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

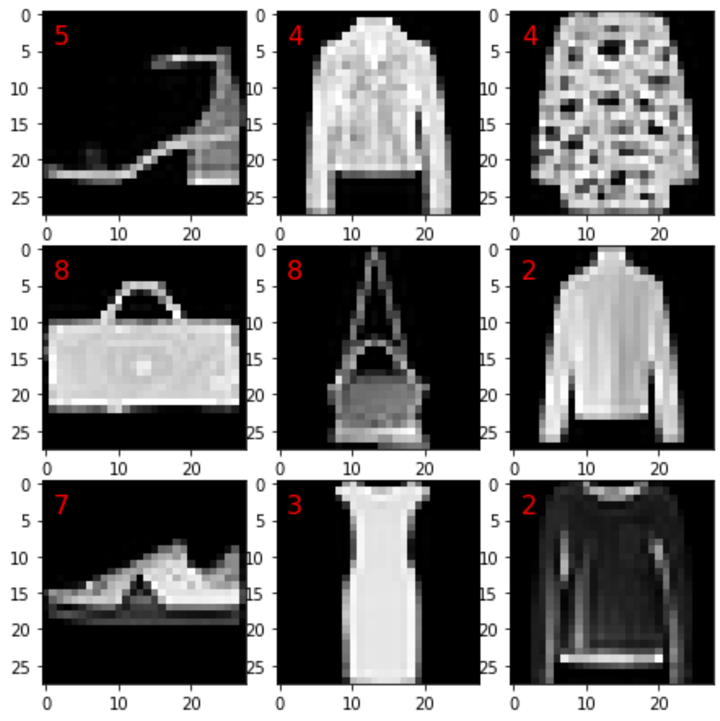
`Fashion-MNIST` is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. (from https://github.com/zalandoresearch/fashion-mnist (https://github.com/zalandoresearch/fashion-mnist))

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

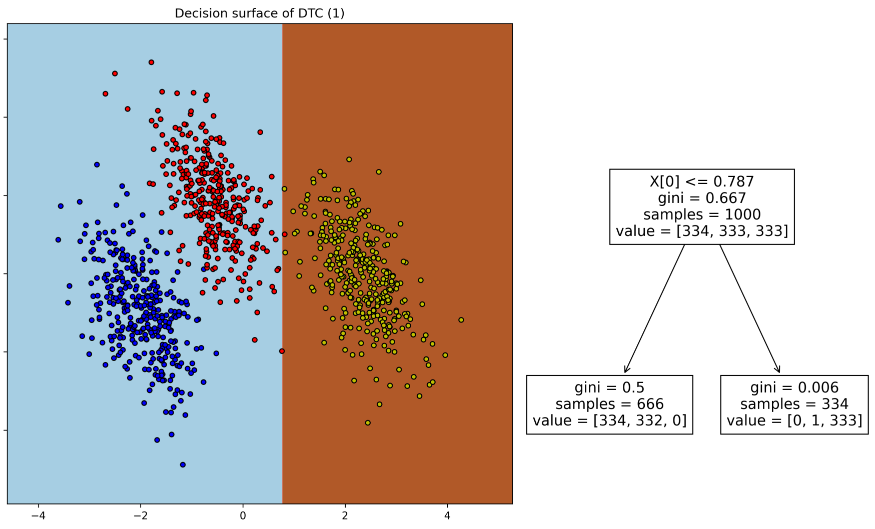
1. Decision Tree

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.

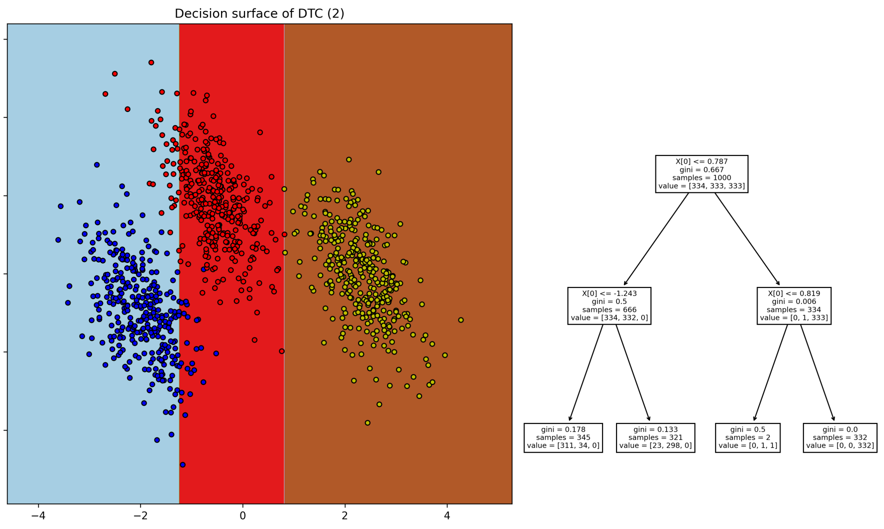
They are fast to train, easily interpretable and require small amount of data.

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()
```

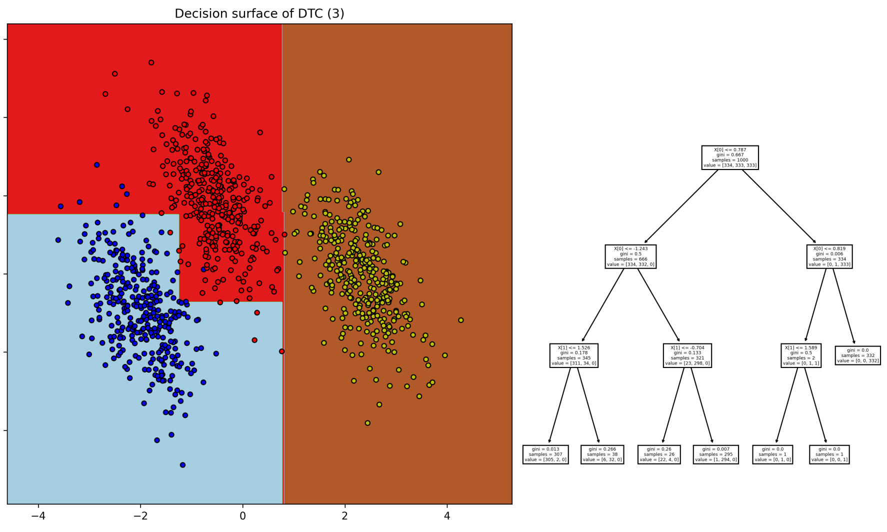
```
training score : 0.667 (depth=1)
```



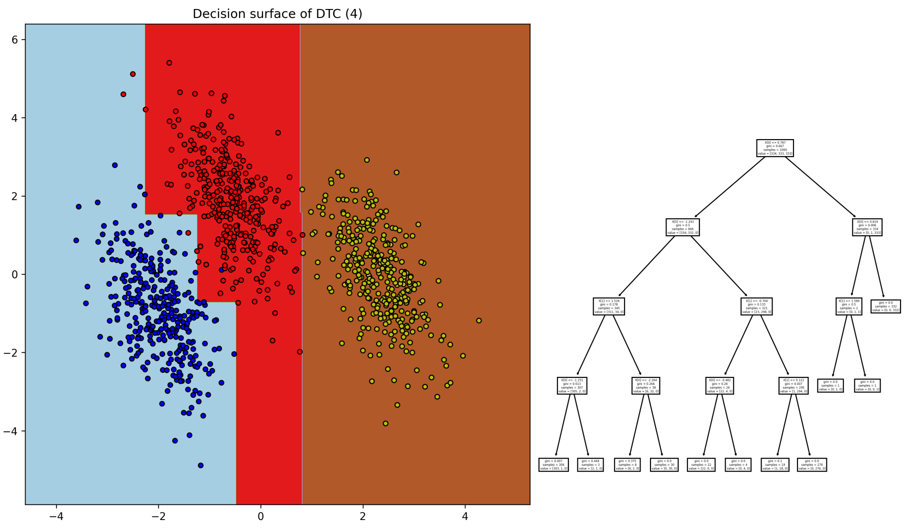
```
training score : 0.942 (depth=2)
```



```
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.

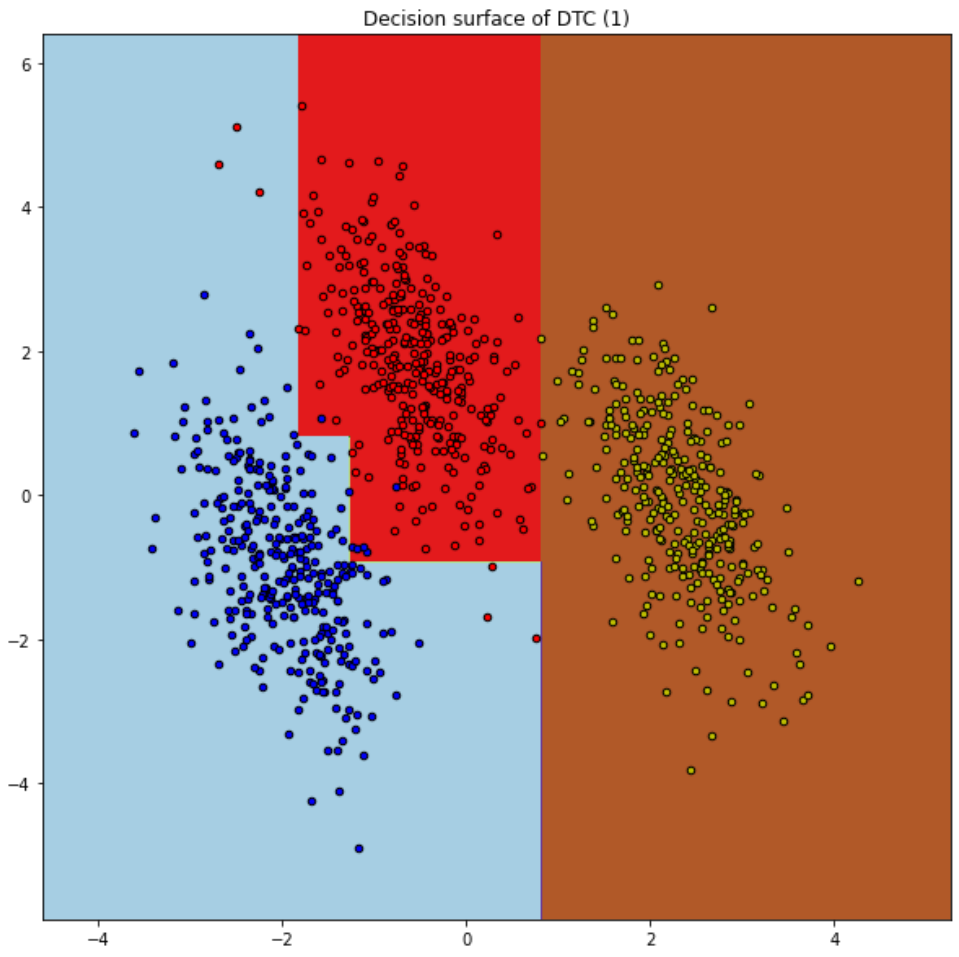
Individual decision trees typically exhibit high variance and tend to overfit. In random forests:

- 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.

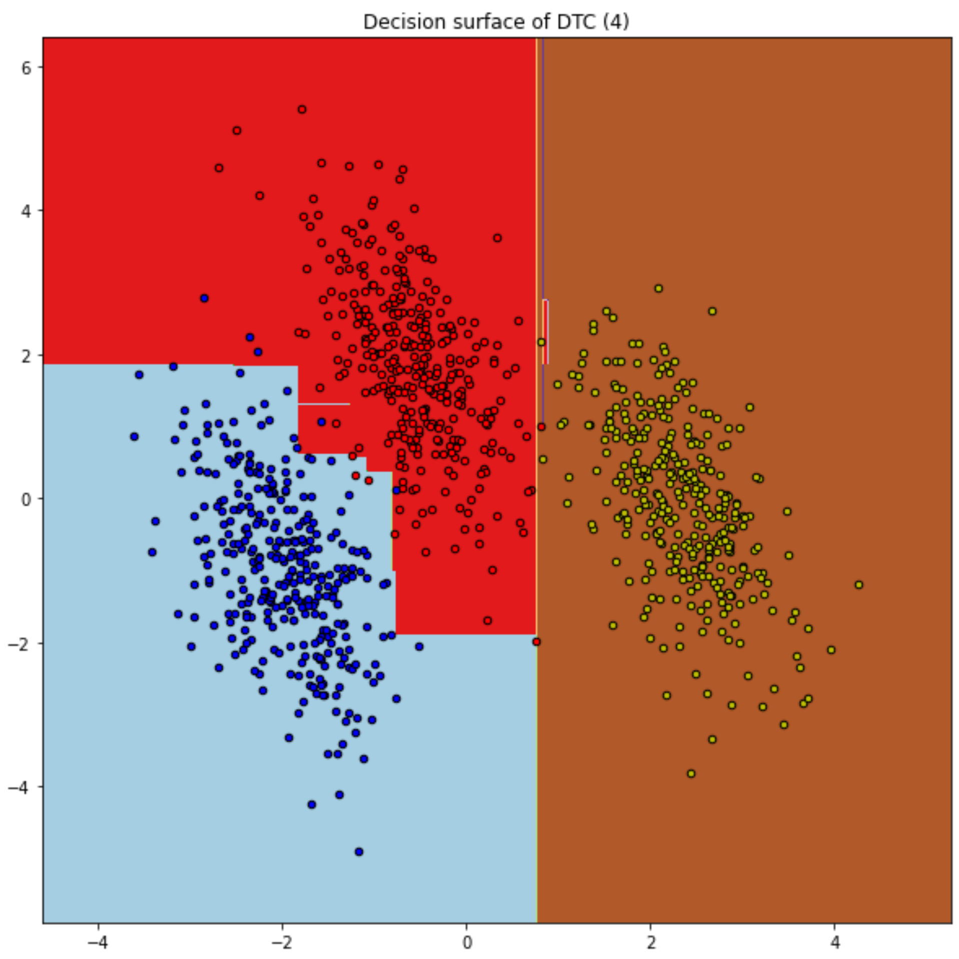
The injected randomness in forests yield decision trees with somewhat decoupled prediction errors. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice 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()
```

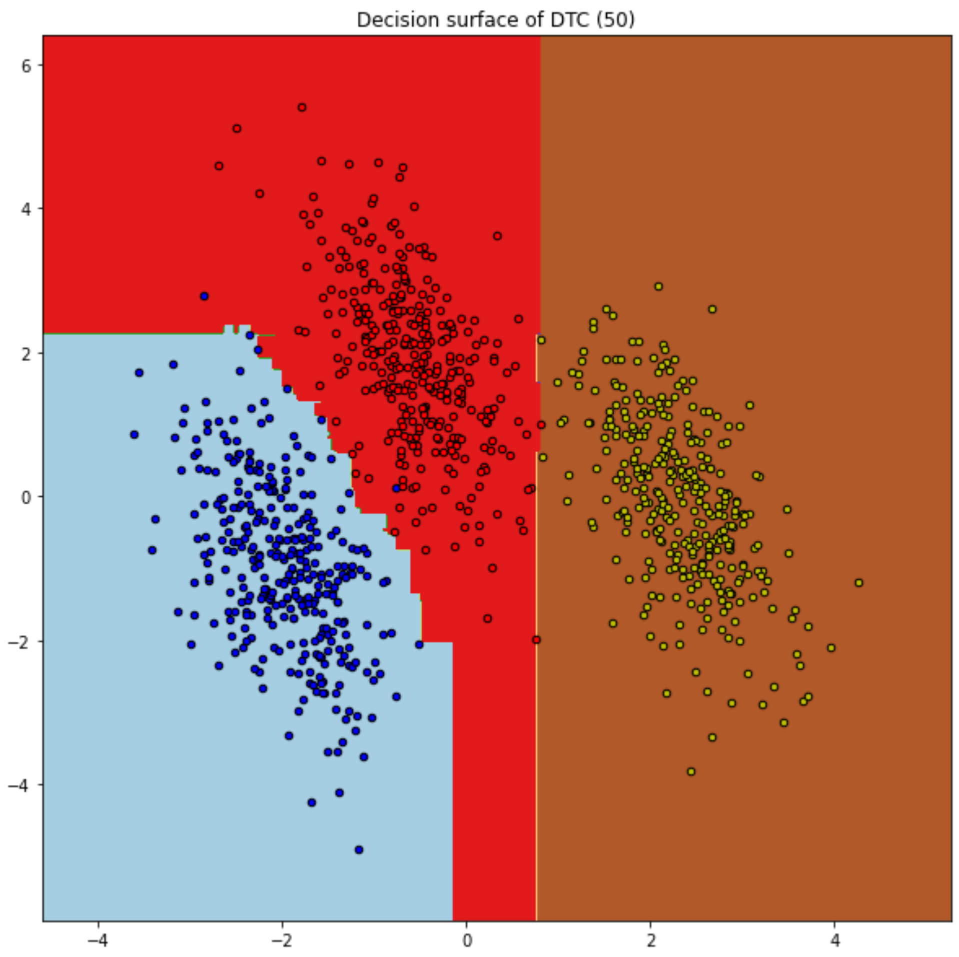
```
training score : 0.987 (n_est=1)
```



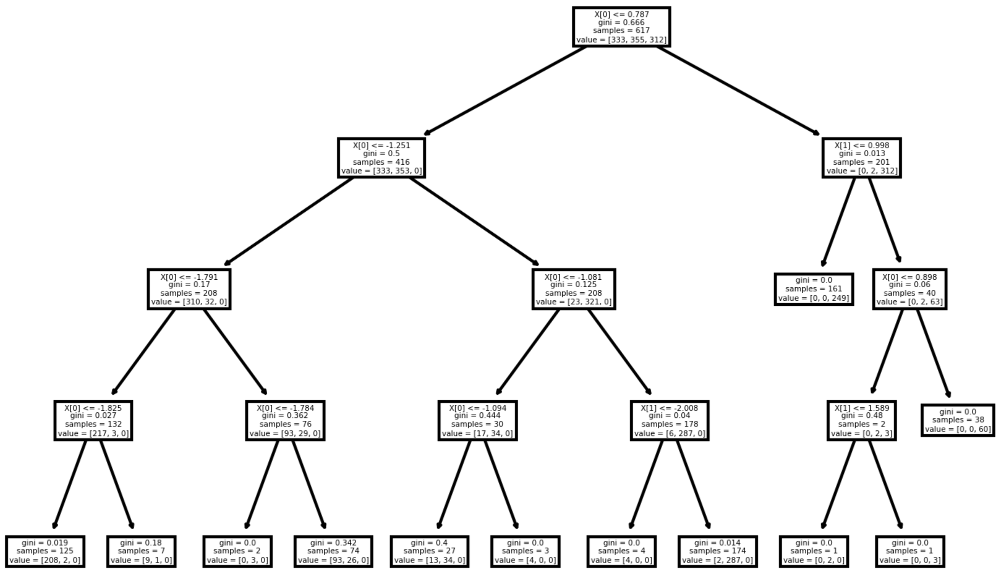
```
training score : 0.992 (n_est=4)
```



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



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



3. Boosted Decision Trees

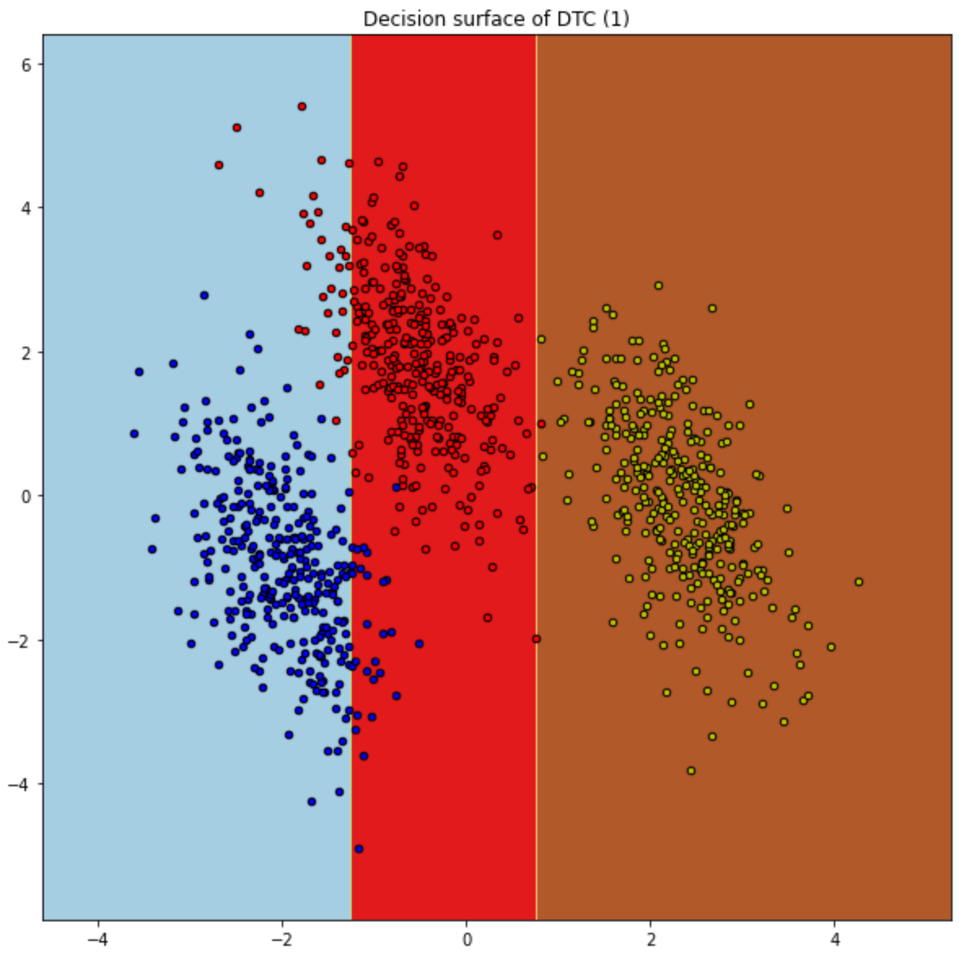
Anouther approach to the ensemble tree modeling is Boosted Decision Trees.

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.

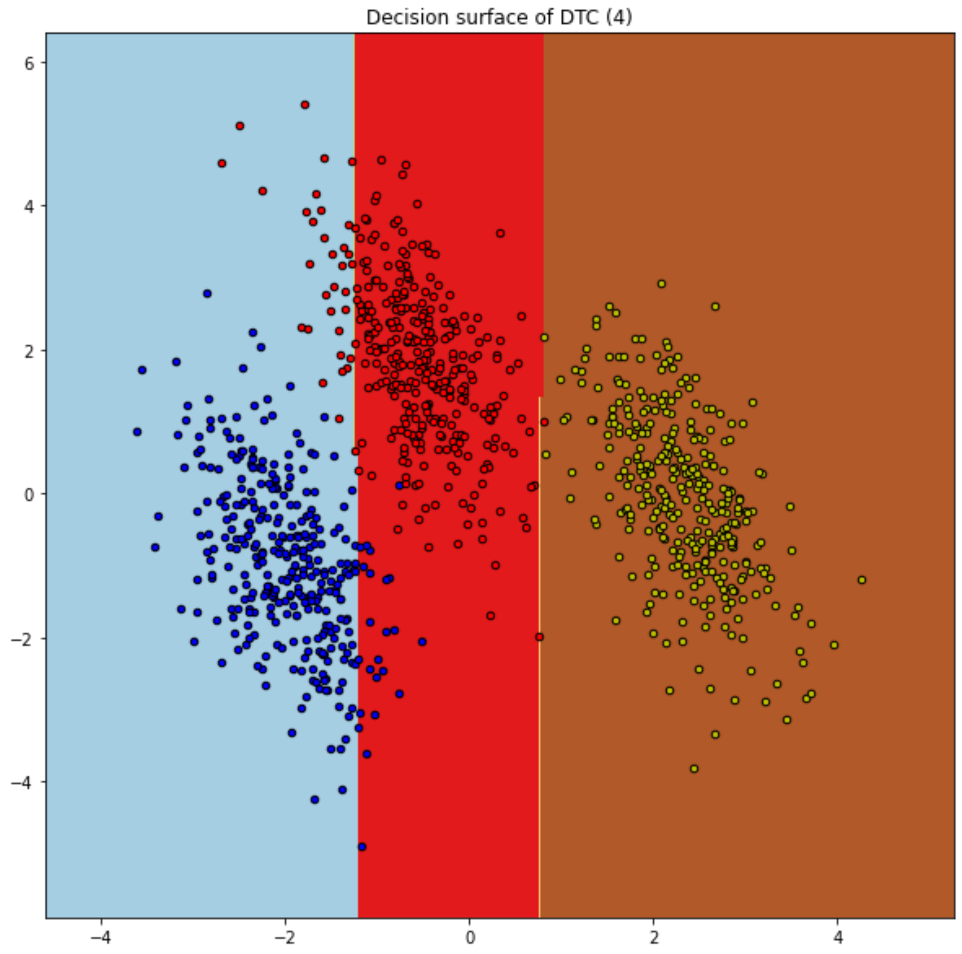
Boosting is based on weak learners, i.e. shallow trees. In boosting primarily the bias is reduced.

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()
```

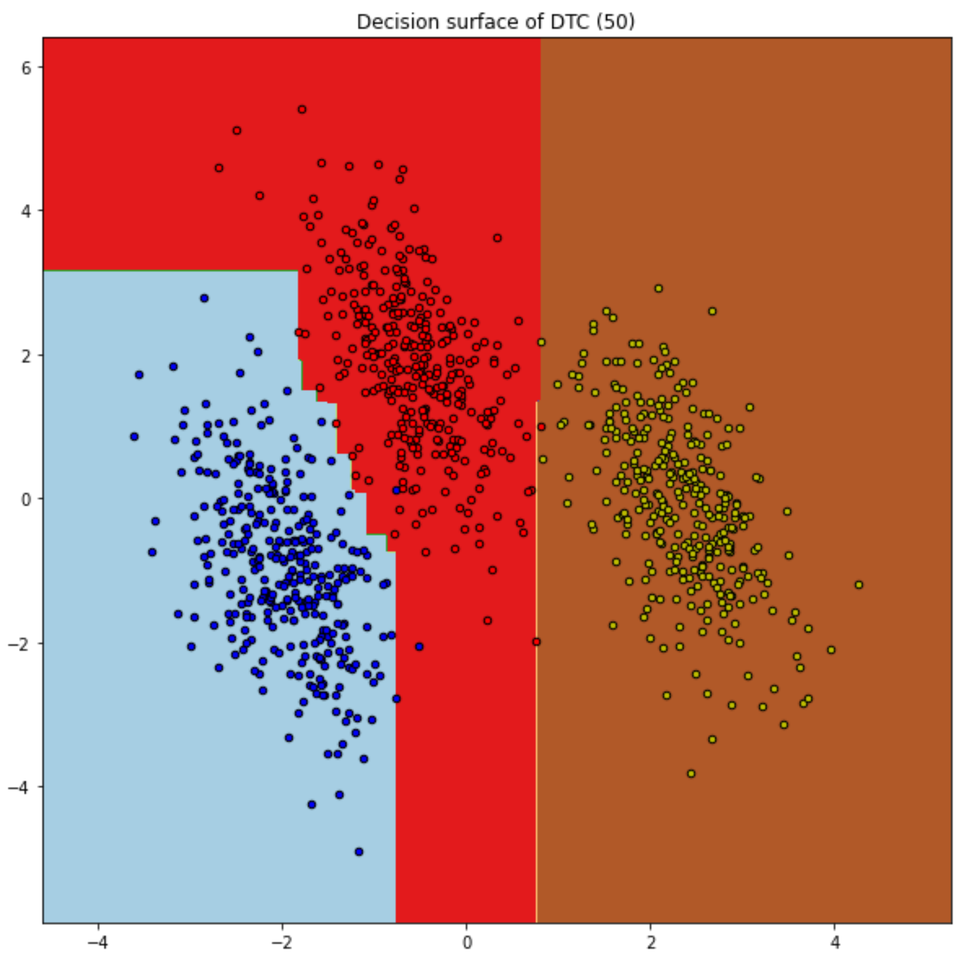
```
training score : 0.942 (n_est=1)
```



```
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 ;

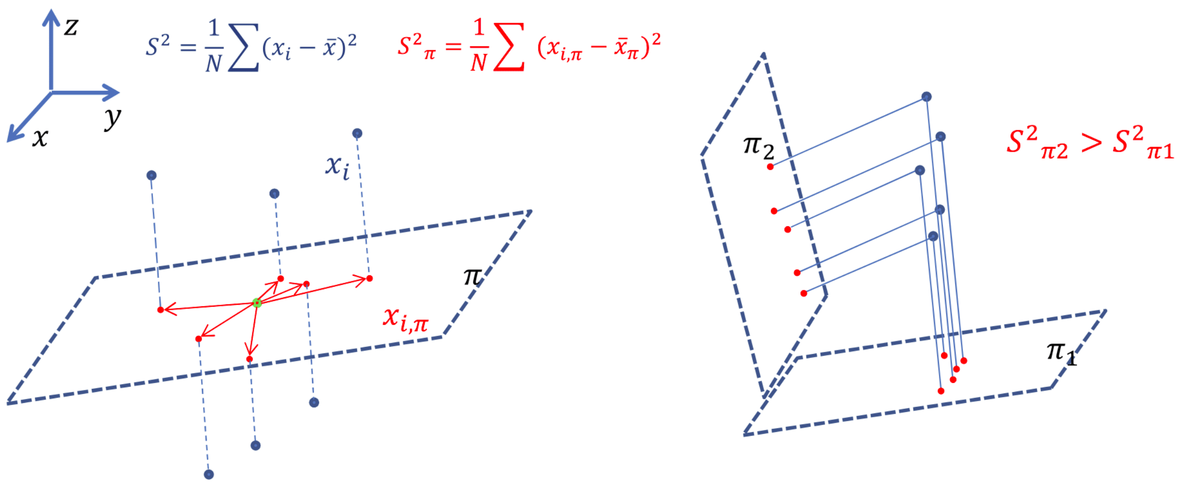
If $M \ll D$ we get an effective dimensionality reduction.

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.


z
y
x
$S^2 = \frac{1}{N}\sum(x_i - \bar{x})^2$
$S^2{}_\pi = \frac{1}{N}\sum(x_{i,\pi} - \bar{x}_\pi)^2$
x_i
π
$x_{i,\pi}$
π_2
$S^2{}_{\pi 2} > S^2{}_{\pi 1}$
π_1

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*

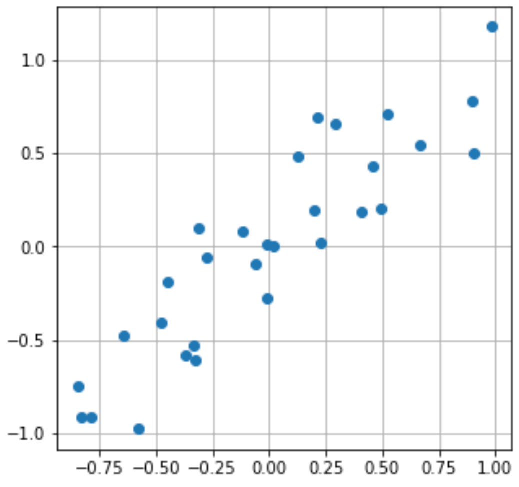
Sklearn: implementation and usage of PCA.

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.

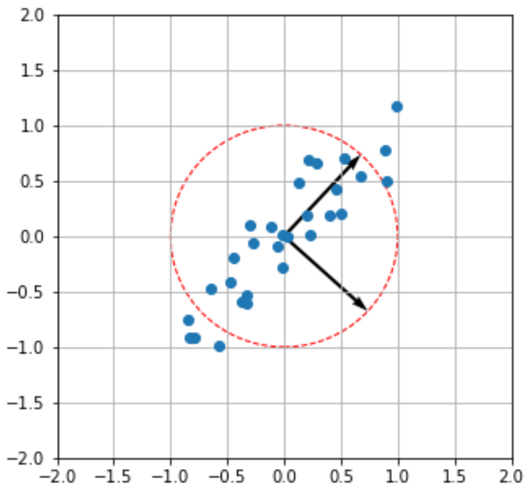
```
In [0]: pca=PCA(n_components=2)
```

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.

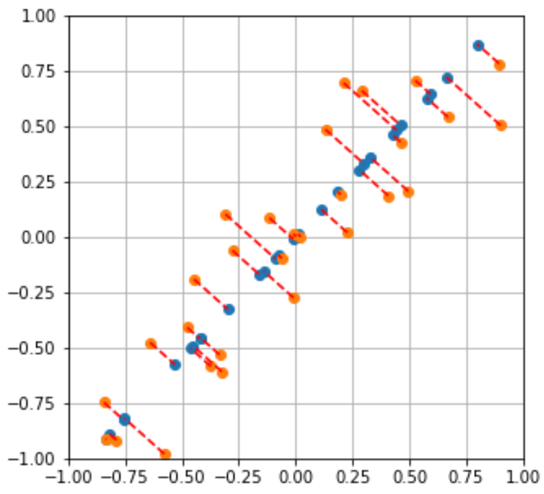
```
In [0]: print(pca.explained_variance_ratio_)

        [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)
```

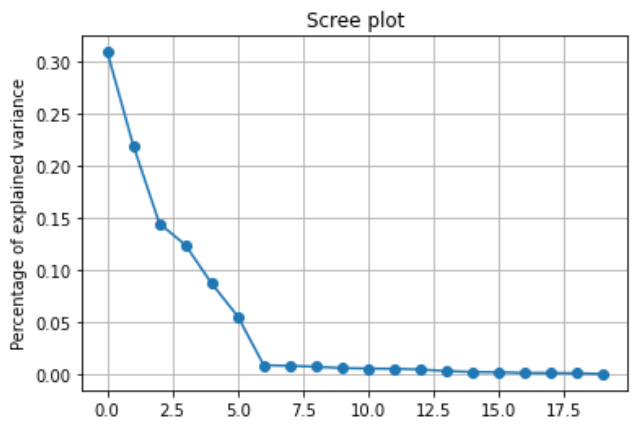


The same procedure is followed for high dimensional data-sets. Here we generate random data which lies almost on a 6 dimensional subspace. The resulting screen plot can be used to find this result in a semi-automaic fashion.

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

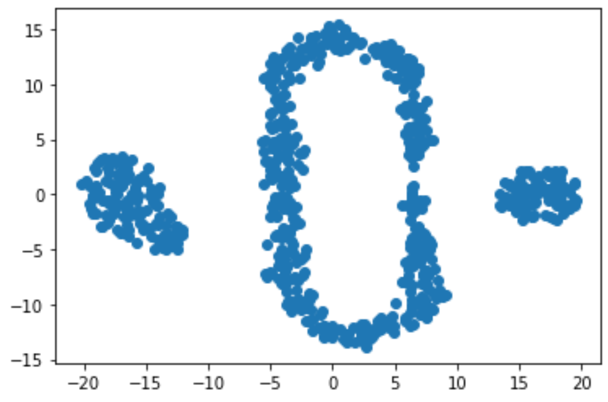
We first of all visualize our multi-dimensional heart using t-sne:

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])
```

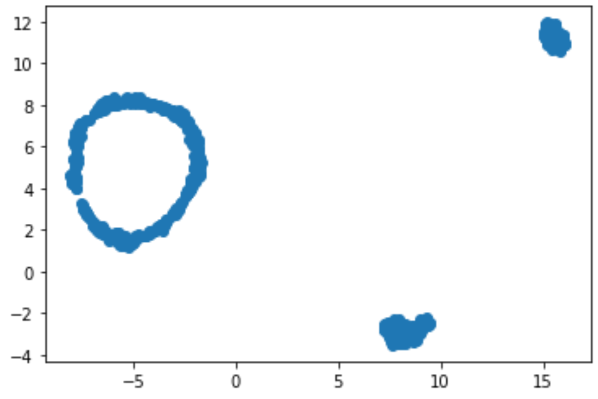
Out[7]:
```
<matplotlib.collections.PathCollection at 0x2a483798d30>
```



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])
```

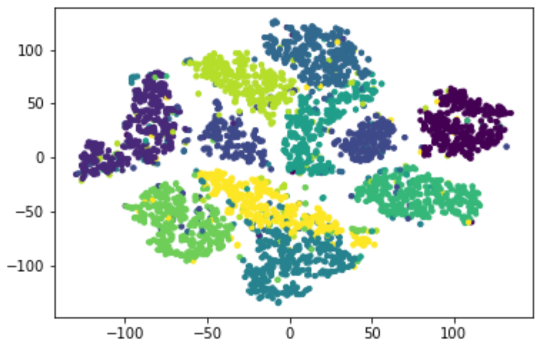
```
Out[9]: <matplotlib.collections.PathCollection at 0x2a485bb0a90>
```



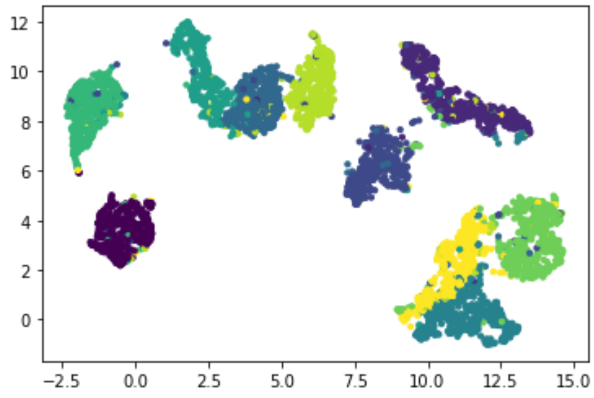
Example 2: Mnist dataset

```
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)
```



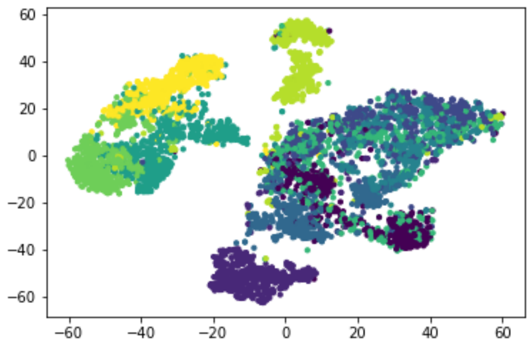
```
In [26]: umap_model = umap.UMAP(n_neighbors=10, n_components=2, random_state=171
         1)
         umap_mnist = umap_model.fit_transform(data)
         plt.scatter(umap_mnist[:, 0], umap_mnist[:, 1], c=labels, s=10)

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



Example 3: Fashion_Mnist dataset

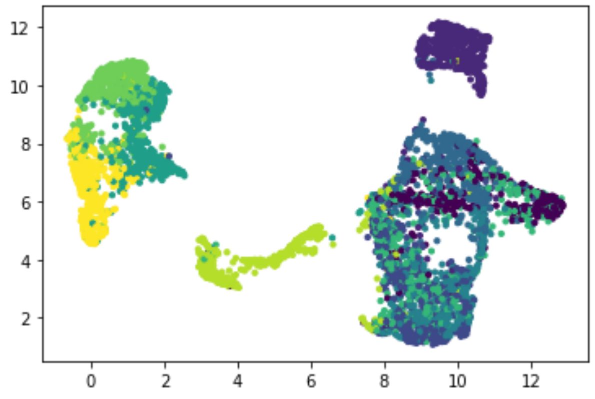
```
In [20]: fmnist = tf.keras.datasets.fashion_mnist
         (train_images, train_labels), (test_images, test_labels) = fmnist.load_d
         ata()

         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)
```

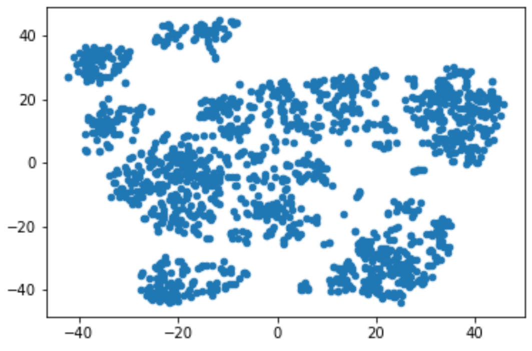
Out[23]:
```
<matplotlib.collections.PathCollection at 0x2a490ad1cf8>
```



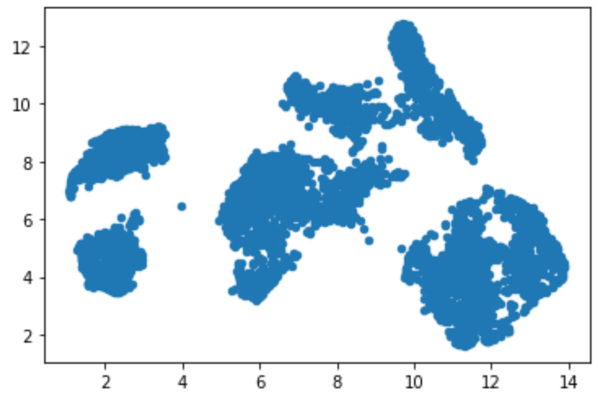
Example 4: House prices

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.