Linear models

James Scott

ECO 395M: Data Mining and Statistical Learning

Linear models

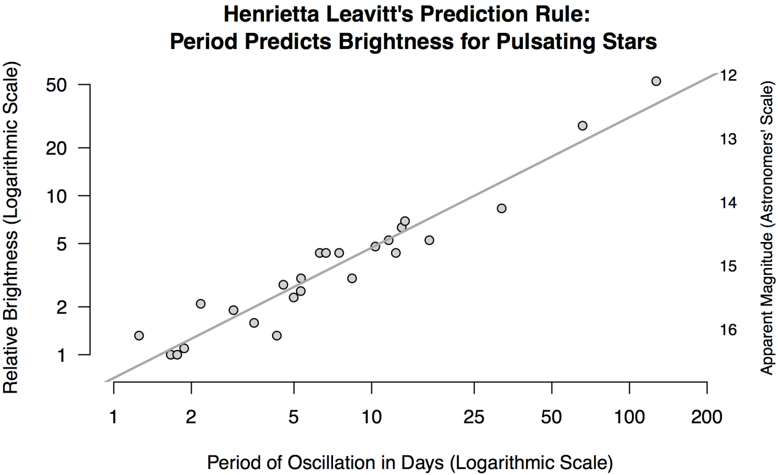
Linear modeling is the most widely used tool in the world for fitting a predictive model of the form

$$y = f(x) + e$$

Linear models, despite their apparent simplicity, are used throughout the worlds of science and industry.

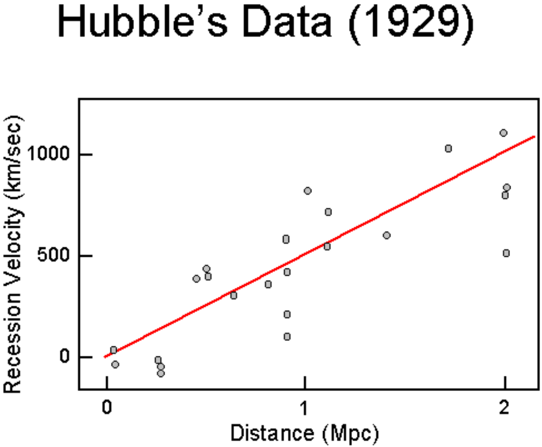
They have been at the heart of some of history's greatest scientific discoveries.

Linear models

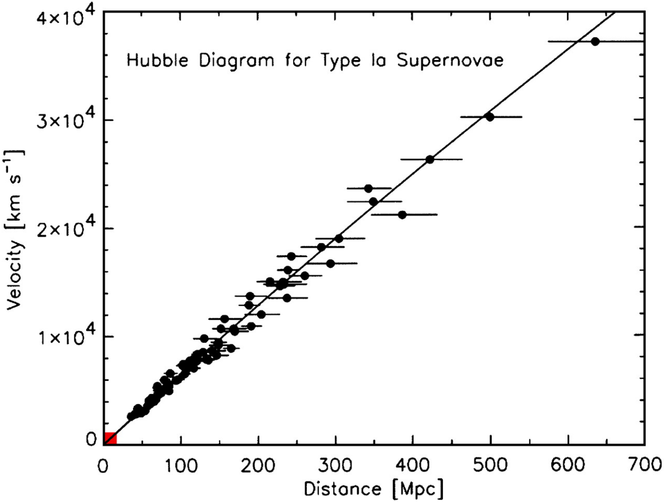


From *AIQ: How People and Machines are Smarter Together*

Linear models



Linear models



Linear models

A linear model is parametric model; we can can write down $f(x)$ in the form of an equation:

$$\begin{aligned} y &= \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + e \\ &= x \cdot \beta + e \end{aligned}$$

A notational convenience: the intercept gets absorbed into the vector of predictors by including a leading term of 1:

- $x = (1, x_1, x_2, \ldots, x_p)$
- $\beta = (\beta_0, \beta_1, \beta_2, \ldots, \beta_p)$

Linear models: pros

- They often work pretty well for prediction.
- They're a simple foundation for more sophisticated techniques.
- They have lower estimation variance than most nonlinear/nonparametric alternatives.
- They're easier to interpret than nonparametric models.
- Techniques for feature selection (choosing which x_j's matter) are very well developed for linear models.
- They're computationally easy to estimate.
    - This is true even for extremely large data sets.
    - This lends itself towards a fast exploration/development loop.

Linear models: cons

- Linear models are pretty much always wrong, sometimes subtly and sometimes spectacularly. If the truth is nonlinear, then the linear model will provide a biased estimate.
- Linear models sometimes require extensive fiddling in order to yield good performance. Three simple examples:
  - Linear models need to represent categorical features in terms of dummy variables (0/1 indicators).
  - Linear models depend on which *transformation* of the features and outcome you use, e.g. x versus $\log(x)$). Without theory to fall back on, you often have to try out different options.
  - Linear models don't handle *interactions* among the predictors unless you explicitly build them in.

Typical focus in stats/econ: inference

Linear models are the workhorse of classical statistics and econometrics. If you learned about linear models in one of those contexts, you might have learned about topics like:

- model assumptions
- confidence intervals and “robust” standard errors, e.g. if residuals are heteroskedastic and/or clustered.
- hypothesis tests for regression coefficients
- large-sample theory for regression
- causal identification using regression.

This is all important stuff. But our focus in this class is **very different**: it’s about getting linear models to work well for prediction and data exploration.

Our focus: feature engineering for linear models

Feature engineering is a very common step in data-science pipelines. It means applying domain-specific knowledge in order to:

1) extract useful features from raw data, and/or
2) improve the usefulness of existing features for predicting y.

Better features almost always beat a better model!

- Good feature engineering is absolutely vital to success in ML applications.
- Data scientists who work on ML pipelines spend a huge amount of their time thinking about how to engineer/extract more useful features from data sets.

Common steps in feature engineering

We'll see examples of several common feature engineering steps:

- Encode categorical variables as dummy variables ("one-hot encoding")
- Combine existing features to form new, derived features, including interactions.
- Merge existing data with new sources
- Apply nonlinear transformations (logs, powers, etc) to existing numerical features
- Extract useful variables from raw text fields (e.g. month, day of week, hour of day from time stamps)

Common steps in feature engineering

But there are many other common feature-engineering steps that we won't cover in detail:

- Discretize: convert continuous features into categorical/discrete features
- Simplify complex features (E.g. heart-rate trace every second –> max/min/average heart rates for the day)
- Extract information from images/videos (this is a huge field unto itself called *machine vision*)
- Etc.

Think of feature engineering as an "a la carte" menu: you choose the options that suit a given problem.

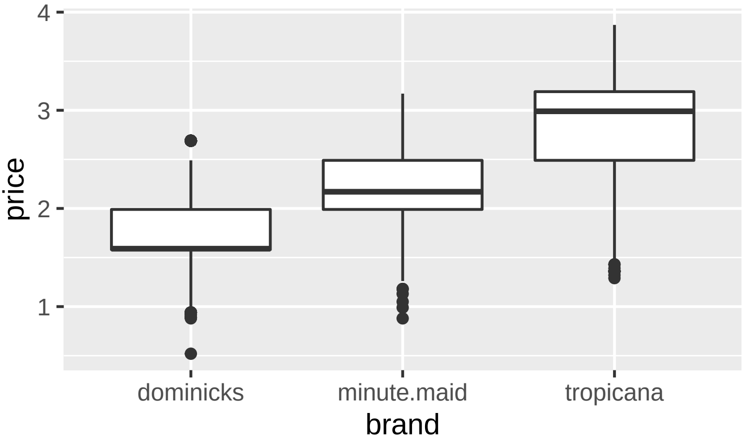
Good feature engineering sometimes involves a lot of creativity!

Example: orange juice sales

We'll consider an example to illustrate some of these ideas:

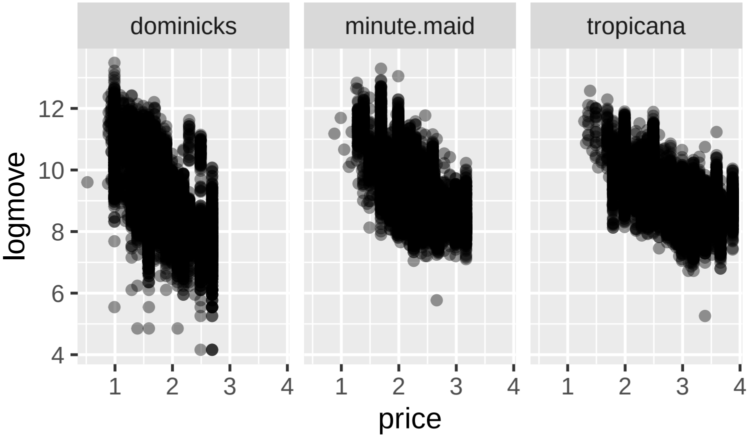
- Three brands of OJ: Tropicana, Minute Maid, Dominicks
- 83 Chicago-area stores and demographic info for each store
- Price, sales (log units moved), and whether advertised (`feat`)
- data in `oj.csv`, code in `oj.R`.

Example: orange juice sales



Each brand occupies a well-defined price range.

Example: orange juice sales



Sales decrease with price (duh).

OJ example: teaching points

This example will illustrate three common feature engineering steps:

- transformations: thinking about *scale* and the appropriate *transformation* in linear models.
- dummy-variable encoding of categorical features
- hand-building new features from existing features, in the form of interactions.

We'll see this in a familiar econ example, but these three steps come up nearly every time you use a linear model outside of really simple problems.

Transformations

When fitting a linear model (*this* goes up, *that* goes down), it's crucial that you think about the *scale* on which you expect to find linearity.

A very common scale is to model the mean for $\log(y)$ rather than y. Remember, this allows us to model *multiplicative* rather than *additive* change. For example:

$$log(y) = \log(\beta_0) + \beta_1 x \iff y = \beta_0 e^{\beta_1 x}$$

Change x by one unit $\longrightarrow$ multiply $E(y \mid x)$ by e^{β_1} units.

Transformations

General guideline: whenever change in y is naturally described on a percentage scale, rather than an absolute scale, use $\log(y)$ as a response.

- prices: "Foreclosed homes sell at a 20% discount. . . "
- sales: "Our Super Bowl ad improved y.o.y. sales by 7% on average, across all product categories."
- volatility, failures, weather events. . . lots of things that are non-negative are expressed most naturally on a log scale.

OJ: price elasticity

A simple "elasticity model" for orange juice sales y might be:

$$\log(y) = \alpha + \gamma \log(\text{price}) + x \cdot \beta$$

The rough interpretation of a log-log regression like this: for every 1% increase in price, we can expect a γ% change in sales.

- Notice here we've used log price on the right-hand side, i.e. transforming price rather than using it in its "raw" state.
- This functional form actually comes from microeconomics, but you could also "discover" it empirically in the data.
- The $x \cdot \beta$ term captures the effect of other, non-price effects on demand.

OJ: price elasticity

Let's try this in R, using `brand` as a feature (x):

```
reg = lm(logmove ~ log(price) + brand, data=oj)
coef(reg) %>% round(2) ## look at coefficients
```

```
##      (Intercept)       log(price)
##            10.83            -3.14
## brandminute.maid   brandtropicana
##             0.87             1.53
```

What happened to `branddominicks`?

OJ: model matrix and dummy variables

Our regression formulas look like $\beta_0 + \beta_1 x_1 + \ldots$ But brand is not a number, so you can't do $\beta \cdot \mathrm{brand}$.

The first step of `lm` is to create a numeric "model matrix" from the input variables. **This is probably the most basic "feature engineering" step in all of data science.**

In `lm` this happens completely behind the scenes, i.e. you don't have to do it by hand. But it's really important to know what's going on, so that we can generalize the idea to other, non-linear models.

OJ: model matrix and dummy variables

Input: feature expressed as a category label.

Brand
dominicks
minute maid
tropicana

OJ: model matrix and dummy variables

Input: feature expressed as a category label.

Brand
dominicks
minute maid
tropicana

Output: that same feature coded as a set of numeric "dummy variables" that we can multiply against β coefficients.

Intercept	minutemaid	tropicana
1	0	0
1	1	0
1	0	1

OJ: model matrix and dummy variables

Our OJ model used `model.matrix` behind the scenes to build a 4 column matrix:

```
> x = model.matrix(~ log(price) + brand, data=oj)
> head(x, 1)
Intercept log(price) brandminute.maid brandtropicana
  1.00000   1.353255         0.000000       1.000000
```

Each factor's reference level is absorbed by the intercept. Coefficients represent change relative to reference level (dominicks here).

Let's go to `oj.R` and take a look.

Interactions

Our second feature engineering step here: *combining existing features* to form new ones. Here this will allow us to model an interaction.

We use the term **interaction** in statistical learning to describe situations where the effect of some feature x on the outcome y is **context-specific.**

Interactions aren't the only reason to create combinations of existing features, but they're a very common use case.

Interactions

Example:

- Biking on flat ground in a low gear: easy (say 2 out of 10)
- Biking on flat ground in a high gear: a bit hard (4 out of 10)
- Biking uphill in a low gear: a bit hard (5 out of 10)
- Biking uphill in a high gear: *very* hard (9 out of 10)

Interactions

Example:

- Biking on flat ground in a low gear: easy (say 2 out of 10)
- Biking on flat ground in a high gear: a bit hard (4 out of 10)
- Biking uphill in a low gear: a bit hard (5 out of 10)
- Biking uphill in a high gear: *very* hard (9 out of 10)

So what's the "effect" of changing the gear from low to high?
There is no one answer... **it depends on context!**

- On flat ground: "high gear" effect = 4 - 2 = 2.
- Uphill: "high gear" effect = 9 - 5 = 4.

One feature (slope) changes how another feature (gear) affects y.

That's an interaction.

Other simple examples

What's the effect of a 5 mph breeze on comfort? It depends!

- When it's hot outside, a 5 mph breeze makes it more pleasant.
- What about the effect of the same breeze when it's cold outside?

Other simple examples

What's the effect of a 5 mph breeze on comfort? It depends!

- When it's hot outside, a 5 mph breeze makes it more pleasant.
- What about the effect of the same breeze when it's cold outside?

What's the effect of two Tylenol pills on a headache? It depends!

- This dose will most likely help a headache in a 165 pound adult human being.
- What about the effect of that same dose on a 13,000 pound African elephant?

Linear models and interactions

Interactions are about capturing these context-specific effects by correctly modeling the *joint effect* of more than one feature at once.

The real world is full of interactions!

- house price versus (square footage, distance to downtown)
- college GPA versus (SAT Math score, college major)
- health outcomes vs (ACE inhibitors, pregnancy)

Interactions play a *massive* role in statistical learning. They are important for making good predictions, and they are often central to social science and business questions.

Linear models *can* accommodate interactions among feature variables—**but only** if we build these interaction terms in "by hand." A super common feature-engineering step in linear modeling!

Interactions

In regression, an interaction is expressed as the product of two features:

$$E(y \mid x) = f(x) = \beta_0 + \beta_1 x_2 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \cdots$$

The term $x_1 x_2$ is our engineered feature: we literally calculate it by multiplying together the two original features to form a new feature.

When we include this new feature in the model, it has a profound effect on the regression equation!

- The effect on y of a one-unit increase in x_1 is now $\beta_1 + \beta_{12} x_2$.
- That is, the x_1 effect depends on x_2!

Interactions in R: use *

The following model statement says: elasticity (the `log(price)` coefficient) is different from one brand to the next:

```r
reg3 = lm(logmove ~ log(price)*brand, data=oj)
coef(reg3) %>% round(1)
```

```
##                  (Intercept)                   log(price)
##                         11.0                         -3.4
##            brandminute.maid               brandtropicana
##                          0.9                          1.0
## log(price):brandminute.maid   log(price):brandtropicana
##                          0.1                          0.7
```

This output is telling us that the elasticities are:

dominicks: -3.4   minute maid: -3.3   tropicana: -2.7

Where do these numbers come from? Do they make sense? Let's try to write out an explicit equation for $E(y \mid x)$.

OJ model as an explicit equation

OJ: bigger models

Let's add a third variable, to give you a sense of what more complex interactions start to look like.

A key question: what changes when we feature a brand? Here, this means in-store display promo or flier ad (`feat` in `oj.csv`). Here are three different options:

1. Main effect only of `feat` on sales volume, without interactions...
2. Two-interaction of `feat` with price (so that elasticity changes when there's a promo or ad)...
3. Three-way interaction of `feat`, brand, and elasticity (so that elasticity changes when there's a promo or ad, but in a different way for each brand!)

In feature-engineering terms: which features (`feat`, `brand`, and `price`) should we combine, and in what ways? **Back to `oj.R`.**

Brand-specific elasticities

	Dominicks	Minute Maid	Tropicana
Not featured	-2.8	-2.0	-2.0
Featured	-3.2	-3.6	-3.5

Findings:

- Ads seem to be associated with decreased elasticity.
- Minute Maid and Tropicana elasticities drop 1.5% with ads, moving them from less to more price sensitive than Dominicks.

Why does marketing seem to increase price sensitivity? And how does this influence pricing/marketing strategy?

Brand-specific elasticities

Before including feat, Minute Maid behaved like Dominicks.

```
reg_interact = lm(logmove ~ log(price)*brand, data=oj)
coef(reg_interact) %>% round(1)
```

```
##                 (Intercept)                   log(price) 
##                        11.0                         -3.4 
##            brandminute.maid               brandtropicana 
##                         0.9                          1.0 
## log(price):brandminute.maid   log(price):brandtropicana 
##                         0.1                          0.7 
```

(Compare the elasticities!)

Brand-specific elasticities

With feat, Minute Maid looks more like Tropicana. Why?

```r
reg_ads3 <- lm(logmove ~ log(price)*brand*feat, data=oj)
coef(reg_ads3) %>% round(1)
```

```
##                      (Intercept)                       log(price) 
##                             10.4                             -2.8 
##                 brandminute.maid                   brandtropicana 
##                              0.0                              0.7 
##                             feat      log(price):brandminute.maid 
##                              1.1                              0.8 
##        log(price):brandtropicana                  log(price):feat 
##                              0.7                             -0.5 
##            brandminute.maid:feat              brandtropicana:feat 
##                              1.2                              0.8 
## log(price):brandminute.maid:feat   log(price):brandtropicana:feat 
##                             -1.1                             -1.0
```

(Again, compare the elasticities!)

What happened?

Minute Maid was more heavily promoted!

```
##      brand
## feat dominicks minute.maid tropicana
##    0     0.743       0.711     0.834
##    1     0.257       0.289     0.166
```

Because Minute Maid was more heavily promoted, AND promotions have a negative effect on elasticity, we were confounding the two effects on price when we estimated an elasticity common to all brands.

Including an interaction with `feat` helped deconfound the estimate! This shows how getting the right engineered features in the model is often critical for interpretation.

Take-home messages: transformations

Transformations are a very common feature-engineering step. They allow us to specify a scale of the relationship between x and y.

The log transformation is easily the most common and is appropriate when changes in the variable in question are most naturally expressed on a percentage scale.

- Exponential growth/decay in y versus x:

$$\widehat{log(y)} = \beta_0 + \beta_1 x \iff \hat{y} = e^{\beta_0} \cdot e^{\beta_1 x}$$

- Power law (elasticity model) in y versus x:

$$\widehat{log(y)} = \beta_0 + \beta_1 \log(x) \iff \hat{y} = e^{\beta_0} \cdot x^{\beta_1}$$

Take-home messages: dummy variables

A super-common feature engineering step is to encode categorical variables using dummy (0/1) variables. In `lm` this happens behind the scenes, but not in all models! Need to know how it works.

Intercept	brand=minute maid	brand = tropicana
1	0	0
1	1	0
1	0	1

One category is arbitrarily chosen as the baseline/intercept. Any baseline is as good as another:

- JJ Watt is 6'5" (baseline), Yao Ming is +13 inches, Simone Biles is -21 inches.
- Simone Biles is 4'8" (baseline), Yao Ming is +34 inches, JJ Watt is +21 inches.

Take-home messages: interactions

An interaction is when one variable changes the effect of another variable on y.

In linear models, we express interactions as products of variables:

$$f(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + \cdots$$

so that the effect on y of a one-unit increase in x_1 is $\beta_1 + \beta_{12} x_2$.

In linear models, these interactions are engineered, or derived, features. Getting the pattern of interactions right is important for:

- good predictive performance
- correct interpretation of model coefficients

Other machine-learning methods will be capable of *automatically* detecting interactions. You won't have to engineer them!

Example 2: house price prediction

Just like with K-nearest neighbors in one x variable, it's important to consider out of sample fit for a linear model.

- The out-of-sample fit can be a useful, easily interpretable way to decide among competing models. It's often the ultimate decider as to whether an engineered feature should be included in the model.
- In models with lots of features (original or engineering), the in-sample fit and out-of-sample fit can be very different.

Let's see these ideas in practice, by comparing three predictive models for house prices in Saratoga, New York.

Variables in the data set

- price (y)
- lot size, in acres
- age of house, in years
- living area of house, in square feet
- number of bedrooms
- number of bathrooms
- number of total rooms
- number of fireplaces
- heating system type (hot air, hot water, electric)
- fuel system type (gas, fuel oil, electric)
- central air conditioning (yes/no)

Three models for house prices

We'll consider three possible models for price constructed from these 10 predictors.

- Small model: price versus lot size, bedrooms, and bathrooms (4 total parameters, including the intercept).
- Medium model: price versus all variables above, main effects only (14 total parameters, including the dummy variables).
- Big model: price versus all variables listed above, together with all pairwise interactions between these variables (90 total parameters, include dummy variables and interactions).

Three models for house prices

A starter script is in `saratoga_lm.R`. Goals for today:

- See if you can “hand-build” a model for price that outperforms all three of these baseline models! Use any combination of transformations, polynomial terms, and interactions that you want.
- When measuring out-of-sample performance, there is *random variation* due to the particular choice of data points that end up in your train/test split. Make sure your script addresses this, e.g. by just averaging the estimate of out-of-sample RMSE over many different random train/test splits, or using cross-validation.

Example 3: feature engineering with dates

Let's see another example of feature engineering in `feature_engineer.R`.

This time we'll see two ideas in practice:

- adding powers of x to get nonlinearities into a linear model
- extracting useful features from raw text fields—in this example, time stamps

Comparison with K-nearest-neighbors

- A linear model makes strong assumptions about the form of $f(x)$: that y and all the features are related linearly.
- If these assumptions are wrong, then a linear model can predict poorly (large bias).
- You have to explicitly engineer interactions into a linear model.
- A nonparametric model like KNN is different:
  - It makes fewer assumptions (and thus can have lower bias).
  - You don't have to "hand build" interactions into the model.
- But it typically has higher estimation variance.
- Which approach leads to a more favorable spot along the bias-variance tradeoff?

Comparison with K-nearest-neighbors

Recall the basic recipe:

- Given a value of K and a point x_0 where we want to predict, identify the K nearest training-set points to x_0. Call this set $\mathcal{N}_0$.
- Then estimate $\hat{f}(x_0)$ using the average value of the target variable y for all the points in $\mathcal{N}_0$.

$$\hat{f}(x_0) = \frac{1}{K} \sum_{i \in \mathcal{N}_l} y_i$$

On your homework, you'll run a "bake-off," comparing KNN with linear models for the Saratoga and power-load forecasting problems.

Measuring closeness

There's a caveat here in running KNN: how do we measure which K points are the closest to x_0? To see why this is trickier than it sounds at first, consider three houses:

- House A: 4 bedrooms, 2 bathrooms, 2200 sq ft
- House B: 4 bedrooms, 2 bathrooms, 2300 sq ft
- House C: 2 bedrooms, 1.5 bathrooms, 2125 sq ft

Which house (B or C) should be "closer to" A?

Measuring closeness

Most people would that A and B are the most similar: they both have 4 BR/2 BA, and their size is similar. House C is has only 2BR/1.5 BA; it's *enormous* for a house with that number of bedrooms. Yet if we naively try to calculate pairwise Euclidean distances, we find the following:

$$\begin{aligned}
d(A, B) &= \sqrt{(4-4)^2 + (2-2)^2 + (2200-2300)^2} \\
&= 100 \\
d(A, C) &= \sqrt{(4-2)^2 + (2-1.5)^2 + (2200-2125)^2} \\
&\approx 75.03
\end{aligned}$$

So A's nearest neighbor is C, not B!

Measuring closeness

What happened here is that the `sqft` variable completely overwhelmed the BR and BA variables in the distance calculations.

The root of the problem:

- square footage is measured on an *entirely different scale* than bedrooms or bathrooms.
- Treating them as if they're on the same scale leads to counter-intuitive distance calculations.

The simple way around this: weighting. That is, we treat some variables as more important than others in the distance calculation.

Weighting

Ordinary Euclidean distance:

$$d(x_1, x_2) = \sqrt{\sum_{j=1}^{p} \{x_{1,j} - x_{2,j}\}^2}$$

Weighted Euclidean distance:

$$d_w(x_1, x_2) = \sqrt{\sum_{j=1}^{p} \{w_j \cdot (x_{1,j} - x_{2,j})\}^2}$$

This depends upon the choice of weights $w_1, \ldots, w_p$ for each feature variable.

Weighting

We can always choose the scales/weights to reflect our substantive knowledge of the problem. For example: - a "typical" bedroom is about 200 square feet (roughly 12x16).
- a "typical" bathroom is about 50 square feet (roughly 8x6).

This might lead us to scales like the following:

$$d_w(x, x') = \sqrt{(x_1 - x_1')^2 + \{200(x_2 - x_2')\}^2 + \{50(x_3 - x_3')\}^2}$$

where x_1 is square footage, x_2 is bedrooms, and x_3 is bathrooms. Thus a difference of 1 bedroom counts 200 times as much as a difference of 1 square foot.

Weighting by scaling

But choosing weights by hand can be a real pain.

A “default” choice of weights that requires no manual specification is to weight by the inverse standard deviation of each feature variable:

$$d_w(x_1, x_2) = \sqrt{\sum_{j=1}^{p} \left(\frac{x_{1,j} - x_{2,j}}{s_j}) \right)^2}$$

where s_j is the sample standard deviation of feature j across all cases in the training set.

Weighting

This is equivalent to calculating "ordinary" distances using a rescaled feature matrix, where we center and scale each feature variable to have mean 0 and standard deviation 1:

$$\tilde{X}_{ij} = \frac{(x_{ij} - \mu_j)}{s_j}$$

where (μ_j, s_j) are the sample mean and standard deviation of feature variable j.

Then we can run ordinary (unweighted) KNN using Euclidean distances based on $\tilde{X}$.

Example: KNN on the house-price data

Some packages in R will scale things automatically for you, but it's good to know how to do it yourself.

This requires a bit more prep work on the data:

```r
saratoga_split = initial_split(SaratogaHouses, prop = 0.8)
saratoga_train = training(saratoga_split)
saratoga_test = testing(saratoga_split)

# construct the training and test-set feature matrices
# note the "-1": this says "don't add a column of ones for the intercept"
Xtrain = model.matrix(~ age + livingArea + bedrooms - 1, data=saratoga_train)
Xtest = model.matrix(~ age + livingArea + bedrooms - 1, data=saratoga_test)

# training and testing set responses
ytrain = saratoga_train$price
ytest = saratoga_test$price

# now rescale:
scale_train = apply(Xtrain, 2, sd)  # calculate std dev for each column
Xtilde_train = scale(Xtrain, scale = scale_train)
Xtilde_test = scale(Xtest, scale = scale_train)  # use the training set scales!
```

Example: KNN on the house-price data

```r
head(Xtrain, 2)
```

```
##   age livingArea bedrooms
## 1  42        906        2
## 3 133       1944        4
```

```r
head(Xtilde_train, 2) %>% round(2)
```

```
##    age livingArea bedrooms
## 1 0.47      -1.36    -1.41
## 3 3.56       0.30     1.05
```

Take-home messages

- Linear models always require a choice of scale. Using no transformation is still a choice!
- Selecting and engineering features/interactions are really important components of building a high-performing linear model.
- Other predictive-modeling methods we'll learn usually require at least some feature engineering, to one degree or another. But linear models are especially labor intensive in this regard.
- Selecting/engineering features by hand is laborious! Stay tuned for more automated methods :-)