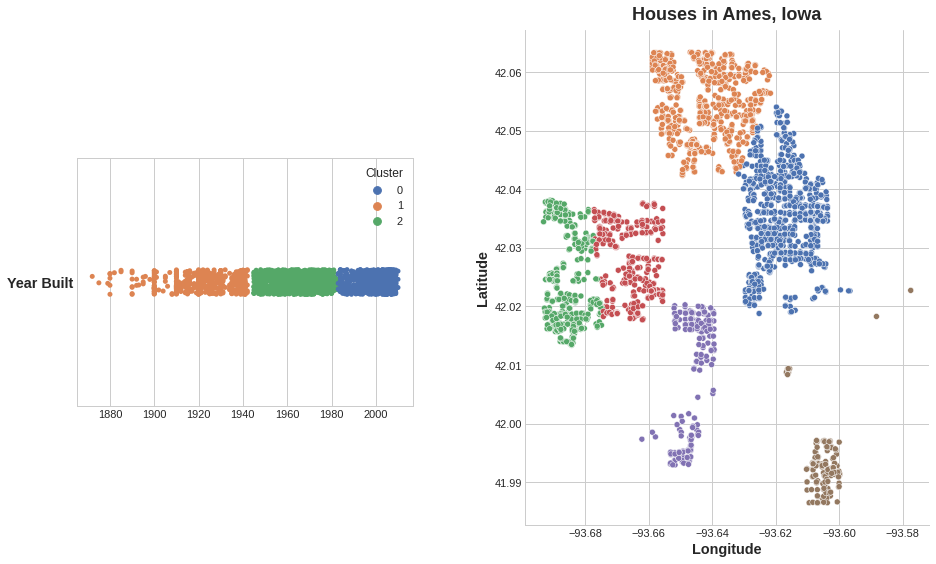
Feature Engineering: 4th lesson – Clustering With K-means

This lesson and the next make use of what are known as unsupervised learning algorithms. Unsupervised algorithms don't make use of a target; instead, their purpose is to learn some property of the data, to represent the structure of the features in a certain way. In the context of feature engineering for prediction, you could think of an unsupervised algorithm as a "feature discovery" technique.

Clustering simply means the assigning of data points to groups based upon how similar the points are to each other. A clustering algorithm makes "birds of a feather flock together," so to speak. When used for feature engineering, we could attempt to discover groups of customers representing a market segment, for instance, or geographic areas that share similar weather patterns. Adding a feature of cluster labels can help machine learning models untangle complicated relationships of space or proximity.

Cluster labels as a feature:

Applied to a single real-valued feature, clustering acts like a traditional "binning" or "discretization" transform. On multiple features, it's like "multi-dimensional binning" (sometimes called vector quantization).



Added to a dataframe, a feature of cluster labels might look like this:

Longitude Latitude Cluster

-93.619 42.054 3

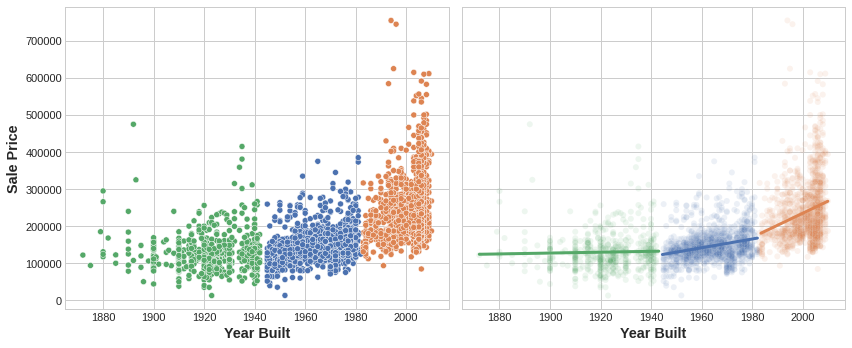
-93.619 42.053 3

-93.638 42.060 1

-93.602 41.988 0

It's important to remember that this Cluster feature is categorical. Here, it's shown with a label encoding (that is, as a sequence of integers) as a typical clustering algorithm would produce; depending on your model, a one-hot encoding may be more appropriate.

The motivating idea for adding cluster labels is that the clusters will break up complicated relationships across features into simpler chunks. Our model can then just learn the simpler chunks one-by-one instead having to learn the complicated whole all at once. It's a "divide and conquer" strategy.



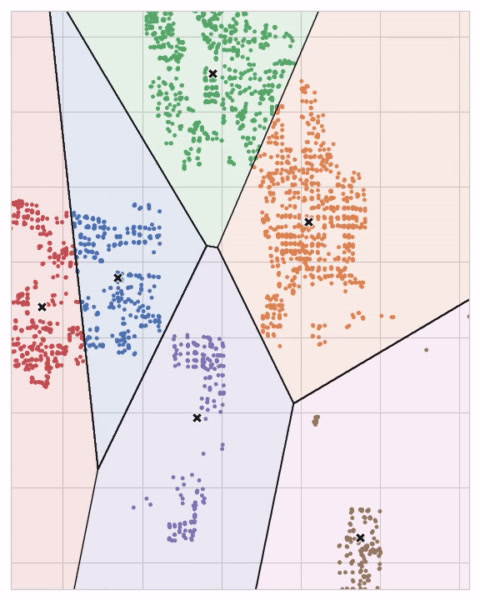
The figure shows how clustering can improve a simple linear model. The curved relationship between the YearBuilt and SalePrice is too complicated for this kind of model -- it underfits. On smaller chunks however the relationship is almost linear, and that the model can learn easily.

K-means clustering:

There are a great many clustering algorithms. They differ primarily in how they measure "similarity" or "proximity" and in what kinds of features they work with. The algorithm we'll use, k-means, is intuitive and easy to apply in a feature engineering context. Depending on your application another algorithm might be more appropriate.

***K-means clustering*** measures similarity using ordinary straight-line distance (Euclidean distance, in other words). It creates clusters by placing a number of points, called centroids, inside the feature-space. Each point in the dataset is assigned to the cluster of whichever centroid it's closest to. The "k" in "k-means" is how many centroids (that is, clusters) it creates. You define the k yourself. You could imagine each centroid capturing points through a sequence of radiating circles. When sets of circles from competing centroids overlap they form a line. The result is what's called a ***Voronoi tessallation***. The tessallation shows you to what clusters future data will be assigned; the tessallation is essentially what k-means learns from its training data.

The clustering on the *Ames* dataset above is a k-means clustering. Here is the same figure with the tessallation and centroids shown.



Let's review how the k-means algorithm learns the clusters and what that means for feature engineering. We'll focus on three parameters from scikit-learn's implementation: n\_clusters, max\_iter, and n\_init. It's a simple two-step process. The algorithm starts by randomly initializing some predefined number (n\_clusters) of centroids. It then iterates over these two operations:

* Assign points to the nearest cluster centroid
* Move each centroid to minimize the distance to its points

It iterates over these two steps until the centroids aren't moving anymore, or until some maximum number of iterations has passed (max\_iter). It often happens that the initial random position of the centroids ends in a poor clustering. For this reason the algorithm repeats a number of times (n\_init) and returns the clustering that has the least total distance between each point and its centroid, the optimal clustering.

You may need to increase the max\_iter for a large number of clusters or n\_init for a complex dataset. Ordinarily though the only parameter you'll need to choose yourself is n\_clusters (k, that is). The best partitioning for a set of features depends on the model you're using and what you're trying to predict, so it's best to tune it like any hyperparameter (through cross-validation, say).