SQUARE ROOT LASSO: WELL-POSEDNESS, LIPSCHITZ STABILITY AND NUMERICAL CONSEQUENCES*

AARON BERK†, SIMONE BRUGIAPAGLIA‡, AND TIM HOHEISEL†

Numerical implementation details

In this document we describe additional details of our numerical implementation to ensure transparency and reproducibility, particularly for the experiments in subsection 7.1. Refer, *e.g.,* to Python, `numpy` and `CVXPY` documentation for further details beyond what is covered here and/or our code repository.

N1. Checking set membership and numerical equality. Perhaps the most critical assumption underlying this work is that the solution $\bar{x}$ satisfy $A\bar{x} \neq b$. The Python numerical implementation mandates checking this condition up to some numerical tolerance, rather than checking for strict equality. We deemed a numerical solution `x_bar` to be *inexact* (*i.e.,* `b - A @ x_bar` $\neq 0$) if the following condition evaluates to `True`.

```
not np.allclose(b, A @ x_bar)
```

Equality of numerical objects was likewise evaluated similarly in other relevant cases. For instance we used the closely related function `np.isclose` to compare the left- and righthand sides of the (vector) relationship defining the equicorrelation set. As an aside, note that we computed the equicorrelation set using the numerical

```
np.isclose(np.abs(A.T.dot(y_bar)), lamda)
```

quantity `y_bar` rather than relying on the residual `r = b - A @ x_bar`. Theoretically, $r/\|r\|$ and $\bar{y}$ are equivalent, but this is not necessarily the case for their numerical counterparts. Empirically, we expected better robustness from using `y_bar` and found no good reason to extensively compare the two approaches.

Finally, to determine if `b` belongs to the range of `A[:, supp]`, where `supp` is the (numerical) support of `x_bar` (see section N2), we used `numpy` least squares, `np.linalg.lstsq`, with default value for the relative cut-off of small singular values of `A[:, supp]` (*i.e.,* `rcond=None`). Then, we deemed that `b` does not lie in the range of `A[:, supp]` if the resulting residual is numerically 0 (according to `np.allclose`). Refer to `is_submatrix_ranges_b` in our code for full details.

N2. Support computation. The support of a solution $\bar{x}$ can be described as $\{i \in [n] : \bar{x}_i \neq 0\}$. For its numerical counterpart `x_bar`, an effective computation of the support requires a numerical tolerance, due to the reasons mentioned above. Generally, this operation was performed using element-wise Boolean negation of `np.isclose`. Specifically, we selected the subset of the equicorrelation set containing (numerically) nonzero elements. Refer to the definition of `support` in our code for full details.

---

*March 24, 2023

†Department of Mathematics and Statistics, McGill University (aaron.berk@mcgill.ca, tim.hoheisel@mcgill.ca).

‡Department of Mathematics and Statistics, Concordia University (simone.brugiapaglia@concordia.ca).

Similarly, when checking the injectivity of the numerical submatrix analgoue to A_I, `A[:, supp]`, the matrix rank was computed using `np.linalg.matrix_rank` with the default numerical tolerance settings used by `numpy`.

N3. Empirical uniqueness sufficiency. We present the following Python pseudocode used to describe the chain of logic to determine if a given solution numerically satisfies the sufficient condition to be unique. The logic is written to verify each component of Assumption 1 and handle a few computational edge cases. This function by default returns `False` if the solution is not inexact (see line 5 of Algorithm N3.1), because our theory does not apply in this case. This is the only instance where `False` could be returned by the function in a setting where the solution could (potentially) be unique. We argue that this is a minor ambiguity due to the way the logic is handled in the remainder of the code for the generation of Figures 5 and 6. Refer to our code repository for full implementation details that clarify this point.

```python
def is_uniqueness_sufficiency(A, b, lamda, x_bar, y_bar, aux_value):
    if aux_value >= lamda:
        return False # violation!
    if not is_inexact_solution(A, b, x_bar):
        return False # N/A!
    supp = support(x_bar, A, b, lamda, y_bar)
    if supp.sum() == 0:
        return True # 0 in range(A[:, I])

    max_rank = supp.sum()
    rank = la.matrix_rank(A[:, supp])
    if rank < max_rank:
        return False # lacks rank!

    if is_submatrix_ranges_b(A, b, supp):
        return False # b in range(A[:, I])!

    # be strict in case of numerical discrepancy:
    Z0 = la.norm(A[:, mask].T.dot(y_bar), np.inf)
    lamda_empir = la.norm(A.T.dot(y_bar), np.inf)
    lamda_ = np.minimum(lamda, lamda_empir)

    J = equicorrelation(A, y_bar, lamda)
    if supp.sum() == J.sum():
        if np.isclose(Z0, lamda_):
            return False # contradicts |I| = |J|!
        return True # Strong Assumption holds
    ZZ = np.minimum(Z0, lamda_)
    if np.isclose(aux_value, ZZ) or (aux_value >= ZZ):
        return False # violation!
    return True
```

Algorithm N3.1: Implementation sketch for uniqueness sufficiency computation in Python-flavoured pseudocode.