Report on Practical Bayes-True Data Generators For Evaluation of Machine Learning, Pattern Recognition and Data Mining Methods

Aleksander Lodwich, Janick Frasch, Thomas Breuel
aleksander.lodwich@iupr.dfki.de, janick.frasch@iupr.org, tmb@iupr.dfki.de

DFKI
Trippstadter Str. 122, 67663 Kaiserslautern, Germany

Abstract. Benchmarking pattern recognition, machine learning and data mining methods commonly relies on real-world data sets. However there exist a couple of reasons for not using real-world data. On one hand collecting real-world data can become difficult or impossible for many reasons, on the other hand real-world variables are difficult to control, even in the problem domain; in the feature domain, where most statistical learning methods operate, control is even more difficult to achieve and hence rarely attempted. This is at odds with the scientific experimentation guidelines mandating the use of as directly controllable and as directly observable variables as possible. Because of this, synthetic data is a necessity for experiments with algorithms. In this report we present four algorithms that produce data with guaranteed global and intra-data statistical or geometrical properties. The data generators can be used for algorithm testing and fair performance evaluation of statistical learning methods.

1 Introduction

For better understanding of performance of methods like MLPs (Multi Layer Perceptrons in combination with the many learning algorithms) or SVMs (Support Vector Machines) it would be desirable to have a set of measurable data characterisitics on whose basis the methods can be controlled and compared to each other. Such comparisons could help answer the questions like: Does a particular type of algorithm tend to overfit or underfit? Do initial theoretical justifications hold true in experiments or do some algorithms have unexpected strengths or weaknesses in particular areas? But there are also practical questions beyond academia, like what are the true costs of focusing on one or another technology from a software engineering point of view.
According to the work done on the No-Free-Lunch theorem [1] it's almost for sure, that even the basic statistical characteristics of data have an impact on the performance of any statistical Pattern Recognition or Data Mining algorithm. Due to the lack of really objective criteria for measuring the influence of data

and its structure on the algorithm performance it occurs frequently that different experts argue feverishly for their favorite methods in the context of their application field.
For example a hitherto unsolved riddle is why Support Vector Machines (SVMs) seem to be currently preferred over Multi-Layer Perceptrons (MLPs) (both explained in [2]) in areas like speech or face recognition, even though no better explanation can be given than the observation that they seem to work better. Osowski et. al. conclude that SVMs are superior classifiers while MLPs are superior regressors [3]. However, their explanation is not universal since their data was highly artificial (spiral problem and Mackey-Glass time series) and therefore it is hard to transfer the result to any particular other dataset.
As Geman points out in [4] that is maybe because non-parametric statistical estimators (like MLP) are generally inadequate for the classification task (which supports [3]). However, Werbos reports that MLPs play an important role in his ADP (approximate dynamic programming) designs [5], because of their ability to approximate non-linear dynamics by gradient methods. Werbos argues that decisions made by ADP systems resemble brain-like intelligence level. Such statements contradict each other.
Backing up theoretical consideration with generalizable empirical insight is of high practical importance as the MLP and SVM models are universal function approximators [6][7].
In order to approach a problem of this kind systematically, the right questions need to be asked and answered. We think that in any given situation a preeminent way for choosing a pattern recognition method would lead through answering one or both of the following two questions:

- What data characteristics allow for a more informed decision about the choice of classification, regression or clustering method(s) (or combinations thereof)?
- How does the data need to be transformed in order to exploit the capabilities of a method optimally?

Answering these questions, properly, often relies on a great amount of data. Naturally, for methods motivated by real world applications, the most common approach so far is to use real world data e.g. from the UCI repository [8] or by collecting it on one's own. Unfortunately, as Rachkovskij and Kussul point out in [9] there are two major drawbacks in using such data:

- First of all, accessing real-world data in most domains is difficult for budget, technical or ethic reasons to name some. This typically results in a limited amount of data available for testing purposes, which can impede and even preclude important theoretical and practical considerations.
- The second category of drawbacks concerns the controllability of standard characteristics (e.g. covariance, Bayes error, geometric aspects, external and intrinsic dimensions), which for many collections of real world data are not fully known and extremely difficult to control.

Even large scale benchmarking experiments, as they were done e.g. by Caruana et al. [10], by Bauer and Kohavi [11] or Statlog [12], which give a fair hint on which methods to use in an uninformed case, do not control all variables. Even if it was attempted to control some variables like the type of variables and value ranges, no one can rule out probatively that there is no bias towards some property in the training data that made one method to be favored over another. Synthetic data can play an important role in the gaps of current research. In that we disagree with the statement of the Statlog report [12], saying that using synthetic data will predefine the measured results. The exact statement in [12] is:
Using artificial data has the advantage that conditions can be altered at will, but the disadvantage that in defining the class and type of noise one almost defines the best algorithm for finding the class. For example, if the class is defined by some logical rule, then some symbolic algorithm would be expected to be successful, but linear discrimination may have problems. To judge on the empirical applicability of algorithms on large real-world problems, it is best to use large real-world data.
As we have argued above, this conclusion is not as solid as it may seem. Of course, synthetic dataset generators should not be chosen according to their fit to specific learning methods' models. Instead, for full usefulness a generation model should be chosen that generates data according to parameters that can be estimated on real-world data and which are significant for the assignment (e.g. classification or regression). Picking up the example of [12] it means that in future we would need data generators controlling a quality that might be well called "logicality" or "discrimination linearity". This implies that new generators also entail research in new synthetic measures for data.
Newer research [13] shows that artificial data indeed is useful and opens up new ways of approaching solutions for classification systems. Given the importance of the issue, publications on methods, providing synthetic data directly as feature vectors, mainly based on distributional or statistical properties, are quite rare. Indeed, Yaling Pei and Osmar Zaïane observe the same in the closely related field of data mining and comment this situation with:
Surprisingly, little work has been done on systematically generating artificial datasets for the analysis and evaluation of data analysis algorithms in data mining area. [14]

2 Framework for Generalizable and Transferable Knowledge Discovery

In this report a strong point for synthetic data generation is being made. However, there is also a need to give a bigger picture for their proper use and future development.
Firstly, generators must control data characteristics that are actually measurable on real-world datasets, as the ultimate goal of using synthetic data generators is to learn how these characteristics relate to the choice of algorithms. If generators

control irrelevant parameters, then research should concentrate on developing more relevant generators and more relevant characteristics.
Generators should be used primarily for the study of algorithms and should not be developed detached from data characterization. As it was already cited, there are other uses like integrity tests and replacement of missing data. From this study an extensive, well transferable knowledge should be collected based on problem type and measures of data involved.

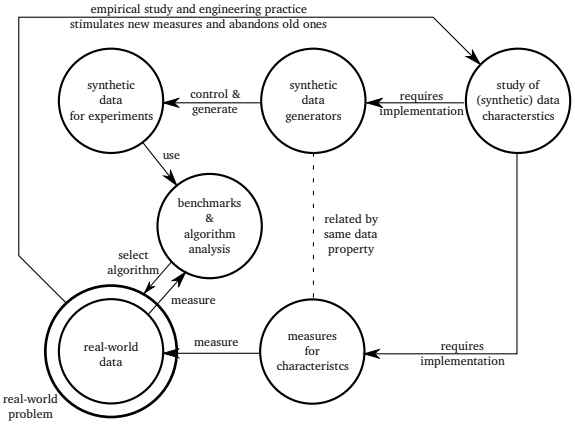


Fig. 1: Framework for generalizable and transferable knowledge discovery (FGTKC).

Our work complies with this framework, as the here proposed generators control the Bayes error rate and it can be estimated via classifier ensembles [15]. There also exist methods for measuring the intrinsic dimensionality (e.g. [16]). Other characteristics can be measured using standard tools from statistics.

3 State-Of-The-Art

Those few examples where synthetic dataset generation has been used in the past spread on a wide range. Most publications focus on the generation of artificial data for one specific application domain.

3.1 Specific Approaches

Helmers and Bunke [17] build handwriting samples from actual character templates for testing handwriting recognition systems. Baird [18] gives a good overview on other methods generating synthetic document image data from real images through the use of degradation models. Purely synthetic data, generated from

statistical models based on gaussian mixture models trained on real-world data has been used for evaluation purposes in protein spot detection methods for 2D images [19]. Other examples of domain specific synthetic data generators are the ones from document image defect models discussed by Baird in [20] and the ones which had been investigated by the US Census Bureau for maintaining the confidentiality of original micro data [21]. The latter generators were also trained from real-world data. For knowledge discovery systems Jeske, Samadi et al. proposed a system for generating synthetic social data based on a to-be-defined semantic graph [22]. Davidov Gabrilovich and Markovitch describe a method for an automated dataset acquisition of labeled textual content from the World Wide Web to be used in text categorization systems [23].
For completeness, some preceding work to be mentioned is the IBM Quest Data Generator [24] and the GSTD [25] where IBM's generator is relational and is targeting at concept learning from relational databases which is a similar work to [22] and where GSTD is motivated by geodesian applications and simulates brownian movement of rectangulary grouped vertices through spatio-temporal feature space. None of the methods controls global statistical properties, however.
There is a series of Hidden Markow Model (HMM) based data generators. Contrary to the methods presented above, for these generators hardly any control over the generation can be excercised. This is due to the fact, that although HMM is a quite general process model final HMM models are trained to be specific, e.g., like a natural source of data. This restricts the value of this method for benchmarking purposes since it follows a totally different objective to the one emphasized in this paper.
All of the above methods have in common that they are narrowly motivated and only work for a very specific problem domain where specific problem dependent variables are known but the resulting feature distributions are quite complex and statistically not fully understood.

3.2 More General Approaches

Rachkovskij and Kussul [9] demonstrated a more general algorithm generating samples from a partitioned feature space including a uniform background noise. The proposed method offers parameters for partitioning the feature space into different regions ("classes") as well as for the distribution of the samples generated from each class. One major drawback of the *DataGen* method is that it only produces class distributions with many uncontrolled properties, which make it difficult to interpret the generated data as feature vectors from any actual real-world scenario. As the authors acknowledge, generating data with internal dependencies is not directly possible with the presented method, but anyway desirable.
Another example for an attempt to create generic data generators is the work [14]. The main goal of this work is to generate data usable for unsupervised learning and outlier detection. The parameters are the number of clusters, distance between the clusters, the total number of points, cluster distributions and

outlier controlling parameters. The generator operates only in two dimensions and in five levels of difficulty. The generator can add uniform background noise and transform the densities into a higher dimensional space. The authors report to have created datasets with up to 4 million samples. The generator does not control any statistical properties. Instead it focuses on geometric relationships. The work we think is most closely related to the one presented here is [26]. This publication has used very simple Gaussian Mixture data models in order to debunk myths or "common wisdom" like *"discriminative classifiers tend to be more accurate than model-based classifiers at classification tasks"* or that *"k-nearest-neighbors (kNN) classifiers are almost always close to optimal in accuracy, for an appropriate choice of k"*. This work most clearly demonstrates the usefulness of well controllable dataset generators. However, the authors have set the centers of the mixtures manually and considered 10 dimensions, 4 classes and 1000 samples/density at maximum which are not representative for real classification problems today. With better tools for dataset generation this promissing path of research can be extended.

4 Proposed Generation Methods

In the following, we will present two new methods on this issue, yielding four synthetic dataset generators, producing datasets that have many parameters in common and few specific ones in order to account for their specialties. Since the generators deliver datasets with a specific Bayes optimal error rate we call them Bayes-true generators. The common characteristics are:

1. Bayes error between the class densities
2. global sample size
3. number of classes
4. number of (intrinsic) dimensions

Through post-processing steps a generator pipeline can also control:

5. dataset covariance
6. centroid
7. external dimensionality

We have designed the generators with the thought in mind that data generated should be qualified and quantified in such a way so that approximatively correct characteristics can be retrieved from real data. All our experiments have been performed in the Python environment. Generators can be downloaded from [27].

4.1 White Gaussians on a *k*-Simplex (WGKS)

The purpose of the WGKS generator is to have a simple, ideal and symmetric model representing very simple statistical problems. These problems are characterized by the absence of any advantages/disadvantages in learning particular

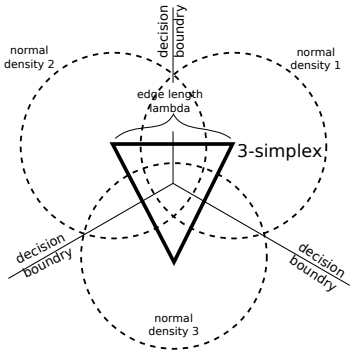


Fig. 2: WGKS places white normal densities on the corners of a k-Simplex.

parts of the data. No class density shall have the chance of being modeled better than any other class density. This generator is creating unimodal problems for each class.

The generation model is based on white Gaussian densities located at the corners of a regular k-Simplex. The k-Simplex explicitly defines the number of densities and implicitly defines the number of dimensions $d = k-1$. This setup guarantees that all density centers are having the same euclidean distance to each other.

The central difficulty in generating data according to this model lies in the fact that there is no known closed solution for computation of the edge length λ of the simplex given a target Bayes error e_t. The obvious step to solve this problem is to perform some kind of optimization of the parameter λ given a measured error e on the data.

The following steps are performed for building a WGKS instance with target error e_t:

initialize λ to some start value $\lambda := \lambda_0$;
repeat
  setup/update of WGKS instance with edge lenght λ;
  compute Bayes error rate e;
  adjust edge lenght λ;
until $\|e - e_t\| \leq \epsilon$;
setup WGKS instance with optimal edge length λ^*;
generate data;

Algorithm 1– Steps in WGKS instance generation

In detail, the steps are implemented as follows:

Setting up the *k*-Simplex The k-simplex S_k can be created recursively by

$$S_1 = \{\overrightarrow{\mu_1}\} = \{\overrightarrow{0}\} \qquad \text{and}$$
$$S_k = \{\overrightarrow{\mu_1}, ..., \overrightarrow{\mu_k}\} = S_{(k-1)} \cup \{\overrightarrow{(centroid(S_{(k-1)}), y_k)}\}.$$

The height $y_k = \sqrt{\lambda^2 - \|base_{k-1}\|^2}$ by the simple Pythagorean consideration that $\lambda^2 = base_{k-1}^2 + y_k^2$, where $base_{k-1} = ||\overrightarrow{centroid(S_{k-1})} - \overrightarrow{\mu_1}||$.

Computing the Bayes error The likelihood of each class c for a sample x_h is given by

$$f_c(x_h, \lambda) = \frac{1}{\sqrt{2\pi}\sqrt{|\Sigma|}} e^{-\frac{1}{2}(x_h - v_c\lambda)^T \Sigma^{-1}(x_h - v_c\lambda)}, \tag{1}$$

where Σ is the covariance matrix (can be omitted when only white Gaussians are considered), λ the edge length of the simplex and v_c are the vertices of the k-simplex with edge length 1.
We are considering two different Monte Carlo methods for computing the Bayes error:

(A) Direct evaluating of the Confusion Matrix. The Bayes error computes by $e = E(l_2 = l) = E(argmax(P) = l)$, where $E(\cdot)$ is the empirical expectation, P is the matrix of the posterior probabilities P (which can be computed by equation 2), l_2 are the Bayesian classifier labels and l are the ground truth labels (which are recorded during sample production).

(B) Importance Sampling. The probabilities of each class c for a sample x_h are given by

$$p(c|x_h) = \frac{f_c(x_h, c)}{\sum_{i=0}^{k} f_i(x_h, i)}. \tag{2}$$

Computing their mean over all samples is guided by the idea of importance sampling and is an estimator of the Bayes error rate $e = E\left(\frac{\max p_h}{\sum_{c \in C} p_{h,c}}\right)$. For a detailed mathematical derivation see Appendix B.2

It can be shown (see Appendix B.2) that the the variance of method *(B)* is always smaller than the variance of method *(A)*, hence on the average method *(B)* converges faster. However, there is a trade-off, since the computation of the likelihoods of each sample in method *(B)* usually takes more time as the check for misclassification in method *(A)* and method *(A)* can be sped up for many kinds of identical densities (e.g. for white Gaussians, this reduces to distance calculations between the sample and the center of each density). Also, method *(A)* is inevitable when the posteriors of the samples are not available but only classifier labels of the Bayesian classifier.

Revision of Bayes Error Computation in WGKS Since the Bayes error computation relies on Monte Carlo simulation, a large number of samples is needed for reasonable accuracy. However, the amount of samples is highly limited by the available computation time and memory. We propose a more efficient approach that exploits the total symmetry of this setup and make CPU and RAM consumption less dependent of the number of densities k. However, the importance sampling method *(B)* for the Bayes error computation is not supported with this approach.

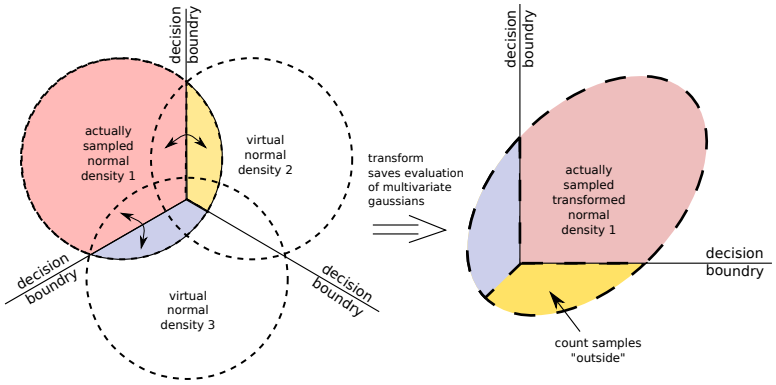


Fig. 3: Exploiting the symmetry of the setup for error computation.

The basic idea is that the total space is divided in equal compartments by the decision hyperplanes. The hyperplanes consist of all equidistant points between pairs of centers. Since all density functions are symmetrically arranged and are identical, the total Bayes error must be equal to the Bayes error in each compartment. (for formal details see Theorem 1 in Appendix C) Therefore, we can reduce the problem to computing the Bayes error in a single compartment of a single prototype density. As it can be seen in Figure 3 the total mass of a density functions outside the decision manifolds containing its center equals the mass of density functions constituting the error in that compartment. That means that the Bayes error e can be computed as the ratio between the density mass lying outside the boundaries and the total density mass. Using Monte Carlo estimation this can be approximated by simply counting the ratio between samples lying outside the compartment and the overall number of samples, where the samples are generated from only one density.
In order to avoid computations of the likelihoods $f_c(x_h, \lambda)$ for each sample x_h, the density can be transformed in a way that the compartment containing the center of the density coincides with the first orthant in the Cartesian coordinate system. Thus the task of performing a Bayes optimal classification on a sample x_h is reduced to the task of checking for a negative sign in one of the coordinates of x_h; subsequent manipulations of λ during optimization are reduced to a translation of the samples. The method can also resample from the density and transform relocate its center in every optimization step. However, this results in higher computational complexity and has the disadvantage of not having a deterministically monotone cost function.
In detail, the transformation works as follows. The densities are denoted by X_i and are centered at μ_i. Note that all vector coordinates are expressed in terms of a global coordinate system where X_1 is located at the origin and X_2 is located on the e_1-axis.

1. Let $\overrightarrow{m}$ be the centroid of the simplex. Let S (cf. Figure 4) be a coordinate system where its first orthant spans the compartment of μ_1. The basis vectors

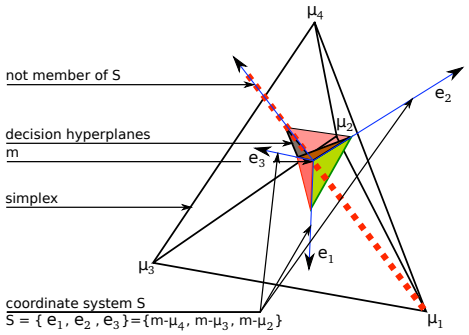


Fig. 4: Obtaining of the coordinate system S.

e'_i spanning S can be computed by $e_i = m - \mu_i$ for all μ_i except for μ_1, where μ_i is the i^{th} corner of the simplex, and normalizing them afterwards.

2. Transform the coordinate system to a Cartesian one S'. Compute μ'_1 by $\mu'_1 = S^{-1} \cdot \mu_1$. Note that it always holds $\mu'_1 = \lambda' \cdot (1, 1, ..., 1)^T$.
3. Since S^{-1} is a linear (and hence affine) transformation, to ensure equivalence of the problems, it suffices to sample from a density $X'_1 = \mathcal{N}(\mu'_1, \Sigma')$ in the new coordinate system, where $\Sigma' = S^{-1} \cdot \Sigma \cdot {S^{-1}}^T = S^{-1} \cdot {S^{-1}}^T$ if the original Σ was the identity matrix.
4. Sample vectors x from a normal density according to Σ'.
5. Find optimal λ'^* for the problem $e(\lambda') \stackrel{!}{=} e_t$, where $e(\lambda') = 1 - E(J(x + \lambda' \cdot \overrightarrow{1}))$ with $x \sim \mathcal{N}(\overrightarrow{0}, \Sigma)$. J is a function that returns a vector of Booleans $\overrightarrow{s}$, being 1 if all component signs of the input vector are positive, i.e. the sample lies in the first orthant.
6. Transform μ_1 from the default simplex with edge length $\lambda_1 = \lambda = 1$ using S into the auxiliary coordinate system and compute the corner reference $\overrightarrow{\mu}_{ref} = S \cdot \overrightarrow{\mu}_1$. The values of $\overrightarrow{\mu}_{ref}$ are all identical ($\mu_{ref,0} = ... = l_{ref,k-1}$) due to the symmetry and the original definitions. Obtain true edge length λ by renormalization $\lambda = \frac{\lambda_1 \cdot \lambda'}{\mu_{ref,0}} = \frac{\lambda'}{\mu_{ref,0}}$.

For more details read Appendix C.

Fast-WGKS using lookup-table Setting up a WGKS dataset can require extensive optimization with repetetive sampling and error estimation. This is quite time consuming. Additionally, the optimization of error in areas of large absolute derivatives will yield λ estimates with high variance. Both problems can be overcome with the lookup-table method, which is a part-analytical approach. The lookup-table method makes use of the fact, that the functions correlating error e, λ for each k are based on the error function. Its main advantage is that it computes λ directly without involvement of any optimization technique. The family type of the functions has been experimentally validated to be the inverse

error function by least-squares approximation in the first step. In contrast, inverse polynomial and trigonometric functions and combination thereof failed to approximate the collected evidence which is presented in figure 5.
In order to optimize as exactly as possible it was attempted to reduce the number of coefficients for approximation as far as possible, possibly to a single one. We achieved that by the observation, that for for $\lambda = 0$ the total Bayes error must be $(k-1)/k$ because all densities have the same posterior distribution function (pdf). There are multiple expressions that could fullfill this goal, however based on the idea of a generalized error function [28] it was intuitively right to assume that mixtures of normal densities do not scale the error function. Indeed, with this restriction in mind there exists a single value on the regular complementary error function that fullfills the condition $\frac{1}{2} \cdot erfc(\frac{x+c_2}{c_1}) = (k-1)/k$ (cf. Figure 7). This allows for a quick computation of c_1 if c_2 is known. The coefficients were found through an expensive computational experiment where λ was optimized 10 times based on 20K samples per density (for high dimensions due to memory limitations). This setup resulted in averages with standard deviations on the order of magnitude of 10^{-4} for λ.

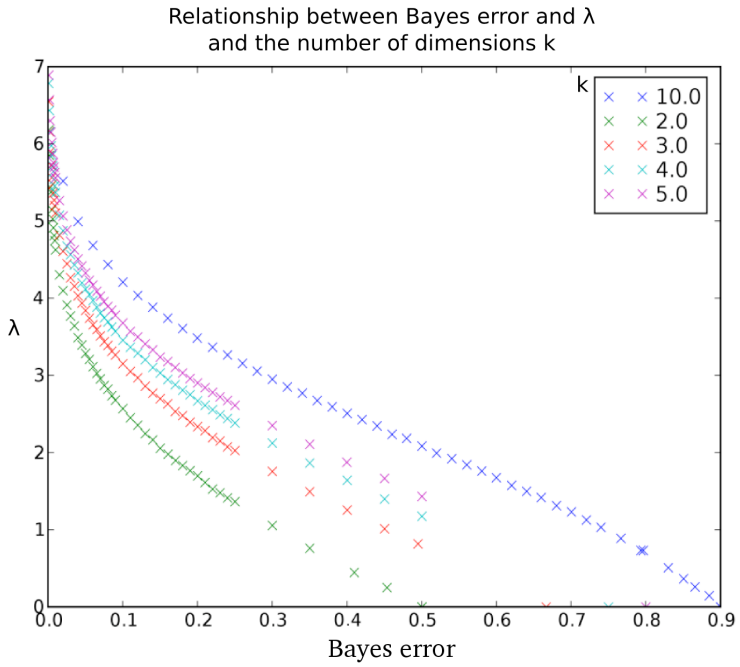


Fig. 5: Relationship between the error and the distance λ for a given number of white normal densities positioned on a k-simplex. The implicit number of dimensions is $k-1$.

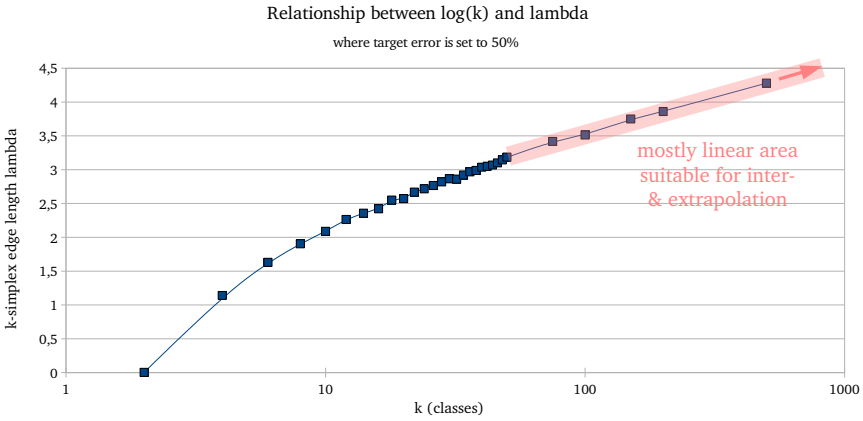


Fig. 6: How λ and k depend on each other (for 50% error).

The coefficient $c_1 = \frac{s_k}{y_s}$, where s_k is listed in Table 2 (support points for 50% Bayes error). For a fixed k, it holds $c_2 = y_s$ and is computed by

$$y_s = erfc^{-1}(\frac{2k-2}{k}).$$

In the case of two densities and a single feature dimension this value is 0, because 50% error is reached exactly when densities overlap each other. In general λ is computed using

$$\lambda_{k,e_t} = \frac{s_k}{y_s} \cdot \left(erfc^{-1}(2e_t) - y_s\right). \tag{3}$$

We consider two special cases, however. The first special case is $k = 2$ where eq. 3 cannot be computed because $y_s = 0$. This is not a problem because in this case the function in question is the regular inverse error function. The other special cases are $k > 100$ where we cannot provide a full list of values.
The extrapolation beyond epoch 100 is footed on 5 supporting points (tab. 1) that we observed to be fairly linear in a log(k)-scale (cf. Figure 6). This dominating linearity allows to interpolate values up to 500 densities and probably extrapolate beyond that to a fair degree. However, 500 densities was the outer limit that we could compute. We applied the least squares method in order to define the parameters of the inter- and extrapolating line and use this line as an estimating function of s_k for class numbers higher than 100. The so obtained s_k is used in the same way as the s_k from the table.

Optimization of λ for non-white Gaussian Mixtures The generator can be modified in order to generate samples from non-white density prototypes. We assume that every density has its own covariance. The target of this section

k	75	100	150	200	500
s_k	3.3438	3.5305	3.75	3.86	4.28

Table 1: Look-Up-Table for extrapolation

k	s_k	k	s_k	k	s_k	k	s_k	k	s_k
1	n.A.	21	2.6265	41	3.0405	61	3.2392	81	3.3885
2	0.0000	22	2.6554	42	3.0554	62	3.2466	82	3.3960
3	0.7855	23	2.6843	43	3.0703	63	3.2541	83	3.4035
4	1.1861	24	2.7132	44	3.0852	64	3.2616	84	3.4109
5	1.4391	25	2.7421	45	3.1002	65	3.2691	85	3.4184
6	1.6308	26	2.7648	46	3.1115	66	3.2765	86	3.4259
7	1.7703	27	2.7875	47	3.1229	67	3.2840	87	3.4334
8	1.8870	28	2.8102	48	3.1343	68	3.2915	88	3.4408
9	2.0058	29	2.8329	49	3.1456	69	3.2989	89	3.4483
10	2.0835	30	2.8556	50	3.1570	70	3.3064	90	3.4558
11	2.1550	31	2.8746	51	3.1645	71	3.3139	91	3.4632
12	2.2266	32	2.8935	52	3.1719	72	3.3213	92	3.4707
13	2.2879	33	2.9124	53	3.1794	73	3.3288	93	3.4782
14	2.3493	34	2.9313	54	3.1869	74	3.3363	94	3.4856
15	2.3981	35	2.9503	55	3.1943	75	3.3438	95	3.4931
16	2.4469	36	2.9653	56	3.2018	76	3.3512	96	3.5006
17	2.4836	37	2.9804	57	3.2093	77	3.3587	97	3.5081
18	2.5202	38	2.9954	58	3.2168	78	3.3662	98	3.5155
19	2.5589	39	3.0105	59	3.2242	79	3.3736	99	3.5230
20	2.5976	40	3.0255	60	3.2317	80	3.3811	100	3.5305

Table 2: Look-Up-Table for WGKS. The table can be regenerated by the module itself according to preset parameters. If higher precision is required, then number of samples can be increased.

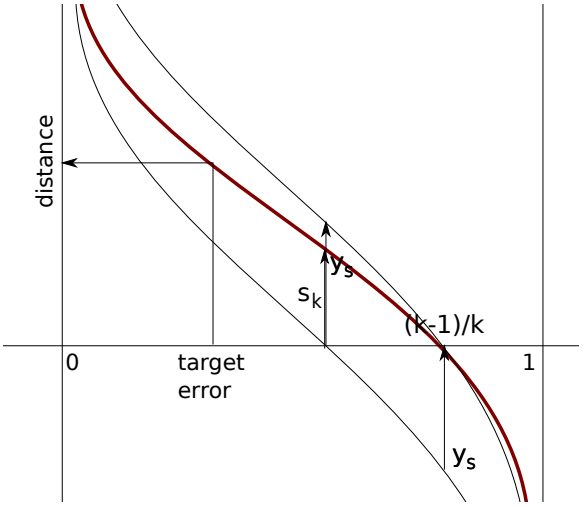


Fig. 7: Interpretation of eq. 3 on how λ is computed for a given k and error e_t given the look up values s_k.

is to show how λ can be found using a gradient descent method. Technically, this approach is not strictly limited to gaussian mixtures but will also work for any mixture of differentiable densities. It is reminded that given the many different possible configurations the distance-to-error-function will most likely become non error-function like and maybe even non-monotonic.
Optimization is done with a gradient descent method with a heuristic for the step length adaptation. We define a cost-function

$$K(X, e_t, \lambda) = (e(X, \lambda) - e_t)^2$$

for optimizing the edge length λ. The necessary derivative $\frac{\partial e(\lambda)}{\partial \lambda}$ is given by

$$\frac{\partial K(X, e_t, \lambda)}{\partial \lambda} = 2(e(X, \lambda) - e_t) \cdot \frac{\partial e(X, \lambda)}{\partial \lambda}.$$

This approach is part-analytical. By part-analytical we mean that we circumvent the problem of knowing the exact integration limits through the use of a Monte-Carlo method. The samples must come from the provided mixture since the equations for computing the error (5) and gradients (6) are based on importance sampling. The practical advantage of this method is that it converges much better than versions based on numerical differentiation.
For the following equations we define:

- n: Number of densities.
- $\mathcal{N}(\cdot, \cdot)$: Normal density.
- d: Integer number of dimensions $d = n - 1$.
- v_k: Vector k from the simplex.

- i, j: Integer counter for dimensions 1 to d.
- a_{ij}: Float values of matrix Σ.
- $\overrightarrow{x}_h$: h^{th} sample vector from F in $\mathbb{R}^d$.
- $x_{hi} = x_i$: i^{th} component of the supplied vector.
- h: Integer counter of samples.
- X: set of $\overrightarrow{x}$.
- $|X|$: total number of vectors/samples.
- e: Bayes error.
- e_t: target Bayes error.
- λ: edge length of simplex.

We define x to be a sample from the Gaussian mixture:

$$x \in \bigcap_k \mathcal{N}(c_k, \theta_k)$$

The probability of error for a sample x from the data set X (defined by the edge length of the simplex λ) is given by

$$P(error|x, \lambda) = 1 - \frac{f_{max}(x, \lambda)}{\sum_k f_k(x, \lambda)}. \tag{4}$$

The total Bayes error e then results to expected value of the error probabilities as specified in eq. 4:

$$\begin{aligned} e &= \frac{1}{|X|} \sum_h^{|X|} P(error|x_h, \lambda) \\ &= \frac{1}{|X|} \sum_h^{|X|} \left(1 - \frac{f_{max}(x_h, \lambda)}{\sum_k f_k(x_h, \lambda)}\right) \\ &= 1 - \sum_h^{|X|} \frac{f_{max}(x_h, \lambda)}{\frac{1}{|X|} \sum_k f_k(x_h, \lambda)} \end{aligned} \tag{5}$$

For the gradient descent method, the derivative of the cost function 4.1 requires the derivative of the Bayesian error rate w.r.t. λ:

$$\begin{aligned} \frac{\partial e(X, \lambda)}{\partial \lambda} &= \frac{\partial}{\partial \lambda} \cdot \left(1 - \sum_h^{|X|} \frac{f_{max}(x_h, \lambda)}{|X| \sum_k f_k(x_h, \lambda)}\right) \\ &= -\sum_h^{|X|} \frac{\partial}{\partial \lambda} \frac{f_{max}(x_h, \lambda)}{|X| \sum_k f_k(x_h, \lambda)} \\ &= -\sum_h^{|X|} \frac{1}{|X|} \frac{u'v - v'u}{v^2} \\ &= -\sum_h^{|X|} \frac{\frac{\partial}{\partial \lambda} f_{max}(x_h, \lambda) \cdot [\sum_k f_k(x_h, \lambda)] - \left[\sum_k \frac{\partial}{\partial \lambda} f_k(x_h, \lambda)\right] \cdot f_{max}(x_h, \lambda)}{|X| \cdot (\sum_k f_k(x_h, \lambda))^2} \end{aligned} \tag{6}$$

The necessary derivative of the density function

$$f_k(x_h,\lambda) = \frac{1}{\sqrt{2\pi}\sqrt{|\Sigma|}} e^{-\frac{1}{2}(x_h - v_k\lambda)^T \Sigma^{-1}(x_h - v_k\lambda)}$$
$$= \frac{1}{\sqrt{2\pi}\sqrt{|\Sigma|}} e^{\left(\sum_i^d \sum_j^d -\frac{a_{ij}}{2}x_i x_j + \frac{a_{ij}}{2}x_i v_{jk}\lambda + \frac{a_{ij}}{2}x_j v_{ik}\lambda - \frac{a_{ij}}{2}v_{ik}v_{jk}\lambda^2\right)}$$

is given by:

$$\frac{\partial f_k(x_h,\lambda)}{\partial\lambda} = \frac{1}{\sqrt{2\pi}\sqrt{|\Sigma|}} \cdot \left(\sum_{i,j}^{d} \frac{a_{ij}}{2}x_i v_{jk} + \frac{a_{ij}}{2}x_j v_{ik} - a_{ij}v_{ik}v_{jk}\lambda\right)$$
$$\cdot\, e^{\left(\sum_i^d \sum_j^d -\frac{a_{ij}}{2}x_i x_j + \frac{a_{ij}}{2}x_i v_{jk}\lambda + \frac{a_{ij}}{2}x_j v_{ik}\lambda - \frac{a_{ij}}{2}v_{ik}v_{jk}\lambda^2\right)}$$

4.2 The GCG, SCG and WMCG Chain Generators

While the purpose of WGKS is to produce extremely transparent lower end problems used for plausibility testing, debugging and basic performance evaluation, there's also a need to produce more sophisticated problems that can actually challenge machine learning and pattern recognition algorithms. The chain generators are able to generate data that can be also interpreted as parallel non stationary processes but can also be understood as a difficult clustering problems. This is done by constructing paths evolving through the feature space and generating samples alongside it with a small perpendicular spread. Examples for that kind of problems are found for adaptive filters (e.g. Haykin [29]) or fiber detection done by Wirjadi [30] where fibers had to be found and their orientation estimated. Borth and Ulges [31] extracted similar data for keyframe extraction in video tagging. From this we see practical motivation for the chain generators, in general.

The Gaussian Chain Generator (GCG), the Spherical Chain Generator (SCG) and the White Markov Chain Generator (WMCG) are all cascades of two generation phases (cf. Fig. 8). The difference between the three lies in the fact that the first phase operates by different rules for path generation. Think of the output of the first stage as of a reusable *skeletton*. Skelettons are paths of variable length and variable number of support points through a high dimensional space. GCG produces skelettons in a cocentrated region, SCG produces skelettons on a hypersphere and WMCG simulates white Markovian motion.

Around this skeletton the second generator adds normally distributed samples with a free σ parameter. This σ controlls the *thickness* of the path. The paths are sought to be of controllable curvature. Therefore, chain generators can be supplied a maximum or a target angle parameter (α) that constrict the bending of a path. A maximum angle of 0° would deliver straight paths only while the predefined angle of 180° results ad ultimo in a white Markov random walk through the feature space.

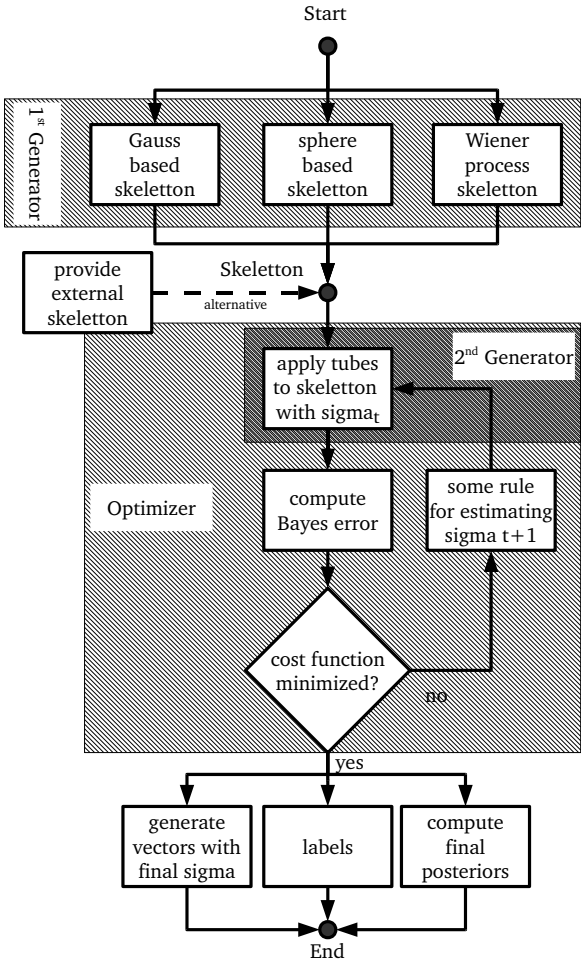


Fig. 8: Relationship between of GCG, SCG and WMCG

GCG Skeletton The GCG skeletton is grown on a multivariate normally distributed set of source (or seed) vertices (denoted as the set S) (By default, this distribution has a covariance matrix $\Sigma = \sigma \cdot I$, where I is the identity matrix). The user can either supply such a set or can specify the mean and the covariances of such a density and the number of seeds to be sampled. The higher this number, the more likely it is that the predefined angle α between adjacent path lines can be found. Examples can be seen in Figures 11h, 11i and 11j.
There is a threefold purpose of this type of generator:

1. *data can be explained with piecewise linear models* (i.e. very simple models). This way even the simplest linear discriminators and regressors can be theoretically justified.

2. *avoids preprocessing.* The generated data is effectively bounded within a box of 7σ units edge length and its overall structure is symmetric. This guarantees that preprocessing is simple or is not not required at all in order to apply the data to any particular learning algorithm. This implies control over the spread in specific dimensions for studying the effects of input scaling on algorithm performance.
3. *known strucutre.* This feature can be beneficial for visualization and clustering analysis. The exact uses are not yet fully investigated.

The generation of a GCG skeletton occurs int two phases: Initalization & Growth. Before the generation can take place it is necessary to make sure that the size of the set S is $|S| >= k \cdot l$, where k is the number of chains and l is chain length. In the first step of the initialization k chain seeds are randomly selected and moved into sorted sets of vertices $X_i \; \forall i \in [1..k]$, which resemble the chains. These seeds are also removed from S. In the second step for every chain X_i and its seed vertex $x_{i,1}$ a nearest neighbor is computed and moved from S to X. With this the initialization of the chains is accomplished and it is guaranteed, that the first section of the chain does not spread across the whole space.
After the two step initialization $l - 2$ additional vertices must be chosen for the chains. This process is sequential, hence the term *growing the Skeletton.* All remaining vertices in S are considered. First, the distances and angles between all head vertices in X and all potential successors in S are computed as shown in Figure 9.
In order to control the smoothness of the curve it is necessary to consider both values, angle (α_j) and distance (d_j) of every source vertex j that is still in S. The selection of the next chain vertex is based on a regularized cost function (eq. 7) with user defined coefficients α (target angle) and s (speed).

$$k_j(\alpha, s, \alpha_j, d_j) = (\alpha_j - \alpha)^2 + \frac{d_j}{s} \tag{7}$$

The selection criterion for the next vertex is then the smallest cost k_j. As in the cost function angle and distance are summed, a larger speed parameter s will make the growing algorithm prefer neighbor vertices that can establish this angle. The control over the angle is important for controlling the internal structure of every chain. E.g. using an $\alpha = 90°$ will deliver chains that grow approximately perpendicular.
After selection the chosen vertices are moved from S to X by

$$X_{i,t+1} = S_{\underset{j}{\operatorname{argmin}} k_j}.$$

This method is performed for each chain in sequence which prevents the situation that a vertex is being shared among chains. The higher the number of seeds, the better the matches found by this approach.

SCG Skeletton The SCG skeletton generator works similar to the GCG. The difference between the two lies in the fact, that the (white) Gaussian vertex

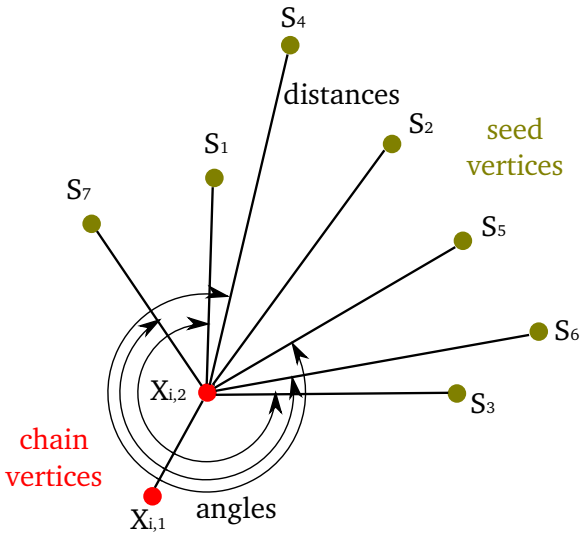


Fig. 9: Every potential new chain vertex is qualified by distance and angle to the current chain head.

source is normalized to a specific user defined radius r. This guarantees that the intrinsic data dimensionality is always $n-1$. Since the chains are 1 dimensional objects, the minimum reasonable number of dimensions for this kind of problem is 3. Otherwise the source points are mapped on two points (1D) or on a circle (2D) where parallel allocation of chains would occur. In such cases the optimizer will fail to deliver the requested Bayes error. From practical considerations, the SCG generator should always use long chains, in order to guarantee a fair level of interference between the chains.

WMCG Skeletton Many problems in physics, biology or chemistry can be modeled through the idea of a Wiener processes or a Random Walk in a high dimensional space. Since statistical pattern recognition and machine learning is used in these fields for automatic experiment evaluation and analysis it appears consistent to also provide a synthetic data generator to cover this ground. Together with the parameters α for the angle, s for the speed, as well as the flags *fix_angle* and *resample* a wide range of interesting datasets can be generated that can be interpreted as particle movement or proteins. Examples can be seen in Figures 11a, ..., 11g.
The starting vertices are randomly picked from a user supplied distribution (in form of samples). This allows control of the overall position of the chains. Starting from these head vertices $l-1$ additional vertices are generated automatically. In every step a white distribution is generated at each head vertex of every chain. This distribution is multiplied with a user defined constant s accounting for the propagation speed. There are two modes for using these distributions. In the first

mode a random vector is picked from that distribution directly, in the other the potential candidate vectors are normalized to be of length σ. Again, X_i denotes every chain and after $|X_i| > 2$ it is possible to restrict the maximum bending of the chain to no more, than the user defined parameter α. However, we provide an option *fix_angle* in order to interpret α in the same way as GCG and SCG do, namely as the *target angle*.
We provide a *resample* option which is on by default. If activated, then the normal distributions are resampled every time a chain is being extended and will yield the distributions known from random walks in the features space. If the distributions are not resampled, then the chains will alter between few or even just two directions creating predominantly unidirectional chains.

All three generators can fail to obtain the target error for specific skelettons. In this case ($|\Delta e| \geq 2\%$) exception is thrown. (However, the framework allows for tighter limits.) This case is common when the initial seeds are widely spread, the speed constant and the chain length are low. The first attempt to fix a chain is to generate a new skeletton for a few times. If for some reason this is not effective, then probably the chains are too short. If chains cannot be made longer, then either the initial seeds variances should be set smaller or the speed constant increased. For unexpected problems the generator package includes a visualization function (projecting the datasets on two user-defined dimensions) that can help identify the problems in the skeleton generation settings.

Adding Perpendicular Spread to the Skeleton In this section the second step in the overall generation process is described which is common to GCG, SCG and WMCG. The generator iterates over every segment t of the chain i and generates a tube (see fig. 10) which is then transformed to the coordinates of the segment.
The longitudinal distribution of samples $\overrightarrow{s}^0_{i,j}$ over a complete chain i is uniform (0 stands for not transformed). Every j^{th} sample's first dimension gets assigned a distance value $d_j \in [0, L]$, where L is the length of the chain ($L = \sum_{t=0}^{segments-1} ||x_{i,t+1} - x_{i,t}||$). All samples are distributed over the chain segments according to their random value d_j for length. Hence the tube (a single chain segment) consists of n_t uniformely sampled scalars, where the other $n-1$ dimensions are sampled from $\mathcal{N}(0, \sigma)$.
After the tube has been generated it must be rotated and translated into the right position. What is known is the relative path vector $\overrightarrow{v}_t$ and its head position $\overrightarrow{x}_{i,t}$, where the path vector $\overrightarrow{v}_t$ is defined as $\overrightarrow{v}_t = \overrightarrow{x}_{i,t+1} - \overrightarrow{x}_{i,t}$. This is an ill posed problem as there exists an infinite number of transformations that would place the tube along the path vector (due to rotational symmetry which is irrelevant to our purposes). However, the singular value decomposition can "guess" the missing orthogonal basis vectors. It only must be made sure that the first basis vector is pointing into the right direction.

$$U \cdot \Sigma \cdot V = \overrightarrow{v}_t \tag{8}$$

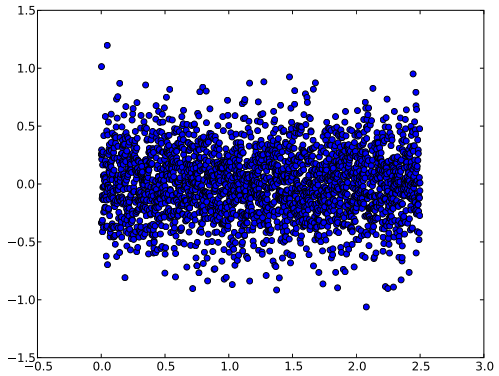


Fig. 10: A 2D tube of length 2.5, a $\sigma = 0.3$ and a density of $longitudinal\_density = 1000$.

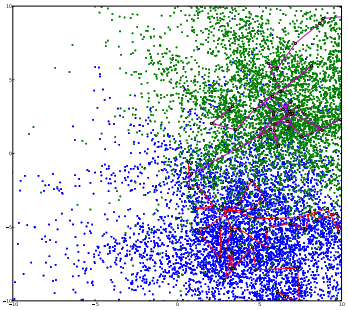
The final transformation of the original tube samples $\overrightarrow{s}^0_{i,j}$ into the well positionend tube samples $\overrightarrow{s}^1_{i,j}$ is then:

$$\overrightarrow{s}^1_{i,j} = V \cdot \overrightarrow{s}^0_{i,j} + \overrightarrow{x}_{i,t} \tag{9}$$

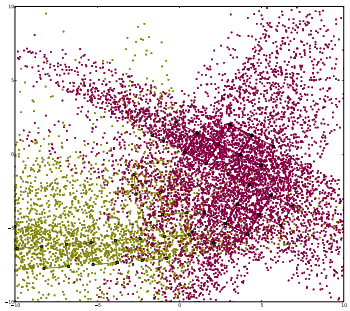
where i is the chain index, t is the index of the chain support vector and where $j \in [0, n]$ is the sample index.

Exercising Control over the Bayes Error The only density parameter that is available in the GCG, SCG or the WMCG generators is the σ parameter, the variance of the perpendicular spread. Theoretically, every skeleton with a $\sigma \to 0$ should deliver $e \to 0$ (in the sense of an *almost sure convergence*). The other direction does not hold true: Increasing $\sigma \to \infty$ does not imply that the Bayes error will reach any specific target error ($e \nrightarrow e_t$). In practice, the adaptation of σ will be found using some kind 1D optimization method. The generators package supports two methods for performing this: downhill simplex and gradient descent.

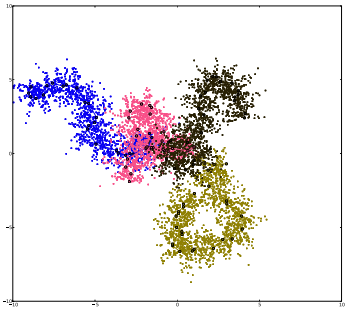
Time Complexity The time complexity of the generators is $O(l \cdot b \cdot k \cdot m \cdot n)$, where l is the number of optimization steps, b the number of samples used, k the number of chains, m the number of sections per chain and n the number of dimensions. This complexity is based on the posterior computation which is the most complex operation. It expresses that during every optimization step $j = 1..l$ a matrix of posterior probabilities of size b by k must be computed, where each value is a sum of m segment posteriors where each segment operation is proportional to n.



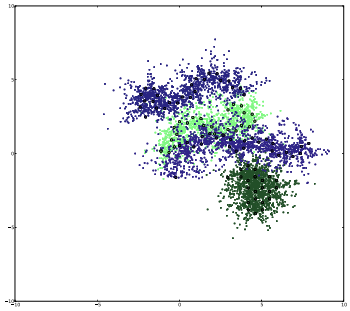
(a) 2D WMCG with 10% error, 2 chains



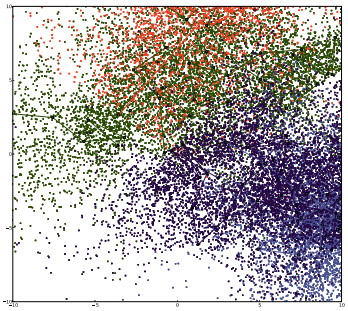
(b) 2D WMCG with 10% error, 2 chains, 50 chain links, angle fixed to 90°



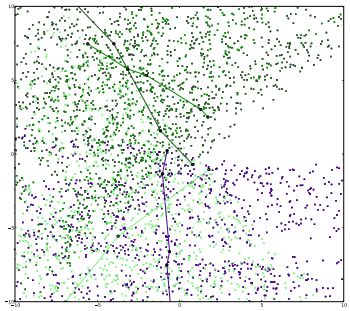
(c) 2D WMCG with 10% error, 4 chains, 25 chain links and fixed angle of 30°



(d) 2D WMCG with 25% error, 4 chains, 25 chain links and fixed angle of 60°

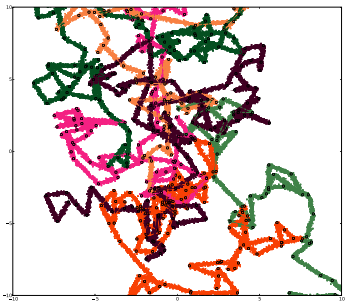


(e) 2D WMCG with 25% error, 4 chains, 50 chain links and no restrictions

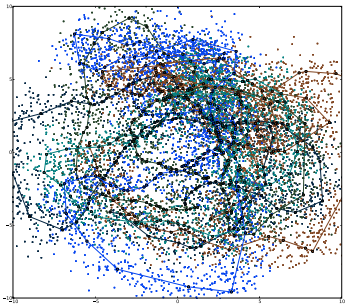


(f) 2D WMCG with 25% error, 4 chains with a fixed angle of 15°

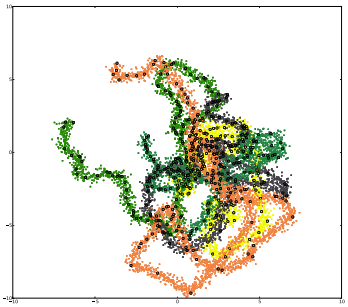
Fig. 11: Examples of data generated by the WMCG generator.



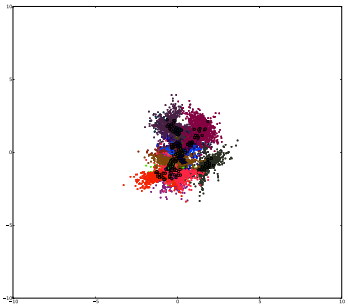
(g) 2D WMCG with 10% error, 6 chains with 100 links



(h) 2D GCG with 50% error, 5 chains with 100 links, speed of 10, target angle $\alpha = 0°$ and stray radius of 3



(i) 2D GCG with 30% error, 5 chains with 100 links, speed of 1, target angle $\alpha = 15°$ and stray radius of 3



(j) 2D GCG with 35% error, 15 chains with 15 links, and stray radius of 1

Fig. 11: Examples of data generated by the different chain generators.

4.3 Post Processing for Guaranteeing Intrinsic Dimensionality

Our generator package provides a series of post processing steps that can be used to create variants of the same data set. In this section we propose arbitrary translations, scalings, shearings and rotations. The main purpose of these methods is to embed the generated data into higher dimensional spaces, maintaining intrinsic dimensionality by adding zero-filled extra feature-columns and applying transformations afterwards.

Random Translation Translations do not change any other parameters than the means and are simply implemented by adding a random vector to all samples. There are different variations for how this random vector can be obtained:

1. given a direction vector multiplied with a random scalar. Scalars can be chosen uniformely or normally.
2. given a radius, a random sample from a white normal density is generated and normalized to the radius.
3. a random vector from a parameterized normal density is picked.

Random Rotation Another possible post processing step is rotation. It changes the means and the absolute covariance characteristics but the shape and the relative covariance characteristics are preserved.
In order to obtain a random rotation matrix with each rotation equally likely, we make use of the fact, that each orthogonal matrix with determinant 1 represents a rotation.
We start out with a $n \times n$ matrix with entries $x_{i,j} \sim \mathcal{N}(0,1)$. Since this resembles n n-dimensional normally distributed random vectors, we get an equally distributed likelihood concerning the direction (angles). We then orthogonalize this matrix, e.g. by a QR decomposition, taking Q (which already has determinant 1), and yield an orthogonal matrix with each rotation direction equally likely, since the directions of the starting vectors were uniformly random. This procedure has been described by Genz in [36].

Random Scaling As scaling parameter we chose a log-normal distributed variable $z \sim \mathcal{N}_{log}(0,\sigma)$. According to the survey done by Limpert et. al. [37] this is the most dominant form of one-sided distributions for natural problems. The characteristic of the chosen distribution parameters is that the distribution median is at 1. The σ-parameter, which is accounting for the spread, is user-defined.

Random Shearing The shearing operation is understood as breaking a perpendicularity of the transformation matrix T_s. An easy way to break the perpendicularity is to start with a orthogonal matrix I (identity matrix) and modify few (w) specific linear dependencies. At random a subset of matrix values is modified (not including the diagonal). The total number of modifications is specified by the user. The new values are sampled from a normal distribution with a user-defined σ. The following operation is performed w times:

$$M_{i,j} = N(0,\sigma) \mid i \neq j$$

Fixing outer $\boldsymbol{\Sigma}$ This method transforms a given dataset affinely in a way, such that it follows a given covariance matrix. This is achieved in three steps. In the first step a PCA is performed on the data X, so that eigenvectors are obtained and, scaled by their eigenvalues, stored in a Matrix M_1. Applying M_1^{-1} to X will

rotate and whiten the supplied data and yield X'. Next, we perform eigenvalue decomposition of the target covariance matrix Σ and thus obtain a matrix M_2 of the eigenvectors, again scaled by their eigenvalues. We then transform X' by M_2 and yield X_t which satisfies the desired covariance matrix Σ.
Formally, for a data point x, we have:

$$x_t = M_2 \cdot (M_1^{-1} \cdot (x - \boldsymbol{c})) + \boldsymbol{c}$$

5 Discussion

The main purpose of synthetic data generation is the possibility to perform controlled experiments that are reproducible by others. Natural data is often quite large or inaccessible, sometimes for such profane reasons as data privacy, inconvenient size or file format or due to licensing issues. In such cases experiments are not reproducible. Even if the datasets from the UCI repository are used for reference, all the problems in there are from natural sources and hence their characterizations do not systematically cover any arbitrarily chosen feature space. This withdraws them from the use in systematic feature analysis.
Our generators help in both ways: On one hand, instead of actually providing data any author can describe the procedure of data generation that can be reproduced according to these descriptions. The implementations of the here proposed generators produce the descriptions automatically, so that they can be either attached to a publication or made available for download on the web. On the other hand our generators allow to perform a variety of hypothesis testing experiments by guaranteeing specific statistical properties. Our generator can guarantee a specific global Bayes error rate, sample size, structure (relationships), number of external and intrinsic dimensions, dataset covariance and the data centroid. Apart from that it is possible to add transformations to test for affine data variations.
This is helpful for finding bugs in new algorithms as they should not get influenced by these transformations. With synthetic Bayes-true generators it is much easier to experimentally test systems and track down their sensitivity to unique factors. Such data generators can be used in unforeseen ways. One idea that comes into mind is that of comparative training. It seems plausible to expect that real world data with specific statistical properties should behave similar during training. This could allow for plausibility training. Plausibility training could be used as a test whether a training result performs up to expectation or whether it should be rerun. We believe that similar ideas are worth further investigation.
The proposed generators can deliver data for testing supervised and unsupervised training in high dimensional space. This data can be embedded in higher dimensional spaces while keeping the structure of the data and probability relationships. In this way the generators indirectly control the ratio between intrinsic and external dimensions.
The structure of the data is predictable, repeatable and comprehensive. The WGKS generator simulates a trivial lower end problem while WMCG can sim-

ulate complex structures that resemble fibers and loops and partially linear, smooth processes. Our generators deliver true posteriors that allow to compare how well classifiers estimate the posteriors. In systems where classifiers are used in conjunction it is assumed that the interacting parts communicate via probabilities. It is natural to expect that since many algorithms assume Bayesian theory for explanation of machine learning algorithms they will perform worse if proliferated values are not matching the true posteriors, well.
The generators are open source (MIT license) and are available from our project website [27] and are running in Python. SciPy is required, but beyond this requirement the generators can run almost in any computer environment. SciPy allows exporting of data to many formats including Matlab so that Matlab users can harness the provided generators.

6 Conclusion

Synthetic data generators which can control the statistic properties of the feature vectors are important tools for experimental inquiries performed in context of machine learning and pattern recognition. We propose two new types of generators that can be used to model static and dynamic processes. We consider this work as a cue to stimulate further work in this field. In general, we like to see the contributions of this report to be seen within the synthesis framework in Figure 1 that we think would make the benchmarking efforts more systematic.

7 Acknowledgments

This report has been made possible though funding of the PaREn (Pattern Recognition and Engineering) [27] project by the BMBF (Federal Ministry for Education and Science, Germany).

References

1. David H. Wolpert, William G. Macready: *No Free Lunch Theorems for Optimization.* IEEE Transactions on Evolutionary Computation, Vol. 1, No. 1, pp. 67-82, 1997
2. Trevor Hastie, Robert Tibshirani and Jerome Friedman: *The Elements of Statistical Learning.* ISBN 0387952845, Springer, 2001
3. Stanislaw Osowski, Krzysztof Siwek, Tomasz Markiewicz: *MLP and SVM Networks a Comparative Study.* Proceedings of the 6th Nordic Signal Processing Symposium - NORSIG 2004, Espoo, Finland, June 9-11, 2004
4. S. Geman, E. Bienenstock, R. Doursat: *Neural networks and the Bias/Variance dilemma.* Neural Computation, Vol. 4, pp. 158, 1992
5. P. J. Werbos: *Intelligence in the brain: A theory of how it works and how to build it.* Neural Networks, Vol. 22, No. 3, pp. 200-212, 2009

6. James Ting-Ho Lo: *Multilayer perceptrons and radial basis functions are universal robust approximators*. Neural Networks Proceedings 1998, EEE World Congress on Computational Intelligence, The 1998 IEEE International Joint Conference on Neural Networks, Vol. 2, pp. 1311-1314, May, 1998
7. Barbara Hammer, Kai Gersmann: *A Note on the Universal Approximation Capability of Support Vector Machines*. Neural Processing Letters, Vol. 17, pp. 43-53, Kluwer Academic Publishers, 2003
8. UC Irvine Machine Learning Repository: http://archive.ics.uci.edu/ml
9. Dmitri A. Rachkovskij, Ernst M. Kussul: *DataGen: a generator of datasets for evaluation of classification algorithms*. Pattern Recognition Letters, Vol. 19, pp. 537544, 1998
10. Rich Caruana, Alexandru Niculescu-Mizil: *An Empirical Comparison of Supervised Learning Algorithms*. ICML '06: Proceedings of the 23rd international conference on Machine learning, Pittsburgh, Pennsylvania, pp. 161-168, ACM, New York, USA, 2007
11. Eric Bauer, Ron Kohavi: *An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting, and Variants*. Machine Learning, Vol. 36, pp. 105139, Kluwer Academic Publishers, 1999
12. R. D. King, C. Feng, A. Sutherland: *StatLog: Comparison of Classification Algorithms on Large Real-World Problems*. EU Project, 1995
13. C.M. van der Walt: *Data Measures that Characterize Classification Problems*. Master Thesis, University of Pretoria, Pretoria, South Africa, Feb. 2008
14. Yaling Pei, Osmar Zaïane: *A Synthetic Data Generator for Clustering and Outlier Analysis*. Computing Science Department University of Alberta, Edmonton, Canada T6G 2E8, 2006
15. K. Turner, J. Ghosh: *Estimating the Bayes Error Rate through Classifier Combining*. ICPR'96, Pattern Recognition, 13th International Conference on, vol. 2, pp. 695, 1996.
16. B. Kegl: *Intrinsic dimension estimation using packing numbers*. NIPS, Vol. 15, pp 681–688. 2002
17. Muriel Helmers, Horst Bunke: *Generation and Use of Synthetic Training Data in Cursive Handwriting Recognition*. Pattern Recognition and Image Analysis, Vol. 2652, pp. 336-345, Springer, Berlin/Heidelberg, Sep. 2003
18. Henry S. Baird: *The State of the Art of Document Image Degradation Modeling*. In Proc. of 4 th IAPR International Workshop on Document Analysis Systems, pp. 1-16, Rio de Janeiro, 2000
19. Mike Rogers, Jim Graham, Robert P. Tonge: *Using statistical image models for objective evaluation of spot detection in two-dimensional gels*. Proteomics, Vol. 3, pp. 879886, 2003
20. Henry S. Baird: *Document Image Defect Models and Their Uses*. In Proceedings of the Second International Conference on Document Analysis and Recognition ICDAR-93, pp. 62-67, 1993
21. John M. Abowd and Julia I. Lane: *Synthetic Data and Confidentiality Protection*. Technical paper No. TP-2003-10, U.S. Census Bureau, LEHD Program, 2003
22. Daniel R. Jeske et al.: *Generation of Synthetic Data Sets for Evaluating the Accuracy of Knowledge Discovery Systems*. KDD05, Chicago, USA, Aug. 21-24, ACM, 2005
23. Dmitry Davidov, Evgeniy Gabrilovich, Shaul Markovitch: *Parameterized Generation of Labeled Datasets for Text Categorization Based on a Hierarchical Directory*. SIGIR04, Sheffield, South Yorkshire, UK, ACM, Jul. 25-29, 2004

24. R. Srikant. IBM Quest Synthetic Data Generation Code, 1999. Not available online anymore.
25. Yannis Theodoridis, Jefferson R.O. Silva, Mario A. Nascimento: *On the Generation of Spatiotemporal Datasets*. SSD99, LNCS 1651, pp. 147-164, Springer, Berlin/Heidelberg, 1999
26. C.M. van der Walt, E. Barnard: *Data characteristics that determine classifier performance*. SAIEE Africa Research Journal, Vol 98 (3), pp 87-93, September 2007.
27. Pattern Recognition and Engineering (PaREn), DFKI, BMBF Project: http://paren.iupr.com/, 2008, 2009, 2010
28. M. Brown: *A Generalized Error Function in N Dimensions*. Technical Memorandum No. NMC-TM-63-8, U.S. Naval Missile Center, Point Mugu, California, USA, 12. Apr. 1963
29. Haykin: *Neural Networks: A Comprehensive Foundation*. 2nd ed., Chapter on ANN filtering, Prentice-Hall, Englewood Cliffs, NJ, 1999
30. Oliver Wirjadi: *Models and Algorithms for Image-Based Analysis of Microstructures*. PhD Thesis, Kaiserslautern University of Technology, Kaiserslautern, Germany, 2009.
31. Damian Borth, Adrian Ulges, Christian Schulze, Thomas M. Breuel: *Keyframe Extraction for Video Tagging & Summarization*. Informatiktage 2008, pages 45-48, pubs.iupr.org
32. Richard M. Kane: *Reflection Groups and Invariant Theory*. 1st rev., Springer, New York, July, 2001
33. Alan F. Beardon: *Algebra and Geometry*. Cambrdidge University Press, 2005
34. Ezra Miller, Victor Reiner, Bernd Sturmfels: *Geometric Combinatorics*. 1.7 rev., p. 70, IAS/Parc City Mathematics Institute, AMS, 2007
35. Richard O. Duda, Peter E. Hart, David G. Stork: *Pattern Classification*. 2nd ed., sec. 2.6.1 - 2.6.2, John Wiley & Sons, Canada, 2001
36. Alan Genz: *Methods for Generating Random Orthogonal Matrices*. MCQMC98, 1998
37. Eckhard Limpert, Werner A. Stahel, Markus Abbt: *Log-normal Distributions across the Sciences: Keys and Clues*. BioScience, Vol. 51, No. 5, pp. 341-352, May, 2001

APPENDIX

A Speed up for Bayes error computation

The total speed up of this method is depending on the variants used. If the WGKS generator is based on all data and is using arbitrary normal densities then the total runtime F is proportional to $k^5 \cdot samples$. If the densities can be restricted to just white normal densities, then a matrix operation is removed and then $F \propto k^3 \cdot samples$. For the symmetry exploiting optimized method there exist two versions, one with and one without resampling.
Figure 12 explains that depending on the setup the total speed up can be as low as k and in other cases as high as k^4. The optimized implementations published in [27] are all without resampling, so that at least a speed up proportional to k^3 is experienced.

B Variance Comparison of Bayes Error Computation Methods

The methods *(A)* and *(B)* from section 4.1 for the Bayesian error rate computation will be compared formally in terms of their variance in the following.
We compute the error rate by $e = 1 - cor$, where *cor* is the fraction of samples being classified correct by the optimal Bayesian classifier with equal priors. For comparing the two methods of computing *cor* we will need the following definitions:

- $\mathbb{1}_M(x)$ is the characteristic function of the set M,
- $c_o(x)$ denotes the ground truth, i.e. the originator class of a sample x,
- $\hat{c}(x)$ denotes the the class with the largest Bayesian posterior probability,
- $f(x)$ is the density function of any of the identical densities in the WGKS model,
- $p(M)$ is the probability of a set M
- A_j denotes the Bayesian acceptance region of density j, A_o denotes the one of the originator class c_o
- k is number of densities,
- $f^*(x) = \frac{1}{k}\sum_{j=1}^{k} f_j(x)$ is the mixture density of all WGKS densities and
- $|X|$ is the number of samples.

B.1 Method *(A)*: Confusion Matrix Evaluation

We have

$$cor = \int \mathbb{1}_{\hat{c}(\cdot)=c_o(\cdot)}(x) \cdot f^*(x)dx \approx \frac{1}{|X|}\sum_{h=1}^{|X|} \mathbb{1}_{\hat{c}(\cdot)=c_o(\cdot)}(x)$$

by standard Monte Carlo estimation, where x is sampled from the mixture of all densities f^*. For the estimator $\mathbb{1}_{\hat{c}(\cdot)=c_o(\cdot)}(x)$ of a sample x we have

$$\mathbb{1}_{\hat{c}(\cdot)=c_o(\cdot)}(x) = \mathbb{1}_{A_o}(x)$$

Hence for the variance of the estimator it holds:

$$var(\mathbb{1}_{A_o}(x)) = p(A_o) \cdot (1 - p(A_o)) = (1 - e) \cdot e.$$

B.2 Method *(B)*: Importance Sampling

Derivation: It holds

$$\begin{aligned} cor &= \int \mathbb{1}_{\hat{c}(\cdot)=c_o(\cdot)}(x) \cdot f^*(x)dx \\ &= \frac{1}{k} \sum_{j=1}^{k} \int \mathbb{1}_{A_j}(x) \cdot f_j(x)dx \\ &= \frac{1}{k} \int f_{max}(x)dx, \end{aligned}$$

where $f_{max}(x) = \max_{j=1..k} f_j(x)$. This can be approximated, using the idea of importance sampling:

$$\begin{aligned} \tfrac{1}{k} \int f_{max}(x)dx &= \frac{1}{k} \int \frac{f_{max}}{\frac{1}{k}\sum_{j=1}^{k} f_j(x)} \cdot f^* dx \\ &= \int \frac{f_{max}}{\sum_{j=1}^{k} f_j(x)} \cdot f^*(x)dx \\ &= E^* \left(\frac{f_{max}(x)}{\sum_{j=1}^{k} f_j(x)} \right) \\ &\approx \frac{1}{|X|} \sum_{h=1}^{|X|} \frac{f_{max}(x_h)}{\sum_{j=1}^{k} f_j(x_h)}. \end{aligned}$$

Note that the x_h are sampled from the joined mixture density.

Variance: We can compute the variance of the estimator $\frac{f_{max}(x)}{\sum f_j(x)}$ by

$$\begin{aligned} var\left(\frac{f_{max}(x)}{\sum f_j(x)}\right) &= E^*\left(\left(\frac{f_{max}(x)}{\sum f_j(x)}\right)^2\right) \\ &\qquad - \left(E^*\left(\frac{f_{max}(x)}{\sum f_j(x)}\right)\right)^2 \\ &= \frac{1}{k}\int \frac{(f_{max}(x))^2}{(\sum f_j(x))^2}\cdot\sum_{i=1}^{k} f_i(x)dx \\ &\qquad - \left(\frac{1}{k}\int \frac{f_{max}}{\sum f_j(x)}\cdot\sum_{i=1}^{k} f_i(x)dx\right)^2 \\ &= \frac{1}{k}\int \underbrace{\frac{f_{max}(x)}{\sum f_j(x)}}_{\leq 1}\cdot f_{max}(x)dx \\ &\qquad - \left(\underbrace{\frac{1}{k}\int f_{max}(x)dx}_{=1-e \text{ by (10)}}\right)^2 \end{aligned}$$

since the densities are identical and have the same error rate. This can further be reduced to

$$\begin{aligned} var\left(\frac{f_{max}(x)}{\sum f_j(x)}\right) \leq (1-e)-(1-e)^2 &= (1-e)\cdot e \\ &= var(\mathbb{1}_A(x)), \end{aligned}$$

the variance of the confusion matrix estimator.
Hence it holds var(estimator (A)) $\geq$ var(estimator (B)) and it is easy to see that estimator (B) is better in the case of $f_{max}(x) < \sum_{j=1}^{n} f_j(x)$.

C Transformation for Bayes Error Computation Speed-Up and its Formal Verification

Lemma 1 *Let $D_1, D_2, ..., D_k$ be $k = n+1$ densities with centers $\mu_1, \mu_2, ..., \mu_k$ located on the corners of a regular k-Simplex in an n dimensional Euclidean Space. Let $B_{j,l}$ be the $n-1$ dimensional perpendicular line bisector of the centers of the densities D_j and D_l , $j \neq l$. If*

1. *all pairs of densities D_j and D_l are symmetric by reflection through $B_{j,l}$ and*
2. *each density D_i is mirror symmetric by reflection through each $B_{j,l}$ $\forall j, l \in \{1, ..., k\} \setminus \{i\},\ j \neq l$,*

then all densities are positioned rotationally invariant, i.e. a rotation by any true rotation $S \in \text{Sym}(k-\text{simplex})$, the symmetrical group of the k-Simplex, does not change the overall setup except for the labels of the densities in the respective global position.

Proof. From elementary geometry we know that in 2D a rotation is equivalent to two concatenated reflections through lines α and β [32]. The angle of the rotation equals 2 times the angle between α and β. This can be generalized to 3D [33] and arbitrary dimensions [32],using dihedral angles (the angles between two Hyperplanes). Hence we can express a rotation $S \in \text{Sym}(k-\text{simplex})$ by a concatenation of an even number of reflections through some hyperplanes $B_{.,.}$. Note that due to the setup of the densities on a regular k-Simplex, each $B_{j,l}$ runs through all other centers $x_i,\ i \neq j,l$, hence fixes them [34]. By assumption (2), for each reflection through $B_{j,l}$, all other densities X_i remain in an equivalent state. By assumption (1) the densities X_j and X_l are symmetric by reflection through $B_{j,l}$, hence for the overall setup this just results in a swap of the labels j and l. All in all this means the simplex is rotation invariant to any rotation from $\text{Sym}(k-\text{simplex})$.

Theorem 1 *Let $N_1, N_2, ..., N_k$ be $k = n+1$ identical Normal distributions with covariance matrices $\Sigma_i = \sigma^2 I$ and centers $\mu_i,\ i = 1..k$ located on the corners of a regular k-simplex in an n-dimensional Euclidean Space E. Let $B_{i,j}$ be the $(n-1)$-dimensional perpendicular line bisector of the centers of the densities N_i and N_j , $i \neq j$.*
Then $B_{i,j}$ are the Bayesian decision boundaries. Furthermore, the overall Bayesian error rate is equivalent to the error rate of each density.

Proof. $B_{i,j}$ is the Hyperplane consisting of all equidistant points between μ_i and μ_j. Since N_i and N_j are identical and have diagonal covariance matrices (being scalar multiples of the identity matrix), the Bayesian decision boundary (assuming equal priors) is $B_{i,j}$ [35]. The boundaries $B_{i,j}\ \forall i,j = 1..k,\ i \neq j$ subdivide E into k compartments, each defined by $\{B_{i,j} \mid j \in \{1..k\} \setminus \{i\}\}$ corresponding to the space where one density N_i dominates the others. Obviously a pair of densities N_j and N_l are symmetric by reflection through $B_{j,l}$ and clearly each density N_i is invariant to reflection through any other $B_{l,j},\ l,j \in \{1..k\} \setminus \{i\},\ l \neq j$, since such a $B_{l,j}$ runs through μ_i. By Lemma 1 this means the densities arranged on the simplex are rotational invariant and hence the error rate in each compartment must be the same and hence be equivalent to the overall error rate. Again as a result from the symmetry of N_i and N_j (and the invariance of all other densities), the density mass that is misclassified as "Class i" though belonging to N_j equals the density mass that is misclassified as "Class j" though belonging to N_i. Using this argument for all densities N_j neighboring N_i, we get that the error rate in each compartment is equivalent to the error rate of each single density N_i.

D Usage of the DFKI Bayes-True Generators Package V1.05

We provide reference implementation of the proposed generators. The advantages of the package are:

- a "Micro Code" configuration allows collecting and verifying sequences of generation steps
- such Micro Code programs can be either run directly or get compiled into a selfsufficient python program
- the package knows "transactions" that can be automatically parallelized for multicore systems
- data exporting functionality into various formats
- proliferation of compact generation programs instead of large chunks of data

Here we briefly want to demonstrate the way the generators can be used.

D.1 Getting Started

First, the generators must be downloaded from the PaREn website [27] and copied and unpacked into the target directory. If the generators shall be used from a different folder the right PYTHONPATH has to be set. After that the library can be imported via:

```
from generators import *
```

D.2 Generators

The package supports five codes for each of the generators WGKS, KWHITES, GCG, SCG, WMCG. The first parameter is always a string-name that will identify the dataset in further process. If the name is no supplied, then the set is given an automatic name "default" plus some number. KWHITES is a WGKS generator, but instead of supplying the target error rate, λ must be defined. The following examples are not examples of calls to a function, instead they are examples of *Micro Code* generators. After invoking a series of these calls one must call EXEC() in order to start the code generation. Micro Code calling examples are:

```
WGKS()

WGKS( "first_wgks", 3, 0.1, 1000 )

WGKS( "second_wgks", no_of_densities=3, target_error=0.1,
samples_per_density=1000, optimizer="lut",
samples_for_optimization=1e5 )

KWHITES( "third_wgks", no_of_densities=3,
```

```
distance=1.333, samples_per_density=250 )

GCG()

GCG( "some_gcg", 0.1, 20, 40, 1000, total_samples=100000 )

GCG( "another_gcg", target_error=0.15, no_of_chains=2, chain_lengths=400,
no_of_chain_seeds=1000, seed_sigma=scipy.eye(2),
target_angle=180.0, speed=1.0, total_samples=10000 )

SCG( "spherically_organized_chains", target_error=0.1, radius=2.0,
no_of_chains=2, chain_lengths=50, no_of_chain_seeds=100,
seed_sigma=scipy.eye(3), target_angle=180.0, speed=1.0,
total_samples=100000 )

WMCG( "markov_chain", target_error=0.05, no_of_chains=12, chain_lengths=5,
seed_sigma=scipy.eye(2)*2, target_angle=180.0, speed=1.0,
longitudinal_density=100.0, test_samples=1000,
fix_speed=0, fix_angle=0, resample=1 )
```

D.3 Transformations

The generators do not allow to control statistical parameters like outer covariance, data centroid or intrinsic dimensionality directly. Therefore, in Chapter 4.3 we have proposed to use specific transformations:

`EMBED( "source_name", "target_name", dims=100 )`
- embed data into 100 dimensions

`RSCALE( "source_name", "target_name", sigma=1.0 )`
- randomly scale features according to a log-normal distribution parameterized by `sigma`

```
RMOVE( "source_name", "target_name", scale=0.1, move_type="gauss" )
RMOVE( "source_name", "target_name", scale=2.0, move_type="uniform"
)
```
- randomly translate the centroid of the dataset

`RROT( "source_name", "target_name" )`
- randomly rotate, no parameters

`RSHEAR( "source_name", "target_name", sigma_list=[1.2, 0.3, 0.1] )`
- example of random shearing along three random dimensions

`NORMCOV( "source_name", "target_name", cov_matrix=eye(5) )`
- apply outer covariance and make sure the dimensions of the matrix fit the data

`CENTER( "source_name", "target_name", new_centroid=zeros(5) )`
- move centroid to specified location

Source name and target name are mandatory parameters. Source name must be a dataset that has been generated before and target name can be a new token or a previously defined token. If an old token is reused, then the previously stored data under this token gets inaccessible.

D.4 Generating & Exporting Data

Calling the *Micro Code* does not generate any data. It only checks, whether the provided parameters are possible and whether the sequences of data reuse are legitimate. It is a frequent error that (especially in python) a token is requested that hasn't been declared, yet. Since data generation can take extensive amount of time it is favorable to check the execution plan in advance. This happens during calls like `WGKS`, `EMBED`, `RROT` etc. Once the declarations are finished the call to `EXEC()` will execute the whole sequence.
During execution the data is placed into the module variable `generators.data_sets`. After calling `EXEC()` the generated data can be stored using `generators.save_data_sets( "some_filename.shelve" )` and retrieve it by using `generators.restore_data_sets( "some_filename.shelve" )`.
Data can be exported to specific formats by calling `generators.export_data_sets()`. A list of names must supplied and the export-backend. Depending on the backend the filenames are chosen automatically. E.g. a dataset with filename "x" will be saved with the csv exporter in three or four files of the kind "x.vectors.csv", "x.labels.csv" and "x.probabilities.csv" (and "x.skeletton.csv", if applicable). Some calling examples are:

`export_data_sets( "some_set" )`

`export_data_sets( [ "set1", "set2", "set3" ] )`

`export_data_sets( "setX", matlab_exporter )`

`export_data_sets( "setY", shelve_exporter )`

`export_data_sets( "setZ", numpy_exporter )` default exporter

`export_data_sets( [ "set4", "set5", "set6" ], csv_exporter )`

`export_data_sets( [ "set4", "set5", "set6" ] )`

`export_data_sets( exporting_function=csv_exporter )`
this will export all datasets

Available memory can limit the number of datasets that can be created. Since often the datasets generated in the past are not required for future datasets you can use the `FLUSH()` call in order to indicate that memory shall be freed. This command is stored in the Micro Code and will be called repeatedly when program is reexecuted.
Your code can be parallized via `TRANSACTION()` calls. All the commands between those calls are considered *atomic*.
Calling `CLEAN()` will delete current program on the stack.

D.5 Program Proliferation

If your experiment relies on great amount of data it can be cumbersome or even impossible to share your data with other researchers. However, since the generated data is having a well defined source model it is not necessary to proliferate the full dataset. Instead it is enough to use data with the same characteristics. The DFKI Bayes-True Generator package can do this proliferation easy for you. When you specify a generation program via the *Micro Code* all important aspects of randomness are canceled and your researching colleagues can use own computing resources in order to reproduce the data in place. If you want to exactly reproduce the samples, then set a seed and send it along with the generation program.
There are two ways of program proliferation. The first is to use `generators.save_stack()` and `generators.restore_stack()` functions. This will save the `generators.stack` variable in a shelve file. However, this possibility requires that the receiving partner has the DFKI Bayes-True Generator package on his computer.
In order to make the code proliferation even easier the DFKI Bayes-True Generator package provides a second way of program proliferation. It can compile a self-sufficient python file with all the necessary code to regenerate the data. After the whole generation program has been specified you can call `generators.compile2python()` function and give it the filename of the python generator file. The default name is "generation_script.py". You can call it by invoking the python interpreter:

`python generation_script.py`
`python generation_script.py 4` use four CPU cores (this will only work if transactions were specified!

D.6 Visualization

As long as the generated datasets are in memory, they can be visualized with the `VISUALIZE()` command. The syntax of this function is:

`VISUALIZE( names=None, y_lim=[−10, 10], x_lim=[−10, 10], wait=3 )`
You can supply a list of names or a single dataset name for visualization. Datasets will be shown for as many seconds as specified in *wait*. The graphs are saved into PDF files in the current directory where the filename is always `dataset-name+.PDF`.

D.7 Examples

This example is found in demo1.py:

```
from generators import *

WGKS( "abc", target_error=0.3 )
KWHITES( "kunk", 5, 0.25 )
EMBED( "kunk", "kunk_high", 20 )
RROT( "kunk_high", "kunk_high_rot1" )
CENTER( "kunk_high_rot1", "kunk_high_rot1", scipy.ones(20)*5 )
RSHEAR( "kunk_high_rot1", "kunk_high_rot1" )
RMOVE( "kunk_high_rot1", "kunk_high_rot1" )
NORMCOV( "kunk_high_rot1", "white_kunk", scipy.eye(20) )
FLUSH( "white_kunk" )
TRANSACTION()
GCG( "gongi", target_error=0.1, no_of_chains=2, chain_lengths=5, no_of_chain_seeds=10000,
seed_sigma=scipy.eye(3) )
SCG( "rukutuku", target_error=0.1, radius=10.0, no_of_chains=2, chain_lengths=50
)
WMCG( "wobodobo", target_error=0.4, no_of_chains=20, chain_lengths=5,
seed_sigma=scipy.eye(2)*2, fix_speed=1, fix_angle=1, resample=0 )
RROT( "wobodobo", "wobodobo1" )
FLUSH( "wobodobo1" )
TRANSACTION()
compile2python()
EXEC()
```

Example from demo2.py:

```
from generators import *

for i in range( 20 ):
GCG( "gcg"+str(i), target_error=0.1, no_of_chains=2, chain_lengths=5,
no_of_chain_seeds=10000, seed_sigma=scipy.eye(3) )
RROT( "gcg"+str(i), "gcg_rot_"+str(i) )
FLUSH( "gcg_rot_"+str(i) )
TRANSACTION() #enables parallelization
compile2python()
```

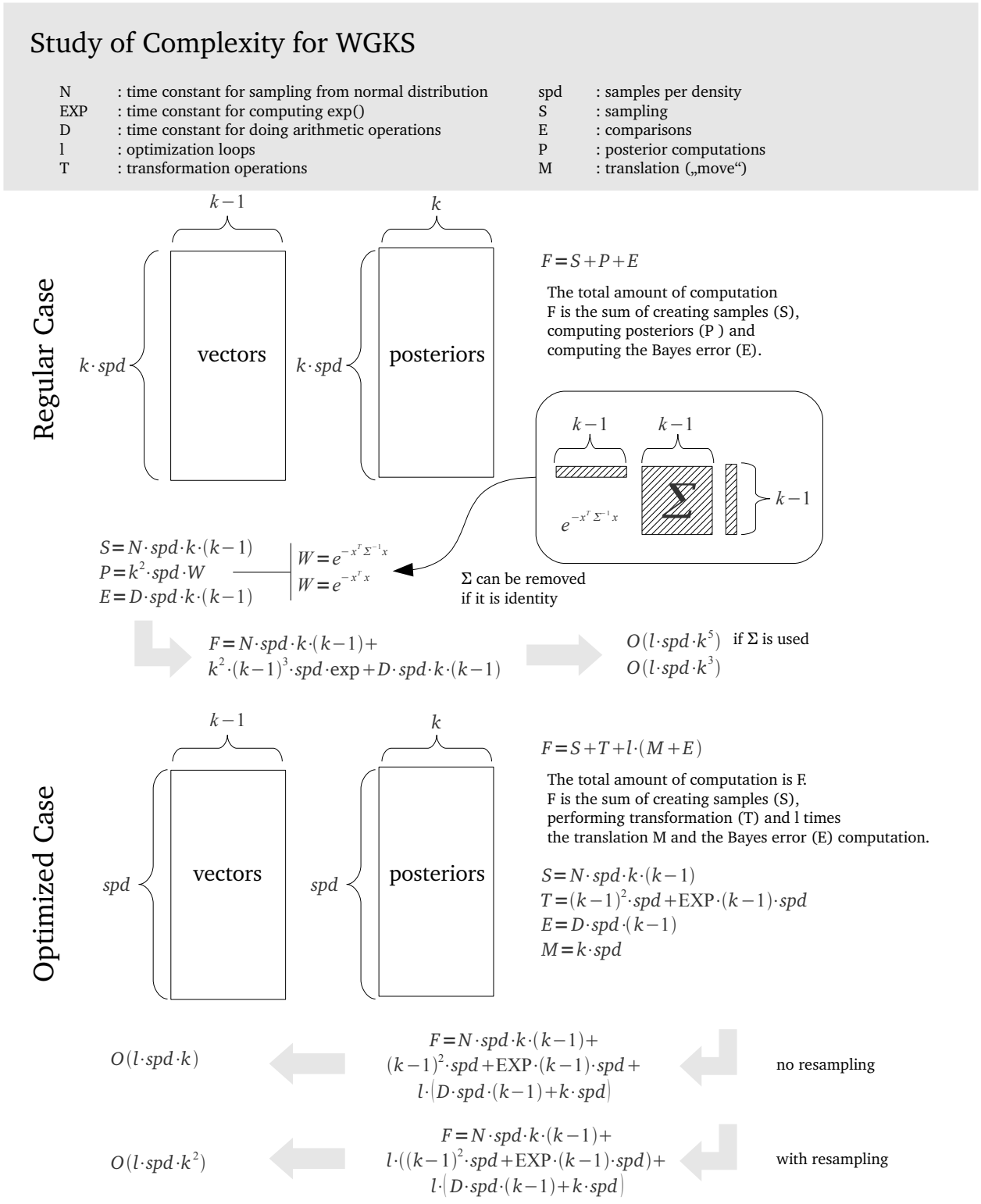


Fig. 12: How speed up is computed.