Data Science Fundamentals 5

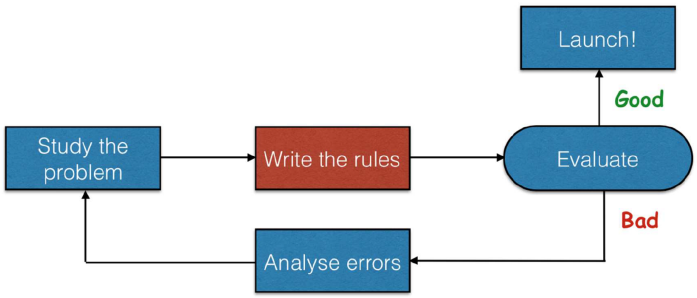
Basic introduction on how to perform typical machine learning tasks with Python.

Prepared by Mykhailo Vladymyrov & Aris Marcolongo, Science IT Support, University Of Bern, 2020

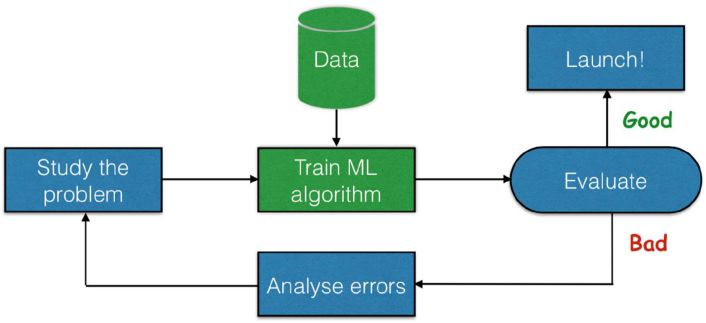
Part 1.

What is Machine learning?

Unlike classical algorithms, created by human to analyze some data:

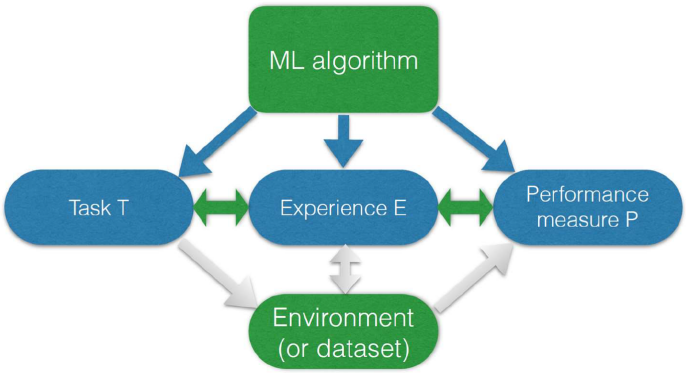


in machine learning the data itself is used for to define the algorithm:



ML:

- improves performance according to measure **P**
- on a task **T**
- with experience **E**



The boundary is a bit fuzzy. In fact when we create algorithms, the problem in hand, namely the data related to the problem, drives us to chose one or another algorithm. And we then tune it, to perform well on a task in hand. ML formalized this procedure, allowing us to automate (part) of thise process.

In this 2-day course you will get acquainted with the basics of ML, where the approach to handling the data (the algorithm) is defined, or as we say "learned" from data in hand.

Classification vs regression.

The two main tasks handled by (supervised) ML is regression and classification. In regression we aim at modeling the relationship between the system's response (dependent variable) and one or more explanatory variables (independent variables).

Examples of regression would be predicting the temperature for each day of the year, or expenses of the household as a function of the number of children and adults.

In classification the aim is to identify what class does a data-point belong to. For example the spieces or the iris plant based on the size of it's petals, or whether an email is spam or not based on it's content.

Performance measures

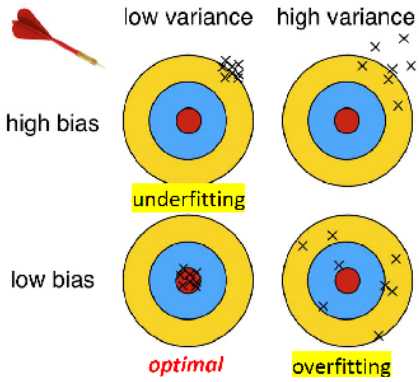
1. Regression:
2. Mean Square Error (MSE): $mse = \frac{1}{n}\sum_i (y_i - \hat{y}(\bar{x}_i))^2$
3. Mean Absolute Error (MAE): $mae = \frac{1}{n}\sum_i |y_i - \hat{y}(\bar{x}_i)|$
4. Median Absolute Deviation (MAD): $mad = median(|y_i - \hat{y}(\bar{x}_i)|)$
5. Fraction of the explained variance: $R^2 = 1 - \frac{\sum_i (y_i - \hat{y}(\bar{x}_i))^2}{\sum_i (y_i - \bar{y}_i)^2}$, where $\bar{y} = \frac{1}{n}\sum_i y_i$
6. Classification:
7. Confusion matrix

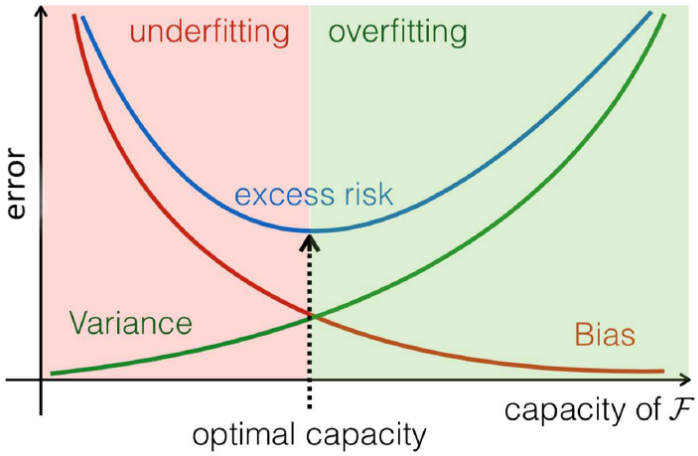
		Prediction	
		Predicted 1	Predicted 0
Ground truth	Class 1	**TP**	**FN**
	Class 0	**FP**	**TN**

- Accuracy $= \frac{TP+TN}{TP+FP+FN+TN}$
- Precision $= \frac{TP}{TP+FP}$
- Recall $= \frac{TP}{TP+FN}$
- F1 $= 2\frac{Precision \cdot Recall}{Precision + Recall} = \frac{2TP}{2TP+FP+FN}$
- Threat score(TS), or Intersection over Union (IoU): IoU=$\frac{TP}{TP+FN+FP}$

During model optimization the used measure in most cases must be differentiable. To this end usually some measure of similarities of diestributions are employed (e.g. cross-entropy).

Actual aim: Generalization





To measure model performace in an unbiassed way, we need to use different data than the data that the model was trained on. For this we use the 'train-test' split: e.g. 20% of all available dataset is reserved for model performance test, and the remaining 80% is used for actual model training.

Load libraries

In [0]:
```
from sklearn import linear_model

from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn import metrics

from matplotlib import pyplot as plt
import numpy as np
import os
from imageio import imread
import pandas as pd
from time import time as timer

import tensorflow as tf

%matplotlib inline
from matplotlib import animation
from IPython.display import HTML
```

In [0]:
```
if not os.path.exists('data'):
    path = os.path.abspath('.')+'/colab_material.tgz'
    tf.keras.utils.get_file(path, 'https://github.com/ne
worldemancer/DSF5/raw/master/colab_material.tgz')
    !tar -xvzf colab_material.tgz > /dev/null 2>&1
```

Datasets

In this course we will use several synthetic and real-world datasets to ilustrate the behavior of the models and excercise our skills.

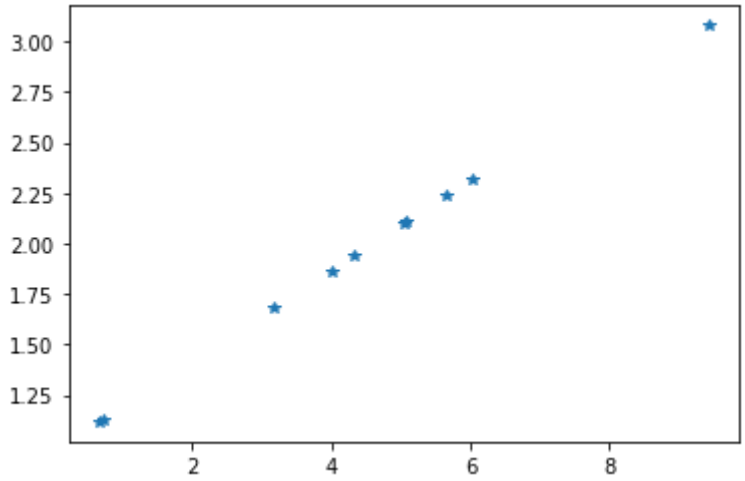
1. Synthetic linear

In [0]:
```
def get_linear(n_d=1, n_points=10, w=None, b=None, sigma
=5):
  x = np.random.uniform(0, 10, size=(n_points, n_d))

  w = w or np.random.uniform(0.1, 10, n_d)
  b = b or np.random.uniform(-10, 10)
  y = np.dot(x, w) + b + np.random.normal(0, sigma, size
=n_points)

  print('true w =', w, ';  b =', b)

  return x, y
```

In [0]:
```
x, y = get_linear(n_d=1, sigma=0)
plt.plot(x[:, 0], y, '*')
```

```
true w = [0.22340972] ;  b = 0.9711035072823542
```

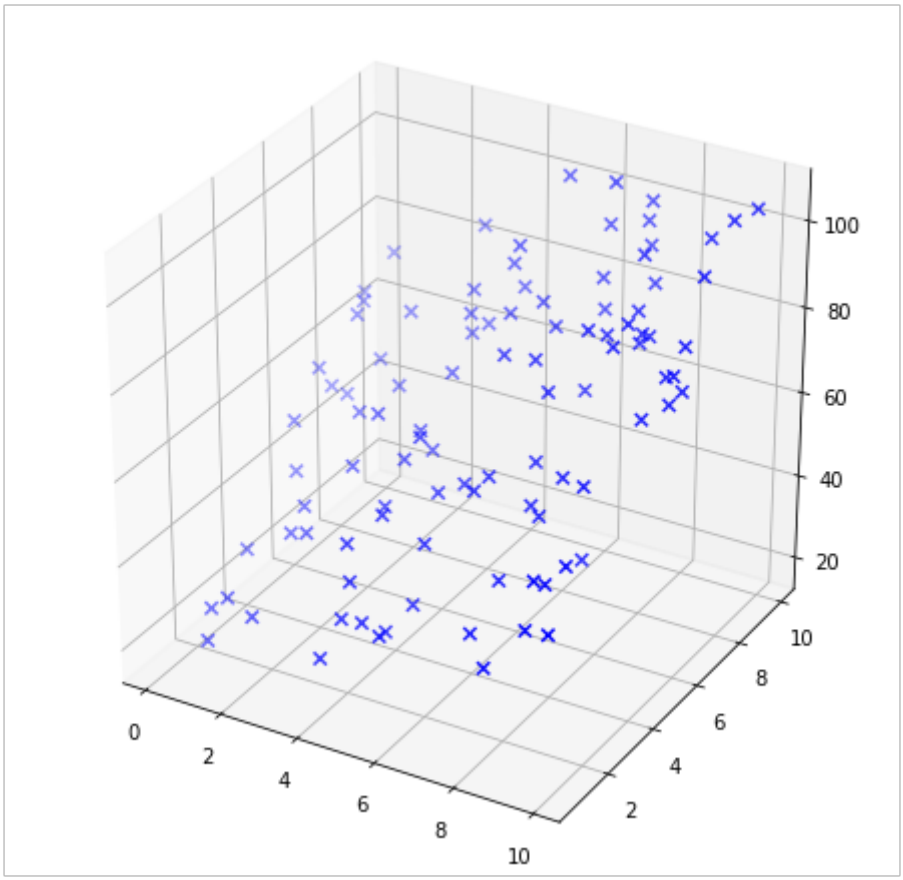
Out[0]:
```
[<matplotlib.lines.Line2D at 0x7fb2c6f56940>]
```



In [0]:
```
n_d = 2
x, y = get_linear(n_d=n_d, n_points=100)

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x[:,0], x[:,1], y, marker='x', color='b',s=40)
```

```
true w = [3.53409832 6.67393245] ;  b = 7.180485793619763
```

Out[0]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7fb2c6a40ba8>
```



2. House prices

Subset of the the hous pricess kaggle dataset: https://www.kaggle.com/c/house-prices-advanced-regression-techniques (https://www.kaggle.com/c/house-prices-advanced-regression-techniques)

In [0]:
```
def house_prices_dataset(return_df=False, price_max=400000, area_max=40000):
  path = 'data/train.csv'

  df = pd.read_csv(path, na_values="NaN", keep_default_na=False)

  useful_fields = ['LotArea',
                   'Utilities', 'OverallQual', 'OverallCond',
                   'YearBuilt', 'YearRemodAdd', 'ExterQual', 'ExterCond',
                   'HeatingQC', 'CentralAir', 'Electrical',
                   '1stFlrSF', '2ndFlrSF','GrLivArea',
                   'FullBath', 'HalfBath',
                   'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual', 'TotRmsAbvGrd',
                   'Functional','PoolArea',
                   'YrSold', 'MoSold'
                   ]
  target_field = 'SalePrice'

  cleanup_nums = {"Street":      {"Grvl": 0, "Pave": 1},
                   "LotFrontage": {"NA":0},
                   "Alley":       {"NA":0, "Grvl": 1, "Pave": 2},
                   "LotShape":    {"IR3":0, "IR2": 1, "IR1": 2, "Reg":3},
                   "Utilities":   {"ELO":0, "NoSeWa": 1, "NoSewr": 2, "AllPub": 3},
                   "LandSlope":   {"Sev":0, "Mod": 1, "Gtl": 3},
                   "ExterQual":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "Ex":4},
                   "ExterCond":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "Ex":4},
                   "BsmtQual":    {"NA":0, "Po":1, "Fa": 2, "TA": 3, "Gd": 4, "Ex":5},
                   "BsmtCond":    {"NA":0, "Po":1, "Fa": 2, "TA": 3, "Gd": 4, "Ex":5},
                   "BsmtExposure":{"NA":0, "No":1, "Mn": 2, "Av": 3, "Gd": 4},
                   "BsmtFinType1":{"NA":0, "Unf":1, "LwQ": 2, "Rec": 3, "BLQ": 4, "ALQ":5, "GLQ":6},
                   "BsmtFinType2":{"NA":0, "Unf":1, "LwQ": 2, "Rec": 3, "BLQ": 4, "ALQ":5, "GLQ":6},
                   "HeatingQC":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "Ex":4},
                   "CentralAir":  {"N":0, "Y": 1},
                   "Electrical":  {"NA":0, "Mix":1, "FuseP":2, "FuseF": 3, "FuseA": 4, "SBrkr": 5},
                   "KitchenQual": {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "Ex":4},
                   "Functional":  {"Sal":0, "Sev":1, "Maj
```

In [0]:
```
x, y, df = house_prices_dataset(return_df=True)
print(x.shape, y.shape)
df.head()
```

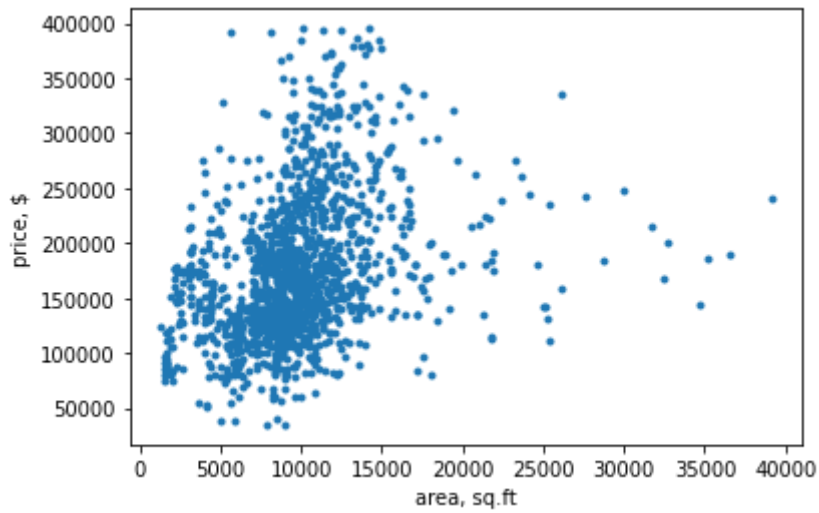
```
(1420, 24) (1420,)
```

Out[0]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotSha
0	1	60	RL	65	8450	Pave	NA	Reg
1	2	20	RL	80	9600	Pave	NA	Reg
2	3	60	RL	68	11250	Pave	NA	IR1
3	4	70	RL	60	9550	Pave	NA	IR1
4	5	60	RL	84	14260	Pave	NA	IR1

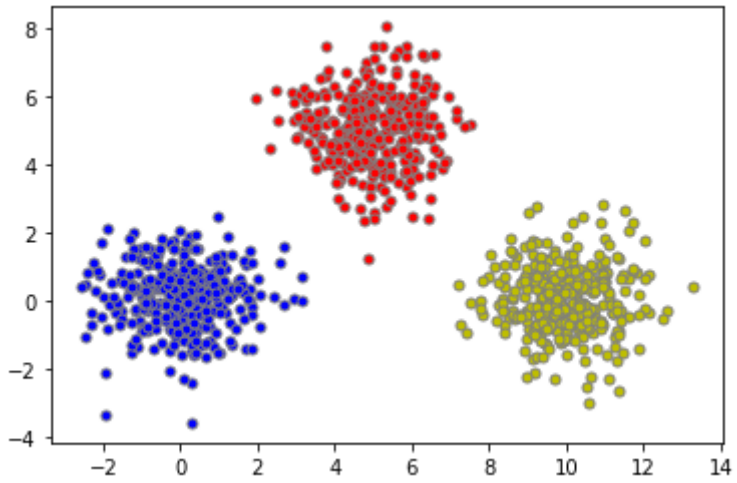
5 rows × 81 columns

In [0]:
```
plt.plot(x[:, 0], y, '.')
plt.xlabel('area, sq.ft')
plt.ylabel('price, $');
```



3. Blobs

In [0]:
```
x, y = make_blobs(n_samples=1000, centers=[[0,0], [5,5], [10, 0]])
colors = "bry"
for i, color in enumerate(colors):
    idx = y == i
    plt.scatter(x[idx, 0], x[idx, 1], c=color, edgecolor='gray', s=25)
```



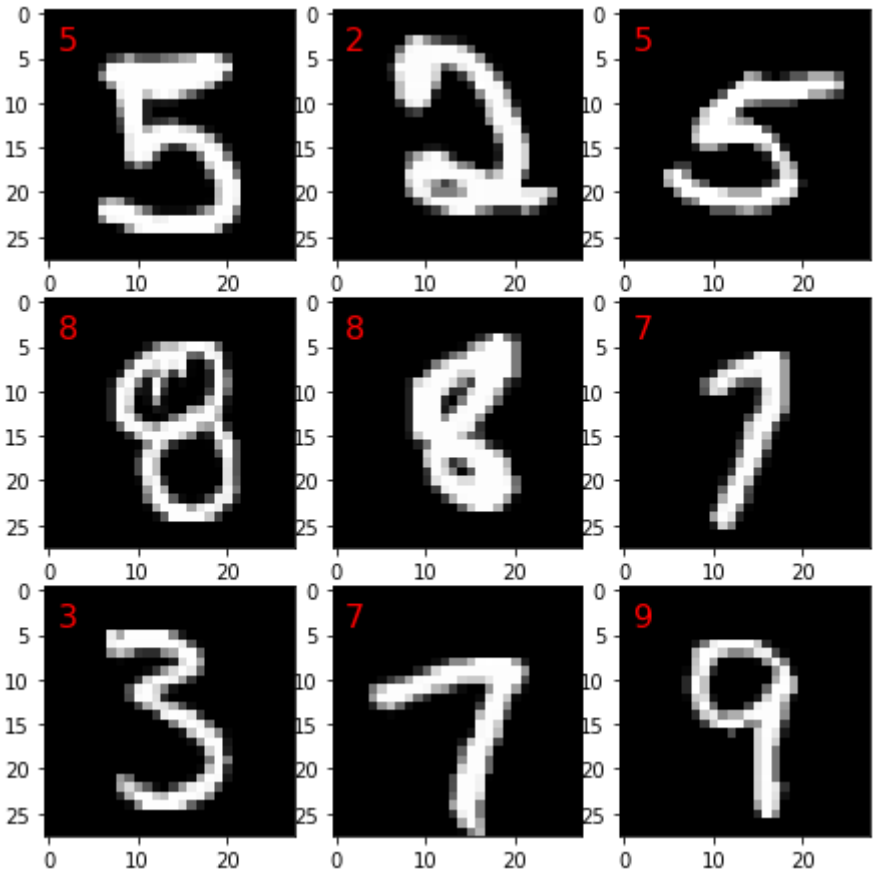
4. MNIST

The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting. (taken from http://yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/)) . Each example is a 28x28 grayscale image and the data-set can be readily downloaded from Tensorflow.

In [0]:
```
mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Let's chech few samples:

In [0]:
```
n = 3
fig, ax = plt.subplots(n, n, figsize=(2*n, 2*n))
ax = [ax_xy for ax_y in ax for ax_xy in ax_y]
for axi, im_idx in zip(ax, np.random.choice(len(train_im
ages), n**2)):
  im = train_images[im_idx]
  im_class = train_labels[im_idx]
  axi.imshow(im, cmap='gray')
  axi.text(1, 4, f'{im_class}', color='r', size=16)
plt.tight_layout(0,0,0)
```



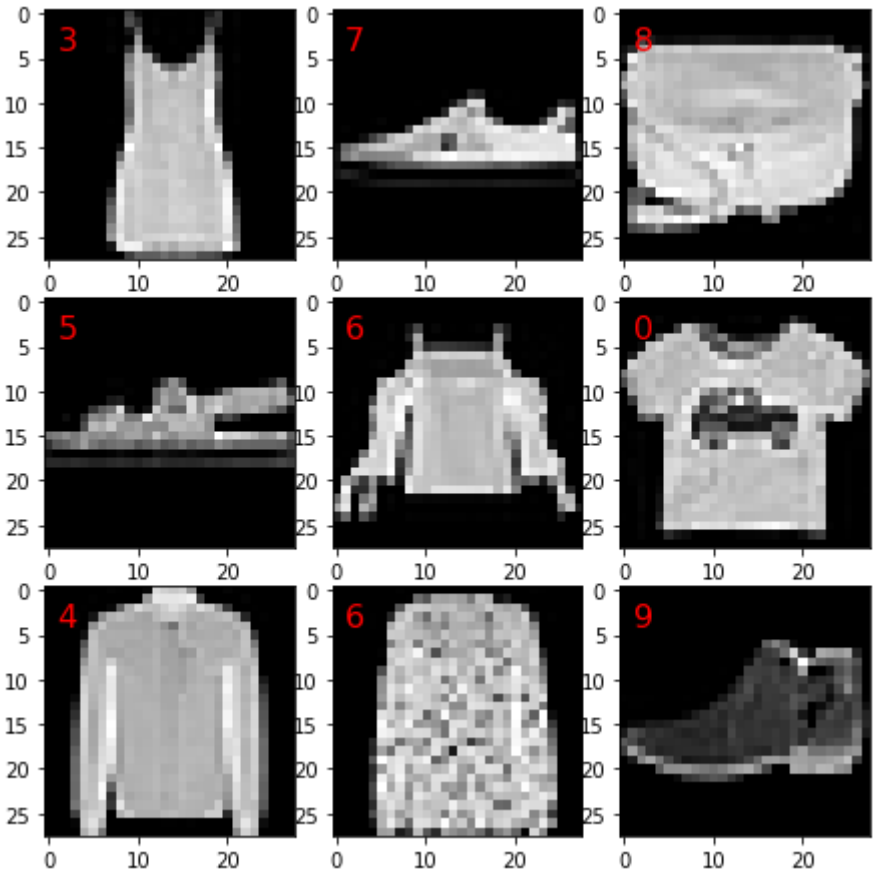
5. Fashion MNIST

`Fashion-MNIST` is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. (from https://github.com/zalandoresearch/fashion-mnist (https://github.com/zalandoresearch/fashion-mnist))

In [0]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels)
= fashion_mnist.load_data()
```

Let's chech few samples:

In [0]:
```
n = 3
fig, ax = plt.subplots(n, n, figsize=(2*n, 2*n))
ax = [ax_xy for ax_y in ax for ax_xy in ax_y]
for axi, im_idx in zip(ax, np.random.choice(len(train_im
ages), n**2)):
  im = train_images[im_idx]
  im_class = train_labels[im_idx]
  axi.imshow(im, cmap='gray')
  axi.text(1, 4, f'{im_class}', color='r', size=16)
plt.tight_layout(0,0,0)
```



Each training and test example is assigned to one of the following labels:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

`scikit-learn` interface

In this course we will primarily use the `scikit-learn` module. You can find extensive documentation with exmples in the user guide (https://scikit-learn.org/stable/user_guide.html)

The module contains A LOT of diffenet mashine learning methods, and here we will cover only few of them. What is great about `scikit-learn` is that it has a uniform and consistent interface.

All the different ML approaches are implemented as classes with a set of same main methods:

1. `fitter = ...`: Create object.
2. `fitter.fit(x, y[, sample_weight])`: Fit model.
3. `y_pred = fitter.predict(X)`: Predict using the linear model.
4. `s = score(x, y[, sample_weight])`: Return an appropriate measure of model performace.

This allows one to easily replace one approach with another, and find the best one for the probkem at hand, by simply using another regression/classification object, while the rest of the code can remain the same.

1.Linear models

In many cases the the scalar value of interest - dependent variable - is (or can be aproximated as) linear combination of the independent variables.

In linear regression the estimator is searched in the form:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + \ldots + w_p x_p$$

The parameters $w = (w_1, \ldots, w_p)$ and w_0 are designated as `coef_` and `intercept_` in `sklearn`.

Reference: https://scikit-learn.org/stable/modules/linear_model.html (https://scikit-learn.org/stable/modules/linear_model.html)

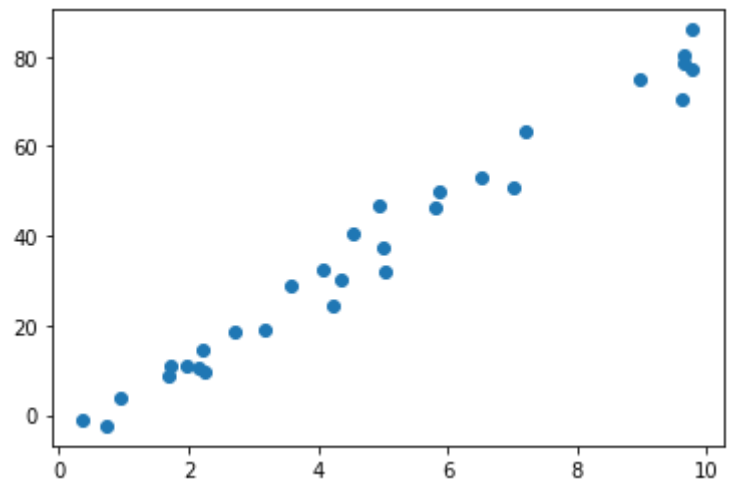
1. Linear regression

LinearRegression fits a linear model with coefficients $w = (w_1, \ldots, w_p)$ and w_0 to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\min_w \; ||Xw - y||_2^2$$

In [0]:
```
x, y = get_linear(n_d=1, sigma=3, n_points=30)  # p==1, 1D input
plt.scatter(x, y);
```

```
true w = [8.45446474] ;  b = -3.1145687950426026
```



In [0]:
```
reg = linear_model.LinearRegression()
reg.fit(x, y)
```

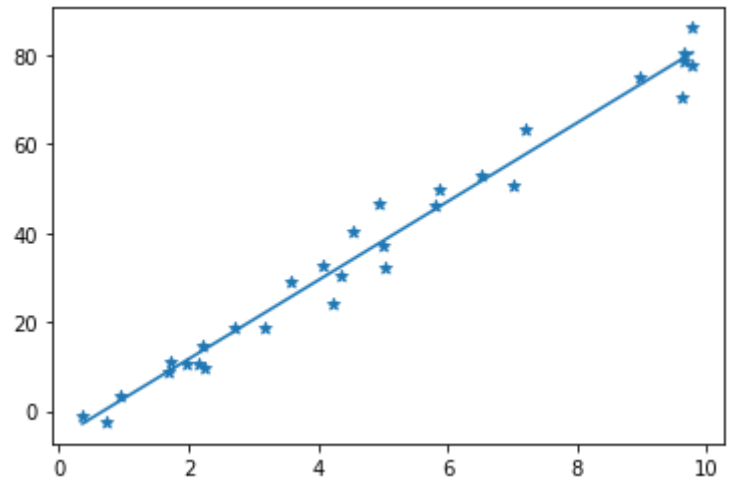
Out[0]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs
=None, normalize=False)
```

In [0]:
```
w, w0 = reg.coef_, reg.intercept_
print(w, w0)
```

```
[8.83547846] -5.992466828806293
```

In [0]:
```
plt.scatter(x, y, marker='*')
x_f = np.linspace(x.min(), x.max(), 10)
y_f = w0 + w[0] * x_f
plt.plot(x_f, y_f)
```

Out[0]:
```
[<matplotlib.lines.Line2D at 0x7fb2c253cd30>]
```



In [0]:
```
# mse
np.std(y - reg.predict(x))  # or use metrics.mean_square
d_error(..., squared=False)
```

Out[0]:
```
3.957683648148916
```

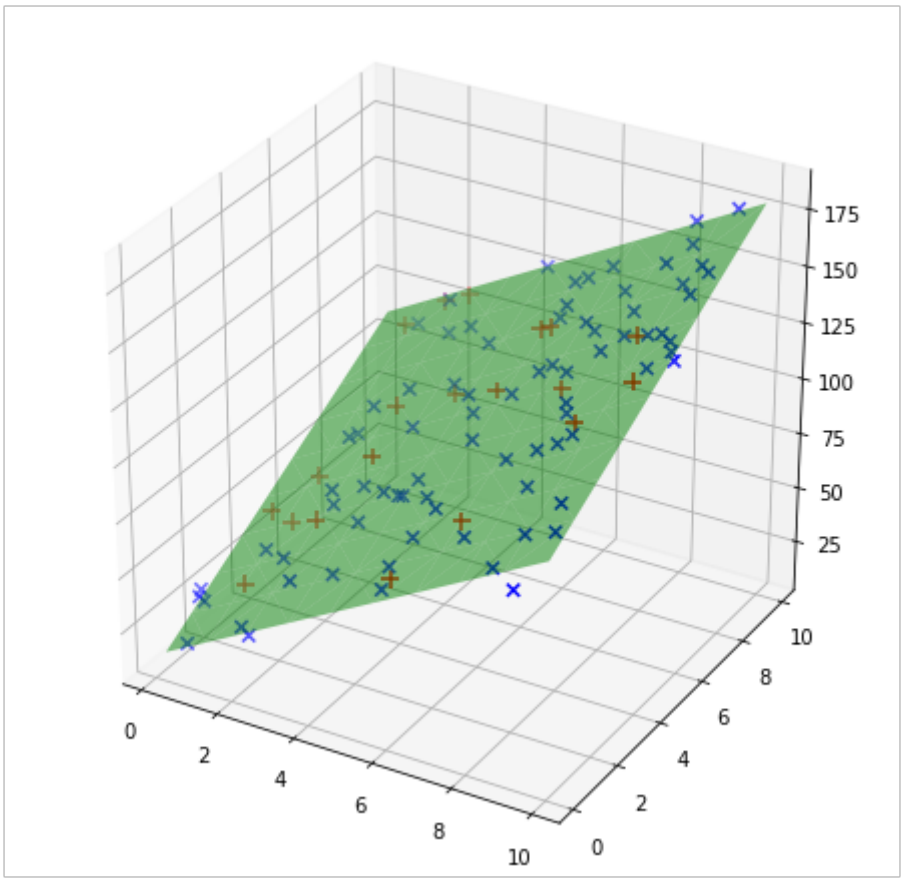
In [0]:
```
# R2
reg.score(x, y)
```

Out[0]:
```
0.9774962341905981
```

Let's try 2D input. Aditionally here we will split the whole dataset into training and test subsets using the `train_test_split` function:

In [0]:
```
n_d = 2
x, y = get_linear(n_d=n_d, n_points=100, sigma=5)

# train test split
x_train, x_test, y_train, y_test = train_test_split(x,
y, test_size=0.2)

reg = linear_model.LinearRegression()
reg.fit(x_train, y_train)


fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x_train[:,0], x_train[:,1], y_train, marker='
x', color='b',s=40)
ax.scatter(x_test[:,0], x_test[:,1], y_test, marker='+',
color='r',s=80)

xx0 = np.linspace(x[:,0].min(), x[:,0].max(), 10)
xx1 = np.linspace(x[:,1].min(), x[:,1].max(), 10)
xx0, xx1 = [a.flatten() for a in np.meshgrid(xx0, xx1)]
xx = np.stack((xx0, xx1), axis=-1)
yy = reg.predict(xx)
ax.plot_trisurf(xx0, xx1, yy, alpha=0.5, color='g');
```

```
true w = [9.67781514 7.94527226] ;  b = 7.37399277228391
6
```



In [0]:
```
# mse
print('train mse =', np.std(y_train - reg.predict(x_trai
n)))
print('test mse =', np.std(y_test - reg.predict(x_tes
t)))
```

```
train mse = 4.825382240735499
test mse = 5.946155392271906
```

In [0]:
```
# R2
print('train R2 =', reg.score(x_train, y_train))
print('test R2 =', reg.score(x_test, y_test))
```

```
train R2 = 0.9839542964836011
test R2 = 0.9619708379511813
```

EXERCISE 1.

Use linear regression to fit house prices dataset.

In [0]:
```
x, y = house_prices_dataset()

# 1. make train/test split

# 2. fit the model

# 3. evaluate MSE, MAE, and R2 on train and test dataset
s

# 4. plot y vs predicted y for test and train parts
```

2. Logistic regression

Logistic regression, despite its name, is a linear model for classification rather than regression. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

In logistic regression the probability p of a point belonging to a class is modeled as:

$$\frac{p}{1-p} = e^{w_0 + w_1 x_1 + \ldots + w_p x_p}$$

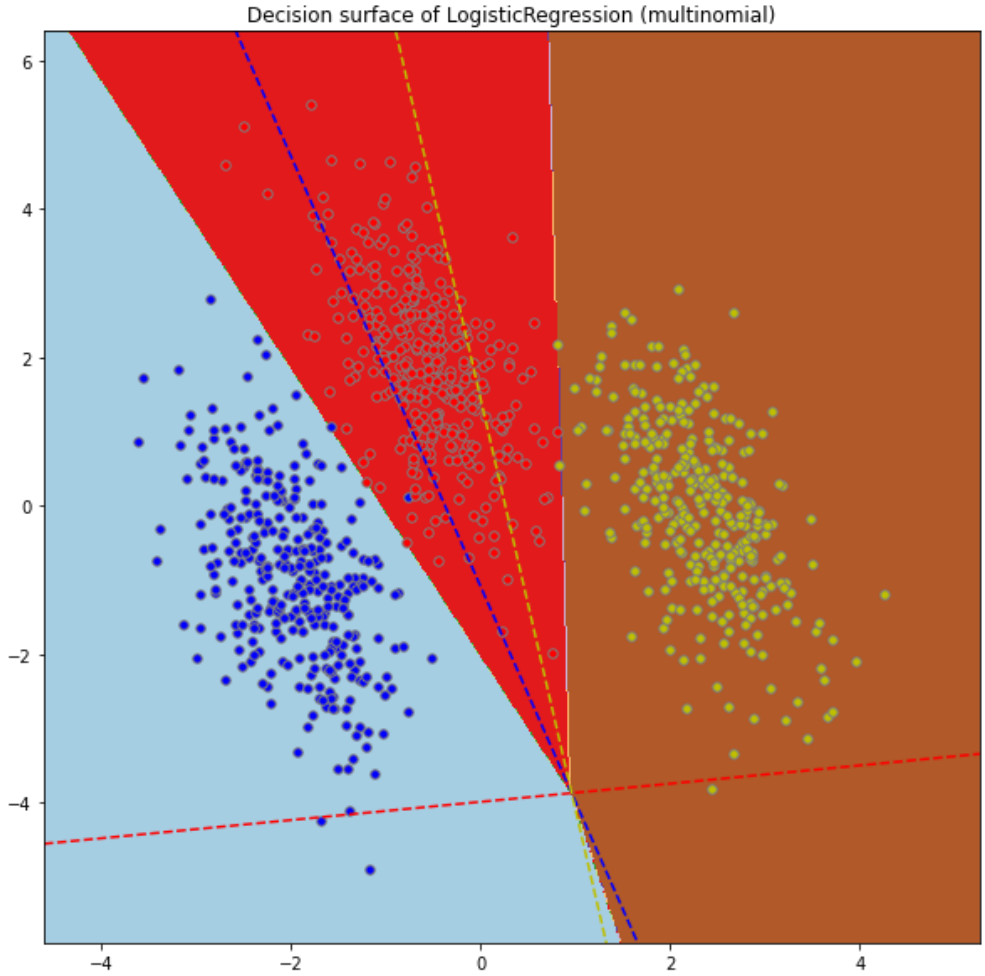
The binary class ℓ_2 penalized logistic regression minimizes the following cost function:

$$\min_{w,c} \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1) + \lambda \frac{1}{2} w^T w$$

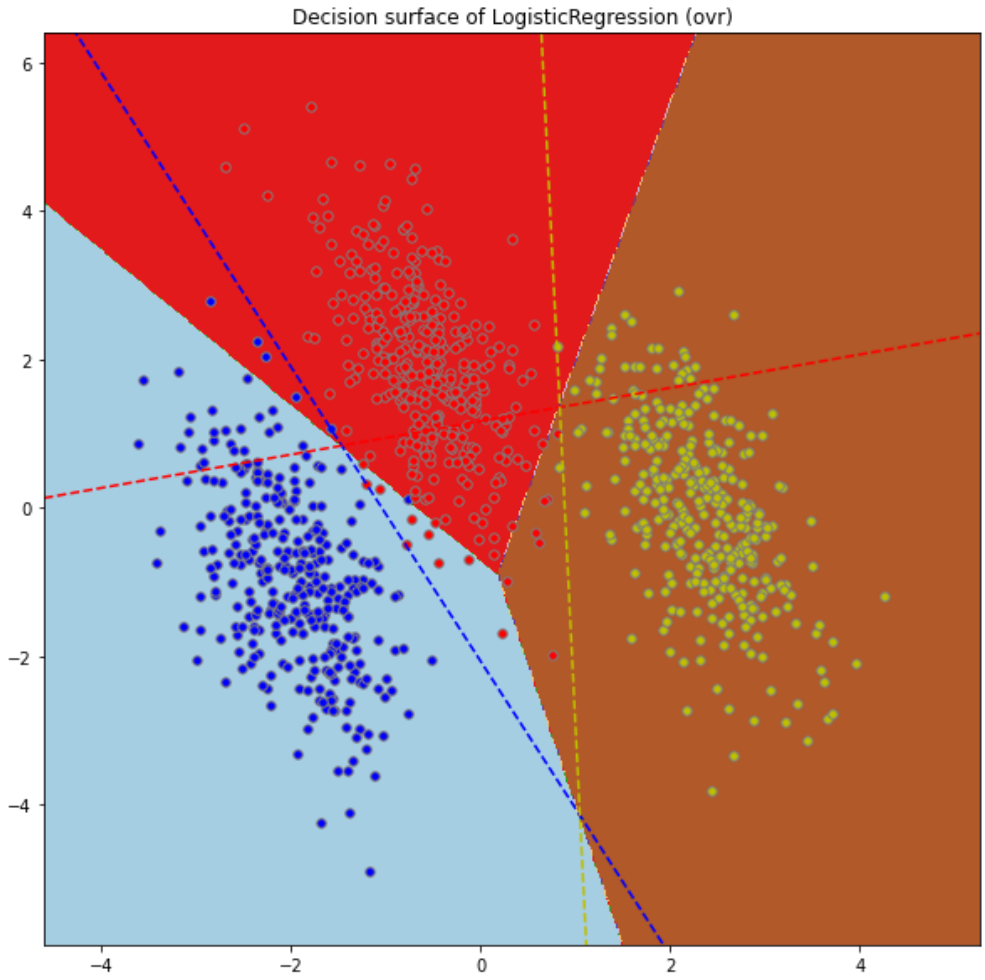
.

In [0]:
```
# make 3-class dataset for classification
centers = [[-5, 0], [0, 1.5], [5, -1]]
x, y = make_blobs(n_samples=1000, centers=centers, rando
m_state=40)
transformation = [[0.4, 0.2], [-0.4, 1.2]]
x = np.dot(x, transformation)

for multi_class in ('multinomial', 'ovr'):
    clf = linear_model.LogisticRegression(solver='sag',
max_iter=100,
                                          multi_class=multi_class)
    clf.fit(x, y)

    # print the training scores
    print("training accuracy : %.3f (%s)" % (clf.score
(x, y), multi_class))

    # create a mesh to plot in
    h = .02  # step size in the mesh
    x_min, x_max = x[:, 0].min() - 1, x[:, 0].max() + 1
    y_min, y_max = x[:, 1].min() - 1, x[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # Plot the decision boundary. For that, we will assi
gn a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    # Put the result into a color plot
    z = z.reshape(xx.shape)
    plt.figure(figsize=(10,10))
    plt.contourf(xx, yy, z, cmap=plt.cm.Paired)
    plt.title("Decision surface of LogisticRegression (%
s)" % multi_class)
    plt.axis('tight')

    # Plot also the training points
    colors = "bry"
    for i, color in zip(clf.classes_, colors):
        idx = np.where(y == i)
        plt.scatter(x[idx, 0], x[idx, 1], c=color, cmap=
plt.cm.Paired,
                    edgecolor='gray', s=30)

    # Plot the three one-against-all classifiers
    xmin, xmax = plt.xlim()
    ymin, ymax = plt.ylim()
    coef = clf.coef_
    intercept = clf.intercept_

    def plot_hyperplane(c, color):
        def line(x0):
            return (-(x0 * coef[c, 0]) - intercept[c]) /
coef[c, 1]
        plt.plot([xmin, xmax], [line(xmin), line(xmax)],
```

```
training accuracy : 0.995 (multinomial)
```




```
training accuracy : 0.976 (ovr)
```



EXERCISE 2.

In [0]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels)
= fashion_mnist.load_data()
```

We will reshape 2-d images to 1-d arrays for use in scikit-learn:

In [0]:
```
n_train = len(train_labels)
x_train = train_images.reshape((n_train, -1))
y_train = train_labels

n_test = len(test_labels)
x_test = test_images.reshape((n_test, -1))
y_test = test_labels
```

Now use a multinomial logistic regression classifier, and measure the accuracy:

In [0]:
```
# 1. Create classifier

# 2. fit the model

# 3. evaluate accuracy on train and test datasets
```

Data Science Fundamentals 5

Basic introduction on how to perform typical machine learning tasks with Python.

Prepared by Mykhailo Vladymyrov & Aris Marcolongo, Science IT Support, University Of Bern, 2020

Part 2.

In [ ]:
```
from sklearn import tree
from sklearn import ensemble

from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

from matplotlib import  pyplot as plt
from time import time as timer
from imageio import imread
import pandas as pd
import numpy as np
import os

from sklearn.manifold import TSNE
import umap

import tensorflow as tf

%matplotlib inline
from matplotlib import animation
from IPython.display import HTML
```

In [2]:
```
if not os.path.exists('data'):
    path = os.path.abspath('.')+'/colab_material.tgz'
    tf.keras.utils.get_file(path, 'https://github.com/neworldemancer/DSF
5/raw/master/colab_material.tgz')
    !tar -xvzf colab_material.tgz > /dev/null 2>&1
```

```
Downloading data from https://github.com/neworldemancer/DSF5/raw/master/c
olab_material.tgz
37371904/37370165 [==============================] - 1s 0us/step
```

In [5]:
```
from utils.routines import *
```

Datasets

In this course we will use several synthetic and real-world datasets to ilustrate the behavior of the models and excercise our skills.

1. House prices

Subset of the the hous pricess kaggle dataset: https://www.kaggle.com/c/house-prices-advanced-regression-techniques (https://www.kaggle.com/c/house-prices-advanced-regression-techniques)

In [16]:
```
def house_prices_dataset(return_df=False, price_max=400000, area_max=400
00):
  path = 'data/train.csv'
  df = pd.read_csv(path, na_values="NaN", keep_default_na=False)

  useful_fields = ['LotArea',
                   'Utilities', 'OverallQual', 'OverallCond',
                   'YearBuilt', 'YearRemodAdd', 'ExterQual', 'ExterCond',
                   'HeatingQC', 'CentralAir', 'Electrical',
                   '1stFlrSF', '2ndFlrSF','GrLivArea',
                   'FullBath', 'HalfBath',
                   'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual', 'TotRms
AbvGrd',
                   'Functional','PoolArea',
                   'YrSold', 'MoSold'
                   ]
  target_field = 'SalePrice'

  cleanup_nums = {"Street":      {"Grvl": 0, "Pave": 1},
                   "LotFrontage": {"NA":0},
                   "Alley":       {"NA":0, "Grvl": 1, "Pave": 2},
                   "LotShape":    {"IR3":0, "IR2": 1, "IR1": 2, "Reg":3},
                   "Utilities":   {"ELO":0, "NoSeWa": 1, "NoSewr": 2, "Al
lPub": 3},
                   "LandSlope":   {"Sev":0, "Mod": 1, "Gtl": 3},
                   "ExterQual":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "ExterCond":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "BsmtQual":    {"NA":0, "Po":1, "Fa": 2, "TA": 3, "G
d": 4, "Ex":5},
                   "BsmtCond":    {"NA":0, "Po":1, "Fa": 2, "TA": 3, "G
d": 4, "Ex":5},
                   "BsmtExposure":{"NA":0, "No":1, "Mn": 2, "Av": 3, "G
d": 4},
                   "BsmtFinType1":{"NA":0, "Unf":1, "LwQ": 2, "Rec": 3, "
BLQ": 4, "ALQ":5, "GLQ":6},
                   "BsmtFinType2":{"NA":0, "Unf":1, "LwQ": 2, "Rec": 3, "
BLQ": 4, "ALQ":5, "GLQ":6},
                   "HeatingQC":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "CentralAir":  {"N":0, "Y": 1},
                   "Electrical":  {"NA":0, "Mix":1, "FuseP":2, "FuseF":
3, "FuseA": 4, "SBrkr": 5},
                   "KitchenQual": {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "Functional":  {"Sal":0, "Sev":1, "Maj2": 2, "Maj1":
3, "Mod": 4, "Min2":5, "Min1":6, 'Typ':7},
                   "FireplaceQu": {"NA":0, "Po":1, "Fa": 2, "TA": 3, "G
d": 4, "Ex":5},
                   "PoolQC":      {"NA":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "Fence":       {"NA":0, "MnWw": 1, "GdWo": 2, "MnPrv":
3, "GdPrv":4},
                   }

  df_X = df[useful_fields].copy()
  df_X.replace(cleanup_nums, inplace=True)  # convert continous categori
al variables to numerical
  df_Y = df[target_field].copy()

  x = df_X.to_numpy().astype(np.float32)
  y = df_Y.to_numpy().astype(np.float32)

  if price_max>0:
    idxs = y<price_max
    x = x[idxs]
    y = y[idxs]
```

In [17]:
```
def house_prices_dataset_normed():
    x, y = house_prices_dataset(return_df=False, price_max=-1, area_max=
-1)

    scaler=StandardScaler()
    features_scaled=scaler.fit_transform(x)

    return features_scaled
```

2. Fashion MNIST

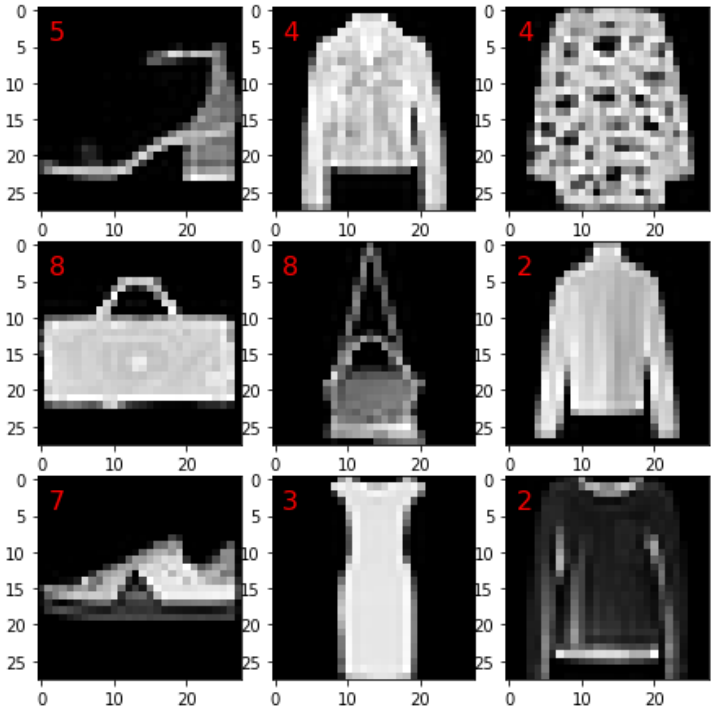
Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. (from https://github.com/zalandoresearch/fashion-mnist (https://github.com/zalandoresearch/fashion-mnist))

In [6]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnis
t.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
32768/29515 [================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

Let's chech few samples:

In [7]:
```
n = 3
fig, ax = plt.subplots(n, n, figsize=(2*n, 2*n))
ax = [ax_xy for ax_y in ax for ax_xy in ax_y]
for axi, im_idx in zip(ax, np.random.choice(len(train_images), n**2)):
  im = train_images[im_idx]
  im_class = train_labels[im_idx]
  axi.imshow(im, cmap='gray')
  axi.text(1, 4, f'{im_class}', color='r', size=16)
plt.tight_layout(0,0,0)
```



Each training and test example is assigned to one of the following labels:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

1. Trees & Forests

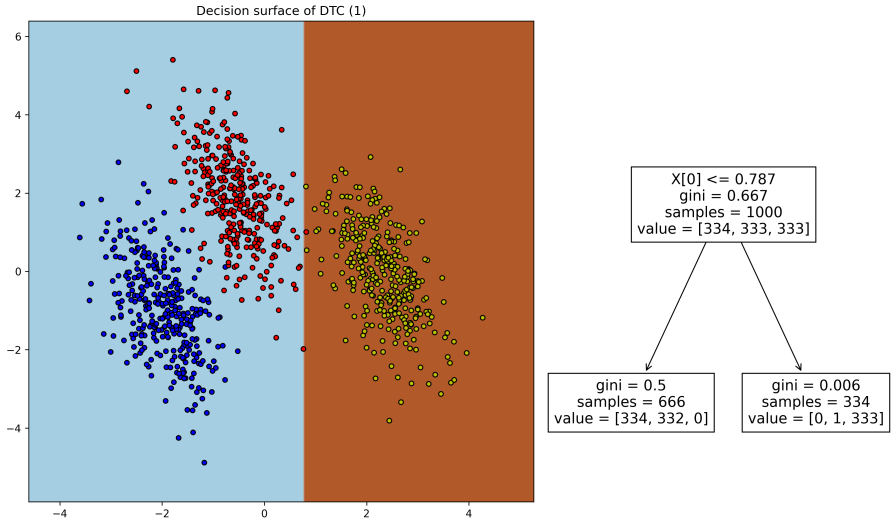
1. Decision Tree

Decision Trees are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning **simple** decision rules inferred from the data features.

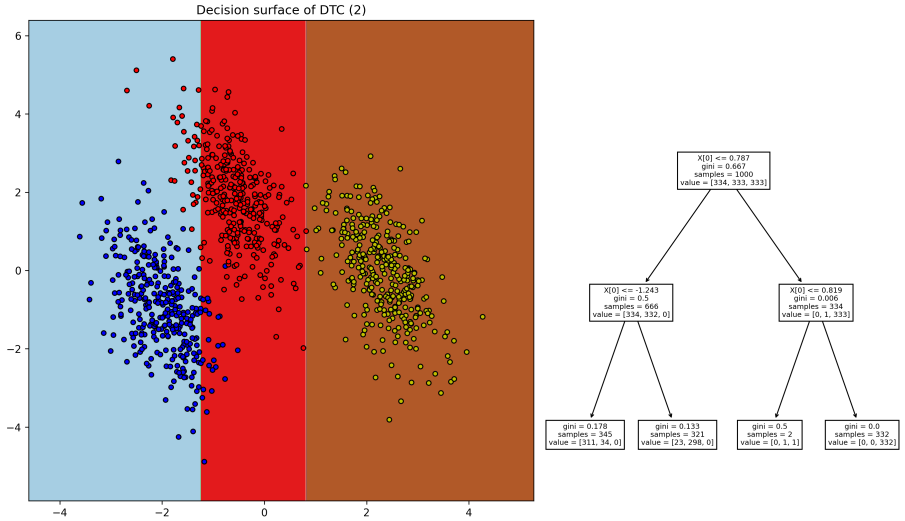
They are fast to train, easily interpretable and require small amount of data.

In [17]:
```
# make 3-class dataset for classification
centers = [[-5, 0], [0, 1.5], [5, -1]]
X, y = make_blobs(n_samples=1000, centers=centers, random_state=40)
transformation = [[0.4, 0.2], [-0.4, 1.2]]
X = np.dot(X, transformation)
for depth in (1, 2, 3, 4):
    dtc = tree.DecisionTreeClassifier(max_depth=depth)
    dtc.fit(X, y)

    # print the training scores
    print("training score : %.3f (depth=%d)" % (dtc.score(X, y), depth))

    # create a mesh to plot in
    h = .02  # step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # Plot the decision boundary. For that, we will assign a color to ea
ch
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    Z = dtc.predict(np.c_[xx.ravel(), yy.ravel()])
    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    fig, ax = plt.subplots(1, 2,  figsize=(14,7), dpi=300)
    ax[0].contourf(xx, yy, Z, cmap=plt.cm.Paired)
    ax[0].set_title("Decision surface of DTC (%d)" % depth)

    # Plot also the training points
    colors = "bry"
    for i, color in zip(dtc.classes_, colors):
        idx = np.where(y == i)
        ax[0].scatter(X[idx, 0], X[idx, 1], c=color, cmap=plt.cm.Paired,
                      edgecolor='black', s=20)

    tree.plot_tree(dtc, ax=ax[1]);

    plt.tight_layout(0.5,0)
    plt.show()
```

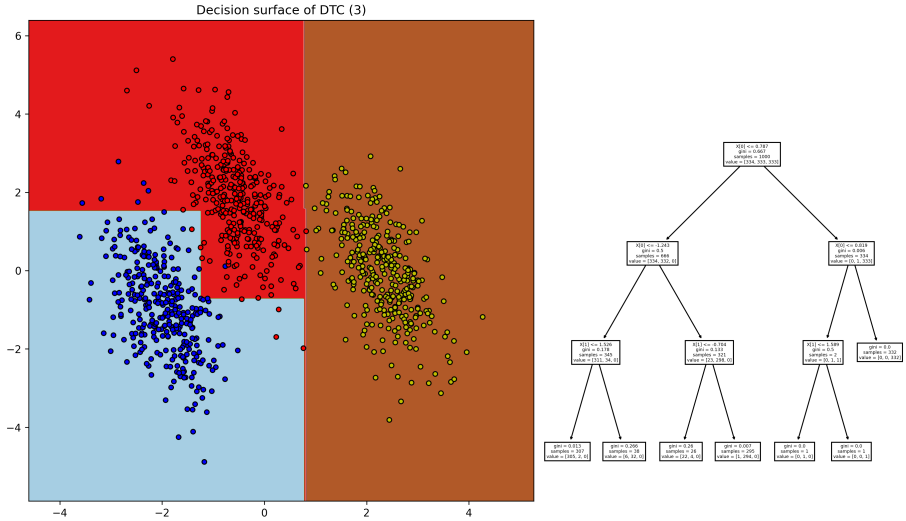
```
training score : 0.667 (depth=1)
```



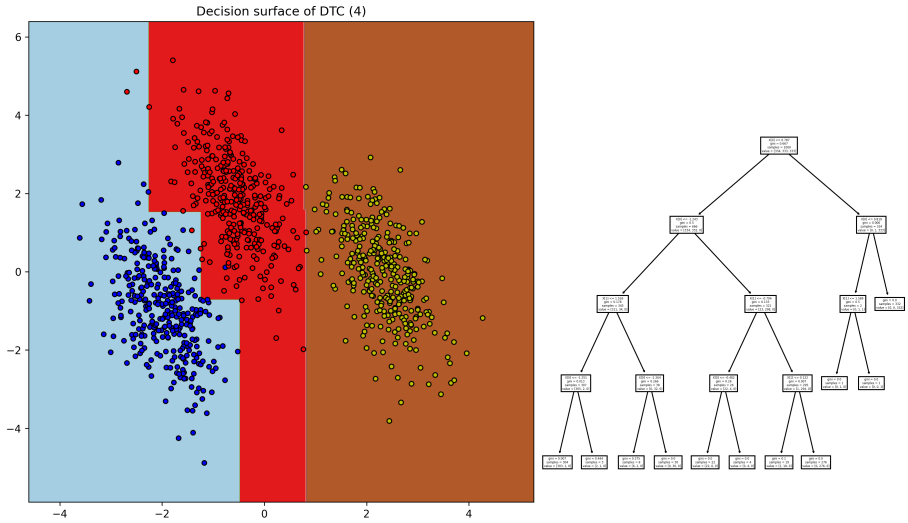

```
training score : 0.942 (depth=2)
```




```
training score : 0.987 (depth=3)
```




```
training score : 0.995 (depth=4)
```



2. Random Forest

The `sklearn.ensemble` provides sevral enasmble algorithms. RandomForest is an averaging algorithm based on randomized decision trees. This means a diverse set of classifiers is created by introducing randomness in the classifier construction. The prediction of the ensemble is given as the averaged prediction of the individual classifiers.

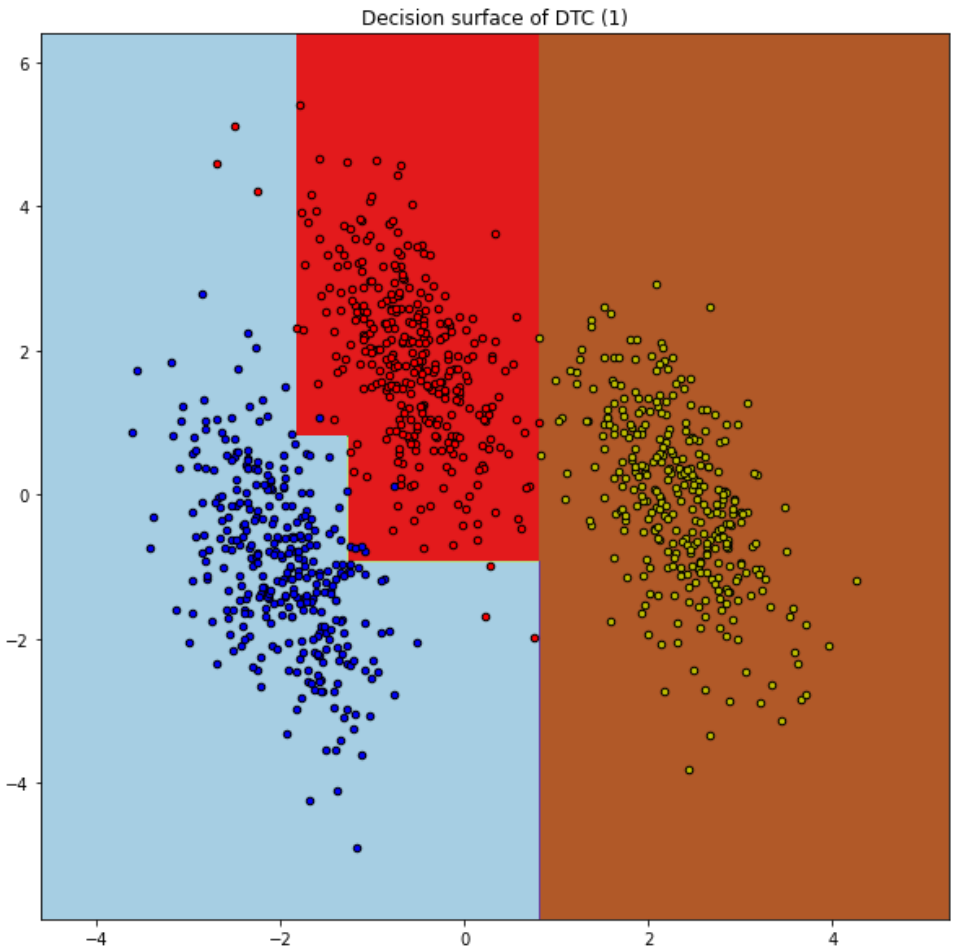
Individual decision trees typically exhibit high variance and tend to overfit. In random forests:

- each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set.
- when splitting each node during the construction of a tree, the best split is found either from all input features or a random subset.

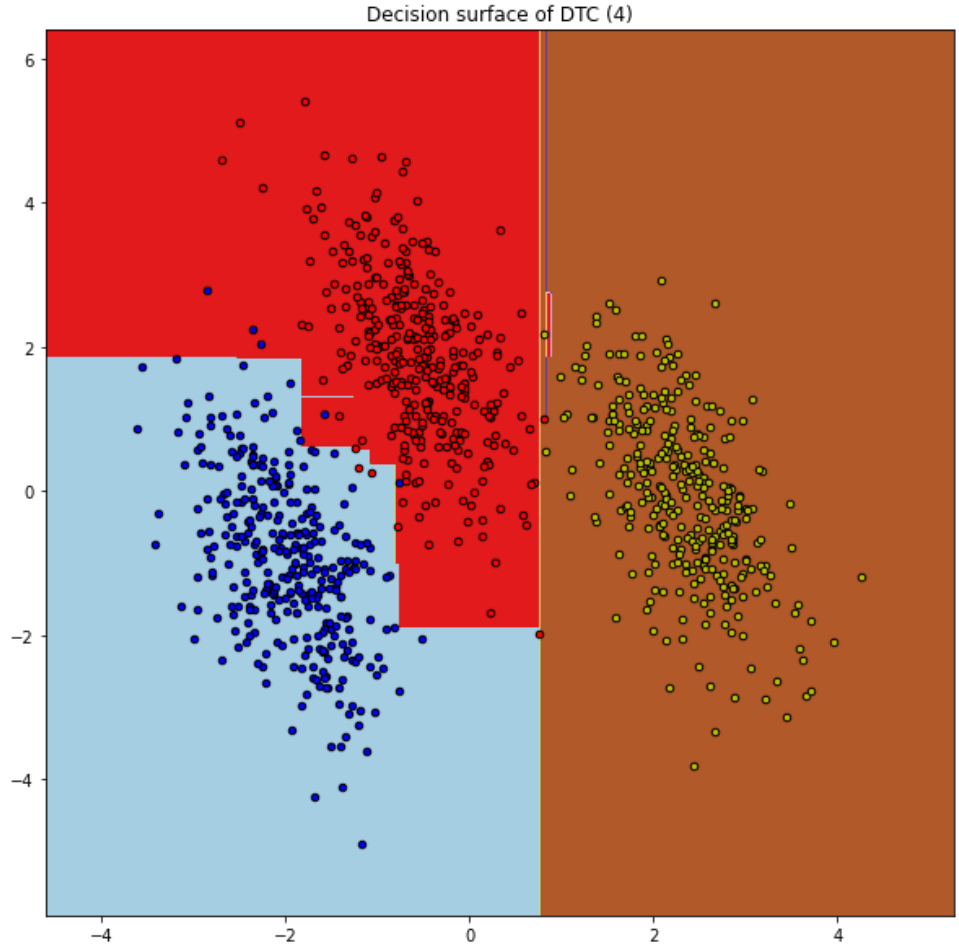
The injected randomness in forests yield decision trees with somewhat decoupled prediction errors. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice the variance reduction is often significant hence yielding an overall better model.

In [0]:
```
# make 3-class dataset for classification
centers = [[-5, 0], [0, 1.5], [5, -1]]
X, y = make_blobs(n_samples=1000, centers=centers, random_state=40)
transformation = [[0.4, 0.2], [-0.4, 1.2]]
X = np.dot(X, transformation)

for n_est in (1, 4, 50):
    dtc = ensemble.RandomForestClassifier(max_depth=4, n_estimators=n_es
t)
    dtc.fit(X, y)

    # print the training scores
    print("training score : %.3f (n_est=%d)" % (dtc.score(X, y), n_est))

    # create a mesh to plot in
    h = .02  # step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # Plot the decision boundary. For that, we will assign a color to ea
ch
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    Z = dtc.predict(np.c_[xx.ravel(), yy.ravel()])
    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(10,10))
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
    plt.title("Decision surface of DTC (%d)" % n_est)
    plt.axis('tight')

    # Plot also the training points
    colors = "bry"
    for i, color in enumerate(colors):
        idx = np.where(y == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=color, cmap=plt.cm.Paired,
                    edgecolor='black', s=20)

    # Plot the three one-against-all classifiers
    xmin, xmax = plt.xlim()
    ymin, ymax = plt.ylim()


    plt.show()
```

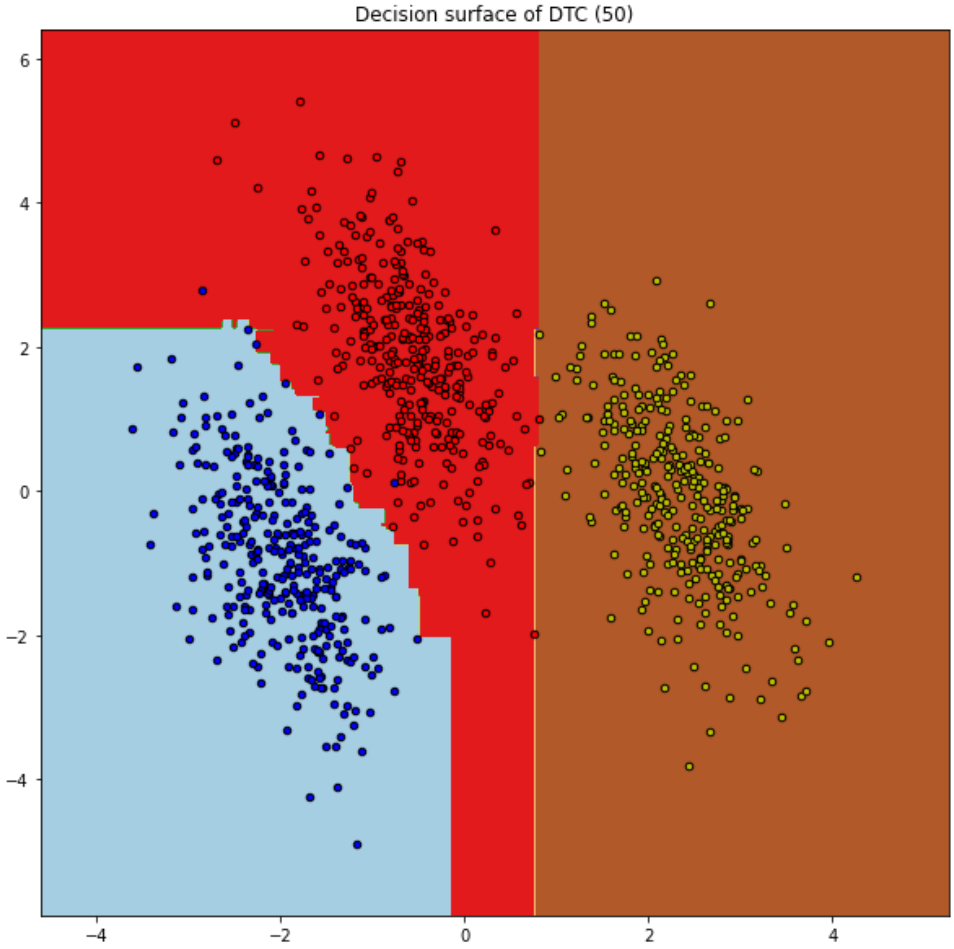
```
training score : 0.987 (n_est=1)
```



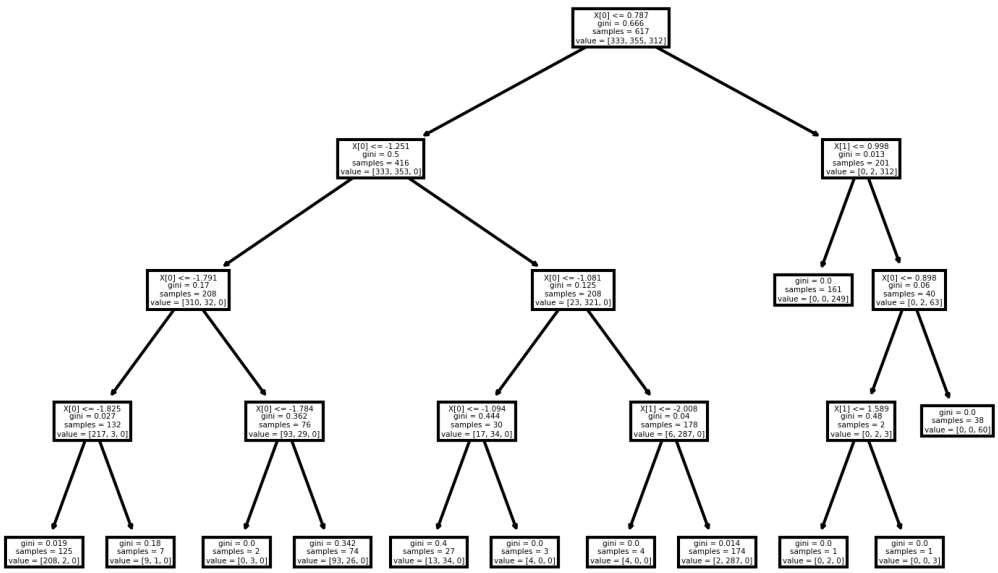

```
training score : 0.992 (n_est=4)
```




```
training score : 0.998 (n_est=50)
```



In [0]:
```
plt.figure(dpi=300)
tree.plot_tree(dtc.estimators_[20]);
```



3. Boosted Decision Trees

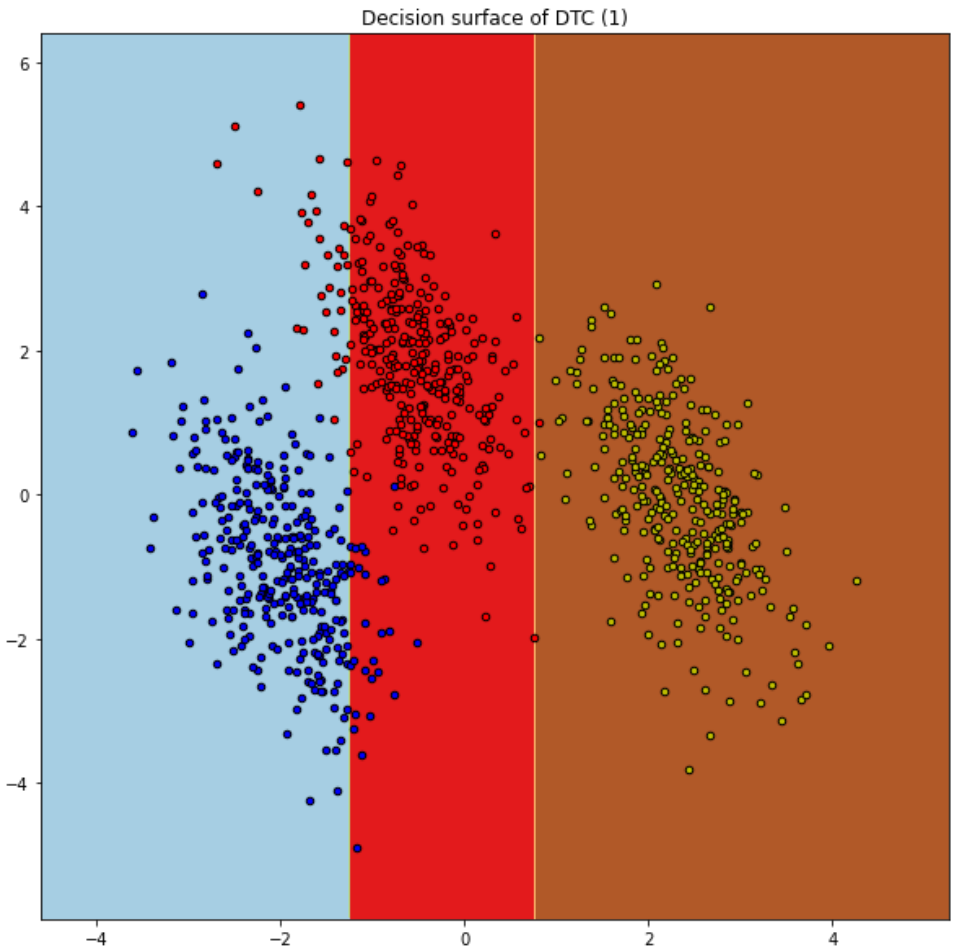
Anouther approach to the ensemble tree modeling is Boosted Decision Trees.

Gradient Tree Boosting or Gradient Boosted Decision Trees (GBDT) is a generalization of boosting to arbitrary differentiable loss functions. GBDT is an accurate and effective off-the-shelf procedure that can be used for both regression and classification problems in a variety of areas including Web search ranking and ecology.

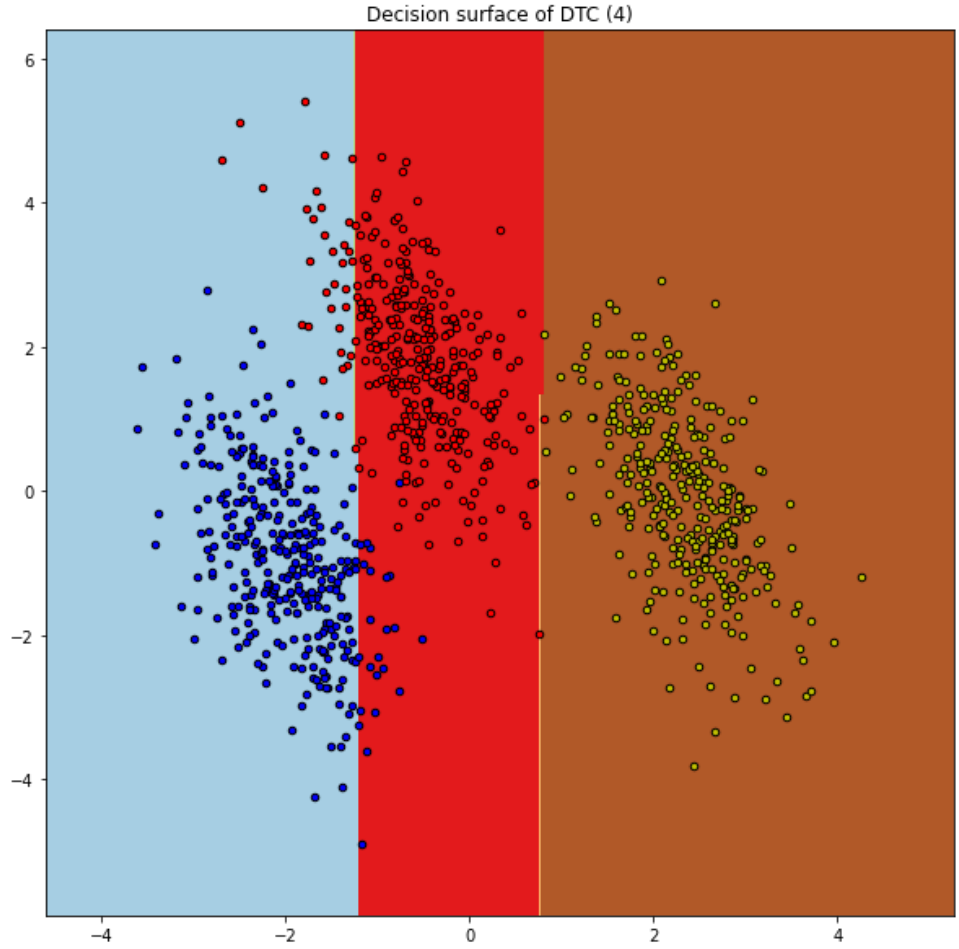
Boosting is based on weak learners, i.e. shallow trees. In boosting primarily the bias is reduced.

In [0]:
```
# make 3-class dataset for classification
centers = [[-5, 0], [0, 1.5], [5, -1]]
X, y = make_blobs(n_samples=1000, centers=centers, random_state=40)
transformation = [[0.4, 0.2], [-0.4, 1.2]]
X = np.dot(X, transformation)

for n_est in (1, 4, 50):
    dtc = ensemble.GradientBoostingClassifier(max_depth=1, n_estimators=
n_est)
    dtc.fit(X, y)

    # print the training scores
    print("training score : %.3f (n_est=%d)" % (dtc.score(X, y), n_est))

    # create a mesh to plot in
    h = .02  # step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    # Plot the decision boundary. For that, we will assign a color to ea
ch
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    Z = dtc.predict(np.c_[xx.ravel(), yy.ravel()])
    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(10,10))
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
    plt.title("Decision surface of DTC (%d)" % n_est)
    plt.axis('tight')

    # Plot also the training points
    colors = "bry"
    for i, color in enumerate(colors):
        idx = np.where(y == i)
        plt.scatter(X[idx, 0], X[idx, 1], c=color, cmap=plt.cm.Paired,
                    edgecolor='black', s=20)

    # Plot the three one-against-all classifiers
    xmin, xmax = plt.xlim()
    ymin, ymax = plt.ylim()


    plt.show()
```

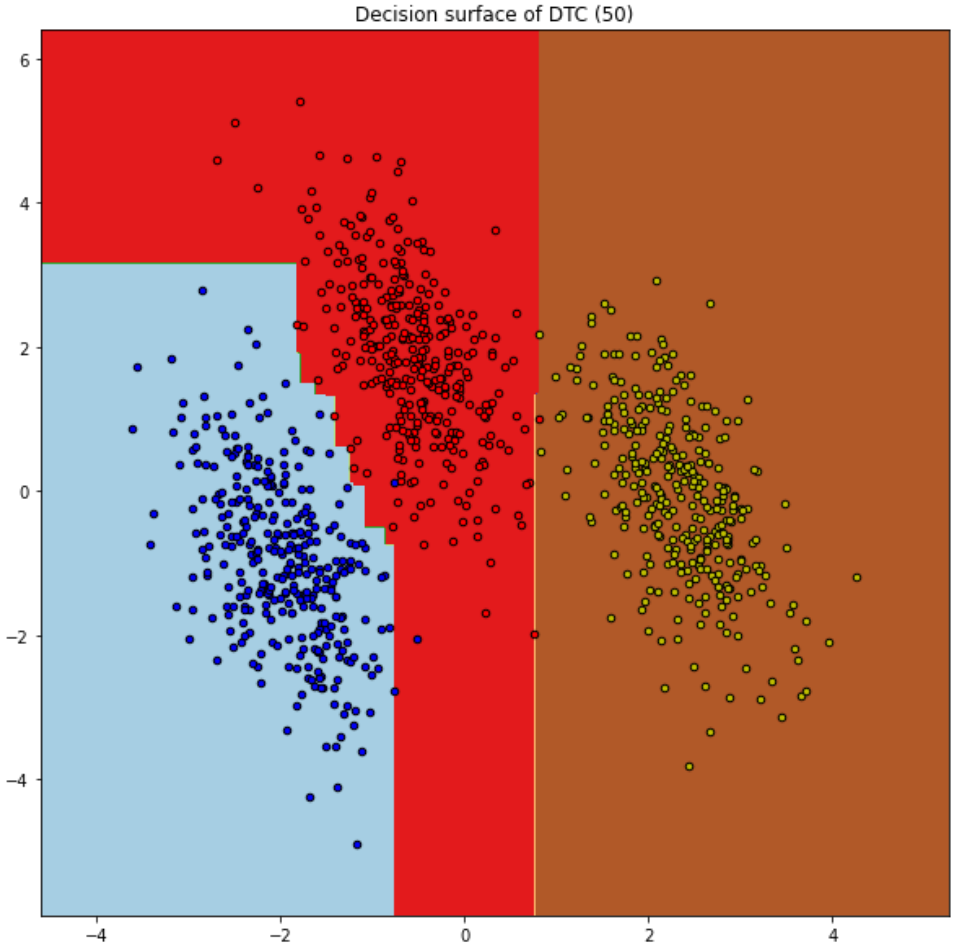
```
training score : 0.942 (n_est=1)
```




```
training score : 0.944 (n_est=4)
```




```
training score : 0.994 (n_est=50)
```



EXERCISE 1 : Random forest classifier for FMNIST

In [0]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnis
t.load_data()

n = len(train_labels)
x_train = train_images.reshape((n, -1))
y_train = train_labels

n_test = len(test_labels)
x_test = test_images.reshape((n_test, -1))
y_test = test_labels
```

Classify fashion MNIST images with Random Forest classifier.

In [0]:
```
# 1. Create classifier. As the number of features is big, use bigger tre
e depth
# (max_depth parameter). in the same time to reduce variance, one should
limit the
# total number of tree leafes. (max_leaf_nodes parameter)
# Try different number of estimators (n_estimators)

# 2. fit the model
# 3. Inspect training and test accuracy
```

2. Unsupervised Learning Techniques

1. Principal Component Analysis (PCA)

Theory overview.

Objective: PCA is used for dimensionality reduction when we have a large number D of features with non-trivial intercorrelation (data redundance) and to isolate relevant features.

PCA provides a new set of uncorrelated M features for every data point, with $M \leq D$. The new features are:

- a linear combination of the original ones ;
- uncorrelated between each other ;

If $M \ll D$ we get an effective dimensionality reduction.

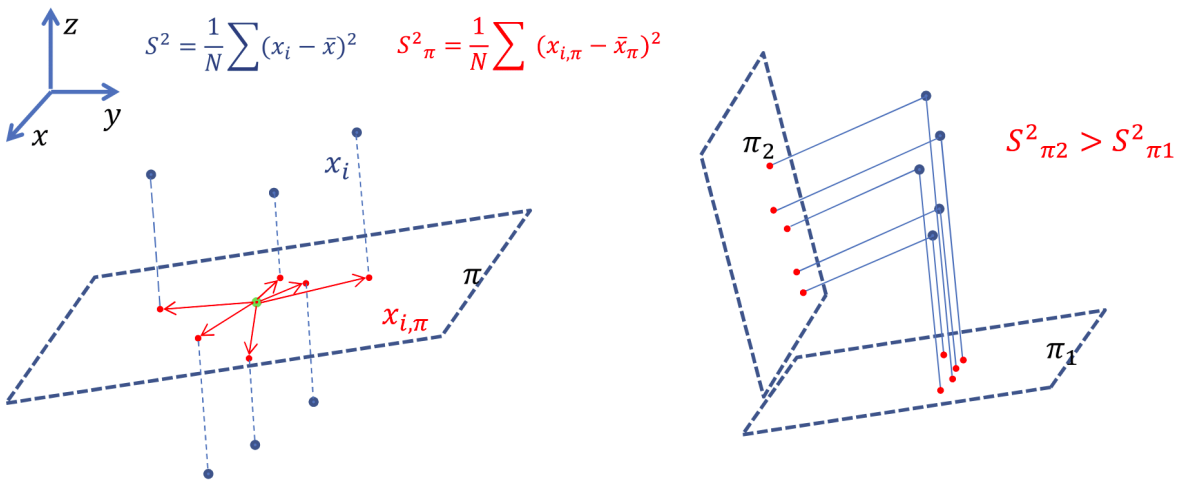
Methods: Each data point indexed by $p = 1..N$ can be seen as an element $\mathbf{x}_p \in \mathbf{R}^D$.

The variance of the data-cloud measures the spread around its centroid:

$$S^2 = \frac{1}{N}\sum_{p=1}^{N}(\mathbf{x}_p - \overline{\mathbf{x}})^2$$

$$\overline{\mathbf{x}} = \frac{1}{N}\sum_{p=1}^{N}\mathbf{x}_p$$

We fix a number $1 \leq k \leq D$ and consider a subspace V_k of dimension k. Each data point $\mathbf{x}_p$ is projected onto V_k, leading to points $\mathbf{x}_p^k$ with spread $S^{2,k}$. PCA chooses V_k such that the variance $S^{2,k}$ is maximized, as shown in the picture.



Terminology and output of a PCA computation:

- *Principal components*: A sequence of orthonormal vectors $k_1, .., k_n$ spanning optimal subspaces: $\mathrm{Span}\{k_1, .., k_m\} = V_m$;
- *Scores*: For every sample-point p. the new features are called scores are given by the component of p along the k vectors;
- *Reconstructed vector:* For every k, the projection of V on V_k ;
- *Explained variance*: For every k, the ratio between the variance of the reconstructed vectors and and total variance. The number of components is chosen selecting an optimal k. The plot of the explained variance as a function of k is called a *scree plot*

Sklearn: implementation and usage of PCA.

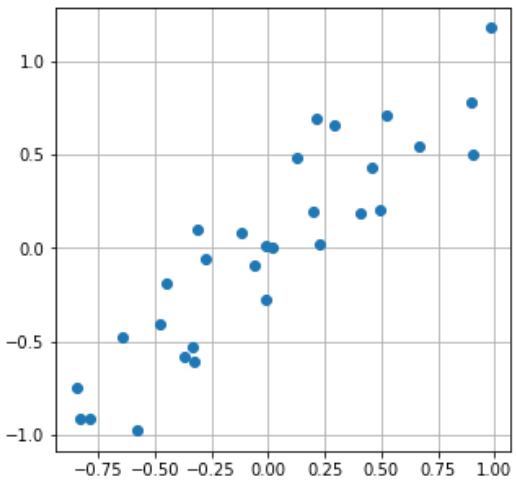
We start showing a two dimensional example that can be easy visualized.

We load the data-sets that we are going to use for the examples:

In [0]:
```
data=load_sample_data_pca()

n_samples,n_dim=data.shape

print('We have ',n_samples, 'samples of dimension ', n_dim)

plt.figure(figsize=((5,5)))
plt.grid()
plt.plot(data[:,0],data[:,1],'o')
```

```
We have  30 samples of dimension  2
```

Out[0]:
```
[<matplotlib.lines.Line2D at 0x7f6b3fed36d8>]
```



The data set is almost one dimensional. PCA will confirm this result.

As with most of sklearn functionalities, we need first to create a PCA object. We will use the object methods to perform PCA.

In [0]:
```
pca=PCA(n_components=2)
```

A call to the pca.fit method computes the principal components

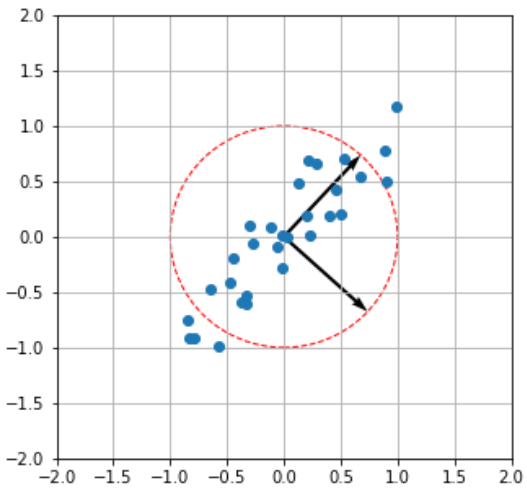
In [0]:
```
pca.fit(data)
```

Out[0]:
```
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)
```

Now the the pca.components_ attribute contains the principal components. We can print them alongside with the data and check that they consitute an orthonormal bases.

In [0]:
```
plt.figure(figsize=((5,5)))
plt.grid()
plt.plot(data[:,0],data[:,1],'o')

circle=plt.Circle((0, 0), 1.0, linestyle='--', color='red',fill=False)
ax=plt.gca()
ax.add_artist(circle)

for vec in pca.components_:
    plt.quiver([0], [0], [vec[0]], [vec[1]], angles='xy', scale_units='x
y', scale=1)

plt.xlim(-2,2)
plt.ylim(-2,2)
```

Out[0]:
```
(-2.0, 2.0)
```



The pca.explained_variance*ratio* attribute containes the explained variance. In this case we see that already the first reconstructed vector explains 95% of the variance.

In [0]:
```
print(pca.explained_variance_ratio_)
```

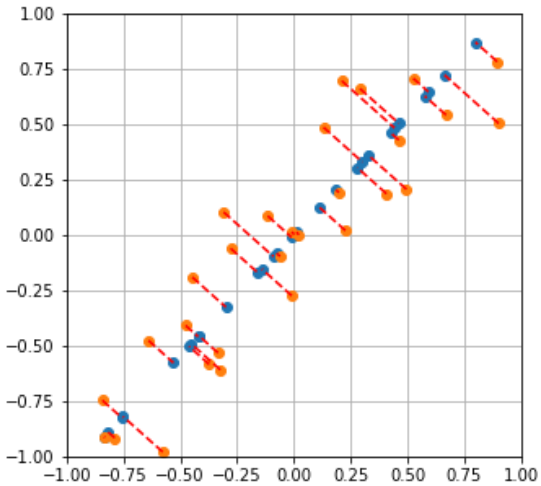
```
[0.95140729 0.04859271]
```

To compute the reconstructed vectors for k=1 we first need to compute the scores and than multiply by the basis vectors:

In [0]:
```
k=1
scores=pca.transform(data)
res=np.dot(scores[:,:k], pca.components_[:k,:] )
```

In [0]:
```
plt.figure(figsize=((5,5)))
plt.plot(res[:,0],res[:,1],'o')
plt.plot(data[:,0],data[:,1],'o')

for a,b,c,d in zip(data[:,0],data[:,1],res[:,0],res[:,1]) :
    plt.plot([a,c],[b,d],'-', linestyle = '--', color='red')

plt.grid()

plt.xlim(-1.0,1.0)
plt.ylim(-1.0,1.0)
```

Out[0]: (-1.0, 1.0)

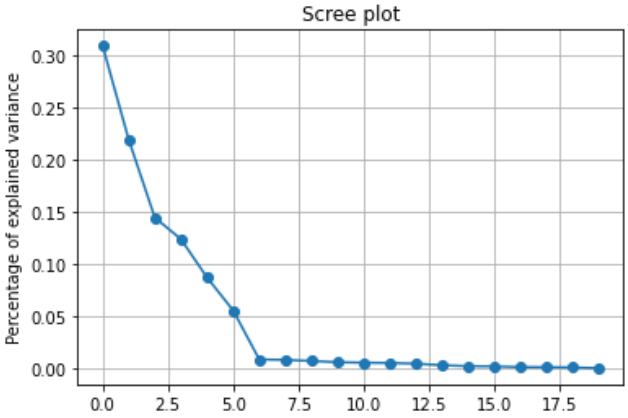


The same procedure is followed for high dimensional data-sets. Here we generate random data which lies almost on a 6 dimensional subspace. The resulting screen plot can be used to find this result in a semi-automaic fashion.

In [0]:
```
high_dim_dataset=load_multidimensional_data_pca(n_data=40 ,n_vec=6, dim=
20, eps= 0.5)
n_samples,n_dim=high_dim_dataset.shape

print('We have ',n_samples, 'samples of dimension ', n_dim)
```

```
We have  40 samples of dimension  20
```

In [0]:
```
pca=PCA()
pca.fit(high_dim_dataset)
plt.plot(pca.explained_variance_ratio_,'-o')
plt.title('Scree plot')
plt.ylabel('Percentage of explained variance')
plt.grid()
```



As an exercise, you can change the value of eps and see how the screen plot changes.

EXERCISE 2 : PCA with a non-linear data-set

In [0]:
```
### In this example you will try to do PCA with the simplest non-linear
function, a parabola.

# 1. Load the data using the function data=load_ex1_data_pca() , check t
he dimensionality of the data and plot them.

# 2. Define a PCA object and perform the PCA fitting.

# 3. Check the explained variance ratio and select best number of compon
ents.

# 4. Plot the reconstructed vectors for different values of k.
```

EXERCISE 3 : Find the hidden drawing.

In [0]:
```
### In this exercise you will take a high dimensional data-set, find the
optimal number of principal components
# and visualize the reconstructed vectors with k=2. The pipeline is the
same as Ex. 2.

# 1. Load the data using the function data=load_ex2_data_pca(seed=1235)
, check the dimensionality of the data and plot them.

# 2. Define a PCA object and perform the PCA fitting.

# 3. Check the explained variance ratio and select best number of compon
ents.

# 4. Plot the reconstructed vectors for the best value of k.
```

2. Data visualization (t-SNE / UMAP)

Theory overview

PCA is a linear embedding technique where the scores are a linear function of the original variables. This forces the number of principal components to be used to be high, if the manyfold is highly non-linear. Curved manyfolds needs to be embedded in higher dimensions.

Other non-linear embedding techniques consider more generic embeddings and try to minimize the loss of information according to different criteria, either statistical (t-SNE) or based on advanced topological descriptions (UMAP). It is not the goal of this short introduction to discuss the derivation of such approaches.

In the following, we will show how to apply practically these dimensionality reductions techniques. Keep in mind that the embedding is given by an iterative solution of a minimizazion problem and therefore the results may depend on the value of the random seed.

Utilization in Python and examples

To begin with, we create a t-SNE object that we are going to use.

In [3]:
```
tsne_model = TSNE(perplexity=30, n_components=2, learning_rate=200, earl
y_exaggeration=4.0,init='pca',
                      n_iter=2000, random_state=2233212, metric='euclide
an', verbose=100 )

umap_model = umap.UMAP(n_neighbors=30, n_components=2, random_state=171
1)
```

Example 1: Exercise 3 Cont'd

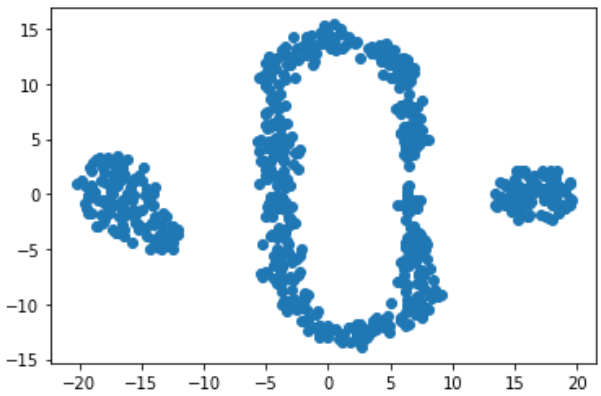
We first of all visualize our multi-dimensional heart using t-sne:

In [6]:
```
data= load_ex2_data_pca(seed=1235, n_add=20)

tsne_model = TSNE(perplexity=30, n_components=2, learning_rate=200, earl
y_exaggeration=4.0,init='pca',
                      n_iter=300, random_state=2233212, metric='euclidea
n', verbose=100 )

tsne_heart = tsne_model.fit_transform(data)
```

```
[t-SNE] Computing 91 nearest neighbors...
[t-SNE] Indexed 651 samples in 0.001s...
[t-SNE] Computed neighbors for 651 samples in 0.012s...
[t-SNE] Computed conditional probabilities for sample 651 / 651
[t-SNE] Mean sigma: 0.060286
[t-SNE] Computed conditional probabilities in 0.026s
[t-SNE] Iteration 50: error = 10.6543207, gradient norm = 0.0495624 (50 i
terations in 1.014s)
[t-SNE] Iteration 100: error = 10.5104313, gradient norm = 0.0211065 (50
iterations in 1.044s)
[t-SNE] Iteration 150: error = 10.4393501, gradient norm = 0.0135420 (50
iterations in 0.903s)
[t-SNE] Iteration 200: error = 10.3963175, gradient norm = 0.0162105 (50
iterations in 0.934s)
[t-SNE] Iteration 250: error = 10.3679504, gradient norm = 0.0112098 (50
iterations in 0.996s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 10.36
7950
[t-SNE] Iteration 300: error = 0.6041825, gradient norm = 0.0007995 (50 i
terations in 0.748s)
[t-SNE] KL divergence after 300 iterations: 0.604182
```

In [7]:
```
plt.scatter(tsne_heart[:,0],tsne_heart[:,1])
```

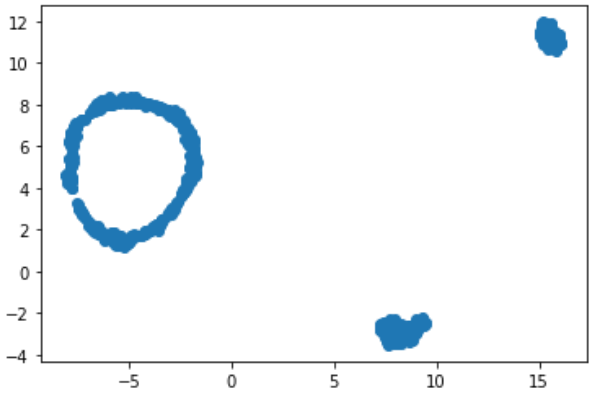
Out[7]:
```
<matplotlib.collections.PathCollection at 0x2a483798d30>
```



And using UMAP :

In [9]:
```
umap_model = umap.UMAP(n_neighbors=30, n_components=2, random_state=171
1)

umap_hart = umap_model.fit_transform(data)
plt.scatter(umap_hart[:, 0], umap_hart[:, 1])
```

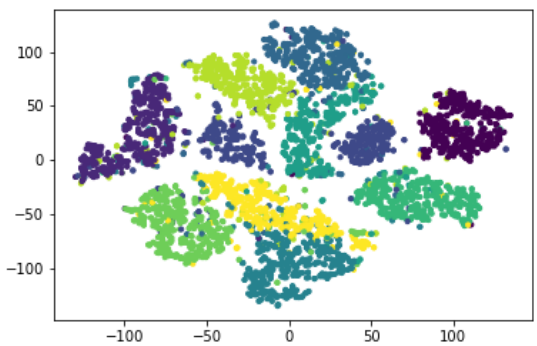
Out[9]: <matplotlib.collections.PathCollection at 0x2a485bb0a90>



Example 2: Mnist dataset

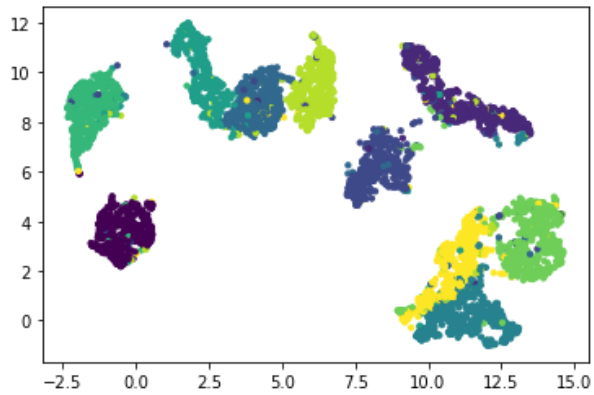
In [25]:
```
mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_da
ta()

n_examples = 5000
data=train_images[:n_examples,:].reshape(n_examples,-1)
data=data/255

labels=train_labels[:n_examples]
```

In [0]:
```
# not to run on COLAB

# tsne_model = TSNE(perplexity=10, n_components=2, learning_rate=200,
#                   early_exaggeration=4.0,init='pca',
#                   n_iter=2000, random_state=2233212,
#                   metric='euclidean', verbose=100, n_jobs=1)

# tsne_mnist = tsne_model.fit_transform(data)

# plt.scatter(tsne_mnist[:,0],tsne_mnist[:,1],c=labels,s=10)
```



In [26]:
```
umap_model = umap.UMAP(n_neighbors=10, n_components=2, random_state=171
1)
umap_mnist = umap_model.fit_transform(data)
plt.scatter(umap_mnist[:, 0], umap_mnist[:, 1], c=labels, s=10)
```

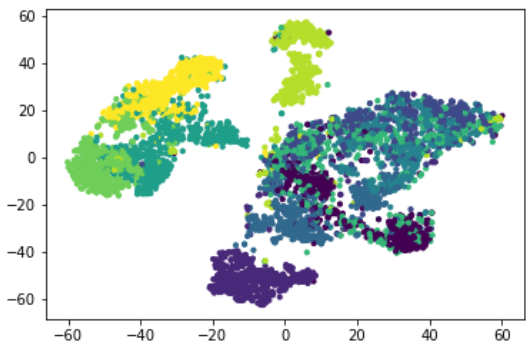
Out[26]: <matplotlib.collections.PathCollection at 0x2a4840109e8>



Example 3: Fashion_Mnist dataset

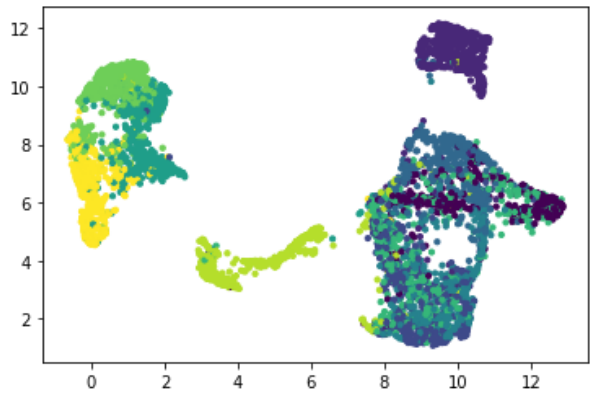
In [20]:
```
fmnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fmnist.load_d
ata()

n_examples = 5000
data=train_images[:n_examples,:].reshape(n_examples,-1)
data=data/255

labels=train_labels[:n_examples]
```

In [21]:
```
# not to run on COLAB

# tsne_model = TSNE(perplexity=50, n_components=2, learning_rate=200, ea
rly_exaggeration=4.0,init='pca',
#                       n_iter=1000, random_state=2233212, metric='euclid
ean', verbose=100 )

# tsne_fmnist = tsne_model.fit_transform(data)

# plt.scatter(tsne_fmnist[:,0],tsne_fmnist[:,1],c=labels,s=10)
```



In [23]:
```
umap_model = umap.UMAP(n_neighbors=50, n_components=2, random_state=171
1)
umap_fmnist = umap_model.fit_transform(data)
plt.scatter(umap_fmnist[:, 0], umap_fmnist[:, 1], c=labels, s=10)
```

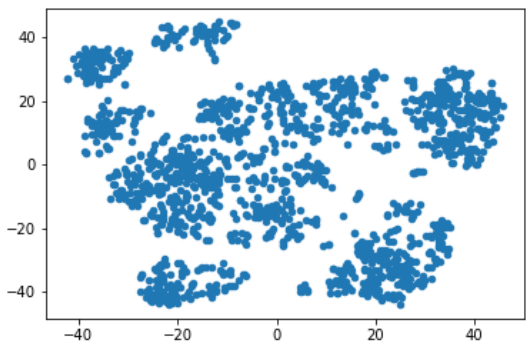
Out[23]: <matplotlib.collections.PathCollection at 0x2a490ad1cf8>



Example 4: House prices

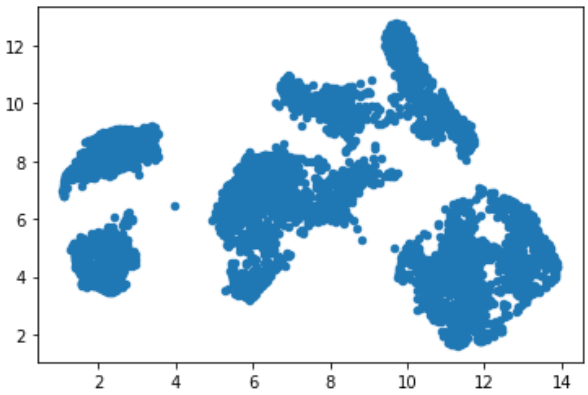
In [18]:
```
data=house_prices_dataset_normed()
```

In [0]:
```
# not to run on COLAB

#tsne_model = TSNE(perplexity=30, n_components=2, learning_rate=200,
#                  early_exaggeration=4.0,init='pca', n_iter=1000,
#                  random_state=2233212, metric='euclidean', verbose=10
0)

#tsne_houses = tsne_model.fit_transform(data)

#plt.scatter(tsne_houses[:,0],tsne_houses[:,1],s=20)
#plt.savefig('t_sne_houses.png')
```



In [27]:
```
umap_model = umap.UMAP(n_neighbors=30, n_components=2, random_state=171
1)
umap_houses = umap_model.fit_transform(data)
plt.scatter(umap_houses[:, 0], umap_houses[:, 1], s=20)
```

Out[27]: <matplotlib.collections.PathCollection at 0x2a490d4e518>



Message: Visualization techniques are useful for having an initial grasp of multi-dimensional datasets and guide further analsis and the choice of the modelling data strategy.

Data Science Fundamentals 5

Basic introduction on how to perform typical machine learning tasks with Python.

Prepared by Mykhailo Vladymyrov & Aris Marcolongo, Science IT Support, University Of Bern, 2020

Part 3.

In [0]:
```
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.mixture import GaussianMixture

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

from matplotlib import  pyplot as plt
import numpy as np
import pandas as pd
from imageio import imread
from time import time as timer
import os

import tensorflow as tf

%matplotlib inline
from matplotlib import animation
from IPython.display import HTML

import umap
from scipy.stats import entropy
```

In [2]:
```
if not os.path.exists('data'):
    path = os.path.abspath('.')+'/colab_material.tgz'
    tf.keras.utils.get_file(path, 'https://github.com/neworldemancer/DSF
5/raw/master/colab_material.tgz')
    !tar -xvzf colab_material.tgz > /dev/null 2>&1
```

```
Downloading data from https://github.com/neworldemancer/DSF5/raw/master/c
olab_material.tgz
98304/96847 [==============================] - 0s 0us/step
```

In [0]:
```
from utils.routines import *
```

1. Clustering

1. K-Means

Theory overview.

Objective: clustering techniques divide the set of data into group of atoms having common features. Each data point p gets assigned a label $l_p \in \{1, .., K\}$. In this presentation the data points are supposed to have D features, i.e. each data point belongs to $\mathbf{R}^D$.

Methods: We call P_k the subset of the data set which gets assigned to class k. K-means aims at minimizing the objective function:

$$L = \sum_k L_k$$

$$L_k = \frac{1}{|P_k|} \sum_{p,p' \in L_k} |\mathbf{x}_p - \mathbf{x}_{p'}|^2$$

One could enumerate all possibilities. The Llyod algorithm is iterative:

- start with an initial guess of the assignements ;
- compute the centroid $\mathbf{c}_k$ for every cluster, defined as:

$$\mathbf{c}_k = \frac{1}{|P_k|} \sum_{p \in L_k} \mathbf{x_p}$$

- re-assign each data point to the class of the nearest centroid
- re-compute the centroids and iterate till convergence

The Lloyd algorithm finds local minima and may need to be started several times with different initializations.

Terminology and output of a K-means computation:

- *Within-cluster variation* : L_k is called within cluster variation. It can be shown that L_k can be interpreted as the sum os squared variation with respect to the centroid
- *Silhouette score*: K-means clustering fixes the number of clusters a priori. Some tecnhique must be chosen to score the different optimal clusterings for different k. One technique chooses the best *Silouhette score*

Sklearn: implementation and usage of K-means.

We start with a 2D example that can be visualized.

First we load the data-set.

In [0]:
```
points=km_load_th1()
```

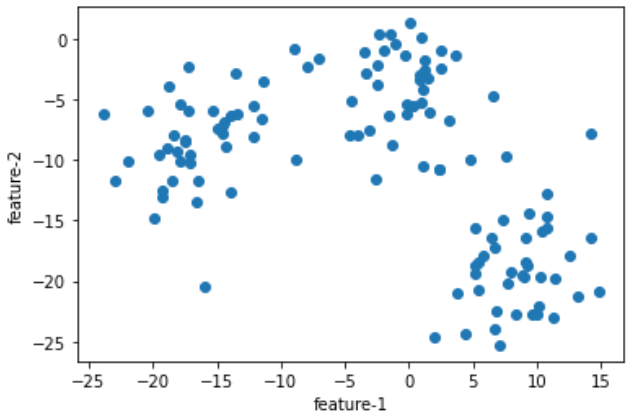
Explore the data-set checking the dataset dimensionality.

In [5]:
```
print(points.shape)
print('We have ', points.shape[0], 'points with two features')
```

```
(120, 2)
We have  120 points with two features
```

In [6]:
```
plt.plot(points[:,0],points[:,1],'o')
plt.xlabel('feature-1')
plt.ylabel('feature-2')
```

Out[6]: Text(0, 0.5, 'feature-2')



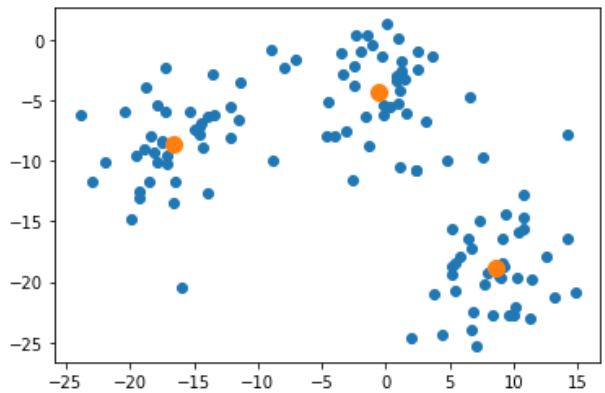
It looks visually that the data set has three clusters. We will cluster them using K-means. As usual, we create a KMeans object. Note that we do not need to initialize it with a data-set.

In [0]:
```
clusterer = KMeans(n_clusters=3, random_state=10)
```

A call to the fit method computes the cluster centers which can be plotted alongside the data-set. They are accessible from the cluster*centers* attribute:

In [8]:
```
clusterer.fit(points)
plt.plot(points[:,0],points[:,1],'o')
plt.plot(clusterer.cluster_centers_[:,0],clusterer.cluster_centers_[:,
1],'o',markersize=10)
```

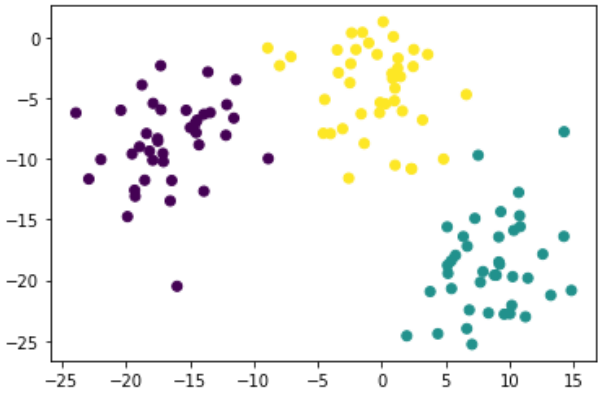
Out[8]: [<matplotlib.lines.Line2D at 0x7fa6eba4e1d0>]



The predict method assigns a new point to the nearest cluster. We can use predict with the training dataset and color the data-set according to the cluster label.

In [9]:
```
cluster_labels=clusterer.predict(points)
plt.scatter(points[:,0],points[:,1],c=cluster_labels)
```

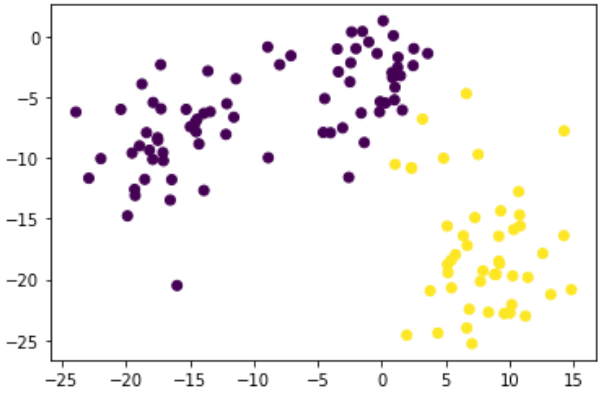
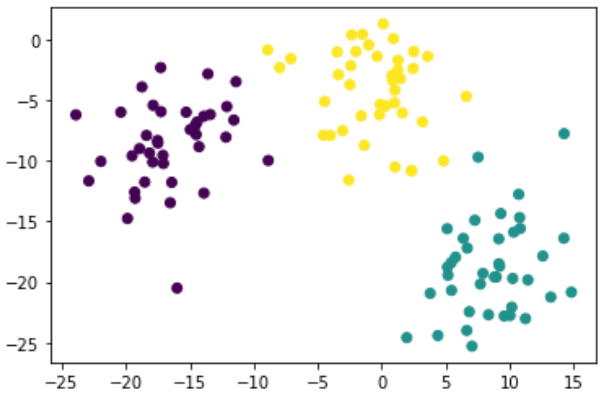
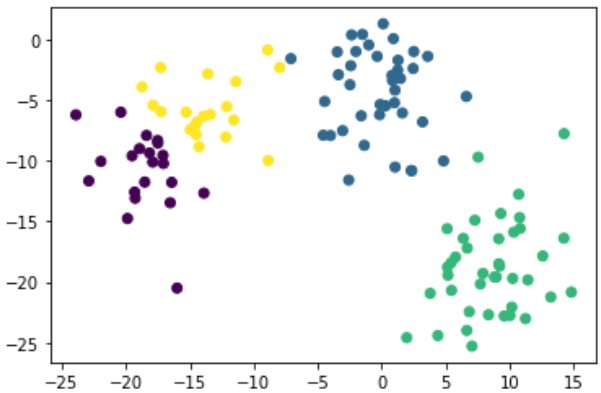
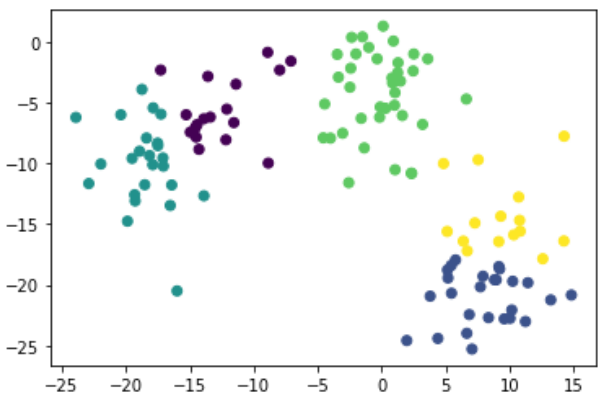
Out[9]: <matplotlib.collections.PathCollection at 0x7fa680121828>

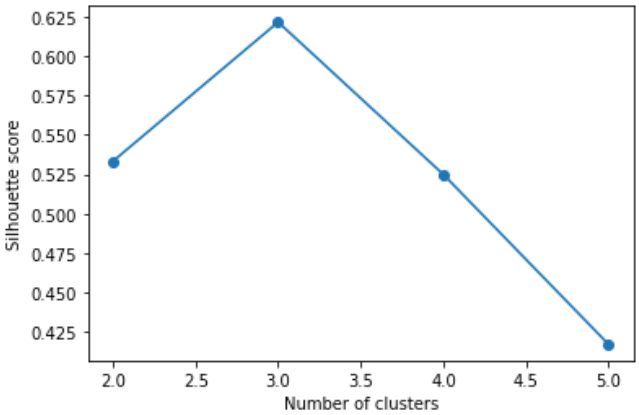


Finally, we can try to vary the number of clusters and score them with the Silhouette score.

In [10]:
```
sil=[]

for iclust in range(2,6):
    clusterer = KMeans(n_clusters=iclust, random_state=10)
    cluster_labels = clusterer.fit_predict(points)
    score=silhouette_score(points,cluster_labels)
    sil.append(score)
    plt.figure()
    plt.scatter(points[:,0],points[:,1],c=cluster_labels)

plt.figure()
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette score')
plt.plot(np.arange(len(sil))+2, sil,'-o')
```

```
Out[10]: [<matplotlib.lines.Line2D at 0x7fa6812c2ac8>]
```



The same techniques can be used on high dimensional data-sets. We use here the famoust MNIST dataset for integer digits, that we are downloading from tensorflow.

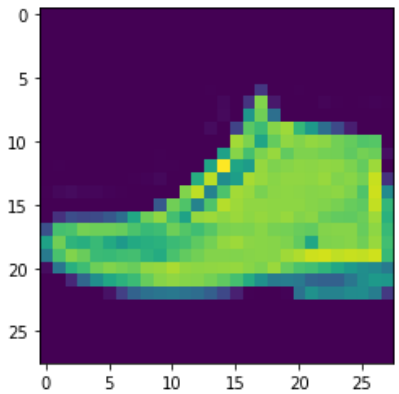
In [11]:
```
fmnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fmnist.load_d
ata()

X=train_images[:5000,:].reshape(5000,-1)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
32768/29515 [================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

In [12]:
```
print(X.shape)
image=X[1232,:].reshape(28,28)
plt.imshow(image)
```

```
(5000, 784)
```

Out[12]:
```
<matplotlib.image.AxesImage at 0x7fa68175f9b0>
```

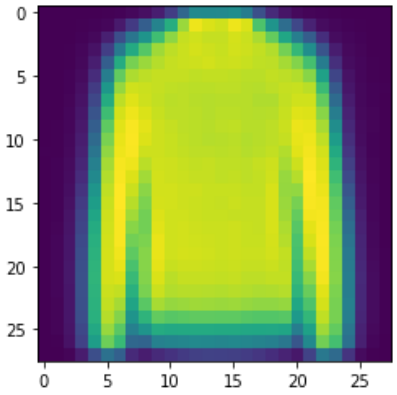


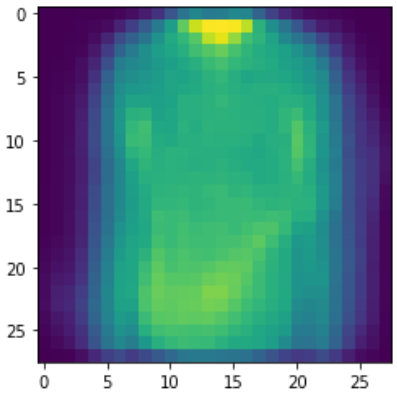
We can cluster the images exactly as we did for the 2d dataset.

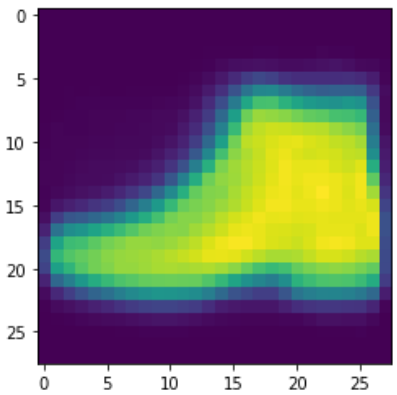
In [0]:
```
clusterer = KMeans(n_clusters=10, random_state=10)
cluster_labels = clusterer.fit_predict(X)
```

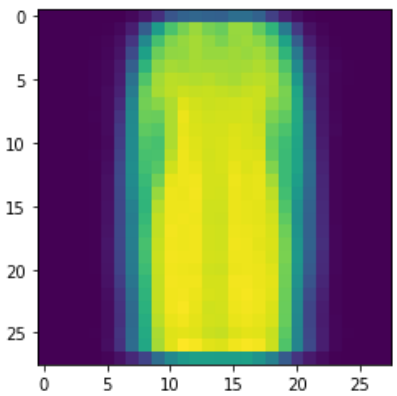
We can plot the cluster centers (wich are 2D figures!) to see if the clustering is learning correct patterns!

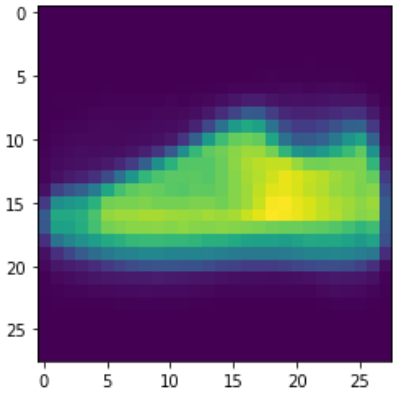
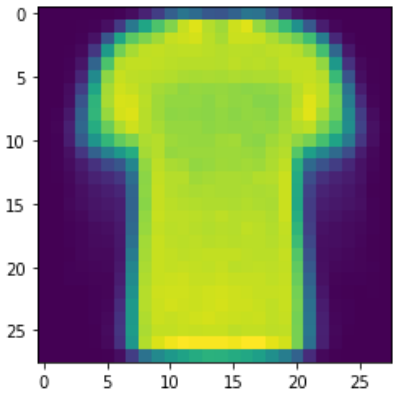
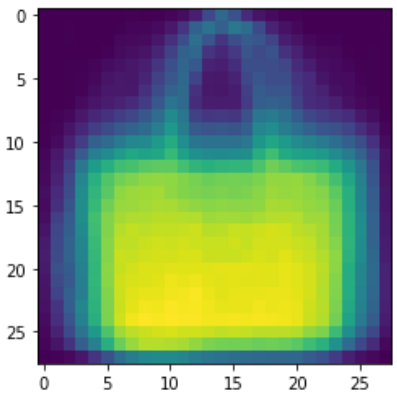
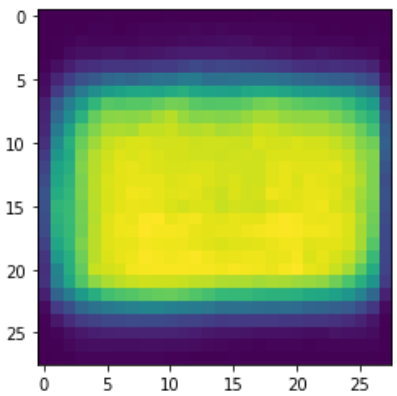
In [14]:
```
for iclust in range(10):
    plt.figure()
    plt.imshow(clusterer.cluster_centers_[iclust].reshape(28,28))
```

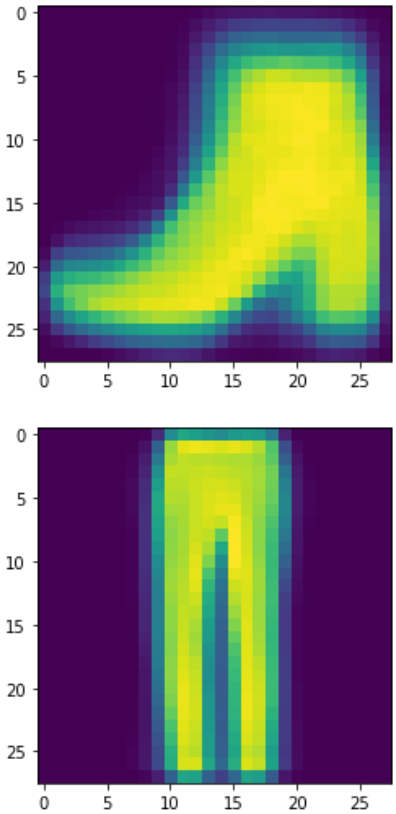

0
5
10
15
20
25
0 5 10 15 20 25


0
5
10
15
20
25
0 5 10 15 20 25


0
5
10
15
20
25
0 5 10 15 20 25


0
5
10
15
20
25
0 5 10 15 20 25



You can see that the model looks to assign one class to the same good. Nevertheless, using the cluster centers and with a further trick, in exercise 2 you will build a digit recognition model !

EXERCISE 1: Discover the number of Gaussians

In [0]:
```
### In this exercise you are given the dataset points, consisting of hig
h-dimensional data. It was built taking random
#samples from a number k of multimensional gaussians. The data is theref
ore made of k clusters but, being
#very high dimensional, you cannot visualize it. Your task it too use K-
means combined with the Silouhette
#score to find the number of k.

# 1. Load the data using the function load_ex1_data_clust() , check the
dimensionality of the data.

# 2. Fix a number of clusters k and define a KMeans clusterer object. Pe
rform the fitting and compute the Silhouette score.
# Save the results on a list.

# 3. Plot the Silhouette scores as a function ok k? What is the number o
f clusters ?

# 4. Optional. Check the result that you found via umap.
```

EXERCISE 2: Predict the good using K-Means

In [0]:
```
#In this exercise you are asked to use the clustering performed by K-mea
ns to predict the good in the f-mnist dataset.
#Here we are using the clustering as a preprocessing for a supervised ta
sk. We need therefore the correct labels
#on a training set and #o test the result on a test set:

# 1. Load the dataset.

#fmnist = tf.keras.datasets.fashion_mnist
#(train_images, train_labels), (test_images, test_labels) = fmnist.load_
data()

#X_train=train_images[:5000,:].reshape(5000,-1)
#y_train=train_labels[:5000]

#X_test=test_images[:1000,:].reshape(1000,-1)
#y_test=test_labels[:1000]


# 2. FITTING STEP: The fitting step consists first here in the computati
on of the cluster center, which was done during
# the presentation. Second, to each cluster center we need than to assig
n a good-label, which will be given by the
# majority class of the sample belonging to that cluster.
#
# You can use, if you want, the helper function most_common for this pur
pose.
#
# In detail you should.
# - fix a number of clusters (start with k=10) and define the cluster KM
eans object. fit the model on the training set
# using the fit method
# - call the predict method of the KMeans object you defined on the trai
ning set and compute the cluster labels.
# Call them cluster_labels
# - use the function most_common with arguments (k,y_train, cluster_labe
ls) to compute the assignement list.
#  assignement[i] will be the majority class of the i-cluster

def most_common(nclusters, supervised_labels, cluster_labels):

    """
    Args:
    - nclusters : the number of clusteres
    - supervised_labels : for each sample, the labelling provided by the
training data ( e.g. in y_train or y_test)
    - cluster_labels : for each good, the cluster it was assigned by K-M
eans using the predict method of the Kmeans object

    Returns:
    - a list "assignement" of lengths nclusters, where assignement[i] is
the majority class of the i-cluster
    """

    assignement=[]
    for icluster in range(nclusters):
        indices=list(supervised_labels[cluster_labels==icluster])
        try:
            chosen= max(set(indices), key=indices.count)
        except ValueError :
            print('Em')
            chosen=1
        assignement.append(chosen)

    return assignement


# 3. Using the assignment list and the clusterer, check the performance
```

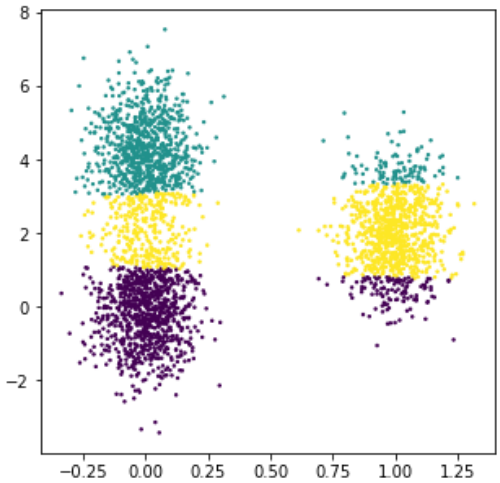
2. Gaussian mixtures

Theory overview.

K-Means is a modelling procedure which is biased towards clusters of circular shape and therefore does not always work perfectly, as the following examples show:

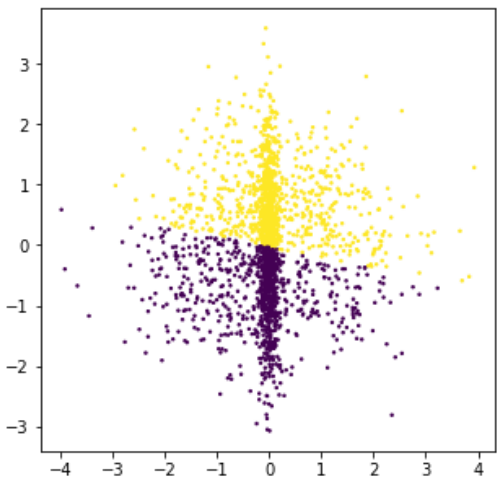
In [19]:
```
points=gm_load_th1()
clusterer = KMeans(n_clusters=3, random_state=10)
cluster_labels=clusterer.fit_predict(points)
plt.figure(figsize=(5,5))
plt.scatter(points[:,0],points[:,1],c=cluster_labels, s=2)
```

Out[19]: <matplotlib.collections.PathCollection at 0x7fa67d6ed390>



In [20]:
```
points=gm_load_th2()
clusterer = KMeans(n_clusters=2, random_state=10)
cluster_labels=clusterer.fit_predict(points)
plt.figure(figsize=(5,5))
plt.scatter(points[:,0],points[:,1],c=cluster_labels, s=2)
```

Out[20]: <matplotlib.collections.PathCollection at 0x7fa67bfefef0>



A Gaussian mixture model is able to fit these kinds of clusters. In a Gaussian mixture model each data-set is supposed to be a random point from the distribution:

$$f(\mathbf{x}) = \sum_c \pi_c N(\mu_\mathbf{c}, \mathbf{\Sigma_c})(\mathbf{x})$$

, which is called a Gaussian mixture. The parameters $\{\pi_c, \mu_\mathbf{c}, \mathbf{\Sigma_c}\}$ are fitted from the data using a minimization procedure (maximum likelyhood via the EM algorithm) and N_c is the chosen number of clusters.

Output of a GM computation:

- *Cluster probabilities:* A gaussian mixtures model is an example of soft clustering, where each data point p does not get assigned a unique cluser, but a distribution over clusters $f_p(c), c = 1, \dots, N_c$.

Given the fitted parameters, $f_p(c)$ is computed as:

$$f_p(c) = \frac{\pi_c N(\mu_\mathbf{c}, \mathbf{\Sigma_c})(\mathbf{x_p})}{\sum_c \pi_c N(\mu_\mathbf{c}, \mathbf{\Sigma_c})(\mathbf{x_p})}, c = 1...N_c$$

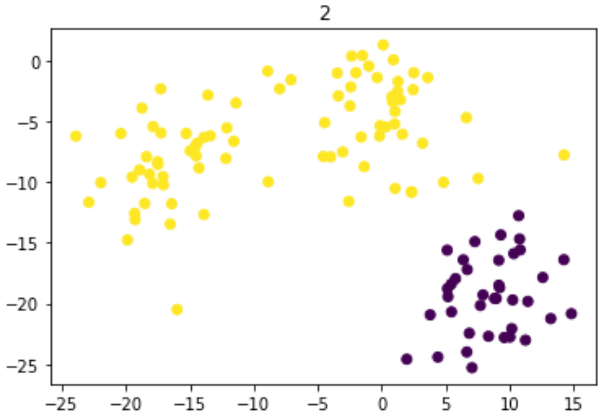
- *AIC/BIC:* after each clustering two numbers are returned. These can be used to select the optimal number of Gaussians to be used, similar to the Silhouette score. (AIC and BIC consider both the likelihood of the data given the parameters and the complexity of the model related to the number of Gaussians used). The lowest AIC or BIC value is an indication of a good fit.

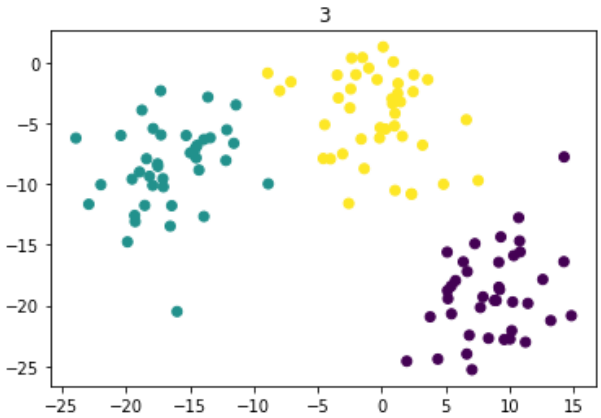
Sklearn: implementation and usage of Gaussian mixtures

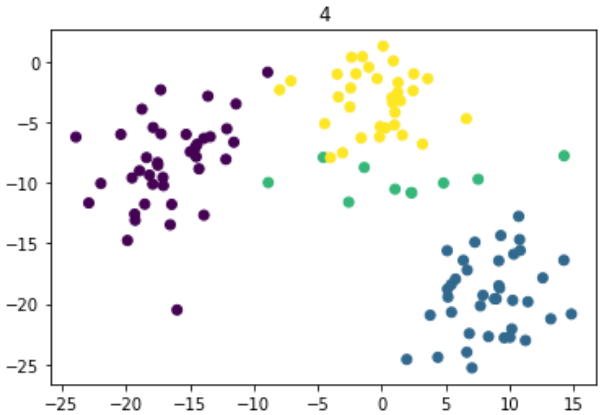
First of all we see how the Gaussian model behaves on our original example:

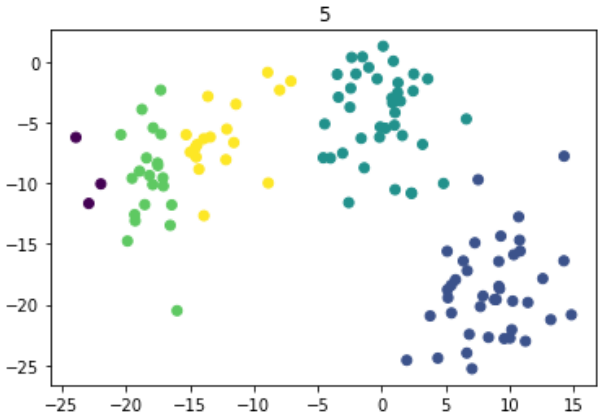
In [21]:
```
points=km_load_th1()

aic=[]
bic=[]
sil=[]

for i_comp in range(2,6):
    plt.figure()
    plt.title(str(i_comp))
    clf = GaussianMixture(n_components=i_comp, covariance_type='full')
    clf.fit(points)
    cluster_labels=clf.predict(points)
    plt.scatter(points[:,0],points[:,1],c=cluster_labels)
    print(i_comp,clf.aic(points),clf.bic(points))
    score=silhouette_score(points,cluster_labels)
    aic.append(clf.aic(points))
    bic.append(clf.bic(points))
    sil.append(score)
```

```
2 1592.1418091070063 1622.804218277609
3 1550.4974051432473 1597.884764770542
4 1553.5290520513045 1617.6413621352915
5 1560.815736563503 1641.6529971041823
```

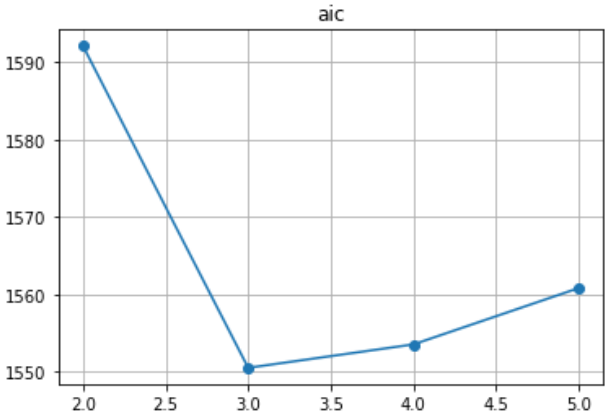


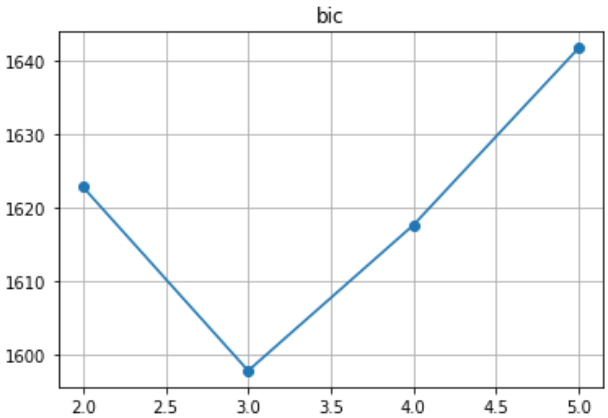


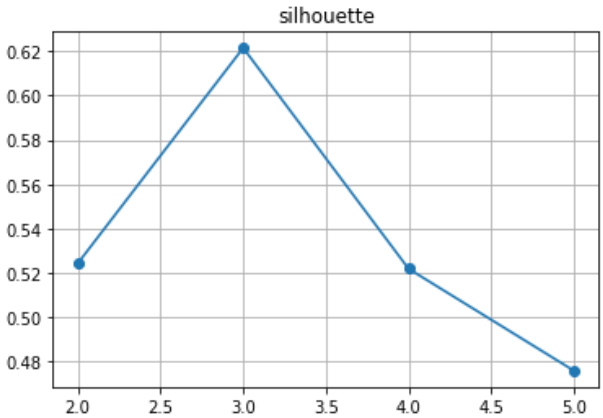



5
0
−5
−10
−15
−20
−25
−25
−20
−15
−10
−5
0
5
10
15

In [22]:
```
plt.plot(np.arange(2,6),aic,'-o')
plt.title('aic')
plt.grid()
plt.figure()
plt.plot(np.arange(2,6),bic,'-o')
plt.title('bic')
plt.grid()
plt.figure()
plt.plot(np.arange(2,6),sil,'-o')
plt.title('silhouette')
plt.grid()
```



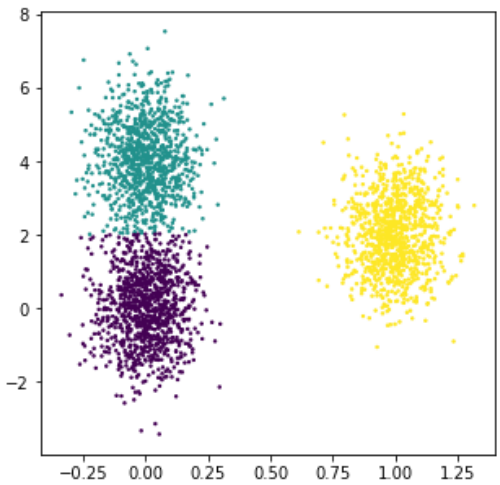




So in this case we get a comparable results, and also the probabilistic tools agree with the Silhouette score ! Let's see how the Gaussian mixtures behave in the examples where K-means was failing.

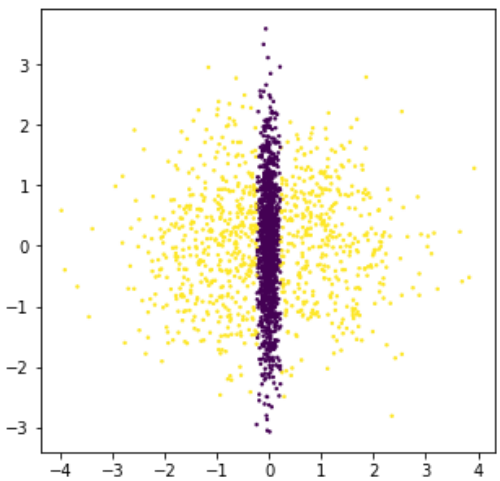
In [23]:
```
points=gm_load_th1()
clf = GaussianMixture(n_components=3, covariance_type='full')
clf.fit(points)
cluster_labels=clf.predict(points)
plt.figure(figsize=(5,5))
plt.scatter(points[:,0],points[:,1],c=cluster_labels, s=2)
```

Out[23]: <matplotlib.collections.PathCollection at 0x7fa67bfc72e8>



In [24]:
```
points=gm_load_th2()
clf = GaussianMixture(n_components=2, covariance_type='full')
clf.fit(points)
cluster_labels=clf.predict(points)
plt.figure(figsize=(5,5))
plt.scatter(points[:,0],points[:,1],c=cluster_labels, s=2)
```

Out[24]: <matplotlib.collections.PathCollection at 0x7fa67bacf3c8>



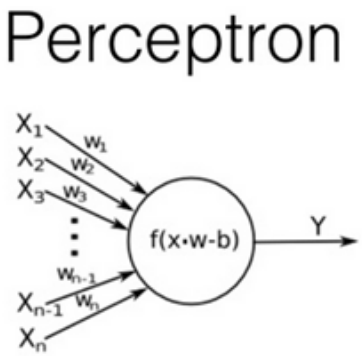
EXERCISE 3 : Find the prediction uncertainty

In [0]:
```
#In this exercise you need to load the dataset used to present K-means ( def km_load_th1() ) or the one used to discuss
# the Gaussian mixtures model ( def km_load_th1() ).
#As discussed, applying a fitting based on gaussian mixtures you can not only predict the cluster label for each point,
#but also a probability distribution over the clusters.

#From this probability distribution, you can compute for each point the entropy of the corresponging
#distribution (using for example scipy.stats.entropy) as an estimation of the undertainty of the prediction.
#Your task is to plot the data-cloud with a color proportional to the uncertainty of the cluster assignement.

# In detail you shoud:
# 1. Instantiate a GaussianMixture object with the number of clusters that you expect
# 2. fit the object on the dataset with the fit method
# 3. compute the cluster probabilities using the method predict_proba. This will return a matrix of dimension
# npoints x nclusters
# 4. use the entropy function ( from scipy.stats import entropy ) to evaluate for each point the uncertainty of the prediction
# 5. Plot the points colored accordingly to their uncertanty. You can use for example the code

#cm = plt.cm.get_cmap('RdYlBu')
#plt.scatter(x, y, c=colors, cmap=cm)
#plt.colorbar(sc)

# where colors is the list of entropies computed.
```

2. Neural Networks Introduction

1. Perceptron

(Artificial) Neural network consists of layers of neurons. Artificial neuron, or perceptron, is in fact inspired by a biological neuron.



Such neuron first calculates the linear transformation of the input vector $\bar{x}$:

$$z = \bar{W} \cdot \bar{x} + b = \sum W_i x_i + b$$

where $\bar{W}$ is vector of weights and b - bias.

2. Nonlinearity

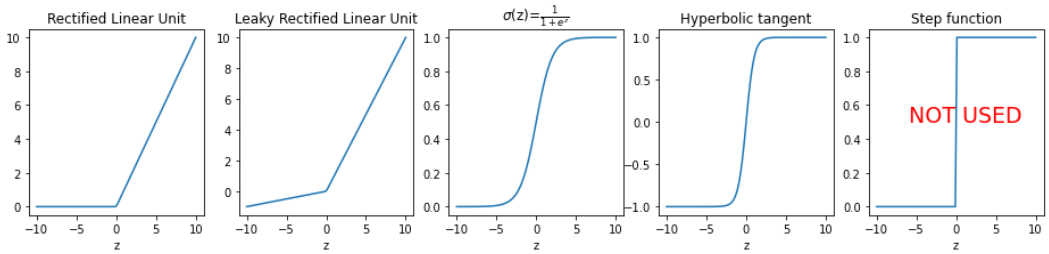
Combining multiple of such objects performing linear transformation would not bring any additional benefit, as the combined output would still be a linear combination of the inputs.

What gives actual power to neurons, is that they additionally perform the nonlinear transformation of the result using activation function f

$$y = f(z)$$

The most commonly used non-linear transformations are:

In [27]:
```
def ReLU(z):
  return np.clip(z, a_min=0, a_max=np.max(z))
def LReLU(z, a=0.1):
  return np.clip(z, a_min=0, a_max=np.max(z)) + np.clip(z, a_min=np.min
(z), a_max=0) * a
def sigmoid(z):
  return 1/(1 + np.exp(-z))
def step(z):
  return np.heaviside(z, 0)
fig, ax = plt.subplots(1, 5, figsize=(16, 3))
z = np.linspace(-10, 10, 100)
ax[0].plot(z, ReLU(z))
ax[0].set_title('Rectified Linear Unit')
ax[1].plot(z, LReLU(z))
ax[1].set_title('Leaky Rectified Linear Unit')
ax[2].plot(z, sigmoid(z))
ax[2].set_title(r'$\sigma$(z)=$\frac{1}{1+e^z}$')
ax[3].plot(z, np.tanh(z))
ax[3].set_title('Hyperbolic tangent');
ax[4].plot(z, step(z)
)
ax[4].text(-6, 0.5, 'NOT USED', size=19, c='r')
ax[4].set_title('Step function');
for axi in ax:
  axi.set_xlabel('z')
```



And the reason we don't use a simple step function, is that it's not differentiable or it's derivative is zero everywhere.

The last nonlinearity to mention here is *softmax*:

$$y_i = Softmax(\bar{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

While each z_i can hav any value, the corresponding $y_i \in [0, 1]$, and $\sum_i y_i = 1$, just like probabilities!

While these y_i are only pseudo-probabilities, this nonlinearity allowes one to model probabilities, e.g. of a data-point belonging to a certain class.

3. Fully connected net

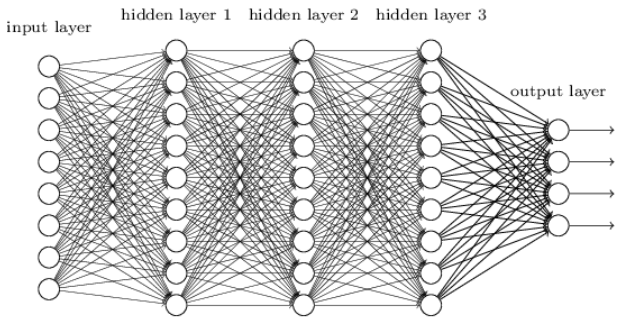
In a fully connected neural network each layer is a set of N neurons, performing different transformations of all the same layer's inputs $\bar{x} = [x_i]$ producing output vector $\bar{y} = [y_j]_{i=1..N}$:

$$y_j = f(\bar{W}_j \cdot \bar{x} + b_j)$$

Since output of each layer forms input of next layer, one can write for layer l:

$$x_j^l = f(\bar{W}_j^l \cdot \bar{x}^{l-1} + b_j^l)$$

where $\bar{x}^0$ is network's input vactor.



4. Loss function

The last part of the puzzle is the measure of network performance, which is used to optimize the network's parameters W_j^l and b_j^l. Denoting the network's output for an input x_i as $\hat{y}_i = \hat{y}_i(x_i)$ and given the label y_i:

1. In case of regression loss shows "distance" from target values:
2. L2 (MSE): $L = \sum_i (y_i - \hat{y}_i)^2$
3. L1 (MAE): $L = \sum_i |y_i - \hat{y}_i|$
4. In case of classification we can use cross-entropy, which shows "distance" from target distribution:

$$L = -\sum_i \sum_c y_{i,c} \log(\hat{y}_{i,c})$$

Here $\hat{y}_{i,c}$ - pseudo-probability of x_i belinging to class c and $y_{i,c}$ uses 1-hot encoding:

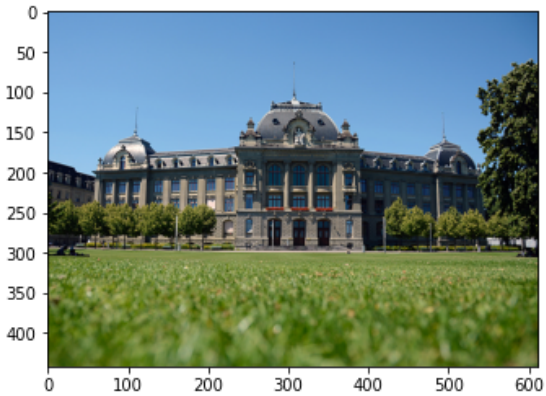
$$y_{i,c} = \begin{cases} 1, & \text{if } x_i \text{ belongs to class } c \\ 0, & \text{otherwise} \end{cases}$$

3. Regression with neural network

Here we will build a neural network to fit an image.

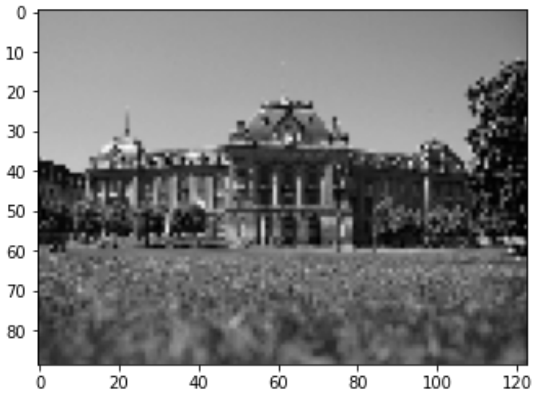
In [28]:
```
image_big = imread('https://www.unibe.ch/unibe/portal/content/carousel/s
howitem940548/UniBE_Coronavirus_612p_eng.jpg')
image_big = image_big[...,0:3]/255
plt.imshow(image_big)
```

Out[28]: <matplotlib.image.AxesImage at 0x7fa6816a3668>



In [29]:
```
image = image_big[::5, ::5]
image = image.mean(axis=2, keepdims=True)
plt.imshow(image[...,0], cmap='gray')
```

Out[29]: <matplotlib.image.AxesImage at 0x7fa68101ca90>



In [0]:
```
h, w, c = image.shape
```

In [31]:
```
X = np.meshgrid(np.linspace(0, 1, w), np.linspace(0, 1, h))
X = np.stack(X, axis=-1).reshape((-1, 2))

Y = image.reshape((-1, c))
X.shape, Y.shape
```

Out[31]: ((10947, 2), (10947, 1))

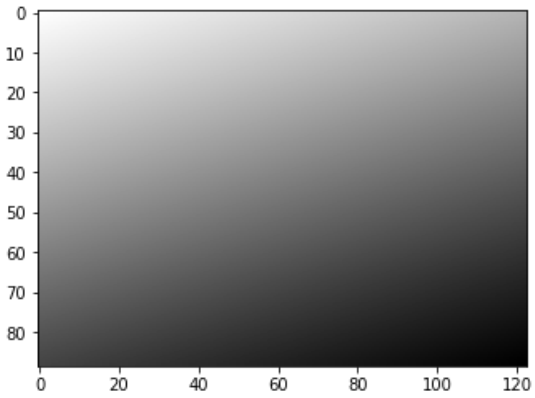
In [32]:
```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(2,)),
  tf.keras.layers.Dense(c, activation='sigmoid'),
])
model.compile(optimizer='adam',
              loss='mae',
              metrics=['mse'])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten (Flatten)            (None, 2)                 0         
_________________________________________________________________
dense (Dense)                (None, 1)                 3         
=================================================================
Total params: 3
Trainable params: 3
Non-trainable params: 0
_________________________________________________________________
```

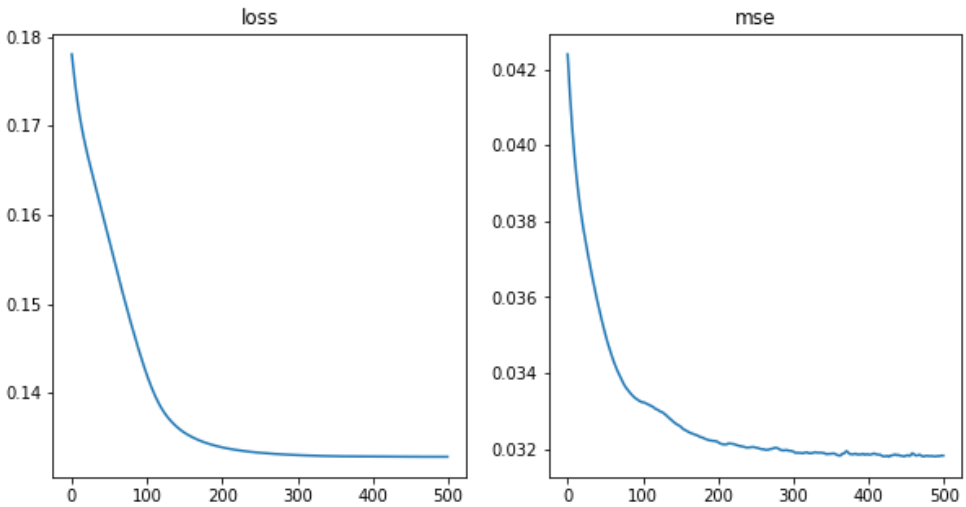
In [33]:
```
hist = model.fit(X, Y, epochs=500, batch_size=2048)
```

```
Epoch 1/500
6/6 [==============================] - 0s 4ms/step - loss: 0.1780 - mse:
0.0424
Epoch 2/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1772 - mse:
0.0420
Epoch 3/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1765 - mse:
0.0417
Epoch 4/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1757 - mse:
0.0414
Epoch 5/500
6/6 [==============================] - 0s 4ms/step - loss: 0.1750 - mse:
0.0411
Epoch 6/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1743 - mse:
0.0408
Epoch 7/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1737 - mse:
0.0405
Epoch 8/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1730 - mse:
0.0402
Epoch 9/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1724 - mse:
0.0400
Epoch 10/500
6/6 [==============================] - 0s 4ms/step - loss: 0.1719 - mse:
0.0397
Epoch 11/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1713 - mse:
0.0395
Epoch 12/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1708 - mse:
0.0393
Epoch 13/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1704 - mse:
0.0391
Epoch 14/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1699 - mse:
0.0389
Epoch 15/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1695 - mse:
0.0388
Epoch 16/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1690 - mse:
0.0386
Epoch 17/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1686 - mse:
0.0385
Epoch 18/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1682 - mse:
0.0383
Epoch 19/500
6/6 [==============================] - 0s 4ms/step - loss: 0.1678 - mse:
0.0382
Epoch 20/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1674 - mse:
0.0380
Epoch 21/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1671 - mse:
0.0379
Epoch 22/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1667 - mse:
0.0378
Epoch 23/500
6/6 [==============================] - 0s 3ms/step - loss: 0.1664 - mse:
```

In [34]:
```
Y_p = model.predict(X)
Y_p = Y_p.reshape((h,w,c))
im = plt.imshow(Y_p[...,0], cmap='gray')
```



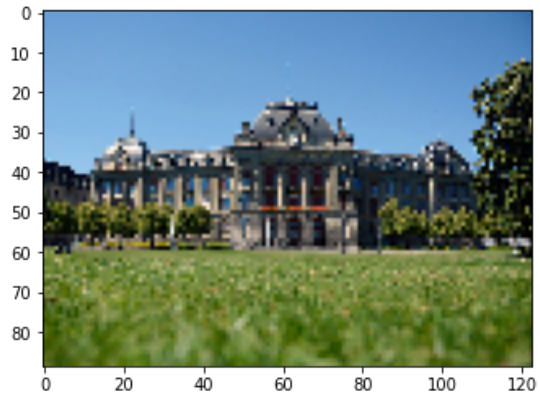
In [35]:
```
fig, axs = plt.subplots(1, 2, figsize=(10,5))
axs[0].plot(hist.epoch, hist.history['loss'])
axs[0].set_title('loss')
axs[1].plot(hist.epoch, hist.history['mse'])
axs[1].set_title('mse')
plt.show()
```



Let's try the same with an RGB image:

In [36]:
```
image = image_big[::5, ::5]
plt.imshow(image)
```

Out[36]: <matplotlib.image.AxesImage at 0x7fa67be874a8>



In [37]:
```
h, w, c = image.shape
X = np.meshgrid(np.linspace(0, 1, w), np.linspace(0, 1, h))
X = np.stack(X, axis=-1).reshape((-1, 2))

Y = image.reshape((-1, c))
X.shape, Y.shape
```

Out[37]: ((10947, 2), (10947, 3))

In [38]:
```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(2,)),
  tf.keras.layers.Dense(c, activation='sigmoid'),
])
model.compile(optimizer='adam',
              loss='mae',
              metrics=['mse'])
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 2)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 9
=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
```

But now we will save images during the course of training, at first every 2 epochs, then every 20, every 200 and finally every 1000. (**Remember**: call to `model.fit` does NOT reinitialize trainable variables. Every time it continues from the previous state):

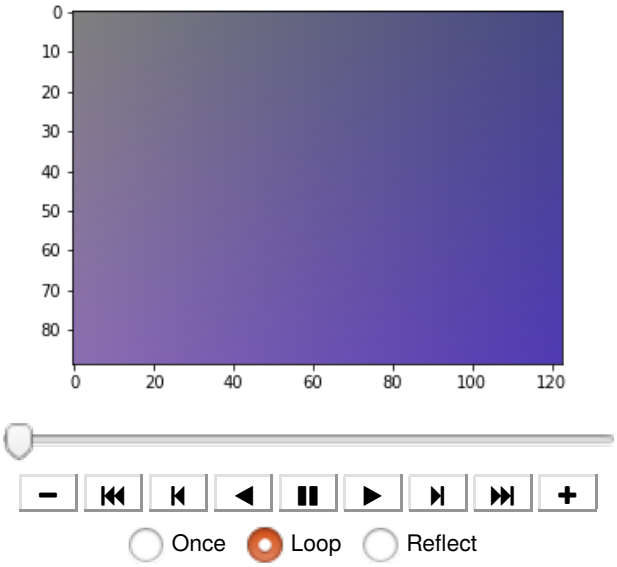
In [39]:
```
ims = []
n_ep_tot = 0
for i in range(170):
  if i % 10 == 0:
    print(f'epoch {i}', end='\n')
  ne = (2 if (i<50) else (20 if (i<100) else (200 if (i<150) else 100
0)))
  model.fit(X, Y, epochs=ne, batch_size=1*2048, verbose=0)

  Y_p = model.predict(X)
  Y_p = Y_p.reshape((h, w, c))
  ims.append(Y_p)
  n_ep_tot += ne

print(f'total numer of epochs trained:{n_ep_tot}')
```

```
epoch 0
epoch 10
epoch 20
epoch 30
epoch 40
epoch 50
epoch 60
epoch 70
epoch 80
epoch 90
epoch 100
epoch 110
epoch 120
epoch 130
epoch 140
epoch 150
epoch 160
total numer of epochs trained:31100
```

In [0]:
```
%%capture
plt.rcParams["animation.html"] = "jshtml"  # for matplotlib 2.1 and abov
e, uses JavaScript

fig = plt.figure()
im = plt.imshow(ims[0])


def animate(i):
    img = ims[i]
    im.set_data(img)
    return im

ani = animation.FuncAnimation(fig, animate, frames=len(ims))
```

In [41]:
```
ani
```

Out[41]:



While the colors properly represent the target image, out model still poses very limited capacity, allowing it to effectively represent only 3 boundaries.

Let's upscale out model:

In [42]:
```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(2,)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(8, activation='relu'),
  tf.keras.layers.Dense(c, activation='sigmoid'),
])
model.compile(optimizer='adam',
              loss='mae',
              metrics=['mse'])
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 2)                 0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               384
_________________________________________________________________
dense_3 (Dense)              (None, 8)                 1032
_________________________________________________________________
dense_4 (Dense)              (None, 3)                 27
=================================================================
Total params: 1,443
Trainable params: 1,443
Non-trainable params: 0
_________________________________________________________________
```

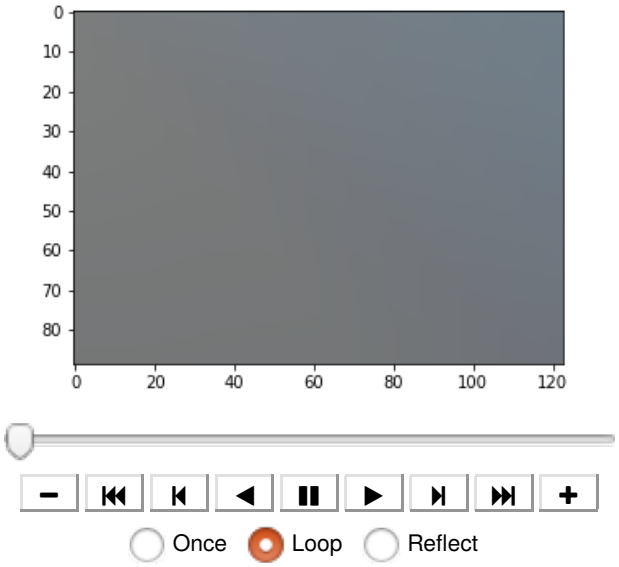
In [43]:
```
ims = []
n_ep_tot = 0
for i in range(180):
  if i % 10 == 0:
    print(f'epoch {i}', end='\n')
  ne = (2 if (i<50) else (20 if (i<100) else (200 if (i<150) else 100
0)))
  model.fit(X, Y, epochs=ne, batch_size=1*2048, verbose=0)

  Y_p = model.predict(X)
  Y_p = Y_p.reshape((h, w, c))
  ims.append(Y_p)
  n_ep_tot += ne

print(f'total numer of epochs trained:{n_ep_tot}')
```

```
epoch 0
epoch 10
epoch 20
epoch 30
epoch 40
epoch 50
epoch 60
epoch 70
epoch 80
epoch 90
epoch 100
epoch 110
epoch 120
epoch 130
epoch 140
epoch 150
epoch 160
epoch 170
total numer of epochs trained:41100
```

In [0]:
```
%%capture
fig = plt.figure()
im = plt.imshow(ims[0])

def animate(i):
    img = ims[i]
    im.set_data(img)
    return im

ani = animation.FuncAnimation(fig, animate, frames=len(ims))
```

In [45]:
```
ani
```

Out[45]:



In [0]:
```
%%capture
fig = plt.figure()
im = plt.imshow(imsa[0])

def animate(i):
    img = imsa[i]
    im.set_data(img)
    return im

ani = animation.FuncAnimation(fig, animate, frames=len(imsa))
```

EXERCISE 4.

Load some image, downscale to a similar resolution, and train a deeper model, for example 5 layers, more parameters in widest layers.

In [0]:
```
# 1. Load your image

# 2. build a deeper model

# 3. inspect the evolution
```

Data Science Fundamentals 5

Basic introduction on how to perform typical machine learning tasks with Python.

Prepared by Mykhailo Vladymyrov & Aris Marcolongo, Science IT Support, University Of Bern, 2020

Part 4.

In [0]:
```
from matplotlib import  pyplot as plt
import numpy as np
from imageio import imread
import pandas as pd
from time import time as timer

import tensorflow as tf

%matplotlib inline
from matplotlib import animation
from IPython.display import HTML
```

1. Classification with neural network

1. Bulding a neural network

The following creates a 'model'. It is an object containing the ML model itself - a simple 3-layer fully connected neural network, optimization parameters, as well as tha interface for model training.

In [0]:
```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Model summary provides information about the model's layers and trainable parameters

In [3]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 10)                7850
=================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
_________________________________________________________________
```

2. Model training

The `fit` function is the interface for model training. Here one can specify training and validation datasets, minibatch size, and the number of training epochs.

We will also save the state of the trainable variables after each epoch:

In [4]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
x_train = x_train/255
x_test = x_test/255

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
32768/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

In [5]:
```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.summary()
```

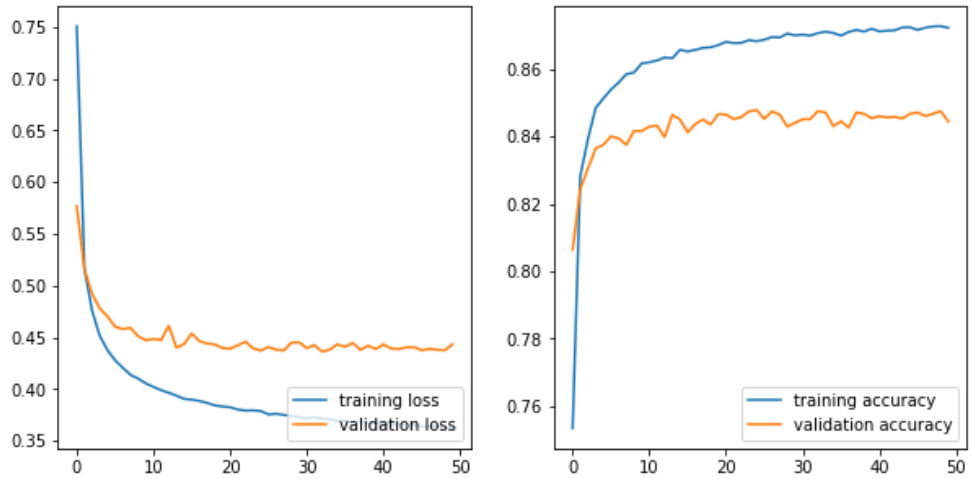
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
flatten_1 (Flatten)          (None, 784)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 10)                7850      
=================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
_________________________________________________________________
```

Here during training we also save the trained models checkpoints after each epoch of training.

In [6]:
```
save_path = 'save/mnist_{epoch}.ckpt'
save_callback = tf.keras.callbacks.ModelCheckpoint(filepath=save_path, s
ave_weights_only=True)

hist = model.fit(x=x_train, y=y_train,
                 epochs=50, batch_size=128,
                 validation_data=(x_test, y_test),
                 callbacks=[save_callback])
```

```
Epoch 1/50
469/469 [==============================] - 2s 4ms/step - loss: 0.7509 - a
ccuracy: 0.7534 - val_loss: 0.5770 - val_accuracy: 0.8064
Epoch 2/50
469/469 [==============================] - 2s 4ms/step - loss: 0.5161 - a
ccuracy: 0.8283 - val_loss: 0.5155 - val_accuracy: 0.8245
Epoch 3/50
469/469 [==============================] - 2s 4ms/step - loss: 0.4758 - a
ccuracy: 0.8393 - val_loss: 0.4922 - val_accuracy: 0.8305
Epoch 4/50
469/469 [==============================] - 2s 4ms/step - loss: 0.4515 - a
ccuracy: 0.8484 - val_loss: 0.4780 - val_accuracy: 0.8364
Epoch 5/50
469/469 [==============================] - 2s 4ms/step - loss: 0.4378 - a
ccuracy: 0.8512 - val_loss: 0.4704 - val_accuracy: 0.8375
Epoch 6/50
469/469 [==============================] - 2s 4ms/step - loss: 0.4279 - a
ccuracy: 0.8539 - val_loss: 0.4603 - val_accuracy: 0.8400
Epoch 7/50
469/469 [==============================] - 2s 4ms/step - loss: 0.4206 - a
ccuracy: 0.8559 - val_loss: 0.4580 - val_accuracy: 0.8394
Epoch 8/50
469/469 [==============================] - 2s 4ms/step - loss: 0.4138 - a
ccuracy: 0.8584 - val_loss: 0.4591 - val_accuracy: 0.8375
Epoch 9/50
469/469 [==============================] - 2s 4ms/step - loss: 0.4102 - a
ccuracy: 0.8589 - val_loss: 0.4512 - val_accuracy: 0.8416
Epoch 10/50
469/469 [==============================] - 2s 4ms/step - loss: 0.4056 - a
ccuracy: 0.8616 - val_loss: 0.4472 - val_accuracy: 0.8416
Epoch 11/50
469/469 [==============================] - 2s 4ms/step - loss: 0.4022 - a
ccuracy: 0.8620 - val_loss: 0.4486 - val_accuracy: 0.8429
Epoch 12/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3990 - a
ccuracy: 0.8625 - val_loss: 0.4475 - val_accuracy: 0.8432
Epoch 13/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3965 - a
ccuracy: 0.8634 - val_loss: 0.4612 - val_accuracy: 0.8398
Epoch 14/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3937 - a
ccuracy: 0.8632 - val_loss: 0.4401 - val_accuracy: 0.8464
Epoch 15/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3906 - a
ccuracy: 0.8657 - val_loss: 0.4436 - val_accuracy: 0.8450
Epoch 16/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3899 - a
ccuracy: 0.8652 - val_loss: 0.4538 - val_accuracy: 0.8412
Epoch 17/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3886 - a
ccuracy: 0.8656 - val_loss: 0.4463 - val_accuracy: 0.8437
Epoch 18/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3867 - a
ccuracy: 0.8662 - val_loss: 0.4440 - val_accuracy: 0.8450
Epoch 19/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3843 - a
ccuracy: 0.8664 - val_loss: 0.4431 - val_accuracy: 0.8435
Epoch 20/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3831 - a
ccuracy: 0.8670 - val_loss: 0.4398 - val_accuracy: 0.8466
Epoch 21/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3825 - a
ccuracy: 0.8680 - val_loss: 0.4391 - val_accuracy: 0.8465
Epoch 22/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3803 - a
ccuracy: 0.8676 - val_loss: 0.4423 - val_accuracy: 0.8451
Epoch 23/50
469/469 [==============================] - 2s 4ms/step - loss: 0.3793 - a
```

In [7]:
```
fig, axs = plt.subplots(1, 2, figsize=(10,5))
axs[0].plot(hist.epoch, hist.history['loss'])
axs[0].plot(hist.epoch, hist.history['val_loss'])
axs[0].legend(('training loss', 'validation loss'), loc='lower right')
axs[1].plot(hist.epoch, hist.history['accuracy'])
axs[1].plot(hist.epoch, hist.history['val_accuracy'])

axs[1].legend(('training accuracy', 'validation accuracy'), loc='lower r
ight')
plt.show()
```



Current model performance can be evaluated on a dataset:

In [8]:
```
model.evaluate(x_test,  y_test, verbose=2)
```

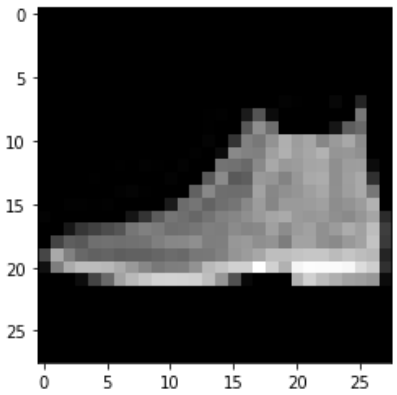
```
313/313 - 1s - loss: 0.4433 - accuracy: 0.8444
```

Out[8]:
```
[0.4432746469974518, 0.8443999886512756]
```

We can test trained model on a image:

In [9]:
```
im_id = 0
y_pred = model(x_test)

y_pred_most_probable = np.argmax(y_pred[im_id])
print('true lablel: ', y_test[im_id],
      '; predicted: ',  y_pred_most_probable,
      f'({class_names[y_pred_most_probable]})')
plt.imshow(x_test[im_id], cmap='gray');
```

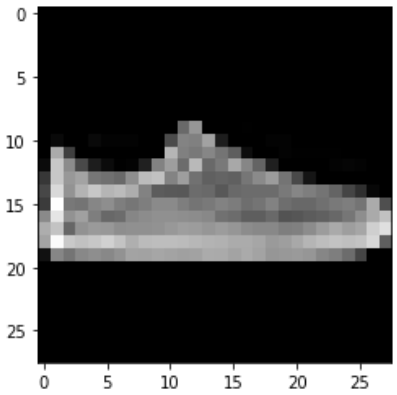
```
true lablel:  9 ; predicted:  9 (Ankle boot)
```



As well as inspect on which samples does the model fail:

In [10]:
```
y_pred_most_probable_all = np.argmax(y_pred, axis=1)
wrong_pred_map = y_pred_most_probable_all!=y_test
wrong_pred_idx = np.arange(len(wrong_pred_map))[wrong_pred_map]

im_id = wrong_pred_idx[0]

y_pred_most_probable = y_pred_most_probable_all[im_id]
print('true lablel: ', y_test[im_id],
      f'({class_names[y_test[im_id]]})',
      '; predicted: ',  y_pred_most_probable,
      f'({class_names[y_pred_most_probable]})')
plt.imshow(x_test[im_id], cmap='gray');
```

```
true lablel:  7 (Sneaker) ; predicted:  5 (Sandal)
```



3. Loading trained model

In [11]:
```
model.load_weights('save/mnist_1.ckpt')
model.evaluate(x_test,  y_test, verbose=2)

model.load_weights('save/mnist_12.ckpt')
model.evaluate(x_test,  y_test, verbose=2)

model.load_weights('save/mnist_18.ckpt')
model.evaluate(x_test,  y_test, verbose=2)
```

```
313/313 - 1s - loss: 0.5770 - accuracy: 0.8064
313/313 - 1s - loss: 0.4475 - accuracy: 0.8432
313/313 - 1s - loss: 0.4440 - accuracy: 0.8450
```

Out[11]: [0.4440324306488037, 0.8450000286102295]

4. Inspecting trained variables

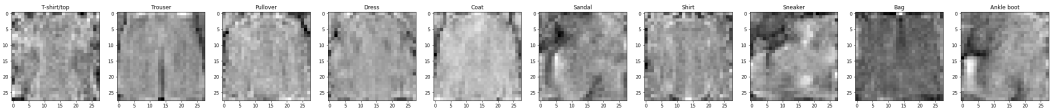
We can obtain the trained variables from model layers:

In [12]:
```
l = model.get_layer(index=1)
w, b = l.weights

w = w.numpy()
b = b.numpy()
print(w.shape, b.shape)
w = w.reshape((28,28,-1)).transpose((2, 0, 1))
```

```
(784, 10) (10,)
```

Let's visualize first 5:

In [13]:
```
n = 10
fig, axs = plt.subplots(1, n, figsize=(4.1*n,4))
for i, wi in enumerate(w[:n]):
  axs[i].imshow(wi, cmap='gray')
  axs[i].set_title(class_names[i])
```



6. Inspecting gradients

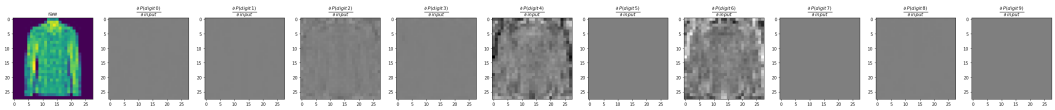
We can also evaluate the gradients of each output with respect to an input:

In [17]:
```
idx = 112
inp_v = x_train[idx:idx+1]  # use some image to compute gradients with r
espect to

inp = tf.constant(inp_v)  # create tf constant tensor
with tf.GradientTape() as tape:  # gradient tape for gradint evaluation
  tape.watch(inp)  # take inp as variable
  preds = model(inp) # evaluate model output

grads = tape.jacobian(preds, inp)  # evaluate d preds[i] / d inp[j]
print(grads.shape, '<- (Batch_preds, preds[i], Batch_inp, inp[y], inp
[x])')
grads = grads.numpy()[0,:,0]
```

```
(1, 10, 1, 28, 28) <- (Batch_preds, preds[i], Batch_inp, inp[y], inp[x])
```

In [18]:
```
print('prediction:', np.argmax(preds[0]))
fig, axs = plt.subplots(1, 11, figsize=(4.1*11,4))
axs[0].imshow(inp_v[0])
axs[0].set_title('raw')
vmin,vmax = grads.min(), grads.max()
for i, g in enumerate(grads):
  axs[i+1].imshow(g, cmap='gray', vmin=vmin, vmax=vmax)
  axs[i+1].set_title(r'$\frac{\partial\;P(digit\,%d)}{\partial\;input}$'
% i, fontdict={'size':16})
```

```
prediction: 6
```



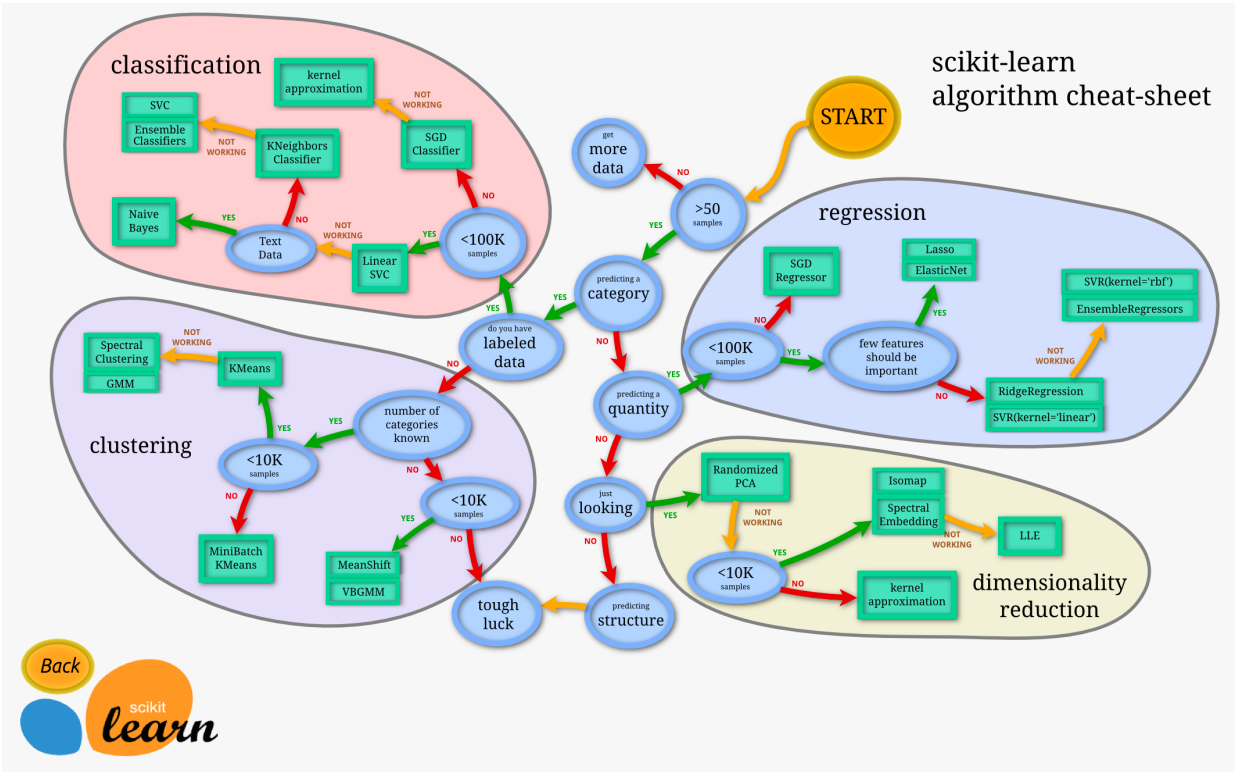
EXERCISE 1: Train deeper network

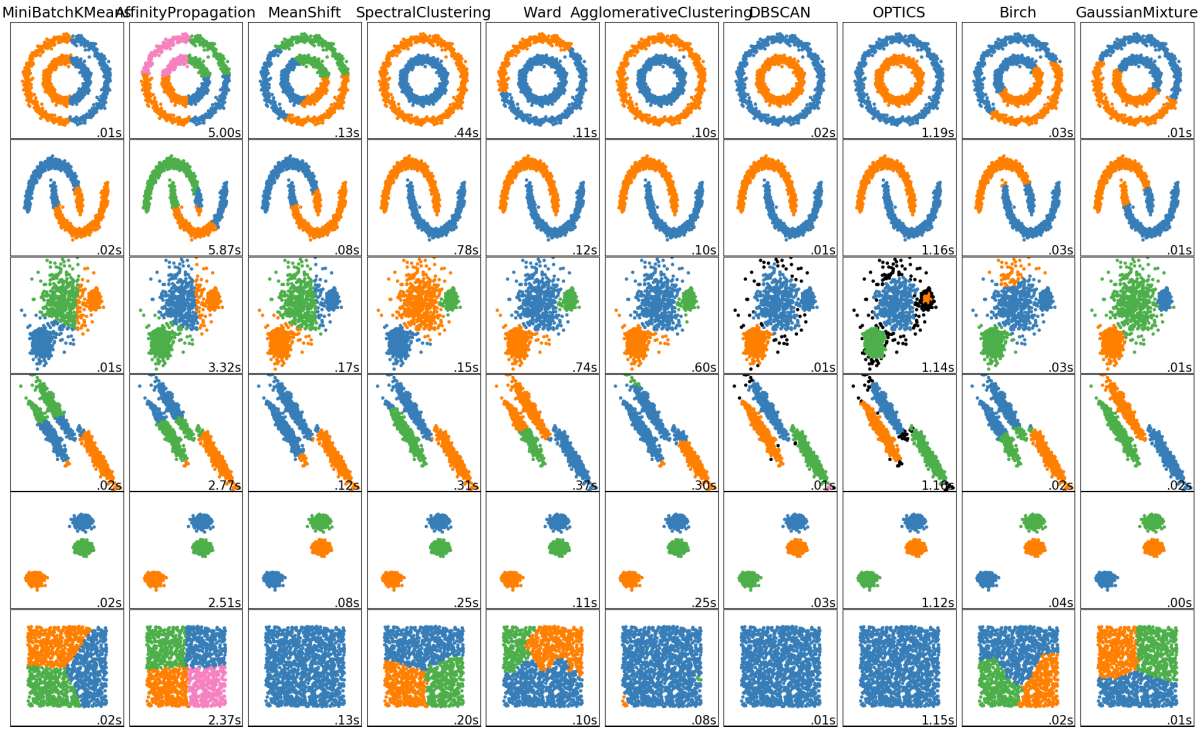
Make a deeper model, with wider layers. Remember to `'softmax'` activation in the last layer, as required for the classification task to encode pseudoprobabilities. In the other layers you could use `'relu'`.

Try to achieve 90% accuracy. Does your model overfit?

In [0]:
```
# 1. create model
# 2. train the model
# 3. plot the loss and accuracy evolution during training
# 4. evaluate model in best point (before overfitting)
```

2. Extras and Q&A


scikit-learn
algorithm cheat-sheet
START
classification
kernel approximation
NOT WORKING
SVC
Ensemble Classifiers
NOT WORKING
KNeighbors Classifier
SGD Classifier
NO
Naive Bayes
YES
NO
Text Data
NOT WORKING
Linear SVC
YES
<100K samples
NO
get more data
NO
>50 samples
YES
regression
SGD Regressor
Lasso
ElasticNet
predicting a category
YES
YES
NO
<100K samples
YES
few features should be important
YES
NO
NOT WORKING
SVR(kernel='rbf')
EnsembleRegressors
RidgeRegression
SVR(kernel='linear')
do you have labeled data
NO
YES
predicting a quantity
YES
NO
Spectral Clustering
GMM
NOT WORKING
KMeans
clustering
YES
number of categories known
YES
<10K samples
NO
NO
<10K samples
YES
MiniBatch KMeans
MeanShift
VBGMM
just looking
YES
NO
Randomized PCA
NOT WORKING
Isomap
Spectral Embedding
NOT WORKING
LLE
YES
<10K samples
NO
kernel approximation
dimensionality reduction
predicting structure
tough luck
Back
scikit
learn


MiniBatchKMeans
AffinityPropagation
MeanShift
SpectralClustering
Ward
AgglomerativeClustering
DBSCAN
OPTICS
Birch
GaussianMixture
.01s 5.00s .13s .44s .11s .10s .02s 1.19s .03s .01s
.02s 5.87s .08s .78s .12s .10s .01s 1.16s .03s .01s
.01s 3.32s .17s .15s .74s .60s .01s 1.14s .03s .01s
.02s 2.77s .12s .31s .37s .30s .01s 1.10s .02s .02s
.02s 2.51s .08s .25s .11s .25s .03s 1.12s .04s .00s
.02s 2.37s .13s .20s .10s .08s .01s 1.15s .02s .01s

Data Science Fundamentals 5

Basic introduction on how to perform typical machine learning tasks with Python.

Prepared by Mykhailo Vladymyrov & Aris Marcolongo, Science IT Support, University Of Bern, 2020

Solutions to Part 1.

In [0]:
```
from sklearn import linear_model

from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn import metrics

from matplotlib import  pyplot as plt
import numpy as np
import os
from imageio import imread
import pandas as pd
from time import time as timer

import tensorflow as tf

%matplotlib inline
from matplotlib import animation
from IPython.display import HTML
```

In [2]:
```
if not os.path.exists('data'):
    path = os.path.abspath('.')+'/colab_material.tgz'
    tf.keras.utils.get_file(path, 'https://github.com/neworldemancer/DSF
5/raw/master/colab_material.tgz')
    !tar -xvzf colab_material.tgz > /dev/null 2>&1
```

```
Downloading data from https://github.com/neworldemancer/DSF5/raw/master/c
olab_material.tgz
98304/96847 [==============================] - 0s 0us/step
```

Datasets

In this course we will use several synthetic and real-world datasets to ilustrate the behavior of the models and excercise our skills.

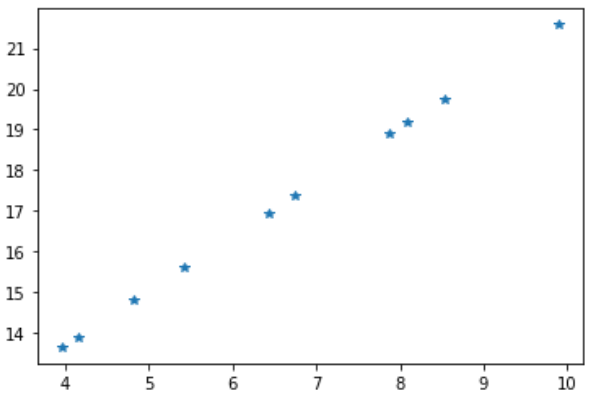
1. Synthetic linear

In [0]:
```
def get_linear(n_d=1, n_points=10, w=None, b=None, sigma=5):
  x = np.random.uniform(0, 10, size=(n_points, n_d))

  w = w or np.random.uniform(0.1, 10, n_d)
  b = b or np.random.uniform(-10, 10)
  y = np.dot(x, w) + b + np.random.normal(0, sigma, size=n_points)

  print('true w =', w, ';  b =', b)

  return x, y
```

In [4]:
```
x, y = get_linear(n_d=1, sigma=0)
plt.plot(x[:, 0], y, '*')
```

```
true w = [1.34032066] ;  b = 8.326857960042354
```

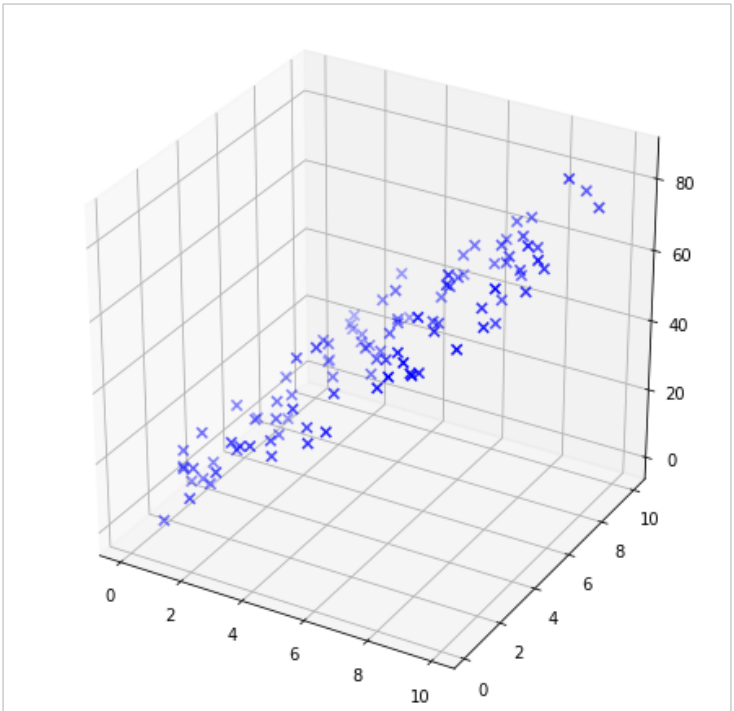
Out[4]: [<matplotlib.lines.Line2D at 0x7f08a22aef28>]



In [5]:
```
n_d = 2
x, y = get_linear(n_d=n_d, n_points=100)

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x[:,0], x[:,1], y, marker='x', color='b',s=40)
```

```
true w = [7.03409766 0.83697333] ;  b = 2.7928040906788194
```

Out[5]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7f08a1dab198>
```



2. House prices

Subset of the the hous pricess kaggle dataset: https://www.kaggle.com/c/house-prices-advanced-regression-techniques (https://www.kaggle.com/c/house-prices-advanced-regression-techniques)

In [0]:
```
def house_prices_dataset(return_df=False, price_max=400000, area_max=400
00):
  path = 'data/train.csv'

  df = pd.read_csv(path, na_values="NaN", keep_default_na=False)

  useful_fields = ['LotArea',
                   'Utilities', 'OverallQual', 'OverallCond',
                   'YearBuilt', 'YearRemodAdd', 'ExterQual', 'ExterCond',
                   'HeatingQC', 'CentralAir', 'Electrical',
                   '1stFlrSF', '2ndFlrSF','GrLivArea',
                   'FullBath', 'HalfBath',
                   'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual', 'TotRms
AbvGrd',
                   'Functional','PoolArea',
                   'YrSold', 'MoSold'
                   ]
  target_field = 'SalePrice'

  cleanup_nums = {"Street":      {"Grvl": 0, "Pave": 1},
                   "LotFrontage": {"NA":0},
                   "Alley":       {"NA":0, "Grvl": 1, "Pave": 2},
                   "LotShape":    {"IR3":0, "IR2": 1, "IR1": 2, "Reg":3},
                   "Utilities":   {"ELO":0, "NoSeWa": 1, "NoSewr": 2, "Al
lPub": 3},
                   "LandSlope":   {"Sev":0, "Mod": 1, "Gtl": 3},
                   "ExterQual":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "ExterCond":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "BsmtQual":    {"NA":0, "Po":1, "Fa": 2, "TA": 3, "G
d": 4, "Ex":5},
                   "BsmtCond":    {"NA":0, "Po":1, "Fa": 2, "TA": 3, "G
d": 4, "Ex":5},
                   "BsmtExposure":{"NA":0, "No":1, "Mn": 2, "Av": 3, "G
d": 4},
                   "BsmtFinType1":{"NA":0, "Unf":1, "LwQ": 2, "Rec": 3, "
BLQ": 4, "ALQ":5, "GLQ":6},
                   "BsmtFinType2":{"NA":0, "Unf":1, "LwQ": 2, "Rec": 3, "
BLQ": 4, "ALQ":5, "GLQ":6},
                   "HeatingQC":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "CentralAir":  {"N":0, "Y": 1},
                   "Electrical":  {"NA":0, "Mix":1, "FuseP":2, "FuseF":
3, "FuseA": 4, "SBrkr": 5},
                   "KitchenQual": {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "Functional":  {"Sal":0, "Sev":1, "Maj2": 2, "Maj1":
3, "Mod": 4, "Min2":5, "Min1":6, 'Typ':7},
                   "FireplaceQu": {"NA":0, "Po":1, "Fa": 2, "TA": 3, "G
d": 4, "Ex":5},
                   "PoolQC":      {"NA":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "Fence":       {"NA":0, "MnWw": 1, "GdWo": 2, "MnPrv":
3, "GdPrv":4},
                   }

  df_X = df[useful_fields].copy()
  df_X.replace(cleanup_nums, inplace=True)  # convert continous categori
al variables to numerical
  df_Y = df[target_field].copy()

  x = df_X.to_numpy().astype(np.float32)
  y = df_Y.to_numpy().astype(np.float32)

  if price_max>0:
    idxs = y<price_max
    x = x[idxs]
```

In [7]:
```
x, y, df = house_prices_dataset(return_df=True)
print(x.shape, y.shape)
df.head()
```

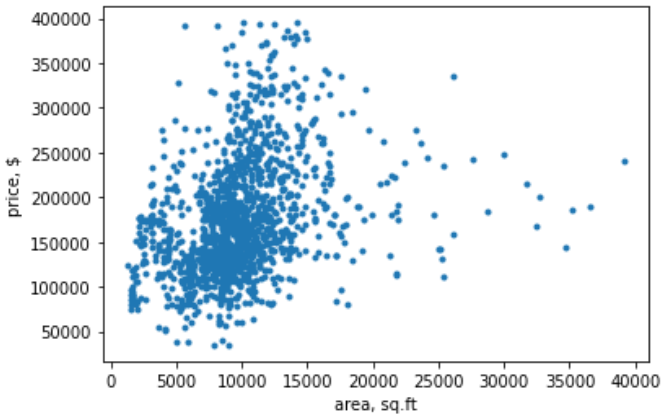
```
(1420, 24) (1420,)
```

Out[7]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Util
0	1	60	RL	65	8450	Pave	NA	Reg	Lvl	AllP
1	2	20	RL	80	9600	Pave	NA	Reg	Lvl	AllP
2	3	60	RL	68	11250	Pave	NA	IR1	Lvl	AllP
3	4	70	RL	60	9550	Pave	NA	IR1	Lvl	AllP
4	5	60	RL	84	14260	Pave	NA	IR1	Lvl	AllP

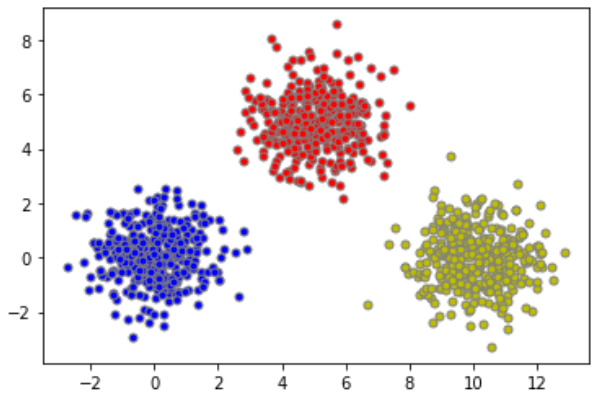
5 rows × 81 columns

In [8]:
```
plt.plot(x[:, 0], y, '.')
plt.xlabel('area, sq.ft')
plt.ylabel('price, $');
```



3. Blobs

In [9]:
```
x, y = make_blobs(n_samples=1000, centers=[[0,0], [5,5], [10, 0]])
colors = "bry"
for i, color in enumerate(colors):
    idx = y == i
    plt.scatter(x[idx, 0], x[idx, 1], c=color, edgecolor='gray', s=25)
```



4. Fashion MNIST

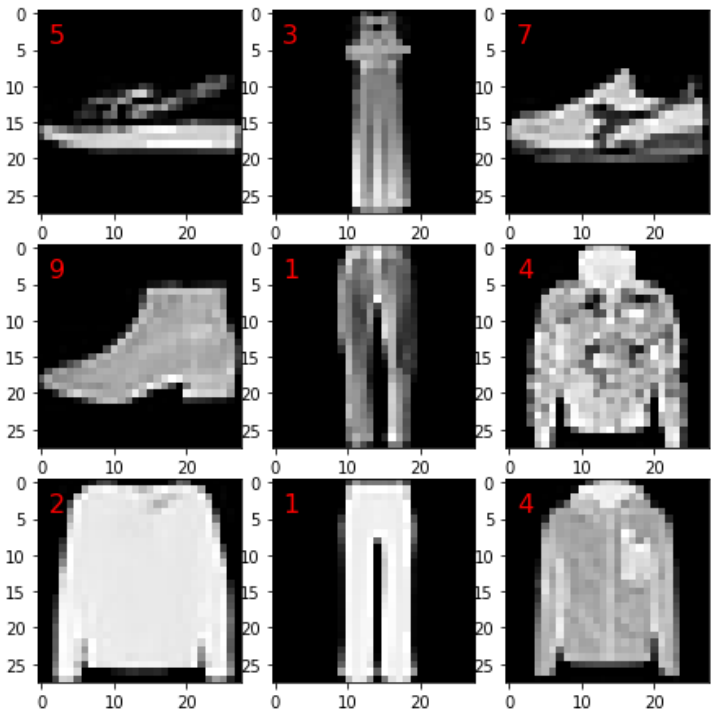
`Fashion-MNIST` is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. (from https://github.com/zalandoresearch/fashion-mnist (https://github.com/zalandoresearch/fashion-mnist))

In [10]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnis
t.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
32768/29515 [================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

Let's chech few samples:

In [11]:
```
n = 3
fig, ax = plt.subplots(n, n, figsize=(2*n, 2*n))
ax = [ax_xy for ax_y in ax for ax_xy in ax_y]
for axi, im_idx in zip(ax, np.random.choice(len(train_images), n**2)):
  im = train_images[im_idx]
  im_class = train_labels[im_idx]
  axi.imshow(im, cmap='gray')
  axi.text(1, 4, f'{im_class}', color='r', size=16)
plt.tight_layout(0,0,0)
```



Each training and test example is assigned to one of the following labels:

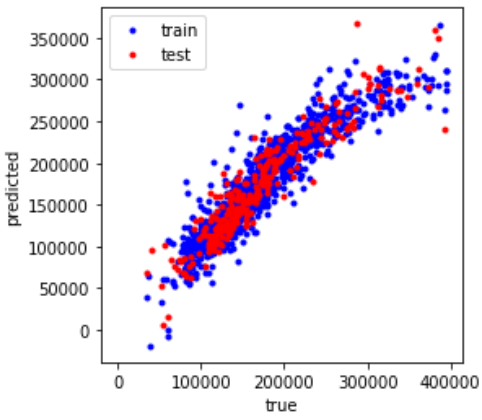
Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

EXERCISE 1.

In [12]:
```
# Solution:
x, y = house_prices_dataset()

# 1. make train/test split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

# 2. fit the model
reg = linear_model.LinearRegression()
reg.fit(x_train, y_train)

# 3. evaluate MSE, MAD, and R2 on train and test datasets
#prediction:
y_p_train = reg.predict(x_train)
y_p_test = reg.predict(x_test)

# mse
print('train mse =', np.std(y_train - y_p_train))
print('test mse =', np.std(y_test - y_p_test))
# mse
print('train mae =', np.mean(np.abs(y_train - y_p_train)))
print('test mae =', np.mean(np.abs(y_test - y_p_test)))
# R2
print('train R2 =', reg.score(x_train, y_train))
print('test R2 =', reg.score(x_test, y_test))

# 4. plot y vs predicted y for test and train parts
plt.plot(y_train, y_p_train, 'b.', label='train')
plt.plot(y_test, y_p_test, 'r.', label='test')

plt.plot([0], [0], 'w.')  # dummy to have origin
plt.xlabel('true')
plt.ylabel('predicted')
plt.gca().set_aspect('equal')
plt.legend()
```

```
train mse = 24834.885
test mse = 24293.932
train mae = 18221.04
test mae = 17279.08
train R2 = 0.8534532342221349
test R2 = 0.8706671727120681
```

Out[12]: <matplotlib.legend.Legend at 0x7f089d87e278>



EXERCISE 2.

In [0]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnis
t.load_data()
```

We will reshape 2-d images to 1-d arrays for use in scikit-learn:

In [0]:
```
n_train = len(train_labels)
x_train = train_images.reshape((n_train, -1))
y_train = train_labels

n_test = len(test_labels)
x_test = test_images.reshape((n_test, -1))
y_test = test_labels
```

Now use a multinomial logistic regression classifier, and measure the accuracy:

In [15]:
```
#solution
# 1. Create classifier
multi_class = 'multinomial'
clf = linear_model.LogisticRegression(solver='sag', max_iter=20,
                                      multi_class=multi_class)

# 2. fit the model
t1 = timer()
clf.fit(x_train, y_train)
t2 = timer()
print ('training time: %.1fs'%(t2-t1))

# 3. evaluate accuracy on train and test datasets
print("training score : %.3f" % (clf.score(x_train, y_train)))
print("test score : %.3f" % (clf.score(x_test, y_test)))
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_sag.py:330:
ConvergenceWarning: The max_iter was reached which means the coef_ did no
t converge
  "the coef_ did not converge", ConvergenceWarning)
```

```
training time: 40.6s
training score : 0.874
test score : 0.843
```

Data Science Fundamentals 5

Basic introduction on how to perform typical machine learning tasks with Python.

Prepared by Mykhailo Vladymyrov & Aris Marcolongo, Science IT Support, University Of Bern, 2020

Solutions to Part 2.

In [0]:
```
from sklearn import tree
from sklearn import ensemble

from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

from matplotlib import  pyplot as plt
from time import time as timer
from imageio import imread
import pandas as pd
import numpy as np
import os

from sklearn.manifold import TSNE
import umap

import tensorflow as tf

%matplotlib inline
from matplotlib import animation
from IPython.display import HTML
```

In [0]:
```
if not os.path.exists('data'):
    path = os.path.abspath('.')+'/colab_material.tgz'
    tf.keras.utils.get_file(path, 'https://github.com/neworldemancer/DSF
5/raw/master/colab_material.tgz')
    !tar -xvzf colab_material.tgz > /dev/null 2>&1
```

In [0]:
```
from utils.routines import *
```

Datasets

In this course we will use several synthetic and real-world datasets to ilustrate the behavior of the models and excercise our skills.

1. House prices

Subset of the the hous pricess kaggle dataset: https://www.kaggle.com/c/house-prices-advanced-regression-techniques (https://www.kaggle.com/c/house-prices-advanced-regression-techniques)

In [0]:
```
def house_prices_dataset(return_df=False, price_max=400000, area_max=400
00):
  path = 'data/train.csv'
  df = pd.read_csv(path, na_values="NaN", keep_default_na=False)

  useful_fields = ['LotArea',
                   'Utilities', 'OverallQual', 'OverallCond',
                   'YearBuilt', 'YearRemodAdd', 'ExterQual', 'ExterCond',
                   'HeatingQC', 'CentralAir', 'Electrical',
                   '1stFlrSF', '2ndFlrSF','GrLivArea',
                   'FullBath', 'HalfBath',
                   'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual', 'TotRms
AbvGrd',
                   'Functional','PoolArea',
                   'YrSold', 'MoSold'
                   ]
  target_field = 'SalePrice'

  cleanup_nums = {"Street":      {"Grvl": 0, "Pave": 1},
                   "LotFrontage": {"NA":0},
                   "Alley":       {"NA":0, "Grvl": 1, "Pave": 2},
                   "LotShape":    {"IR3":0, "IR2": 1, "IR1": 2, "Reg":3},
                   "Utilities":   {"ELO":0, "NoSeWa": 1, "NoSewr": 2, "Al
lPub": 3},
                   "LandSlope":   {"Sev":0, "Mod": 1, "Gtl": 3},
                   "ExterQual":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "ExterCond":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "BsmtQual":    {"NA":0, "Po":1, "Fa": 2, "TA": 3, "G
d": 4, "Ex":5},
                   "BsmtCond":    {"NA":0, "Po":1, "Fa": 2, "TA": 3, "G
d": 4, "Ex":5},
                   "BsmtExposure":{"NA":0, "No":1, "Mn": 2, "Av": 3, "G
d": 4},
                   "BsmtFinType1":{"NA":0, "Unf":1, "LwQ": 2, "Rec": 3, "
BLQ": 4, "ALQ":5, "GLQ":6},
                   "BsmtFinType2":{"NA":0, "Unf":1, "LwQ": 2, "Rec": 3, "
BLQ": 4, "ALQ":5, "GLQ":6},
                   "HeatingQC":   {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "CentralAir":  {"N":0, "Y": 1},
                   "Electrical":  {"NA":0, "Mix":1, "FuseP":2, "FuseF":
3, "FuseA": 4, "SBrkr": 5},
                   "KitchenQual": {"Po":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "Functional":  {"Sal":0, "Sev":1, "Maj2": 2, "Maj1":
3, "Mod": 4, "Min2":5, "Min1":6, 'Typ':7},
                   "FireplaceQu": {"NA":0, "Po":1, "Fa": 2, "TA": 3, "G
d": 4, "Ex":5},
                   "PoolQC":      {"NA":0, "Fa": 1, "TA": 2, "Gd": 3, "E
x":4},
                   "Fence":       {"NA":0, "MnWw": 1, "GdWo": 2, "MnPrv":
3, "GdPrv":4},
                   }

  df_X = df[useful_fields].copy()
  df_X.replace(cleanup_nums, inplace=True)  # convert continous categori
al variables to numerical
  df_Y = df[target_field].copy()

  x = df_X.to_numpy().astype(np.float32)
  y = df_Y.to_numpy().astype(np.float32)

  if price_max>0:
    idxs = y<price_max
    x = x[idxs]
    y = y[idxs]
```

In [0]:
```
def house_prices_dataset_normed():
    x, y = house_prices_dataset(return_df=False, price_max=-1, area_max=
-1)

    scaler=StandardScaler()
    features_scaled=scaler.fit_transform(x)

    return features_scaled
```

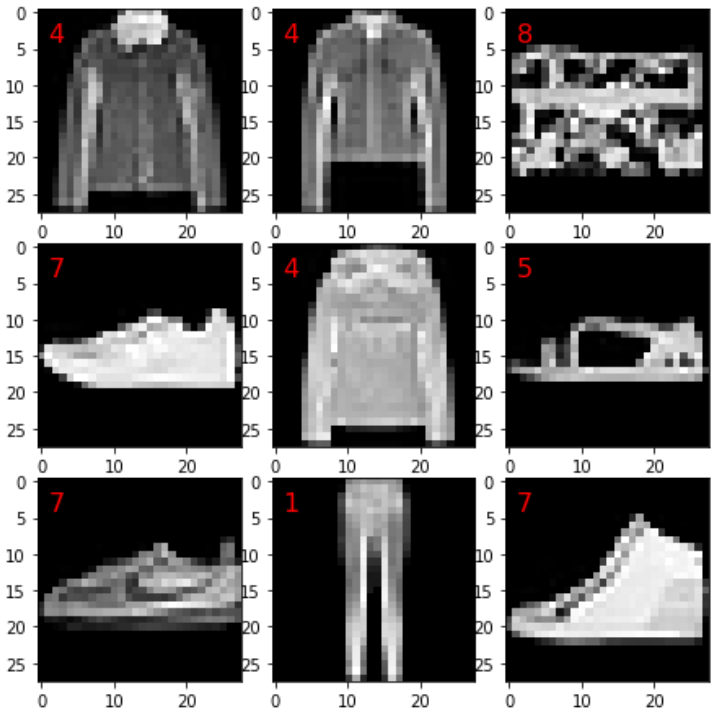
2. Fashion MNIST

Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. (from https://github.com/zalandoresearch/fashion-mnist (https://github.com/zalandoresearch/fashion-mnist))

In [0]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnis
t.load_data()
```

Let's chech few samples:

In [7]:
```
n = 3
fig, ax = plt.subplots(n, n, figsize=(2*n, 2*n))
ax = [ax_xy for ax_y in ax for ax_xy in ax_y]
for axi, im_idx in zip(ax, np.random.choice(len(train_images), n**2)):
  im = train_images[im_idx]
  im_class = train_labels[im_idx]
  axi.imshow(im, cmap='gray')
  axi.text(1, 4, f'{im_class}', color='r', size=16)
plt.tight_layout(0,0,0)
```



Each training and test example is assigned to one of the following labels:

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

EXERCISE 1 : Random forest classifier for FMNIST

In [0]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnis
t.load_data()

n = len(train_labels)
x_train = train_images.reshape((n, -1))
y_train = train_labels

n_test = len(test_labels)
x_test = test_images.reshape((n_test, -1))
y_test = test_labels
```

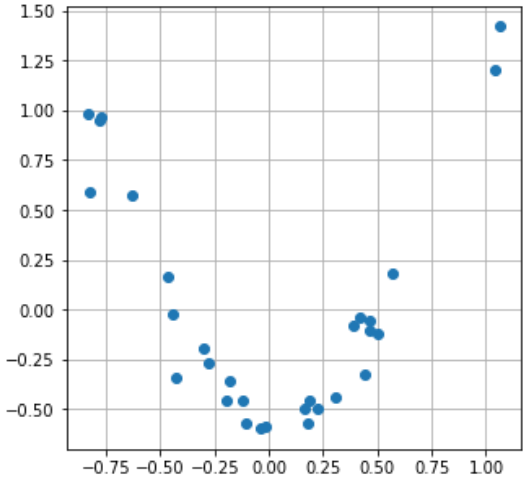
In [9]:
```
# 1. Create classifier. As the number of features is big, use bigger tre
e depth
# (max_depth parameter). in the same time to reduce variance, one should
limit the
# total number of tree leafes. (max_leaf_nodes parameter)
# Try different number of estimators (n_estimators)

n_est = 20

dtc = ensemble.RandomForestClassifier(max_depth=700, n_estimators=n_est,
max_leaf_nodes=500)

# 2. fit the model
t1 = timer()
dtc.fit(x_train, y_train)
t2 = timer()
print ('training time: %.1fs'%(t2-t1))

# 3. Inspect training and test accuracy
print("training score : %.3f (n_est=%d)" % (dtc.score(x_train, y_train),
n_est))
print("test score : %.3f (n_est=%d)" % (dtc.score(x_test, y_test), n_es
t))
```

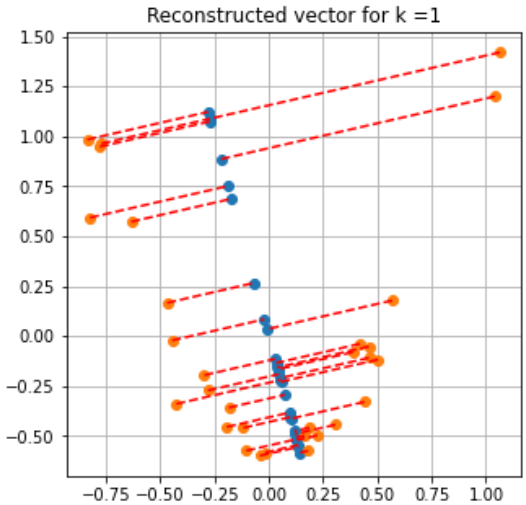
```
training time: 13.0s
training score : 0.893 (n_est=20)
test score : 0.855 (n_est=20)
```

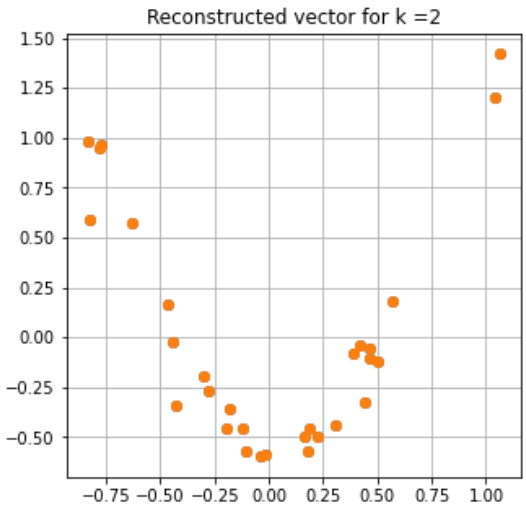
EXERCISE 2 : PCA with a non-linear data-set

In [10]:
```
# 1. Load the data using the function load_ex1_data_pca() , check the di
mensionality of the data and plot them.
# Solution:

data = load_ex1_data_pca()

n_samples,n_dim=data.shape

print('We have ',n_samples, 'samples of dimension ', n_dim)

plt.figure(figsize=((5,5)))
plt.grid()
plt.plot(data[:,0],data[:,1],'o')

# 2. Define a PCA object and perform the PCA fitting.
pca=PCA()
pca.fit(data)

# 3. Check the explained variance ratio and select best number of compon
ents.

print('Explained variance ratio: ' ,pca.explained_variance_ratio_)

# 4. Plot the reconstructed vectors for different values of k.

scores=pca.transform(data)
for k in range(1,3):
    res=np.dot(scores[:,:k], pca.components_[:k,:])

    plt.figure(figsize=((5,5)))
    plt.title('Reconstructed vector for k ='+ str(k))
    plt.plot(res[:,0],res[:,1],'o')
    plt.plot(data[:,0],data[:,1],'o')

    for a,b,c,d in zip(data[:,0],data[:,1],res[:,0],res[:,1]) :
        plt.plot([a,c],[b,d],'-', linestyle = '--', color='red')

    plt.grid()

# Message: if the manyfold is non-linear one is forced to use a high num
ber of principal components.
# For example, in the parabola example the projection for k=1 looks bad.
But using too many principal components
# the reconstructed vectors are almost equal to the original ones (for k
=2 we get exact reconstruction in our example )
# and the advanteges of dimensionality reduction are lost. This is a gen
eral pattern.
```

```
We have  30 samples of dimension  2
Explained variance ratio:  [0.57388642 0.42611358]
```





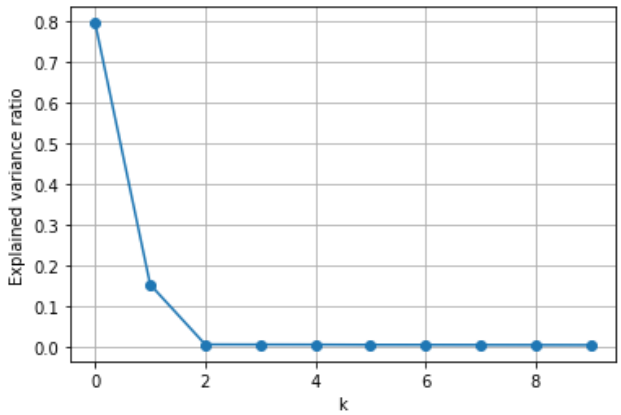


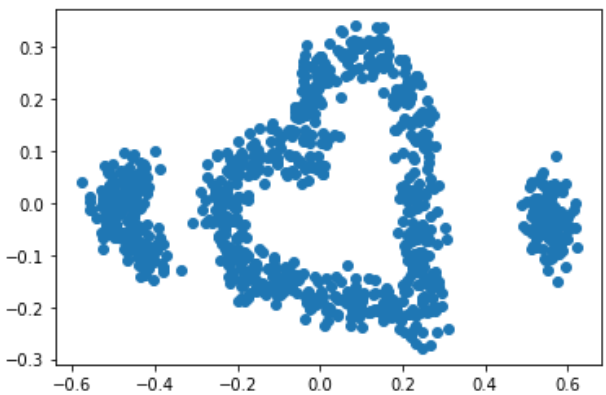
EXERCISE 3 : Find the hidden drawing.

In [11]:
```
# 1. Load the data using the function load_ex2_data_pca(seed=1235) , che
ck the dimensionality of the data and plot them.

data= load_ex2_data_pca(seed=1235)
n_samples,n_dim=data.shape
print('We have ',n_samples, 'samples of dimension ', n_dim)

# 2. Define a PCA object and perform the PCA fitting.
pca=PCA()
pca.fit(data)

# 3. Check the explained variance ratio and select best number of compon
ents.
plt.figure()
print('Explained variance ratio: ' ,pca.explained_variance_ratio_)
plt.plot(pca.explained_variance_ratio_,'-o')
plt.xlabel('k')
plt.ylabel('Explained variance ratio')
plt.grid()

# 4. Plot the reconstructed vectors for the best value of k.
plt.figure()
k=2
data_transformed=pca.transform(data)
plt.plot(data_transformed[:,0],data_transformed[:,1],'o')

# **Message:** Sometimes the data hides simple patterns in high dimensio
nal datasets.
# PCA can be very useful in identifying these patterns.
```

```
We have  961 samples of dimension  10
Explained variance ratio:  [0.79700994 0.15407412 0.00688753 0.00667879
0.00652795 0.00605738
 0.00596107 0.00576693 0.00561825 0.00541804]
```

Out[11]: [<matplotlib.lines.Line2D at 0x7f44fa3c3f28>]





Data Science Fundamentals 5

Basic introduction on how to perform typical machine learning tasks with Python.

Prepared by Mykhailo Vladymyrov & Aris Marcolongo, Science IT Support, University Of Bern, 2020

Solutions to Part 3.

In [0]:
```
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.mixture import GaussianMixture

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

from matplotlib import  pyplot as plt
import numpy as np
import pandas as pd
from imageio import imread
from time import time as timer
import os

import tensorflow as tf

%matplotlib inline
from matplotlib import animation
from IPython.display import HTML

import umap
from scipy.stats import entropy
```

In [2]:
```
if not os.path.exists('data'):
    path = os.path.abspath('.')+'/colab_material.tgz'
    tf.keras.utils.get_file(path, 'https://github.com/neworldemancer/DSF
5/raw/master/colab_material.tgz')
    !tar -xvzf colab_material.tgz > /dev/null 2>&1
```

```
Downloading data from https://github.com/neworldemancer/DSF5/raw/master/c
olab_material.tgz
98304/96847 [==============================] - 0s 0us/step
```

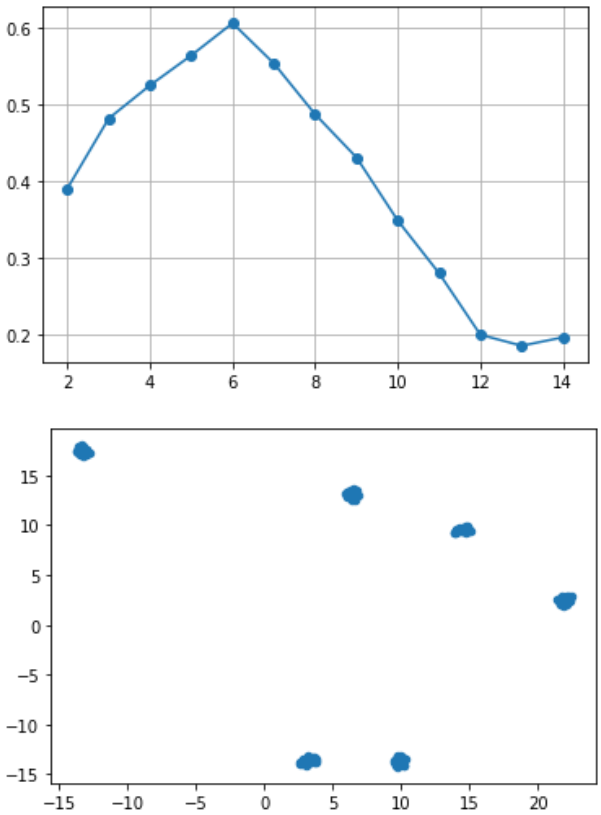
In [0]:
```
from utils.routines import *
```

EXERCISE 1: Discover the number of Gaussians

In [4]:
```
### In this exercise you are given the dataset points, consisting of hig
h-dimensional data. It was built taking random
#samples from a number k of multimensional gaussians. The data is theref
ore made of k clusters but, being
#very high dimensional, you cannot visualize it. Your task it too use K-
means combined with the Silouhette
#score to find the number of k.

# 1. Load the data using the function load_ex1_data_clust() , check the
dimensionality of the data.

points=load_ex1_data_clust()

# 2. Fix a number of clusters k and define a KMeans clusterer object. Pe
rform the fitting and compute the Silhouette score.
# Save the results on a list.

plt.figure()
scores=[]
for itrial in range(2,15):
    print('Number of clusters considered: ',itrial)
    clusterer = KMeans(n_clusters=itrial, random_state=10)
    cluster_labels = clusterer.fit_predict(points)
    score=silhouette_score(points,cluster_labels)
    scores.append(score)

# 3. Plot the Silhouette scores as a function ok k? What is the number o
f clusters ?
plt.grid()
plt.plot(np.arange(len(scores))+2,np.array(scores),'-o')

# 4. Optional. Check the result that you found via umap.
plt.figure()
umap_model = umap.UMAP(random_state=1711)
umap_gs = umap_model.fit_transform(points)
plt.scatter(umap_gs[:, 0], umap_gs [:, 1], s=20)
```

```
Number of clusters considered:  2
Number of clusters considered:  3
Number of clusters considered:  4
Number of clusters considered:  5
Number of clusters considered:  6
Number of clusters considered:  7
Number of clusters considered:  8
Number of clusters considered:  9
Number of clusters considered:  10
Number of clusters considered:  11
Number of clusters considered:  12
Number of clusters considered:  13
Number of clusters considered:  14
```

```
Out[4]: <matplotlib.collections.PathCollection at 0x7fb682fd9ac8>
```



EXERCISE 2: Predict the good using K-Means

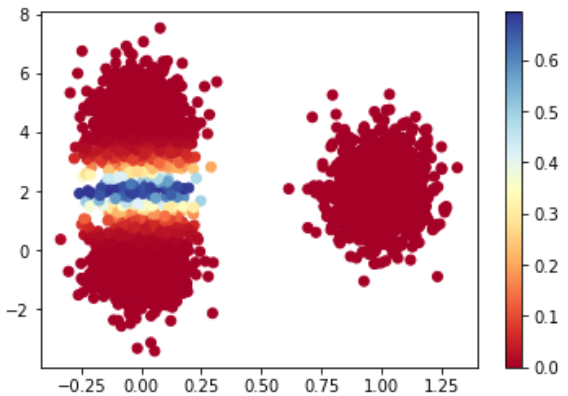
In [5]:
```
#In this exercise you are asked to use the clustering performed by K-mea
ns to predict the good in the f-mnist dataset.
#Here we are using the clustering as a preprocessing for a supervised ta
sk. We need therefore the correct labels
#on a training set and #o test the result on a test set:

# 1. Load the dataset.

fmnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fmnist.load_d
ata()

X_train=train_images[:5000,:].reshape(5000,-1)
y_train=train_labels[:5000]

X_test=test_images[:1000,:].reshape(1000,-1)
y_test=test_labels[:1000]

# 2. FITTING STEP: The fitting step consists first here in the computati
on of the cluster center, which was done during
# the presentation. Second, to each cluster center we need than to assig
n a good-label, which will be given by the
# majority class of the sample belonging to that cluster.

def most_common(nclusters, supervised_labels, cluster_labels):

    """
    Args:
    - nclusters : the number of clusteres
    - supervised_labels : for each sample, the labelling provided by the
training data ( e.g. in y_train or y_test)
    - cluster_labels : for each good, the cluster it was assigned by K-M
eans using the predict method of the Kmeans object

    Returns:
    - a list "assignement" of lengths nclusters, where assignement[i] is
the majority class of the i-cluster
    """

    assignement=[]
    for icluster in range(nclusters):
        indices=list(supervised_labels[cluster_labels==icluster])
        try:
            chosen= max(set(indices), key=indices.count)
        except ValueError :
            print('Em')
            chosen=1
        assignement.append(chosen)

    return assignement


clusterer = KMeans(n_clusters=10, random_state=10)

clusterer.fit(X_train)

cluster_labels = clusterer.predict(X_train)
assignement=most_common(10, y_train, cluster_labels)

print(assignement)

print('Training set')

cluster_labels = clusterer.predict(X_train)

new_labels=[assignement[i] for i in cluster_labels]

cm=metrics.confusion_matrix( y_train, new_labels)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
32768/29515 [================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
[4, 6, 9, 3, 7, 0, 8, 8, 9, 1]
Training set
0.5492
[[248   6   0  62   6   0 131   1   3   0]
 [  1 482   0  37   8   0  28   0   0   0]
 [  5   0   0  14 324   0 153   1   7   0]
 [  7 165   0 223   5   0 101   0   0   0]
 [  2   2   0  77 333   0  68   0   6   0]
 [  0   0   0   0   0   0 111 291   2  89]
 [ 68   3   0  52 181   0 178   1   9   1]
 [  0   0   0   0   0   0   0 410   0 102]
 [  0   0   0   7  16   0  46  25 378  18]
 [  0   0   0   0   0   0  10   2   0 494]]

Test set
0.558
[[60  0  0 12  4  0 31  0  0  0]
 [ 0 94  0  6  0  0  5  0  0  0]
 [ 2  0  0  2 71  0 35  0  1  0]
 [ 2 25  0 48  1  0 17  0  0  0]
 [ 0  1  0 23 79  0 12  0  0  0]
 [ 0  0  0  0  0  0 17 55  0 15]
 [12  1  0  9 32  0 41  0  2  0]
 [ 0  0  0  0  0  0  0 78  0 17]
 [ 0  0  0  1  5  0 14  3 67  5]
 [ 0  0  0  0  0  0  1  3  0 91]]
[4, 6, 9, 3, 7, 0, 8, 8, 9, 1]
Test set with 10 clusters
0.558
[[60  0  0 12  4  0 31  0  0  0]
 [ 0 94  0  6  0  0  5  0  0  0]
 [ 2  0  0  2 71  0 35  0  1  0]
 [ 2 25  0 48  1  0 17  0  0  0]
 [ 0  1  0 23 79  0 12  0  0  0]
 [ 0  0  0  0  0  0 17 55  0 15]
 [12  1  0  9 32  0 41  0  2  0]
 [ 0  0  0  0  0  0  0 78  0 17]
 [ 0  0  0  1  5  0 14  3 67  5]
 [ 0  0  0  0  0  0  1  3  0 91]]
[2, 3, 5, 1, 0, 5, 1, 4, 2, 9, 2, 3, 9, 2, 8, 8, 9, 8, 7, 0]
Test set with 20 clusters
0.648
[[87  0  6 11  2  0  0  0  1  0]
 [ 5 93  1  6  0  0  0  0  0  0]
 [15  0 85  1 10  0  0  0  0  0]
 [18 16  3 55  1  0  0  0  0  0]
 [ 6  0 57 17 35  0  0  0  0  0]
 [ 0  0  0  0  0 73  0  4  0 10]
 [34  0 36  9 16  2  0  0  0  0]
 [ 0  0  0  0  0 14  0 59  0 22]
 [ 6  0  7  1  1  8  0  0 69  3]
 [ 0  0  0  0  0  3  0  0  0 92]]
[6, 0, 7, 3, 9, 1, 4, 9, 5, 4, 2, 8, 8, 1, 1, 9, 0, 2, 3, 4, 5, 7, 3, 8,
6, 5, 3, 9, 5, 2]
Test set with 30 clusters
0.652
```

EXERCISE 3 : Find the prediction uncertainty

In [6]:
```
#In this exercise you need to load the dataset used to present K-means (
def km_load_th1() ) or the one used to discuss
# the Gaussian mixtures model ( def km_load_th1() ).
#As discussed, applying a fitting based on gaussian mixtures you can not
only predict the cluster label for each point,
#but also a probability distribution over the clusters.

#From this probability distribution, you can compute for each point the
entropy of the corresponging
#distribution (using for example scipy.stats.entropy) as an estimation o
f the undertainty of the prediction.
#Your task is to plot the data-cloud with a color proportional to the un
certainty of the cluster assignement.

# In detail you shoud:
# 1. Instantiate a GaussianMixture object with the number of clusters th
at you expect
# 2. fit the object on the dataset with the fit method

from scipy.stats import entropy
points=gm_load_th1()

plt.figure()
clf = GaussianMixture(n_components=3, covariance_type='full')

clf.fit(points)

# 3. compute the cluster probabilities using the method predict_proba. T
his will return a matrix of
# dimension npoints x nclusters
# 4. use the entropy function ( from scipy.stats import entropy ) to eva
luate for each point the uncertainty of the
#prediction

cluster_labels_prob=clf.predict_proba(points)

entropies=[]
for point in range(len(cluster_labels_prob)):
    entropies.append(entropy(cluster_labels_prob[point]))

# 5. Plot the points colored accordingly to their uncertanty.

cm = plt.cm.get_cmap('RdYlBu')
sc = plt.scatter(points[:,0], points[:,1], c=entropies, cmap=cm)
plt.colorbar(sc)
```

Out[6]: <matplotlib.colorbar.Colorbar at 0x7fb680f58e48>



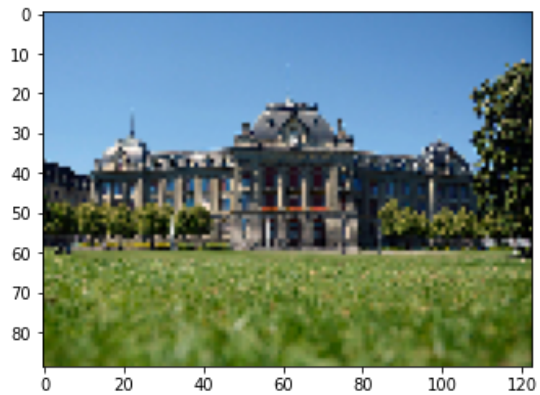
EXERCISE 4.

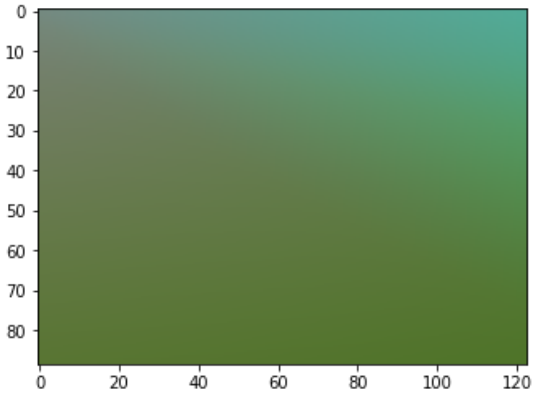
Load some image, downscale to a similar resolution, and train a deeper model, for example 5 layers, more parameters in widest layers.

In [7]:
```
# solution

# 1. Load your image
image_big = imread('https://www.unibe.ch/unibe/portal/content/carousel/s
howitem940548/UniBE_Coronavirus_612p_eng.jpg')
image_big = image_big[...,0:3]/255
plt.imshow(image_big)

image = image_big[::5, ::5]
plt.imshow(image)
plt.show()

h, w, c = image.shape
X = np.meshgrid(np.linspace(0, 1, w), np.linspace(0, 1, h))
X = np.stack(X, axis=-1).reshape((-1, 2))

Y = image.reshape((-1, c))
X.shape, Y.shape


# 2. build a deeper model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(2,)),
  tf.keras.layers.Dense(32, activation=tf.keras.layers.LeakyReLU()),
  tf.keras.layers.Dense(512, activation=tf.keras.layers.LeakyReLU()),
  tf.keras.layers.Dense(64, activation=tf.keras.layers.LeakyReLU()),
  tf.keras.layers.Dense(16, activation=tf.keras.layers.LeakyReLU()),
  tf.keras.layers.Dense(8, activation=tf.keras.layers.LeakyReLU()),
  tf.keras.layers.Dense(c, activation='sigmoid'),
])
model.compile(optimizer='adam',
              loss='mae',
              metrics=['mse'])
model.summary()

# 3. inspect the evolution

ims = []
n_ep_tot = 0
for i in range(200):
  if i % 10 == 0:
    print(f'epoch {i}', end='\n')
  ne = (2 if (i<50) else (20 if (i<100) else (200 if (i<150) else 100
0)))
  model.fit(X, Y, epochs=ne, batch_size=1*2048, verbose=0)

  Y_p = model.predict(X)
  Y_p = Y_p.reshape((h, w, c))
  ims.append(Y_p)
  n_ep_tot += ne

print(f'total numer of epochs trained:{n_ep_tot}')

plt.rcParams["animation.html"] = "jshtml"  # for matplotlib 2.1 and abov
e, uses JavaScript
fig = plt.figure()
im = plt.imshow(ims[0])

def animate(i):
    img = ims[i]
    im.set_data(img)
    return im

ani = animation.FuncAnimation(fig, animate, frames=len(ims))

ani
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 2)                 0
_________________________________________________________________
dense (Dense)                (None, 32)                96
_________________________________________________________________
dense_1 (Dense)              (None, 512)               16896
_________________________________________________________________
dense_2 (Dense)              (None, 64)                32832
_________________________________________________________________
dense_3 (Dense)              (None, 16)                1040
_________________________________________________________________
dense_4 (Dense)              (None, 8)                 136
_________________________________________________________________
dense_5 (Dense)              (None, 3)                 27
=================================================================
Total params: 51,027
Trainable params: 51,027
Non-trainable params: 0
_________________________________________________________________
epoch 0
epoch 10
epoch 20
epoch 30
epoch 40
epoch 50
epoch 60
epoch 70
epoch 80
epoch 90
epoch 100
epoch 110
epoch 120
epoch 130
epoch 140
epoch 150
epoch 160
epoch 170
epoch 180
epoch 190
total numer of epochs trained:61100
```

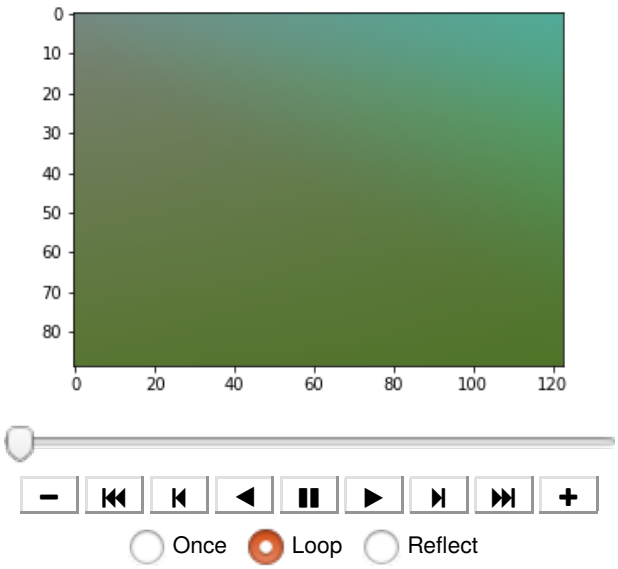
Out[7]: <matplotlib.animation.FuncAnimation at 0x7fb686a04198>




0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
120

In [17]:

Out[17]:


0
10
20
30
40
50
60
70
80
0
20
40
60
80
100
120
Once
Loop
Reflect

Data Science Fundamentals 5

Basic introduction on how to perform typical machine learning tasks with Python.

Prepared by Mykhailo Vladymyrov & Aris Marcolongo, Science IT Support, University Of Bern, 2020

Solutions to Part 4.

In [0]:
```
from matplotlib import  pyplot as plt
import numpy as np
from imageio import imread
import pandas as pd
from time import time as timer

import tensorflow as tf

%matplotlib inline
from matplotlib import animation
from IPython.display import HTML
```

EXERCISE 1: Train deeper network

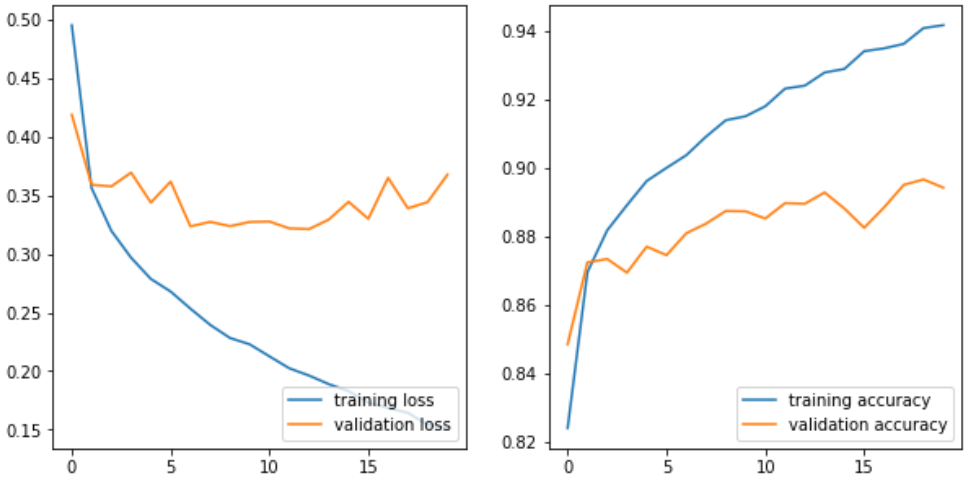
Make a deeper model, with wider layers. Remember to `'softmax'` activation in the last layer, as required for the classification task to encode pseudoprobabilities. In the other layers you could use `'relu'`.

Try to achieve 90% accuracy. Does your model overfit?

In [0]:
```
fashion_mnist = tf.keras.datasets.fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
x_train = x_train/255
x_test = x_test/255

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

In [3]:
```
# 1. create model
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(1024, activation='relu'),
  tf.keras.layers.Dense(256, activation='relu'),
  tf.keras.layers.Dense(64, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])


model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.summary()

# 2. train the model
save_path = 'save/mnist_{epoch}.ckpt'
save_callback = tf.keras.callbacks.ModelCheckpoint(filepath=save_path, s
ave_weights_only=True)

hist = model.fit(x=x_train, y=y_train,
                 epochs=20, batch_size=128,
                 validation_data=(x_test, y_test),
                 callbacks=[save_callback])

# 3. plot the loss and accuracy evolution during training
fig, axs = plt.subplots(1, 2, figsize=(10,5))
axs[0].plot(hist.epoch, hist.history['loss'])
axs[0].plot(hist.epoch, hist.history['val_loss'])
axs[0].legend(('training loss', 'validation loss'), loc='lower right')
axs[1].plot(hist.epoch, hist.history['accuracy'])
axs[1].plot(hist.epoch, hist.history['val_accuracy'])

axs[1].legend(('training accuracy', 'validation accuracy'), loc='lower r
ight')
plt.show()

# 4. evaluate model in best point (before overfitting)
model.load_weights('save/mnist_10.ckpt')
model.evaluate(x_test,  y_test, verbose=2)
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 1024)              803840
_________________________________________________________________
dense_1 (Dense)              (None, 256)               262400
_________________________________________________________________
dense_2 (Dense)              (None, 64)                16448
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 1,083,338
Trainable params: 1,083,338
Non-trainable params: 0
_________________________________________________________________
Epoch 1/20
469/469 [==============================] - 2s 4ms/step - loss: 0.4948 - a
ccuracy: 0.8242 - val_loss: 0.4185 - val_accuracy: 0.8486
Epoch 2/20
469/469 [==============================] - 2s 3ms/step - loss: 0.3561 - a
ccuracy: 0.8697 - val_loss: 0.3588 - val_accuracy: 0.8725
Epoch 3/20
469/469 [==============================] - 2s 3ms/step - loss: 0.3196 - a
ccuracy: 0.8819 - val_loss: 0.3574 - val_accuracy: 0.8735
Epoch 4/20
469/469 [==============================] - 2s 3ms/step - loss: 0.2966 - a
ccuracy: 0.8892 - val_loss: 0.3692 - val_accuracy: 0.8695
Epoch 5/20
469/469 [==============================] - 2s 3ms/step - loss: 0.2786 - a
ccuracy: 0.8963 - val_loss: 0.3436 - val_accuracy: 0.8771
Epoch 6/20
469/469 [==============================] - 2s 3ms/step - loss: 0.2678 - a
ccuracy: 0.9001 - val_loss: 0.3615 - val_accuracy: 0.8746
Epoch 7/20
469/469 [==============================] - 2s 3ms/step - loss: 0.2533 - a
ccuracy: 0.9038 - val_loss: 0.3234 - val_accuracy: 0.8810
Epoch 8/20
469/469 [==============================] - 2s 3ms/step - loss: 0.2394 - a
ccuracy: 0.9092 - val_loss: 0.3272 - val_accuracy: 0.8838
Epoch 9/20
469/469 [==============================] - 2s 3ms/step - loss: 0.2283 - a
ccuracy: 0.9140 - val_loss: 0.3236 - val_accuracy: 0.8875
Epoch 10/20
469/469 [==============================] - 2s 3ms/step - loss: 0.2228 - a
ccuracy: 0.9151 - val_loss: 0.3271 - val_accuracy: 0.8874
Epoch 11/20
469/469 [==============================] - 2s 3ms/step - loss: 0.2126 - a
ccuracy: 0.9180 - val_loss: 0.3275 - val_accuracy: 0.8853
Epoch 12/20
469/469 [==============================] - 1s 3ms/step - loss: 0.2024 - a
ccuracy: 0.9232 - val_loss: 0.3216 - val_accuracy: 0.8898
Epoch 13/20
469/469 [==============================] - 2s 3ms/step - loss: 0.1962 - a
ccuracy: 0.9241 - val_loss: 0.3210 - val_accuracy: 0.8896
Epoch 14/20
469/469 [==============================] - 2s 3ms/step - loss: 0.1889 - a
ccuracy: 0.9279 - val_loss: 0.3292 - val_accuracy: 0.8929
Epoch 15/20
469/469 [==============================] - 2s 3ms/step - loss: 0.1829 - a
ccuracy: 0.9289 - val_loss: 0.3443 - val_accuracy: 0.8882
Epoch 16/20
469/469 [==============================] - 2s 3ms/step - loss: 0.1732 - a
ccuracy: 0.9341 - val_loss: 0.3298 - val_accuracy: 0.8826
Epoch 17/20
469/469 [==============================] - 2s 3ms/step - loss: 0.1686 - a
```




```
313/313 - 1s - loss: 0.3272 - accuracy: 0.8838
```

```
Out[3]: [0.3271946609020233, 0.8838000297546387]
```

```python
import numpy as np

def load_sample_data_pca():

    np.random.seed(3)
    eps=0.5
    n=30
    x=np.random.uniform(-1,1,n)
    y=x+eps*np.random.uniform(-1,1,n)
    x=x-np.mean(x)
    y=y-np.mean(y)
    data=np.vstack((x,y)).transpose()

    return data


def load_multidimensional_data_pca(n_data, n_vec, dim, eps ):

    points=[]
    vectors=np.random.uniform(-1,1,(dim,n_vec))
    for idata in range(n_data):
        alphas=np.random.normal(size=n_vec)
        points.append(np.sum(np.dot(vectors,np.diag(alphas)),axis=1))

    points=np.array(points)
    pert=eps*np.random.normal(size=points.shape)

    return points+pert

def load_ex1_data_pca(eps=0.1):

    np.random.seed(1231)
    n=30
    x=np.random.uniform(-1,1,n)

    y=2*x*x

    epsx=eps*np.random.uniform(-1,1,n)
    epsy=eps*np.random.uniform(-1,1,n)

    x=x+epsx
    y=y+epsy

    x=x-np.mean(x)
    y=y-np.mean(y)

    data=np.vstack((x,y)).transpose()
    return data

def load_ex2_data_pca(dim=10 , eps=0.0 , seed=8, fat=True, eps1=0.05, n_add=30):

    group = np.array([[0.067, 0.21], [0.092, 0.21],
   [0.294, 0.445], [0.227, 0.521], [0.185, 0.597],
   [0.185, 0.689], [0.235, 0.748], [0.319, 0.773],
   [0.387, 0.739], [0.437, 0.672], [0.496, 0.739],
   [0.571, 0.773], [0.639, 0.765], [0.765, 0.924],
   [0.807, 0.933], [0.849, 0.941], [0.118, 0.143], [0.118, 0.176],
   [0.345, 0.378], [0.395, 0.319], [0.437, 0.261],
   [0.496, 0.328], [0.546, 0.395], [0.605, 0.462],
   [0.655, 0.529], [0.697, 0.597], [0.706, 0.664],
   [0.681, 0.723], [0.849, 0.798], [0.857, 0.849],
   [0.866, 0.899]])

    points=[]
    np.random.seed(seed)
    n_data=group.shape[0]
```



```python
    vectors=np.random.uniform(-1,1,(dim,2))
    vectors[:,0]=vectors[:,0]/np.linalg.norm(vectors[:,0])
    vectors[:,1]=vectors[:,1]-np.dot(vectors[:,1],vectors[:,0])*vectors[:,0]
    vectors[:,1]=vectors[:,1]/np.linalg.norm(vectors[:,1])

    for idata in range(n_data):
        points.append(np.sum(np.dot(vectors,np.diag(group[idata,:])),axis=1))

    points=np.array(points)
    pert=eps*np.random.normal(size=points.shape)

    data=points+pert

    data=data-np.mean(data,axis=0)
    if (fat):
        data_added={}
        for iadd in range(n_add):
            data_added[iadd]=np.zeros((n_data,dim))
            for idata in range(n_data):
                noise=np.random.uniform(-eps1,eps1,dim)
                data_added[iadd][idata,:]=data[idata,:]+noise[:]
        for iadd in range(n_add):
            data=np.concatenate([data,data_added[iadd]],axis=0)

    return data

def load_ex1_data_clust(dim=5, n_clusters=6, eps=12.0, dist=20, seed=13124,
n_points=20, return_centers=False):

    np.random.seed(seed)
    centers=np.random.uniform(-dist,dist,(dim,n_clusters))
    cov=np.identity(dim)*eps

    data={}
    for iclust in range(n_clusters):
        data[iclust] = np.random.multivariate_normal(centers[:,iclust], cov,
n_points)

    if (return_centers):
        return centers,np.concatenate([data[iclust] for iclust in data.keys
()],axis=0)
    else:
        return np.concatenate([data[iclust] for iclust in data.keys()],axis=0)

def km_load_th1():
    return load_ex1_data_clust(dim=2, n_clusters=3, eps=12.0, dist=20,
seed=13124, n_points=40, return_centers=False)


def gm_load_th1():
    np.random.seed(1321)
    points1=np.random.multivariate_normal([0,0], [[0.01,0.0],[0.0,1.0]], 1000)
    points2=np.random.multivariate_normal([0,4], [[0.01,0.0],[0.0,1.0]], 1000)
    points3=np.random.multivariate_normal([1,2], [[0.01,0.0],[0.0,1.0]], 1000)

    points=np.concatenate([points1,points2, points3], axis=0)
    return points


def gm_load_th2():
    np.random.seed(14321)
    points1=np.random.multivariate_normal([0,0.0], [[0.01,0.0],[0.0,1.0]], 1000)
    points2=np.random.multivariate_normal([0,0.0], [[1.5,0.0],[0.0,1.0]], 1000)
    points=np.concatenate([points1,points2], axis=0)
    return points
```