TP APPRENTISSAGE STATISTIQUE

Compte rendu

Kilian Saint-Chely

Professeur : Bilel Bensaid

Master Statistiques et Science des Données

Université de Montpellier

3 octobre 2025

Table des matières

Chapitre 1

Classification des iris

1.1 Importation des librairies et préparation des données

Avant de commencer, nous importons les librairies et le jeu de données nécessaire au TP

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC

from svm_source import *
from sklearn import svm
from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.datasets import fetch_lfw_people
from sklearn.decomposition import PCA
from time import time

scaler = StandardScaler()

import warnings
warnings.filterwarnings("ignore")

plt.style.use('ggplot')

iris = datasets.load_iris()
X = iris.data
X = scaler.fit_transform(X)
y = iris.target
X = X[y != 0, :2]
y = y[y != 0]
```

Listing 1.1 – Préparation

Ensuite nous séparons les données dans l'optique de réaliser une validation croisée. On garde 25% de nos données pour le test et 75% pour l'apprentissage. Enfin, nous fixons une graine pour la reproductibilité du TP.

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.25, random_state=42, shuffle=True)
```

Listing 1.2 – Séparation pour validation croisée

1.2 Classification à partir d'un noyau linéaire

Dans cette section, nous mettons en place un code qui va classifier la classe 1 contre la classe 2 à l'aide d'un noyau linéaire. Nous regardons alors le meilleur paramètre C et les scores sur les données d'entraînement et de tests.

```
# fit the model and select the best hyperparameter C
parameters = {'kernel': ['linear'], 'C': list(np.logspace(-3, 3, 200))
    }
clf_linear = GridSearchCV(SVC(), parameters, n_jobs=-1)
clf_linear.fit(X_train, y_train)

# compute the score
print(clf_linear.best_params_)
print('Generalization score for linear kernel: %s, %s' %
      (clf_linear.score(X_train, y_train),
       clf_linear.score(X_test, y_test)))
```

Listing 1.3 – Classification des classes 1 contre 2 avec noyau linéaire

Les observations que l'on peut faire sont les suivantes :
- le meilleur paramètre est C=0.84
- le score du jeu d'entraînement est 0.75
- le score sur l'ensemble test est 0.68 (<0.75, ce qui est logique puisqu'un modèle colle toujours mieux aux données d'entraînement)

1.3 Classification à partir d'un noyau polynomial

Nous effectuons à présent la même classification mais avec un noyau polynomial où on teste les degrés 1, 2 et 3.

```
Cs = list(np.logspace(-3, 3, 5))
gammas = 10. ** np.arange(1, 2)
degrees = np.r_[1, 2, 3]

# fit the model and select the best set of hyperparameters
parameters = {'kernel': ['poly'], 'C': Cs, 'gamma': gammas, 'degree':
    degrees}

clf_poly = GridSearchCV(SVC(), parameters, n_jobs=-1)
clf_poly.fit(X_train, y_train)

print(clf_poly.best_params_)
print('Generalization score for polynomial kernel: %s, %s' %
      (clf_poly.score(X_train, y_train),
       clf_poly.score(X_test, y_test)))
```

Listing 1.4 – Classification des classes 1 contre 2 avec noyau polynomial

Il ressort de ce test, que le noyau polynomial le plus performant est celui de degré 1 ; avec exactement le même score que le noyau linéaire. Ceci est cohérent puisque le degré d'un noyau linéaire est 1.

1.4 Visualisation de nos classifications

On écrit un code permettant de visualiser les données et les frontières que nous avons tracées.

```python
# display your results using frontiere (svm_source.py)

def f_linear(xx):
    return clf_linear.predict(xx.reshape(1, -1))

def f_poly(xx):
    return clf_poly.predict(xx.reshape(1, -1))

plt.ion()
plt.figure(figsize=(15, 5))
plt.subplot(131)
plot_2d(X, y)
plt.title("iris␣dataset")

plt.subplot(132)
frontiere(f_linear, X, y)
plt.title("linear␣kernel")

plt.subplot(133)
frontiere(f_poly, X, y)
plt.title("polynomial␣kernel")

plt.tight_layout()
plt.draw()
```

Listing 1.5 – Visualisation des données et des classifications

On peut constater sur la figure 1.1 que la frontière est sensiblement différente malgré un score de performance identique. Cela s'explique par le fait que le noyau linéaire est sans terme de biais, alors que le noyau polynomial de degré 1 est une forme affine (avec biais).

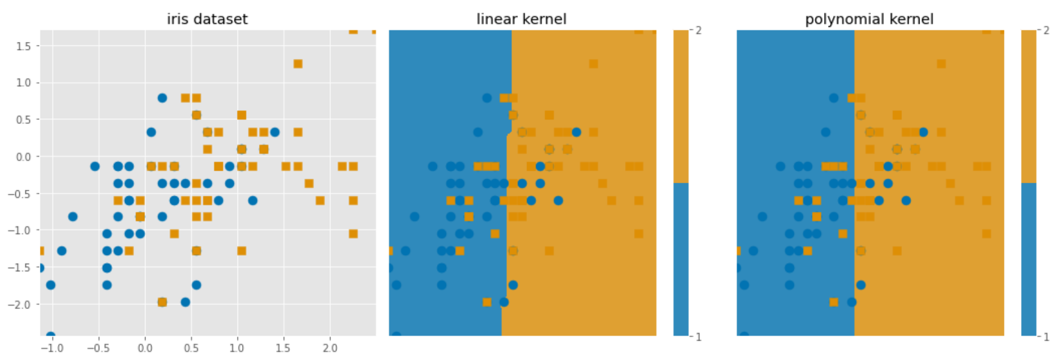

FIGURE 1.1 – Visualisation des données et des frontières

1.5 Reprise d'un noyau polynomial en retirant le degré 1

Dans cette partie, nous reprenons le même procédé que les deux sections précédentes, mais en retirant la possibilité de degré 1 pour le noyau polynomial afin de visualiser ce que cela donnerait comme résultat.

```python
Cs = list(np.logspace(-3, 3, 5))
gammas = 10. ** np.arange(1, 2)
degrees = np.r_[2, 3]

```

```
# fit the model and select the best set of hyperparameters
parameters = {'kernel': ['poly'], 'C': Cs, 'gamma': gammas, 'degree':
    degrees}

clf_poly2 = GridSearchCV(SVC(), parameters, n_jobs=-1)
clf_poly2.fit(X_train, y_train)

print(clf_poly2.best_params_)
print('Generalization score for polynomial kernel: %s, %s' %
      (clf_poly2.score(X_train, y_train),
       clf_poly2.score(X_test, y_test)))

def f_poly2(xx):
    return clf_poly2.predict(xx.reshape(1, -1))

plt.ion()
plt.figure(figsize=(15, 5))
plt.subplot(131)
plot_2d(X, y)
plt.title("iris dataset")

plt.subplot(132)
frontiere(f_linear, X, y)
plt.title("linear kernel")

plt.subplot(133)
frontiere(f_poly2, X, y)
plt.title("polynomial kernel")

plt.tight_layout()
plt.draw()
```

Listing 1.6 – Programme de classification et de visualisation avec un noyau polynomial de degré supérieur à 1

Les résultats montrent que le noyau polynomial le plus approprié est celui de degré 2. Les scores sont :
- jeu d'entraînement : 0.65
- ensemble test : 0.52

On peut voir sur la figure 1.2 que la frontière est bien différente de celle tracée à partir d'un noyau linéaire.

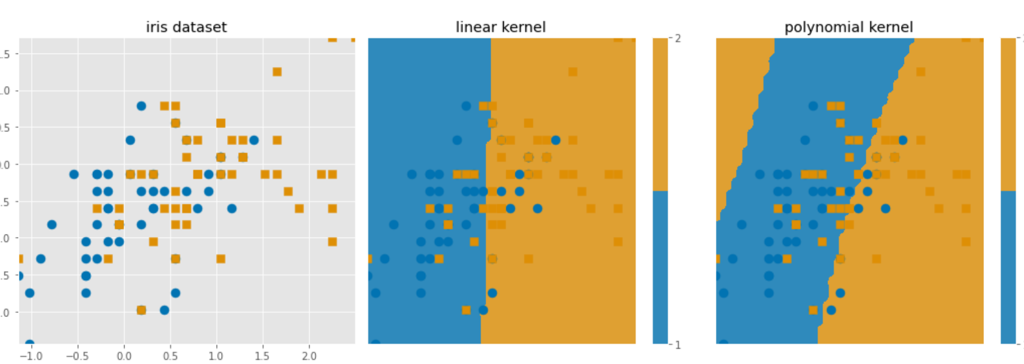

Figure 1.2 – Visualisation des données et des frontières avec noyau polynomial de degré 2

Chapitre 2

Classification des visages

2.1 Influence du paramètre de régularisation

Dans un premier temps, nous importons, préparons et nous familiarisons avec les données.

```
"""
The dataset used in this example is a preprocessed excerpt
of the "Labeled Faces in the Wild", aka LFW_:

  http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz (233MB)

  _LFW: http://vis-www.cs.umass.edu/lfw/
"""

# Download the data and unzip; then load it as numpy arrays
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4,
                              color=True, funneled=False, slice_=None,
                              download_if_missing=True)

# introspect the images arrays to find the shapes (for plotting)
images = lfw_people.images
n_samples, h, w, n_colors = images.shape

# the label to predict is the id of the person
target_names = lfw_people.target_names.tolist()

# Pick a pair to classify
names = ['Tony Blair', 'Colin Powell']

idx0 = (lfw_people.target == target_names.index(names[0]))
idx1 = (lfw_people.target == target_names.index(names[1]))
images = np.r_[images[idx0], images[idx1]]
n_samples = images.shape[0]
y = np.r_[np.zeros(np.sum(idx0)), np.ones(np.sum(idx1))].astype(int)

# plot a sample set of the data
plot_gallery(images, np.arange(12))
plt.show()

# Extract features (grayscale, mean per pixel)
X = (np.mean(images, axis=3)).reshape(n_samples, -1)

# Scale features
X -= np.mean(X, axis=0)
```

```
X /= np.std(X, axis=0)

# Split data into a half training and half test set
X_train, X_test, y_train, y_test, images_train, images_test =
    train_test_split(X, y, images, test_size=0.5, random_state=0)
```

Listing 2.1 – Mise en place des données

Nous regardons à présent l'influence de C en le faisant varier.

```
print("---␣Linear␣kernel␣---")
print("Fitting␣the␣classifier␣to␣the␣training␣set")
t0 = time()

Cs = 10. ** np.arange(-5, 6)   # from 1e-5 to 1e5
scores = []
for C in Cs:
    clf = SVC(kernel='linear', C=C)
    clf.fit(X_train, y_train)
    scores.append(clf.score(X_test, y_test))

ind = np.argmax(scores)
print("Best␣C:␣{}".format(Cs[ind]))

plt.figure()
plt.plot(Cs, scores, marker="o")
plt.xlabel("Param tre␣de␣r gularisation␣C")
plt.ylabel("Score␣de␣test")
plt.xscale("log")
plt.title("Influence␣de␣C")
plt.tight_layout()
plt.show()
print("Best␣score:␣{}".format(np.max(scores)))
```

Listing 2.2 – Programme de visualisation de l'influence de C

Ce code nous dit que la meilleure valeur de C est 0.001 et que le score associé est 0.88. Nous pouvons visualiser ce résultat sur la figure 2.1

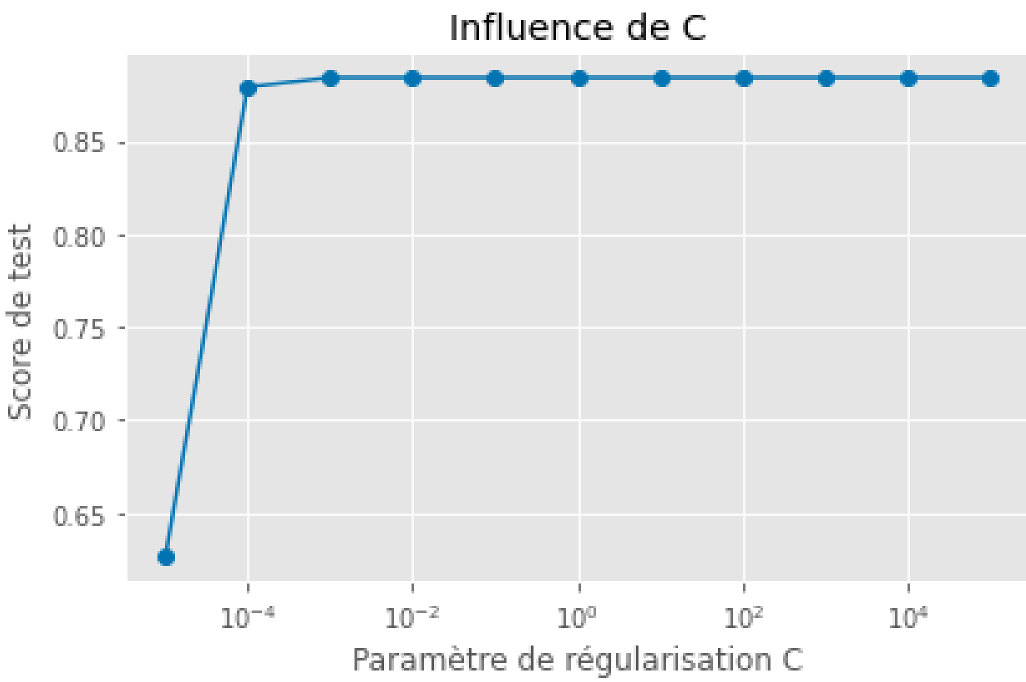

FIGURE 2.1 – Score de test en fonction de la valeur du paramètre de régularisation C

A présent, nous réentraînons le modèle avec la meilleure valeur de C trouvée, nous effectuons une prédiction sur les données de test.Nous prenons soin de mesurer le temps d'exécution et nous comparons la précision réelle du modèle avec le niveau de hasard.

```
print("Predicting the people names on the testing set")
t0 = time()

# retrain best model
clf = SVC(kernel='linear', C=Cs[ind])
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print("done in %0.3fs" % (time() - t0))
# The chance level is the accuracy that will be reached when constantly predicting the majority class.
print("Chance level : %s" % max(np.mean(y), 1. - np.mean(y)))
print("Accuracy : %s" % clf.score(X_test, y_test))
```

Listing 2.3 – Utilisation de la meilleure valeur de C

Les résultats obtenus sont les suivants :
- temps d'exécution : 1.149 secondes
- niveau de hasard : 0.62
- précision réelle du modèle : 0.88

On constate sans surprise que le modèle est "utile" puisqu'il est plus performant que le hasard.

2.2 Évaluation qualitative du modèle de prédiction et visualisation des coefficients

Nous voulons à présent faire un test qualitatif sur la prédiction d'image et visualiser les coefficients appris par le modèle.

```
# Qualitative evaluation of the predictions using matplotlib

prediction_titles = [title(y_pred[i], y_test[i], names)
                     for i in range(y_pred.shape[0])]

plot_gallery(images_test, prediction_titles)
plt.show()

# Look at the coefficients
plt.figure()
plt.imshow(np.reshape(clf.coef_, (h, w)))
plt.show()
```

Listing 2.4 – Test de prédiction et visualisation des coefficients

On obtient dans ce cas là une prédiction 100% juste malgré un score théorique de 88% (figure 2.2). On peut voir sur la figure 2.3 les coefficients, c'est-à-dire les traits discriminants pour le modèle, qui lui permettent de prédire une image.

Figure 2.2 – Prédiction de 12 visages

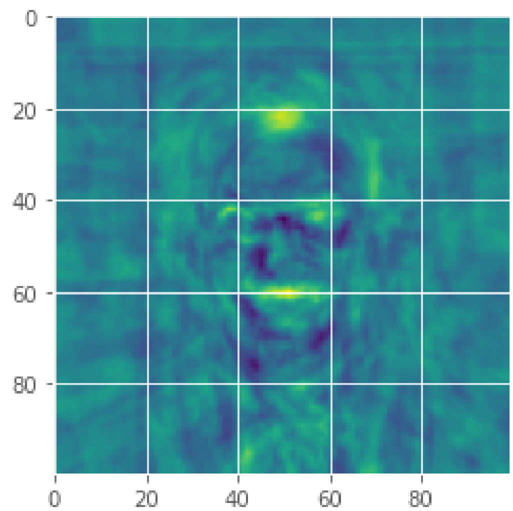

FIGURE 2.3 – Visualisation des coefficients appris par le modèle

2.3 Influence de l'ajout de variables de nuisance

Nous voulons tester l'influence de l'ajout de variables de nuisance. Pour cela, nous calculons le score sans variables de nuisance puis nous ajoutons 300 variables de nuisance Gaussiennes réduites.

```
def run_svm_cv(_X, _y):
    _indices = np.random.permutation(_X.shape[0])
    _train_idx, _test_idx = _indices[:_X.shape[0] // 2], _indices[_X.
        shape[0] // 2:]
    _X_train, _X_test = _X[_train_idx, :], _X[_test_idx, :]
    _y_train, _y_test = _y[_train_idx], _y[_test_idx]

    _parameters = {'kernel': ['linear'], 'C': list(np.logspace(-3, 3,
        5))}
    _svr = svm.SVC()
    _clf_linear = GridSearchCV(_svr, _parameters)
    _clf_linear.fit(_X_train, _y_train)

    print('Generalization score for linear kernel: %s, %s \n' %
          (_clf_linear.score(_X_train, _y_train), _clf_linear.score(
              _X_test, _y_test)))

print("Score sans variable de nuisance")
run_svm_cv(X, y)

print("Score avec variable de nuisance")
n_features = X.shape[1]
# On rajoute des variables de nuisance
sigma = 1
noise = sigma * np.random.randn(n_samples, 300, )
#with gaussian coefficients of std sigma
X_noisy = np.concatenate((X, noise), axis=1)
X_noisy = X_noisy[np.random.permutation(X.shape[0])]
run_svm_cv(X_noisy, y)
```

Listing 2.5 – Programme de calcul de l'influence de l'ajout de variables de nuisance

En généralisation, nous avons un score sans variable de nuisance de 0.94 contre un score avec variables de nuisance de 0.58. On peut donc en conclure que l'ajout de variables de nuisance fait chuter la performance du modèle.

2.4 Amélioration de la prédiction à l'aide d'une réduction de dimension

Nous effectuons une ACP afin de réduire la dimension et de voir si cela améliore comme on peut le supposer la prédiction.

```
print("Score␣apres␣reduction␣de␣dimension")

n_components = 80  # jouer avec ce parametre
pca = PCA(n_components=n_components, svd_solver='randomized').fit(
    X_noisy)
X_pca = pca.transform(X_noisy)
run_svm_cv(X_pca, y)
```

Listing 2.6 – Prédiction avec l'utilisation de l'ACP pour réduire la dimension

Nous obtenons un score de 0.83 sur l'échantillon d'entraînement et de 0.52 sur l'échantillon test. Ce qui paraît anormal car une sélection de variables sachant que 300 d'entre elles sont du bruit (nous le savons puisque c'est nous qui les avons ajoutées) devrait augmenter le score de prédiction.

2.5 Biais dans le prétraitement des données

En prenant du recul sur notre travail, on peut constater qu'il y a un biais dans le prétraitement de nos données. En effet, nous effectuons la moyenne sur les canaux de couleurs, pour ne garder que l'illumination moyenne par pixel, ce qui engendre une perte d'information.

```
X = (np.mean(images, axis=3)).reshape(n_samples, -1)
```

Listing 2.7 – Hypothèse d'une commande source de perte d'information

Chapitre 3

Code complet

```
#%%
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC

from svm_source import *
from sklearn import svm
from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.datasets import fetch_lfw_people
from sklearn.decomposition import PCA
from time import time

scaler = StandardScaler()

import warnings
warnings.filterwarnings("ignore")

plt.style.use('ggplot')

#%%
###############################################################################
#               Toy dataset : 2 gaussians
###############################################################################

n1 = 200
n2 = 200
mu1 = [1., 1.]
mu2 = [-1./2, -1./2]
sigma1 = [0.9, 0.9]
sigma2 = [0.9, 0.9]
X1, y1 = rand_bi_gauss(n1, n2, mu1, mu2, sigma1, sigma2)

plt.show()
plt.close("all")
plt.ion()
plt.figure(1, figsize=(15, 5))
plt.title('First data set')
plot_2d(X1, y1)

```

```
X_train = X1[::2]
Y_train = y1[::2].astype(int)
X_test = X1[1::2]
Y_test = y1[1::2].astype(int)

# fit the model with linear kernel
clf = SVC(kernel='linear')
clf.fit(X_train, Y_train)

# predict labels for the test data base
y_pred = clf.predict(X_test)

# check your score
score = clf.score(X_test, Y_test)
print('Score : %s' % score)

# display the frontiere
def f(xx):
    """Classifier: needed to avoid warning due to shape issues"""
    return clf.predict(xx.reshape(1, -1))

plt.figure()
frontiere(f, X_train, Y_train, w=None, step=50, alpha_choice=1)

# Same procedure but with a grid search
parameters = {'kernel': ['linear'], 'C': list(np.linspace(0.001, 3,
    21))}
clf2 = SVC()
clf_grid = GridSearchCV(clf2, parameters, n_jobs=-1)
clf_grid.fit(X_train, Y_train)

# check your score
print(clf_grid.best_params_)
print('Score : %s' % clf_grid.score(X_test, Y_test))

def f_grid(xx):
    """Classifier: needed to avoid warning due to shape issues"""
    return clf_grid.predict(xx.reshape(1, -1))

# display the frontiere
plt.figure()
frontiere(f_grid, X_train, Y_train, w=None, step=50, alpha_choice=1)

#%%
###################################################################
#                Iris Dataset
###################################################################

iris = datasets.load_iris()
X = iris.data
X = scaler.fit_transform(X)
y = iris.target
X = X[y != 0, :2]
y = y[y != 0]

# split train test (say 25% for the test)
# Split train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.25, random_state=42, shuffle=True)
```

```python

########################################################################
# fit the model with linear vs polynomial kernel
########################################################################

#%%
# Q1 Linear kernel

# fit the model and select the best hyperparameter C
parameters = {'kernel': ['linear'], 'C': list(np.logspace(-3, 3, 200))
    }
clf_linear = GridSearchCV(SVC(), parameters, n_jobs=-1)
clf_linear.fit(X_train, y_train)

# compute the score
print(clf_linear.best_params_)
print('Generalization score for linear kernel: %s, %s' %
      (clf_linear.score(X_train, y_train),
       clf_linear.score(X_test, y_test)))

#%%
# Q2 polynomial kernel
Cs = list(np.logspace(-3, 3, 5))
gammas = 10. ** np.arange(1, 2)
degrees = np.r_[1, 2, 3]

# fit the model and select the best set of hyperparameters
parameters = {'kernel': ['poly'], 'C': Cs, 'gamma': gammas, 'degree':
    degrees}

clf_poly = GridSearchCV(SVC(), parameters, n_jobs=-1)
clf_poly.fit(X_train, y_train)

print(clf_poly.best_params_)
print('Generalization score for polynomial kernel: %s, %s' %
      (clf_poly.score(X_train, y_train),
       clf_poly.score(X_test, y_test)))

#sans le degre 1
Cs = list(np.logspace(-3, 3, 5))
gammas = 10. ** np.arange(1, 2)
degrees = np.r_[2, 3]

# fit the model and select the best set of hyperparameters
parameters = {'kernel': ['poly'], 'C': Cs, 'gamma': gammas, 'degree':
    degrees}

clf_poly2 = GridSearchCV(SVC(), parameters, n_jobs=-1)
clf_poly2.fit(X_train, y_train)

print(clf_poly2.best_params_)
print('Generalization score for polynomial kernel: %s, %s' %
      (clf_poly2.score(X_train, y_train),
       clf_poly2.score(X_test, y_test)))


#%%
# display your results using frontiere (svm_source.py)

```

```
def f_linear(xx):
    return clf_linear.predict(xx.reshape(1, -1))

def f_poly(xx):
    return clf_poly.predict(xx.reshape(1, -1))

plt.ion()
plt.figure(figsize=(15, 5))
plt.subplot(131)
plot_2d(X, y)
plt.title("iris dataset")

plt.subplot(132)
frontiere(f_linear, X, y)
plt.title("linear kernel")

plt.subplot(133)
frontiere(f_poly, X, y)
plt.title("polynomial kernel")

plt.tight_layout()
plt.draw()

def f_poly2(xx):
    return clf_poly2.predict(xx.reshape(1, -1))

plt.ion()
plt.figure(figsize=(15, 5))
plt.subplot(131)
plot_2d(X, y)
plt.title("iris dataset")

plt.subplot(132)
frontiere(f_linear, X, y)
plt.title("linear kernel")

plt.subplot(133)
frontiere(f_poly2, X, y)
plt.title("polynomial kernel")

plt.tight_layout()
plt.draw()

#%%
####################################################################
#                Face Recognition Task
####################################################################
"""
The dataset used in this example is a preprocessed excerpt
of the "Labeled Faces in the Wild", aka LFW_:

  http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz (233MB)

  _LFW: http://vis-www.cs.umass.edu/lfw/
"""

####################################################################
# Download the data and unzip; then load it as numpy arrays
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4,
```

```
                                     color=True, funneled=False, slice_=None,
                                     download_if_missing=True)

# introspect the images arrays to find the shapes (for plotting)
images = lfw_people.images
n_samples, h, w, n_colors = images.shape

# the label to predict is the id of the person
target_names = lfw_people.target_names.tolist()

####################################################################
# Pick a pair to classify
names = ['Tony Blair', 'Colin Powell']

idx0 = (lfw_people.target == target_names.index(names[0]))
idx1 = (lfw_people.target == target_names.index(names[1]))
images = np.r_[images[idx0], images[idx1]]
n_samples = images.shape[0]
y = np.r_[np.zeros(np.sum(idx0)), np.ones(np.sum(idx1))].astype(int)

# plot a sample set of the data
plot_gallery(images, np.arange(12))
plt.show()

####################################################################
# Extract features (grayscale, mean per pixel)
X = (np.mean(images, axis=3)).reshape(n_samples, -1)

# Scale features
X -= np.mean(X, axis=0)
X /= np.std(X, axis=0)

####################################################################
# Split data into a half training and half test set
X_train, X_test, y_train, y_test, images_train, images_test = train_test_split(X, y, images, test_size=0.5, random_state=0)

####################################################################

# Q4 Influence of regularization parameter C
print("--- Linear kernel ---")
print("Fitting the classifier to the training set")
t0 = time()

Cs = 10. ** np.arange(-5, 6)   # from 1e-5 to 1e5
scores = []
for C in Cs:
    clf = SVC(kernel='linear', C=C)
    clf.fit(X_train, y_train)
    scores.append(clf.score(X_test, y_test))

ind = np.argmax(scores)
print("Best C: {}".format(Cs[ind]))

plt.figure()
plt.plot(Cs, scores, marker="o")
plt.xlabel("Param tre de r gularisation C")
plt.ylabel("Score de test")
plt.xscale("log")
```

```python
plt.title("Influence de C")
plt.tight_layout()
plt.show()
print("Best score: {}".format(np.max(scores)))

print("Predicting the people names on the testing set")
t0 = time()

# retrain best model
clf = SVC(kernel='linear', C=Cs[ind])
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print("done in %0.3fs" % (time() - t0))
# The chance level is the accuracy that will be reached when constantly predicting the majority class.
print("Chance level : %s" % max(np.mean(y), 1. - np.mean(y)))
print("Accuracy : %s" % clf.score(X_test, y_test))

####################################################################

#%%
####################################################################
# Qualitative evaluation of the predictions using matplotlib

prediction_titles = [title(y_pred[i], y_test[i], names)
                     for i in range(y_pred.shape[0])]

plot_gallery(images_test, prediction_titles)
plt.show()

####################################################################
# Look at the coefficients
plt.figure()
plt.imshow(np.reshape(clf.coef_, (h, w)))
plt.show()


#%%
# Q5

def run_svm_cv(_X, _y):
    _indices = np.random.permutation(_X.shape[0])
    _train_idx, _test_idx = _indices[:_X.shape[0] // 2], _indices[_X.shape[0] // 2:]
    _X_train, _X_test = _X[_train_idx, :], _X[_test_idx, :]
    _y_train, _y_test = _y[_train_idx], _y[_test_idx]

    _parameters = {'kernel': ['linear'], 'C': list(np.logspace(-3, 3, 5))}
    _svr = svm.SVC()
    _clf_linear = GridSearchCV(_svr, _parameters)
    _clf_linear.fit(_X_train, _y_train)

    print('Generalization score for linear kernel: %s, %s \n' %
          (_clf_linear.score(_X_train, _y_train), _clf_linear.score(_X_test, _y_test)))

print("Score sans variable de nuisance")
```

```python
run_svm_cv(X, y)

print("Score avec variable de nuisance")
n_features = X.shape[1]
# On rajoute des variables de nuisances
sigma = 1
noise = sigma * np.random.randn(n_samples, 300, )
#with gaussian coefficients of std sigma
X_noisy = np.concatenate((X, noise), axis=1)
X_noisy = X_noisy[np.random.permutation(X.shape[0])]
run_svm_cv(X_noisy, y)

#%%
# Q6
print("Score apres reduction de dimension")

n_components = 100  # jouer avec ce parametre
pca = PCA(n_components=n_components, svd_solver='randomized').fit(
    X_noisy)
X_pca = pca.transform(X_noisy)
run_svm_cv(X_pca, y)
```

Listing 3.1 – Code python complet