Project real text

October 19, 2021

[1]:
```
import sys
from operator import add
from pyspark.sql import SparkSession
from pyspark import SparkContext
import pyspark
from pyspark.ml.linalg import Vectors
import numpy as np
from sklearn.linear_model import LinearRegression
from pyspark.sql.types import *
from pyspark.sql import functions as func
from pyspark.sql.functions import *
from pyspark.sql import SQLContext
import matplotlib.pyplot as plt
import time
from pandas import Series,DataFrame
import pandas as pd
import re
from collections import Counter
from sklearn.linear_model import LinearRegression
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.classification import LinearSVC


# building functions

def isfloat(value):
    try:
        float(value)
        return True
    except:
        return False

def correctRows(p):
    if isfloat(p[3]) and isfloat(p[4]) and isfloat(p[6]) and isfloat(p[7]) and␣
↪isfloat(p[9]):
        return p
```

```python
def to_list(a):
    return [a]

def addToList(x, y):
    x.append(y)
    return x

def extend(x,y):
    x.extend(y)
    return x


if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: wordcount <file> <output> ", file=sys.stderr)
        exit(-1)

    spark = SparkSession.builder.master("local[*]").getOrCreate()
    sc = SparkContext.getOrCreate()
    sqlContext = SQLContext(sc)

    # load data set
    lines2 = sc.textFile("Google-Playstore.csv")

    # generate test case file
    df = pd.read_csv("Google-Playstore.csv")
    test_case = df.sample(n = 10000)
    test_case.to_csv('Google-Playstore_test.csv', index=False)

    """

    Simple Linear Regression

    """

    print("##### Finding Simple Linear Regression Equation #####")

    # data pre-processing

    correctLine = lines2.map(lambda x: x.split(','))
    cleaned = correctLine.filter(correctRows)

    max_install = cleaned.map(lambda p: (float(p[7])))
    rating = cleaned.map(lambda p: (float(p[3])))

    # apply linear regression
```

```
x = np.array(max_install.collect())
y = np.array(rating.collect())

X = np.stack([x], axis = 1)

reg = LinearRegression(fit_intercept=True).fit(X, y)

print("The m (coefficient) =",reg.coef_)
print("The b (y-intercept) =",reg.intercept_)
print("The equation is: y = "+str(reg.coef_[0])+"X + "+str(reg.intercept_))


"""

Gradient Descent for parameters

"""

print("##### Finding the parameters using gradient descent #####")

start1 = time.time()
df = np.stack([y, x], axis=1)
dff = map(lambda x: (float(x[0]), Vectors.dense(x[1:])), df)
mydf = spark.createDataFrame(dff, schema=["Money", "Distance"])
myRDD=mydf.rdd.map(tuple).map(lambda x: (float(x[0]), np.array(x[1]) ))

learningRate = 0.00001
num_iteration = 100
size = float(len(y))
beta = np.array([0.1])
costs = []

for i in range(num_iteration):
    gradientCost=myRDD.map(lambda x: (x[1], (x[0] - x[1] * beta) ))\
                            .map(lambda x: (x[0]*x[1], x[1]**2 )).
↪reduce(lambda x, y: (x[0] +y[0], x[1]+y[1] ))
    cost= gradientCost[1]
    gradient=(-1/float(size))* gradientCost[0]
    print(i, "Beta", beta, " Cost", cost)
    beta = beta - learningRate * gradient
    costs.append(cost[0])

end1 = time.time()

print(f"Computation time of BGD is {(end1 - start1)/60} Minutes")

# making plot
```

```python
xValues = [i for i in range(len(costs))]
plt.plot(xValues, costs, 'o', markersize=2)
plt.xlabel("Number of Iteration")
plt.ylabel("Cost")
plt.title("Cost with the number of iteration")
plt.show()


"""

Multi-Linear Regression

"""

print("##### Finding the parameters of multi-linear regression using␣
↪gradient descent #####")

start2 = time.time()

rating = cleaned.map(lambda p: (p[0], p[3]))
rating_count = cleaned.map(lambda p: (p[0], p[4]))
min_install = cleaned.map(lambda p: (p[0], p[6]))
max_install = cleaned.map(lambda p: (p[0], p[7]))
price = cleaned.map(lambda p: (p[0], p[9]))

rating = rating.combineByKey(to_list, addToList, extend)
rating = rating.collect()
rating_count = rating_count.combineByKey(to_list, addToList, extend)
rating_count = rating_count.collect()
min_install = min_install.combineByKey(to_list, addToList, extend)
min_install = min_install.collect()
max_install = max_install.combineByKey(to_list, addToList, extend)
max_install = max_install.collect()
price = price.combineByKey(to_list, addToList, extend)
price = price.collect()


ratingKey = []
ratingValue = []
for i in range(len(rating)):
    ratingKey.append(rating[i][0])

    rate = 0
    total = 0
    for j in [float(i) for i in rating[i][1]]:
        rate += j
        total += 1
```

```
    ratingValue.append(rate/total)


ratingCountKey = []
ratingCountValue = []
for i in range(len(rating_count)):
    ratingCountKey.append(rating_count[i][0])

    rate = 0
    for j in [float(i) for i in rating_count[i][1]]:
        rate += j

    ratingCountValue.append(rate)


min_installKey = []
min_installValue = []
for i in range(len(min_install)):
    min_installKey.append(min_install[i][0])

    count = 0
    for j in [float(i) for i in min_install[i][1]]:
        if j != 0:
            count += 1

    min_installValue.append(count)


max_installKey = []
max_installValue = []
for i in range(len(max_install)):
    min_installKey.append(max_install[i][0])

    count = 0
    for j in [float(i) for i in max_install[i][1]]:
        if j != 0:
            count += 1

    max_installValue.append(count)


priceKey = []
priceValue = []
for i in range(len(price)):
    priceKey.append(price[i][0])
```

```
        amount = 0
        for j in [float(i) for i in price[i][1]]:
            amount += j

        priceValue.append(amount)


    app = ratingKey
    rating = ratingValue
    countOfRating = ratingCountValue
    mi_install = min_installValue
    Price = priceValue

    ma_install = max_installValue


    x = []
    y = []
    for i in range(len(app)):
        x.append([float(rating[i]), float(countOfRating[i]),␣
↪float(mi_install[i]), float(Price[i])])
        y.append(float(ma_install[i]))

    learningRate = 0.000001
    num_iteration = 100
    size = len(y)
    costs = []
    beta = np.array([0.1, 0.1, 0.1, 0.1])

    data = {'y':y, 'x':x}
    df = DataFrame(data)
    spark_df_from_pandas = spark.createDataFrame(df, schema=['x', 'y'])
    myRDD=spark_df_from_pandas.rdd.map(lambda x: (float(x[0]), np.array(x[1])))

    for i in range(num_iteration):
        gradientCost=myRDD.map(lambda x: (x[1], (x[0] - x[1] * beta)))\
                               .map(lambda x: (x[0]*x[1], x[1]**2 )).
↪reduce(lambda x, y: (x[0] +y[0], x[1]+y[1] ))

        cost = 0
        for j in gradientCost[1]:
            cost += j

        gradient=(-1/float(size))* gradientCost[0]
        print(i, "Beta", beta, " Cost", cost)
        beta = beta - learningRate * gradient
```

```
        costs.append(cost)

    end2 = time.time()

    print(f"Computation time of multi-linear regression by BGD is {(end2 -␣
↪start2)/60} minutes")

    xValues = [i for i in range(len(costs))]

    plt.plot(xValues, costs, 'o', markersize=2)
    plt.xlabel("Number of Iteration")
    plt.ylabel("Cost")
    plt.title("Cost with the number of iteration")
    plt.show()


    """

    Logistic Regression

    """

    print("##### gradient descent algorithm to learn a logistic regression␣
↪model #####")

    start3 = time.time()

    total_install = cleaned.map(lambda p: (p[0], p[7]))
    tuples = total_install.collect()

    appWords = []

    for i in app:
        words = i.split(" ")
        for j in words:
            j = re.sub('[^A-Za-z0-9]+', '', j)
            appWords.append(j)

    appWords = ' '.join(appWords).split()

    allWords = sc.parallelize(appWords)
    allCount = allWords.map(lambda x: (x, 1)).reduceByKey(add)
    topWords = allCount.top(20000, lambda x: x[1])

    topWordsK = sc.parallelize(range(20000))
    dictionary = topWordsK.map(lambda x: (topWords[x][0], x))
```

```
def TF(words_list, top_words):
    words_dict = dict(Counter(words_list))
    tf_vector = []
    for word in top_words:
        if word in words_dict.keys():
            tf = words_dict[word]
            tf_vector.append(tf)
        else:
            tf_vector.append(0)
    return tf_vector

key_id = [i for i in range(len(tuples))]
key_values = {tuples[i][0]: i for i in key_id}
topWordsBC = sc.broadcast(dictionary.keys().collect())
feat = cleaned.map(lambda x: (key_values[x[0]], TF(x[1], topWordsBC.value)))
labels = cleaned.map(lambda x: (key_values[x[0]], int(x[0][0] == 'A' and␣
↪x[0][1] == 'U')))
trainRDD = feat.join(labels)

learningRate = 0.0003
num_iteration = 5
lambda_cof = 0.01
size = len(tuples)

loss_list = list()

parameter_vector = np.random.normal(0, 0.1, (dictionary.count(), 1))

def sigmoid(x):
    return 1.0 / (1 + np.exp(-x))

def loss_func(feat_line, y, parameter_vector):
    feat_line = np.array(feat_line)
    pred = sigmoid(np.dot(feat_line, parameter_vector))
    return -y * np.log(pred + 1e-12) - (1 - y) * np.log((1 - pred) + 1e-12)

def accuracy_score(feat_line, y, parameter_vector):
    feat_line = np.array(feat_line)
    pred = sigmoid(np.dot(feat_line, parameter_vector))
    pred = 1 if pred >= 0.5 else 0
    acc = int(pred == y)
    return acc

def grad_func(feat_line, y, parameter_vector):
    feat_line = np.array(feat_line)
    pred = sigmoid(np.dot(feat_line, parameter_vector))
    grad = (pred - y) @ feat_line[None, :]
```

```
        return grad

    acc_his = []
    loss_his = []
    grad_his = []
    prev_param_norm = 0

    for i in tqdm.trange(num_iteration):
        parameter_vector_BC = sc.broadcast(parameter_vector)

        loss = trainRDD.map(lambda x: loss_func(x[1][0], x[1][1],␣
↪parameter_vector_BC.value)).reduce(add) / size
        acc = trainRDD.map(lambda x: accuracy_score(x[1][0], x[1][1],␣
↪parameter_vector_BC.value)).reduce(add) / size
        grad = trainRDD.map(lambda x: grad_func(x[1][0], x[1][1],␣
↪parameter_vector_BC.value)).reduce(add) / size

        parameter_vector = parameter_vector - learningRate * grad[:, None]

        if np.abs(np.linalg.norm(parameter_vector) - prev_param_norm) < 1e-7:
            print('Break')
            break

        prev_param_norm = np.linalg.norm(parameter_vector)
        # L2
        parameter_vector = parameter_vector - 2 * lambda_cof * parameter_vector

        acc_his.append(acc)
        loss_his.append(loss)
        grad_his.append(np.linalg.norm(grad))

    end3 = time.time()

    print(f"Computation time of multi-linear regression by BGD is {(end3 -␣
↪start3)/60} minutes")

    fig, ax = plt.subplots(3, figsize=(13, 13))
    ax[0].set_title('Accurary')
    ax[0].plot(acc_his)
    ax[1].set_title('Loss')
    ax[1].plot(loss_his)
    ax[2].set_title('GradNorm')
    ax[2].plot(grad_his)
    plt.savefig("TrainingProcess.png")
    plt.show()

    print('The five words with the largest coefficients',
```

```
        np.array(topWordsBC.value)[np.argsort(parameter_vector[:, 0])[-5:]])


    """

    Logistic Regression Model Evaluation

    """

    print("##### model evaluation #####")

    start4 = time.time()

    t_tuples = total_install.collect()
    key_id = [i for i in range(len(tuples))]
    key_values = {tuples[i][0]: i for i in key_id}
    topWordsBC = sc.broadcast(dictionary.keys().collect())
    feat = cleaned.map(lambda x: (key_values[x[0]], TF(x[1], topWordsBC.value)))
    labels = cleaned.map(lambda x: (key_values[x[0]], int(x[0][0] == 'A' and␣
↪x[0][1] == 'U')))
    testRDD = feat.join(labels)


    # val
    def TP_func(feat_line, y, parameter_vector):
        feat_line = np.array(feat_line)
        pred = sigmoid(np.dot(feat_line, parameter_vector))
        pred = 1 if pred >= 0.5 else 0
        TP = int(pred == 1 and y == 1)
        return TP


    def FP_func(feat_line, y, parameter_vector):
        feat_line = np.array(feat_line)
        pred = sigmoid(np.dot(feat_line, parameter_vector))
        pred = 1 if pred >= 0.5 else 0
        FP = int(pred == 1 and y != 1)
        return FP


    def FN_func(feat_line, y, parameter_vector):
        feat_line = np.array(feat_line)
        pred = sigmoid(np.dot(feat_line, parameter_vector))
        pred = 1 if pred >= 0.5 else 0
        FN = int(pred != 1 and y == 1)
        return FN
```

```
def TN_func(feat_line, y, parameter_vector):
    feat_line = np.array(feat_line)
    pred = sigmoid(np.dot(feat_line, parameter_vector))
    pred = 1 if pred >= 0.5 else 0
    TN = int(pred != 1 and y != 1)
    return TN


parameter_vector_BC = sc.broadcast(parameter_vector)
acc = testRDD.map(lambda x: accuracy_score(x[1][0], x[1][1],␣
↪parameter_vector_BC.value)).reduce(add) / size
TP = testRDD.map(lambda x: TP_func(x[1][0], x[1][1], parameter_vector_BC.
↪value)).reduce(add)
FP = testRDD.map(lambda x: FP_func(x[1][0], x[1][1], parameter_vector_BC.
↪value)).reduce(add)
FN = testRDD.map(lambda x: FN_func(x[1][0], x[1][1], parameter_vector_BC.
↪value)).reduce(add)
TN = testRDD.map(lambda x: TN_func(x[1][0], x[1][1], parameter_vector_BC.
↪value)).reduce(add)

F1 = 2 * TP / (2 * TP + FN + FP)
print('The Acc of Test: ', acc)
print('The F1 score of Test: ', F1)

print(f"Computation time of multi-linear regression by BGD is {(end4 -␣
↪start4)/60} minutes")


"""

SVM Model

"""

print("##### SVM model #####")

start5 = time.time()

d_corpus = sc.textFile("Google-Playstore.csv")
d_keyAndText = d_corpus.map(lambda x: (x[x.index('id="') + 4: x.index('"␣
↪url=')], x[x.index('">') + 2:][:-6]))
regex = re.compile('[^a-zA-Z]')

d_keyAndListOfWords = d_keyAndText.map(lambda x: (str(x[0]), regex.sub(' ',␣
↪x[1]).lower().split())).sortByKey(False)
```

```python
    tuples = d_keyAndListOfWords.collect()
    allWordsList = []

    for i in range(len(tuples)):
        for j in tuples[i][1]:
            allWordsList.append(j)

    allWords = sc.parallelize(allWordsList)
    allCount = allWords.map(lambda x: (x, 1)).reduceByKey(add)
    topWords = allCount.top(20000, lambda x: x[1])

    topWordsK = sc.parallelize(range(20000))
    dictionary = topWordsK.map(lambda x: (topWords[x][0], x))


    def TF(words_list, top_words):
        words_dict = dict(Counter(words_list))
        tf_vector = []
        for word in top_words:
            if word in words_dict.keys():
                tf = words_dict[word]
                tf_vector.append(tf)
            else:
                tf_vector.append(0)
        return Vectors.dense(tf_vector)

    key_id = [i for i in range(len(tuples))]
    key_values = {tuples[i][0]: i for i in key_id}
    topWordsBC = sc.broadcast(dictionary.keys().collect())
    feat = d_keyAndListOfWords.map(lambda x: (key_values[x[0]], TF(x[1], topWordsBC.value)))
    labels = d_keyAndListOfWords.map(lambda x: (key_values[x[0]], int(x[0][0] == 'A' and x[0][1] == 'U')))
    train_feat_df = sqlContext.createDataFrame(feat, ['ind', 'features'])
    train_labels_df = sqlContext.createDataFrame(labels, ['ind', 'labels'])
    train_df = train_feat_df.join(train_labels_df, on=['ind']).sort(['ind'])
    train_df.cache()

    svc = LinearSVC(labelCol='labels')
    svg_model = svc.fit(train_df)

    st_test_read = time.time()
    t_corpus = sc.textFile('Google-Playstore_test.csv')
    t_keyAndText = t_corpus.map(lambda x: (x[x.index('id="') + 4: x.index('" url=')], x[x.index('">') + 2:][:-6]))
    regex = re.compile('[^a-zA-Z]')
```

```
    t_keyAndListOfWords = t_keyAndText.map(lambda x: (str(x[0]), regex.sub(' ',␣
↪x[1]).lower().split())).sortByKey(False)
    t_tuples = t_keyAndListOfWords.collect()
    t_key_id = [i for i in range(len(t_tuples))]
    t_key_values = {t_tuples[i][0]: i for i in t_key_id}
    t_feat = t_keyAndListOfWords.map(lambda x: (t_key_values[x[0]], TF(x[1],␣
↪topWordsBC.value)))
    t_labels = t_keyAndListOfWords.map(
        lambda x: (t_key_values[x[0]], int(x[0][0] == 'A' and x[0][1] == 'U')))
    test_feat_df = sqlContext.createDataFrame(t_feat, ['ind', 'features'])
    test_labels_df = sqlContext.createDataFrame(t_labels, ['ind', 'labels'])
    test_df = test_feat_df.join(test_labels_df, on=['ind']).sort(['ind'])
    test_df.cache()

    st_test = time.time()
    test_pred = svg_model.evaluate(test_df).predictions
    evaluator = MulticlassClassificationEvaluator(labelCol='labels')

    print('Acc of Test: ', evaluator.evaluate(test_pred, {evaluator.metricName:␣
↪"accuracy"}))
    print('F1 of Test:', evaluator.evaluate(test_pred, {evaluator.metricName:␣
↪"f1"}))

    end5 = time.time()

    print(f"Computation time of SVM regression by BGD is {(end5 - start5)/60}␣
↪minutes")
```

```
##### Finding Simple Linear Regression Equation #####
The m (coefficient) = [0.00285053]
The b (y-intercept) = 0.3456192319738405
The equation is: y = 0.0028505272834074033X + 0.3456192319738405
##### Finding the parameters using gradient descent #####
0 Beta [0.1]  Cost [1.22465805e+09]
1 Beta [-0.0045326]  Cost [12179629.77029447]
2 Beta [0.00420567]  Cost [3706927.27275984]
3 Beta [0.00347521]  Cost [3647720.70333904]
4 Beta [0.00353627]  Cost [3647306.97255329]
5 Beta [0.00353116]  Cost [3647304.08143576]
6 Beta [0.00353159]  Cost [3647304.06123297]
7 Beta [0.00353156]  Cost [3647304.06109216]
8 Beta [0.00353156]  Cost [3647304.06109071]
9 Beta [0.00353156]  Cost [3647304.06109042]
10 Beta [0.00353156]  Cost [3647304.06109096]
```

```
11 Beta [0.00353156]  Cost [3647304.06109107]
12 Beta [0.00353156]  Cost [3647304.06109113]
13 Beta [0.00353156]  Cost [3647304.06109092]
14 Beta [0.00353156]  Cost [3647304.06109097]
15 Beta [0.00353156]  Cost [3647304.06109096]
16 Beta [0.00353156]  Cost [3647304.06109097]
17 Beta [0.00353156]  Cost [3647304.06109097]
18 Beta [0.00353156]  Cost [3647304.06109097]
19 Beta [0.00353156]  Cost [3647304.06109097]
20 Beta [0.00353156]  Cost [3647304.06109097]
21 Beta [0.00353156]  Cost [3647304.06109097]
22 Beta [0.00353156]  Cost [3647304.06109097]
23 Beta [0.00353156]  Cost [3647304.06109097]
24 Beta [0.00353156]  Cost [3647304.06109097]
25 Beta [0.00353156]  Cost [3647304.06109097]
26 Beta [0.00353156]  Cost [3647304.06109097]
27 Beta [0.00353156]  Cost [3647304.06109097]
28 Beta [0.00353156]  Cost [3647304.06109097]
29 Beta [0.00353156]  Cost [3647304.06109097]
30 Beta [0.00353156]  Cost [3647304.06109097]
31 Beta [0.00353156]  Cost [3647304.06109097]
32 Beta [0.00353156]  Cost [3647304.06109097]
33 Beta [0.00353156]  Cost [3647304.06109097]
34 Beta [0.00353156]  Cost [3647304.06109097]
35 Beta [0.00353156]  Cost [3647304.06109097]
36 Beta [0.00353156]  Cost [3647304.06109097]
37 Beta [0.00353156]  Cost [3647304.06109097]
38 Beta [0.00353156]  Cost [3647304.06109097]
39 Beta [0.00353156]  Cost [3647304.06109097]
40 Beta [0.00353156]  Cost [3647304.06109097]
41 Beta [0.00353156]  Cost [3647304.06109097]
42 Beta [0.00353156]  Cost [3647304.06109097]
43 Beta [0.00353156]  Cost [3647304.06109097]
44 Beta [0.00353156]  Cost [3647304.06109097]
45 Beta [0.00353156]  Cost [3647304.06109097]
46 Beta [0.00353156]  Cost [3647304.06109097]
47 Beta [0.00353156]  Cost [3647304.06109097]
48 Beta [0.00353156]  Cost [3647304.06109097]
49 Beta [0.00353156]  Cost [3647304.06109097]
50 Beta [0.00353156]  Cost [3647304.06109097]
51 Beta [0.00353156]  Cost [3647304.06109097]
52 Beta [0.00353156]  Cost [3647304.06109097]
53 Beta [0.00353156]  Cost [3647304.06109097]
54 Beta [0.00353156]  Cost [3647304.06109097]
55 Beta [0.00353156]  Cost [3647304.06109097]
56 Beta [0.00353156]  Cost [3647304.06109097]
57 Beta [0.00353156]  Cost [3647304.06109097]
58 Beta [0.00353156]  Cost [3647304.06109097]
```

```
59 Beta [0.00353156]  Cost [3647304.06109097]
60 Beta [0.00353156]  Cost [3647304.06109097]
61 Beta [0.00353156]  Cost [3647304.06109097]
62 Beta [0.00353156]  Cost [3647304.06109097]
63 Beta [0.00353156]  Cost [3647304.06109097]
64 Beta [0.00353156]  Cost [3647304.06109097]
65 Beta [0.00353156]  Cost [3647304.06109097]
66 Beta [0.00353156]  Cost [3647304.06109097]
67 Beta [0.00353156]  Cost [3647304.06109097]
68 Beta [0.00353156]  Cost [3647304.06109097]
69 Beta [0.00353156]  Cost [3647304.06109097]
70 Beta [0.00353156]  Cost [3647304.06109097]
71 Beta [0.00353156]  Cost [3647304.06109097]
72 Beta [0.00353156]  Cost [3647304.06109097]
73 Beta [0.00353156]  Cost [3647304.06109097]
74 Beta [0.00353156]  Cost [3647304.06109097]
75 Beta [0.00353156]  Cost [3647304.06109097]
76 Beta [0.00353156]  Cost [3647304.06109097]
77 Beta [0.00353156]  Cost [3647304.06109097]
78 Beta [0.00353156]  Cost [3647304.06109097]
79 Beta [0.00353156]  Cost [3647304.06109097]
80 Beta [0.00353156]  Cost [3647304.06109097]
81 Beta [0.00353156]  Cost [3647304.06109097]
82 Beta [0.00353156]  Cost [3647304.06109097]
83 Beta [0.00353156]  Cost [3647304.06109097]
84 Beta [0.00353156]  Cost [3647304.06109097]
85 Beta [0.00353156]  Cost [3647304.06109097]
86 Beta [0.00353156]  Cost [3647304.06109097]
87 Beta [0.00353156]  Cost [3647304.06109097]
88 Beta [0.00353156]  Cost [3647304.06109097]
89 Beta [0.00353156]  Cost [3647304.06109097]
90 Beta [0.00353156]  Cost [3647304.06109097]
91 Beta [0.00353156]  Cost [3647304.06109097]
92 Beta [0.00353156]  Cost [3647304.06109097]
93 Beta [0.00353156]  Cost [3647304.06109097]
94 Beta [0.00353156]  Cost [3647304.06109097]
95 Beta [0.00353156]  Cost [3647304.06109097]
96 Beta [0.00353156]  Cost [3647304.06109097]
97 Beta [0.00353156]  Cost [3647304.06109097]
98 Beta [0.00353156]  Cost [3647304.06109097]
99 Beta [0.00353156]  Cost [3647304.06109097]
Computation time of BGD is 38.446050798892976 Minutes
```

```
<Figure size 640x480 with 1 Axes>
```

```
##### Finding the parameters of multi-linear regression using gradient descent
#####
0 Beta [0.1 0.1 0.1 0.1]  Cost 5.0307575633564e+18
```

```
1 Beta [ 1.00000561e-01 -6.52332938e+03  1.00001340e-01 -4.33113272e+07]  Cost
9.435631463641919e+35
2 Beta [1.00001122e-01 4.25538269e+08 1.00002679e-01 1.87587107e+16]  Cost
1.7700028199286235e+53
3 Beta [ 1.00001683e-01 -2.77592634e+13  1.00004019e-01 -8.12464663e+24]  Cost
3.3202971042668924e+70
4 Beta [1.00002245e-01 1.81082821e+18 1.00005359e-01 3.51889230e+33]  Cost
6.228449320237621e+87
5 Beta [ 1.00002806e-01 -1.18126291e+23  1.00006698e-01 -1.52407897e+42]  Cost
1.1683767963088338e+105
6 Beta [1.00003367e-01 7.70576714e+27 1.00008038e-01 6.60098831e+50]  Cost
2.1917242446163375e+122
7 Beta [ 1.00003928e-01 -5.02672580e+32  1.00009378e-01 -2.85897566e+59]  Cost
4.111392129332666e+139
8 Beta [1.00004489e-01 3.27909886e+37 1.00010717e-01 1.23826031e+68]  Cost
7.712441600561638e+156
9 Beta [ 1.00005050e-01 -2.13906423e+42  1.00012057e-01 -5.36306976e+76]  Cost
1.4467546167076145e+174
10 Beta [1.00005611e-01 1.39538207e+47 1.00013396e-01 2.32281670e+85]  Cost
2.7139251476631882e+191
11 Beta [ 1.00006172e-01 -9.10253702e+51  1.00014736e-01 -1.00604275e+94]  Cost
5.090973702147291e+208
12 Beta [1.00006734e-001 5.93788481e+056 1.00016076e-001 4.35730468e+102]  Cost
9.550010345079675e+225
13 Beta [ 1.00007295e-001 -3.87347791e+061  1.00017415e-001 -1.88720650e+111]
Cost 1.791458823538236e+243
14 Beta [1.00007856e-001 2.52679726e+066 1.00018755e-001 8.17374184e+119]  Cost
3.360545800965015e+260
15 Beta [ 1.00008417e-001 -1.64831311e+071  1.00020095e-001 -3.54015609e+128]
Cost 6.303950686446002e+277
16 Beta [1.00008978e-001 1.07524895e+076 1.00021434e-001 1.53328859e+137]  Cost
1.1825398792580455e+295
17 Beta [ 1.00009539e-001 -7.01420317e+080  1.00022774e-001 -6.64087643e+145]
Cost inf
18 Beta [1.00010100e-001 4.57559584e+085 1.00024113e-001 2.87625173e+154]  Cost
inf
19 Beta [ 1.00010661e-001 -2.98481192e+090  1.00025453e-001 -1.24574281e+163]
Cost inf
20 Beta [1.00011222e-001 1.94709116e+095 1.00026793e-001 5.39547745e+171]  Cost
inf
21 Beta [ 1.00011783e-001 -1.27015172e+100  1.00028132e-001 -2.33685290e+180]
Cost inf
22 Beta [1.00012345e-001 8.28561815e+104 1.00029472e-001 1.01212201e+189]  Cost
inf
23 Beta [ 1.00012906e-001 -5.40498172e+109  1.00030812e-001 -4.38363479e+197]
Cost inf
24 Beta [1.00013467e-001 3.52584766e+114 1.00032151e-001 1.89861041e+206]  Cost
inf
```

```
25 Beta [ 1.00014028e-001 -2.30002660e+119  1.00033491e-001 -8.22313373e+214]
Cost inf
26 Beta [1.00014589e-001 1.50038313e+124 1.00034830e-001 3.56154837e+223]  Cost
inf
27 Beta [ 1.00015150e-001 -9.78749352e+128  1.00036170e-001 -1.54255388e+232]
Cost inf
28 Beta [1.00015711e-001 6.38470450e+133 1.00037510e-001 6.68100560e+240]  Cost
inf
29 Beta [ 1.00016272e-001 -4.16495311e+138  1.00038849e-001 -2.89363221e+249]
Cost inf
30 Beta [1.00016833e-001 2.71693614e+143 1.00040189e-001 1.25327052e+258]  Cost
inf
31 Beta [ 1.00017394e-001 -1.77234697e+148  1.00041528e-001 -5.42808098e+266]
Cost inf
32 Beta [1.00017955e-001 1.15616032e+153 1.00042868e-001 2.35097392e+275]  Cost
inf
33 Beta [ 1.00018516e-001 -7.54201471e+157  1.00044208e-001 -1.01823801e+284]
Cost inf
34 Beta [1.00019077e-001 4.91990469e+162 1.00045547e-001 4.41012400e+292]  Cost
inf
35 Beta [ 1.00019638e-001 -3.20941593e+167  1.00046887e-001             -inf]
Cost nan
36 Beta [1.00020199e-001 2.09360776e+172 1.00048226e-001             nan]  Cost
nan
37 Beta [ 1.00020761e-001 -1.36572932e+177  1.00049566e-001              nan]
Cost nan
38 Beta [1.00021322e-001 8.90910235e+181 1.00050906e-001             nan]  Cost
nan
39 Beta [ 1.00021883e-001 -5.81170100e+186  1.00052245e-001              nan]
Cost nan
40 Beta [1.00022444e-001 3.79116405e+191 1.00053585e-001             nan]  Cost
nan
41 Beta [ 1.00023005e-001 -2.47310122e+196  1.00054924e-001              nan]
Cost nan
42 Beta [1.00023566e-001 1.61328541e+201 1.00056264e-001             nan]  Cost
nan
43 Beta [ 1.00024127e-001 -1.05239923e+206  1.00057603e-001              nan]
Cost nan
44 Beta [1.00024688e-001 6.86514692e+210 1.00058943e-001             nan]  Cost
nan
45 Beta [ 1.00025249e-001 -4.47836153e+215  1.00060283e-001              nan]
Cost nan
46 Beta [1.00025810e-001 2.92138278e+220 1.00061622e-001             nan]  Cost
nan
47 Beta [ 1.00026371e-001 -1.90571424e+225  1.00062962e-001              nan]
Cost nan
48 Beta [1.00026932e-001 1.24316019e+230 1.00064301e-001             nan]  Cost
nan
```

```
49 Beta [ 1.00027493e-001 -8.10954352e+234  1.00065641e-001              nan]
Cost nan
50 Beta [1.00028054e-001 5.29012243e+239 1.00066980e-001             nan]  Cost
nan
51 Beta [ 1.00028615e-001 -3.45092116e+244  1.00068320e-001              nan]
Cost nan
52 Beta [1.00029176e-001 2.25114958e+249 1.00069659e-001             nan]  Cost
nan
53 Beta [ 1.00029737e-001 -1.46849904e+254  1.00070999e-001              nan]
Cost nan
54 Beta [1.00030298e-001 9.57950309e+258 1.00072339e-001             nan]  Cost
nan
55 Beta [ 1.00030859e-001 -6.24902549e+263  1.00073678e-001              nan]
Cost nan
56 Beta [1.00031420e-001 4.07644522e+268 1.00075018e-001             nan]  Cost
nan
57 Beta [ 1.00031981e-001 -2.65919953e+273  1.00076357e-001              nan]
Cost nan
58 Beta [1.00032542e-001 1.73468347e+278 1.00077697e-001             nan]  Cost
nan
59 Beta [ 1.00033103e-001 -1.13159118e+283  1.00079036e-001              nan]
Cost nan
60 Beta [1.00033664e-001 7.38174212e+287 1.00080376e-001             nan]  Cost
nan
61 Beta [ 1.00034225e-001 -4.81535360e+292  1.00081715e-001              nan]
Cost nan
62 Beta [0.10003479        inf 0.10008305        nan]  Cost nan
63 Beta [0.10003535        nan 0.10008439        nan]  Cost nan
64 Beta [0.10003591        nan 0.10008573        nan]  Cost nan
65 Beta [0.10003647        nan 0.10008707        nan]  Cost nan
66 Beta [0.10003703        nan 0.10008841        nan]  Cost nan
67 Beta [0.10003759        nan 0.10008975        nan]  Cost nan
68 Beta [0.10003815        nan 0.10009109        nan]  Cost nan
69 Beta [0.10003871        nan 0.10009243        nan]  Cost nan
70 Beta [0.10003927        nan 0.10009377        nan]  Cost nan
71 Beta [0.10003983        nan 0.10009511        nan]  Cost nan
72 Beta [0.1000404         nan 0.10009645        nan]  Cost nan
73 Beta [0.10004096        nan 0.10009779        nan]  Cost nan
74 Beta [0.10004152        nan 0.10009913        nan]  Cost nan
75 Beta [0.10004208        nan 0.10010047        nan]  Cost nan
76 Beta [0.10004264        nan 0.10010181        nan]  Cost nan
77 Beta [0.1000432         nan 0.10010315        nan]  Cost nan
78 Beta [0.10004376        nan 0.10010449        nan]  Cost nan
79 Beta [0.10004432        nan 0.10010583        nan]  Cost nan
80 Beta [0.10004488        nan 0.10010717        nan]  Cost nan
81 Beta [0.10004544        nan 0.10010851        nan]  Cost nan
82 Beta [0.10004601        nan 0.10010985        nan]  Cost nan
83 Beta [0.10004657        nan 0.10011118        nan]  Cost nan
```

```
84 Beta [0.10004713        nan 0.10011252        nan]  Cost nan
85 Beta [0.10004769        nan 0.10011386        nan]  Cost nan
86 Beta [0.10004825        nan 0.1001152         nan]  Cost nan
87 Beta [0.10004881        nan 0.10011654        nan]  Cost nan
88 Beta [0.10004937        nan 0.10011788        nan]  Cost nan
89 Beta [0.10004993        nan 0.10011922        nan]  Cost nan
90 Beta [0.10005049        nan 0.10012056        nan]  Cost nan
91 Beta [0.10005105        nan 0.1001219         nan]  Cost nan
92 Beta [0.10005161        nan 0.10012324        nan]  Cost nan
93 Beta [0.10005218        nan 0.10012458        nan]  Cost nan
94 Beta [0.10005274        nan 0.10012592        nan]  Cost nan
95 Beta [0.1000533         nan 0.10012726        nan]  Cost nan
96 Beta [0.10005386        nan 0.1001286         nan]  Cost nan
97 Beta [0.10005442        nan 0.10012994        nan]  Cost nan
98 Beta [0.10005498        nan 0.10013128        nan]  Cost nan
99 Beta [0.10005554        nan 0.10013262        nan]  Cost nan
Computation time of multi-linear regression by BGD is 24.002660648028055 minutes
```

```
<Figure size 640x480 with 1 Axes>
```

```
##### Logistic regression model #####
Acc of Test:  0.9975432599871822
F1 of Test: 0.9974863907576629
Computation time of logistic regression is 28.70752596855166 minutes
##### SVM model #####
Acc of Test:  0.982736100123734
F1 of Test: 0.980016837261283
Computation time of logistic regression is 29.631058894040428 minutes
```

[ ]:

```

```