1 Methods Used

We have used the following methods to solve the problem:

1. *Naive Technique*
2. *Coordinate Descent*
3. *Proximal Gradient Descent*
4. *Projected Gradient Descent*

1.1 Naive Technique

In this technique, we first solve the least squares problem by using the lstsq solver offered by the Numpy model. We then sparsify the solution using hard thresholding.

The lstsq solver returns a tuple of 4 outputs of which the first is the linear model which is why we used [0] to index out the first element of the tuple.

This method is not accurate as the model lacks parameters that gradually do the fitting.In this method, the model assigns the same importance to all data points and does not take into account the sparsity of the data set. That is why after hard thresholding the method gives inaccurate results

1.2 Coordinate Descent Method

In this approach, we solved the sparse recovery problem using coordinate gradient descent.

We implemented the coordinate descent algorithm for solving Lasso (Least Absolute Shrinkage and Selection Operator) and OLS (Ordinary Least Squares) regression problems. The coordinate descent algorithm is an iterative optimization algorithm that updates the weights of the regression model by optimizing one weight at a time while keeping the others fixed. It iterates over each feature and updates the corresponding weight based on specific calculations.

- *Lasso regression*: Lasso regression is a linear regression model that incorporates a regularization term using the L1 norm. It encourages sparsity in the weights, leading to feature selection.
- *OLS regression*: OLS regression is a linear regression model that aims to minimize the mean squared error without any regularization.

We first implemented the Lasso technique, but the weights were not converging and the model was taking a lot of training time. So, we then implemented the OLS technique which gave significantly better results. The weights were converging and keeping in mind the constraint of the problem, we applied soft thresholding after every iteration. It shrinks the weights by subtracting the threshold value and taking the maximum of zero and the result.

1.2.1 Equations

- Lasso Regression:

$$L(w) = \frac{1}{2N}\sum_{i=1}^{n}(y^{(i)} - \tilde{w}\cdot\tilde{x}^{(i)})^2 + \alpha\sum_{j=1}^{d}|w_j| = \text{MSE} + \alpha\cdot\text{norm}(|w|)$$

- OLS Regression

$$\frac{\partial L}{\partial w_j} = \frac{1}{N}\left(\sum_{i=1}^{n}(-x_j^{(i)})\left(y^{(i)} - \left(\sum_{k\neq j} w_k\tilde{x}_k^{(i)}\right)\right)\right) + \frac{1}{N}w_j\left(\sum_{i=1}^{n}x_j^{(i)}\right)^2 = -\frac{1}{N}\rho_j + \frac{1}{N}w_j z_j$$

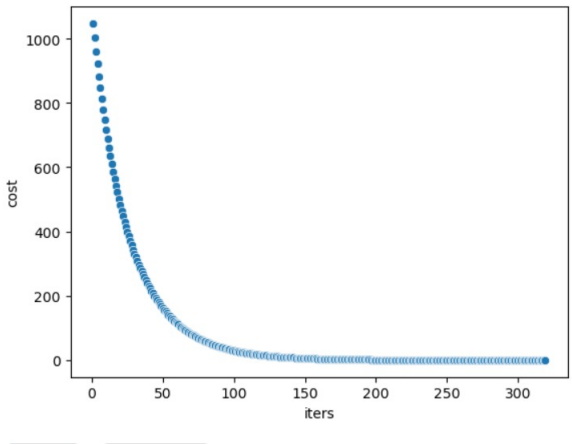


Figure 1: Cost vs. Iteration graph: Optimization progress visualization

```
Iteration 312 - MSE: 0.0796
Iteration 313 - MSE: 0.0772
Iteration 314 - MSE: 0.0749
Iteration 315 - MSE: 0.0727
Iteration 316 - MSE: 0.0706
Iteration 317 - MSE: 0.0685
Iteration 318 - MSE: 0.0665
Iteration 319 - MSE: 0.0645
```

Figure 2:

1.3.1 Proximal Gradient Descent

We model the problem as a least square regression problem and use $L1$ regularization in the LASSO to find sparse solutions.

$$\min_{w \in \mathbb{R}^D} \frac{1}{2}(\|b - Ax\|_2)^2 + \lambda(\|x\|_1)$$

The approaches mentioned above - *ISTA, FISTA and Adaptive FISTA* were compared in two different setups, i.e. with and without warm initialization. For warm initialization two approaches were used, initializing w by the all-zero vector i.e. $w_0 = 0$, the solution of the least squares problem i.e.

$$w_0 = \arg\min_{w \in \mathbb{R}^D} \ell(w)$$

For validation purposes, we made a random split of our data of 1600 CRPs into 1500 training and 100 for validation. For validation accuracy, Mean Absolute Error was used.

1.3.2 ISTA

Iterative Shrinkage/Thresholding Algorithm (ISTA) [**?**,]]daubechies2003iterative is a popular algorithm to solve with non-smooth convex regularizers. ISTA works by iterative gradient descent update followed by a soft-threshold step, a proximal operator in our case.

Algorithm: We iterate until x^{k+1} reaches convergence condition,

$$x^{(k+1)} = \tau_\alpha(x^{(k)} - \mu A^T(Ax^k - b))$$

where, $\mu = \frac{1}{L}$ is an appropriate step size, and L must be an upper bound on the largest *eigenvalue* of A^TA. We have $L = \|A\|_2^2$
.

$$\tau_\alpha = soft(x, \alpha)$$
$$soft(x, \alpha) = sign(x_i)\max\{|x_i| - \alpha, 0\}$$

The ISTA algorithm promotes sparsity by utilizing the $\|L_1\|_2$
term. This allows it to recover sparse solutions by encouraging most coefficients of the solution vector to be zero.

1.3.3 FISTA

[**?**,]]inproceedings Due to the unoptimized low convergence rate of ISTA, a faster version of it, the Fast Iterative Shrinkage/Thresholding Algorithm is used. The main improvement of FISTA is that the iterative shrinkage operator τ_α is not applied on the previous estimation x^k, but rather at y^k which adopts a well-designed linear combination of the last two estimates $x^{(k)}$, $x^{(k-1)}$. It adds a momentum term to increase the convergence rate, which is proportional to $x^{(k)} - x^{(k-1)}$.

$$x^{(k)} = \tau_\alpha(y^{(k)} - \mu A^T(Ay^{(k)} - b))$$
$$t^{(k+1)} = \frac{1 + \sqrt{1 + 4(t^{(k)})^2}}{2}$$
$$y^{(k+1)} = x^{(k)} + \left(\frac{t^{(k)} - 1}{t^{(k+1)}}\right)(x^{(k)} - x^{(k-1)})$$

FISTA achieves an $O\left(\frac{1}{k^2}\right)$ convergence rate, where k is the iteration number.

1.3.4 Adaptive FISTA

The convergence rate of FISTA can be still improved as found in [**?**,]]liang2018faster. In

$$t^{(k)} = \frac{1 + \sqrt{1 + 4(t^{(k-1)})^2}}{2}$$

we replace *1,1,4* with *p,q and r* respectively, such that –

$$t^{(k)} = \frac{p + \sqrt{q + r(t^{(k-1)})^2}}{2}$$

taking, $a_k = \frac{t^{(k-1)}-1}{t^{(k)}}$, it was found that *r* controls the limiting value of a_k and *p,q* control the speed of a_k converging to 1.
Thus, finally we took *p=1, q=1 and* $r_k = f(\alpha_k)$ where –

$$f(\alpha) = 4\frac{(1-\sqrt{\gamma\alpha})^2}{1-\gamma\alpha}$$

where, $\gamma = \frac{1}{L}$.

1.3.5 Results

We obtain the following results, when we execute the code for different conditions as mentioned in the table:

Table 1: Performance over different epochs and batch iterations

Epoch	Batch Iterations	MAE	RMSE
2	1000	1.30e-05	1.55e-05
2	1500	1.38e-05	1.66e-05
3	1000	2.92e-06	3.66e-06
4	1500	2.92e-06	3.66e-06
4	1000	2.92e-06	3.66e-06

We obtain the graph, as shown in *Figure 3* when we compare all the three approaches we used for the above method– *ISTA, FISTA, ADA-FISTA*

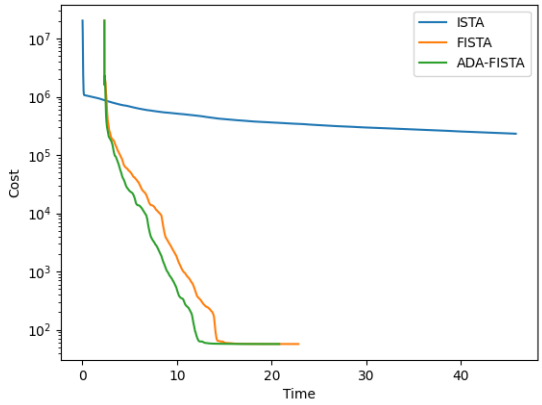


Figure 3: ISTA vs FISTA vs ADA-FISTA performance

We observe that the best result is obtained by the ADA-FISTA approach with **MAE: 2e-6** and **Time:15.8 secs**.

1.4 Projected Gradient Descent

In this approach, we solved the sparse recovery problem using projected gradient descent. To further improve the convergence rate of the algorithm, we implemented various initialization techniques like least square initialization, zero initialization, and one initialization, and noticed a little improvement in each of them. To improve further, we implemented the correction technique but it took 1 to 2 minutes to converge.

We further improved the algorithm minimizing the total mean squared error and reducing the training time significantly using the Nesterov momentum, which is a modification of traditional momentum that takes into account the expected future update of the weights.

By using Nesterov momentum, the algorithm estimates the next position of the weights before calculating the gradient. It then uses this estimation to adjust the momentum term. This technique allows the algorithm to take into account the future position of the weights, helping to anticipate and correct overshooting. As a result, Nesterov momentum accelerates the convergence rate, leading to faster optimization.

At last, we implemented L2 regularization to prevent overfitting and improve the general capability of the model. The objective function is defined as follows:

$$\ell(w) = \frac{1}{2n} \sum_i (w^\top x_i - y_i)^2 + \frac{\lambda}{2n} \|w\|^2$$

All the approaches mentioned above were compared in two different setups, i.e., with and without warm initialization. For warm initialization, three approaches were used: initializing w by the all-zero vector (i.e., $w_0 = 0$), initializing w by the all-one vector (i.e., $w_0 = 1$), and the solution of the least squares problem (i.e., $w_0 = \arg\min_{w \in \mathbb{R}^D} \ell(w)$).

For validation purposes, we made a random split of our data of 1600 CRPs into 1500 for training and 100 for validation. After this, we also made a random split into 1400 for training and 200 to check the robustness of the model. Mean Squared Error was used for validation accuracy.

1.4.1 Nesterov Momentum

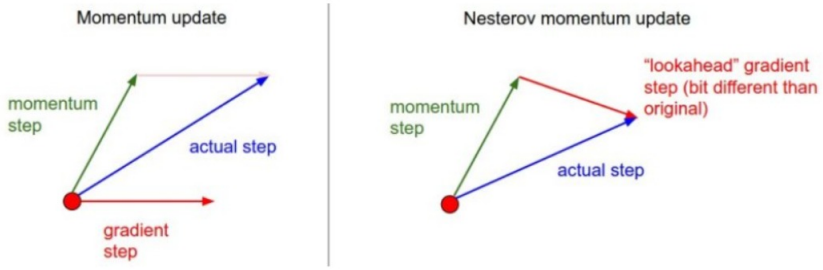
Nesterov momentum is an optimization technique that improves upon standard momentum by considering the gradient lookahead. Instead of using the current position for computing the gradient, it uses the position that would be reached by taking a step in the direction of the previous momentum. This "lookahead" position is calculated before applying the gradient update. By using the lookahead position, Nesterov momentum allows the algorithm to anticipate the effect of the momentum on the next step and adjust the current update accordingly. This correction helps to reduce oscillations and overshooting, leading to faster convergence.

1.4.2 Algorithm

1. Compute the lookahead position: $\phi_{t+1} = \theta_t + \mu \cdot v_t$
2. Compute the gradient at the lookahead position: $\nabla f(\phi_{t+1}) = \text{compute\_gradient}(\phi_{t+1})$
3. Update the momentum: $v_{t+1} = \gamma \cdot v_t - \eta \cdot \nabla f(\phi_{t+1})$
4. Update the parameters using the momentum-adjusted gradient: $\theta_{t+1} = \theta_t + v_{t+1}$

In these equations:

- θ_t represents the current parameters at iteration t.
- v_t represents the momentum at iteration t.
- μ is the momentum decay rate.
- $\nabla f(\phi_{t+1})$ is the gradient of the objective function evaluated at the lookahead position ϕ_{t+1}.
- γ is the momentum factor.
- η is the learning rate or step size.



1.4.3 Results

Grid search was used to find the best performing i.e best consistent validation accuracy giving learning rate, max iterations momentum factor, and regularization constant, which were –

```
Best Parameters:
Learning Rate: 0.005
Max Iterations: 3000
Momentum Factor: 0.99
Regularization Lambda: 0.01
```

Figure 4:

```
Iteration: 0, Loss: 5036308.14095696
Iteration: 300, Loss: 18.201323377308878
Iteration: 600, Loss: 0.4311015128420305
Iteration: 900, Loss: 0.18576183845731895
Iteration: 1200, Loss: 0.18268284944474997
Iteration: 1500, Loss: 0.182645742822262
Iteration: 1800, Loss: 0.18264589620124216
Iteration: 2100, Loss: 0.18264596596576885
Iteration: 2400, Loss: 0.1826459736076891
```

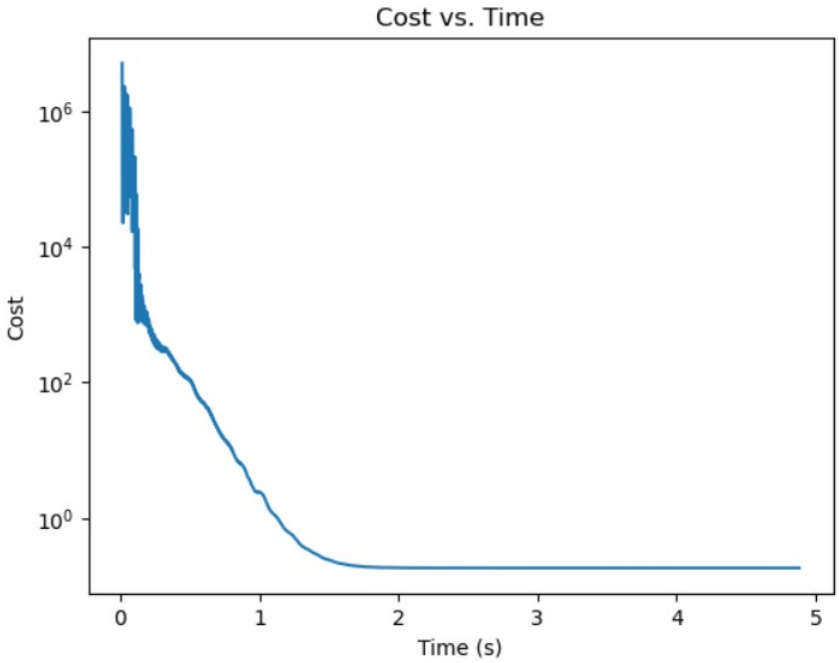


Figure 5: Cost vs. Time graph: Optimization progress visualization

```python
y_pred = X_val.dot(weights)
mse = np.mean((y_pred - y_val) ** 2)
print(f"Test MSE: {mse}")
```

```
Test MSE: 2.1453442588382175e-12
```

2 HyperParameter Description

1. *Regularization constant (Lambda)*:When taken in large values the model had low convergence rate, whereas using smaller values made it unable to learn sparse representations.

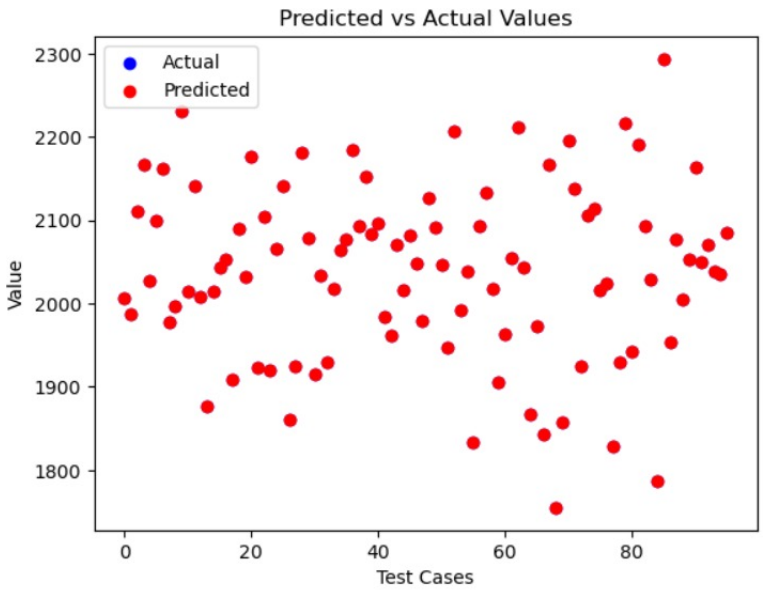


Figure 6: Actual vs. Projected Values

2. *Max iterations*:Tradeoff was made between model accuracy and convergence time, to give optimal sparse representation in minimal time.

3. *Learning rate*:Large learning rate lead to an oscillating descent when close to convergence, whereas taking a small value required more iterations to converge.

4. *momentum*:On taking large values it was observed that it model overshoots minima, which made the model to diverge and oscillate prooving it difficult to converge in limited iterations, whereas lowering it made the model to take too many iterations to converge.

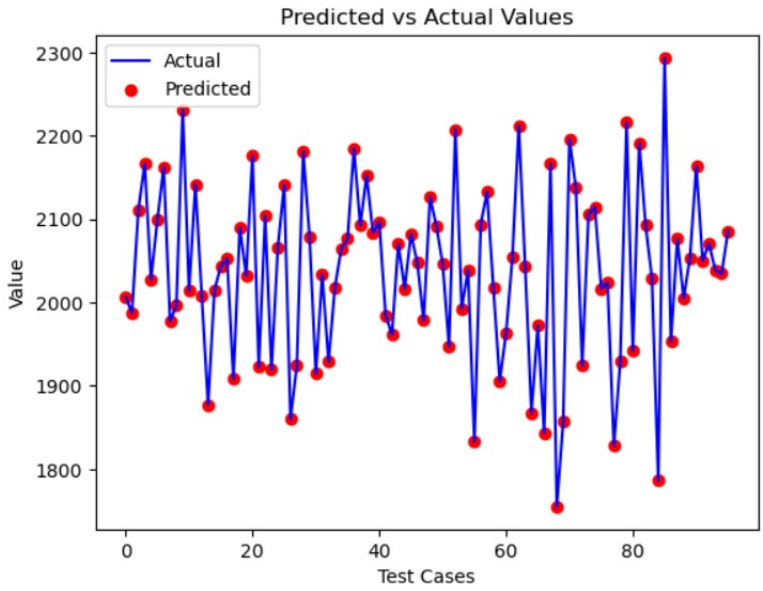
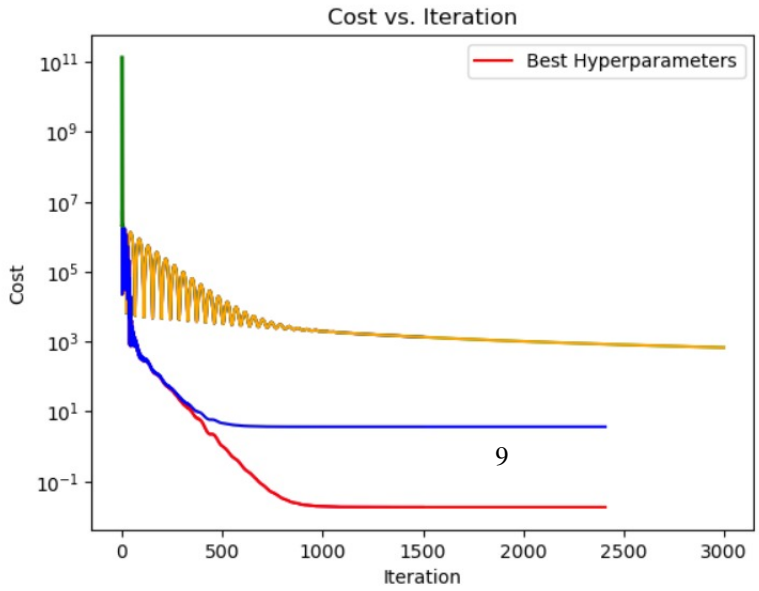


Figure 7: Actual vs. Projected Values

```
count = np.count_nonzero(weights ==0)
count1 = np.count_nonzero(weights <1)
count2 = np.count_nonzero(weights >=1)
count3 = np.count_nonzero(weights <0.000001) ## ignoring weights tending to zero
print('number of connected CDUs : ', count2 + count1 - count3)
```

```
number of connected CDUs :  512
```

Figure 8: Result



```
Iteration: 0, Loss: 5036307.970530282
Iteration: 400, Loss: 4.808139424476537
Iteration: 800, Loss: 0.032448758294341344
Iteration: 1200, Loss: 0.018309185419122985
{'learning_rate': 0.005, 'max_iterations': 1500, 'momentum_factor': 0.99, 'regularization_lambda': 0.001}

Iteration: 0, Loss: 5036308.14095696
Iteration: 400, Loss: 4.970154016196994
Iteration: 800, Loss: 0.196715599689074
Iteration: 1200, Loss: 0.18268284944474997
{'learning_rate': 0.005, 'max_iterations': 1500, 'momentum_factor': 0.99, 'regularization_lambda': 0.01}

Iteration: 0, Loss: 5036307.970530282
Iteration: 400, Loss: 4.808139424476537
Iteration: 800, Loss: 0.032448758294341344
Iteration: 1200, Loss: 0.018309185419122985
Iteration: 1600, Loss: 0.01826469521422942
Iteration: 2000, Loss: 0.018264595634363164
Iteration: 2400, Loss: 0.018264597361943143
{'learning_rate': 0.005, 'max_iterations': 3000, 'momentum_factor': 0.99, 'regularization_lambda': 0.001}

Iteration: 0, Loss: 5036308.14095696
Iteration: 400, Loss: 4.970154016196994
Iteration: 800, Loss: 0.196715599689074
Iteration: 1200, Loss: 0.18268284944474997
Iteration: 1600, Loss: 0.18264575198581343
Iteration: 2000, Loss: 0.18264595243371104
Iteration: 2400, Loss: 0.1826459736076891
{'learning_rate': 0.005, 'max_iterations': 3000, 'momentum_factor': 0.99, 'regularization_lambda': 0.01}

Iteration: 0, Loss: 19799719.009645972
{'learning_rate': 0.008, 'max_iterations': 1500, 'momentum_factor': 0.99, 'regularization_lambda': 0.001}

Iteration: 0, Loss: 19799719.44593827
{'learning_rate': 0.008, 'max_iterations': 1500, 'momentum_factor': 0.99, 'regularization_lambda': 0.01}

Iteration: 0, Loss: 19799719.009645972
{'learning_rate': 0.008, 'max_iterations': 3000, 'momentum_factor': 0.99, 'regularization_lambda': 0.001}

Iteration: 0, Loss: 19799719.44593827
{'learning_rate': 0.008, 'max_iterations': 3000, 'momentum_factor': 0.99, 'regularization_lambda': 0.01}
```

Hyperparameters value

References

1

[1] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm with application to wavelet-based image deblurring. pages 693–696, 04 2009.

[2] Ingrid Daubechies, Michel Defrise, and Christine De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint, 2003.

[3] Jingwei Liang and Carola-Bibiane Schönlieb. Faster fista, 2018.