Estimating the inverse trace using random forests on graphs

Simon BARTHELMÉ[1], Nicolas TREMBLAY[1], Alexandre GAUDILLIÈRE[2], Luca AVENA[3], Pierre-Olivier AMBLARD[1]

[1]Univ Grenoble-Alpes, CNRS, Grenoble-INP, GIPSA-lab, Grenoble, France

[2]Aix-Marseille Univ, CNRS, I2M, Marseille, France

[3]Leiden University, Netherlands

prenom.nom@gipsa-lab.fr, alexandre.gaudilliere@math.cnrs.fr, l.avena@math.leidenuniv.nl

Résumé – Dans certains problèmes de traitement de données, il est nécessaire de calculer la trace de l'inverse d'une matrice, de la forme $\mathrm{Tr}(q\mathbf{I}+\mathbf{L})^{-1}$. Si la matrice est de dimension trop élevée, les méthodes directes deviennent trop coûteuses et il faut se contenter de méthodes approchées, comme celles fondées sur l'estimateur de Girard (aussi connu sous le nom d'estimateur de Hutchinson). Dans cet article, nous proposons une méthode alternative basée sur l'algorithme de Wilson, initialement développé pour tirer des arbres couvrants uniformes. Cette méthode est applicable à de très grandes matrices, rapide, et facile à implémenter. En revanche, elle est uniquement adaptée aux matrices $\mathbf{L}$ qui sont diagonalement dominantes.

Abstract – Some data analysis problems require the computation of (regularised) inverse traces, i.e. quantities of the form $\mathrm{Tr}(q\mathbf{I}+\mathbf{L})^{-1}$. For large matrices, direct methods are unfeasible and one must resort to approximations, for example using a conjugate gradient solver combined with Girard's trace estimator (also known as Hutchinson's trace estimator). Here we describe an unbiased estimator of the regularized inverse trace, based on Wilson's algorithm, an algorithm that was initially designed to draw uniform spanning trees in graphs. Our method is fast, easy to implement, and scales to very large matrices. Its main drawback is that it is limited to diagonally dominant matrices $\mathbf{L}$.

1 Background

Monte Carlo methods are increasingly popular in large-scale linear algebra problems [14]. Among the many different quantities one may need to compute on large matrices, spectral summaries of the form $\sum_{i=1}^{n} f(\lambda_i(\mathbf{L}))$, where the λ_i's are the eigenvalues of $\mathbf{L}$ and f is some function, are often required. Here we focus on the following quantity :

$$s(q) = q\,\mathrm{Tr}((\mathbf{L}+q\mathbf{I})^{-1}) = \sum_{i=1}^{n} \frac{q}{\lambda_i + q} \tag{1}$$

which we seek to evaluate for real $q > 0$. We call the quantity $s(q)$ because it is equivalent (up to scaling) to the Stieltjes transform of the eigenvalue density evaluated on the negative real axis [1].

In practice, the problem of estimating efficiently $s(q)$ may arise when looking for the optimal regularization parameter in a regularized optimization problem. Say we measure a signal $\mathbf{x} = [x_1, \ldots, x_n]^t$ under white Gaussian noise ϵ. The measurements read $y_i = x_i + \epsilon_i$ for $i = 1$ to n. Many estimation methods (smoothing splines, semi-supervised learning, Gaussian process regression) define an estimator of $\mathbf{x}$ as :

$$\hat{\mathbf{x}} = \underset{\mathbf{z}\in\mathbb{R}^n}{\mathrm{argmin}}\; q\,||\mathbf{y}-\mathbf{z}||^2 + \frac{1}{2}\mathbf{z}^t\mathbf{L}\mathbf{z} \tag{2}$$

where $\mathbf{L}$ is a semi-definite positive matrix defining the penalty (regularisation) term, and q parametrizes the regularisation's strength. The solution to this optimisation problem equals :

$$\hat{\mathbf{x}} = q(q\mathbf{I}+\mathbf{L})^{-1}\mathbf{y}. \tag{3}$$

In most cases the optimal value of q is unknown and must be estimated, for instance using AIC (Akaike's Information Criterion) or Generalised-Cross Validation (GCV). AIC requires computing the number of degrees of freedom of the estimator, which here can be taken to equal $s(q)$ (see [10], ch. 5, [9, 8]).

The simplest solution to compute eq. (1) is of course to compute the eigenvalues of $\mathbf{L}$, which comes at $\mathcal{O}(n^3)$ cost if $\mathbf{L}$ is $n \times n$. Moreover, there is no particular gain to expect from the sparsity of $\mathbf{L}$. In fact, iterative methods for eigenvalues, that look to estimate the smallest or largest eigenvalues of $\mathbf{L}$, cannot be used directly here, as $s(q)$ involves the whole spectral density. An alternative is to consider Monte Carlo methods. A famous estimator for the trace of a matrix was first suggested by Girard in [9] : let $\mathbf{r}$ denote a length-n vector of independent, standard Gaussian entries. Let $\mathbf{M}$ denote a $n \times n$ matrix. Then :

$$\mathbb{E}(\mathbf{r}^t\mathbf{M}\mathbf{r}) = \mathbb{E}\left(\mathrm{Tr}(\mathbf{M}\mathbf{r}\mathbf{r}^t)\right) = \mathrm{Tr}\left(\mathbf{M}\mathbb{E}(\mathbf{r}\mathbf{r}^t)\right) = \mathrm{Tr}\,\mathbf{M} \tag{4}$$

This leads immediately to estimating $\mathrm{Tr}\,\mathbf{M}$ using the empirical mean $\mathrm{Tr}\,\mathbf{M} \approx \frac{1}{k}\sum_{l=1}^{k}\mathbf{r}_l^t\mathbf{M}\mathbf{r}_l$. Note that eq. (4) is valid for any random vector with diagonal covariance, so we may use other random vectors [12]. Various options have been studied in the literature, see [5]. In this work we use Gaussian vectors for simplicity (as we will see, it is not the main factor here).

In our case, $\mathbf{M} = q(q\mathbf{I}+\mathbf{L})^{-1}$, and Girard estimator of $s(q)$ reads

$$\hat{s}_k^{\mathrm{G}}(q) = \frac{q}{k}\sum_{l=1}^{k}\mathbf{r}_l^t(q\mathbf{I}+\mathbf{L})^{-1}\mathbf{r}_l. \tag{5}$$

In the Gaussian case, the variance of the estimation for $k = 1$ (see, e.g., lemma 9 of [5]) is :

$$\mathrm{Var}(\hat{s}_1^{\mathrm{G}}(q)) = \sum_{i=1}^{n}\frac{2q^2}{(q+\lambda_i)^2}. \tag{6}$$

We still need to figure out how to compute the quadratic forms $\mathbf{r}^t(q\mathbf{I}+\mathbf{L})^{-1}\mathbf{r}$ in eq. (5). This involves solving a large linear system, a task for which algorithms abound. If $\mathbf{L}$ is sparse, computing a sparse Cholesky factor will give good results for many systems, up to a certain size [1]. Alternatively, for very large systems, iterative solvers such as Conjugate Gradients may be used [6]. Another approach is to use an order p polynomial approximation [2] of the function $f(x) = q/(q+x) \simeq \sum_{j=0}^{p} \alpha_j x^j$. Estimating $\mathbf{r}^t(q\mathbf{I}+\mathbf{L})^{-1}\mathbf{r}$ then boils down to computing $\mathbf{r}^t \sum_{j=0}^{p} \alpha_j \mathbf{L}^j \mathbf{r}$, that is : p matrix vector multiplications and one scalar product. Iterative solvers and polynomial methods only provide approximate solutions, but we expect the error induced by these approximations to be small relative to the Girard variance of Eq. (6). A combination of Girard's trace estimator and iterative solvers has been used, e.g., in [18].

Below, we describe an alternative method that is very natural and intrinsic when $\mathbf{L}$ is actually a graph Laplacian, a particular class of matrices associated with graphs. At the end of section 2 we extend the technique to diagonally dominant matrice, *i.e.* the set of matrices that verify $\forall i \quad L_{ii} \geq \sum_{j \neq i} |L_{ij}|$.

2 Uniform spanning trees, random forests, and inverse traces

In this section we recall some facts on graphs and spanning trees that should help understand our method. Mathematical details can be found in [2] and [3].

Consider a weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of $n = |\mathcal{V}|$ nodes and $|\mathcal{E}|$ edges. We restrict ourselves to undirected graphs in this paper, even though the results may be extended to strongly connected [3] directed graphs. We denote by $\mathbf{A} \in \mathbb{R}^{n \times n}$ the graph's adjacency matrix, where $A_{ij} = A_{ji} \geq 0$ is the weight of the connection between nodes i and j. The graph Laplacian of $\mathcal{G}$ equals $\mathbf{L} = \mathbf{D} - \mathbf{A} \in \mathbb{R}^{n \times n}$, where $\mathbf{D} = \mathrm{diag}(d_1, \ldots, d_n) \in \mathbb{R}^{n \times n}$ is the diagonal degree matrix with $d_i = \sum_j A_{ij}$ the degree of node i. The graph Laplacian is a fascinating object with many applications in machine learning and graph signal processing, see eg. [7].

A tree is a cycle-free graph, and a spanning tree $\mathcal{T}$ of $\mathcal{G}$ is a cycle-free connected subgraph of $\mathcal{G}$ that spans all n nodes of $\mathcal{G}$. A typical graph has more than one spanning tree. For instance, the complete graph of size n contains n^{n-2} different spanning trees. A tree sampled uniformly from the set of all spanning trees of $\mathcal{G}$ is called a uniform spanning tree (UST).

A fast algorithm for sampling USTs, now known as "Wilson's algorithm" was developed in [19] . In a nutshell, the algorithm runs as follows : pick a node at random, and call it the root of the tree. Now pick another node, and run a random walk until it hits the root. The trajectory of the random walk may include loops : we simply erase them as they come. The resulting "loop-erased" random walk will form the first branch of the spanning tree. Next, pick a node that is not yet in the tree, run a random walk until it hits the tree, erase the possible loops, add this new branch to the tree, etc. Wilson's algorithm runs in time proportional to $\mathcal{O}(\tau)$ where τ is the average "commute time" : the time it takes a random walk to reach node j starting from node i for two nodes picked uniformly on the graph.

Algorithm 1 A variant of Wilson's algorithm

Input : A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of size n and $q > 0$
$\mathcal{R} \leftarrow \emptyset$, $\mathcal{W} \leftarrow \emptyset$
Add a node, called Δ, to $\mathcal{G}$ and connect it to all n nodes in $\mathcal{V}$ with edges of weight q. Call this augmented graph $\mathcal{G}'$.
while $\mathcal{W} \neq \mathcal{V}$ **do** :
· Do a random walk on $\mathcal{G}'$ starting from any node $i \in \mathcal{V} \setminus \mathcal{W}$ until it reaches either Δ, or a node in $\mathcal{W}$.
· Erase all the loops of the trajectory, in the order of appearance.
· Add all the nodes of this loop-erased trajectory to the set of visited nodes $\mathcal{W}$.
if the last node of the trajectory is Δ **do** :
· Denote by l the last visited node before Δ
· $\mathcal{R} \leftarrow \mathcal{R} \cup \{l\}$
Output : $\mathcal{R}$, the root set of the sampled forest spanning $\mathcal{G}$.

Wilson in [19] noted that his algorithm could be used to generate random spanning forests, and not just USTs. A forest is a set of trees, and a spanning forest is a set of disjoint trees that, taken together, span the whole graph. The algorithm [4] is given as alg. 1 : it uses loop-erased random walks (LERW), but these LERWs may be interrupted early. At each node, the random walk is interrupted with probability $\frac{q}{q+d_i}$. Of course, the larger q, the shorter the walks, the larger the number of roots, the faster the algorithm. In the implementation given in alg. 1, the average runtime is [5] $\mathcal{O}(|E|/q)$.

The resulting process has many fascinating aspects, some of which have been investigated in [4]. For our purposes, we focus on the fact that the number of roots is in fact an unbiased estimator of $s(q)$:

$$\mathbb{E}(|\mathcal{R}|) = \sum_{i=1}^{n} \frac{q}{q+\lambda_i} = s(q). \tag{7}$$

This suggests to define Wilson estimator of s as :

$$\hat{s}_k^{\mathbb{W}}(q) = \frac{1}{k} \sum_{l=1}^{k} |\mathcal{R}_l|. \tag{8}$$

where the k sets of roots $\{\mathcal{R}_l\}_{l=1,\ldots,k}$ are obtained by running alg. 1 k times. A further property of alg. 1 is, in the case $k = 1$ (see [4]) :

$$\mathrm{Var}(\hat{s}_1^{\mathbb{W}}(q)) = q \sum_{i=1}^{n} \frac{\lambda_i}{(q+\lambda_i)^2}. \tag{9}$$

This variance can be compared with Girard's (eq. (6)) : we see that for both very small and very large values of q, Girard's estimator is less effective per sample. Unfortunately, identifying exactly the interval of q for which Wilson's estimator is preferable (on a per-sample basis) is heavily dependent on the eigenvalue distribution.

Since $\mathrm{Var}(\hat{s}_1^{\mathbb{W}}) \leq \mathbb{E}(\hat{s}_1^{\mathbb{W}})$ and $s(q) \geq 1$ the relative error verifies :

$$\frac{\mathrm{Var}(\hat{s}_k^{\mathbb{W}})}{\mathbb{E}(\hat{s}_k^{\mathbb{W}})^2} = \frac{\mathrm{Var}(\hat{s}_1^{\mathbb{W}})}{k\mathbb{E}(\hat{s}_1^{\mathbb{W}})^2} \leq \frac{1}{ks(q)} \leq \frac{1}{k}. \tag{10}$$

1. In fact, if the Cholesky factor is available, the Takahashi equations may also be used to obtain the trace, see [16]

2. Using Chebychev polynomials for instance if one wants to ensure the smallest infinite-norm error : $\sup_{x \in [0, \lambda_{\max}]} |f(x) - \sum_{j=0}^{p} \alpha_j x^j|$

3. Given any pair of nodes (i, j), there is a directed path to go from i to j, *and* from j to i.

4. Alg. 1 is written in order to only output the set of roots of the sampled forest, as this is the information we will use in this paper. Much more information can in practice be extracted.

5. This figure assumes that, when at node i, picking a neighbour at random is $\mathcal{O}(d_i)$. This can be marginally improved by some preprocessing tricks, for example by using the alias method for sampling. In addition, in the case of unweighted graphs there is no dependency on the degree (picking a random neighbor is $\mathcal{O}(1)$)

Let us point out several advantages of the suggested algorithm. First, no preprocessing is required. The graph does even not need to be precomputed : essentially, all we need is the ability to run a random walk on the graph. Second, it is very easy to implement (our implementation runs under 20 lines of Julia code). Third, it is is easy to parallelise, as we can just generate several forests concurrently. Fourth, its memory footprint is minimal, requiring a handful of $\mathcal{O}(n)$ quantities. However, the main disadvantage is that the algorithm can only estimate $s(q)$ if $\mathbf{L}$ is a graph Laplacian. The next section partly lifts that restriction to allow the use of diagonally-dominant matrices.

Generalising to diagonally-dominant matrices. We borrow a trick from the rich literature on Laplacian solvers (see for instance [13, 11]). Let $\mathbf{G}$ be a diagonally dominant matrix, that we decompose as $\mathbf{G} = \mathbf{D}_1 + \mathbf{D}_2 + \mathbf{A}_p + \mathbf{A}_n$ where :

— $\mathbf{A}_p$ contains the positive off-diagonal elements, $\mathbf{A}_n$ contains the negative ones

— $\mathbf{D}_1$ is a diagonal matrix, with $D_1(i,i) = \sum_{j \neq i} |G_{ij}|$ (sum of off-diagonal elements)

— $\mathbf{D}_2$ is also diagonal, with entries $D_2(i,i) = G_{ii} - D_1(i,i)$. Diagonal dominance of $\mathbf{G}$ implies that $\forall i, D_2(i,i) \geq 0$.

In the same way we restricted the previous discussion to undirected graphs, we here restrict ourselves to symmetric diagonally dominant matrices, implying that $\mathbf{A}_p$ and $\mathbf{A}_n$ are symmetric. We form the following two graph Laplacians, both representing undirected weighted graphs, and of respective size n and $2n$:

$$\mathbf{L}_1 = \mathbf{D}_1 + \mathbf{A}_n - \mathbf{A}_p \tag{11}$$

$$\mathbf{L}_2 = \begin{pmatrix} \mathbf{D}_1 + \mathbf{D}_2/2 + \mathbf{A}_n & -\mathbf{D}_2/2 - \mathbf{A}_p \\ -\mathbf{D}_2/2 - \mathbf{A}_p & \mathbf{D}_1 + \mathbf{D}_2/2 + \mathbf{A}_n \end{pmatrix}. \tag{12}$$

It can be easily verified that an eigenvector basis for $\mathbf{L}_2$ can be constructed as follows : n eigenvectors of the form $\begin{pmatrix} \mathbf{x} \\ \mathbf{x} \end{pmatrix}$, where $\mathbf{x}$ is an eigenvector of $\mathbf{L}_1$; and n other eigenvectors of the form $\begin{pmatrix} \mathbf{y} \\ -\mathbf{y} \end{pmatrix}$, where $\mathbf{y}$ is an eigenvector of $\mathbf{G}$. This implies that $\lambda(\mathbf{L}_2) = \lambda(\mathbf{L}_1) \cup \lambda(\mathbf{G})$ and consequently that :

$$s_{\mathbf{G}}(q) = s_{\mathbf{L}_2}(q) - s_{\mathbf{L}_1}(q). \tag{13}$$

Given eq. (13), the extension to symmetric diagonally dominant matrices is thus straightforward : form the two Laplacians $\mathbf{L}_1$ and $\mathbf{L}_2$, run the algorithm on each graph, and subtract.

3 Empirical results

We implemented our algorithm in the Julia programming language [6], and compared its performance on a number of graphs to alternatives based on Girard's estimator. We ran all algorithms on a single core on a desktop PC. Specifically, the alternative algorithms are as follows. First generate k Gaussian vectors of size n, of zero mean and variance 1, then compute $\hat{s}_k^{\mathcal{G}}(q) = (q/k) \sum_{l=1}^{k} \mathbf{r}_l^t (\mathbf{L} + q\mathbf{I})^{-1} \mathbf{r}_l^t$ using one of the following methods :

1. `direct` : use Julia's backslash operator (which calls CHOLMOD internally)
2. `amg` : Algebraic Multigrid (AMG) with Ruge-Stüben coarsening [17], implemented in the AlgebraicMultigrid package [7]
3. `cg` : Conjugate Gradients : we used the implementation in the IterativeSolvers.jl package [8], with diagonal preconditioning
4. `cg-amg` : same as above, with AMG preconditioning

All methods defined here are based on Monte Carlo, and have an asymptotic relative error of $\epsilon^2 = \text{Var}(\hat{s}_1)/k$. In order to ensure a fair comparison, we report effective runtimes as the time needed per iteration multiplied by the number of iterations needed in order to reach a fixed relative error ϵ. For each value of q, we run each method 100 times on each graph. This gives us an estimate $\hat{s}_{100}(q)$, along with an estimated standard deviation $\hat{\sigma}_{s(q)}$. The asymptotic relative error is given by :

$$\epsilon = \frac{\hat{\sigma}_{s(q)}}{\hat{s}(q)\sqrt{k}}. \tag{14}$$

We solve for k given a relative error of $\epsilon = 0.02$. The time per iteration is then computed as the total time divided by 100. We note that this tends to be unfavourable to our method, which has zero set-up time, unlike the direct method (which needs to compute a decomposition) or AMG (which needs to setup the preconditioner).

Recall that $1 \leq s(q) \leq n$, where n is the number of nodes of the graph, and that $s(q)$ is the average number of roots alg. 1 outputs. Generally, the higher $s(q)$ is, the faster our algorithm. $s(q)$ will of course vary depending on the graph, and so in the comparisons we pick a range that is appropriate for each graph. We set the range such that $s(q)$ would vary approximately between 1% and 50% of n, the number of nodes. We picked 8 values on a logarithmic scale.

The graphs we tested are as follows :

— "circle" : a ring graph of size $27,000$

— "grid_2d" : a 2D lattice of size $164 \times 164 = 26,896$

— "grid_3d" : a 3D lattice of size $30^3 = 27,000$

— "barabasi_albert" : A Barabasi-Albert random graph with $n = 3000$ and $k = 30$ (average degree)

— "noisy_heart" : a k-nearest neighbour graph obtained from $n = 4096$ points sampled from the parametric surface $x = \sin(\theta)\cos(\phi), \sin(\theta)y = \sin(\phi)(1 + \exp(-0.1\theta)), z = \cos(\theta)(0.1 + \theta)$ for $\theta \in [0, \pi], \phi \in [0, 2\pi]$. We added a small random Gaussian offset to each point, and the surface looks heart-shaped when plotted, hence the name.

Results are shown in Fig. 1. We plot run-time as a function of $s(q)$, to ease comparison across graphs. Our method is competitive compared to a direct solver for a range of values of q. Iterative methods make a relatively poor showing here, but they are expected to scale better with n. Also, we need to solve for several right-hand sides, and block CG methods may be more appropriate [15]. Finally, we have also checked that our algorithm scales to very large graphs. On a Barabasi-Albert random graph of size $n = 1,000,000$ and 40 links per node, running our algorithm even at low $q = 6 \cdot 10^{-3}$ (corresponding to $s(q) \approx 100$) takes a very reasonable 1/5 sec per realisation.

4 Discussion

Random forests on graphs lead to simple estimators for inverse traces of diagonally dominant matrices, and we find good practical performance. The small memory footprint is especially notable (all quantities stored scale in $\mathcal{O}(n)$). There are also several promising avenues for improvement. In many scenarios, what is needed is to evaluate $s(q)$ for a range of values of q, and the "coupled forests" algorithm of [3] can be use to directly estimate $s(q)$ over a range much

6. `julialang.org`
7. https ://github.com/JuliaLinearAlgebra/AlgebraicMultigrid.jl
8. https ://juliamath.github.io/IterativeSolvers.jl/dev/

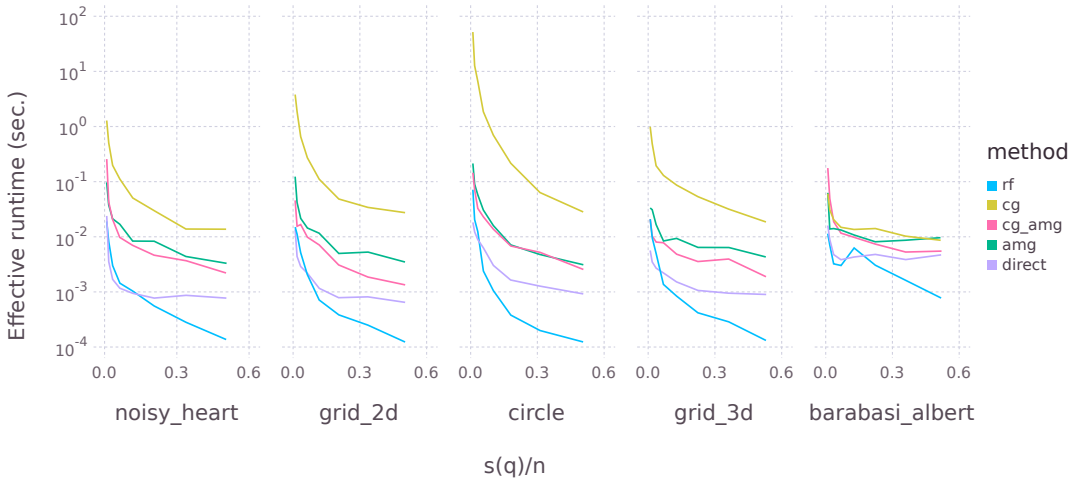

FIGURE 1 – Runtime of the proposed method ("`rf`") compared to alternatives, on 5 graphs. See text for details.

more cheaply than by running independent forests for a grid of values. The method can also be extended to estimate the values on the diagonal of $(\mathbf{L} + q\mathbf{I})^{-1}$, a refinement we will describe in future work.

Références

[1] Greg W Anderson, Alice Guionnet, and Ofer Zeitouni. *An introduction to random matrices*, volume 118. Cambridge university press, 2010.

[2] L Avena and A Gaudilliere. On some random forests with determinantal roots. *arXiv preprint arXiv :1310.1723*, 2013.

[3] L. Avena and A. Gaudillière. Two Applications of Random Spanning Forests. *Journal of Theoretical Probability*, July 2017.

[4] Luca Avena, Fabienne Castell, Alexandre Gaudillière, and Clothilde Mélot. Random forests and networks analysis. *Journal of Statistical Physics*, 173(3-4) :985–1027, 2018.

[5] Haim Avron and Sivan Toledo. Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix. *Journal of the ACM*, 58(2) :1–34, April 2011.

[6] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems : building blocks for iterative methods*, volume 43. Siam, 1994.

[7] Fan RK Chung and Linyuan Lu. *Complex graphs and networks*, volume 107. American mathematical society Providence, 2006.

[8] A Girard. A fast 'monte-carlo cross-validation'procedure for large least squares problems with noisy data. *Numerische Mathematik*, 56(1) :1–23, 1989.

[9] Didier Girard. Un algorithme simple et rapide pour la validation croisée généralisée sur des problèmes de grande taille. Technical report, 1987.

[10] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning*. Springer, 2009.

[11] Timothy Hunter, Ahmed El Alaoui, and Alexandre Bayen. Computing the log-determinant of symmetric, diagonally dominant matrices in near-linear time. *arXiv preprint arXiv :1408.1693*, 2014.

[12] Michael F Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 19(2) :433–450, 1990.

[13] Jonathan A Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. A simple, combinatorial algorithm for solving sdd systems in nearly-linear time. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 911–920. ACM, 2013.

[14] Michael W Mahoney et al. Randomized algorithms for matrices and data. *Foundations and Trends® in Machine Learning*, 3(2) :123–224, 2011.

[15] Dianne P O'Leary. The block conjugate gradient algorithm and related methods. *Linear algebra and its applications*, 29 :293–322, 1980.

[16] Havard Rue and Leonhard Held. *Gaussian Markov random fields : theory and applications*. Chapman and Hall/CRC, 2005.

[17] John W Ruge and Klaus Stüben. Algebraic multigrid. In *Multigrid methods*, pages 73–130. SIAM, 1987.

[18] Michael L Stein, Jie Chen, Mihai Anitescu, et al. Stochastic approximation of score functions for gaussian processes. *The Annals of Applied Statistics*, 7(2) :1162–1191, 2013.

[19] David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the Twenty-eighth Annual ACM Symposium on the Theory of Computing (STOC)*, volume 96, pages 296–303. Citeseer, 1996.