Assignment 2

By Niklas Alexsander

How to run

To run the assignment you simply have to open up the jupyter notebook provided and run the code IN ORDER. The order is really important! Start at the top. When testing the code I only used the run button at the top of the jupyter notebook to ensure the right order :)

Binary classifiers

Linear regression

To implement the diff-variable I made a function called MSE. This function calculate the mean squared error. Then I check the MSE on the a new run compared to the previous run to see if there has been any improvements/change. If the change is less than the diff-variable the loop will stop. For each iteration of the while-loop inside of the fit-function inside of LinReg-classifiers is an epoch. The code used to represent the LinReg-classifier is close to the one given in one of the weekly-exercises.

The calculated accuracies for (X_train, t2_train, X_val, t2_val) is as follows with a gamma of 0.01:

diff	Accuracy	Epochs
0.001	0.51	19
0.0001	0.56	51
0.00001	0.58	83
0.000001	0.6	115
0.0000001	0.605	148
0.01	0.4775	3
0.1	0.47	2

The best accuracy is is 0.605, diff= 0.0000001

Logistic Regression

To add the diff for Logistic regression I had to use another loss function than the MSE since MSE is not 'using' the 'flexible' state that the logistic regression is obtaining. While testing I tried to use the LLS (logistic loss function), but ended up with the use of CEL (Cross-entropy-loss) as advised on a lecture. The LLS-function is not removed, it is only as a comment inside of LogReg.

The principle of using diff is the same as in LinReg, the difference being the CEL instead of MSE.

The calculated accuracies is as follows with a gamma of 0.01:

diff	Accuracy	Epochs

diff	Accuracy	Epochs
0.001	0.5975	19
0.0001	0.605	51
0.00001	0.605	83
0.000001	0.6075	115
0.0000001	0.6075	148
0.01	0.5875	3
0.1	0.5875	2

The best accuracy is is 0.6075, diff= 0.000001

K-Nearest Neighbours (kNN)

Since the provided knn-code from one of the weekly exercises is made to work on numpy lists, this is the code that's the slowest to run in the whole assignment (same speed with knn multi-class). The accuracies is stored together with the corresponding k-value. While testing I used a k-value of between 1 to 30, skipping 1 every time. This to ensure that we do not get a 50/50 when picking a label.

The task wants us to test on (X2_val, x2_val). These do not exist from the given sets. I therefore tested on the X_val and t2_val.

The calculated accuracies for (X_train, t2_train, t_val, t2_val) is as follows:

k-value	Accuracy
1	0.67
3	0.68
5	0.7125
7	0.735
9	0.7325
11	0.7375
13	0.7475
15	0.7525
17	0.75
19	0.7525
21	0.7575
23	0.75
25	0.7525
27	0.7475
29	0.755

The most optimal k found in range(1, 30, 2) is: 21
k=21 gives an accuracy of: 0.7575

Simple Perceptron

Using the perceptron-code given in the weekly exercise (week05) as suggested.

The calculated accuracies for (X_train, t2_train, t_val, t2_val) is as follows:

Epochs	Accuracy
1	0.647
2	0.650
3	0.593
4	0.588
5	0.647
6	0.625
7	0.583
8	0.650
9	0.650
10	0.595
11	0.583
12	0.565
13	0.613
14	0.647
15	0.593
16	0.583
17	0.662
18	0.598
19	0.595

The best accuracy is is 0.662, 17 epochs

SUMMARY

Linear regression:

The results I got when testing the code for linear regression seems to me to be correct. Depending on where the line ends in each run but the line seems to give a result of around 60% with a diff of 0.00001. My thoughts about this percentage is that a linear regression on the given points is impossible with an accuracy of close to 100%. This is because the classes are scattered around and on top each other, there is no clear separable 'visual' line between the classes. The 'orange' class is more denser packed than the 'blue' class, but there is still 'blue' around 'orange' and vice versa.

Logistic regression:

There is a cluster of 'orange' points in the middle and the 'blue' points looks like a wide rectangle going through the orange points. This made me believe that the logistic regression could not see any great improvement over the Linear regression. I think one of the better solutions to the points given would be a non-linear logistic regression that would create a circle, that could create a regression line (circle) around the 'orange' class. The best accuracy percentage while testing was around 60% with a diff of 0.0001.

k-nearest neighbour:

The k-nearest neighbour gives the best result out of all of the four tested method. This was my predicted outcome as well. This because of the principle of k-nearest which is to check the neighbouring-points, which with a great enough k would 'find' clusters of points that is close to each other. This could find the great amount of 'orange' points that's close to each other and give a greater accuracy to both the 'orange' points in the middle and the 'blue' points that is a great distance away from the 'orange' cluster. The best accuracy percentage while testing was around 75%, with k=21 giving the best result from 1 to 30.

Simple perceptron:

Since the simple perceptron is a type of linear classifier my prediction of the results given by the perceptron would be around the same as for the linear regression and the logistic regression. While looking at the test-results we see that this seems true while testing with different epochs from 1 to 20. The best accuracy percentage while testing was around 66% with 17 epochs, which is actually a small improvement over linear and logistic!

OVERALL BEST ACCURACY TABLE:

LinReg	LogReg	kNN	PerTrn
0.605	0.6075	0.7575	0.662

Multi-class Classifiers

kNN

Since the code for knn works with multi-labeled sets there is no change to this code-part.

k-value	Accuracy
1	0.6625
3	0.675
5	0.7075
7	0.73
9	0.7425
11	0.7475
13	0.75
15	0.7575
17	0.7525
19	0.755

k-value	Accuracy
21	0.76
23	0.7525
25	0.7525
27	0.75
29	0.755

The most optimal k found in range(1, 30, 2) is: 21
k=21 gives an accuracy of: 0.76

Logistic Regression

To make the Logistic Regression to work with multiple more than 2 labels, I created a "one-vs-rest" solution where I split the data by it's labels, calculate an accuracy for each label, and then pick the highest rated accuracy as the one for the given targets. To find the overall accuracy of this approach I take the product of the accuracies calulated divided by the number of accuracies, which in this case is 3.

I created the scalar-to-array-function but did not use it in this task. I however am using it in the later tasks.

The calculated accuracy of each class with a gamma of 0.0001 and diff of 0.00000000001:

Had to use a large diff in order to see the runs. but not too large, because the class1 takes a lot longer to calculate than the other classes

Class 0	Class 1	Class 2
0.81	0.61	0.7425

The overall accuracy while testing was 0.7208333333333333.

While running the 'ovr' I saw an great time difference with class 0 and class 2 VS class 1. Class 1 with the dots in the middle is the hardest of them all. This also reflects greatly in the calculated acccuracies where Class 1 is rather poor compared to Class 0 and class 2.

In this part of the assignment in the Jupyter notebook I have added the plots for the different sets used on the LogReg.

results discussion

How do the two classifiers compare?

- The two classifiers KNN and OVR-logistic regression is fairly similar in performance, but the KNN is a bit better than the OVR. The KNN got an accuracy of 0.76 with k= 21 while testing while the OVR got an accuracy of 71%, so not a giant difference but a significant one nonetheless. The OVR can get an accuracy of 60% on class 1 but an 80% on class 0 (numbers while testing) and 74% on class 2. So the classifier would be the best at finding the right label for class 0, and not quite as good on finding the label of class 1.

How do the results on the three-class classification task compare to the results on the binary task?

- Since the OVR is basically a binary classification used on different 'views' of a given set of values, the results should be the same as running the binary three times (with different sets ofc)! But as 'one-run-only' the OVR is better than the binary LogReg with over 10% while testing. The KNN gives a slightly better score compared to the binary test earlier. But this will depend on the test-values given, so the overall stability of KNN is better with all the classes.

What do you think are the reasons for the differences?

- When there is one more class to pick from the probability changes from 1/2 to 1/3, but in the same time the 'clusters', as seen in the representation of the points in a graph, does not get noticed, and the blend between two classes could be separated by another class, which we can see with 'class 0' and 2 which got 'class 1' in the middle, separating the classes. With 'class 1' in the middle, the accuracy of a point from 'class 0' that will be wrongly predicted to be in 'class 2' will be reduced by having another class in between the two classes. This is because if the given training set labels does not contain the 'class 1' the points which should be in class 1 will be wrongly calculated to either 'class 0' or 'class 2'. Since the classes are on top of and around each other the separation of the classes with a single line will be quite poor.

Adding non-linear features

Explain in a couple of sentences what effect the non-linear features have on the various classifiers:

- When testing with an learning rate of '0.1' I couldn't find any advantages of using the added weights, but when I changed the learning rate to '0.001' there's a quite a bit of improvement to the results.
  When using the non-linear features on the various classifiers makes the classifiers 'see' the 'vectors' in a new way, which may lead to a better separation of points that are close to each other with the basic (x,y) points.
  This could lead the classifiers to 'wrap' around the classes aswell.

- As we see, the linear regression gets quite a boost, from around 60% to 73%, nearly as good as knn!
  knn is actually a non-linear classifier, so any result that is close to knn is actually pretty good!

- No super advantage with perceptron at this point.

Accuracy table comparing with/without non-linear features:

LinReg	LogReg	kNN	PerTrn	Nmb W
0.605	0.6075	0.7575	0.662	2
0.7325	0.52	0.7525	0.618	5

Part II

Multi-layer neural networks

The first thing I did was to make a min_max scaler and a normalization scaler. I tested both and found the best result with normalization. Code for both scalers are in the jupyter notebook.

Step1: One round of training

Starting with initializing the learning rate as 0.01, and the number of hidden nodes as 6.

The number of incoming nodes is of 2. This is the number of features for each instance of the X_train set. (800 instances, 2 features)

The number of outgoing nodes is of 3. This is because we got 3 classes to work with. This number is found by finding the unique numbers in the t_train.

Then the w1 is being created by dim_in, dim_hidden.
w2 from dim_hidden, dim_out.
Using the numpy.random.rand().

I created two arrays as bias containing -1, the size of dim_hidden and dim_out.

Forwards phase

To do the forwards-motion of calculating from in to hidden then hidden to out, I calculated the activations with the dot product of X_train and w + bias, sending this to the sigmoid-function squeezing the calculations to numbers between 0 and 1.

Inspecting the numbers and I see that the values are indeed between 0 and 1.

Backwards phase

Here I am using the scalar_to_array-function to convert t_train to y_train. This is so we can calculate the delta_output using matrix-calculations.

```
delta_output = (y_train - output_activations)
                    * (output_activations *
                    (1 - output_activations))
```

delta_hidden:

```
delta_hidden = hidden_activations_with_bias *
                    (1 - hidden_activations_with_bias) *
                    (delta_output @ w_hid_out.T)
```

At the end we need to update the weights and see that they have changed.

Step2: A Multi-layer neural network classifier

The multi-later neural network classifier is more or less the same code as the code given in the step1, it's just cleaner with some parts of the code placed in their own function.

While working on the accuracy-function I tested with scikit to see that the results are the same. So it should do as well as sklearn.metrics.accuracy :)

Part III Final testing

While testing my MLP I got an accuracy of 0.75-77%! That's around the same as my expectations :) I believed that the MLP would get around the same accuracy of the KNN, and it did! And the most fun thing about this achievement is the speed of MLP compared to KNN. KNN needs to go through a lot of k-values before we find the one that is optimal. While the MLP finds a reasonable answer in split second!

I took the kNN vs MLP as the best classifiers made in the jupyter notebook.

The kNN gets an accuracy of around 0.76 and k=27 on both (X_test, t_test) and (X_test, t2_test) while training on (X_train, t_train) for the first one, and (X_train, t2_train) for the second.

MLP got a really good accuracy with a learning rate of 0.001 and 3000 ephocs on the first one with an accuracy of 0.755 and an accuracy of 0.7425 on the second!
Thats a really good achievement compared to the speed differences between kNN and MLP!

Confusion matrix for MLP

	predicted	
152	30	13
34	79	0
20	0	72

Confusion matrix for kNN

	predicted	
160	27	8
38	75	0
23	0	69

The confusion matrixes are not that different from each other.

Precisions MLP

Class0	Class1	Class2
0.7379	7248	0.8470

Recalls MLP

Class0	Class1	Class2
0.7785	0.5752	0.7826

Precision kNN

Class0	Class1	Class2
0.7240	0.7343	8961

Recalls kNN

Class0	Class1	Class2
0.8205	0.6637	0.75

So overall the kNN may be slower, but it will deliver a result that is not as affected by the nuances of the training set if the set is of good shape or is well scaled. While my MLP got custom tweaks by me while testing on the scaled (X_train, t_train). This may have led to me tweaking to much in the favor of X_train, t_train, rather than taking the most logical 'performance enhancers'. There could also be an improvement from changing the scaler to something else. While Testing I found out that the scaler is a really important factor to the accuracy of the MLP. Changing the scaler gave me a difference in accuracy from 50% or below to 70+%! Quite a huge gap with a simple correction. Maybe even adding the three extra features would be a good place to start looking at improving the MLP?

I hope that my explanations has been adequate :)