Numerical Computing Cheatsheet

Edoardo Riggio

January 13, 2022

Numerical Computing - SA. 2021
Computer Science
Università della Svizzera Italiana, Lugano

Contents

1 PageRank Algorithm

The PageRank algorithm is entirely determined by the link structure of the World Wide Web. A page will thus have a high rank if other pages with high ranks link to it.

1.1 Random Surfer Model

This algorithm is based on the **Random Surfer Model**. Here we imagine a user going from one page to the other by randomly choosing an outgoing link from one page to the next. This process is also called **exploitation**.

The problem with exploitation is that it could lead to dead-ends – i.e. a page with no outgoing links, or cycles around cliques of interconnected pages. For this reason, we sometimes choose a random page from the web to navigate to. This process is called **exploration**.

1.2 Markov Chains

The random walk generated by the combination of exploitation and exploration is known as a **Markov Chain**. A Markov Chain – or Markov Process – is a stochastic process. Differently from other stochastic processes, it has the property of being memory-less. This means that the probability of future states are not dependent upon the steps that led up to the present state.

1.2.1 Markov Matrix

Let W be the set of webpages that can be reached by a Markov Chain of hyperlinks, n the number of pages in W, and G the $n \times n$ connectivity matrix of a portion of the Web. The matrix G will be composed as follows:

$$g_{ij} = \begin{cases} 1 & \text{if there is a hyperlink from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

From matrix G we can determine the in-degree and out-degree of a page j. This can be computed as follows:

$$r_i = \sum_j g_{ij}$$

$$c_J = \sum_i g_{ij}$$

Where r_i is the in-degree and c_j is the out-degree. Let now p be the probability that the random walk follows a link – i.e. performs exploitation. A typical value for p is 0.85. Let δ be the probability that a particular random page is chosen,

and $1 - p$ is the probability that some random page is chosen – i.e. perform exploration. Then δ will have the following formulation:

$$\delta = \frac{1-p}{n}$$

Now let A be a matrix that comes from scaling G by its column sums. The elements of matrix A will be:

$$a_{ij} = \begin{cases} \rho \cdot \frac{g_{ij}}{c_j} + \delta & \text{if } c_j \neq 0 \\ \frac{1}{n} & \text{if } c_j = 0 \end{cases}$$

Matrix A is the transition probability matrix for the Markov Chain – for this reason this matrix is also known as the Markov matrix. All of its elements are strictly between 1 and 0, and its column sums are all equal to 1.

1.3 Eigenvectors and Eigenvalues

1.3.1 Eigenvector

An **eigenvector** is a non-zero vector that changes at most by a scalar factor when a linear transformation is applied to it.

1.3.2 Eigenvalue

An **eigenvalue** is the factor by which the eigenvector is scaled. Thus, this relation holds true:

$$A\overrightarrow{v} = \lambda\overrightarrow{v}$$

Where A is the transformation matrix, $\overrightarrow{v}$ is the eigenvector, and λ is the eigenvector.

1.3.3 Eigenbasis

An **eigenbasis** is the basis vector space which consists entirely of eigenvectors. Mathematically, this is represented by a diagonal matrix in which all of its columns are eigenvectors. This basis is not always computable.

1.3.4 Rayleigh Quotient

The **Rayleigh Quotient** is a simple way to find – given an eigenvector – its corresponding eigenvalue. The following is the quotient:

$$\mu(v) = \frac{v^T A v}{v^T v}$$

Where v is the vector for which we need to compute the eigenvalue, and A is the transformation matrix. One thing to note, is that if the vector v is an eigenvector, then $\mu(v)$ will be the corresponding eigenvalue. On the other hand, if v is not an eigenvector, then $\mu(v)$ will be the eigenvalue of the closest eigenvector of v.

This quotient is useful in the computation of the dominant eigenvalue. This is done by the power iteration and by the inverse iteration.

1.4 Power Iteration

The **power iteration** is used in order to compute the dominant eigenvector λ_1 of a matrix A. In order to find it, we need to follow this algorithm:

1. Start with an initial guess v_0, which is a vector
2. Compute w, s.t. $w = Av_k$
3. Set $v_{k+1} = ||w||$
4. Find the eigenvalue of v_{k+1} by using the Rayleigh quotient
5. Repeat steps 2 to 4 until the difference between the previous and current eigenvectors is under a certain threshold. Increment k by 1

1.5 Inverse Iteration

The **inverse iteration** is very similar to the power method. In the case of the inverse iteration, we can use it to find a particular eigenvalue – which is not necessarily the dominant one.

This can be done by applying the power method to $(A - \alpha I)^{-1}$ instead of A. By doing so, the α constant will contribute in making the non-dominant eigenvector the dominant one.

This algorithm converges much faster than the previous one. On the other hand, it is also more computationally expensive than the previous. In this case we will need to compute a system of equations for every step of the algorithm.

$$(A - \alpha I)^{-1}\overrightarrow{v} = \frac{1}{\lambda - \alpha}\overrightarrow{v} \quad \Longleftrightarrow \quad (A - \alpha I)x_{n+1} = x_n$$

2 Spectral Graph Partitioning

In order to find a partition in a graph, we need to use the second smallest eigenvector – i.e. eigenvector v_2. This eigenvector is also known as the **Fiedler eigenvector**, and plays a very important role in graph partitioning. This is

because all of the indices corresponding to vector entries can be divided as follows:

$$v_{2i} \in \begin{cases} \text{Set 1} & \text{if } v_{2i} > 0 \\ \text{Set 2} & \text{if } v_{2i} < 0 \end{cases}$$

The arising partition minimizes the number of edges between the two sets. The second smallest eigenvalue λ_2 is greater than 0 iff we are considering a connected graph. The magnitude of λ_2 gives us an indication on how well-connected the overall graph is.

2.1 Laplacian Matrix

The **Laplacian Matrix** is the matrix representation of a graph. It is computed by doing the following:

$$L = D - A$$

Where A is the adjacency matrix of the graph G, and D is the degree matrix of A. The **degree matrix** is a diagonal matrix which contains information about the degree of each vertex – i.e. to how many other nodes is a node connected to.

From the Laplacian Matrix, we can compute the eigenvector w_2 and the respective eigenvalue λ_2.

2.2 Degree Centrality

The **degree centrality** is defined as the number of links incident upon a node. In other words, it represents the number of vertices a node has.

2.3 Reverse Cuthill McKee Ordering and Cholesky Factor

The **Reverse Cuthill McKee Ordering** is an algorithm used to permute a sparse matrix that has a symmetric sparsity pattern, into a band matrix form with a small bandwidth. In other words, the non-zero elements of the matrix get nearer to the diagonal of the matrix.

The **Cholesky Factorization** is the decomposition of a positive-definite matrix into the product of a lower-triangular matrix and its conjugate transpose.

3 Graph Partitioning

3.1 Spectral Partitioning

The most common example of a global approach to graph partitioning is **spectral partitioning**. This is carried out through a method known as spectral bisection. The spectral method utilizes the spectral graph theorem of linear algebra. This enables the decomposition of a real symmetric matrix into eigenvalues, within an orthonormal basis of eigenvectors.

A spectral bisection is computed using the second eigenvector corresponding with the second eigenvalue of the graph Laplacian Matrix. By thresholding the values of w_2 around 0, it will result in two roughly balanced partitions with minimum edgecut. On the other hand, by thresholding the values around the median of w_2, it will result in two strictly balanced partitions.

3.2 Inertial Bisection

Another common approach is **inertial bisection**. In this case the nodes of the graph are associated with a coordinate list. it thus relies on the geometric layout of the graph.

The main goal of this method is to find a hyperplane running through the centre of gravity of the points. Such a line is chosen in a way that the sum of squares of the distances of the nodes to the line is minimized.

3.3 Recursive Partitioning

Recursive bisection is a technique which is highly dependent on the decisions made during the earlier stages of the partitioning process. Furthermore, it lacks of global information.

On the other hand, **K-way partitioning** starts with the partitioning of a small set of vertices. After this, the obtained partition is projected back towards the original set of nodes in order to refine it.

The main difference between these two recursive approaches is that k-way partitioning stores global information about the graph – i.e. it takes into consideration the whole graph when creating the partitions, while recursive partitioning does not.

4 Graph Clustering

The main reason as to why we use clustering is the need to extract information from a set of data. This is done by grouping elements that are similar based

on some metric. The main goal of clustering is to put together elements which display similar behaviours, while separating them from the others.

4.1 Similarity Graphs

Given a set of data points and some notion of similarity between two pairs of nodes, the goal of clustering is to divide the data points into several groups such that points in the same group are similar and points in different groups are dissimilar to each other.

In order to build such similarity graphs, we can use one of the following methods:

- **ϵ-neighbourhood graph**

  Here we connect all of the points whose pairwise distances are smaller than a threshold value ϵ.

- **k-nearest neighbour graph**

  Here we connect vertex v_i to vertex v_j, if v_j is among the k-nearest neighbours of v_i.

- **Fully connected graph**

  Here we connect all points with positive similarity with each other, and we weight all edges by s_{ij} – i.e. the similarity function.

4.2 Spectral Clustering

The **spectral clustering** algorithm creates clusters that have the same number of nodes inside them, without considering the spacial characteristics of each node.

4.3 K-Means Clustering

The **k-means clustering** algorithm divides the clusters based on centroids. **Centroids** are imaginary or real locations representing the center of the cluster. The clusters are thus formed based on the distance of a node to a cluster centroid. In this case the number on nodes per cluster could be substantially different.

5 Image Deblurring

Here we used the Conjugate Gradient in order to deblur an image given the exact blurred image and the original transformation matrix.

Matrix B represents the blurred image, vector b is the vectorized form of matrix

B, matrix X represents the original grayscale image – where every entry corresponds to a pixel, the vector x is the vectorized form of matrix X, and matrix A indicates the transformation matrix coming from the repeated application of the **image kernel** – which is the bulr effect in this case. The sizes of the presented matrices and vectors are:

- $A \to n^2 \times n^2$
- $B \to n^2 \times n^2$
- $X \to n \times n$
- $\overrightarrow{b} \to n^2$
- $\overrightarrow{x} \to n^2$

With all of these matrices and vectors defined, we can write the following system of equations:

$$Ax = b$$

By solving this system, we can recover the original image, getting rid of the blur effect. The blurred effect in the image is given by a weighted average of the surrounding pixels to a pixel. These weights are defined by the kernel matrix K. From this matrix, we can obtain the matrix A such that the non-zero elements of each row of A correspond to the values of K.

A is a d^2 banded matrix – where $d \ll n$. This means that A is a sparse matrix.

5.1 Conjugate Gradient Method

5.1.1 Motivation

The complexity of Gaussian Elimination for very large linear systems of equations is too high. For this reason we need to find other ways to solve these linear systems of equations. There are two types of methods that we can use:

- **Direct Methods**

  Used to compute the exact solution in n steps. This method has a very high memory consumption due to additional fill-ins.

- **Iterative Methods**

  This methods use an arbitrary initial starting point in order to compute an approximate arbitrary solution. For this method there is no need for additional memory and it will converge after a few iterations. On the other side, these methods are very often less robust and not as general as direct methods.

5.1.2 Definitions

The **error** in step m is the deviation of the computed point from the exact solution. This can be written as:

$$e^{(m)} = x - x^{(m)}$$

Where $x^{(m)}$ is the computed point and x is the exact solution. This error is not known during the iterations – otherwise we would know the solution.

The **residual** provides us with a measure of the real error. This is computed as:

$$r^{(m)} = b - Ax^{(m)}$$

Where $x^{(m)}$ is the computed point.

5.1.3 Steepest Descent

The **steepest descent** algorithm is a precursor to the Conjugate Gradient algorithm. Here we do the following operations:

1. Start with a random initial guess
2. Take the gradient – which is the direction of the deepest descent
3. Compute the minimum of the gradient
4. Take the new gradient s.t. it is orthogonal to the previous gradient
5. Repeat steps 2 to 4 until a certain convergence criterion is met

This algorithm as a very low convergence rate, but it does not take optimal steps in order to find the next approximate solution.

5.1.4 Conjugate Gradient

The speed of convergence of the Conjugate Gradient is determined by the condition number $\kappa(A)$ of matrix A. The larger $\kappa(A)$, the slower the improvement.

If matrix A were not to be positive-definite, we can apply this trick in order to obtain a positive-definite matrix back:

$$A^T Ax = A^T b \rightarrow \tilde{A}x = \tilde{b}$$

The premultiplication of A with A^T will result in the positive-definite matrix $\tilde{A}$.

In order to carry out the actual method, we can use a slightly modified version of the steepest descent. The difference between the two methods is that in the case of the Conjugate Method, the two gradients need to be A-orthogonal. This means that:

$$d_i^T A d_j = 0$$

Where d_i and d_j are two vectors. By doing so, this would result in the new residual being orthogonal to the old residual. Furthermore, it would mean that every new step will never redo or undo optimizations of previous steps.

5.2 Condition Number

If a small modification in b results in a big change in x, then the system is said to be **ill-conditioned**, and the condition number $\kappa(A)$ will be big. This condition number can be computed as:

$$\kappa(A) = \frac{\sigma_{\max}}{\sigma_{\min}}$$

Where σ indicates the singular values of A. For a real symmetric matrix, the singular values are equal to the absolute value of the eigenvalues.

$$\sigma = |\lambda|$$

5.3 Preconditioner

A symmetric positive preconditioner P is selected such that P^{-1} approximates to $\tilde{A}^{-1}$.

$$\begin{aligned} P^{-1}\tilde{A}x &= P^{-1}\tilde{b} \\ (L^{-1}\tilde{A}L^{-1})(Lx) &= L^{-1}\tilde{b} \\ \bar{\tilde{A}}\bar{x} &= \bar{\tilde{b}} \end{aligned}$$

Where

$$\kappa(M^{-1}\tilde{A}) \ll \kappa(\tilde{A})$$

This is done in order to both decrease the condition number and to decrease the range of the eigenvalues. The computation of the preconditioner should be inexpensive to find. This is why we use the **Incomplete Cholesky factorization** in order to compute the preconditioner. By using this method, we only compute the Cholesky factors for the non-zero elements of $\tilde{A}$. This will give us the preconditioner:

$$P = F^T F$$

Where F is composed by the sparse IC factors. This routine can fail since the existence of F is not guaranteed. In order to amend to this issue, we can apply a diagonal shift to P. By doing so, we enforce positive-definiteness, thus making F computable.

6 Linear Programming

Linear programming is an optimization method which aims to maximize or minimize a linear objective function subject to linear equality or inequality constraints. Such an example is:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} c_i x_i \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{1,j} x_j \leq h_1 \\ & \vdots \\ & \sum_{j=1}^{n} a_{m,j} x_j \leq h_m \end{aligned}$$

In this case we have the presence of linear constrains. This means that the values of the vector x_i cannot assume any arbitrary value. The following needs to be respected:

- Satisfy a set of relationships
- Satisfy the non-negativity condition
  - Only positive values are accepted
  - Values need to belong to the feasible region

The previous linear problem can be re-written by using the following simplified notation:

$$\begin{array}{rl} \max & z = C^T x \\ \text{s.t.} & Ax \leq h \\ & x \geq 0 \end{array} \qquad \begin{array}{rl} \min & z = C^T x \\ \text{s.t.} & Ax \geq h \\ & x \geq 0 \end{array}$$

Where $z = f(x)$ represents the value of the objective function, c is a vector of coefficients, x is a vector of unknowns, A is the coefficient matrix, and h is the right hand side of the constraints. The sizes of these matrices are vectors are:

- $A \to m \times n$
- $\overrightarrow{c} \to n$
- $\overrightarrow{x} \to n$
- $\overrightarrow{h} \to m$

This is known as the **standard form** of a linear program. After writing the problem in a standard form, we need to graphically plot the feasible region which satisfies all of the constraints.

6.1 Fundamental Theorem of Linear Programming

In order to find the optimal value, we can make use of the **Fundamental Theorem of Linear Programming**.

> If a linear program admits a solution – i.e. if its neither unfeasible nor unbouded, this solution will lie in a vertex of the polytope defined by the feasible region. In the case in which two vertices are both maximizers or minimizers of the objective function, then also all the points lying on the line segment between them will represent optimal solutions of the linear programming problem.

6.2 Simplex Method

The **simplex method** is an algorithm used for solving linear programming problems. It has an exponential worst-case complexity. In order to solve a linear programming problem with the simplex method, we need to do the following:

- Write the problem in standard form
- Transform inequalities into equalities by adding:
  - **Slack Variables** – for maximization
  - **Surplus Variables** – for minimization
- Add the variables above also to the standard form – with their coefficient equal to 0

The steps above will result in, for example:

$$\begin{aligned} \max \quad & z = 3x + 2y + 0s_1 + 0s_2 \\ \text{s.t.} \quad & x + 2y + s_1 = 4 \\ & x - y + s_2 = 1 \\ & x, y \geq 0;\ s_1, s_2 \geq 0 \end{aligned}$$

The existence of a feasible solution implies also the existence of an optimal basic solution. The existence of an optimal solution implies the existence of an optimal basic solution.

6.3 Auxiliary Minimization Problem

By solving an initial **auxiliary minimization problem**, we find a feasible starting solution. It can be defined as follows – by introducing artificial variables $u_1, ..., u_m$:

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} u_i \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{1,j}x_j + s_1 + u_1 = h_1 \\ & \vdots \\ & \sum_{j=1}^{n} a_{m,j}x_j + s_m + u_m = h_m \\ & x_i, ..., x_n \geq 0;\ s_1, ..., s_m \geq 0;\ u_1, ..., u_m \geq 0 \end{aligned}$$

The auxiliary problem aims at minimizing the sum of the artificial variables. The optimal solution of the auxiliary problem would be achieved when all of the artificial variables have a value of 0.

6.4 Stopping Criterion

At every operation of the simplex method, we need to check if a **stopping criterion** is met. This criterion is:

$$r_D = c_D - c_B B^{-1} D$$

Where c_D represents the non-basic coefficients of the objective function, c_B represents the basic coefficient of the objective function, B^{-1} represents the inverse of the basic matrix, and D represents the non-basic matrix. The optimality condition is given by $r_D \geq 0$ for minimization problems, and $r_D \leq 0$ for maximization problems.

6.5 Iterative Rule

If the stopping criterion is not met, we select the highest reduced cost coefficient (in the case of maximization), or the lowest reduced cost coefficient (in the case of minimization). In case of multiple variables having the same values, we could end up swapping the same two variables over and over. In order to remove this

problem, we use the **iterative rule**.

The case described above can be fixed by computing the following ratio for every variable:

$$\frac{B^{-1}h}{B^{-1}D}$$

Where B^{-1} represents the inverse of the basic matrix, h represents the right hand side of the constraints, and D represents the non-basic matrix. Among all of the obtained ratios, we need to select the one that has the smallest positive value.