Chong Zhen Jie A0201613Y
MA5232: Convex Optimization and Its Applications

Assignment for Part 3

The codes for questions 3,4 and 5 are attached at the end of this document; the `ipynb` files are also included in a separate folder. All codes are implemented in `Python`.

For those questions that involve only coding, a brief description is given on the implementation details. For the results of those questions, interpretations and conclusions are given as well.

Question 1: Convexity

(i) Suppose $\{C_i\}_{i=1}^K$ are convex sets. For any $x, y \in \cap_{i=1}^K C_i$, this also means that for all $i = 1, 2, \ldots, K$, we have that $x, y \in C_i$.

By convexity of C_i, $\theta x + (1-\theta)y \in C_i$ for all $\theta \in [0,1]$. Since points of the form $\theta x + (1-\theta)y$ belong to every C_i, these points must then belong to the intersection of all C_i, that is $\theta x + (1-\theta)y \in \cap_{i=1}^K C_i$.

Hence, $\cap_{i=1}^K C_i$ is also convex.

(ii) Suppose $C_i, i \in \mathcal{I}$, is a collection of convex sets.

We define $K \coloneqq |\mathcal{I}|$, where we denote K as the cardinality of the index set $\mathcal{I}$. Without loss of generality, we relabel C_i in any way with values $j = 1, 2, \ldots, K$ to get C_j.

Then, we have that $\cap_{i\in\mathcal{I}} C_i = \cap_{j=1}^K C_j$ and in part (i), we have already shown that $\cap_{j=1}^K C_j$ is convex.

Hence, $\cap_{i\in\mathcal{I}} C_i$ is also convex.

(iii) Define any polyhedron $P \coloneqq \{x : Ax \leq b\}$. For any $x, y \in P$, we have that $Ax \leq b$ and $Ay \leq b$. Then, for all $\theta \in [0,1]$,

$$\begin{aligned} A\Big(\theta x + (1-\theta)y\Big) &= \theta Ax + (1-\theta)Ay \\ &\leq \theta b + (1-\theta)b \\ &= b \end{aligned}$$

Since $A\Big(\theta x + (1-\theta)y\Big) \leq b$, this implies that $\theta x + (1-\theta)y \in P$, and thus P is a convex set.

Hence, polyhedra are convex sets.

(iv) Suppose $\{f_i\}_{i\in\mathcal{I}}$ is a collection of convex functions.

We note that for any $i \in \mathcal{I}, x \in \mathbb{R}^n$,

$$f_i(x) \leq \sup\{f_i(x) : i \in \mathcal{I}\} = f(x)$$

Following that, together with the definition of a convex function, for any $i \in \mathcal{I}$, $\theta \in [0,1]$ and $x, y \in \mathbb{R}^n$, we have:

$$f_i\Big(\theta x + (1-\theta)y\Big) \leq \theta f_i(x) + (1-\theta) f_i(y) \leq \theta f(x) + (1-\theta) f(y) \tag{1.0}$$

Suppose at some $j \in \mathcal{I}$, we have $f\Big(\theta x + (1-\theta)y\Big) = f_j\Big(\theta x + (1-\theta)y\Big)$. Then, we have the following result:

$$\begin{aligned} f\Big(\theta x + (1-\theta)y\Big) &= f_j\Big(\theta x + (1-\theta)y\Big) \\ &\leq \theta f_j(x) + (1-\theta) f_j(y) \\ &\leq \theta f(x) + (1-\theta) f(y), \end{aligned}$$

where the inequalities are due to the fact that (1.0) holds for any $i \in \mathcal{I}$.

Hence, $f(x) = \sup\{f_i(x) : i \in \mathcal{I}\}$ is also a convex function.

(v) For the first part, we want to show that $\max\limits_{\|u\|_2=1} u^\top X u$ is well-defined using the Lagrange multiplier method.

By squaring the equality constraint, we get $u^\top u = \|u\|_2^2 = 1$. Then, we get the following problem:

$$\begin{aligned} \max \quad & u^\top X u \\ \text{subject to} \quad & u^\top u = 1 \end{aligned}$$

We now set up the Lagrangian problem.

$$\begin{aligned} \mathcal{L}(u, \lambda) &= u^\top X u - \lambda(u^\top u - 1) \\ \frac{\partial \mathcal{L}}{\partial u} &= 2Xu - 2\lambda u = 0 \\ Xu &= \lambda u \end{aligned}$$

We have shown that $Xu = \lambda u$ must hold, satisfying the form where u is an eigenvector of X, associated with the eigenvalue λ. From here, the objective function evaluates to:

$$\begin{aligned} u^\top X u = u^\top(\lambda u) = \lambda \cdot (u^\top u) = \lambda \cdot (1) = \lambda \\ u^\top X u = \lambda \end{aligned}$$

To maximize $u^\top Xu$, λ must be as large as possible. As such, since the maximum of $u^\top Xu$ is well-defined, it is also equal to the supremum. In other words,

$$\sup_{\|u\|_2=1} u^\top Xu = \max_{\|u\|_2=1} u^\top Xu = \lambda(X)$$ ■

For the second part, we want to show that $u^\top Xu$ is convex.

Define $f_u(X) \coloneqq u^\top Xu$ for any $u \in \mathcal{D} \coloneqq \{u : \|u\|_2 = 1\}$. We note that $f_u(X)$ is a linear function because for any symmetric matrices X, Y of the same size and $a, b \in \mathbb{R}$,

$$f_u(aX + bY) = u^\top(aX + bY)u = au^\top Xu + bu^\top Yu = af_u(X) + bf_u(Y)$$

$f_u(X)$ are thus convex functions because if we fix $a = \theta \in [0, 1]$ and $b = 1 - a$, we get

$$f_u\Big(\theta X + (1-\theta)Y\Big) = \theta f_u(X) + (1-\theta)f_u(Y)$$

Then, $\lambda(X)$ is just the supremum of a collection of convex functions over the domain $\mathcal{D}$ defined above. That is to say,

$$\lambda(X) = \sup_{\|u\|_2=1} u^\top Xu = \sup\{f_u(X) : u \in \mathcal{D}\}$$

With similar arguments shown in part (iv), $\lambda(X) = \sup\{f_u(X) : u \in \mathcal{D}\}$ is also a convex function. ■

Question 2: Geometric Programming

(i) A GP is of the form

$$\begin{aligned} \min_{x_1,\ldots,x_n>0} \quad & f_0(x_1,\ldots,x_n) \\ \text{s.t.} \quad & f_i(x_1,\ldots,x_n) \leq 1, \qquad 1 \leq i \leq m \\ & h_j(x_1,\ldots,x_n) = 1, \qquad 1 \leq j \leq p \end{aligned}$$

where $f_0, f_1, \ldots, f_m$ are posynomials and $h_1, \ldots, h_p$ are monomials.

We consider a change of variable $y_i = \log x_i \Rightarrow x_i = e^{y_i}$, followed by a logarithmic transformation on the objective function and constraints.

For any posynomial f,

$$\begin{aligned} f(x_1,\ldots,x_n) &= \sum_{j=1}^{N} c_j x_1^{a_{1j}} x_2^{a_{2j}} \ldots x_n^{a_{nj}} \\ &= \sum_{j=1}^{N} \exp(\log c_j) \exp(a_{1j} y_1) \exp(a_{2j} y_2) \ldots \exp(a_{nj} y_n) \\ &= \sum_{j=1}^{N} \exp(\log c_j + a_{1j} y_1 + a_{2j} y_2 + \ldots + a_{nj} y_n) \\ &= \sum_{j=1}^{N} \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y}) \\ &= f(e^{y_1}, \ldots, e^{y_n}) \end{aligned}$$

where we define vectors $\boldsymbol{a}_j \coloneqq (a_{1j}, a_{2j}, \ldots, a_{nj})^\top$ and $\boldsymbol{y} \coloneqq (y_1, y_2, \ldots, y_n)^\top$.

We then take the natural logarithm of f.

$$\log\Big(f(e^{y_1}, \ldots, e^{y_n})\Big) = \log\left(\sum_{j=1}^{N} \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y})\right)$$

We need to show that $\log\Big(f(e^{y_1}, \ldots, e^{y_n})\Big)$ is a convex function.

Proof: To show that $\log\Big(f(e^{y_1},\ldots,e^{y_n})\Big)$ is convex, we show that the Hessian matrix $\mathbf{H}$ of $\log f$ is positive semi-definite. $\mathbf{H}$ has the following form:

$$
\begin{aligned}
&(\mathbf{H})_{i,k} \\
&= \frac{\partial^2}{\partial y_i \partial y_k} \log\left(\sum_{j=1}^{N} \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y})\right) \\
&= \frac{\partial}{\partial y_k} \frac{\sum_{j=1}^{N} a_{ji} \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y})}{\sum_{j=1}^{N} \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y})} \\
&= \frac{1}{\Big(\sum_{j=1}^{N} \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y})\Big)^2} \left(\sum_{j=1}^{N} \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y}) \sum_{j=1}^{N} a_{ji} a_{jk} \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y})\right. \\
&\left. - \sum_{j=1}^{N} a_{ji} \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y}) \sum_{j=1}^{N} a_{jk} \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y})\right)
\end{aligned}
$$

Define $v_j \coloneqq \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y})$. Then, we have

$$
(\mathbf{H})_{i,k} = \frac{\sum_{j=1}^{N} a_{ji} a_{jk} v_j}{\sum_{j=1}^{N} v_j} - \frac{\sum_{j=1}^{N} a_{ji} v_j \sum_{j=1}^{N} a_{jk} v_j}{\Big(\sum_{j=1}^{N} v_j\Big)^2}
$$

Define matrix $\mathbf{A} \in \mathbb{R}^{N \times n}$, where $\boldsymbol{a}_j^\top \in \mathbb{R}^n$ defined above are row vectors in A, and vector $\boldsymbol{v} \coloneqq (v_1, v_2, \ldots, v_N)$. Using this fact, we observe some patterns in $(\mathbf{H})_{i,k}$ as follows:

$$
\text{From first term of } (\mathbf{H})_{i,k}: \quad \sum_{j=1}^{N} a_{ji} a_{jk} v_j = \sum_{j=1}^{N} a_{ji} v_j a_{jk} = \boldsymbol{A}_i^\top \Big(\operatorname{diag}(\boldsymbol{v})\Big) \boldsymbol{A}_k
$$

$$
\begin{aligned}
\text{From second term of } (\mathbf{H})_{i,k}: \quad \sum_{j=1}^{N} a_{ji} v_j \sum_{j=1}^{N} a_{jk} v_j &= \sum_{j=1}^{N} a_{ji} v_j \sum_{j=1}^{N} v_j a_{jk} \\
&= \boldsymbol{A}_i^\top \Big(\cancel{\operatorname{diag}(\boldsymbol{v})}\Big) \Big(\cancel{\operatorname{diag}(\boldsymbol{v})}\Big)^\top \boldsymbol{A}_k \\
&= \boldsymbol{A}_i^\top \Big(\cancel{\operatorname{diag}(\boldsymbol{v})}\Big)^2 \boldsymbol{A}_k
\end{aligned}
$$

where $\boldsymbol{A_i}, \boldsymbol{A_k} \in \mathbb{R}^N$ are column vectors of $\mathbf{A}$ as we can infer from the fact that the summation runs for $j = 1, \ldots, N$. As such, we deduce that the values of v_j must be the elements of a diagonal matrix, $\operatorname{diag}(\boldsymbol{v}) \in \mathbb{R}^{N \times N}$. We also define $\mathbf{1} \in \mathbb{R}^N$ as a vector of ones such that $\mathbf{1}^\top \boldsymbol{v} = \sum_{j=1}^{N} v_j$.

Substituting the values in terms of their vectors,

$$\begin{aligned}(\mathbf{H})_{i,k} =& \frac{\boldsymbol{A}_i^\top \Big(\operatorname{diag}(\boldsymbol{v}) \Big) \boldsymbol{A}_k}{\mathbf{1}^\top \boldsymbol{v}} - \frac{\boldsymbol{A}_i^\top \Big(\cancel{\operatorname{diag}(\boldsymbol{v})} \Big)^2 \boldsymbol{A}_k}{(\mathbf{1}^\top \boldsymbol{v})^2} \\ =& \frac{\boldsymbol{A}_i^\top \left(\mathbf{1}^\top \boldsymbol{v} \cdot \operatorname{diag}(\boldsymbol{v}) - \Big(\cancel{\operatorname{diag}(\boldsymbol{v})} \Big)^2 \right) \boldsymbol{A}_k}{(\mathbf{1}^\top \boldsymbol{v})^2}\end{aligned}$$

From $(\mathbf{H})_{i,k}$, we can see that $\mathbf{H}$ has the form:

$$\mathbf{H} = \frac{\mathbf{A}^\top \left(\mathbf{1}^\top \boldsymbol{v} \cdot \operatorname{diag}(\boldsymbol{v}) - \Big(\cancel{\operatorname{diag}(\boldsymbol{v})} \Big)^2 \right) \mathbf{A}}{(\mathbf{1}^\top \boldsymbol{v})^2}$$

We need to show that $\langle \boldsymbol{w}, \mathbf{H}\boldsymbol{w} \rangle \geq 0$ for any vector $\boldsymbol{w} \in \mathbb{R}^n$.

$$\begin{aligned}\langle \boldsymbol{w}, \mathbf{H}\boldsymbol{w} \rangle &= \boldsymbol{w}^\top \mathbf{H} \boldsymbol{w} \\ &= \frac{\boldsymbol{w}^\top \mathbf{A}^\top \left(\mathbf{1}^\top \boldsymbol{v} \cdot \operatorname{diag}(\boldsymbol{v}) - \Big(\cancel{\operatorname{diag}(\boldsymbol{v})} \Big)^2 \right) \mathbf{A}\boldsymbol{w}}{(\mathbf{1}^\top \boldsymbol{v})^2}\end{aligned}$$

We have $v_j \coloneqq \exp(\log c_j + \boldsymbol{a}_j^\top \boldsymbol{y}) > 0 \Rightarrow (\mathbf{1}^\top \boldsymbol{v})^2 > 0$. Therefore, we only need to show $\boldsymbol{w}^\top \mathbf{A}^\top \left(\mathbf{1}^\top \boldsymbol{v} \cdot \operatorname{diag}(\boldsymbol{v}) - \Big(\cancel{\operatorname{diag}(\boldsymbol{v})} \Big)^2 \right) \mathbf{A}\boldsymbol{w} \geq 0$.

Define $\mathbf{A}\boldsymbol{w} \coloneqq (z_1, z_2, \ldots, z_N)^\top$. Then,

$$\begin{aligned}&\boldsymbol{w}^\top \mathbf{A}^\top \left(\mathbf{1}^\top \boldsymbol{v} \cdot \operatorname{diag}(\boldsymbol{v}) - \Big(\cancel{\operatorname{diag}(\boldsymbol{v})} \Big)^2 \right) \mathbf{A}\boldsymbol{w} \\ &= \sum_{k=1}^{N} z_k^2 \left(\Big(\sum_{j=1}^{N} v_j \Big) v_k - v_k^2 \right) \\ &\geq 0\end{aligned}$$

because $v_k > 0 \Rightarrow \Big(\sum_{j=1}^{N} v_j \Big) v_k - v_k^2 = \Big(\sum_{j \neq k} v_j \Big) v_k > 0$.

This shows that $\langle \boldsymbol{w}, \mathbf{H}\boldsymbol{w} \rangle \geq 0$ so the Hessian matrix $\mathbf{H}$ of $\log f$ is positive semi-definite. Hence, $\log \Big(f(e^{y_1}, \ldots, e^{y_n}) \Big)$ is convex. ■

On the other hand, for any monomial h,

$$\begin{aligned} h(x_1, \ldots, x_n) &= c x_1^{a_1} x_2^{a_2} \ldots x_n^{a_n} \\ &= \exp(\log c) \exp(a_1 y_1) \exp(a_2 y_2) \ldots \exp(a_n y_n) \\ &= \exp(\log c + a_1 y_1 + a_2 y_2 + \ldots + a_n y_n) \\ &= \exp(\log c + \boldsymbol{a}^\top \boldsymbol{y}) \\ &= h(e^{y_1}, \ldots, e^{y_n}) \end{aligned}$$

where vectors $\boldsymbol{a}$ and $\boldsymbol{y}$ are similarly defined as above.

We then take the natural logarithm of h.

$$\begin{aligned} \log\Big(h(e^{y_1}, \ldots, e^{y_n})\Big) &= \log\Big(\exp(\log c + \boldsymbol{a}^\top \boldsymbol{y})\Big) \\ &= \log c + \boldsymbol{a}^\top \boldsymbol{y} \end{aligned}$$

We need to show that $\log\Big(h(e^{y_1}, \ldots, e^{y_n})\Big)$ is a convex function.

Proof: $\log\Big(h(e^{y_1}, \ldots, e^{y_n})\Big)$ is convex because it is an affine function in $\boldsymbol{y}$, and for any $\boldsymbol{y}, \boldsymbol{w} \in \mathbb{R}^n$ and $\theta \in [0, 1]$, it satisfies

$$\log c + \boldsymbol{a}^\top\Big(\theta \boldsymbol{y} + (1-\theta)\boldsymbol{w}\Big) = \theta(\log c + \boldsymbol{a}^\top \boldsymbol{y}) + (1-\theta)(\log c + \boldsymbol{a}^\top \boldsymbol{w}) \qquad \blacksquare$$

After showing that the transformed objective function and constraints are convex, we now have a convex program of the form:

$$\begin{aligned} \min_{y_1, \ldots, y_n} \quad & \log\Big(f_0(e^{y_1}, \ldots, e^{y_n})\Big) = \sum_{k=1}^{N} \exp(\log c_{0k} + \boldsymbol{a}_{0k}^\top \boldsymbol{y}) \\ \text{s.t.} \quad & \log\Big(f_i(e^{y_1}, \ldots, e^{y_n})\Big) = \sum_{k=1}^{N} \exp(\log c_{ik} + \boldsymbol{a}_{ik}^\top \boldsymbol{y}) \leq 0, \qquad 1 \leq i \leq m \\ & \log\Big(h_j(e^{y_1}, \ldots, e^{y_n})\Big) = \log c_j + \boldsymbol{a}_j^\top \boldsymbol{y} = 0, \qquad 1 \leq j \leq p \end{aligned}$$

where we also took the natural logarithm on the RHS of the original constraints to get $\log(1) = 0$.

(ii) We define the terms below.

$$h \coloneqq \text{height}, \qquad A \coloneqq \text{upper limit of corresponding area},$$
$$l \coloneqq \text{length}, \qquad U \coloneqq \text{upper bound of corresponding ratio},$$
$$w \coloneqq \text{width}, \qquad L \coloneqq \text{lower bound of corresponding ratio}.$$

The three-dimensional box container is of the form:

$$\begin{aligned} \max_{h,l,w>0} \quad & hlw \\ \text{s.t.} \quad & 2hl + 2hw + 2lw \leq A_{total} \\ & lw \leq A_{ceiling} \\ & L_{hw} \leq \frac{h}{w} \leq U_{hw} \\ & L_{hl} \leq \frac{h}{l} \leq U_{hl} \end{aligned}$$

We can transform the above problem into a GP using simple transformations. Maximizing hlw is equivalent to minimizing its inverse, $h^{-1}l^{-1}w^{-1}$. For any posynomial f and monomial g, we express the above inequality constraints in terms of $f(h,l,w) \leq 1$ and $g(h,l,w) \leq 1$.

Hence, we can now express the problem in terms of a GP:

$$\begin{aligned} \min_{h,l,w>0} \quad & h^{-1}l^{-1}w^{-1} \\ \text{s.t.} \quad & \frac{2}{A_{total}}\, hl + \frac{2}{A_{total}}\, hw + \frac{2}{A_{total}}\, lw \leq 1, \\ & \frac{1}{A_{ceiling}}\, lw \leq 1, \\ & L_{hw}\, h^{-1}w \leq 1, \quad \frac{1}{U_{hw}}\, hw^{-1} \leq 1, \\ & L_{hl}\, h^{-1}l \leq 1, \quad \frac{1}{U_{hl}}\, hl^{-1} \leq 1. \end{aligned}$$

Question 3: Compressed Sensing

We first solve part (ii) as parts (i),(iii), (iv) involve coding.

(ii) The original problem is the following:

$$\begin{aligned} \min \quad & \|x\|_1 \\ \text{s.t.} \quad & Ax = y \end{aligned} \tag{3.0}$$

Since $\|x\|_1 = \sum_{i=1}^n |x_i|$ is not linear, we can let $t_i := |x_i| \geq 0$ and add the constraint $|x_i| \leq t_i$, or equivalently $-t_i \leq x_i \leq t_i$. We get a Linear Program of the form:

$$\begin{aligned} \min \quad & \mathbf{1}^\top t = \sum_{i=1}^n t_i \\ \text{s.t.} \quad & Ax = y \\ & t_i \geq x_i \qquad \text{for } i = 1, \ldots, n \\ & t_i \geq -x_i \quad \text{for } i = 1, \ldots, n \\ & t \geq \mathbf{0} \end{aligned}$$

where $\mathbf{1}, \mathbf{0} \in \mathbb{R}^n$ are vectors of ones and zeros respectively. The above Linear Program is still not in standard form as there are inequality constraints.

We introduce another non-negative slack variable s to get $|x_i| - t_i + s_i = 0$, or equivalently $x_i - t_i + s_i = 0$ and $x_i + t_i - s_i = 0$. Because x_i is a free variable, we denote $x_i = x_i^+ - x_i^-$, where $x_i^+, x_i^- \geq 0$. We can then concatenate the matrices and vectors to get the following form:

$$\min \quad c^\top \tilde{x} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \end{bmatrix}^\top \begin{bmatrix} x^+ \\ x^- \\ t \\ s \end{bmatrix} = \mathbf{1}^\top t$$

$$\text{s.t.} \quad \tilde{A}\tilde{x} = \begin{bmatrix} A & A & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} x^+ \\ -x^- \\ t \\ s \end{bmatrix} = A(x^+ - x^-) = Ax = y$$

$$\tilde{I}\tilde{x} = \begin{bmatrix} I & -I & -I & I \\ I & -I & I & -I \end{bmatrix} \begin{bmatrix} x^+ \\ x^- \\ t \\ s \end{bmatrix} = \begin{bmatrix} x^+ - x^- - t + s \\ x^+ - x^- + t - s \end{bmatrix} = \begin{bmatrix} x - t + s \\ x + t - s \end{bmatrix} = \mathbf{0}$$

$$\tilde{x} = \begin{bmatrix} x^+ \\ x^- \\ t \\ s \end{bmatrix} \geq \mathbf{0}$$

where I is the identity matrix, and the terms $c, \tilde{x}, \tilde{A}, \tilde{I}$ are defined above. The terms $\mathbf{1}, \mathbf{0}$ are matrices or vectors of ones and zeros respectively of their corresponding dimensions.

We can further combine the constraints by defining the following terms:

$$\hat{y} = \begin{bmatrix} y \\ \mathbf{0} \end{bmatrix}, \ \hat{A} = \begin{bmatrix} \tilde{A} \\ \tilde{I} \end{bmatrix}$$

Hence, the problem reduces to a Linear Program in standard form:

$$\begin{aligned} \min \quad & c^\top \tilde{x} \\ \text{s.t.} \quad & \hat{A}\tilde{x} = \hat{y} \\ & \tilde{x} \geq \mathbf{0} \end{aligned}$$

(i) The remaining parts of this question is mainly implemented using `numpy` and `cvxpy`.

Generating random matrix $A \in \mathbb{R}^{m \times n}$ and random sparse vector $x^* \in \mathbb{R}^n$

Matrix A can be generated using `numpy.random.standard_normal`. On the other hand, for the sparse vector x^*, we index an initial array of zeroes on a uniform random Boolean mask to select the locations of the non-zero entries.

The Boolean mask is created by comparing an array of uniform random generated values in the unit interval to its j-th percentile value, using functions `numpy.random.rand` and `numpy.percentile`, where

$$j = \frac{s}{n} \times 100\%$$

This ensures that there are exactly s entries, where we define s to be the number of non-zero entries. With the Boolean mask, we can easily assign the non-zero entries with 1.

(iii) **Defining error tolerance**

Declaring success if $\|\hat{x} - x^*\| / \|x^*\| \leq 10^{-4}$ means that our relative error is $\|\hat{x} - x^*\| / \|x^*\|$ under the L2-norm while its tolerance is 10^{-4}. If the relative error is within the tolerance level, we are assumed to have successfully reconstructed the sparse vector.

We define a function here to return True if it is a success and return False otherwise. This function will help perform the recording of successes later on in part (iv).

(iv) **Solving the problem**

We can then construct the problem using `cvxpy` by following the original problem in (3.0) over every pair of m and s, where $m, s \in \{1, \ldots, 50\}$. To record the successes, we initialize a matrix of zeros of size $m \times s$. This means that each row denotes the choice of m while each column denotes the choice of s. Each entry denotes the number of success at that pair of m and s. At every single iteration, we update this matrix using the function mentioned previously in part (iii).

Results

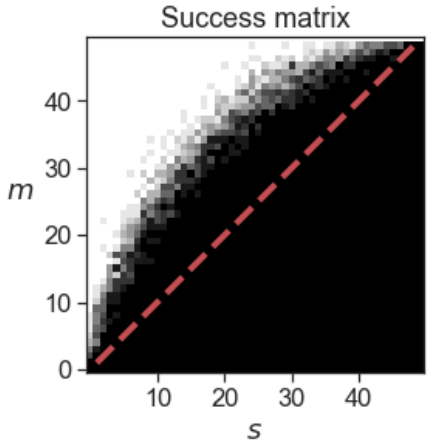


FIGURE 1. Grayscale image of success matrix

In Figure (1), the red dotted 45° line is included as a reference.

Since $y \in \mathbb{R}^m$, the value m denotes the number of measurements. We observe a pattern in the form of a curve. For all points where m exceeds the curve, we see a region of high intensity, as indicated by its white colour. This corresponds to the high probability of success in our experiment. On the other hand, for all points below the curve, we see a region of low intensity, as indicated by its black colour. This represents the low probability of success in our experiment.

Across all pairs of m and s, we have an average of only 2.726 successes out of the 10 trials. This can be seen in Figure (1), where the curve is much higher than the 45° line; the white region is so much smaller than the black region.

Interpretations and conclusions

From the results, we can conclude the following:

(a) In general, given a vector of fixed sparsity s that we want to reconstruct, a greater number of measurements, m, leads to a higher probability of reconstructing the vector. This can be seen from the white region lying above the curve. This is because at low levels of m such that $m < n$, we are dealing with an underdetermined system[1] of linear equations in $y = Ax$, with infinitely many solutions. By

increasing m, we are restricting the degree of freedom imposed by the unknowns, allowing us to potentially reconstruct the vector.

(b) On the other hand, given a fixed number of measurements m, if we decrease s, we are imposing a constraint to only allow smaller numbers of non-zero entries in the vector we want to reconstruct. Similar to part (a), we are restricting the degree of freedom imposed by the unknowns here, giving us a higher probability of reconstructing the vector. This is also indicated by the white region lying to the left of the curve.

(c) Finally, combining parts (a) and (b), this forms the curve above the 45° line, where $m \gg s$ for high probability of success. Here we can also see the curve as a threshold. At every level of s, we only need a minimum number of measurements to be able to reconstruct the vector. As s increases, the threshold also increases. This continues until $m = s$, where we have the usual determined system of linear equations $y = Ax$, where the unique solution x exists if and only if A is invertible. This can be seen in Figure (1), where the point at the top right corner is white. We can also check that at $m = s = 50$, the number of successes recorded is 10 out of 10 trials. In other words, the probability of generating a singular matrix A is very small.

Question 4: Matrix Completion

(i) The original problem is of the form:

$$\begin{aligned} \min_{X} \quad & \|X\|_* \\ \text{s.t.} \quad & X_{ij} = Z_{ij} \quad \text{for all} \quad (i,j) \in \Omega \end{aligned} \tag{4.0}$$

Define $\tilde{X} \coloneqq \begin{pmatrix} W_1 & X \\ X^\top & W_2 \end{pmatrix}$ and $t \coloneqq \|X\|_* = \frac{1}{2}\min\left\{\operatorname{tr}(W_1) + \operatorname{tr}(W_2) : \tilde{X} \text{ is PSD}\right\}$. If we simply substitute t in, the problem becomes an unbounded one as follows:

$$\begin{aligned} \min \quad & t \\ \text{s.t.} \quad & X_{ij} = Z_{ij} \quad \text{for all} \quad (i,j) \in \Omega \\ & 2t \leq \operatorname{tr}(W_1) + \operatorname{tr}(W_2) \\ & \tilde{X} \text{ is PSD} \end{aligned}$$

where we have the constraint $2t \leq \operatorname{tr}(W_1) + \operatorname{tr}(W_2)$ because t is defined to be the minimum of the mentioned set. However, since this is a minimization problem, and $2t$ is the lower bound in the constraint, the objective function may be improved indefinitely to $-\infty$ without violating this constraint.

To ensure that the constraint is tight, we impose an equality constraint instead, that is $2t = \operatorname{tr}(W_1) + \operatorname{tr}(W_2)$. Then, we have

$$\begin{aligned} \min \quad & t = \frac{1}{2}\operatorname{tr}(W_1) + \frac{1}{2}\operatorname{tr}(W_2) \\ \text{s.t.} \quad & X_{ij} = Z_{ij} \quad \text{for all} \quad (i,j) \in \Omega \\ & \tilde{X} \text{ is PSD} \end{aligned}$$

Now, to formulate the problem into the given SDP form, we note that

$$\operatorname{tr}(\tilde{X}) = \operatorname{tr}\begin{pmatrix} W_1 & X \\ X^\top & W_2 \end{pmatrix} = \operatorname{tr}(W_1) + \operatorname{tr}(W_2)$$

This is true because we know $X \in \mathbb{R}^{m\times n}$, so $\tilde{X} \in \mathbb{R}^{(m+n)\times(m+n)}$. From there, we can infer that $W_1 \in \mathbb{R}^{m\times m}$ and $W_2 \in \mathbb{R}^{n\times n}$. Since $\tilde{X}, W_1$, and W_2 are square matrices, the trace of $\tilde{X}$ is equivalent to the sum of the diagonal elements of W_1 and W_2. The objective function then evaluates to $\frac{1}{2}\operatorname{tr}(\tilde{X})$.

Hence, we get the following SDP:

$$\begin{aligned} \min \quad & \frac{1}{2}\operatorname{tr}(\tilde{X}) \\ \text{s.t.} \quad & X_{ij} = Z_{ij} \quad \text{for all} \quad (i,j) \in \Omega \\ & \tilde{X} \text{ is PSD} \end{aligned} \tag{4.1}$$

Check: To show that solving (4.1) indeed solves the original problem in (4.0), or in other words, minimizing $\frac{1}{2}\operatorname{tr}(\tilde{X})$ is equivalent to minimizing $\|X\|_*$.

Since this is a matrix completion problem that aims to minimize the rank of X, using the SVD of X, suppose the rank-r approximation of X is $X = U\Sigma V^\top$, where $\Sigma \in \mathbb{R}^{r\times r}$ and $\operatorname{rank}(X) = r$.

Then, we have

$$\tilde{X} = \begin{pmatrix} W_1 & U\Sigma V^\top \\ V\Sigma U^\top & W_2 \end{pmatrix} \tag{4.2}$$

To find feasible W_1 and W_2 such that $\|X\|_* = \frac{1}{2}\operatorname{tr}(\tilde{X})$. Since $\|X\|_* = \operatorname{tr}(\Sigma)$, it is easy to see from (4.2) that $W_1 \coloneqq U\Sigma U^\top$ and $W_2 \coloneqq V\Sigma V^\top$ and we get

$$\tilde{X} = \begin{pmatrix} U\Sigma U^\top & U\Sigma V^\top \\ V\Sigma U^\top & V\Sigma V^\top \end{pmatrix} = \begin{pmatrix} U \\ V \end{pmatrix} \Sigma \begin{pmatrix} U \\ V \end{pmatrix}^\top$$

$\tilde{X}$ is clearly PSD as we have shown the proof for a similar matrix in Question 2.

Proof: For any vector $y \in \mathbb{R}^{m+n}$, define $\begin{pmatrix} U \\ V \end{pmatrix}^\top y \coloneqq (z_1, \ldots, z_r)^\top$. Then, we have

$$\begin{aligned} y^\top \tilde{X} y &= y^\top \begin{pmatrix} U \\ V \end{pmatrix} \Sigma \begin{pmatrix} U \\ V \end{pmatrix}^\top y \\ &= \sum_{k=1}^{r} \sigma_k z_k^2 \\ &\geq 0 \end{aligned}$$

where σ_k is defined to be the singular value of $\tilde{X}$ so σ_k is always non-negative. ■

As such, there exists feasible $W_1 \coloneqq U\Sigma U^\top$ and $W_2 \coloneqq V\Sigma V^\top$ and the objective function evaluates to

$$\frac{1}{2}\operatorname{tr}(\tilde{X}) = \frac{1}{2}\operatorname{tr}(W_1) + \frac{1}{2}\operatorname{tr}(W_2) = \operatorname{tr}(\Sigma) = \|X\|_*$$

where $\operatorname{tr}(W_1) = \operatorname{tr}(U\Sigma U^\top) = \operatorname{tr}(U^\top U\Sigma) = \operatorname{tr}(\Sigma)$ because $U^\top U = I$; the columns of $U \in \mathbb{R}^{m\times r}$ are orthonormal. The same argument can be made for W_2, where the columns of $V \in \mathbb{R}^{n\times r}$ are orthonormal. Furthermore, minimizing $\frac{1}{2}\operatorname{tr}(\tilde{X})$ is equivalent to minimizing $\operatorname{tr}(\tilde{X})$ so we can just drop the constant $\frac{1}{2}$.

Hence, the original problem can be expressed as a SDP of the form:

$$\begin{aligned} \min \quad & \operatorname{tr}(\tilde{X}) \\ \text{s.t.} \quad & X_{ij} = Z_{ij} \quad \text{for all} \quad (i,j) \in \Omega \\ & \tilde{X} \text{ is PSD} \end{aligned}$$

noting that the objective function, $\operatorname{tr}(\tilde{X}) = \operatorname{tr}(I\tilde{X})$, where I is the identity matrix. This agrees with the general form of $\operatorname{tr}(CX)$ as stated in the question.

(ii) The remaining parts of this question is mainly implemented using `numpy` and `cvxpy`.

Generating random matrix $Z \in \mathbb{R}^{m \times n}$

To create the random matrix $Z = UV^\top$, we simply use functions like `numpy.random.standard_normal` to generate random matrices U and V with values from the standard normal distribution. We then perform matrix multiplication to get Z.

(iii) **Selecting random entries Ω, where $|\Omega| = k$**

To select the k entries from $Z \in \mathbb{R}^{m \times n}$, we create a uniform random Boolean mask to indicate the indices of the locations we want to pick.

We first generate a matrix with uniform random values in the unit interval using `numpy.random.rand`. Then, to create the Boolean mask, we compare this matrix to its j-th percentile value using `numpy.percentile`, where

$$j = \frac{k}{mn} \times 100\%$$

Comparing using the j-th percentile value ensures that exactly k entries are chosen. Now that we have created our Boolean mask, we can just index the matrices X and Z in part (iv) later such that the values at these picked locations are equal, that is

$$X_{ij} = Z_{ij} \text{ for all } (i,j) \in \Omega$$

(iv) **Solving the problem**

We can then construct the problem using `cvxpy` by following the original problem in (4.0) over the different choices of $k \in \{100, 200, \ldots, 3000\}$. As explained in part (iii), we can now easily set the constraint using the Boolean mask created.

Results using evaluation metric: mean-squared error (MSE)

To compute the MSE, we can easily obtain the locations of the unobserved entries by inverting the Boolean mask created in part (iii). We get the following results:

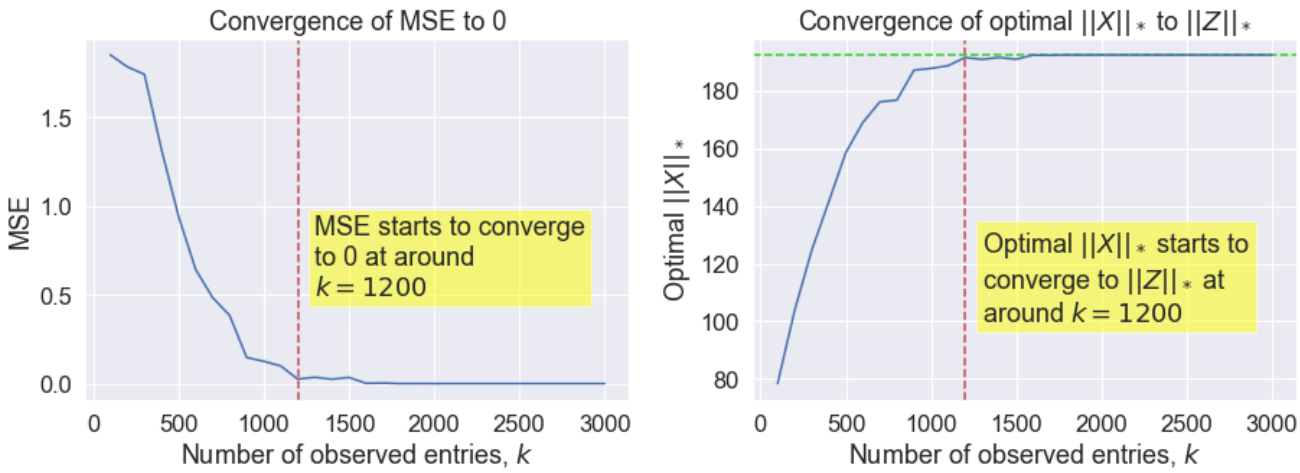


FIGURE 2. Graphs of MSE and optimal $\|X\|_*$ over different choices of k

In Figure (2), for the MSE graph on the left, we observe a pattern that as k increases, the MSE decreases. Beyond $k = 1200$, the MSE starts to converge to 0.

On the other hand, the graph on the right in Figure (2) shows the changes in the optimal $\|X\|_*$ over the different choices of k. As a reference, the green dotted horizontal line indicates $\|Z\|_* \approx 192.444$. We see that as k increases, optimal $\|X\|_*$ increases, starting to converge to $\|Z\|_*$ at around $k = 1200$ as well.

Interpretations and conclusions

From the results, we can infer the following conclusions:

(a) In general, the more observed data we have, the better we are able to approximate the unobserved data with the optimal solution X. This is intuitive because we can consider the extreme case; it would be difficult for the optimization model to learn with only 1 data point, as compared to when we have a nearly complete dataset.

(b) However, we only need a minimum number of observed data to generate a fairly well approximation of Z. In this question, we have $mn = 10000$ entries. But, we only needed roughly 1200 observed entries to get a relatively good approximation with MSE ≈ 0. This is also supported by the convergence of optimal $\|X\|_*$ to $\|Z\|_*$ at around $k = 1200$. We only need around 12% of the total entries from Z.

A side thing to note is that, this matrix completion problem is similar to the compressed sensing problem above. The nuclear norm of X is equivalent to the L1-norm of $\sigma(X)$, where $\sigma(X)$ is the vector of singular values of X, because

$$\|X\|_* = \sum_i \sigma_i = \sum_i |\sigma_i| = \|\sigma(X)\|_1$$

where the singular values σ_i are defined to be non-negative. In both questions, we have rather similar conclusions from the results, that is we only need a minimum number of known data to be able to recover the unknown data.

The difference here is that this problem involves sparse matrices while the compressed sensing problem involves sparse vectors. This can also be seen from the fact that the Linear Program is a special case[2] of the SDP, where we can just arrange the scalar quantities in the Linear Program into diagonal matrices to transform the variables to the form in the SDP.

Question 5: Movie Lens

This question is mainly implemented using `pandas` and `numpy`.

(i) **Loading of data**

We first load the dataset using `pandas` for ease of data exploration before working with `numpy` arrays for the matrix completion algorithm in part (iv).

(ii) **Random train-test split**

To perform a random train-test split, we first create a uniform random Boolean mask to indicate the indices of the observations we want in the training dataset.

The Boolean mask can be created by comparing an array of uniform random generated values in the unit interval to its 90th percentile value, using functions `numpy.random.rand` and `numpy.percentile`.

This ensures that we get a train-test ratio of exactly 90:10. Then, we can just index the entire dataset based on the Boolean mask to perform the split.

(iii) Before we move on to build the baseline estimator, we perform some simple exploratory data analysis to determine the main characteristics of the dataset. This is because the baseline estimator is a naive average model, which requires consistency between training and testing datasets. Thus, there may be potential complications.

Exploratory data analysis

We know for a fact that every user rates a subset of movies. Hence, we focus on movie ID and movie ratings.

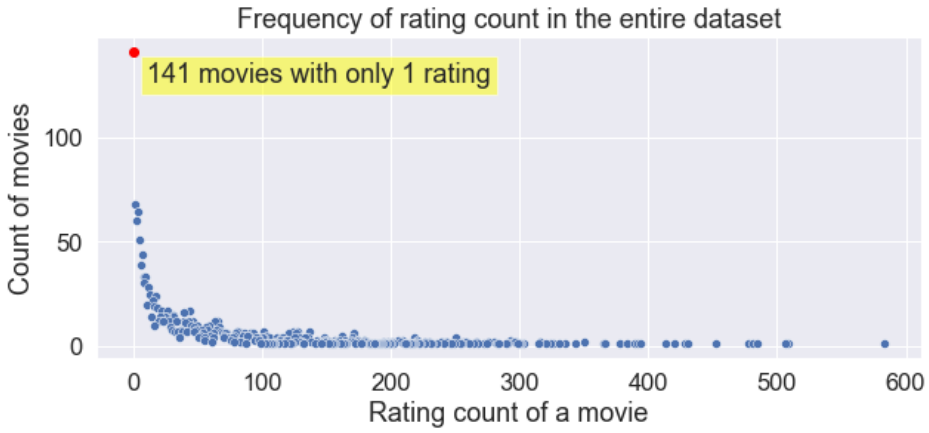


FIGURE 3. Distribution of rating count in entire dataset

In Figure (3), we define rating count to be the number of users who rated the movie, regardless of the rating value. We observe that there are some movies with very little ratings. In fact, we have 141 of such movies with only 1 rating. This could affect our baseline estimator, which requires that movie IDs in the testing dataset must also be in the training dataset to be able to compute the estimated rating.

As such, there may be missing values in the baseline estimates. Since this is just a baseline estimator, to keep it simple, we choose to impute missing values, if any.

To find an appropriate value, we look further at the distribution of movie ratings, but only in the training dataset. We do not want to examine the entire dataset as the value chosen would have taken into account the characteristics of the testing dataset, creating potential bias.

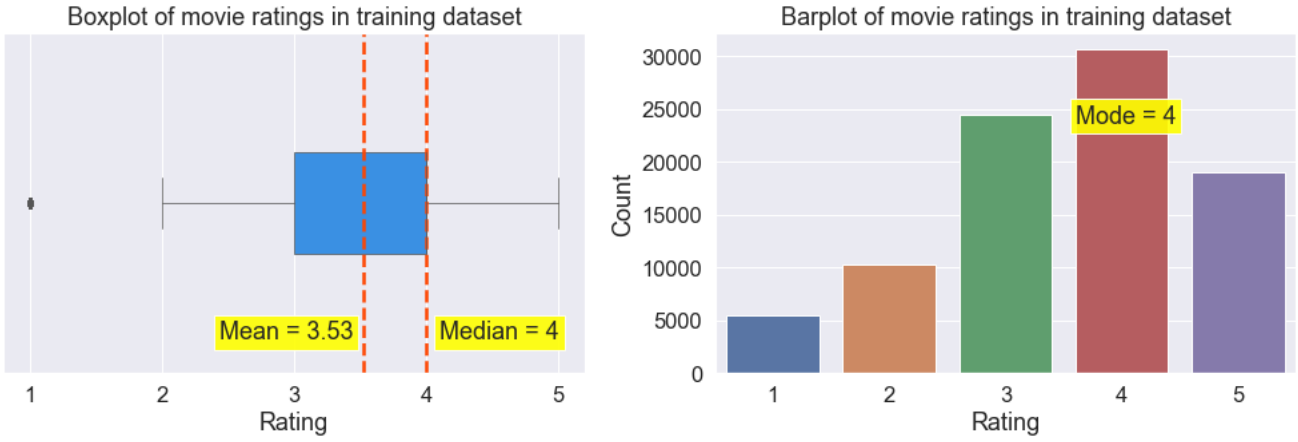


FIGURE 4. Distribution of movie ratings in training dataset

From Figure (4), it is evident that the distribution of movie ratings in the training dataset is skewed, with the majority of users voting 3 and above. Nevertheless, the rounded-off mean, the mode, and the median values, all equal to 4. Hence, the value of 4 may be an appropriate estimate for imputing missing values.

Building baseline estimator

We can now build our baseline estimator, imputing missing values with 4, if any. Since we are still working with `pandas` DataFrame, the estimated ratings can be simply computed by grouping the training dataset by movie IDs, and then computing the mean for each movie ID. We check that we have indeed imputed 13 missing values.

Integer constraint of movie ratings

We should note that for this question, the movie ratings only take integer values. As such, it would be inappropriate to evaluate our estimated ratings based on non-integer values because the non-integer estimates are not feasible.

Assumption: Here we assume to round off all estimates to the nearest integer. An interpretation could be that, for example, a rating of 3.6 would suggest that the user is more inclined to give a rating of 4 than a rating of 3.

Evaluation metric: root-mean-squared-error (RMSE)

To compute the squared error loss, we use the RMSE metric for better interpretability because it is measured in the same units as our target variable, the movie ratings. The general form of RMSE is given by

$$\sqrt{\frac{1}{|\Omega^c|}\sum_{ij\in\Omega^c}(X_{ij}-Z_{ij})^2}$$

where Ω^c denotes the testing dataset, X_{ij} denotes the estimated solution, and Z_{ij} refers to the given dataset.

Results of baseline estimator

For the baseline estimator, we computed RMSE ≈ 1.0599. This means that on average, the estimated ratings differ from the actual ratings by 1.0599.

(iv) **Matrix completion algorithm**

We define $X \in \mathbb{R}^{943\times 1682}$, where each row denotes a unique user, while each column denotes a unique movie. The algorithm is mainly implemented using `numpy.linalg` functions to compute the SVD and Frobenius norm, and other `numpy` operations and functions to handle the matrices.

(v) To clip the values of the estimated ratings to be within the scale of 1 to 5, we can just use the function `numpy.clip` and specify the limits, 1 and 5.

Testing the algorithm at $r = 10$

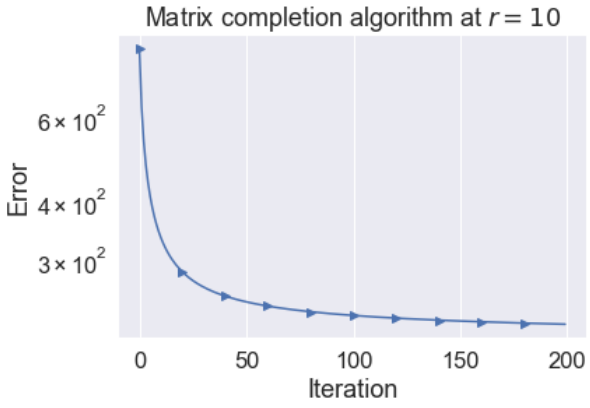


FIGURE 5. Graph of error $\|X - X_{new}\|_F$ at $r = 10$

We first test the algorithm at $r = 10$. In Figure (5), we use a log scale on the y-axis for better visualization. We observe that at around 50 iterations, the errors start to converge, suggesting that the algorithm is indeed converging to the solution X.

At $r = 10$, we get RMSE ≈ 1.0465, which is slightly smaller than the RMSE of the baseline estimator of 1.0599. This lower value of RMSE indicates that the algorithm is performing better than the baseline estimator.

We go on to run the algorithm for all $r = 1, \ldots, 20$ to see if we get even better results.

(vi) **Interpretations of results for algorithm for all** $r = 1, ..., 20$

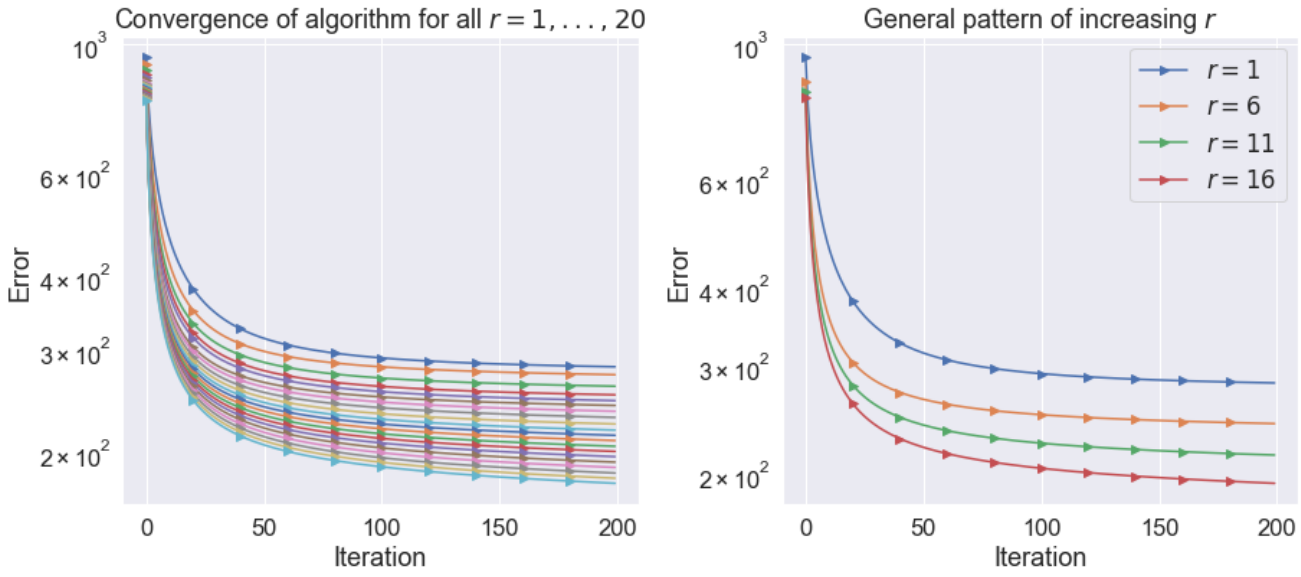


FIGURE 6. Graphs of error $\|X - X_{new}\|_F$

The graph to the left in Figure (6) shows the results for all $r = 1, \ldots, 20$. We see that all the errors tend to converge at around 50 iterations as well. This indicates that the algorithm is consistent in reaching the solution X.

What we are effectively computing is to find X_{new} such that

$$\|X - X_{new}\|_F \leq \|X - A\|_F \text{ for any matrix } A \text{ of rank-}r,$$

implying that over all rank-r matrices, X_{new} minimizes the Frobenius norm. Hence, we can regard $\|X - X_{new}\|_F$ as our training error.

As it is hard to visualize all the different choices of r, we plot again just a subset of it in the graph to the right in Figure (6). We observe that as r decreases, the training errors converge at larger values.

This makes sense because at low levels of r, we are keeping less non-zero singular values, which essentially translates to capturing less information, thus our lower rank approximations of X have larger training errors.

We now examine the RMSE of the estimated solutions for all r.

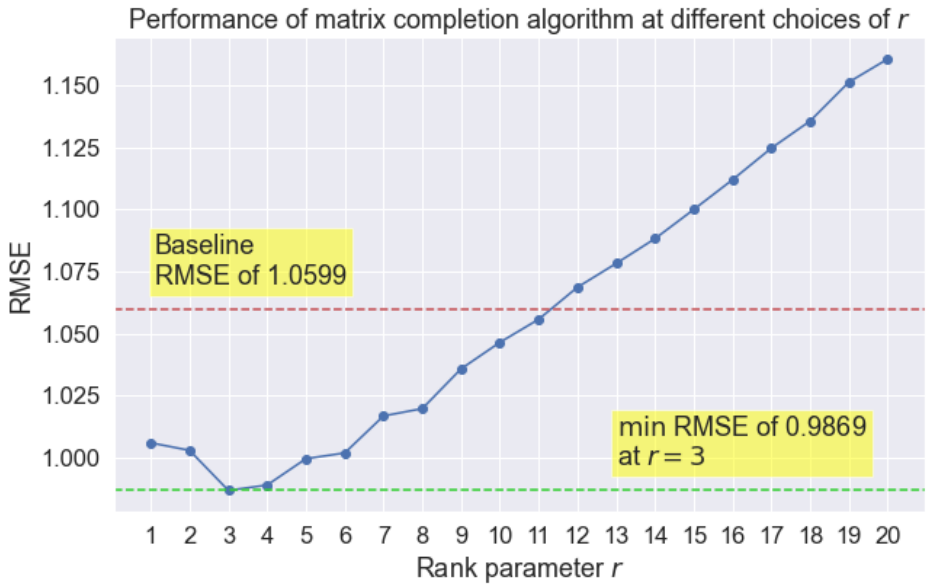


FIGURE 7. Graph of RMSE over different choices of r

In Figure (7), as r decreases, the RMSE decreases and reaches a minimum of roughly 0.9869 at $r = 3$. On average, the estimated ratings from the algorithm at $r = 3$ differ from the actual ratings by 0.9869, which is significantly better than the baseline estimator of 1.0599. However, we also note that at $k \geq 12$, the algorithm performs worse than the baseline estimator.

The results suggest that the recovered matrix $X \in \mathbb{R}^{943\times1682}$ is approximately low-rank at $r = 3$ when the entire dataset has high dimensionality of 943 unique users and 1682 unique movies. A low-rank matrix having many linearly dependent vectors implies that there may be many correlations in our entire dataset. This causes the algorithm to perform poorly at larger $k \geq 12$.

Conclusions

Combining the observations from both Figures (6) and (7), we can make the following conclusions:

(a) If we choose the value of r to be too large, this is similar to the case of redundant data, due to large amounts of correlations found in the dataset. This may cause our estimated solutions to be imprecise due to additional noise.

(b) This may be equivalent to overfitting. This is supported by the fact that at high levels of r, we see that the algorithm converges with a smaller training error in Figure (6), but the RMSE is larger as shown in Figure (7).

(c) By minimizing r, we are overcoming the curse of dimensionality by reducing the number of non-zero singular values to keep. This in turn denoises the data, allowing the algorithm to improve its performance by decreasing r.

(d) However, we also need to seek a balance in the model complexity. In Figure (7), at $r < 3$, the RMSE starts to increase again because we kept too little non-zero singular values, resulting in information loss.

In short, this question is an example of how big data in real life have approximately low-ranks, where the dataset in the form of a matrix have dimensions that grow so much more quickly than its rank. This is most likely due to the fact that data collected often have high correlations. For example, a movie with subsequent sequels tend to be correlated. Hence, by minimizing the rank in our optimization problem, we are trying to find an underlying low-dimensional structure of the high-dimensional dataset.

P.S. : If we drop the integer constraint for movie ratings, we get similar conclusions on the relative performance of the algorithm to the baseline estimator as well.

References

[1] Wikipedia contributors. Underdetermined system — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Underdetermined_system&oldid=1010727449, 2021. [Online; accessed 27-March-2022].

[2] Wikipedia contributors. Semidefinite programming — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Semidefinite_programming&oldid=1078182931, 2022. [Online; accessed 27-March-2022].

Chong Zhen Jie A0201613Y

Assignment Question 3: Compressed Sensing

In [1]:
```
import cvxpy as cp
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
```

In [2]:
```
sns.set_theme(style="ticks", font_scale=1.5)
SEED = 5232
```

3(i) Generating random matrix $A \in \mathbb{R}^{m \times n}$ and random sparse vector $x^* \in \mathbb{R}^n$

In [3]:
```
n = 50

# Range of values for m and s
idx = range(1,51)

# Define function to create sparse vector
def create_sx(s, n=n):
    """
    Args:
    ===
    s: Sparsity, or number of non-zero entries
    n: Size of sparse vector

    Output:
    ===
    Return a sparse vector of size n, with sparsity s.
    """
    # Percent of non-zero entries
    percent = (s / n) * 100

    # Create Boolean mask
    rand = np.random.rand(n)
    mask = rand <= np.percentile(rand, percent)

    # Initialize zero vector
    sx = np.zeros(n)

    # Fill in entries based on index randomly generated
    sx[mask] = 1

    return sx
```

In [4]:
```
# Random matrix A
np.random.standard_normal(size=(idx[-1],n)).shape
```

Out[4]: (50, 50)

In [5]:
```
# Random sparse vector
create_sx(idx[2])
```

Out[5]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 1., 0.,
       0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

3(iii) Defining error tolerance

In [6]:
```
# Define function to check for success
def success(sx, soln, TOL=1e-4):
    """
    Args:
    ===
    sx: Sparse vector
    soln: Optimal solution x
    TOL: Tolerance level

    Output:
    ===
    Return True if relative error is within tolerance level, otherwise return False.
    """
    # Relative error under L2-norm
    err = np.linalg.norm(soln-sx) / np.linalg.norm(sx)

    return err <= TOL
```

3(iv) Solving the problem

Run solver

In [7]:
```
# Define function for solver
def solver(A, y):
    """
    Args:
    ===
    A: Random matrix A
    y: Matrix multiplication of random matrix A and sparse vector sx

    Output:
    ===
    Return optimal solution x.
    """
    # Construct problem and run solver
    x = cp.Variable(shape=n)
    objective = cp.Minimize(cp.norm(x,1))
    constraints = [y == A@x]
    problem = cp.Problem(objective, constraints)
    problem.solve()

    # Retrieve optimal solution x
    soln = x.value

    return soln
```

In [8]:
```
# Define function to run solver for all values of m and s
def run_solver(solver=solver, create_sx=create_sx, n=n, niter=10, idx=idx):
    """
    Args:
    ===
    solver: Function for solver
    create_sx: Function to create sparse vector
    n: Fixed parameter
    niter: Number of iterations for each pair of m and s
    idx: Range of values for m and s

    Output:
    ===
    Return matrix recording the successes for each pair of m and s.
    """
    # Initialize matrix to record success for each pair of m and s
    success_mat = np.zeros(shape=(n,n))

    for m in tqdm(idx, desc="Progress bar"):
        for s in idx:
            for _ in range(niter):
                # Initialize parameters for solver
                sx = create_sx(s)
                A = np.random.standard_normal(size=(m,n))
                y = A @ sx

                # Run solver
                soln = solver(A,y)

                # Record success
                success_mat[m-1, s-1] += success(sx, soln)

    return success_mat
```

In [9]:
```
##############################
### Around 3 minutes to run ###
##############################

# Run solver for all values of m and s
np.random.seed(SEED)
success_mat = run_solver()
```

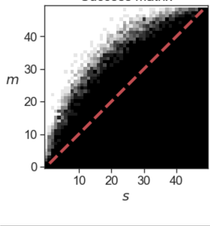
```
Progress bar: 100%|██████████| 50/50 [02:28<00:00, 2.97s/it]
```

Results

In [10]:
```
# Average across all pairs of m and s
avg_success = np.mean(success_mat)

print(f"Across all pairs of m and s, we have an average of {avg_success} successes out of the 10 trials.")
```

```
Across all pairs of m and s, we have an average of 2.726 successes out of the 10 trials.
```

In [11]:
```
# Plot grayscale image of success matrix
plt.imshow(success_mat, cmap="gray", origin="lower")
# Plot 45-degree reference line
plt.plot([1,n-2], [1,n-2], ls="--", c="r", lw=4)

plt.xlabel("$s$")
plt.xticks([i*10 for i in range(1,5)])
plt.ylabel("$m$", rotation="horizontal", labelpad=15)
plt.title("Success matrix")

plt.show()
```



Chong Zhen Jie A0201613Y

Assignment Question 4: Matrix Completion

In [1]:
```
import cvxpy as cp
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
```

In [2]:
```
sns.set_theme(font_scale=1.5)
SEED = 5232
```

4(ii) Generating random matrix $Z \in \mathbb{R}^{m \times n}$

In [3]:
```
# Size of Z is m by n
m, n = 100, 100

# Rank of Z is r=2
r = 2

np.random.seed(SEED)
U = np.random.standard_normal(size=(m,r))
V = np.random.standard_normal(size=(n,r))
Z = U @ V.T

print(f"Size of Z: {Z.shape}")
```

```
Size of Z: (100, 100)
```

4(iii) Selecting random entries Ω, where $|\Omega| = k$

In [4]:
```
# Define function to select the locations of k number of entries from Z
def select_entries(k, m=m, n=n):
    """
    Args:
    ===
    k: Number of random entries to select
    m: Number of rows in Z
    n: Number of columns in Z

    Output:
    ===
    Return Boolean masks for both observed and unobserved entries.
    """
    # Percent of entries to select
    percent = (k / (m*n)) * 100

    # Create Boolean mask
    rand = np.random.rand(m,n)
    mask = rand <= np.percentile(rand, percent)

    # Boolean mask for observed entries used in optimization, tr for train
    tr = mask

    #Boolean mask for unobserved entries used to compute MSE, tt for test
    tt = ~mask

    return tr, tt
```

4(iv) Solving the problem

Evaluation metric: mean-squared error (MSE)

In [5]:
```
# Define function to compute MSE
def compute_MSE(actual, pred, tt):
    """
    Args:
    ===
    actual: given data in the form of matrix Z
    pred: estimated X after optimization
    tt: Boolean mask of unobserved entries

    Output:
    ===
    Return MSE.
    """
    # Retrieve unobserved entries by indexing, and then take the difference to find an array of errors
    error = pred[tt] - actual[tt]
    # Compute MSE by formula
    MSE = np.mean(np.power(error, 2))

    return MSE
```

Running the solver

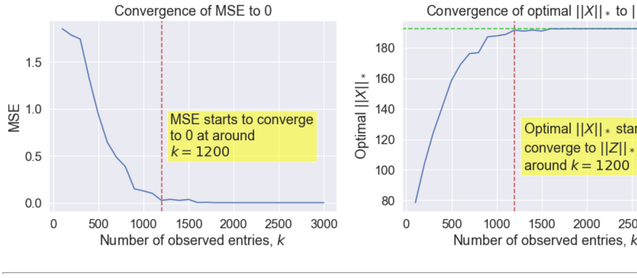
In [6]:
```
# Define function for solver
def run_solver(k, Z=Z):
    """
    Args:
    ===
    k: Number of random entries to select
    Z: given data in the form of matrix Z

    Output:
    ===
    Return optimal objective value and MSE.
    """
    # Generate Boolean masks, tr for solver and tt for computing MSE
    tr, tt = select_entries(k)

    # Construct problem and run solver
    X = cp.Variable(shape=(m,n))
    obj_fn = cp.normNuc(X)
    constraints = [ X[tr] == Z[tr] ]
    problem = cp.Problem(cp.Minimize(obj_fn), constraints)
    problem.solve()

    # Retrieve optimal objective value and optimal solution X
    obj, soln = problem.value, X.value

    # Compute MSE
    MSE = compute_MSE(Z, soln, tt)

    return obj, MSE
```

In [7]:
```
################################
### Around 45 minutes to run ###
################################

# Run the solver for all k = 100,200,...,3000
k_list = [i*100 for i in range(1,31)]
results = list( map(run_solver, tqdm(k_list, desc="Progress bar")) )

obj_list = [i[0] for i in results]
MSE_list = [i[1] for i in results]
```

```
Progress bar: 100%|██████████████████████████████████████████████████████████| 30/30 [41:02<00:00, 82.10s/it]
```

Results

In [8]:
```
plt.figure(figsize=(13,5))

# MSE against r
plt.subplot(1,2,1)
plt.plot(k_list, MSE_list)
plt.axvline(1200, ls="--", c="r")
plt.text(1200+100, 0.5,
         s="MSE starts to converge\nto 0 at around\n$k=1200$",
         bbox=dict(facecolor="yellow",alpha=0.5))
plt.xlabel("Number of observed entries, $k$")
plt.ylabel("MSE")
plt.title("Convergence of MSE to 0")

# Nuclear norm of Z
nuc_Z = np.linalg.norm(Z, ord="nuc")

# Objective value against r
plt.subplot(1,2,2)
plt.axvline(1200, ls="--", c="r")
plt.axhline(nuc_Z, ls="--", c="limegreen")
plt.plot(k_list, obj_list)
plt.text(1200+100, 100,
         s="Optimal $||X||_*$ starts to\nconverge to $||Z||_*$ at\naround $k=1200$",
         bbox=dict(facecolor="yellow",alpha=0.5))
plt.xlabel("Number of observed entries, $k$")
plt.ylabel("Optimal $||X||_*$")
plt.title("Convergence of optimal $||X||_*$ to $||Z||_*$")

plt.tight_layout()
plt.show()
```



Chong Zhen Jie A0201613Y

Assignment Question 5: Movie Lens

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
```

In [2]:
```
sns.set_theme(font_scale=1.5)
SEED = 5232
```

5(i) Loading of data

In [3]:
```
col_names = ["user_id", "movie_id", "ratings"]
filepath = "data\data.csv"

data = pd.read_csv(filepath, names=col_names)
print(f"Size of dataset: {data.shape}")

data.head()
```

```
Size of dataset: (100000, 3)
```

Out[3]:

	user_id	movie_id	ratings
0	196	242	3
1	186	302	3
2	22	377	1
3	244	51	2
4	166	346	1

5(ii) Random train-test split

In [4]:
```
np.random.seed(SEED)

# Create boolean mask
rand = np.random.rand(len(data))
mask = rand <= np.percentile(rand, 90)

# Index dataset based on boolean mask
train = data.iloc[mask]
test = data.iloc[~mask]

print(f"Size of training dataset: {train.shape}")
print(f"Size of testing dataset: {test.shape}")
```

```
Size of training dataset: (90000, 3)
Size of testing dataset: (10000, 3)
```

5(iii) Baseline estimator

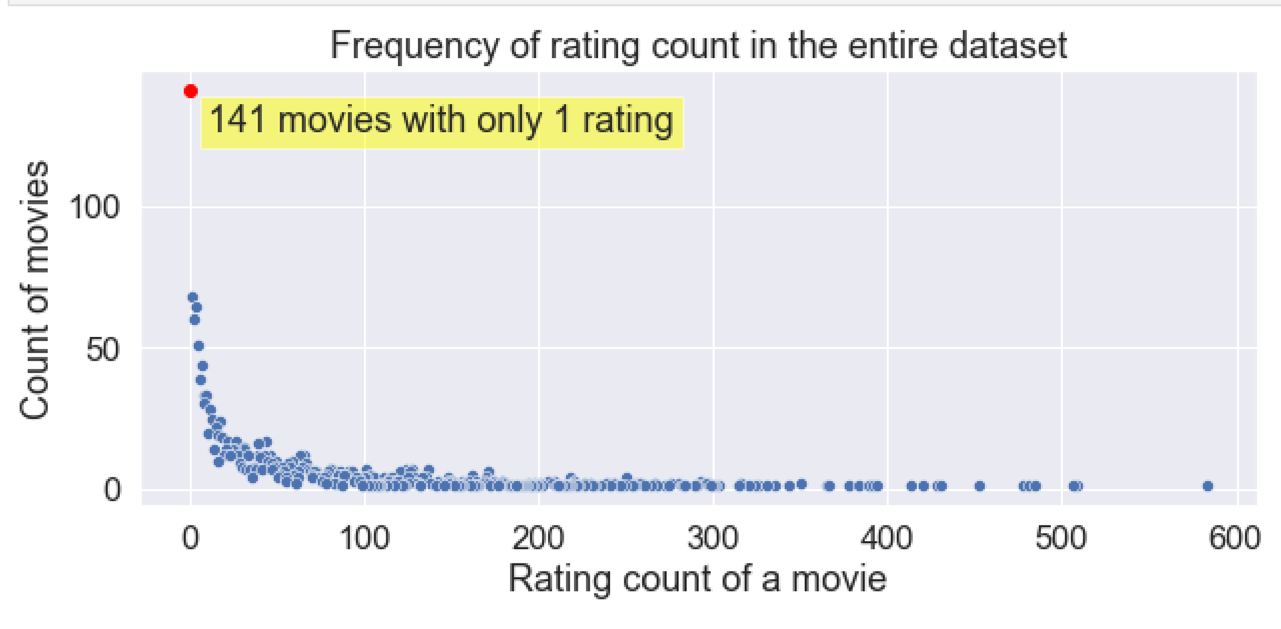
Exploratory data analysis

In [5]:
```
data.describe().T
```

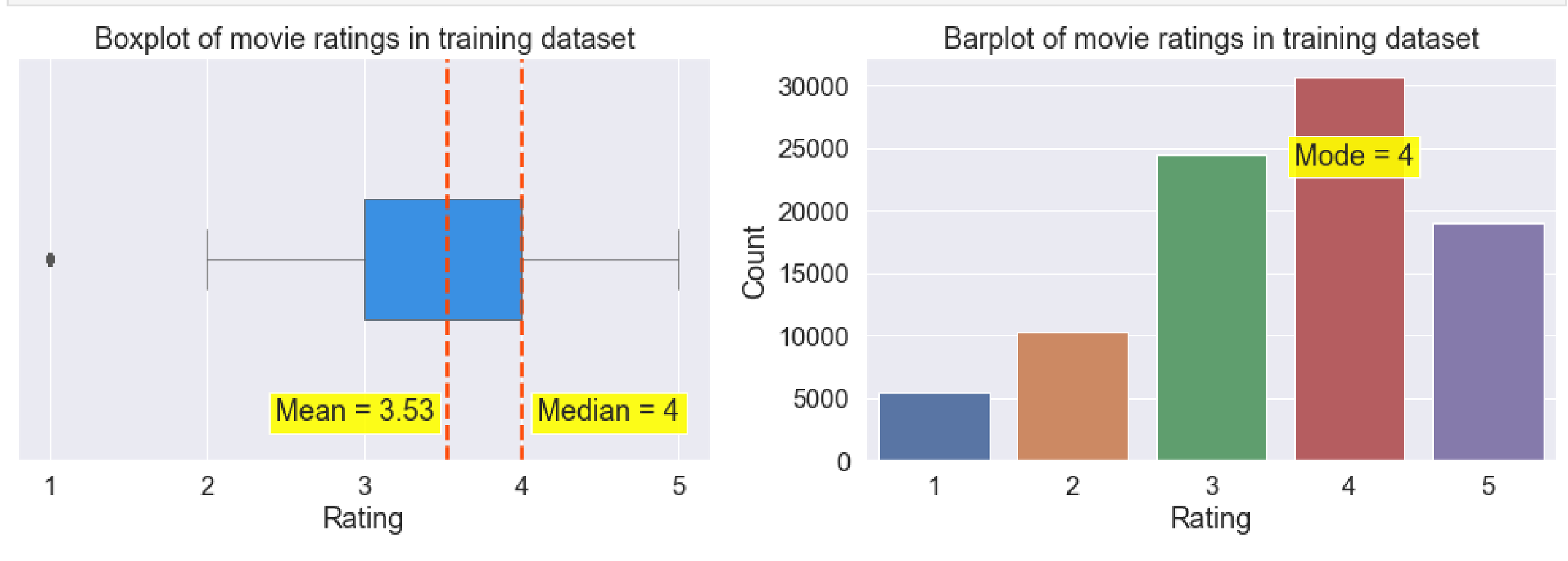
Out[5]:

	count	mean	std	min	25%	50%	75%	max
user_id	100000.0	462.48475	266.614420	1.0	254.0	447.0	682.0	943.0
movie_id	100000.0	425.53013	330.798356	1.0	175.0	322.0	631.0	1682.0
ratings	100000.0	3.52986	1.125674	1.0	3.0	4.0	4.0	5.0

In [6]:
```
movie_counts = data["movie_id"].value_counts() # Counts total ratings for each movie
rating_freq = movie_counts.value_counts() # Counts the frequency of each unique total ratings
min_ratings = rating_freq.min()

# Scatterplot
plt.figure(figsize=(10,4))
sns.scatterplot(x=rating_freq.index, y=rating_freq.values)
plt.scatter(min_ratings, rating_freq.loc[min_ratings], color="red")
plt.text(min_ratings+10, rating_freq.loc[min_ratings]-15,
         s=f"{rating_freq.loc[min_ratings]} movies with only {min_ratings} rating",
         bbox=dict(facecolor="yellow",alpha=0.5))
plt.ylabel("Count of movies")
plt.xlabel("Rating count of a movie")
plt.title("Frequency of rating count in the entire dataset")

plt.show()
```



In [7]:
```
rating_col = train["ratings"]
median_rating, mean_rating, mode_rating = int(rating_col.median()), rating_col.mean(), rating_col.mode()[0]
rating_counts = rating_col.value_counts() # Counts number of ratings for each integer value

plt.figure(figsize=(14,5))
# Boxplot
plt.subplot(1,2,1)
sns.boxplot(x=rating_col, color="dodgerblue", width=0.3, linewidth=.7)
plt.axvline(median_rating, color="orangered", ls="--", lw=2.5)
plt.axvline(mean_rating, color="orangered", ls="--", lw=2.5)
plt.text(median_rating + .1, .4, s=f"Median = {median_rating}", bbox=dict(facecolor="yellow",alpha=0.9))
plt.text(mean_rating - 1.1, .4, s=f"Mean = {mean_rating:.2f}", bbox=dict(facecolor="yellow",alpha=0.9))
plt.xlabel("Rating")
plt.xticks([i for i in range(1,6)])
plt.title("Boxplot of movie ratings in training dataset")

# Barplot
plt.subplot(1,2,2)
sns.barplot(x=rating_counts.index, y=rating_counts.values)
plt.text(mode_rating-1.4, rating_counts.max()-7000, s=f"Mode = {mode_rating}", bbox=dict(facecolor="yellow",alp
plt.xlabel("Rating")
plt.ylabel("Count")
plt.title("Barplot of movie ratings in training dataset")

plt.tight_layout()
plt.show()
```



Building baseline estimator

In [8]:
```
# Compute predictions of baseline estimator
base_pred = train.groupby("movie_id").mean()[["ratings"]].reset_index().rename(columns={"ratings": "base_pred"}

# Merge testing data with predictions using left-join
baseline = test.merge(base_pred, how="left", on="movie_id")
print(f"Number of missing entries in baseline predictions: {baseline.isna().sum().sum()}")

# Fill NA values with the median rating in training dataset
baseline["base_pred"] = baseline["base_pred"].fillna(median_rating)
print(f"Number of missing entries after filling NA values: {baseline.isna().sum().sum()}")

baseline.head()
```

```
Number of missing entries in baseline predictions: 13
Number of missing entries after filling NA values: 0
```

Out[8]:

	user_id	movie_id	ratings	base_pred
0	305	451	3	3.328947
1	200	222	5	3.674772
2	97	194	3	4.072398
3	25	181	5	3.975281
4	279	154	5	3.700000

Evaluation metric: root-mean-squared error (RMSE)

In [9]:
```
# Define function to compute RMSE
def compute_RMSE(pred, test=test):
    """
    Args:
    ===
    pred: Predicted ratings
    test: Testing dataset

    Output:
    ===
    Return RMSE.
    """
    # Extract actual ratings
    actual = test["ratings"].to_numpy()

    # Convert to integers due to integer constraint
    pred = np.round(pred)

    # Evaluate metric
    val = np.mean(np.power(actual-pred, 2)) ** 0.5

    return val
```

Results of baseline estimator

In [10]:
```
# Compute RMSE for baseline estimator
baseline_RMSE = compute_RMSE(baseline["base_pred"].to_numpy())

print(f"RMSE of baseline estimator: {baseline_RMSE:.4f}")
```

```
RMSE of baseline estimator: 1.0599
```

5(iv) Matrix completion algorithm

In [11]:
```
# Size of X is m by n
m, n = data["user_id"].nunique(), data["movie_id"].nunique()

# Initialize zero matrix
X_init = np.zeros(shape=(m,n))

print(f"Size of X: {X_init.shape}")
```

```
Size of X: (943, 1682)
```

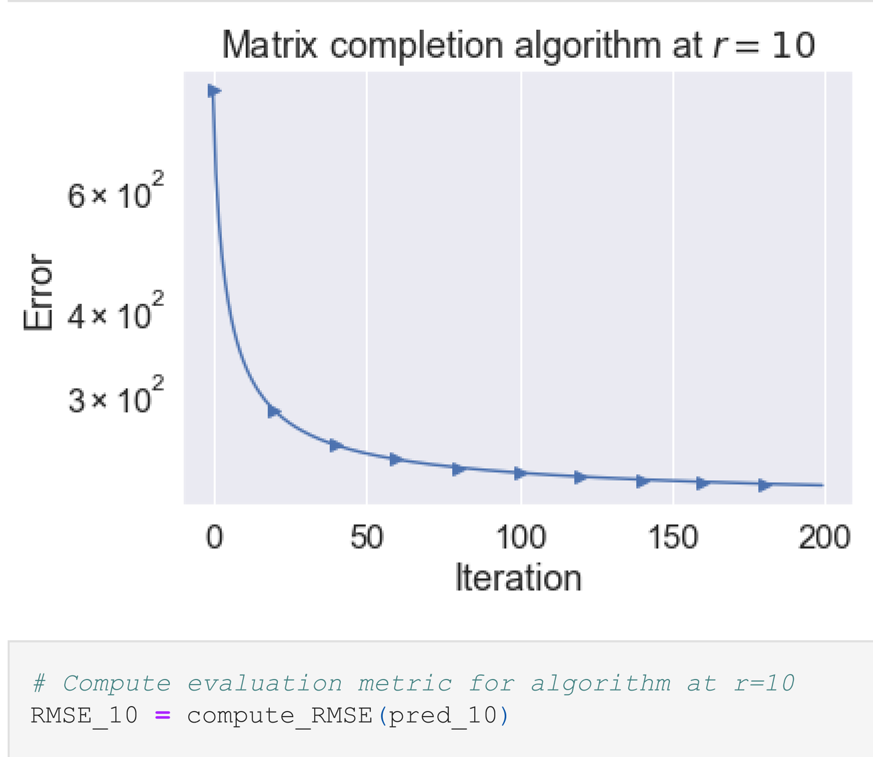
In [12]:
```
# Define function for matrix completion algorithm
def mat_com_algo(r, niter=200, X_init=X_init, train=train, test=test):
    """
    Args:
    ===
    r: rank-r approximation of X
    niter: Number of iterations to run the algorithm
    X_init: Initial value of X
    train: Training dataset
    test: Testing dataset

    Output:
    ===
    Return the list of errors and predicted ratings.
    """
    errors = []

    # Set X to initial value
    X = X_init

    for _ in range(niter):
        # (a) Set entries to correspond to training dataset
        X[train["user_id"]-1, train["movie_id"]-1] = train["ratings"]

        # (b) Compute SVD of X
        U, S, V_T = np.linalg.svd(X, full_matrices=False)

        # (c) Set all but the first r singular values to be zero
        S_new = S.copy()
        S_new[r:] = 0

        # (d) Compute low rank approximation of X with rank = r
        X_new = U @ np.diag(S_new) @ V_T

        # (e) Record error, defined by frobenius norm of the matrix difference
        err = X - X_new
        err = np.linalg.norm(err, ord="fro")
        errors.append(err)

        # (f) Set X to be the new approximated X
        X = X_new

    # Clip the values
    X = np.clip(X, a_min=1, a_max=5)

    # Extract predicted ratings
    pred = X[test["user_id"]-1, test["movie_id"]-1]

    return errors, pred
```

5(v) Testing the algorithm at $r=10$

In [13]:
```
# Run algorithm at r=10
r = 10
errors_10, pred_10 = mat_com_algo(r)
```

In [14]:
```
# Define function for plot customizations
def plot_custom(title):
    plt.xlabel("Iteration")
    plt.ylabel("Error")
    plt.yscale("log")
    plt.title(title)
```

In [15]:
```
# Plot errors for matrix completion at r=10
plt.plot(errors_10, "->", markevery=20)
plot_custom(f"Matrix completion algorithm at $r={r}$")

plt.show()
```



In [16]:
```
# Compute evaluation metric for algorithm at r=10
RMSE_10 = compute_RMSE(pred_10)

print(f"RMSE of algorithm at r=10: {RMSE_10:.4f}")
```

```
RMSE of algorithm at r=10: 1.0465
```

5(vi) Results of algorithm for all $r=1,...,20$

In [17]:
```
################################
### Around 40 minutes to run ###
################################

# Run algorithm for r=1,...,20
results = list( map(mat_com_algo, tqdm(range(1,21), desc="Progress bar")) )
```

```
Progress bar: 100%|████████████████████████████████████████| 20/20 [39:48<00:00, 11
9.45s/it]
```

In [18]:
```
errors_list = [i[0] for i in results]
pred_list = [i[1] for i in results]
```

In [19]:
```
# Plot errors for all r=1,...,20
plt.figure(figsize=(13,6))
for i in range(len(errors_list)):
    plt.subplot(1,2,1)
    plt.plot(errors_list[i], "->", markevery=20)
    plot_custom("Convergence of algorithm for all $r=1,...,20$")
    if i%5 == 0:
        plt.subplot(1,2,2)
        plt.plot(errors_list[i], "->", markevery=20, label=f"$r={i+1}$")
        plt.legend()
        plot_custom("General pattern of increasing $r$")

plt.tight_layout()
plt.show()
```

In [20]:
```
# Collate RMSE together with the corresponding rank parameter r
RMSE_list = [compute_RMSE(pred) for pred in pred_list]
RMSE_df = pd.DataFrame({"r": [f"{i}" for i in range(1,21)], "RMSE": RMSE_list}).set_index("r")

RMSE_df.T
```

Out[20]:

r	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
RMSE	1.006032	1.002996	0.986864	0.98904	0.99965	1.001948	1.016858	1.019804	1.035809	1.04647	1.055746	1.068644	1.078239	1.08821	1

In [21]:
```
# Plot RMSE against r
plt.figure(figsize=(10,6))
plt.plot(RMSE_df.index, RMSE_df["RMSE"], marker="o")

# Plot RMSE of baseline estimator as reference
plt.axhline(baseline_RMSE, ls="--", c="r")
plt.text(0.1, baseline_RMSE+0.01, s=f"Baseline\nRMSE of {baseline_RMSE:.4f}", bbox=dict(facecolor="yellow",alph

# Plot reference horizontal line to indicate minimum RMSE
min_RMSE = RMSE_df["RMSE"].min()
min_r = RMSE_df[RMSE_df["RMSE"]==min_RMSE].index[0]
plt.axhline(min_RMSE, ls="--", c="limegreen")
plt.text(12, min_RMSE+0.01, s=f"$\min$ RMSE of {min_RMSE:.4f}\nat $r={min_r}$", bbox=dict(facecolor="yellow",al

plt.xlabel("Rank parameter $r$")
plt.ylabel("RMSE")
plt.title("Performance of matrix completion algorithm at different choices of $r$")

plt.show()
```