np.sum((h_theta - y)**2)
        cost_history.append(cost)
        #get new theta
        theta = theta - (learning_rate/len(y)) * np.dot(x.T, error)

    #storing cost history to a .txt file for debugging
    cost_df = pd.DataFrame({"Iteration": np.arange(1, iterations + 1),
"Cost": cost_history})
    cost_df.to_csv('cost_history.csv', index=False)

    return theta, cost_history

def predict(theta, X_new):
    X_new = np.insert(X_new, 0, 1)
    y_pred = np.dot(theta, X_new)
    return y_pred
```

cleaning.py

```
import pandas as pd
import numpy as np

def clean_empty(df: pd.DataFrame) -> pd.DataFrame:
```

```
    # Replaces empty cells with the average value of the column
    df = df.apply(pd.to_numeric, errors='coerce')
    df.fillna(df.mean(), inplace=True)
    return df
```

proj1.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sys

from cleaning import clean_empty
from regression import gradient_descent

def runProj1(fileName, outputFileName, alpha, iterations):
    df = pd.read_csv(fileName, na_values="########")
    df = clean_empty(df)

    X_vars = ['X1', 'X2', 'X3']
    y = df['Y'].values

    plt.figure(figsize=(12, 10))

    for i, var in enumerate(X_vars):
        X = df[[var]].values
        X = np.column_stack((np.ones(X.shape[0]), X))  # adding column of
ones for the intercept

        theta_initial = np.zeros(X.shape[1])
        theta_final, cost_history = gradient_descent(X, y, theta_initial,
alpha, iterations)

        # regression model
        plt.subplot(3, 2, 2 * i + 1)
        plt.scatter(df[var], y, color='blue', label="Actual Data")
        plt.plot(df[var], np.dot(X, theta_final), color='red',
label="Predicted")
        plt.xlabel(var)
        plt.ylabel("Y")
        plt.title(f"Regression Model for {var}")
        plt.legend()

        # loss over iterations
        plt.subplot(3, 2, 2 * i + 2)
        plt.plot(range(1, iterations + 1), cost_history, linestyle='-')
        plt.xlabel("Iterations")
```

```
        plt.ylabel("Cost")
        plt.title(f"Loss Over Iterations ({var})")
        plt.grid(True)

    plt.tight_layout()
    plt.savefig(outputFileName + ".png")

    print("final theta values:", theta_final)

    final_losses = {}

    for i, var in enumerate(X_vars):
        theta_initial = np.zeros(X.shape[1])
        theta_final, cost_history = gradient_descent(X, y, theta_initial,
alpha, iterations)

        final_losses[var] = cost_history[-1]  #store final loss as array

    best_var = min(final_losses, key=final_losses.get)
    print(f"The best explanatory variable is {best_var} with the lowest
final loss: {final_losses[best_var]}")

    return
```

proj2.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sys

from cleaning import clean_empty
from regression import gradient_descent, predict

def runProj2(fileName, outputFileName, alpha, iterations):
    learning_rates = []
    for i in range(1, 50):
        learning_rates.append(i*alpha/50)

    df = pd.read_csv(fileName, na_values="########")

    df = clean_empty(df)

    X = df[['X1', 'X2', 'X3']].values
    y = df['Y'].values

    X = np.column_stack((np.ones(X.shape[0]), X))  # adding a column of
```

```
ones for the intercept. required for the linear regression formula

    theta_initial = np.zeros(X.shape[1])  # initial values for theta

    results = {}
    cost_values=[]
    cost_histories = {}

    for lr in learning_rates:
        theta_final, cost_history = gradient_descent(X, y, theta_initial,
lr, iterations)
        results[lr] = theta_final
        cost_values.append(cost_history[-1])
        cost_histories[lr] = cost_history

    best_lr = learning_rates[np.argmin(cost_values)]
    best_cost_history = cost_histories[best_lr]

    # Plot Learning Rate vs. Final Cost
    plt.figure(figsize=(12, 6))

    plt.subplot(1, 2, 1)
    plt.plot(learning_rates, cost_values, marker='o', linestyle='-')
    plt.xlabel('Learning Rate')
    plt.ylabel('Final Cost')
    plt.title('Effect of Learning Rate on Final Cost')
    plt.grid(True)

    plt.subplot(1, 2, 2)
    plt.plot(range(1, iterations + 1), best_cost_history, marker='.',
linestyle='-')
    plt.xlabel('Iterations')
    plt.ylabel('Cost')
    plt.title(f'Cost Over Iterations (Best LR={best_lr:.4f})')
    plt.grid(True)

    plt.savefig(outputFileName + ".png")

    print("final theta values:", theta_final)

    #these are the values required by the assignment so I just hardcoded
them in
    X_new_values = [
        [1, 1, 1],
        [2, 0, 4],
        [3, 2, 1],
    ]
```

```
    for X_new in X_new_values:
        y_pred = predict(theta_final, X_new)
        print(f"Predicted y for input {X_new}: {y_pred}")

    return
```

help.txt

```
Parameters:
-p   Project part                    1 or 2
-a   Alpha value (learning rate)     Default: 0.05
-i   Number of iterations            Default: 1000
-f   Input csv file name             Default: D3.csv
-o   Output plot png file name       Defaults: project_1_plot,
project_2_plot
```

Plots

Plot 1: Project part 1

Plot 2: Project Part 2, Cost vs Iterations

Plot 3: Project Part 2, Cost vs Learning Rate