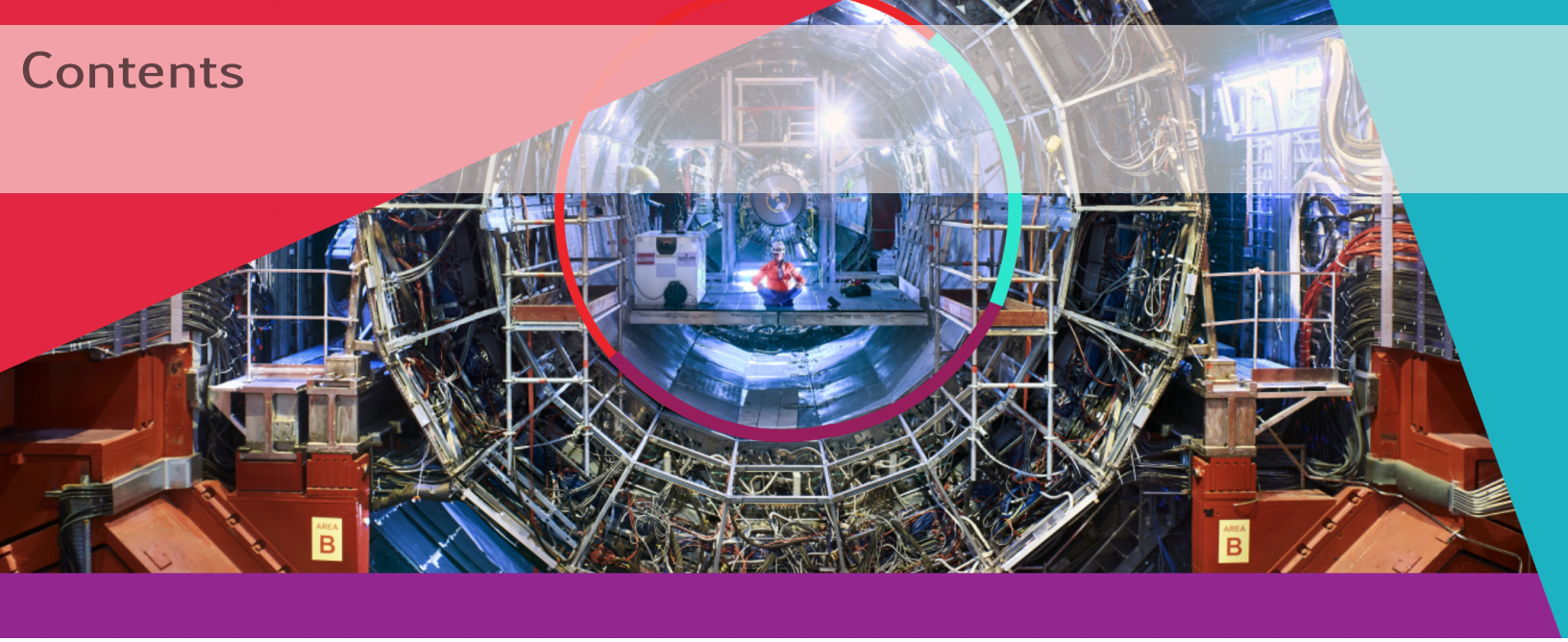
Course introduction, lecture notes and coding examples

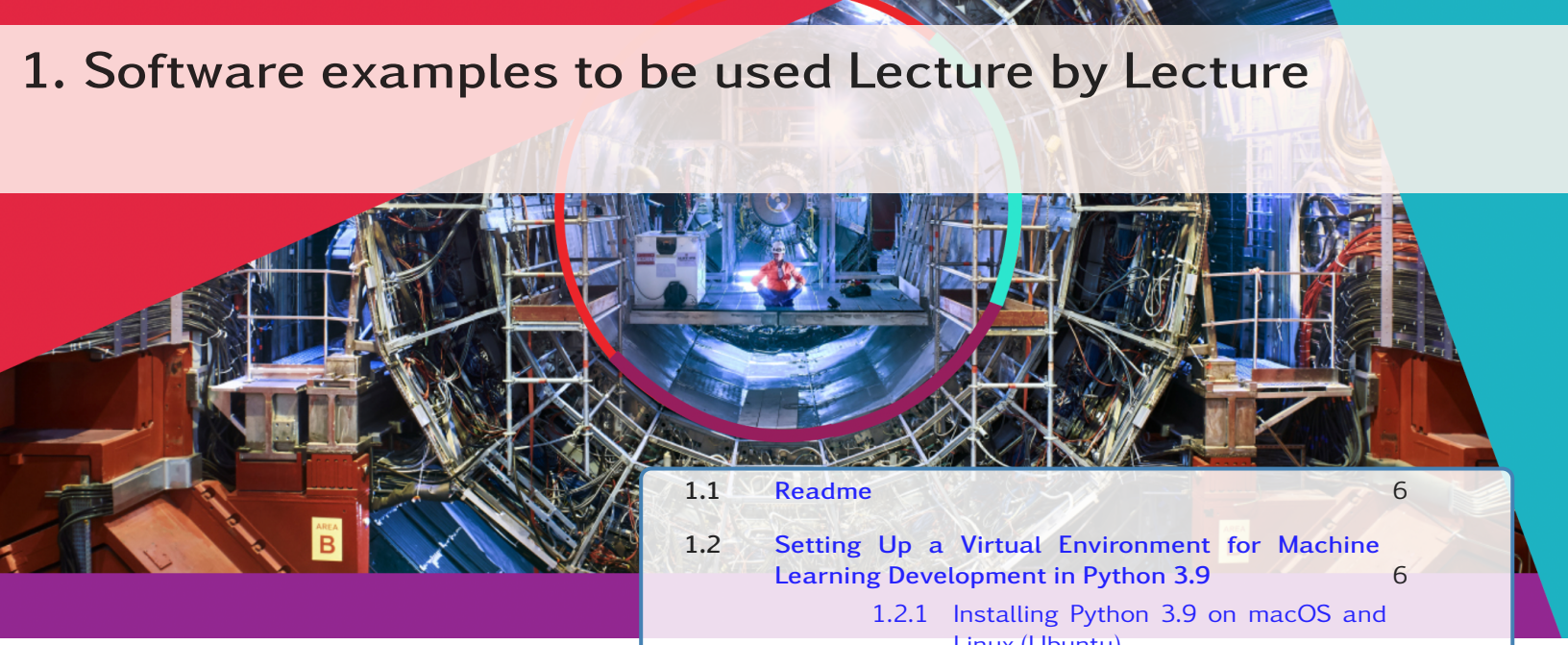
Alessandro Grelli, Marc van der Sluys, Sarah Caudill and Melissa Lopez

Computational aspects of machine learning

Contents



1. Software examples to be used Lecture by Lecture



1.1 Readme

The examples that follows are meant introducing you to machine learning in python. The examples have an increasing level of complexity, going from simple programs in lecture one to fully fledged neural networks in Lecture 9. As discussed in the **introduction to the course** chapter, I will not test you on the coding part, you are assumed to know already python. However, I consider important that you have real life examples connected to the theory lectures in order to get acquitted with the algorithms in a gradual way before starting your final project.
Please consider that the following examples will run out of the box once you add them to a Jupyter notebook. However, for each one of them you need data. You can find the corresponding data at:

$$https://github.com/agrelli/ML\_course223\_24/tree/d23a06baf21c95be850211e18c38c0d0bb50a773/examples\_python \quad (1.1)$$

Can you run them? **In case of questions, do not hesitate to contact your TAs**

1.2 Setting Up a Virtual Environment for Machine Learning Development in Python 3.9

In this guide, we will walk through the process of setting up a virtual environment in Python 3.9 specifically tailored for machine learning development. A virtual environment helps isolate your project dependencies, ensuring that different projects can have their own set of libraries without interference.

1.2.1 Installing Python 3.9 on macOS and Linux (Ubuntu)

For macOS

1. Install Python 3.9 using Homebrew:

```
/bin/bash -c "$(curl -fsSL
    https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)" brew install
    python@3.9
```

2. Verify Python installation:

```
python3.9 --version
```

1.2.2 For Linux (Ubuntu)

1. Update package list and install dependencies:

```
sudo apt update sudo apt install build-essential zlib1g-dev libncurses5-dev
    libgdbm-dev libnss3-dev libssl-dev libreadline-dev libffi-dev wget
```

2. Download and install Python 3.9:

```
cd /tmp wget https://www.python.org/ftp/python/3.9.7/Python-3.9.7.tgz tar -xf
    Python-3.9.7.tgz cd Python-3.9.7 ./configure --enable-optimizations sudo make
    altinstall
```

3. Verify Python installation:

```
python3.9 --version
```

1.2.3 Installing `virtualenv` and Setting Up a Virtual Environment

1. Install `virtualenv`:

```
python3.9 -m pip install virtualenv
```

2. Navigate to your project directory and create a virtual environment named `venv`:

```
cd /path/to/your/project python3.9 -m virtualenv venv
```

3. Activate the virtual environment:

```
source venv/bin/activate
```

1.2.4 Installing Necessary Packages for Machine Learning

Now that your virtual environment is active, install the necessary packages for machine learning:

```
python -m pip install numpy pandas matplotlib scikit-learn tensorflow jupyter seaborn
```

These packages provide a solid foundation for various aspects of machine learning development, from data manipulation and visualization to building and training models.

1.2.5 Deactivating the Virtual Environment

When you're done working in the virtual environment, deactivate it using:

```
deactivate
```

1.2.6 Conclusion

By following these steps, you have successfully set up a virtual environment in Python 3.9 on both macOS and Linux (Ubuntu), and installed essential packages for machine learning development. This setup allows you to work on machine learning projects with a clean environment, ensuring dependency management and compatibility.

1.3 Setting Up a Virtual Environment with GPU Support for Machine Learning on macOS

In this guide, we will walk through the process of setting up a virtual environment on macOS with GPU support for machine learning tasks using TensorFlow or PyTorch. This setup allows you to leverage NVIDIA GPU acceleration for faster computations in deep learning models.

1.3.1 Installing CUDA Toolkit and cuDNN

1. **Check macOS Version Compatibility:** Ensure your macOS version is compatible with the CUDA Toolkit and cuDNN version you intend to install. Refer to NVIDIA's official documentation for compatibility details.
2. **Download CUDA Toolkit:** Visit the NVIDIA CUDA Toolkit download page and download the installer for macOS.
3. **Install CUDA Toolkit:** Follow the installation instructions provided by NVIDIA. This usually involves running the downloaded installer and following on-screen prompts.
4. **Download cuDNN:** Visit the NVIDIA cuDNN download page and download the version compatible with your CUDA Toolkit version and macOS.
5. **Install cuDNN:** Extract the downloaded cuDNN archive and copy the relevant files into the CUDA Toolkit installation directory according to NVIDIA's installation instructions.

1.3.2 Setting Up the Virtual Environment

1. **Create and Activate Virtual Environment:** Follow the steps to create and activate a virtual environment using `virtualenv`:

```
python3.9 -m venv venv
source venv/bin/activate
```

2. **Install TensorFlow or PyTorch with GPU support:**
   - **TensorFlow:**

```
    pip install tensorflow
```

   - **PyTorch:** Visit the PyTorch website to get the specific installation command for PyTorch with CUDA support.

```
    pip install torch torchvision torchaudio
```

1.3.3 Verify GPU Support

- **Check TensorFlow GPU Support:** After installing TensorFlow, run the following Python code snippet within your virtual environment:

```
import tensorflow as tf
print("Num GPUs Available:",
    len(tf.config.experimental.list_physical_devices('GPU')))
```

- **Check PyTorch GPU Support:** For PyTorch, you can check GPU availability with:

```
import torch
print("CUDA Available:", torch.cuda.is_available())
```

  Additionally, print the CUDA device properties:

```
if torch.cuda.is_available():
    print("CUDA Device Name:", torch.cuda.get_device_name(0))
```

1.3.4 Deactivating the Virtual Environment

- **Deactivate the Virtual Environment:** When you're done working with your virtual environment, deactivate it as follows:

```
deactivate
```

By following these steps, you can set up a virtual environment on macOS with GPU support for TensorFlow or PyTorch. This configuration leverages the power of NVIDIA GPUs for accelerated deep learning computations, enhancing performance significantly compared to CPU-only computations. Adjustments may be necessary based on specific GPU models, macOS versions, and deep learning framework versions you are using.

1.4 Lecture1: Bayes

1.4.1 A trivial example

This is the first example of the course, and it is your introduction to coding. Please, do not be surprised by its simplicity!. In this first lecture, my only intent is to put you in front of your programming skills. What I want to say is that, if you consider this example difficult, then I would suggest you to spend some time refreshing your Phyton before the project part of the course start.

Listing 1.1: Writing Bayes theorem in python.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Let's put in code form our example 1 of this lecture (cancer patient). Note this is
    really really simple so consider it a warmup that allow you to self evaluate your
    python skills.

# ------------------------------
# calculate bayes theorem factors
#-------------------------------

def bayes_theorem(p_a, p_b_given_a, p_b_given_not_a):
    # calculate P(not A)
    not_a = 1 - p_a
    # calculate P(B) - evidence
    p_b = p_b_given_a * p_a + p_b_given_not_a * not_a
    # calculate P(A|B) with bayes theorem
    p_a_given_b = (p_b_given_a * p_a) / p_b
    return p_a_given_b

# ------------------------------
# Initial values
# ------------------------------

# P(A) - general person with cancer - prior
p_a = 0.0002
# P(B|A) - sensitivity f the test
p_b_given_a = 0.85
# P(B|not A) - fail rate of the test
p_b_given_not_a = 0.05
# calculate P(A|B)
result = bayes_theorem(p_a, p_b_given_a, p_b_given_not_a)
# print on screen the result
print('P(A|B) = %.3f%%' % (result * 100))

```

```
#Define style for plotting
plt.style.use('seaborn-white')

# ------------------------------------------------
# Let's try for fun all the possible priors and see
# what posterior distribution we get
#-------------------------------------------------

#Initializing vectors for prior and posterior
p_a1 = [] # prior
b = [] # posterior

# generating random priors between 0 and 1 (10000 times) and filling the vector
p_a1 = np.random.uniform(0,1,10000)

for x in range(0,10000):
    print(p_a1[x])
    b.append(bayes_theorem(p_a1[x], p_b_given_a, p_b_given_not_a))

#make the plotting
# not very elegant but effective:
# define a figure and then divide in 4 your space
# afterwards plot one histogram per box
fig = plt.figure()
#slot 1
plt.subplot(2, 2, 1)
plt.hist2d(p_a1, b, bins =[100, 50], cmap=plt.cm.jet)
plt.title('Bayes')
plt.xlabel('prior')
plt.ylabel('posterior')
# slot 2
plt.subplot(2, 2, 2)
plt.hist(b, bins=50)
plt.title('Bayes')
plt.xlabel('posterior')
plt.ylabel('Entries')
# slot 3
plt.subplot(2, 2, 3)
plt.hist(p_a1, bins=50)
plt.xlabel('prior')
plt.ylabel('Entries')
# show
plt.show()
```

The code runs with just *Phyton bayes.py* once you properly installed NumPy extension, and you can expect as a result:

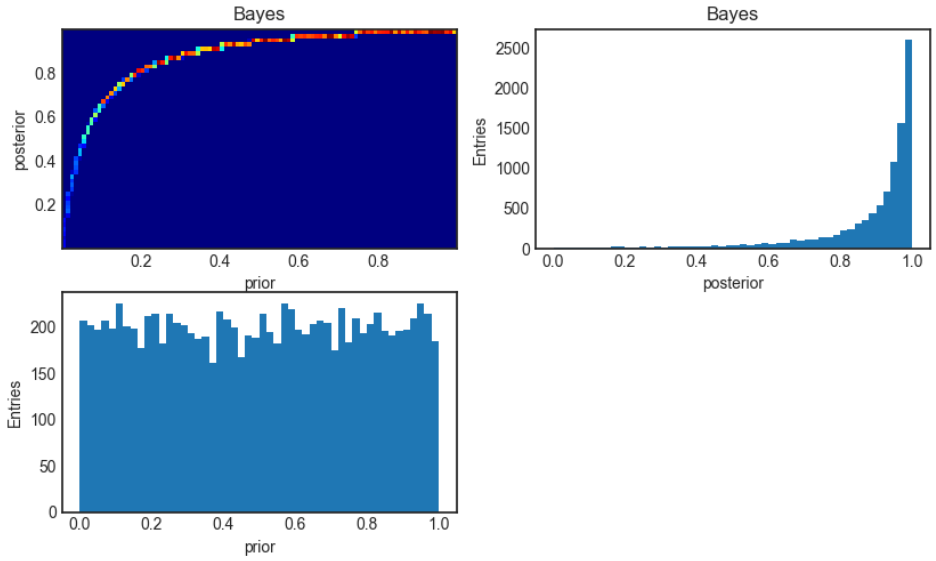


Figure 1.1: Posterior vs prior distributions.

1.5 Lecture 3: Mixture of Gaussian, Clustering

1.6 Clustering and mixture of gaussian

In this example we want to cluster the data in 3 classes by using k-Means and then we repeat the process with a mixture of gaussians. The software example is fully working on a Jupyter notebook If you have questions about this notebook contact Melissa Lopez at m.lopez@uu.nl

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.mixture import GaussianMixture

color = ['cornflowerblue', 'darkorange', 'mediumseagreen']
```

Here we load the data and resample it to have a $(samples, features)$ structure.

```
# We load the data
data = np.load('./data/1000_glitches.npy')

# (samples, 1, features) --> (samples, features)
data = data.reshape(data.shape[0], data.shape[2])
print(data.shape)
```

```
(1000, 938)
```

We want to cluster the data in 3 classes. Let's use k-Means.

```
# We fix the random state to reproduce the results.
random_state = 170
clusters = 3
# This is our clustering algorithm.
kmeans = KMeans(n_clusters=clusters, random_state=random_state)

# And we want to fit and predict all our data.
kmeans_pred = kmeans.fit_predict(data)
```

How can we visualize our data? We have 938 $features$, so we might want to reduce our dimensionality with PCA.

```
# We decompose our data in 2 components for better visualization
pca = PCA(n_components=2)

# And we want to fit and decompose all our data.
pca_data = pca.fit_transform(data)
print(pca_data.shape)

# 1st PCA component
pca1 = pca_data[:, 0]

# 2nd PCA component
pca2 = pca_data[:, 1]
```

```
(1000, 2)
```

We will store PCA components and k-Means labels in a pandas DataFrame

```
results = pd.DataFrame({'PCA 1': pca1, 'PCA 2': pca2, 'Labels': kmeans_pred})
display(results)
```

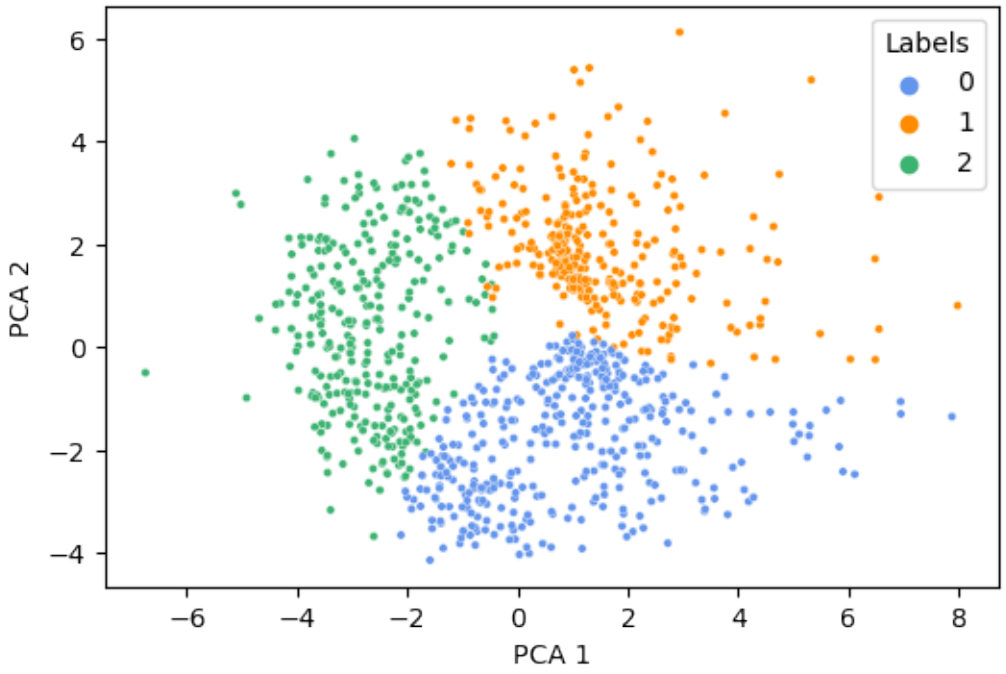


Figure 1.2: png

	PCA 1	PCA 2	Labels
0	1.629584	-0.507421	0
1	-2.018157	-0.721535	2
2	-0.738842	-1.951903	0
3	1.826200	4.669151	1
4	0.868556	-0.608222	0
...	...	...	...
995	1.160501	0.973785	1
996	1.243735	0.971269	1
997	0.321739	-1.591889	0
998	-2.422662	2.523152	2
999	0.623661	4.481843	1

1000 rows × 3 columns

Let's visualize our data.

```
fig = plt.figure(dpi=100)
sns.scatterplot(data = results, x='PCA 1', y='PCA 2', hue='Labels',
                palette=color, s=10)
```

```
<AxesSubplot:xlabel='PCA 1', ylabel='PCA 2'>
```

Let's repeat the process with Gaussian Mixtures.

```
random_state = 170
clusters = 3

gm = GaussianMixture(n_components=clusters, random_state=random_state).fit(data)
```

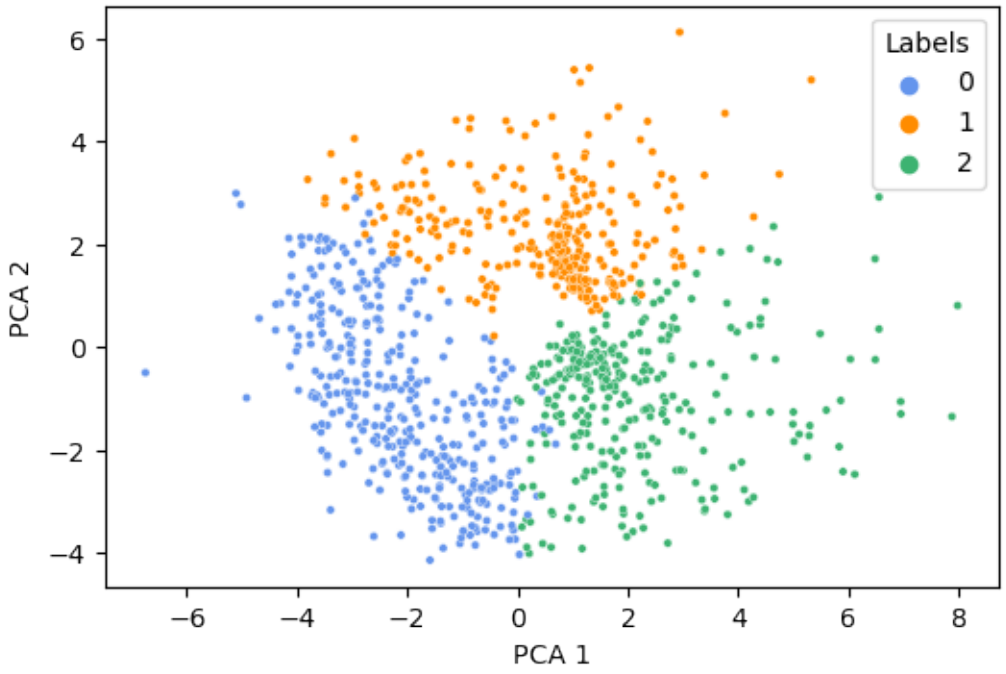


Figure 1.3: png

```
gm_pred = gm.predict(data)

results_gm = pd.DataFrame({'PCA 1': pca1, 'PCA 2': pca2, 'Labels': gm_pred})

fig = plt.figure(dpi=100)
sns.scatterplot(data = results_gm, x='PCA 1', y='PCA 2', hue='Labels',
                palette=color, s=10)
```

```
<AxesSubplot:xlabel='PCA 1', ylabel='PCA 2'>
```

We can see some differences: some samples that were classified in a certain class with K-means, now are in separated classes with Gaussian Mixtures. Can you plot some examples where this is the case?

Can you plot some of the samples in the outskirts of the distributions? Are they similar or different to the ones in the bulk?

1.7 Lecture 4: Linear regression and maximum likelihood

1.8 Linear regression

If you have questions about this notebook contact Melissa Lopez at m.lopez@uu.nl

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
```

```
!ls data
```

```
A_vs_A.npy A_vs_B.npy
```

Let's load the data that are stored in numpy arrays. These can be understood as correlation matrices between population A and B, and A against itself. Note that population A has 1000 samples while population B has 3265.

```
# Note that we drop 2 matrices for simplicity.
#These matrices that we drop are ids of the samples, but we do not need them.

A_vs_B = np.load('./data/A_vs_B.npy')[:, :, :3]
A_vs_A = np.load('./data/A_vs_A.npy')[:, :, :3]

print(A_vs_B.shape, A_vs_A.shape)
```

```
(1000, 3265, 3) (1000, 1000, 3)
```

From their shapes we can observe that they have 3 dimensions. However, we want to get the average of A over B for A_vs_B, and the average of A over A for A_vs_A. Their shapes transform as:

A_vs_B $\longrightarrow (A\ samples, B\ samples, metrics) \longrightarrow (A\ samples, metrics)$

A_vs_A $\longrightarrow (A\ samples, A\ samples, metrics) \longrightarrow (A\ samples, metrics)$

What are these *metrics*? They are similarity distance. For simplicity let's say $metrics = ${$dist1, dist2, dist3$}

We want to check if A_vs_A and A_vs_B are linearly correlated. Let's do that for $dist1$

```
# We average along dim 1. In that way x and y should be the same shape.

x = np.mean(A_vs_A, axis=1)[:, 0]
y = np.mean(A_vs_B, axis=1)[:, 0]

print(x.shape, y.shape)
```

```
(1000,) (1000,)
```

We want to fit x and y variales to a line, so we use least squares estimate. We use Scipy lineare regression (see https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.linregress.html).

```
m, n, R, _, _ = stats.linregress(x, y)
print(m, n, R)
```

```
0.8988249130413917 0.0064757515395796016 0.9925763297325655
```

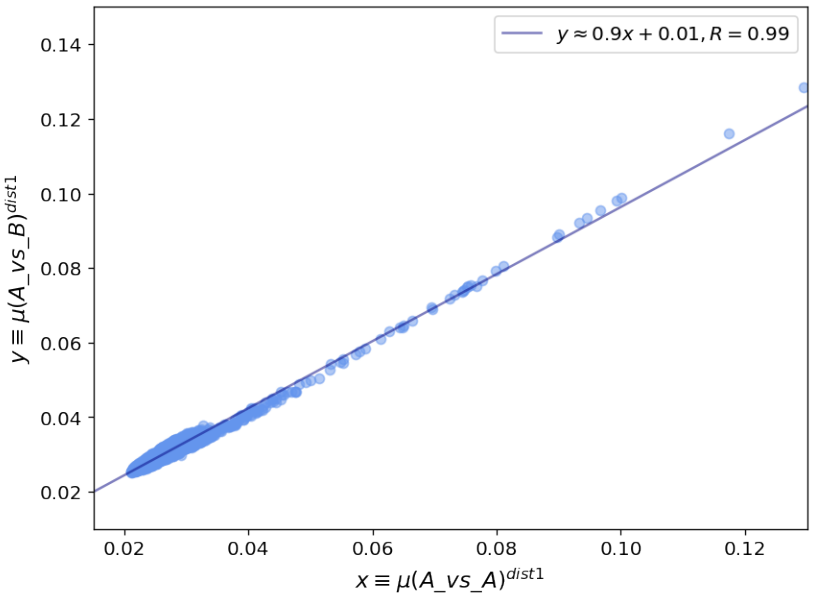


Figure 1.4: png

We are only interested in the slope m, intercept n and Pearson coefficient R. Pearson coefficient is almost one meaning that x and y are strongly linearly correlated. Let's plot the results.

```
# We define X to plot the line.
X = np.array([0, 0.15])

fig = plt.figure(figsize=(8,6), dpi=120)
plt.scatter(x, y, alpha=0.5, c='cornflowerblue')
plt.plot(X,  X* m + n, c='navy', alpha=0.5,
        label=r'$y \approx '+str(np.round(m,2))+' x +'+str(np.round(n,2))+', R
            =$'+str(np.round(R,2)))
plt.legend(fontsize=12)
plt.xticks(fontsize=12), plt.yticks(fontsize=12)
plt.xlabel('$x \equiv \mu(A\_vs\_A)^{dist1}$', fontsize=14)
plt.ylabel('$y \equiv \mu(A\_vs\_B)^{dist1}$', fontsize=14)
plt.xlim(0.015, 0.13)
plt.ylim(0.01, 0.15)
```

```
(0.01, 0.15)
```

Can you do the same for $dist2$ and $dist3$? How are these magnitudes similar or different to $dist1$? Can you relate them some how?

1.9 Lecture 5 and 6: A simple classifier

Lecture 5 of Computational Aspectes of Machine Learning: course 2021-2022

If you have questions about this notebook contact Melissa Lopez at m.lopez@uu.nl

```
import numpy as np
import pandas as pd
from sklearn import preprocessing as pp
import matplotlib.pyplot as plt
from sklearn.preprocessing import MaxAbsScaler, MinMaxScaler
from sklearn.utils import shuffle
from sklearn.metrics import confusion_matrix

# Load classifiers
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier,
    AdaBoostClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.gaussian_process import GaussianProcessClassifier

dtc = DecisionTreeClassifier(random_state=0)
rfc = RandomForestClassifier(random_state=0)
gbc = GradientBoostingClassifier(random_state=0)
abc = AdaBoostClassifier(n_estimators=100, random_state=0)
mlpc = MLPClassifier(random_state=1, max_iter=300)
gpc = GaussianProcessClassifier()
```

In this example we intend to distinguish triggers generated by:

- A type of transient noise called Blip glitch.
- A gravitational wave entering the detector.

For this aim we coded the following functions:

- load_data: this function loads the pre-processed data set. The inputs have 2 variables: chirp mass and effective spin.
- decision_boundariesplots: this function plots the boundaries defined by the classifiers (modified from https://scikit-learn.org/stable/auto_examples/tree/plot_iris_dtc.html#sphx-glr-auto-examples-tree-plot-iris-dtc-py)
- get_scores_with_decision_boundaries: to get the accuracy of the classifier and plot decision boundaries.

```
def load_data(train_percentage):
    """
        We load a balanced (N injections, N glitches) and shuffle the dataset.
        We choose how much data we want to train and test (in percentage).
    Input
    -----
    train_percentage: training and testing split (train: train_percentage , test:
        1-train_percentage)

    Output
    ------
    X, y: dataset for classification in the form of (input, target)
    """
    X = pd.read_pickle('./data/input.csv')
    y = np.load('./data/target.npy')

    X, y = shuffle(X, y)

```

```
    X_train = X[:int(len(X)*train_percentage)]
    y_train = y[:int(len(X)*train_percentage)]
    X_test = X[int(len(X)*train_percentage):]
    y_test = y[int(len(X)*train_percentage):]
    return X_train, y_train, X_test, y_test
```

```
def decision_boundaries_plots(input_list, target_list, label_list, clf):
    """
        This function plots the decission boundaries of the desired classifier.

    Input
    ------
    input_list: (list) contains training input and testing input as numpy arrays
    target_list: (list) contains training target and testing target as numpy arrays
    label_list: (list) contains label of training and testing as str

    Output
    ------
    plots

    """

    ct=0
    fig, axs = plt.subplots(1, 2, figsize=(10,6))
    for x, y, stage in zip(input_list, target_list, label_list):

        X = np.array(x)
        plot_step = 0.02
        x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
        y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
        xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step), np.arange(y_min, y_max,
            plot_step))

        Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        cs = axs[ct].contourf(xx, yy, Z, cmap=plt.cm.RdYlBu, alpha=0.6)

        glt = list()
        inj = list()
        for i in range(len(X)):
            if y[i] == 0:
                glt.append(X[i])
            else:
                inj.append(X[i])
        glt = np.array(glt); inj = np.array(inj)

        axs[ct].scatter(glt[:, 0], glt[:, 1], c='crimson', label='Glitches',
            edgecolor='maroon')
        axs[ct].scatter(inj[:, 0], inj[:, 1], c='blue', label='Injections',
            edgecolor='navy')
        axs[ct].set_xlabel(r'$\chi_{eff}$', fontsize=13),
            axs[ct].set_ylabel(r'$\mathcal{M}$', fontsize=13)
        axs[ct].set_title(stage, fontsize=15)
        axs[ct].set_xlim(-0.05, 1.05), axs[ct].set_ylim(-0.05, 1.05)
        axs[ct].legend()
        ct=ct+1
    fig.suptitle('Glitches against injections classified by '+str(c), fontsize=15)
    plt.tight_layout()
    plt.show()
```

```
def get_scores_with_decision_boundaries(train_percentage, classifier):
    """
        We return the accuracy of each classifier for a certain glitch type. TODO: extend
            to multi-class problem.

    Input
    -----
```

```
    template_bank: dataset with 'bags' of feature vectors
    inj_data: dataset with injections. TODO: update dataset
    types: glitch type to classify
    train_percentage: training and testing split (train: train_percentage , test:
        1-train_percentage)
    classifier: type of classifier defined above

    Output
    ------
    acc_test: test accuracy of the classifier

    """
    X_train, y_train, X_test, y_test = load_data(train_percentage=0.8)
    clf = classifier.fit(X_train, y_train)

    decision_boundaries_plots([X_train, X_test], [y_train, y_test], ['Training',
        'Testing'], clf)

    y_pred_train = clf.predict(X_train)
    y_pred = clf.predict(X_test)
    acc_test = clf.score(X_test, y_test)
    print('Training accuracy: '+str(np.round(clf.score(X_train, y_train), 3))+', testing
        accuracy: '+str(np.round(acc_test, 3)))
    print('Confusion matrix of training:'+str(confusion_matrix(y_train,
        y_pred_train).ravel()))
    print('Confusion matrix of training:'+str(confusion_matrix(y_test, y_pred).ravel()))
    return acc_test
```

```
classifiers = ['DecissionTree', 'RandomForest', 'GradientBoosting', 'AdaBoost',
    'NeuralNet', 'GaussianProcess']

matrix = np.empty([1, len(classifiers)])
for j, c in enumerate(classifiers):

    if c == 'DecissionTree':
        classifier = dtc

    if c == 'RandomForest':
        classifier = rfc

    if c == 'GradientBoosting':
        classifier = gbc

    if c == 'AdaBoost':
        classifier = abc

    if c == 'NeuralNet':
        classifier = mlpc

    if c == 'GaussianProcess':
        classifier = gpc


    acc_test = get_scores_with_decision_boundaries(0.8, classifier)
    matrix[0][j]= acc_test

df = pd.DataFrame(data=np.round(matrix, 3), index= ['Blips'],
                  columns = classifiers)

display(df)
```

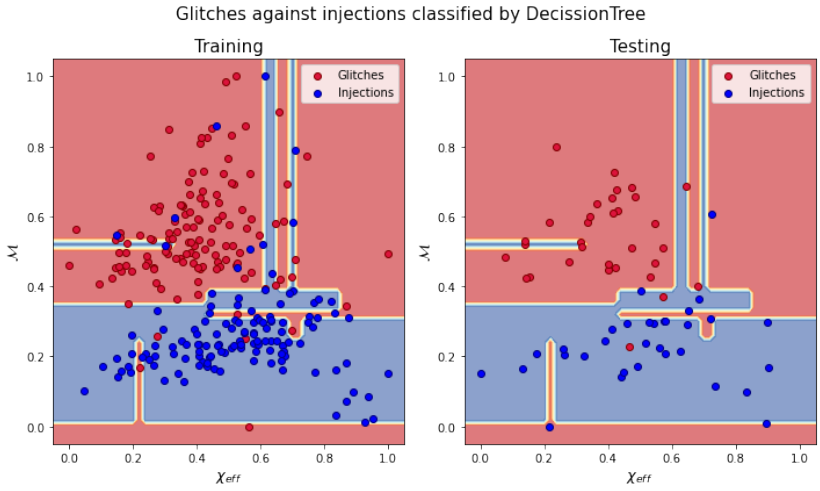


Figure 1.5: png

```
Training accuracy: 1.0, testing accuracy: 0.906
Confusion matrix of training:[127   0   0 127]
Confusion matrix of training:[29  3  3 29]
```

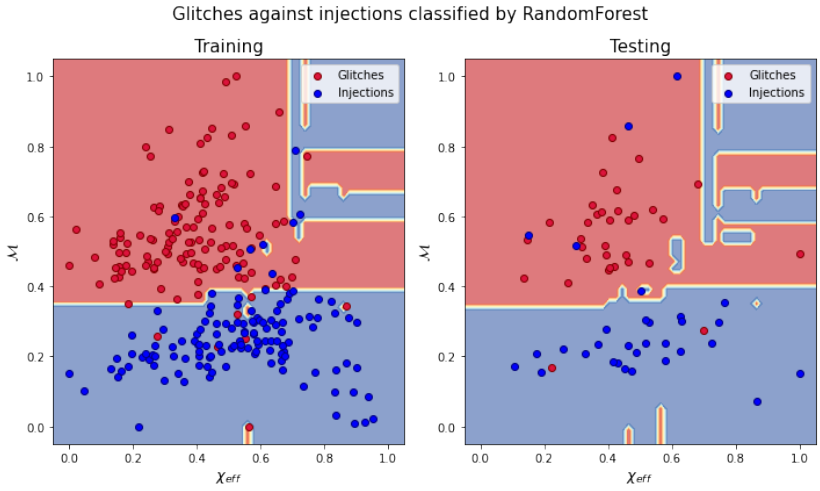


Figure 1.6: png

```
Training accuracy: 1.0, testing accuracy: 0.906
Confusion matrix of training:[126   0   0 128]
Confusion matrix of training:[31  2  4 27]
```

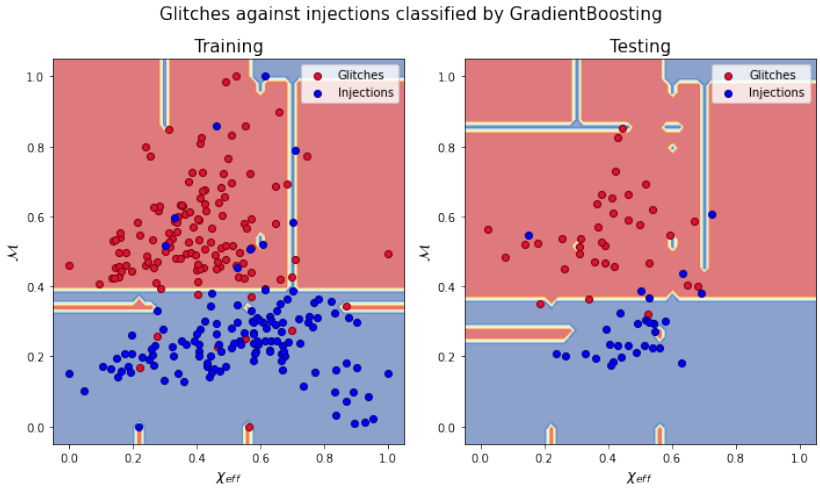


Figure 1.7: png

```
Training accuracy: 1.0, testing accuracy: 0.859
Confusion matrix of training:[124   0   0 130]
Confusion matrix of training:[31  4  5 24]
```

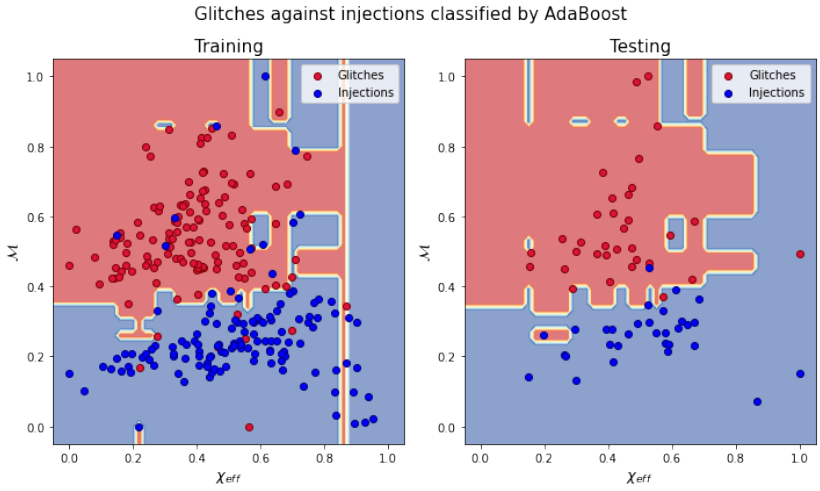


Figure 1.8: png

```
Training accuracy: 0.957, testing accuracy: 0.875
Confusion matrix of training:[120   6   5 123]
Confusion matrix of training:[30  3  5 26]
```

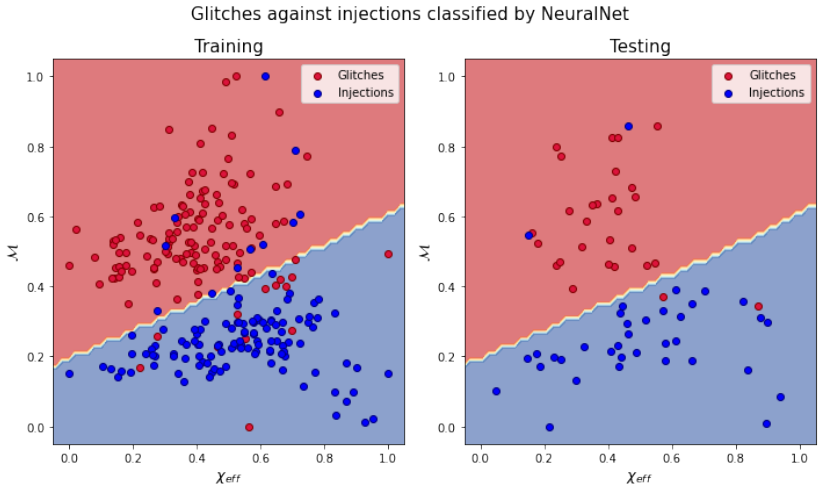


Figure 1.9: png

```
Training accuracy: 0.902, testing accuracy: 0.938
Confusion matrix of training:[117  14  11 112]
Confusion matrix of training:[26  2  2 34]
```

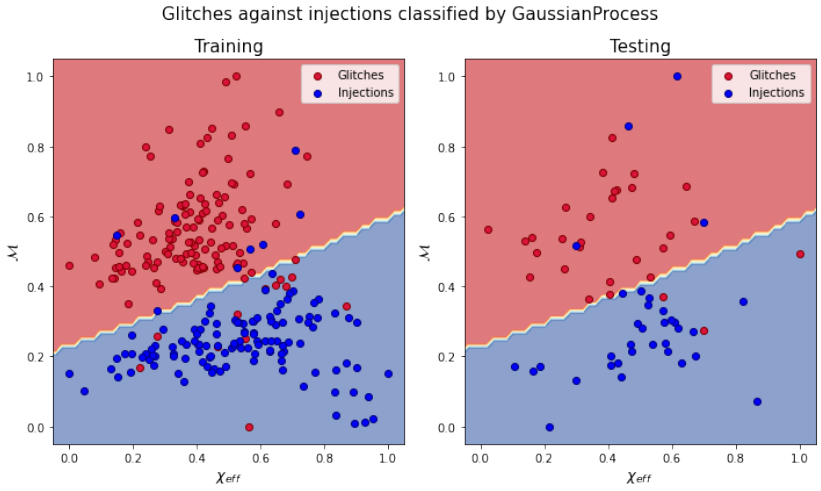


Figure 1.10: png

```
Training accuracy: 0.917, testing accuracy: 0.891
Confusion matrix of training:[116  13   8 117]
Confusion matrix of training:[27  3  4 30]
```

In summary:

	DecissionTree	RandomForest	GradientBoosting	AdaBoost	NeuralNet	GaussianProcess
Blips	0.906	0.906	0.859	0.875	0.938	0.891

Which is the best classifier for this glitch? Do the same process for koy fish glitch (input2.csv adn target2.npy). Is the best classifier still the same? Why? Explore other classifiers from sklearn (https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html).

1.10 Lecture 7: Kernel methods

Listing 1.2: Gaussian Kernel.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as sig

area, river_flow = np.loadtxt('area_flow.txt', dtype=float, delimiter=',', skiprows=1,
    unpack=True)

##########################
# Plot data
##########################
plt.figure()
plt.plot(area, river_flow, 'bo')
plt.xlabel('Area')
plt.ylabel('River Flow')
plt.xlim(0,120)
plt.ylim(0,3000)
plt.savefig('data_set.png')

##########################
# Try kernel regression
##########################
#
# Construct the Gaussian kernel function
# around each data point.

def gaussian_kernel(xj, xi, h):

        K = 1./(h*np.sqrt(2.*np.pi))*np.exp(-0.5*((xj-xi)/h)**2.0)
        return K

# Define linearly spaced series of data points which
# include observed data points and can be used to
# evaluate the Gaussian kernel
xj = np.array(range(5, 101, 1))

# Define the bandwidth
h = 10

# For each area data point, calculate the value of each kernel,
# the corresponding weights, and then the predicted river
# flow value.
river_flow_pred = []
for idx, query_point in enumerate(area):
        K = 1./(h*np.sqrt(2.*np.pi))*np.exp(-0.5*((area-query_point)/h)**2.0)
        weights = K/np.sum(K)
        kj = np.sum(river_flow * weights)
        print (river_flow * weights)
        print (kj)
        print (idx)
        river_flow_pred.append(np.sum(river_flow * weights))

plt.figure()
plt.plot(area, river_flow, 'bo')
plt.plot(area, np.array(river_flow_pred), 'b-')
plt.xlabel('Area')
plt.ylabel('River Flow')
plt.xlim(0,120)
plt.ylim(0,3000)
plt.show()
#plt.savefig('data_set.png')
```

The code runs with just *phyton gaussian$_k$ernel$_r$egression.py* once you properly installed NumPy extension, and you can expect as a result:

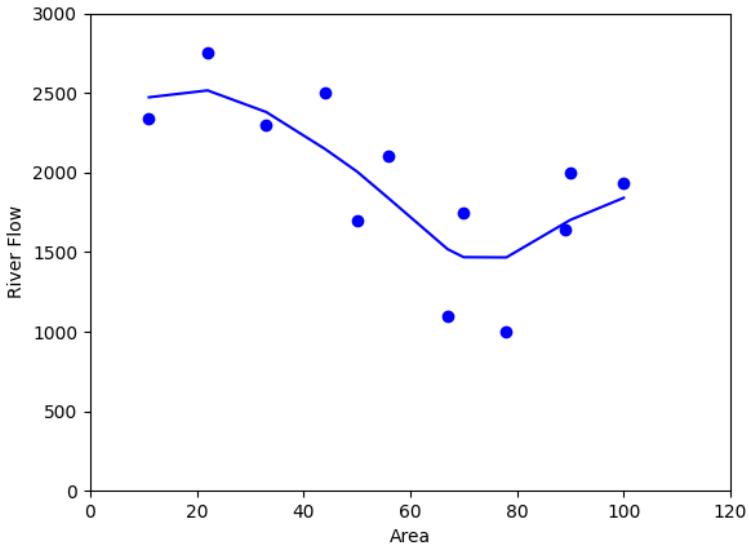


Figure 1.11: Regression result on top of the original data.

1.11 Lecture 9: A simple neural network

1.11.1 Environment

The neuronal network is implemented in PyCharm with a Python 3.6 interpreter, but can also be executed with Jupyter. The overall program consists of two files and uses the packages NumPy and Matplotlib:

1. SimpleNN.py In this file the class SimpleNN is implemented, which contains all necessary functions for the neuronal network, like training, predicting and visualizing.

    - **init: init (self,** *par_alpha*, *par_lambda*) The constructor of the class has the parameters α and λ as inputs and stores them in the class for later use.

    - **training: trainNN(self,** $input\_values, output\_values, number\_epochs$) The number of the layers and neurons is fixed (three layers with 8x3x8 neurons in total). This means that this particular implementation of a NN can only be trained with 8 different training sets, which respectively contain 8 elements (8x8 matrix). As a result, the input parameters input values and output values must be a 8x8 matrix. The parameter number of epochs specifies the number of times the training sets are processed by the network.

    - **predicting: predict(**$self, testset\_x, visualize$) After the neuronal network is trained, the predicting function can be called. This function runs the forward propagation steps for the given inputs in test set x. This input is a Nx8 matrix (N = number of test sets), which means that the number of test sets is not fixed. With the remaining visualize flag (True/False), it is possible to trigger the visualization function of the NN after the test sets are processed.

    - **visualization: visualizeNN(...)** The visualization function has all values of the

NN as input parameters (neurons, biases, weights) and displays them in a 2D plot. The corresponding values are coded in colours.

2. Main.py In this file all training sets and test sets are specified. Those are used to train and test the NN (SimpleNN class). By default, the complete training process (10000 runs) is executed for $\alpha = 0$ to 0.9 (0.1 steps) and fixed $\lambda = 0.0001$. After the execution, two plots are loaded and show error rates and nn-weights. The total run time is approx. 55 seconds (if the Main.py file is executed without change).

1.11.2 SimpleNN.py

Listing 1.3: a Simple Neural Network in Python.

```
import numpy as np
import matplotlib.pyplot as plt

#---------------------------
def sigmoid(x):
    return 1/(1 + np.exp(-x))
#---------------------------

#this class assumes three layers with 8x3x8 nodes
class SimpleNN:
    def __init__(self, par_alpha, par_lambda):

        #learing rate parameter
        self.par_alpha = par_alpha

        #weight decay parameter
        self.par_lambda = par_lambda

        # init the weight matrices1
        np.random.seed(1)

        self.theta_layer1 = np.random.rand(3,8)
        self.theta_layer2 = np.random.rand(8,3)

        # init biases
        self.b_layer1 = np.random.rand(3)
        self.b_layer2 = np.random.rand(8)

        # init activations
        self.a_layer1 = np.array([[0, 0, 0, 0, 0, 0, 0, 0]])
        self.a_layer2 = np.array([[0, 0, 0]])
        self.a_layer3 = np.array([[0, 0, 0, 0, 0, 0, 0, 0]])

        # init storage for all standard deviatons for delta_layer 3 (all runs)
        self.std_d3_CompleteRun = np.array([[0,0,0,0,0,0,0,0]])

    # input_values and output_values are a nx8 matrix for n number testsets
    def trainNN(self, input_values, output_values, number_epochs):
        for epoch in range(number_epochs):

            # Step 1: Init

            Dtheta_layer1 = 0 * np.ones(self.theta_layer1.shape)
            Dtheta_layer2 = 0 * np.ones(self.theta_layer2.shape)
            Db_layer1 = 0 * np.ones(3)
            Db_layer2 = 0 * np.ones(8)

            std_d3_current_epoche = np.zeros(8)

            # Step 2
            for i in range(len(input_values)):
                # get input and output as vector from the input and output matrices
```

```
53                  y = output_values[i, :]
54                  self.a_layer1 = input_values[i, :]
55
56                  # forward propagation
57                  z_layer2 = (self.theta_layer1 @ self.a_layer1) + self.b_layer1
58                  self.a_layer2 = sigmoid(z_layer2)
59
60                  z_layer3 = (self.theta_layer2 @ self.a_layer2) + self.b_layer2
61                  self.a_layer3 = sigmoid(z_layer3)  # layer 3 is already the output (=y)
62
63                  # backpropagation
64                  d_layer3 =  -(y - self.a_layer3) * self.a_layer3 * (1 - self.a_layer3)
65                  std_d3_current_epoche[i] = np.std(d_layer3) # store standard deviation
                        values
66
67                  d_layer2 = self.a_layer2 * (1 - self.a_layer2) * (self.theta_layer2.T @
                        d_layer3)
68
69                  # partial derivatives
70                  jDtheta_layer2 = d_layer3[np.newaxis, :].T * self.a_layer2
71
72                  Dtheta_layer2 = Dtheta_layer2 + jDtheta_layer2
73                  Db_layer2 = Db_layer2 + d_layer3
74
75                  # partial derivatives
76                  jDtheta_layer1 = d_layer2[np.newaxis, :].T * self.a_layer1
77
78                  Dtheta_layer1 = Dtheta_layer1 + jDtheta_layer1
79                  Db_layer1 = Db_layer1 + d_layer2
80
81              # add the vector standard deviation vector (value for all testsets in the
                    current epoche) to the global storage
82              self.std_d3_CompleteRun = np.r_[self.std_d3_CompleteRun,
                    [std_d3_current_epoche]]
83
84              #Step 3
85              self.theta_layer2 = self.theta_layer2 - (self.par_alpha * ((1 /
                    len(input_values)) * Dtheta_layer2 + self.par_lambda * self.theta_layer2))
86              self.b_layer2 = self.b_layer2 - (self.par_alpha * ((1 / len(input_values)) *
                    Db_layer2))
87
88              self.theta_layer1 = self.theta_layer1 - (self.par_alpha * ((1 /
                    len(input_values)) * Dtheta_layer1 + self.par_lambda * self.theta_layer1))
89              self.b_layer1 = self.b_layer1 - (self.par_alpha * ((1 / len(input_values)) *
                    Db_layer1))
90
91          # selete first row, because the first row was just initialized with zeros
92          self.std_d3_CompleteRun = np.delete(self.std_d3_CompleteRun, 0, 0)
93
94      # forward propagation for the inputs. Each row is a test stet. With visualize=True the
            Network is visualized
95      def predict(self, testset_x, visualize):
96
97          z_layer2 = (self.theta_layer1 @ testset_x.T) + np.array([self.b_layer1, ] *
                len(testset_x)).T
98          a_layer2 = sigmoid(z_layer2)
99
100         z_layer3 = (self.theta_layer2 @ a_layer2) + np.array([self.b_layer2, ] *
                len(testset_x)).T
101         a_layer3 = sigmoid(z_layer3)
102
103         y = a_layer3
104
105         if(visualize):
106             self.visualizeNN(testset_x, a_layer2, a_layer3, self.b_layer1, self.b_layer2,
                    self.theta_layer1, self.theta_layer2);
107
108         #for x in testset_x:
```

```
109         #    # forward propagation
110         #    z_layer2 = (self.theta_layer1 @ x) + self.b_layer1;
111         #    self.a_layer2 = hf.sigmoid(z_layer2);
112 
113         #    z_layer3 = (self.theta_layer2 @ self.a_layer2) + self.b_layer2;
114         #    self.a_layer3 = hf.sigmoid(z_layer3);  # layer 3 is already the output
115 
116         #    y = np.r_[y, [self.a_layer3]];
117         #y = np.delete(y, 0, 0);
118 
119 
120         return y
121 
122     def visualizeNN(self, a1_matrix, a2_matrix, a3_matrix, b1_vector, b2_vector,
            weight_matrix1, weight_matrix_2):
123 
124         plt.figure(figsize=(15, 8))
125 
126         plt.subplot(2, 5, 1)
127         plt.imshow(np.expand_dims(b1_vector, axis=0), cmap='seismic',
                interpolation='nearest', vmin=-3, vmax=3)
128         plt.xlabel('neurons')
129         plt.title('b Layer 1')
130         #plt.colorbar()
131 
132         plt.subplot(2, 5, 6)
133         plt.imshow(a1_matrix, cmap='seismic', interpolation='nearest', vmin=-3, vmax=3)
134         plt.xlabel('neurons')
135         plt.ylabel('test set')
136         plt.title('A Layer 1')
137         #plt.colorbar()
138 
139         plt.subplot(2, 5, 7)
140         plt.imshow(weight_matrix1.T, cmap='seismic', interpolation='nearest', vmin=-3,
                vmax=3)
141         plt.xlabel('neurons layer 2')
142         plt.ylabel('neurons layer 1')
143         plt.title('Weights Layer 1')
144         #plt.colorbar()
145 
146         plt.subplot(2, 5, 3)
147         plt.imshow(np.expand_dims(b2_vector, axis=0), cmap='seismic',
                interpolation='nearest', vmin=-3, vmax=3)
148         plt.xlabel('neurons')
149         plt.title('b Layer 2')
150         #plt.colorbar()
151 
152         plt.subplot(2, 5, 8)
153         plt.imshow(a2_matrix.T, cmap='seismic', interpolation='nearest', vmin=-3, vmax=3)
154         plt.ylabel('test set')
155         plt.xlabel('neurons')
156         plt.title('A Layer 2')
157         #plt.colorbar()
158 
159         plt.subplot(2, 5, 9)
160         plt.imshow(weight_matrix_2.T, cmap='seismic', interpolation='nearest', vmin=-3,
                vmax=3)
161         plt.xlabel('neurons layer 3')
162         plt.ylabel('neurons layer 2')
163         plt.title('Weights Layer 2')
164         #plt.colorbar()
165 
166         plt.subplot(2, 5, 10)
167         plt.imshow(a3_matrix.T, cmap='seismic', interpolation='nearest', vmin=-3, vmax=3)
168         plt.title('A Layer 3')
169         plt.ylabel('test set')
170         plt.xlabel('neurons')
171         plt.colorbar()
```

```
172
173         plt.show()
174
175         return
```

1.11.3 mainNN

Lecture 4 of Computational Aspectes of Machine Learning: course 2021-2022

Credits: Armen Yeritsyan

In this example we code our own NN (see SimpleNN.py for details). Here we give a training set and a test set. For details about the procedure read SimpleNN.pdf.

```
import numpy as np
import matplotlib.pyplot as plt
import SimpleNN as nn
```

MUST be Nx8 Matrix (number of testsets/rows N is variable. Just add or delete rows in this matrix)

```
testset_input = np.array([[1, 0, 0, 0, 0, 0, 0, 0],
                          [0, 1, 0, 0, 0, 0, 0, 0],
                          [0, 0, 1, 0, 0, 0, 0, 0],
                          [0, 0, 0, 1, 0, 0, 0, 0],
                          [0, 0, 0, 0, 1, 0, 0, 0],
                          [0, 0, 0, 0, 0, 1, 0, 0],
                          [0, 0, 0, 0, 0, 0, 1, 0],
                          [0, 0, 0, 0, 0, 0, 0, 1],
                          [0, 0, 1, 0, 0, 0, 0, 0],
                          [0, 1, 0, 0, 0, 0, 0, 0],
                          [0, 0, 0, 0, 1, 0, 0, 0],
                          [0, 0, 0, 0, 0, 0, 0, 0],
                          [0, 0, 0, 0, 0, 0, 1, 0],
                          [1, 0, 0, 0, 0, 0, 0, 0],
                          [0, 0, 0, 0, 0, 0, 0, 1],
                          [0, 0, 0, 1, 0, 0, 0, 0],])
```

MUST bet 8x8 matrix! You can change the order and the values of the rows, but can't change the number of the rows!

```
training_inputs = np.array([[1, 0, 0, 0, 0, 0, 0, 0],
                            [0, 1, 0, 0, 0, 0, 0, 0],
                            [0, 0, 1, 0, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 0, 0, 0],
                            [0, 0, 0, 0, 1, 0, 0, 0],
                            [0, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 0, 0, 0, 0, 0, 1],])
```

In this case the outputs are the same as the inputs

```
training_outputs = training_inputs
```

We want to see the influence of different learning rates α and weight decays λ across different layers, so we loop over some values.

```
alphas = np.arange(0.0, 1, 0.1)
lambdas = 0.0001
```

```
fig, axs = plt.subplots(2, 5, figsize=(15,10))
axs = axs.ravel()

for c, par_alpha in enumerate(alphas):

    print('Run number ' + str(c))

    simple_nn = nn.SimpleNN(par_alpha, lambdas)

```

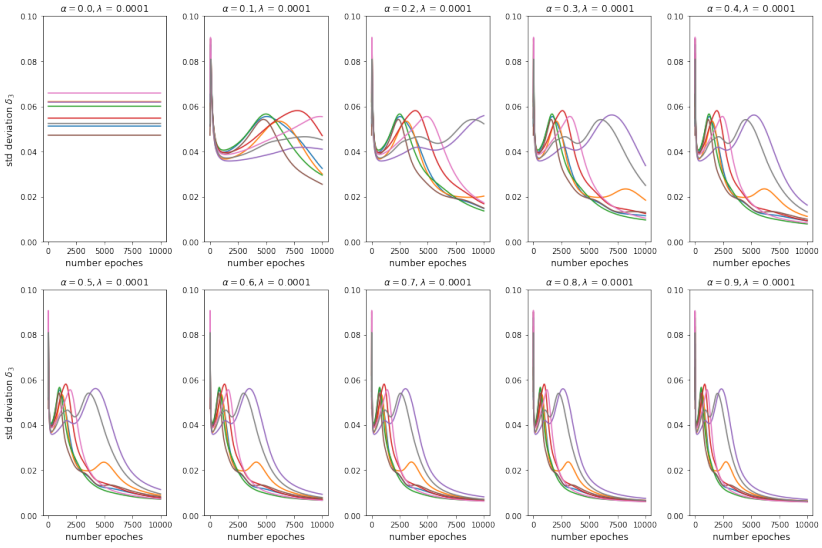


Figure 1.12: png

```
    # run the training routine
    simple_nn.trainNN(training_inputs, training_outputs, 10000)

    # plot standard deviation of the errors in layer 3 of all 8 test sets for this iteration
    axs[c].set_title(r'$\alpha = ' + str(np.round(par_alpha,2)) + ', \lambda$ = ' +
        str(np.round(0.0001,4)))

    for i in range(8):
        axs[c].plot(simple_nn.std_d3_CompleteRun[:,i])
        axs[c].set_ylim(0, 0.1)
        axs[c].set_xlabel('number epoches', fontsize=12)

        if (c == 0) or (c == 5):
            axs[c].set_ylabel(r'std deviation $\delta_3$', fontsize=12)
plt.tight_layout()
plt.show()
```

```
Run number 0
Run number 1
Run number 2
Run number 3
Run number 4
Run number 5
Run number 6
Run number 7
Run number 8
Run number 9
```

Now we visualise our layers

```
y = simple_nn.predict(testset_input, True);
```

We compare the input test and the predicted output

```
# Input
plt.subplot(1,2,1)
plt.imshow(testset_input)
plt.colorbar()

```

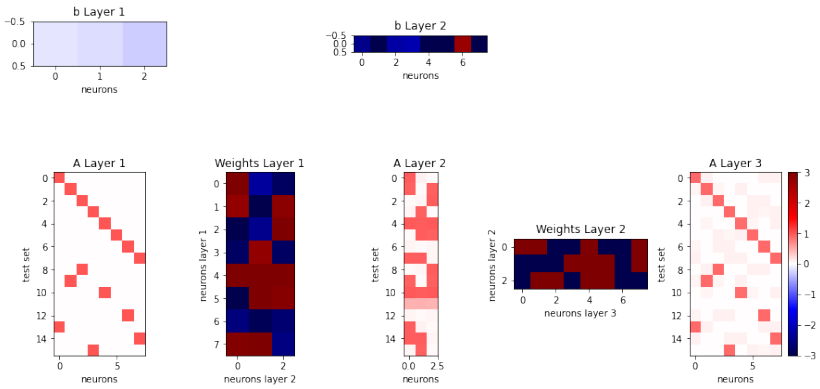


Figure 1.13: Heat map of activations, weights and biases

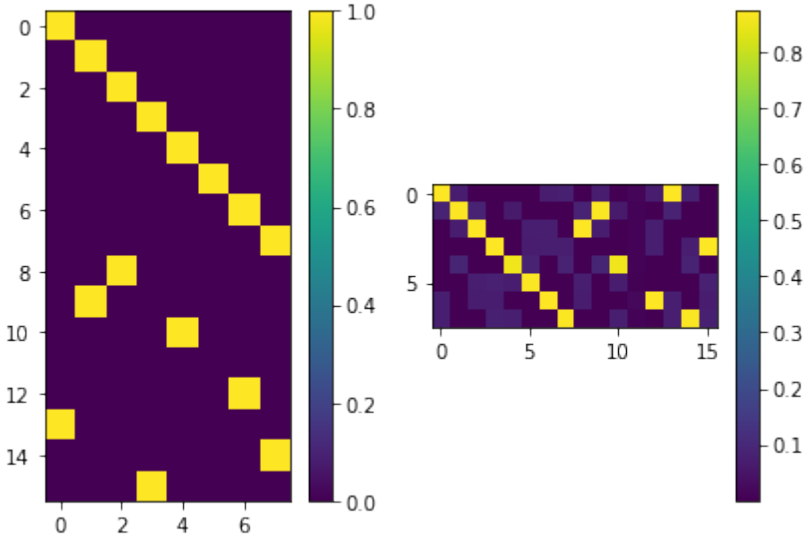


Figure 1.14: png

```
# Predictions
plt.subplot(1,2,2)
plt.imshow(y)
plt.colorbar()
plt.tight_layout()
```

In Fig. 1.11.3 we can see the heat map of $a^{(l)}$ for $l = 1,2,3$. We define $a^{(1)}$ (left side of the figure) to be the input where the vectors are ordered at first but then they introduced randomly in our neural network. In the same way we define $a^{(3)}$ as our output that needs to be exactly the same as our input. Note that this output is only showing the probability of the one being in that position.
We initialize the weights and the biases to random value between 0 and 1 (instead of random values around 0.01) because our results were slightly improved by using this initialization. After this we can compute the activation of layer 2 and 3 in a matrix form as it follows:

$$a^{l+1} = f(W^l a^l + b^l) \tag{1.2}$$

In Fig. 1.11.3 we plotted heat maps of the activations, the weights and the biases. We discovered a fascinating pattern related to $a^{(2)}, W^{(1)}$ and $W^{(2)}$ as we show in Fig. 1.11.3. If we invert $W^{(2)}$ and rotate it 90 degree we get the same pattern as $W^{(1)}$ and the part of $a^{(2)}$ that corresponds to the 'ordered' part of our input, as we have indicated with a blue curly bracket . This might mean that the weights are ordering the inputs to perform a better estimation. The estimation performed in the weights is sent to the different neurons transforming the dimensions to fit the activations. The activation must vary between 0 and 1. We must also note that the white color of the activation means that $a^{(l)} \sim 0$ and the neuron is not active. The red color of the activation means that $a^{(l)} \sim 1$ and the neuron is active.