SupervisedLearning

March 29, 2022

0.0.1 Initialization

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import sklearn
from sklearn import datasets
import random
from operator import itemgetter #I import "itemgetter" to get the index of␣
↪element
```

1 Part 1: Comparing classifiers

1.1 Datasets

We start by making a synthetic dataset of 1600 datapoints and three classes, with 800 individuals in one class and 400 in each of the two other classes. (See https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html#sklearn.datasets.make_blobs regarding how the data are generated.)

When we are doing experiments in supervised learning, and the data are not already split into training and test sets, we should start by splitting the data. Sometimes there are natural ways to split the data, say training on data from one year and testing on data from a later year, but if that is not the case, we should shuffle the data randomly before splitting. (OK, that is not necessary with this particular synthetic data set, since it is already shuffled by default by scikit, but that will not be the case with real-world data.) We should split the data so that we keep the alignment between X and t, which may be achieved by shuffling the indices. We split into 50% for training, 25% for validation, and 25% for final testing. The set for final testing *must not be used* till the end of the assignment in part 3.

We fix the seed both for data set generation and for shuffling, so that we work on the same datasets when we rerun the experiments.

[2]:
```
from sklearn.datasets import make_blobs
X, t = make_blobs(n_samples=[400,800,400], centers=[[0,0],[1,2],[2,3]],
                  n_features=2, random_state=2019)
```

[3]:
```
indices = np.arange(X.shape[0])
random.seed(2020)
random.shuffle(indices)
indices[:10]
```

[3]:
```
array([1301,  293,  968,  624,  658,  574,  433,  368,  512,  353])
```

[4]:
```
X_train = X[indices[:800],:]
X_val = X[indices[800:1200],:]
X_test = X[indices[1200:],:]
t_train = t[indices[:800]]
t_val = t[indices[800:1200]]
t_test = t[indices[1200:]]
```

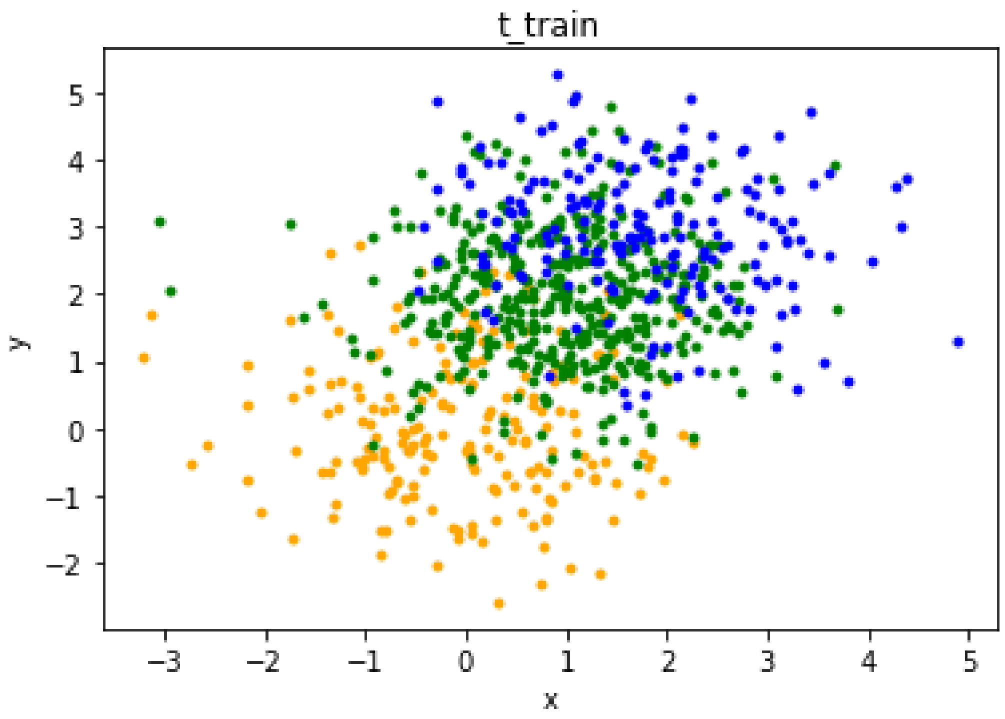
Next, we will make a second dataset by merging the two smaller classes in (X,t) and call the new set (X, t2). This will be a binary set.

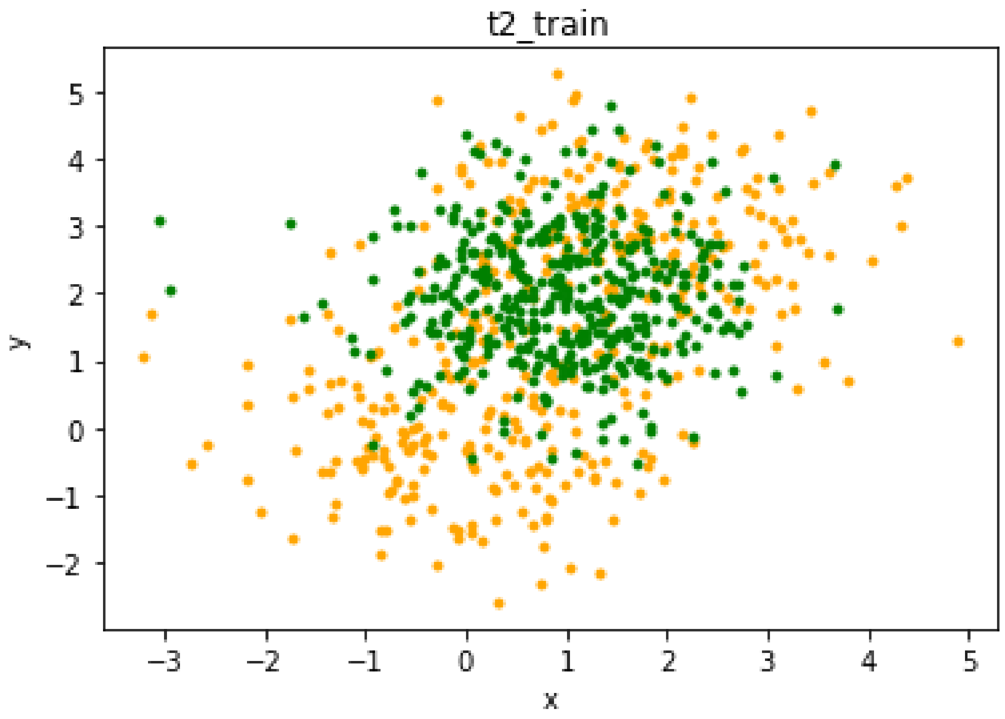
[5]:
```
t2_train = t_train == 1
t2_train = t2_train.astype('int')
t2_val = (t_val == 1).astype('int')
t2_test = (t_test == 1).astype('int')
```

Plot the two training sets.

[64]:
```
#appending in to a dictionary
cls = {0:[], 1:[], 2:[]}
for (a,b) in zip(X_train, t_train):
    cls[b].append(a)

#plotting
for i in cls[0]:
    plt.plot(i[0], i[1], ".", label="class 0", color="orange")

for i in cls[1]:
    plt.plot(i[0], i[1], ".", label="class 1",  color="green")

for i in cls[2]:
    plt.plot(i[0], i[1], ".", label="class 2",  color="blue")

#plt.legend()
plt.title("t_train")
plt.xlabel("x")
plt.ylabel("y")
plt.show()



#appending in to a dictionary
cls = {0:[], 1:[], 2:[]}
```

```python
for (a,b) in zip(X_train, t2_train):
    cls[b].append(a)

#plotting
for i in cls[0]:
    plt.plot(i[0], i[1], ".", label="class 0", color="orange")

for i in cls[1]:
    plt.plot(i[0], i[1], ".", label="class 1",  color="green")

#plt.legend()
plt.title("t2_train")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```





1.2 Binary classifiers

1.2.1 Linear regression

We see that that set (X, t2) is far from linearly separable, and we will explore how various classifiers are able to handle this. We start with linear regression. You may use the implementation from exercise set week07 or make your own. You should make one improvement. The implementation week07 runs for a set number of epochs. You provide the number of epochs with a parameter to the fit-method. However, you do not know what a reasonable number of epochs is. Add one more argument to the fit-method *diff* (with defualt value e.g. 0.001). The training should stop when the update is less than *diff*. The *diff* will save training time, but it may also be wise to not set it too small – and not run training for too long – to avoid overfitting.

Train the classifier on (X_train, t2_train) and test for accuracy on (X_val, t2_val) for various values of *diff*. Choose what you think is optimal *diff*. Report accuracy and save it for later.

[7]:
```
def add_bias(X):
    # Put bias in position 0
    sh = X.shape
    if len(sh) == 1:
        #X is a vector
        return np.concatenate([np.array([1]), X])
    else:
        # X is a matrix
        m = sh[0]
        bias = np.ones((m,1)) # Makes a m*1 matrix of 1-s
        return np.concatenate([bias, X], axis  = 1)



class NumpyClassifier():
    def accuracy(self,X_test, t2_test, **kwargs):
        pred = self.predict(X_test, **kwargs)
        if len(pred.shape) > 1:
            pred = pred[:,0]
        return sum(pred==t2_test)/len(pred)

class NumpyLinRegClass(NumpyClassifier):
    def fit(self, X_train, t_train, eta = 0.1, epochs=1, diff=0.001):
        (k, m) = X_train.shape
        X_train = add_bias(X_train)
        self.weights = weights = np.zeros(m+1)

        update = 1
        weights -= eta / k *  X_train.T @ (X_train @ weights - t2_train)
        temp = weights.copy()
        while update > diff:
            weights -= eta / k *  X_train.T @ (X_train @ weights - t2_train)
            update = sum(abs(weights - temp))
            temp = weights.copy()
            epochs += 1



    def predict(self, x, threshold=0.5):
        z = add_bias(x)
        score = z @ self.weights
        return score>threshold




lin_cl = NumpyLinRegClass()
lin_cl.fit(X_train, t2_train)
accuracy_lin = lin_cl.accuracy(X_val, t2_val)
print("Linear regression accuracy is", accuracy_lin)
```

```
Linear regression accuracy is 0.58
```

1.2.2 Logistic regression

Do the same for logistic regression, i.e., add the *diff*, tune it, report accuracy, and store it for later.

[8]:
```python
def logistic(x):
    return 1/(1+np.exp(-x))

class NumpyLogReg(NumpyClassifier):

    def fit(self, X_train, t_train, eta = 0.1, epochs=1, diff=0.001):
        (k, m) = X_train.shape
        X_train = add_bias(X_train)

        self.weights = weights = np.zeros(m+1)

        update = 1
        weights -= eta / k *  X_train.T @ (self.forward(X_train) - t_train)
        temp = weights.copy()
        while update > diff:
            weights -= eta / k *  X_train.T @ (self.forward(X_train) - t_train)
            update = sum(abs(weights - temp))
            temp = weights.copy()
            epochs += 1


    def forward(self, X):
        return logistic(X @ self.weights)

    def score(self, x):
        z = add_bias(x)
        score = self.forward(z)
        return score

    def predict(self, x, threshold=0.5):
        z = add_bias(x)
        score = self.forward(z)
        return (score>threshold).astype('int')

lr_cl = NumpyLogReg()
lr_cl.fit(X_train, t2_train)
print("Logistic regression accuracy is", lr_cl.accuracy(X_val, t2_val))
```

```
Logistic regression accuracy is 0.625
```

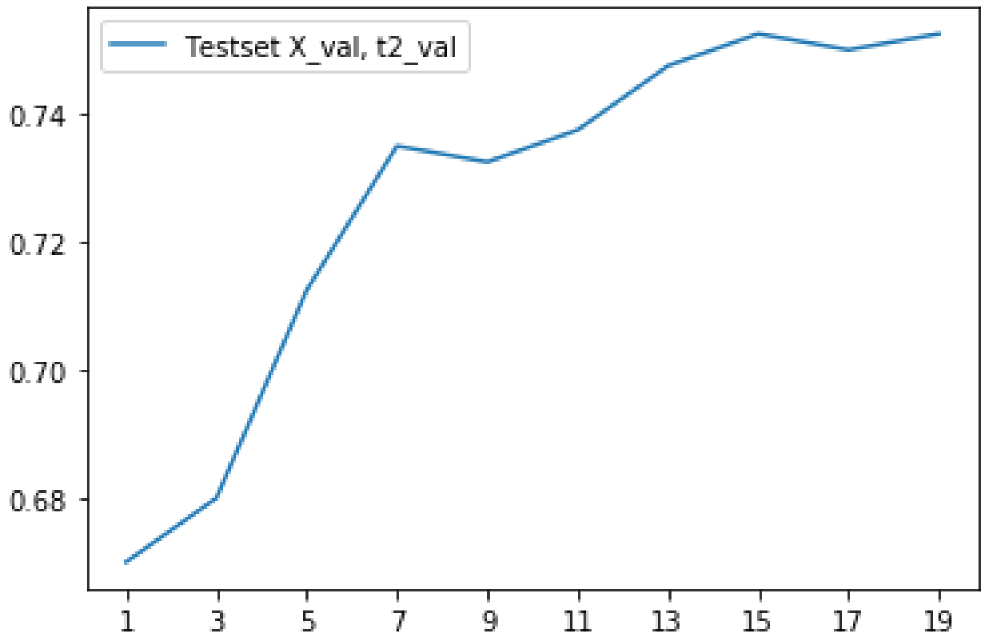
1.2.3 *k*-nearest neighbors (*k*NN)

We will now compare to the *k*-nearest neighbors classifier. You may use the implementation from the week05 exercise set. Beware, though, that we represented the data differently from what we do here, using Python lists instead of numpy arrays. You might have to either modify the representation of the data or the code a little.

Train on (X_train, t2_train) and test on (X2_val, t2_val) for various values of *k*. Choose the best *k*, report accuracy and store for later.

[66]:
```python
from collections import Counter

class PyClassifier():
    def accuracy(self,X_test, y_test, **kwargs):
        predicted = [self.predict(a, **kwargs) for a in X_test]
        equal = len([(p, g) for (p,g) in zip(predicted, y_test) if p==g])
        return equal / len(y_test)


def distance_L2(a, b):
    "L2-distance using comprehension"
    s = sum((x - y) ** 2 for (x,y) in zip(a,b))
    return s ** 0.5

def majority(a):
    counts = Counter(a)
    return counts.most_common()[0][0]


class PykNNClassifier(PyClassifier):
    def __init__(self, k=3, dist=distance_L2):
        self.k = k
        self.dist = dist

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train


    def predict(self, a):
        X = self.X_train
        y = self.y_train
        distances = [(self.dist(a, b), b, c) for (b, c) in zip(X, y)]
        distances.sort()
        predictors = [c for (_,_,c) in distances[0: 1]]
        return majority(predictors)
```

```
x = range(1, 20, 2)
accuracies = []
for l in x:
    cls = PykNNClassifier(k=l)
    cls.fit(X_train.tolist(), t2_train.tolist())
    accuracies.append(cls.accuracy(X_val.tolist(), t2_val.tolist()))


plt.plot(x, accuracies, label="Testset X_val, t2_val")
plt.xticks(x)
plt.legend()
plt.show()



idx = max(enumerate(accuracies), key=itemgetter(1))[0]
print("KNN accuracy is", max(accuracies), "with k =", idx)
```



```
KNN accuracy is 0.7525 with k = 7
```

1.2.4 Simple perceptron

Finally, run the simple perceptron (week06) on the same set, and report and store accuracy.

[11]:
```python
class PyClassifier():
    """Common methods to all python classifiers --- if any"""

    def accuracy(self,X_test, y_test, **kwargs):
        """Calculate the accuracy of the classifier
        using the predict method"""
        predicted = [self.predict(a, **kwargs) for a in X_test]
        equal = len([(p, g) for (p,g) in zip(predicted, y_test) if p==g])
        return equal / len(y_test)

class PyPerClassifier(PyClassifier):
    """Simple perceptron python classifier"""

    def fit(self, X_train, y_train, eta=1, epochs=1):
        """Train the self.weights on the training data with learning
        rate eta, running epochs many epochs"""
        X_train = [[1]+list(x) for x in X_train] # Put bias in position 0
        self.dim = dim = len(X_train[0])
        self.weights = weights = [0 for _ in range(dim)]
        # Initialize all weights to 0. There are better ways!

        for e in range(epochs):
            for x, t in zip(X_train, y_train):
                y = int(self.forward(x)>0)
                for i in range(dim):
                    weights[i] -= eta * (y - t) * x[i]

    def forward(self, x):
        """Calculate the score for the item x"""
        score = sum([self.weights[i]*x[i] for i in range(self.dim)])
        return score

    def predict(self, x):
        """Predict the value for the item x"""
        x = [1] + list(x)
        score = self.forward(x)
        return int(score > 0)

test = []
for i in range(0,20):
    cl = PyPerClassifier()
    cl.fit(X_train, t2_train, eta=0.1, epochs = i)
    test.append(cl.accuracy(X_val, t2_val))


epochs_index = max(enumerate(test), key=itemgetter(1))[0]
print("Simple perceptron accuracy is", max(test), "with k =", epochs_index)
```

```
Simple perceptron accuracy is 0.6625 with k = 17
```

1.2.5 Summary

Report the accuracies for the four classifiers in a table.

Write a couple of sentences where you comment on what you see. Are the results as you expected?

Classifiers:	Accuracy
Linear Regreesion	0.58
Logical Regression	0.625
k-nearest neighbors (kNN)	0.7525
Simple perceptron	0.6625

I see that KNN got the best accuracy, while Linear Regression got the worst accuracy. The variation in accuracy varies between 0.58 and 0.76, which I think is big difference in accruracy.

KNN supports non-linear solutions, but Linear Regression on the other gives linear solutions. That can be the reason that we got so low accuracy when testing Linear Regression.

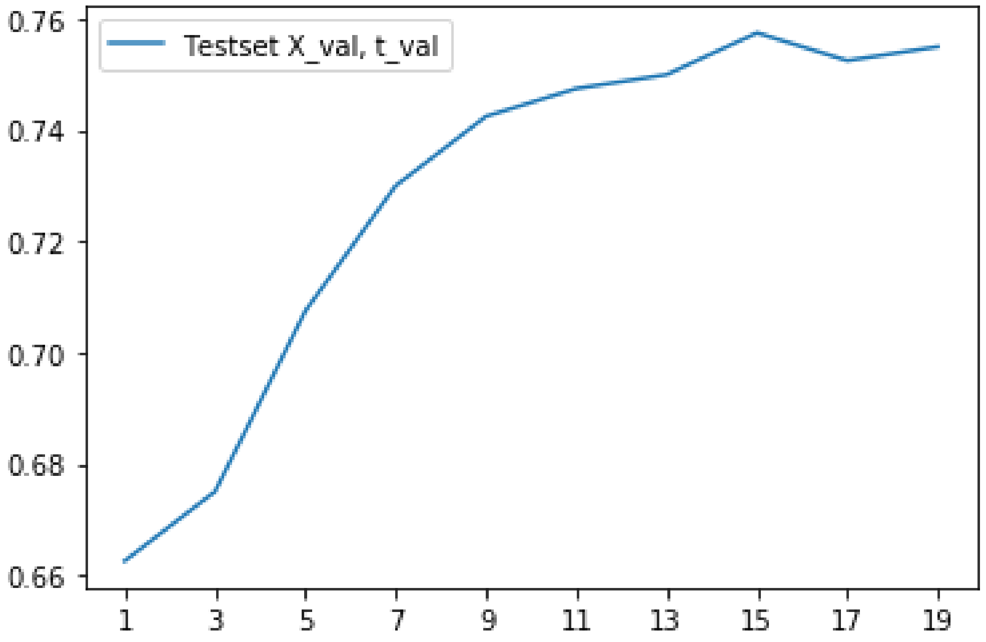
1.3 Multi-class classifiers

We turn to the task of classifying when there are more than two classes, and the task is to ascribe one class to each input. We will now use the set (X, t).

1.3.1 *k*NN

One of the classifiers can handle multiple classes without modifications: the *k*-nearest neighbors classifier. Train it on (X_train, t_train), test it on (X_val, t_val) for various values of *k*. Choose the one you find best and report the accuracy.

[12]:
```python
xMulti = range(1, 20, 2)
accuraciesMulti = []
for l in xMulti:
    cls = PykNNClassifier(k=l)
    cls.fit(X_train.tolist(), t_train.tolist())
    accuraciesMulti.append(cls.accuracy(X_val.tolist(), t_val.tolist()))



plt.plot(xMulti, accuraciesMulti, label="Testset X_val, t_val")
plt.xticks(xMulti)
plt.legend()
plt.show()



idxMulti = max(enumerate(accuraciesMulti), key=itemgetter(1))[0]
print("KNN accuracy is", max(accuraciesMulti), "with k =", idxMulti)
```



```
KNN accuracy is 0.7575 with k = 7
```

1.3.2 Logistic regression "one-vs-rest"

We saw in the lecture how a logistic regression classifier can be turned into a multi-class classifier using the one-vs-rest approach. We train one classifier for each class and assign the class which ascribes the highest probability.

Extend the logisitc regression classifier to a multi-class classifier. To do this, you must modify the target values from scalars to arrays. Train the resulting classifier on (X_train, t_train), test it on (X_val, t_val), and report the accuracy.

[62]:
```
def logistic(x):
    return 1/(1+np.exp(-x))



list1 = []
```

```
A = NumpyLogReg()
binary_t_val_A = (t_val == 0).astype('int') #Isolating the class in val
binary_t_train_A = (t_train == 0).astype('int') #Isolating the class in train
A.fit(X_train, binary_t_train_A)
list.append(A.accuracy(X_train, binary_t_train_A))


list2 = []
B = NumpyLogReg()
binary_t_val_B = (t_val == 1).astype('int') #Isolating the class in val
binary_t_train_B = (t_train == 1).astype('int') #Isolating the class in train
B.fit(X_train, binary_t_train_B)
list2.append(B.accuracy(X_train, binary_t_train_B))


list3 = []
C = NumpyLogReg()
binary_t_val_C = (t_val == 2).astype('int') #Isolating the class in val
binary_t_train_C = (t_train == 2).astype('int') #Isolating the class in train
C.fit(X_train, binary_t_train_C)
list3.append(C.accuracy(X_train, binary_t_train_C))

print(f"Logistic regression accuracy: t=0: {max(list)}, t=1: {max(list2)}, with␣
↪t=2: {max(list3)}")
```

```
Logistic regression accuracy: t=0: 0.90875, t=1: 0.6175, with t=2: 0.8225
```

Discuss the results in a couple of sentences, addressing questions like

- How do the two classfiers compare?
- How do the results on the three-class classification task compare to the results on the binary task?
- What do you think are the reasons for the differences?

After running both codes I see that I get better accuracy with logistic regresion. I think it has something to do with the third class, where it is harder for KNN to achive high accuracy.

1.4 Adding non-linear features

We are returning to the binary classifier and the set (X, t2). As we see, some of the classifiers are not doing too well on the (X, t2) set. It is easy to see from the plot that this data set is not well suited for linear classifiers. There are several possible options for trying to learn on such a set. One is to construct new features from the original features to get better discriminants. This works e.g., for the XOR-problem. The current classifiers use two features: x_1 and x_2 (and a bias term x_0). Try to add three additional features of the form ${x_1}^2$, ${x_2}^2$, $x_1 * x_2$ to the original features and see what the accuracies are now. Compare to the results for the original features in a 4x2 table.

Explain in a couple of sentences what effect the non-linear features have on the various classifiers. (By the way, some of the classifiers could probably achieve better results if we scaled the data, but we postpone scaling to part 2 of the assignment.)

I see thath KNN got the best results, and I think it is because KNN is best suited for non-linear solution (as I mentioned above). I also see that especially linear regression got a bit worse results than above, where the accuracy dropped from 0.58 to 0.5275 because we added non-linear features

[30]:
```
new_X_train = np.insert(X_train, 2, X_train[:,0] ** 2, axis=1)
new_X_train = np.insert(new_X_train, 3, X_train[:,1] ** 2, axis=1)
new_X_train = np.insert(new_X_train, 4, X_train[:,0] *  X_train[:,1], axis=1)


new_X_val = np.insert(X_val, 2, X_val[:,0] ** 2, axis=1)
new_X_val = np.insert(new_X_val, 3, X_val[:,1] ** 2, axis=1)
new_X_val = np.insert(new_X_val, 4, X_val[:,0] *  X_val[:,1], axis=1)


#linear regression
lin_cl = NumpyLinRegClass()
lin_cl.fit(new_X_train, t2_train)
new_accuracy_lin = lin_cl.accuracy(new_X_val, t2_val)
print("Linear regression accuracy:", new_accuracy_lin)


#logical regression
lr_cl = NumpyLogReg()
lr_cl.fit(new_X_train, t2_train)
print("Logical regression accuracy:", lr_cl.accuracy(new_X_val, t2_val))

#KNN
x = range(1, 20, 2)
accuracies = []
for l in x:
    cls = PykNNClassifier(k=l)
    cls.fit(new_X_train.tolist(), t2_train.tolist())
    accuracies.append(cls.accuracy(new_X_val.tolist(), t2_val.tolist()))
print("KNN:", max(accuracies))


#preseption

test2 = []
for i in range(1,20):
    cl2 = PyPerClassifier()
    cl2.fit(new_X_train.tolist(), t2_train, eta=0.1, epochs = i)
    test2.append(cl2.accuracy(new_X_val.tolist(), t2_val))
```

```


epochs_index2 = max(enumerate(test2), key=itemgetter(1))[0]
print("Simple perceptron accuracy is", max(test2), "with k =", epochs_index2)

#for some reason, I can not run perceptron-code after I ran the␣
↪simple-perceptron-code above.
#Sometimes I can run this code but not the one above. Then the accuracy is 0.
↪6475.
#I think it has something to do with values being stored.
```

```
/srv/conda/envs/notebook/lib/python3.6/site-packages/ipykernel_launcher.py:32:
RuntimeWarning: overflow encountered in matmul
/srv/conda/envs/notebook/lib/python3.6/site-packages/ipykernel_launcher.py:32:
RuntimeWarning: invalid value encountered in matmul
/srv/conda/envs/notebook/lib/python3.6/site-packages/ipykernel_launcher.py:41:
RuntimeWarning: invalid value encountered in matmul
/srv/conda/envs/notebook/lib/python3.6/site-packages/ipykernel_launcher.py:42:
RuntimeWarning: invalid value encountered in greater
```

```
Linear regression accuracy: 0.5275
Logical regression accuracy: 0.7325
KNN: 0.7625
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-1bd5549171ef> in <module>
     36 for i in range(1,20):
     37     cl2 = PyPerClassifier()
---> 38     cl2.fit(new_X_train.tolist(), t2_train, eta=0.1, epochs = i)
     39     test2.append(cl2.accuracy(new_X_val.tolist(), t2_val))
     40 

<ipython-input-11-20b8e84c5f3e> in fit(self, X_train, y_train, eta, epochs)
     15         """Train the self.weights on the training data with learning
     16         rate eta, running epochs many epochs"""
---> 17         X_train = [[1]+list(x) for x in X_train] # Put bias in position 0
     18         self.dim = dim = len(X_train[0])
     19         self.weights = weights = [0 for _ in range(dim)]

<ipython-input-11-20b8e84c5f3e> in <listcomp>(.0)
     15         """Train the self.weights on the training data with learning
     16         rate eta, running epochs many epochs"""
---> 17         X_train = [[1]+list(x) for x in X_train] # Put bias in position 0
     18         self.dim = dim = len(X_train[0])
     19         self.weights = weights = [0 for _ in range(dim)]
```

```
TypeError: 'list' object is not callable
```

Classifiers:	Accuracy (2 features)	Accuracy (5 features)
Linear Regreesion	0.58	0.52
Logical Regression	0.625	0.7325
k-nearest neighbors (kNN)	0.7525	0.7625
Simple perceptron	0.6625	0.6475

2 Part II

2.1 Multi-layer neural networks

We will implement the Multi-layer feed forward network (MLP, Marsland sec. 4.2.1). We will do so in two steps. In the first step, we will work concretely with the dataset (X, t). We will initialize the network and run a first round of training, i.e. one pass throught the algorithm at p. 78 in Marsland.

In the second step, we will turn this code into a more general classifier. We can train and test this on (X, t), but also on other datasets.

First of all, you should scale the X.

[31]:
```
X_scaler = sklearn.preprocessing.normalize(X, axis=1, norm="l1")
X_train_scaler = X_scaler[indices[:800],:]
X_val_scaler = X_scaler[indices[800:1200],:]
X_test_scaler = X_scaler[indices[1200:],:]
```

2.2 Step1: One round of training

2.2.1 Initialization

We will only use one hidden layer. The number of nodes in the hidden layer will be a hyper-parameter provided by the user; let's call it *dim_hidden*. (*dim_hidden* is called *M* by Marsland.) Initially, we will set it to 6. This is a hyper-parameter where other values may give better results, and the hyper-parameter could be tuned.

Another hyper-parameter set by the user is the learning rate. We set the initial value to 0.01, but also this may need tuning.

[32]:
```
eta = 0.01 #Learning rate
dim_hidden = 6
```

We assume that the input *X_train* (after scaling) is a matrix of dimension *P x dim_in*, where *P* is the number of training instances, and *dim_in* is the number of features in the training instances (*L* in Marsland). Hence we can read *dim_in* off from *X_train*. Similarly, we can read *dim_out* off from *t_train*. Beware that *t_train* must be given the form of *P x dim_out* at some point, cf. the "one-vs-all" exercise above.

[33]:
```
dim_in = len(X_train[0])
dim_out = len(set(t_train))

t_train_2 = np.zeros([len(X_train), dim_out])
for i in range(len(t_train)):
    if t_train[i] == 0:
        t_train_2[i] = [1,0,0]

    elif t_train[i] == 1:
        t_train_2[i] = [0,1,0]

    elif t_train[i] == 2:
        t_train_2[i] = [0,0,1]
```

We need two sets of weights: weights1 between the input and the hidden layer, and weights2, between the hidden layer and the output. Make the weight matrices and initialize them to small random numbers. Make sure that you take the bias terms into consideration and get the correct dimensions.

[34]:
```
weights1 = np.random.uniform(-1, 1, size = (dim_in, dim_hidden))  #Marshland p.
 ↪48 and p.49
weights2 = np.random.uniform(-1, 1, size = (dim_hidden, dim_out)) #Marshland p.
 ↪48 and p.49
```

2.2.2 Forwards phase

We will run the first step in the training, and start with the forward phase. Calculate the activations after the hidden layer and after the output layer. We will follow Marsland and use the logistic (sigmoid) activation function in both layers. Inspect whether the results seem reasonable with respect to format and values.

[35]:
```
def logistic(x):
    return 1 / (1 + np.exp(-x))
```

[36]:
```
hidden_activations = logistic(X_train @ weights1)
```

[37]:
```
output_activations = logistic(hidden_activations @ weights2)
```

2.2.3 Backwards phase

Calculate the delta terms at the output. We assume, like Marsland, that we use sums of squared errors. (This amounts to the same as using the mean square error).

[38]:
```
deltao = (t_train_2 - output_activations)*output_activations*(1.
↪0-output_activations)
```

Calculate the delta terms in the hidden layer.

[39]:
```
deltah = hidden_activations*(1.0-hidden_activations)*(np.dot(deltao,np.
↪transpose(weights2)))
```

Update the weights. Check that they have changed. As the weights depend on the random initialization, there is no unique correct solution at this point. But you should be able to see that the weights have been updated.

[40]:
```
updatew1 = np.zeros((np.shape(weights1)))
updatew2 =  np.zeros((np.shape(weights2)))

updatew1 = eta*(np.dot(np.transpose(X_train),deltah))
updatew2 = eta*(np.dot(np.transpose(hidden_activations),deltao))
weights1 += updatew1
weights2 += updatew2
```

2.3 Step 2: A Multi-layer neural network classifier

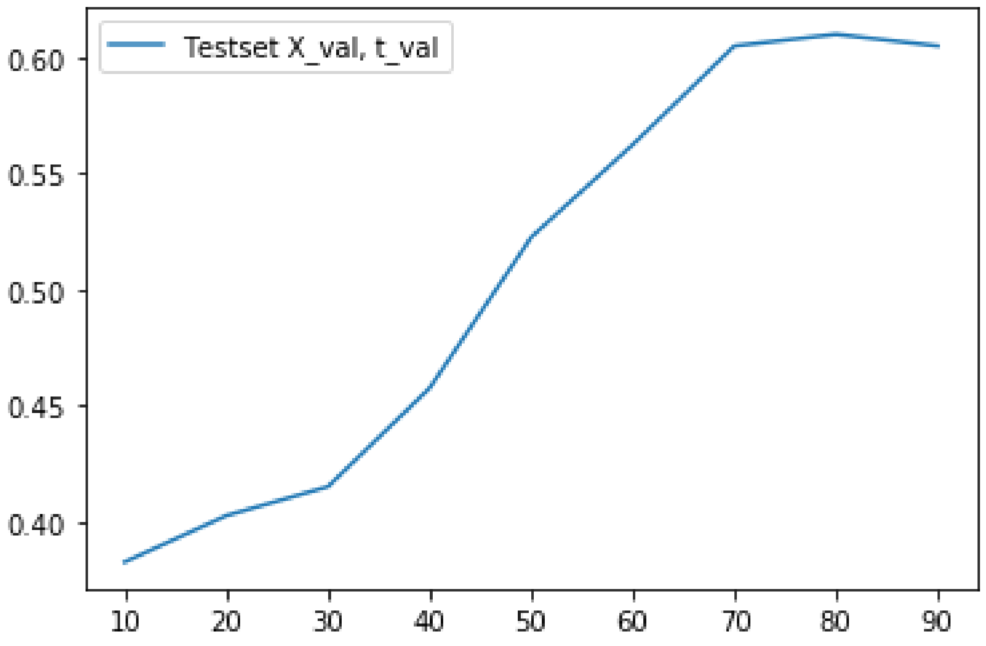
You want to train and test a classifier on (X, t). You could have put some parts of the code in the last step into a loop and run it through some iterations. But instead of copying code for every network we want to train, we will build a general Multi-layer neural network classfier as a class. This class will have some of the same structure as the classifiers we made for linear and logistic regression. The task consists mainly in copying in parts from what you did in step 1 into the template below. Remember to add the *self*- prefix where needed, and be careful in your use of variable names.

[41]:
```
class MNNClassifier():
    """A multi-layer neural network with one hidden layer"""

    def __init__(self,eta = 0.001, dim_hidden = 6):
        """Initialize the hyperparameters"""
        self.eta = eta
        self.dim_hidden = dim_hidden
        self.X_train = X_train
        self.t_train = t_train

    def fit(self, X_train, t_train, epochs = 100):
        """Initialize the weights. Train *epochs* many epochs."""
        self.dim_in = len(self.X_train[0])
        self.dim_out = len(set(self.t_train))

        self.t_train_2 = np.zeros([len(self.X_train), self.dim_out])
        for i in range(len(self.t_train)):
            if self.t_train[i] == 0:
                self.t_train_2[i] = [1,0,0]

            elif self.t_train[i] == 1:
                self.t_train_2[i] = [0,1,0]

            elif self.t_train[i] == 2:
                self.t_train_2[i] = [0,0,1]

        self.weights1 = np.random.uniform(-1, 1, size = (self.dim_in, self.
↪dim_hidden))
        self.weights2 = np.random.uniform(-1, 1, size = (self.dim_hidden, self.
↪dim_out))

        # Initilaization
        # Fill in code for initalization


        for e in range(epochs):
            # Run one epoch of forward-backward
            #Fill in the code
            self.forward(self.X_train)
            self.backward(self.t_train)





    def backward(self, t):
        self.deltao = (t_train_2 - self.output_activations)*self.
↪output_activations*(1.0-self.output_activations)
        self.deltah = self.hidden_activations*(1.0-self.hidden_activations)*(np.
↪dot(deltao,np.transpose(weights2)))

        self.updatew1 = np.zeros((np.shape(self.weights1)))
        self.updatew2 =  np.zeros((np.shape(self.weights2)))

        self.updatew1 = eta*(np.dot(np.transpose(self.X_train),self.deltah))
        self.updatew2 = eta*(np.dot(np.transpose(self.hidden_activations),self.
↪deltao))
        self.weights1 += self.updatew1
        self.weights2 += self.updatew2

    def forward(self, X):
        """Perform one forward step.
        Return a pair consisting of the outputs of the hidden_layer
        and the outputs on the final layer"""
        self.hidden_activations = logistic(self.X_train @ self.weights1)
        self.output_activations = logistic(self.hidden_activations @ self.
↪weights2)
        return self.hidden_activations, self.output_activations


    def distance_L2(a, b):
        "L2-distance using comprehension"
        s = sum((x - y) ** 2 for (x,y) in zip(a,b))
        return s ** 0.5

    def majority(a):
        counts = Counter(a)
        return counts.most_common()[0][0]


    def accuracy(self, X_test, t_test):
        """Calculate the accuracy of the classifier for the pair (X_test,␣
↪t_test)
        Return the accuracy"""
        predicted = [self.predict(a) for a in X_test]
        equal = len([(p, g) for (p,g) in zip(predicted, t_test) if p==g])
        return equal / len(t_test)

    def predict(self, a):
        X = self.X_train
        t = self.t_train
        distances = [(distance_L2(a, b), b, c) for (b, c) in zip(X, t)]
        distances.sort()
        predictors = [c for (_,_,c) in distances[0: l]]
        return majority(predictors)
```

Train the network on (X_train, t_train) (after scaling), and test on (X_val, t_val). Adjust hyperparameters or number of epochs if you are not content with the result.

[53]:
```
MNN_cl = MNNClassifier()
MNN_cl.fit(X_train_scaler, t_train, epochs=100)
accuracy = MNN_cl.accuracy(X_val_scaler, t_val)




x = range(10, 100, 10)
acc = []

for l in x:
    MNN_cl = MNNClassifier()
    MNN_cl.fit(X_train_scaler, t_train, epochs=l)
    acc.append(MNN_cl.accuracy(X_val_scaler, t_val))


print(acc)
maximum = max(acc)

epochs_index3 = max(enumerate(acc), key=itemgetter(1))[0]
print("The max accuracy is", maximum, "with k =", epochs_index3)
plt.plot(x, acc, label="Testset X_val, t_val")
plt.xticks(x)
plt.legend()
plt.show()
```

```
[0.3825, 0.4025, 0.415, 0.4575, 0.5225, 0.5625, 0.605, 0.61, 0.605]
The max accuracy is 0.61 with k = 7
```



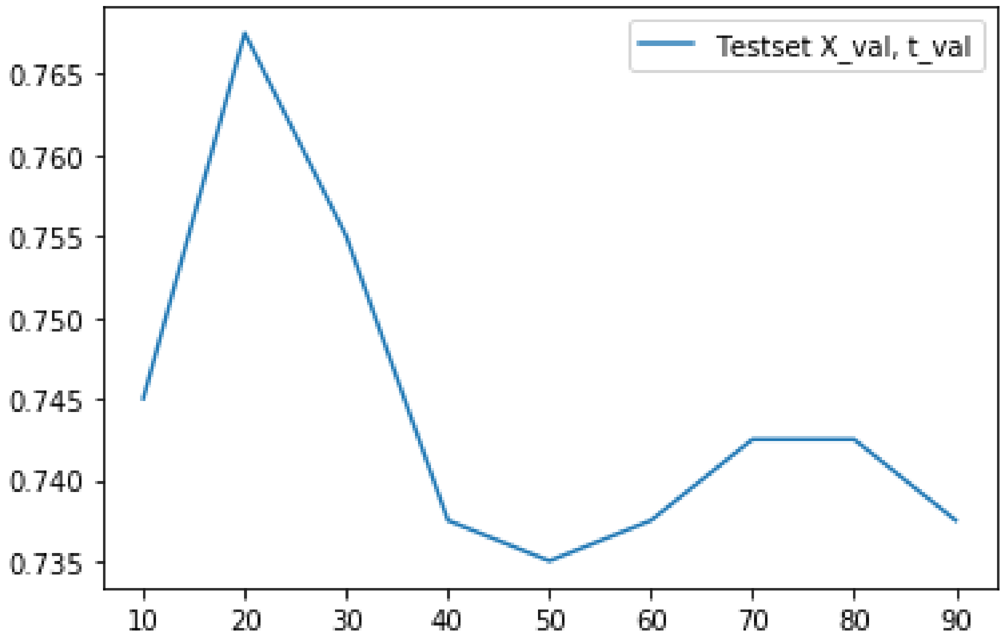
2.4 Make a neural network classifier for (X,t)

Let us see whether a multilayer neural network can learn a non-linear classifier. Train it on (X_train, t_train) and test it on (X_val, t_val). Tune the hyper-parameters for the best result.

```
[54]: MNN_cl = MNNClassifier()
      MNN_cl.fit(X_train_scaler, t_train, epochs=100)
      accuracy = MNN_cl.accuracy(X_val_scaler, t_val)
```

```python
x = range(10, 100, 10)
acc = []

for l in x:
    MNN_cl = MNNClassifier()
    MNN_cl.fit(X_train, t_train, epochs=l)
    acc.append(MNN_cl.accuracy(X_val, t_val))


print(acc)
maximum = max(acc)

epochs_index3 = max(enumerate(acc), key=itemgetter(1))[0]
print("The max accuracy is", maximum, "with k =", epochs_index3)
plt.plot(x, acc, label="Testset X_val, t_val")
plt.xticks(x)
plt.legend()
plt.show()
```

```
[0.745, 0.7675, 0.755, 0.7375, 0.735, 0.7375, 0.7425, 0.7425, 0.7375]
The max accuracy is 0.7675 with k = 1
```



3 Part III: Final testing

Take the best classifiers that you found for the training sets (X, t) and (X, t2) and test them on (X_test, t_test) and (X_test, t2_test), respectively. Compute accuracy, the confusion matrix, precision and recall. Answer in 2-3 sentences: How do the accuracies compare to the results on the validation sets?

[50]:
```python
#Confusion matrix
def cf_matrix(predicted, gold):
    table = np.zeros((2,2))
    for p,g in zip(predicted, gold):
        table[int(p), g] += 1
    print(28*" "+"gold")
    print("{:20}|{:>9}|{:>9}|".format(" ","neg", "pos"))
    print(10*" "+30*"-")
    print("{:10}{:10}|{:9}|{:9}|".format(" ","neg",table[0,0], table[0,1]))
    print("predicted "+30*"-")
    print("{:10}{:10}|{:9}|{:9}|".format(" ","pos",table[1,0], table[1,1]))
    print(10*" "+30*"-")
    return table
```

[51]:
```python
#confusion matrix
cls = PykNNClassifier(k=7)
cls.fit(X_train.tolist(), t2_train.tolist())
predicted = [cls.predict(x) for x in X_test]
table = cf_matrix(predicted, t2_test)
```

```
                            gold
                    |      neg|      pos|
          ------------------------------
          neg       |    142.0|     35.0|
predicted ------------------------------
          pos       |     63.0|    160.0|
          ------------------------------
```

[52]:
```python
#KNN with t2_test
#I found the equation to havo to find accuracy, precision and recall on:
#https://scikit-learn.org/stable/auto_examples/model_selection/
 ↪plot_precision_recall.html

print("KNN (t2_train):")

#accuracy
accuracy = (table[0,0] + table[1,1]) / (table[0,0] + table[0,1] + table[1,0] +␣
 ↪table[1,1])
print(f"accuracy: {accuracy:0.3f}")
```

```
#presision
presicion = table[1,1] / (table[1,1] +  table[0,1])
print(f"presicion: {presicion:0.3f}")

#recall
recall = table[1,1] / (table[1,1] + table[1,0])
print(f"recall: {recall:0.3f}")
```

```
KNN (t2_train):
accuracy: 0.755
presicion: 0.821
recall: 0.717
```

[55]:
```
#Confusion matrix
def cf_matrix2(predicted, gold):
    table = np.zeros((3,3))
    for p,g in zip(predicted, gold):
        table[int(p), g] += 1
    print(28*" "+"gold")
    print("{:20}|{:>9}|{:>9}|".format(" "," ", " "))
    print(10*" "+30*"-")
    print("{:10}{:10}|{:9}|{:9}|".format(" ", table[0,0], table[0,1],␣
 ↪table[0,2]))
    print("predicted "+30*"-")
    print("{:10}{:10}|{:9}|{:9}|".format(" ", table[1,0], table[1,1],␣
 ↪table[1,2]))
    print(10*" "+30*"-")
    print("{:10}{:10}|{:9}|{:9}|".format(" ", table[2,0], table[2,1],␣
 ↪table[2,2]))
    print(10*" "+30*"-")
    return table
```

[56]:
```
cls2 = PykNNClassifier(k=7)
cls2.fit(X_train.tolist(), t_train.tolist())
predicted = [cls2.predict(x) for x in X_test]
table = cf_matrix2(predicted, t_test)
```

```
                            gold
                    |         |         |
          ------------------------------
                67.0|      7.0|      0.0|
predicted ------------------------------
                25.0|    167.0|     48.0|
          ------------------------------
                 0.0|     21.0|     65.0|
          ------------------------------
```

[57]:
```python
#KNN with t_train
print("KNN:")


#accuracy
x = range(1, 20, 2)
accuracy = []
for k in x:
    cls2 = PykNNClassifier(k=7)
    cls2.fit(X_train.tolist(), t_train.tolist())
    accuracy.append(cls2.accuracy(X_test.tolist(), t_test.tolist()))


print(f"accuracy {max(accuracy):0.3f}")
print()


#presision
presision_class0 = table[0,0] / (table[0,0] + table[0,1] +  table[0,2])
presision_class1 = table[1,0] / (table[1,0] + table[1,1] +  table[1,2])
presision_class2 = table[2,0] / (table[2,0] + table[2,1] +  table[2,2])
print(f"presicion class 0: {presision_class0:0.3f}")
print(f"presicion class 1: {presision_class1:0.3f}")
print(f"presicion class 2: {presision_class2:0.3f}")
print()



#recall
print()
recall_class0 = table[0,0] / (table[0,0] + table[1,0] +  table[2,0])
recall_class1 = table[0,1] / (table[0,1] + table[1,1] +  table[2,1])
recall_class2 = table[0,2] / (table[0,2] + table[1,2] +  table[2,2])
print(f"recall class 0: {recall_class0:0.3f}")
print(f"recall class 1: {recall_class1:0.3f}")
print(f"recall class 2: {recall_class2:0.3f}")
print()
```

```
KNN:
accuracy 0.748

presicion class 0: 0.905
presicion class 1: 0.104
presicion class 2: 0.000


recall class 0: 0.728
recall class 1: 0.036
recall class 2: 0.000
```

[58]:
```
#Confusion matrix
def cf_matrix3(predicted, gold):
    table = np.zeros((3,3))
    for p,g in zip(predicted, gold):
        table[int(p), g] += 1
    print(28*" "+"gold")
    print("{:20}|{:>9}|{:>9}|".format(" "," ", " "))
    print(10*" "+30*"-")
    print("{:10}{:10}|{:9}|{:9}|".format(" ", table[0,0], table[0,1],␣
↪table[0,2]))
    print("predicted "+30*"-")
    print("{:10}{:10}|{:9}|{:9}|".format(" ", table[1,0], table[1,1],␣
↪table[1,2]))
    print(10*" "+30*"-")
    print("{:10}{:10}|{:9}|{:9}|".format(" ", table[2,0], table[2,1],␣
↪table[2,2]))
    print(10*" "+30*"-")
    return table
```

[59]:
```
#accuracy
MNN_cl = MNNClassifier()
MNN_cl.fit(X_train_scaler, t_train, epochs=7)
predicted = [MNN_cl.predict(a) for a in X_test_scaler]
table = cf_matrix3(predicted, t_test)
```

```
                            gold
                    |         |         |
          ------------------------------
                66.0|     17.0|      1.0|
predicted ------------------------------
                26.0|    178.0|    112.0|
          ------------------------------
                 0.0|      0.0|      0.0|
          ------------------------------
```

[60]:
```
print("MNN (t_train):")

#accuracy
accuracy3 = MNN_cl.accuracy(X_test_scaler, t_test)
print("accuracy", accuracy3)

#presision
presision_class0 = table[0,0] / (table[0,0] + table[0,1] +  table[0,2])
```

```
presision_class1 = table[1,0] / (table[1,0] + table[1,1] +  table[1,2])
presision_class2 = table[2,0] / (table[2,0] + table[2,1] +  table[2,2])
print(f"presicion class 0: {presision_class0:0.3f}")
print(f"presicion class 1: {presision_class1:0.3f}")
print(f"presicion class 2: {presision_class2:0.3f}")
print()



#recall
print()
recall_class0 = table[0,0] / (table[0,0] + table[1,0] +  table[2,0])
recall_class1 = table[0,1] / (table[0,1] + table[1,1] +  table[2,1])
recall_class2 = table[0,2] / (table[0,2] + table[1,2] +  table[2,2])
print(f"recall class 0: {recall_class0:0.3f}")
print(f"recall class 1: {recall_class1:0.3f}")
print(f"recall class 2: {recall_class2:0.3f}")
print()


#Something went wrong in the last line. I think it has something to do with the␣
 ↪"k"-value being stored.
#I tried to change all "k" above to "l", and I got better results, but still I␣
 ↪think a have a problem for
#"presicion class 2" and "recall class 2"
```

```
MNN (t_train):
accuracy 0.61
presicion class 0: 0.786
presicion class 1: 0.082
presicion class 2: nan


recall class 0: 0.717
recall class 1: 0.087
recall class 2: 0.009


/srv/conda/envs/notebook/lib/python3.6/site-packages/ipykernel_launcher.py:10:
RuntimeWarning: invalid value encountered in double_scalars
  # Remove the CWD from sys.path while we load stuff.
```