Linnaeus University
Introduction to Machine learning, 2DV516
Jonas Nordqvist
jonas.nordqvist@lnu.se

Assignment 3: Supervised learning algorithms

Introduction

In this Assignment you will use Python to handle a number of exercises related to various supervised learning methods: decision trees, support vectors machines and feed forward neural networks. The datasets used in the assignment can be downloaded in Moodle. The exercises are further divided into lectures.

Submission instructions

All exercises are individual. We expect you to submit at least one py- file (or Jupyter Notebook) for each exercise and your submission should include all the datasets and files we need to run your programs. Execises A, and B are just to get you started and should not be submitted. When grading your assignments we will in addition to functionality also take into account code quality. We expect well structured and efficient solutions. Finally, keep all your files in a single folder named as `username_A3` (e.g. ab123de_A3) and submit a zipped version of this folder.

Certain quantitive questions such as: *What is the MSE of the model?*, can simply be handled as a print statement in the program. More qualitative questions such as: *Motivate your choice of model.*, should be handled as a comment in the notebook or in a separate text file. (All such answers can be grouped into a single text-file.) The non-mandatory VG-exercise will require a separate report.

The following libraries are allowed to use in the assignment: `sklearn`[1], `pandas`, `matplotlib`, `scipy.stats`, `seaborn`, `numpy`, `tensorflow` (for the neural network exercise).

1 Support vector machines

Exercise A: Testing the flexibility of SVM

Support vector machines may be very flexible, and may handle highly non-linear classification. In the following example you will utilize the `sklearn` classifier `svm.SVC` to build an SVM classifier for the data in the file `bm.csv`. This is a two-class problem and the data consists of 10,000 data points.

1. Create a dataset which consists of a random sample of 5,000 datapoints in `bm.csv`. To be able to compare with the subsequent results you can use the following code.

```
np.random.seed(7)
r = np.random.permutation(len(y))
X, y = X[r, :], y[r]
X_s, y_s = X[:n_s, :], y[:n_s]
```

Listing 1: Settings for the Batman dataset

2. Use `sklearn` to create and train a support vector machine using a Gaussian kernel and compute its training error ($\gamma = .5$ and $C = 20$ should yield a training error of .0102, however note that these hyperparams are not optimized and the results may be improved).

3. Plot the decision boundary, the data and the support vectors in two plots, *c.f.* Figure 1. The indices of support vectors are obtained by `clf.support_`, where `clf` is your trained model.

[1]Note that in some assignments `sklearn` is not allowed.

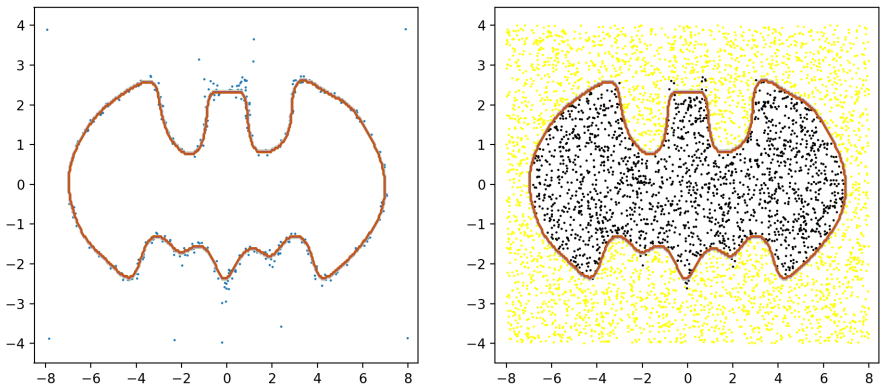

Figure 1: Support vector machine fitted on the BM dataset. To the left are the support vectors plotted together with the decision boundary and the to the right the data with the decision boundary.

Exercise 1: Various kernels

Much of the flexibility in SVM lies in the *kernel*. In this exercise you will solve the same problem using three (or four) different kernels: linear, RBF, polynomial and a custom kernel (VG-exercise). The latter is of course not implemented in `sklearn`, and you will have to make your own implementation. These should later be compared in terms of predictive performance.

Use the dataset `mnistsub.csv`, which is a small, (and difficult) 2d dummy version of MNIST consisting only of examples from the digits `1, 3, 5` and `9`. Thus, each 2d coordinate represent a digit among the four classes. There are a total of 796 examples in the dataset. You should set aside a part of these for a validation set used in the hyperparameter search discussed below. Your task is to construct the three (or four) classifiers earlier mentioned. For each classifier you should do the following

1. Tune the necessary hyperparameters by for instance grid search. In this exercise we are concerned with the hyperparameters given in Table 1. Every hyperparameter should be tested for at least 3 values but you are free to add more testings.[2]

Kernel	Hyperparameters to tune
Linear	`C`
rbf	`C, gamma`
poly	`C, d, gamma` (optional)
(ANOVA)	`C, sigma, d`

Table 1: Hyperparameters to tune in Exercise 1. Note that Anova is only necessary for students wanting a higher grade (*c.f.* Exercise 1.1)

2. Produce a plot of the decision boundary for the best models together with the data.

Note: If *not* using Jupyter notebooks all plots may ideally also be saved in a single pdf (preferred) or as images (wisely named) for the instructors as the reproduction of these plots may take some time to generate by code.

Exercise 1.1 (VG-exercise): Implementing your own kernel

The VG-part of this exercise is to implement a kernel yourselves. In this exercise it will be executed by precomputing the kernel by means of a so-called *Gram matrix* explained further down in this text.

[2] On my Macbook the longest training time was for the ANOVA kernel and that was done in ~ 6 s. Hence, a grid search over 3^3 different parameter values would amount to a total of less than 3 minutes. The others are done in negligible time, so it is easy to test for many values

The *ANOVA kernel* function can be computed as follows. Let x, x' be two ℓ-dimensional vectors, and denote by x_i the ith component of x. Then we let

$$k(x, x') := \left(\sum_{j=1}^{\ell} \exp\left(-\sigma \left(x_j - x'_j \right)^2 \right) \right)^d,$$

where $\sigma \in \mathbb{R}$ and $d \in \mathbb{N}$ are two hyperparameters to be tuned. In `sklearn` this (or basically any) kernel can be implemented as follows. Given two matrices X and Y of dimension $n \times p$ and $m \times p$. We compute the so-called Gram matrix G of X and Y, given by

$$G_{ij} := k(x^{(i)}, y^{(j)}),$$

where $x^{(i)}$ and $y^{(j)}$ are the ith and jth row of X and Y respectively. A correct implementation would thus yield a matrix G of the size $n \times m$. To train a classifier you should instead of passing X in the fitting, pass the gram matrix of X and X and include the argument `kernel='precomputed'` as an argument when constructing your classifier. When predicting you should instead pass the gram matrix of X_p and X, where X_p is the set of examples to be predicted.

Bonus credit! Prove that k is a kernel. You may use any result without proof from the lecture.

Exercise 2: One versus all MNIST

Support vector machines using rbf-kernels perform very well on the MNIST dataset. By tuning your parameters you should be able to get over 95% test accuracy. So, the first part of this exercise is to find `C` and `gamma` to obtain that kind of scores. You may use a smaller part of MNIST for training and still obtain good scores. Recall that the hyperparameters have to be found without laying your hands on the test set, *i.e.* use either cross-validation, a validation set or some other technique to distinguish between different models. Report in your code as comments, or in a separate document, the grid (or whatever technique for hyperparameter search your are using) which was searched and the resulting best hyperparameters.

The second part of this exercise is to compare the built-in binarization scheme used for the `SVC` class, namely one-vs-one, against the one-vs-all scheme, which was discussed in Lecture 5. You should implement your own version of one-vs-all SVM and compare your results against the built in version. To make the comparison simple you should keep the same hyperparameters which you found in the first part of this exercise. Which was the best classifier? If studying the confusion matrix was there any apparent difference between the two methods in terms of misclassifications? Include your findings either as comments in your code, in your Jupyter notebook or as a separate text document.

The dataset is very common and easily obtainable online. Google is your friend here!

2 Ensemble methods and Decision Trees

Exercise 3: (VG-exercise): An ensemble effect

The intent of this exercise is not to test some new methods or models, but instead to study the effects of using ensembles of *weak learners*[3]. In this exercise we will have a different approach and study an artificial problem. We've learned in class that several independent weak learners may together form a strong learner. The weak learners in this assignment will be constructed artificially and thus are not actual learners. Since the assignment is done artificially we've provided a generated dataset `gen.csv`. This dataset contains $n = 1600$ samples, and it consists of only a response y, which is a binary response. The dataset is evenly divided among zeros and ones.

We will construct the "weak learners" w in the following way. Let w be a random binary vector of length n. If we were to consider w as a model for predicting the output y it would have roughly 50% correct predictions – just what we would expect of a monkey! The idea is now to force some correct predictions on w to give a slight edge in its ability to predict. On its own it is not sufficient, but together with other "weak learners" it would be quite good at the job.

Let p be a real number in the unit interval $[0, 1]$, and generate $q := \lfloor np \rfloor$ random numbers without replacement[4] among the integers $\{1, 2, \ldots, n\}$. We'll denote these integers by $\alpha_1, \alpha_2, \ldots, \alpha_q$. Consider these

[3] In this context this means models with relatively low prediction accuracy.

[4] Without replacement means no number may be chosen more than once.

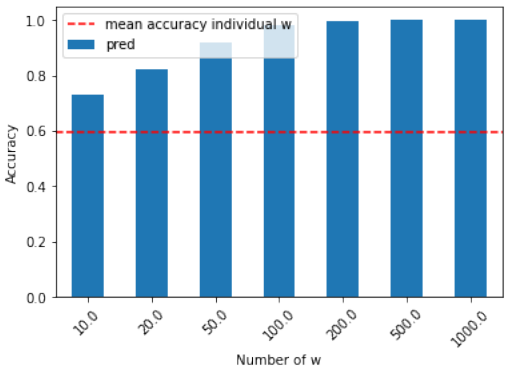

Figure 2: Accuracy in prediction over various number of w with a fixed $p = 0.2$

q integers as indices of w and put $w[\alpha_i] = y[\alpha_i]$, for all $1 \leq i \leq q$. Doing so means that w now contains roughly $(n+q)/2$ correct "predictions" of y. In terms of percentage this figure is $0.5+p/2$. An implementation of w could be as given in the code below.

```
from numpy.random import default_rng

rng = np.random.default_rng()
w = rng.choice(2, size=n, replace=True) # generates a binary vector of length n
q = n*p
alpha = rng.choice(n, size = q, replace=False) # creates the set of indices
w[alpha] = y[alpha] # gives w the q correct predictions
```

Listing 2: Constructing artificial weak learner

By considering several "learners" w, we may construct an ensemble of such: $w_1, w_2, \ldots, w_m$ for some integer $m \geq 1$. The prediction procedure for the ensemble is by majority vote.[5]

Write code to generate these "weak learners", and answer the following questions. Note that due to the random element in these exercises the answers won't be exactly the same for everyone.

a) Generate an ensemble of 100 w with $p = 0.1$. What is the accuracy of the ensemble in predicting y?

b) If $p = 0.05$, how many w are needed in the ensemble in order to surpass an accuracy of 0.9?

c) If you have five w, what seems to be a sufficient value for p in order to have a prediction accuracy above 0.95?

d) In Figure 2 you can see a plot of the results of ensebmles of different sizes when $p = 0.2$. Construct a similar plot for the case $p = 0.3$, and the sizes `[3,5,10,20,50]` of the ensembles.

Comment: In practice the various models are *not* independent as they are here. For instance, hard examples to classify (if it is a classification task) are typically hard for most models. Although, there are two main strengths of ensembles – variance reduction and potentially increased accuracy.

Exercise 4: Ensemble of Batman Trees

In this exercise, we'll revisit the Batman dataset from Exercise A. Start by splitting the data into a training and test set of 5000 examples each at random.

We will work with decision trees and the sklearn implementation of those.[6] In this assignment we will create our own simplified version of a random forest. We will construct 100 decision trees. Train them on different subsets for each classifier, and then combine their knowledge in a majority vote to use as a classifier.

The subsets should be constructed using the bootstrap technique, which in short means that we should create new training sets for each individual decision tree in the following manner: sample 5000 training samples with replacement (that means any sample may be chosen more than once). This means that for

[5] This is done using the statistical mode of the individual predictions.

[6] The implementation of decision trees is the only place you should use sklearn for this exericse.

Figure 3: Decision boundaries for 99 decision trees on the Batman-dataset (the 100th model is removed for visual purposes) and also including the ensemble voting model (down-right corner)

instance X_1 is a subset of 5000 training samples, but most likely there are several which are duplicates. Thus, you will create $X_1, X_2, \ldots, X_{100}$ and the corresponding $y_1, y_2, \ldots y_{100}$. Then each of the 100 decision trees should be trained on the corresponding dataset. Some pointers on a suggestion on how to create these datasets is given in the code below.

```
# Assuming that X is the data matrix in the exercise, the following representation could be
    used for the 100 bootstrapped training data matrices for each decision tree

from numpy.random import default_rng
rng = np.random.default_rng()

n = 5000
r = np.zeros([n,100], dtype=int)
XX = np.zeros([n,2,100])

for i in range(100):
    r[:,i] = rng.choice(n, size = n, replace=True)
    XX[:,:,i] = X[r[:,i], :]
```

Listing 3: Ideas for generating bootstrap samples for the different models

For the sake of the assignment there is no need to work further with the different models in searching for hyperparameters. The default values are good enough!

The output of this exercise should be the following:

a) The estimate of the generalization error using the test set of the ensemble of 100 decision trees.

b) The average estimated generalization error of the individual decision trees.

c) A plot of the decision boundaries of all the models , and including the ensemble model (*c.f* Figure 3).

d) Finally, a short comment on the results. Was it expected? Surprising? Do you see any benefits, downsides with this method?

3 Neural networks

Exercise B: Yes or no but not both

Train a multilayer perceptron (MLP) to learn the XOR function, and print its parameters. Confirm for yourself that the network learned the function correctly by studying its weights.

Exercise 5: ML in Fashion

Image classification is classical task in machine learning. MNIST serves as a good starting point for testing out algorithms and learning. Some criticism that has been pointed out is that it is fairly easy to get good (well above 95%) accuracy. In this exercise you will perform image classification on a similar dataset known as Fashion MNIST. The dataset consists of 10 categories of different clothes, and your overall objective is to find a feed-forward neural network which can distinguish images on the different sets of clothes. The dataset contains 60,000 images for training and 10,000 for testing just as the ordinary MNIST. The images are 28×28 pixels.

The following elements should be included and printed/plotted in your code.

1. Plot 16 random samples from the training set with the corresponding labels.
2. Train a multilayer perceptron to achieve as good accuracy as you can. There are numerous hyperparameters that we discussed in class which you can tweak, for instance: learning rate, number of and size of hidden layers, activation function and regularization (*e.g.* Ridge (known here as L2), and early stopping). You should make a structured search for the best hyperparameters that you can find.
3. Plot the confusion matrix. Which are the easy/hard categories to classify? Are there any particular classes that often gets mixed together?