o1

September 12, 2025

Gruppe 30

Navn	Studienummer
Lasse Borring Petersen	202208165
Benjamin Harboe Strunge	202209864
Esben Inglev	202210050
Asbjørn Vad	202208512

1 Modules and Classes

Path setup for libs:

[1]:
```
import sys,os
sys.path.append(os.path.expanduser('./libitmal'))

from libitmal import utils as itmalutils
print(dir(itmalutils))
print(itmalutils.__file__)
```

```
['AssertInRange', 'CheckFloat', 'InRange', 'Iterable', 'PrintMatrix',
'ResetRandom', 'TEST', 'TestAll', 'TestCheckFloat', 'TestPrintMatrix',
'TestVarName', 'VarName', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'ctxlib', 'inf', 'inspect',
'isFloat', 'isList', 'isNumpyArray', 'nan', 'np', 'random', 're']
/home/lassebp7/code/6.Semester/MAL/libitmal/utils.py
```

1.1 Qa - Load and test libitmal

[2]:
```
from libitmal import utils as itmalutils

itmalutils.TestAll()
```

```
TestPrintMatrix…(no regression testing)
X=[[   1.    2.]
   [   3. -100.]
   [   1.   -1.]]
X=[[ 1.  2.]
   …
   [ 1. -1.]]
X=[[   1.
       2.    ]
   [   3.0001
    -100.    ]
   [   1.
      -1.    ]]
X=[[   1.    2.]
   [   3. -100.]
   [   1.   -1.]]
OK
TEST: OK
ALL OK
```

1.2 Qb Create your own module, with some functions, and test it

Below is two small printer functions placed in `malutils` and imported:

[3]:
```python
import malutils

malutils.HelloWorld()
malutils.Greeter("Pokemon!")
```

```
Hello World!
Hello Pokemon!!
```

1.3 Qc How do you 'recompile' a module?

1.3.1 Answer

Reload of modules can be done in serveral ways. One simple way is to just restart the kernal. Another is the code below.

[4]:
```python
import importlib
importlib.reload(malutils)
```

[4]:
```
<module 'malutils' from
'/home/lassebp7/code/6.Semester/MAL/mal_grp30/01/malutils/__init__.py'>
```

If you are using VSCode, it is also possible to add the following code to settings.json, which will make the notebook auto reload the module changes.

```
"jupyter.runStartupCommands": ["%load_ext autoreload", "%autoreload 2"],
```

1.4 Qe Extend the class with some public and private functions and member variables

1.4.1 Answers

As can be seen below, private function and member variables are represented in python by two ___ prefixed to the name.

The meaning of `self` is that it is a reference to the class instance itself. Other languages have 'this' as a reference to the class instance itself.

Calling a function without `self` in the parameter list is not allowed in python, as can be seen from the output of the exception catch.

[5]:
```python
class MyClass:
    def myFun(self):
        self.myvar = "Public function"
        print(f"This is a message inside the class, myvar={self.myvar}.")

    #private function
    def __myfun(self):
        self.myvar = "Private"
        print(f"This is a private message inside the class, myvar={self.myvar}.
 ↪")
```

```
    def callToPrivate(self):
        print(f"Calling private function, myvar={self.myvar}.")
        self.__myfun()
        print("Done with private function")

    def myFun2(): # this wont work!
        print("No self")


instance = MyClass()

instance.myFun()
try:
    instance.__myfun()
except:
    print("Exception: can't call private function")

instance.callToPrivate()
try:
    instance.myFun2()
except:
    print("Exception: no self class method!")
```

```
This is a message inside the class, myvar=Public function.
Exception: can't call private function
Calling private function, myvar=Public function.
This is a private message inside the class, myvar=Private.
Done with private function
Exception: no self class method!
```

1.5 Qf Extend the class with a Constructor

1.5.1 Answers

As can be seen below, the constructor is named ___init___ and takes the 'self' parameter and an arbitrary number of parameters. There is no real destructor compared to the C++ destructor. Python is a managed language, so objects that are no longer in use are garbage collected.

The ___del___ function is not a destructor. It's just a function that gets called when the garbage collector destroys the instance.

[6]:
```
class MyCtorClass:
    def __init__(self,x):
        self.x = x
        print(f"Constructor called with x={x}")

    def GetX(self):
        return self.x
```

```
ctorInstance = MyCtorClass(42)
num = ctorInstance.GetX()
print(f"numn from instance = {num}")
```

```
Constructor called with x=42
numn from instance = 42
```

1.6 Qg Extend the class with a to-string function

Below is a small class with a “to string” method:

[7]:
```
class MyToStringClass:
    def __init__(self,x):
        self.x = x

    def __str__(self):
        return f"MyToStringClass (x={self.x})"

strClass = MyToStringClass(420)
print(strClass)
```

```
MyToStringClass (x=420)
```

2 Intro

[8]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.linear_model
import os

def prepare_country_stats(oecd_bli, gdp_per_capita):
    oecd_bli = oecd_bli[oecd_bli["INEQUALITY"]=="TOT"]
    oecd_bli = oecd_bli.pivot(index="Country", columns="Indicator",␣
 ↪values="Value")
    gdp_per_capita.rename(columns={"2015": "GDP per capita"}, inplace=True)
    gdp_per_capita.set_index("Country", inplace=True)
    full_country_stats = pd.merge(left=oecd_bli, right=gdp_per_capita,
                                  left_index=True, right_index=True)
    full_country_stats.sort_values(by="GDP per capita", inplace=True)
    remove_indices = [0, 1, 6, 8, 33, 34, 35]
    keep_indices = list(set(range(36)) - set(remove_indices))
    return full_country_stats[["GDP per capita", 'Life satisfaction']].
 ↪iloc[keep_indices]

datapath = os.path.join("./datasets", "lifesat", "")

# Load the data
```

```
oecd_bli = pd.read_csv(datapath + "oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv(datapath + "gdp_per_capita.
 ↪csv",thousands=',',delimiter='\t',
                             encoding='latin1', na_values="n/a")

# Prepare the data
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# Visualize the data
#country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
#plt.show()

# Select a linear model
model = sklearn.linear_model.LinearRegression()
# Train the model
model.fit(X, y)

print("OK")
```

```
OK
```

2.1 Qa) The θ parameters and the R^2 Score

2.1.1 Answers

Maximum for R^2 is 1.

Minimum for R^2 is negative infinity. This is if the model makes predictions worse than just guessing the average, and can be a result of overfitting, bad test data etc.

It's better to have a higher R^2 score. This measures the fitness of the model.

[9]:
```
# skæring ved x-aksen
theta_0 = model.intercept_
# koefficienten
theta_1 = model.coef_[0]
print(f"h(x) = {theta_0[0]:.4f} + {theta_1[0]}x")

u = np.sum((y - model.predict(X))**2)
v = np.sum((y - np.mean(y))**2)

R2 = 1 - u/v
R2_skl = model.score(X, y)
print(f"R2 = {R2}")
print(f"R2_skl = {R2_skl}")
```

```
h(x) = 4.8531 + 4.911544589158484e-05x
R2 = 0.7344414355437031
```

```
R2_skl = 0.7344414355437031
```

2.2 Qb) Using k-Nearest Neighbors

2.2.1 Answers

KNN regressor also uses R^2 as a score, so in that regard they can be compared to each other. However, the knn model might overfit to the data, since k=3 allows for the model to fluctuate a bit.

This information about the score function was found at the following locations in the documentation.

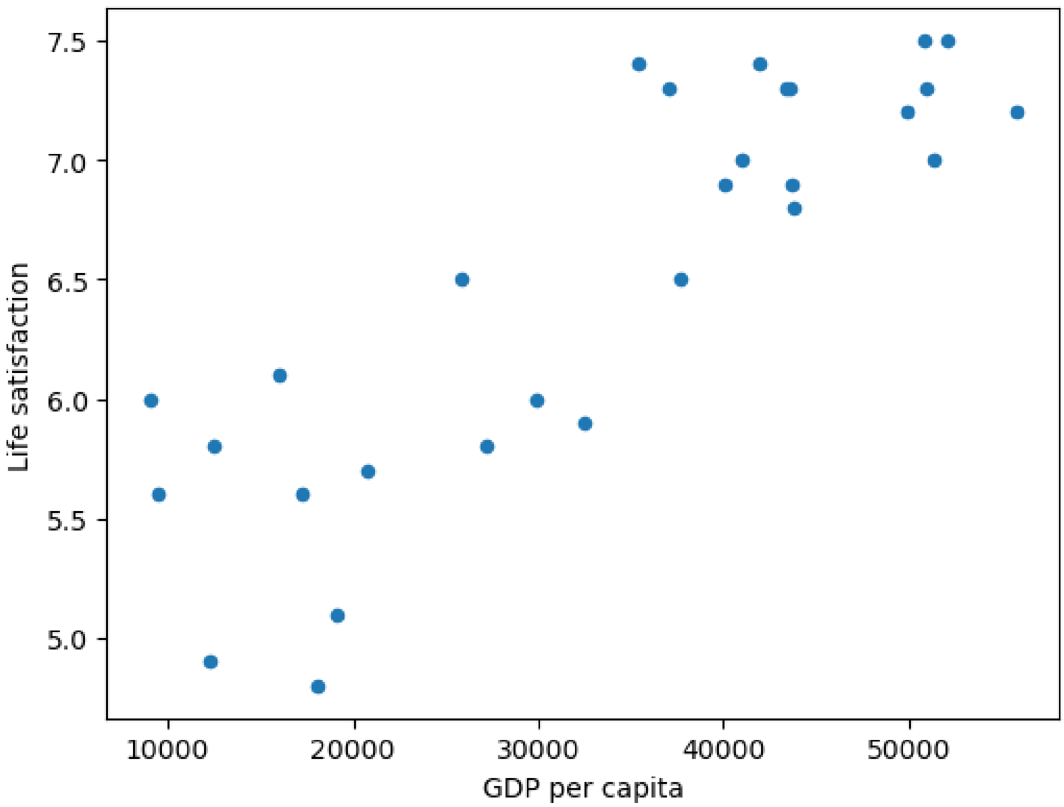
Linear Reg https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklea

KNN https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html#sklearn.ne

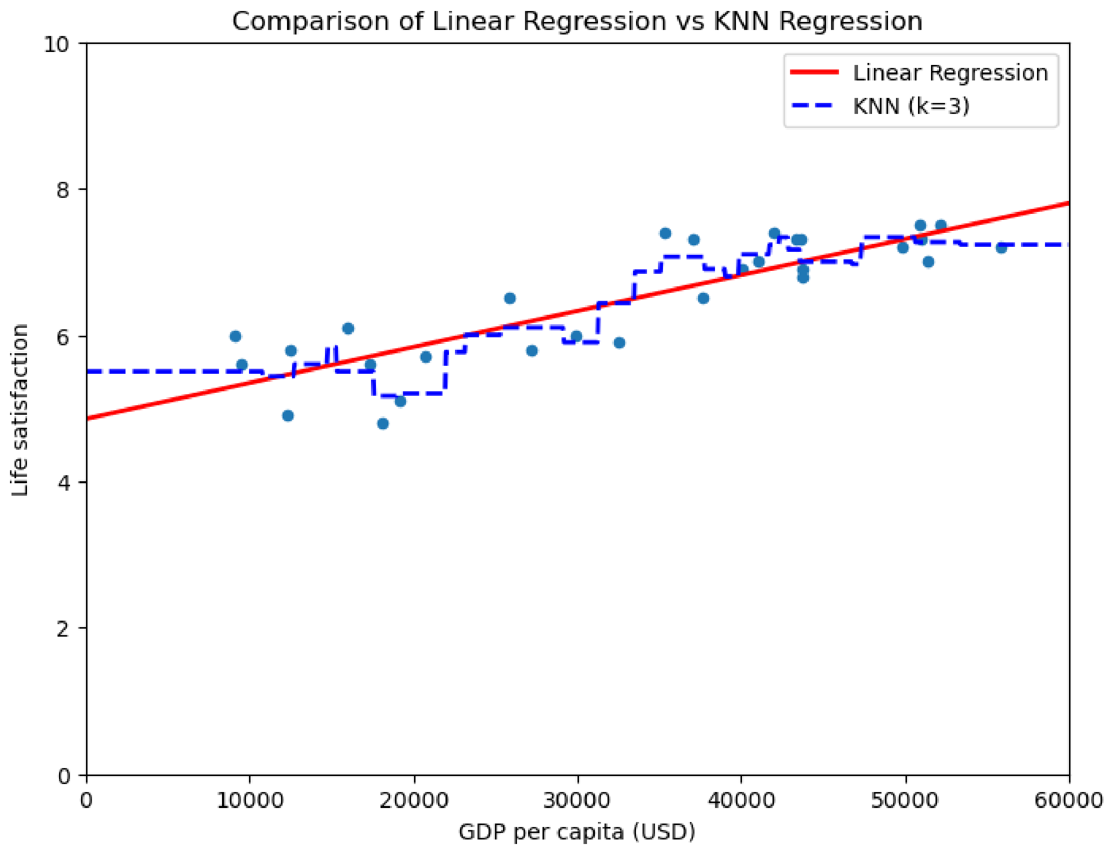
[10]:
```
# Prepare the data
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

print("X.shape=",X.shape)
print("y.shape=",y.shape)

# Visualize the data
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

# Select and train a model
k = 3
knn = sklearn.neighbors.KNeighborsRegressor(k)
knn.fit(X, y)

# Plot knn
m = np.linspace(0, 60000, 1000)
M = np.empty([m.shape[0], 1])
M[:, 0] = m

y_pred_lin = model.predict(M)  # Linear regression predictions
y_pred_knn = knn.predict(M)

# Create the plot
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction',␣
 ↪figsize=(8, 6))
plt.axis([0, 60000, 0, 10])

# Plot both model predictions
plt.plot(m, y_pred_lin, "r-", label="Linear Regression", linewidth=2)
plt.plot(m, y_pred_knn, "b--", label=f"KNN (k={k})", linewidth=2)
```

```python
# Add labels and legend
plt.xlabel("GDP per capita (USD)")
plt.ylabel("Life satisfaction")
plt.legend()
plt.title("Comparison of Linear Regression vs KNN Regression")
plt.show()

# Print model performance
print(f"Linear Regression Score (R²): {model.score(X, y):.3f}")
print(f"KNN Score (R²): {knn.score(X, y):.3f}")

# Make prediction for Cyprus
X_cyprus = [[22587]]
lin_pred = model.predict(X_cyprus)
knn_pred = knn.predict(X_cyprus)

print(f"\nPredictions for Cyprus (GDP = 22587 USD):")
print(f"Linear Regression: {lin_pred[0][0]:.2f}")
print(f"KNN (k={k}): {knn_pred[0]}")

knn_score = knn.score(X, y)
print(f"KNN score              : {knn_score}")
print(f"Linear Regression score: {R2_skl}")
```

```
X.shape= (29, 1)
y.shape= (29, 1)
```



7.5
7.0
6.5
6.0
5.5
5.0
Life satisfaction
10000
20000
30000
40000
50000
GDP per capita




```
Linear Regression Score (R²): 0.734
KNN Score (R²): 0.853

Predictions for Cyprus (GDP = 22587 USD):
Linear Regression: 5.96
KNN (k=3): [5.76666667]
KNN score              : 0.8525732853499179
Linear Regression score: 0.7344414355437031
```

2.3 Qc) Tuning Parameter for k-Nearest Neighbors and A Sanity Check

2.3.1 Answers

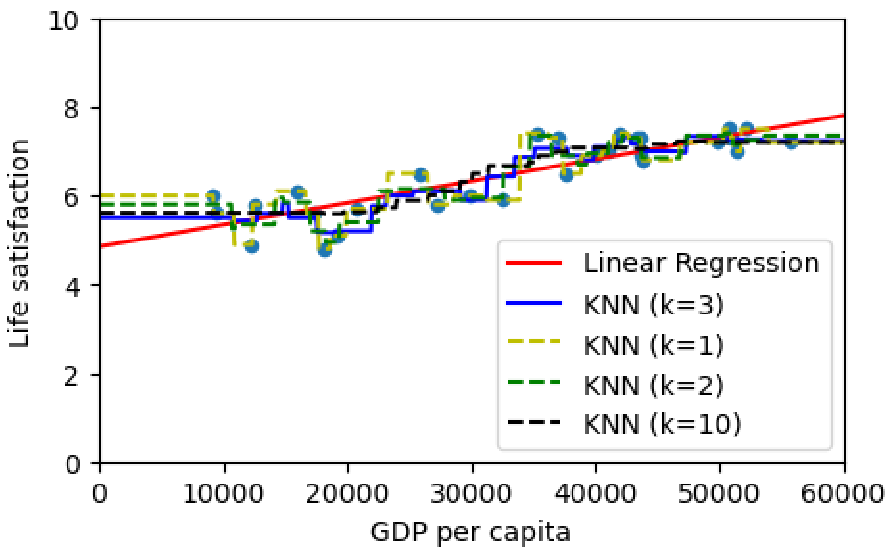
K=1 gives score 1 This obviously looks good because of what is discussed earlier about higher R2 values, but by looking at the graph, it just fits the model 100%, because it just draws a line between each point. This means it has no idea how to predict anything depending on the training data, and it will just predict the training data value closest to whatever value we are trying to predict using the model.

k=5, k=10... The model uses more neighbors to predict new data, where the higher k means it looks at more neighbors to make its prediction. This will make the generalization better, but lower the training score and the R2 value.

k20... When setting the neighbor count this high, we risk overfitting, where the line just becomes straight and gives bad predictions again.

[11]:
```
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction',␣
 ↪figsize=(5,3))
plt.axis([0, 60000, 0, 10])

# create an test matrix M, with the same dimensionality as X, and in the range␣
 ↪[0;60000]
# and a step size of your choice
m=np.linspace(0, 60000, 1000)
M=np.empty([m.shape[0],1])
M[:,0]=m

# from this test M data, predict the y values via the lin.reg. and k-nearest␣
 ↪models
y_pred_lin = model.predict(M)
y_pred_knn = knn.predict(M)   # ASSUMING the variable name 'knn' of your␣
 ↪KNeighborsRegressor

# use plt.plot to plot x-y into the sample_data plot..
plt.plot(m, y_pred_lin, "r", label="Linear Regression")
plt.plot(m, y_pred_knn, "b", label="KNN (k=3)")
knn_score3 = knn.score(X, y)

knn = sklearn.neighbors.KNeighborsRegressor(1)
knn.fit(X, y)
knn_score1 = knn.score(X, y)
y_pred_knn1 = knn.predict(M)
plt.plot(m, y_pred_knn1, "y--", label="KNN (k=1)")

knn = sklearn.neighbors.KNeighborsRegressor(2)
knn.fit(X, y)
knn_score2 = knn.score(X, y)
y_pred_knn2 = knn.predict(M)
plt.plot(m, y_pred_knn2, "g--", label="KNN (k=2)")

knn = sklearn.neighbors.KNeighborsRegressor(10)
knn.fit(X, y)
knn_score10 = knn.score(X, y)
y_pred_knn10 = knn.predict(M)
plt.plot(m, y_pred_knn10, "k--", label="KNN (k=10)")

plt.legend()
plt.show()
```

```
#scores
print(f"KNN score (k=1)    : {knn_score1}")
print(f"KNN score (k=2)    : {knn_score2}")
print(f"KNN score (k=3)    : {knn_score3}")
print(f"KNN score (k=10)   : {knn_score10}")
print(f"Linear Regression score: {R2_skl}")
```




```
KNN score (k=1)    : 1.0
KNN score (k=2)    : 0.9091881835016248
KNN score (k=3)    : 0.8525732853499179
KNN score (k=10)   : 0.7833080605150065
Linear Regression score: 0.7344414355437031
```

2.4 Qd) Trying out a Neural Network

[12]:
```
from sklearn.neural_network import MLPRegressor

# Setup MLPRegressor
mlp = MLPRegressor( hidden_layer_sizes=(10,), solver='adam', activation='relu',
 ↪tol=1E-5, max_iter=100000, verbose=True)
mlp.fit(X, y.ravel())

# lets make a MLP regressor prediction and redo the plots
y_pred_mlp = mlp.predict(M)

mlp_score = mlp.score(X, y)
```

```python
plt.plot(m, y_pred_lin, "r", label="Linear Regression")
plt.plot(m, y_pred_knn, "b", label="KNN (k=3)")
plt.plot(m, y_pred_mlp, "k", label="MLP (h=10)")

plt.legend()
plt.show()

y_nn_pred = mlp.predict(X_cyprus)
print(f"Predictions for Cyprus (GDP = 22587 USD): {y_nn_pred[0]}")

# Scores
print(f"KNN score (k=3)   : {knn_score3}")
print(f"Linear Regression score: {R2_skl}")
print(f"MLP score         : {mlp_score}")
```

```
Iteration 1, loss = 241269671.70241207
Iteration 2, loss = 238627172.97791645
Iteration 3, loss = 236002859.05747169
Iteration 4, loss = 233396981.59040323
Iteration 5, loss = 230809782.89627913
Iteration 6, loss = 228241495.31556553
Iteration 7, loss = 225692340.60716501
Iteration 8, loss = 223162529.39726055
Iteration 9, loss = 220652260.68325421
Iteration 10, loss = 218161721.39591217
Iteration 11, loss = 215691086.02211234
Iteration 12, loss = 213240516.28986719
Iteration 13, loss = 210810160.91656119
Iteration 14, loss = 208400155.42062542
Iteration 15, loss = 206010621.99617901
Iteration 16, loss = 203641669.44951037
Iteration 17, loss = 201293393.19566348
Iteration 18, loss = 198965875.31284085
Iteration 19, loss = 196659184.65184656
Iteration 20, loss = 194373376.99736831
Iteration 21, loss = 192108495.27754754
Iteration 22, loss = 189864569.81800640
Iteration 23, loss = 187641618.63629106
Iteration 24, loss = 185439647.77255344
Iteration 25, loss = 183258651.65221578
Iteration 26, loss = 181098613.47635674
Iteration 27, loss = 178959505.63559490
Iteration 28, loss = 176841290.14334562
Iteration 29, loss = 174743919.08446059
Iteration 30, loss = 172667335.07543904
Iteration 31, loss = 170611471.73260182
Iteration 32, loss = 168576254.14485231
Iteration 33, loss = 166561599.34789708
```

```
Iteration 34, loss = 164567416.79705611
Iteration 35, loss = 162593608.83606538
Iteration 36, loss = 160640071.15954152
Iteration 37, loss = 158706693.26704603
Iteration 38, loss = 156793358.90695116
Iteration 39, loss = 154899946.50856197
Iteration 40, loss = 153026329.60119081
Iteration 41, loss = 151172377.21911317
Iteration 42, loss = 149337954.29154515
Iteration 43, loss = 147522922.01698315
Iteration 44, loss = 145727138.22143203
Iteration 45, loss = 143950457.70020947
Iteration 46, loss = 142192732.54316923
Iteration 47, loss = 140453812.44331515
Iteration 48, loss = 138733544.98889995
Iteration 49, loss = 137031775.93920210
Iteration 50, loss = 135348349.48426476
Iteration 51, loss = 133683108.48895615
Iteration 52, loss = 132035894.72177133
Iteration 53, loss = 130406549.06884965
Iteration 54, loss = 128794911.73372027
Iteration 55, loss = 127200822.42332166
Iteration 56, loss = 125624120.52086209
Iteration 57, loss = 124064645.24610542
Iteration 58, loss = 122522235.80367298
Iteration 59, loss = 120996731.51995675
Iteration 60, loss = 119487971.96923667
Iteration 61, loss = 117995797.08958735
Iteration 62, loss = 116520047.28915091
Iteration 63, loss = 115060563.54333788
Iteration 64, loss = 113617187.48350392
Iteration 65, loss = 112189761.47763158
Iteration 66, loss = 110778128.70352770
Iteration 67, loss = 109382133.21502563
Iteration 68, loss = 108001620.00166319
Iteration 69, loss = 106636435.04228225
Iteration 70, loss = 105286425.35297588
Iteration 71, loss = 103951439.02978800
Iteration 72, loss = 102631325.28654654
Iteration 73, loss = 101325934.48819205
Iteration 74, loss = 100035118.17994155
Iteration 75, loss = 98758729.11260703
Iteration 76, loss = 97496621.26436983
Iteration 77, loss = 96248649.85929106
Iteration 78, loss = 95014671.38282225
Iteration 79, loss = 93794543.59456091
Iteration 80, loss = 92588125.53848058
Iteration 81, loss = 91395277.55084935
```

```
Iteration 82, loss = 90215861.26603399
Iteration 83, loss = 89049739.62037455
Iteration 84, loss = 87896776.85430108
Iteration 85, loss = 86756838.51284938
Iteration 86, loss = 85629791.44472325
Iteration 87, loss = 84515503.80003873
Iteration 88, loss = 83413845.02687503
Iteration 89, loss = 82324685.86674777
Iteration 90, loss = 81247898.34911059
Iteration 91, loss = 80183355.78498366
Iteration 92, loss = 79130932.75979878
Iteration 93, loss = 78090505.12554350
Iteration 94, loss = 77061949.99228118
Iteration 95, loss = 76045145.71911576
Iteration 96, loss = 75039971.90466595
Iteration 97, loss = 74046309.37710661
Iteration 98, loss = 73064040.18383175
Iteration 99, loss = 72093047.58078739
Iteration 100, loss = 71133216.02151915
Iteration 101, loss = 70184431.14597580
Iteration 102, loss = 69246579.76910537
Iteration 103, loss = 68319549.86927830
Iteration 104, loss = 67403230.57656796
Iteration 105, loss = 66497512.16091702
Iteration 106, loss = 65602286.02021486
Iteration 107, loss = 64717444.66830928
Iteration 108, loss = 63842881.72297354
Iteration 109, loss = 62978491.89384738
Iteration 110, loss = 62124170.97036944
Iteration 111, loss = 61279815.80971681
Iteration 112, loss = 60445324.32476501
Iteration 113, loss = 59620595.47208157
Iteration 114, loss = 58805529.23996442
Iteration 115, loss = 58000026.63653474
Iteration 116, loss = 57203989.67789454
Iteration 117, loss = 56417321.37635545
Iteration 118, loss = 55639925.72874756
Iteration 119, loss = 54871707.70481382
Iteration 120, loss = 54112573.23569584
Iteration 121, loss = 53362429.20251665
Iteration 122, loss = 52621183.42506416
Iteration 123, loss = 51888744.65057988
Iteration 124, loss = 51165022.54265586
Iteration 125, loss = 50449927.67024303
Iteration 126, loss = 49743371.49677356
Iteration 127, loss = 49045266.36939914
Iteration 128, loss = 48355525.50834722
Iteration 129, loss = 47674062.99639702
```

```
Iteration 130, loss = 47000793.76847580
Iteration 131, loss = 46335633.60137734
Iteration 132, loss = 45678499.10360270
Iteration 133, loss = 45029307.70532422
Iteration 134, loss = 44387977.64847327
Iteration 135, loss = 43754427.97695119
Iteration 136, loss = 43128578.52696498
Iteration 137, loss = 42510349.91748600
Iteration 138, loss = 41899663.54083306
Iteration 139, loss = 41296441.55337832
Iteration 140, loss = 40700606.86637669
Iteration 141, loss = 40112083.13691761
Iteration 142, loss = 39530794.75899878
Iteration 143, loss = 38956666.85472170
Iteration 144, loss = 38389625.26560767
Iteration 145, loss = 37829596.54403398
Iteration 146, loss = 37276507.94478952
Iteration 147, loss = 36730287.41674873
Iteration 148, loss = 36190863.59466326
Iteration 149, loss = 35658165.79107032
Iteration 150, loss = 35132123.98831711
Iteration 151, loss = 34612668.83069985
Iteration 152, loss = 34099731.61671689
Iteration 153, loss = 33593244.29143488
Iteration 154, loss = 33093139.43896661
Iteration 155, loss = 32599350.27506009
Iteration 156, loss = 32111810.63979723
Iteration 157, loss = 31630454.99040159
Iteration 158, loss = 31155218.39415381
Iteration 159, loss = 30686036.52141373
Iteration 160, loss = 30222845.63874827
Iteration 161, loss = 29765582.60216374
Iteration 162, loss = 29314184.85044175
Iteration 163, loss = 28868590.39857747
Iteration 164, loss = 28428737.83131916
Iteration 165, loss = 27994566.29680800
Iteration 166, loss = 27566015.50031687
Iteration 167, loss = 27143025.69808733
Iteration 168, loss = 26725537.69126338
Iteration 169, loss = 26313492.81992099
Iteration 170, loss = 25906832.95719263
Iteration 171, loss = 25505500.50348486
Iteration 172, loss = 25109438.38078905
Iteration 173, loss = 24718590.02708301
Iteration 174, loss = 24332899.39082321
Iteration 175, loss = 23952310.92552606
Iteration 176, loss = 23576769.58443736
Iteration 177, loss = 23206220.81528873
```

```
Iteration 178, loss = 22840610.55513996
Iteration 179, loss = 22479885.22530622
Iteration 180, loss = 22123991.72636889
Iteration 181, loss = 21772877.43326911
Iteration 182, loss = 21426490.19048291
Iteration 183, loss = 21084778.30727661
Iteration 184, loss = 20747690.55304185
Iteration 185, loss = 20415176.15270870
Iteration 186, loss = 20087184.78223605
Iteration 187, loss = 19763666.56417816
Iteration 188, loss = 19444572.06332613
Iteration 189, loss = 19129852.28242355
Iteration 190, loss = 18819458.65795476
Iteration 191, loss = 18513343.05600524
Iteration 192, loss = 18211457.76819243
Iteration 193, loss = 17913755.50766657
Iteration 194, loss = 17620189.40517988
Iteration 195, loss = 17330713.00522345
Iteration 196, loss = 17045280.26223053
Iteration 197, loss = 16763845.53684537
Iteration 198, loss = 16486363.59225636
Iteration 199, loss = 16212789.59059250
Iteration 200, loss = 15943079.08938219
Iteration 201, loss = 15677188.03807320
Iteration 202, loss = 15415072.77461288
Iteration 203, loss = 15156690.02208753
Iteration 204, loss = 14901996.88541983
Iteration 205, loss = 14650950.84812329
Iteration 206, loss = 14403509.76911299
Iteration 207, loss = 14159631.87957101
Iteration 208, loss = 13919275.77986609
Iteration 209, loss = 13682400.43652623
Iteration 210, loss = 13448965.17926313
Iteration 211, loss = 13218929.69804778
Iteration 212, loss = 12992254.04023575
Iteration 213, loss = 12768898.60774168
Iteration 214, loss = 12548824.15426146
Iteration 215, loss = 12331991.78254160
Iteration 216, loss = 12118362.94169438
Iteration 217, loss = 11907899.42455803
Iteration 218, loss = 11700563.36510100
Iteration 219, loss = 11496317.23586909
Iteration 220, loss = 11295123.84547483
Iteration 221, loss = 11096946.33612777
Iteration 222, loss = 10901748.18120513
Iteration 223, loss = 10709493.18286146
Iteration 224, loss = 10520145.46967664
Iteration 225, loss = 10333669.49434126
```

```
Iteration 226, loss = 10150030.03137824
Iteration 227, loss = 9969192.17490010
Iteration 228, loss = 9791121.33640066
Iteration 229, loss = 9615783.24258047
Iteration 230, loss = 9443143.93320493
Iteration 231, loss = 9273169.75899435
Iteration 232, loss = 9105827.37954490
Iteration 233, loss = 8941083.76127974
Iteration 234, loss = 8778906.17542941
Iteration 235, loss = 8619262.19604050
Iteration 236, loss = 8462119.69801188
Iteration 237, loss = 8307446.85515777
Iteration 238, loss = 8155212.13829647
Iteration 239, loss = 8005384.31336427
Iteration 240, loss = 7857932.43955352
Iteration 241, loss = 7712825.86747420
Iteration 242, loss = 7570034.23733807
Iteration 243, loss = 7429527.47716471
Iteration 244, loss = 7291275.80100865
Iteration 245, loss = 7155249.70720685
Iteration 246, loss = 7021419.97664568
Iteration 247, loss = 6889757.67104687
Iteration 248, loss = 6760234.13127148
Iteration 249, loss = 6632820.97564131
Iteration 250, loss = 6507490.09827702
Iteration 251, loss = 6384213.66745221
Iteration 252, loss = 6262964.12396290
Iteration 253, loss = 6143714.17951161
Iteration 254, loss = 6026436.81510546
Iteration 255, loss = 5911105.27946762
Iteration 256, loss = 5797693.08746152
Iteration 257, loss = 5686174.01852715
Iteration 258, loss = 5576522.11512882
Iteration 259, loss = 5468711.68121388
Iteration 260, loss = 5362717.28068178
Iteration 261, loss = 5258513.73586283
Iteration 262, loss = 5156076.12600620
Iteration 263, loss = 5055379.78577661
Iteration 264, loss = 4956400.30375906
Iteration 265, loss = 4859113.52097133
Iteration 266, loss = 4763495.52938342
Iteration 267, loss = 4669522.67044378
Iteration 268, loss = 4577171.53361160
Iteration 269, loss = 4486418.95489479
Iteration 270, loss = 4397242.01539328
Iteration 271, loss = 4309618.03984703
Iteration 272, loss = 4223524.59518854
Iteration 273, loss = 4138939.48909922
```

```
Iteration 274, loss = 4055840.76856948
Iteration 275, loss = 3974206.71846190
Iteration 276, loss = 3894015.86007734
Iteration 277, loss = 3815246.94972352
Iteration 278, loss = 3737878.97728569
Iteration 279, loss = 3661891.16479921
Iteration 280, loss = 3587262.96502358
Iteration 281, loss = 3513974.06001773
Iteration 282, loss = 3442004.35971622
Iteration 283, loss = 3371334.00050607
Iteration 284, loss = 3301943.34380402
Iteration 285, loss = 3233812.97463394
Iteration 286, loss = 3166923.70020417
Iteration 287, loss = 3101256.54848450
Iteration 288, loss = 3036792.76678271
Iteration 289, loss = 2973513.82032040
Iteration 290, loss = 2911401.39080787
Iteration 291, loss = 2850437.37501807
Iteration 292, loss = 2790603.88335918
Iteration 293, loss = 2731883.23844599
Iteration 294, loss = 2674257.97366970
Iteration 295, loss = 2617710.83176616
Iteration 296, loss = 2562224.76338234
Iteration 297, loss = 2507782.92564105
Iteration 298, loss = 2454368.68070373
Iteration 299, loss = 2401965.59433127
Iteration 300, loss = 2350557.43444279
Iteration 301, loss = 2300128.16967232
Iteration 302, loss = 2250661.96792343
Iteration 303, loss = 2202143.19492152
Iteration 304, loss = 2154556.41276414
Iteration 305, loss = 2107886.37846896
Iteration 306, loss = 2062118.04251962
Iteration 307, loss = 2017236.54740937
Iteration 308, loss = 1973227.22618252
Iteration 309, loss = 1930075.60097378
Iteration 310, loss = 1887767.38154550
Iteration 311, loss = 1846288.46382281
Iteration 312, loss = 1805624.92842679
Iteration 313, loss = 1765763.03920566
Iteration 314, loss = 1726689.24176411
Iteration 315, loss = 1688390.16199086
Iteration 316, loss = 1650852.60458443
Iteration 317, loss = 1614063.55157737
Iteration 318, loss = 1578010.16085891
Iteration 319, loss = 1542679.76469623
Iteration 320, loss = 1508059.86825440
Iteration 321, loss = 1474138.14811514
```

```
Iteration 322, loss = 1440902.45079453
Iteration 323, loss = 1408340.79125980
Iteration 324, loss = 1376441.35144529
Iteration 325, loss = 1345192.47876780
Iteration 326, loss = 1314582.68464139
Iteration 327, loss = 1284600.64299189
Iteration 328, loss = 1255235.18877116
Iteration 329, loss = 1226475.31647136
Iteration 330, loss = 1198310.17863926
Iteration 331, loss = 1170729.08439097
Iteration 332, loss = 1143721.49792706
Iteration 333, loss = 1117277.03704834
Iteration 334, loss = 1091385.47167244
Iteration 335, loss = 1066036.72235145
Iteration 336, loss = 1041220.85879072
Iteration 337, loss = 1016928.09836898
Iteration 338, loss = 993148.80466014
Iteration 339, loss = 969873.48595670
Iteration 340, loss = 947092.79379523
Iteration 341, loss = 924797.52148389
Iteration 342, loss = 902978.60263235
Iteration 343, loss = 881627.10968422
Iteration 344, loss = 860734.25245216
Iteration 345, loss = 840291.37665596
Iteration 346, loss = 820289.96246373
Iteration 347, loss = 800721.62303639
Iteration 348, loss = 781578.10307568
Iteration 349, loss = 762851.27737586
Iteration 350, loss = 744533.14937939
Iteration 351, loss = 726615.84973661
Iteration 352, loss = 709091.63486978
Iteration 353, loss = 691952.88554163
Iteration 354, loss = 675192.10542856
Iteration 355, loss = 658801.91969872
Iteration 356, loss = 642775.07359517
Iteration 357, loss = 627104.43102434
Iteration 358, loss = 611782.97314987
Iteration 359, loss = 596803.79699215
Iteration 360, loss = 582160.11403360
Iteration 361, loss = 567845.24883003
Iteration 362, loss = 553852.63762814
Iteration 363, loss = 540175.82698939
Iteration 364, loss = 526808.47242035
Iteration 365, loss = 513744.33700986
Iteration 366, loss = 500977.29007293
Iteration 367, loss = 488501.30580180
Iteration 368, loss = 476310.46192411
Iteration 369, loss = 464398.93836848
```

```
Iteration 370, loss = 452761.01593757
Iteration 371, loss = 441391.07498890
Iteration 372, loss = 430283.59412335
Iteration 373, loss = 419433.14888177
Iteration 374, loss = 408834.41044960
Iteration 375, loss = 398482.14436981
Iteration 376, loss = 388371.20926420
Iteration 377, loss = 378496.55556322
Iteration 378, loss = 368853.22424446
Iteration 379, loss = 359436.34557990
Iteration 380, loss = 350241.13789204
Iteration 381, loss = 341262.90631912
Iteration 382, loss = 332497.04158938
Iteration 383, loss = 323939.01880461
Iteration 384, loss = 315584.39623305
Iteration 385, loss = 307428.81411169
Iteration 386, loss = 299467.99345817
Iteration 387, loss = 291697.73489232
Iteration 388, loss = 284113.91746734
Iteration 389, loss = 276712.49751095
Iteration 390, loss = 269489.50747626
Iteration 391, loss = 262441.05480279
Iteration 392, loss = 255563.32078740
Iteration 393, loss = 248852.55946539
Iteration 394, loss = 242305.09650185
Iteration 395, loss = 235917.32809313
Iteration 396, loss = 229685.71987872
Iteration 397, loss = 223606.80586342
Iteration 398, loss = 217677.18734999
Iteration 399, loss = 211893.53188219
Iteration 400, loss = 206252.57219835
Iteration 401, loss = 200751.10519546
Iteration 402, loss = 195385.99090386
Iteration 403, loss = 190154.15147250
Iteration 404, loss = 185052.57016484
Iteration 405, loss = 180078.29036542
Iteration 406, loss = 175228.41459706
Iteration 407, loss = 170500.10354879
Iteration 408, loss = 165890.57511450
Iteration 409, loss = 161397.10344220
Iteration 410, loss = 157017.01799412
Iteration 411, loss = 152747.70261745
Iteration 412, loss = 148586.59462584
Iteration 413, loss = 144531.18389154
Iteration 414, loss = 140579.01194838
Iteration 415, loss = 136727.67110530
Iteration 416, loss = 132974.80357065
Iteration 417, loss = 129318.10058713
```

```
Iteration 418, loss = 125755.30157739
Iteration 419, loss = 122284.19330018
Iteration 420, loss = 118902.60901723
Iteration 421, loss = 115608.42767055
Iteration 422, loss = 112399.57307037
Iteration 423, loss = 109274.01309353
Iteration 424, loss = 106229.75889236
Iteration 425, loss = 103264.86411398
Iteration 426, loss = 100377.42412993
Iteration 427, loss = 97565.57527628
Iteration 428, loss = 94827.49410385
Iteration 429, loss = 92161.39663883
Iteration 430, loss = 89565.53765350
Iteration 431, loss = 87038.20994713
Iteration 432, loss = 84577.74363694
Iteration 433, loss = 82182.50545913
Iteration 434, loss = 79850.89807983
Iteration 435, loss = 77581.35941599
Iteration 436, loss = 75372.36196608
Iteration 437, loss = 73222.41215063
Iteration 438, loss = 71130.04966240
Iteration 439, loss = 69093.84682628
Iteration 440, loss = 67112.40796870
Iteration 441, loss = 65184.36879657
Iteration 442, loss = 63308.39578567
Iteration 443, loss = 61483.18557834
Iteration 444, loss = 59707.46439052
Iteration 445, loss = 57979.98742797
Iteration 446, loss = 56299.53831158
Iteration 447, loss = 54664.92851180
Iteration 448, loss = 53074.99679197
Iteration 449, loss = 51528.60866057
Iteration 450, loss = 50024.65583232
Iteration 451, loss = 48562.05569790
Iteration 452, loss = 47139.75080239
Iteration 453, loss = 45756.70833227
Iteration 454, loss = 44411.91961082
Iteration 455, loss = 43104.39960198
Iteration 456, loss = 41833.18642246
Iteration 457, loss = 40597.34086208
Iteration 458, loss = 39395.94591224
Iteration 459, loss = 38228.10630241
Iteration 460, loss = 37092.94804455
Iteration 461, loss = 35989.61798546
Iteration 462, loss = 34917.28336680
Iteration 463, loss = 33875.13139285
Iteration 464, loss = 32862.36880589
Iteration 465, loss = 31878.22146896
```

```
Iteration 466, loss = 30921.93395619
Iteration 467, loss = 29992.76915032
Iteration 468, loss = 29090.00784751
Iteration 469, loss = 28212.94836927
Iteration 470, loss = 27360.90618145
Iteration 471, loss = 26533.21352021
Iteration 472, loss = 25729.21902479
Iteration 473, loss = 24948.28737715
Iteration 474, loss = 24189.79894821
Iteration 475, loss = 23453.14945077
Iteration 476, loss = 22737.74959887
Iteration 477, loss = 22043.02477364
Iteration 478, loss = 21368.41469543
Iteration 479, loss = 20713.37310218
Iteration 480, loss = 20077.36743402
Iteration 481, loss = 19459.87852383
Iteration 482, loss = 18860.40029387
Iteration 483, loss = 18278.43945826
Iteration 484, loss = 17713.51523127
Iteration 485, loss = 17165.15904135
Iteration 486, loss = 16632.91425079
Iteration 487, loss = 16116.33588089
Iteration 488, loss = 15614.99034268
Iteration 489, loss = 15128.45517292
Iteration 490, loss = 14656.31877549
Iteration 491, loss = 14198.18016796
Iteration 492, loss = 13753.64873324
Iteration 493, loss = 13322.34397640
Iteration 494, loss = 12903.89528636
Iteration 495, loss = 12497.94170249
Iteration 496, loss = 12104.13168608
Iteration 497, loss = 11722.12289648
Iteration 498, loss = 11351.58197190
Iteration 499, loss = 10992.18431482
Iteration 500, loss = 10643.61388190
Iteration 501, loss = 10305.56297827
Iteration 502, loss = 9977.73205619
Iteration 503, loss = 9659.82951804
Iteration 504, loss = 9351.57152342
Iteration 505, loss = 9052.68180043
Iteration 506, loss = 8762.89146102
Iteration 507, loss = 8481.93882028
Iteration 508, loss = 8209.56921969
Iteration 509, loss = 7945.53485416
Iteration 510, loss = 7689.59460290
Iteration 511, loss = 7441.51386399
Iteration 512, loss = 7201.06439254
Iteration 513, loss = 6968.02414249
```

```
Iteration 514, loss = 6742.17711187
Iteration 515, loss = 6523.31319152
Iteration 516, loss = 6311.22801718
Iteration 517, loss = 6105.72282494
Iteration 518, loss = 5906.60430988
Iteration 519, loss = 5713.68448791
Iteration 520, loss = 5526.78056081
Iteration 521, loss = 5345.71478425
Iteration 522, loss = 5170.31433887
Iteration 523, loss = 5000.41120431
Iteration 524, loss = 4835.84203615
Iteration 525, loss = 4676.44804563
Iteration 526, loss = 4522.07488222
Iteration 527, loss = 4372.57251891
Iteration 528, loss = 4227.79514010
Iteration 529, loss = 4087.60103223
Iteration 530, loss = 3951.85247687
Iteration 531, loss = 3820.41564638
Iteration 532, loss = 3693.16050206
Iteration 533, loss = 3569.96069462
Iteration 534, loss = 3450.69346713
Iteration 535, loss = 3335.23956019
Iteration 536, loss = 3223.48311945
Iteration 537, loss = 3115.31160529
Iteration 538, loss = 3010.61570469
Iteration 539, loss = 2909.28924525
Iteration 540, loss = 2811.22911130
Iteration 541, loss = 2716.33516199
Iteration 542, loss = 2624.51015145
Iteration 543, loss = 2535.65965087
Iteration 544, loss = 2449.69197248
Iteration 545, loss = 2366.51809544
Iteration 546, loss = 2286.05159353
Iteration 547, loss = 2208.20856462
Iteration 548, loss = 2132.90756195
Iteration 549, loss = 2060.06952707
Iteration 550, loss = 1989.61772442
Iteration 551, loss = 1921.47767770
Iteration 552, loss = 1855.57710766
Iteration 553, loss = 1791.84587156
Iteration 554, loss = 1730.21590415
Iteration 555, loss = 1670.62116015
Iteration 556, loss = 1612.99755814
Iteration 557, loss = 1557.28292595
Iteration 558, loss = 1503.41694744
Iteration 559, loss = 1451.34111057
Iteration 560, loss = 1400.99865690
Iteration 561, loss = 1352.33453235
```

```
Iteration 562, loss = 1305.29533923
Iteration 563, loss = 1259.82928952
Iteration 564, loss = 1215.88615936
Iteration 565, loss = 1173.41724478
Iteration 566, loss = 1132.37531848
Iteration 567, loss = 1092.71458785
Iteration 568, loss = 1054.39065402
Iteration 569, loss = 1017.36047206
Iteration 570, loss = 981.58231216
Iteration 571, loss = 947.01572189
Iteration 572, loss = 913.62148942
Iteration 573, loss = 881.36160780
Iteration 574, loss = 850.19924007
Iteration 575, loss = 820.09868544
Iteration 576, loss = 791.02534627
Iteration 577, loss = 762.94569600
Iteration 578, loss = 735.82724796
Iteration 579, loss = 709.63852491
Iteration 580, loss = 684.34902960
Iteration 581, loss = 659.92921592
Iteration 582, loss = 636.35046102
Iteration 583, loss = 613.58503807
Iteration 584, loss = 591.60608979
Iteration 585, loss = 570.38760279
Iteration 586, loss = 549.90438248
Iteration 587, loss = 530.13202881
Iteration 588, loss = 511.04691257
Iteration 589, loss = 492.62615241
Iteration 590, loss = 474.84759250
Iteration 591, loss = 457.68978079
Iteration 592, loss = 441.13194787
Iteration 593, loss = 425.15398647
Iteration 594, loss = 409.73643148
Iteration 595, loss = 394.86044056
Iteration 596, loss = 380.50777530
Iteration 597, loss = 366.66078291
Iteration 598, loss = 353.30237844
Iteration 599, loss = 340.41602747
Iteration 600, loss = 327.98572931
Iteration 601, loss = 315.99600072
Iteration 602, loss = 304.43186004
Iteration 603, loss = 293.27881180
Iteration 604, loss = 282.52283178
Iteration 605, loss = 272.15035248
Iteration 606, loss = 262.14824901
Iteration 607, loss = 252.50382546
Iteration 608, loss = 243.20480151
Iteration 609, loss = 234.23929961
```

```
Iteration 610, loss = 225.59583240
Iteration 611, loss = 217.26329057
Iteration 612, loss = 209.23093106
Iteration 613, loss = 201.48836554
Iteration 614, loss = 194.02554939
Iteration 615, loss = 186.83277080
Iteration 616, loss = 179.90064037
Iteration 617, loss = 173.22008088
Iteration 618, loss = 166.78231749
Iteration 619, loss = 160.57886813
Iteration 620, loss = 154.60153421
Iteration 621, loss = 148.84239165
Iteration 622, loss = 143.29378209
Iteration 623, loss = 137.94830445
Iteration 624, loss = 132.79880671
Iteration 625, loss = 127.83837791
Iteration 626, loss = 123.06034045
Iteration 627, loss = 118.45824256
Iteration 628, loss = 114.02585104
Iteration 629, loss = 109.75714423
Iteration 630, loss = 105.64630516
Iteration 631, loss = 101.68771489
Iteration 632, loss = 97.87594616
Iteration 633, loss = 94.20575709
Iteration 634, loss = 90.67208520
Iteration 635, loss = 87.27004149
Iteration 636, loss = 83.99490487
Iteration 637, loss = 80.84211657
Iteration 638, loss = 77.80727488
Iteration 639, loss = 74.88612996
Iteration 640, loss = 72.07457883
Iteration 641, loss = 69.36866058
Iteration 642, loss = 66.76455162
Iteration 643, loss = 64.25856116
Iteration 644, loss = 61.84712683
Iteration 645, loss = 59.52681036
Iteration 646, loss = 57.29429350
Iteration 647, loss = 55.14637401
Iteration 648, loss = 53.07996178
Iteration 649, loss = 51.09207507
Iteration 650, loss = 49.17983688
Iteration 651, loss = 47.34047147
Iteration 652, loss = 45.57130091
Iteration 653, loss = 43.86974181
Iteration 654, loss = 42.23330212
Iteration 655, loss = 40.65957804
Iteration 656, loss = 39.14625104
Iteration 657, loss = 37.69108499
```

```
Iteration 658, loss = 36.29192329
Iteration 659, loss = 34.94668623
Iteration 660, loss = 33.65336836
Iteration 661, loss = 32.41003591
Iteration 662, loss = 31.21482439
Iteration 663, loss = 30.06593616
Iteration 664, loss = 28.96163821
Iteration 665, loss = 27.90025984
Iteration 666, loss = 26.88019061
Iteration 667, loss = 25.89987820
Iteration 668, loss = 24.95782640
Iteration 669, loss = 24.05259320
Iteration 670, loss = 23.18278889
Iteration 671, loss = 22.34707425
Iteration 672, loss = 21.54415877
Iteration 673, loss = 20.77279899
Iteration 674, loss = 20.03179681
Iteration 675, loss = 19.31999794
Iteration 676, loss = 18.63629037
Iteration 677, loss = 17.97960284
Iteration 678, loss = 17.34890344
Iteration 679, loss = 16.74319825
Iteration 680, loss = 16.16152992
Iteration 681, loss = 15.60297645
Iteration 682, loss = 15.06664993
Iteration 683, loss = 14.55169529
Iteration 684, loss = 14.05728917
Iteration 685, loss = 13.58263877
Iteration 686, loss = 13.12698079
Iteration 687, loss = 12.68958034
Iteration 688, loss = 12.26972996
Iteration 689, loss = 11.86674859
Iteration 690, loss = 11.47998069
Iteration 691, loss = 11.10879525
Iteration 692, loss = 10.75258498
Iteration 693, loss = 10.41076537
Iteration 694, loss = 10.08277396
Iteration 695, loss = 9.76806948
Iteration 696, loss = 9.46613108
Iteration 697, loss = 9.17645763
Iteration 698, loss = 8.89856698
Iteration 699, loss = 8.63199525
Iteration 700, loss = 8.37629621
Iteration 701, loss = 8.13104060
Iteration 702, loss = 7.89581551
Iteration 703, loss = 7.67022378
Iteration 704, loss = 7.45388346
Iteration 705, loss = 7.24642719
```

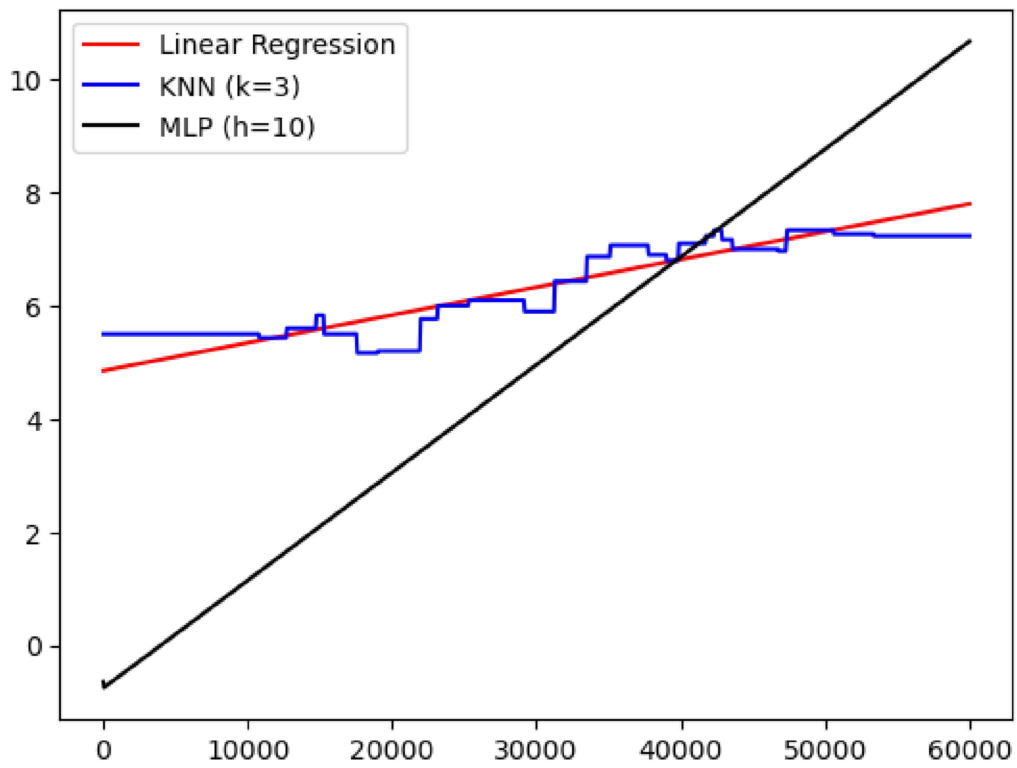
```
Iteration 706, loss = 7.04750172
Iteration 707, loss = 6.85676735
Iteration 708, loss = 6.67389744
Iteration 709, loss = 6.49857793
Iteration 710, loss = 6.33050688
Iteration 711, loss = 6.16939401
Iteration 712, loss = 6.01496026
Iteration 713, loss = 5.86693741
Iteration 714, loss = 5.72506761
Iteration 715, loss = 5.58910306
Iteration 716, loss = 5.45880559
Iteration 717, loss = 5.33394630
Iteration 718, loss = 5.21430525
Iteration 719, loss = 5.09967110
Iteration 720, loss = 4.98984074
Iteration 721, loss = 4.88461908
Iteration 722, loss = 4.78381866
Iteration 723, loss = 4.68725939
Iteration 724, loss = 4.59476828
Iteration 725, loss = 4.50617917
Iteration 726, loss = 4.42133244
Iteration 727, loss = 4.34007482
Iteration 728, loss = 4.26225908
Iteration 729, loss = 4.18774384
Iteration 730, loss = 4.11639335
Iteration 731, loss = 4.04807724
Iteration 732, loss = 3.98267036
Iteration 733, loss = 3.92005253
Iteration 734, loss = 3.86010840
Iteration 735, loss = 3.80272721
Iteration 736, loss = 3.74780266
Iteration 737, loss = 3.69523270
Iteration 738, loss = 3.64491941
Iteration 739, loss = 3.59676877
Iteration 740, loss = 3.55069060
Iteration 741, loss = 3.50659832
Iteration 742, loss = 3.46440888
Iteration 743, loss = 3.42404257
Iteration 744, loss = 3.38542293
Iteration 745, loss = 3.34847660
Iteration 746, loss = 3.31313320
Iteration 747, loss = 3.27932523
Iteration 748, loss = 3.24698792
Iteration 749, loss = 3.21605916
Iteration 750, loss = 3.18647939
Iteration 751, loss = 3.15819148
Iteration 752, loss = 3.13114064
Iteration 753, loss = 3.10527433
```

```
Iteration 754, loss = 3.08054219
Iteration 755, loss = 3.05689591
Iteration 756, loss = 3.03428920
Iteration 757, loss = 3.01267766
Iteration 758, loss = 2.99201873
Iteration 759, loss = 2.97227164
Iteration 760, loss = 2.95339728
Iteration 761, loss = 2.93535818
Iteration 762, loss = 2.91811843
Iteration 763, loss = 2.90164361
Iteration 764, loss = 2.88590075
Iteration 765, loss = 2.87085825
Iteration 766, loss = 2.85648582
Iteration 767, loss = 2.84275446
Iteration 768, loss = 2.82963639
Iteration 769, loss = 2.81710497
Iteration 770, loss = 2.80513471
Iteration 771, loss = 2.79370119
Iteration 772, loss = 2.78278100
Iteration 773, loss = 2.77235175
Iteration 774, loss = 2.76239198
Iteration 775, loss = 2.75288113
Iteration 776, loss = 2.74379955
Iteration 777, loss = 2.73512838
Iteration 778, loss = 2.72684959
Iteration 779, loss = 2.71894593
Iteration 780, loss = 2.71140085
Iteration 781, loss = 2.70419854
Iteration 782, loss = 2.69732385
Iteration 783, loss = 2.69076229
Iteration 784, loss = 2.68449998
Iteration 785, loss = 2.67852366
Iteration 786, loss = 2.67282061
Iteration 787, loss = 2.66737867
Iteration 788, loss = 2.66218623
Iteration 789, loss = 2.65723215
Iteration 790, loss = 2.65250578
Iteration 791, loss = 2.64799694
Iteration 792, loss = 2.64369589
Iteration 793, loss = 2.63959330
Iteration 794, loss = 2.63568028
Iteration 795, loss = 2.63194829
Iteration 796, loss = 2.62838918
Iteration 797, loss = 2.62499516
Iteration 798, loss = 2.62175877
Iteration 799, loss = 2.61867289
Iteration 800, loss = 2.61573071
Iteration 801, loss = 2.61292570
```

```
Iteration 802, loss = 2.61025165
Iteration 803, loss = 2.60770260
Iteration 804, loss = 2.60527286
Iteration 805, loss = 2.60295700
Iteration 806, loss = 2.60074981
Iteration 807, loss = 2.59864634
Iteration 808, loss = 2.59664183
Iteration 809, loss = 2.59473175
Iteration 810, loss = 2.59291178
Iteration 811, loss = 2.59117776
Iteration 812, loss = 2.58952574
Iteration 813, loss = 2.58795195
Iteration 814, loss = 2.58645278
Iteration 815, loss = 2.58502479
Iteration 816, loss = 2.58366467
Iteration 817, loss = 2.58236929
Iteration 818, loss = 2.58113564
Iteration 819, loss = 2.57996086
Iteration 820, loss = 2.57884221
Iteration 821, loss = 2.57777708
Iteration 822, loss = 2.57676297
Iteration 823, loss = 2.57579749
Iteration 824, loss = 2.57487839
Iteration 825, loss = 2.57400348
Iteration 826, loss = 2.57317069
Iteration 827, loss = 2.57237805
Iteration 828, loss = 2.57162367
Iteration 829, loss = 2.57090575
Iteration 830, loss = 2.57022257
Iteration 831, loss = 2.56957249
Iteration 832, loss = 2.56895395
Iteration 833, loss = 2.56836545
Iteration 834, loss = 2.56780558
Iteration 835, loss = 2.56727297
Iteration 836, loss = 2.56676634
Iteration 837, loss = 2.56628443
Iteration 838, loss = 2.56582608
Iteration 839, loss = 2.56539017
Iteration 840, loss = 2.56497562
Iteration 841, loss = 2.56458140
Iteration 842, loss = 2.56420656
Iteration 843, loss = 2.56385015
Iteration 844, loss = 2.56351129
Iteration 845, loss = 2.56318914
Iteration 846, loss = 2.56288290
Iteration 847, loss = 2.56259180
Iteration 848, loss = 2.56231510
Iteration 849, loss = 2.56205211
```

```
Iteration 850, loss = 2.56180217
Iteration 851, loss = 2.56156465
Iteration 852, loss = 2.56133894
Iteration 853, loss = 2.56112447
Iteration 854, loss = 2.56092069
Iteration 855, loss = 2.56072708
Iteration 856, loss = 2.56054314
Iteration 857, loss = 2.56036841
Iteration 858, loss = 2.56020244
Iteration 859, loss = 2.56004478
Iteration 860, loss = 2.55989504
Iteration 861, loss = 2.55975283
Iteration 862, loss = 2.55961778
Iteration 863, loss = 2.55948953
Iteration 864, loss = 2.55936775
Iteration 865, loss = 2.55925212
Iteration 866, loss = 2.55914234
Iteration 867, loss = 2.55903811
Iteration 868, loss = 2.55893916
Iteration 869, loss = 2.55884524
Iteration 870, loss = 2.55875608
Iteration 871, loss = 2.55867145
Iteration 872, loss = 2.55859114
Iteration 873, loss = 2.55851491
Iteration 874, loss = 2.55844257
Iteration 875, loss = 2.55837392
Iteration 876, loss = 2.55830879
Iteration 877, loss = 2.55824698
Iteration 878, loss = 2.55818834
Iteration 879, loss = 2.55813270
Iteration 880, loss = 2.55807992
Iteration 881, loss = 2.55802985
Iteration 882, loss = 2.55798235
Iteration 883, loss = 2.55793730
Iteration 884, loss = 2.55789456
Iteration 885, loss = 2.55785403
Iteration 886, loss = 2.55781559
Iteration 887, loss = 2.55777913
Iteration 888, loss = 2.55774456
Iteration 889, loss = 2.55771177
Iteration 890, loss = 2.55768068
Iteration 891, loss = 2.55765120
Iteration 892, loss = 2.55762324
Iteration 893, loss = 2.55759674
Iteration 894, loss = 2.55757161
Iteration 895, loss = 2.55754778
Iteration 896, loss = 2.55752519
Iteration 897, loss = 2.55750377
```

```
Iteration 898, loss = 2.55748346
Iteration 899, loss = 2.55746422
Iteration 900, loss = 2.55744597
Iteration 901, loss = 2.55742867
Iteration 902, loss = 2.55741227
Iteration 903, loss = 2.55739672
Iteration 904, loss = 2.55738198
Iteration 905, loss = 2.55736801
Iteration 906, loss = 2.55735476
Iteration 907, loss = 2.55734221
Iteration 908, loss = 2.55733030
Iteration 909, loss = 2.55731902
Iteration 910, loss = 2.55730832
Iteration 911, loss = 2.55729817
Iteration 912, loss = 2.55728855
Iteration 913, loss = 2.55727943
Iteration 914, loss = 2.55727079
Iteration 915, loss = 2.55726259
Iteration 916, loss = 2.55725481
Iteration 917, loss = 2.55724743
Iteration 918, loss = 2.55724044
Iteration 919, loss = 2.55723380
Iteration 920, loss = 2.55722751
Iteration 921, loss = 2.55722154
Iteration 922, loss = 2.55721587
Training loss did not improve more than tol=0.000010 for 10 consecutive epochs.
Stopping.
```




```
Predictions for Cyprus (GDP = 22587 USD): 3.544095102346475
KNN score (k=3)    : 0.8525732853499179
Linear Regression score: 0.7344414355437031
MLP score          : -6.514105598162307
```

2.4.1 Answers

Can the score for MLP be compared with LinReg and KNN?

Yes, in the docs for the MLP's score function, it is calculated the same way as they are in both other fits, so they can be compared. For the MLP, we get a pretty bad fit, as we can see in both the negative R2 score, and by looking at the plot.

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html#sklearn.neural_

3 Cost Function

3.1 Qa Given the following $\mathbf{x}^{(i)}$'s, construct and print the X matrix in python.

Definitions of $x^{(i)}$'s is stripped from this pdf - can be found in original exercise notebook.

[13]:
```
import numpy as np

y_true = np.array([1,2,3,4]) # NOTE: you'll need this later
```

```
X = np.array([[1,2,3],[4,2,1],[3,8,5],[-9,-1,0]])

print("X_true = \n", X)
```

```
X_true =
 [[ 1  2  3]
 [ 4  2  1]
 [ 3  8  5]
 [-9 -1  0]]
```

3.2 Qb Implement the L1 and L2 norms for vectors in python.

Defined without using any methods from libraries, og python primitives.

[14]:
```
import math

def L1(x):
    if x.ndim != 1:
        raise ValueError("expected x to be of ndim=1, got ndim=",X.ndim)
    sum = 0
    for i in x:
        if i > 0:
            sum += i
        else:
            sum += -i
    return sum


def L2(x):
    if x.ndim != 1:
        raise ValueError("expected x to be of ndim=1, got ndim=",X.ndim)
    sum = 0
    for i in x:
        sum += i**2
    sum = sum**0.5
    return sum

def L2Dot(x):
    assert x.ndim == 1 and isinstance(x, np.ndarray)
    return x.dot(x)**0.5

# TEST vectors: here I test your implementation...calling your L1() and L2()␣
 ↪functions
tx=np.array([1, 2, 3, -1])
ty=np.array([3,-1, 4,  1])

expected_d1=8.0
expected_d2=4.242640687119285
```

```
d1=L1(tx-ty)
d2=L2(tx-ty)

print(f"tx-ty={tx-ty}, d1-expected_d1={d1-expected_d1},␣
 ↪d2-expected_d2={d2-expected_d2}")

eps=1E-9
assert math.fabs(d1-expected_d1)<eps, "L1 dist seems to be wrong"
assert math.fabs(d2-expected_d2)<eps, "L2 dist seems to be wrong"

print("OK(part-1)")

d2dot=L2Dot(tx-ty)
print("d2dot-expected_d2=",d2dot-expected_d2)
assert math.fabs(d2dot-expected_d2)<eps, "L2Ddot dist seem to be wrong"
print("OK(part-2)")
```

```
tx-ty=[-2  3 -1 -2], d1-expected_d1=0.0, d2-expected_d2=0.0
OK(part-1)
d2dot-expected_d2= 0.0
OK(part-2)
```

3.3 Qc Construct the Root Mean Square Error (RMSE) function (Equation 2-1 [HOML]).

[15]:
```
def RMSE(y_pred, y_true):
    assert len(y_pred) == len(y_true) and y_pred.ndim == 1 and y_true.ndim == 1
    err_vec = y_pred - y_true
    l2 = L2(err_vec)
    return l2 / len(err_vec)**0.5

# Dummy h function:
def h(X):
    if X.ndim!=2:
        raise ValueError("excpeted X to be of ndim=2, got ndim=",X.ndim)
    if X.shape[0]==0 or X.shape[1]==0:
        raise ValueError("X got zero data along the 0/1 axis, cannot continue")
    return X[:,0]

# Calls your RMSE() function:
r=RMSE(h(X), y_true)

eps=1E-9
expected=6.57647321898295
print(f"RMSE={r}, diff={r-expected}")
assert math.fabs(r-expected)<eps, "your RMSE dist seems to be wrong"
```

```
print("OK")
```

```
RMSE=6.576473218982953, diff=2.6645352591003757e-15
OK
```

3.4 Qd Similar construct the Mean Absolute Error (MAE) function (Equation 2-2 [HOML]) and evaluate it.

[16]:
```
def MAE(y_pred, y_true):
    assert len(y_pred) == len(y_true) and y_pred.ndim == 1 and y_true.ndim == 1
    err_vec = y_pred - y_true
    return L1(err_vec) / len(err_vec)


# Calls your MAE function:
r=MAE(h(X), y_true)

# TEST vector:
expected=3.75
print(f"MAE={r}, diff={r-expected}")
assert math.fabs(r-expected)<eps, "MAE dist seems to be wrong"

print("OK")
```

```
MAE=3.75, diff=0.0
OK
```

3.5 Qe Robust Code

The functions above in this journal section are already made robust with asserts.

3.6 Qf Conclusion

3.6.1 Answer

Cost functions are at the basis of what the algorithms view as a success or not. A cost function is essentially, what the measure the machine should look for when training, or try to minimize. What lies behind the cost function is therefore essential. The L1 and L2 calculations are some of the key ingredients in the basic cost functions, and understanding how they work, and what they represent is key.

When L1 is used, the absolute errors are simply summed up. But the L2 squares all of the errors, which means it is more sensitive to outlier data, and is more punishing at larger errors.

4 Dummy Classifier

4.1 Qa Load and display the MNIST data

We create the MNIST_GetDataSet() and MNIST_PlotDigit() functions, so they can be reused later.

[17]:
```
import matplotlib.pyplot as plt
import matplotlib
from sklearn.datasets import fetch_openml

def MNIIST_GetDataSet():
    X, y = fetch_openml('mnist_784', return_X_y=True, cache=True,␣
 ↪as_frame=False)
    return X, y

X, y = MNIIST_GetDataSet()
print(f"Shape of X: {X.shape}")
```

```
Shape of X: (70000, 784)
```

[18]:
```
def MNIST_PlotDigit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = matplotlib.cm.binary)
    plt.axis("off")

MNIST_PlotDigit(X[0])
plt.show()
```

4.2 Qb Add a Stochastic Gradient Decent [SGD] Classifier

Below the data is split into train and test sets, and the SGD classifier is trained.

[19]:
```
from sklearn.linear_model import SGDClassifier

X_train, X_test, y_train, y_test = X[:50000], X[50000:], y[:50000], y[50000:]

y_train_5 = (y_train == '5')
y_test_5 = (y_test == '5')

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)

sgd_predict_5 = sgd_clf.predict(X_test)
```

In order to print some correctly classified and incorrectly classified digits, we first get some of the indecies of these digits:

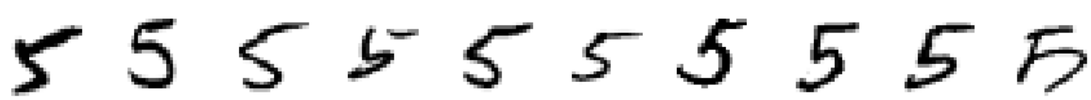
[20]:
```
true_positives = []
false_positives = []
false_negatives = []
for i in range(len(y_test_5)):
    if y_test_5[i] == True and sgd_predict_5[i] == True:
```

```
        true_positives.append(i)
    elif y_test_5[i] == False and sgd_predict_5[i] == True:
        false_positives.append(i)
    elif y_test_5[i] == True and sgd_predict_5[i] == False:
        false_negatives.append(i)
print(f"Number of true positives : {len(true_positives)}")
print(f"Number of false positives: {len(false_positives)}")
print(f"Number of false negatives: {len(false_negatives)}")
```

```
Number of true positives : 1523
Number of false positives: 615
Number of false negatives: 284
```

Now we plot some digits:

[21]:
```
# Plotting some true positives:
def PlotMultiple(amount, indicies, X):
    fig, axes = plt.subplots(1, amount, figsize=(12, 4))
    for i in range(amount):
        plt.subplot(1, amount, i + 1)
        MNIST_PlotDigit(X[indicies[i]])

    plt.tight_layout()
    plt.show()

PlotMultiple(10, true_positives, X_test)
```



[22]:
```
# Plotting some false positives
PlotMultiple(10, false_positives, X_test)
```

[23]:
```
# Plotting some false negatives
PlotMultiple(10, false_negatives, X_test)
```

4.3 Qc Implement a dummy binary classifier

Below is a (very stupid) DummyClassifier, that simply takes in a 'strategy' and classifies everything as that strategy:

[24]:
```python
from sklearn.metrics import accuracy_score
from sklearn.base import BaseEstimator, ClassifierMixin
import numpy as np

class DummyClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, strategy):
        self.strategy = strategy

    def fit(self, X, y=None):
        # Actually do nothing
        return self

    def predict(self, X):
        n_samples = X.shape[0]
        return np.full(n_samples, self.strategy)

    def score(self, X, y):
        y_pred = self.predict(X)
        return accuracy_score(y, y_pred)
```

Testing the classifier and printing the score:

[25]:
```python
dummy = DummyClassifier(False)
dummy.fit(X_train, y_train_5)

dummy_pred = dummy.predict(X_test)
dummy_score = dummy.score(X_test, y_test_5)
print(f"dummy_score: {dummy_score}")
```

```
dummy_score: 0.90965
```

4.3.1 Comparison with book result

Our score is 0.909, which fits perfectly with the books result of 0.909 as well. They are of course compatible as both classifiers are just guessing false on all images. This should theoretically give us an accuraccy of 90%, which fits

(10 numbers, 1 of those is 5…, 90%)

4.4 Qd Conclusion

4.4.1 Answer

In general, this exercise was mostly about getting the basic machine learning and sklearn techniques into our fingers. Splitting the MNIST test set for common ML best practices, and creating a very basic binary classifier adds greatly to the understanding of how ML works - and also that it is not magic. Even a fairly simple human catagorization of “5 or not 5” turns out to be not so easy, as can be seen on the plots of the false positives and false negatives. This was quite revealing, as some of the 5’s looks like they should be easily recognizable.

Also, this exercise gave some insight into how python classes work together with the sklearn library, when creating our own classifier. Something that might be useful when a very specific type of classification is needed, or later in the course.

Lastly, we can also conclude that it is important to think about what kind of data and classification that you are working with. Just because you have an accuracy of 90%, does not necessarily make the model good, as we see when using the dummy classifier.

5 Performance Metrics

5.1 Qa Implement the Accuracy function and test it on the MNIST data.

We added the assert at the top of `UnpackPerfMetrics()`, to make sure the denom is above 0.

[26]:
```
import math

def UnpackPerfMetrics(y_true, y_pred):
    assert y_true.shape == y_pred.shape and y_true.shape[0] > 0
    TP, TN, FP, FN = 0, 0, 0, 0
    for i, _ in enumerate(y_pred):
        if y_true[i] == True and y_pred[i] == True:
            TP += 1
        elif y_true[i] == False and y_pred[i] == False:
            TN += 1
        elif y_true[i] == True and y_pred[i] == False:
            FN += 1
        else:
            FP += 1
    return TP, TN, FP, FN

def MyAccuracy(y_true, y_pred):
    TP, TN, FP, FN = UnpackPerfMetrics(y_true, y_pred)
    accuracy = (TP + TN) / y_true.shape[0]
    return accuracy

from sklearn.metrics import accuracy_score

def TestAccuracy(y_true, y_pred):
```

```
    a0=MyAccuracy(y_true, y_pred)
    a1=accuracy_score(y_true, y_pred)

    print(f"MyAccuracy     = {a0}")
    print(f"accuracy_score = {a1}")

    eps = 1E-9
    if math.fabs(a0 - a1) > eps:
        raise ValueError("Difference in MyAccuracy and accuracy_score too big!")

TestAccuracy(y_test_5, sgd_predict_5)
TestAccuracy(y_test_5, dummy_pred)
```

```
MyAccuracy     = 0.95505
accuracy_score = 0.95505
MyAccuracy     = 0.90965
accuracy_score = 0.90965
```

5.2 Qb Implement Precision, Recall and F_1-score and test it on the MNIST data for both the SGD and Dummy classifier models

Check for denom = 0 is in `UnpackPerfMetrics()`, as shown in Qa.

[27]:
```
from sklearn.metrics import precision_score, recall_score, f1_score
def MyPrecision(y_true, y_pred):
    TP, TN, FP, FN = UnpackPerfMetrics(y_true, y_pred)
    if TP + FP == 0: return 0.0
    return TP / (TP + FP)

def MyRecall(y_true, y_pred):
    TP, TN, FP, FN = UnpackPerfMetrics(y_true, y_pred)
    if TP + FN == 0: return 0.0
    return TP / (TP + FN)

def MyF1Score(y_true, y_pred):
    precision = MyPrecision(y_true, y_pred)
    recall = MyRecall(y_true, y_pred)
    if precision == 0 or recall == 0: return 0.0
    return 2 / (1/precision + 1/recall)

def TestMetrics(y_true, y_pred):
    p0 = MyPrecision(y_true, y_pred)
    p1 = precision_score(y_true, y_pred)

    r0 = MyRecall(y_true, y_pred)
    r1 = recall_score(y_true, y_pred)

    f1_0 = MyF1Score(y_true, y_pred)
```

```python
    f1_1 = f1_score(y_true, y_pred)

    eps = 1E-9

    print(f"MyPrecision     = {p0}")
    print(f"precision_score = {p1}")
    if math.fabs(p0 - p1) > eps:
        raise ValueError("Difference in MyPrecision and precision_score too big!
↪")

    print(f"MyRecall        = {r0}")
    print(f"recall_score    = {r1}")
    if math.fabs(r0 - r1) > eps:
        raise ValueError("Difference in MyRecall and recall_score too big!")

    print(f"MyF1Score       = {f1_0}")
    print(f"f1_score        = {f1_1}")
    if math.fabs(f1_0 - f1_1) > eps:
        raise ValueError("Difference in MyF1Score and f1_score too big!")

print("SGD Performance Metrics")
TestMetrics(y_test_5, sgd_predict_5)
print("=============")
print("Dummy Performance Metrics")
TestMetrics(y_test_5, dummy_pred)
```

```
SGD Performance Metrics
MyPrecision     = 0.7123479887745556
precision_score = 0.7123479887745556
MyRecall        = 0.8428334255672385
recall_score    = 0.8428334255672385
MyF1Score       = 0.7721166032953104
f1_score        = 0.7721166032953105
=============
Dummy Performance Metrics
```

```
/home/lassebp7/anaconda3/lib/python3.13/site-
packages/sklearn/metrics/_classification.py:1565: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 due to no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
MyPrecision     = 0.0
precision_score = 0.0
MyRecall        = 0.0
recall_score    = 0.0
MyF1Score       = 0.0
f1_score        = 0.0
```

5.3 Qc The Confusion Matrix

[28]:
```python
from sklearn.metrics import confusion_matrix

M_dummy = confusion_matrix(y_test_5, dummy_pred)
M_SGD = confusion_matrix(y_test_5, sgd_predict_5)

print("M_dummy:\n", M_dummy)
print("M_SGD:\n", M_SGD)

M_dummy_swapped = confusion_matrix(dummy_pred, y_test_5)
M_SGD_swapped = confusion_matrix(sgd_predict_5, y_test_5)

print("M_dummy_swapped:\n", M_dummy_swapped)
print("M_SGD_swapped:\n", M_SGD_swapped)
```

```
M_dummy:
 [[18193     0]
 [ 1807     0]]
M_SGD:
 [[17578   615]
 [  284  1523]]
M_dummy_swapped:
 [[18193  1807]
 [    0     0]]
M_SGD_swapped:
 [[17578   284]
 [  615  1523]]
```

5.3.1 Answer

From the result of running the code, the matrix returned from confusion matrix must be

[[TN, FP]

[FN, TP]]

As the dummy model never predicts true, no positives are to be found, and the second column will be all 0's

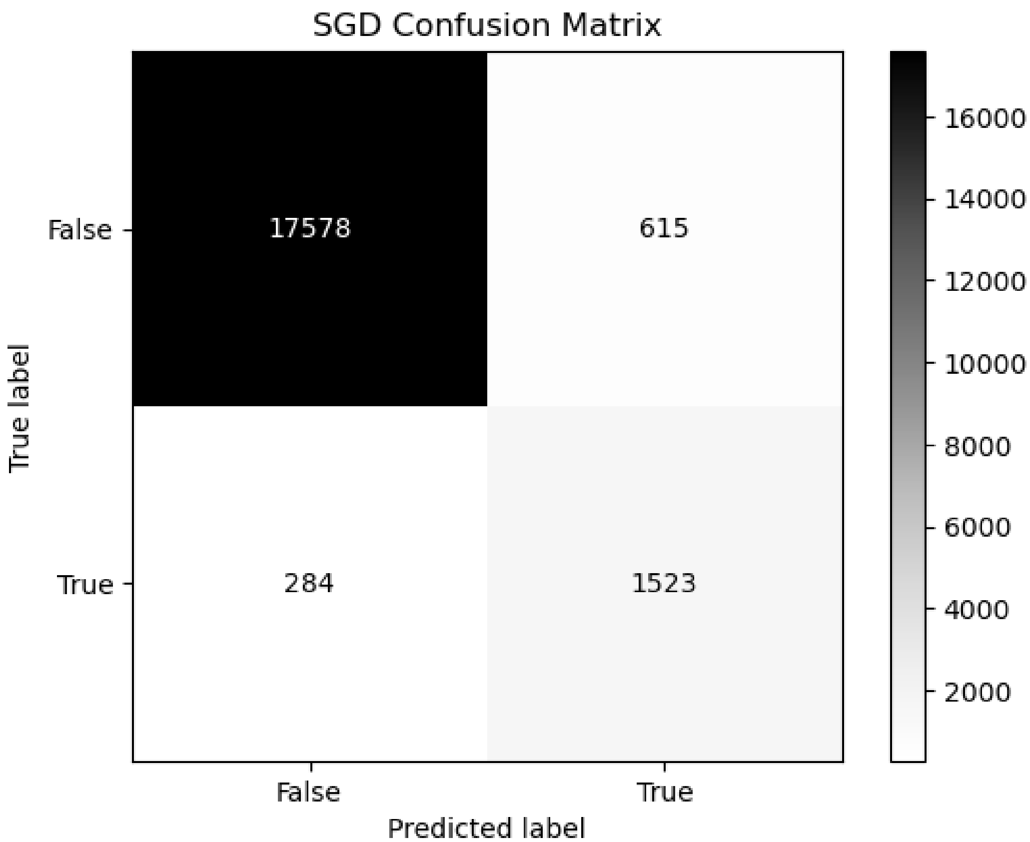
In the SDG we see that it predicts some positives, so the second column has numbers. It is mostly correct, but there are still some false positives and negatives

Swapped variable order Swapping the variable order swaps the places of FP and FN, and can therefore lead to some quite different conclusions.

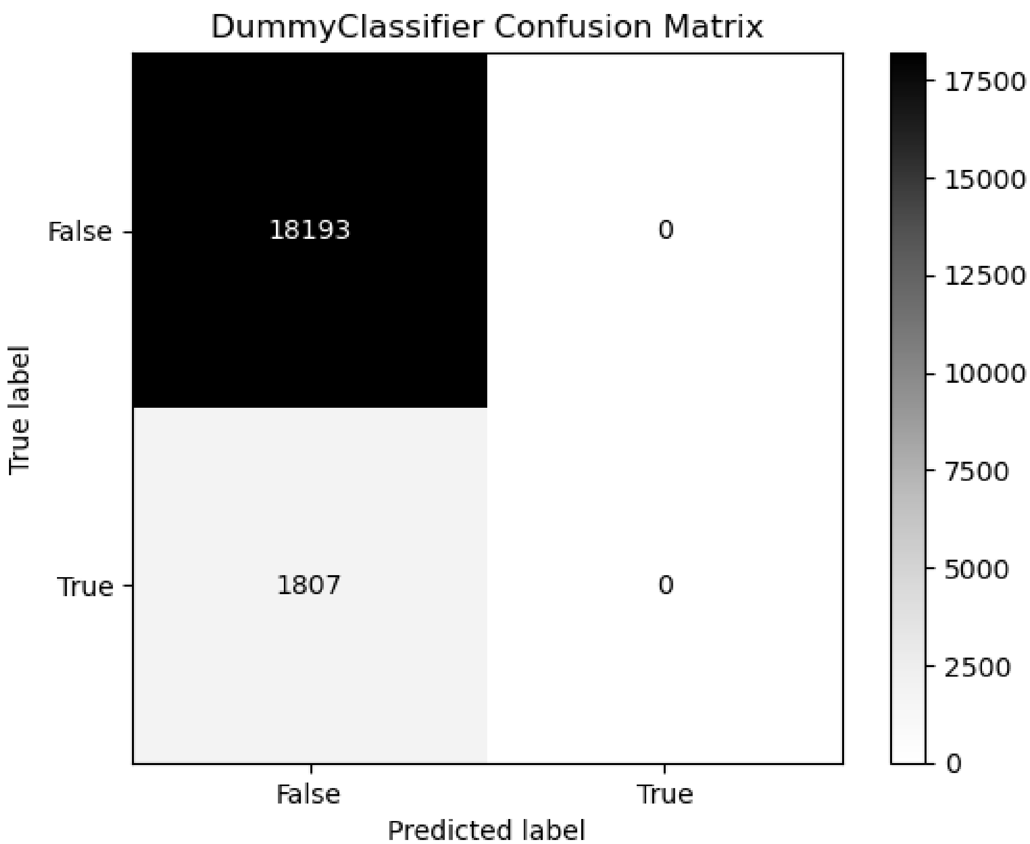
5.4 Qd A Confusion Matrix Heat-map

First we created heat maps for the 2x2 confusion matrixes, seen below

[29]:
```
from sklearn.metrics import ConfusionMatrixDisplay
import matplotlib.pyplot as plt

ConfusionMatrixDisplay.from_predictions(y_test_5, sgd_predict_5, cmap="Grays")
plt.title("SGD Confusion Matrix")
plt.figure()
ConfusionMatrixDisplay.from_predictions(y_test_5, dummy_pred, cmap="Grays")
plt.title("DummyClassifier Confusion Matrix")
plt.show()
```



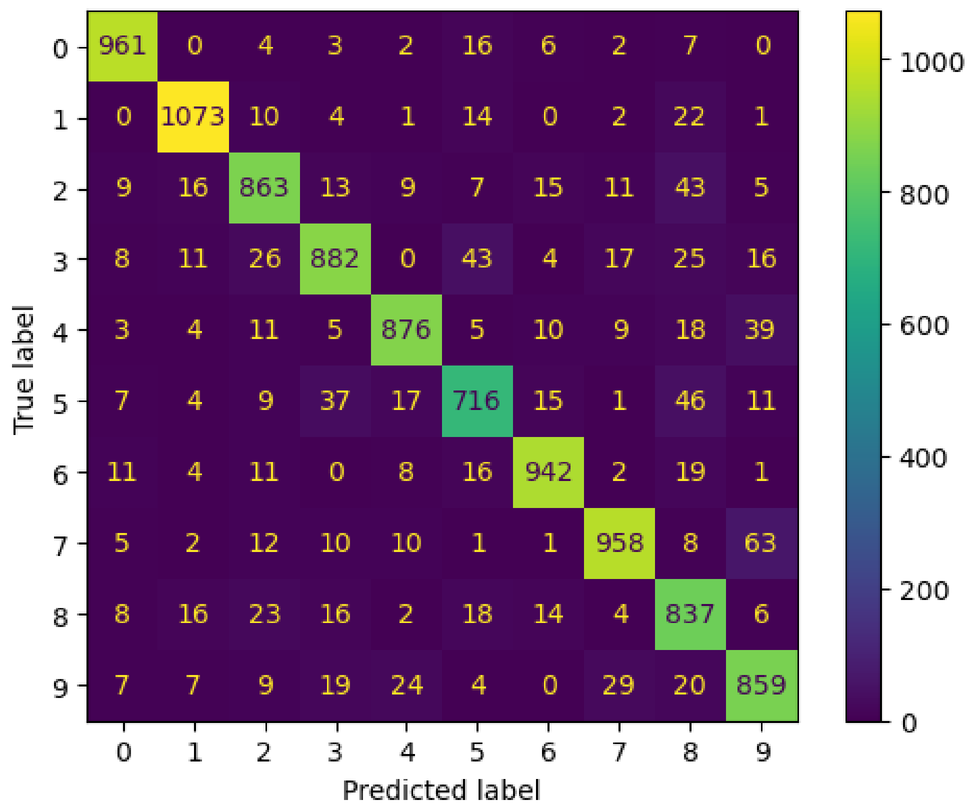

```
<Figure size 640x480 with 0 Axes>
```

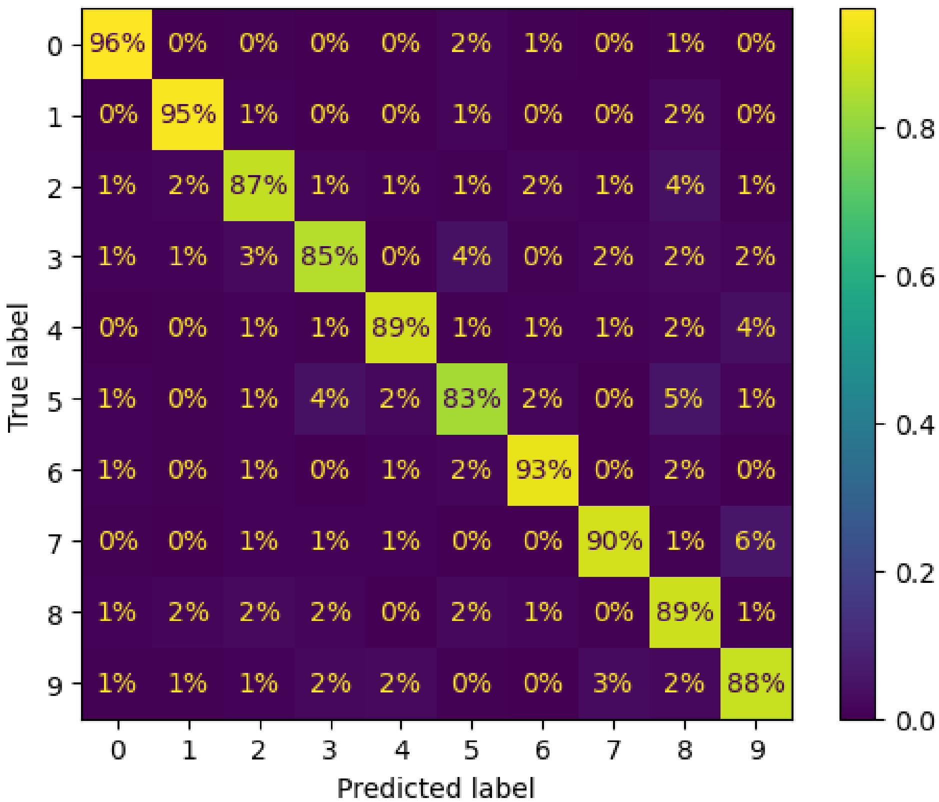


But as they dont tell us much about where the errors occur, we also created them on the 10x10 data

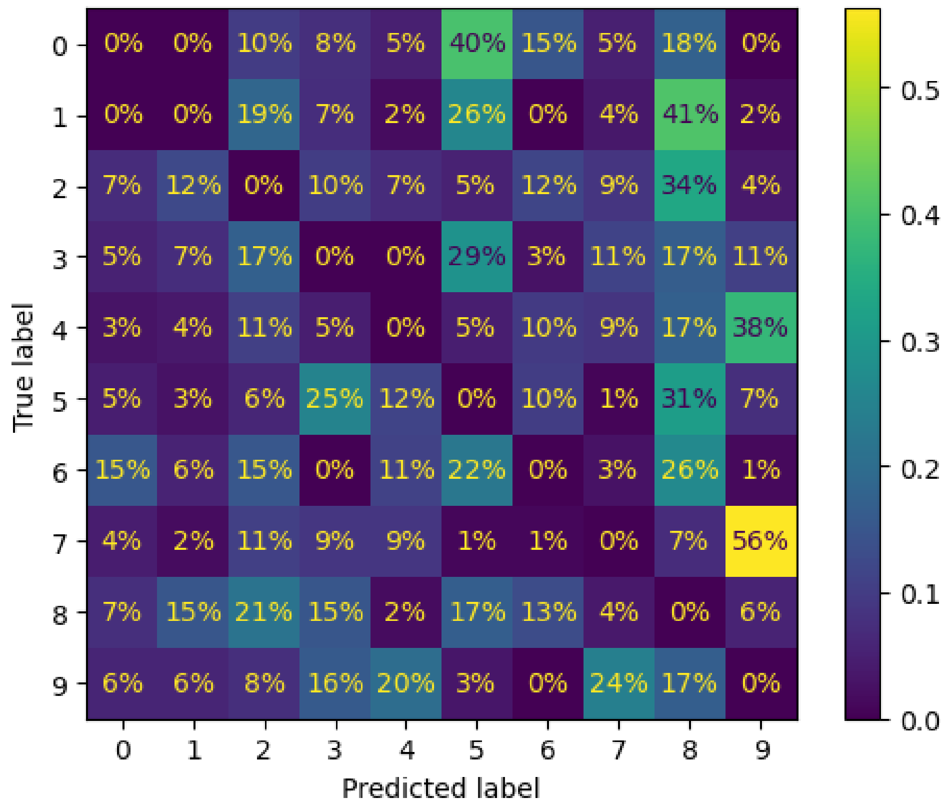
[30]:
```
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.model_selection import cross_val_predict
from sklearn.preprocessing import StandardScaler

# had to reduce size due to taking too long to create plots

scaler = StandardScaler()

X_train_small = X_train[:10000]
y_train_small = y_train[:10000]

X_train_scaled_small = scaler.fit_transform(X_train_small.astype(np.float64))

y_train_pred = cross_val_predict(sgd_clf, X_train_scaled_small, y_train_small,␣
↪cv=3, n_jobs=4)
ConfusionMatrixDisplay.from_predictions(y_train_small, y_train_pred)
plt.show()
```

```python
ConfusionMatrixDisplay.from_predictions(y_train_small, y_train_pred,
normalize="true", values_format=".0%")
plt.show()

sample_weight = (y_train_pred != y_train_small)
ConfusionMatrixDisplay.from_predictions(y_train_small, y_train_pred, sample_weight=sample_weight, normalize="true", values_format=".0%")
plt.show()
```





96% 0% 0% 0% 0% 2% 1% 0% 1% 0%
0% 95% 1% 0% 0% 1% 0% 0% 2% 0%
1% 2% 87% 1% 1% 1% 2% 1% 4% 1%
1% 1% 3% 85% 0% 4% 0% 2% 2% 2%
0% 0% 1% 1% 89% 1% 1% 1% 2% 4%
1% 0% 1% 4% 2% 83% 2% 0% 5% 1%
1% 0% 1% 0% 1% 2% 93% 0% 2% 0%
0% 0% 1% 1% 1% 0% 0% 90% 1% 6%
1% 2% 2% 2% 0% 2% 1% 0% 89% 1%
1% 1% 1% 2% 2% 0% 0% 3% 2% 88%
True label
Predicted label
0 1 2 3 4 5 6 7 8 9
0.0 0.2 0.4 0.6 0.8



In the first heatmap, we see each row being the true digit label, and the column being the predicted label.

The diagonal are the true positives, and the other numbers in each row are the falsely classified numbers, and what else they were identified as.

This is then elaborated on in the other heatmaps, in total percentages, and percentages per row. This helps us see where our model typically classifies wrong, thereby where it should be improved.

5.5 Qe Conclusion

In this exercise notebook we have looked at different types of metrics, that we can use to evaluate our models. We went through some of the functions that are used for this, such as recall, F1 and precision, and implemented them by hand, to better our understanding of how they work behind the scenes.

After that we looked at confusion matrices, that we can use to look at the numbers directly, to see how many TP, FP, TN and TP we are getting from our dataset through our model. We also compared our SGD model to the dummy model, to directly see how the dummy model works. We could see from the numbers produced here that due to how our dataset is structured and how our classification is made, why the dummy model achieved a high accuracy in the previous exercise.

Lastly, we looked at ways to visualize these heatmaps in different versions, to analyze where our model makes the most mistakes. This information can then be used to see where it needs to be improved to achieve better results.