

Machine Learning - Assignment 1

Frederik Predut - StudieNr: 201502305

Magnus Hauge Kyneb - StudieNr: 201807279

Sebastian Laczek Nielsen - StudieNr: 201806678

Gruppe 22 - September 2020

Table of Contents

1	Introduction	3
2	<u>L01 Intro:</u>	4
2.1	Qa	4
2.2	Qb	5
2.3	Qc	6
2.4	Trying out a Neural Network	7
3	<u>L01 Modules And Classes:</u>	8
3.1	Qa	8
4	<u>L02 Cost Function:</u>	10
4.1	Qa Given the following $X^{(i)}$, construct and print the X matrix in python	10
4.2	Qb Implement the L1 and L2 norms for vectors in pynthon - without using methods from libraries	11
4.3	Qc Construct the Root Mean Square Error (RMSE) function (Equation 2-1 [HOML]). . .	12
4.4	Qd Similar construct the Mean Absolute Error (MAE) function (Equation 2-2 [HOML]) and evaluate it.	13
4.5	Qe Robust Code	13
4.6	Qf Conclusion	14
5	<u>L02 Dummy Classifier:</u>	16
5.1	Qa Loading and displaying MNIST-dataset	16
5.2	Qb Adding an SGD-Classifer	16
5.3	Qc Dummy binary classifier	19
5.4	Qd Conclusion	19
6	<u>L02 Performance Metrics:</u>	20
6.1	Qa	20
6.2	Qa	21
6.3	Qc	22
6.4	Qc	22
6.5	Conclusion	23
7	<u>L03 Description of MAL Project</u>	24
7.1	Description of dataset	24
7.2	Description of data	25
7.3	How we can use the data	25

1 Introduction

This Exercise set includes exercises from MAL from the first three lectures. Each lecture contains multiple exercises, the following exercises are all described in here:

L01 Intro, L01 Modules and classes, L02 Cost Function, L02 Dummy Classifier, L02 Performance Metrics and lastly our Project description for this semester.

Front page image source:

<https://www.ie.edu/exponential-learning/blog/wp-content/uploads/2018/01/MachineLearninginMarketing-1621x1000.jpg?fbclid=IwAR2WIEtNNqDDLCSH0KE4tDtNyHsRA3xvmqRFzGhU7PJPcDaOBT5Dlwa8XE>

2 L01 Intro:

2.1 Qa

How do you extract the theta 0 and theta 1 coefficients in his life-satisfaction figure form the linear regression model?):

The two coefficients can be extracted as attributes from the `lin1 = linearmodel.LinearRegression()` object as shown below on picture1:

```
# TODO: add your code here..
#THETA 0
print("Theta zero: ",lin1.intercept_);

#THETA 1
print("Theta one: ",lin1.coef_);

#score from score method using the lin1 obj from example
r2 = lin1.score(X,y);
r3 = model.score(X,y);
print("r^2: ",r2);
print(r3);

Theta zero: [4.8530528]
Theta one: [[4.91154459e-05]]
r^2: 0.7344414355437029
0.7344414355437029
```

Figur 1: Qa

What are the minimum and maximum values for R2 ?:

The values for r2 are ranging from -1 to 1, 1 being a "perfect fit", while values lower represents a lesser fit. 0 being, meaning there is no fit at all

Is it best to have a low R2 score (a measure of error/loss via a cost-function) or a high R2 score (a measure of fitness/goodness)?:

It's best to have a high R2 score since that represents a high fit/goodness measure, where as a low R2 score means that there is a high varians between true and predicted values

2.2 Qb

Change the linear regression model to a `sklearn.neighbors.KNeighborsRegressor` with `k=3` (as in [HOML:p21,bottom]), and rerun the fit and predict using this new model.

as seen below in picture 2, We import a new ML model (KneighborRegressor) to train our model with.

```
# 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
from sklearn.neighbors import KNeighborsRegressor
knN = KNeighborsRegressor(3)
# Train the model
knN.fit(X,y)

# Make a prediction for Cypress
X_new = [[22587]] # Cypres' GDP per capita
y_pred = knN.predict(X_new)
KnScore = knN.score(X,y)
print("y_pred/KNearst: ", y_pred)
print("KNearst score: ", KnScore)

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

Figure 2: Code snippet of KNNRegressor model

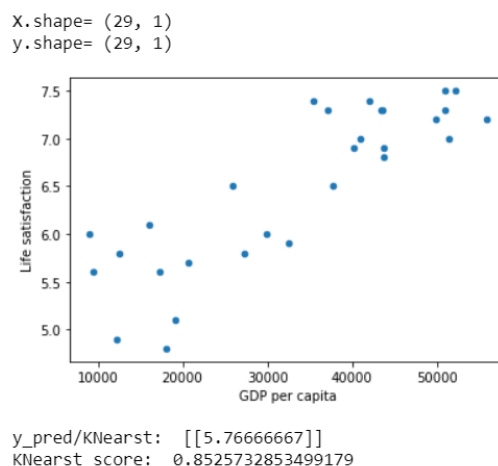


Figure 3: Visualized Data of GDP per Capita and life satisfaction

What do the k-nearest neighbours estimate for Cyprus, compared to the linear regression (it should yield=5.77)?:

y-pred KNearst score is 5.77 (life satisfaction) whereas the linear regression model is 5.96 (in the book p21 bottom)

What score-method does the k-nearest model use, and is it comparable to the linear regression model?:

According to the scikit documentation, the score method returns 'the coefficient of determination R^2 of the prediction', which means that both the linear regression model and K-nearest model are comparable

Seek out the documentation in Scikit-learn, if the scoring methods are not equal, can they be compared to each other at all then?:

In the documentation for both models the scoring methods are equal (returns the same) hence comparing scores wouldn't make much sense.

2.3 Qc

Tuning Parameter for k-Nearest Neighbors and A Sanity Check

Choosing `kneighbor=1` produces a nice score=1, that seems optimal...but is it really so good?:

When `KNeighbor` is 1, it accounts only for datapoints relative to itself, and gives a wrong impression about the model. hence `KNeighbor > 2`

Does a score=1 with `kneighbor=1` also mean that this would be the preferred estimator for the job?:

No as it doesn't tell us much, as we can't train the model properly without regarding other current data points

2.4 Trying out a Neural Network

Can the MLPRegressor score function be compared with the linear and KNN-scores? On figure 4 we instantiate our MLPRegressor and make a prediction for cypress the output can be seen on figure 5. Although scikit learn documentation states that the score method of MLPRegressor returns the same as Linear and KNN regressors, but its not comparable to linear and KNN-scores in our case here.

```
from sklearn.neural_network import MLPRegressor

# Setup MLPRegressor, can be very tricky for the tiny-data
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)
plt.plot(m, y_pred_lin, "r")
plt.plot(m, y_pred_knn, "b")
plt.plot(m, y_pred_mlp, "k")

cypress_pred = mlp.predict(X_new)

mlpscore = mlp.score(X,y)
print("score for cypress", cypress_pred)
print("mlpscore: ", mlpscore)
```

Figure 4: Code snippet

```
Iteration 718, loss = 1.59071692
Training loss did not improve more than tol=0.000010 for 10 consecutive epochs. Stopping.
score for cypress [4.07671125]
mlpscore: -3.6741440524089084
```

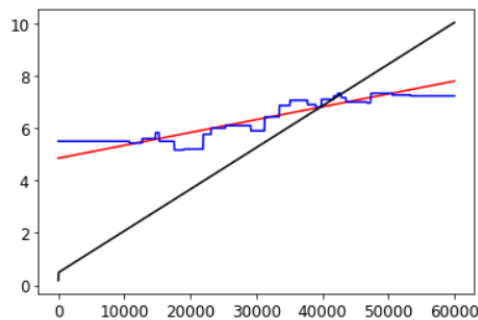


Figure 5: MLP Regressor plot and output

3 L01 Modules And Classes:

3.1 Qa

Qa Load and test the libitmal module

As can be seen below on Figure 6 the library is loaded and all tests are ok.

```
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
```

Figure 6: Code and output of loading a library and running the tests

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

Below on Figure 7 the module is shown and below on figure 8 is shown loading it and using some functions to test it is loaded properly.

```
1 def HelloModule():
2     print("Im Alive")
3
4 def TestFunc(param=True):
5     if param == True:
6         print("true")
7         return
8
9     print("untrue")
```

Figure 7: The module called MyModule

```
# TODO: Qb...
from libitmal import MyModule as m
m.HelloModule()
m.TestFunc(False)
m.TestFunc(True)

Im Alive
untrue
true
```

Figure 8: basic tests of module

Qc How do you 'recompile' a module?

Simply reloading it by typing the name of the library and add ".reload(module)" where module is the name of the loaded module.

Qe + Qf + Qg Extend the class with some public and private functions, member variables, a Constructor and a to-string function

On figure 9 the class MyClass is shown with a private var notated with double "__". The constructor is always the "`__init__`" function. The to string function is shown as "`__str__`".

```
# TODO: Qe...
class MyClass:
    __myVar = "Private Var"
    myVar = "public var"

    def __init__(self, var):
        print("setting private var to: ", var)
        self.__myVar = var

    def printVar(self):
        print("Private var is: ", self.__myVar)
        self.__myPrivFunc()

    def __myPrivFunc(self):
        print("Private Func")

    def __str__(self):
        return self.myVar + ", " + str(self.__myVar)

c = MyClass(10)
c.printVar()

print (c)
```

```
setting private var to: 10
Private var is: 10
Private Func
public var, 10
```

Figur 9: MyClass with all extensions

4 L02 Cost Function:

4.1 Qa Given the following $X^{(i)}$, construct and print the X matrix in python

$$\begin{aligned}\mathbf{x}^{(1)} &= [1, 2, 3]^T \\ \mathbf{x}^{(2)} &= [4, 2, 1]^T \\ \mathbf{x}^{(3)} &= [3, 8, 5]^T \\ \mathbf{x}^{(4)} &= [-9, -1, 0]^T\end{aligned}$$

Figur 10: X matrix

```
# Qa
import numpy as np

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

x1 = np.array([1,2,3])
x2 = np.array([4,2,1])
x3 = np.array([3,8,5])
x4 = np.array([-9,-1,0])

X = np.array([x1,x2,x3,x4])

print("X:",X)
```

X: [[1 2 3]
[4 2 1]
[3 8 5]
[-9 -1 0]]

Figur 11: X matrix printet out in python

As shown above we print out the matrix X figure 10 in python figure 11

4.2 Qb Implement the L1 and L2 norms for vectors in python - without using methods from libraries

```
import numpy as np
import math

# Euclidian distance function
def L2(vector):
    return (sum(vector**2))**0.5

# City block norm function
def L1(vector):
    return sum((vector**2)**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(d2)

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

eps=1E-9 # remember to import math for fabs
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"

def L2Dot(vector):
    return math.sqrt(np.dot(vector, vector))

# comment-in once your L2Dot fun is ready...
d2dot=L2Dot(tx-ty)
print("d2dot-expected_d2=",d2dot-expected_d2)
assert math.fabs(d2dot-expected_d2)<eps, "L2Dot dist seem to be wrong"

4.242640687119285
tx-ty=[-2  3 -1 -2], d1-expected_d1=0.0, d2-expected_d2=0.0
d2dot-expected_d2= 0.0
```

Figure 12: output from our functions

As shown on figure 12 we see the implementation for L2 and L1 functions. The functions are pure pythonic since they don't rely on numpy functions. and we see the result we expected.

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

```
import math

# L2 is length(norm) of difference vector. In a way, the total of the errors in each dimension.
# MSE is the the average of the errors, in every dimension.

# L2 being sqrt(x_1^2+x_2^2*x_3^2...x_n^2)
# MSE being (x_1^2+x_2^2*x_3^2...x_n^2)/n

# We extract the x_1^2+x_2^2*x_3^2...x_n^2 from the L2 by squaring, removing the sqrt.
# We can then divide by the size of the vector.
def MSE(vector1, vector2):
    assert vector1.size == vector2.size, "Vector sizes must match"
    return (L2(vector1-vector2)**2)/vector1.size

# RMSE is sqrt of MSE.
def RMSE(vector1, vector2):

    assert vector1.size == vector2.size, "Vector sizes must match"
    return math.sqrt((L2(vector1-vector2)**2)/vector1.size)

# 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)

# TEST vector:
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"

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

Figur 13: Our RMSE function and its output

In figure 13 we see the MSE function using our L2 definition inside its stomach. And we pretty much get the expected result

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

```
# TODO: solve Qd

# L1 is already the sum of the absolute error values. We just need to find the m
def MAE(vector1, vector2):
    assert vector1.size == vector2.size, "Vector sizes must match"
    return L1(vector1-vector2)/vector1.size

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

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

MAE=3.75, diff=0.0

Figure 14: Our MAE function and its output

In figure 14 we've implemented our MAE function, that calls L1 internally, and we get the expected result

4.5 Qe Robust Code

Add error checking code (asserts or exceptions), that checks for right y' - y sizes of the MSE and MAE functions.

We've added some robustness to our functions by using asserts. As shown on figure 15 both the MSE and MAE function, asserts on its input and confirms that the input is in fact a vector in 1 dimension. If not we raise a ValueError. Last we assert that the vectors indeed are of the same size.

```
import math

# L2 is Length(norm) of difference vector. In a way, the total of the errors in each dimension.
# MSE is the the average of the errors, in every dimension.

# L2 being sqrt(x_1^2+x_2^2*x_3^2...x_n^2)
# MSE being (x_1^2+x_2^2*x_3^2...x_n^2)/n

# We extract the x_1^2+x_2^2*x_3^2...x_n^2 from the L2 by squaring, removing the sqrt.
# We can then divide by the size of the vector.
def MSE(vector1, vector2):
    if not vector1.ndim==1:
        raise ValueError("Vector 1 is not a vector")
    assert vector1.size == vector2.size, "Vector sizes must match"
    return (L2(vector1-vector2)**2)/vector1.size

def MAE(vector1, vector2):
    if not vector1.ndim==1:
        raise ValueError("Vector 1 is not a vector")
    assert vector1.size == vector2.size, "Vector sizes must match"
    return L1(vector1-vector2)/vector1.s

# RMSE is sqrt of MSE.
def RMSE(vector1, vector2):
    if not vector1.ndim==1:
        raise ValueError("Vector 1 is not a vector")
    assert vector1.size == vector2.size, "Vector sizes must match"
    return math.sqrt((L2(vector1-vector2)**2)/vector1.size)
```

Figure 15: Added robustness to our code - asserts

```
# Testing for size match.
matrix = np.array([[2],[3],[4]])
long = np.array([2,3,4,3,6,1])
short = np.array([2,3,4])

# Getting error when calling index 1, shape vector is acting weird.
# print(short.shape[1])
print(matrix.shape)

print(MAE(short, long))
#print(MAE(matrix, long))

(3, 1)

-----
AssertionError                                Traceback (most recent call last)
<ipython-input-37-c128a8043855> in <module>
      8 print(matrix.shape)
      9
----> 10 print(MAE(short, long))
      11 #print(MAE(matrix, long))

<ipython-input-4-397ca31b19d3> in MAE(vector1, vector2)
      5     if not vector1.ndim==1:
      6         raise ValueError("Vector 1 is not a vector")
----> 7     assert vector1.size == vector2.size, "Vector sizes must match"
      8     return L1(vector1-vector2)/vector1.size
      9

AssertionError: Vector sizes must match
```

Figur 16: AssertionError - Vector sizes must match

```
# Testing for size match.
matrix = np.array([[2],[3],[4]])
long = np.array([2,3,4,3,6,1])
short = np.array([2,3,4])

# Getting error when calling index 1, shape vector is acting weird.
# print(short.shape[1])
print(matrix.shape)

#print(MAE(short, long))
print(MAE(matrix, long))

(3, 1)

-----
ValueError                                Traceback (most recent call last)
<ipython-input-36-922005438cc3> in <module>
      9
     10 #print(MAE(short, long))
----> 11 print(MAE(matrix, long))

<ipython-input-4-397ca31b19d3> in MAE(vector1, vector2)
      4 def MAE(vector1, vector2):
      5     if not vector1.ndim==1:
----> 6         raise ValueError("Vector 1 is not a vector")
      7     assert vector1.size == vector2.size, "Vector sizes must match"
      8     return L1(vector1-vector2)/vector1.size

ValueError: Vector 1 is not a vector
```

Figur 17: Raised ValueError - Not a vector

4.6 Qf Conclusion

The purpose of this exercise was to get more a in-depth understanding how a ML-model works, and which parts the makes up a particular model then implementing the various methods and norms. We've implemented the

L1 and L2 norms which calculates the 'Euclidian distance' between two vectors and tested it. Next we've implemented the RMSE and MAE functions which we use to detect error on the model. We've also added some robustness to our code in terms of validation, and gotten acquainted with the overall Jupyter environment.

5 L02 Dummy Classifier:

5.1 Qa Loading and displaying MNIST-dataset

First, the dataset is loaded and a digit is visualized from vector form. A GetDataSet function is created to make future MNIST loads easier.

```
import numpy as np
from sklearn.datasets import fetch_openml

# Loading dataset.
X, y = fetch_openml('mnist_784', return_X_y=True, cache=True)

# Plotting method for digit.
%matplotlib inline
def MNIST_PlotDigit(data):
    import matplotlib
    import matplotlib.pyplot as plt
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = matplotlib.cm.binary, interpolation="nearest")
    plt.axis("off")

# Plotting first digit of dataset.
MNIST_PlotDigit(X[0])

# Defining a MNIST Loader method.
def MNIST_GetDataSet():
    from sklearn.datasets import fetch_openml
    X, y = fetch_openml('mnist_784', return_X_y=True, cache=True)
    return X, y
```

Figur 18: Opgave Qa kode

5.2 Qb Adding an SGD-Classifer

The dataset is split into a training set and a testing set. The "truth"(y)-vectors are simplified to be true when the digit is '5' and false in any other case.

An SGD-Classifer model is trained by fitting to the training data.

```
# Splitting data into training and test datasets.
X_train, X_test, y_train, y_test = X[:65000], X[65000:], y[:65000], y[65000:]

# Get a vector that shows every '5' as being true.
y_train_5 = (y_train == '5')
y_test_5 = (y_test == '5')

# Create SGDClassifier object.
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(random_state=69)

# Fitting training data.
sgd_clf.fit(X_train, y_train_5)

# Didn't need any reshaping for this part it seems, perhaps because called in PlotDigit.
```

Figur 19: Opgave Qb kode - del 1

The SGD-model is used on the test datasets and an afternoon was spent creating a method for visualizing the results.

```
# Spent way too long making this visualizer for a bunch of numbers in the dataset.
# Shows a number and the SGD-prediction.
def MNIST_PlotDigitGrid(data_vector, rows, columns):
    import matplotlib
    import matplotlib.pyplot as plt

    fig=plt.figure(figsize=(20, 20))

    for i in range(1, data_vector.shape[0]):
        image = data_vector[i - 1].reshape(28,28)
        sub = fig.add_subplot(rows, columns, i)

        prediction = sgd_clf.predict([X_test[i - 1]])

        if (prediction == True):
            text_pred = 'This is a 5!'
            color_pred = 'black'
        else:
            text_pred = 'Not 5!'
            color_pred = 'red'

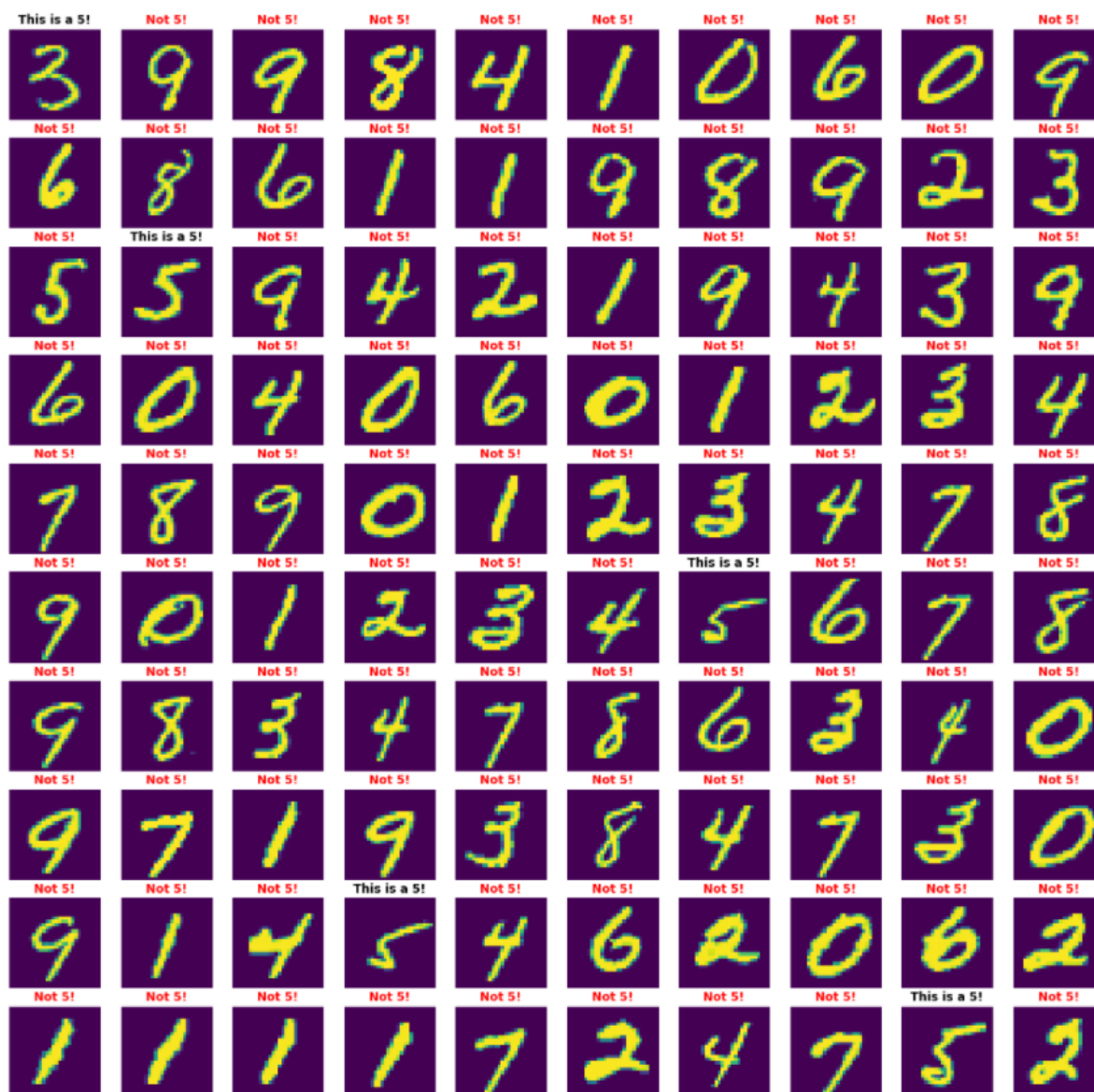
        sub.set_title(text_pred, color=color_pred, fontweight='bold')
        sub.text = 'red'

        plt.imshow(image)
        plt.axis("off")

MNIST_PlotDigitGrid(X_test[0:101], 10, 10)
```

Figur 20: Opgave Qb kode - del 2

The output of the MNISTPlotDigitGrid can be seen below. The model's prediction is located above the digit it's trying to predict. A false positive can be seen from the start, as it classifies a 3 as 5. A false negative can be seen in the 1st column on the 3rd row. Other than those two cases, it's doing a good job predicting the first 100 digits of the test set, which it has never seen before.



Figur 21: Opgave Qb - Digit + prediction grid output

5.3 Qc Dummy binary classifier

The a DummyClassifier is created, which always guesses a digit isn't '5'. The DummyClassifier is "fit" (not really) to the training data and is used to predict on the testing data. Amazingly (or not so), it gets a 91 percent accuracy score! , Naturally, this is because it's only around 10 percent of digits that are a '5'.

```
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin

class DummyClassifier(BaseEstimator, ClassifierMixin):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X),1), dtype=bool)

import sklearn.metrics as sklm

dumb = DummyClassifier()

dumb.fit(X_train, y_train_5)
dumb_prediction = dumb.predict(X_test)

# Accuracy of 91%!
sklm.accuracy_score(dumb_prediction, y_test_5)

0.9128
```

Figur 22: Opgave Qa kode

5.4 Qd Conclusion

A SGD-Classifier can be used to predict numbers relatively well on the MNIST dataset, or so it seems looking at the guesses on the grid. However, it's also clear that an accuracy score isn't enough to judge this predictive power, as the dummy-model, which is blindly guessing false, is able to achieve a similar score.

6 L02 Performance Metrics:

6.1 Qa

Implement the Accuracy function and test it on the MNIST data.):

The accuracy function is implemented in the MyAccuracy class that takes the sum of all truths equal to the predictions, then divides with the size, as seen on figure 23.

```
def MyAccuracy(y_true, y_pred):
    assert y_true.size == y_pred.size
    n_correct = sum(y_true == y_pred)
    return n_correct/y_true.size
```

Figur 23: Class for accuracy calculation

On figure 24, is shown the dummy classifier(from previous exercise) and the sgd classifier. Both classifiers are used for a cross value prediction, and then run through the accuracy. The results are the same for both accuracy tests, meaning MyAccuracy calculates the same as the library function.

```
X, y = fetch_openml('mnist_784', return_X_y = True)

X_train = X[:60000]
X_test = X[60000:]
y_train = y[:60000]
y_test = y[60000:]

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

dummy = DummyClassifier()
sgd_clf = SGDClassifier(random_state=42)

sgd_clf.fit(X_train, y_train_5)

from sklearn.model_selection import cross_val_predict

sgd_pred = cross_val_predict(sgd_clf, X_test, y_test_5, cv=3)
dum_pred = cross_val_predict(dummy, X_test, y_test_5, cv=3).T[0]
TestAccuracy(sgd_pred, y_test_5)
TestAccuracy(dum_pred, y_test_5)
```

```
my a          =0.9581
scikit-learn a=0.9581
Good
```

```
my a          =0.9108
scikit-learn a=0.9108
Good
```

Figur 24: the code testing and the resulting accuracy

6.2 Qa

Implement Precision, Recall and F1-score and test it on the MNIST data for both the SGD and Dummy classifier models:

Below on figure 25 is seen the three functions compared to the given library functions, as can be seen on the results they yield the same answer.

```
def MyPrecision(y_true, y_pred):
    TP = sum(y_pred & y_true)
    FP = sum((y_pred == True) & (y_true == False))
    return TP / (TP + FP)

def MyRecall(y_true, y_pred):
    TP = sum(y_pred & y_true)
    FN = sum((y_pred == False) & (y_true == True))

    return TP / ( TP + FN )

def MyF1Score(y_true, y_pred):
    p = MyPrecision(y_true, y_pred)
    r = MyRecall(y_true, y_pred)
    return (2*p*r) / (p + r)
# TODO: your test code here!
y_pred = sgd_clf.predict(X_test)

print(MyPrecision(y_test_5, y_pred))
print(precision_score(y_test_5, y_pred))
print("-----")

print(MyRecall(y_test_5, y_pred))
print(recall_score(y_test_5, y_pred))

print("-----")

print(MyF1Score(y_test_5, y_pred) - f1_score(y_test_5, y_pred))

0.6618887015177066
0.6618887015177066
-----
0.8800448430493274
0.8800448430493274
-----
0.0
```

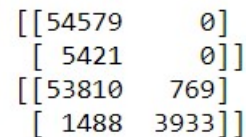
Figur 25: functions and test code to confirm right implementation

6.3 Qc

The Confusion Matrix:

On figure 26 is seen the code to generate the confusion matrices. The first matrix is the dummy matrix, it shows it finds 54k True Negatives, which makes sense in that it is just an array that assumes everything is not 5, which is also why it gets the type II error, in that it assumes all 5's are not 5 as seen on the number 5421 which is also called a False Negative. The two zeros are because it never tries to predict anything to be a 5, which mean it can't contain True Positives or Negative Positives.

```
y_train_pred = cross_val_predict(dummy, X_train, y_train_5, cv = 3)
sgd_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv = 3)
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_train_5, y_train_pred))
print(confusion_matrix(y_train_5, sgd_pred))
```



[[54579	0]
[5421	0]]
[[53810	769]
[1488	3933]]

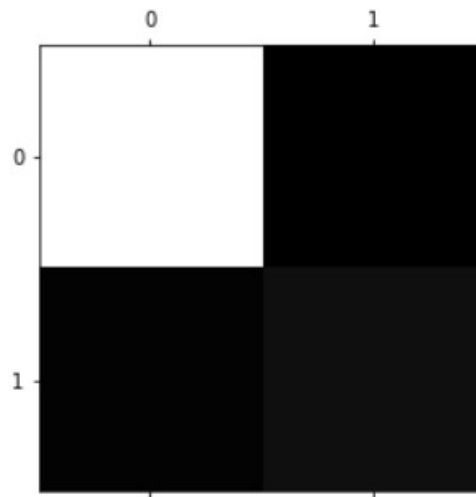
Figur 26: Confusion matrix for both dummy- and sgd-classifier

6.4 Qc

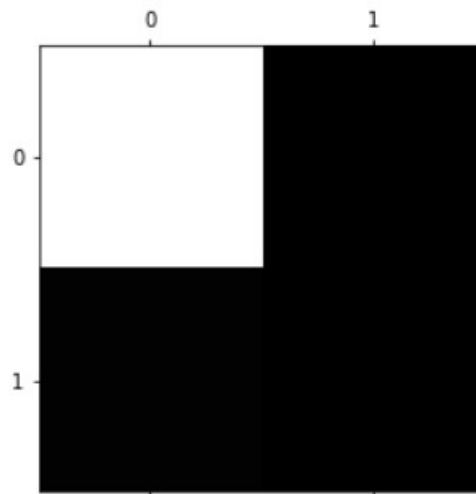
The Confusion Matrix heatmap:

Below on Figure 27 is the heatmaps plotted, inspired by the code from HOLM.

M_SGD heatmap:



M_Dummy heatmap:



Figur 27: Confusion matrix for both dummy- and sgd-classifier

6.5 Conclusion

These exercises shows how to calculate performance of different classifiers by making our own functions and using existing library functions. Both our own functions and library functions are compared and used to evaluate the predictions. As shown above all the calculations match and confirm the right implementation of the functions. The Confusion matrix also shows how the different classifiers perform, and how the dummyclassifier really isn't usefull at all.

7 L03 Description of MAL Project

7.1 Description of dataset

We've chosen a dataset from Google's "Quick, Draw!" browser game. This game challenges players to doodle an object based on a word and have Google's ML-model guess what the object is. Here's an example, the dataset containing doodles trying to match the word "Duck".

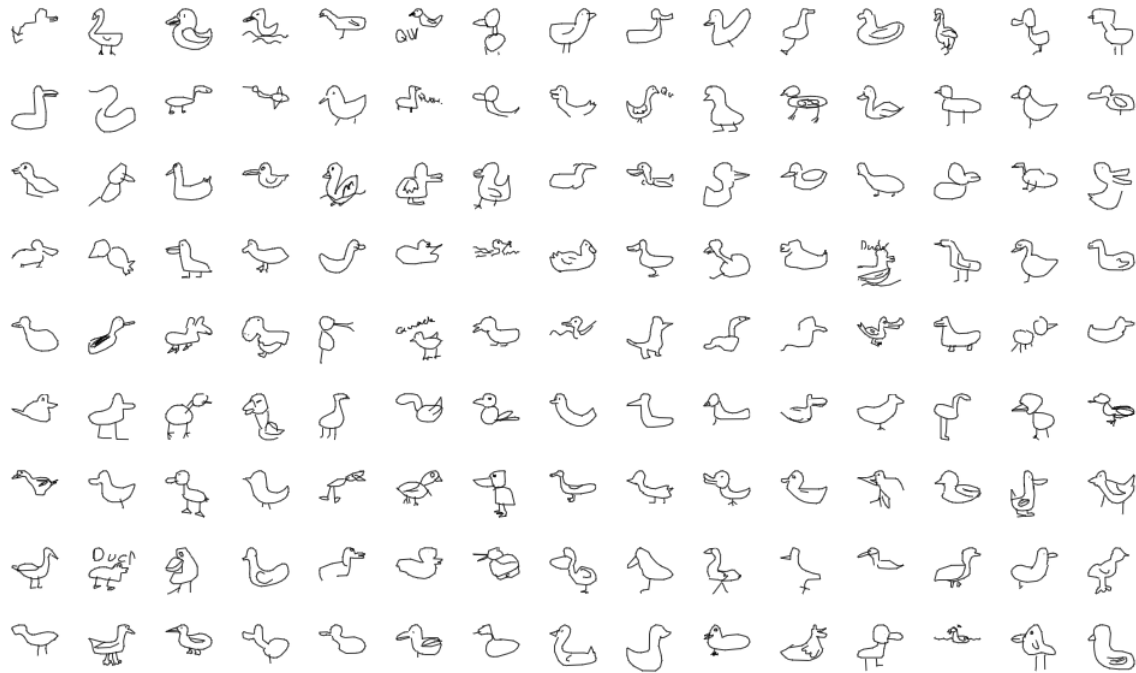


Figure 28: Duck doodles by users.

Over a 100.000 duck doodles have been made and are available for download. The duck-doodle is just one of many different target-words. There's everything from brain to boomerang. Below you can see a small cut-out of the different word collections.



Figure 29: Different doodle categories.

7.2 Description of data

Google has made it easy to work with the doodle-data. They're providing a few data options, preprocessed in different ways. They have datasets with and without timing data. There are simplified versions (.ndjson) that have been scaled down to a 256x256 region. It's also available as binary data files (.bin) and numpy bitmaps (.npy).

Source: <https://github.com/googlecreativelab/quickdraw-dataset>

The raw moderated dataset

The raw data is available as `ndjson` files separated by category, in the following format:

Key	Type	Description
key_id	64-bit unsigned integer	A unique identifier across all drawings.
word	string	Category the player was prompted to draw.
recognized	boolean	Whether the word was recognized by the game.
timestamp	datetime	When the drawing was created.
countrycode	string	A two letter country code (ISO 3166-1 alpha-2) of where the player was located.
drawing	string	A JSON array representing the vector drawing

Figur 30: A snippet of the documentation on the datasets.

7.3 How we can use the data

We want to split the data into training sets and test sets and see if we're able to predict what object an unknown doodle is supposed to look like. The difficulty of this is easily scaled up and down, depending on how tough it turns out to be. We can start with a binary classification, as in, "is this a donut or is this a house?" Once we get that working, we can expand to more categories and even try inserting our very own doodles and see how it handles them.