# Tarea 1: Machine Learning

| | |
|---|---|
| Date | @March 24, 2023 |
| Day | Thursday |
| Status | Not started |
| Type | University |


## Alumno: Bastián Barraza Morales.

## Profesor: Alejandro Pereira.


En primer lugar, se importan las bibliotecas necesarias, como NumPy y Matplotlib:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
```

Luego, definimos nuestros datos de entrenamiento. En este ejemplo, utilizaremos un conjunto de datos extraídos desde Kaggle, los cuales relacionan el índice de felicidad de personas agrupadas por país durante 2015-2020, en relación a un conjunto de variables. En este caso se utilizará este índice como la variable regresora, y el ingreso per capita como la variable explicatoria. Luego inicializamos los parámetros (theta) y establecemos algunos hiperparámetros, como la tasa de aprendizaje y el número de iteraciones para el descenso de gradiente. (Ver sitio del dataset en el siguiente link)

```
df = pd.read_csv('C:\\Users\\Bastian Barraza M\\OneDrive\\Documentos\\Freelancer works\\Personal project\\WorldHappiness_Corruption_20
X = np.array(df['happiness_score'])
Y = np.array(df['gdp_per_capita'])

# Define initial values of parameters
m = 0
b = 0

# Define hyperparameters
learning_rate = 0.001
epochs = 200
```

```
fig = px.scatter(x=X, y= Y, marginal_x="histogram", marginal_y="histogram", trendline="ols", width=800, height=800)
fig.update_layout(title='Happiness score by each GDP Per Capita')
fig.update_xaxes(title='Happiness Score',row=1, col=1)
fig.update_yaxes(title='GDP Per Capita',row=1, col=1)
fig.show()
```


A continuación, definimos una función de costo para calcular el error cuadrático medio (MSE) entre los valores predichos y los valores reales, y una función de descenso de gradiente para actualizar los parámetros utilizando los gradientes de la función de costo con respecto a cada parámetro.

```
# Define cost function
def compute_cost(Y, Y_pred):
    cost = np.sum((Y - Y_pred)**2) / (2 * len(Y))
    return cost

# Perform gradient descent
costs = []
for epoch in range(epochs):
    Y_pred = m * X + b
    cost = compute_cost(Y, Y_pred)
    dm = -(2/len(Y)) * np.sum(X * (Y - Y_pred))
    db = -(2/len(Y)) * np.sum(Y - Y_pred)
    m = m - learning_rate * dm
    b = b - learning_rate * db
    costs.append(cost)
```

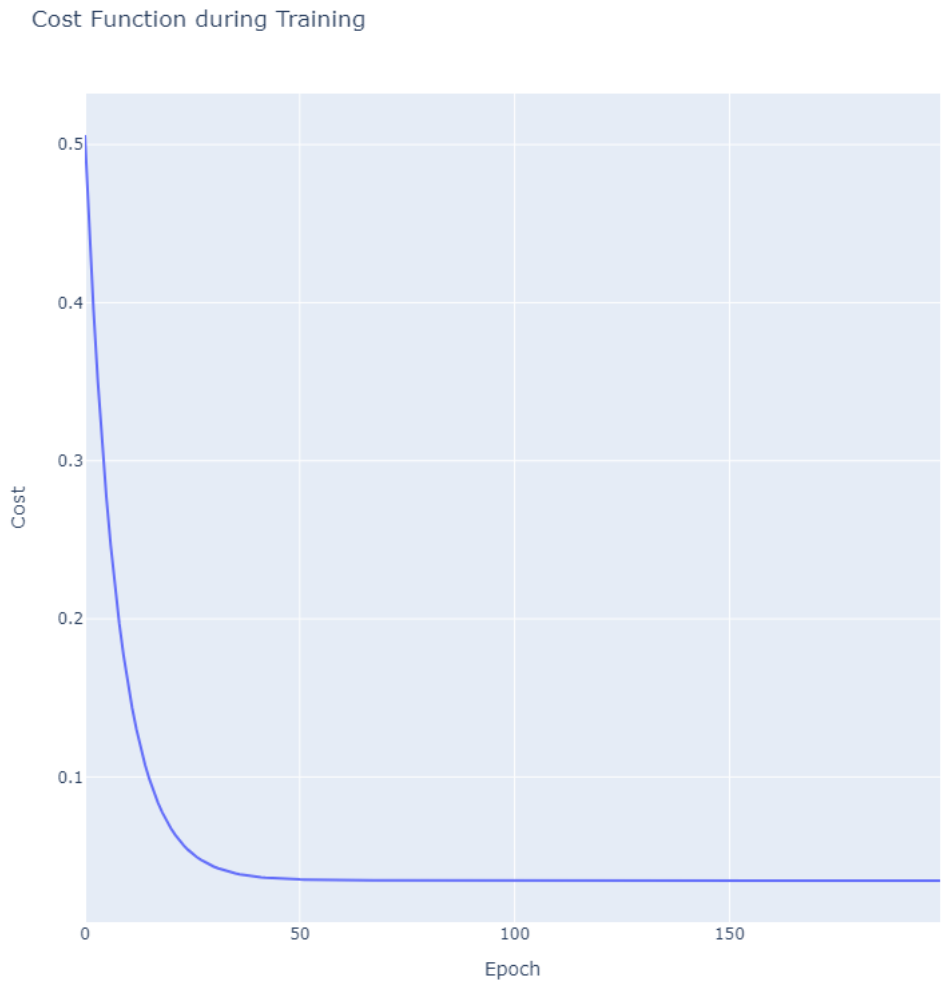
```
# Define cost function
def compute_cost(Y, Y_pred):
    cost = np.sum((Y - Y_pred)**2) / (2 * len(Y))
    return cost

# Perform gradient descent
costs = []
for epoch in range(epochs):
```

```
    Y_pred = m * X + b
    cost = compute_cost(Y, Y_pred)
    dm = -(2/len(Y)) * np.sum(X * (Y - Y_pred))
    db = -(2/len(Y)) * np.sum(Y - Y_pred)
    m = m - learning_rate * dm
    b = b - learning_rate * db
    costs.append(cost)
```

Después de ejecutar el descenso de gradiente, trazamos la historia del costo para verificar si el algoritmo ha convergido.

```
# Visualization of cost function during training
fig = px.line(x=list(range(epochs)), y=costs, title='Cost Function during Training', width=800, height=800)
fig.update_layout(xaxis_title='Epoch', yaxis_title='Cost')
fig.show()
```




Se observa que el algoritmo converge cercano a la iteración 50, por lo que no es necesario que se realice el algoritmo 200 veces.