Scalable learning through linearithmic time kernel approximation techniques

Johan von Tangen Sivertsen

March 24, 2014

Abstract

Random Feature mappings as first suggested in 2007 by Rahimi and Recht provide mappings that closely approximate feature spaces in $O(nDd)$. Two new approaches for random feature mapping were presented in 2013. "Fastfood" replaces the random matrix of Rahimi and Recht with an approximation that uses the Walsh-hadamard transform to gain significant speed and storage advantages. It promises to create accurate mappings for RBF kernels in $O(nD \log d)$ time. Tensor sketching applies recent results in tensor sketch convolution to deliver approximations for polynomial kernels in $O(n(d + D \log D))$ time. This thesis investigates the theoretic foundation of both methods. Providing a point of reference for future users, it highlights several relevant tradeoffs and considerations for putting these methods to use in machine learning. Through independent analysis and experiments it verifies speed and accuracy of the methods and for the first time provides an accessible open-source implementation of both.

Contents

Chapter 1

Introduction

In an increasingly digitally connected world the amount of data collected and generated is undergoing explosive growth with no indication of slowing down[7]. 90% of all data was generated in the last two years[1]. Unfortunately data on its own is of little value. Without powerful tools to analyze and learn from it, the data may never find its way into meaningful and valuable information. This creates a growing demand for fast techniques that uncover complex relationships between thousands of variables.

This thesis takes its offset in kernel methods. They allow linear classification and regression methods to discover non-linear relationships at little extra cost. Unfortunately the Support Vector Machines(SVMs) that traditionally utilize kernel functions scale poorly in the size of the dataset. Recent developments have produced SVMs that scale well in dataset size[10], but cannot take advantage of kernel functions. A promising compromise was presented in 2007 by Ali Rahimi and Ben Recht who suggested using random feature mappings to get the best of both worlds[18]. In 2013 two new papers[13, 16] presented novel new approaches along this line that both deliver significant speed improvements at the cost of a, hopefully, negligible loss of accuracy. This would bridge the gap and provide a method of discovering non-linear relationships between multiple variables in linear time.

This thesis project aims a exploring these new techniques and making them more available through four goals:

- To provide a reference point to allow anybody with an insight into computer programming and linear algebra to understand and use these methods.
- To provide a fast and accessible open source implementation of the techniques to the public.
- To discuss important kernel and data properties and how they effect the accuracy and speed of the methods.
- To independently test the accuracy and speed of the techniques through experiments.

[1]according to IBM research. See `http://www-03.ibm.com/systems/power/software/linux/powerlinux/bigdata.html`

We begin this thesis with a section aimed at grounding the methods in machine learning and expanding on the potential benefits of using them in this context. The second part of the thesis consists of a theoretical breakdown and discussion of the techniques. We first present the idea of random projection mapping as developed by Ali Rahimi and Ben Recht in the Random Kichen Sinks method[18]. The first of the new methods, Fastfood[13], is then presented as a direct extension to Random Kitchen Sinks. We will use the word Fastfood to generally refer to this technique throughout this thesis. The second new technique, tensor sketching[16], is presented next. It is also a random projection technique, but follows a different basic approach than Random Kitchen Sinks. A series of experiments are carried out to determine the accuracy and speed of Fastfood and tensor sketching, and to test the methods in a machine learning scenario. The results of these efforts are presented in the final chapter of the thesis, following a short chapter that details the C++ implementation of the methods created as part of the thesis project.

1.1 The Kernel Trick

We begin by introducing the Kernel Trick along with some general machine learning concepts following the terminology of Bishop[3]. This will provide a basis of discussion throughout the rest of the thesis and demonstrate the usefulness of the random projections presented later.

In many machine learning scenarios we wish to find the parameters of a decision function $f(\boldsymbol{x})$ of the form:

$$f(\boldsymbol{x}) = w_0 + w_1\phi_1(\boldsymbol{x}) + \cdots + w_d\phi_d(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}) \tag{1.1}$$

Such a function is called a linear model, since it represents a linear combination of fixed nonlinear functions ϕ_j, or *features*, of the input variables [3]. Evaluating this function has uses in regression and classification. Learning the parameters of such a function, $\boldsymbol{\omega}$, is a common machine learning task and many popular algorithms like *Support Vector Machines* or *Adaboost* refer to various ways of solving this optimization task[17]. Evaluating the decision function with known parameters is referred to as *testing* while learning the parameters is *training* and generally requires at least $O(n)$ evaluations of the decision function for a data set with n entries.

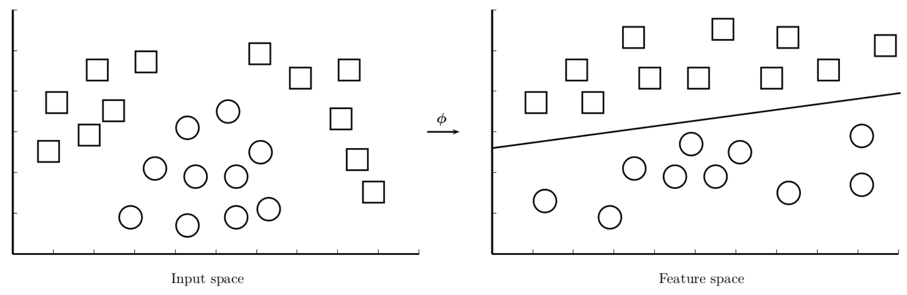


Figure 1.1: Mapping to a feature space where the classes are linearly separable

By using a feature space $\boldsymbol{\phi}$, data that originally had a very complex decision boundary can be mapped to a space where a simple linear separation of the classes exists. The idea is illustrated in fig.1.1. Unfortunately the feature space can often be of very high or even infinite dimensionality, making a direct computation of the mapping undesirable and risking incurring effects of the curse of dimensionality.

However, many learning algorithms do not require actually mapping the input to the feature space $\boldsymbol{\phi}$, but just some notion of the 'difference' between datapoints in this space. They can be reformulated in a *dual representation*:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) = \sum_{n=1}^{N} \alpha_n \langle \boldsymbol{\phi}(\boldsymbol{x_n}), \boldsymbol{\phi}(\boldsymbol{x}) \rangle \tag{1.2}$$

The dual representation relies only on inner products so the feature space could be any Hilbert space. Therefore it is desireable to find functions $k(\boldsymbol{x}, \boldsymbol{y})$ with the property that:

$$k(\boldsymbol{x}, \boldsymbol{x}') = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}') \tag{1.3}$$

The big benefit of using this formulation is that the problem becomes expressed only in terms of $k(\boldsymbol{x}, \boldsymbol{x'})$ and we no longer have to work with $\boldsymbol{\phi}$. Such functions are called *kernel functions* or just kernels. Kernel Methods were first introduced in the field of pattern recognition in 1964[2] but have flourished with their applications in Support Vector Machines introduced in 1995[5]. Their properties are described by Mercers Theorem[14] which guarantees the expansion of kernel functions to Hilbert spaces as in eq.1.3, requiring the kernels to be positive, continuous and semi-definite. The representer Theorem[20] states that the expansion exists under manageable conditions, even when the feature space is of infinite dimensionality. See appendix B for further details. Using kernel functions we can evaluate the decision function as:

$$f(\boldsymbol{x}) = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_n, \boldsymbol{x}) \tag{1.4}$$

Using a dual representation a Support Vector Machine, originally a linear classifier, can discover non-linear relationships. The *kernel trick* refers to the idea of replacing expensive mappings to high dimensional feature spaces, $\boldsymbol{\phi}$, with computationally inexpensive evaluations of a kernel function $k(\boldsymbol{x}, \boldsymbol{x}')$ and learning through the dual representation of a problem.

1.1.1 Common Kernel functions

The choice of kernel function is a matter of the intended application and the data. It is desirable to choose a kernel function that corresponds to a mapping $\boldsymbol{\phi}(\boldsymbol{x})$ into some feature space where the input is easily separable. Two of the most commonly used types of kernel functions are *Radial Basis Function kernels* and *polynomial kernels.*

Definition 1 (RBF kernels)**.** Kernels depending only on the magnitude of the distance between inputs, $k(\boldsymbol{x}, \boldsymbol{x'}) = k(||\boldsymbol{x} - \boldsymbol{x'}||)$, e.g. the Gaussian kernel:

$$k(\boldsymbol{x}, \boldsymbol{y}) = \exp(-\frac{||\boldsymbol{x} - \boldsymbol{y}||^2}{2\sigma^2}) \tag{1.5}$$

Where σ is a free parameter. The corresponding feature space is of infinite dimensionality[3].

Definition 2 (Polynomial Kernels)**.** Polynomials of the inner product of the input.

$$k(\boldsymbol{x}, \boldsymbol{y}) = (\langle \boldsymbol{x}, \boldsymbol{y} \rangle + c)^p \tag{1.6}$$

Where c is a constant. If $c = 0$, we call the kernel homogeneous. The corresponding mapping consists of all p-degree terms of the input. We will see how this fact becomes useful in the tensor sketching approach.

Table 1.1: Common definitions

n	Size of a dataset. The number of entries.
d	Dimensionality of input.
D	Dimensionality of feature space or mapping.
p	Degree of polynomial kernel.
C	Offset of polynomial kernel.
σ	Width of Gaussian RBF kernel.

Kernel Conversion

The Gaussian RBF and the polynomial kernel are related through the property of the inner product:

$$||\boldsymbol{x} - \boldsymbol{y}||^2 = ||\boldsymbol{x}||^2 + ||\boldsymbol{y}||^2 - 2\langle \boldsymbol{x}, \boldsymbol{y} \rangle \tag{1.7}$$

So knowing the length of the involved vectors we can establish a linear time conversion between an RBF kernel and a polynomial kernel.

1.1.2 The curse of support

When using the kernel trick to train a linear model, like in eq.1.2, methods like Support Vector Machines solve a quadratic optimization problem on the Gram matrix K containing all possible $k(x, x')$ values. This can lead to a prohibitive cost if the support is large, that is, if the number of non-zero α_n in 1.4 is large. Unfortunately this is often the case for large scale problems [13] and we come close to a cubic growth in run time for increasing n. As datasets grow to hundreds of thousands of entries, the size of the optimization quickly becomes prohibitive. The problem has been dubbed *The Curse of Support*[11].

The same problem arises when testing such models. Using a decision function learned from a dual representation, see eq.1.2, it is necessary to evaluate $k(x, x')$ for all pairs with the vectors that have non-zero α_n. This has an $O(Nd)$ cost. If we would instead learn $\boldsymbol{w}$ directly the decision function could be evaluated in $O(d)$, see eq.1.1.

In 2006 T. Joachims introduced the first linear time algorithm for training linear SVMs[10]. This is done by reformulating the SVM optimization problem. Unfortunately this benefit is lost if we are still bound to the Gram matrix of the kernel trick.

1.2 A scaling solution

We can now recap and consider the heart of our problem. The well known kernel trick provides a fast way to learn non-linear relations, but it does not scale well as datasets grow large. Fast learning methods exists, but they do not support using the kernel trick. This paper investigates two approaches that try to bridge the gap. This would provide a method for training linear models that scales well in the size of the dataset and allow the use of feature spaces to capture non-linear relations. Both take their offset from the idea of approximating kernel functions through random mappings to inner product spaces.

Using random features

Ali Rahimi and Ben Recht in 2007 suggested an alternative to the kernel trick[18, 17]. Instead of using the kernel function directly they propose explicitly mapping data to a low dimensional inner product space using a randomized feature map $Z : \mathbb{R}^d \to \mathbb{R}^D$ such that:

$$E[\langle \boldsymbol{z}(\boldsymbol{x}), \boldsymbol{z}(\boldsymbol{x'}) \rangle] = k(\boldsymbol{x}, \boldsymbol{x'}) = \langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x'}) \rangle \tag{1.8}$$

Having such a mapping with good bounds we no longer need to learn through the Gram matrix. We can work directly with a linear decision function:

$$f(\boldsymbol{x}) = w_0 + w_1 z_1(\boldsymbol{x}) + \cdots + w_D z_D(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{z}(\boldsymbol{x}) \tag{1.9}$$

This decision function can be evaluated in $O(D)$ time, the complexity depends only on the dimensionality of the space $\boldsymbol{z} \in \mathbb{R}^D$.

In other words a good mapping $\boldsymbol{z}$ would lift both the curse of dimensionality from using high or infinite dimensional $\boldsymbol{\phi}$ mappings and the curse of support from using kernels. In the rest of the thesis we investigate two recent techniques for constructing such mappings. In the end we return to the full machine learning scenario and put the results to use in a Support Vector Machine.

Chapter 2

Random Feature mappings

2.1 Random Kitchen Sinks

The first technique has been dubbed Fastfood[13] and is a direct extension of the Random Kitchen Sinks framework introduced by Ali Rahimi and Ben Recht[18],[17]. In this section we introduce the Random Kitchen Sink application to approximating shift invariant kernels, in the next we look at the Fastfood extension. Recall that shift invariant kernels consider only the distance between the inputs, and we can generally rewrite them as single input functions $k(\boldsymbol{x},\boldsymbol{y}) = k(\boldsymbol{x}-\boldsymbol{y})$ like e.g. the Gaussian RBF kernel:

$$k(\boldsymbol{x}-\boldsymbol{y}) = \exp(-\frac{||\boldsymbol{x}-\boldsymbol{y}||^2}{2\sigma^2}) \tag{2.1}$$

So the purpose of the method will be to construct a mapping $\boldsymbol{z(x)} = \boldsymbol{x'}$ such that $\langle \boldsymbol{x'}, \boldsymbol{y'} \rangle = e^{-\frac{||\boldsymbol{x}-\boldsymbol{y}||^2}{2\sigma^2}}$.

2.1.1 Bochner's Theorem

The mathematical foundation for the Random Kitchen Sink method is Bochner's Theorem.

Theorem 2.1.1 (Bochner's theorem[1])**.** *Every positive definite function is the Fourier transform of a positive measure. This implies that for any shift invariant kernel $k(\boldsymbol{x}-\boldsymbol{y})$, there exists a positive measure μ such that the kernel is the Fourier transform of that measure:*

$$k(\boldsymbol{x}-\boldsymbol{y}) = \int_{\mathbb{R}^d} \mu(\boldsymbol{\omega}) e^{-i\langle \boldsymbol{\omega}, (\boldsymbol{x}-\boldsymbol{y})\rangle} d\boldsymbol{\omega} \tag{2.2}$$

and the measure $p = \frac{\mu(\boldsymbol{\omega})}{\int \mu(\boldsymbol{\omega})}$ is a probability measure.

If the kernel $k(\boldsymbol{x}-\boldsymbol{y})$ is properly, scaled Bochner's theorem guarantees that its Fourier transform $p(\boldsymbol{\omega})$ is a proper probability distribution[18].

[1]The theorem was first published in 1932, the version given here is based on [8],[6],[18]

This gives rise to the central step in the Random Kitchen Sinks method. Looking at Bochner's theorem we are given a measure $p(\boldsymbol{\omega})$ that assign weights or probabilities to each $\boldsymbol{\omega} \in \mathbb{R}^d$. Since $p(\boldsymbol{\omega})$ is a probability distribution $\int p(\boldsymbol{\omega}) = 1$ and we would expect that $k(\boldsymbol{x}-\boldsymbol{y}) \approx e^{-i\langle \boldsymbol{\omega}_\mu, (\boldsymbol{x}-\boldsymbol{y})\rangle}$ if $\boldsymbol{\omega}_\mu$ is the average of $p(\boldsymbol{\omega})$. Since $E[p(\boldsymbol{\omega})] = \boldsymbol{\omega}_\mu$ we can estimate the kernel by $k(\boldsymbol{x}-\boldsymbol{y}) = E_{\boldsymbol{\omega}}[e^{-i\langle \boldsymbol{\omega}, (\boldsymbol{x}-\boldsymbol{y})\rangle}]$ where $\boldsymbol{\omega}$ is sampled from the $p(\boldsymbol{\omega})$ distribution. We can improve the estimate by drawing multiple samples from the distribution $\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \cdots, \boldsymbol{\omega}_D \sim p(\boldsymbol{\omega})$ and averaging their evaluations:

$$E\left[\frac{1}{D}\sum_{j=1}^{D} e^{i\langle \boldsymbol{\omega}_j, (\boldsymbol{x}-\boldsymbol{y})\rangle}\right] = k(\boldsymbol{x}-\boldsymbol{y}) \tag{2.3}$$

Since both $p(\boldsymbol{\omega})$ and $k(\boldsymbol{x}-\boldsymbol{y})$ are real we can expect the imaginary part of the approximation to have no contribution and $e^{i\langle \boldsymbol{\omega}_j, (\boldsymbol{x}-\boldsymbol{y})\rangle}$ can be replaced by $\cos \boldsymbol{\omega}_j^T(\boldsymbol{x}-\boldsymbol{y})$.

$$\frac{1}{D}\sum_{j=1}^{D} \cos \boldsymbol{\omega}_j^T(\boldsymbol{x}-\boldsymbol{y}) \tag{2.4}$$

We can rewrite this form using the sum of angles rule: $\cos \boldsymbol{\omega}_j^T(\boldsymbol{x}-\boldsymbol{y}) = \cos(\boldsymbol{\omega}_j^T\boldsymbol{x})*\cos(\boldsymbol{\omega}_j^T\boldsymbol{y})+\sin(\boldsymbol{\omega}_j^T\boldsymbol{x})*\sin(\boldsymbol{\omega}_j^T\boldsymbol{y})$. By setting $z_j(x) = \{\cos \boldsymbol{\omega}_j^T\boldsymbol{x}, \sin \boldsymbol{\omega}_j^T\boldsymbol{x}\}$ we can express the sum as a sum of dot products:

$$E\left[\frac{1}{D}\sum_{j=1}^{D} z_j(x)^T z_j(y)\right] = k(\boldsymbol{x}-\boldsymbol{y}) \tag{2.5}$$

So to build a mapping $\boldsymbol{z}(\boldsymbol{x})$ we normalize with respect to the number of samples D. $\boldsymbol{z}(x)$ consist of D entries where:

$$z_j(x) = \frac{1}{\sqrt{D}}\{\cos \boldsymbol{\omega}_j^T x, \sin \boldsymbol{\omega}_j^T x\} \tag{2.6}$$

The resulting map fulfills our purpose. It approximates the kernel space in D dimensions according to the number of samples drawn. $E[\langle \boldsymbol{z}(x), \boldsymbol{z}(y)\rangle] = k(\boldsymbol{x}-\boldsymbol{y})$. We can use Random Kitchen Sinks to find a low dimensional random feature space that approximates any shift-invariant kernel.

2.1.2 Alternative mappings

We may also follow a more direct route from eq.2.3 by splitting the sum to get:

$$\frac{1}{D}\sum_{j=1}^{D} e^{i\boldsymbol{\omega}_j(\boldsymbol{x}-\boldsymbol{y})} = \frac{1}{D}\sum_{j=1}^{D} e^{i\boldsymbol{\omega}_j(\boldsymbol{x})}\overline{e^{i\boldsymbol{\omega}_j(\boldsymbol{y})}} \tag{2.7}$$

Looking at a single term of the sum:

$$\begin{aligned} e^{i\boldsymbol{\omega}_j(\boldsymbol{x})}\overline{e^{i\boldsymbol{\omega}_j(\boldsymbol{y})}} &= (\cos\langle \boldsymbol{\omega}_j, \boldsymbol{x}\rangle + i*\sin\langle \boldsymbol{\omega}_j, \boldsymbol{x}\rangle) * (\cos\langle \boldsymbol{\omega}_j, \boldsymbol{y}\rangle - i*\sin\langle \boldsymbol{\omega}_j, \boldsymbol{y}\rangle) \\ &= \cos\langle \boldsymbol{\omega}_j, \boldsymbol{x}\rangle \cos\langle \boldsymbol{\omega}_j, \boldsymbol{y}\rangle + \sin\langle \boldsymbol{\omega}_j, \boldsymbol{x}\rangle \sin\langle \boldsymbol{\omega}_j, \boldsymbol{y}\rangle + img \\ &= z_j(x)^T z_j(y) \end{aligned} \tag{2.8}$$

As before the imaginary part, *img*, is expected to give zero contribution. Looking at 2.8 we see that if we take the complex conjugate of one of the mappings when evaluating the inner product the mapping in 2.6 is equivalent to a mapping $\boldsymbol{z'}$ with D entries where:

$$z'_j(x) = \frac{1}{\sqrt{D}} e^{i\boldsymbol{\omega}_j x} \tag{2.9}$$

We show both of these mappings here because the $\boldsymbol{z}$ and $\boldsymbol{z'}$ forms are used interchangeably in [13] and [18] but do have an important difference in the latter requiring a conjugation when taking inner products. Ali Rahimi and Ben Recht further suggest using a mapping:

$$z''_j(x) = \frac{1}{\sqrt{D}} \sqrt{2} \cos(\boldsymbol{\omega_j^t} x + b) \tag{2.10}$$

where b is drawn uniformly from $[0, 2\pi]$ [18]. In our implementation we have used the $\boldsymbol{z}$ mapping. It has a clear benefit over the z' mapping in that it can be implemented without the use of complex numbers. However, the $\boldsymbol{z}$ mapping requires storing both the sine and cosine for each sample $\boldsymbol{\omega}$, effectively doubling the feature dimensionality D. The $\boldsymbol{z''}$ mapping avoids this cost but unfortunately little mathematical evidence is provided in favor of this mapping[2] and experiments also showed the $\boldsymbol{z}$ mapping to be more accurate in practice. See p.30.

2.1.3 Matrix representation

The main computations in these mappings are the dot products like $\boldsymbol{\omega}_j^T \boldsymbol{x}$. To perform the mapping we construct a matrix $\Omega \in \mathbb{R}^{D \times d}$ so that these can more easily be found through matrix multiplication:

$$\Omega \boldsymbol{x} = \begin{bmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,d} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{D,1} & \omega_{D,2} & \cdots & \omega_{D,d} \end{bmatrix} * \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} = \begin{bmatrix} \boldsymbol{\omega_1^T x} \\ \vdots \\ \boldsymbol{\omega_D^T x} \end{bmatrix} \tag{2.11}$$

Using the Ω matrix we can map the data in $O(Dd)$ using:

$$z_j(x) = \frac{1}{\sqrt{D}} \{\cos([\Omega \boldsymbol{x}]_j), \sin([\Omega \boldsymbol{x}]_j)\} \tag{2.12}$$

Or for the $\boldsymbol{z'}$ map:

$$z'_j(\boldsymbol{x}) = \frac{1}{\sqrt{D}} \exp(i[\Omega \boldsymbol{x}]_j) \tag{2.13}$$

The matrix representation makes it clear how mapping can be performed in $O(Dd)$ through matrix multiplication.

[2]There exist at least two different version of the 2007 paper, putting different emphasis on this mapping.

2.1.4 Sampling Ω

To use the method we need to sample the entries in Ω from a distribution that corresponds to the chosen kernel. According to Bochner's theorem we can use the inverse Fourier Transform to find the sample distribution matching a given shift invariant kernel.

2.1.5 Gaussian kernel

Returning to the Gaussian RBF kernel we set the bandwidth $\sigma = 1$.

$$k(\boldsymbol{x} - \boldsymbol{y}) = e^{-\frac{||\boldsymbol{x}-\boldsymbol{y}||^2}{2}} \tag{2.14}$$

Bochner's theorem guarantees the existence of a measure $p(\boldsymbol{\omega})$ such that:

$$\int_{\mathbb{R}^d} p(\boldsymbol{\omega}) e^{(-i\langle \boldsymbol{\omega}, (\boldsymbol{x}-\boldsymbol{y})\rangle)} = e^{-(\frac{||\boldsymbol{x}-\boldsymbol{y}||^2}{2})} \tag{2.15}$$

So to find $p(\boldsymbol{\omega})$ we apply the inverse Fourier Transform to the kernel:

$$p(\boldsymbol{\omega}) = \frac{1}{\sqrt{(2\pi)^d}} \int_{\mathbb{R}^d} k(\boldsymbol{x} - \boldsymbol{y}) e^{\langle \boldsymbol{\omega}, (\boldsymbol{x}-\boldsymbol{y})\rangle} d(\boldsymbol{x} - \boldsymbol{y}) = (2 * \pi)^{-\frac{d}{2}} e^{-\frac{||\boldsymbol{\omega}||}{2}} \tag{2.16}$$

The sampling distribution for $\boldsymbol{\omega}$ is a Gaussian function. The same as given in the original paper[18]. However, if we rewrite it slightly we see this corresponds to the regular form of the normal distribution for $\sigma = 2\pi^{\frac{d-1}{2}}$.

$$p(\boldsymbol{\omega}) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{||\boldsymbol{\omega}||^2}{2}}, \sigma = 2\pi^{\frac{d-1}{2}} \tag{2.17}$$

If we choose another bandwidth for the Gaussian RBF kernel we adjust the variance of the distribution accordingly.

2.1.6 Error Bound

Hoeffdings inequality guarantees a fast convergence between the kernel and the approximation[18]:

$$Pr[|\langle \boldsymbol{z}(x), \boldsymbol{z}(\boldsymbol{y})\rangle - k(\boldsymbol{x}, \boldsymbol{y})| \geq \epsilon] \leq 2 \exp \frac{-D\epsilon^2}{4} \tag{2.18}$$

2.2 Fastfood

The key idea of Fastfood is to replace the Ω matrix of Random Kitchen Sinks method with another matrix, V. Specifically one with properties that allow for very fast vector multiplication while still delivering a mapping comparable to using Ω. The Fast Walsh Hadamard Transform (FWHT) allows us to multiply with a Hadamard matrix in log-linear time[9]. This allows us to exploit the fact that Hadamard matrices, when combined with diagonal Gaussian matrices, have properties very similar to dense Gaussian matrices, like Ω[13].

2.2.1 Matrix approximation

Recall that the matrix Ω from Random Kitchen Sinks is a $D \times d$ Gaussian random matrix. d is the dimensionality of the input space and each $\boldsymbol{\omega}_{i \in \{1..D\}}$ is an independent sample from a d dimensional Gaussian. To approximate the Ω matrix we construct V:

$$\Omega \approx V = \frac{1}{\sigma\sqrt{d}} SHG\Pi HB \tag{2.19}$$

Where $d = 2^l$ for some $l \in \mathbb{N}$.

The construction looks complex but the underlying idea is simple. A number of samples are drawn and then combined through a series of permutation, transforms and scaling to form a matrix which resembles Ω. To see how the process works we will examine the components used to form V.

Components

The matrix V is a product of several matrices:

Diagonal random matrices:
The matrices S, G and B are diagonal random matrices. B has random ± 1 values on its main diagonal, G has random Gaussian entries and S is a random scaling matrix. All of these are computed once and then stored.

A permutation matrix:
The Π matrix is a random permutation matrix $\Pi \in \{0,1\}^{dxd}$. It can be implemented as a lookup table by sorting random numbers.

The Walsh-Hadamard matrix:
H is the Walsh-Hadamard matrix:

Definition 3 (Walsh-Hadamard Matrix)**.**
$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ and $H_{2d} = \begin{bmatrix} H_d & H_d \\ H_d & -H_d \end{bmatrix}$
The matrix is only defined for powers of 2. With a normalization factor $d^{-\frac{1}{2}}$ it forms the Walsh-Hadamard Transform[9].

2.2.2 Constructing the approximation

Sampling and transforming

The random Kitchen Sinks method rely on samples from a distribution matching the kernel of interest. In the Fastfood method we use a combination of samples from a normal distribution $\mathcal{N}(0,1)$ on the diagonal of G and samples from a spectrum depending on the kernel of interest on the diagonal of S[13].

The Walsh-Hadamard transform

The Walsh-Hadamard transform $\frac{1}{\sqrt{d}}H$ is closely related to the Fourier Transform[12, 9]. It decomposes an arbitrary vector $\boldsymbol{x} \in \mathbb{R}^d$ into a superposition of Walsh functions. This makes a sparse input vector more dense[1]. In Fastfood it rotates the Gaussians of G resulting in a denser matrix more similar to a fully random Gaussian matrix, like Ω. Equation 2.20 shows an example for a $d = 2, HG$ computation.

$$HG = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} g_1 & 0 \\ 0 & g_2 \end{bmatrix} = \begin{bmatrix} g_1 & g_2 \\ g_1 & -g_2 \end{bmatrix} \tag{2.20}$$

This is done twice through $\frac{1}{\sqrt{d}}HG\Pi H$. The permutation matrix Π ensures that the order of the decomposition is scrambled so no single G_{ij} gets too influential.

A longer chain of Walsh-Hadamards and permutations will bring the result even closer to a fully random Gaussian matrix, but it has been shown that two steps are enough to provide a sufficient amount of decorrelation[13].

Disregarding the normalization, this forms a matrix G':

$$G' = HG\Pi HB \tag{2.21}$$

Each entry G'_{ij} is a result of adding and subtracting zero-mean independent Gaussian random variables. Since sign changes retain Gaussian properties, each entry in G' is a zero-mean Gaussian. The B matrix ensures that there is no correlation between the entries of each row, thus any row of G' is i.i.d. Gaussian. The entries can be calculated by $G'_{ij} = B_{jj}H_i^T G\Pi H_j$[13].

Scaling

The permutation and sign changes due to Π and B do not affect the length of the rows:

$$\begin{aligned} ||G'_i||^2 &= [HG\Pi HB(HG\Pi HB)^T]_{ii}, i \in [d] \\ &= [HGH(HGH)^T]_{ii} \\ &= ||G||^2_{Frob} d \end{aligned} \tag{2.22}$$

Because of the nature of the Hadamard transform all rows end up having the same length. One way to see this is to consider the first HG which will always yield a matrix with all elements from G on each row. No matter the permutation and sign inversion to the second H it will maintain the same combination of pairs with same or different signs on each column. The result when combined with the first HG will be rows with d elements. Each element will square to $||G||^2_{frob} + k$ and the k's will always sum to zero. This is illustrated in fig.2.1

$$HGH_1 = \left[\begin{bmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} & \mathbf{d} \\ a & -b & c & -d \\ a & b & -c & -d \\ a & -b & -c & d \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & 1 \\ \mathbf{1} & \mathbf{-1} & \mathbf{1} & \mathbf{-1} \\ \mathbf{1} & \mathbf{1} & \mathbf{-1} & \mathbf{-1} \\ 1 & -1 & -1 & 1 \end{bmatrix}\right]_1$$

$$= \{(..+b+c)(..-b+c)(..+b-c)(..-b-c)\}$$

$$||HGH_1||^2 = (a^2+b^2+c^2+d^2)d + k = ||G||^2_{frob}d$$

Figure 2.1: **Properties of HGH** Looking across the colums of the hadamard we see that it contains the same number of adding and subtracting combinations. When squaring the result the first and fourth column will cause additional $2bc$ terms and the second and third $-2bc$ terms. In this way k always sums to zero. This property is not affected by permutation of the rows or sign inversion along the columns.

All the rows of $HG\Pi HB$ have the same length and by scaling with $||G||^{-1}_{frob}d^{-\frac{1}{2}}$ we can get length 1 rows.[3] This is an undesired property since we want the matrix to behave like the entries of a Gaussian matrix and this property is a sign of row correlation. Having correlated rows means that the map different entries $z_j(\boldsymbol{x})$ will be correlated. Take the extreme case where each row in V' is the same. The result is that each $z_j(x)$ is the same and this corresponds to only drawing a single $\boldsymbol{\omega}$ when approximating the integral given by Bochner's theorem. To decorrelate the rows we use the scaling matrix S. Using S we can adjust the V matrix depending on the kernel of interest. In this case we are focused on the Gaussian RBF kernel, but we may choose any RBF kernel[13].

For the Gaussian RBF Le et al.[13] suggest sampling entries in S from a distribution:

$$[13](2\pi)^{\frac{d}{2}} A^{-1}_{d-1} r^{d-1} e^{\frac{r^2}{2}} \tag{2.23}$$

While this solution appears elegant it also provides for an unnecessarily complex implementation. Since the purpose in this case is to simulate the behavior of a d dimensional vector with each element sampled from $\mathcal{N}(0,1)$ we implement the scaling matrix as:

```
Require: $G^{-1}_{frob}, N \leftarrow \mathcal{N}(0,1)$
Ensure: Scaling matrix diagonals $S_i$
1: for all $i \leq d$ do
2:     $Length \leftarrow 0$
3:     for all $j \leq d$ do
4:         $Length \leftarrow Length + N.sample()^2$
5:     end for
6:     $S[i] \leftarrow \sqrt{(Length)} * G^{-}_{frob}1$
7: end for
8: return $S$
```

[3][13] is very brief on this point and also suggest scaling with $||G||^{-\frac{1}{2}}_{frob}d^{-\frac{1}{2}}$. We have attempted to contact the authors of the paper but with no luck. We have tested both approaches and it appears to be a typo in the original text.

While this means taking (d^2) samples instead of just d it only needs to be done once when initializing the mapping. The benefit is that the implementation is much simpler and rely on a standard normal distribution already available in most programming languages.

Normalization

Finally the matrix is normalized to fit the width of the kernel, σ, and the dimensionality of the input space, d.

$$\frac{1}{\sigma\sqrt{d}} \tag{2.24}$$

This last step can be added to the S matrix implementation to reduce the run time complexity of the algorithm

Stacking

Since V is a square $d \times d$ matrix it cannot directly replace Ω which is $D \times d$. Normally we will want have many more samples than dimensions of the input space $D \gg d$. To do this we will need to form D/d instances of V and stack them to achieve the desired $D \times d$ size:

$$V^T = [V_1, V_2, ..., V_{D/d}] \tag{2.25}$$

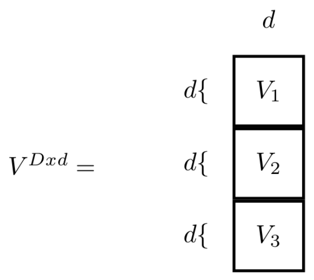


Figure 2.2: Building V in the case where D=3d.

The resulting matrix V will have properties similar to Ω but entails several computational advantages.

2.2.3 Computational advantages

Using the Fast Walsh-Hadamard Transform it is possible to compute each $H_i\boldsymbol{x}$ in $O(d \log d)$ for any vector $\boldsymbol{x} \in \mathbb{R}^d$ [1]. For D/d blocks the total time for matrix-vector multiplication $H\boldsymbol{x}$ becomes $O(D \log d)$. The remaining matrix multiplications in V can all be carried out in linear time in D. It is easy to see for the diagonal matrices G, S, B and can be achieved for the Π matrix by implementing it as a lookup table[13]. This adds a total storage and operation cost of $4D$ for multiplication. The total operation count of Fastfood is $O(D \log d)$ and it uses $O(D)$ storage.

This is a significant improvement of the $O(Dd)$ CPU and storage cost of Random Kitchen Sinks.[4]

2.2.4 Feature map

The feature map is the same as derived in Random Kitchen Sinks, eq2.6, but now replacing the Ω matrix with V.

$$z_j(\boldsymbol{x}) = \frac{1}{\sqrt{D}}\{\cos{[V\boldsymbol{x}]_j}, \sin{[V\boldsymbol{x}]_j}\} \tag{2.26}$$

This would have the same desirable property: $\langle \boldsymbol{z}(\boldsymbol{x}), \boldsymbol{z}(\boldsymbol{y'}) \rangle = e^{-\frac{||\boldsymbol{x}-\boldsymbol{y}||^2}{2\sigma^2}}$. By changing way we sample the S matrix we can make the mapping match any RBF kernel[13].

2.2.5 Error Bound

The error bound for Fastfood closely follows the one given for Random Kitchen Sinks in eq.2.18. The approximation error of a single $d \otimes d$ block has been shown to be within a factor $O(\sqrt{\log d/\delta})$ of Random Kitchen Sinks for a given error probability δ. See theorem 6[13]. We will not go into further proofs here, but the expected inverse relationship between the error and D is confirmed experimentally in section 4.1.2.

[4]If $D \ll d$ the Walsh-Hadamard transform will still bound Fastfood to $O(d \log d)$ time and will make the approximation slower than the exact calculation if $D < \log d$). For practical uses of Fastfood $D \gg d$.

2.3 Tensor Sketching

The second technique we will explore is based on tensor sketching. Tensor sketching in turn relies on fast convolution of Count Sketches[15] to approximate polynomial kernels[16]. We first look at using the tensor product directly as a mapping for the polynomial kernel. We then introduce a version of the Count Sketch algorithm. Finally we show how Count Sketch can be used make fast and accurate sketches of the tensor product using the fast convolution method of Pagh[15]. Such sketches fulfill the purpose of approximating kernel values, but here for the polynomial kernel. End the end of the section we will have a mapping $\boldsymbol{z}$ such that:

$$E[\langle \boldsymbol{z}(\boldsymbol{x}), \boldsymbol{z}(\boldsymbol{x'})\rangle] = (\langle \boldsymbol{x}, \boldsymbol{x'}\rangle + c)^p \tag{2.27}$$

2.3.1 The tensor product and the polynomial kernel

Definition 4 (Tensor Product)**.** Given a vector $\boldsymbol{x} \in \mathbb{R}^d$ its 2-level tensor product is defined as:

$$\boldsymbol{x}^{(2)} = \boldsymbol{x} \otimes \boldsymbol{x} = \begin{bmatrix} x_1x_1 & x_1x_2 & \cdots & x_1x_d \\ x_2x_1 & x_2x_2 & \cdots & x_2x_d \\ \vdots & \vdots & \ddots & \vdots \\ x_dx_1 & x_dx_2 & \cdots & x_dx_d \end{bmatrix} \in \mathbb{R}^{d^2} \tag{2.28}$$

Here written as a matrix, but we normally interpret the tensor product as a vector. The general p-level tensor product:

$$x^{(p)} = x \otimes ... \otimes x, \text{ p - 1 times} \tag{2.29}$$

is a mapping $\mathbb{R}^d \to \mathbb{R}^{d^p}$. It is similar to a Cartesian product, but ordered and with multiplication between the entries in each tuple. We introduce a general notation along those lines:

$$x^{(p)} = \bigotimes_{i_1,..,i_p=1}^{d} \prod_{j \in \{i_1,\cdots,i_p\}} x_j \tag{2.30}$$

The big $\otimes$ is understood to provide p indices between 1 and d, the multiplication part provides the product of the corresponding entries of $\boldsymbol{x}$. The resulting vector will be $x_1^p, x_1^{p-1}x_2, \ldots, x_d^p$. In this way the tensor product contains all possible p-degree ordered multiples of the elements of $\boldsymbol{x}$. This makes it an explict mapping for the p-degree polynomial kernel as stated in definition 2.

Lemma 2.3.1. *Given two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$ the inner product of their p-level tensor products is exactly equal to the value returned by the homogeneous polynomial kernel:*

$$\langle \boldsymbol{x}^{(p)}, \boldsymbol{y}^{(p)}\rangle = \langle \boldsymbol{x}, \boldsymbol{y}\rangle^p \tag{2.31}$$

Proof. Given two vectors $\boldsymbol{x} = x_1, x_2, ..., x_d$ and $\boldsymbol{y} = y_1, y_2, ..., y_d$. Their p-level tensor products are :

$$\boldsymbol{x}^{(p)} = \bigotimes_{i_1,..,i_p=1}^{d} \prod_{j \in \{i_k,\cdots,i_p\}} x_j \tag{2.32}$$

and

$$\boldsymbol{y}^{(p)} = \bigotimes_{i_1,..,i_p=1}^{d} \prod_{j\in\{i_k,\cdots,i_p\}} y_j \tag{2.33}$$

We can directly write the inner product between them as:

$$\langle \boldsymbol{x}^{(p)}, \boldsymbol{y}^{(p)} \rangle = \sum_{i_1,\ldots,i_p=1}^{d,\cdots,d} (\prod_{j\in\{i_1,\cdots,i_p\}} x_j) * (\prod_{j\in\{i_1,\cdots,i_p\}} y_j) \tag{2.34}$$

$$\langle \boldsymbol{x}^{(p)}, \boldsymbol{y}^{(p)} \rangle = \sum_{i_1,\ldots,i_p=1}^{d,\cdots,d} (\prod_{j\in\{i_1,\cdots,i_p\}} x_j y_j) \tag{2.35}$$

Change the order of evaluation.

$$\langle \boldsymbol{x}^{(p)}, \boldsymbol{y}^{(p)} \rangle = \prod^{p} \sum_{i=1}^{d} x_i y_i$$

$$= (\sum_{i=1}^{d} x_i y_i)^p = \langle \boldsymbol{x}, \boldsymbol{y} \rangle^p \tag{2.36}$$

□

Lemma 2.3.2. *By appending a constant $\sqrt{c}$ to a vector $\boldsymbol{x} \in \mathbb{R}^d$ the tensor sketch of $\boldsymbol{x}$ can provide an explict mapping for any polynomial kernel.*

$$\langle \boldsymbol{x}^{(p)}, \boldsymbol{y}^{(p)} \rangle = (\langle \boldsymbol{x}, \boldsymbol{y} \rangle + c)^p \tag{2.37}$$

Proof. appending $\sqrt{c}$ to the end of x and y in the previous proof we arrive at the statement:

$$= (\sum_{i}^{d+1} x_i y_i)^p = (\sum_{i}^{d} x_i y_i + \sqrt(c) * \sqrt(c))^p = (\langle \boldsymbol{x}, \boldsymbol{y} \rangle + c)^p \tag{2.38}$$

□

So the tensor product provides a mapping $\boldsymbol{z}(\boldsymbol{x})$ allowing us to work directly in a feature space corresponding to any polynomial kernel. Since the tensor product belongs to $\mathbb{R}^{d^p}$ it is not a scalable approach however. Calculating the mapping directly takes $O(Nd^p)$ for a dataset of N points in $\mathbb{R}^d$. For cases with low dimensional data and using a polynomial kernel of low degree, the tensor product approach scales well in the number of points. However if d or p are large a direct tensor product mapping quickly becomes computationally more expensive than using the kernel trick and suffering the curse of support. To improve on this we can use recently introduced methods[15],[4] of approximating the tensor product. These methods make use of the Count Sketch algorithm first introduced by Chrikar et. al. in 2011[4].

2.3.2 Count Sketch

Count sketch is an algorithm for approximating counts of high frequency elements in a stream. It maintains two families of t hash functions. One family $h_t(x) : \mathbb{N} \to [b]$ and another $s_t(x) : \mathbb{N} \to \pm 1$. For storage it uses an array of t hash tables, each containing b buckets. Each item x encoutered on the stream is run through the t $h(x)$ and $s(x)$ functions. Each $h(x)$ provides an index and each $s(x)$ indicates whether to increment or decrement the counter at that position. The resulting table can be used to approximate a number of properties of the stream. The tensor sketching approach uses a slightly modified version of this algorithm adapted to work on vectors rather than streams:

Definition 5 (Count Sketch)**.** Given a vector $\boldsymbol{x} \in \mathbb{R}^d$ and two 2-wise independent hash functions $h : \mathbb{N} \to \{1, ..., D\}$ and $s : \mathbb{N} \to \pm 1$. The Count Sketch of $\boldsymbol{x}$ is defined as $CS(\boldsymbol{x}) = \{C_1, ..., C_D\} \in \mathbb{R}^D$ where:

$$C_j = \sum_{i \in S_j} s(i)x_i, S_j = \{i \in [d] | h(i) = j\} \tag{2.39}$$

The iterator set S_j includes all integers $i = \{1, ..., d\}$ where $h(i) = j$.

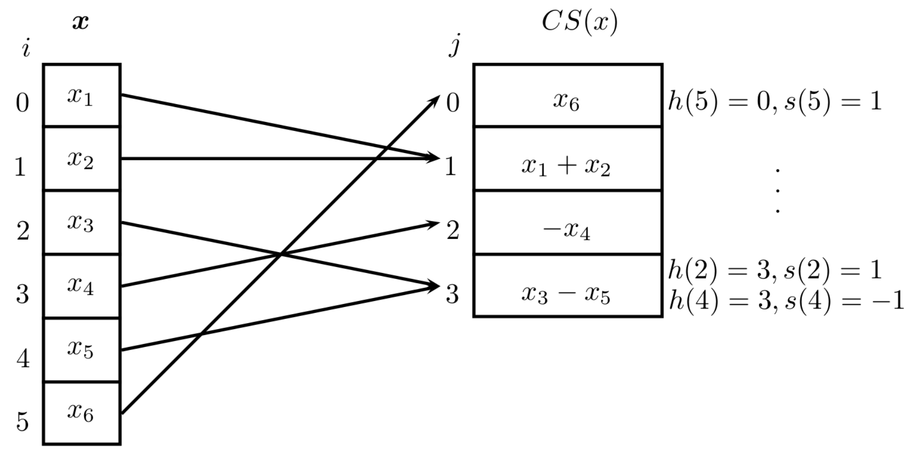


Figure 2.3: An example of a CountSketch with $d = 6$ and $D = 4$.

Since $h(i)$ maps to only one j computing the entire count sketch thus requires only one run over $\boldsymbol{x}$, for each element subtracting or adding its value to the corresponding entry in $CS(\boldsymbol{x})$. This means that the Count Sketch can be constructed in linear time in the dimensionality of the vector, $O(d)$. An example is given in Figure 2.3. This version of the Count Sketch has also been introduced as *The hashing trick*[22]. It has the useful property that the inner product of two vectors is maintained by Count Sketch within a bounded variance.

Lemma 2.3.3.

$$E[\langle CS(\boldsymbol{x}), CS(\boldsymbol{y})\rangle] = \langle \boldsymbol{x}, \boldsymbol{y}\rangle \tag{2.40}$$

Proof. Given $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$ and Count Sketch as defined in 2.39.

$$\langle CS(\boldsymbol{x}), CS(\boldsymbol{y})\rangle = \sum_g^D (\sum_{i \in S_g} s(i)x_i) * (\sum_{j \in S_g} s(j)y_j) \tag{2.41}$$

Keeping the same iterator set S we can simplify the expression.

$$\langle CS(\boldsymbol{x}), CS(\boldsymbol{y})\rangle = \sum_{g}^{D} \sum_{i,j \in S_g} s(i)s(j)x_i y_j \tag{2.42}$$

We now recall the definition of $s(i)$. Since $s : \mathbb{N} \to \pm 1$ is uniformly distributed:

$$E[s(i)s(j)] = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{2.43}$$

So:

$$E[\langle CS(\boldsymbol{x}), CS(\boldsymbol{y})\rangle] = \sum_{g}^{D} \sum_{i \in S_g} x_i y_i = \sum_{i}^{d} x_i y_i = \langle \boldsymbol{x}, \boldsymbol{y}\rangle \tag{2.44}$$

□

To see that the variance is tightly bounded we introduce another lemma:

Lemma 2.3.4.

$$Var[\langle CS(\boldsymbol{x}), CS(\boldsymbol{y})\rangle] = \frac{1}{D}(\sum_{i!=j} x_i^2 y_j^2 + \sum_{i!=j} x_i y_i x_j y_j) \tag{2.45}$$

Proof. See [22] Appendix A. □

Representing the count sketch as a polynomial

The current representation of the count sketch as a vector of sums(see 2.39) can be changed to a polynomial representation. We construct a polynomial where each term corresponds to an element of the vector. Consider the polynomial $f_x(u)$ below:

$$f_x(u) = \sum_{i=1}^{d} s(i) x_i u^{h(i)} = a_0 u^0 + a_1 u^1 + a_2 u^2 + \cdots + a_D u^D \tag{2.46}$$

The a_0 term corresponds to $\sum_{i \in S_0} s(i)x_i$, the a_1 term corresponds to $\sum_{i \in S_1} s(i)x_i$ etc. The coeficcients of the polynomial matches the entries of the vector representation. In this way a $D-1$ degree polynomial can be made that represents the Count Sketch $CS(\boldsymbol{x})$:

Definition 6 (polynomial representation of Count Sketch)**.** Given a vector $x \in \mathbb{R}$ and two 2-wise independent hash functions $h : \mathbb{N} \to \{1, ..., D\}$ and $s : \mathbb{N} \to \pm 1$, the polynomial:

$$p(u) = \sum_{i=1}^{n} s(i) x_i u^{h(i)} \tag{2.47}$$

represents the Count Sketch of x by its coefficients.

The polynomial representation of the Count Sketch will be useful in understanding the Tensor Sketching approach.

2.3.3 Tensor Sketching

The count sketch algorithm can be used to provide count sketches of tensor products. We call these tensor sketches.

Definition 7 (Tensor Sketch)**.** A tensor sketch is a Count Sketch of a tensor product.

$$TS_p(\boldsymbol{x}) = CS(\boldsymbol{x}^{(p)}) \tag{2.48}$$

Using lemma2.40 we see:

$$E[\langle TS_p(\boldsymbol{x}), TS_p(y)\rangle] = E[\langle CS(\boldsymbol{x}^{(p)}), CS(\boldsymbol{y}^{(p)})\rangle] = \left\langle \boldsymbol{x}^{(p)}, \boldsymbol{y}^{(p)} \right\rangle \tag{2.49}$$

combining this with lemma 2.3.1 and 2.3.2 we get:

$$E[\langle TS_p(\boldsymbol{x}), TS_p(y)\rangle] = (\langle \boldsymbol{x}, \boldsymbol{y}\rangle + c)^p \tag{2.50}$$

So $TS_p(\boldsymbol{x})$ provides an approximate mapping for the polynomial kernel as given in definition 2.

Fast convolution

A recent paper by Pagh[15] introduces a fast technique for constructing tensor sketches through fast convolution of count sketches. The heart of this technique is polynomial multiplication through the Fast Fourier Transformation.

Returning to the polynomial representation of the Count Sketch in eq.2.46 we now consider the Fast Fourier Transformation of two such polynomials.

$$FFT(f_x(u)) = \{f_x(\omega^0), f_x(\omega^1), \cdots, f_x(\omega^{D-1})\} = \{\sum_{i=1}^{d} s_1(i) x_i (\omega^0)^{h_1(i)}, \cdots \tag{2.51}$$

$$FFT(f_y(u)) = \{f_y(\omega^0), f_y(\omega^1), \cdots, f_y(\omega^{D-1})\} = \{\sum_{j=1}^{d} s_2(j) y_i (\omega^0)^{h_2(j)}, \cdots \tag{2.52}$$

By taking the componentwise multiplication of the two we can get a vector with D entries.

$$FFT(f_x) * FFT(f_y)_D = \sum_{i,j}^{d} s_1(i) s_2(j) x_i y_j (\omega^D)^{h_1(i)+h_2(j) \bmod D} \tag{2.53}$$

Notice that the exponents are constrained by a $\bmod D$ to concentrate the sums to D terms. By choosing hash functions $S(i,j) = s_1(i)s_2(j)$ and $H(i,j) = h_1(i) + h_2(j) \bmod D$ we can rewrite the entries as:

$$\sum_{i,j}^{d} S(i,j) x_i y_j (\omega^t)^{H(i,j)} \tag{2.54}$$

At this point we consider a direct computation of the Count Sketch of $x \otimes y$ using $S(i,j)$ and $H(i,j)$:

$$f_{x\otimes y}(u) = \sum_{i,j}^{d} S(i,j) x_i y_j u^{H(i,j)} \tag{2.55}$$

It is clear that the discrete fourier transform of the sketch yields the same entries as 2.54.

$$FFT(f_{x\otimes y}(u)) = FFT(f_x(u)) * FFT(f_y(u)) \tag{2.56}$$

So by performing the Fourier Inverse on 2.54 we have a fast method of computing $CS(\boldsymbol{x}^{(2)})$ from two different Count Sketches of x. The method generalizes to sketching any p-level tensor product:

Definition 8. Fast computation of p-level tensor sketch.

Compute p Count Sketches using p different hash-functions $h(\boldsymbol{x}) : \mathbb{N} \to [D]$ and $s(\boldsymbol{x}) : \mathbb{N} \to \pm 1$. Now using component-wise multiplication compute the p-level tensor sketch as:

$$CS(\boldsymbol{x}^{(p)}) = FFT^{-1}(FFT(CS_1(\boldsymbol{x})) * \dots * (FFT(CS_p(\boldsymbol{x}))) \tag{2.57}$$

Corresponding to $CS(\boldsymbol{x}^{(p)})$ using hash-functions H and S as:

$$H(i_1, ..., i_p) = \sum_{k=1}^{p} h_k(i_k) \mod D \tag{2.58}$$

$$S(i_1, ..., i_p) = \prod_{k=1}^{p} s_k(i_k) \tag{2.59}$$

Since computing the Count Sketch is $O(d)$ and FFT is $O(DlogD)$ this technique delivers a total runtime in $O(p(d + DlogD))$.

Error bounds

Approximation improves with D. The variance of the mapping is bounded by:

$$Var[\langle CS(\boldsymbol{x}^{(p)}), CS(\boldsymbol{y}^{(p)})\rangle] \leq \frac{1}{D}(\langle \boldsymbol{x}, \boldsymbol{y}\rangle^{2p} + ||x||^{2p}||y||^{2p}) \tag{2.60}$$

See [16] for a proof of the bound.

Chapter 3

Implementation

As part of this project Fastfood and tensor sketching was implemented in C++. For the sake of future usefulness both algorithms are implemented to work with the Shark machine learning library. Shark is an award winning open source C++ library[1]. Hopefully providing implementations of these algorithms for an efficient open source library like Shark might make them more likely to find uses in production. The Shark library is available at:
`http://image.diku.dk/shark/`
The C++ implementation of Fastfood and tensor sketch are available at:
`https://bitbucket.org/johanvts/fastkernel`
The code is available as a working extension of Shark including unit tests etc. The repository includes build instructions and precompiled versions ready for use with Shark. Figure 3.1 presents a short overview of the central parts of the codebase. To use the methods we leverage the powerful Shark "transform" method. An example call looks like this:

```
transform(shark::Data<shark::RealVector> Dataset,transformerClass)
```

Where transformerClass is an instance of "TensorSketch" or "FastFood-Stacked". Details about the implementation is reserved for the online documentation, but we will discuss a few central decisions in the rest of this section.

[1] `http://image.diku.dk/shark/`

```
/
├─ client ............ Binary for testing methods and generating data set
├─ Common
│  ├─ hadamardmatrix.cpp
│  │  ├─ FWHT(RealVector x) ........................ FWHT transform
│  │  └─ FWHTn(RealVecor x) .................... Normalized transform
│  ├─ Uniform distribution.cpp ........ Tools for uniform distributions
│  ├─ Normal distribution.cpp .......... Tools for normal distributions
│  └─ ··· ........... Tools for kernels, data management, timekeeping etc
├─ FastFood
│  └─ fastfood.cpp
│     ├─ FastFood(unsigned int d) ........... d dimensional V matrix
│     ├─ vProduct(RealVector x) .......................... returns Vx
│     ├─ gausssampler.cpp ......................... Gaussian support
│     └─ stacker.cpp ............................... Stacked V matrix
│        ├─ FastFoodStacked(unsigned int d,unsigned int D) . D x d
│        │  Matrix V
│        └─ map(const RealVector x) ...................... returns Vx
├─ TensorSketch
│  ├─ CountSketch.cpp ................................ Sketches vectors
│  └─ TensorSketch.cpp .............................. Tensor Product
│     ├─ TensorSketch(int d, int D, int p) .... d vector in Dp skecth
│     │  class
│     └─ () x ............................... return TensorSketch x(p)
├─ Experiement.cpp ........ Implements Speed and accuracy experiments
└─ SVM.cpp .... Examples for using the methods with internal and external
   SVM trainers
```

Figure 3.1: Implementation Overview

3.0.4 Fourier transform

Central to the performance of tensor sketching is a fast fourier transform. For this purpose the FFTW[2] library was used. This library was chosen because it has a proven track record of excellent performance across multiple platforms and is free for all purposes. Since FFTW is a C library the FFTW++[3] wrapper was used.

3.0.5 Hadamard transform

The performance of the Fastfood library hinges on a Fast Walsh-Hadamard transform. The Walsh-Hadamard transform can be implemented through FFTW as a multidimensional Discrete Fourier Transform, but this proved troublesome using the FFTW++ wrapper. The IT++ library[4] provides a FWHT method, but little information on performance was available. Spiral[5] provides a opti-

[2] `http://www.fftw.org/`
[3] `http://fftwpp.sourceforge.net/`
[4] `http://itpp.sourceforge.net/4.3.1/`
[5] `http://www.spiral.net/software/wht.html`

mized C package for FWHT but no wrapper. To keep the number of external library dependencies low and the code readable we instead used a simple direct implementation which takes only a few lines of code. A small experiment was carried out to verify speed and accuracy, see appendix A. Implementing an optimized Fast Walsh Hadamard transform library might provide a speed up but our native solution delivers on the $O(d \log d)$ time expectancy of the FWHT.

3.0.6 Testing probability based methods

The central methods of Fastfood and tensor Sketching are listed in table 3.1. The output of these methods is expected to have the listed properties, but a specific run being completely off does not necessarily mean that the method is wrongly implemented. This poses a challenge to normal unit testing and makes it harder to debug and test code.

Class	**Method**	**Output Properties**	**Error**
Count Sketch	sketchVector	$E[\langle CS(x), CS(y)\rangle] = \langle x, y\rangle$	2.45
Tensor Sketch	map	$E[\langle TS_p(x), TS_p(y)\rangle] = \left\langle x^{(p)}, y^{(p)}\right\rangle$	2.60
Fastfood	vProduct	$E[Vx] = E[Gx]$	2.18
Stacker	map	$E[Vx] = E[Gx]$	2.18

Table 3.1: Probability based testing

To test these requirements we used the statistics component of the Boost library[6]. The methods were evaluated multiple times and the average was checked to be within the expected error range. For the Count Sketch and Tensor Sketch methods the output is checked against $\langle x, y\rangle$. For the FastFood methods the output is checked against the expectancy $E[Gx] = M_\mu x$ where M_μ contains averages of the used Gaussian.

[6]`http://www.boost.org/`

Chapter 4

Experiments

Fastfood and tensor sketching both promise to deliver fast and accurate kernel estimates[13, 16]. To test the speed and performance of the methods we conducted a series of experiments on the implementation introduced in the previous chapter. We first investigate the accuracy of each method and try to pinpoint the conditions that give the most accurate kernel approximation. In section.4.2 we look at the speed of each method and how the theoretical expectations compare to our observations. Finally we put each method to use on the MNIST classification task to see how the combination of kernel approximation and linear classification compare to using the kernel trick and non-linear support vector machines.

4.1 Kernel approximation

Previous work has shown that the methods can work[16, 13], but do not provide much insight in the technical challenges involved in implementing and optimizing these methods. These experiments have two purposes. To verify the theoretic expectations on accuracy and speed independent of the original implementations and to extract new information on the kernel and input data properties involved in attaining the best results.

To recap, the promise of the mappings are that:

$$E[\langle \boldsymbol{z}(x), \boldsymbol{z}(y)] = k(x, y) \tag{4.1}$$

So to test this promise experimentally on a dataset of n entries we can use these error functions:

$$Error_{abs.} = \frac{1}{n} \sum_{x,y}^{n} |\langle \boldsymbol{z}(x), \boldsymbol{z}(y) \rangle - k(x, y)| \tag{4.2}$$

$$Error_{rel.} = \frac{1}{n} \sum_{x,y}^{n} \frac{|\langle \boldsymbol{z}(x), \boldsymbol{z}(y) \rangle - k(x, y)|}{k(x, y)} \tag{4.3}$$

The relative error function is troublesome to work with here because some kernels are very close to zero, a small error in the estimation of these values can skew the average relative error severely. The absolute value on the other hand is hard to compare across different kernels and parameters. To get an easily comparable and readable error measure we measured the relative error as above, but only include pairs where the relative error was $\leq 100\%$. If the percentage of inputs fulfilling this condition is less than 100% it is reported along with $Error_{rel}$ as $E_{\%}$.

Figure 4.1 show kernel estimates from the Fastfood method plotted against exact kernel values. Each point represents a combination of two vectors x, y. The coordinate corresponds to $(k(x, y), \langle \boldsymbol{z}(x), \boldsymbol{z}(y) \rangle)$. A perfect mapping would manifest as a narrow 45-degree line. Previous results analyze the precision on an aggregated level[16]. Using this plot we can more easily get a nuanced understanding of the methods ability to estimate the kernel value.

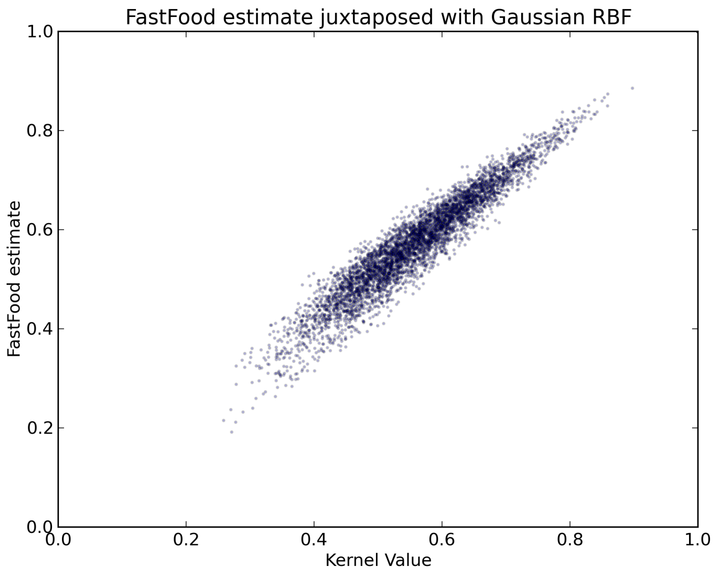
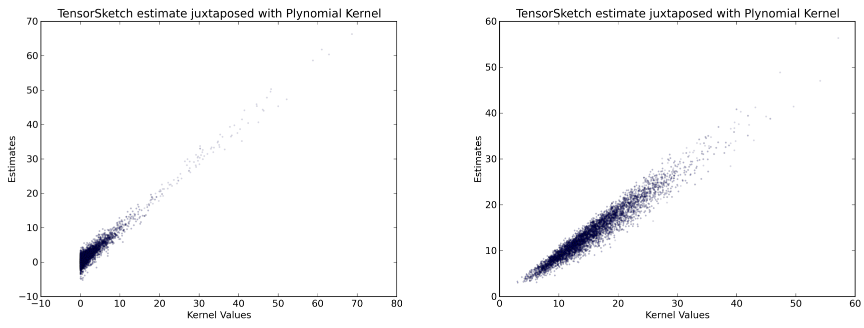


Figure 4.1: Accuracy test of Fastfood.$d = 16, D = 512, \sigma = 1.5, Error_{Rel.} = 3.5\%$

Figure 4.1 shows Fastfood estimates for a dataset of 100 vectors sampled uniformly from $[0, 1]^{16}$ using 512 features.

4.1.1 Tensor Sketch input normalization

Using randomly generated data sampled from $[-1, 1]$ we experienced much lower accuracy for Tensor Sketching than expected. Vectors sampled uniformly i.i.d from $[-1, 1]^d$ have an expected inner product of zero. This means that we get a high relative error, and it also makes it difficult to investigate a larger spectrum of the kernel range because values concentrate on zero. With no cost to generality we can transform data into a positive range before sketching. This gives a much broader range of values and should make classification easier. It also demonstrates that the polynomial kernel is not shift invariant. Looking at figure 4.2 we see the difference in the range and accuracy of estimation kernel.



(a) $\boldsymbol{x} \in [-1,1]^d, Error_{rel.} = 42\%, E_{\%} = 39\%$ (b) $\boldsymbol{x} \in [0,1]^d, Error_{rel.} = 13\%$

Figure 4.2: **Tensor sketching random data**
$d = 16, D = 128, p = 2, C = 0$

Unless anything else is stated explicitly the experiments described here were conducted with data sampled i.i.d. from a uniform distribution between $[0,1]$.

4.1.2 Increasing the feature count

The first experiment concerns the feature count, D. Looking at the error bounds given for Fastfood,p.14, and tensor sketching,eq.2.60, we would expect the approximation error to fall sharply with increasing feature count.

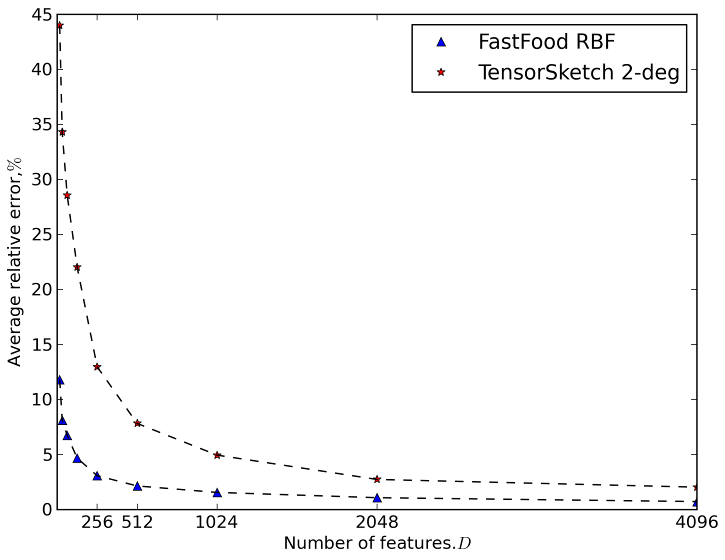


Figure 4.3: Error decreases as a function of feature count,D.$d = 16, \sigma = 2, p = 2$

The results in fig.4.3,4.4 and table.4.1 are averages over 16 runs of each algorithm. Each run on a randomly generated dataset of 10000 vector pairs from vectors $\in [0,1]^{16}$. The relative error depends heavily on the choice of kernel parameters. By choosing dominating C and σ that effectively place a strict limit on the range of the kernel we can artificially lower the error. We will discuss this further in the next experiment. For this experiment we use $C = 0$ and $\sigma = 2$ as these parameters give a decent spread over the kernel range.

Table 4.1: **Kernel estimation error** $d = 16, \sigma = 2, p = 2, C = 0$

D	$Fastfood_{Abs.}$	$Fastfood_{Rel.}\%$	$TensorSketch_{Abs.}$	$TensorSketch_{Rel.}\%$
16	0.083	11.82	8.39	44.01
32	0.057	8.14	5.56	34.32
64	0.047	6.74	4.99	28.57
128	0.033	4.68	3.63	22.02
256	0.022	3.08	2.19	12.98
512	0.015	2.15	1.32	7.83
1024	0.011	1.56	0.81	4.95
2048	0.0077	1.08	0.39	2.74
4096	0.0052	0.73	0.32	2.04
8192	0.0036	0.51	0.31	1.73

The performance seems on par with the results presented in the original papers[13, 16]. This is particularly interesting for Fastfood since our implementation uses a novel way of sampling the scaling matrix.

Tensor sketching does show a more fluctuating performance than Fastfood. As an example, for $D = 4096$ the best tensor sketch mapping had an accuracy of 1% but the worst 4.1%. This is likely due to the way we choose h functions. They are mappings sampled uniformly i.i.d. to map from $[d]$ to $[D]$. While this gives a good performance expectancy, it also allows cases where the performance is off. In the worst case the whole input is hashed to the same position in the sketch. Future work could study how to best construct these hash functions. It seems that for many applications collisions could be completely avoided since we can choose $D \gg d$. This could improve the accuracy of tensor sketching beyond the results shown here.

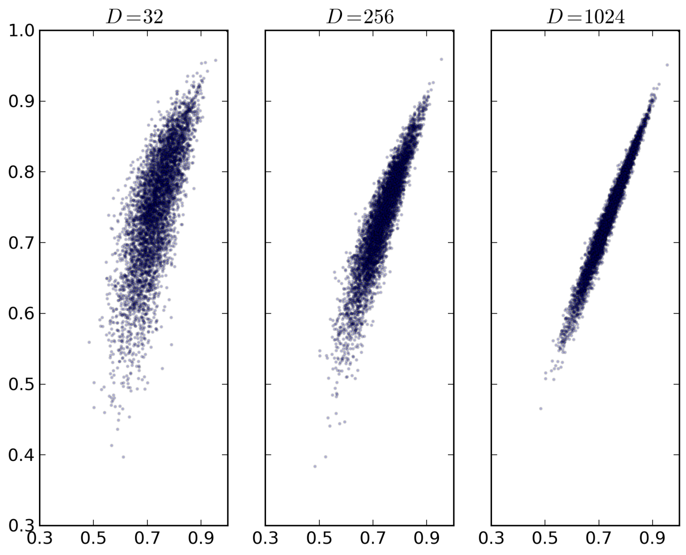


Figure 4.4: Fastfood estimates improve with D.$d = 16, \sigma = 3$

4.1.3 Gaussian RBF Kernel range

The estimates are not equally good across the range of the kernel. As we mentioned briefly in the previous experiment kernel parameters greatly effect accuracy of the estimates.

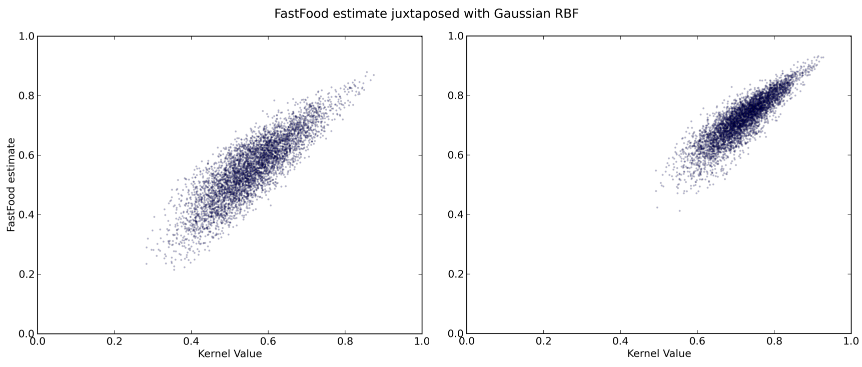


Figure 4.5: Changing the kernel range. Left:$sigma = 1.5, Error_{rel.} = 8.3\%$, right $\sigma = 2, Error_{rel.} = 3.6\%$. Both $d = 16, D = 128$.

Fig.4.5 gives an impression of how the approximation suffers for lower kernel values. There is an interesting tradeoff between kernel resolution and the ability to accuratly estimate the kernel. Figure 4.6 shows how the average relative error, in discreet sections of the kernel range, drops for higher values. This highlights

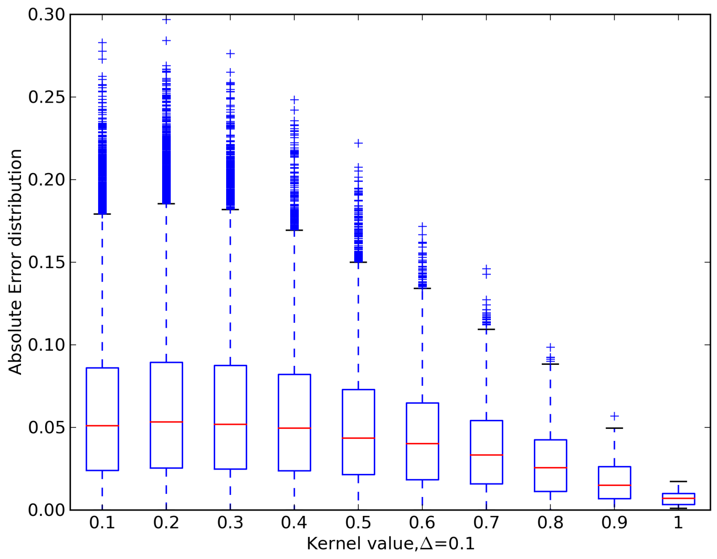


Figure 4.6: Error over kernel range. $d = 8, D = 128, sigma = 0.6, Error_{Abs.} = 0.059$. Shows the distribution in absolute error

the importance of choosing the right σ value. By choosing sigma to fit data in the high part of the Gaussian RBF range estimates are expected to be more similar to using the kernel, but with a downside of less resolution for a classifier to work with.

This behavior is likely due to the way vector entries are summed in the Fastfood map. The $||\boldsymbol{x} - \boldsymbol{y}||$ of the kernel captures the difference between inputs on each dimension, the $\langle \boldsymbol{v_j}, \boldsymbol{x} \rangle$ of the mapping captures the difference on an aggregated level. Consider in the extreme case two orthogonal unit vectors. The kernel will easily identify these as different, but they have the same expected value in the Fastfood mapping. The B and S matrices of Fastfood ensure that they will not likely have the exact same value, but this effect makes it hard for Fastfood to maintain differences based on variations across the input dimensions. Vector pairs that are dissimilar across all dimensions and similar vectors are less distorted by the aggregation. Increasing σ improves the error because it lessens the kernels discrimination between dissimilar pairs, not because it improves the accuracy of Fastfood.

4.1.4 Mappings for Fastfood

Testing the mappings presented for Random Kitchen Sinks in chapter 2.1 we found that the $\boldsymbol{z}''$ mapping was faster, as expected, but also represented a step down in terms of accuracy. Hence the implementation and experiments presented as part of this thesis uses the $\boldsymbol{z}$ mapping as defined in eq.2.12:

$$z_j(x) = \frac{1}{\sqrt{D}}\{\cos([\Omega\boldsymbol{x}]_j), \sin([\Omega\boldsymbol{x}]_j)\} \tag{4.4}$$

Results for the $\boldsymbol{z}''$ mapping can be found in appendix C.

4.1.5 Sparse input for Tensor Sketch

The main loss of information suffered in the Tensor Sketch approach is from collisions in the Count Sketching step. In illustration 2.3 e.g. we see a collision occurring on the second and last entry in the sketch. Having sparse vectors should reduce the loss of information suffered in such collisions because the chance that one of the colliding inputs is zero increases. At the same time all non-zero entries are still appearing in the sketch. To verify this experimentally we conducted a series of experiments with differing input sparsity.

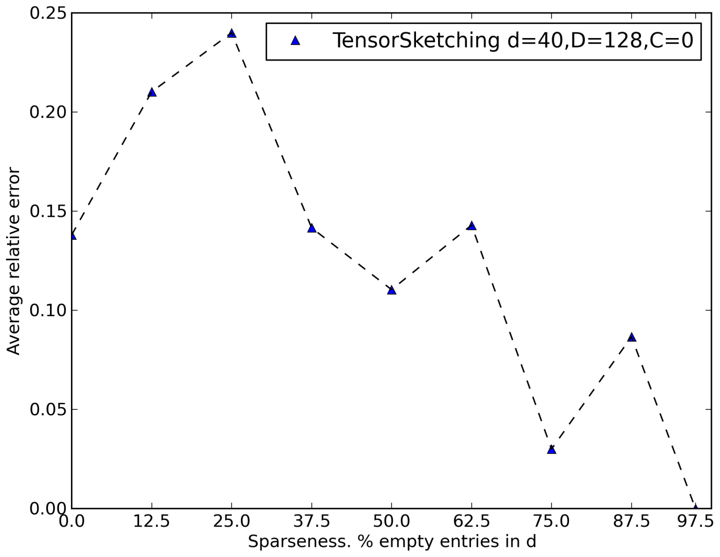


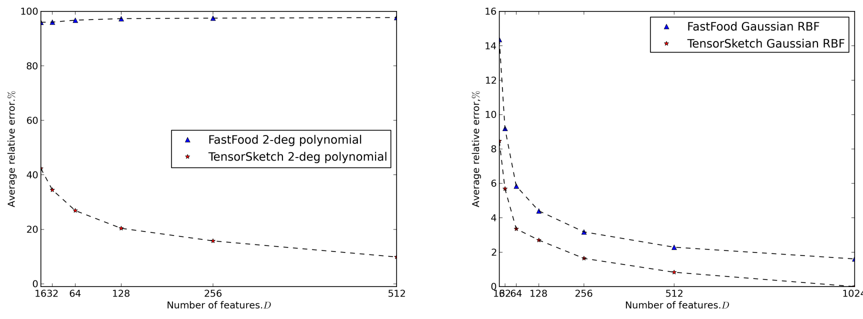
Figure 4.7: Relative error over input sparsity. $d = 40, D = 128, p = 2, n = 100$. Average over 12 runs.

When sparsity goes up in our random dataset, the polynomial kernel values generally drop, making it hard to isolate the effect of sparsity alone. To measure it we need to scale the non-zero entries up as sparsity increases. We construct data vectors $\boldsymbol{x}$ with entries $x_d = \frac{\sqrt{k}}{d-s}$ where s is the number of sparse entries. We must also make sure that the zero-entries remain in a fixed part of the constructed input, having random sparsity would again make the actual kernel smaller and thus increase relative error. With this rather artificial input construction kernel value remains constant. Notice that this construction of the input data is only necessary for measuring the isolated effect of sparsity on kernel approximation.

Looking at fig.4.7 we see that the effects of increasing sparsity are complicated. While the error generally does drop as sparsity increases, it also rises especially in the beginning. This might be due to the fact that scaling the entries up also mean that each collision that does occur causes greater distortion.

4.1.6 Accuracy comparison

We can use eq.1.7 to make a direct comparison between Fastfood and tensor sketching. The Fastfood method works on shift-invariant kernels and the length of the original input is lost when mapping. Tensor sketching on the other hand maintains a good estimate of the original input length. Fig.4.8 shows this difference when we use eq.1.7 to directly transform the polynomial kernel estimates into estimates for the Gaussian RBF and vice versa.



(a) Fastfood mapping converted

(b) Tensor sketch mapping converted

Figure 4.8: **Converting estimates**
$d = 16, p = 2, \sigma = 2$

This does not provide a way to use tensor sketching as a map for RBF kernels, but it gives some impression of the amount of information loss. It is clear that Fastfood maintains much less information after mapping than tensor sketching. The converted estimates are very inaccurate for Fastfood and very precise for tensor sketch. If we use information on the lengths from the original input Fastfood does provide an accurate conversion, see fig.D.1 in the appendix. This shows how Fastfood captures only the directions of vectors while tensor sketching also conserves the length. Pham and Pagh presents the idea of applying tensor sketching for the Gaussian kernel through Taylor-series approximations[16]. This experiment suggests that such methods might provide more accurate mappings for the Gaussian RBF than Fastfood.

4.2 Speed

The attraction of both methods is the promise of a significant speed improvement over other random feature map methods. The asymptotic performances presented in the previous sections are listed in table4.2. Each experiment shows a table breaking down the expected run-times in the varying parameter.

Method	**Time**	**Storage**
Fastfood	$O(nD\log d)$	$O(D)$
Tensor Sketch	$O(np(d + D\log D))$	$O(pd\log D)$

Table 4.2: **Time and Storage costs**.Number of samples n. Input dimensions d.Number of features D. Degree of polynomial p.

4.2.1 Samples n

Fastfood		**Tensor Sketch**	
Evaluate map	$O(n)$	Evaluate map	$O(n)$
$O(n)$		$O(n)$	

Table 4.3: Speed costs of n

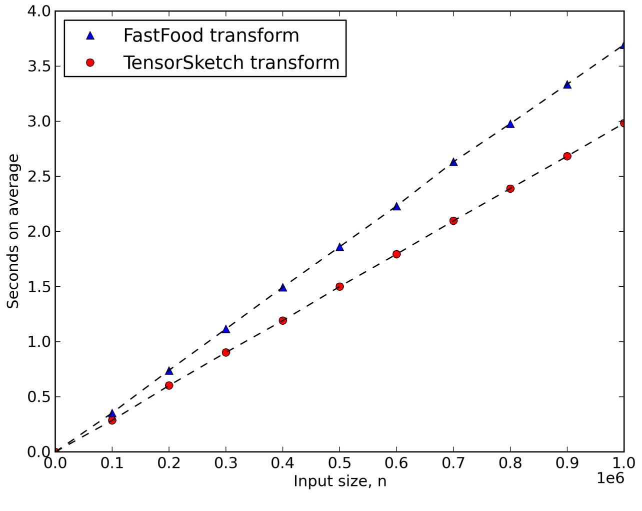


Figure 4.9: Speed measured as a function of data samples. $d = 16, D = 16$

Both algorithms show linear dependency on the number of samples in the input. This is in accordance with the theoretical expectation. Recall that the problem with using kernel methods to begin with is the cubic growth of the Gram matrix, we would be no better off if these methods did not scale much better in the data set size.

4.2.2 Features D

Fastfood		**Tensor Sketch**	
Draw samples	$O(D)$	Choose range of h functions	$O(1)$
Repeat the FWHT	$O(D)$	Perform FFT	$O(D \log D)$
Evaluate map	$O(D)$	Evaluate map	$O(D)$
$O(D)$		$O(D \log D)$	

Table 4.4: Speed costs of D

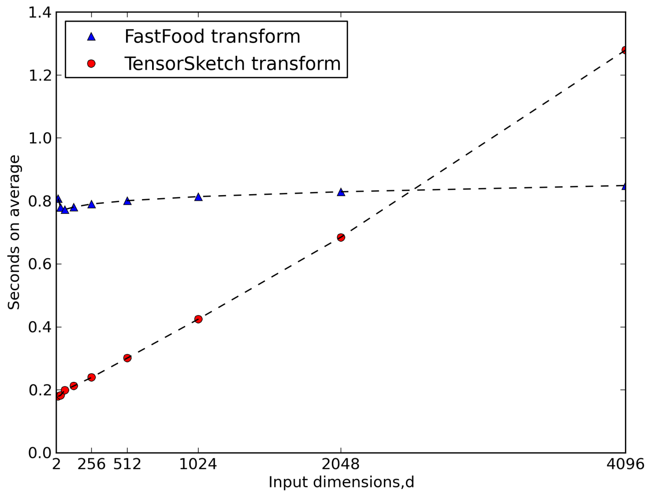


Figure 4.10: Speed measured as a function of Features. $d = 16, n = 1000$

Again both algorithms look very linear. While tensor sketching is expected to show log-linear performance when varying D it is still very fast and the additional $\log D$ factor does not dominate for even a very high amount for features. Using the z mapping,eq.2.26), requires evaluating and storing both the sine and cosine of each feature. The alternative z'' mapping improves the speed of Fastfood as show in appendixC. Still tensor sketching is faster which is likely due to the highly optimized FFTW[1] library handling the Fourier transforms at the heart of our tensor sketch implementation. D does not affect the size of the transform in FastFood which is always determined by d, but it does affect the number of transforms needed, D/d, as illustrated in fig.2.2. As such an optimized FWHT library might improve FastFood speed over D.

4.2.3 Input dimensions d

While FWHT is $O(d \log d)$ Fastfood always performs D/d transforms. For all practical applications of Fastfood $D \gg d$. As d doubles D/d halves and $\frac{D}{d} d \log d)$

[1]http://www.fftw.org/

Fastfood		Tensor Sketch	
	Choose domain of h,s functions	$O(1)$	
FWHT	$O(d \log d)$	Sketch input	$O(d)$
Evaluate map	$O(1)$	Evaluate map	$O(1)$
	$O(d \log d)$		$O(d)$

Table 4.5: Speed costs of d

remains constant except for the logarithmic term. time consumption grows in a logarithmic fashion in d. This is reflected in the results shown in fig4.11.

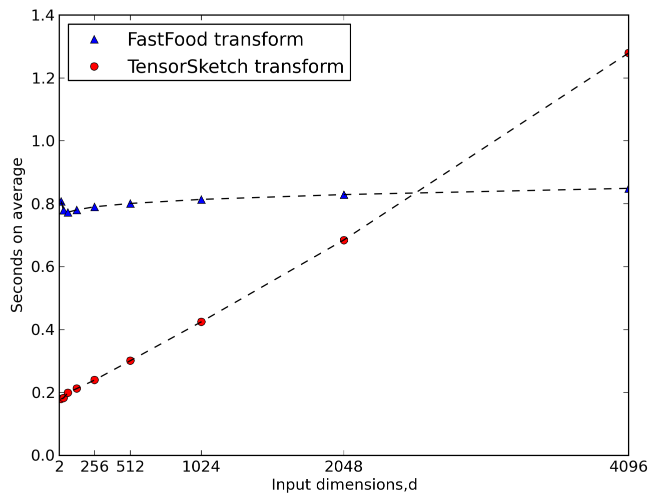


Figure 4.11: Speed measured as a function of input dimensions.$D = 4096, n = 1000$

Initially Fastfood seems to be become faster when increasing the input dimensionality. A possible explanation is the cost of object allocation. The Fastfood implementation creates and stacks D/d matrices V', see fig.2.2. For low d and high D as in the left side of fig.4.11 D/d becomes very large and object instantiating appears to dominate the run time. Other than that, asymptotic results are as expected. Tensor sketching exhibits linear growth in time consumption while Fastfood delivers on the promise of logarithmic dependency on the input dimensions.

4.3 Image Recognition

The accuracy and speed experiments suggest that the methods are suitable for use in training linear models. To test the methods in a complete machine learning scenario we used the popular MNIST[2] dataset. MNIST contains 60.000

[2]http://yann.lecun.com/exdb/mnist/

images of handwritten digits and the task is to classify each image as a number $[0 - 10]$. There are already excellent results for this dataset in a number of implementations and the goal here is not to improve the results on MNIST. We use MNIST to test how the speed and accuracy of Fastfood and tensor sketching combined with a linear SVM compares to using the classic kernel and non-linear SVM approach in an end-to-end machine learning scenario.

4.3.1 Kernel method

For training using the kernel trick we trained a linear model using Sharks build in multiclass One-Versus-All SVM. Kernel parameters were chosen to be in the area known to give good results. For the Gaussian RBF we used $\sigma = 8.27$. For the polynomial kernel we chose a 4-degree polynomial with no offset. We use the regularization parameter $C = 100$ for training in all cases. The MNIST webpage reports a best known error rate of 1.4% for a Gaussian RBF and 1.1 for a 4-degree polynomial. Our results are very similar. Tuning the parameters might improve the accuracy further.

Method	d/D	**Error%**	**Training** s	**Trans.** s
SVM w. 4-deg poly.kernel	780	**1.8**	10029	NA
Tensor Sketch + LIBLINEAR	4096	2.6	509	35
SVM w. Gaussian RBF	780	**1.5**	76299	NA
Fastfood + LIBLINEAR	4096	1.9	1177	93

Table 4.6: **MNIST results** All experiments conducted on a 2.1Ghz 64bit AMD processor. Numbers show for random mappings are averages over 10 runs.

4.3.2 Random Feature Mappings

Using Fastfood or tensor sketching for learning is a two step procedure: transformation and training. First we transformed the training and test datasets using our implementation of Fastfood and tensor sketching. The parameters were chosen to match the kernels used for the kernel based approach: $\sigma = 8.27$,$p = 2$. For maximum performance we chose a number of features significantly larger than the inputs 780. Looking at the accuracy experiments this is necessary to get good kernel estimates. After transforming data we used the LIBLINEAR[3]. library to train a support vector machine. LIBLINEAR is a widely used linear SVM trainer[21].

An issue that requires careful consideration when working with the transforms is data handling. As an example the Fastfood mapping conducted here uses 4096 features. Each feature requires storing two values, the sine and cosine of $\langle \boldsymbol{\omega_j}, \boldsymbol{x} \rangle$. The result is that the transformed dataset is around 8-9 times larger than the original. For MNIST the transformed training and test data took up around 7GB. Shark implements many useful tools to handle such challenges[4]. However, when using the external SVM library it was necessary to write the transformed data to disk at a considerable disadvantage. This can be

[3]`http://www.csie.ntu.edu.tw/~cjlin/liblinear/`

[4]`http://image.diku.dk/shark/sphinx_pages/build/html/rest_sources/tutorials/concepts/data/datasets.html`Shark Data Containers

improved by integrating a linear time trainer in Shark, but the transforms will always carry a considerable storage cost because the number of features D has to be significantly larger than the original input dimension d. However this is a cost that is paid before training, both methods offer an $O(D)$ storage cost for classification.

4.3.3 Results

The results can be seen in table.4.6. The accuracy of the model created by using kernel methods should be seen as a benchmark for the random features based methods, it represents the result achievable by a 100% accurate mapping $z(x)^T z(y) = k(x, y)$. LIBLINEAR found very accurate models using both Fastfood and the tensor sketching. The results clearly support the idea that using random features can provide significant speedups to learning. Accuracy suffers slightly, but training times drops to fractions of the time needed for non-linear trainers.

Chapter 5

Conclusions and future directions

This thesis described the Fastfood and tensor sketching methods for kernel approximation based on random projections. The theoretical foundation of each method was presented and experiments were carried out to analyze the strengths and weaknesses of each method. Additionally an open source C++ implementation of both methods was presented and made available as part of the project.

We demonstrated that the Fastfood and tensor sketch algorithms can produce low dimensional mappings that provide accurate estimates of the Gaussian RBF kernel and the polynomial kernel respectively. And perhaps more importantly that they can do it in linearithmic time. Experiments on the MNIST dataset demonstrated performance on par with kernel based methods on a well known classification problem. We took a novel approach to analyzing the methods in looking more closely at the properties of kernel parameters, normalization, input sparsity and mapping choices. We saw how choosing σ represents a trade off between accuracy and resolution for the Fastfood method. Normalization was shown to have a large impact on tensor sketching accuracy, due to the nature of the polynomial kernel, and we discussed the diverse effects of input sparsity on tensor sketching. A novel way of sampling the Fastfood scaling matrix for Gaussian RBF kernels is introduced in this thesis. Experiments verify that results remain on par with the more cumbersome original method. All of these findings cover important choices when putting the methods to use.

Fastfood and tensor sketch both give fast and accurate results and both scale linearly in dataset size. This thesis provides a point of reference for developers and others who wish to make use of this power to implement effective machine learning techniques on large datasets.

During the course of this thesis a number of possibilities for future research naturally appeared. For tensor sketching future work could include investigating the optimal choice for the h hash-functions. The theoretical and experimental work in this thesis suggest that accuracy can be further improved for the tensor sketch method by eliminating all count-sketch collisions. Future work along this line could improve the average accuracy performance, and perhaps provide tighter bounds on the precision for the tensor sketch approach.

For Fastfood we suggested the integration of an optimized Fast Walsh Hadamard Transform library into the implementation. This might reduce time consumption for the Fastfood transform. Future work might also well focus on developing a more accurate version of the $\boldsymbol{z}''$ mapping to capitalize on the potential speed improvement show in this work.

In this thesis we have focused on applications of Fastfood and tensor sketching in machine learning, but the methods represent general embedding of information and might well find uses in other fields.

Bibliography

[1] Nir Ailon and Bernard Chazelle. Faster dimension reduction. *Commun. ACM*, 53(2):97–104, February 2010.

[2] M. Aizerman, E. Braverman, and L. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[3] Christopher M Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[4] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, January 2004.

[5] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[6] Greg Fasshauer. Positive Definite and Completely Monotone Functions, 2003.

[7] By John Gantz and David Reinsel. Extracting Value from Chaos State of the Universe : An Executive Summary, 2011.

[8] Aicke Hinrichs and Jan Vybíral. On positive positive-definite functions and Bochner s Theorem. *Journal of Complexity*, 27(3-4):264–272, June 2011.

[9] N Honda, J T Butler, K J Breeding, K Chakrabarti, D Kolp, C Minnick, and V Summary. Unified Matrix Treatment of the Fast Walsh-Hadamard Transform. *IEEE Transactions on Computers*, C-25(11):1142–1146, 1976.

[10] Thorsten Joachims. Training linear SVMs in linear time. *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '06*, page 217, 2006.

[11] Purushottam Kar and Harish Karnick. Random feature maps for dot product kernels. *arXiv preprint arXiv:1201.6530*, 22, 2012.

[12] H.O. Kunz. On the Equivalence Between One-Dimensional Discrete Walsh-Hadamard and Multidimensional Discrete Fourier Transforms. *IEEE Transactions on Computers*, C-28, 1979.

[13] Quoc Le, T Sarlós, and Alex Smola. Fastfood - Approximating Kernel Expansions in Loglinear Time. *ai.stanford.edu*, 28, 2013.

[14] J Mercer. Functions of Positive and Negative Type, and their Connection with the Theory of Integral Equations. *Philosophical Transactions of the Royal Society of London*, CCIX(1904):415–446, 1909.

[15] Rasmus Pagh. Compressed matrix multiplication. *Proceedings of the 3rd Innovations in Theoretical . . .*, 2012.

[16] Ninh Dang Pham and Rasmus Pagh. *Fast and Scalable Polynomial Kernels via Explicit Feature Maps*, pages 239–249. Association for Computing Machinery, 2013.

[17] Ali Rahimi and B Recht. Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning. *Advances in neural information processing systems*, 1(1):1–8, 2008.

[18] Ali Rahimi and Ben Recht. Random features for large-scale kernel machines. *Advances in neural information processing systems*, pages 1–8, 2007.

[19] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge University Press, 2004.

[20] George S. Kimeldorf Wahba and Grace. A Correspondence Between Bayesian Estimation on Stochastic Processes and Smoothing by Splines, 1970.

[21] R.-E. Fan Wang, K.-W. Chang, C.-J. Hsieh, X.-R., and C.-J. Lin. LIBLINEAR: A Library for Large Linear Classification. *Journal of Machine Learning Research*, pages 1871–1874, 2008.

[22] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1113–1120, New York, NY, USA, 2009. ACM.

List of Figures

List of Tables

Appendix A

Verify speed of Walsh-Hadamard Transform

Optimized libraries like Spiral[1] are available to perform system optimized Fast Walsh-Hadamard Transform(FWHT). We also implemented a simple FWHT to be a native part of the Fastfood package[2], it's simpler to test and time and reduces the amount of external libraries needed to get started. However for use in production it is recommended to use an optimized FWHT. When installing the FastFood library through CMAKE use the "Use Spiral WHT" setting to enable the third party solution from Spiral.

It is central to the Fastfood technique that the Walsh-Hadamard be performed in $O(d \log d)$ where d is the dimensionality of the input. A small test was carried out to show that the native solution adheres to the bound.

A.1 Experiment

1000 vectors of random numbers wheres generated for each power of two from 2^1 to 2^{10}. Each of the corresponding dimension. Every vector was transformed in two ways. First a transformation using the FWHT implementation. Secondly a transform by matrix multiplication with the Walsh-Hadamard matrix using the Boost library.

A.2 Results

The results support the expected $O(d \log d)$ bound for Fast Walsh-Hadamard Transform.

[1]http://www.spiral.net/software/wht.html
[2]common/hadamardmatrix.cpp

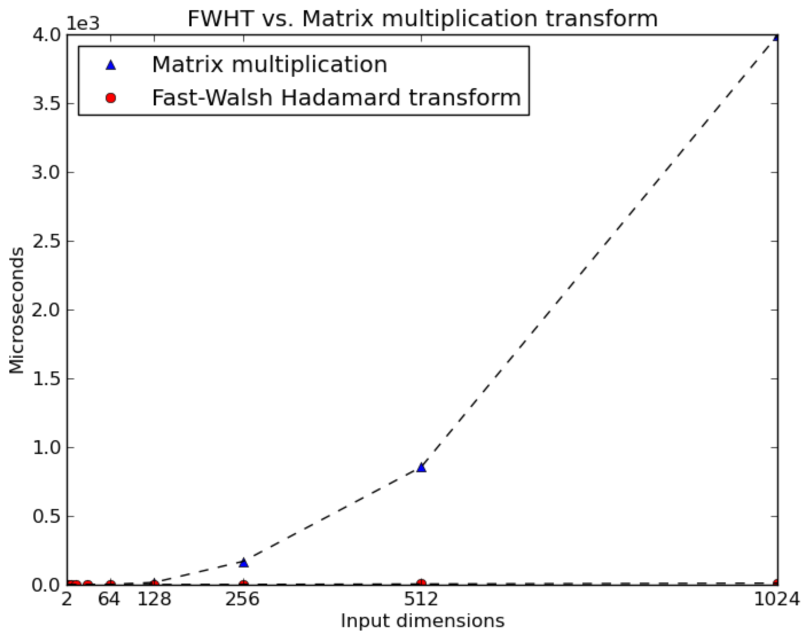


Figure A.1: Speed comparison of FWHT to Boost optimized matrix multiplication

A.3 Using the Timer class

The timer class is located in common.

```
#include <timer.h>
```

An instance of timer is started, stopped or reset. It can be queried for hours, minutes, seconds, milliseconds, microseconds or nanoseconds. A query on a started clock gives a split time. Querying or stopping a clock before starting it gives an error. The best way to use to class is:

- instantiate timer
- start timer
- —perform action—
- stop timer
- query timer at will

Appendix B

Mercers Theorem

The matehematical foundation for the Kernel trick is Mercers theorem. It can be used to guarantee the existence of kernel functions with the property of 1.3.

Theorem B.0.1 (Mercers Theorem)**.** *Let the function $k : X \times X \to \mathbb{R}$ be symmetric, continous and positive semi-definite. Then there exists a countable sequence of functions $\phi = \{\phi_i\}_{i \in \mathbb{N}}$ and a sequence of positive real numbers $\lambda = \{\lambda_i\}_{i\mathbb{N}}$ such that,*

$$k(x, x') = \sum_j \lambda_j \phi_j(\boldsymbol{x}) \phi_j(\boldsymbol{x}') = \langle \phi(\boldsymbol{x}), \phi(\boldsymbol{x}') \rangle \tag{B.1}$$

by

$$\phi(\boldsymbol{x}) = \{\phi_j(\boldsymbol{x}) + \sqrt{\lambda_j}\} \tag{B.2}$$

Where $\lambda_j > 0$ and ϕ_j is orthonormal on $L^2(X)$.

The theorem is very usefull since it garantees that a function satifying the conditions actullay correspond to an inner product in some vector space. In order for $k(\boldsymbol{x}, \boldsymbol{y})$ to be a valid kernel function, the Gram matrix K, whose elements are given by $k(\boldsymbol{x_n}, \boldsymbol{x_m})$, should be positive semidefinite for all possible choices of the set x_n.[3, p.295][19].

Appendix C

Alternative mapping for Fastfood

The alternative mapping[1]:

$$z''_j(x) = \frac{1}{\sqrt{D}}\sqrt{2}\cos(\boldsymbol{\omega_j}^t x + b) \tag{C.1}$$

Has the clear benefit that it requires storing only one number pr. feature. As expected this reduces the time necessary to perform the Fastfood transform:

Unfortunately this speedup comes at the expense of accuracy in the kernel estimate. As can be seen in fig.C.4 there seems to be a lower bound on the accuracy which might well stem from the adding of the random variable b in the z'' map.

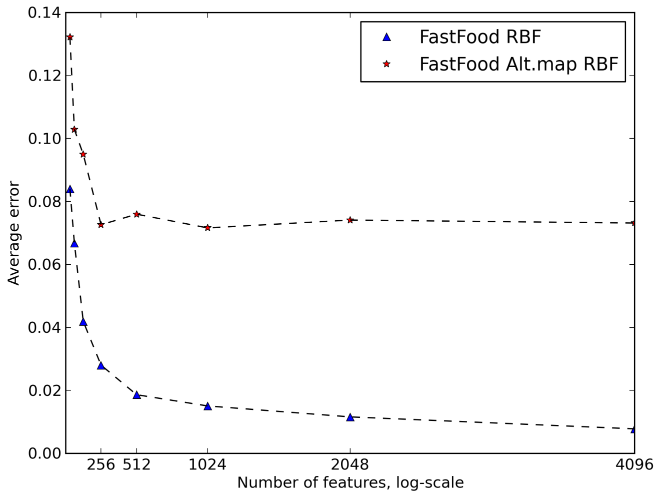


Figure C.4: Error decrease as a function of feature count,D.$d = 16, \sigma = 3$

[1]See p.8

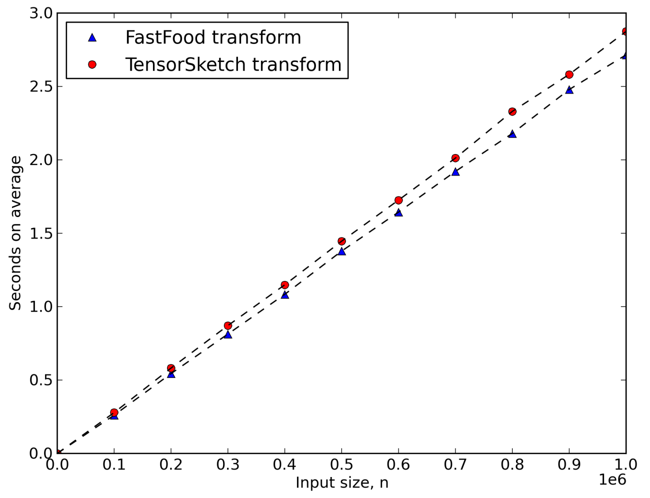


Figure C.1: Speed measured as a function of data samples. $d = 16, D = 16$

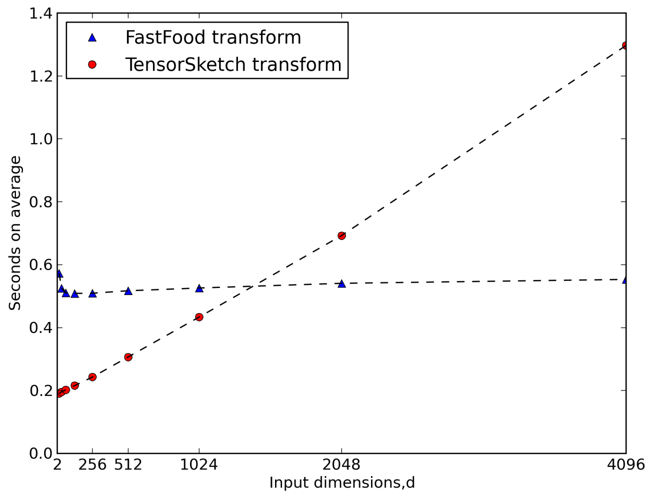


Figure C.2: Speed measured as a function of Features. $d = 16, n = 1000$

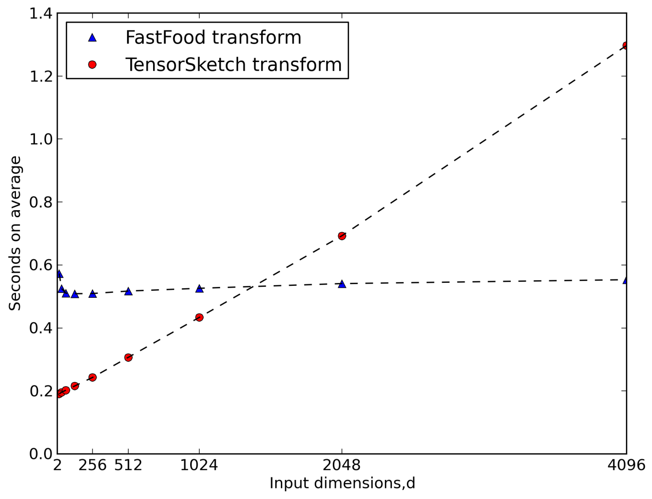


Figure C.3: Speed measured as a function of input dimensions. $D = 4096, n = 1000$

Appendix D

Converting Fastfood with original vector lengths

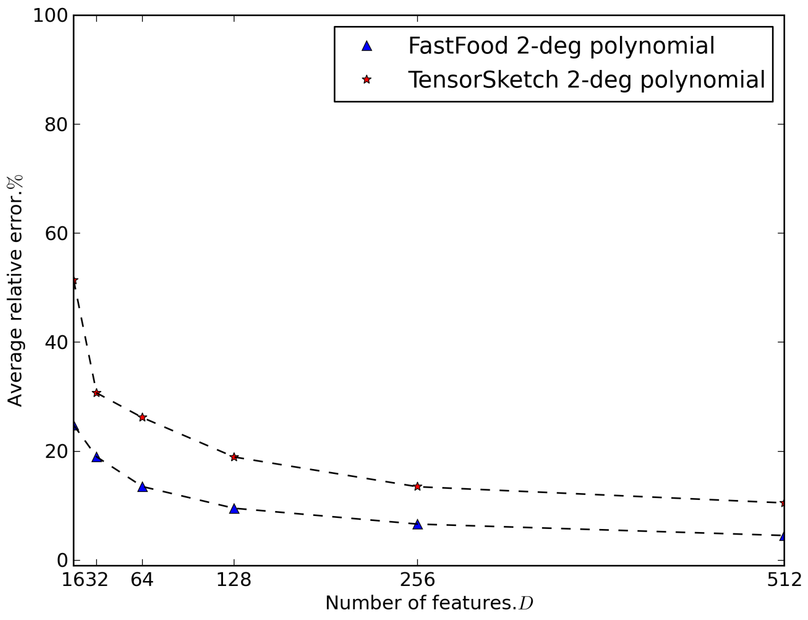


Figure D.1: Fastfood mapping converted. $d = 16, p = 2, \sigma = 2$

In fig.D.1 we converted a Fastfood Gaussian RBF mapping to estimate the polynomial kernel. Unlike the result shown in fig.4.8 we here used information on the original vector lengths of the input. The results indicate that Fastfood does provide good estimates of the directions of vectors.