04 - Model Selection

In the last exercise section, we have seen how to manage the bias-variance trade-off and how to properly evaluate our model for a given task. In this exercise section, we will see some of the methods to select a model in the first place. These includes:

- feature selection, which is a supervised methodology to identify the relevant features for a given task;
- principal component analysis for dimensionality reduction, which is an unsupervised methodology to reduce the dataset dimensionality;
- regularization with the k paramater in the k-nearest neighbours classifier.

Feature Selection

Let us suppose to face a common classification task on the Iris Dataset, e.g., we would like to train a model to classify 'Iris-virginica' against 'Non-virginica'.

First, we can load the dataset, which is provided by sklearn

In [1]:
```
from sklearn import datasets

iris = datasets.load_iris()

# the variable iris now contains inputs ('data' field), targets ('target', field)
# and other stuff. Let us keep the inputs and targets in two separate variables

X = iris.data
y = iris.target
```

we need targets! SUPERVISED FEATURES SELECTION

The standard targets are suited for a 3-class classification, where the class 0 is 'Setosa', 1 is 'Versicolor', 2 is 'Virginica'. Let us rearrange the target y to have just class 1 for 'Virginica', 0 for 'Non-virginica'

In [2]:
```
print('Original targets\n', y)

# the dataset is stored in the variable iris
y = iris.target.copy()
# 0 - setosa, 1 - versicolor, 2 - virginica
y[y == 1] = 0
y[y == 2] = 1

print('Iris-Virginica targets\n', y)
```

```
Original targets
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
Iris-Virginica targets
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1]
```

As it is well-known, this dataset has four input features. But how many of them are actually relevant for our classification task?

Backward Feature Selection

Let us try to identify a convenient subset of k features for our classification task. In principle, we would have to train $\binom{4}{k}$ models, one for each $k \in \{1, 2, 3, 4\}$, and to compare one model against the others. This sounds quite inefficient. Instead, we start from the complete set of features and we try to iteratively remove the least relevant feature.

But first, do not forget to split the data in train/test/validation sets (here we will use validation to make a decision on the subset of features).

In [3]:
```
import numpy as np
# set the seed to get consistent splits
np.random.seed(0)

# let us concatenate X and y in a single matrix
D = np.concatenate((X, y.reshape(len(y), -1)), axis=1)

# then we shuffle D
np.random.shuffle(D)

# and we slip the data as 50/20/20
X_train = D[:90,:4]
y_train = D[:90,4]
X_vali = D[90:120,:4]
y_vali = D[90:120,4]
X_test = D[120:150,:4]
y_test = D[120:150,4]
```

Let us train the full model (e.g., logistic regression) at first.

In [4]:
```
from sklearn.linear_model import LogisticRegression
```

0. all features

```
# Logistic regression on the original dataset
log_classifier = LogisticRegression(penalty='none') # regularization is applied as default
log_classifier.fit(X_train, y_train)
y_hat_vali = log_classifier.predict(X_vali)
vali_accuracy = sum(y_hat_vali == y_vali) / len(y_vali)
print('Validation accuracy:', vali_accuracy)
```

```
Validation accuracy: 1.0
```

The task is quite simple, and the full model is perfect on the validation set. But all the features are necessary here?

Let us train a model M_{-i} which consider the set of features $\{x_0, x_1, x_2, x_3\} \setminus \{x_i\}$ for each $i \in \{0, 1, 2, 3\}$

1. -1 feature

In [5]:
```
# full set of features
F = [0, 1, 2, 3]

for i in range(len(F)):
  # remove the feature i from train and validation
  X_train_i = np.delete(X_train, i, axis=1)
  X_vali_i = np.delete(X_vali, i, axis=1)
  # fit the classifier
  log_classifier_i = LogisticRegression(penalty='none') # regularization is applied as default
  log_classifier_i.fit(X_train_i, y_train)
  # evaluate on validation
  y_hat_vali = log_classifier_i.predict(X_vali_i)
  vali_accuracy = sum(y_hat_vali == y_vali) / len(y_vali)
  print('The model with features', F, 'without', F[i], 'has validation accuracy:', vali_accuracy)
```

```
The model with features [0, 1, 2, 3] without 0 has validation accuracy: 1.0
The model with features [0, 1, 2, 3] without 1 has validation accuracy: 1.0
The model with features [0, 1, 2, 3] without 2 has validation accuracy: 1.0
The model with features [0, 1, 2, 3] without 3 has validation accuracy: 1.0
```

As you can see, removing one feature does not seem to impact the performance. Let us remove a feature randomly (e.g., x_0), so that we can repeat the process with three features

2. -2 features

In [6]:
```
F = [1, 2, 3]
X_train_new = X_train[:, F]
X_vali_new = X_vali[:, F]
for i in range(len(F)):
  # remove the feature from train and validation
  X_train_i = np.delete(X_train_new, i, axis=1)
  X_vali_i = np.delete(X_vali_new, i, axis=1)
  # fit the classifier
  log_classifier_i = LogisticRegression(penalty='none') # regularization is applied as default
  log_classifier_i.fit(X_train_i, y_train)
  # evaluate on validation
  y_hat_vali = log_classifier_i.predict(X_vali_i)
  vali_accuracy = sum(y_hat_vali == y_vali) / len(y_vali)
  print('The model with features', F, 'without', F[i], 'has validation accuracy:', vali_accuracy)
```

```
The model with features [1, 2, 3] without 1 has validation accuracy: 1.0
The model with features [1, 2, 3] without 2 has validation accuracy: 0.9666666666666667
The model with features [1, 2, 3] without 3 has validation accuracy: 0.9666666666666667
```

Removing feature x_1 we are still perfect on the validation accuracy, thus, we can remove x_1 without any drop in the (estimated) performance

3. -3 features

In [7]:
```
F = [2, 3]
X_train_new = X_train[:,F]
X_vali_new = X_vali[:, F]
for i in range(len(F)):
  # remove the feature from train and validation
  X_train_i = np.delete(X_train_new, i, axis=1)
  X_vali_i = np.delete(X_vali_new, i, axis=1)
  # fit the classifier
  log_classifier_i = LogisticRegression(penalty='none') # regularization is applied as default
  log_classifier_i.fit(X_train_i, y_train)
  # evaluate on validation
  y_hat_vali = log_classifier_i.predict(X_vali_i)
  vali_accuracy = sum(y_hat_vali == y_vali) / len(y_vali)
  print('The model with features', F, 'without', F[i], 'has validation accuracy:', vali_accuracy)
```

```
The model with features [2, 3] without 2 has validation accuracy: 0.9666666666666667
The model with features [2, 3] without 3 has validation accuracy: 0.9666666666666667
```

Removing another feature from $\{x_2, x_3\}$ seems to reduce the performance in any case. Thus, we can stop here the backward feature selection procedure and keep only $\{x_2, x_3\}$ in the final dataset. Going from a four features to two might seem a minor feat, but this is a very small example, and we are still halving the dimensionality of the dataset. The final model is also simpler, i.e., less prone to overfitting.

Until now we have evaluated the feature selection with the validation set, let us look how the model behave on the test set

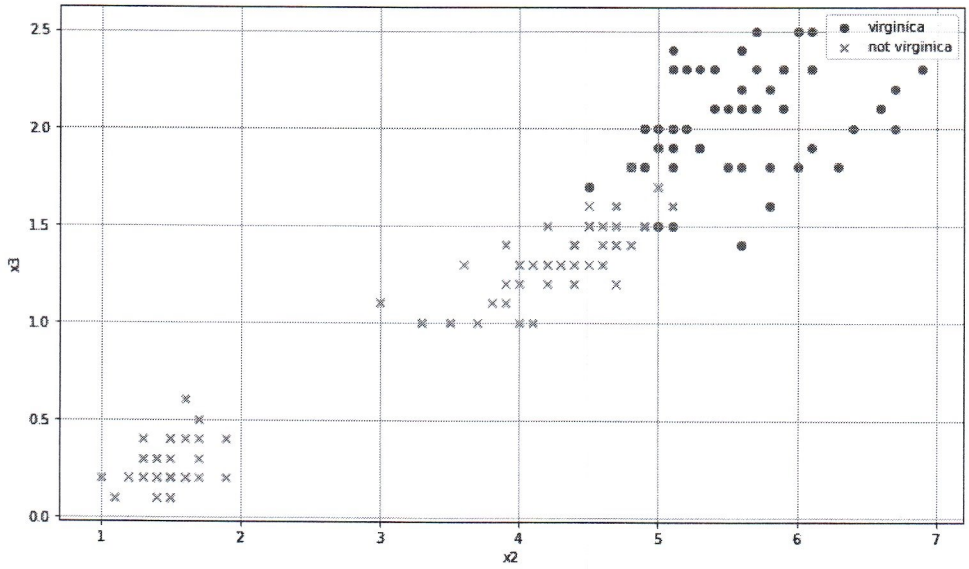
In [8]:
```
# we can merge back the validation into the train set
X_train = np.concatenate((X_train, X_vali))
y_train = np.concatenate((y_train, y_vali))

# and extract the dataset after feature selection
F = [2, 3]
X_train_fs = X_train[:, F]
X_test_fs = X_test[:, F]

# we fit the model once again on the new training
log_classifier_fs = LogisticRegression(penalty='none')
log_classifier_fs.fit(X_train_fs, y_train)
# and we evaluate it on the test set
```

```
y_hat_test = log_classifier_fs.predict(X_test_fs)
test_accuracy = sum(y_hat_test == y_test) / len(y_test)
print('The model with feature', F, 'has test accuracy:', test_accuracy)
```

```
The model with feature [2, 3] has test accuracy: 0.9333333333333333
```

Now that the dataset has only two features it is also easier to visualize

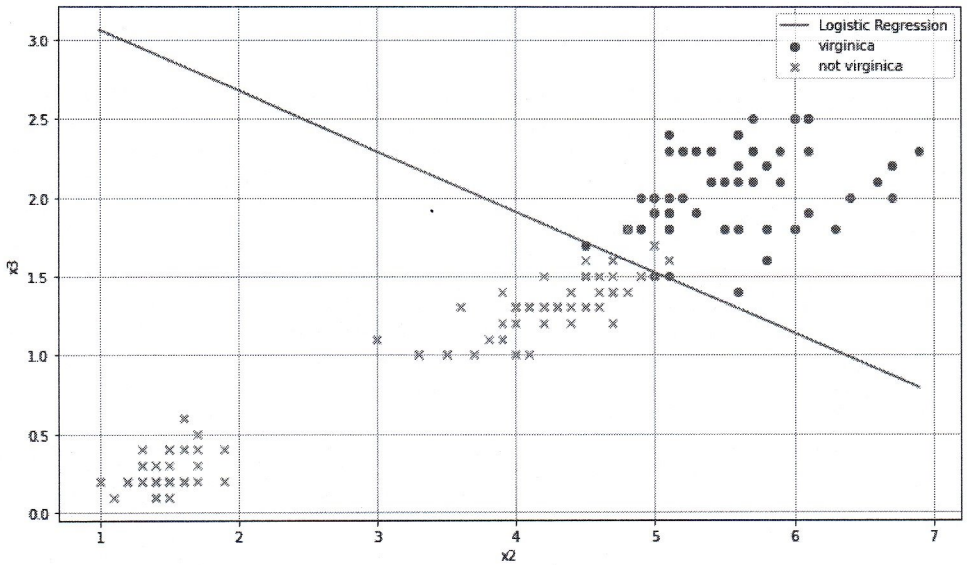
In [9]:
```
from matplotlib import pyplot as plt

virginica = X[y == 1]
not_virginica = X[y == 0]

plt.figure(figsize=(12,7))
plt.scatter(virginica[:, 2], virginica[:, 3], label='virginica')
plt.scatter(not_virginica[:, 2], not_virginica[:, 3], label='not virginica', marker='x')

plt.xlabel('x2')
plt.ylabel('x3')
plt.grid()
plt.legend()
plt.show()
```



We can also plot the discriminative surface of the perceptron

In [10]:
```
# Same function of Exercise Session 2
def plot_ds(X, w, step=100, label='DS'):
  ds_x1 = np.linspace(X[:,2].min(), X[:,2].max(), step)
  ds_x2 = [-(w[0] + w[1]*x1) / w[2] for x1 in ds_x1]
  plt.plot(ds_x1, ds_x2, label=label)

plt.figure(figsize=(12,7))
coef = log_classifier_fs.coef_.flatten() # weights
w0 = log_classifier_fs.intercept_ # bias
log_w = np.array([w0, coef[0], coef[1]])
plot_ds(X, log_w, label='Logistic Regression')

plt.scatter(virginica[:, 2], virginica[:, 3], label='virginica')
plt.scatter(not_virginica[:, 2], not_virginica[:, 3], label='not virginica', marker='x')

plt.xlabel('x2')
plt.ylabel('x3')
plt.grid()
plt.legend()
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:10: VisibleDeprecationWarning: Creating an n
darray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with differe
nt lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creati
ng the ndarray
  # Remove the CWD from sys.path while we load stuff.
```



Dimensionality Reduction: Principal Component Analysis

Principal Component Analysis (PCA) is an unsupervised technique to perform dimensionality reduction, i.e., to extract some low dimensional features from a dataset. Especially, we aim to find a linear transformation of the original data X s.t. the largest variance lies on the first transformed feature, the second largest variance on the second transformed feature and so on. At last, we only keep some of the features we extracted.

To see how to perform PCA in practice, let us start once again from the Iris Dataset

In [11]:
```
from sklearn import datasets

iris = datasets.load_iris()

# the variable iris now contains inputs ('data' field), targets ('target', field)
# and other stuff. Let us keep the inputs and targets in two separate variables

X = iris.data
y = iris.target

# Note: PCA is an unsupervised technique, so we do not need y actually
```

To identify the principal components (i.e., the extracted features), we have to:

1. Normalize the original inputs X to obtain a matrix ($\tilde{X}$) with zero (column-wise) mean;
2. Compute the covariance matrix of $\tilde{X}$, i.e., $C = \tilde{X}^\top \tilde{X}$;
3. Compute the eigenvectors e and eigenvalues λ of C:
   - The eigenvector e_1 corresponding to the largest eigenvalue λ_1 will be the first principal component direction;
   - The eigenvector e_2 corresponding to the largest eigenvalue λ_2 will be the second principal component direction;
   - ...

Let us center the data in X (step 1)

In [12]:
```
import numpy as np

X_tilde = X - np.mean(X, axis=0)

# Note: np.mean(X, axis=0) returns a vector containing the mean of each column
np.mean(X, axis=0)
```

Out[12]:
```
array([5.84333333, 3.05733333, 3.758     , 1.19933333])
```

We now compute the covariance matrix C (step 2)

In [13]:
```
C = np.dot(X_tilde.T, X_tilde)

print(C)
```

```
[[102.16833333  -6.32266667 189.873      76.92433333]
 [ -6.32266667  28.30693333 -49.1188    -18.12426667]
 [189.873       -49.1188    464.3254    193.0458    ]
 [ 76.92433333 -18.12426667 193.0458     86.56993333]]
```

We extract the eigenvectors and eigenvalues of C with the eig function of numpy
(https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html)

In [14]:
```
eigenvalues, eigenvectors = np.linalg.eig(C)

print('eigenvalues:', eigenvalues)
print('eigenvectors:', eigenvectors.T)

# Note: each column of eigenvectors is the unit-length eigenvector corresponding to the i-th eigenvalue
```

```
eigenvalues: [630.0080142   36.15794144  11.65321551   3.55142885]
eigenvectors: [[ 0.36138659 -0.08452251  0.85667061  0.3582892 ]
 [-0.65658877 -0.73016143  0.17337266  0.07548102]
 [-0.58202985  0.59791083  0.07623608  0.54583143]
 [ 0.31548719 -0.3197231  -0.47983899  0.75365743]]
```

The columns of the eigenvectors matrix identify the directions of the principal components. Thus, we have obtained the linear transformation matrix (W) we were looking for, and we can now transform the data as $T = \tilde{X}W$

In [15]:
```
W = eigenvectors

T = np.dot(X_tilde, W)

print('Original data point:', X[21])
print('Transformed data point:', T[21])
```

```
Original data point: [5.1 3.7 1.5 0.4]
Transformed data point: [-2.54370523 -0.43299606  0.20845723  0.0410654 ]
```

PCA with Scikit Learn

Notably, there exist a much easier strategy to obtain the same linear transformation: use the scikit learn implementation (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html)

In [16]:
```
from sklearn.decomposition import PCA

# we create a PCA object
pca = PCA()

# we fit the dataset in X
pca.fit(X)

# we extract the principal components directions
print('first principal component direction:', pca.components_[:,0])

# and the corresponding explained variance
explained = pca.explained_variance_
print('explained variance:', explained)
```

```
first principal component direction: [ 0.36138659  0.65658877 -0.58202985 -0.31548719]
explained variance: [4.22824171 0.24267075 0.0782095  0.02383509]
```

Finally, we can transoform the data as $T = \tilde{X}W$

In [17]:
```
T = pca.transform(X)

print('The same data point as before:', T[21])
```

```
The same data point as before: [-2.54370523  0.43299606  0.20845723 -0.0410654 ]
```

From Linear Transformation to Dimensionality Reduction

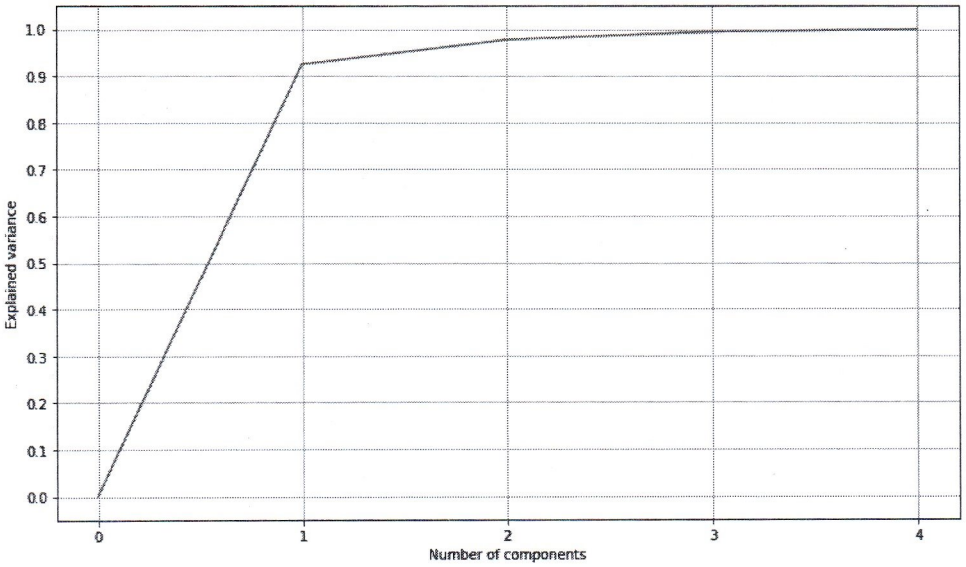
Until now, we have simply transoformed the dataset, but we didn't reduce its dimensions. To perform dimensionality reduction, we can keep just the first k columns of the linear transformation matrix W (i.e., the first k new axes) instead of all of them.

But how can we choose k properly? There exist several different strategies, such as:

- Keep all the principal components until we cumulate at least 90% (or 95%) of the variance;
- Keep all the principal components which explain more of the 5% of the variance;
- Find the elbow in the cumulated variance.

E.g., let us look for the elbow of the explained variance (as a function of the number of components)

In [18]:
```
from matplotlib import pyplot as plt

explained_variance = np.cumsum(explained) / sum(explained)
explained_variance = np.insert(explained_variance, 0, 0.)

plt.figure(figsize=(12,7))
plt.plot(range(5), explained_variance)
plt.xlabel('Number of components')
plt.ylabel('Explained variance')
plt.xticks(range(5))
plt.yticks(np.arange(0., 1.1, 0.1))
plt.grid()
plt.show()
```



We can see that the first two principal components dimensions accounts for the largest share of the explained variance (around 98%). Thus we may choose $k = 2$ to obtain the new dataset $T_2 = \tilde{X}W_{1,2}$, where $W_{1,2}$ is obtained from W keeping the first two columns.

In [19]:
```
# without Scikit Learn
T_12 = np.dot(X_tilde, W[:,:2])

print('Data point with reduced dimensionality', T_12[21])
```

```
Data point with reduced dimensionality [-2.54370523 -0.43299606]
```

In [20]:
```
# with Scikit Learn
pca2 = PCA(n_components=2)
T_12 = pca2.fit_transform(X)

print('Data point with reduced dimensionality', T_12[21])
```

```
Data point with reduced dimensionality [-2.54370523  0.43299606]
```

PCA for Classification

There are multiple purposes to perform the PCA and project the dataset into a lower dimensional space. At first, we could consider the principal components analysis as a feature extraction technique for classification and regression tasks.

Let us get back to the 'Iris-Setosa' classification task. First, we recover the corresponding targets from the dataset.

In [21]:
```
# the dataset is stored in the variable iris
y = iris.target.copy()
# 0 - setosa, 1 - versicolor, 2 - virginica
y[y == 1] = 2
y[y == 0] = 1
y[y == 2] = 0

print('Iris-Setosa targets\n', y)
```

```
Iris-Setosa targets
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0]
```

Let us consider a logistic regression trained over different input dataset, i.e. , the original one (X), the first two principal components ($T_{1,2}$) and the last two ($T_{3,4}$).

In [22]:
```
from sklearn.linear_model import LogisticRegression

# Logistic regression on the original dataset
log_classifier_original = LogisticRegression(penalty='none') # regularization is applied as default
log_classifier_original.fit(X, y)
y_hat_original = log_classifier_original.predict(X)
acc_original = sum(y_hat_original == y) / len(y)
print('Accuracy original:', acc_original)

# Logistic regression on the first two pc
T_12 = np.dot(X_tilde, W[:,:2])
log_classifier_12 = LogisticRegression(penalty='none')
log_classifier_12.fit(T_12, y)
y_hat_12 = log_classifier_12.predict(T_12)
acc_12 = sum(y_hat_12 == y) / len(y)
print('Accuracy first two pc:', acc_12)

# Logiastic regression on the last two pc
T_34 = np.dot(X_tilde, W[:,2:])
log_classifier_34 = LogisticRegression(penalty='none')
log_classifier_34.fit(T_34, y)
y_hat_34 = log_classifier_34.predict(T_34)
```

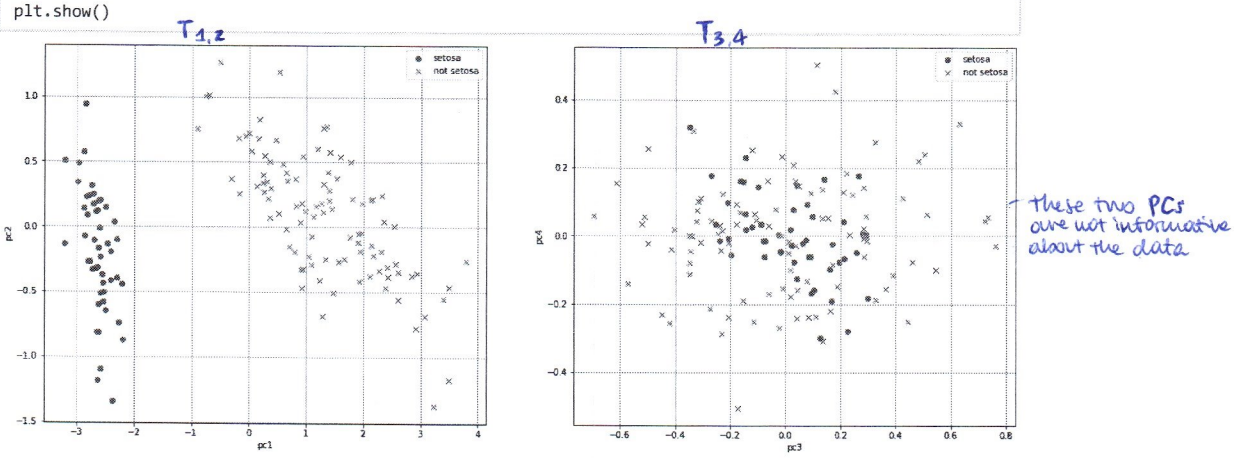
```
acc_34 = sum(y_hat_34 == y) / len(y)
print('Accuracy last two pc:', acc_34)
```

```
Accuracy original: 1.0
Accuracy first two pc: 1.0
Accuracy last two pc: 0.6666666666666666
```

As you can see, the use of the first two principal components gives similar results, since they accounts for around 98% of the variance of the original data. Instead, it makes little sense to use the last two principal components (which explain a small share of the variance) and the corresponding accuracy drops significantly.

Indeed, let us visualize the data in the space defined by the first two principal components as opposed to the last two

In [23]:
```
from matplotlib import pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,8))

setosa = T_12[y == 1]
not_setosa = T_12[y == 0]

ax1.scatter(setosa[:, 0], setosa[:, 1], label='setosa')
ax1.scatter(not_setosa[:, 0], not_setosa[:, 1], label='not setosa', marker='x')
ax1.set(xlabel='pc1', ylabel='pc2')
ax1.grid()
ax1.legend()

setosa = T_34[y == 1]
not_setosa = T_34[y == 0]

ax2.scatter(setosa[:, 0], setosa[:, 1], label='setosa')
ax2.scatter(not_setosa[:, 0], not_setosa[:, 1], label='not setosa', marker='x')
ax2.set(xlabel='pc3', ylabel='pc4')
ax2.grid()
ax2.legend()

plt.show()
```



As a final remark we should underline that the PCA does not make use of the label when is performed on a dataset, while other model selection techniques are based on the a posteriori performance of the classifier (e.g., feature selection).

Regularization

We have already seen regularization techniques in relation with linear regression (e.g., Lasso, Ridge regression). This is actually a model selection technique, as it allows to tune the model complexity through a hyper-paramter (commonly denoted as λ in linear regression)

Let us now consider the regularization technique in a different context, i.e., a classification task with the k-Nearest Neighbors (k-NN) classifier. k-NN is a non-parametric method that coarsely works as follows: Suppose you have to predict the class of a new data point, look at the k data points in the train set that are the closest to this new one, and decides the class of it accordingly (usually via majority voting).

As it is common in non-parametric methods, the k-NN classifier does not involve a real training phase, it just has to keep in memory the train set to make predictions. However, we need to properly choose the hyper-parameter k beforehand, which is actually a regularization parameter.

Let us look at how this works in practice on a classification task

As usual, we first load the Iris Dataset

In [24]:
```
from sklearn import datasets

iris = datasets.load_iris()

# the variable iris now contains inputs ('data' field), targets ('target', field)
# and other stuff. Let us keep the inputs and targets in two separate variables

X = iris.data
y = iris.target

print(iris.feature_names)
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

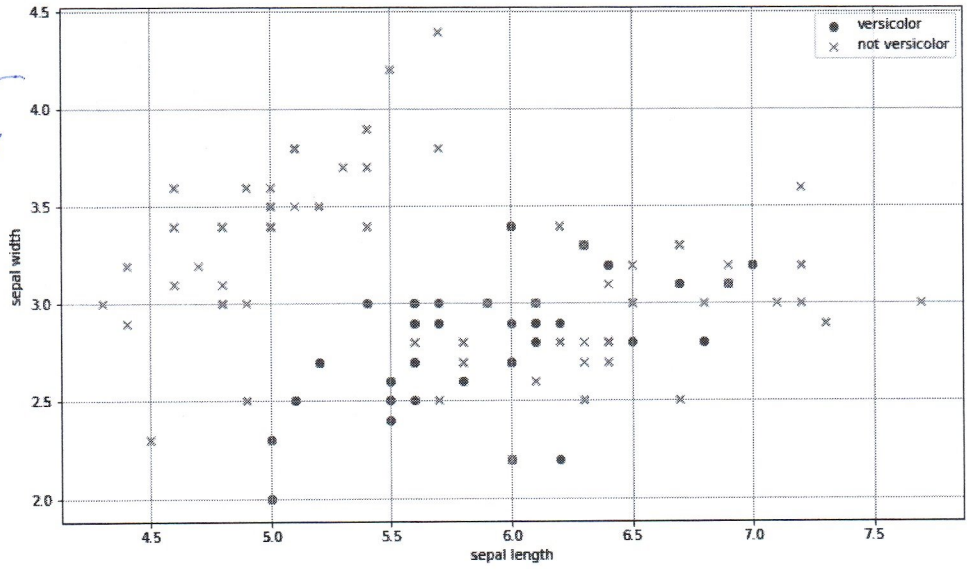
In this case, we would like to learn a classifier to discriminate between 'versicolor' and 'not-versicolor' by looking at the speal length and width. To do so, we split the data in a train set and a test set

In [25]:
```
import numpy as np
# set the seed to get consistent splits
np.random.seed(0)

# we update the targets according to our preferred task
# 0 - setosa, 1 - versicolor, 2 - virginica
y[y == 2] = 0

# let us concatenate X and y in a single matrix
D = np.concatenate((X[:, :2], y.reshape(len(y), -1)), axis=1)

# then we shuffle D
np.random.shuffle(D)

# and we slip the data as 50/20/20
X_train = D[:100,:2]
y_train = D[:100,2]
X_test = D[100:150,:2]
y_test = D[100:150,2]
```

Let us plot the train data on the 2D sepal-length/sepal-width space

In [26]:
```
from matplotlib import pyplot as plt

versicolor = X_train[y_train == 1]
not_versicolor = X_train[y_train == 0]

plt.figure(figsize=(12,7))
plt.scatter(versicolor[:, 0], versicolor[:, 1], label='versicolor')
plt.scatter(not_versicolor[:, 0], not_versicolor[:, 1], label='not versicolor', marker='x')

plt.xlabel('sepal length')
plt.ylabel('sepal width')
plt.grid()
plt.legend()
plt.show()
```

Not an easy task since they're **not** linearly separable

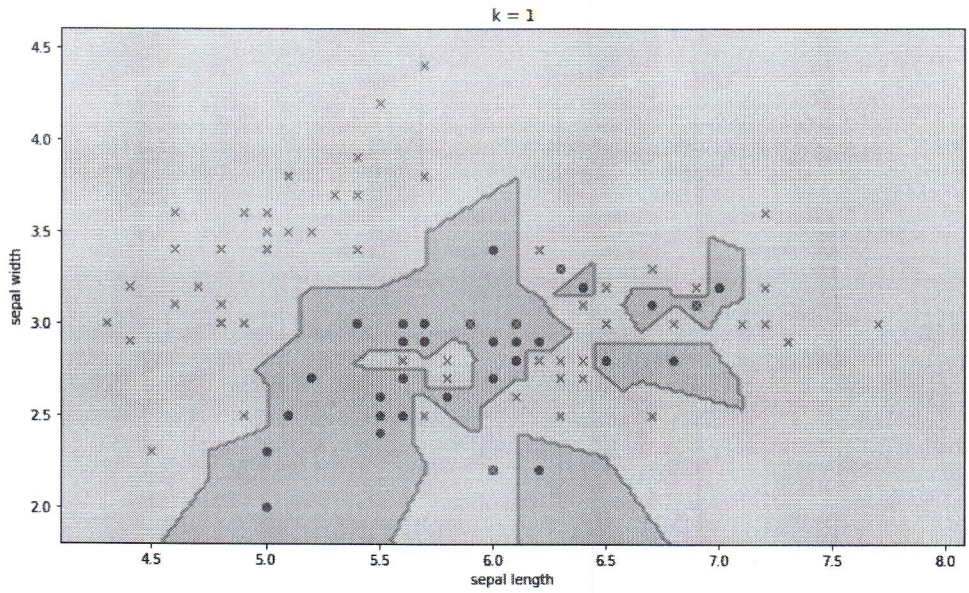


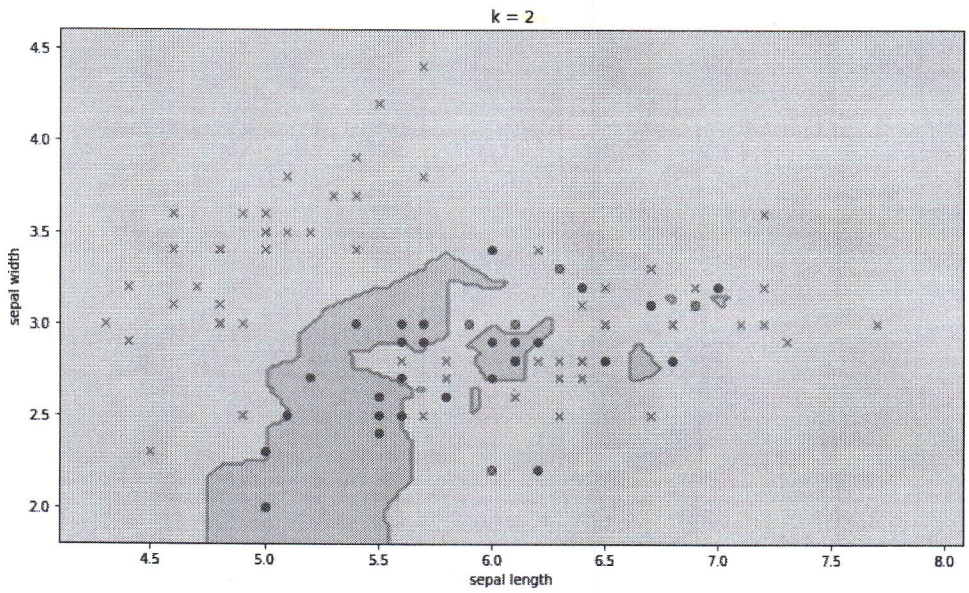
The task looks quite hard, as data are not linearly separable. Let us fit a familiy of k-NN with different values for k, e.g., $k \in \{1, \ldots, 10\}$. For each classifier, we plot the corresponding decision surface with the function below

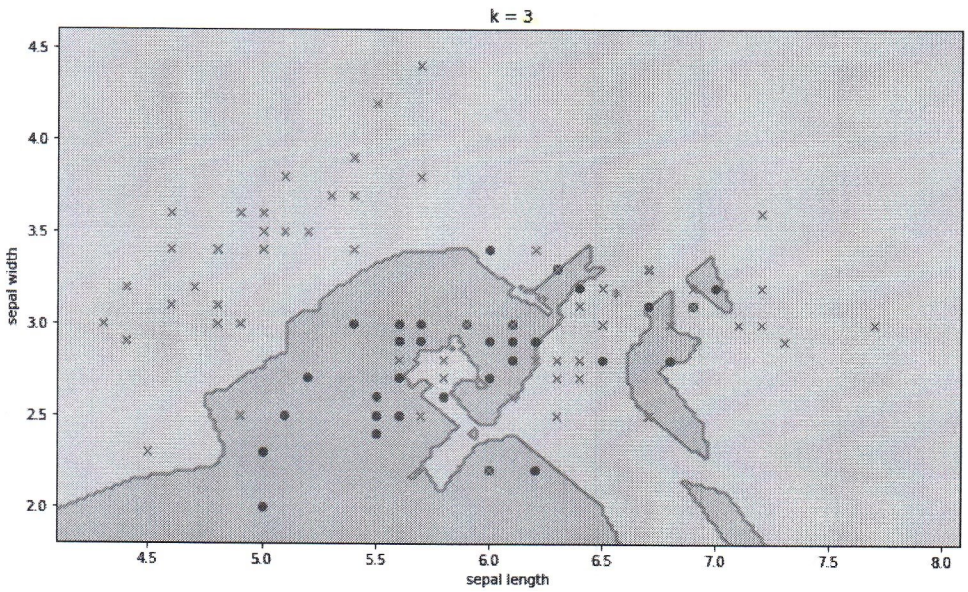
In [27]:
```
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap

# function to plot the decision surface of the k-NN classifier,
# see https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.html#sphx-glr-auto-exa

def knn_decision_surface(X, y, clf, k):

  # let us prepare the decision surface by calling the predict over a grid of points
  h = .02
  x_min, x_max = X[:, 0].min() - .2, X[:, 0].max() + .2
  y_min, y_max = X[:, 1].min() - .2, X[:, 1].max() + .2
  xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
  Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
  Z = Z.reshape(xx.shape)

  # colormap for the contour plot
  cmap_light = ListedColormap(['orange', 'cornflowerblue'])

  # plotting functions
  plt.figure(figsize=(12, 7))
  plt.contourf(xx, yy, Z, cmap=cmap_light, alpha=0.4)
  plt.scatter(versicolor[:, 0], versicolor[:, 1], label='versicolor')
  plt.scatter(not_versicolor[:, 0], not_versicolor[:, 1], label='not versicolor', marker='x')
  plt.xlabel('sepal length')
  plt.ylabel('sepal width')
```

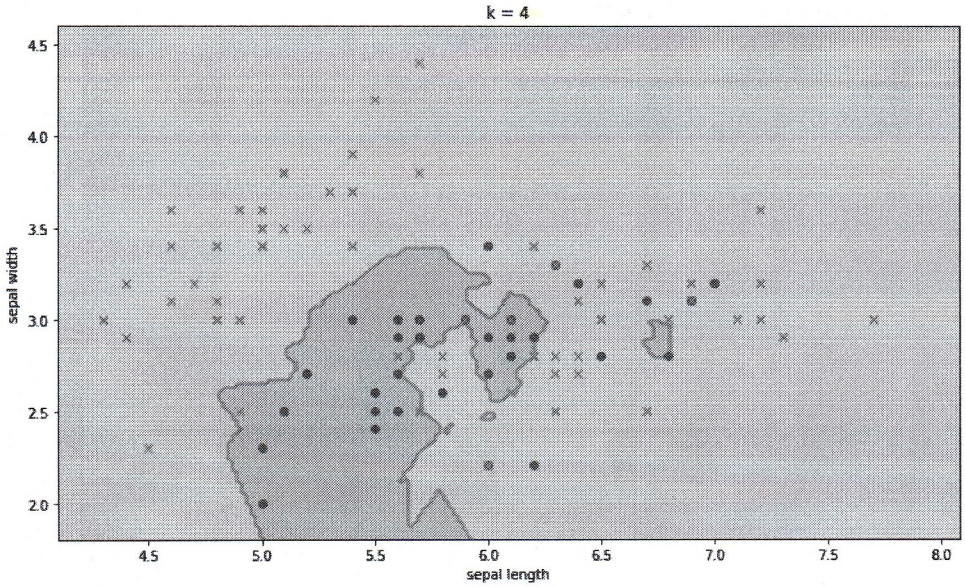
```
    plt.title("k = {}".format(k))
    plt.show()
```

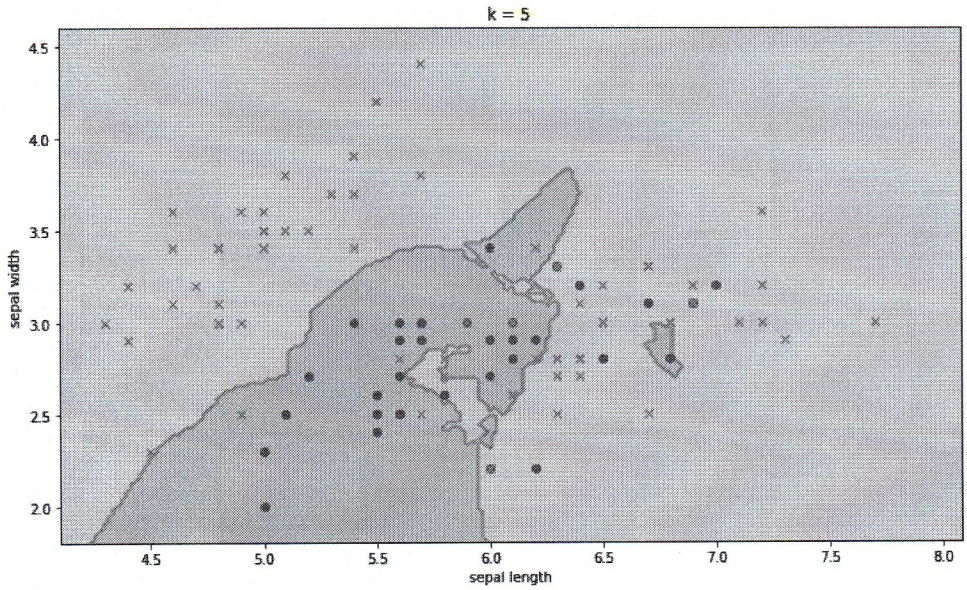
In [28]:
```
from sklearn import neighbors

for k in np.arange(1, 11):
    knn_classifier = neighbors.KNeighborsClassifier(k)
    knn_classifier.fit(X_train[:,:2], y_train)
    knn_decision_surface(X, y, knn_classifier, k)
```

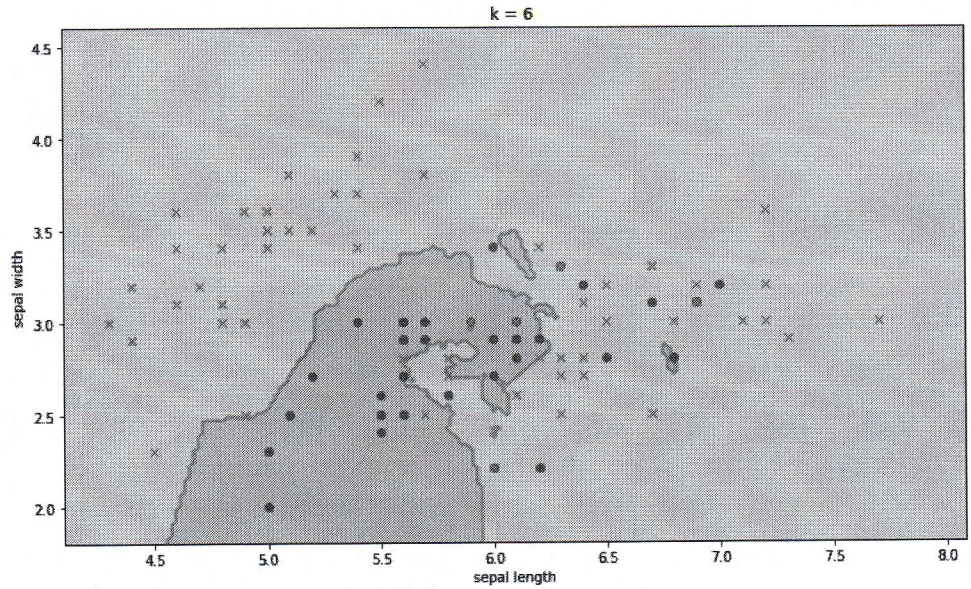


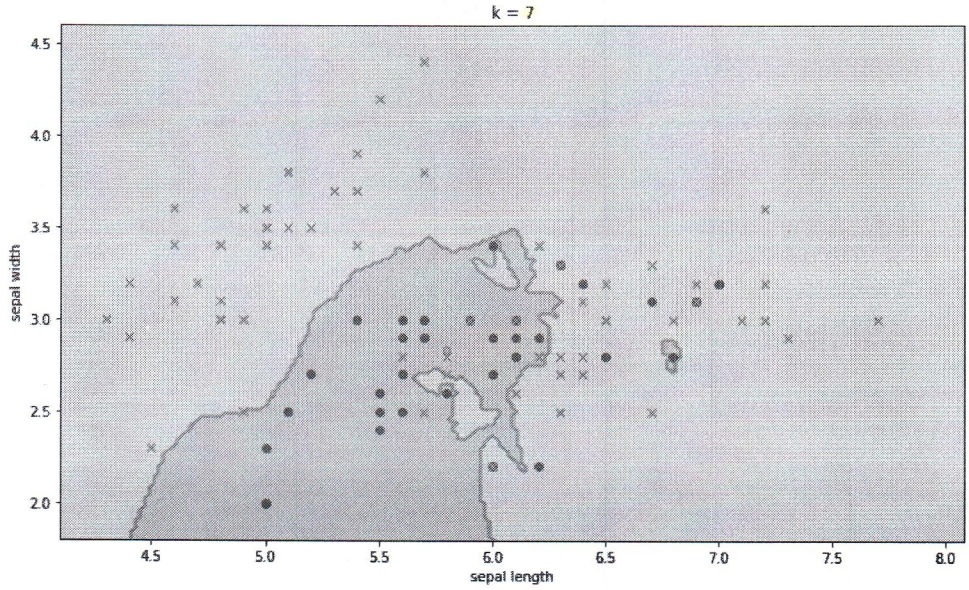


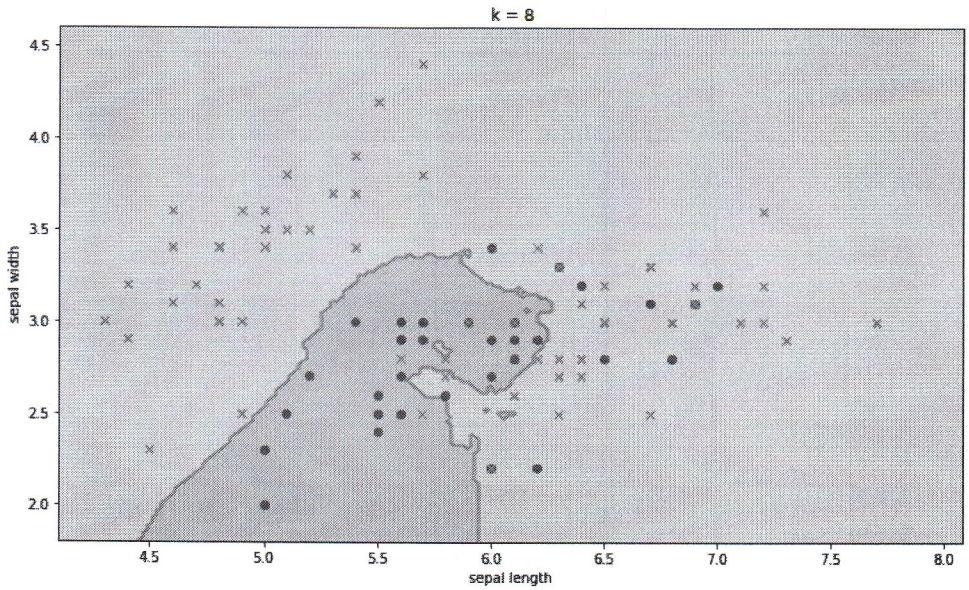


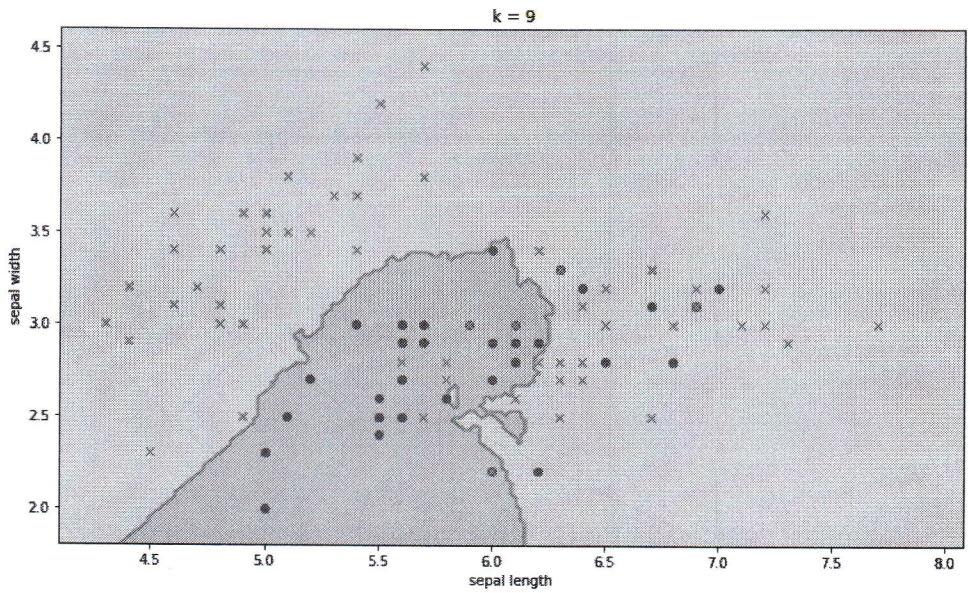

k = 4
sepal width
sepal length
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

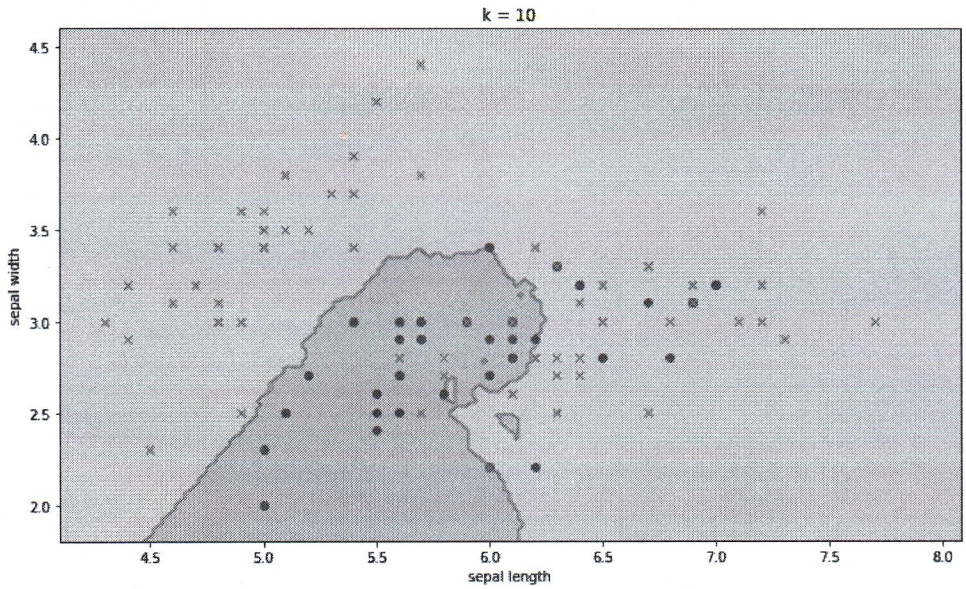

k = 5
sepal width
sepal length
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


k = 6
sepal width
sepal length
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


k = 7
sepal width
sepal length
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

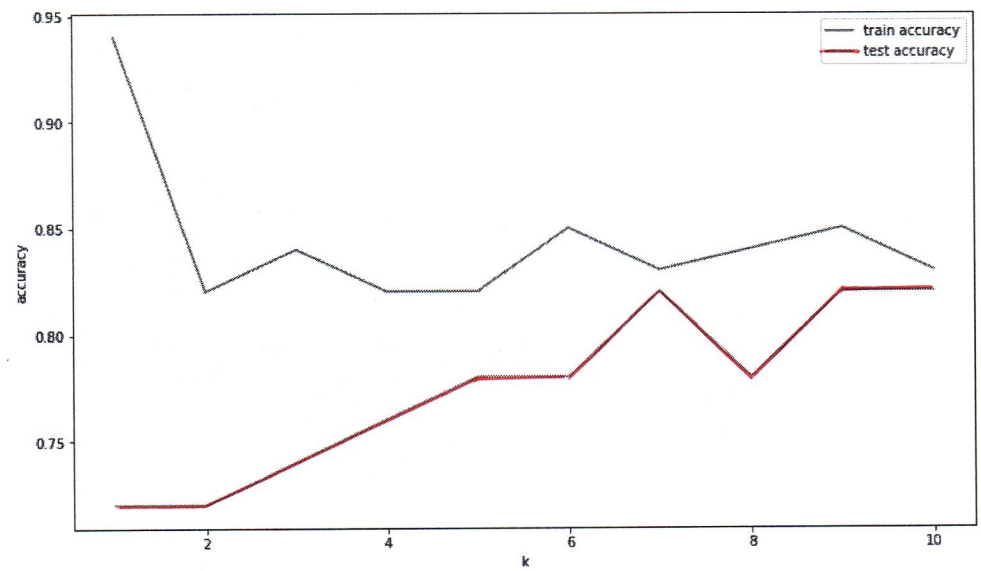

k = 8
sepal width
sepal length
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


k = 9
sepal width
sepal length
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0



As you can see, the shape of the decision surface is quite crazy for a small value of k, whereas it becomes smoother and smoother when we increse k, but several data points are misclassified. What is going on? Let us evaluate the classifiers with the test set

In [29]:
```
parameters = np.arange(1, 11)
train_accuracy = []
test_accuracy = []

for k in parameters:
  knn_classifier = neighbors.KNeighborsClassifier(k)
  knn_classifier.fit(X_train[:,:2], y_train)

  y_hat = knn_classifier.predict(X_train)
  accuracy = sum(y_hat == y_train) / len(y_train)
  train_accuracy.append(accuracy)

  y_hat = knn_classifier.predict(X_test)
  accuracy = sum(y_hat == y_test) / len(y_test)
  test_accuracy.append(accuracy)


plt.figure(figsize=(12, 7))
plt.plot(parameters, train_accuracy, label='train accuracy')
plt.plot(parameters, test_accuracy, label='test accuracy')
plt.xlabel('k')
plt.ylabel('accuracy')
plt.legend()
plt.show()
```



We have a bias-variance trade-off once again. With a small k, the model overfits the train set (small training error, but large test error) and incurs in a large variance. With a greater k, the train accuracy decresases while the test accuracy increases. However, the model struggles to correctly classify all the samples, thus there is supposedly some bias.

Homeworks

Forward Feature Selection

1) Implement the Forward Feature Selection method on the same classification task we considered above with the Backward version. Do you expect any change in hte results?

2) Use the cross-validation to evaluate the features instead of a validation set. Why cross-validation might be preferable in this case?

k-Nearest Neighbors from Scratch

Implement the k-NN classifier from scratch instead of relying on the sklearn off-the-shelf implementation.

Hint: a brute-force strategy to search for the k-nearest neighbors is definitely sufficient for the classification task above.

Principal Components Analysis for Compression

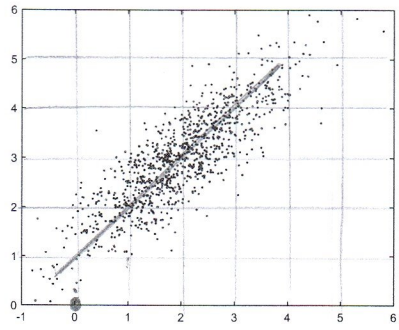
PCA can be seen as a compression method. Try to reconstruct the original dataset X from the transformed dataset $T_{1,2}$ and the weights $W_{1,2}$. Do the same with $T_{3,4}$ and $W_{3,4}$. Which one works better? In which case we could perfectly reconstruct the original dataset?

In [29]:

5 Model Selection

✕ Exercise 5.1

Consider the following dataset:



Draw the direction of the principal components and provide an approximate and consistent guess of the values of the loadings. Are they unique?

✕ Exercise 5.2

Consider the following statement regarding PCA and tell if they are true or false. Provide motivation for your answers.

1. Even if all the input features are on very similar scales, we should still perform mean normalization (so that each feature has zero mean) before running PCA.
2. Given only scores t_i and the loadings W, there is no way to reconstruct any reasonable approximation to x_i.
3. Given input data $x_i \in \mathbb{R}^d$, it makes sense to run PCA only with values of k that satisfy $k \leq d$.
4. PCA is susceptible to local optima, thus trying multiple random initializations may help.

Exercise 5.3

You are analysing a dataset where each observation is an age, height, length, and width of a turtle. You want to know if the data can be well described by fewer than four dimensions, so you decide to do Principal Component Analysis. Which of the following is most likely to be the loadings of the first Principal Component?

1. $(1, 1, 1, 1)$
2. $(0.5, 0.5, 0.5, 0.5)$
3. $(0.71, -0.71, 0, 0)$
4. $(1, -1, -1, -1)$

Provide motivations for your answer.

Exercise 5.4

Which of the following is a reasonable way to select the number of principal components k in a datset with N samples?

1. Choose k to be 99% of N, i.e., $k = \lceil 0.99N \rceil$;
2. Choose the value of k that minimizes the approximation error $\sum_{i=1}^{N} ||x_i - \hat{x}_i||_2^2$;
3. Choose k to be the smallest value so that at least 99% of the variance is retained;
4. Choose k to be the smallest value so that at least 1% of the variance is retained;
5. Identify the elbow of the cumulated variance function.

What changes if the purpose of PCA is visualization?

Exercise 5.5

Are the following statement regarding the *No Free Lunch* (NFL) theorem true or false? Explain why.

1. On a specific task all the ML algorithms perform in the same way;
2. It is always possible to find a set of data where an algorithm performs arbitrarily bad;
3. In a real scenario, when we are solving a specific task all the concepts f belonging to the concept space $\mathcal{F}$ have the same probability to occur;

4. We can design an algorithm which is always correct on all the samples on every task.

Exercise 5.6

A hypercube with side length 1 in d dimensions is defined to be the set of points $(x_1, x_2, \ldots, x_d)$ such that $0 \leq x_j \leq 1$ for all $j = 1, 2, \ldots, d$. The boundary of the hypercube is defined to be the set of all points such that there exists a j for which $0 \leq x_j \leq 0.05$ or $0.95 \leq x_j \leq 1$, i.e., the set of all points that have at least one dimension in the most extreme 10% of possible values). What proportion of the points in a hypercube of dimension 50 are in the boundary?

In this case, we are considering points with small or big L_∞ norm, i.e., a boundary point x is s.t. $||x||_\infty \leq 0.05 \vee ||x||_\infty \geq 0.95$. What happens if we consider L_2 norm for vectors?

Remember that the sphere volume in $2k$ and $2k+1$ dimensions are:

$$V_{2k}(R) = \frac{\pi^k}{k!} R^{2k},$$
$$V_{2k+1}(R) = \frac{2(k!)(4\pi)^k}{(2k+1)!} R^{2k+1},$$

respectively.

Interpret the following result in the light of the so called *Curse of Dimensionality*.

Exercise 5.7

1. Show that the VC dimension of an axis aligned rectangle is 4.
2. Show that the VC dimension of a linear classifier in 2D is 3.
3. Show that the VC dimension of a triangle in the plane is at least 7.

Exercise 5.8

Consider the hypothesis space of the decision trees with attributes with $n = 4$ binary features with at most $k = 10$ leaves (in this case you have less than $n^{k-1}2^{2k-1}$ different trees) and the problem of binary classification.

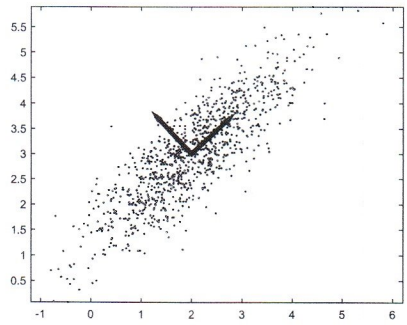
Suppose you found a learning algorithm which is able to perfectly classify a training set of $N = 1000$ samples. What is the lowest error ε you can guarantee to have with probability greater than $1 - \delta = 0.95$? How many samples do you need to halve this error?

Another classifier is able only to get an error of $L_{train}(h) = 0.02$ on your original training set. It is possible to use the same error bound derived in the first case? If not, derive

a bound with the same probability for this case? How many samples do we need to halve the error bound?

Answers

Answer of exercise 5.1



The computed principal components loadings are:

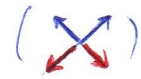
$$\begin{matrix} 0.7287 & -0.6849 \\ 0.6849 & 0.7287 \end{matrix}$$

and a reasonable guess would be:

$$\begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix}$$

we start from the 1st: $[1,1] \rightarrow [1/\sqrt{2}, 1/\sqrt{2}] \rightarrow [\sqrt{2}/2, \sqrt{2}/2]$
from here we can find the second (knowing they're ⊥)
remember to normalize!

The properties satisfied by the principal components are the unity norm and that they are orthogonal. They're not unique, we can change the signs!



Answer of exercise 5.2

1. TRUE Since the principal components are identifying the directions where the most of the variance of the data is present, where the directions is defined as a vector with tail in the origin, we should remove the mean values for each component in order to identify correctly these directions.

$t_i = \tilde{x}_i W$
$t_i \cdot W^T = \tilde{x}_i W W^T$
since it's orthonormal
$t_i \cdot W^T = \tilde{x}_i$

2. FALSE By applying again the loadings matrix W to the scores t_i, thanks to the orthogonality property of W, we are able to reconstruct perfectly the original mean normalized vectors. If we want to reconstruct the original vectors we should also store the mean values for each dimension.

3. TRUE Running it with $k = d$ is possible but usually not helpful and $k > d$ does not make sense.

4. FALSE There is no source of randomization and no initialization point in the algorithm to perform PCA.

Answer of exercise 5.3

Options 1 and 4 cannot be right because the sum of the squared loadings (i.e., their norm) exceeds 1. Options 2 and 3 are both reasonable options.

The second option is most likely correct because we expect this particular set of four variables to be positively correlated with each other in the first principal component. Note that it is fairly common for the loadings of the first principal component to all have the same sign. In such a case, the principal component is often referred to as a size component.

Answer of exercise 5.4

1. FALSE This way we could either include principal components which provides explanation for small amount of variance or exclude some of the most important ones.
2. FALSE This would mean to include all the principal components, which are able to perfectly reconstruct the original dataset;
3. TRUE The cumulated variance of the principal components provides us an estimates on how much of the variance of the original dataset is considered. Keeping an high percentage of it would imply that we are discarding components which does not vary too much or noise.
4. FALSE The same reason of the previous point.
5. TRUE If there is an elbow in the cumulated variance, the inclusion of the following principal components would not improve the representation of the data too much.

Answer of exercise 5.5

1. FALSE Given a specific task we are able to find an algorithm which is likely to perform better than a random guess. This does not mean that the algorithm will perform well also on a generic task.
2. TRUE This is exactly how we are able to prove the NFL theorem, i.e., by showing that on a specific concept an algorithm performs arbitrarily bad and, therefore, on average it is not able to beat a random guess.
3. FALSE In the NFL theorem we are considering all the sample as likely as any

other to be seen, while in real applications some of them have low probability to happen. This is why we are considering specific algorithm for specific tasks.

4. FALSE The NFL theorem does not allow that, since we can always built an instance where we perform arbitrarily bad. That is why we usually consider PAC-Learning which allows to get a limited amount of mistakes with a given probability.

Answer of exercise 5.7

1. Consider 4 points. It is possible to show by enumeration that all the possible labeling are shattered by the rectangle. Consider 5 points. Consider the set of points with maximum and minimum x coordinate and maximum and minimum y coordinates. If all the points are on the rectangle, we consider the labeling which assign alternate labels to the points if you follow the rectangle perimeter. Otherwise, there are at most 4 points in this set. If we label them $+$ and label $-$ the other, it is not possible to shatter this labeling.

2. Again it is at least 3, since it is possible to pick a set of 3 not aligned points and shatter each one of the possible labeling. It is not 4 since:

   - if they form a convex hull, the labeling of different diagonal points with different labels does not allow to shatter the set;
   - if three of them form an hull and the fourth is in the hull we can just label $+$ the three external point and $-$ the remaining one.

3. If we consider a set of points on a circle it is possible to show by enumeration that a triangle ia able to shatter all of them.

Answer of exercise 5.8

Since we have a learner in the version space, we are able to resort to the theorem which states that the error on a hypothesis of the version space ε follows:

$$\mathbb{P}(\exists h \in H,\ L_{true}(h) > \varepsilon) \leq |H| e^{-N\varepsilon} = \delta.$$

Thus:

$$\varepsilon \geq \frac{1}{N}\left(\ln|H| + \ln\left(\frac{1}{\delta}\right)\right) = 0.0286.$$

By looking at this inequality, to halve this error we need to double the number of samples we had originally. Clearly we still require to have have a perfect classifier on the new training set.

In the case we are not allowed to use the previous bound, since the classifier is not able to perfectly classify all the points in the training set. Thus, we might consider an agnostic approach and rely on the fact that:

$$\mathbb{P}(L_{train}(h) - L_{true}(h) > \varepsilon) \leq |H|e^{-2N\varepsilon^2} = \delta.$$

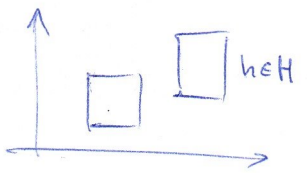
Thus the error bound is:

$$err = L_{train}(h) + \varepsilon \leq L_{train}(h) + \sqrt{\frac{\ln|H| - \ln(\delta)}{2N}} = 0.1397.$$

If we want to halve the error bound we should have $err' = \frac{err}{2}$ and we have:

$$L_{train}(h) + \sqrt{\frac{\ln|H| - \ln(\delta)}{2N'}} \leq err'$$
$$N' \geq \frac{\ln|H| - \ln(\delta)}{2(\frac{err}{2} - L_{train})^2} = 5766.34 \approx 5767.$$

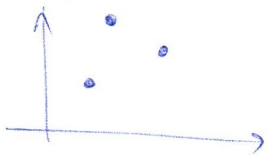
5.7



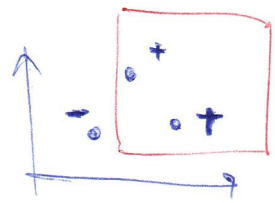
To check that $VC(H) = 4$:

1. prove that an instance of 4 points can be shattered by H
2. prove that an instance of more than 4 points can not be shattered

Consider **3** points in the input space :



an adversary puts the labels →

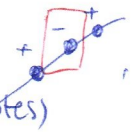


can we shatter? yes
all + are inside, all - are outside

$\Longrightarrow VC(H) \geq 3$

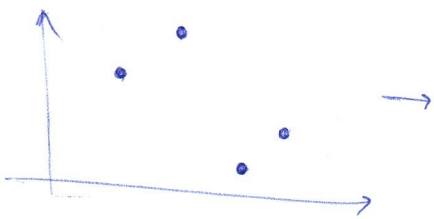
Notice, we choose points, we have to choose them smartly, otherwise the adversary wins!

(actually here yes, because we can but look at the example in notes)

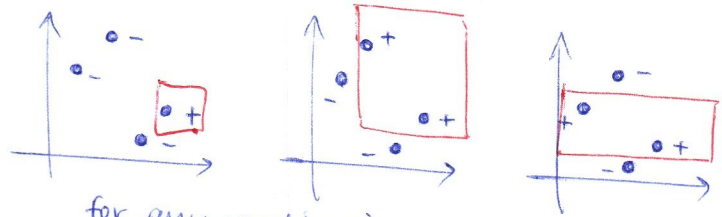


> We don't have to say that all the combinations of points must be shatter-able, we have to say that all the combinations of labels for one particular instance (position)

Let's consider **4** points in the input space :



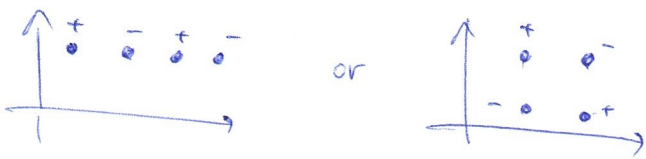
→



for any combination ∃ □

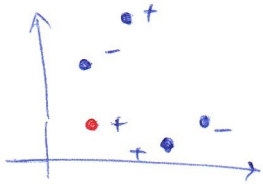
$\Longrightarrow VC(H) \geq 4$

We can fail to choose the points:



Let's consider **5** :

We start from the previous and we see that whenever we add a point the adversary can win:



$\Longrightarrow VC(H) < 5$