Pretraining and the Lasso

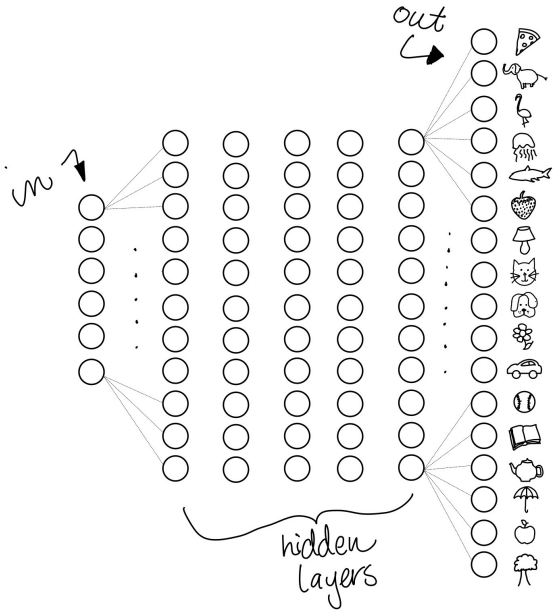
and an

Introduction to `ptLasso`

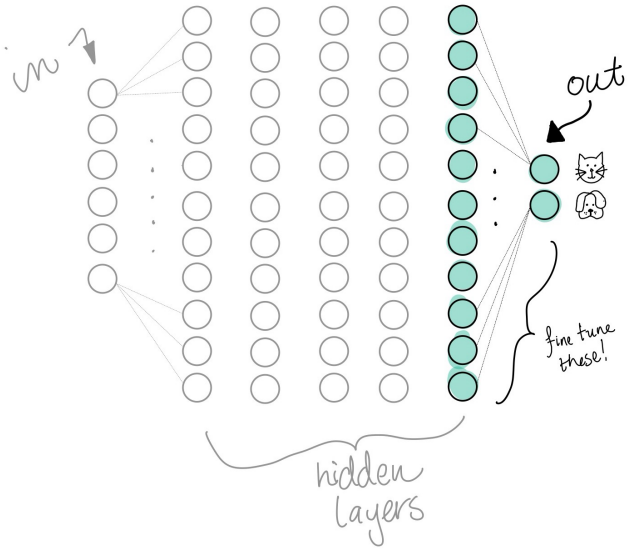
Erin Craig

Mert Pilanci, Thomas Le Menestrel, Balasubramanian Narasimhan,
Manuel Rivas, Roozbeh Dehghannasiri, Julia Salzman,
Jonathan Taylor, Robert Tibshirani

Pretraining with neural networks



1. Fit a model using e.g. ImageNet to distinguish cats, dogs, jellyfish, strawberries...

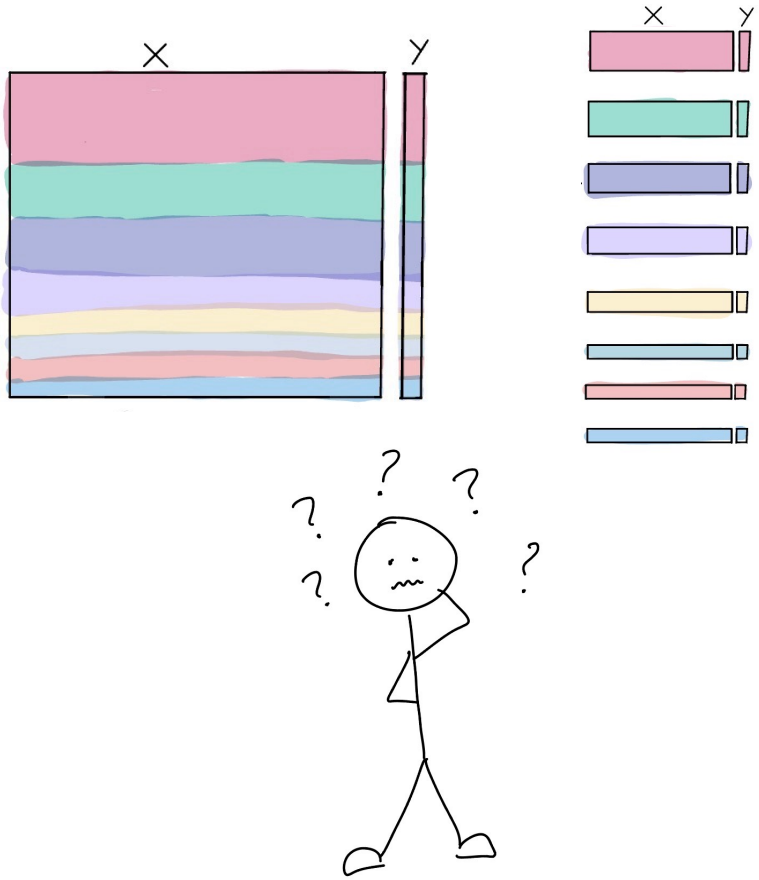


2. Fine tune using a smaller dataset, with the goal to distinguish cats and dogs.

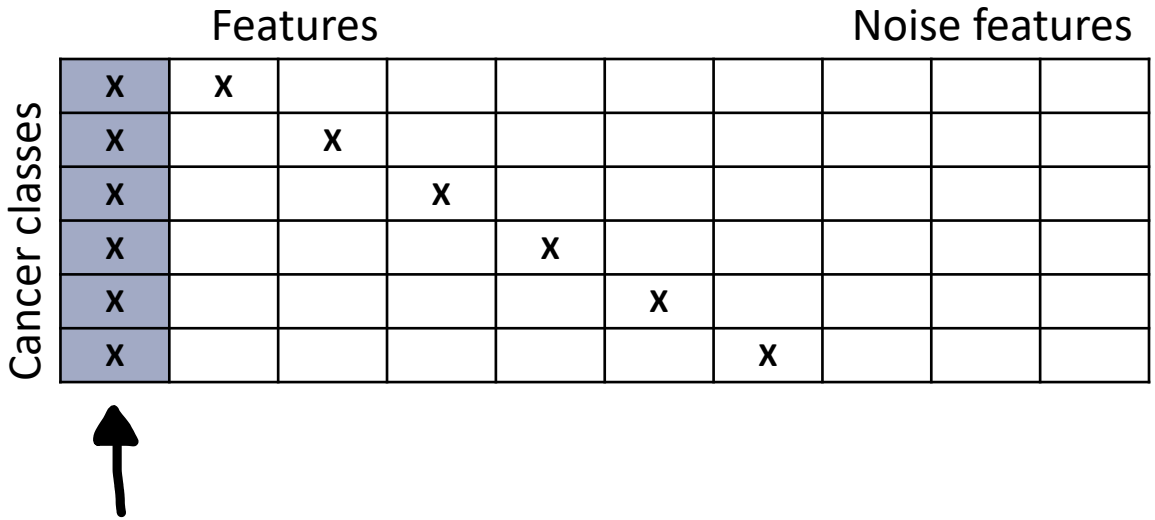
A challenge

- Suppose we have a dataset spanning ten cancers and we want to fit a lasso logistic regression model to predict 10-year survival.
- Some cancers have more observations (e.g. breast) and some have fewer (e.g. head and neck).
- Is it better to pool data from all cancers together and fit *one* model?

  Or, would it be better to fit a separate model for *each* cancer?

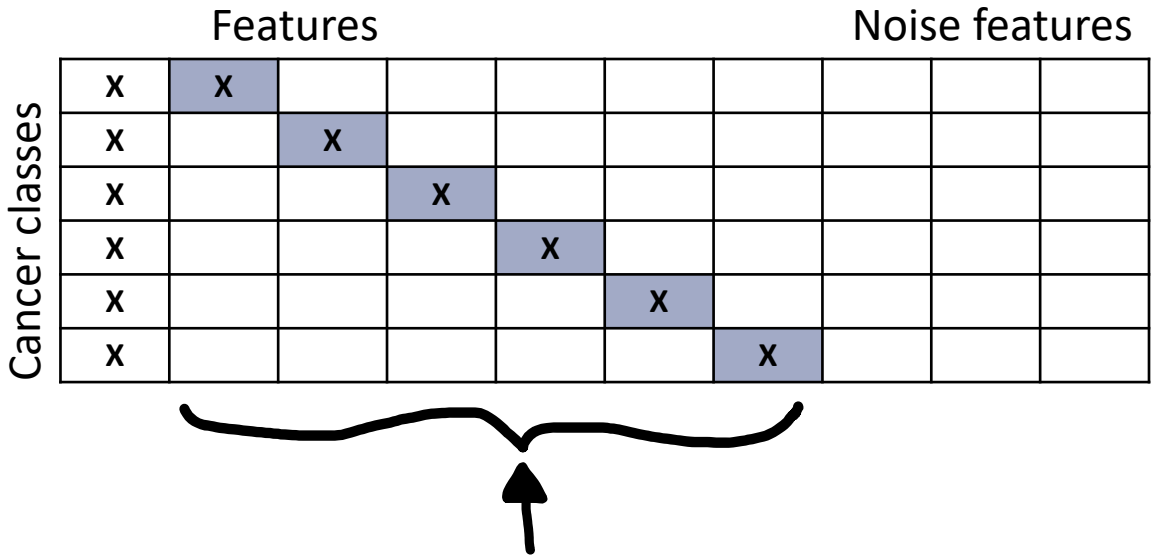


Our conceptual model



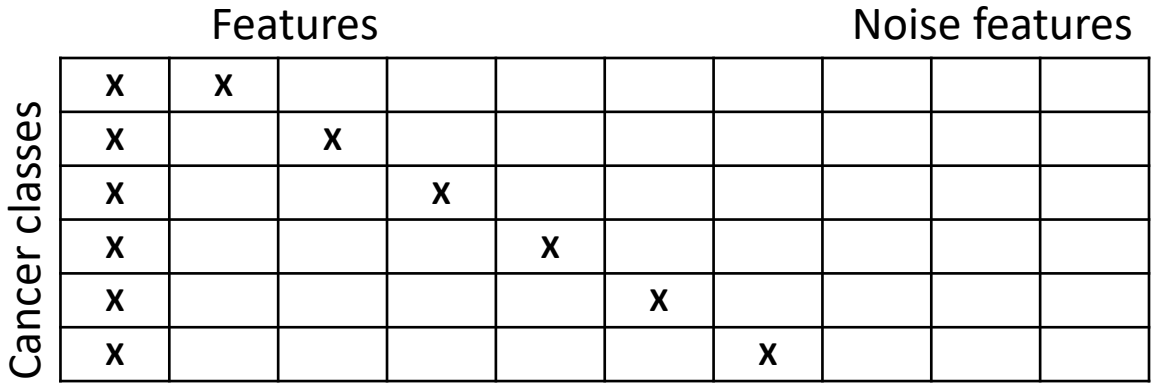
Some features are predictive across all classes.
Combining all cancer classes gives us the most power to discover these features.

Our conceptual model



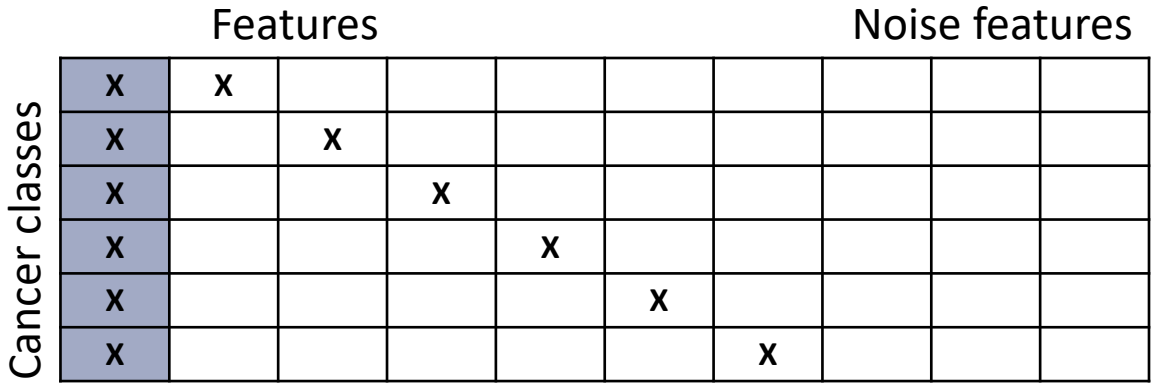
Some features are predictive in just one (or a few) classes.
Discovering them is easiest when we fit a
separate model for each class.

Our conceptual model



Pretraining with the lasso gives us the best of both worlds.

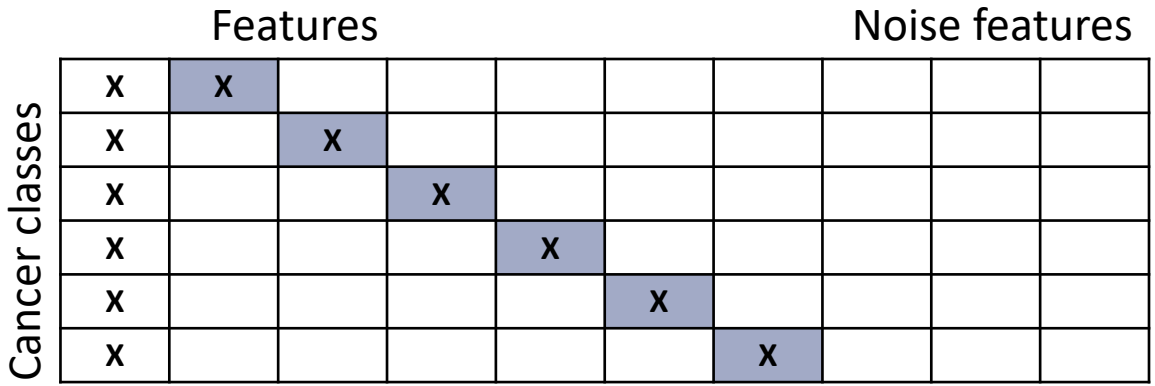
Our conceptual model



Pretraining with the lasso gives us the best of both worlds.

Pretraining for the lasso does model fitting in two steps.
In the first, we pool all data and fit a model.

Our conceptual model



Pretraining with the lasso gives us the best of both worlds.

Pretraining for the lasso does model fitting in two steps.
In the first, we pool all data and fit a model.
In the second, we fine tune this model
separately for each class.

Lasso pretraining is a *general method* to pass information across models

Pretraining can be used for:

- Grouped data (as in our example)
- Data with a multinomial outcome
- Time series data
- Multi-response data
- Conditional average treatment effect estimation
- Datasets with large amounts of unlabeled data
- … and more!

Before describing pretraining in more detail, we will review the lasso.

Review: lasso linear regression

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \frac{1}{2\,N} \sum_{i=1}^{N} \left(y_i - x_i^T \beta\right)^2 + \lambda \sum_{k=1}^{p} |\beta_k|$$

Review: lasso linear regression

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \frac{1}{2\,N} \sum_{i=1}^{N} \left(y_i - x_i^T \beta\right)^2 + \lambda \sum_{k=1}^{p} |\beta_k|$$

Encourage $x_i^T \beta$ to be close to y_i.

Review: lasso linear regression

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \frac{1}{2\,N} \sum_{i=1}^{N} \left(y_i - x_i^T \beta\right)^2 + \lambda \sum_{k=1}^{p} |\beta_k|$$

Encourage β to be sparse.

This does *feature selection*.

Sparsity can yield better prediction accuracy and interpretability.

Review: the lasso

- Though I showed a lasso linear regression problem, the lasso can be applied to the entire GLM family (including logistic and Poisson regression for example), as well as Cox's proportional hazards model.

- There are many software packages that implement the lasso, a popular choice is `glmnet` in R.

- Varying the regularization parameter $\lambda \geq 0$ yields a path of solutions: an optimal value $\hat{\lambda}$ is usually chosen by cross-validation using for example the `cv.glmnet` function from the package `glmnet`.

Two modeling options that we use

1. We specify an **offset** O_i for each observation $\mathrm{x_i}$.

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \frac{1}{2\,N} \sum_{i=1}^{N} \left(y_i - \boldsymbol{O_i} - x_i^T \beta\right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

For the case of linear regression, using an offset means that we are fitting a *residual*.

Two modeling options that we use

1. We specify an **offset** O_i for each observation $\mathrm{x_i}$.

2. We specify a **penalty factor** $\mathrm{pf_j}$ that modifies the lasso penalty for the $\mathrm{j^{th}}$ feature.

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \frac{1}{2\,N} \sum_{i=1}^{N} \left(y_i - O_i - x_i^T \beta\right)^2 + \lambda \sum_{j=1}^{p} \mathbf{pf_j} |\beta_j|$$

If $\mathrm{pf_j}$ is 0, β_j is not penalized at all.
If $\mathrm{pf_j}$ is big *enough*, β_j is omitted.

Two modeling options that we use

1. We specify an **offset** O_i for each observation $\mathrm{x_i}$.

2. We specify a **penalty factor** $\mathrm{pf_j}$ that modifies the lasso penalty for the $\mathrm{j^{th}}$ feature.

Both options are standard in generalized linear models (and in the software package `glmnet`).

Now we are ready to show how pretraining works.

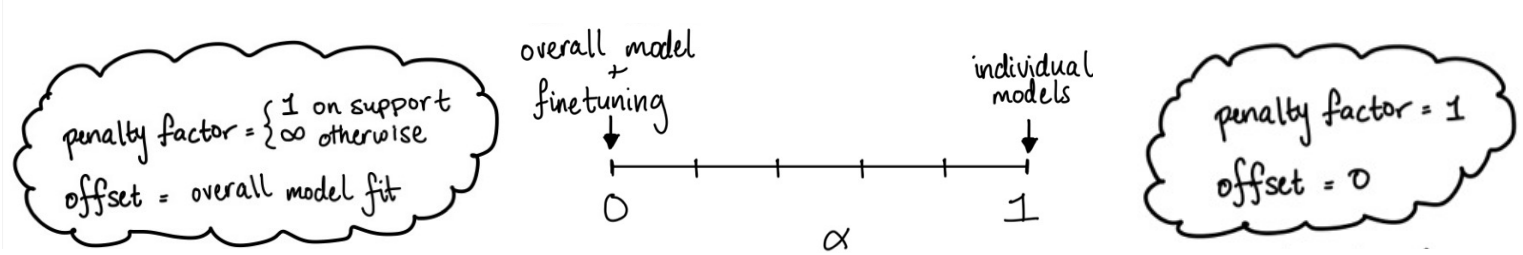
Lasso pretraining details

1. *Overall model*:
   Fit a lasso penalized model to the full dataset X, y to get coefficients $\hat{\beta}_0$.

2. *Group-specific models:*
   Choose the hyperparameter α $(0 \leq \alpha \leq 1)$. Fit a model for each *group* X_k, y_k:

   a. Offset: $(1 - \alpha)\, X_k \hat{\beta}_0$

   b. Penalty factor: $\mathrm{pf_j} = 1$ (features selected by the overall model)
      Penalty factor: $\mathrm{pf_j} = \frac{1}{\alpha}$ (other features)



The `ptLasso` package in R

- `ptLasso` is a package that fits pretrained models using the `glmnet` package, including lasso, elasticnet and ridge models. All model fitting in `ptLasso` is done with `cv.glmnet`.

- The first step of pretraining is a straightforward call to `cv.glmnet`.
  The second step calls `cv.glmnet` with an offset and penalty factor for each group.

- One call to `ptLasso` fits an overall model, pretrained class specific models, and class specific models for each group (without pretraining).

- The `ptLasso` package also includes methods for prediction and plotting,
  and a function that performs cross-validation to choose α.

A quick start example

A typical modeling pipeline looks like this:

```
# Data: features X, response y and group identifier "groups"

# Fit a model:
fit = ptLasso(X, y, groups)
# ...optionally using cross-validation for parameter selection:
cvfit = cv.ptLasso(X, y, groups)

# Visualize the model:
plot(fit)
plot(cvfit)

# And make predictions:
predict(fit, Xtest, groupstest)
predict(cvfit, Xtest, groupstest)

# Inspect the coefficients
coef(fit)
coef(cvfit)
```

We'll do an example using `ptLasso` with a simple simulated dataset.

Simulate data

We simulate data with 3 groups and a continuous response.

- $n = 200$ observations in each group and $p = 80$ features.
- Groups share 10 informative features with different coefficient values.
- Each group has 10 additional group-specific features.
- Other features are noise.

```
require(ptLasso)

set.seed(1234)

out = gaussian.example.data(k = 3)
x = out$x; y = out$y; groups = out$groups

outtest = gaussian.example.data(k = 3)
xtest = outtest$x; ytest = outtest$y; groupstest = outtest$groups
```

Simulate data

We simulate data with 3 groups and a continuous response.

- $n = 200$ observations in each group and $p = 80$ features.
- Groups share 10 informative features with different coefficient values.
- Each group has 10 additional group-specific features.
- Other features are noise.

Sanity check:

```
dim(x)
## [1] 600  80
head(y, 3)
## [1]  -7.593043389  10.842131973  -9.845550692
```

Model fitting and prediction

We are ready to fit a model using `ptLasso`.

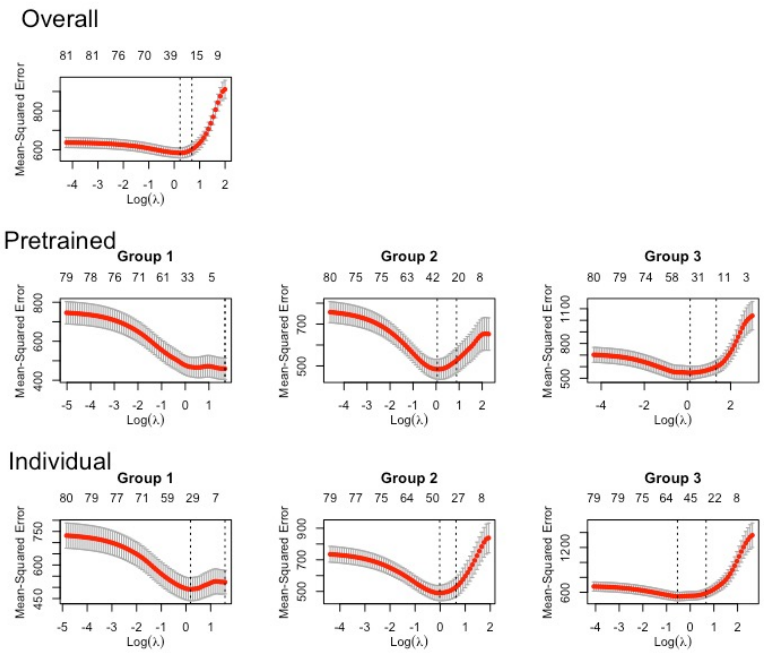
We'll choose the pretraining parameter $\alpha = 0.5$ (randomly chosen).

Here, we'll go through three steps:

1. fitting a model,
2. predicting with a validation set, and
3. extracting the model coefficients.

Model fitting

```
fit <- ptLasso(x, y, groups, alpha = 0.5)
plot(fit)
```



`ptLasso` used `cv.glmnet` to fit 7 models:

- 1 *overall* model (trained using all 3 groups),
- 3 *pretrained* models (one for each group),
- 3 *individual* models (one for each group).

`plot` shows their cross validation curves.

Prediction

```
preds = predict(fit, xtest, groupstest=groupstest)
```

`predict` makes predictions from all 7 models. It returns a list containing:
1. `yhatoverall` (predictions from the overall model),
2. `yhatpre` (predictions from the pretrained models) and
3. `yhatind` (predictions from the individual models).

By default, `predict` uses `lambda.min` – the value of the regularization parameter λ that minimized the cross-validated mean squared error – for all 7 `cv.glmnet` models.

Prediction

If you also provide `ytest` (for model validation), `predict` will measure performance.

```
preds = predict(fit, xtest, groupstest=groupstest, ytest=ytest)
preds
```

```
## Performance (Mean squared error):
##            allGroups  mean group_1 group_2 group_3    r^2
## Overall        566.1 566.1   492.7   574.7   630.7 0.3591
## Pretrain       551.5 551.5   517.5   595.0   542.1 0.3756
## Individual     575.7 575.7   528.5   596.7   602.0 0.3482
##
## Support size:
## Overall    23
## Pretrain   60 (13 common + 47 individual)
## Individual 76
```

Model interpretation

```
coefs = coef(fit)
names(coefs)
## [1] "individual" "pretrain"   "overall"
length(coefs$pretrain)
## [1] 3
head(coefs$pretrain[[3]])
## 6 x 1 sparse Matrix of class
## "dgCMatrix"
##                      s1
## (Intercept) -0.3221821
## V1           5.0591788
## V2           4.9309031
## V3           3.7828370
## V4           5.1038363
## V5           5.6944842
```

`coef` returns a list with the coefficients for the overall, individual and pretrained models.

`coefs$pretrain` is a list of length 3, containing the coefficients for each group.

Cross validation to choose α

In our example, we chose the parameter α randomly.

We recommend making a more thoughtful choice by using:

1. a validation set to measure performance for a few different choices of α (e.g. $0, 0.25, 0.5, 0.75, 1.0$), or
2. `cv.ptLasso`, which will recommend a choice of α based on CV performance.

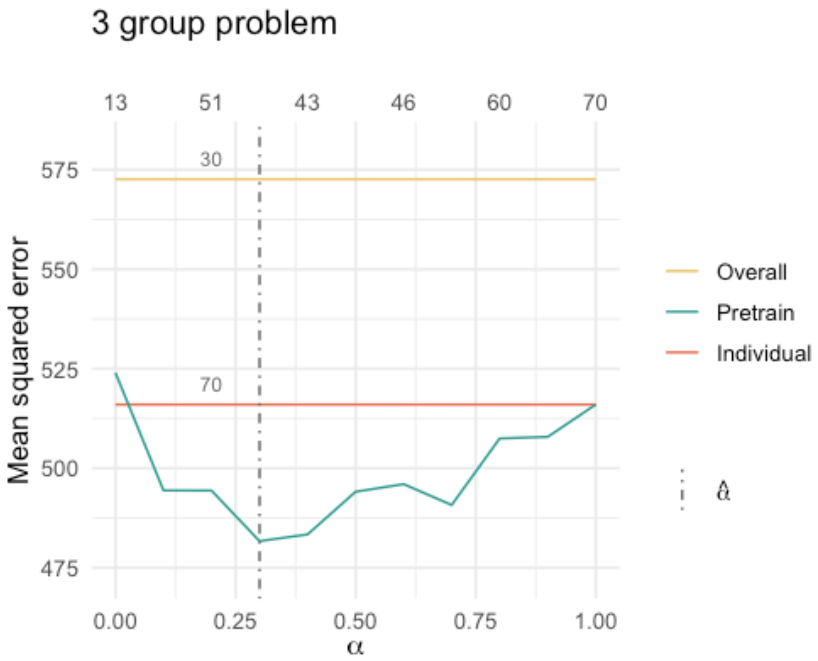
Here, we'll try `cv.ptLasso`.

Cross validation to choose α

```
cvfit <- cv.ptLasso(x, y, groups)
cvfit

## type.measure:  mse
##            alpha overall  mean wtdMean group_1 group_2 group_3
## Overall            572.6 572.6   572.6   517.0   509.3   691.5
## Pretrain     0.0   524.0 524.0   524.0   486.8   549.4   535.6
## Pretrain     0.1   494.4 494.4   494.4   470.3   492.4   520.6
## Pretrain     0.2   494.4 494.4   494.4   467.1   493.9   522.1
...
## Pretrain     0.9   507.9 507.9   507.9   503.8   488.1   531.8
## Pretrain     1.0   516.0 516.0   516.0   504.4   523.0   520.6
## Individual         516.0 516.0   516.0   504.4   523.0   520.6
##
## alphahat (fixed) = 0.3
## alphahat (varying):
## group_1 group_2 group_3
##     0.4     0.4     0.3
```

Cross validation to choose α

```
plot(cvfit, plot.alphahat = TRUE)
```



`plot` visualizes performance as a function of α and draws a vertical line to show the value of α that minimized the CV MSE.

What if we don’t know the input groups?

In our motivating example, each row of X belonged to one cancer class.
But what if we don’t have a predefined grouping on our data?

An example of this may be data with clinical variables like age and sex as well as genomic variables; we often have some prior belief that risks are different for different subpopulations.

In this case, we can use clinical variables to fit a shallow CART tree; each node of the tree defines a separate group.

Given these groups defined by clinical variables, we can then fit pretrained models using the genomic variables as described.

What if we don’t know the input groups?

Here is a quick simulation with 3 groups and additional clinical variables:

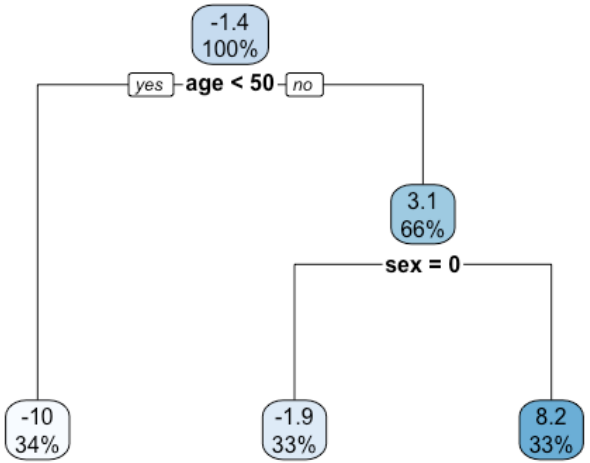
- in group 1, age < 50 and sex is equally likely male or female,
- in group 2, age > 50 and sex is male,
- in group 3, age > 50 and sex is female.

What if we don't know the input groups?

```
out = gaussian.example.data(k = 3, intercepts = c(-10, 0, 10))
x = out$x; y = out$y; groups = out$groups

outtest = gaussian.example.data(k = 3, intercepts = c(-10, 0, 10))
xtest = outtest$x; ytest = outtest$y; groupstest = outtest$groups

n = nrow(x)
clinvars = cbind(
  age = ifelse(c(groups, groupstest) == 1,
               runif(2*n, min = 20, max = 50),
               runif(2*n, min = 50, max = 90)),
  sex = ifelse(c(groups, groupstest) == 1,
               sample(c(0, 1), 2*n, replace = T),
               ifelse(c(groups, groupstest) == 2, 0, 1)
               )
)
clinvars.train = clinvars[1:n, ]
clinvars.test  = clinvars[-(1:n), ]
```

What if we don't know the input groups?

```
require(rpart)
treefit = rpart(y~.,
    data = data.frame(clinvars.train, y),
    control = rpart.control(
                    maxdepth=2,
                    minsplit=50
                )
            )

rpart.plot::rpart.plot(treefit)
```



What if we don’t know the input groups?

We want our tree to return the ID of the terminal node for each observation. The following is a trick that causes `predict` to behave as desired.

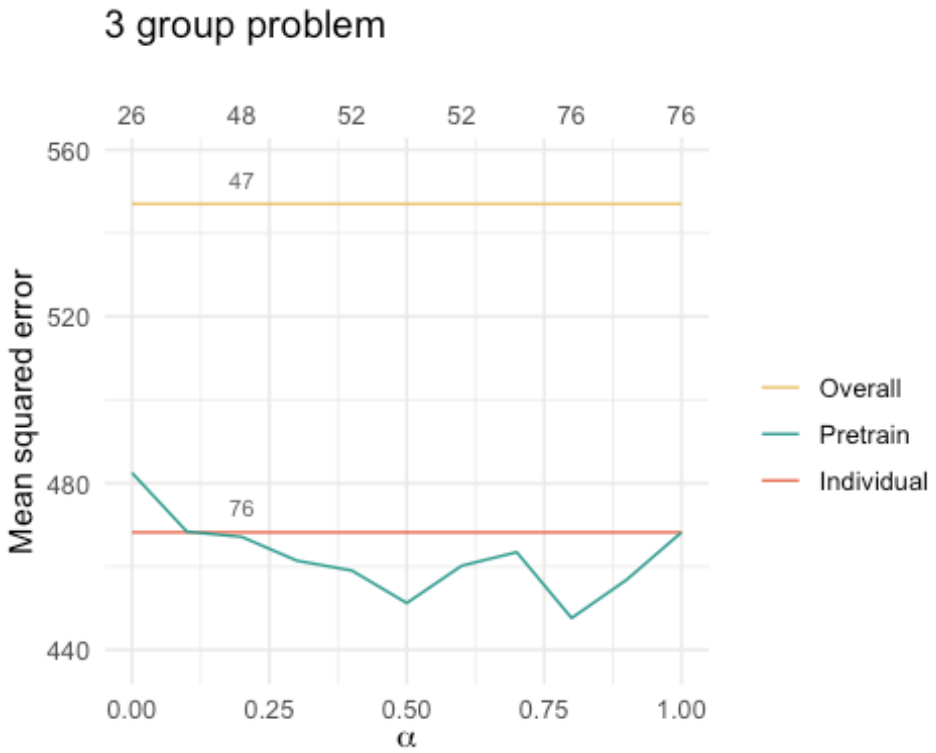
```
leaf=treefit$frame[,1]=="<leaf>"
treefit$frame[leaf,"yval"]=1:sum(leaf)

predgroups.train = predict(treefit, data.frame(clinvars.train))
predgroups.test  = predict(treefit, data.frame(clinvars.test))
```

And now we can fit models with `cv.ptLasso` using our predicted groups:

```
cvfit = cv.ptLasso(x, y, predgroups.train)
```

What if we don’t know the input groups?

```
plot(cvfit)
```



Wrap up

Thank you for watching!

Now we have all the ingredients we need to use `ptLasso` with a real dataset!

See below for links to these slides and an R markdown to accompany them.

Part 2 of this series will cover more use cases of `ptLasso` including datasets with a multinomial target, multi-response data and time series data. I'll see you there!