

ITMAL Opgavesæt 1

28/2 2021

ITMAL Gruppe 41

Kristian Lau Jespersen

Studienummer: 201807157

Oskar Vedel

Studienummer: 201804993



Ingeniørhøjskolen Katrinebjerg

Aarhus Universitet

[1]

L01/Intro

Qa - The 0 parameters and the R2 Score

We use `model.fit()` to train the `sklearn.linear_model.LinearRegression()` model to the life satisfaction data. Then we use `model.predict()` to find the life satisfaction for a given income value:

```
y_pred = model.predict([[22587]]) = 5.96
```

We extract the θ_0 and θ_1 values using:

```
1 >>>print(model.coef_)  
2 4.91154459e-05
```

Snippet 1: Printing the slope

```
1 >>>print(model.intercept_)  
2 4.8530528
```

Snippet 2: Printing the y intercept

We extract the R^2 score using:

```
1 >>>print(model.score(X,y))  
2 0.734441435543703
```

The R^2 score reveals how well the model fits the test data. The typical range of the R^2 -score is 0.0 to 1.0, with 1.0 being the best possible score. Since it is best to have a high score, R^2 is a measure of fitness.

What R^2 describes is in essence how far from a line a set of points are distributed, where points further away from the line have an exponentially larger negative impact on the score than closer points. The score is normalized over the variance of the dataset. Otherwise the R^2 of different datasets could not be compared, since the respective scale of the datasets would not be taken into account.

Qb - Using k-Nearest Neighbors

We change the algorithm to `sklearn.neighbors.KNeighborsRegressor` with $k=3$ using:

```
model = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
```

KNeighborsRegressor is an example of instance based learning whereas linear regression is model based. With this particular dataset it can be assumed that very few extrapolated data points will occur as long as the training data has a proper representation. Therefore it should be fine to use instance based learning for this dataset, as this form of learning performs better when close to all of the data is expected to be interpolated. Using a k-value of 3 for the KNeighborsRegressor yields a score of 5.76666667. It should be noted, that this score is not reliable, since it is not the result of a cross validation. There is no guarantee that the data used for training is representative of data to come.

What score-method does the k-nearest model use, and is it comparable to the linear regression model? Both the the k-nearest model and the linear regression models use R^2 score method.[2][3]. It does seem reasonable that both KNN and the linear regression model use R^2 as their default score method. While the two methods of regression are quite different in how they are trained, they both output a line and are thereby suitable for R^2 evaluation.

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

The k-nearest neighbor algorithm is will perform very poorly when given either a k-value too small or too large. It overfits when the k value is too low, and vice versa. The maximum overfitting occurs at a k-value of 1, where when evaluating new data, the algorithm only evaluates a single value of the nearest neighbor. If the algorithm is trained with a $k=1$, and a the same training data is used for scoring, it will look as though the algorithm is the best predictor imaginable.

This is due to the non-existent bias, knn with a k-value of 1 will simply draw a crooked line that intercepts all datapoints of the training set, see figure 1. Therefore it is completely overfit and will perform very badly on anything other than the training set.

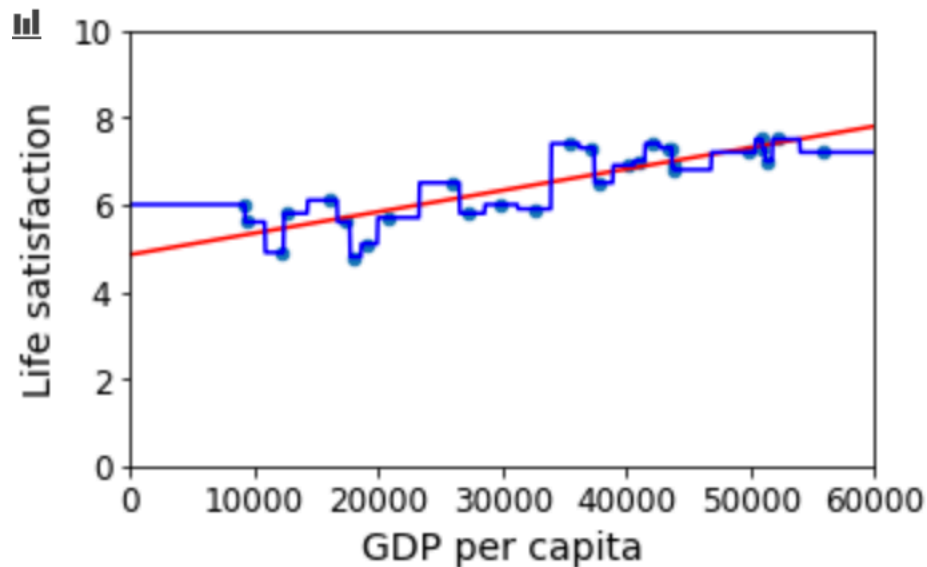


Figure 1: Red line: function obtained by linear regression. Blue line: The resulting line from training KNN with $k=1$. Points: The training set.

If we switch to using a train test split for creating and evaluating our models it becomes clear that the linear regressor significantly outperforms the KNN regressor. The linear bias of the linear regressor prevents it from overfitting too much to the training data and is thus better than the KNN regressor which completely overfits. Plotting the two regression results against the test-set shows the performance difference, figure 2. The respective R^2 scores for this particular test-set are 0,74 for the linear regressor and 0,44 for the KNN regressor. To get a proper score it would of course be necessary to do a cross validation and average the k scores.

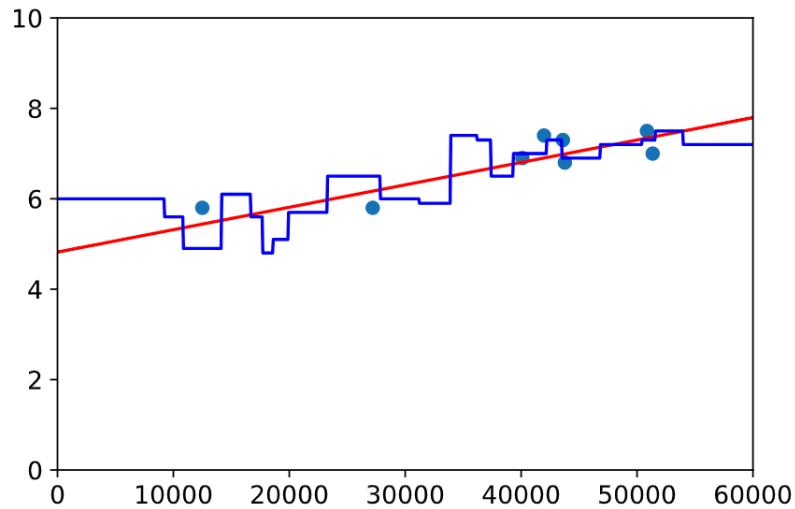


Figure 2: A train test split reveals that $k=1$ should be avoided. Red line: function obtained by linear regression. Blue line: The resulting line from training KNN with $k=1$. Points: The test set.

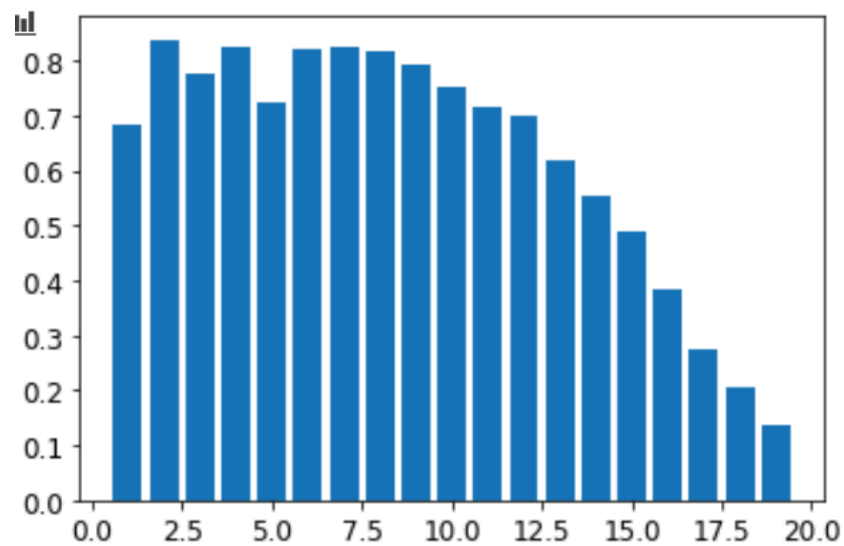


Figure 3: A plot of the scores obtained from using different k -values on a random train test split of the data

The above figure 3 was created using the following code:

```
1 X = np.c_[country_stats["GDP per capita"]]
2 y = np.c_[country_stats["Life satisfaction"]]
3
4 X_train, X_test, y_train, y_test = train_test_split(X, y)
5
6 scores = []
7 k_range = range(1,20)
8 for k in k_range:
9     knn = KNeighborsRegressor(n_neighbors=k).fit(X_train,y_train)
10    scores.append(knn.score(X_test,y_test))
11 plt.bar(k_range,scores)
```

It is important to note that the scores shown in figure 3 are very much a product of that particular train test split, since we have such a small selection of data available. If another random train test split is evaluated with different k-vals it becomes apparent that the scores are very different, see figure 4.

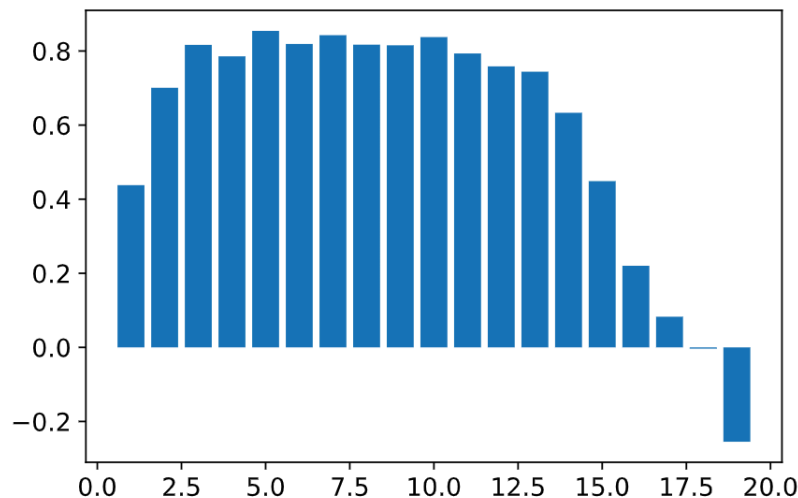


Figure 4: Scores for different k-values after a new train test split

Finding the optimal k-value for this dataset is best done by cross validation as shown below. The optimal k-val was found to be 6.

```
1 scores = []
```

```
2 for k in range(1,15):  
3     knn = KNeighborsRegressor(n_neighbors=k)  
4     scores.append(cross_val_score(knn,X,y,cv=5).mean())  
5 print("BEST K VAL:", np.argmax(scores)+1)
```

Qd - Trying out a Neural Network

Training the neural network on data with absolute values is not a viable approach to the problem. When using data that has not been normalized to train the MLP-model, it scores very low. Scores such as -70900 can be observed, which concludes that the model is useless. Plotting the resulting mlp-regression line against the other regression lines shows that it does a very bad job at recognizing the linear tendency in the data, compared to the other regressors, see figure 5

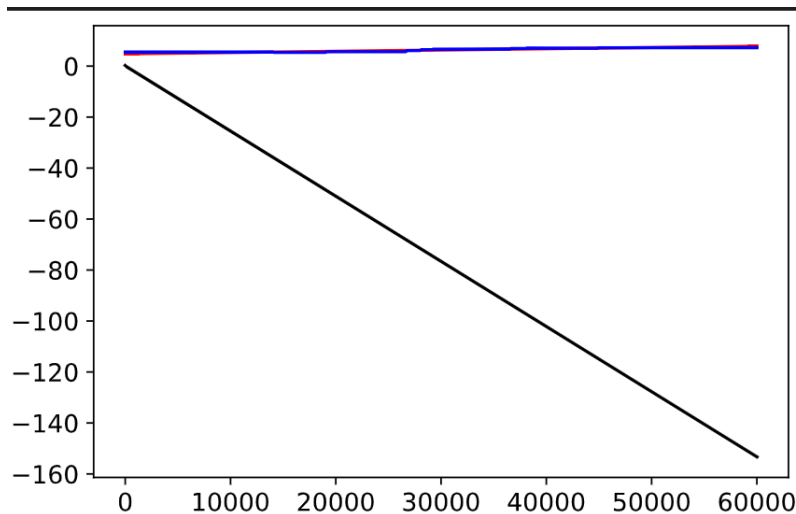


Figure 5: Without the prescaler the mlp regressor cannot recognize the linear tendency. Black line represents the mlp regression line, while blue and red are the knn and linear regressors respectively.

Qe - Neural Network with pre-scaling

It is however possible to train a usable mlp-model from this dataset. In order to do this we need to normalize the dataset. This can be done using the `sklearn.preprocessing.MinMaxScaler` or manually. Setting up a pipeline that preprocesses the data to a normalized range and feeds it to the mlp regressor can be done as seen below.

```
1 mlp_pipe = Pipeline([('scaler', MinMaxScaler()), ('mlp', MLPRegressor(  
    hidden_layer_sizes=(10,), solver='adam', activation='relu', tol=1E  
    -5, max_iter=100000, verbose=False))])  
2 mlp_pipe.fit(X_train, y_train.ravel())  
3 print("Score:", mlp_pipe.score(X_test, y_test.ravel()))  
4 >>>0.7182297549230252
```

Snippet 3: Setting up a pipeline that normalizes input data before handing it to the mlp-regressor. Training and scoring of the model is done on line 2 and 3 (No cross validation).

The score of 0.718 is a significant improvement over the previous negative scores. All the three regression tools use the same scoring method, R^2 . Therefore it is reasonable to compare them. The following scores are all derived from one particular train test split, and should therefore only be taken as just a reasonable estimate of the performance.

lin: 0.739

knn: 0.819

mlp: 0.718

Plotting the three regression lines against the validation set shows that they did indeed perform similarly. But again, the dataset is very small, so it is not a very reliable estimation of performance.

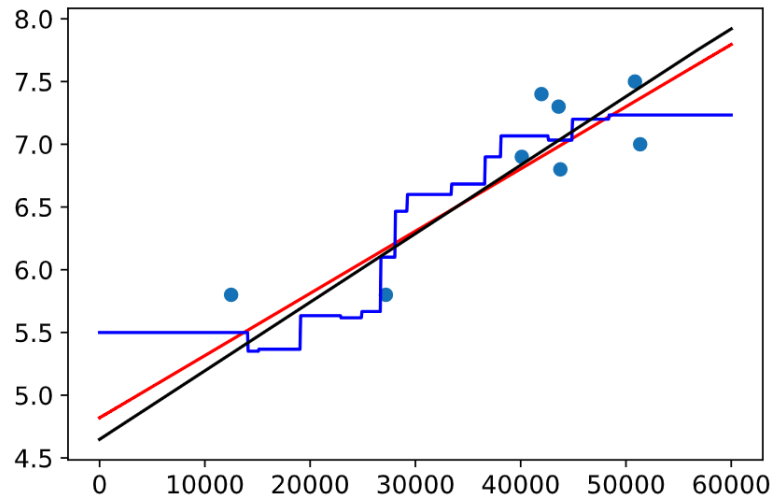


Figure 6: Red: linear reg, Blue: knn, Red: mlp, Dots: test set

L01/Modules and classes

Qa - Load and test the libitmal module

We try out the libitmal module, running `utils.utils.TestAll()` via:

```
1 from libitmal import utils as itmalutils
2 itmalutils.TestAll()
```

Snippet 4: Running `utils.TestAll()` from libitmal

This outputs a test matrix:

```
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 7: Output from libitmal utils.TestAll()

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

We create the module `it_minds_of_99` and add some helper functions to it. Since we use Anaconda on macOS, we have to add the module path to anaconda. We do this by using

```
1 conda-develop /Users/[USER]/python\_modules
```

Snippet 5: Adding path to anaconda

This adds a file `conda.pth` containing the path to our designated python modules folder (`/Users/[USER]/python_modules/`) to `/Users/[USER]/opt/anaconda3/lib/python3.8/site-packages/`

Qc - How do you 'recompile' a module?

To force recompilation of a module in Jupyter, we write this in the notebook before we use the module:

```
1 %load_ext autoreload
2 %autoreload 2
```

Snippet 6: Printing the slope

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

We extend the class with some private and public functions and some member variables:

```
1 class MyClass():
2     class_level_var = "asdf"
3     _semi_private_class_level_var = [4,4,4]
4     __private_class_level_var = 420
5
6     def my_func(self):
7         print("This is a message from a MyClass instance.")
8
9     #Private method
10    def __my_func(self):
11        print("This method should only be called within MyClass")
12
13    @staticmethod
14    def my_static_func():
15        print("I do not belong to a MyClass instance")
16
17    # Calling the static method
18    MyClass.my_static_func()
19    myobjectx = MyClass()
20    # Calling the public method
21    myobjectx.my_func()
```

Snippet 7: MyClass

The concept of private members is not present in python. However, the notion of privacy in programming languages mostly exists to differentiate between internal and external interfaces. Privacy can therefore just as easily be achieved by naming convention as it is done in Python. Two different ways to declare private members exists:

- `_single_leading_underscore`
- `__double_leading_underscore`

There is a slight difference between the two naming conventions. Symbols with a single leading underscore will not be included when an import statement is executed for the module in which the symbol resides. However, the symbols can be accessed with the dot operator on an instance of the class in question. It is merely a convention not to use symbols prefixed with a single underscore from outside the class.

When naming a symbol with a double underscore prefix, the python interpreter will utilize *name mangling* when it reads the function definition. It will create a new name for the symbol in question, which can be used to prevent name ambiguity when subclassing. Below is an example of the name mangling:

```
1 class PrivacyExample:
2     def __init__(self):
3         self.public = 4
4         self._private_by_convention = 5
5         self.__private_by_name_mangling = 6
6
7 example = PrivacyExample()
8 print(dir(example))
9 >>>['_PrivacyExample__private_by_name_mangling', '__class__', '
    __delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '
    __format__', '__ge__', '__getattr__', '__gt__', '__hash__', '
    __init__', '__init_subclass__', '__le__', '__lt__', '__module__', '
    __ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '
    __setattr__', '__sizeof__', '__str__', '__subclasshook__', '
    __weakref__', '_private_by_convention', 'public']
```

Snippet 8: The instance level member `__private_by_name_mangling` has been prefixed with the class name as seen from the print output.

When trying to access `__private_by_name_mangling` on the class instance it becomes clear, that the member variable cannot be referred to by this identifier.

```
1 print(example.__private_by_name_mangling)
2 >>>AttributeError: 'PrivacyExample' object has no attribute '
```

```
__private_by_name_mangling'
```

Snippet 9: Using the private identifier yields an error since the symbol has been renamed by the interpreter.

```
1 print(example._PrivacyExample__private_by_name_mangling)
2 >>>6
```

Snippet 10: The private variable can still be accessed by its mangled name but it is a convention not to do this.

In Python the `self`-parameter is a reference to the running instance of the class, and can be compared to the `this`-keyword in C#. If you forget to use `self` when declaring a parameter, the function becomes class level. If you declare a function and forget `self` in the parameter list, the function fails to run. We try this out:

```
1 class MyClass:
2     def myfun():
3         print("This is a message inside the class.")
4 myobjectx = MyClass()
5 myobjectx.myfun() //run the function
```

Snippet 11: MyClass

This results in an error, since the function was defined on the class level, but was called on an instance of the class:

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-19-1d25a04063c1> in <module>
      8 myobjectx = MyClass()
      9
--> 10 myobjectx.myfun()

TypeError: myfun() takes 0 positional arguments but 1 was given
```

Figure 8: Error when calling function that takes no self-parameter with a self-parameter

Qf - Extend the class with a Constructor

In the following code snippet `MyClass` has been extended with a constructor that initializes two instance level member variables.

```
1 class MyClass():
2     def __init__(self, a_constructor_var=4):
3         self.instance_level_var = a_constructor_var
4         self.list_of_things = [4,2,3,5,61,7,9]
5
6         class_level_var = 420
7         _semi_private_class_level_var = [4,4,4]
8         __private_class_level_var = "asdf"
9
10    def my_func(self):
11        print("This is a message from a MyClass instance.")
12
13    #Private method
14    def __my_func(self):
15        print("This method should only be called within MyClass")
16
17    @staticmethod
18    def my_static_func():
19        print("I do not belong to a MyClass instance")
20
21
22 # Calling the static method
23 MyClass.my_static_func()
24 myobjectx = MyClass()
25 # Calling the public method
26 myobjectx.my_func()
```

Snippet 12: MyClass with a constructor

When dealing with python it is very important to know the difference between class level variables and instance level variables. Class level variables behave much like static members do in C-style languages in the sense that they are not bound to an instance, they can be set and read without instantiating instances. What we mostly want when programming is instead the instance level variable, which is defined in the `__init__` function. By referencing `self` in the definition of the variable, the variable is bound to an instance of the class.

```
1 >>>instance_1 = MyClass()
```

```
2 >>>print(hasattr(instance_1,"class_level_var"))
3 True
4 >>>print(hasattr(MyClass,"class_level_var"))
5 True
6 >>>print(hasattr(instance_1,"instance_level_var"))
7 True
8 >>>print(hasattr(MyClass,"instance_level_var"))
9 False
```

Snippet 13: From this snippet it is important to note that the class level variable can be accessed from both the class and the instance.

Qg - Extend the class with a to-string function

We extend the class with the to-string functionality by providing a custom `__str__` magic method on the class. Hereby instances of the class can be converted to string by calling :

```
1 str(instance_of_MyClass)
```

Snippet 14: the `str` function is the standard interface used when converting classes to string

This is useful since other classes conform to the same interface of exposing a `__str__` method when they can be converted to string.

The `__str__` method on `MyClass` can be seen below. Note that the method itself uses the `__str__` of the dict object returned by the `vars` method.

```
1 def __str__(self):
2     return str(vars(self))
```

Calling the to-string method would look like this.

```
1 >>>instance = MyClass()
2 >>>instance_as_string = str(instance)
3 >>>print(instance_as_string)
4 {'instance_level_var': 4, 'list_of_things': [4, 2, 3, 5, 61, 7, 9]}
5 >>>print("Was the string conversion successful?",isinstance(
6     instance_as_string,str))
6 Was the string conversion successful? True
```

We chose to implement the string conversion in a way that outputs a stringified dict, since this is a common syntax and easily parsed if necessary.

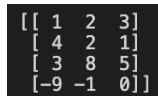
L02/Cost function

Qa - a Given the following x's, construct and print the matrix in python

We construct the matrix using np.array:

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

Snippet 15: MyClass



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

Figure 9: Printing matrix

Qb - Implement the L1 and L2 norms for vectors in python

We define the L1 and L2 norms for vectors as 'low-level' as possible:

```
1 def L1(x):
2     return np.sum(np.abs(x))
3
4 def L2(x):
5     return (np.sum(x*x))**0.5
```

Snippet 16: MyClass

Then we define L2 again using numpys dot operator to dot the array:

```
1 def L2Dot(x):
2     return np.sqrt(x.dot(x))
```

Snippet 17: L2Dot

We run the test functions and observe the output. It seems to be positive:

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

Figure 10: Output from Qb

Qc - Construct the RMSE function

The euclidian distance between two vectors in n dimensions is in itself the root of the summed squares of errors in all n dimensions. By convention this value is divided by two. As such we end up with the following formula.

```
1 def RMSE(y_pred,y_true):  
2     return 1/2 * L2Dot(y_pred - y_true)
```

Snippet 18: RMSE

Qd - Construct the MAE function

```
1 def MAE(y_pred,y_true):  
2     return L1(y_pred - y_true)/len(y_true)
```

Snippet 19: MAE

Qe - Pythonic code

```
1 def L1(x):  
2     if not (x.ndim == 1):  
3         raise Exception("Faulty input dimensions for L1")  
4     result = np.sum(np.abs(x))  
5     if not isinstance(result,Number):  
6         raise Exception("L1 return val was not a number")  
7     return result  
8  
9 def L2Dot(x):  
10    if not (x.ndim == 1):  
11        raise Exception("Faulty input dimensions for L2Dot")
```

```
12     result = np.sqrt(x.dot(x))
13     if not isinstance(result,numbers.Number):
14         raise Exception("L2Dot return val was not a number")
15     return result
16
17 def RMSE(y_pred,y_true):
18     if not (y_pred.ndim == 1 and y_true.ndim == 1 and y_pred.shape ==
19             y_true.shape):
20         raise Exception("Faulty input dimensions for RMSE")
21     result = 1/2 * L2Dot(y_pred - y_true)
22     if not isinstance(result,numbers.Number):
23         raise Exception("RMSE return val was not a number")
24     return result
25
26 def MAE(y_pred,y_true):
27     if not (y_pred.ndim == 1 and y_true.ndim == 1 and y_pred.shape ==
28             y_true.shape):
29         raise Exception("Faulty input dimensions for MAE")
30     result = 1/len(y)*(L1(x - y))
31     if not isinstance(result,numbers.Number):
32         raise Exception("MAE return val was not a number")
33     return result
```

In order to show that the exceptions are raised on appropriate errors, the following code will run our functions, catch exceptions and print out their respective exception messages. Of course it depends on the context, whether or not it is desirable to catch and handle exceptions, it is often to be preferred that the process terminates immediately after an exception is thrown.

```
1 multi_dimensional = np.array([[1,2,3], [4,2,1], [3,8,5], [-9,-1,0]])
2 len_4_a = np.array([1, 1, 3, -1])
3 len_4_b = np.array([1, 2, 2, 7])
4 len_5 = np.array([1, 2, 3, -1, 5])
5
6 try:
7     L1(multi_dimensional)
8 except Exception as e:
9     print(e)
10
```

```
11 try:
12     L2Dot(multi_dimensional)
13 except Exception as e:
14     print(e)
15
16 try:
17     RMSE(multi_dimensional, len_4_a)
18 except Exception as e:
19     print(e)
20
21 try:
22     RMSE(len_4_b, len_4_a)
23     print("RMSE ran succesfully")
24 except Exception as e:
25     print(e)
26
27 try:
28     RMSE(len_4_b, len_5)
29 except Exception as e:
30     print(e)
```

Output:

```
1 Faulty input dimensions for L1
2 Faulty input dimensions for L2Dot
3 Faulty input dimensions for RMSE
4 RMSE ran succesfully
5 Faulty input dimensions for RMSE
```

Qf - Conclusion

The concept of loss functions is essential to machine learning. In order to find suitable model parameters we need to minimize a loss function with respect to the parameters in question. What is the right loss function, does of course depend on the task at hand, but in order to make the choice, a certain amount of insight into loss functions is required. For instance, when looking at RMSE and MAE it becomes apparent, that these behave differently. The loss produced by RMSE increases exponentially when the a residual increases. This is not the

case with MAE. So choosing between these two is a question of how much the *magnitude* of the error matters.

L02/Dummy classifier

Qa - Load and display the MNIST data

Several different parameter options are available when using the `fetch_openml` function. These are mostly related to the datatype in which the data is returned. Boolean parameters such as `return_X_y` and `as_frame` decide what the returned object looks like. It is possible to get the dataset as an `sklearn.Bunch` with a `pandas.DataFrame`. It is also an option to get the dataset as an `(X,y)` tuple, where `X` is the data matrix and `y` contains target values. An option also exists for caching the dataset in which case the dataset is saved to disk when `fetch_openml` is called. Three examples of the fetch call can be seen below.

```
1 >>>mnist = fetch_openml("mnist_784",as_frame=True)
2 >>>print(type(mnist))
3 <class 'sklearn.utils.Bunch'>
4 >>>print(type(mnist.frame))
5 <class 'pandas.core.frame.DataFrame'>
```

Snippet 20: Fetching MNIST as bunch object with dataframe.

```
1 >>>mnist = fetch_openml("mnist_784",as_frame=False)
2 >>>print(type(mnist))
3 <class 'sklearn.utils.Bunch'>
4 >>>print(type(mnist.frame))
5 <class 'NoneType'>
```

Snippet 21: Fetching MNIST as bunch object without a dataframe.

```
1 >>>mnist = fetch_openml("mnist_784", return_X_y=True)
2 >>>print(type(mnist))
3 <class 'tuple'>
4 >>>print(len(mnist_3))
5 2
```

```
6 >>> print(len(mnist_3[0]))
7 70000
8 >>> print(len(mnist_3[1]))
9 70000
```

Snippet 22: Fetching MNIST as a tuple of data and target.

The tuple of X and y can be printed using MNIST_PlotDigit by indexing the tuple as shown below:

```
1 mnist = fetch_openml("mnist_784", return_X_y=True)
2 MNIST_PlotDigit(mnist[0][0])
```

Snippet 23: Fetching MNIST as a tuple of data and target.



Figure 11: The first row of the dataset displayed as a 28x28 image

Finally, the fetch_openml call is embedded a function called MNIST_GetDataSet for later use.

```
1 def MNIST_GetDataSet():
2     return fetch_openml("mnist_784", return_X_y=True)
```

Snippet 24: MNIST_GetDataSet

Qb - Add a Stochastic Gradient Decent SGD Classifier

Contrary to the expected, the MNIST data did have the correct dimensions when loaded.

```
1 >>> X, y = MNIST_GetDataSet()
2 >>> print(f"X.shape={X.shape}")
```

```
3 X.shape=(70000, 784)
```

Snippet 25: MNIST dataset dimensions

The dataset is divided with a train test split and the SGD model is trained on the training set.

```
1 from sklearn.linear_model import SGDClassifier
2 y_is_it_a_five = y == '5'
3 X_train, X_test, y_train, y_test = train_test_split(X, y_is_it_a_five)
4 # Initialize Etta Cameron classifier.
5 sgd_classifier = SGDClassifier(random_state=27)
6 sgd_classifier.fit(X_train, y_train)
```

Snippet 26: Splitting data and training the SGD model.

After training the classifier we create a predicted y column by calling the predict method on the classifier with our y_test as parameter. Afterwards we extract all the indexes with errors using list comprehension.

```
1 y_pred = sgd_classifier.predict(X_test)
2 error_indexes = [i for i, (y_p, y_t) in enumerate(zip(y_pred, y_test))
3                  if y_p != y_t]
```

Snippet 27: Predicting y values with the trained model and extracting indexes with errors.

When we evaluate the errors it is possible to find both false positives and negatives. Out of 17500 entries in the test set, 620 predictions were wrong. The accuracy score for the classifier was 0.965. Below are two examples of prediction errors, one false positive and one false negative.

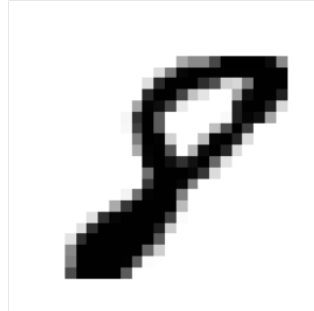


Figure 12: A false positive from the test set

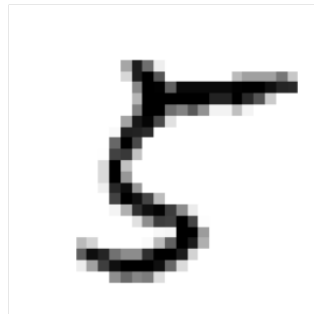


Figure 13: A false negative from the test set

Qc - Implement a dummy binary classifier

A dummy binary classifier, that always classifies as false can be implemented as shown below. Calling `fit()` does nothing, while `predict` simply returns an array with the length of the input vector, filled with the value false.

```
1 class DummyClassifier:
2     def fit(self, X_train, y_train):
3         pass
4     def predict(self, X_test):
5         return np.zeros((len(X_test), 1), dtype=bool)
```

Snippet 28: DummyClassifier class

We use the function `sklearn.metrics.accuracy_score` to calculate the accuracy score for the dummy classifier.

```
1 dummy_classifier = DummyClassifier()  
2 dummy_classifier.fit(X_train, y_train)  
3 y_pred_dummy = dummy_classifier.predict(X_test)  
4 print(accuracy_score(y_test, y_pred_dummy))
```

Snippet 29: Calculating the accuracy score for the dummy classifier

Output:

```
1 0.9098285714285714
```

In HOML chapter 3, subsection *Measuring Accuracy Using Cross-Validation* the same test is done, the output is indeed very comparable. The `dummyclassifier` in that example yields three cross val scores, which are all $> 0,9$.

A binary classifier that always classifies as false will quite obviously have an advantage if the dataset it predicts for has an overrepresentation of actual false values. In the case of the MNIST dataset 10% of dataset consists of fives, therefore it will still have an accuracy of 90% even though it yields false negatives for all the actual 5-values. This is a great example of why it is important to look at other metrics than just the accuracy. Evaluating the amount of false negatives would reveal that the dummy classifier is in fact useless.

Qd - Conclusion

The key takeaway from this exercise is that an accuracy score isn't a good metric in and of itself. It needs to be compared against precision and recall of the classifier. The accuracy depends on the representation of the different classes in the dataset. In the case of the dummy classifier, the recall score would be zero, even though the accuracy score was seemingly high.

L02/Performance metrics

Qa - Implement the Accuracy function

Accuracy is defined as:

$$a = \frac{TP + TN}{N_P + N_N}$$

To find all correctly identified results (TP + TN):

```
1 y_true == y_pred.T
```

Snippet 30: TP + TN

To find the total number of samples: ($N_P + N_n$):

```
1 np.array(y_true).shape[0]
```

Snippet 31: $N_P + N_n$

Combining to find accuracy:

```
1 def MyAccuracy(y_true, y_pred):  
2     return np.sum(y_true == y_pred.T) / np.array(y_true).shape[0]
```

Snippet 32: Accuracy function definition

The MyAccuracy function was tested using the supplied function TestAccuracy(y_true, y_pred). The output from this testfunction did show that our accuracy function worked as anticipated.

```
1 sgd_classifier.fit(X_train, y_train)  
2 dummy_classifier.fit(X_train, y_train)  
3  
4 y_pred_sgd = sgd_classifier.predict(X_test)  
5 y_pred_dummy = dummy_classifier.predict(X_test)  
6  
7 TestAccuracy(y_test, y_pred_sgd)  
8 TestAccuracy(y_test, y_pred_dummy)
```

Output:

```
1 my a          =0.9661714285714286  
2 scikit-learn a=0.9661714285714286
```

```
3
4 my a = 0.9096
5 sklearn a=0.9096
```

Qb - Implement Precision, Recall and F1-score

Implement precision

Precision is defined as:

$$p = \frac{TP}{TP + FP}$$

To find true positives (TP):

```
1 np.logical_and((y_true == True), (y_pred.T == True)))
```

Snippet 33: TP

To find all predicted positives (TP + FP):

```
1 np.sum(y_pred == True)
```

Snippet 34: TP + FP

Combining to find precision:

```
1 def MyPrecision(y_true, y_pred):
2     if np.sum(y_pred == True) == 0:
3         return 0
4     return np.sum(np.logical_and((y_true == True), (y_pred.T == True)))
5         / np.sum(y_pred == True)
6
7 #Alternative
8 def MyPrecision2(y_true, y_pred):
9     tp = np.sum(np.logical_and((y_true == True), (y_pred==True)))
10    fp = np.sum(np.logical_and((y_true == False), (y_pred==True)))
11    return 0 if ((tp + fp) == 0) else (tp/(tp+fp))
```

Snippet 35: Precision function definition

Implement recall

Recall is defined as:

$$r = \frac{TP}{N_P}$$

To find all positive samples (N_P):

```
1 np.sum(y_true == True)
```

Snippet 36: N_P

Precision can be defined as:

```
1 def MyRecall(y_true, y_pred):
2     if np.sum(y_pred == True) == 0:
3         return 0
4     return np.sum(np.logical_and((y_true == True), (y_pred.T == True)))
5         / np.sum(y_true == True)
6
7 # Alternative
8 def MyRecall2(y_true, y_pred):
9     tp = np.sum(np.logical_and((y_true == True), (y_pred==True)))
10    fn = np.sum(np.logical_and((y_true == True), (y_pred==False)))
11    return 0 if ((tp + fn) == 0) else (tp/(tp+fn))
```

Snippet 37: Recall function definition

Implement F1Score

F1Score is defined as:

$$F_1 = \frac{2}{\frac{1}{p} + \frac{1}{r}}$$

Using our definitions of precision and recall, we can define F1Score as:

```
1 def MyF1Score(y_true, y_pred):
2     p = MyPrecision(y_true, y_pred)
3     r = MyRecall(y_true, y_pred)
4     return np.sum(2/(1/p+1/r))
```

Snippet 38: F1Score function definition

The metric methods are tested with the following methods. If the metric methods do not correspond to the sklearn supplied equivalent methods an assertion error is thrown.

```
1 def TestPrecision(y_true, y_pred):
2     a0=MyPrecision(y_true, y_pred)
3     a1=precision_score(y_true, y_pred)
4
5     print(f"\nmy a          ={a0}")
6     print(f"scikit-learn a={a1}")
7     assert a0 == a1
8
9 def TestRecall(y_true, y_pred):
10    a0=MyRecall(y_true, y_pred)
11    a1=recall_score(y_true, y_pred)
12
13    print(f"\nmy a          ={a0}")
14    print(f"scikit-learn a={a1}")
15    assert a0 == a1
```

Snippet 39: Test methods for precision.

Qc - The Confusion Matrix

```
1 from sklearn.metrics import confusion_matrix
2
3 print(confusion_matrix(y_test,y_pred_sgd))
4 print()
5 print(confusion_matrix(y_test,y_pred_dummy))
```

Output:

```
1 [[15575   343]
2  [   249 1333]]
3
4 [[15918     0]
5  [ 1582     0]]
```

Snippet 40: Confusion matrix for the sgd classifier(top) and the dummy classifier(bottom)

The documentation for the `confusion_matrix` method specifies that the matrix is configured as follows:

TN FP

FN TP

When looking at the confusion matrix for the dummy classifier it becomes obvious that it is a dysfunctional classifier, as we can see that it has zero true positives.

If we accidentally reverse the order of parameters in the `confusion_matrix` call, we get the following output:

```
1 [[15575    249]
2  [   343  1333]]
3
4 [[15918    1582]
5  [     0     0]]
```

Snippet 41: Confusion matrix for the `sgd` classifier(top) and the dummy classifier(bottom). Reversed parameters in the `confusion_matrix` call

From the above matrixes it can be seen that the output data has not been corrupted, but rather that the matrixes have been transposed. In this case the composition as follows:

TN FN

FP TP

Qd - A Confusion Matrix Heat-map

Since the only about a ninth of our dataset is the of the 5-class, a heat map of the errors does offer much insight unless it is first normalized. The amount of errors needs to be evaluated with respect to the number of potential errors. The code snippet below shows how the confusion matrix is created, normalized and plotted.

```
1 sgd_conf_matrix = confusion_matrix(y_test,y_pred_sgd)
2
3 sums_of_classes_sgd = sgd_conf_matrix.sum(axis=1, keepdims=True)
```

```
4 norm_conf_matrix_sgd = sgd_conf_matrix / sums_of_classes_sgd
5 np.fill_diagonal(norm_conf_matrix_sgd, 0)
6 plt.matshow(norm_conf_matrix_sgd, cmap=plt.cm.gray)
7 plt.show()
```

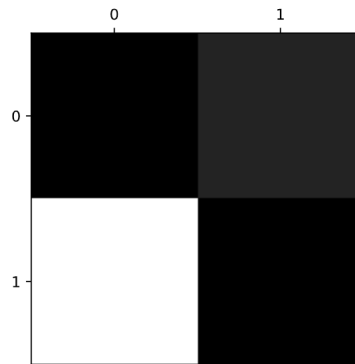


Figure 14: Normalized heat map for the sgd classifier confusion matrix.

In the confusion matrix for the SGD classifier we have 343 false negatives and 249 false positives. So in absolute values, we have more false negatives than false positives. However, figure 14 shows us, that it is in fact the false positives that are the main issue of this classifier. We see that the upper right corner, FN, has a dark grey color while the lower left corner, FP, is white. This means that the normalized amount of false positives is significantly larger than that of false negatives. This is because the proportions of the two classes are quite different. The model only misclassified 343 out of 15.918 possible negative misclassifications, while it misclassified 249 out of 1.582 possible misclassifications, where the latter is a much larger proportion.

Below on figure 15 is a confusion matrix heat map for the dummy classifier. By looking at this heat map it is very obvious that the false negatives in the lower left corner are the only errors present. This is due to the fact that the dummy classifier never classifies anything as positive.

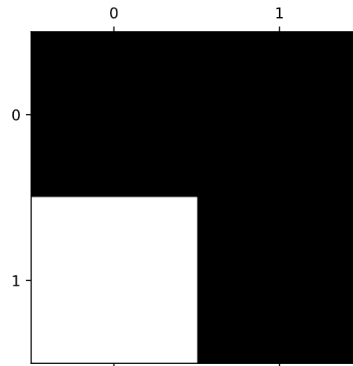


Figure 15: Normalized heat map for the dummy classifier confusion matrix.

Qe - Conclusion

An important lesson from this exercise has been, that no performance metric can be used as a standalone solution. Multiple metrics need to be evaluated in order to determine the performance of a model. We saw a great example of this with the dummy classifier, where the accuracy was quite high even though the classifier was useless. The false flag of the high accuracy could easily be noticed, just by looking at the recall and precision scores. Another great thing to note from this exercise is the importance of normalizing data, for instance when looking at error values. In the heat map exercise we saw how FP and FN values are only comparable when they have been normalized with respect to the number of possible errors.

L03/Pipelines

Qa - Create a Min/max scaler for the MLP

We do a test-train-split as we did in L01/intro:

```
1 X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=29)
```

Snippet 42: Test-train-split

We create a min/max-scaler that scales an array so that no number is smaller than 0 and larger than 1:

```
1 def Scaler(array):  
2     diff = max(array) - min(array)  
3     offset = min(array)  
4     return (array - offset)/diff
```

Snippet 43: Scaler

We test with some random numbers:

```
1 array = np.array([4,7,1,6,8,-4])  
2 print(Scaler(array))
```

Snippet 44: Test of Scaler

Output:

```
1 [0.66666667 0.91666667 0.41666667 0.83333333 1. 0.]
```

The scaler works, and so we try using it on our training set before handing it to the MLP:

```
1 X_train_scaled = Scaler(X_train)  
2 y_train_scaled = Scaler(y_train)  
3 X_test_scaled = Scaler(X_test)  
4 y_test_scaled = Scaler(y_test)  
5  
6 mlp = MLPRegressor( hidden_layer_sizes=(10,), solver='adam', activation  
    = 'relu', tol=1E-5, max_iter=100000, verbose=False)  
7  
8 mlp.fit(X_train_scaled,y_train_scaled.ravel())  
9  
10 print("Score: ", mlp.score(X_test_scaled,y_test_scaled.ravel()))
```

Snippet 45: Using our scaler with MLP

Output:

```
1 0.5586629905642001
```

The output indicates that the data is scaled correctly.

Qb - Scikit-learn Pipelines

We scale the data again using `sklearn.preprocessing.MinMaxScaler` and `sklearn.pipeline.Pipeline`:

```
1 mlp_pipe = Pipeline([('scaler', MinMaxScaler()), ('mlp', MLPRegressor(  
2     hidden_layer_sizes=(10,), solver='adam',  
3     activation='relu', tol=1E-5, max_iter=100000, verbose=False))])  
4  
5 mlp_pipe.fit(X_train, y_train.ravel())  
6  
7 print("Score:", mlp_pipe.score(X_test, y_test.ravel()))
```

Output:

```
1 Score: 0.7163382280969754
```

The Min-max scaler gives a much better score than our selfmade scaler did. It seems that using the correct scaling for the dataset can have a big impact on the accuracy of the regression.

Qc - Outliers and the Min-max Scaler vs. the Standard Scaler

When a dataset has a number of outliers, the mean can be skewed in relation to the inliers, which leads to poorer performance by scalers that calculate the mean and use that to scale the data.

To achieve a better score, scaling algorithms that use the median or percentiles to standardize the data can be used.

Both the Min-max-scaler and Standard scaler use the mean and are very sensitive to outliers, and so the `sklearn.preprocessing.StandardScaler` should not perform better in this case.

However, in testing the Standard scaler has actually performed better than the Min-max-scaler, giving fits about 0.05 better than the Min-max-scaler.

Qd - Modify the MLP Hyperparameters

We adjust the number of neurons by changing the `hidden_layer_sizes` input parameter. Going as low as 2 neurons doesn't seem to affect the score. Using a single neuron seems to be hit or miss with either a good score in the .7x range or a bad score in the -.6x range.

We try some different activation methods: Using the "logistic" activation method consistently gives a score in the .71 to .72-range. The "tanh" method is less reliable and produces scores from .70 to .78.

We try using the "sgd" solver method, which we find produces wildly different results, varying from -2 to 0.7.

Litteratur

- [1] Aarhus Universitet. Aarhus universitet logo. https://projects.au.dk/fileadmin/projects/ausat/ausegl_blaa.png. Accessed on 2020-06-03.
- [2] Scikit linear regression model. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression.score.
- [3] Scikit k-nearest neighbors model. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html#sklearn.linear_model.LinearRegression.score.