Nonlinear Methods: Regression

Signal Data Science

In this lesson, we will explore a collection of standard nonlinear regression techniques through predicting wine quality in terms of chemical properties. At the end, we'll combine all of these methods with stacking to create a model which performs better than any individual technique alone.

Getting started

First, we'll load and visualize the data with which we'll be working.

The wine quality dataset can be found in the `wine-quality` dataset folder (with documentation in `winequality.names.txt`) or downloaded on the UCI Datasets page. The white wine dataset is three times the size of the red wine dataset, so we'll be focusing on the former – with more data and consequently a greater "resolution", the advantages of nonlinear techniques become more apparent.

- Load the white wine dataset from `winequality-white.csv`. Read the associated documentation. Replace each space in the column names with an underscore (`"_"`).
- Use `qplot(...) + geom_smooth()` to plot wine quality against each of the individual variables representing chemical properties in the white wine dataset. Which variables are strongly *and* nonlinearly associated with wine quality?

Elastic net regularization

Before using nonlinear methods to predict white wine quality, we will use elastic net regularized linear regression to calculate a performance "baseline" to which we can compare the performance of nonlinear methods.

First, we'll write a utility function for easily accessing `caret`'s `train()`, which will serve as a standardized interface for applying a variety of different nonlinear techniques. `caret` supports a very large number of different models!

- Load the following code into R:

```
caret_reg = function(x, y, method, grid, ...) {
  set.seed(1)
  control = trainControl(method="repeatedcv", repeats=1,
                         number=3, verboseIter=TRUE)
  train(x=x, y=y, method=method, tuneGrid=grid,
        trControl=control, metric="RMSE",
        preProcess=c("center", "scale"), ...)
}
caret_rmse = function(caret_fit) min(caret_fit$results$RMSE)
```

- Write a function `caret_reg(x, y, method, grid, ...)` which returns the output of calling `caret`'s `train()` for a regression task with predictors `x`, a target variable `y`, a model type `method`, and a grid of hyperparameters `grid`. Follow these specifications:
  - The function should set the seed to 1.
  - The `trControl` parameter of `train()` should be set for repeated cross-validation with 1 repeat and 3 folds. Set `verboseIter=TRUE` as well.
  - In the call to `train()`, pass in the parameters `metric="RMSE"` and `preProcess=c("center", "scale")`.
  - The call to `train()` should end like 'train(..)
- Use `expand.grid()` to create a grid of hyperparameters to search over with `alpha=seq(0, 1, 0.1)` and `lambda=2^seq(-4, 1, length.out=20)`.
- Use `caret_reg()` with `method="glmnet"` to fit an elastic net regularized linear model for wine quality. Create a data frame `results` with two columns, `results` and `rmse`, and add in a row with values corresponding to the model you just fit.

As you work through this assignment, continually update `results` with a row for each method you try.

K-Nearest Neighbors

K-Nearest Neighbors (KNN) is one of the simplest possible nonlinear regression techniques.

First, we pick a value of k. Next, suppose that we have a dataset of n points, where each $\mathbf{x}_i$ is associated with a target variable taking on value y_i. Finally, suppose that we have a new point $\mathbf{x}^\star$ and we want to predict the associated

value of the target variable. To do so, we find the k points $\mathbf{x}_i$ which are closest to $\mathbf{x}^\star$, look at the associated values of y_i, and take their average.

The only hyperparameter to tune is k. A larger value of k helps guard against overfitting but will make the model less sensitive to fine-grained structure in the data.[1]

- Evaluate the performance of k-Nearest Neighbors on the wine quality dataset by setting `method="knn` and searching over values of k from 1 to 20.

Multivariate adaptive regression splines

Multivariate adaptive regression splines (MARS) is an extension of linear models which uses *hinge functions*. MARS models a target variable as being a linear combination of functions of the form $\max(0, \pm(x_i - c))$, where x_i is one of the predictor variables.

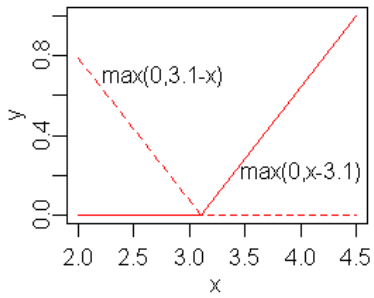

Figure 1: Two examples of hinge functions.

Combining hinge functions allows us to model nonlinear relationships in the data, like so:

[1] One way to interpret k-NN is that it's equivalent to the imposition of a Bayesian prior saying that your dataset is sufficiently fine-grained enough that the value of the target variable at any given point is completely determined by the value of the target variable at nearby points. Given enough granularity, this is in some sense guaranteed to be true (assuming your target variable is a reasonably smooth function of the feature space), but often it's not *perfectly* true. Later on, we'll be looking at support vector machines with a Gaussian basis / radial kernel function, which can be interpreted as a sort of "regularized k-nearest neighbors model" and performs very well in practice for classification.

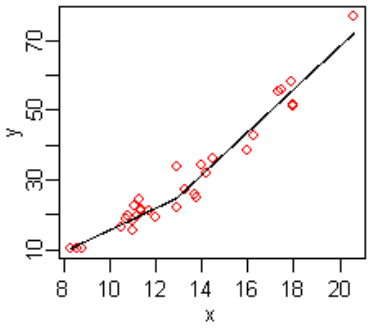

Figure 2: A simple MARS model.

By increasing the *degree* of a MARS model, one can allow for *products* of multiple hinge functions (*e.g.*, $\max(0, x_1 - 10) \times \max(0, 2 - x_3)$), which models interactions between the predictor variables.

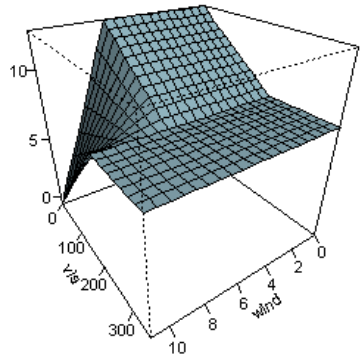

Figure 3: A 2-dimensional MARS model with variable interactions. Note the curvature in the middle of the graph.

Intuitively, one can think of a 1st MARS model with p predictor variables as being a piecewise linear combination of hyperplanes: with 1 predictor variable our resulting model looks like various straight lines connected together, with 2 predictors our resulting model looks like various planes pasted together, and so on and so forth. Raising the degree then allows for polynomial-like nonlinear *curvature*, so instead of connecting together linear functions in a piecewise manner we connect together quadratics, cubics, etc.

MARS models have two hyperparameters: `degree`, the degree of the model, and `nprune`, the maximum number of additive terms allowed in the final model. The latter controls the maximum complexity of the model.

- Evaluate the performance of MARS on the wine quality dataset by setting

`method="earth"` and searching over `degree=1:5` and `nprune=10:30`.[2]

`caret` fits a model at the end with the whole dataset and the best hyperparameters, storing the model in `$finalModel`. Use `summary()` to view the model and interpret the results. (Note that `h(...)` represents a term of the form $\max(0, \ldots)$.)

The advantage of MARS models is that they are easy to fit and interpret, especially 1st degree MARS models.

Decision trees

The previous two nonlinear algorithms we have seen are not particularly impressive, outperforming elastic net regularized linear regression only by a relatively small margin. However, we'll move into a discussion of *decision trees*. By themselves, decision trees are *also* not particularly good and tend to overfit badly, but they can be *combined* into various different regression algorithms which *do* perform quite well.

Standard regression trees

Regression trees are easiest to understand visually:

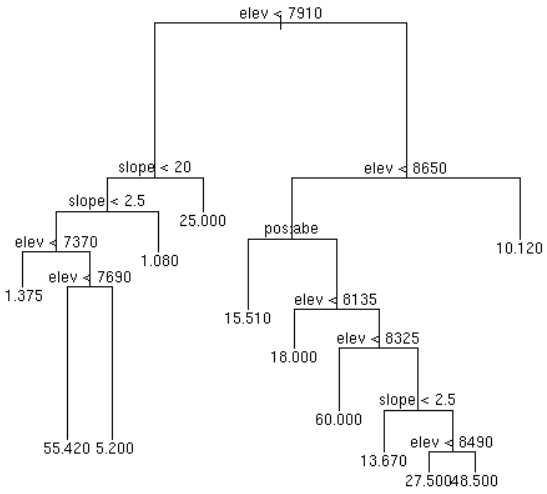

Figure 4: An illustration of a regression tree model.

[2]Why "earth"? Wikipedia says: "The term 'MARS' is trademarked and licensed to Salford Systems. In order to avoid trademark infringements, many open source implementations of MARS are called 'Earth'."

Predictions are made on new data by following the tree down from the top to one of the terminal nodes and then taking the average of the values of all the training points which are located at that node. At each split, the recursive partitioning algorithm tries to find the variable which, if used for that split, would improve the algorithm's predictions the most. The algorithm stops growing the tree when no split would improve the prediction quality above a predefined limit.

The *complexity parameter*, usually denoted `cp`, is the single hyperparameter used for fitting regression tree models. It is the "predefined limit" mentioned above.

- Evaluate the performance of regression trees on the wine quality dataset by setting `method="rpart"` and searching over `cp=10^seq(-3, 0, length.out=10)`.
- View the final model by directly printing it in the console. (Unlike with MARS, the result of `summary()` is *less* interpretable than just printing the fit object directly.) Interpret the results.

Random forests

In short, a *random forest* trains many different different regression trees and averages their predictions together, with these two conditions on the regression trees:

1. Each regression tree is trained on a bootstrapped sample of the original dataset.[3] This technique is known as *bagging* and helps combat overfitting.
2. At each split of each regression tree, only a random subset of the original predictors are considered as candidate variables for the split. This prevents very strong predictors from dominating certain splits, *decorrelating* the regression trees from each other.

The size of the random subset of predictors considered at each split is the single hyperparameter involved in fitting a random forest model, typically denoted `mtry`. For a dataset with p predictors, it is typically advised to try setting `mtry` to floor$(\sqrt{p})$, floor$(p/3)$, and p, with further refinement of the search space afterward if desired.[4] The computation time required for larger datasets can be quite significant, so `mtry = floor(sqrt(p))` works well for obtaining a baseline level of performance with nonlinear techniques.

[3] Recall that a bootstrapped sample of a dataset is a dataset of the same size formed by repeatedly sampling the original dataset *with replacement*.

[4] Setting the value of `mtry` carefully is of debatable importance. Random Forests for Classification in Ecology by Cutler *et al.* (2007) reports that performance isn't very sensitive to `mtry`, whereas Conditional variable importance for random forests by Strobl *et al.* (2008) reports the opposite. Finally, Random Forests: some metholodological insights by Genuer *et al.* (2008) finds varying importance for `mtry` depending on properties of the dataset. *All considered,* I think it's fine to initially just try $p/3$, $\sqrt{p}$, and p, and to decide if further tuning is warranted based on those results.

- Evaluate the performance of random forests on the wine quality dataset by setting `method="ranger"` and searching over `mtry=2:6`.[5]

As a bonus, we can fit each data point in the training data with the trees which were *not* trained on that data point to obtain an *out-of-bag error*, which is an estimate for the generalizable error of our model. This actually saves us the trouble of using

With this, we don't really have to use cross-validation to estimate the generalizable error of our model.

Second, when using the `predict()` function on a random forest model, there is an important point to keep in mind. Suppose that we've run `rf = randomForest(y ~ x, df)` and we want to evaluate the RMSE associated with that fit. To that end, we'd like to generate predictions on the original dataset. We can run one of two commands:

1. `predict(rf)`, which will make predictions for each data point only with trees which weren't trained on that data point, thereby allowing us to calculate a generalizable *out-of-bag error*, and
2. `predict(rf, df)`, which will use the *entire tree* and seem to indicate severe problems with overfitting if we calculate the associated RMSE.

Usually, what you want is `predict(rf)`, not `predict(rf, df)`.

Gradient boosted trees

Boosting is a technique that iteratively improves a decision tree ensemble with more and more decision trees.[6] Specifically, *gradient boosting* is a very powerful nonlinear technique and is one of the best "off-the-shelf" machine learning models.[7] They train relatively quickly, they can pick up on fairly complicated nonlinear interactions, you can guard against overfitting by increasing the shrinkage parameter, and their performance is difficult to beat.

However, they're a little more complicated than random forests; there are more hyperparameters to tune, and it's much more difficult to parallelize gradient boosted trees.[8]

Intuitively, one can think of boosting as iteratively improving a regression tree ensemble by repeatedly training a new regression tree on the *residuals* of the ensemble (when making predictions on the dataset) and then incorporating that regression tree into the ensemble.

[5] The `ranger` method is a faster *and* parallelized implementation of the random forests algorithm. One can set `method="rf"` for the standard (and slower) version.

[6] The first implementation of boosting – or at least the most famous – is AdaBoost, which can be considered to be like a special case of gradient boosting, a more modern form of boosting.

[7] See Ben Kuhn's comments on gradient boosting.

[8] See StackExchange for a brief overview of tuning `gbm()` hyperparameters.

Gradient boosted trees are implemented in R's `gbm` package as the `gbm()` function. They're also compatible with `caret`'s `train()` – just set `method="gbm"`.

- Use `train()` to perform a *grid search* to optimize the hyperparameters for a gradient boosted tree model (predicting white wine quality from chemical properties).
  - Instead of passing in the `tuneLength` parameter like earlier, use `expand.grid()` to create a grid with `n.trees` set to 500, `shrinkage` set to `10^seq(-3, 0, 1)`, `interaction.depth` set to `1:3`, and `n.minobsinnode` set to `seq(10, 50, 10)`.
- With the optimal values of the hyperparameters determined with `train()`, run `train()` again and tune only the value of `n.tree`, trying values from 500 to 5000 in steps of 5000. Compare the minimum RMSE to previously obtained RMSEs for other models.

Cubist

Cubist is a nonlinear *regression* algorithm developed by Ross Quinlan with a proprietary implementation. (The single-threaded code is open source and has been ported to R

In practice, Cubist performs approximately as well as a gradient boosted tree (as far as predictive power is concerned).[9] Having only two hyperparameters to tune, Cubist is a little simpler to use, and the hyperparameters themselves are very easily interpretable.

Broadly speaking, Cubist works by creating a *tree of linear models*, where the final linear models are *smoothed* by the intermediate models earlier in the tree. It's usually referred to as a *rule-based model*.

- Cubist incorporates a *boosting-like scheme* of iterative model improvement where the residuals of the ensemble model are taken into account when training a new tree. Cubist calls its trees *committees*, and the number of committees is a hyperparameter which must be tuned.
- Cubist can also adjust its final predictions using a more complex version of k-NN. When Cubist is finished building a rule-based model, Cubist can make predictions on the training set; subsequently, when trying to make a prediction for a new point, it can incorporate the predictions of the K nearest points in the training set into the new prediction.

As such, there are two hyperparameters to tune, called `committees` and `neighbors`. `committees` is the number of boosting iterations, and the functionality of `neighbors` is easily intuitively understandable as a more complex

[9] In Subpixel Urban Land Cover Estimation by Walton (2008), Cubist, random forests, and support vector regression are compared for a prediction task, and Cubist is found to be superior to gradient boosted trees. A comment on a Ben Kuhn post reports the same result.

version of k-NN. The Cubist algorithm is available as `cubist()` in the `Cubist` package and can be used with `train()` by setting `method="cubist"`.

- Use `caret`'s `train()` to fit a Cubist model for white wine quality. Use a grid search to find the optimal hyperparameter combination, searching over `committees=seq(10, 30, 5)` and `neighbors=0:9`.
- Compare the RMSE of the best Cubist fit with previously obtained RMSEs, particularly the RMSE corresponding to a gradient boosted tree.

Note that Cubist can only be used for *regression*, not for *classification*. Quinlan also developed the C5.0 algorithm, which is for classification instead of regression.

Stacking

Stacking is a technique in which multiple different learning algorithms are trained and then *combined* together into an ensemble.[10] The final 'stack' is very computationally expensive to compute, but usually performs better than any of the individual models used to create it.

Ensemble stacking using a `caret`-based interface is implemented in the `caretEnsemble` package. We'll start off by illustrating how to combine (1) MARS, (2) K-Nearest Neighbors, and (3) regression trees into a stack.

We'll first have to specify which methods we're using and the control parameters:

```
ensemble_methods = c('glmnet', 'kknn', 'rpart')
ensemble_control = trainControl(method="repeatedcv", repeats=1,
                                number=3, verboseIter=TRUE,
                                savePredictions="final")
```

Next, we have to specify the tuning parameters for all three methods:

```
ensemble_tunes = list(
  glmnet=caretModelSpec(method='glmnet', tuneLength=10),
  kknn=caretModelSpec(method='kknn', tuneLength=10),
  rpart=caretModelSpec(method='rpart', tuneLength=10)
)
```

We then create a list of `train()` fits using the `caretList()` function:

```
ensemble_fits = caretList(quality ~ ., df_whitewine,
                          trControl=ensemble_control,
                          methodList=ensemble_methods,
                          tuneList=ensemble_tunes)
```

[10] The canonical paper on stacking is Stacked Generalization by Wolpert (1992).

Finally, we can find the best *linear combination* of our many models by calling `caretEnsemble()` on our list of models:

```
fit_ensemble = caretEnsemble(ensemble_fits)
print(fit_ensemble)
summary(fit_ensemble)
```

By combining three simple methods, we've managed to get a cross-validated RMSE lower than the RMSE for any of the three individual models!

- How much lower does the RMSE get if you add in gradient boosted trees to the ensemble model? (The `caretModelSpec()` function can take a `tuneGrid` parameter instead of `tuneLength`.)

In the `caretEnsemble` documentation, read about how to use `caretStack()` to make a more sophisticated *nonlinear ensemble* from `ensemble_fits`.

- If you use a gradient boosted tree for `caretStack()`, is it any better than the simple linear combination?
- If you use all of the techniques you've just learned about in a big ensemble, how low can the RMSE get? (This might take a lot of computation time, so it's **optional**, but is fun to look at.)

Closing notes

By now, you've tried a fairly wide variety of nonlinear fitting techniques and gotten some sense for how each of them works. *In practice*, people usually use tree-based methods, especially random forests and gradient boosted trees – they tend to be fairly easily tuned and robust to overfitting. However, it's useful to have a broader overview of the field as a whole.

Also, there are a lot of peculiarities to the interfaces of different nonlinear techniques – when comparing them, `caret` offers a very well-designed interface for all of them, so it's nice to stick to using `train()` and other `caret` methods when possible.

Hyperparameter optimization

You may have noticed that tuning hyperparameters is a very big part of fitting nonlinear methods well! As the techniques become more complex, the number of hyperparameters to tune can grow significantly. Grid search is fine for ordinary usage, but in very complicated situations (10-20+ hyperparameters) it's better to

use random search – otherwise there would just be far too many hyperparameter combinations to evaluate![11]

- Read the first 4 paragraphs of the `caret` package's documentation on random hyperparameter search.

The `caret` package is very well-designed, and grid search will usually suffice for your purposes, especially because of its internal optimizations. It's good to be aware that alternatives to grid search exist.

Which model to use?

When trying to do predictive regression modeling, it's usually advised to start out with random forests or gradient boosted trees because they're fairly well understood and perform well out of the box with fairly straightforward tuning.[12] Random forests are simpler than gradient boosted trees, but both are much simpler than, say, a deeps neural net.

For fast parallelized gradient boosted trees in R, use the `xgboost` package – it's currently the state of the art. For random trees, the currently best implementation can be used by setting `method="parRF"` in `caret`'s `train()`, which is a parallelized combination of the `randomForest`, `e1071`, and `foreach` packages.

In the future, you'll learn about more complex nonlinear regression techniques, which can either be used on their own or be combined with the techniques you've already learned in a larger ensemble. However, defaulting to either random forests or gradient boosted trees works quite well in practice if you want to get a sense of how much predictive improvement you can get from using a nonlinear method.

[11] If you have, say, 15 hyperparameters, even the simplest possible grid search that selects one of two possible values for each hyperparameter still has 2^{15} configurations to iterate over. That will almost assuredly take far too long.

[12] One of the only good comparison of nonlinear regression techniques is in BART: Bayesian Additive Regression Trees by Chipman *et al.* (2010), which gives the following ordering (from better to worse): BART, 1-layer neural nets, gradient boosted trees, random forests. Cubist isn't used very much, mostly because almost nobody really knows what it does, even if its results are pretty good in practice. See also Performance Analysis of Some Machine Learning Algorithms for Regression Under Varying Spatial Autocorrelation by Santibanez *et al.* (2015).