Language and Linguistics Compass 9/2 (2015): 55–76, 10.1111/lnc3.12123

A Gentle Introduction to Machine Learning for Natural Language Processing: How to Start in 16 Practical Steps

Barbora Hladka* and Martin Holub
Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics, Charles University in Prague

Abstract
We present a gentle introduction to machine learning in natural language processing. Our goal is to navigate readers through basic machine learning concepts and experimental techniques. As an illustrative example we practically address the task of word sense disambiguation using the R software system. We focus especially on students and junior researchers who are not trained in experimenting with machine learning yet and who want to start. To some extent, machine learning process is independent on both addressed task and software system used. Therefore readers who deal with tasks from different research areas or who prefer different software systems will gain useful knowledge as well.

1. *Introduction*

We compose our paper as a gentle introduction to machine learning (ML). We explain ML on its application in the field of natural language processing (NLP) and illustrate it with the task of assigning correct word senses called word sense disambiguation (WSD). Moreover, the paper addresses the WSD task practically in the R software system, which is an environment for data analysis and running ML methods.[1] Figure 1 shows that the data are common denominator of NLP, ML, and R: data in NLP are represented by written and/or spoken corpora; ML uses data for learning. Once we computerize real world objects (e.g., email messages, or words in sentences, or flowers in the garden), we form their *data representations*, which means that the real objects become data. Since ML deals with data regardless a field they come from, NLP in Figure 1 can be replaced with any other field that can profit from existing data.

We aim at readers who want to approach an NLP task using ML and who do not know how to start. To some extent, machine learning process is independent on both addressed task and software system used. Therefore readers who prefer systems different from R will gain useful knowledge as well. Our goal is to navigate readers through machine learning process and to illustrate it on the example of the WSD classification task. The reader will learn both theoretical foundations of three machine learning methods and basic practical steps to develop a simple classifier. This paper does not require any strong mathematical or programming background. However, we assume basic mathematics knowledge, basic experience with NLP data, elementary programming skills, and basic knowledge of the R language as well. We illustrate the practical steps directly in the form of R code. Additional supplementary material (henceforth SM) including experimental data sets and their description, demo R code with its output, and more detailed description of experiments with their results are posted at a web page attached to this paper (Hladká and Holub 2013).[2] We recommend having the SM opened while reading this tutorial to follow its content easily.

A considerable amount of literature has been published on machine learning, on natural language processing, and on the R system. We point out some introductory readings and serious

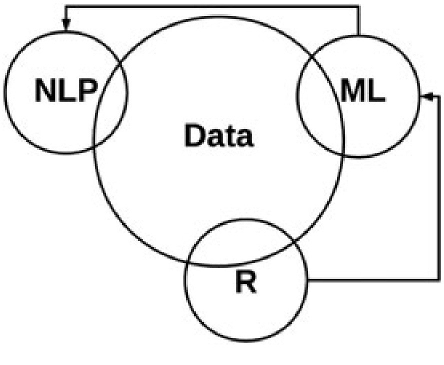

Fig. 1. Interconnection between data, natural language processing, machine learning, and R.

textbooks related to the topics of our paper: a non-technical view on ML (Domingos, 2012), an introduction to NLP (Prakash et al. 2011), an extensive practical tutorial on ML with R (Lantz 2013), a cartoon covering all the fundamental ideas of statistics (Gonick and Smith 1993), a practical introduction to statistics using R focused on NLP applications (Baayen 2008), an introduction to statistical learning including applications in R (James et al. 2013), a tutorial on Support Vector Machines (Burges 1998), a comprehensive introduction to the fields of pattern recognition and machine learning (Bishop 2006), and a book on ideas in learning explained in a statistical framework (Hastie et al. 2009). Our tutorial guides readers through the three things, NLP and ML and R, at once.

This paper is organized as follows. In Section 2, we introduce the WSD task in more detail. Then we focus on the ML process itself. Sections 3 and 6 contain practical demonstrations on the WSD task, accompanied by both short theoretical descriptions and commented sample code in R. A deeper view on the evaluation of classifier performance is given in Section 12. Last two sections briefly discuss slightly more advanced topics. Section 16 provides a view on feature engineering, which often belongs to the crucial parts of the ML process. Finally, in Section 17, we focus on formal foundations of ML.

2. *Example NLP task*

A task is called CLASSIFICATION if its goal is to classify input objects into a given set of discrete TARGET CLASSES. Then, a CLASSIFIER is an automatic procedure performing classification. In fact, a classifier works as a prediction function that takes a description of an input object and predicts a target class as its output. For example, a spam filter classifies e-mail messages into two classes `SPAM` and `NOT-SPAM`.

Natural language processing deals with human–machine communication in both written and spoken natural language. This communication can work if computers are able to recognize proper senses of words in sentences or utterances. People usually recognize senses easily because they understand the context in the sentence and use their knowledge of the world. If we want computers to master some problem using ML, we should prepare a *training set of examples* labeled with *true classifications* and guide computers to learn from them. When we provide computers with such training examples, we become in fact their supervisors. Therefore this way of learning is called *supervised*.

We address the task of WSD to illustrate ML in NLP practically. Namely, we work with the target word *line* and its six possible senses `CORD`, `DIVISION`, `FORMATION`, `PHONE`, `PRODUCT`, and `TEXT` selected out of all possible senses of *line*. These six senses are possible values of the target class `SENSE`. For illustration, *line* in the sentence *I've got Inspector Jackson on the line for you.* has the correct sense `PHONE`, `FORMATION` in *Outside, a line of customers waited to get in.*, `PRODUCT` in *The company has just launched a new line of small, low-priced computers.*, `DIVISION` in *Draw a line that passes through the points P and Q.*, `CORD` in *Important parameters of a fishing line are its length, material, and weight.*, and `TEXT` in *He quoted a few lines from Shakespeare.*

Both humans and computers need to know the context of the target word (*line*) to recognize correct sense. Computers need to extract a limited set of useful context clues that are subsequently used for automatic decision about the correct sense. Formally, the context clues are called *features*, and the developer should define them exactly and explicitly. Thus each object (a sentence) is characterized by an ordered list of features, which is called a *feature vector*. The set of all feature vectors forms a *feature space*. There are two types of features, *numerical* and *categorical*. Numerical features have numerical values (either discrete or continuous), while categorical features have discrete, non-numerical values. A special kind of categorical features are *binary* features that have only two possible values `TRUE/FALSE`. A developer designs a set of features based on his/her intuition about their usefulness. An example of a possible feature is the occurrence of typical verbs preceding *line*, which could be useful for recognizing the sense `PHONE`. Once features are specified, the example objects are analyzed, feature values are extracted from them, and feature vectors are created (see Figure 2). A feature vector together with the corresponding target class value form a data *instance*. In the WSD task, we use 20 categorical features including 11 binary ones (see SM, file wsd.pdf).

In this paper we work with an experimental data set, which contains 3524 sentences that were manually classified (Leacock et al. 1993). We analyzed this data by a chain of fundamental NLP tools to extract feature vectors (Toutanova and Manning 2000), (de Marneffe et al. 2006), namely a tokenizer, part-of-speech tagger and syntactic parser. The result is stored in the wsd. development.csv file as 3524 classified data instances.

3. *Preparing ML experiment in R*

Figure 3 visualizes a story of teaching computers to classify objects. The story starts with task formulation and results in a classifier. The action in between is called *machine learning process*, and it consists of three main procedures, namely getting data, building classifier, and final evaluation. *Getting data* requires a sequence of activities, mainly gathering data, assigning true classification, cleaning and preprocessing data. *Building classifier* is the central and most complex procedure. First, the developer should choose an appropriate ML method that enters the *development cycle*. Then the selected method is practically implemented, i.e., a set of features is designed and extracted from data, a classifier is trained, tuned, and evaluated typically in a

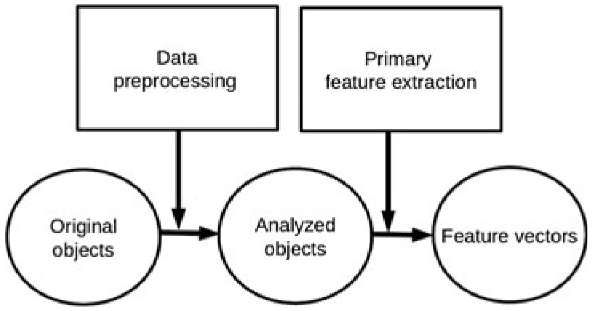

Fig. 2. Data preprocessing precedes feature extraction.

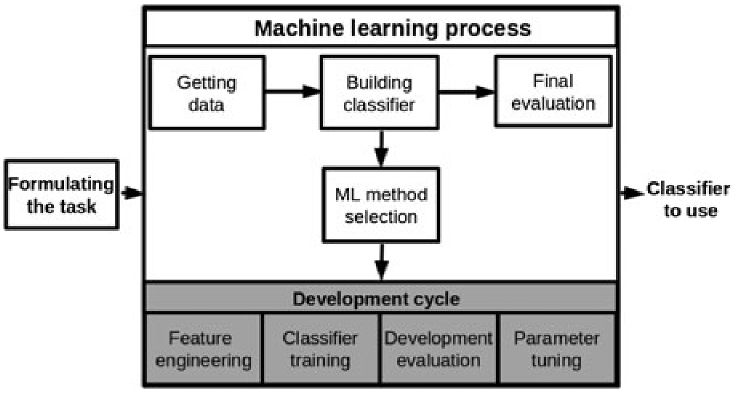

Fig. 3. Machine learning process and development cycle.

number of iterations. Iterations in the development cycle test developer's expectations and answer their questions like what does work and what does not, what features to exclude, what learning parameters set to what values. Finally, the best classifier's performance undertakes *final evaluation* on the test data set. In the next sections, we present sixteen practical steps (see *Steps 1–16*) that describe these procedures in details and address them using R scripts practically.

3.1. FORMULATING THE WSD TASK

First necessary step towards the machine learning process is an exact definition of the task. We need to formally describe both the nature of objects to be classified and the set of target class values that will be used for classification.

Step 1: *Formal definition of the task*

Task description – Assign the correct sense to the target word *line*.
Objects specification – Natural language sentences containing the given target word (*line* or some of its (derived) forms *lines* or *lined*).
Target class specification – The target class is a categorical variable and has six possible values. SENSE ∈ {`CORD`, `DIVISION`, `FORMATION`, `PHONE`, `PRODUCT`, `TEXT`}.

□

Another important and necessary task related to objects specification is to define an exact list of object's features that will be used in feature vectors. In this paper we use 20 features described in wsd.pdf (see SM). For example, the binary feature A9 has `TRUE` value if the target word *line* is followed by the preposition *between*, otherwise it has `FALSE` value.

3.2. DEALING WITH DATA IN R

The example data set is available in the wsd.development.csv file. It is a simple comma-separated values (CSV) file, so it can be browsed or even edited using any text editor. Also, the file can be loaded into a spreadsheet. Here we show how to read it using R.

Step 2: *Loading data into R*

To read the whole data file we use the read.table function. The reading process results in a data frame structure stored in the examples object.

```
> examples <- read.table("wsd.development.csv", header=T, colClass="factor")
> str(examples)
'data.frame':        3524 obs. of  21 variables:
 $ SENSE: Factor w/ 6 levels "cord","division",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ A1   : Factor w/ 2 levels "0","1": 2 1 1 1 1 1 1 2 2 1 ...
 . . .
 $ A19  : Factor w/ 3 levels "line","lined",..: 3 1 3 3 1 1 3 3 1 1 ...
 $ A20  : Factor w/ 80 levels "advcl","agent",..: 12 6 2 12 12 18 12 73 18 12 ...
```

The new examples object contains 21 factors of the same length (3524 observations, i.e., instances), which are stored as columns in the data frame. Each row represents one data instance as row feature vector.

□

Before we start the learning process, at least elementary data analysis should be done. The following example shows how to estimate the probability of different senses by calculating their *relative frequency* (also known as *empirical probability*). More examples are given in SM.

Step 3: *Basic data analysis*

```
## probability of the target class values calculated as relative frequencies (in %)
> round(100 * table(examples$SENSE)/nrow(examples), 2)
     cord  division formation     phone   product      text
     9.53      9.14      8.40     10.78     52.16      9.99
```

□

Now the developer should randomly split his development data into two disjoint sets. One part will be used for training, and the other portion will be used for development tests. The correct way of working with all available data is discussed in more detail in Section 13.

Step 4: *Creating development working set and development test set*

```
# first, the size of both training and test portion should be decided
> num.examples <- nrow(examples)          # total number of input examples
> num.train <- round(0.9 * num.examples)  # number of training examples = 90% of all
> num.test  <- num.examples - num.train   # number of test examples = 10% of all

# to generate a random split of all examples we randomly permutate the index set
# to be able to reconstruct exactly the same experiment later we use set.seed()
> set.seed(123); s <- sample(num.examples)
> train <- examples[s[1:num.train], ]     # randomly selected training examples
> test  <- examples[s[(num.train+1):num.examples], ] # the rest makes the test set
```

□

4. *Running ML methods in R*

In general, no learning algorithm dominates all others on all problems. We illustrate ML in NLP using three out of all known ML algorithms, namely decision tree algorithm (DT, (Breiman 1984)), Naïve Bayes algorithm (NB, e.g., (James et al. 2013)), and Support Vector Machines algorithm (SVM, e.g., (Cristianini and Shawe-Taylor 2000)). Our intention is to present methods that are based on a different mathematical framework. DT is based on information theory, NB on probability, and SVM on linear algebra. It is appropriate to start with DT since it is simple both to understand and to interpret. On the other hand, NB deals with tables of probabilities that are not so easy to interpret. A classifier can be viewed as a procedure that partitions feature space into decision regions using a decision boundary. The SVM algorithm represents an ML model that clearly shows searching for a linear decision boundary.

4.1. DECISION TREES

Figure 4 shows an example of a decision tree structure. To classify an instance using a decision tree, we start at the root node and go through internal nodes to a terminal node associated with a target class value. We make decisions defined as conditions on the instance feature values on the path from the root to a terminal node. These conditions are put on values of the best feature selected using a

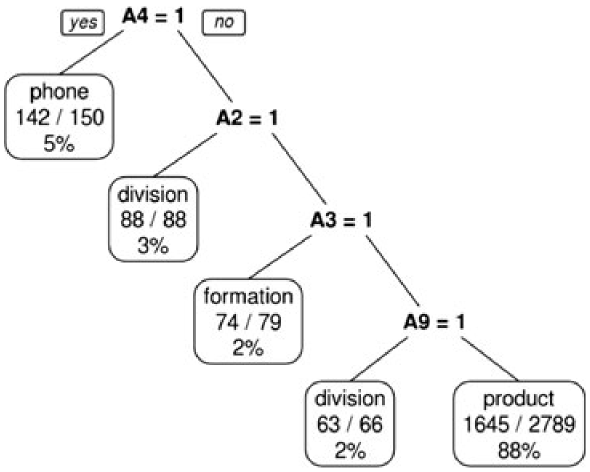

Fig. 4. Decision tree model M1. In each terminal node there are the *classification rate* (the number of correct classifications/the number of instances in the node) and the total percentage of training instances.

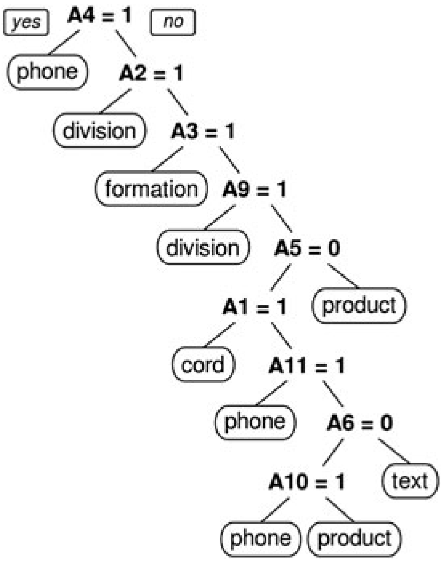

Fig. 5. Decision tree model M2. It improves the classification of M1 in the terminal node with the lowest classification rate (PRODUCT).

specific criterion. For example, one of the widely used measure is *information gain* based on the concept of conditional entropy anchored in information theory (e.g., Mitchell 1997, Lantz 2013).

For illustration, the decision tree in Figure 4 classifies the sense of *line* in sentence '*Draw a line between the points P and Q.*' correctly since the first three conditions on features `A4`, `A2`, `A3` are evaluated as negative and the fourth condition on feature A9 is fulfilled. Therefore the result is `DIVISION`.

Step 5: *Decision tree learning*

To learn a basic decision tree model we use the rpart package and the development working data. Since using categorical features with many different values is computationally too demanding, only the binary features will be used.

```
> library(rpart)                 # to load the package ‘rpart’ into the memory
# building a simple decision tree model M1 using all BINARY features
> M1 <- rpart(SENSE ~ A1+A2+A3+A4+A5+A6+A7+A8+A9+A10+A11,
              data=train, method="class")
```

A detailed description of the created decision tree can be obtained using the summary(`M1`) function. Also, a nice simple graphical representation of the tree can be easily obtained using package `rpart.plot`. The function `rpart.plot()` provides a lot of parameters to control the format of the output picture. For details see `help(rpart.plot)`.

```
> install.packages(rpart.plot)  # rpart.plot is not included in default instalation
> library(rpart.plot)           # to load the package into the memory
> rpart.plot(M1, tweak=1.5, extra=102, varlen=0)   # to make a picture; see Fig. 4
```

Now, the developed decision tree can be used to classify the test instances. The result will be stored in the prediction. M1 object, a factor with 352 values, which is the number of the test instances. The table at the end of the example code shows the distribution of the predicted classes, e.g., 13 of the test instances were classified as `PHONE`.

```
> prediction.M1 <- predict(M1, test, type="class")
> length(prediction.M1)
[1] 352
> summary(prediction.M1)
     cord  division formation     phone   product      text
        0        16         7        13       316         0
```

□

4.2. BASIC CLASSIFIER EVALUATION

The first thing we necessarily need to observe progress during the development is a *baseline classifier*. Later, its performance will be compared with performance of more sophisticated models. The most trivial baseline classifier is the classifier that always outputs the most frequent class, often called an *MFC classifier*. Our developed classifiers should *never* be worse than the MFC baseline. Sometimes, another simple classifier is considered to be a baseline, e.g., a classifier with default learning parameters settings.

Step 6: *Baseline classifier* (*MFC*)

To compute the MFC baseline we need to find the most frequent class and its relative frequency. In our development data set the most frequent target class value is product (see the previous step); the result below shows that there are 52.16% examples labeled with it.

```
> round(100 * max(table(examples$SENSE))/num.examples, 2)    # MFC baseline accuracy
[1] 52.16
```

□

First step towards a more detailed evaluation is usually observing and analyzing *confusion matrix*, which is a square matrix indexed by all possible target class values. Columns represent the instances in predicted classes, while each row represents the instances in an actual class. An example is given in Step 7 where number 13 in the second row of the output table indicates that 13 instances with a true label `DIVISION` were incorrectly classified as `PRODUCT`. Thus, the matrix makes it easy to count how often the classifier is confusing (i.e., misclassifying) two different classes. The numbers of correctly predicted examples for each target class value are displayed on the diagonal.

Step 7: *Confusion matrix*

In R we produce confusion matrix by comparing two factors: a factor with true classes (`test$SENSE`) and a factor that contains predicted values, i.e., the result of classification (`prediction.M1`).

```
>  table(test$SENSE, prediction.M1)
           prediction.M1
            cord division formation phone product text
  cord         0        0         0     0      33    0
  division     0       15         0     0      13    0
  formation    0        0         6     0      22    0
  phone        0        0         0    12      21    0
  product      0        1         0     0     191    0
  text         0        0         1     1      36    0
```

□

Of course, we need to quantify the classifier performance by a single number. The most basic performance measure, called *sample accuracy* is the number of correctly predicted examples divided by the number of all examples in the predicted set. A similar measure is *error rate*, also called *sample error*, which is defined as

$$\text{error rate} = 1 - \text{accuracy}.$$

A good classifier should be able to *generalize* from training data, which means that it should have good *predictive performance* even on *unseen data instances*. In contrast to sample error, *generalization error* of a classifier is the probability that a randomly chosen instance (possibly in future) will be misclassified. The goal of ML process is in fact to build a classifier with low generalization error. To estimate generalization error we use a test data set that should *not* be used during training at all. If test data is used as a sample for measuring accuracy or error, then the measures are called *test accuracy* or *test error*, respectively.

Step 8: *Computing accuracy and error rate on the test set*

```
> round(100 * sum(prediction.M1 == test$SENSE)/num.test, 2)    # test accuracy in %
[1] 63.64
> round(100 * sum(prediction.M1 != test$SENSE)/num.test, 2)    # test error in %
[1] 36.36
```

□

During evaluation we should observe and analyze also *training error*, which is the sample error measured on the training data. A low training error indicates that the developed model *fits* the training data well. However, if there is a big difference between the test error (which is usually worse) and the training error, it can happen that classifier's generalization is poor because it *overfits* the training data. Hence, we should compare the sample error on the test set and on the training set.

Step 9: *Comparing the test error and the training error*

```
> prediction.M1.train <- predict(M1, train, type="class")
> round(100 * sum(prediction.M1.train != train$SENSE)/num.train, 2)
[1] 36.57                 # training error is almost the same as the test error
```

The results obtained in Steps 8 and 9 show that the model is not *overtrained*, i.e., it does not overfit. □

4.3. MODEL TUNING

When an ML method has already been selected, the purpose of the ML process is to search for the best values of the prediction function parameters, which are generally called *hypothesis parameters*. To optimize a classifier means to optimize its hypothesis parameters. In the case of decision trees, for example, conditions in decision tree nodes are hypothesis parameters. In fact, the search for the optimal hypothesis parameters is often too computationally demanding. To reduce the computational complexity of the search, ML algorithms sometimes approximate the optimum.

Also, learning methods usually have some *learning parameters*, which are parameters of the learning process itself. Learning parameters should be set by the developer and typically need tuning. To find the best classifier the learning process requires optimal values of learning parameters. Searching for them is done by optimization techniques, which is a complex field and cannot be explained in this paper.

The `rpart()` function for building decision trees provides a number of learning parameters among which the *complexity parameter* (`cp`) is the most important one. The complexity parameter controls the depth of the learned tree. Obviously, to get a good classifier the depth should not be too small. On the other hand, when the learned tree is too deep, it can easily overfit training data. Therefore the cp value should be carefully tuned. For more details including the default values see help(`rpart.control`).

Step 10: *Improving classifier's performance by setting a better learning parameter value*

The default value of the complexity parameter is `cp=0.01`. In this experiment a lower value will be set, which will lead to a more complex decision tree model.

```
# building a decision tree model with a lower value of complexity parameter
M2 <- rpart(SENSE ~ A1+A2+A3+A4+A5+A6+A7+A8+A9+A10+A11,
            data=train, method="class", cp=0.001)
```

Resulting decision tree displayed in Figure 5 tries to be more sophisticated and is more complex than the previous one displayed in Figure 4. However, we should check the new model and see whether its performance improves or not.

```
> prediction.M2 <- predict(M2, test, type="class")
> round(100 * sum(prediction.M2 == test$SENSE)/num.test, 2)
[1] 65.34          # M2's test accuracy is better than M1's test accuracy = 63.64%
```

The improved test accuracy (65.34%) shows that the new cp setting (`cp=0.001`) improves the classifier. One can also measure the training accuracy, which is 66.2%. So, in comparison with `M1`, the gap between the training and the test accuracy increased. It is quite a typical case that a more complex model leads to both lower training error and a bit worse generalization. However, the test error (34.66 % = 1 − 65.34 %), which estimates the generalization error, decreased; and therefore `M2` should be considered better than `M1`.

□

4.4. NAÏVE BAYES CLASSIFIER

Probability theory provides a framework for the quantification and manipulation of uncertainty. We can quantify uncertainty when predicting an output value of an instance **x** using conditional probability. For each target class value *y*, we calculate the conditional probability of this value given an instance **x**, Pr (y | **x**). Then a target class value with the highest conditional probability is selected as an output. In our WSD task, to assign the most probable target class value to an instance ***x***, we compute the six conditional probabilities Pr(`CORD`|**x**), Pr(`DIVISION`|**x**), …, Pr(`TEXT`|**x**) and select the class with the highest conditional probability. The Naïve Bayes classifier computes Pr (y | **x**) using probabilities Pr (**x** | y) and Pr (y) based on the Bayes rule and on a naïve assumption that features are independent of each other. Although this assumption is rarely true in real world applications, Naïve Bayes classifier surprisingly often shows good performance.

Step 11: *Building and evaluating a Naïve Bayes classifier*

```
# to create a Naïve Bayes model using binary features
> library(e1071)    # package 'e1071' is needed
> M3 <- naïveBayes(SENSE ~ A1+A2+A3+A4+A5+A6+A7+A8+A9+A10+A11, data=train)

# prediction and evaluation on the test data set
> prediction.M3 <- predict(M3, test[2:12], type="class")
> round(100 * sum(prediction.M3 == test$SENSE)/num.test, 2)
[1] 65.62          # accuracy of M3 in percentage
```

Although Naïve Bayes model can work with all available features, it does not reach better performance in comparison with `M3`.

```
# building and evaluating Naïve Bayes model M3b using ALL features
> M3b <- naïveBayes(SENSE ~ ., data=train)   # '.' stands for all available features
> prediction.M3b <- predict(M3b, test[2:12], type="class")
> round(100 * sum(prediction.M3b == test$SENSE)/num.test, 2)
[1] 65.62                                    # M3b does not outperform M3
```

□

4.5. SUPPORT VECTOR MACHINES

Support Vector Machines represents a 'geometric' approach to ML since it searches for the optimal *hyperplane* in the feature space, a linear decision boundary that partitions instances into two output classes. Consequently, SVM primarily does binary classification. If there are only two features, a hyperplane is a line – see the thick black line in Figure 6. Having a separating hyperplane, we classify a new instance by its position. Instances on the left-hand side of the hyperplane are classified as black ones, while instances on the right-hand side are classified as white ones. Mathematically, a hyperplane takes a form of a linear function $f(\mathbf{x}) = \mathbf{wx} + b$, where parameters $\mathbf{w}$ and b represent hypothesis parameters of SVM classifier. A multiclass task can be reduced into several binary *one-versus-all* or *one-to-one* tasks.

When searching for an optimal hyperplane, we first suppose that training data is *linearly separable*, as illustrated in Figure 6. There may be infinitely many separating hyperplanes among which the SVM algorithm selects the hyperplane with maximum distance from the closest instances of different classes. The instances closest to the selected hyperplane are referred to as *support vectors*, and the maximized distance between them is called *margin*. If training data set is *not* linearly separable, see Figure 7, the following more advanced techniques are often used: (i) We allow a few examples on the wrong side of the separating hyperplane and penalize them for misclassification. Then we control the balance between margin and misclassification. (ii) To turn the data into a linearly separable case, we use *kernel functions* that map the training feature vectors into a new space of a higher dimension where the non-linear properties can appear linear.

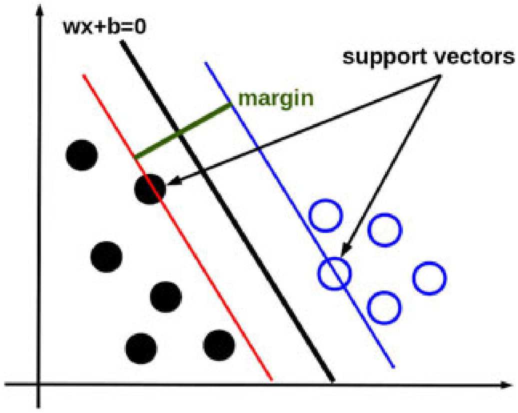

Fig. 6. Linearly separable data.

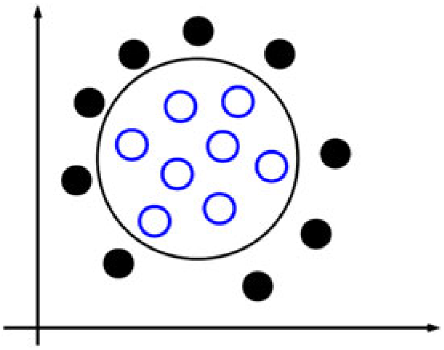

Fig. 7. Non-linearly separable data.

Step 12: *Building an SVM model*

```
# to create an SVM model using all binary features
> library(e1071)          # package ‘e1071’ is needed
> M4 <- svm(SENSE ~ A1+A2+A3+A4+A5+A6+A7+A8+A9+A10+A11, data=train, kernel=’linear’)

# prediction and evaluation on the test data set
> prediction.M4 <- predict(M4, test, type="class")
> round(100 * sum(prediction.M4 == test$SENSE)/num.test, 2)
[1] 65.91
```

As the following code shows, the SVM classifier dramatically improves upon the Naïve Bayes model: The test accuracy increases to 78.69% when we use all 20 features.

```
> M4b <- svm(SENSE ~ ., data=train, kernel=’linear’)
> prediction.M4b <- predict(M4b, test, type="class")
> round(100 * sum(prediction.M4b == test$SENSE)/num.test, 2)
[1] 78.69
```

□

Although M4b is much better than all previous models (it has significantly better accuracy on the development test set), its generalization is relatively weak. As shown below, the training error of M4b is less than 1%.

```
> prediction.M4b.train <- predict(M4b, train, type="class")
> round(100 * sum(prediction.M4b.train == train$SENSE)/num.train, 2)
[1] 99.05           # training accuracy is better than 99%
```

□

5. *Working with data: training and evaluation*

When a classifier has been developed, we always need to do thorough evaluation to get a right picture about the quality of the classifier. Especially we need to

(1) get a reliable estimate of the classifier’s performance on new (so far unseen) data instances and
(2) compare a number of competing classifiers that have been developed to decide which one is the best.

This kind of evaluation process is often called *model assessment and selection*. Reliable model assessment requires proper working with the development data set. The fundamental reason is that the ML process should result in a classifier with good performance not only on the data available during the development but also on any data that can be expected in future.

5.1. DEVELOPMENT DATA AND TEST DATA

As illustrated in Figure 8, all available data examples are typically divided into two basic parts. While *development data* is the portion that the developer works with, the other part, *unseen test data* is the portion that the developer will never see. At the beginning of the development process the developer should split the available development set into two parts. The purpose of the *development test set* is in fact to simulate the real (i.e., unseen) test data and should be used only for the final development evaluation to get an estimate of the classifier’s performance. It happens only at the end of the development when the classifier has already been tuned and all learning parameters are finally set. Development test set is also used for model selection.

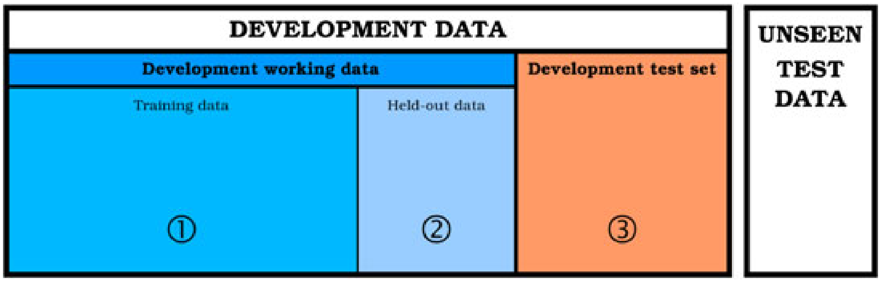

Fig. 8. Working with data during the development process.

Development working data set is used both for training and for evaluation when we tune the learning parameters. Therefore development working data set is split in training data, used for training with a particular learning parameter setting, and *held-out data* used for evaluation with the particular learning parameter settings.

Finally, after model selection, when the whole development has been finished, all available development data are usually used for training the best model that we are able to develop. This final model can be later evaluated on the unseen test data. However, the test on unseen test set is not a part of the development process.

Step 13: *Evaluation on unseen test data*

Evaluation of a finished classifier using unseen test data is a means of estimating the generalization error and should not be a part of the development process. It is important that the development test set should *not* be used for this kind of evaluation because this development test set has already been used for the model parameter tuning. The following code shows the sample accuracy of three selected classifiers measured on the unseen test data. This time the three models are trained using all development data.

```
## Loading the wsd.test data set into R
> unseen <- read.table("wsd.test.csv", header=T, colClass="factor")
> num.unseen <- nrow(unseen)

### final Decision Tree model M2
> M2.final <- rpart(SENSE ~ A1+A2+A3+A4+A5+A6+A7+A8+A9+A10+A11,
                    data=examples, method="class", cp=0.001)
> prediction.M2.final <- predict(M2.final, unseen, type="class")
> round(100 * sum(prediction.M2.final == unseen$SENSE)/num.unseen, 2)
[1] 64.54

### final Naive Bayes model M3
> M3.final <- naiveBayes(SENSE ~ A1+A2+A3+A4+A5+A6+A7+A8+A9+A10+A11, data=examples)
> prediction.M3.final <- predict(M3.final, unseen, type="class")
> round(100 * sum(prediction.M3.final == unseen$SENSE)/num.unseen, 2)
[1] 64.74

### final SVM model M4b
> M4b.final <- svm(SENSE ~ ., data=examples, kernel='linear')
> prediction.M4b.final <- predict(M4b.final, unseen, type="class")
> round(100 * sum(prediction.M4b.final == unseen$SENSE)/num.unseen, 2)
[1] 79.12
```

□

Table 1 gives an overview of the performance of four selected classifiers. Last two columns show the average cross-validation accuracy and confidence intervals (CI), which will be explained in the next section.

5.2. CROSS-VALIDATION AND CONFIDENCE INTERVALS

Using only one small test set during the whole evaluation process is risky because we can get a good or bad result only by chance. Generally, the more test data, the more confident evaluation. On the other hand, we also need training data set as large as possible. However, it is quite typical that we have only a small sample of examples to train a classifier. To overcome the difficulty with getting a sufficient amount of data one can apply techniques like *cross-validation* or other kinds of bootstrapping methods that generate a number of data samples using the available data set. Cross-validation is a simple and widely used technique that boosts evaluation confidence without the need of collecting more example data.

The idea of *k*-fold cross-validation is that development working data is partitioned into *k* subsets of equal size and then we do training and test in *k* iterations. In the *i*th step of the iteration, the *i*th subset is used as a test set, while the remaining parts form the training set. An example of sixfold cross-validation process is depicted in Figure 9.

Step 14: *Running cross-validation experiment*

```
> source("Do-CV-test.R")          # running code stored in a file (available in SM)
Decision Tree model M1  --  cross-validation accuracies:
 [1] 0.664 0.648 0.596 0.634 0.669 0.603 0.659 0.659 0.634 0.590
Decision Tree model M2  --  cross-validation accuracies:
 [1] 0.613 0.664 0.650 0.675 0.675 0.637 0.659 0.644 0.703 0.672
```

□

Using the code in `Do-CV-test.R` we can run a 10-fold cross-validation test, which results in 10 accuracy values. Now the question is what can we conclude from the observation of 10 different accuracy values? Or, in other words, what is the 'real' classifier accuracy? Obviously, we can compute the average accuracy, which is more robust than a simple test accuracy measured on a fixed test set. However, to quantify our uncertainty we need to compute a *confidence interval* for the mean of the classifier accuracy. When we measure classifier accuracy or error rate on random data samples, the population of resulting accuracy values has approximately normal distribution. Confidence intervals for the mean of a normally distributed population are based on a special statistical procedure called *one sample t-test* and are calculated

Table 1. Overview of selected classifiers' performance.

		Development accuracy		Test accuracy	Cross-validation	
Model		**Training**	**Test**	**Unseen**	**Avg**	**CI**
DT	M1	63.4%	63.6%	63.5%	63.6%	[0.614, 0.657]
DT	M2	66.2%	65.3%	64.5%	65.9%	[0.641, 0.677]
NB	M3	65.6%	65.6%	64.7%	65.8%	[0.639, 0.671]
SVM	M4b	99.1%	78.7%	79.1%	80.4%	[0.787, 0.822]

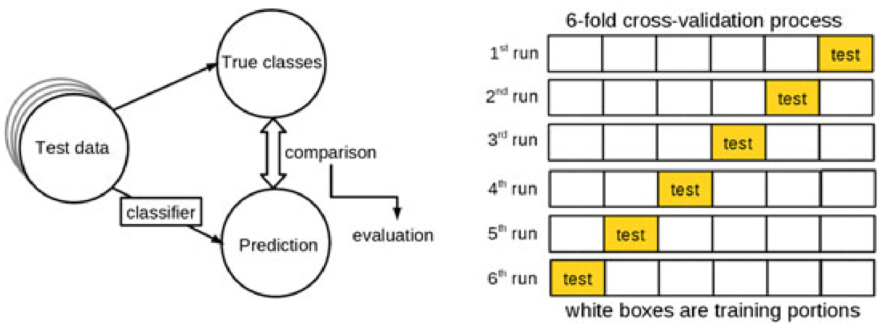

Fig. 9. Scheme of the cross-validation process.

from a given set of observed data. A confidence interval gives an estimated range which is likely to include the (always unknown) population mean.

In R we can run the `t.test()` function that takes a vector of observed accuracy values and the *confidence level* parameter. Confidence level can be understood as the probability that the confidence interval includes the true population mean. In other words, it is the probability that one cannot make a mistake rejecting a hypothesis about the population mean. The higher the confidence level, the wider the confidence interval.

Step 15: *Computing confidence intervals*

```
> Acc.M1 <- c(0.664, 0.648, 0.596, 0.634, 0.669, 0.603, 0.659, 0.659, 0.634, 0.590)
> t.test(Acc.M1, conf.level = 0.95)
 . . .
95 percent confidence interval:
 0.6144241 0.6567759

> Acc.M2 <- c(0.613, 0.664, 0.650, 0.675, 0.675, 0.637, 0.659, 0.644, 0.703, 0.672)
> t.test(Acc.M2, conf.level = 0.95)
 . . .
95 percent confidence interval:
 0.6414395 0.6769605
```

□

One should be careful and should not misinterpret the meaning of confidence intervals. Practically, if a hypothesized mean is *outside* the confidence interval, we *reject* the hypothesis. Otherwise, we *cannot reject* the hypothesis. For example, we can hypothesize that the `M1`'s accuracy mean is only 60%. Since this value falls outside the confidence interval [0.614, 0.657], we can reject this hypothesis. On the other hand, if we hypothesize 65%, it cannot be rejected.

Confidence intervals can also be used to compare two different classifiers, e.g., classifiers `M1` and `M2`. The relation between the two confidence intervals is illustrated in Figure 10. Since the

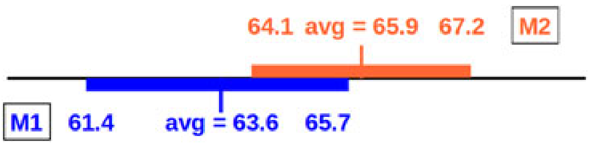

Fig. 10. Relation between two confidence intervals.

confidence intervals have non-empty intersection, there are numbers, 65%, that can be successfully hypothesized as the same accuracy mean for both classifiers, and thus we cannot reject the hypothesis that both classifiers have the same accuracy.

5.3. BINARY CLASSIFICATION AND ITS EVALUATION

Binary classification (also known as 2-class classification, or sometimes as 0/1 classification) is a classification task with just two different target class values. In such tasks training examples are very often regarded as divided into two disjoint subsets: 'positive examples' (those that should be retrieved, 'ones') and 'negative examples' (those that should not be retrieved, 'zeros'). Then the confusion matrix looks as shown in Table 2. The most commonly used performance measures for binary classification are defined in Table 3. If we want to work with a single measure combining precision and recall, we can use *F-measure* (also known as *balanced F-score*), defined as follows.

$$F = 2\ \frac{Precision * Recall}{Precision + Recall}$$

6. *Feature engineering*

Feature engineering belongs to the most important processes during the development cycle (see Figure 3). Feature engineering generally includes various methods and/or processes related to the construction of feature vectors suitable for a given ML task. All these methods can be summarized under the terms *feature extraction* (which is used in two different senses) and *feature selection*. A general scheme of feature engineering is given in Figure 11, showing the relationship between feature extraction and feature selection. The very first step is *primary feature extraction*

Table 2. Confusion matrix for binary classification task.

		Predicted class	
		Positive	**Negative**
True class	Positive	True positive (TP)	False negative (FN)
	Negative	False positive (FP)	True negative (TN)

'True' instances are correctly classified, while 'False' are those incorrectly classified. 'Positives' are instances predicted as positive (correctly or incorrectly), while 'Negatives' are those predicted as negative (correctly or incorrectly).

Table 3. Performance measures for binary classification.

Measure	Formula
Precision	TP/(TP + FP)
Recall/sensitivity	TP/(TP + FN)
Specificity	TN/(TN + FP)
Accuracy	(TP + TN)/(TP + FP + TN + FN)

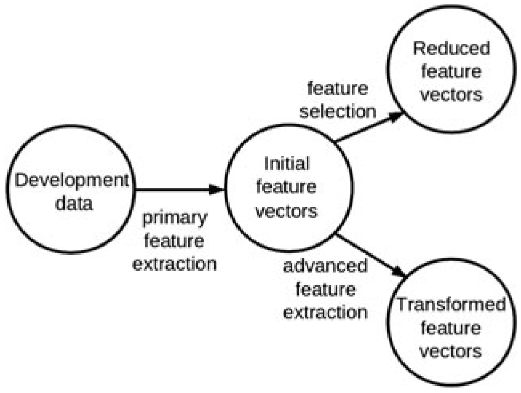
Development data
primary feature extraction
Initial feature vectors
feature selection
Reduced feature vectors
advanced feature extraction
Transformed feature vectors

Fig. 11. Basic scheme of feature engineering.

(see also Figure 2). The purpose of primary feature extraction is to exploit learning examples and collect all pieces of information that could be useful for prediction. Resulting sets of feature values form initial feature vectors. The choice of initial features highly depends on both developer's intuition and understanding of the task, and on his/her experience as well. Experiments are often necessary to do primary feature extraction well. Usually the developer should test his/her expectations during the development cycle and should find out what actually works and what does not.

Initial feature vectors are typically not optimal since they may contain many redundant or irrelevant features. Hence their further processing is needed. Feature selection does not change existing features, and instead it aims at selecting a subset of the most relevant features in order to both reduce the dimension (i.e., the length) of the feature vectors and remove unuseful 'noisy' initial features. Whereas the purpose of *advanced feature extraction* is to create a set of completely new features from the initial feature set by transforming the initial feature vectors. For an overview of basic approaches to advanced feature extraction refer to, e.g., (Kononenko and Kukar 2007), Chap. 7.7. Of course, automatic methods for feature selection and for advanced feature extraction can even be combined. In the rest of this section we concentrate on basic approaches to feature selection.

Feature selection is necessary especially when we work with a lot of initial features, which typically happens when one does, e.g., a text categorization task, see (Scott and Matwin 1999). A brief introductory survey of feature selection techniques can be found in Guyon and Elisseeff (2003). Main reasons why feature selection is needed are

- lower model complexity and improved model interpretability – a feature set which is more compact can help to a better understanding of the underlying process that generated the data;
- lower computational complexity – reducing the number of features can rapidly shorten training times;
- improved prediction performance – lower dimension of feature vectors can enhance the classifier's generalization power; hence it can reduce the risk of overfitting; also, some learning methods do not work well if the feature set contains highly dependent features (e.g., Naïve Bayes learner or even SVM model).

Two simplest approaches to feature subset selection are *filters* and *wrappers*. Filters select feature subsets as a preprocessing step, independently of the learning method; for details see, e.g., (Duch 2004). A brief description of a number of simple heuristic filter methods particularly useful for classification tasks can be found in Kononenko and Kukar (2007), Chap. 6.1.

Wrappers use an ML algorithm in conjunction with internal cross-validation to score feature subsets by measuring their predictive power. Moreover, there is another kind of feature selection algorithms called *embedded methods* that perform feature selection during the process of training; for more detailed explanation and examples see, e.g., (Lal et al. 2004). In the following practical step we demonstrate feature selection on a simple wrapper using the SVM learner.

Step 16: *Demonstration of a simple feature selection procedure using FSelector package*

The following code uses a greedy forward feature selection method implemented by the function `forward.search()` available in FSelector package. For details refer to the integrated help, e.g., `?FSelector`, `?forward.search`. The function `eval.svm()` provides a criterion for feature subset optimization. The greedy algorithm will search for a feature subset with the best result of `eval.svm()`.

```
> install.packages("FSelector")
> library(FSelector)

> eval.svm <- function(feature.set) {
      formula <- as.simple.formula(feature.set, "SENSE")
      model <- svm(formula, data=train, kernel='linear', cross=10)
      return(model$tot.accuracy)
  }

> init.features <- names(train)[-1]            # to start with all initial features
> subset <- forward.search(init.features, eval.svm)        # this will take a while

> print(formula <- as.simple.formula(subset, "SENSE"))        # to print the result
SENSE ~ A2 + A3 + A10 + A12 + A13 + A14 + A15 + A16 + A17 + A18

> print(avg.accuracy <- eval.svm(subset))          # to print the average accuracy
[1] 80.54855

# to check the performance of the classifier that uses the full feature set
> model.all <- svm(SENSE ~ ., data=train, kernel='linear', cross=10)
> model.all$tot.accuracy
[1] 80.45397
```

This example shows that the resulting feature subset is significantly more compact than the original set of all initial features (10 selected features in contrast to 20 original ones), yet the performance of the classifier with the reduced feature set (80.55%) measured by 10-fold cross-validation is more or less the same as the performance of the classifier that uses the full set of initial features (80.45%).

□

7. *Formal foundations of ML*

So far, we have presented basic concepts of ML from practical points of so far. Now, we enrich this practical experience with elementary formal foundations of ML related to the basic procedures of ML process as shown in Figure 3. For more details about ML formalisms see, e.g., (Mitchell 1997).

Before addressing a task, the developer has to describe the task exactly, specify its objects and a target class variable with *k*-possible values $C = \{y_1, y_2, \ldots, y_k\}$.

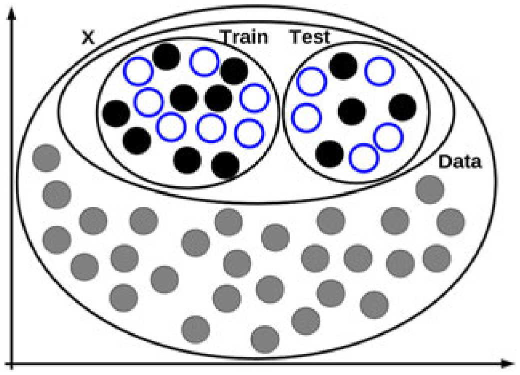

Fig. 12. Training and test data.

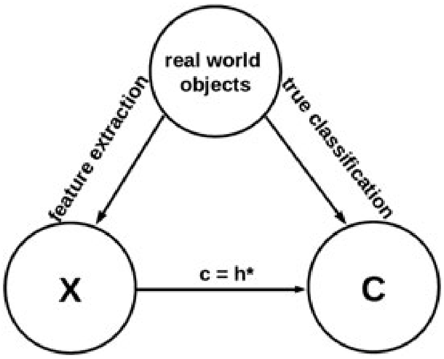

Fig. 13. Prediction function.

7.1. GETTING DATA

Once the objects are specified, the developer creates a set of features that characterize them. Using *m* features $A_1,\ldots,A_m$, a data instance is described as a feature vector $\mathbf{x} = \langle x_1, \ldots, x_m \rangle$, where $x_i \in A_i$. Feature vectors are elements in an *m*-dimensional space, and thus we deal with a set of instances $X = \{\mathbf{x} : \mathbf{x} = \langle x_1, \ldots, x_m \rangle, x_i \in A_i\}$. For supervised ML, objects with their true classification must be available. We need to select a sufficiently large and representative sample of objects from *X* and then assign true classification to each of them. Each selected example is classified into one of *k* classes $y_1, y_2, \ldots, y_k$. Then feature vectors of the selected examples with their true classifications form a set of pairs $Data = \{\langle \mathbf{x}, y \rangle : \mathbf{x} \in X, y \in C\}$. Finally, *Train*, a training set, and *Test*, a test set, are selected from *Data* so that $Train \cap Test = \emptyset$ and $Train \cup Test = Data$. To get representative samples, both the selection of *Data* and the split into *Train* and *Test* should be random. In Figure 12, the instances from *X* for which we do not know their true classification are filled with gray. A number of them are truly classified into either 'white' class or 'black' class to compose *Data*. We illustrate one possible split of *Data* into training and test set.

7.2. BUILDING CLASSIFIER

Formally we define a classifier as a prediction function $c(\mathbf{x}) = y$, $\mathbf{x} \in X$, $y \in C$, see Figure 13. At the beginning we do not know the prediction function so we need to approximate it using a hypothesis $h: X \rightarrow C$. Then, the *building classifier* procedure in Figure 3 searches the hypothesis space. The best hypothesis $h^\star$ is finally taken as c.

Classifier performance is influenced by both feature selection and learning parameters setting. These procedures are undertaken in the *development cycle*. We need to control the balance between the performance on training data and the ability to generalize beyond training data. *Overfitting* occurs when the test error increases while the training error steadily decreases, see Figure 14. For illustration, in Figure 15 we show three classifiers together with training examples. The classifiers are represented by the line h, parabola g, and curved line k, respectively. All of them separate the space of instances into two subspaces, and each instance is classified according to its position. The classifiers differ in the number of their hypothesis parameters: h is described by two parameters (a, b), g by three (c, d, e), and k would need many. As one can observe, the more hypothesis parameters, the lower training error. The classifier h misclassifies two training examples, g one, while k fits the training data perfectly. However, we can expect that k will generalize poorly. There are some techniques how to prevent overfitting, namely cross-validation, feature engineering, parameter tuning, and regularization.

An ML task can also be addressed with ensemble techniques, like bagging and boosting, that combine more classifiers. If classifiers are complementary, i.e., if they produce different types of errors then combining their outputs can increase the overall performance, e.g., by majority voting.

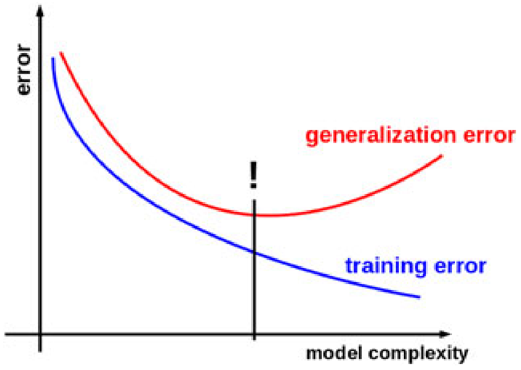

Fig. 14. Overfitting.

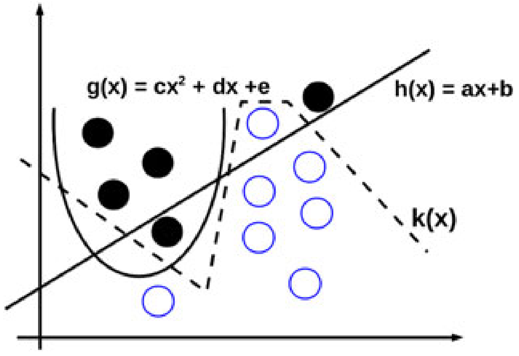

Fig. 15. Overfitting in 2-d feature space.

7.3. FINAL EVALUATION

Test data *Test*, unseen during the training, is now classified using $h^\star$, i.e., $\forall \langle \mathbf{x}, y \rangle \in Test$: Get $h^\star(\mathbf{x})$. To evaluate $h^\star$ on test data means to compare the true classification with the automatic classification by $h^\star$, i.e., $\forall \langle \mathbf{x}, y \rangle \in Test$: Compare y and $h^\star(\mathbf{x})$.

8. Conclusion

In our paper, we first helped readers to orientate in the field before they start with ML experiments. Then we presented an example NLP task (WSD) to illustrate the ML approach practically. The paper guided readers through 16 practical steps using the R system, and then it concluded the tutorial with an elementary ML formalism. In summary, we demonstrated the very basic concepts of supervised ML.

We are aware that providing a single NLP task to illustrate ML concepts cannot be sufficient enough to provide a broad understanding of the underlying ML problems. However, our tutorial is exclusively intended for the very beginners in the field, and our intention is to introduce them to the field step by step since the very beginning. Therefore, we have not investigated some other core topics related to the field, like concepts of overfitting, feature extraction and selection, and the curse of dimensionality. Also, we have not discussed theoretical aspects of ML at all. In our opinion, advanced parts of ML could be studied only when one is familiar with the basic ML concepts both theoretically and practically.

Acknowledgement

We gratefully acknowledge that this work was supported by the Grant Agency of the Czech Republic, grant project no. P103/12/G084. We would like to thank Jirka Hana for his English corrections. Also, we would like to thank the students who attended our course at ESSLLI 2013. Last but not least, we would like to thank the anonymous reviewers for their valuable comments.

Short Biographies

Barbora Hladka is a senior research associate at Charles University in Prague, Czech Republic. She is affiliated with the Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics. Her research focuses on issues from natural language processing, mainly corpus annotation, entity relationship extraction, and application of machine learning in various domains.

Martin Holub is a senior research associate at Charles University in Prague, Czech Republic. He got his PhD in software engineering in 2005 and is currently affiliated with the Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics. His research focuses on lexical semantics, on lexical disambiguation, and on machine learning applications, especially in Natural Language Processing.

Notes

* Correspondence address: Barbora Hladka, Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics, Charles University in Prague, Malostranske namesti 25, Prague 1, 11800 Czech Republic. E-mail: hladka@ufal.mff.cuni.cz

[1] http://www.r-project.org/

[2] http://ufal.mff.cuni.cz/Machine-Learning-for-NLP

Works Cited

Baayen, Rolf Harald. 2008. *Analyzing linguistic data: a practical introduction to statistics using R*. Cambridge: Cambridge University Press.

Bishop, Christopher M. 2006. *Pattern recognition and machine learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Breiman, L. 1984. Classification and regression trees. The Wadsworth and Brooks-Cole statistics-probability series. New York: Chapman & Hall.

Burges, Christopher J. C. 1998. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery* 2. 121–167.

Cristianini, Nello, John Shawe-Taylor. 2000. *An introduction to support vector machines: and other kernel-based learning methods*. New York, NY, USA: Cambridge University Press.

de Marneffe, Marie-Catherine, Bill MacCartney, Christopher D. Manning. 2006. Generating typed dependency parses from phrase structure parses. In *Proceedings of the Conference on Language Resources and Evaluation* (*LREC*), 449–454.

Domingos, Pedro. 2012. A few useful things to know about machine learning. *Communications of the ACM* 55. 78–87.

Duch, Wlodzislaw. 2004. Filter methods. In *Feature extraction, foundations and applications*, 89–118. Berlin Heidelberg New York: Physica Verlag, Springer.

Gonick, Larry, Woollcott Smith. 1993. *The cartoon guide to statistics*. New York: HarperPerennial, A Division of HarperCollins Publisher.

Guyon, Isabelle, André Elisseeff. 2003. An introduction to variable and feature selection. *Journal of Machine Learning Research* 3. 1157–1182.

Hastie, Trevor, Robert Tibshirani, Jerome Friedman. 2009. *The elements of statistical learning: data mining, inference and prediction*. 2 edition. New York: Springer.

Hladká, Barbora, Martin Holub. 2013. Supplementary data and other material for "A Gentle Introduction to Machine Learning for Natural Language Processing". http://ufal.mff.cuni.cz/Machine-Learning-for-NLP.

James, G., T. Hastie, D. Witten, R. Tibshirani. 2013. *An introduction to statistical learning: with applications in R*. Springer Texts in Statistics: Springer London, Limited.

Kononenko, Igor, Matjaz Kukar. 2007. *Machine learning and data mining*. West Sussex: Elsevier Science.

Lal, Thomas Navin, Olivier Chapelle, Jason Weston, André Elisseeff. 2004. Embedded methods. In *Feature extraction, foundations and applications*, 137–166. Berlin Heidelberg New York: Physica Verlag, Springer.

Lantz, Brett. 2013. *Machine learning with R*. Birmingham: PACKT Publishing.

Leacock, Claudia, Geoffrey Towell, Ellen Voorhees. 1993. Corpus-based statistical sense resolution. In *Proceedings of the ARPA Workshop on Human Language Technology*. 260–265.

Mitchell, Thomas M. 1997. *Machine learning*. 1 edition. New York, NY, USA: McGraw-Hill, Inc.

Prakash, M. Nadkarni, Lucila Ohno-Machado, Wendy W. Chapman. 2011. Natural language processing: an introduction. *JAMIA* 18. 544–551.

Scott, Sam, Stan Matwin. 1999. Feature engineering for text classification. In *Proceedings of the Sixteenth International Conference on machine learning*, ICML '99, 379–388. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Toutanova, Kristina, Christopher D. Manning. 2000. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proceedings of the 2000 Joint SIGDAT Conference on empirical methods in NLP*, 63–70, Stroudsburg, PA, USA: Association for Computational Linguistics.