# 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.
- Use `caret`'s `train()` function with `method="rf"` to tune over `mtry=c(3, 5, 7, 11)`. Compare the minimum cross-validated RMSE with the out-of-bag RMSE estimate.

## 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

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

`cv.folds=3`. (The `gbm()` algorithm will automatically use 3-fold cross-validation to estimate the test error.) Compare the minimum RMSE to previously obtained RMSEs for other models.

## 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.