McKernel: A Library for Approximate Kernel Expansions in Log-linear Time

J. D. Curtó* CURTO@VISION.EE.ETHZ.CH

I. C. Zarza* ZARZA@VISION.EE.ETHZ.CH

Computer Vision Laboratory, Eidgenössische Technische Hochschule Zürich
The Robotics Institute, Carnegie Mellon University
Department of Computer Science and Engineering, The Chinese University of Hong Kong
Department of Electronic Engineering, City University of Hong Kong

F. Yang FENGYANG@GOOGLE.COM

Google Research

A. J. Smola ALEX@SMOLA.ORG

Machine Learning Department, Carnegie Mellon University

L. Van Gool VANGOOL@VISION.EE.ETHZ.CH

Computer Vision Laboratory, Eidgenössische Technische Hochschule Zürich

**Both authors contributed equally.*

Abstract

McKernel[1] is a C++ library for large-scale machine learning. It contains a CPU optimized implementation of the Fastfood algorithm in Le et al. (2013), that allows the computation of approximated kernel expansions in log-linear time. The algorithm requires to compute the product of Walsh Hadamard Transform (WHT) matrices. A cache friendly SIMD Fast Walsh Hadamard Transform (FWHT) that achieves compelling speed and outperforms current state-of-the-art methods has been developed. McKernel allows to obtain non-linear classification combining Fastfood and a linear classifier.

Keywords: Approximate Kernel Expansions, Fast Walsh Hadamard Transform, SIMD, RBF Gaussian kernel, RBF Matern kernel

1. Introduction

Kernel methods offer state-of-the-art estimation performance. They provide function classes that are flexible and easy to control in terms of regularization properties. However, the use of kernels in large-scale machine learning problems has been beset with difficulty. This is because using kernel expansions in large datasets is too expensive in terms of computation and storage. In order to solve this problem, Le et al. (2013) proposed an approximation algorithm, called Fastfood, based on Random Kitchen Sinks by Rahimi and Recht (2007), that speeds up the computation of a large range of kernel functions, allowing us to use them in big data. Recent works on the topic build on Fastfood to propose state-of-the-art deep

[1] McKernel, https://www.github.com/curto2/mckernel/

learning embeddings, Yang et al. (2015) and Hong et al. (2017).

At its heart, McKernel requires scalar multiplications, a permutation, access to trigonometric functions, and two Walsh Hadamard Transforms (WHT) for implementation. The key computational bottleneck here is the WHT. We provide a fast, cache friendly SIMD oriented implementation that outperforms state-of-the-art codes such as Spiral Johnson and Püschel (2000). To allow for very compact distribution of models, we use hash functions and a Pseudorandom Permutation Generator for portability. In this way, for each feature dimension, we only need one floating point number. In summary, our implementation serves as a drop-in feature generator for linear methods where attributes are generated on-the-fly, such as for regression, classification, or two-sample tests. This obviates the need for explicit kernel computations, particularly on large amounts of data.

Outline: We begin with a brief operational description of the feature construction McKernel in Section 2. This is followed by a discussion of the computational issues for a SIMD implementation in Section 3. The concepts governing the API are described in Section 4.

2. McKernel

Kernel methods work by defining a kernel function $k(x, x')$ on a domain $\mathcal{X}$. We can write k as inner product between feature maps, as follows

$$k(x, x') = \langle \phi(x), \phi(x') \rangle \tag{1}$$

for some suitably chosen ϕ. Random Kitchen Sinks (Rahimi and Recht, 2007) approximate this feature mapping ϕ by a Fourier expansion in the case of radial basis function kernels, i.e. whenever $k(x, x') = \kappa(x - x')$. This is possible since the Fourier transform diagonalizes the corresponding integral operator (Smola et al., 1988). This leads to

$$k(x, x') = \int \exp i \langle w, x \rangle \exp -i \langle w, x \rangle d\rho(w) \tag{2}$$

for some L_2 measurable function $\rho(\omega) \geq 0$ that is given by the Fourier transform of κ. Random Kitchen Sinks exploit this by replacing the integral by sampling $\omega_j \sim \rho(\omega)/\left\|\rho\right\|_1$. This allows for finite dimensional expansions but it is costly due to the large number of inner products required. Fastfood resolves this for rotationally invariant κ by providing a fast approximation of the matrix $W := [\omega_1, \ldots \omega_n]$.
This is best seen for the RBF Gaussian kernel where $k(x, x') = \exp\left(-\frac{1}{2\sigma^2}\left\|x - x'\right\|^2\right)$. Since Fourier transforms of Gaussians are Gaussians, albeit with inverse covariance, it follows that $\rho(\omega) \propto \exp\left(-\frac{\sigma^2}{2}\left\|\omega\right\|^2\right)$ and that W contains i.i.d Gaussian random variables. It is this matrix that Fastfood approximates via

$$V := SHG\Pi HB \tag{3}$$

Here S, G and B are diagonal matrices, Π is a random permutation matrix and H is the Hadamard matrix. Whenever the number of rows in W exceeds the dimensionality of the data, we can simply generate multiple instances of V, drawn i.i.d, until the required number of dimensions is obtained.

Diagonal Binary Matrix B This is a matrix with entries $B_{ii} \in \{\pm 1\}$, drawn from the uniform distribution. To avoid memory footprint, we simply use Murmurhash[2] as hash function and extract bits from $h(i, j)$ with $j \in \{0, \ldots N\}$.

Walsh Hadamard Matrix H This matrix is iteratively composed of $H_{2n} = \begin{bmatrix} H_n & H_n \\ -H_n & H_n \end{bmatrix}$. It is fixed and matrix-vector products are carried out efficiently in $O(n \log n)$ time using the Fast Walsh Hadamard Transform (FWHT). We will discuss implementation details for a fast variant below.

Permutation Matrix Π We generate a random permutation using the Fisher-Yates shuffle. That is, given a list $L = \{1, \ldots n\}$ we generate permutations recursively as follows: pick a random element from L. Use this as the image of n and move n to the position where the element was removed. The algorithm runs in linear time and its coefficients can be stored in $O(n)$ space. Moreover, to obtain a deterministic mapping, replace the random number generator with calls to the hash function.

Gaussian Scaling G This is a diagonal matrix with i.i.d Gaussian entries. We generate the random variates using the Box-Muller transform Box and Muller (1958) while substituting the random number generator by calls to the hash function to allow us to recompute the values at any time without the need to store random numbers.

Diagonal Scaling S This is a random scaling operator whose behavior depends on the type of kernel chosen, such as the Matern kernel, the RBF Gaussian kernel or any other radial spectral distribution, Yang et al. (2014).

3. FWHT Implementation

A key part of the library is a FWHT efficient implementation. In particular, McKernel offers considerable improvement over the Spiral[3] library, due to automatic code generation, the use of SIMD intrinsics (SSE2 using 128 bit registers) and loop unrolling. This decreases the memory overhead.

McKernel proceeds with vectorized sums and subtractions iteratively for the first $\frac{n}{2^k}$ input vector positions (where n is the length of the input vector and k the iteration starting from 1), computing the intermediate operations of the Cooley-Tukey algorithm till a small Hadamard routine that fits in cache. Then the algorithm continues in the same way but starting from the smallest length and doubling on each iteration the input dimension until the whole FWHT is done in-place.

For instance, on an intel i5-4200 CPU @ 1.60GHz laptop the performance obtained can be observed in Figure 1.

[2] https://code.google.com/p/smhasher/

[3] http://www.spiral.net

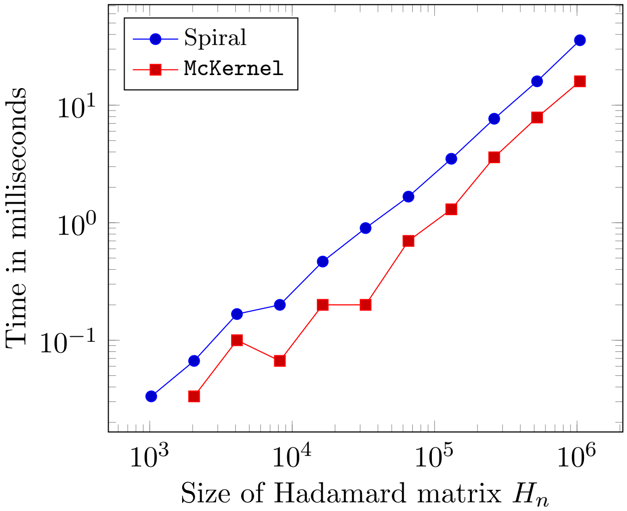

Figure 1: **FWHT Comparison**. McKernel (red) outperforms Spiral (blue) across the range of arguments.

Our code outperforms Spiral by a factor of 2 consistently throughout the range of arguments. Furthermore, Spiral needs to precompute trees and by default can only perform the computation up to matrix size $n = 2^{20}$. On the other hand, our implementation works for any size since it computes the high-level partitioning dynamically.

4. API Description

The API follows the factory design pattern. That is, while the McKernel object is fairly generic in terms of feature computation, we have a factory that acts as a means of instantiating the parameters according to pre-specified sets of parameters for e.g. a RBF Gaussian or a Matern kernel. The so-chosen parameters are deterministic, given by the values of a hash function. The advantage of this approach is that there is no need to save the coefficients generated for McKernel when deploying the functions. That said, we also provide a constructor that consumes B, Π, G and S to allow for arbitrary kernels.

```
void fwht(float* data, unsigned long lgn)
```

Computes the FWHT in-place on `data`. The length of the vector it operates on is given by $2^{\texttt{lgn}}$. Note that `fwht` does not check whether `data` points to a sufficiently large amount of memory.

4.1 Customizing McKernel

For instance, to generate each S_{ii} entry for Matern kernel we draw t iid samples from the n-dimensional unit ball S_d, add them and compute its Euclidean norm. To draw efficiently samples from S_d we use the algorithm provided below.

Let $\mathbf{X} = (X_1, \ldots, X_n)$ be a vector of iid random variables drawn from $\mathcal{N}(0,1)$, and let $||\mathbf{X}||$ be the Euclidean norm of $\mathbf{X}$, then $\mathbf{Y} = (\frac{X_1}{||\mathbf{X}||}, \ldots, \frac{X_n}{||\mathbf{X}||})$ is uniformly distributed over the n-sphere, where it is obvious to see that $\mathbf{Y}$ is the projection of $\mathbf{X}$ onto the surface of the n-dimensional sphere. To draw uniform random variables in the n-ball, we multiply $\mathbf{Y}$ by $U^{1/n}$ where $U \sim U(0,1)$. This can be proved as follows: Let $\mathbf{Z} = (Z_1, \ldots, Z_n)$ be a random vector uniformly distributed in the unit ball. Then, the radius $R = ||\mathbf{Z}||$ satisfies $\mathbb{P}(R \leq r) = r^n$. Now, by the inverse transform method we get $R = U^{1/n}$.Therefore to sample uniformly from the n-ball the following algorithm is used: $\mathbf{Z} = rU^{1/n}\frac{\mathbf{X}}{||\mathbf{X}||}$.

4.2 Class McKernel

Class instance McKernel initializes the object by computing and storing B and Π diagonal matrices. The Factory design lets you specify the creation of the matrix G and S by calling the proper kernel constructor.

Objects McKernel contain the following methods:

- `McFeatures(float* data, float* features)` computes the features by using the real version of the complex feature map ϕ in Rahimi and Recht (2007), $\phi'(x) = n^{-\frac{1}{2}}\{\cos([Vx]_j), \sin([Vx]_j)\}$. SIMD vectorized instructions and cache locality are used to increase speed performance. To avoid a bottleneck in the basic trigonometric function computation a SSE2 sincos function[4] is used. These allow an speed improvement of 18x times for a 2^{24} dimension input matrix.

- `float McEvaluate (float* data, float* weights)` computes the inner product between the input vector and the weights.

4.3 Practical Use

McKernel can be embedded in front of a linear classifier. For instance, first doing feature extraction, then McKernel and finally SVM Linear Classification. Another example is to embedded it in a deep neural network architecture either as non-linear mapping to the activation function, or as a separate layer embedding Hong et al. (2017).

[4] http://gruntthepeon.free.fr/ssemath/

References

G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *In: The Annals of Mathematical Statistics*, 1958.

W. Hong, J. Yuan, and S. D. Bhattacharjee. Fried binary embedding for high-dimensional visual features. *In: CVPR*, 2017.

J. Johnson and M. Püschel. In search of the optimal walsh-hadamard transform. *In: IEEE*, 2000.

Q. Le, T. Sarlós, and A. Smola. Fastfood - approximating kernel expansions in loglinear time. *In: ICML*, 2013.

A. Rahimi and B. Recht. Random features for large-scale kernel machines. *In: NIPS*, 2007.

A. J. Smola, B. Schölkopf, and K. R. Müller. General cost functions for support vector regression. *In: Ninth Australian Conference on Neural Networks*, 1988.

Z. Yang, A. J. Smola, L. Song, and A. G. Wilson. À la carte - learning fast kernels. *In: AISTATS*, 2014.

Z. Yang, M. Moczulski, M. Denil, N. de Freitas, A. Smola, L. Song, and Z. Wang. Deep fried convnets. *In: ICCV*, 2015.