**Essential Notes**

Page 30:

The sweet spot in model fitting is where the “generalization curve” is at its peak. It is the classifier/learning algorithm set up that is considered “just right” that is able to achieve this sweet spot. However, this classifier will have less accuracy on the training data than an overfitted one – and that’s completely desired.



The correct curve has generalization at its peak and training data accuracy at something relatively high, but not *too* high.

Page 31:

The more data points you have, the more complex of a model you can build without overfitting. Yes, it is possible you will have many features but the more data points you have the more chances of less overfitting because of the variety that will be in your dataset.

It is almost always that collecting more datapoints will yield more variety, but simply duplicating the same data (….. -\_-) or collecting suspiciously similar data points will NOT help and thus will contribute to overfitting.

If we added 10,000 more rows of customer data to the table on Page 30, and it was evident that the rule of “customer > 45 and less than 3 children and not divorced” was still there, then we would take this rule with much more open arms.

Page 33:

“low-dimensional” datasets are those with few features (akin to how many dimensions we would need to plot the dataset). Intuitions (think: trends and patterns) we see from datasets with few features *might not* hold in datasets with many features (high-dimensional).

Page 35:

In the Boston Dataset, we’ll also look at “interactions” of the dataset. The interactions of a dataset are the relationship between features. Essentially, we will add features to the dataset which are products of one another. The interaction of closeness to the river and crime rate is simply a vector represented by X[:, “CRIME\_RATE”] \* X[:, “CLOSENESS\_TO\_RIVER”]. This new vector would be appended to the end of the dataset. This is called **feature engineering**.

The new shape is (506, 104) because we have added the product of every two distinct feature to the dataset. This is because for every feature we decrease in 1 distinct combination we can make with it

13 (original) + 12 (feature 1 & other 12) + 11 (feature 2 & other 11) + … 1 = **104**

Page 39:

Single neighbor in k-nearest Neighbors corresponds to a more *complex* model (inverse of how many neighbors there are == complexity). By looking at one neighbor, we’re fixating and making decision on just a single data point. As we increased our classifier to accommodate 3 neighbors, and ultimately 9, we notice that our model is actually becoming more *simple*. The decision boundary that separates both of the classes in this binary classification problem actually becomes smoother. On the other extreme, if all of the training data points are considered neighbors, then we run into the problem that our classifier is **completely deterministic**. A prediction on a k-nearest neighbors classifier that has n\_neighbors set to the size of the training set would always lead to the same prediction for any new test set feature set: the majority representative class in the training set would always be predicted.

Confirming relationship between model complexity and generalization is important, otherwise it would seem like this is not the case. Here is a graph that shows the relationship between the increase of neighbors and the accuracy. 

Page 41:

k-nearest neighbors works for both regression and classification. Use KNeighborsRegressor.

Page 44:

In essence, the independent variable is plotted, and then, for every possible y value at a given granularity, a closest point distance is set. You keep picking the closest distances for however many neighbors you have, averaging them to get the test data point’s response/prediction. Of course, for KNeighborsRegressor(n\_neighbors=1), you can only have the test point match the response of the training data point it is closest to.

knn.score(test\_set, real\_results) is actually called the **R2 score**, also known as the **coefficient of determination**. A value of 1 is a perfect prediction, 0 (or even negative!!!) is a prediction of the mean of the training set (all neighbors, an extreme!).

Key points about KNeighborsClassifier and KNeighborsRegressor:

The two parameters that matter to k-nn is the number of neighbors and the method of measuring distance between training and test points. Always adjust your neighbors to see what works, and you’ll see anywhere from 3 to 5 neighbors on most datasets will give you a good prediction(s). The way we measure distance is typically with Euclidean distance.

KNN Strengths:

* Easy to understand
* Reasonable performance with a lot of adjustments
* Usually fast to build the model

It’s good to use KNN as a baseline before considering other machine learning algorithms to apply to your dataset.

KNN Weaknesses:

* When training set is large in either the number of features or in the number of samples… or both, prediction is slow.

Before using KNN, preprocess your data. Preprocessing does not do justice with many features (high dimensional, 100s +) or where the dataset is sparse.

Page 48:

(Linear Models)

Using a linear model would seem to be very restrictive, and it would also make us feel that key details of the data are being lost. However, looking at one-dimensional data is giving us a skewed perspective. *When considering a dataset with many features, a linear function can perfectly model any target y. \*\* What?*

Page 49:

There are a variety of different linear models, and they differ in

1. how the weighted parameters, *w*, are learned, and.…
2. how they can be controlled in model complexity.

OLS == Linear Regression. OLS finds the values of w and b that minimize the **squared error** in between the hypothesis function predictions and the true results from the training set. In sklearn, there is no parameters in building LinearRegression, which also means it is not possible to control model complexity.

Page 50:

R^2

🡪 How close the data is to the fitted line.

Here, we notice bad accuracy even on the training set (along with the test set). That being said, we may be *underfitting* our regressor. **This is due to the fact that our training and test set are equally bad in accuracy.** In linear models with low features, it’s very hard to overfit, because that is caused by a high number of features, which leads to high variance amongst the parameters. However, when the number of features increases, we higher the chance of overfitting, and then need to take action to avoid it.

Page 51:

The discrepancy between the training set (very high.. 90%) and the test set (very low… 60%), should automatically tell us that we are overfitting. Because LinearRegression() in sklearn.linear\_model is a black box, in the sense that I can’t really edit the algorithm, I have no control over the model complexity. So, we need to use a regression technique or a new type of model that allows us to control complexity … effectively allowing us to make simpler models.

Introducing **ridge regression**, a linear model technique that allows us to fit a line through a high dimensional dataset and accommodate for overfitting. Ridge regression fits an additional constant to the resulting line that actually reduces the contribution and significance of each individual parameter, making it less likely to ovefit. We want the parameter values to be as small as possible (small slope), while still predicting well. This constraint is called *regularization*. The regularization in ridge regression is called **L2 regularization**.

The regularization parameter is what we care about in Ridge regression. Now, this parameter is what balances trade-off between model simplicity (artificial increase of parameters, but overall decrease), and accuracy against the training set. We didn’t choose Ridge for any other reason but the fact that it should give us more customizability. Indeed, it does. Specify the *alpha* parameter and you can control the regularization parameter.

The value of theta/alpha that is just right depends on the dataset. We can always try to fiddle around and see what works for us, but a good way of measuring to see if we are improving with increases/decreases in alpha is by checking the .coef\_ variable. The coefficients should be decreasing when we increase alpha, and should be increasing when we decrease alpha.

Page 53:



This diagram represents on the X axis all of the coefficients up to Θ100. We notice that those values of 10 lambda/alpha happen to have coefficients very much near 0, and on the opposite side, Linear Regression has its parameter magnitudes at much higher absolute Y values. Increasing the regularization parameter (Ridge(alpha)) leads to a model with parameters near 0 and are therefore regularized.

Left off on “Another way” on bottom of 53