Project 1: Boston Housing Prices

Udacity: Machine Learning Nanodegree
Hamilton Hoxie Ackerman
hoxiea@gmail.com
12/21/2015

Introduction

In this project, we play the role of a real estate agent who's interested in predicting the value of a house in the Boston area, given various properties of the house, geographic location, and neighborhood. Since the target (selling price) is quantitative, this is a regression problem.

The dataset analyzed is a well-known dataset in the machine learning community [1]. This particular instance of the dataset comes from the `scikit-learn datasets` repository.

Section 1: Statistical Analysis and Data Exploration

In our real estate dataset, we have 506 observations (houses). Each observation has 13 features; these predictive features are given in Table 1. Note that most features are quantitative but that at least one (CHAS) is categorical.

Feature	**Description**
CRIM	per capita crime rate by town
ZN	proportion of residential land zoned for lots over 25,000 sq.ft.
INDUS	proportion of non-retail business acres per town
CHAS	Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
NOX	nitric oxides concentration (parts per 10 million)
RM	average number of rooms per dwelling
AGE	proportion of owner-occupied units built prior to 1940
DIS	weighted distances to five Boston employment centres
RAD	index of accessibility to radial highways
TAX	full-value property-tax rate per $10,000
PTRATIO	pupil-teacher ratio by town
B	1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
LSTAT	% lower status of the population

Table 1: A summary of available features.

The 14th variable, MEDV (the median value of owner-occupied homes), is the target of the prediction problem. Table 2 contains some basic summary statistics for the target variable. Figure 1 shows visually how target values are distributed across the given range: home values appear to have a bimodal distribution, with the majority of home values in the range (10, 30) but with a non-trivial number of high-value homes in the range (45, 50).

Range	(5, 50)
Mean	22.533
Median	21.2
SD	9.188

Table 2: Summary of the target variable, MEDV.

Section 2: Evaluating Model Performance

The model constructed to predict MEDV is a Decision Tree Regression model. Since this is a regression problem, the metric that we'll use to evaluate model performance will be some function of the residuals, to be minimized. Common options include mean absolute error and mean squared error (MSE). In this case, it seemed like punishing larger residuals more strongly was a reasonable penalization strategy, so I opted to use the `scikit-learn` implementation of MSE as my performance metric.

It's often useful in machine learning to split the data into subsections for various aspects of the modeling process. The main issue that we're trying to avoid when we make these splits is the fact that **evaluating a model's performance on data that the model was trained on will lead to an optimistically biased estimate of model performance.**

For example, if we set no data aside and started training models of varying complexity on the entire dataset, we would find that:

1. More complex models have lower training/in-sample error rates, since more complex models are better able to adapt to variations in the data. And because we're using all of our data, this in-sample error rate is the only feedback available to us. If we decide to minimize this error rate, which is the only thing that makes sense to do, then we would just choose the most complex model, even though overfitting could be a real possibility.
2. We have no way to estimate how any of these models will perform on data that the model has never seen before, since all available data was used to train the model. Estimates of how our model generalizes to unseen data are important if we're going to be confident using our model to predict values for new observations.

To overcome these issues, we would ideally split our data into three parts [2]:

- Data on which models can be trained, i.e. *training data*. Since the models will clearly have seen this data, we can't rely on error estimates from this dataset for anything.
- Data on which trained models of varying complexity can be assessed, i.e. *validation data*. We should observe that increasing model complexity initially yields better performance in the validation data as more structure in the training data is captured, but that as increasingly complex models overfit the training data, the validation error rate rises. This leads to classic model complexity diagrams such as Figure 2, and enables us to make intelligent decisions about the appropriate model complexity for the problem. (Addresses Issue #1 above. More about Figure 2 later.)
- Data locked away until the final model has been chosen. The performance of the final model on this *test data* should be a good estimate of how the final model will generalize to unseen observations. (Addresses Issue #2 above.)

Though this is the ideal situation, it relies on having sufficient data for each of the three parts: enough training data to fully exhibit the structure in the data, enough validation data to get good feedback concerning model complexity, and enough test data to see and assess model performance on a reasonable range of new inputs.

In practice, we don't always have sufficient data to follow this procedure; if you're not absolutely drowning in data, then it can be hard to justify just ignoring 20% of your data while you're trying to achieve the best fit possible. If your data is limited, then it becomes important to learn as much as you can from the data you do have.

Before we discuss efficient uses of available data, we should note that model complexity is typically specified by one or more *hyperparameters*. (Once you've decided on a set of hyperparameters, you can use your performance metric and an optimization algorithm to find the best model parameters for the model dictated by your choice of hyperparameters; thus, the hyperparameters are a "level above" the model parameters.)

In some cases, hyperparameters are discrete quantities that can be individually considered. For example, in Decision Tree Regression, the one hyperparameter that controls model complexity is the maximum depth of the decision tree: greater depths correspond to greater complexity. In other cases, hyperparameters are continuous quantities, so it's less clear which specific values should actually be considered as we search for the best model. And often, models have multiple hyperparameters, so combinations must be considered.

Various hyperparameter combinations are typically explored via a **grid search**: given an m-dimensional hyperparameter space, we establish a grid of (typically equally spaced) points in the space and then evaluate model performance at each point in the grid. The combination of optimal hyperparameters specifies the complexity of our final model.

So that's the goal of the validation process: to find the combination of hyperparameters that yields the model with the best fit to the validation data. But like we said previously, we'd like to maximize the amount of available data that we use when fitting and validating our models at each point in the grid.

To use our data as efficiently as possible while searching for the optimal set of hyperparameters, we often use *cross-validation*: instead of setting aside a dedicated validation data set, we split our data into testing data (still set aside until the end, to estimate generalizability) and non-testing data. (If you're not interested in estimating generalizability, then all our your data can be non-testing data.) We then algorithmically split the non-testing data into different partitions of training and validation data, such that, across the partitions, all observations are used for training and for validation at some point.

The most common form of cross-validation is *k-fold cross-validation:* the non-testing data is divided into k folds, and each fold in turn becomes the validation dataset (with the other k-1 folds constituting the training data). For each of these (k-1, 1) partitions, we fit models of varying complexity (different hyperparameter values) and find the complexity that minimizes the validation error rate, as before.

This entire process produces *k* different estimates of the best hyperparameter(s) to use, and these results are somehow combined or averaged into the winning model. We then fit that winning model on the entire non-testing dataset (thereby using as much data as possible for parameter estimation), and apply the fitted model to the testing data to estimate out-of-sample generalization.

Section 3: Analyzing Model Performance

Learning curves let us visually examine the effects of a different aspect of modeling: the effect of increased training data on model performance.

How are training data size and model performance related? Well, if the relationship between target and predictors is fairly simple, then it probably won't require much training data before the relationship has been revealed. And using more data won't actually allow you to do any better than you already were, since you've already seen what there was to see. On the other hand, if the relationship is relatively complex, then more data will be required to establish the structure that can then be learned. (In both cases, you eventually reach the point at which more data won't help; it's just a question of whether this limit can be feasibly reached in practice or not, and whether your model is complex enough to capture all structure that emerges.)

Said another way, if you have relatively little training data, then you should probably restrict your attention to relatively simple models. If you're drowning in training data, then more complex models can be considered and potentially fit well.

Of course, it's all relative when it comes to "less data versus more data" and "simple versus complex models." Visually, we can judge the complexity of our model relative to the amount of data we have using learning curves. The idea behind a learning curve is to, for a fixed model complexity, examine the in-sample (training) and out-of-sample (test) error rates as you use increasing amounts of your available data for training.

We expect to see, for very small amounts of training data:

- A low in-sample error rate, since even relatively simple models can often capture the structure of just a few points. (For example, a line can perfectly capture 2 points.)
- A high out-of-sample error rate, since those few points were unlikely to represent all the structure in the data.

As we increase the amount of training data we use, still with a fixed model complexity,

- the in-sample error rate should rise, since we stop being able to capture all present structure and therefore start making mistakes. (For example, unless the points are all perfectly collinear, a line will have errors associated with 3 or more points.)
- the out-of-sample error rate should fall, since more training data means that we're better capturing the structure in the data, and we can therefore generalize to unseen data better.

With enough training data, in-sample and out-of-sample error rates should eventually "meet in the middle." And the error rate that they converge to is an estimate of the amount of uncapturable variability present in the data.

In the learning curves that result from fitting models with depths (1, ..., 10) to increasing amounts of housing data, we see exactly these patterns emerge. Zooming in on the depth=1 and depth=10 learning curves (Figures 3 and 4), we see that training error starts at 0 in both cases, though it rises quite high in the d=1 case and remains negligible in the d=10 case. In both cases, the test error drops initially, though it plateaus after about 100 observations at a relatively high value in the d=1 case, suggesting that this simple model has reached the limits of what it's able to capture about the data, i.e. we have a high-bias model on our hands. In the d=10 case, on the other hand, the test error continues to drop as larger samples are considered and levels off at a much lower error rate than the d=1 test error did.

It seems like a learning curve for which the training error rate remains negligible across all sample sizes considered should be investigated closely for overfitting, since this model is clearly fitting the contours of the training data quite closely. Indeed, re-examining the model complexity graph (Figure 2) reveals that the training error rate reaches 0 right around depth=10, but that validation error has long since started to rise again as the overfit models struggles with data it hasn't seen before. Validation error appears to be minimized for d=5 or d=6, so the value that we use in our final model will likely be close to 5.

Section 4: Model Prediction

After all this work exploring hyperparameters, varying training sizes, and fitting parameters, it would be nice to be able to make some predictions: given data about a new house, for what price do we expect to be able to sell the house?

To do this, we could simply perform a grid search across a range of feasible max-depth values, using the gridsearch default of 3-fold cross-validation for validation purposes. We would then fit the winning model (in terms of complexity) to our data, and then feed the model the information we have about the house.

In practice, though, the randomness of cross-validation (and the computational feasibility that accompanies our relatively small dataset) implies that it would be a good idea to repeat this procedure a few times to understand how the winning parameter value changes with various resamplings. I performed this procedure 200 times, and the resulting distribution of optimal depth values can be seen in Figure 5. A depth value of 4 was the clear winner, though other parameter values were best a non-trivial amount of the time.

After fitting a Decision Tree Regression model with **max_depth=4** to the entire data, the resulting model predicted a median selling price of **21.630**. This predicted value certainly lies within the range of feasible target values, and indeed, re-examining Figure 1, this neighborhood of target values seems to be quite common in this dataset. This prediction therefore seems quite reasonable in the context of the dataset, and the model seems valid.

Bibliography

[1]: "Housing Data Set." *UCI Machine Learning Repository*. UC Irvine. Web. 15 Dec. 2015. <https://archive.ics.uci.edu/ml/datasets/Housing>.
[2]: Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. "Model Assessment and Selection." *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd ed. 222.

Supplementary Material

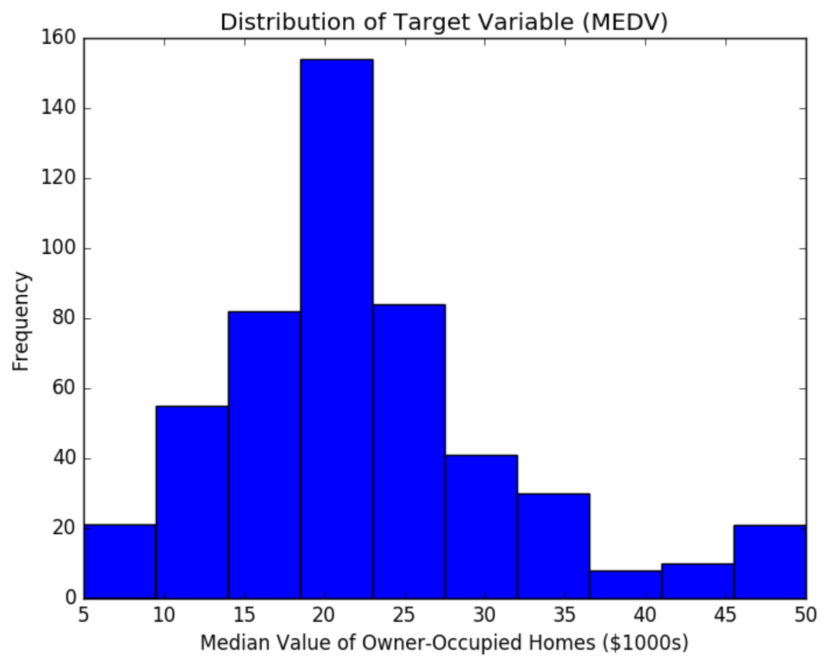

Figure 1: The distribution of the target variable, MEDV.

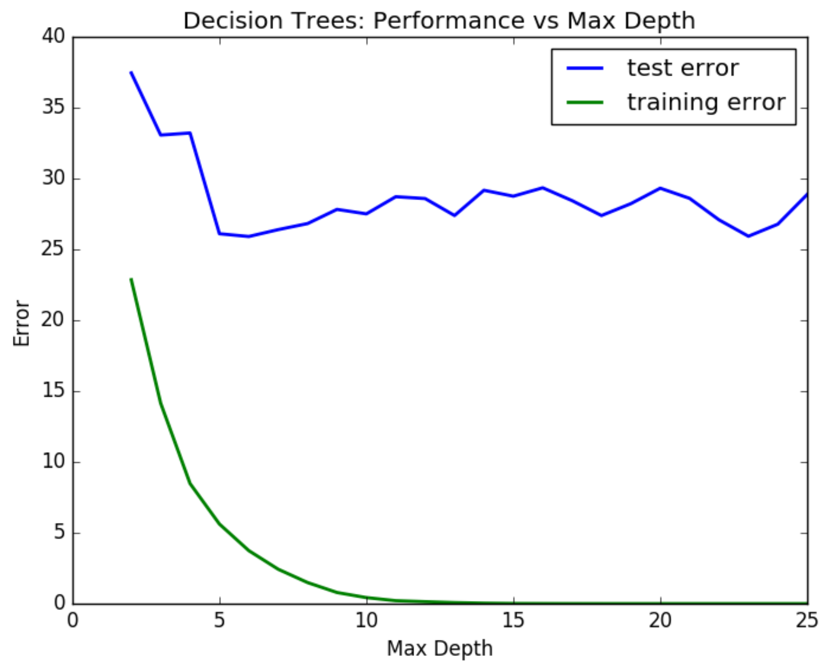

Figure 2: The training and test error achieved by fitting increasingly complex Decision Tree Regressions to our data. A more complex model always does better in the training data, but error in the test data starts to rise around d=5 or 6.

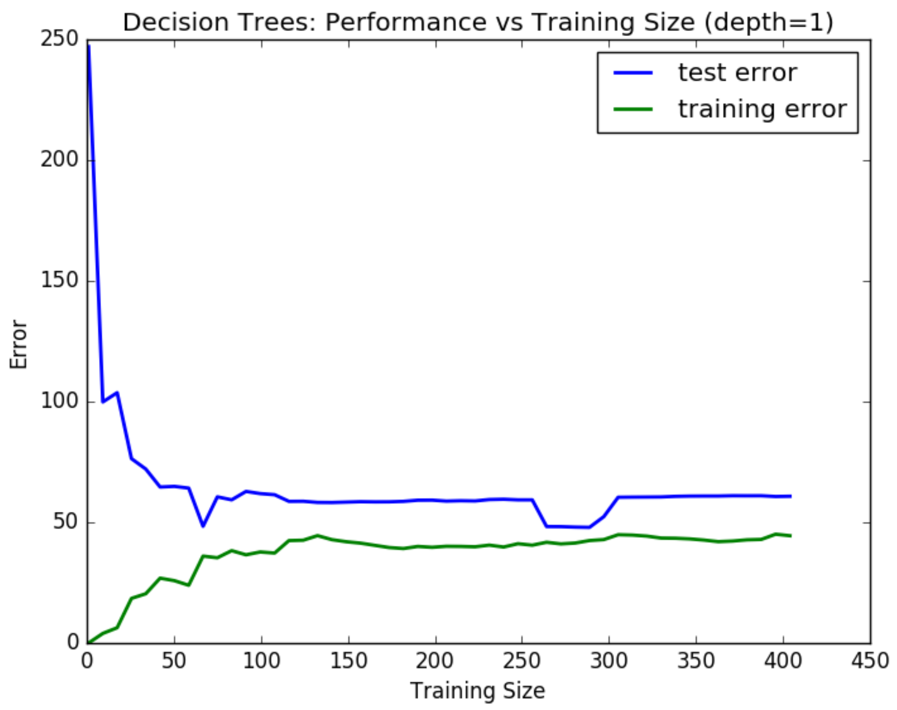

Figure 3: Learning curve for d=1. This relatively simple model quickly starts making sizeable errors in the training data. It also quickly reaches a point (n=~100) where more data actually doesn't improve the model (error curves flatten out).

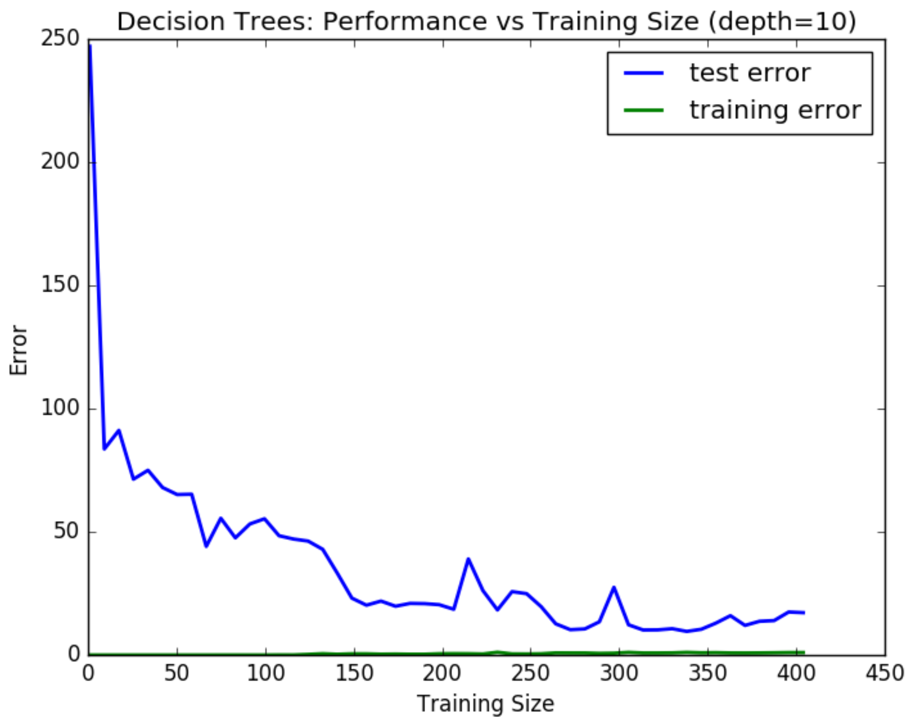

Figure 4: Learning curve for d=10. This complex model is able to keep training error negligible for all datasets examined. Test error also drops to a much lower level than it did in the d=1 case (Figure 3); it's hard to say if this has leveled off, or if more data would cause it to drop even further.

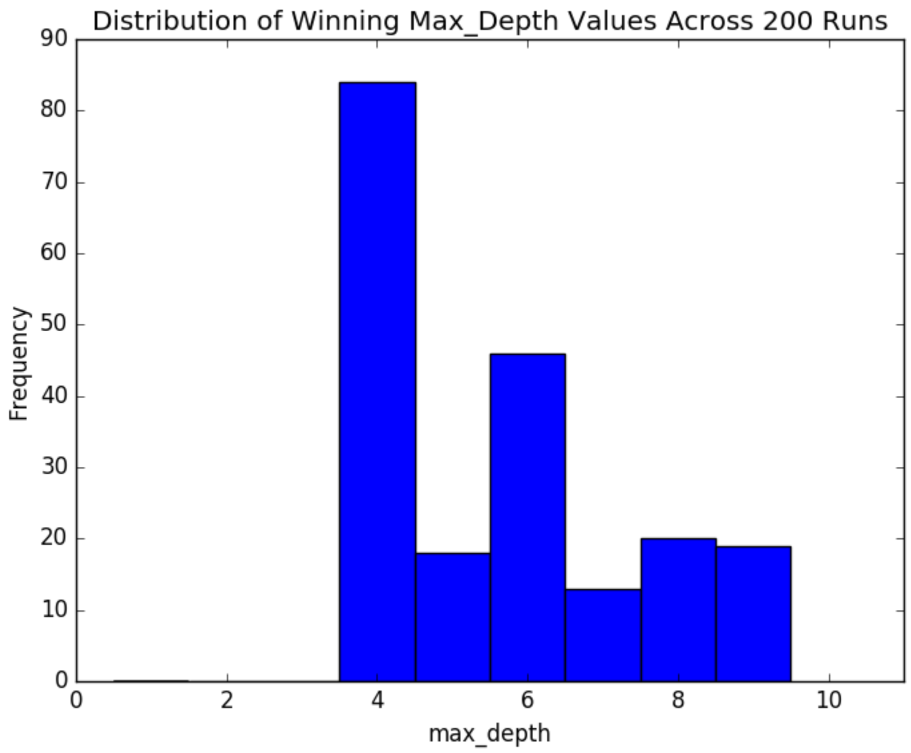

Figure 5: The results of 200 3-fold cross-validated searches for the best max_depth value. By far the most commonly optimal value was max_depth=4, though higher values were seen a non-trivial number of times; max_depth=6 was the winner in approximately 25% of runs.