Nonlinear Techniques

Today, you'll look at nonlinear regression techniques with a wine quality dataset dataset, which pairs up chemical characteristics of wines with their quality ratings. The dataset is split into a red wine dataset and a white wine dataset. You'll mainly be looking at the white wine dataset, which has three times as much data as the red wine dataset and so has more fine-grained nonlinear structure. Our goal will be to use the chemical properties of white wines to predict their associated quality ratings.

You'll also be using the `caret` package to easily get cross-validated estimates of RMSE as well as to easily tune the parameters of these nonlinear models. Since there aren't very many predictors relative to the number of rows in the data, we can use 3-fold cross-validation for simplicity, with:

```
control = trainControl(method="repeatedcv", repeats=1, number=3,
                       verboseIter=TRUE)
caret_fit = train(..., trControl=control)
```

Getting started

The data can be downloaded on the UCI Datasets page.

- Load the data for both the red and white wines. (If you use `read_delim()` for the red wine CSV, it might output a couple warning messages; just remove the two rows with `NA`s in that case.)
- The column names in both data frames have spaces in them, which doesn't work well with formulas. Write a utility function that modifies each string in a character vector, replacing spaces (`' '`) with underscores (`'_'`). (You may find `gsub()` helpful.) Use this to modify the column names of both datasets.

Next, before we do any model fitting, it's always a good idea to visualize the data.

- For each chemical property `p` in the white wine dataset, plot `p` on the x-axis and plot wine quality on the y-axis. On each plot, overlay both a

scatterplot of the individual data points and a smooth fit of the data. Note any evidence of nonlinearity.

A simple linear model

Before using nonlinear methods to predict white wine quality, let's use regularized linear regression to get some sort of baseline for comparison.

- Use `caret` with `train(..., method="glmnet")` to get an estimate for how low you can get the cross validated RMSE to get with just regularized linear regression.
  - For simplicity, instead of passing in a grid of values, you can just pass in `tuneLength=10` to `train()`, which makes it automatically generate a grid of hyperparameters. (This is fine for `glmnet`, but may not work so well for the hyperparameters of more complex nonlinear methods.)
- Examine the coefficients associated with the best linear fit and interpret the results. Based on the graphs you viewed earlier, which nonlinear relationships (between wine quality and chemical properties) are the regularized linear model not successfully modeling?

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. That's all!

KNN is implemented in R as `kknn()` in the `kknn` package. It can be used with `caret`'s `train()` by setting `method="kknn"`. There's just a single hyperparameter to tune – the value of k. A larger value of k helps guard against overfitting, but will make the model less sensitive to fine-grained structure in the data.

- Use `caret` to train a KNN model for white wine quality using `tuneLength=10`. Compare the minimum RMSE obtained to the RMSE for a regularized linear model.

In general, we can get better predictions by using information about what happens at a greater distance from the point of interest.

Regression tree models

We'll proceed to explore three different types of nonlinear techniques: regression trees, random forests, and gradient boosted trees. All three of these can be used for both regression and classification.

Regression trees are the simplest and very easily interpretable, but their performance is often poor and they tend to overfit. We can train an *ensemble* of regression trees and combine them together into a *random forest*, or we can keep training regression trees in an iterative manner to keep improving a single model in a technique known as *boosting*.

Using regression trees

You'll need the `rpart()` function in the `rpart` package to construct regression tree models. It's used in the same way as `lm()`, both in how the model is constructed and in how predictions are made using the model.

- For both red and white wines, create regression tree models predicting wine quality with the other features in the dataset. View them (by just calling `print()` on the models) and interpret the differences between the two models.

There is a single hyperparameter involved in the fitting of a regression tree: the *complexity parameter*, usually denoted `cp`. (It defaults to `0.01` if not explicitly specified in the call to `rpart()`.) As we grow a regression tree, we only make another 'split' in the tree if the associated incremental increase in the overall R-squared is greater than the value of `cp`. A higher value of `cp` helps us guard against overfitting to the data, but it can also stop us from growing the tree to a sufficient depth.

As before, we can use `caret`'s `train()` to test different values of `cp`. Since there's only a single hyperparameter to optimize, we can again use the `tuneLength=10` parameter.

- Use the `caret` package to fit a regression tree for the prediction of white wine quality with its chemical characteristics. Compare the RMSE value for the best fit with the RMSE from regularized linear regression and KNN.

Using random forests

Next, you'll be using `randomForest()` from the `randomForest` package, which is a more sophisticated nonlinear regression and classification technique.

Theoretical overview

In short, a *random forest* trains a lot of different regression trees and averages them together, with these two conditions on the regression trees:

1. Each regression is trained on a subset of the original data, sampled *with replacement*. 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 (usually $\sqrt{p}$ candidate predictors for p total predictors). This prevents very strong predictors from dominating certain splits and thereby *decorrelates* the regression trees from each other. The size of this random subset, denoted as `mtry`, is the sole hyperparameter needed to fit a random forest model.

As a bonus, we can fit each data point in the training data to the trees that *weren't* trained on that data point (and average the subsequent predictions) to obtain an *out-of-bag error*, which is an estimate for the generalizable error of our model. With this, we don't really have to use cross-validation to estimate the generalizable error of our model.

- Read Edwin Chen's Quora answer on how random forests work.

Random forests in R

In general, the `randomForest()` function is used in a manner analogous to `rpart()` and `lm()`. There are two details to pay attention to:

First, it's important to pay some attention to the choice of the `mtry` hyperparameter. It's usually advised to try either `mtry = floor(sqrt(p))` or `mtry = floor(p/3)` (for a dataset with `p` predictors); the former should be used when `p/3` rounds to 1-2 or when we don't have very many predictors relative to the number of data points ($p \ll n$), and the latter should be used otherwise. Also, it's *always* wise to try `mtry = p`.

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)`.

Anyway, let's get some practice with random forests:

- With the white wine dataset, fit two random forest models for wine quality as a function of the other variables, setting `mtry = floor(sqrt(p))` and `mtry = p`.
- Make out-of-bag predictions with both of the random forest models and calculate the associated RMSE values. Compare the RMSEs to previously obtained RMSEs.

Using gradient boosted trees

Gradient boosting is a very powerful nonlinear technique which is one of the best "off-the-shelf" machine learning models.[1] 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.[2]

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.

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()`, call `gbm()` on the data directly with 5000 trees instead of 500 and with `cv.folds=3`. (The `gbm()` algorithm will automatically use 3-fold cross-validation to estimate the test error.)

[1] See Ben Kuhn's comments on gradient boosting.
[2] See StackExchange for a brief overview of tuning `gbm()` hyperparameters.

- The `$cv.error` variable of the `gbm()` fit is a vector of cross-validated RMSE estimates after each incremental improvement to the ensemble of regression trees. Find the minimum cross-validated RMSE and plot the cross-validated RMSEs as a function of number of trees added to the model. Compare the minimum RMSE to previously obtained RMSEs for other models.
- Determine the degree of overfitting by using `predict()` to generate predictions on the entire dataset and calculating the RMSE from those predictions. (Running `predict()` on a `gbm()` model will automatically default to using the optimal number of trees in the ensemble model as determined by the RMSE estimates in `$cv.error`.)