Nonlinear Techniques

We'll be continuing with the white wine dataset. In this assignment, we'll consider some less-common (but still important) nonlinear techniques. Finally, you'll use your new knowledge on a Kaggle competition!

Multivariate adaptive regression splines

Multivariate adaptive regression splines (MARS) is an extension of linear models that uses *hinge functions*. It models a target variable as being linear in functions of the form $\max(0, \pm(x_i - c))$ where x_i can be any of the predictors in the dataset.

- Look at the pictures on the Wikipedia page for MARS to get some intuition for how MARS works.

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

Intuitively, one can think of a degree 1 MARS model with p predictor variables as being a piecewise linear combination of hyperplanes – with 1 predictor variable you're pasting different lines together, with 2 predictor variables you're pasting planes together, and so on and so forth. Raising the degree then allows more complicated nonlinear interactions to show up.

MARS is implemented as `earth()` in the `earth` package and can be used with `train()` by setting `method="earth"`. It has two hyperparameters to tune, `degree` and `nprune`; the `nprune` parameter is the maximum number of additive terms allowed in the final model (so it controls model complexity).

- Use `caret`'s `train()` to fit a MARS model for white wine quality. Use a grid search to find the optimal hyperparameters, trying `degree=1:5` and `nprune=10:20`.
- Compare the RMSE of the optimal MARS model with the previously obtained RMSEs for white wine quality.

- Pass the optimal hyperparameters into `earth()` directly and examine the resulting model (with `print()` and `summary()`). Interpret the results, comparing the model with the output of a simple regression tree model for white wine quality.

Although MARS doesn't usually give results as good as those of a random forest or a boosted tree, MARS models are easy to fit and interpret while being more flexible than just a simple linear regression. Degree 1 models can also be built *very* rapidly even for large datasets.

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 boosted tree (as far as predictive power is concerned). 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 KNN. 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 version of KNN. 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. 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)
```

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!

- 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.

Kaggle Bike Sharing Demand competition

Try the nonlinear techniques you've just learned on the Kaggle Bike Sharing Demand competition. In addition to applying various nonlinear modeling techniques, you may find it helpful to explicitly engineer new features from the existing ones, *e.g.*, adding well-chosen indicator variables, or to remove highly collinear variables.