3 Machine learning with sklearn

The open-source series of scikit libraries build on the NumPy and SciPy libraries for more domain specific support. In this chapter we briefly introducing the scikit-learn library, or sklearn for short. This library started as a Google Summer of Code project by David Cournapeau and developed into an open source library which now provides a variety of well-established machine learning algorithms. These algorithm together with excellent documentation are available at http://scikit-learn.org/.

The goal of this chapter is to show how to apply machine learning algorithms in a general setting using some classic methods. In particular we will apply Support Vector Machines (SVMs) and Random Forest (RF) classifiers. While many of the methods studied later go beyond these now classic methods, this does not mean that these classic methods are obsolete. Quite the contrary; many applications have limited amount of data where such techniques where more data hungry deep learning methods might not work. Also, the algorithms discussed here are providing some form of baseline to discuss the advanced methods like probabilist reasoning and deep learning. Our aim here is to demonstrate that applying machine learning methods based on such machine learning libraries is not too difficult.

Fig. 3.1 Illustration of sklearn components and a typical workflow for different data and machine learning goals.

An outline of the algorithms and a typical work flow is shown in Fig. 3.1. As can
be seen, the machine learning methods are there divided into classification, regression, clustering, and dimensionality reduction. While we will later discuss that all of these are ultimately related we start by treating the methods first as a black-box. We specifically outline in this chapter a typical machine learning setting for classification. In some application it is possible to achieve already sufficient performance. However, applying machine learning to more challenging cases and avoiding pitfalls requires some deeper understanding of the algorithms. We will only outline the ideas behind these first classic methods in the second half of this chapter. Our aim for the rest of the book is to dwell much deeper into the principles behind machine learning that includes probabilistic and deep learning methods.

### 3.1 Classification of Iris data

We start with a basic classification example using the classic Iris data set that were already discussed somewhat in the previous chapter to demonstrate how to read data into NumPy arrays. We will now apply a SVM model from the scikit-learn library. This is implementation is actually a wrapper to the very popular SVMLIB implementation by Chih-Chung Chang and Chih-Jen Lin that has been very popular for classification applications. IN the following we discuss the example code to apply this classifier to the Iris data set.

We start as usual by importing the necessary libraries and importing the data similar to the program discussed in the last chapter. We choose here to split the data into a training set and test set by using every second data point as training point and every other as test point. This is accomplished with the index specifications \(0:-1:2\) which is a list that starts at index "0", iterates until the end specified by index "-1" and uses a step of "2". Since the data are ordered and well balanced in the original data file, this will leave us also with a balanced data set. Also, instead of using the names features and target we decided to shorten the notation by denoting the input features as \(x\) and the targets as \(y\) values. Finally, we used the NumPY function \(\text{int32}(\cdot)\) to convert the target values to integers. While the sklearn routines also accept real values, we will also show an example of tensorflow below that requires integer targets.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, metrics, model_selection
from sklearn.ensemble import RandomForestClassifier

iris_data = np.loadtxt('iris.data', delimiter=',')

train_x = iris_data[0:-1:2,0:4]
train_y = np.int32(iris_data[0:-1:2,4])
test_x = iris_data[1:-1:2,0:4]
test_y = np.int32(iris_data[1:-1:2,4])
```

The next section of code encapsulates a basic machine learning session. In the first step we specify the model, which is here a support vector classifier (SVC) from the svm methods of sklearn. We then apply a training algorithm provided in the fit function that
requires the training data, both feature values and labels as this is supervised learning. After the model is trained we can use the trained model to predict new data. We use the predict method on the feature values of the test data to predict the corresponding labels. Finally, evaluate how good the predictions are by comparing the labels with the test labels. In this case we simply count the percentage of correct labels, which is called the accuracy. The accuracy for this evaluation is around 0.97 percent, which corresponds to only two incorrect classification.

```python
# model
model = svm.SVC(kernel='linear')
# train
model.fit(train_x, train_y)
# prediction
predicted_y=model.predict(test_x)
# evaluation
print('Percentage correct (accuracy) of SVM:', np.mean(test_y == predicted_y))
```

Before we move on to discuss evaluations in some more detail, let us apply another model that of the popular random forest classifier. The corresponding code is shown below.

```python
# model
model = RandomForestClassifier(n_estimators=10)
# train
model.fit(train_x, train_y)
# prediction
predicted_y=model.predict(test_x)
# evaluation
print('Percentage correct (accuracy) of RF:', np.mean(test_y == predicted_y))
```

The result of this classifier is slightly less with an accuracy of around 95 percent, equating to four misclassification. This seem to indicate that the SVM method is superior to RFs. However, this conclusion should not be made as argued further below.

Finally, we show below and example implementation if a basic neural network with tensorflow. We are thereby using the contributed ‘learn’ module which is an interface that frame tensorflow in the sklearn syntax. The main point here is to show that neural networks discussed later can also be framed into the basic model definition, training by fitting the mode parameters to the training data, and making predictions on test data.

```python
import tensorflow as tf
from tensorflow.contrib import learn

#model
model=learn.DNNClassifier(
 feature_columns=[tf.contrib.layers.real valued_column(''
 dimension=4)], hidden_units=[10, 20, 10], n_classes=3)
# train
model.fit(train_x, train_y, steps = 100, batch_size = 75)
# prediction
predicted_y = list(model.predict(test_x, as_iterable=True))
# evaluation
print('Percentage correct (accuracy) of MLP:',
      np.mean(test_y == predicted_y))

Running this code repeatedly will give different performances values in every run. Most often these values will be smaller than the ones achieved with SVM and RF above, but some are even larger. This demonstrates already that some more careful consideration of such methods are important.

### 3.2 Performance measures and evaluations

We used above the percentage of misclassification as an objective function to evaluate the performance of the model. This is a common choice and often a good start in our examples, but there are other commonly used evaluation measures we should know about. Let us consider first a binary classification case where it is common to call one class ‘positive’ and the other the ‘negative’. This nomenclature comes from diagnostics such as trying to decide if a person has some disease based on some clinical tests. We can then define the following four performance indicators,

- **True Positive (TP):** Number of correctly predicted positive samples
- **True Negative (TN):** Number of correctly predicted negative samples
- **False Positive (FP):** Number of incorrectly predicted positive samples
- **False Negative (FN):** Number of incorrectly predicted negative samples

The previously used measure of accuracy correspond to

\[
\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (3.1)
\]

in this notation. Also, it is also common to report a rate when normalizing these values to their respective maximum number. That is

- **True Positive Rate (TPR):** \( \frac{\text{TP}}{\text{all positive samples}} \)
- **True Negative Rate (TNR):** \( \frac{\text{TN}}{\text{all negative samples}} \)
- **False Positive Rate (FPR):** \( \frac{\text{FP}}{\text{all negative samples}} \)
- **False Negative Rate (FNR):** \( \frac{\text{FN}}{\text{all positive samples}} \)

These numbers are often reported in a table that is called a **confusion matrix.** Such a confusion matrix can also be generalized to more than two classes when comparing the ratio of correctly predicted labels out of the number of examples of this or another class. An example is shown in Fig. 3.2 for the Iris example. This plot is produced with the following code.

```python
# Confusion Matrix
cm = metrics.confusion_matrix(test_y, predicted_y)
```
fig. 3.2  example of a confusion matrix for the iris data set when classified with an svm.

there are also many derived measures. for example, in many cases we want to have a good prediction of at least the positive class. for example, we might want to diagnose an illness and it hence important not to miss any samples. so we would like to maximize the TPR, which is also called recall or precision as it gives us a measure of how good we can detect positive samples. another way of writing this is

\[
Recall = \frac{TP}{TP+FN} = TPR \tag{3.2}
\]

Of course, we could always make this 100% when simply recalling all samples as positive. It is hence important to balance this with how precise the prediction is by counting how many of the positively predicted samples are correct out of all the predicted,

\[
Precision = \frac{TP}{TP+FP} = 1 - FPR \tag{3.3}
\]

Another popular way to summarize the precision and recall is to take their arithmetic mean

\[
F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{3.4}
\]

This measure has the index 1 as it is the balanced choice of the more general definition that weights the two terms differently

\[
F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}} \tag{3.5}
\]
These values can be easily calculated from a test set, and sklearn includes the calculation of these measures in a method to calculate the confusion matrix. For example, the values mentioned above can be calculated for the Iris example as shown in the following code.

```python
# Model Evaluation
print('The main classification metrics for iris data:

', metrics.classification_report(test_y, predicted_y))
```

The main classification metrics for iris data:

<table>
<thead>
<tr>
<th>precision</th>
<th>recall</th>
<th>f1-score</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.00</td>
<td>1.00</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>0.80</td>
<td>0.96</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>0.95</td>
<td>0.75</td>
<td>24</td>
</tr>
</tbody>
</table>

avg / total 0.92 0.91 0.90 74

**Fig. 3.3** Example of ROC curve. The ideal classifier is in the upper left corner.

Note that the application of these measures encapsulate the importance that a user places onto specific characteristics. This is similar of discussing which car is better. Some might find that larger horsepowers are good, while others want a car to consume as little gas as possible. Hence, there is not a simple best measure. In the same sense, in many cases we have the option of a trade-off between precision and recall, for example by tuning some parameters of the machine learning algorithm. This trade-off is often visualized as a ROC curve. ROC stands for a receiver operating characteristics, which come from their historical use in evaluating communication equipment. An example is shown in Fig. 3.3. Ideally we want the TPR to be one and the FPR to be zero, which corresponds to a point in the upper left corner. While this is not typically not always possible, the next best is that the TPR is as close to one for all possible values of FPR. In contrast a random binary classification corresponds to the diagonal in this
plot, which has a value of 0.5 as the area under this curve. Hence, when comparing two algorithms we generally prefer an algorithm that has a larger area under the ROC curve, or an area that is close to one. For many applications we have curves that are somewhere in between.

### 3.3 Support Vector Machines (SVM) *

#### 3.3.1 Linear classifiers with large margins

In this section we briefly outline the basic idea behind Support Vector Machines (SVM) that has been instrumental in a first wave of industrial applications due to its robustness and ease of use. SVMs, and the underlying statistical learning theory, has been worked out by Vladimir Vapnik since the early 1960, but some further breakthroughs were made in the late 1990 with some collaborators like Corinna Cortes, Chris Burges, Alex Smola, and Bernhard Schölkopf to name but a few. While we outline some of the underlying formulas, we do not derive all the steps and will mainly try to provide some intuition. We basically show the formulas to enable some discussion.

The basic SVMs are concerned with binary classification. Figure 3.4 shows an example of two classes, depicted by different symbols, in a two dimensional attribute space. We distinguish here attributes from features as follows. Attributes are the raw measurements, whereas features can be made up by combining attributes. For example, the attributes $x_1$ and $x_2$ could be combined in a feature vector $(x_1, x_1x_2, x_2, x_1^2, x_2^2)^T$. This will become important later. Our training set consists of $m$ data with attribute values $x^{(i)}$ and labels $y^{(i)}$. We put the superscript index $i$ in brackets so it is not mistaken as a power. For this discussion we chose the binary labels of the two classes as represented with $y \in \{-1, 1\}$. This will nicely simplify some equations.

\[
 w^T x + b = 0
\]

Fig. 3.4 Illustration of linear support vector classification.

The two classes in the figure 3.4 can be separated by a line, which can be parameterized by

\[
 w_1 x_1 + w_2 x_2 - b = w^T x - b = 0.
\]

While the left hand side shows the lines equation with its components in two dimensions, the next expression in a matrix notation is the same in any dimension. Of course,
Support Vector Machines (SVM) *

In three dimensions we would talk about a plane. In general, we will talk about a hyperplane in any dimensions. The particular hyperplane is the dividing or separating hyperplane between the two classes. We also introduce what the margin $\gamma$, which is the perpendicular distance between the dividing hyperplane and the closest point.

The main point to realize now is that the dividing hyperplane that maximizes the margin is likely a good choice. Why is that? We should assume that the training data, shown in the figure, are some unbiased examples of the true underlying density function describing the distribution of points within each class and thus representative of the most likely data. It is then likely that new data points, which we want to classify, are close to the already existing data points. Hence, if we make the separating hyperplane as far as possible from each point, than it is most likely to not make wrong classification on future data points. Or, with other words, a separating hyperplane like the one shown as dashed line in the figure, is likely to generalize much worse than the maximum margin hyperplane. So the maximum margin hyperplane can be considered the best choice if we assume that both classes have the same expected variation from the training data. Such a maximum margin classifier, can hence be expected be a good choice.

Let us formalize the maximization of the margin a bit more mathematical. Learning a linear maximum margin classifier on labeled data means finding the parameters ($w$) and $b$ that maximizes the margin. For this we could computer the distances of each point from the hyperplane, which is simply a geometric exercise,

$$
\gamma^{(i)} = y^{(i)} \left( \frac{w}{||w||}^T x^{(i)} + \frac{b}{||w||} \right).
$$

(3.7)

The vector $w/||w||$ is the normal vector of the hyperplane, a vector of unit length perpendicular to the hyperplane ($||w||$) is the Euclidean length of the vector $w$. The margin we want to maximize is the distance to the closest point,

$$
\gamma = \min_i \gamma^{(i)}.
$$

(3.8)

By looking at equation 3.7 we see that maximizing $\gamma$ is equivalent to minimizing $||w||$, or, equivalently, of minimizing

$$
\min_{w,b} \frac{1}{2} ||w||^2.
$$

(3.9)

More precisely, we want to maximize this margin under the constraint that no training data lies within the margin,

$$
\begin{align*}
  w^T x^{(i)} + b &\geq 1 \quad \text{for } y^{(i)} = 1 \\
  w^T x^{(i)} + b &\leq -1 \quad \text{for } y^{(i)} = -1,
\end{align*}
$$

(3.10) (3.11)

which can nicely be combines with our choice of class representation as

$$
-(y^{(i)}(w^T x + b) - 1) \leq 0.
$$

(3.12)

Thus we have a quadratic minimization problem with linear inequalities as constraint. Taking a constrain into account can be done with a Lagrange formalism. For this we
simply add the constraints to the main objective function with parameters $\alpha_i$ called Lagrange multipliers,

$$L^P(w, b, \alpha_i) = \frac{1}{2}||w||^2 - \sum_{i=1}^{m} \alpha_i [y^{(i)}(w^T x + b) - 1].$$  \hspace{1cm} (3.13)

The Lagrange multipliers determine how well the constrains are observed. In the case of $\alpha_i = 0$, the constrains do not matter. In order conserve the constrains, we should thus make these values as big as we can. Finding the maximum margin classifier is given by

$$p^* = \min_{w, b} \max_{\alpha_i} L^P(w, b, \alpha_i) \leq p^* = \max_{\alpha_i} \min_{w, b} L^D(w, b, \alpha_i) = d^*. $$ \hspace{1cm} (3.14)

In this formula we also added the formula when interchanging the min and max operations, and the reason for this is the following. It is straight forward to solve the optimization problem on the left hand side, but we can also solve the related problem on the right hand side which turns out to be essential when generalizing the method to nonlinear cases below. Moreover, the equality hold when the optimization function and the constrains are convex$^1$. So, if we minimize $L$ by looking for solutions of the derivatives $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$, we get

$$w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}$$ \hspace{1cm} (3.15)

$$0 = \sum_{i=1}^{m} \alpha_i y^{(i)}$$ \hspace{1cm} (3.16)

Substituting this into the optimization problem we get

$$\max_{\alpha_i} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j} y^{(i)} y^{(j)} \alpha_i \alpha_j x^{(i)T} x^{(j)},$$ \hspace{1cm} (3.17)

subject to the constrains

$$\alpha_i \geq 0$$ \hspace{1cm} (3.18)

$$\sum_{i=1}^{m} \alpha_i y^{(i)} = 0.$$ \hspace{1cm} (3.19)

From this optimization problem it turns out that the $\alpha_i$'s of only a few examples, those ones that are lying on the margin, are the only ones with have $\alpha_i \neq 0$. The corresponding training examples are called support vectors. The actual optimization can be done with several algorithms. In particular, John Platt developed the sequential minimal optimization (SMO) algorithm that is very efficient for this optimization problem. Please note that the optimization problem is convex and can thus be solved very efficiently without the danger of getting stuck in local minima.

$^1$Under these assumptions there are other conditions that hold, called the Karush-Kuhn-Tucker conditions, that are useful in providing proof of the convergence of the methods outlined here.
Once we found the support vectors with corresponding \( \alpha_i \)'s, we can calculate \( (w) \) from equation 3.15 and \( b \) from a similar equation. Then, if we are given a new input vector to be classified, this can then be calculated with the hyperplane equation 3.6 as

\[
y = \begin{cases} 
1 & \text{if } \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)T} x > 0 \\
-1 & \text{otherwise}
\end{cases}
\]  

(3.20)

Since this is only a sum over the support vectors, which should be only a few data points from the training set, classification becomes very efficient after training.

### 3.3.2 Soft margin classifier

So far we only discussed the linear separable case. But how about the case when there are overlapping classes? It is possible to extend the optimization problem by allowing some data points to be in the margin while penalizing these points somewhat. We include therefore some slag variables \( \xi_i \) that reduce the effective margin for each data point, but we add to the optimization a penalty term that penalizes if the sum of these slag variables are large,

\[
\min_{w,b} \frac{1}{2} ||w||^2 + C \sum_i \xi_i,
\]

subject to the constrains

\[
y^{(i)}(w^T x + b) \geq 1 - \xi_i \\
\xi_i \geq 0
\]

(3.22)  
(3.23)

The constant \( C \) is a free parameter in this algorithm. Making this constant large means allowing less points to be in the margin. This parameter must be tuned and it is advisable to at least try to vary this parameter to verify that the results do not dramatically depend on an initial choice.

### 3.3.3 Nonlinear Support Vector Machines

We have treated the case of overlapping classes while assuming that the best we can do is a linear separation. But what if the underlying problem is separable with a function \( f \) that just might be more complex, for example such as the case shown in Fig. 3.5. We will now look into the non-linear generalization of the SVM.

When discussing regression we started with the linear case and then discussed non-linear extensions such as regressing with polynomial functions. For example, a linear function in two dimensions (two attribute values) is given by

\[
y = w_0 + w_1 x_1 + w_2 x_2,
\]

(3.24)

and an example of a non-linear function, that of an polynomial of 3rd order, is given by

\[
y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2.
\]

(3.25)

The first case is a linear regression of a feature vector...
We can also view the second equation as that of linear regression on a feature vector

\[ \mathbf{x} \rightarrow \phi(\mathbf{x}) = \begin{pmatrix} x_1 \\ x_2 \\ x_1x_2 \\ x_1^2 \\ x_2^2 \end{pmatrix}, \]  

(3.27)

which can be seen as a mapping function \( \phi(\mathbf{x}) \) of the original attribute vector. We call this mapping a feature map. Thus, we can use the above maximum margin classification method in non-linear cases if we replace all occurrences of the attribute vector \( \mathbf{x} \) with the mapped feature vector \( \phi(\mathbf{x}) \). There are only three problems remaining. One is that we don’t know what the mapping function should be. The somewhat solution to this will be that we try out some functions and see which work best. We will get into this a bit more below. The second problem is that we have the problem of overfitting as we might use too many feature dimensions and corresponding free parameters \( w_i \). In the next section we provide a glimpse of an argument why SVMs might somewhat address this problem. And the third problem is that with an increased number of dimensions, the evaluation of the equations becomes more computational intensive. However, there is a great trick to alleviate the last problem in the case when the methods only rely on dot products, like in the case of the formulation in the dual problem. The function to be minimized in this formulation, equation 3.17 with the feature maps, only depends on the dot products \( \phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)}) \). Also, when predicting the class for a new input vector \( \mathbf{x} \) from equation 3.15 when adding the feature maps, we only need the resulting values for the dot products \( \phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}) \) which can sometimes be represented as function called Kernel function.

\[ K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z}). \]

(3.28)

Instead of actually specifying a feature map, which is often a guess to start, we could actually specify a Kernel function. For example, let us consider a quadratic feature map.
\[ K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T \mathbf{z} + c)^2. \] (3.29)

We can then try to write this in the form of equation 3.28 to find the corresponding feature map. That is

\[
K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T \mathbf{z})^2 + 2c \mathbf{x}^T \mathbf{z} + c^2
= (\sum_i x_i z_i)^2 + 2c \sum_i x_i z_i + c^2
= \sum_j \sum_i (x_i x_j) (z_i z_j) + \sum_i (\sqrt{2c} x_i) (\sqrt{2c} z_i) + c^2
= \phi(\mathbf{x})^T \phi(\mathbf{z}),
\] (3.33)

with

\[
\phi(\mathbf{x}) = \begin{pmatrix}
x_1 x_1 \\
x_1 x_2 \\
\vdots \\
x_n x_1 \\
x_n x_2 \\
\vdots \\
\sqrt{2c} x_1 \\
\sqrt{2c} x_2 \\
\vdots \\
c
\end{pmatrix},
\] (3.34)

The dimension of this feature vector is \(O(n^2)\) for \(n\) original attributes. Thus, evaluating the dot product in the mapped feature space is much more time consuming than calculating the Kernel function which is just the square of the dot product of the original attribute vector. The dimensionality Kernels with higher polynomials is quickly rising, making the benefit of the Kernel method even more impressive.

While we have derived the corresponding feature map for a specific Kernel function, this task is not always easy and not all functions are valid Kernel functions. We have also to be careful that the Kernel functions still lead to convex optimization problems. In practice, only a small number of Kernel functions is used. Besides the polynomial Kernel mentioned before, one of the most popular is the Gaussian Kernel,

\[ K(\mathbf{x}, \mathbf{z}) = \exp -\frac{||\mathbf{x} - \mathbf{z}||^2}{2\gamma^2}, \] (3.35)

which corresponds to an infinitely large feature map.

As mentioned above, a large feature space corresponds to a complex model that is likely to be prone to overfitting. We must therefore finally look into this problem. The key insight here is that we are already minimizing the sum of the components of the parameters, or more precisely the square of the norm \(||\mathbf{w}||^2\). This term can be viewed as regularization which favours a smooth decision hyperplane. Moreover, we have discussed two extremes in classifying complicated data, one was to use Kernel functions to create high-dimensional non-linear mappings and hence have a high-dimensional separating hyperplane, the other method was to consider a low-dimensional separating hyperplane and interpret the data as overlapping. The last
method includes a parameter $C$ that can be used to tune the number of data points that we allow to be within the margin. Thus, we can combine these two approaches to classify non-linear data with overlaps where the soft margins will in addition allow us to favour more smooth dividing hyperplanes.

### 3.3.4 Regularization and parameter tuning

What if we can not divide the data with a hyperplane and we have to consider non-linear separators. Don’t we then run into the same problems as outlined before, specifically the bias-variance tradeoff? Yes, indeed, this is still a challenge, and our aim is really to work on this problem. But before going there it is useful to formalize the linear separable case in some detail as the representation of the optimization problem will be a key in applying some tricks later.

In practice we have to consider several free parameters when applying support vector machines. First, we have to decide which Kernel function to use. Most packages have a number of choices implemented. We will use for the following discussion the Gaussian Kernel function with width parameter $\gamma$. Setting a small value for $\gamma$ and allowing for a large number of support vectors (small $C$), corresponds to a complex model. In contrast, larger width values and regularization constant $C$ will increase the stiffness of the model and lower the complexity. In practice we have to tune these parameters to get good results. To do this we need to use some form of validation set, as discussed in section 2?, and k-times cross validation is often implemented in the software packages. An example of the SVM performance (accuracy) on some examples (Iris Data set from the UCI repository; From Broadman and Trappenberg, 2006) is shown in figure 3.6 for several values of $\gamma$ and $C$. It is often typical that there is a large area where the SVM works well and has only little variations in terms of performance. This robustness has helped to make SVMs practical methods that often outperform other methods. However, there is often also an abrupt onset of the region where the SVM fails, and some parameter tuning is hence required. While just trying a few settings might be sufficient, some more systematic methods such as grid search or simulated annealing also work well.

![Fig. 3.6](image.png) **Fig. 3.6** Illustration of SVM accuracy for different values of parameters $C$ and $\gamma$. 
3.3.5 Statistical learning theory and VC dimension

SVMs are good and practical classification algorithms for several reasons, including the advantage of being convex optimization problem that can be solved with quadratic programming, have the advantage of being able to utilize the Kernel trick, have a compact representation of the decision hyperplane with support vectors, and turn out to be fairly robust with respect to the hyper parameters. However, in order to be good learners, they need to moderate the variance-bias tradeoff discussed in section ?? A great theoretical contributions of Vapnik and colleagues was the embedding of supervised learning into statistical learning theory and to derive some bounds that make statements on the average ability to learn form data. We outline here briefly the ideas and state some of the results. We discuss this issue here in the context of binary classification, although similar observations can be made in the case of multiclass classification and regression.

We start again by stating our objective, which is to find a hypothesis which minimized the generalization error. To state this a bit more differentiated and to use the nomenclature common in these discussions, we call the error function here the risk function $R$. In particular, the expected risk for a binary classification problem is the probability of misclassification,

$$ R(h) = P(h(x) \neq y) \tag{3.36} $$

Of course, we generally do not know this density function, though we need to approximate this with our validation data. We assume thereby again that the samples are iid (independent and identical distributed) data, and can then estimate what is called the empirical risk,

$$ \hat{R}(h) = \frac{1}{m} \sum_{i=1}^{m} \mathbb{I}(h(x^{(i)}; \theta) = y^{(i)}). \tag{3.37} $$

We use here again $m$ as the number of examples, but note that this is here the number of examples in the validations set, which is the number of all training data minus the ones used for training. Also, we will discuss this empirical risk further, but note that it is better to use the regularized version that incorporates a smoothness constrain such as

$$ \hat{R}_{rmreg}(h) = \frac{1}{m} \sum_{i} \mathbb{I}(h(x^{(i)}; \theta) = y^{(i)}) - \lambda \|w\|^2 \tag{3.38} $$

in the case of SVM, where $\lambda$ is a regularization constant. Thus, wherever $\hat{R}(h)$ is used in the following, we can replace this with $\hat{R}_{rmreg}(h)$. Empirical risk minimization is the process of finding the hypothesis $\hat{h}$ that minimizes the empirical risk,

$$ \hat{h} = \arg \min_{h} \hat{R}(h). \tag{3.39} $$

The empirical risk is the MLE of the mean of a Bernoulli-distributed random variable with true mean $R(h)$. Thus, the empirical risk is itself a random variable for each possible hypothesis $h_i$. Let us first assume that we have $k$ possible hypothesis $h_i$, We
now draw on a theorem by Hoeffding called the **Hoeffding inequality** that provides an upper bound for the sum of random numbers to its mean,

\[ P(|R(h_i) - \hat{R}(h_i)| > \gamma) \leq 2 \exp(-2\gamma^2 m). \] (3.40)

This formula states that there is a certain probability that we make an error larger than \( \gamma \) for each hypothesis of the empirical risk compared to the expected risk, although the good news is that this probability is bounded and that the bound itself becomes exponentially smaller with the number of validation examples. This is already an interesting result, but we now want to know the probability that some, out of all possible hypothesis, are less than \( \gamma \). Using the fact that the probability of the union of several events is always less or equal to the sum of the probabilities, one can show that with probability \( 1 - \delta \) the error of a hypothesis is bounded by

\[ |R(h) - \hat{R}(h)| \leq \sqrt{\frac{1}{2m} \log \frac{2k}{\delta}}. \] (3.41)

This is a great result since it shows how the error of using an estimate the risk, the empirical risk that we can evaluate from the validation data, is getting smaller with training examples and with the number of possible hypothesis.

![Fig. 3.7](image)

**Fig. 3.7** Illustration of VC dimensions for the class of linear functions in two dimensions.

While the error scales only with the log of the number of possible hypothesis, the values goes still to infinite when the number of possible hypothesis goes to infinite, which much more resembles the situation when we have parameterized hypothesis. However, Vapnik was able to show the following generalization in the infinite case, which is that given a hypothesis space with **Vapnic-Chervonenecis** dimension \( VC(\{h\}) \), then, with probability \( 1 - \delta \), the error of the empirical risk compared to the expected risk (true generalization error) is

\[ |R(h) - \hat{R}(h)| \leq O \left( \sqrt{\frac{VC}{m} \log \frac{m}{VC} + \frac{1}{m} \log \frac{1}{\delta}} \right). \] (3.42)

The VC dimensions is thereby a measure of how many points can be divided by a member of the hypothesis set for all possible label combinations of the point. For example, consider three arbitrary points in two dimensions as shown in figure 3.7, and let us consider the hypothesis class of all possible lines in two dimensions. I can always divide the three points under any class membership condition, of which two examples
are also shown in the figure. In contrast, it is possible to easily find examples with four points that cannot be divided by a line in two dimensions. The VC dimension of lines in two dimensions is hence $VC = 3$.\(^2\)

### 3.3.6 Support Vector Regression

While we have mainly discussed classification in the last few sections, it is time to consider the more general case of regression and to connect these methods to the general principle of maximum likelihood estimation outlined in the previous chapter. It is again easy to illustrate the method for the linear case before generalizing it to the non-linear case similar to the strategy followed for SVMs.

![Fig. 3.8 Illustration of support vector regression and the $\epsilon$-insensitive cost function.](image)

We have already mentioned in section 6.2 the $\epsilon$-insensitive error function which does not count deviations of data from the hypothesis that are less than $\epsilon$ form the hypothesis. This is illustrated in figure 3.8. The corresponding optimization problem is

$$
\min_{w,b} \frac{1}{2} ||w||^2 + C \sum \sigma (\xi_i + \xi_i^*),
$$

subject to the constraints

$$
y^{(i)} - w^T x - b \leq \xi_i \quad (3.44)
$$

$$
y^{(i)} - w^T x - b \geq \xi_i^* \quad (3.45)
$$

$$
\xi_i, \xi_i^* \geq 0 \quad (3.46)
$$

The dual formulations does again only depend on scalar products between the training examples, and the regression line can be also be expressed by a scalar product between the support vectors and the prediction vector,

$$
h(x; \alpha_i, \alpha_i^*) = \sum_{i=1}^{m} (\alpha_i - \alpha_i^*) x_i^T x. \quad (3.47)
$$

This, we can again use Kernels to generalize the method to non-linear cases.

\(^2\)Three points of different classes can not be separated by a single line, but these are singular points that are not effective in the definition of VC dimension.
3.4 Decision Trees and Random Forrest *

A quick word about Random Forrest (RF) as we have used this classifier at the beginning of the course without much explanation. A random forest is really a technique based on decision trees. A decision tree looks at the individual features and makes some decision based on them. For example, lets say we want to decide if a patient has a certain illness from some medical observations such as blood pressure, maybe some skin condition or the temperature of the patient. At each level the algorithm looks for the feature that bests predict the split of the data at this level. It often uses some information theoretic measure such as the information gain to determine which feature is is best.

Decision trees have thus some form of feature selection build into them. This can be quite useful. However, it is also clear that such a method takes different features only in a serial manner into account; it hence assumes that there is some form of conditional independence similar to the naive Bayes method. In practice it has often been observed that decision trees tend to overfit easily. This is where the Random Forrest idea comes into play. This is a ensemble learning method for decision trees which uses several decision trees that have some random variations. The final decision is a form of an average or majority vote.

Random Forrest classifiers have gained a good reputation for many applications as they are easy to apply and even lead to simple decision rules. As we have seen in the first chapter, such methods are competitive with other classifiers for simple problems. However, it is clear that they have limitations in representing more complicated multi-feature relations.

3.5 Feature Selection, Dimensionality reduction, t-SNE

Feature selection has been an important topic in machine learning for many years. This was specifically important when building models for limited data. That is, it has been a common strategy to reduce the number of features as much as possible as every feature would add complexity (and hence parameters) to the model. Choosing model inputs carefully was hence an important topic. While this has been largely negated with deep learning on large data, such strategies might still be useful with limited data sets. Furthermore, there is also often the need for a more compressed representation of to visualize results in low dimensions that can be comprehended by humans. Dimensionality reduction is hence an important topic in machine learning.

A traditional method still used frequently for dimensionality reduction is Principle Component Analysis (PCA). PCA attempts to find a new coordinate system of the feature representation which orders the dimensions according to how spread the data are along these dimensions as this dimension would then give most sensitivity in describing the data.

Such data reduction can be illustrated with the example shown in Fig. ???. The direction of the largest variance of the data in this figure is called the first principal component. The variance in the perpendicular direction, which is called the second principal component, is less. In higher dimensions, the next principal components are in further perpendicular directions with decreasing variance along the directions. If one
would be allowed to use only one quantity to describe the data, then one can choose values along the first principal component, since this would capture an important distinction between the individual data points. Of course, we lose some information about the data, and a better description of the data can be given by including values along the directions of higher-order principal components. Describing the data with all principal components is equivalent to a transformation of the coordinate system and thus equivalent to the original description of the data.

Fig. 3.9 Example of feature pairs \((x_1, x_2)\) values drawn from a two-dimensional probability distribution with mean zero. It also shows a new coordinate system with first principle component \(x'_1\) and the perpendicular direction along the second principle component \(x'_2\).

There are several extensions to PCA that are worth considering in the machine learning context. In particular Independent Component Analysis (ICA) seeks to find new coordinates that would minimize the statistical dependence between the new features. Let \(S\) be independent source signals, and consider that the features that we usually measure for a system are linear combinations of these source signals,  

\[ x = Ws \]  

(3.48)

where \(W\) is now a mixing matrix. We are hence looking for the inverse of this mixing matrix that can be solved by trying to find the component of the inverse matrix that minimize some measure of dependencies between the feature values \(x\). This is in itself a learning problem. A generalization of these ideas are captured in non-negative matrix factorization.

Finally, there have been many techniques to visualize high dimensional data in 2-dimensional spaces so they can be visualize graphically. One of the strongest techniques in this realm is currently a technique called t-distributed stochastic neighbor embedding (tSNE). This technique tries to minimize distance between the similarities of data points in the high dimensional feature space with the distance of their low dimensional representation. Such techniques are implemented in sklearn.