Elastic Net Regularization Paths for All Generalized Linear Models

J. Kenneth Tay,

Department of Statistics, Stanford University, 390 Jane Stanford Way, Stanford, California 94305, United States of America

Balasubramanian Narasimhan,

Department of Biomedical Data Sciences, and Department of Statistics, Stanford University, 390 Jane Stanford Way, Stanford, CA 94305

Trevor Hastie

Department of Biomedical Data Sciences, and Department of Statistics, Stanford University, 390 Jane Stanford Way, Stanford, CA 94305

Abstract

The lasso and elastic net are popular regularized regression models for supervised learning. Friedman, Hastie, and Tibshirani (2010) introduced a computationally efficient algorithm for computing the elastic net regularization path for ordinary least squares regression, logistic regression and multinomial logistic regression, while Simon, Friedman, Hastie, and Tibshirani (2011) extended this work to Cox models for right-censored data. We further extend the reach of the elastic net-regularized regression to all generalized linear model families, Cox models with (start, stop] data and strata, and a simplified version of the relaxed lasso. We also discuss convenient utility functions for measuring the performance of these fitted models.

Keywords

lasso; elastic net; ℓ_1 penalty; regularization path; coordinate descent; generalized linear models; survival; Cox model

1. Introduction

Consider the standard supervised learning framework. We have data of the form (x_1, y_1), $\ldots$, (x_n, y_n), where $y_i \in \mathbb{R}$ is the target and $x_i = (x_{i,1}, \ldots, x_{i,p})^T \in \mathbb{R}^p$ is a vector of potential predictors. The ordinary least squares (OLS) model assumes that the response can be modeled as a linear combination of the covariates, i.e. $y_i = \beta_0 + x_i^T \beta$ for some coefficient vector $\beta \in \mathbb{R}^p$ and intercept $\beta_0 \in \mathbb{R}$. The parameters are estimated by minimizing the residual sum of squares (RSS):

J. Kenneth Tay, Department of Statistics, Stanford University, 390 Jane Stanford Way, Stanford, California 94305, United States of America, kjytay@stanford.edu, URL: https://kjytay.github.io/.

$$(\widehat{\beta}_0, \widehat{\beta}) = \underset{(\beta_0, \beta) \in \mathbb{R}^{p+1}}{\mathrm{argmin}} \frac{1}{2n} \sum_{i=1}^{n} (y_i - \beta_0 - x_i^T \beta)^2 . \tag{1}$$

There has been a lot of research on regularization methods in the last two decades. We focus on the elastic net (Zou and Hastie 2005) which minimizes the sum of the RSS and a regularization term which is a mixture of ℓ_1 and ℓ_2 penalties:

$$(\widehat{\beta}_0, \widehat{\beta}) = \underset{(\beta_0, \beta) \in \mathbb{R}^{p+1}}{\mathrm{argmin}} \left[\frac{1}{2n} \sum_{i=1}^{n} (y_i - \beta_0 - x_i^T \beta)^2 + \lambda \left(\frac{1-\alpha}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 \right) \right]. \tag{2}$$

In the above, $\lambda \geq 0$ is a tuning parameter and $\alpha \in [0, 1]$ is a higher level hyperparameter[1].

We always fit a path of models in λ, but set a value of α depending on the type of prediction model we want. For example, if we want ridge regression (Hoerl and Kennard 1970) we set $\alpha = 0$ and if we want the lasso (Tibshirani 1996) we set $\alpha = 1$. If we want a sparse model but are worried about correlations between features, we might set α close to but not equal to 1. The final value of λ is usually chosen via cross-validation: we select the coefficients corresponding to the λ value giving smallest cross-validated error as the final model.

The elastic net can be extended easily to generalized linear models (GLMs) (Nelder and Wedderburn 1972) and Cox proportional hazards models (Cox 1972). Instead of solving the minimization problem (2), the RSS term in the objective function is replaced with a negative log-likelihood term or a negative log partial likelihood term respectively.

The **glmnet** R package (Friedman et al. 2010) contains efficient functions for computing the elastic net solution for an entire path of values $\lambda_1 > \cdots > \lambda_m$. The minimization problems are solved via cyclic coordinate descent (van der Kooij 2007), with the core routines programmed in FORTRAN for computational efficiency. Earlier versions of the package contained specialized FORTRAN subroutines for a handful of popular GLMs and the Cox model for right-censored survival data. The package includes functions for performing K-fold cross-validation (CV), plotting coefficient paths and CV errors, and predicting on future data. The package can also accept the predictor matrix in sparse matrix format: this is especially useful in certain applications where the predictor matrix is both large and sparse. In particular, this means that we can fit unpenalized GLMs with sparse predictor matrices, something the glm function in the **stats** package cannot do.

From version 4.1 and later, **glmnet** is able to compute the elastic net regularization path for all GLMs, Cox models with (start, stop] data and strata, and a simplified version of the relaxed lasso. (Hastie, Tibshirani, and Tibshirani 2020).

---

[1]If the square were removed from the ℓ_2-norm penalty, it would be more natural to have $1 - \alpha$ instead of $(1 - \alpha)/2$ as its mixing parameter. The factor of 1/2 compensates for the fact that a squared ℓ_2-norm penalty is used, in the sense that the gradient of the penalty with respect to β can be seen as a convex combination of the ℓ_1 and ℓ_2 penalty terms. We note also that there is a one-to-one correspondence between these two parameterizations for the penalty.

Friedman et al. (2010) gives details on how the **glmnet** package computes the elastic net solution for ordinary least squares regression, logistic regression and multinomial logistic regression, while Simon et al. (2011) explains how the package fits regularized Cox models for right-censored data. This paper builds on these two earlier works. In Section 2, we explain how the elastic net penalty can be applied to all GLMs and how we implement it in software. In Section 3, we detail extensions to Cox models with (start, stop] data and strata. In Section 4, we describe an implementation of the relaxed lasso implemented in the package, and in Section 5 we describe the package's functionality for assessing fitted models. We conclude with a summary and discussion.

2.   Regularized generalized linear models

2.1.   Overview of generalized linear models

Generalized linear models (GLMs) (Nelder and Wedderburn 1972) are a simple but powerful extension of OLS. A GLM consists of 3 parts:

- A linear predictor: $\eta_i = x_i^T \beta$,

- A link function: $\eta_i = g(\mu_i)$, and

- A variance function as a function of the mean: $V = V(\mu_i)$.

The user gets to specify the link function g and the variance function V. For one-dimensional exponential families, the family determines the variance function, which, along with the link, are sufficient to specify a GLM. More generally, modeling can proceed once the link and variance functions are specified via a quasi-likelihood approach (see McCullagh and Nelder (1983) for details); this is the approach taken by the quasi-binomial and quasi-Poisson models. The OLS model is a special case, with link $g(x) = x$ and constant variance function $V(\mu) = \sigma^2$ for some constant σ^2. More examples of GLMs are listed in Table 1.

The GLM parameter β is determined by maximum likelihood estimation. Unlike OLS, there is no closed form solution for $\hat{\beta}$. Rather, it is typically computed via an iteratively reweighted least squares (IRLS) algorithm known as *Fisher scoring.* In each iteration of the algorithm we make a quadratic approximation to the negative log-likelihood (NLL), reducing the minimization problem to a weighted least squares (WLS) problem. For GLMs with canonical link functions, the negative log-likelihood is convex in β, Fisher scoring is equivalent to the Newton-Raphson method and is guaranteed to converge to a global minimum. For GLMs with non-canonical links, the negative log-likelihood is not guaranteed to be convex[2]. Also, Fisher scoring is no longer equivalent to the Newton-Raphson method and is only guaranteed to converge to a local minimum.

It is easy to fit GLMs in R using the `glm` function from the **stats** package; the user can specify the GLM to be fit using `family` objects. These objects capture details of the GLM

---

[2]It is not true that the negative log-likelihood is always non-convex for non-canonical links. For example, it can be shown via direct computation that the negative log-likelihood for probit regression is convex in β.

such as the link function and the variance function. For example, the code below shows the `family` object associated with probit regression model:

```
R> class(binomial(link = "probit"))
```

```
[1] "family"
```

```
R> str(binomial(link = "probit"))
List of 12
 $ family : chr "binomial"
 $ link : chr "probit"
 $ linkfun : function (mu)
 $ linkinv : function (eta)
 $ variance : function (mu)
 $ dev.resids: function (y, mu, wt)
 $ aic : function (y, n, mu, wt, dev)
 $ mu.eta : function (eta)
 $ initialize: language … # code to set up objects needed for the family
 $ validmu : function (mu)
 $ valideta : function (eta)
 $ simulate : function (object, nsim)
 - attr(*, "class")= chr "family"
```

The `linkfun`, `linkinv`, `variance` and `mu.eta` functions are used in fitting the GLM, and the `dev.resids` function is used in computing the deviance of the resulting model. By passing a class `"family"` object to the `family` argument of a `glm` call, `glm` has all the information it needs to fit the model. Here is an example of how one can fit a probit regression model in R:

```
R> library(glmnet)
R> data(BinomialExample)
R> glm(y ~ x, family = binomial(link = "probit"))
```

2.2.   Extending the elastic net to all GLM families

To extend the elastic net to GLMs, we replace the RSS term in (2) with a negative log-likelihood term:

$$(\hat{\beta}_0, \hat{\beta}) = \underset{(\beta_0, \beta) \in \mathbb{R}^{p+1}}{\operatorname{argmin}} \left[-\frac{1}{n} \sum_{i=1}^{n} \ell(y_i, \beta_0 + x_i^T \beta) + \lambda \left(\frac{1-\alpha}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 \right) \right], \quad (3)$$

where $\ell(y_i, \beta_0 + x_j^T \beta)$ is the log-likelihood term associated with observation i. We can apply the same strategy as for GLMs to minimize this objective function. The key difference is that instead of solving a WLS problem in each iteration, we solve a penalized WLS problem.

The algorithm for solving (3) for a path of λ values is described in Algorithm 1. Note that in Step 2(a), we initialize the solution for $\lambda = \lambda_k$ at the solution obtained for $\lambda = \lambda_{k-1}$. This is known as a *warm start*: since we expect the solution at these two λ values to be similar, the algorithm will likely require fewer iterations than if we initialized the solution at zero.

2.3. Implementation details

There are two main approaches we can take in implementing Algorithm 1. In the original implementation of **glmnet**, the entire algorithm was implemented in FORTRAN for specific GLM families. In version 4.0 and later, we added a second implementation which implemented just the computational bottleneck, the penalized WLS problem in Step 2(b)iii, in FORTRAN, with the rest of the algorithm implemented in R. Here are the relative merits and disadvantages of the second approach compared to the first:

Algorithm 1 *Fitting GLMs with elastic net penalty*

---

1 . Select a value of $\alpha \in [0, 1]$ and a sequence of λ values $\lambda_1 > \dots > \lambda_m$.

2 . For $k = 1, \dots, m$:

    (a) Initialize $(\hat{\beta}_0^{(0)}(\lambda_k), \hat{\beta}^{(0)}(\lambda_k)) = (\hat{\beta}_0(\lambda_{k-1}), \hat{\beta}(\lambda_{k-1}))$.   For $k = 1$, initialize $(\hat{\beta}_0^{(0)}(\lambda_k), \hat{\beta}^{(0)}(\lambda_k)) = (0, \mathbf{0})$. (Here, $(\hat{\beta}_0(\lambda_k), \hat{\beta}(\lambda_k))$ denotes the elastic net solution at $\lambda = \lambda_k$.)

    (b) For $t = 0, 1, \dots$ until convergence:

        i . For $i = 1, \dots, n$, compute $\eta_i^{(t)} = \hat{\beta}_0^{(t)}(\lambda_k) + \hat{\beta}^{(t)}(\lambda_k)^T x_i$ and $\mu_i^{(t)} = g^{-1}(\eta_i^{(t)})$.

        ii . For $i = 1, \dots, n$, compute working responses and weights

$$z_i^{(t)} = \eta_i^{(t)} + (y_i - \mu_i^{(t)}) / \frac{d\mu_i^{(t)}}{d\eta_i^{(t)}}, \quad w_i^{(t)} = \left(\frac{d\mu_i^{(t)}}{d\eta_i^{(t)}}\right)^2 / V(\mu_i^{(t)}). \qquad (4)$$

        iii. Solve the penalized WLS problem

$$(\hat{\beta}_0^{(t+1)}(\lambda_k), \hat{\beta}^{(t+1)}(\lambda_k))$$

$$= \underset{(\beta_0, \beta) \in \mathbb{R}^{p+1}}{\operatorname{argmin}} \left[\frac{1}{2n} \sum_{i=1}^n w_i^{(t)}(z_i^{(t)} - \beta_0 - x_i^T \beta)^2 + \lambda_k \left(\frac{1-\alpha}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 \right) \right].$$

$$(5)$$

- Because the formulas for the working weights and responses in (4) are specific to each GLM, the first approach requires a new FORTRAN subroutine for each GLM family. This is tedious to manage, and also means that users cannot fit regularized models for their bespoke GLM families. The second approach allows the user to pass a class "family" object to glmnet: the working weights and responses can then be computed in R before the FORTRAN subroutine solves the resulting penalized WLS problem.

- As written, Algorithm 1 is a proximal Newton algorithm with a constant step size of 1, and hence it may not converge in certain cases. To ensure convergence, we can implement step-size halving after Step 2(b)iii: as long as the objective function (3) is not decreasing, set $\hat{\beta}^{(t+1)}(\lambda_k) \leftarrow \hat{\beta}^{(t)}(\lambda_k) + \frac{1}{2}\left[\hat{\beta}^{(t+1)}(\lambda_k) - \hat{\beta}^{(t)}(\lambda_k)\right]$ (with a similar formula for the intercept). Since the objective function involves a log-likelihood term, the formula for the objective function differs across GLMs, and the first approach has to maintain different subroutines for step-size halving. For the second approach, we can write a single function that takes in the class `"family"` object (along with other necessary parameters) and returns the objective function value.

- It is computationally less efficient than the first approach because (i) R is generally slower than FORTRAN, and (ii) there is overhead associated with constant switching between R and FORTRAN. Some timing comparisons for Gaussian and logistic regression with the default parameters are presented in Figure 1. The second approach is 10 to 15 times as slow than the first approach.

- Since each GLM family has its own set of FORTRAN subroutines in the first approach, it allows for special computational tricks to be employed in each situation. For example, with `family = "gaussian"`, the predictors can be centered once upfront to have zero mean and Algorithm 1 can be run ignoring the intercept term.

We stress that both approaches have been implemented in **glmnet**. Users should use the first implementation for the most popular GLM families including OLS (Gaussian regression), logistic regression and Poisson regression (see `glmnet`'s documentation for the full list of such families), and use the second implementation for all other GLM families. For example, the code below shows two equivalent ways to fit a regularized Poisson regression model:

```
R> data(PoissonExample)
R> glmnet(x, y, family = "poisson")
R> glmnet(x, y, family = poisson())
```

The first call specifies the GLM family as a character string to the `family` argument, invoking the first implementation. The second call passes a class `"family"` object to the `family` argument instead of a character string, invoking the second implementation. One would never run the second call in practice though, as it returns the same result as the first call but takes longer to fit. The example below fits a regularized quasi-Poisson model that allows for overdispersion, a family that is only available via the second approach:

```
R> glmnet(x, y, family = quasipoisson())
```

2.4. Details on the penalized WLS subroutine

Since the penalized WLS problem in Step 2(b)iii of Algorithm 1 is the computational bottleneck, we elected to implement it in FORTRAN. Concretely, the subroutine solves the problem

$$\underset{(\beta_0, \beta) \in \mathbb{R}^{p+1}}{\text{minimize}} \quad \frac{1}{2n} \sum_{i=1}^{n} w_i\big(z_i - \beta_0 - x_i^T \beta\big)^2 + \lambda_k \sum_{j=1}^{p} \gamma_j \Big(\frac{1-\alpha}{2}\beta_j^2 + \alpha \mid \beta_j \mid \Big) \tag{6}$$

$$\text{subject to} \quad L_j \le \beta_j \le U_j, \quad j = 1, \ldots, p. \tag{7}$$

This is the same problem as (5) except for two things. First, the penalty placed on each coefficient β_j has its own multiplicative factor γ_j. ((7) reduces to (5) if $\gamma_j = 1$ for all j, which is the default value for the glmnet function.) This allows the user to place different penalty weights on the coefficients. An instance where this is especially useful is when the user always wants to include feature j in the model: in that case the user could set $\gamma_j = 0$ so that β_j is unpenalized. Second, the coefficient β_j is constrained to lie in the interval $[L_j, U_j]$. (glmnet's default is $L_j = -\infty$ and $U_j = \infty$ for all j, i.e. no constraints on the coefficients.) One example where these constraints are useful is when we want a certain β_j to always be non-negative or always non-positive.

The FORTRAN subroutine solves (7) by cyclic coordinate descent: see Friedman et al. (2010) for details. Here we describe one major computational trick that was not covered in that paper: the application of *strong rules* (Tibshirani, Bien, Friedman, Hastie, Simon, Taylor, and Tibshirani 2012).

In each iteration of cyclic coordinate descent, the solver has to loop through all p features to update the corresponding model coefficients. This can be time-consuming if p is large, and is potentially wasteful if the solution is sparse: most of the β_j would remain at zero. If we know a priori which predictors will be "active" at the solution (i.e. have $\beta_j \ne 0$), we could perform cyclic coordinate descent on just those coefficients and leave the others untouched. The set of "active" predictors is known as the *active set.* Strong rules are a simple yet powerful heuristic for guessing what the active set is, and can be combined with the Karush-Kuhn-Tucker (KKT) conditions to ensure that we get the exact solution. (The set of predictors determined by the strong rules is known as the *strong set.*) We describe the use of strong rules in solving (7) fully in Algorithm 2.

Algorithm 2 *Solving penalized WLS (7) with strong rules*

Assume that we are trying to solve for $\hat{\beta}(\lambda_k)$ for some $k = 1, \ldots, m$, and that we have already computed $\hat{\beta}(\lambda_{k-1})$. (If $k = 1$, set $\hat{\beta}(\lambda_{k-1}) = \mathbf{0}$.)

1. Initialize the strong set $S_{\lambda_k} = \{j : \hat{\beta}(\lambda_{k-1})_j \neq 0\}$.
2. Check the strong rules: for $j = 1, \ldots, p$, include j in S_{λ_k} if
$$\left| x_j^T \left\{ y - X\hat{\beta}(\lambda_{k-1}) \right\} \right| > \alpha \left[\lambda_k - (\lambda_{k-1} - \lambda_k) \right] \gamma_j.$$
3. Perform cyclic coordinate descent only for features in S_{λ_k}.
4. Check that the KKT conditions hold for each $j = 1, \ldots, p$. if the conditions hold for all j, we have the exact solution. If the conditions do not hold for some features, include them in the strong set S_{λ_k} and go back to Step 3.

Finally, we note that in some applications, the design matrix X is sparse. In these settings, computational savings can be reaped by representing X in a sparse matrix format and performing matrix manipulations with this form. To leverage this property of the data, we have a separate FORTRAN subroutine that solves (7) when X is in sparse matrix format.

2.5. Other useful functionality

In this section, we mention other use functionality that the **glmnet** package provides for fitting elastic net models.

For fixed α, glmnet solves (3) for a path of λ values. While the user has the option of specifying this path of values using the lambda option, it is recommended that the user let glmnet compute the sequence on its own. glmnet uses the arguments passed to it to determine the value of λ_{max}, defined to be the smallest value of λ such that the estimated coefficients would be all equal to zero[3]. The program then computes λ_{min} such that the ratio $\lambda_{min}/\lambda_{max}$ is equal to lambda.min.ratio (default 10^{-2} if the number of variables exceeds the number of observations, 10^{-4} otherwise). Model (3) is then fit for nlambda λ values (default 100) starting at λ_{max} and ending at λ_{min} which are equally spaced on the log scale.

In practice, it common to choose the value of λ via cross-validation (CV). The cv.glmnet function is a convenience function that runs CV for the λ tuning parameter. The returned object has class "cv.glmnet", which comes equipped with plot, coef and predict methods. The plot method produces a plot of CV error against λ (see Figure 2 for an example.) As mentioned earlier, we prefer to think of α as a higher level hyperparameter whose value depends on the type of prediction model we want. Nevertheless, the code below shows how the user can perform CV for α manually using a for loop. Care must be taken to ensure that the same CV folds are used across runs for the CV errors to be comparable.

---

[3]We note that when $\alpha = 0$, λ_{max} is infinite, i.e. all coefficients will always be non-zero for finite λ. To avoid such extreme values of λ_{max}, if $\alpha < 0.001$ we return the λ_{max} value for $\alpha = 0.001$.

```
R> alphas <- c(1, 0.8, 0.5, 0.2, 0)
R> fits <- list()
R> fits[[1]] <- cv.glmnet(x, y, keep = TRUE)
R> foldid <- fits[[1]]$foldid
R> for (i in 2:length(alphas)) {
+       fits[[i]] <- cv.glmnet(x, y, alpha = alphas[i], foldid = foldid)
+   }
```

The returned `cv.glmnet` object contains estimated standard errors for the model CV error at each λ value. (We note that the method for obtaining this estimates is crude and are generally too small due to correlations across CV folds.) By default, the `predict` method returns predictions for the model at the `"lambda.1se"` value, i.e. the value of λ that gives the most regularized model such that the CV error is within one standard error of the minimum. To get predictions at the λ value which gives the minimum CV error, the `s = "lambda.min"` argument is passed.

```
R> cfit <- cv.glmnet(x, y)
R> predict(cfit, x)
R> predict(cfit, x, s = "lambda.min")
```

In large data settings, it may take some time to fit the entire sequence of elastic net models. `glmnet` and `cv.glmnet` come equipped with a progress bar which can be displayed with the argument `trace.it = TRUE`. This gives the user a sense of how model fitting is progressing.

The **glmnet** package provides a convenience function `bigGlm` for fitting a single *unpenalized* GLM but allowing all the options of `glmnet`. In particular, the user can set upper and/or lower bounds on the coefficients, and can provide the x matrix in sparse matrix format: options that are not available for the `stats::glm` function.

```
R> data(BinomialExample)
R> fit <- bigGlm(x, y, family = "binomial", lower.limits = -1)
```

3. Regularized Cox proportional hazards models

We assume the usual survival-analysis framework. Instead of having $y_i \in \mathbb{R}$ as a response, we have instead $(y_i, \delta_i) \in \mathbb{R}_+ \times \{0, 1\}$. Here y_i is the observed time for observation i, and $\delta_i = 1$ if y_i is the failure time and $\delta_i = 0$ if it is the right-censoring time. The Cox proportional hazards model (Cox 1972) is a commonly used model for the relationship between the predictor variables and survival time. It assumes a semi-parametric form for the hazard function

$$h_i(t) = h(t)e^{x_i^T \beta},$$

where $h_i(t)$ is the hazard for observation i at time t, h is the baseline hazard for the entire population of observations, and $\beta \in \mathbb{R}^p$ is the vector of coefficients to be estimated. Let $t_1 < \cdots < t_m$ denote the unique failure times and let $j(i)$ denote the index of the observation failing at time t_i. (Assume for the moment that the y_i's are unique.) If $y_j \geq t_i$, we say that observation j is *at risk* at time t_i. Let R_i denote the *risk set* at time t_i. β is estimated by maximizing the partial likelihood

$$L(\beta) = \prod_{i=1}^{m} \frac{e^{x_{j(i)}^T \beta}}{\sum_{j \in R_i} e^{x_j^T \beta}} . \tag{8}$$

It is the conditional likelihood that the failure occurs for observation $j(i)$ given all the observations at risk. Maximizing the partial likelihood is equivalent to minimizing the negative log partial likelihood

$$-\ell(\beta) = \frac{2}{n} \sum_{i=1}^{m} \left[-x_{j(i)}^T \beta + \log \left(\sum_{j \in R_i} e^{x_j^T \beta} \right) \right] . \tag{9}$$

We put a negative sign in front of ℓ so that ℓ denotes the log partial likelihood, and the scale factor $2/n$ is included for convenience. Note also that the model does not have an intercept term β_0, as it cancels out in the partial likelihood. Simon et al. (2011) proposed an elasticnet regularization path version for the Cox model, as well as Algorithm 3 for solving the minimization problem.

Algorithm 3 *Fitting Cox models with elastic net penalty*

---

1. Select a value of $\alpha \in [0, 1]$ and a sequence of λ values $\lambda_1 > \dots > \lambda_m$. Define $\hat{\beta}(\lambda_0) = \mathbf{0}$.

2. For $\ell = 1, \dots, m$:

    (a) Initialize $\hat{\beta}(\lambda_\ell) = \hat{\beta}(\lambda_{\ell-1})$.

    (b) For $t = 0, 1, \dots$ until convergence (outer loop):

        i. For $k = 1, \dots, n$, compute $\eta_k^{(t)} = \hat{\beta}(\lambda_\ell)^T x_k$.

        ii. For $k = 1, \dots, n$, compute

$$\ell'\left(\eta^{(t)}\right)_k = \delta_k - e^{\eta_k^{(t)}} \sum_{i \in C_k} \left(\frac{1}{\sum_{j \in R_i} e^{\eta_j^{(t)}}} \right), \tag{10}$$

$$\ell''\left(\eta^{(t)}\right)_{k,k} = \sum_{i \in C_k} \left[\frac{e^{\eta_k^{(t)}} \sum_{j \in R_i} e^{\eta_j^{(t)}} - \left(e^{\eta_k^{(t)}}\right)^2}{\left(\sum_{j \in R_i} e^{\eta_j^{(t)}}\right)^2} \right], \tag{11}$$

$$w_k^{(t)} = -\ell''\left(\eta^{(t)}\right)_{k,k}, \tag{12}$$

$$z_k^{(t)} = \eta_k^{(t)} + \frac{\ell'\left(\eta^{(t)}\right)_k}{\ell''\left(\eta^{(t)}\right)_{k,k}}, \tag{13}$$

        where C_k is the set of failure times i such that $t_i < y_k$ (i.e. times for which observation k is still at risk.)

        iii. Solve the penalized WLS problem (inner loop):

$$\hat{\beta}(\lambda_\ell) = \underset{\beta \in \mathbb{R}^p}{\operatorname{argmin}} \left[\frac{1}{2} \sum_{k=1}^n w_k^{(t)} \left(z_k^{(t)} - x_k^T \beta\right)^2 + \lambda_\ell \left(\frac{1-\alpha}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 \right) \right].$$

Algorithm 3 has the same structure as Algorithm 1 except for different formulas for computing the working responses and weights. (We note that these formulas implicitly approximate the Hessian of the log partial likelihood by a diagonal matrix with the Hessian's diagonal entries.) This means that we can leverage the fast implementation of the penalized WLS problem in Section 2.4 for an efficient implementation of Algorithm 3. (As a small benefit, it also means that we can fit regularized Cox models when the design matrix X is sparse.) Such a model can be fit with **glmnet** by specifying `family = "cox"`. The response provided needs to be a `Surv` object from the **survival** package (Therneau 2020).

```
R> glmnet(x, y, family = "cox")
```

The computation of these w_k's and z_k's can be a computational bottleneck if not implemented carefully: since the C_k and R_i have $O(n)$ elements, a naive implementation takes $O(n^2)$ time. Simon et al. (2011) exploit the fact that, once the observations are sorted in order of the observed times y_i, the risk sets are nested ($R_{i+1} \subseteq R_i$ for all i) and the w_k's and z_k's can be computed in $O(n)$ time.

If our data contains tied observed times, `glmnet` uses the Breslow approximation of the partial likelihood for ties (Breslow 1972) and maximizes the elastic net-regularized version of this approximation instead. See Simon et al. (2011) for details.

3.1. Extending regularized Cox models to (start, stop] data

Instead of working with right-censored responses, the Cox model can be extended to work with responses which are a pair of times (called the "start time" and "stop time"), with the possibility of the stop time being censored. This is an instantiation of the counting process framework proposed by Andersen and Gill (1982), and the right-censored data set-up is a special case with the start times all being equal to zero.

As noted in Therneau and Grambsch (2000), (start, stop] responses greatly increase the flexibility of the Cox model, allowing for

- Time-dependent covariates,

- Time-dependent strata,

- Left truncation,

- Multiple time scales,

- Multiple events per subject,

- Independent increment, marginal, and conditional models for correlated data, and

- Various forms of case-cohort models.

From a data analysis viewpoint, this extension amounts to requiring just one more variable: the `time` variable is replaced by (`start`, `stop`] variables, with (`start`, `stop`] indicating the interval where the unit is at risk. The survival package provides the function `tmerge` to aid in the creation of such datasets.

For this more general setup, inference for β can proceed as before. The formulas for the partial likelihood and negative log partial likelihood (Equations (8) and (9)) remain the same; what changes is the definition of what it means for an observation to be at risk at time t_i. If we let $(y_{1j}, y_{2j}]$ denote the (start, stop] times for observation j, then observation j is at risk at time t_i if and only if $t_i \in (y_{1j}, y_{2j}]$. Similarly, the elastic net-regularized version of the Cox model for (start, stop] data can be fitted using Algorithm 3 with this new definition of what it means for an observation to be at risk at a failure time.

With (start, stop] data, it is no longer true that the risk sets are nested. For example, if $t_i < y_{1j} < t_{i+1} < y_{2j}$, then $j \in R_{i+1}$ but $j \notin R_i$. However, as Algorithm 4 shows, it is still possible to compute the working responses and weights in $O(n \log n)$ time. In fact, only the ordering of observations (Step 1) requires $O(n \log n)$ time: the rest of the algorithm requires just $O(n)$ time. Since the ordering of observations never changes, the results of Step 1 can be cached, meaning that only the first run of Algorithm 4 requires $O(n \log n)$ time, and future runs just need $O(n)$ time.

The differences between right-censored data and (start, stop] data for Cox models are hidden from the user, in that the function call for (start, stop] data is exactly the same as that for right-censored data. The difference is in the type of `Surv` object that is passed for the response `y`. `glmnet` checks for the `Surv` object type before routing to the correct internal subroutine.

3.2. Stratified Cox models

An extension of the Cox model is to allow for strata. These strata divide the units into disjoint groups, with each group having its own baseline hazard function but having the same values of β. Specifically, if the units are divided into K strata, then the stratified Cox model assumes that a unit in stratum k has the hazard function

$$h_i(t) = h_k(t)e^{x_i^T \beta},$$

where $h_k(t)$ is the shared baseline hazard for all units in stratum k. In several applications, allowing different subgroups to have different baseline hazards approximates reality more closely. For example, it might be reasonable to have different baseline hazards based on gender in clinical trials, or a separate baseline for each center in multi-center trials.

In this setting, the negative log partial likelihood is

$$\ell(\beta) = \sum_{k=1}^{K} \ell_k(\beta),$$

where $\ell_k(\beta)$ is exactly (9) but considering just the units in stratum k. Since the negative log partial likelihood decouples across strata (conditional on β), regularized versions of stratified Cox models can be fit using a slightly modified version of Algorithm 3.

To fit an unpenalized stratified Cox model, the **survival** package has a special `strata` function that allows users to specify the strata variable in formula syntax. Since `glmnet` does not work with formulas, we needed a different approach for specifying strata. To fit regularized stratified Cox models in **glmnet**, the user needs to add a `strata` attribute to the response `y`. `glmnet` checks for the presence of this attribute and if it is present, it fits a stratified Cox model. We note that the user cannot simply add the attribute manually because R drops attributes when subsetting vectors. Instead, the user should use the `stratifySurv` function to add the `strata` attribute. (`stratifySurv` creates an object of class `"stratifySurv"` that inherits from the class `"Surv"`, ensuring that **glmnet** can reassign the `strata` attribute correctly after any subsetting.) The code below shows an example of how to fit a regularized stratified Cox model with **glmnet**; there are a total of 20 observations, with the first 10 belonging to the first strata and the rest belonging to the second strata.

Algorithm 4 *Computing working responses and weights for* **Algorithm 3**

Input: $\eta_j = x_j^T \hat{\beta}$ (where $\hat{\beta}$ is the current estimate for β), $(y_{1j}, y_{2j}]$, δ_j for $j = 1, ..., n$. For simplicity, assume that the observations are ordered by ascending stop time, i.e. $y_{21} < \cdots < y_{2n}$. As before, let $t_1 < ..., t_m$ denote the failure times in increasing order.

1. Get the ordering for the observations according to start times. Let $start(j)$ denote the index for the observation with the jth earliest start time.

2. Compute the risk set sums $RSS_i = \sum_{j \in R_i} e^{\eta_j}$, $i = 1, ..., m$ using the following steps:

    (a) For $j = 1, ..., n$, set $RSS_j \leftarrow \sum_{\ell = j}^{n} e^{\eta_\ell}$.

    (b) Set $curr \leftarrow 0$, $i \leftarrow m$, $start\_idx \leftarrow n$.

    (c) While $i > 0$ and $start\_idx > 0$:

    i. If $y_{1start(start\_idx)} < t_i$, set $RSS_{j(i)} \leftarrow RSS_{j(i)} - curr$ and $i \leftarrow i - 1$.

    ii. If not, set $curr \leftarrow curr + e^{\eta_{start(start\_idx)}}$ and $start\_idx \leftarrow start\_idx - 1$.

    (d) Take just the elements of RSS corresponding to death times, i.e. set $RSS_i \leftarrow RSS_{j(i)}$.

3. Compute the partial sums $RSK_k = \sum_{i \in C_k} \frac{1}{RSS_i}$, $k = 1, ..., n$ using the following steps:

    (a) For $i = 1, ..., m$, set $RD_i \leftarrow \sum_{\ell = 1}^{i} \frac{1}{RSS_i}$. Set $RD_0 \leftarrow 0$.

    (b) For $k = 1, ..., n$, set $D_k \leftarrow \sum_{\ell = 1}^{k} \delta_\ell$.

    (c) For $k = 1, ..., n$, set $RSK_k \leftarrow RD_{D_k}$.

    (d) Set $curr \leftarrow 0$, $i \leftarrow 1$, $start\_idx \leftarrow 1$.

    (e) While $i \leq m$ and $start\_idx \leq n$:

    i. If $y_{1start(start\_idx)} < t_i$, set $RSk_{start(start\_idx)} \leftarrow RSK_{start(start\_idx)} - curr$ and $start\_idx \leftarrow start\_idx + 1$.

    ii. If not, set $curr \leftarrow curr + \frac{1}{RSS_i}$ and $i \leftarrow i + 1$.

4. Compute the partial sums $RSKSQ_k = \sum_{i \in C_k} \frac{1}{RSS_i^2}$, $k = 1, ..., n$ in a similar manner as Step 3.

5. Compute $\ell'(\eta)_k$ amd $\ell''(\eta)_{k,k}$ using the formulas (10) and (11):

    $$\ell'(\eta)_k = \delta_k - e^{\eta_k} \cdot RSK_k, \qquad \ell''(\eta)_{k,k} = (e^{\eta_k})^2 \cdot RSKSQ_k - e^{\eta_k} \cdot RSK_k.$$

6. Compute the working responses and weights using the formulas (12) and (13).

```
R> strata <- c(rep(1, 10), rep(2, 10))
R> y2 <- stratifySurv(y, strata)
R> glmnet(x, y2, family = "cox")
```

3.3. Plotting survival curves

The beauty of the Cox partial likelihood is that the baseline hazard, $h_0(t)$, is not required for inference on the model coefficients β. However, the estimated hazard is often of interest to users. The **survival** package already has a well-established `survfit` method that can produce estimated survival curves from a fitted Cox model. **glmnet** implements a

`survfit` method for regularized Cox models fit by `glmnet` by creating the `coxph` object corresponding to the model and calling `survival::survfit`.

The code below is an example of calling `survfit` for `coxnet` objects for a particular value of the λ tuning parameter (in this case, λ = 0.05). Note that we had to pass the original design matrix `x` and response `y` to the `survfit` call: they are needed for `survfit.coxnet` to reconstruct the required `coxph` object. The survival curves are computed for the individuals represented in `newx`: we get one curve per individual, as seen in Figure 3.

```
R> fit <- glmnet(x, y, family = "cox")
R> sf_obj <- survfit(fit, s = 0.05, x = x, y = y, newx = x[1:2, ])
R> plot(sf_obj, col = 1:2, mark.time = TRUE, pch = "12")
```

The `survfit` method is available for Cox models fitted by `cv.glmnet` as well. By default, the survival curves are computed for the `lambda.1se` value of the λ hyperparameter. The user can use the code below to compute the survival curve at the `lambda.min` value:

```
R> c.fit <- cv.glmnet(x, y, family = "cox", nfolds = 5)
R> survfit(cfit, s = "lambda.min", x = x, y = y, newx = x[1:2, ])
```

4. The relaxed lasso

Due to the regularization penalty, the lasso tends to shrink the coefficient vector $\hat{\beta}$ toward zero. The relaxed lasso (Meinshausen 2007) was introduced as a way to undo the shrinkage inherent in the lasso estimator. Through extensive simulations, Hastie et al. (2020) conclude that the relaxed lasso performs well in terms of predictive performance across a range of scenarios. It was found to perform just as well as the lasso in low signal-to-noise (SNR) scenarios and nearly as well as best subset selection in high (SNR) scenarios. It also has a considerable advantage over best subset and forward stepwise regression when the number of variables, p, is large. In this section, we describe the simplified version of the relaxed lasso proposed by Hastie et al. (2020) and give details on how it is implemented in **glmnet**.

For simplicity, we describe the method for the OLS setting (`family = "gaussian"`) and for the lasso ($\alpha = 1$). For a given tuning parameter λ, let $\hat{\beta}^{lasso}(\lambda) \in \mathbb{R}^p$ denote the lasso estimator for this value of λ. Let $\mathscr{A}_\lambda$ denote the active set of the lasso estimator, and let $\hat{\beta}^{LS}_{A_\lambda} \in \mathbb{R}^{|A_\lambda|}$ denote the OLS coefficients obtained by regressing y on $X_{\mathscr{A}_\lambda}$ (i.e. the subset of columns of X which correspond to features in the active set $\mathscr{A}_\lambda$). Let $\hat{\beta}^{LS}(\lambda) \in \mathbb{R}^p$ denote the OLS coefficients $\hat{\beta}^{LS}_{A_\lambda}$ padded with zeros to match the zeros of the lasso solution. The (simplified version of the) relaxed lasso estimator is given by

$$\hat{\beta}^{relax}(\lambda, \gamma) = \gamma \hat{\beta}^{lasso}(\lambda) + (1 - \gamma) \hat{\beta}^{LS}(\lambda), \tag{14}$$

where $\gamma \in [0, 1]$ is a hyperparameter, similar to α. In other words, the relaxed lasso estimator is a convex combination of the lasso estimator and the OLS estimator for the lasso's active set.

The relaxed lasso can be fit with `glmnet` function in **glmnet** by setting the argument `relax = TRUE`:

```
R> data(QuickStartExample)
R> fit <- glmnet(x, y, relax = TRUE)
```

When called with this option, `glmnet` first runs the lasso (Algorithm 1 with $\alpha = 1$) to obtain the lasso estimates $\hat{\beta}^{lasso}(\lambda_k)$ and the active sets $\mathscr{A}_{\lambda_k}$ for a path of hyperparameter values $\lambda_1 > \cdots > \lambda_m$. It then goes down this sequence of hyperparameter values again, fitting the unpenalized model of y on each $X_{\mathscr{A}_{\lambda_k}}$ to obtain $\hat{\beta}^{LS}(\lambda_k)$. The refitting is done in an efficient manner. For example, if $\mathscr{A}_{\lambda_\ell} = \mathscr{A}_{\lambda_k}$, `glmnet` does not fit the OLS model for λ_ℓ but sets $\hat{\beta}^{LS}(\lambda_\ell) = \hat{\beta}^{LS}(\lambda_k)$.

The returned object has a `predict` method which the user can use to make predictions on future data. As an example, the code below returns the relaxed lasso predictions for the training data at $\gamma = 0.5$ (the default value is `gamma = 1`, i.e. the lasso estimator):

```
R> predict(fit, x, gamma =0.5)
```

The `cv.glmnet` function works with the relaxed lasso as well. When cross-validating a relaxed lasso model, `cv.glmnet` provides optimal values for both the `lambda` and `gamma` parameters.

We note that we can consider as many values of the γ hyperparameter as we like in CV. Most of the computational time is spent obtaining $\hat{\beta}^{lasso}(\lambda)$ and $\hat{\beta}^{LS}(\lambda)$; once we have have computed them $\hat{\beta}^{relax}(\lambda, \gamma)$ is simply a linear combination of the two. By default, `cv.glmnet` performs CV for `gamma = c(0, 0.25, 0.5, 0.75, 1)`.

In the exposition above we have focused on the `family = "gaussian"` case. Relaxed fits are also available for the rest of the other model families, i.e. any other `family` argument. Instead of fitting the OLS model of the response on the active set to obtain the relaxed fit, `glmnet` fits the unpenalized model for that model family on the active set.

We note that while the relaxation can be applied for α values smaller than 1, we do not recommend doing this. Relaxation is typically applied to obtain sparser models. It achieves this by undoing shrinkage of coefficients in the active set toward zero, allowing the model to have more freedom to fit the response. Together with CV on λ and γ, this often gives us a model that is sparser than the lasso. Selecting α smaller than 1 results in a larger active set than that for the lasso, working against the goal of obtaining a sparser model.

4.1.   Application to forward stepwise regression

One use case for the relaxed fit is to provide a faster version of forward stepwise regression. When the number of variables p is large, forward stepwise regression can be tedious since it only adds one variable at a time and at each step, it needs to try all predictor variables that are not already included in the model to find the best one to be added. On the other hand, because the lasso solves a convex problem, it can identify good candidate sets of variables over 100 values of the λ hyperparameter even when p is in the tens of thousands. In a case like this, one can run `cv.glmnet` and fit the OLS model for a sequence of selected variable sets.

```
R> fitr <- cv.glmnet(x, y, gamma = 0, relax = TRUE)
```

5.   Assessing models

After fitting elastic net models with `glmnet`, we often want to assess their performance on a set of evaluation or test data. After deciding on the performance measure, for each model in the fitted sequence (indexed by the value of λ and possibly γ for relaxed fits) we have to build a matrix of predictions and compute the performance measure for it.

`cv.glmnet` does some of this evaluation automatically. In performing CV, `cv.glmnet` computes the pre-validated fits (Tibshirani and Efron 2002), that is the model's predictions of the linear predictor on the held-out fold, and then computes the performance measure with these pre-validated fits. The performance measures are recorded in the `cvm` element of the returned `cv.glmnet` and are used to make the CV `plot` when the plot method is called.

glmnet supports a variety of performance measures depending on the model family: the full list of measures can be seen via the call `glmnet.measures()`. The user can change the performance measure computed in CV by specifying the `type.measure` argument. For example, the code below computes the area under the curve (AUC) of the pre-validated fits instead of the deviance which is the default for `family = "binomial"`:

```
R> fitr <- cv.glmnet(x, y, family = "binomial", type.measure = "auc")
```

More generally, model assessment can be performed using the `assess.glmnet` function. The user can pass a matrix of predictions, a class `"glmnet"` object, or a class `"cv.glmnet"` object to `assess.glmnet` along with the true response values. The code below shows how one can use `assess.glmnet` with these three objects, where the training design matrix and response is x[itrain,] and y[itrain] respectively and the testing design matrix and response is x[-itrain,] and y[-itrain] respectively.

```
R> fit <- glmnet(x[itrain, ], y[itrain])
R> assess.glmnet(fit, newx = x[-itrain, ], newy = y[-itrain])
```

```
R> pred <- predict(fit, newx = x[-itrain, ])
R> assess.glmnet(pred, newy = y[-itrain])


R> cfit <- cv.glmnet(x[itrain, ], y[itrain])
R> assess.glmnet(cfit, newx = x[-itrain, ], newy = y[-itrain])
```

By default `assess.glmnet` will return all possible performance measures for the model family. Note that if a matrix of predictions is passed, the user has to specify the model family via `family` argument since `assess.glmnet` cannot infer that from the inputs. (The default value for the `family` argument is `"gaussian"`, which is what would have been used in the code above.) If a class `"glmnet"` object is passed to `assess.glmnet`, it returns one performance measure value for each model in the λ sequence while if a class `"cv.glmnet"` object is passed, it returns the performance measure value at the `lambda. lse` value of the λ hyperparameter. The user can get the performance measure values at other values of the hyperparameters using the s and `gamma` arguments as in the `predict` method.

One major use of `assess.glmnet` is to avoid running CV multiple times to get the values for different performance measures. By default, `cv.glmnet` will only return a single performance measure. However, if the user specifies `keep = TRUE` in the `cv.glmnet` call, the pre-validated fits are returned as well. The user can then pass the pre-validated matrix to `assess.glmnet`. The code below is an example of how to do this for the Poisson model family. (The `keep` argument is `FALSE` by default as the pre-validated matrix is large when the number of training observations is large, thus inflating the size of the returned object.)

```
R> cfit <- cv.glmnet(x[itrain, ], y[itrain], keep = TRUE)
R> assess.glmnet(cfit$fit.preval, newy = y, family = "poisson")
```

We have two additional functions that provide test performance which are unique to binomial data. As the function names suggest, `roc.glmnet` and `confusion.glmnet` produce the receiver operating characteristic (ROC) curve and the confusion matrix respectively for the test data. Here is an example of the output the user gets from `confusion.glmnet`:

```
R> data(MultinomialExample)
R> set.seed(101)
R> itrain <- sample(1:500, 400, replace = FALSE)
R> cfit <- cv.glmnet(x[itrain, ], y[itrain], family = "multinomial")
R> cnf <- confusion.glmnet(cfit, newx = x[-itrain, ], newy = y[-itrain])
R> print(cnf)
```

| | True | | | |
Predicted	1	2	3	Total
1	13	6	4	23
2	7	25	5	37
3	4	3	33	40
Total	24	34	42	100

Percent Correct: 0.71

6. Discussion

We have shown how to extend the use of the elastic net penalty to all GLM model families, Cox models with (start, stop] data and with strata, and to a simplified version of the relaxed lasso. We have also discussed how users can use the **glmnet** package to assess the fit of these elastic net models. These new capabilities are available in version 4.1 and later of the **glmnet** package on CRAN.

Acknowledgments

References

Andersen PK, Gill RD (1982). "Cox's Regression Model for Counting Processes: A Large Sample Study" Annals of Statistics, 10(4), 1100–1120.

Breslow NE (1972). "Contribution to the Discussion of the Paper by D. R. Cox" Journal of the Royal Statistical Society: Series B (Methodological), 34, 216–217.

Cox DR (1972). "Regression Models and Life-Tables" Journal of the Royal Statistical Society: Series B (Methodological), 34(2), 187–220.

Friedman J, Hastie T, Tibshirani R (2010). "Regularization Paths for Generalized Linear Models via Coordinate Descent" Journal of Statistical Software, 33(1), 1–24. [PubMed: 20808728]

Hastie T, Tibshirani R, Tibshirani R (2020). "Best Subset, Forward Stepwise or Lasso? Analysis and Recommendations Based on Extensive Comparisons." Statistical Science, 35(4), 579–592.

Hoerl AE, Kennard RW (1970). "Ridge Regression: Biased Estimation for Nonorthogonal Problems." Technometrics, 12(1), 55–67.

McCullagh P, Nelder JA (1983). Generalized Linear Models. Springer US. ISBN 9780412238505. URL https://books.google.com/books?id=OUitAQAACAAJ.

Meinshausen N (2007). "Relaxed Lasso." Computational Statistics & Data Analysis, 52(1), 374–393.

Nelder JA, Wedderburn RWM (1972). "Generalized Linear Models." Journal of the Royal Statistical Society: Series A (General), 135(3), 370–384.

Simon N, Friedman J, Hastie T, Tibshirani R (2011). "Regularization Paths for Cox's Proportional Hazards Model via Coordinate Descent." Journal of Statistical Software, 39(5), 1–13.

Therneau TM (2020). A Package for Survival Analysis in R. R package version 3.2–7, URL https://CRAN.R-project.org/package=survival.

Therneau TM, Grambsch PM (2000). Modeling Survival Data: Extending the Cox Model. Springer.

Tibshirani R (1996). "Regression Shrinkage and Selection via the Lasso." Journal of the Royal Statistical Society: Series B (Methodological), 58(1), 267–288.

Tibshirani R, Bien J, Friedman J, Hastie T, Simon N, Taylor J, Tibshirani RJ (2012). "Strong Rules for Discarding Predictors in Lasso-Type Problems." Journal of the Royal Statistical Society: Series B (Statistical Methodology), 74(2), 245–266. [PubMed: 25506256]

Tibshirani RJ, Efron B (2002). "Pre-Validation and Inference in Microarrays." Statistical Applications in Genetics and Molecular Biology, 1(1), 1–19.

van der Kooij AJ (2007). Prediction Accuracy and Stability of Regression with Optimal Scaling Transformations. Ph.d. thesis, Leiden University.

Zou H, Hastie T (2005). "Regularization and Variable Selection via the Elastic Net." Journal of the Royal Statistical Society: Series B (Statistical Methodology), 67(2), 301–320.

Fitting time: "gaussian" vs. gaussian()

Fitting time: "binomial" vs. binomial()
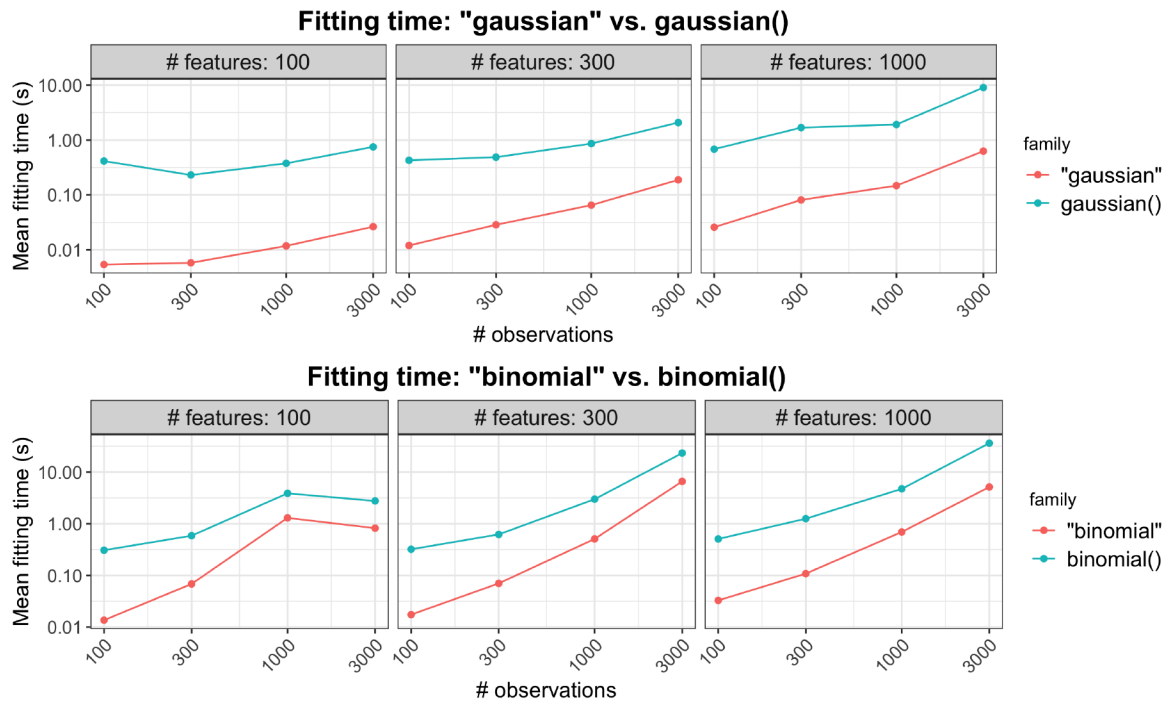
Figure 1:
The top plot compares model fitting times for `family = "gaussian"` and `family = gaussian()` for a range of problem sizes, while the plot below compares that for `family = "binomial"` and `family = binomial()`. Each point is the mean of 5 simulation runs. Note that both the *x* and *y* axes are on the log scale.
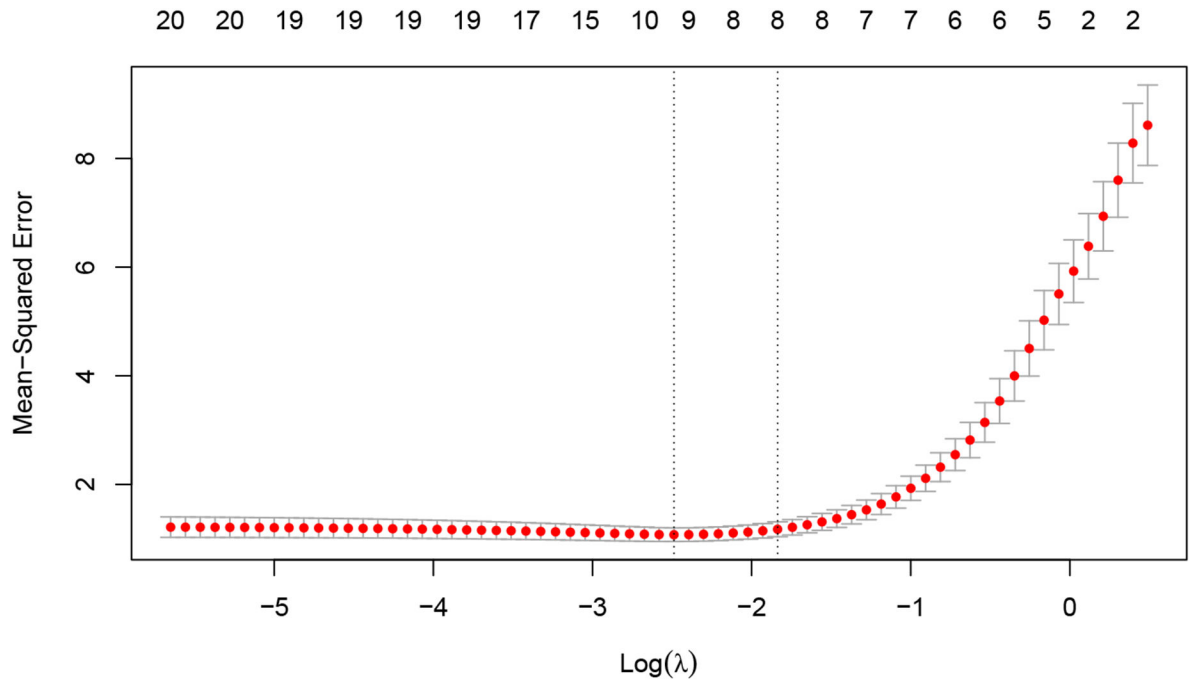
Figure 2:
Example output for plotting a `cv.glmnet` object: a plot of CV error against $\log(\lambda)$. The error bars correspond to ± 1 standard error. The left vertical line corresponds to the minimum error while the right vertical line corresponds to the largest value of λ such that the CV error is within one standard error of the minimum. The top of the plot is annotated with the size of the models, i.e. the number of predictors with non-zero coefficient.
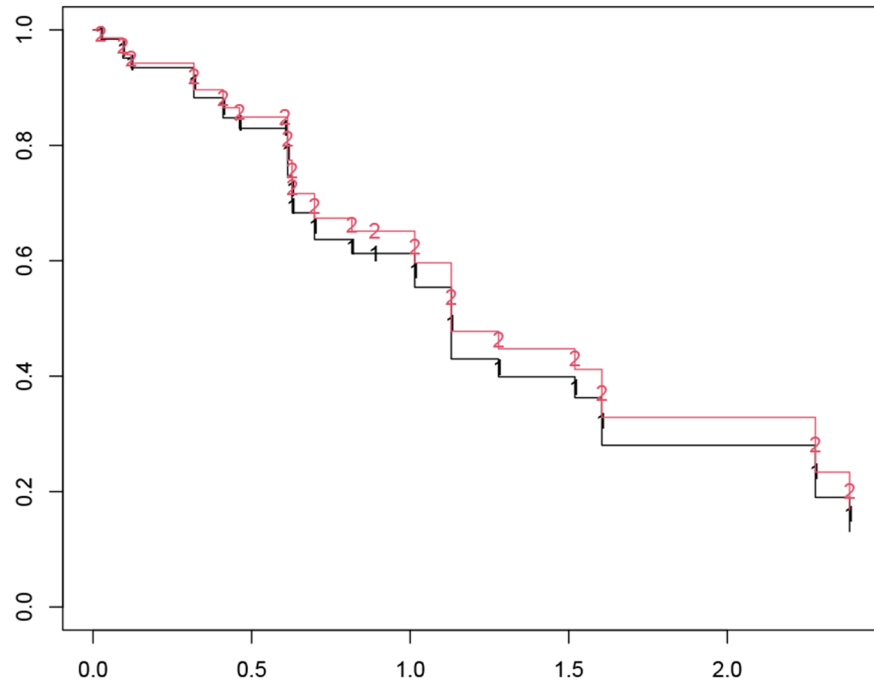
Figure 3:
An illustration of the plotted `survfit` object. One survival curve is plotted for each individual represented in the `newx` argument.

Table 1:

Examples of generalized linear models (GLMs) and their representations in R.

GLM family / Regression type	Response type	Representation in R
Gaussian	$\mathbb{R}$	`gaussian()`
Logistic	$\{0, 1\}$	`binomial()`
Probit	$\{0, 1\}$	`binomial(link = "probit")`
Quasi-Binomial	$\{0, 1\}$	`quasibinomial()`
Poisson	$\mathbb{N}_0 = \{0, 1, \ldots\}$	`poisson()`
Quasi-Poisson	$\mathbb{N}_0$	`quasipoisson()`
Negative binomial	$\mathbb{N}_0$	`MASS::negative.binomial(theta = 3)`
Gamma	$\mathbb{R}_+ = [0, \infty)$	`Gamma()`
Inverse Gaussian	$\mathbb{R}_+$	`inverse.gaussian()`
Tweedie	Depends on variance power parameter	`statmod::tweedie()`