Anne Louise Seekford
Advanced Applied Machine Learning

Comparison of Different Regularization and Variable Selection Techniques

In this project, I have applied and compared several different regularization techniques including Ridge, LASSO, Elastic Net, SCAD, and Square Root Lasso

1. Created my own sklearn compliant functions for Square Root Lasso and SCAD to use them in conjunction with GridSearchCV for finding optimal hyper-parameters when data such as xx and yy are given.

SCAD

```
[ ] # Define functions

    @njit
    def scad_penalty(beta_hat, lambda_val, a_val):
        is_linear = (np.abs(beta_hat) <= lambda_val)
        is_quadratic = np.logical_and(lambda_val < np.abs(beta_hat), np.abs(beta_hat) <= a_val * lambda_val)
        is_constant = (a_val * lambda_val) < np.abs(beta_hat)

        linear_part = lambda_val * np.abs(beta_hat) * is_linear
        quadratic_part = (2 * a_val * lambda_val * np.abs(beta_hat) - beta_hat**2 - lambda_val**2) / (2 * (a_val - 1)) * is_quadratic
        constant_part = (lambda_val**2 * (a_val + 1)) / 2 * is_constant
        return linear_part + quadratic_part + constant_part

    @njit
    def scad_derivative(beta_hat, lambda_val, a_val):
        return lambda_val * ((beta_hat <= lambda_val) + (a_val * lambda_val - beta_hat)*((a_val * lambda_val - beta_hat) > 0) / ((a_val - 1) * lambda_val) * (beta_hat > lambda_val))
```

```
[ ] @njit
    def scad(beta):
      beta = beta.flatten()
      beta = beta.reshape(-1,1)
      n = len(y)
      return 1/n*np.sum((y-x.dot(beta))**2) + np.sum(scad_penalty(beta,lam,a))

    @njit
    def dscad(beta):
      beta = beta.flatten()
      beta = beta.reshape(-1,1)
      n = len(y)
      output = -2/n*np.transpose(x).dot(y-x.dot(beta))+scad_derivative(beta,lam,a)
      return output.flatten()
```

```
[ ] # SKLearn Compliant SCAD Function

    class SCAD(BaseEstimator, RegressorMixin):
        def __init__(self, a=2,lam=1):
            self.a, self.lam = a, lam

        def fit(self, x, y):
            a = self.a
            lam   = self.lam

            @njit
            def scad(beta):
              beta = beta.flatten()
              beta = beta.reshape(-1,1)
              n = len(y)
              return 1/n*np.sum((y-x.dot(beta))**2) + np.sum(scad_penalty(beta,lam,a))

            @njit
            def dscad(beta):
              beta = beta.flatten()
              beta = beta.reshape(-1,1)
              n = len(y)
              output = -2/n*np.transpose(x).dot(y-x.dot(beta))+scad_derivative(beta,lam,a)
              return output.flatten()


            beta0 = np.zeros(p)
            output = minimize(scad, beta0, method='L-BFGS-B', jac=dscad,options={'gtol': 1e-8, 'maxiter': 50000,'maxls': 50,'disp': False})
            beta = output.x
            self.coef_ = beta

        def predict(self, x):
            return x.dot(self.coef_)
```

Square Root Lasso

```
[ ] # SKLearn Compliant SQRTLasso Function

    class SQRTLasso(BaseEstimator, RegressorMixin):
        def __init__(self, alpha=0.01):
            self.alpha = alpha

        def fit(self, x, y):
            alpha=self.alpha
            @njit
            def f_obj(x,y,beta,alpha):
              n =len(x)
              beta = beta.flatten()
              beta = beta.reshape(-1,1)
              output = np.sqrt(1/n*np.sum((y-x.dot(beta))**2)) + alpha*np.sum(np.abs(beta))
              return output
            @njit
            def f_grad(x,y,beta,alpha):
              n=x.shape[0]
              p=x.shape[1]
              beta = beta.flatten()
              beta = beta.reshape(-1,1)
              output = (-1/np.sqrt(n))*np.transpose(x).dot(y-x.dot(beta))/np.sqrt(np.sum((y-x.dot(beta))**2))+alpha*np.sign(beta)
              return output.flatten()

            def objective(beta):
              return(f_obj(x,y,beta,alpha))

            def gradient(beta):
              return(f_grad(x,y,beta,alpha))

            beta0 = np.ones((x.shape[1],1))
            output = minimize(objective, beta0, method='L-BFGS-B', jac=gradient,options={'gtol': 1e-8, 'maxiter': 50000,'maxls': 25,'disp': True})
            beta = output.x
            self.coef_ = beta

        def predict(self, x):
            return x.dot(self.coef_)
```

Anne Louise Seekford
Advanced Applied Machine Learning

2. Simulated 100 data sets, each with 1,200 features, 200 observations, and a toeplitz correlation structure such that the correlation between features i and j is approximately $p^{|i-j|}$ with p=0.8. For the dependent variable y consider the following functional relationship: $y = x\beta + \sigma\epsilon$, where σ=3.5, ϵ is a column vector with $\epsilon_i \in N(0, 1)$.

$$\beta^* = (\underbrace{1, 1, ..., 1}_{7\text{ times}}, \underbrace{0, 0..., 0}_{25\text{ times}}, \underbrace{0.25, 0.25..., 0.25}_{5\text{ times}}, \underbrace{0, 0..., 0}_{50\text{ times}}, \underbrace{0.7, 0.7..., 0.7}_{15\text{ times}}, \underbrace{0, 0..., 0}_{1098\text{ times}})^T$$

```
[ ] n = 200 #num observations
    p = 1200 #num features
```

```
[ ] beta_star = np.concatenate(([1]*7, [0]*25, [0.25]*5, [0]*50, [0.7]*15, [0]*1098))
```

```
[ ] # What we want to detect is the position of the actual information or "signal"

    pos = np.where(beta_star != 0)
```

```
[ ] pos # shows the INDICIES where the value is not equal to zero

    # 27 important conditions

    (array([  0,   1,   2,   3,   4,   5,   6,  32,  33,  34,  35,  36,  87,
             88,  89,  90,  91,  92,  93,  94,  95,  96,  97,  98,  99, 100,
            101]),)
```

```
[ ] len(pos[0])

    27
```

```
[ ] # What we need: toeplitz([1, 0.8, 0.8**2, 0.8**3, 0.8**4, ... , 0.8**1119])

    v = []
    for i in range(p):
      v.append(0.8**i)
```

```
[ ] v
```

```
[ ] # Create covariance matrix (to use later when generating x)

    mu = [0]*p #repeat 0, 1200 times
    r = toeplitz(v) #covariance
    sigma = 3.5 # per project instructions

    # Generate the random samples
    np.random.seed(123)
    x = np.random.multivariate_normal(mu, r, size=n) #this is where we generate fictious data
    y = np.matmul(x,beta_star) + sigma*np.random.normal(0,1,size=n)
```

```
[ ] y.shape #should be (200,1)

    (200,)
```

Anne Louise Seekford
Advanced Applied Machine Learning

3. Applied variable selection methods (such as *Ridge*, *Lasso*, *Elastic Net*, *SCAD* and *Square Root Lasso* with GridSearchCV (for tuning the hyper-parameters)) and recorded the final results, including the overall (on average) quality of reconstructing the sparsity pattern and the coefficients of $\beta *$. The final results include the average number of true non-zero coefficients discovered by each method, the L2 distance to the ideal solution and the Root Mean Squared Error.

Ridge

- `RIDGE mean square error is:  34.84857904255565`
- Number of non-zero values: 1200
- Compared with "ground truth": 27
- Number of true non-zero coefficients: 27
  - `array([  0,   1,   2,   3,   4,   5,   6,  32,  33,  34,  35, 36,  87, 88,  89,  90,  91,  92,  93,  94,  95,  96,  97,  98, 99, 100, 101])`

Lasso

- `LASSO mean square error is:  15.23773830528663`
- Number of non-zero values: 144
- Compared with "ground truth": 27
- Number of true non-zero coefficients: 21
  - `array([  0,   1,   2,   3,   4,   5,   6,  32,  35,  87,  88, 89,  90, 92,  94,  95,  96,  97,  99, 100, 101])`

Elastic Net

- `Elastic Net mean square error is:  14.823625894539939`
- Number of non-zero values: 238
- Compared with "ground truth": 27
- Number of true non-zero coefficients: 25
  - `array([  0,   1,   2,   3,   4,   5,   6,  32,  34,  35,  87, 88,  89, 90,  91,  92,  93,  94,  95,  96,  97,  98,  99, 100, 101])`

SCAD

- `SCAD mean square error is:`
- Number of non-zero values: 1200
- Compared with "ground truth": 27
- Number of true non-zero coefficients:

Square Root Lasso

- `Square Root Lasso mean square error is:`
- Number of non-zero values: 238

Anne Louise Seekford
Advanced Applied Machine Learning

- Compared with “ground truth”: 27
- Number of true non-zero coefficients: 25
    - `array([  0,   1,   2,   3,   4,   5,   6,  32,  34,  35,  87,  88,  89, 90,  91,  92,  93,  94,  95,  96,  97,  98,  99, 100, 101])`