High-Dimensional Regression: LASSO and AdaBoost

Abdallah Abdelsameia, Louis Rodriguez

Toulouse School of Economics - M2 Data Science for Social Sciences

March 2025

Outline

Introduction

- **High-dimensional regression:** We often have $p \gg n$, or at least p comparable to n, where many features may be redundant.
- **LASSO** addresses this by:
  - Minimizing a penalized least squares objective:

$$\min_{b \in \mathbb{R}^p} \frac{1}{n} \|Y - Xb\|_2^2 \; + \; 2\tau \|b\|_1,$$

    where $\|b\|_1 = \sum_j |b_j|$.
  - Encouraging *sparsity* in b: many coefficients become zero.
  - Simultaneously performing variable selection and shrinkage.
- In this presentation, we focus on **four LASSO variants**:
  - Coordinate Descent (CD)
  - ISTA (Iterative Shrinkage-Thresholding)
  - FISTA (Fast ISTA with momentum)
  - Square-root Lasso (scaled Lasso)
- **We run experiments** on synthetic data to see:
  - How these methods behave under different design conditions (correlation, dimensionality),
  - Their ability to recover the true sparse support,
  - Their runtime and convergence properties.

LASSO: Coordinate Descent (Method 1)

```python
import numpy as np

def soft_thresh(r, lam):
    return np.sign(r) * max(abs(r) - lam, 0)

def lasso_cd(X, y, lam, max_iter=5000, tol=1e-4):
    n, p = X.shape
    b = np.zeros(p)
    for it in range(max_iter):
        b_old = b.copy()
        for j in range(p):
            r = y - X @ b + b[j] * X[:, j]
            rho = (X[:, j] @ r) / n
            norm_j = (X[:, j]**2).sum() / n
            b[j] = soft_thresh(rho, lam) / norm_j
        if np.linalg.norm(b - b_old) < tol:
            break
    return b
```

Implementation Description:

- Initialize β to zeros.
- **Loop** over a maximum of `max_iter`:
  - **For each coordinate** j:
    1. Compute a *partial residual* $r = y - X\beta + \beta_j X_{:,j}$.
    2. Find correlation ρ between r and $X_{:,j}$.
    3. Compute $b[j]$ by *soft thresholding* ρ, scaled by the feature norm.
  - Check if the updated β changes less than `tol`.
- Return β when converged or after max iterations.

LASSO: ISTA (Method 2)

```python
def soft_thresh(r, lam):
    return np.sign(r) * max(abs(r) - lam, 0)

def lasso_ista(X, y, lam, L=None,
               max_iter=1e4, tol=1e-4):
    n, p = X.shape
    if L is None:
        L = (np.linalg.norm(X, 2)**2)/n
    b = np.zeros(p)
    for k in range(int(max_iter)):
        b_old = b.copy()
        grad = (X.T @ (X @ b - y)) / n
        b = soft_thresh(b - grad / L, lam / L)
        if np.linalg.norm(b - b_old) < tol:
            break
    return b
```

Implementation Description:

- **Initialize** β as zeros.
- **Estimate Lipschitz constant** $L \approx \frac{\|X\|_2^2}{n}$ if not provided.
- **For each iteration**:
  1. Compute gradient of the least-squares term: $\nabla f(\beta)$.
  2. Perform *proximal* update:
  $$\beta \leftarrow \text{soft\_thresh}\big(\beta - \frac{1}{L}\nabla f,\ \frac{\lambda}{L}\big).$$
  3. Check convergence on $\|\beta - \beta_{\text{old}}\|$.
- Return β.

LASSO: FISTA (Method 3)

```python
def soft_thresh(r, lam):
    return np.sign(r) * max(abs(r) - lam, 0)

def lasso_fista(X, y, lam, L=None,
                max_iter=1e4, tol=1e-4):
    n, p = X.shape
    if L is None:
        L = (np.linalg.norm(X, 2)**2)/n
    b = np.zeros(p)
    yk = b.copy()
    t = 1.0
    for k in range(int(max_iter)):
        b_old = b.copy()
        grad = (X.T @ (X @ yk - y)) / n
        b = soft_thresh(yk - grad / L, lam / L)
        t_new = 0.5 * (1 + (1 + 4*t*t)**0.5)
        yk = b + ((t - 1)/t_new) * (b - b_old)
        t = t_new
        if np.linalg.norm(b - b_old) < tol:
            break
    return b
```

Implementation Description:

- Similar to ISTA but uses an **acceleration** (Nesterov momentum).
- We keep track of:
  - A separate `yk` vector (the "extrapolated" point),
  - A momentum scalar t updated each iteration.
- Each iteration:
  1. Gradient step on `yk`.
  2. Apply soft-threshold with λ/L.
  3. Update t_{k+1} and combine the new b with old b to form `yk`.
- Typically faster convergence than ISTA.

Square-root Lasso (Method 4)

```python
def sqrt_lasso(X, y, tau,
               max_outer=100, tol=1e-4):
    n, p = X.shape
    sigma = np.linalg.norm(y)/np.sqrt(n)
    b = np.zeros(p)
    for it in range(max_outer):
        b_old, s_old = b.copy(), sigma
        # solve LASSO subproblem
        b = lasso_fista(X, y, lam=tau*sigma)
        r = y - X@b
        sigma = np.linalg.norm(r)/np.sqrt(n)
        if (abs(sigma - s_old)<tol and
            np.linalg.norm(b-b_old)<tol):
            break
    return b, sigma
```

Implementation Description:

- We minimize:

$$\min_{b,\sigma>0} \frac{\sigma}{2} + \frac{\|y - Xb\|^2}{2\sigma n} + \tau\|b\|_1.$$

- **Initialize** σ from $\|y\|/\sqrt{n}$.
- **Outer loop**:
  1. *Fix* σ, solve standard LASSO with $\lambda = \tau \cdot \sigma$ (using FISTA).
  2. Update σ from new residual $\|y - Xb\|/\sqrt{n}$.
  3. Check if both b and σ changed below `tol`.
- Returns (b, σ) upon convergence.

Experiment Roadmap

- We design various synthetic setups to test LASSO performance:
  1. **Baseline (make_regression):** Low correlation, moderate dimension.
  2. **Toeplitz Correlated:** Vary correlation ρ.
  3. **High-Dimensional Scale-Up:** Fix n, grow p.
- Compare methods on:
  - Support recovery
  - False positives
  - Runtime
- Next slides: results and highlights for each scenario.

make_regression: Key Highlights

Generate Synthetic Regression Data

What It Does:

- Generates data using a random linear model.
- Constructs outputs from a subset of `n_informative` features.
- Optionally adds Gaussian noise to simulate real-world data.
- Supports both well-conditioned and low-rank data (via `effective_rank`).

Important Parameters:

- `n_samples`: Number of samples.
- `n_features`: Total features.
- `n_informative`: Features used to generate the output.
- `noise`: The standard deviation of the gaussian noise applied to the output.

Example Usage:

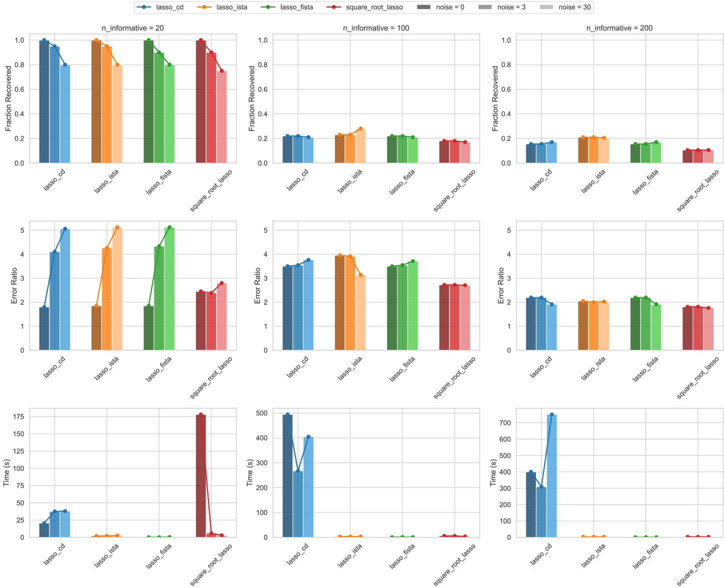
```python
from sklearn.datasets import make_regression

# Generate data with 100 samples, 100 features, 10
    informative, and noise level 10
X, y = make_regression(n_samples=100, n_features=100,
                       n_informative=10, noise=10,
                       random_state=42)
```

Scenario 1: Baseline (make_regression)

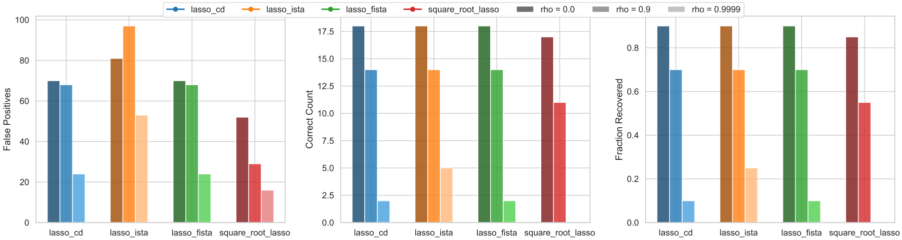
- **Data Generation:**
  - Use sklearn.datasets.make_regression with:
    - $n = 100$ samples, $p = 1000$ features (example),
    - n_informative values in $\{20, 100, 200\}$,
    - noise values in $\{0, 3, 30\}$,
    - Sparse coefficients.
  - Standardize X and center y.
- **Expectations:**
  - Lasso can achieve near-oracle performance if the design is well-behaved (i.e., no extreme correlations) and the true β is sparse.
  - make_regression by default produces a "nice" design, typically satisfying the restricted eigenvalue condition.
  - As noise increases, Lasso's support recovery (identifying nonzero coefficients) becomes more challenging.
  - Error bounds include a penalty factor that grows with the noise variance.

Scenario 1: Results



Scenario 2: Correlated Design (Toeplitz)

- **Data Generation:**
  - Covariance Σ with $\Sigma_{ij} = \rho^{|i-j|}$.
  - Sample $X_i \sim \mathcal{N}(0, \Sigma)$, $i = 1, \ldots, n$.
  - True β sparse, $y = X\beta + \epsilon$.
- **Vary** $\rho \in \{0.0, 0.9, 0.9999\}$ to see effect of correlation.
- **Expectations:**
  - Lasso can achieve near-oracle performance if the design is well-behaved (i.e., no extreme correlations) and the true β is sparse.
  - When correlation increases the RE (or RIP) assumptions can fail, which often hurts Lasso's ability to correctly recover the sparse support.
  - We expect to see more false positives because different columns in the same "correlated cluster" can all partially explain the same signal.
  - Convergence might slow down for iterative methods if the design is "ill-conditioned." and we expect to see longer running times.



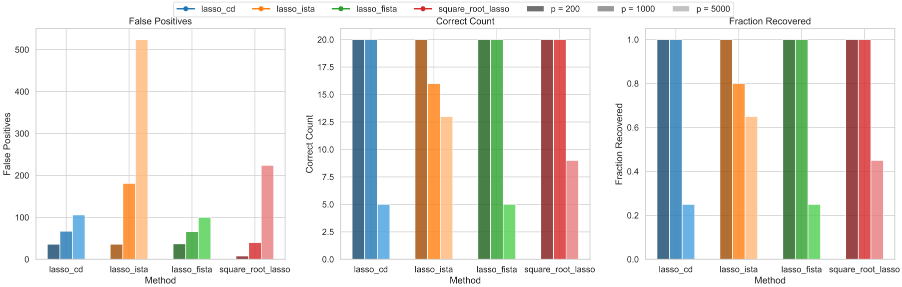
Parameters: $n = 100,\ p = 1000,\ \rho \in \{0.0,\ 0.9,\ 0.9999\},\ n_{\mathtt{informative}} = 20,\ \sigma_{\mathtt{noise}} = 1.0,\ \lambda = 0.1,\ \tau = 0.2,$

$\mathtt{random\_state}{=}42.$

Scenario 3: Scaling Up p

- **Setup:**
  - Fix $n = 100$, let $p \in \{200, 1000, 5000\}$ or more.
  - Keep `n_informative` $= 20$ and moderate noise.
  - Evaluate each method's runtime, fraction recovered, etc.
- **Expectations:**
  - As p grows, naive coordinate descent might slow.
  - $\log(p)$ factor in theory: can see more false positives if λ not tuned carefully.
  - It could be harder for lasso to distinguish the true signal source if some of the features are correlated by chance.

Scenario 3: Results



Parameters: $p_{\textbf{list}} = [200, 1000, 5000]$, $n = 100$, $n_{\text{informative}} = 20$, $\sigma_{\text{noise}} = 1.0$, $\lambda = 0.1$, $\tau = 0.2$, `random_state=42`.

Conclusion of Experiments

- LASSO solutions strongly depend on design matrix properties:
  - Correlation, rank deficiencies, high p all degrade success of feature recovery.
- FISTA often converges faster than ISTA; coordinate descent can be effective in low correlation.
- Square-root Lasso can help handle unknown noise level automatically.

Adaboost

- AdaBoost: adaptive boosting for feature selection and regression.
- 2 versions of AdaBoost for regression:
  - Scikit-learn AdaBoostRegressor.
  - Custom MyAdaBoostRegressor implementation.
- Experiments on synthetic data follow.

MyAdaBoost: Manual Implementation

```python
def fit(self, X, y):
    n_samples = X.shape[0]
    sample_weights = np.ones(n_samples) / n_samples

    for m in range(self.n_estimators):
        estimator = deepcopy(DecisionTreeRegressor(max_depth=
                self.max_depth)) # Stronger weak learners
        estimator.fit(X, y, sample_weight=sample_weights)
        y_pred = estimator.predict(X)

        # Compute absolute errors and median threshold
        errors = np.abs(y - y_pred)
        tau = np.median(errors)
        L = (errors > tau).astype(int) # Binary loss

        # Compute weighted error
        err_m = np.sum(sample_weights * L) / np.sum(
                sample_weights)
        err_m = np.clip(err_m, 1e-10, 0.4999) # Prevent
                division instability

        # Compute model weight (alpha)
        alpha_m = 0.5 * np.log((1 - err_m) / err_m) # Scaling
                alpha
        beta_m = err_m / (1 - err_m) # Compute beta_m

        # Update sample weights using beta_m^(1 - L)
        sample_weights *= beta_m ** (1 - L)
        sample_weights /= np.sum(sample_weights)
        self.estimators_.append(estimator)
        self.estimator_weights_.append(alpha_m)

def predict(self, X):
    predictions = np.array([
        alpha * est.predict(X) for alpha, est in zip(self.
                estimator_weights_, self.estimators_)
    ])
    return np.sum(predictions, axis=0) # Weighted sum
```

Key Points:

- Iteratively trains multiple DecisionTreeRegressors and assigns higher weights to harder-to-predict samples.
- Weak learners are weighted based on their error rate; better models get higher weights.
- The error err_m is computed as a weighted error rate.
- The model weight α_m is calculated as:

$$\alpha_m = \frac{1}{2}\log\left(\frac{1-\mathrm{err}_m}{\mathrm{err}_m}\right)$$

- The beta value (β_m) used for weight updates is:

$$\beta_m = \frac{\mathrm{err}_m}{1-\mathrm{err}_m}$$

- The weight update rule is:

$$w_i = w_i \times \beta_m^{(1-L)}$$

AdaBoost: Scikit-Learn Implementation

```python
from sklearn.ensemble import AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor

# Create AdaBoost with decision stumps
ada_reg = AdaBoostRegressor(
    estimator=DecisionTreeRegressor(max_depth=4),
    n_estimators=50,
    learning_rate=1.0,
    random_state=42
)

# Train the model
ada_reg.fit(X_train, y_train)

# Predictions
y_pred = ada_reg.predict(X_test)
```

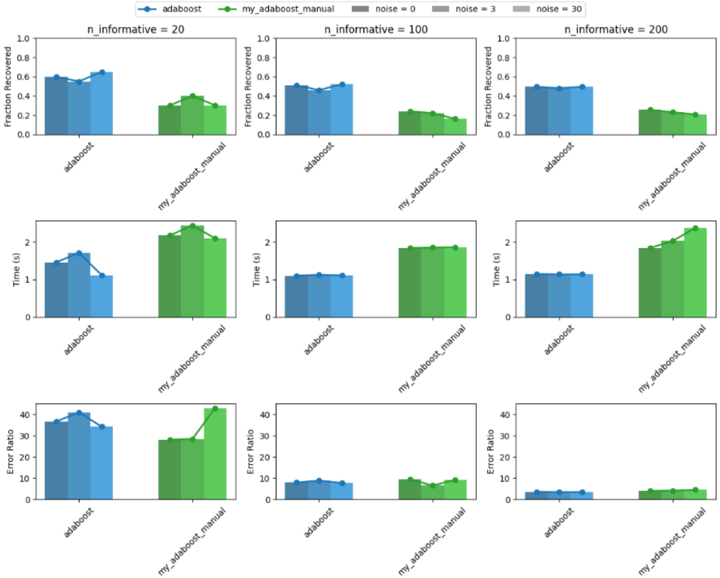
Key Points:

- The weak learner used is a `DecisionTreeRegressor` with `max_depth=4`, meaning each tree can have a maximum depth of 4.
- Unlike classification AdaBoost, the regression version uses weighted residuals to improve performance.
- The model is trained sequentially, where each new tree focuses on correcting the errors of the previous trees.
- `n_estimators=50`: Uses 50 weak learners (decision trees).
- `learning_rate=1.0`: Controls how much influence each tree has on the final prediction.

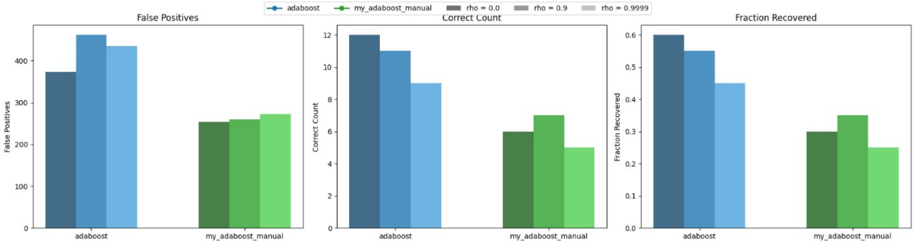
AdaBoost Baseline (make_regression)

- **Data Generation:**
  - Use sklearn.datasets.make_regression with:
    - $n = 100$ samples, $p = 1000$ features.
    - n_informative $= 20, 100, 200$.
    - noise values in $\{0, 3, 30\}$.
    - Sparse coefficients.
  - Standardize X and center y.
  - Split into training (80%) and test (20%).
- **Expectations:**
  - AdaBoost assigns more weight to samples that are harder to predict.
  - Feature importance estimation depends on boosting weight updates.
  - Higher noise levels make it difficult for weak learners to detect patterns.
  - Manual implementation should perform similarly to scikit-learn's, with minor differences due to weight updates.

Scenario 1 on AdaBoost



Scenario 2: Adaboost - Correlated Design (Toeplitz)



Comparison Table

Aspect	Lasso Regression	Adaboost Regression
Type	Linear model	Ensemble method
Regularization	L1 (sparsity)	No direct regularization
Strength	Feature selection	Captures complex non-linearity
Weakness	Struggles with non-linearity	Sensitive to outliers
Use Case	High-dimensional, sparse data	Complex relationships, boosting weak models

Table: Comparison of Lasso and Adaboost Regression

References I

Belloni, A., Chernozhukov, V., & Wang, L. (2011).
"Square-root Lasso: Pivotal recovery of sparse signals via conic programming."
Biometrika, 98(4), 791–806.
(**Square-root Lasso**)

Beck, A., & Teboulle, M. (2009).
"A fast iterative shrinkage-thresholding algorithm for linear inverse problems."
SIAM Journal on Imaging Sciences, 2(1), 183–202.
(**FISTA Algorithm Paper**)

Friedman, J., Hastie, T., & Tibshirani, R. (2010).
"Regularization Paths for Generalized Linear Models via Coordinate Descent."
Journal of Statistical Software, 33(1), 1–22.
(**Coordinate Descent for Lasso**)

Christophe Giraud. *Introduction to High-Dimensional Statistics.* CRC Press, 2021, Chapter 11. https://www.imo.universite-paris-saclay.fr/ christophe.giraud/Orsay/Bookv3.pdf

Our Implementations:
Coordinate Descent, ISTA, FISTA, and Square-root Lasso in Python:
`https://github.com/aabdelsameia1/HighDimensionalProject`