Fundamentals of Data Science

Release Spring 2019.1

Stephyn Butcher

Apr 12, 2019

CONTENTS

CHAPTER

ONE

LINEAR MODELS - PART 1

We started our efforts at modeling with means, rates, and proportions. These suffice surprisingly often. To this repertoire, we added mathematical distributions as a kind of model. What these two types of models have in common is that they only use information about the variable itself. For example, if we want to model some numerical variable, y, we might plot a histogram to get a feel for the shape of the data, calculate the mean, or even estimate a mode of y using moments to estimate the parameters of an appropriate mathematical distribution. But we don't use any information from other variables.

In our EDA, however, we *have* used other variables, often looking at pairwise relationships between variables using visualization and correlation as a guide. In this chapter, we will start building linear models of a single variable (only one "x"). In a later chapter, we'll expand our linear models to include multiple variables.

1.1 Terminology

Before we start, we need to get a bit of terminology out of the way. Data Scientists come from a variety of backgrounds and techniques have been invented and reinvented under a variety of names in a number of fields. In order to get everyone on the same page, we're going to talk a little bit about terminology.

Artificial Intelligence (AI) is concerned with building algorithms that enable computers to learn, understand, and solve problems. One subset of AI is Machine Learning. It might help to contrast ML with one of the older branches of AI, state space search.

In state space search, a problem is specified terms of permitted states and actions that can be taken in those states. If you specify an end state as a goal, a state space search algorithm will find a series of actions that will get you from the current state to the goal.

In contrast, a *supervised* machine learning algorithm takes data in the form of *examples* of inputs and outputs and returns a model of those examples. You can then supply different inputs to the model and get an appropriate output. Because of the relationship between inputs and an output, supervised machine learning is often thought of as function approximation or at least the modeling of pattern recognition with some sort of function (or, more broadly, mapping).

Remember our old friend the coin with a probability of heads, p, and the observed outcomes 6 heads and 4 tails. We noted that it is impossible to know *for sure* what p is from the data. In supervised machine learning, our problem is even more difficult.

1.1.1 Some Computational Learning Theory

First, we have some notion of an output y from a system. We may or may not be measuring that value correctly. Second, we think we know the inputs to the system X but we must remember the known knowns, known unknowns, and unknown unknowns that factor into any data collection endeavor. So, at best, X, the variables we have, are a large subset of $\mathbf{X}$, the true inputs, and are properly measured. It could be instead that X and $\mathbf{X}$ have only a small intersection.

Third, there is, we hope, some actual relationship between y and X but we don't know the actual functional form, $\mathbf{f}(X)$ anymore than we know p. Our approximation is $f(X)$.

Fourth, $f(X)$ must be estimated and even for the same $f(X)$ there are different ways of estimating it, $g(X, y)$.

So we use an algorithm to fit a model:

$$f = g(X, y)$$

where g is the machine learning algorithm (we'll assume we're talking about "supervised" for now), (y, X) are a tuple of output and input vectors, and f is the model. The model can then be used:

$$\hat{y} = f(X)$$

on new inputs X to estimate values of y, which are called (by convention), $\hat{y}$.

It's interesting to note that one of the machine learning libraries for Python (and that we will be using), `scikit-learn`, has an API that mirrors these constructs exactly:

```
algorithm = LinearRegression()
model = algorithm.fit(X, y) # g(X, y)
y_hat = model.predict(X)     # f(X)
```

The constructor creates an instance of the algorithm that holds *metaparameters*. A `fit` method is called on the instance of the algorithm with training data, X and y. This is basically $g(X, y)$. A `predict` method is then called on the model instance to predict new values from new X.

We'll see this a lot more over the next few chapters.

- If we think of the mean as supervised machine learning, what are $g(X, y)$ and $f(X)$?

1.1.2 More About X and y

X is a matrix of variables. Each column is called a *feature* (attribute, factor, variable, inputs) and y is a vector of output values (target, response). Each column of X, X_i, can be a different data type (unlike a mathematical matrix) although the user has to be careful. Sometimes an algorithm g will accept types that don't actually make sense for f.

If y is a numerical variable, this is called *regression* regardless of the actual g used. If y is a categorical variable, this is called *classification*. It is important to distinguish these uses because we can use can use *linear regression* to solve *regression* problems and we can use *logistic regression* to solve *classification* problems.

In concrete terms, if you want to estimate things like price, age, IQ, or leaf volume, this is a regression problem. If you want to estimate a purchase outcome, religion, animal, or creditworthiness outcome, then this is a classification problem. This is important because not all algorithms, metrics, evaluation techniques are appropriate for both regression and classification.

1.1.3 Why Linear and Logistic Regression

So why not dive directly into boosted random forests? Linear and logistic regression regularly top the "must know" lists of algorithms for data scientists. . . and for good reason.

First, we can go very deeply into these models and understand how they work, their advantages, and disadvantages.

Second, one of the key findings in data science is that with "big" data, simple models can perform extremely well and more time should generally be placed on refining the features (inputs, or in regression-speak, the *regressors*) to the models. Facebook does this for many of its machine learning applications.

Third, simple models are often sufficient to get a project off the ground, to go from 0% to 60% instead of 92% to 93%. A fielded, simple data product with 3% lift is better than a complicated data product with 20% lift that never gets fielded, needs to be constantly re-tuned or nobody understands.

Finally, and specifically with regard to all types of regression, they are useful when we are trying to explore relationships in our process rather than simply prediction.

For example, if we use a neural network to model our system, we may get a very good model of the inputs to the outputs. However, we will not be able to understand *how* the inputs relate to the outputs. With a linear regression model, we may get a poorer model (or have to futz with it more) but we will be able to understand the relationship between inputs and outputs. Ultimately it depends on what we're trying to do with the model (and hence the George Box quote from before). This is a general tension between statistics-based methods and machine learning-based methods.

Nevertheless, we will address more complicated approaches in later chapters (machine learning) because *big* is really relative to the problem space and you may not be able to find or create better features than what you have. In any case, you should always start with the simplest thing first.

Regression as a technique comes out of the field of statistics although people have been working with regression for over 130 years. Sir Francis Galton presented regression for the first time in a lecture in 1877.

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%matplotlib inline
```

[3]:
```
import numpy as np
import scipy.stats as stats
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import random
import patsy
import sklearn.linear_model as linear

import sys
sys.path.append('resources')
import models

sns.set(style="whitegrid")
```

1.2 Linear Regression

In a previous chapter we talked about how we can use the mean of a variable as a model of that variable. The logic was as follows:

1. We're going to pick a constant, β_0, as an estimate for any y.
2. We want this constant to minimize mean squared error: $\frac{1}{n}\sum(y - \beta_0)^2$ (MSE).
3. The mean, $\bar{y} = \frac{1}{n}\sum x_i$, minimizes MSE so use $\bar{y} = \beta_0$.

In terms of the previous section:

$$G = \frac{1}{n}\sum x_i$$

which returns:

$$f = \beta_0$$

which we use as:

$$\hat{y} = \beta_0$$

If we use β_0 as our model of y then our estimates of y, $\hat{y}$, will have an MSE equal to $\frac{1}{n}\sum(y-\beta_0)^2$ which is also the formula for variance, σ^2. We can express our model probabilistically as follows:

$$y \sim N(\beta_0, \sigma)$$

which says that y is distributed normally with mean β_0 and error σ. Another way of looking at this is to separate out β_0. We can make this an equality:

$$y = N(\beta_0, \sigma)$$

and subtract β_0 from both sides we have:

$$y - \beta_0 = N(\beta_0, \sigma) - \beta_0$$

$$y - \beta_0 = N(0, \sigma)$$

then rearrange:

$$y = \beta_0 + N(0, \sigma)$$

While $N(0, \sigma)$ is often called "error", as we have seen, this is really just all the other things (the "known unknowns" and the "unknown unknowns") that influence y but we are not taking into account. What we are saying here is that the mean is a "good" model for y if our errors are normally distributed.

But it turns out we might be able to do better. What if all the unknowns aren't actually unknown? How can we include them?

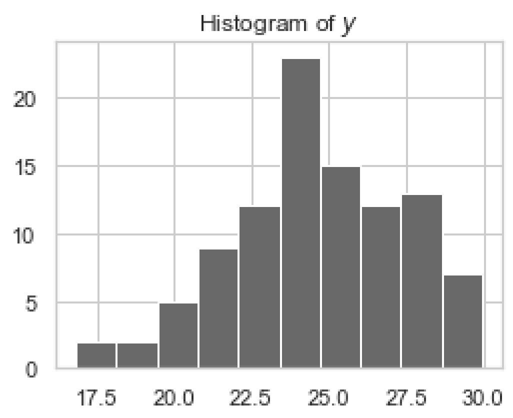
Let's start by making a little synthetic data. We're going to generate both a y and an x but we're only going to use y for right now.

```
[4]: np.random.seed(8472385)
```

```
[5]: data = {}
     data["x"] = stats.norm.rvs(10.0, 1, 100)
     data["e"] = stats.norm.rvs(0, 1, 100)
     data["y"] = data["x"] * 2.5 + data["e"]
     data = pd.DataFrame(data)
```

We'll start by plotting a histogram of y to get a sense of its shape:

```
[6]: figure = plt.figure(figsize=(4,3))
     axes = figure.add_subplot(1, 1, 1)
     axes.hist(data.y,color="dimgray")
     axes.set_title("Histogram of $y$")
     plt.show()
     plt.close()
```



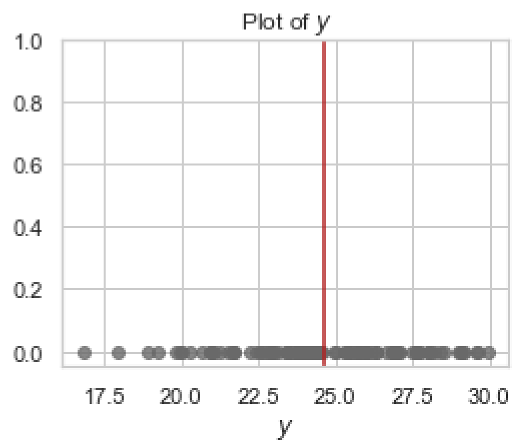
Let's calculate the mean of y:

```
[7]: beta_0 = data.y.mean()
     beta_0
```

```
[7]: 24.638917684363282
```

which is our model of y, β_0. Graphically, it's something like this where values of y are shown as dots and, β_0, the mean–our model–is shown as a red line:

```
[8]: figure = plt.figure(figsize=(4,3))

     axes = figure.add_subplot(1, 1, 1)
     axes.scatter( data.y, [0] * data.y.size, color="dimgray", alpha=0.8)
     axes.axvline(beta_0, color="firebrick")
     axes.set_xlabel(r"$y$")
     axes.set_ylim((-0.05, 1))
     axes.set_title(r"Plot of $y$")

     plt.show()
     plt.close()
```



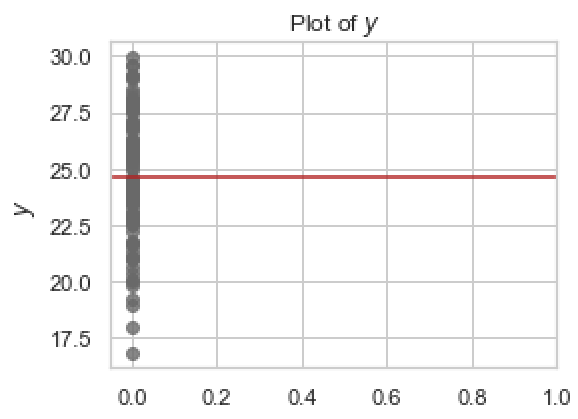
By convention, we often show the *target* variable on the y-axis:

```
[9]: figure = plt.figure(figsize=(4,3))
```

(continues on next page)

(continued from previous page)

```
axes = figure.add_subplot(1, 1, 1)
axes.scatter( [0] * data.y.size, data.y, color="dimgray", alpha=0.8)
axes.axhline(beta_0, color="firebrick")
axes.set_xlim((-0.05, 1))
axes.set_ylabel(r"$y$")
axes.set_title(r"Plot of $y$")

plt.show()
plt.close()
```

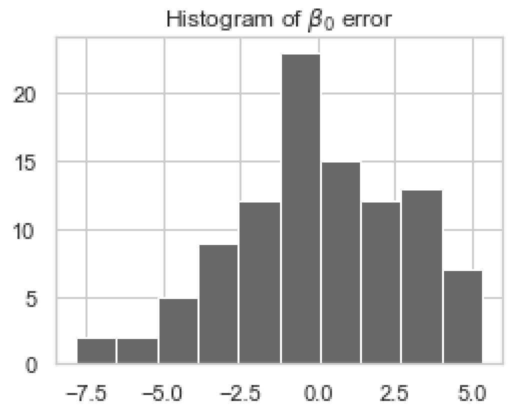


Let's calculate the "error" of this model:

```
[10]: data["error"] = data.y - beta_0
```

and show a histogram of it:

```
[11]: figure = plt.figure(figsize=(4,3))
axes = figure.add_subplot(1, 1, 1)
axes.hist(data.error,color="dimgray")
axes.set_title(r"Histogram of $\beta_0$ error")
plt.show()
plt.close()
```



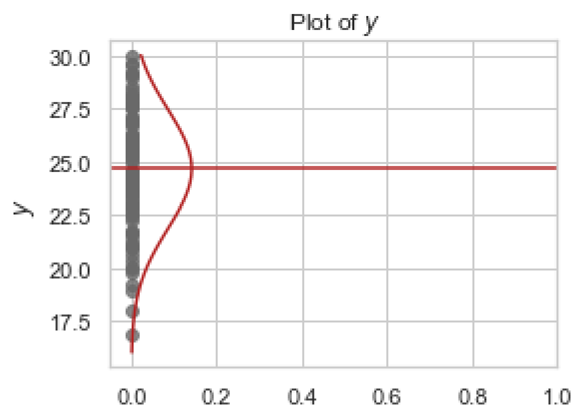
This is an empirical application of our derivation above. The errors are centered at zero (which we expect given the formulas for the mean on the one hand and the formulas for variance and MSE on the other hand). The empirical distribution is not quite normal and that may be okay for our purposes. This is what we mean by "is the normal distribution a good model for this data" but let's put that to the side for now. Ideally, the errors are normally distributed with y's standard deviation (variance):

```
[12]: stdev = data.y.std()
      print(stdev)
```

```
2.809849407158323
```

Let's plot the idealized case:

```
[13]: figure = plt.figure(figsize=(4,3))

      axes = figure.add_subplot(1, 1, 1)
      axes.scatter( [0] * data.y.size, data.y, color="dimgray", alpha=0.8)
      axes.axhline(beta_0, color="firebrick")
      ys = np.linspace(16, 30)
      xs = [stats.norm.pdf( k, loc=beta_0, scale=stdev) for k in ys]
      axes.plot( xs, ys, color="firebrick")
      axes.set_xlim((-0.05, 1))
      axes.set_ylabel(r"$y$")
      axes.set_title(r"Plot of $y$")

      plt.show()
      plt.close()
```



Now we're ready to introduce "better". The basic idea is to replace β_0 with a linear function. Instead of:

$$y = \beta_0 + N(0, \sigma)$$

we have:

$$y = \beta_0 + \beta_1 x_1 + N(0, \sigma)$$

in the case of *univariate* ("one variable") linear regression. As we have seen, another way to look at this is:

$$y \sim N(\beta_0 + \beta_1 x_1, \sigma)$$

So basically, we're saying y is normally distributed with a mean equal to this *function* of one variable, x_1, with "error" (I like "noise") equal to $N(0, \sigma)$. We often refer to $N(0, \sigma)$ as ϵ. So you will often see:

$$y = \beta_0 + \beta_1 x_1 + \epsilon$$

as well.

1.2.1 Implementation

Python is a general purpose language so it doesn't just happen to have regression built in. However, there are Python libraries with linear regression implementations. Here are a few options:

1. scikit-learn
2. statsmodels
3. scipy

Since we're going to be working a bit with scikit-learn throughout the book, we're going to start using it now even though we could probably get by with the included scipy library. It's nice to know there are other options.

We're also going to use a library called patsy to save us some typing. Patsy permits us to describe our linear models using an R-like formula syntax.

We'll wrap both of these libraries in a function called `linear_regression`. The main reason for this is that the `scikit-learn` library has a machine learning focus so that it returns *no* metrics about the regression. This is in contrast to the `statsmodels` library that returns the model along with standard Frequentist metrics (and incorrectly, I might add).

Because we do want *some* (Bayesian) evaluation of the linear regression, we'll need to customize our results, which we will do in stages. For now, let's not worry about inference. The `linear_regression` function in the `models` module of the book's code (in the `resources`) directory, will work. It runs most of the information we need including:

1. the model from Scikit Learn in case we need it.
2. the formula used to fit the model.
3. the coefficients for the features.
4. the σ (error) of the regression.
5. the R^2 of the regression or the coefficient of determination. R^2 is literally the Pearson correlation coefficient squared thus, unlike the Pearson correlation coefficient, it is never negative. It has several interpretations:
    1. The percent of the variability explained by the model.
    2. The improvement of the model over the constant model (see below).
    3. The square of correlation (as we just mentioned).

1.2.2 Linear Regression with a Numerical Variable (Synthetic Data)

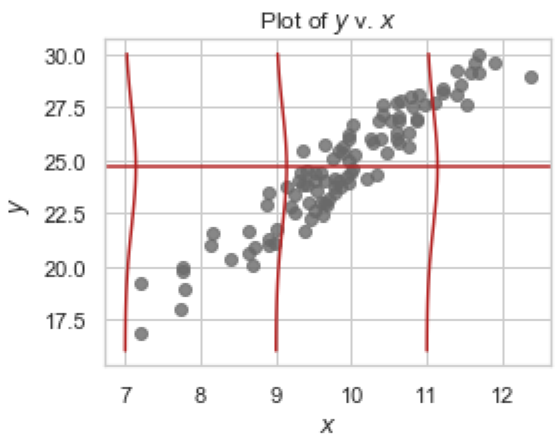
We begin our introduction to univariate linear regression looking at a single numerical feature, x. But first, let's think about what our current (mean) model means in the context of another variable like x

[14]:
```python
figure = plt.figure(figsize=(4,3))

axes = figure.add_subplot(1, 1, 1)
axes.scatter( data.x, data.y, color="dimgray", alpha=0.8)
axes.axhline(beta_0, color="firebrick")

ys = np.linspace(16, 30)
xs = [stats.norm.pdf( k, loc=beta_0, scale=stdev) + 7 for k in ys]
axes.plot( xs, ys, color="firebrick")

xs = [stats.norm.pdf( k, loc=beta_0, scale=stdev) + 9 for k in ys]
axes.plot( xs, ys, color="firebrick")
```

(continues on next page)

(continued from previous page)

```
xs = [stats.norm.pdf( k, loc=beta_0, scale=stdev) + 11 for k in ys]
axes.plot( xs, ys, color="firebrick")

axes.set_ylabel(r"$y$")
axes.set_xlabel(r"$x$")
axes.set_title(r"Plot of $y$ v. $x$")

plt.show()
plt.close()
```



Essentially we are saying, no matter what x is, we predict β_0. This is the red line. It is actually a linear model but not necessarily a good one! The error of the model for different values of x can be visualized by showing normally distributed error overlayed on x. We'll see how a linear regression model changes this.

We fit our first model:

$$\hat{y} = \beta_0 + \beta_1 x$$

or

$$y = \beta_0 + \beta_1 x + \epsilon$$

Notice the slight difference between the two formulations. The estimate $\hat{y}$ lacks the "noise", whereas the real value includes it.

In Patsy, this is expressed as $y \sim x$:

[15]:
```
results = models.linear_regression("y ~ x", data)
```

We can use `simple_describe_lr` to show a "nice" table of results:

[16]:
```
print(models.simple_describe_lr(results))
```

```
Model: y ~ x
---------  -------  ---
Coefficients             Value
              $\beta_0$  -0.48
x             $\beta_1$  2.54

Metrics       Value
```

(continues on next page)

(continued from previous page)

```
$\sigma$      0.92
$R^2$         0.89
--------  ------  ---
```

(Note: the actual nice tables are temporarily turned off. They show up well in the Notebook and as HTML but when the PDF is generated, they look awful. We're stuck with ASCII tables until I can get it fixed.)

We'll return to the coefficients shortly. The σ is the standard deviation of our predictions (strangely, it's closer to error than MSE is because it's not squared). The coefficient of determination (R^2), as previously mentioned, has several related interpretations:

1. The model explains 89.4% of the variability in y.
2. The model is better than the β_0 only model (here R^2 is just an index of "better" from 0 to 1).
3. The correlation between y and the feature(s) is 89.4%.

We will generally pick explanation #1 but you should be aware of the others.

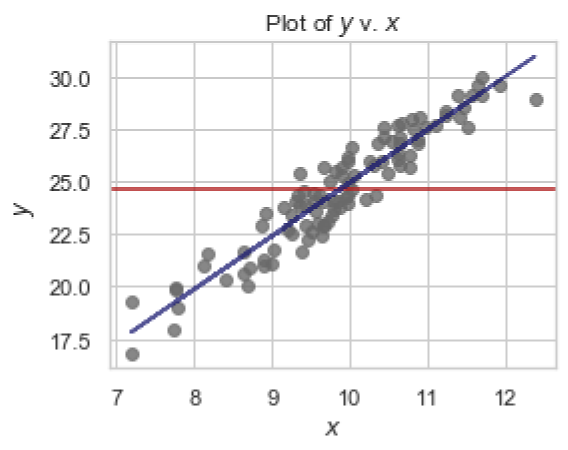
Here, the model explains 89% of the variation of y. This is generally considered excellent but there are many caveats. We'll return to estimation and evaluation later in this chapter and in the next two chapters. For now, we just want to understand what the overall model *means*. In the next section, we'll start to interpret the coefficients as well.

The estimated formula is:

$$\hat{y} = -0.48 + 2.54x$$

which we can plot:

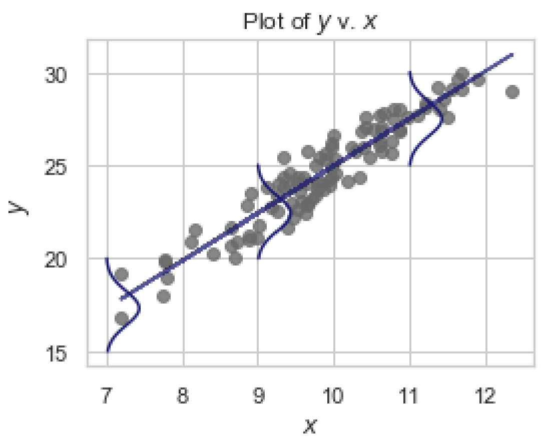
```
[17]: figure = plt.figure(figsize=(4,3))

axes = figure.add_subplot(1, 1, 1)
axes.scatter( data.x, data.y, color="dimgray", alpha=0.8)

axes.axhline(beta_0, color="firebrick")

beta = results[ "coefficients"]
axes.plot(data.x, [beta[ 0] + beta[ 1] * x for x in data.x], '-', color="midnightblue
↪", alpha=0.75)

axes.set_ylabel(r"$y$")
axes.set_xlabel(r"$x$")
axes.set_title(r"Plot of $y$ v. $x$")

plt.show()
plt.close()
```



For comparison, we show the old model (the mean of y) in red and the new model (linear regression $y = -0.48 + 2.54x$) in blue. You can always use the mean as a baseline model.

As we noted above, a few mathematical identities allow us to equate the standard deviation as at least being proportional to the MSE. For the mean, the standard deviation was 2.81. For the linear regression, the standard deviation (σ) is 0.91. What about that error? We showed it before for the mean only model. What about the linear regression?

```
[18]: figure = plt.figure(figsize=(4,3))

axes = figure.add_subplot(1, 1, 1)
axes.scatter( data.x, data.y, color="dimgray", alpha=0.8)

beta = results[ "coefficients"]
axes.plot(data.x, [beta[ 0] + beta[ 1] * x for x in data.x], '-', color="midnightblue
↪", alpha=0.75)

ys = np.linspace(15, 20)
xs = [stats.norm.pdf( k, loc=beta[ 0] + beta[ 1] * 7, scale=results["sigma"]) + 7 for␣
↪k in ys]
axes.plot( xs, ys, color="midnightblue")

ys = np.linspace(20, 25)
xs = [stats.norm.pdf( k, loc=beta[ 0] + beta[ 1] * 9, scale=results["sigma"]) + 9 for␣
↪k in ys]
axes.plot( xs, ys, color="midnightblue")

ys = np.linspace(25, 30)
xs = [stats.norm.pdf( k, loc=beta[ 0] + beta[ 1] * 11, scale=results["sigma"]) + 11␣
↪for k in ys]
axes.plot( xs, ys, color="midnightblue")

axes.set_ylabel(r"$y$")
axes.set_xlabel(r"$x$")
axes.set_title(r"Plot of $y$ v. $x$")

plt.show()
plt.close()
```



Remember when we overlayed the error on the constant mean model? Here we can see that the error is smaller and tight around the regression line. This fully encompasses what we mean by:

$$y = -0.48 + 2.54x + N(0, 0.91)$$

1.2.3 Linear Regression with a Numerical Feature (Actual Data)

Now we're going to look at two variables from a real data. We have some data about child IQ as well as some possible explanatory features including the IQs and education of their mothers. We have a sense that there is a relationship here and we want to build a linear regression model of it. We assume that all the previous steps (ASK, GET, EXPLORE) have been completed.

```
[19]: child_iq = pd.read_csv( "resources/child_iq.tsv", sep="\t")
```

Although for now we're only interested in one variable, let's look at the data types and data examples:

```
[20]: child_iq.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 434 entries, 0 to 433
Data columns (total 5 columns):
child_iq    434 non-null int64
mom_hs      434 non-null int64
mom_iq      434 non-null float64
mom_work    434 non-null int64
mom_age     434 non-null int64
dtypes: float64(1), int64(4)
memory usage: 17.0 KB
```

We have 434 observations with no null values. The variables are as follows:

- **child_iq** - numeric, the child's IQ.
- **mom_hs** - boolean, whether or not the mother finished HS.
- **mom_iq** - numeric, the mother's IQ.
- **mom_work** - categorical, whether and how much the mother works.
- **mom_age** - numeric, the mother's age.

Side Note on "Features"

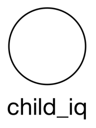
There is not always a one to one relationship between the *variables* in the data and *features* in our model. For example, you might have a variable like *mom_age* your data but you might want to include both mom_age and $\sqrt{mom\_age}$ as features in your model (for reasons we'll talk about later).

Let's look at some actual data:

```
[21]: child_iq.head()
```

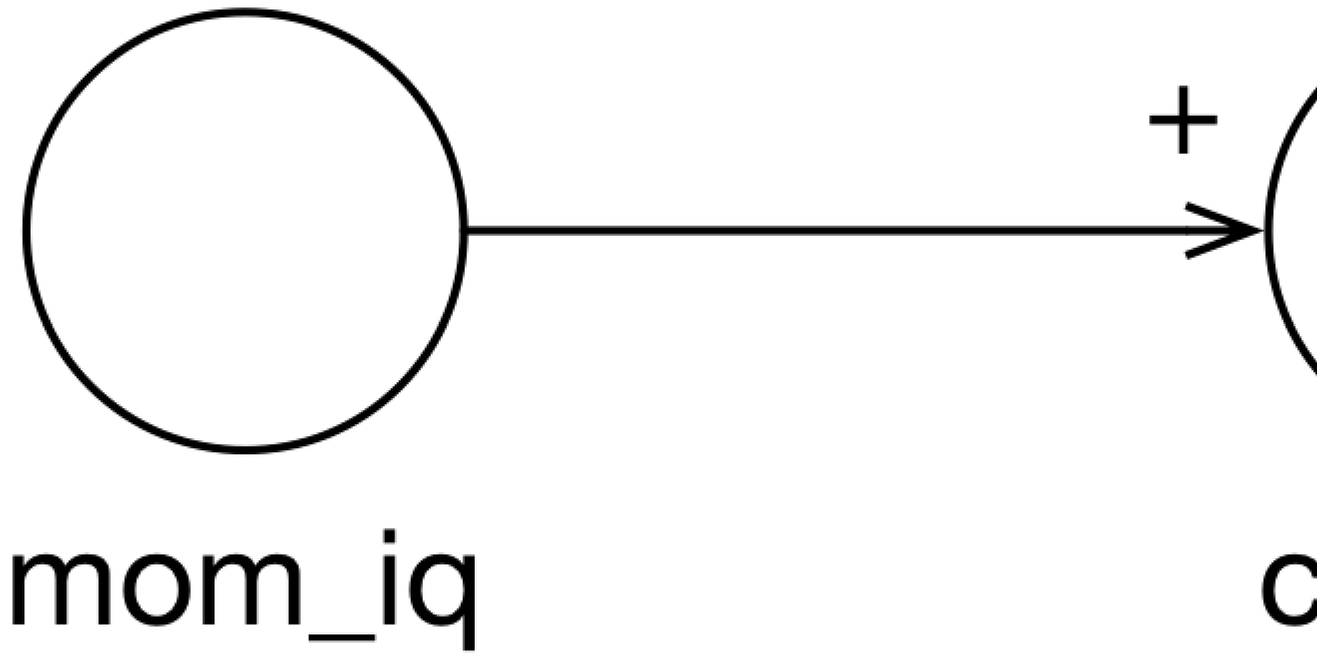
```
[21]:    child_iq  mom_hs      mom_iq  mom_work  mom_age
0         65       1  121.117529         4       27
1         98       1   89.361882         4       25
2         85       1  115.443165         4       27
3         83       1   99.449639         3       25
4        115       1   92.745710         4       27
```

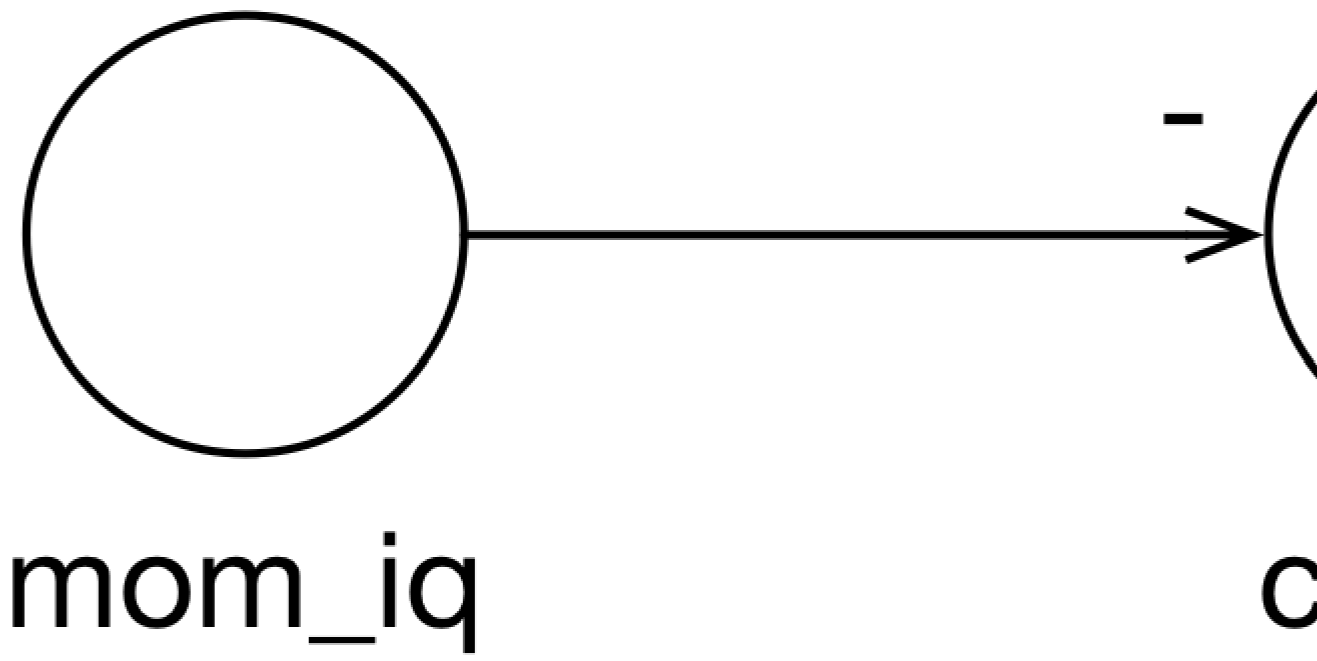
Now's the time to start bringing in our Causal Loop Diagrams. So far, our models have been something like this:



That is, we only looked at a single target variable and sought to model that target variable only in terms of itself. Now we can try to use other variables, although in this chapter we're only going to use one variable. More importantly, however, is that when we created the causal loop diagram, we put a "+" or "-" on the arrow to indicate a positive or negative relationship. When we estimate the model, our β_1 should similarly be "+" or "-".

Think about it. What do you expect the relationship between the mother's IQ and the child's IQ to be?


+
mom_iq
c


-
mom_iq
c

It's a good idea to get a sense of the baseline model:

```
[22]: print("child_iq = {0} ({1})".format(child_iq.child_iq.mean(), child_iq.child_iq.
      ↪std()))
```

```
child_iq = 86.79723502304148 (20.410688494227184)
```

This basically says that if we used the mean (86.8) to predict any given child's IQ, the standard deviation of our estimates would be 20.4 IQ points.

Let's estimate the linear model:

```
[23]: result1 = models.linear_regression( "child_iq ~ mom_iq", data = child_iq)
      print(models.simple_describe_lr(result1))
```

```
Model: child_iq ~ mom_iq
---------  ------  ---
Coefficients              Value
              $\beta_0$   25.80
mom_iq        $\beta_1$   0.61

Metrics       Value
$\sigma$      18.27
$R^2$         0.20
---------  ------  ---
```

The model is:

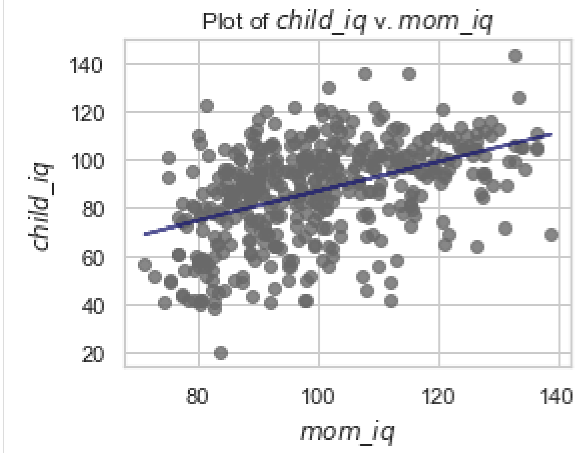
$$\hat{child_i q} = 25.8 + 0.61 mom\_iq$$

The R^2 is not fantastic but this is real life and real data. Our model explains only 20% of the variation in y. This is understandable since we're only trying to use one variable in what is clearly a multivariate process probably full of "unknown unknowns".

The standard deviation of our estimate has only dropped to 18.3 IQ points from 20.4.

Let's plot data and the model:

```
[24]: figure = plt.figure(figsize=(4,3))

      axes = figure.add_subplot(1, 1, 1)
      axes.scatter( child_iq.mom_iq, child_iq.child_iq, color="dimgray", alpha=0.8)

      beta = result1[ "coefficients"]
      axes.plot(child_iq.mom_iq, [beta[ 0] + beta[ 1] * x for x in child_iq.mom_iq], '-',␣
      ↪color="midnightblue", alpha=0.75)

      axes.set_ylabel(r"$child\_iq$")
      axes.set_xlabel(r"$mom\_iq$")
      axes.set_title(r"Plot of $child\_iq$ v. $mom\_iq$")

      plt.show()
      plt.close()
```



1.2.4 Interpreting Linear Regression

There are many different ways to interpret linear regression, which is to say, the βs.

The *causal* interpretation of β_i (where $i > 0$) goes something like this: if you increase x_i by 1 then y increases by β_i, other things being equal. The causal interpretation can lead to some silliness when the relationship is only associative and not actually causal. For example, in the current case, you can't increase a child's IQ by increasing the mother's IQ. This leads to the predictive interpretation.

The *predictive* interpretation can be more difficult to understand so let's use the current example: β_1 is the expected difference in IQ between children whose mothers had an IQ of 80 and children whose mothers had an IQ of 81, in this case, 0.61 IQ points. This takes a little getting used to because it's not the interpretation we generally hear. However, it is consistent with our derivation of linear regression as replacing a constant (β_0) value for the mean with a *function* ($\beta_0 + \beta_1 x$). Thus our model says, given a value for $mom\_iq = 80$, what is the mean value of $child\_iq$ *for those children*?

$chi\hat{l}d\_iq = 25.8 + 0.61 mom\_iq$

$chi\hat{l}d\_iq = 25.8 + 0.61 \times 80$

$chi\hat{l}d\_iq = 25.8 + 48.8$

$chi\hat{l}d\_iq = 25.8 + 48.8$

$chi\hat{l}d\_iq = 74.6$

Overall, this leads to a smaller error than just estimating using the overall mean of 86.8 (although not much of a decrease in error). We will generally try to use the predictive interpretation but the causal interpretation is just so much easier to say. We just have to be careful when the interpretation is at odds with what is actually possible for the real world process.

1.2.5 What about the intercept?

We haven't mentioned β_0 yet. It's worth noting that the β_0 from our original mean only model and the β_0 from our linear regression are not the same. The mean has now been replaced by a function. So what is β_0? From a purely mathematical point of view, it's the value of y when $x = 0$. The problem is that $x = 0$ may not be a legitimate value for x and so β_0 has no real or useful interpretation. In this case, the mother cannot have an IQ of 0 so the intercept is meaningless.

There are ways to transform the data so that the intercept *does* have a meaningful interpretation and we'll talk about those later.

It's worth noting that this is just a special case of a more general problem: prediction outside the observed values of x. Generally speaking, you should use your model to make predictions for values of the features that you did not observe in the data set. This is where the model is likely to be weak and will likely be weaker the farther x is from the observed minimum or maximum when the model was built. $mom\_iq = 0$ is just a very extreme case of this.

This also makes sense in the context of the predictive interpretation of linear regression. If you didn't observe many (or any) values, you probably shouldn't trust your model's prediction of their *mean*.

1.2.6 Linear Regression and a Binary Feature (Actual Data)

As we will see over the course of this and the next chapter, linear regression is fairly flexible. It will take any feature numerical feature and spit out a model. This can be problematic if you don't think about a little bit about what the feature and what the coefficient might *mean*.

For example, we know that the linear regression algorithm (G) won't accept strings as inputs. There's no way to have a feature that's ["yes", "no", "no", "yes", "no"] and have the algorithm work. However, it will work with binary features (categorical variables with two value *codes*) such as [1, 0, 0, 1, 0]. This is the case we're going to look at right shortly. The variable is mom_hs which indicates whether (1) or not (0) the mother went to high school. In regression, these variables are often called "indicator variables" for this reason.

However, if we have an encoding of $\{0, 1, 2, 3\}$ for some categorical feature although this will "work" in the sense that G won't fail, the resulting model is questionable because "2" in this case isn't actually twice "1" and therefore it's not clear what the coefficient means.

This will happen if the feature is *ordered* as well. Suppose we use the *place* that someone finished in a marathon as a feature. Because 16th place isn't twice 8th place, this can lead to problems. It would be better in this case to use the finish time as a feature instead. In that case, if it took 16th place twice as long to finish as 8th place, that *would* make sense.

For categorical features of more than two values, what we generally do in use a "one hot" encoding. We'll talk about this later. Let's return to the simple binary feature case, mom_hs, and our model: $child\_iq \sim mom\_hs$:

```
[25]: result2 = models.linear_regression("child_iq ~ mom_hs", data=child_iq)
```

```
[26]: print(models.simple_describe_lr(result2))
```

```
Model: child_iq ~ mom_hs
---------  ------  ---
Coefficients              Value
               $\beta_0$  77.55
mom_hs         $\beta_1$  11.77

Metrics        Value
$\sigma$       19.85
$R^2$          0.06
---------  ------  ---
```

This model is not even quite as good as the last one but for our purposes what is more important is what it all means. The model is:

$\hat{y} = 77.5 + 11.8 mom\_hs$

but mom_hs can only take on two values: 0 or 1 so we either have:

$\hat{y} = 77.5 + 11.8 mom\_hs$

$\hat{y} = 77.5 + 11.8 \times 0$

$\hat{y} = 77.5$

if the mother did not complete high school or:

$\hat{y} = 77.5 + 11.8 mom\_hs$

$\hat{y} = 77.5 + 11.8 \times 1$

$\hat{y} = 77.5 + 11.8$

$\hat{y} = 89.3$

if the mother did complete high school. In essence, for single binary feature like mom_hs, the linear regression model is equivalent to calculating the mean for each of the groups separately:

```
[27]: grouped = child_iq.groupby("mom_hs")
      grouped["child_iq"].describe()
```

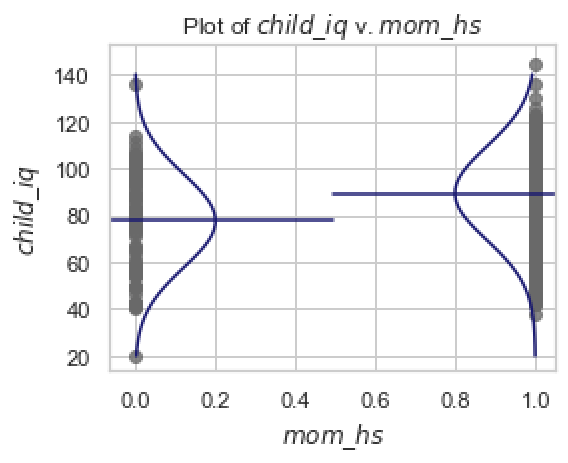
```
[27]:         count       mean        std   min   25%   50%    75%    max
      mom_hs
      0        93.0  77.548387  22.573800  20.0  58.0  80.0   95.0  136.0
      1       341.0  89.319648  19.049483  38.0  77.0  92.0  103.0  144.0
```

We can see that the means in the table above are the same that we calculated using our linear regression model. So linear regression doesn't really *calculate* with categorical features. It "handles" them by separating them into different groups. (The equivalency of the calculations only holds in this simple case, once you introduce additional features, the analogy breaks down).

Let's "plot" the data:

```
[28]: figure = plt.figure(figsize=(4,3))

      beta = result2[ "coefficients"]

      axes = figure.add_subplot(1, 1, 1)
      axes.scatter([0] * child_iq[child_iq.mom_hs == 0].child_iq.size, child_iq[child_iq.
       ↪mom_hs == 0].child_iq, color="dimgray", alpha=0.8)
      axes.axhline(beta[0], xmax=0.5,color="midnightblue")

      axes.scatter([1] * child_iq[child_iq.mom_hs == 1].child_iq.size, child_iq[child_iq.
       ↪mom_hs == 1].child_iq, color="dimgray", alpha=0.8)
      axes.axhline(beta[0] + beta[1], xmin=0.5, color="midnightblue")

      ys = np.linspace(20, 140)
      xs = [stats.norm.pdf( k, loc=beta[0], scale=result2["sigma"]) * 10 for k in ys]
      axes.plot( xs, ys, color="midnightblue")
      print(beta[0])
      ys = np.linspace(20, 140)
      xs = [1.0 - stats.norm.pdf( k, loc=beta[0] + beta[1], scale=result2["sigma"]) * 10␣
       ↪for k in ys]
      axes.plot( xs, ys, color="midnightblue")


      axes.set_ylabel(r"$child\_iq$")
      axes.set_xlabel(r"$mom\_hs$")
      axes.set_title(r"Plot of $child\_iq$ v. $mom\_hs$")

      plt.show()
      plt.close()
```

```
77.5483870967742
```



Here we can see that our model represents two mean values (one for the mother's who did not complete high school and one for the mothers that did). The standard deviation is the same for both because it's the same for the model overall. Note that this is different than what happened above when we calculated the individual means and standard deviations.

This case isn't particularly compelling except that it drives home the idea that linear regression is simply a model of the mean as a function and it sets us up for the next case, the only time we'll have two variables in this chapter.

1.2.7 Linear Regression with a Numerical Feature and Binary Feature

In this section we will see the various ways that a numerical feature and a binary feature (indicator variable) can be combined. This is in anticipation of the next chapter were we look at multivariate models in general.

We can combine the two models above in a number of ways:

$\hat{child\_iq} = \beta_0 + \beta_1 mom\_hs + \beta_2 mom\_iq$

$\hat{child\_iq} = \beta_0 + \beta_1 mom\_iq + \beta_2 mom\_iq \times mom\_hs$

$\hat{child\_iq} = \beta_0 + \beta_1 mom\_hs + \beta_2 mom\_iq + \beta_3 mom\_iq \times mom\_hs$

Let's take the simplest case first.

Numerical Feature and Binary Feature

Remember that mom_hs can only take on two values: 0 or 1. This means that the following equation:

$\hat{child\_iq} = \beta_0 + \beta_1 mom\_hs + \beta_2 mom\_iq$

only has two possibilities as far as mom_hs is concerned. It can either be:

$\hat{child\_iq} = \beta_0 + \beta_1 (mom\_hs = 0) + \beta_2 mom\_iq$

$\hat{child\_iq} = \beta_0 + \beta_1 \times 0 + \beta_2 mom\_iq$

$\hat{child\_iq} = \beta_0 + \beta_2 mom\_iq$

when mom_hs is 0 or it can be:

$\hat{child\_iq} = \beta_0 + \beta_1 (mom\_hs = 1) + \beta_2 mom\_iq$

$$\hat{child\_iq} = \beta_0 + \beta_1 \times 1 + \beta_2 mom\_iq$$

$$\hat{child\_iq} = (\beta_0 + \beta_1) + \beta_2 mom\_iq$$

when $mom\_hs$ is 1. As we saw before, the presence of the binary variable "automagically" creates two models, one for each value. Now let's estimate this model with the $child\_iq$ data:

[29]:
```
result3 = models.linear_regression("child_iq ~ mom_hs + mom_iq", data=child_iq)
print(models.simple_describe_lr(result3))
```

```
Model: child_iq ~ mom_hs + mom_iq
---------  ------  ---
Coefficients             Value
              $\beta_0$  25.73
mom_hs        $\beta_1$  5.95
mom_iq        $\beta_2$  0.56

Metrics       Value
$\sigma$      18.14
$R^2$         0.21
---------  ------  ---
```

Looking at our metrics, the proportion of variation in $child\_id$ explained by the model is about 21.4%. The standard error of the regression is 18.1 IQ points.

We can use our calculations above and insert the values of $\beta_0$, $\beta_1$, and $\beta_2$. The equation when the mother did not graduate high school is:

$$\hat{child\_iq} = \beta_0 + \beta_2 mom\_iq$$

$$\hat{child\_iq} = 25.7 + 0.56 mom\_iq$$

and the equation for when she *did* graduate high school is:

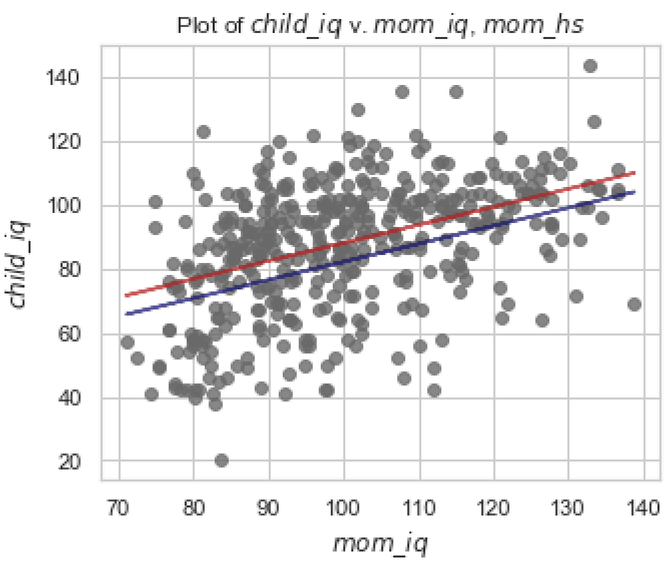
$$\hat{child\_iq} = (\beta_0 + \beta_1) + \beta_2 mom\_iq$$

$$\hat{child\_iq} = (25.7 + 6.0) + 0.56 mom\_iq$$

$$\hat{child\_iq} = 31.7 + 0.56 mom\_iq$$

Graphically, there are two lines, one for each case:

[30]:
```
figure = plt.figure(figsize=(5,4))

axes = figure.add_subplot(1, 1, 1)
axes.scatter( child_iq.mom_iq, child_iq.child_iq, color="dimgray", alpha=0.8)

beta = result3[ "coefficients"]
axes.plot(child_iq.mom_iq, [beta[ 0] + beta[ 2] * x for x in child_iq.mom_iq], '-',␣
↪color="midnightblue", alpha=0.75)
axes.plot(child_iq.mom_iq, [(beta[ 0] + beta[1]) + beta[ 2] * x for x in child_iq.mom_
↪iq], '-', color="firebrick", alpha=0.75)

axes.set_ylabel(r"$child\_iq$")
axes.set_xlabel(r"$mom\_iq$")
axes.set_title(r"Plot of $child\_iq$ v. $mom\_iq$, $mom\_hs$")

plt.show()
plt.close()
```



When the indicator variable is 1, the entire regression line shifts (up in this case because $\beta_1$ is positive).

Numerical Feature and Binary Feature Interaction Term

The second model removes mom_hs by itself and adds the product of mom_iq and mom_hs. This is called an "interaction term" because it captures the "interaction" of the two variables. Mathematically, we start with the following:

$$\hat{child\_iq} = \beta_0 + \beta_1 mom\_iq + \beta_2 mom\_iq \times mom\_hs$$

and when mom_hs is 0, we have:

$$\hat{child\_iq} = \beta_0 + \beta_1 mom\_iq + \beta_2 mom\_iq \times (mom\_hs = 0)$$

$$\hat{child\_iq} = \beta_0 + \beta_1 mom\_iq + \beta_2 mom\_iq \times 0$$

$$\hat{child\_iq} = \beta_0 + \beta_1 mom\_iq$$

which is our regression line without mom_hs. If mom_hs is 1, then we have:

$$\hat{child\_iq} = \beta_0 + \beta_1 mom\_iq + \beta_2 mom\_iq \times (mom\_hs = 1)$$

$$\hat{child\_iq} = \beta_0 + \beta_1 mom\_iq + \beta_2 mom\_iq \times 1$$

$$\hat{child\_iq} = \beta_0 + \beta_1 mom\_iq + \beta_2 mom\_iq$$

$$\hat{child\_iq} = \beta_0 + (\beta_1 + \beta_2) mom\_iq$$

which, as you might imagine, adjusts the *slope* of the line when the mother has completed high school. Let's estimate this model. The Patsy code for an interaction term is "x1:x2".

[31]:
```
result4 = models.linear_regression("child_iq ~ mom_iq + mom_hs:mom_iq", data=child_iq)
print(models.simple_describe_lr(result4))
```

```
Model: child_iq ~ mom_iq + mom_hs:mom_iq
----------  -------  ---
Coefficients              Value
              $\beta_0$   29.77
mom_iq        $\beta_1$   0.53
mom_hs:mom_iq $\beta_2$   0.05
```

(continues on next page)

(continued from previous page)

```

Metrics         Value
$\sigma$        18.18
$R^2$           0.21
----------  -------  ---
```

The R^2 still isn't great but this is illustrative, we're not trying to settle the origins of IQ here.

We can fill in these values and, for $mom\_hs = 0$ we have:

$\hat{child\_iq} = \beta_0 + \beta_1 mom\_iq$

$\hat{child\_iq} = 29.8 + 0.53 mom\_iq$

and for $mom\_hs = 1$, we have:

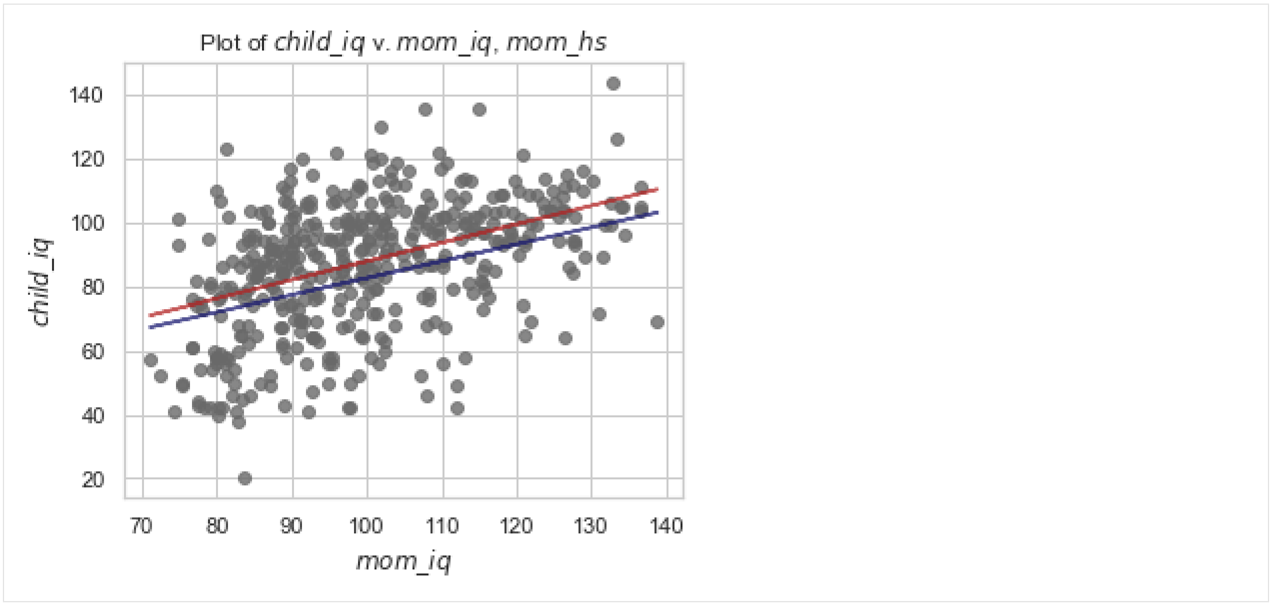
$\hat{child\_iq} = \beta_0 + (\beta_1 + \beta_2) mom\_iq$

$\hat{child\_iq} = 29.8 + (0.53 + 0.05) mom\_iq$

$\hat{child\_iq} = 29.8 + 0.58 mom\_iq$

Graphically, this is as follows:

[32]:
```
figure = plt.figure(figsize=(5,4))

axes = figure.add_subplot(1, 1, 1)
axes.scatter( child_iq.mom_iq, child_iq.child_iq, color="dimgray", alpha=0.8)

beta = result4[ "coefficients"]
axes.plot(child_iq.mom_iq, [beta[ 0] + beta[ 1] * x for x in child_iq.mom_iq], '-',␣
↪color="midnightblue", alpha=0.75)
axes.plot(child_iq.mom_iq, [beta[ 0] + (beta[1] + beta[ 2]) * x for x in child_iq.mom_
↪iq], '-', color="firebrick", alpha=0.75)

axes.set_ylabel(r"$child\_iq$")
axes.set_xlabel(r"$mom\_iq$")
axes.set_title(r"Plot of $child\_iq$ v. $mom\_iq$, $mom\_hs$")

plt.show()
plt.close()
```



They have the same intercept but very slightly different slopes.

Both

Of course, as we saw in the 3rd equation, we can combine both of these. We will speak more about interaction terms in the next chapter. If we start with the full equation:

$\hat{child\_iq} = \beta_0 + \beta_1 mom\_hs + \beta_2 mom\_iq + \beta_3 mom\_iq \times mom\_hs$

and set mom_hs to zero, we end up with a simplified version:

$\hat{child\_iq} = \beta_0 + \beta_2 mom\_iq$

If, however, $mom\_hs = 1$, then we end up shifting both the intercept and the slope:

$\hat{child\_iq} = (\beta_0 + \beta_1) mom\_hs + (\beta_2 + \beta_3) mom\_iq$

Let's estimate this model:

```
[33]: result5 = models.linear_regression("child_iq ~ mom_hs + mom_iq + mom_hs:mom_iq",␣
      ↪data=child_iq)
      print(models.simple_describe_lr(result5))
```

```
Model: child_iq ~ mom_hs + mom_iq + mom_hs:mom_iq
---------  ------  ----
Coefficients               Value
               $\beta_0$  -11.48
mom_hs         $\beta_1$  51.27
mom_iq         $\beta_2$  0.97
mom_hs:mom_iq  $\beta_3$  -0.48

Metrics        Value
$\sigma$       17.97
$R^2$          0.23
---------  ------  ----
```

The R^2 went up just a little bit. We'll talk more about that later. R^2 is a non-decreasing function of the number of features in the model. This means that adding another feature will not (generally) hurt the model and will probably help even if the feature doesn't belong there!

Substituting into our equations above, we have (for children whose mothers that did not graduate high school):

$\hat{child\_iq} = \beta_0 + \beta_2 mom\_iq$

$\hat{child\_iq} = -11.48 + 0.97 mom\_iq$

and for when mothers did graduate high school ($mom\_hs = 1$):

$\hat{child\_iq} = (\beta_0 + \beta_1) mom\_hs + (\beta_2 + \beta_3) mom\_iq$

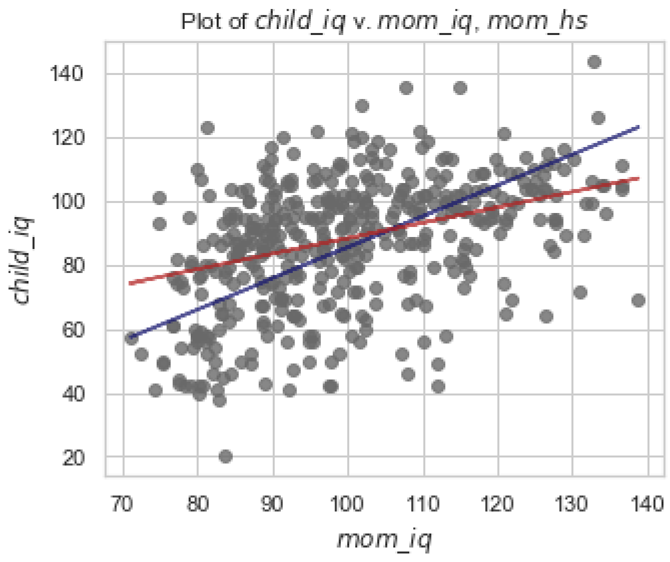
$\hat{child\_iq} = (-11.48 + 51.3) + (0.97 - 0.48) mom\_iq$

$\hat{child\_iq} = (-11.48 + 51.3) + (0.97 - 0.48) mom\_iq$

$\hat{child\_iq} = 39.82 + 0.49 mom\_iq$

How are we to interpret these models? For children whose mothers did not graduate high school, the base IQ starts low and increases fairly fast (nearly 1 to 1) for each IQ point of the mother. For children whose mothers did graduate high school, the base IQ start higher and only increases about 1 for every 2 IQ points of the mother. Graphically,

```
[34]: figure = plt.figure(figsize=(5,4))

axes = figure.add_subplot(1, 1, 1)
axes.scatter( child_iq.mom_iq, child_iq.child_iq, color="dimgray", alpha=0.8)

beta = result5[ "coefficients"]
axes.plot(child_iq.mom_iq, [beta[ 0] + beta[ 2] * x for x in child_iq.mom_iq], '-',␣
↪color="midnightblue", alpha=0.75)
axes.plot(child_iq.mom_iq, [(beta[ 0] + beta[1]) + (beta[ 2] + beta[3]) * x for x in␣
↪child_iq.mom_iq], '-', color="firebrick", alpha=0.75)

axes.set_ylabel(r"$child\_iq$")
axes.set_xlabel(r"$mom\_iq$")
axes.set_title(r"Plot of $child\_iq$ v. $mom\_iq$, $mom\_hs$")

plt.show()
plt.close()
```



It's almost like there are two models (although the results are not quite the same as if you had estimated them separately because the two "models" share a common σ, which is jointly minimized).

1.2.8 Bayesian Inference with the Bootstrap

As is true of all our modeling efforts, the data we observed is consistent with a wide variety of different parameter values. Although in the context of y, we take the βs as given, they are just estimates. Although we would be correct to say:

$\hat{y} = \hat{\beta_0} + \hat{\beta_1} x$

nobody does ("we estimate with estimates").

We can use our domain knowledge to interpret the coefficients in context but we need something else to determine how credible the coefficients are.

In order to determine credibility bounds, we can once again turn to Bayesian inference. Unfortunately, Bayesian inference is more complicated with these more complicated models but, as before, the Bootstrap comes to the rescue. We can generate bootstrap samples from our data, fit our model to each sample, and use those estimates to generate posterior distributions for the coefficients.

Not only that, we can do a bit better than Frequentist statistics and generate posterior distributions for metrics such as σ and R^2.

[35]:
```
result6 = models.bootstrap_linear_regression("child_iq ~ mom_hs + mom_iq + mom_hs:mom_
↪iq", data=child_iq)
print(models.describe_bootstrap_lr(result6))
```

```
Model: child_iq ~ mom_hs + mom_iq + mom_hs:mom_iq
---------  ------  ----  -----  ---
                                  95% BCI
Coefficients                Mean    Lo        Hi
               $\beta_0$  -11.48  -36.86    15.67
mom_hs         $\beta_1$  51.27   21.71     82.11
mom_iq         $\beta_2$  0.97    0.68      1.25
mom_hs:mom_iq  $\beta_3$  -0.48   -0.82     -0.15

Metrics        Mean       Lo      Hi
$\sigma$       17.97      16.99   19.25
$R^2$          0.23       0.19    0.30
---------  ------  ----  -----  ---
```

How do we interpret these (Bayesian) Credible Intervals? Just as we did before. The estimated value of β_0 is -11.48 but there is a 95% probability that the value is between -44.1 and 12.1. The estimated value of β_1 is 52.3 but there is a 95% probability that the value is between 18.9 and 89.6. The estimated value of β_2 is 0.97 but there is a 95% probability the value is between 0.72 and 1.31. Finally, we estimated the value of β_3 as -0.48 but there is a 95% probability that the value is between -0.89 and -0.16.

The width of these bounds is some indication of why our model is so poor. But we will leave it at that for now. In traditional terms, there is only one coefficient whose credible interval contains zero but it is the intercept so it isn't cause for concern.

We can also ask better questions than just "is this coefficient zero or not zero?". If we started with a Causal Loop Diagram, we know what we expect the signs on the coefficients to be. For example, perhaps we expect $mom\_hs = 1$ to be a general overall boost to the prediction of the child's IQ regardless of the mother's *measured* IQ. Understanding what this means in terms of the model we have chosen (linear regression) then our expectation is that $\beta_1 + \beta_0 > \beta_0$ or that $\beta_1 > 0$. We can answer that question. Based on the data, what is the probability that β_1 is positive?

[36]:
```
print('P(mom_hs = 1 > 0)', np.mean(result6['resampled_coefficients']['mom_hs'] > 0))
```

```
P(mom_hs = 1 > 0) 1.0
```

Based on the data, the conjecture has a probability of 99%. And similarly, for the slope. We generally believe that the child's IQ will, on average, increase with the mother's IQ. This seems perfectly reasonable. We do have to be careful though with "regression to the mean". Linear regression got its name from the phenomena identified by Galton. He noted that adults with above average heights tended to have children with average heights and adults with below average heights tended to have children with average heights.

We should then expect that above average IQs would lead to average IQs and similarly for below average IQs. Still, the slope is not going to be *negative*:

```
[37]: print('P(mom_hs = 1 > 0)', np.mean(result6['resampled_coefficients']['mom_iq'] > 0))
```

```
P(mom_hs = 1 > 0) 1.0
```

And here we can see that, based on the data, we can be nearly certain that there is a positive relationship between the mother's IQ and the child's.

Additionally, we have credible bounds on the error of the regression (σ) as 16.8 to 19.0 and on the coefficient of determination (R^2) of 16.4% to 29.4%.

The Bootstrap gives us some additional benefits. In the results above, we have a credible intervals for each coefficient. But, as we noted, these coefficients actually combine to form a shift intercept and a shifted slope.

With traditional Frequentist confidence intervals, there is no way to combine the confidence intervals to get an overall confidence interval for the intercept ($\beta_0 + \beta_1$) nor is there a way to get a confidence interval for the overall slope ($\beta_2 + \beta_3$).

With the bootstrap samples we have, we can estimate those posterior distributions very easily. The first one is:

```
[38]: (result6[ "resampled_coefficients"]["intercept"] +
       result6[ "resampled_coefficients"]["mom_hs"]).quantile( [0.025, 0.975])
```

```
[38]: 0.025    28.525660
      0.975    50.896568
      dtype: float64
```

This represents the 95% credible interval for the combined β_0 and β_1 intercept and the second one is:

```
[39]: (result6[ "resampled_coefficients"]["mom_iq"] +
       result6[ "resampled_coefficients"]["mom_hs:mom_iq"]).quantile( [0.025, 0.975])
```

```
[39]: 0.025    0.374122
      0.975    0.599503
      dtype: float64
```

which represents the 95% credible interval for the combined β_2 and β_3 slope.

We will have more to stay about statistical inference and model building later but let's first examine the second problem of supervised learning: classification.

```
[1]: import warnings
     warnings.filterwarnings('ignore')
```

```
[2]: %matplotlib inline
```

```
[3]: import numpy as np
     import scipy.stats as stats
     import seaborn as sns
     import matplotlib.pyplot as plt
     import pandas as pd
```

(continues on next page)

(continued from previous page)

```
import random
import patsy

import sys
sys.path.append('resources')
import models

sns.set(style="whitegrid")
```

1.3 Logistic Regression

In the last section, we looked at how we can use a linear model to fit a numerical target variable like child IQ (or price or height or weight). Regardless of the method you use, this is called a *regression problem*. Linear regression is one way of solving the regression problem.

When your target variable is a categorical variable, this is a *classification problem*. For now we'll work only with the case where there are two outcomes or labels.

With our numerical y, we started with a model:

$\hat{y} = N(\beta_0, \sigma)$

with the innovation that we could replace the mean in the normal distribution with a linear function of features, $f(X)$.

We can do the same thing for classification. If we have a binary categorical variable y, it has a Bernoulli distribution with probability p as the parameter. We can estimate y as:

$\hat{y} = p$

or the fraction of "successes" in the data. But what if we did the same thing as before? What if p was a function of additional features? We would have:

$\hat{y} = \beta_0 + \beta_1 x$

and we would have a model that represented how the probability of y changes as x changes. Although this sounds good, there is a problem. $\beta_0 + \beta_1 x$ is not bounded to the range (0, 1) which we require for probabilities. But it does turn out that there is a solution: we can use a transformation to keep the value in the range (0, 1), the logistic function.

1.3.1 The Logistic Function

The logistic function is:

$logistic(z) = logit^{-1}(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}$

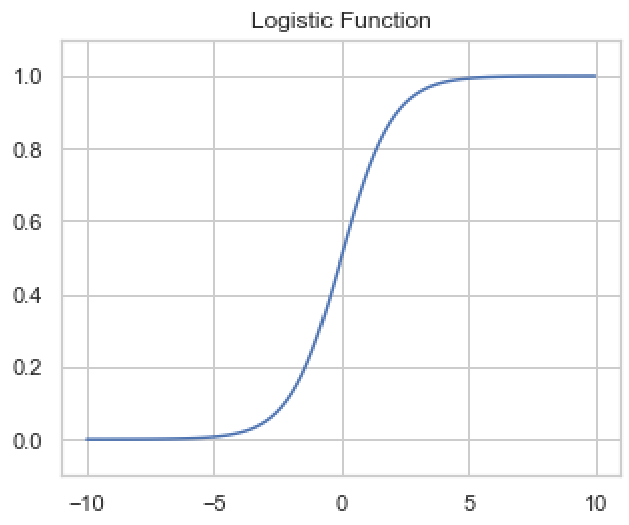
And it looks like the following:

[4]:
```
def logistic( z):
    return 1.0 / (1.0 + np.exp( -z))

figure = plt.figure(figsize=(5,4))

axes = figure.add_subplot(1, 1, 1)

xs = np.linspace( -10, 10, 100)
ys = logistic( xs)
```

(continues on next page)

(continued from previous page)

```
axes.plot( xs, ys)
axes.set_ylim((-0.1, 1.1))
axes.set_title("Logistic Function")
plt.show()
plt.close()
```



No matter what the value of x, the value of y is always between 0 and 1 which is exactly what we need for a probability.

There are a few additional things to note at this point. First, there is not a single definition of logistic regression. Gelman defines logistic regression as:

$P(y = 1) = logit^{-1}(\beta_0 + \beta_1 x)$

in terms of the inverse logit function. Such a function returns the probability that $y = 1$. There are other possibilites (for example, a general maximum entropy model).

Second, interpreting the coefficients becomes a bit of a problem. Let's assume that we have no features and only have:

$P(y = 1) = logit^{-1}(\beta_0)$

what, exactly, is β_0? It's not a probability because the probability interpretation only takes place once we have transformed the result using the inverse logit function. We take note of the following truism:

$logit^{-1}(logit(p)) = p$

This is simply what it means to be an inverse function of some other function. But the interesting thing is that this means that:

$\beta_0 = logit(p)$

and we do know what $logit(p)$ is, it's the $log\ odds$. $logit$ is defined as:

$logit(p) = log(\frac{p}{1-p})$

if p is the probability of an event, then $\frac{p}{1-p}$ is the log odds of the event (the ratio of the probability for an event and the probability against the event).

The third difficulty is that the logistic regression is non-linear. For linear regression, the slopes of the curve (a line) are constant (the βs) and while logistic regression has a linear predictor, the result is non-linear in the probability space. For example, a 0.4 point increase in log odds from 0.0 to 0.4 increases probability from 50% to 60% but a 0.4 point increase in log odds from 2.2 to 2.6 only increases probability from 90% to 93%.

Finally, we lose a lot of our ability to visualize what's going on when we moved from linear regression to logistic regression.

1.3.2 What Loss does Logistic Regression minimize?

Several times now, we've turned the question of variance on its head and asked, "what estimate minimizes my error?". For a single prediction, β_0, of a numerical variable, we know that we want some value that minimizes Mean Squared Error (MSE):

$MSE = \frac{1}{n}\sum(y - \beta_0)^2$

and that this means our prediction of β_0 should be the mean, $\bar{y}$. This is also true for *linear* regression where we minimize MSE:

$MSE = \frac{1}{n}\sum(y - \hat{y})^2$

Also note that we can use MSE to *evaluate* linear regression (it, or some variant, is really the only way we have to evaluate linear regression's predictions).

We do not use MSE for logistic regression, however, mostly because we want something with a better first derivative. Instead of MSE, we often use *cross entropy* (also called *log loss*, we'll stick with cross entropy):

$L(\beta) = -\frac{1}{n}\sum ylog(\hat{y}) + (1-y)log(1-\hat{y})$

This has several implications:

1. Just because "regression" is in the name, we do not use Mean Squared Error to derive or evaluation logistic regression.
2. Although we do use cross entropy to derive logistic regression, we do *not* use it to evaluate logistic regression. We tend to use error rate and other metrics to evaluate it (which we will discuss in a few chapters). For now, we will just use error rate.

1.3.3 Logistic Regression with Continuous Feature (Synthetic Data)

As before, we're going to start with synthetic data to get our proverbial feet wet. Even here, generating synthetic data isn't as easy as it is for linear regression. We basically need to estimate the p for each value of x and then simulate it. The algorithm is something like this:

```
1. generate x using the standard normal distribution or binomial if categorical.
2. for each data point:
3.     z = beta_0 + beta_1 * x
4.     pr = 1/(1_exp(-z))
5.     y = 1 if rand() < pr else 0
```

Of note, the logistic function does not output $\hat{y}$ as it does with linear regression. It outputs the estimated probability of $y = 1$. We can take that probability and compare it to a threshold and assign $y = 0$ or $y = 1$. The y above is the *real* y for the synthetic data.

[5]:
```
np.random.seed(83474722)
```

[6]:
```
data = {}
data["x"] = stats.norm.rvs(0, 1, 100)
data["z"] = 0.5 + data["x"] * 0.5
data["pr"] = list(map(lambda z: logistic(z), data["z"]))
data["y"] = list(map(lambda pr: 1 if np.random.uniform() < pr else 0, data["pr"]))
data = pd.DataFrame(data)
```

It's worth taking a bit more in-depth look at this data even though it's synthetic (or *because* it's synthetic). We generated x from the standard Normal distribution: $x \sim N(0, 1)$. z, an intermediate step, is the actual linear model: $z = \beta_0 + \beta_1 x$ or $z = 0.5 + 0.5x$.

As the earlier discussion mentions, we pass z through the logistic function to bound it to the interval (0, 1). The result, pr, represents a probability. This is a conditional probability: $P(y = 1|x)$. In order to find out the "true" y for each x, we simulate that probability.

As we can see in the table below, we have $pr = 0.663$ and $y = 1$ (obs 1) as well as $pr = 0.857$ and $y = 0$. This is logistic regression's form of "error", "noise", or the "known unknowns and unknown unknowns".

[7]:
```
data.head()
```

[7]:
```
          x         z        pr  y
0  0.355745  0.677873  0.663264  1
1  0.412298  0.706149  0.669550  1
2 -0.830253  0.084874  0.521206  1
3  2.579133  1.789567  0.856874  0
4 -1.067772 -0.033886  0.491529  0
```

We could use a constant model as we have before:

[8]:
```
np.mean(data.y)
```

[8]:
```
0.57
```

No matter what x is, we say there's a 57% probability that the value of y is 1. Since 57% is over 50%, we could just guess that for any x, $\hat{y} = 1$. We would be right 57 of the time and wrong 43% of the time on average. 43% is the model's *error rate*.

Can we do better?

[9]:
```
result = models.logistic_regression("y ~ x", data = data)
```

[10]:
```
print(models.simple_describe_lgr(result))
```

```
Model: y ~ x
---------  ------  ---
Coefficients             Value
               $\beta_0$  0.25
x              $\beta_1$  0.22

Metrics        Value
Error ($\%$)   42.00
Efron's $R^2$  0.01
---------  ------  ---
```

We do a *little* better. The error rate here is 42% instead of (1-0.57) or 43% but that's not very encouraging. Logistic Regression doesn't actually *have* an R^2 metric. What we have shown here is Efron's Pseudo R^2. It basically measures the same thing as interpretation #1 of the "real" R^2: it's the percent of the variability in y explained by the model. Not very much.

Additionally, our estimates of the coefficients, β_0 and β_1, are pretty bad compared to the ground truth in the synthetic data. In the linear regression case we were able to recover them fairly easily. Why is the synthetic data so bad?

Note that our base probability is not really much different than a coin toss (57% versus 50%). Assume a given x leads to a probability of 65%. We need a lot more examples of x to calculate that 65%...if we only have a few, we may never actually observe the case where $y = 1$.

What happens with the current data generator if we just generate more data, n=10,000 instead of n=100?

```
[11]: data = {}
      data["x"] = stats.norm.rvs(0, 1, 10000)
      data["z"] = 0.5 + data["x"] * 0.5
      data["pr"] = list(map(lambda z: logistic(z), data["z"]))
      data["y"] = list(map(lambda pr: 1 if np.random.uniform() < pr else 0, data["pr"]))
      data = pd.DataFrame(data)
```

We can re-run our logistic regression on this data:

```
[12]: result1 = models.logistic_regression("y ~ x", data = data)
      print(models.simple_describe_lgr(result1))
```

```
Model: y ~ x
---------  ------  ---
Coefficients              Value
               $\beta_0$  0.48
x              $\beta_1$  0.49

Metrics        Value
Error ($\%$)   36.28
Efron's $R^2$  0.05
---------  ------  ---
```

Our coefficient estimates are almost exactly the same as the ground truth. Still our error rate is 36.3% instead of 43.0%. This is probably as good as we can get. What happens if bump up the base probability a bit?

```
[13]: data = {}
      data["x"] = stats.norm.rvs(0, 1, 10000)
      data["z"] = 0.75 + data["x"] * 10
      data["pr"] = list(map(lambda z: logistic(z), data["z"]))
      data["y"] = list(map(lambda pr: 1 if np.random.uniform() < pr else 0, data["pr"]))
      data = pd.DataFrame(data)
```

Let's look at this data. The probabilities of each observation are now either very near 0 or very near 1:

```
[14]: data.head()
```

```
[14]:           x         z        pr  y
      0  0.875917  9.509174  0.999926  1
      1 -0.537053 -4.620530  0.009752  0
      2  0.809259  8.842593  0.999856  1
      3 -0.632188 -5.571878  0.003789  0
      4 -0.757485 -6.824855  0.001085  0
```

The constant model shows a probability of 52.2% for $y = 1$. This means it has an error rate of 47.8%!

```
[15]: np.mean(data.y)
```

```
[15]: 0.5215
```

What about our logistic regression model?

```
[16]: result2 = models.logistic_regression("y ~ x", data = data)
      print(models.simple_describe_lgr(result2))
```

```
Model: y ~ x
---------  ------  ---
Coefficients              Value
               $\beta_0$  0.69
```

(continues on next page)

(continued from previous page)

```
x               $\beta_1$   9.45

Metrics         Value
Error ($\%$)    5.09
Efron's $R^2$   0.85
---------  ------  ---
```

The coefficients are almost exact *and* the error rate is only 5.1%. The (pseudo) R^2 shows that our model explains 85% of the variation in y.

This set of experiments shows us a number of things. First, generating synthetic data is very useful for learning how your algorithms work. In fact, let's do one more experiment. Let's reduce the number of observations back to 100:

[17]:
```
data = {}
data["x"] = stats.norm.rvs(0, 1, 10000)
data["z"] = 0.75 + data["x"] * 10
data["pr"] = list(map(lambda z: logistic(z), data["z"]))
data["y"] = list(map(lambda pr: 1 if np.random.uniform() < pr else 0, data["pr"]))
data = pd.DataFrame(data)
```

[18]:
```
result3 = models.logistic_regression("y ~ x", data = data)
print(models.simple_describe_lgr(result3))
```

```
Model: y ~ x
---------  ------  ---
Coefficients                Value
                $\beta_0$   0.64
x               $\beta_1$   8.93

Metrics         Value
Error ($\%$)    5.77
Efron's $R^2$   0.83
---------  ------  ---
```

Here our error rate is still quite a bit lower but the estimates of our coefficients aren't as good. We need both a lot of data and clear underlying pattern *and* this pattern isn't as obvious as it is with linear regression.

1.3.4 Logistic Regression with Real Data

When it comes to either numerical features or a binary categorical features, there is no difference between linear regression and logistic regression. We can have numerical features which will affect the slope and intercept of the line. We can have binary categorical features that will affect the intercept of the line. We can have interaction terms that will affect the slope of the line.

The main difference with linear regression is in the interpretation of the coefficients.

For logistic regression, the coefficients are log-odds and while some people are quite comfortable thinking in terms of log-odds, most are not. How do we convert them into something we can understand?

Let's begin the discussion by looking at real data. This data is from a study of villager behavior in Bangladesh. Wells were examined for natural arsenic contamination and villagers using wells with higher arsenic readings were encouraged to use other wells or dig new ones. The variables are:

- **switch** - yes (1) or no (0), did the respondent switch to a new well.
- **dist** - distance to the nearest safe well in meters.
- **arsenic** - arsenic level of the respondent's well.

- **assoc** - does the respondent or a family member belong to a community association.
- **educ** - the educational attainment of the respondent in years.

Let's start out with a logistic regression model for $\hat{switch}$:

$P(\hat{switch} = 1) = logistic^{-1}(\beta_0 + \beta_1 dist)$

although we really have something like:

$z = \beta_0 + \beta_1 dist$

$\hat{pr} = \frac{1}{1+e^{-z}}$

$\hat{y}$ = 1 if $\hat{pr}$ > 0.5 else 0

which is a bit more complicated to write each time.

```
[19]: wells = pd.read_csv( "resources/arsenic.wells.tsv", sep=" ")
```

Let's check the representations:

```
[20]: wells.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3020 entries, 1 to 3020
Data columns (total 5 columns):
switch     3020 non-null int64
arsenic    3020 non-null float64
dist       3020 non-null float64
assoc      3020 non-null int64
educ       3020 non-null int64
dtypes: float64(2), int64(3)
memory usage: 141.6 KB
```

There is nothing particularly startling here. Let's see a few values:

```
[21]: wells.head()
```

```
[21]:
   switch  arsenic       dist  assoc  educ
1       1     2.36  16.826000      0     0
2       1     0.71  47.321999      0     0
3       0     2.07  20.966999      0    10
4       1     1.15  21.486000      0    12
5       1     1.10  40.874001      1    14
```

The base model (and error rate) are:

```
[22]: mean = np.mean(wells.switch)
      print("P(switch=1) = {0:.2f} ({1:.2f})".format(mean, 1-mean))
```

```
P(switch=1) = 0.58 (0.42)
```

The base model (often called the "null" model) is that $P(switch = 1) = 0.58$ which leads to an error rate of 42%. Let's see what logistic regression can get us:

```
[23]: result = models.logistic_regression( "switch ~ dist", data = wells)
      print(models.simple_describe_lgr(result))
```

```
Model: switch ~ dist
---------  ------  ---
Coefficients            Value
```

(continues on next page)

(continued from previous page)

```
                $\beta_0$  0.60
dist            $\beta_1$  -0.01

Metrics         Value
Error ($\%$)    40.53
Efron's $R^2$   0.01
---------  ------  ---
```

So again, this is the real world and real data and sometimes you only get improvements such as these. Despite the showing on this data, logistic regression is a very powerful modeling technique.

We can see that the error rate and (pseudo) R^2 of this model aren't great but we're much more interested in interpreting the model coefficients. What do they mean?

Intercept

The intercept in this case has a legitimate $dist = 0$ interpretation. If the alternative well is 0 meters away, what is the probability of switching?

We can use our previous identity and use the inverse logit (logistic) function:

```
[24]: logistic( 0.6038)
```

```
[24]: 0.6465252043640823
```

so the probability of switching is 64.7% if the safe well is zero meters away (that doesn't really bode well). If we were to run the logistic regression without any regressors, we could think of β_0 as the *prior* log odds. In essence, logistic regression is a function that calculates conditional probabilities based on the features instead of using a table.

Once you add features, β_0 is no longer a pure prior because it has been optimized in the presence of the other features and therefore is still a conditional probability. . . just with all the features at 0.

Coefficients

Next, we can look at each coefficient (or in this case, the only coefficient). The model basically says there is a 0.0062 decrease in *log odds* (remember that the coefficients are not in probability space until transformed) for every meter to the nearest safe well. Now we have a problem. While β_i as *log odds* is linear in x_i (that's *all* the term "linear model" means), arbitrary transformations of β_i, $t(\beta_i)$ is not necessarily linear in x_i and that is the case here. How do we get around this problem?

There are several options:

No. 1 - Evaluate at the mean of the variable with a unit change.

The mean value of dist(ance) is 48.33. If we evaluate our model using that value, we get:

$P(switch = 1)$ = $logit^{-1}$(0.6038 – 0.0062 × 48.33) = $logit^{-1}$(0.304154) = 0.5755

And if we do the same thing again after adding 1 meter to the average distance, we get:

$P(switch = 1)$ = $logit^{-1}$(0.6038 – 0.0062 × 49.33) = $logit^{-1}$(0.297954) = 0.5739

So. . . that's a decrease of about 0.0016 percentage points (or 0.27%) which isn't huge but then we only increased the difference by a little over 3 feet!

But, you need to be careful with mean scaled data (which we'll talk about in the next chapter). A unit change is equal to one entire standard deviation which may be an extremely large value. . . or an extremely small one.

```
[25]: a = logistic( 0.6038 - 0.0062 * 48.33)
      b = logistic( 0.6038 - 0.0062 * 49.33)
      print(a, b)
      print(a - b)
```

```
0.5754576812550608 0.5739422790895422
0.0015154021655185979
```

If you have more than one x_i, you should set all of them to their mean values and then do a unit change for each variable separately.

Note that this is a good reason to mean *center* data in a logistic regression but not to mean *scale* it. The reason for not mean scaling is that:

1. The coefficients do have clear interpretations and relative magnitudes (changes in probability after transformation).
2. Mean *scaling* makes 1 unit equal to one standard deviation of the standard normal distribution. This might be a very, very large value in the variables actual domain which messes up the approximation.

No. 2 - Calculate the derivative of the logistic and evaluate it at the mean of the variable.

$\frac{\partial}{\partial x_i} logit^{-1}(\beta X) = \frac{\beta_i e^{\beta X}}{(1+e^{\beta X})^2}$

but βX (z) is just the log odds at the mean values of X (if X is indeed $[1.0, \bar{x}_1, \bar{x}_2, ..., \bar{x}_n]$) so if we plug our value for the model evaluated at the mean into the derivative, we get:

$\frac{0.0062 e^{0.0062 \times 0.3042}}{(1+e^{0.0062 \times 0.3042})^2} = 0.0016$

```
[26]: def logistic_slope_at( beta, z):
          return (beta * np.exp( beta * z)) / (1.0 + np.exp( beta * z))**2

      print(logistic_slope_at( 0.0062, 0.3042))
```

```
0.0015499986216064004
```

No. 3 - Divide by 4 Rule

$\beta_1/4 = 0.0062/4 = 0.00155$

It's just a rule of thumb but easy. It has the same general interpretation though. . . the change in probability from a unit change at the mean of the regressor. Again, this approach can get very funky with mean scaled data because a unit change is a full standard deviation which can actually be enormous or infinitesimal. Why does that work?

The slope of the logistic curve is maximized where the first derivative is zero or $\beta_0 + \beta_1 + x = 0$. We can solve for this as:

$\frac{beta_1 e^0}{(1+e^0)^2}$

$\frac{\beta_1 \times 1}{(1+1)^2}$

$\frac{\beta_1}{4}$

This interpretation holds best in the context of mean values for the corresponding feature, x.

No. 4 - Average Predictive Difference

We can also average the probabilities over all of our data points for a specific change in each of our predictors. For a model with only one predictor, this amounts to the same thing as No. 1 so we will save this for later.

1.3.5 Useful Transformations

We'll talk more about transformations in the next chapter. The main point of this chapter is to establish what linear and logistic regression are and how to interpret them. However, there is an especially useful transformation for logistic regression and that involves transforming the units.

This doesn't affect the quality of the model at all. It does, however, change how you interpret it. For example, the current units of *dist* are meters. The probability of change per *meter* is pretty small. But what about the probability of changing per *ten meters*? That's nearly 33 feet.

```
[27]: wells["dist10"] = wells["dist"]/10
```

```
[28]: result = models.logistic_regression( "switch ~ dist10", data = wells)
      print(models.simple_describe_lgr(result))
```

```
Model: switch ~ dist10
---------  ------  ---
Coefficients              Value
               $\beta_0$  0.60
dist10         $\beta_1$  -0.06

Metrics        Value
Error ($\%$)   40.53
Efron's $R^2$  0.01
---------  ------  ---
```

We can now reinterpret the model. The probability of switching decreases by 1.5% (-00.619/4 = -0.015475 or "Divide by 4" rule) at the average when the distance to the safe well increases by 10 meters.

It's interesting to note that Gelman used 100 to scale his data. The problem with this, in an interpetability sense, is that if you look at distance, the median distance to a safe well is 36.7 meters. The 3rd quartile is 64 meters. A 100 meter difference just doesn't figure prominently into the data even though the maximum distance was 339.5 meters. 10 meters seems like a reasonable unit in this case.

Again, this doesn't change how good the model is. But it does it easier to talk about than "15 100ths of a percent per meter".

1.3.6 Plotting Logistic Regression

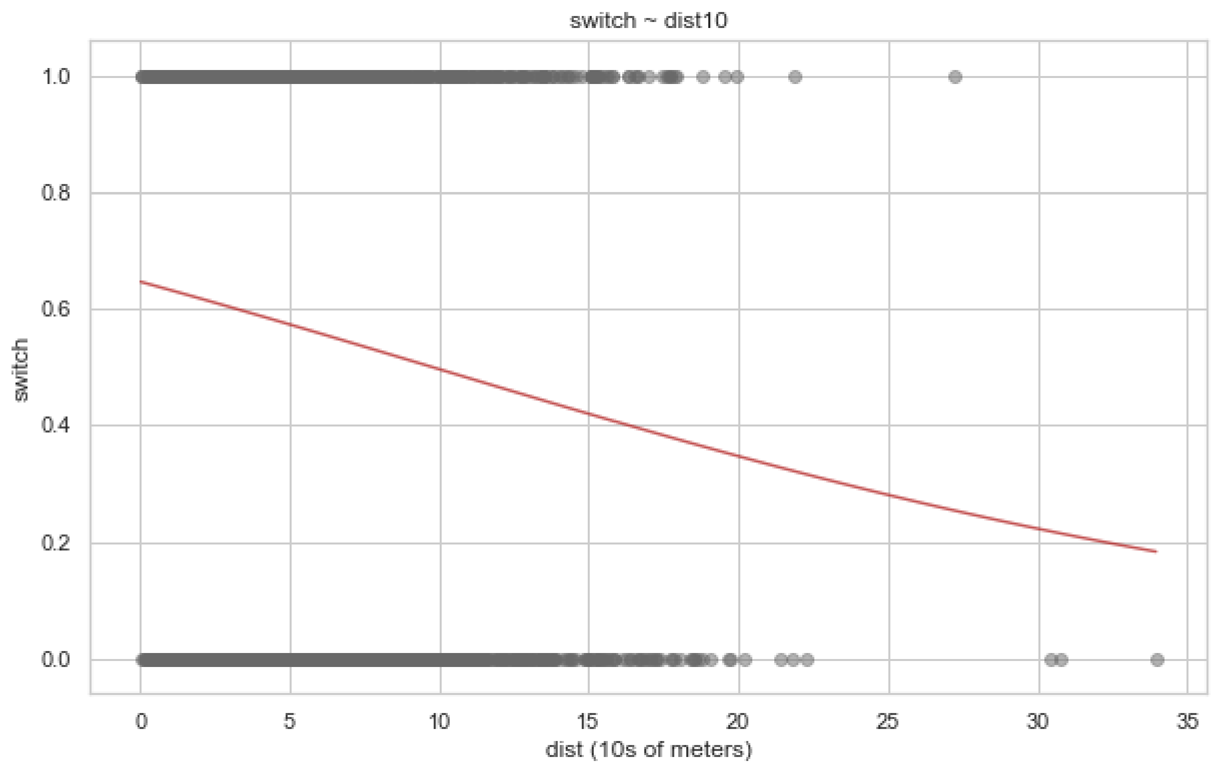
It's not quite as easy to plot a logistic regression as it is linear regression. If we just plot the data, we have:

```
[29]: figure = plt.figure(figsize=(10,6))

      axes = figure.add_subplot(1, 1, 1)

      xs = wells[ "dist10"]
      ys = wells[ "switch"]
      axes.scatter( xs, ys, color="dimgray", alpha=0.5)
      betas = result[ "coefficients"]

      zs = np.linspace( xs.min(), xs.max(), 100)
      ps = [logistic( betas[ 0] + betas[ 1] * x) for x in zs]

      axes.plot(zs, ps, '-', color="firebrick", alpha=0.75)
      axes.set_title( result[ "formula"])
      axes.set_xlabel("dist (10s of meters)")
      axes.set_ylabel("switch")
```

(continues on next page)

(continued from previous page)

```
plt.show()
plt.close()
```



It's just not very interesting or informative on its own. Additionally, you have the problem that logistic regression is nonlinear. We'll get into plotting multivariate regression (linear and logistic) in the next chapter. The solution for logistic regression is usually to plot the *decision boundary* in feature space and not to plot the target at all.

1.3.7 Bootstrap Inference

As with linear regression, we can also apply bootstrap inference to logistic regression and with the same results. Here we only show the function in operation. We also make the innovation that we include "divide by 4" interpretations of our coefficients:

```
[30]: result = models.bootstrap_logistic_regression("switch ~ dist10", wells)
      print(models.describe_bootstrap_lgr(result, 3))
```

```
Model: switch ~ dist10
---------  ------  ----  -----  ----  ----
                                95% BCI
Cofficients                Mean    Lo        Hi       P(y=1)
               $\beta_0$  0.604    0.493     0.715    0.647
dist10         $\beta_1$  -0.062   -0.081    -0.045   -0.015

Metrics        Mean       Lo       Hi
Error ($\%$)   40.530     38.970   42.302
Efron's $R^2$  0.014      0.007    0.022
---------  ------  ----  -----  ----  ----
```

How do we interpret this table? Let's look at "dist10" (AKA β_1).

The mean value is -0.062 *log odds*. Every ten meters we move away from the better well, the *log odds* of switching decrease by 0.062. Looking at "Lo" and "Hi", there is a 95% probability the value is in the range of -0.081 and 0.045. Put differently, and perhaps a bit more Bayesian, the data are consistent with a range of values for "dist10". The values having at least 95% credibility, are in the range -0.081 and -0.045. Finally, using the "divide by 4" rule, the probability of switching, P(y=1), decreases by 1.5 percentage *points* for every 10 meters we move away from the better/safe well.

If we had the visualization skills, we might just show a tiny histogram of the posteriors for each coefficient/metric instead of using intervals.

The development so far has been pedagogical. You should always do bootstrap inference for both linear and logistic regression and going forward, we will.

1.3.8 More than Two Outcomes

Binary classification shows up quite a bit: does it fail or not, does he have a heart attack or not, does she purchase it or not, does he click on it or not. But in many instances, the event is not actually binary. Death is inevitable. It's not "does he have a heart attack or not" but "does he die from a heart attack, cancer, flu, traffic accident, . . . ". It's not "does she purchase it or not" but "does she buy the shoes, the shop-vac, the weedwacker, . . . ". It's not "does he click on it or not" but "does he click on this, does he click on that, does he go back, . . . ".

This gives us some clue as to how to deal with multiclass classification problems.

First, there are classification algorithms that *can* deal with multiclass problems "directly". Decision trees are a good example.

Second, every algorithm that can handle only binary classification can also be made to handle multiclass classification. If the response variable has n possible outcomes then you can train n binary models where each is trained on "the class" and "not the class". For example, if there were three classes: buy, sell, hold. Then you first convert your data (temporarily) into "buy/not buy" and train a model, then convert to "sell/not sell" and train a model, then convert to "hold/not hold" and train a model:

$z_{buy} = \beta_0^{buy} + \beta_1^{buy} x_1$

from data where the classes are now "buy/don't buy".

$z_{sell} = \beta_0^{sell} + \beta_1^{sell} x_1$

from data where the classes are now "sell/don't sell"

$z_{hold} = \beta_0^{hold} + \beta_1^{hold} x_1$

from data where the classes are now "hold/don't hold"

This works as long as each model has the same x_1 (and this generalizes to more than one feature: x_1, x_2,. . . x_n). Actually, multinomial logistic regression does this under the covers for you. . . but things like Support Vector Machines do not and you do have to do it manually.

You now have a metamodel for multiclass classification. When you need to make a prediction, you use all three models and pick the class that has the highest probability. Strangely, this is essentially what neural networks must do as well.

Note that there's a different although related problem of *multilabel* classification. In this case, each observation might be assigned more than one outcome For example, a story might be fiction *and* sports while another one might non-fiction and sports. A discussion of this problem is beyond the scope of these notes.

1.4 Conclusion

In this chapter we developed the basics of the two main linear models: linear regression and logistic regression. There is a whole family of linear models under the umbrella term *Generalized Linear Models* (GLM). For example, there is Poisson regression for estimating counts.

For both models, we motivated the development of the linear model as an alternative to the constant model using the mean of the target variable. As we noted this baseline model is often called the "null" model and we often compare our development efforts against the null model.

For linear regression, we examined models with numerical features and a binary categorical feature and learned to interpret them. We also introduced interaction terms. We applied Bayesian Bootstrap inference to our models and saw how they enable us to estimate credible intervals and probabilities for different parts of the model.

For logistic regression, we concentrated on the difference in interpretation that logistic regression requires. In general, the same techniques that can be used for linear regression (numerical features, binary categorical features, interaction terms) can be used for logistic regression.

In the next chapter, we'll discuss many more topics in linear and logistic regression such as model building, residuals, transformations, troubleshooting, and plotting.

1.4.1 Review

1. How does a linear regression model of a numerical target y extend the concept of the mean of y?
2. If x_1 is a *categorical* feature with two outcomes, how do we interpret the βs for the regression: $\hat{y} = \beta_0 + \beta_1 x_1$?
3. If x_1 is a *numerical* feature, how do we interpret the βs for the regression: $\hat{y} = \beta_0 + \beta_1 x_1$?
4. If x_1 is a *numerical* feature and x_2 is a *categorical* feature of two outcomes, how do we interpret the βs for the regression: $\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$?
5. What is an *interaction term* and why might we include them in a linear regression?
6. Describe the *predictive* and *causal* interpretations of the coefficients of a linear regression.
7. Why can't we use linear regression for a binary categorical y?
8. How is a logistic regression model of a binary categorical y extend the concept of the mean of y?
9. If y takes on more than two values, what are our alternatives?
10. How do we interpret β_0 in logistic regression?
11. How do we interpret any other β_i in logistic regression?

CHAPTER

TWO

LINEAR MODELS - PART 2

In the previous chapter, we learned the concepts behind linear and logistic regression. These concepts included:

1. continuity of linear models with using the mean as an estimate.
2. interpretation of the coefficients of linear models (linear regression and logistic regression) with numeric and binary categorical features as well as simple interaction terms.
3. application of the Bootstrap to linear models.

In this chapter, we're going to delve more deeply into linear models including:

1. how to make estimates and evaluate them.
2. evaluating *residuals* (short for "residual error").
3. extending the models to multiple features of both kinds and how to build models using domain knowledge and metaheuristics.
4. transforming variables to improve performance and or interpretation of the model.

We will defer a full discussion of model evaluation until the next chapter.

Most of this discussion is taken from Andrew Gelman and Jennifer Hill's, *Data Analysis Using Regression and Multi-Level/Hierarchical Models*.

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%matplotlib inline
```

[3]:
```
import numpy as np
import scipy.stats as stats
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import random
import patsy
import sklearn.linear_model as linear

import sys
sys.path.append('resources')
import models

sns.set(style="whitegrid")
```

2.1 Building Linear Models

So far we have only looked at the simplest linear models that use only one feature–two if the other feature is a binary categorical feature ("binary feature", hereafter):

$\hat{y} = \beta_0 + \beta_1 x_1$

The purpose of the simplification was to get you used to interpreting the coefficients of a linear model. There's nothing that prevents us from adding more features to the equation but as we add more features, the line becomes a *hyperplane* and a lot of intuition goes out the window. It's even difficult to chart these.

The full linear model is:

$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots + \beta_n x_n + N(0, \sigma)$

Here we make a (nuanced) distinction between "feature" and "variable" and the proceed to ignore it because the nomenclature is too entrenched. A "variable" is from the data that you collect via ETL and explore during EDA. A feature is a variable put into a model and may be exactly the same as a variable, transformation, or both. For example, it could be the case that:

$x_2 = mother\_iq$

and

$x_3 = \sqrt{mother\_iq}$

so the variable $mother\_iq$ appears directly and indirectly as *two* features in the model.

This means there may not be a one to one correspondence between variables in your database or other data source and the features in the model. Unfortunately, the term "variable" is so common in discussions that instead of fighting a losing battle, we will use "variable" and "feature" synonymously. Just remember, you often can and should transform your *variables* to create more or better *features*. We will discuss how later in this chapter.

Now that we know that a regression can handle many features, how do we know which ones to add?

2.1.1 Building Regression Models

The answer is probably not surprising, we should start with domain knowledge. The variables we have available are based on our discussion of the problem, meetings with domain experts, and qualitative modeling with Causal Loop Diagrams. We have wrestled to get the data (ETL) and wrestled with the data (EDA) and at this point we should have some intuitions about the variables, their relationships, and even possible transformations. This is a good starting point.

As a result, our first pass at a model will likely include all the variables we believe are relevant to the target variable, given our knowledge of the domain and the data that we were able to get (rather than the data we *wanted* to get). Whether they were explicit or implicit, we have our Causal Loop Diagrams to guide us. By using domain knowledge, we also satisfy one of the underlying assumptions of correct modeling, the assumption of *validity*. The assumption of validity assumes all the variables have a reason for being in the model.

Although there are techniques for the automatic selection of features, in general, selecting an optimal subset of features is NP-hard. It's easy to see why this is the case. If we have n variables and we want to select k features and build the best model we can, there are n choose k possibilities to pick from:

$\binom{n}{k} = \frac{n!}{k(n-k)!}$

for $n = 10$ and $k = 9$, that's 10 combinations. But what if the best set is 8? That's 45 more combinations. And if the best set is some 7 of them? 120 *more* combinations. That's a lot of combinations to try. And that doesn't include transformations or interaction terms. Increase the number of variables by 1 and you get even more possibilities.Domain knowledge at least gives us a foot in the door.

We also have the problem that as the sizes of data sets increases, spurious correlations are more likely to result. We don't want to create and field a model that stops working because a relationship we were depending on was ephemeral.

2.1.2 All Variables? Why One Leg is better than Two

Suppose we have data on human height and measurements for both the left and right leg. Here's a bit of Trivial Pursuit domain knowledge for you:

1. leg length is typically 40-50% of total height which would make leg length a fairly good predictor of height.
2. legs, however, are not always the same length. 70% of people typically have one leg longer than the other. The average difference is generally less than 1.1 cm. Differences of 0-0.30 cm is considered a "mild" difference, 0.31-0.60, is considered moderate and greater than 0.61 is considered severe.

Let's simulate this "process". We just need an average height in centimeters. The average Australian female is 161.8 cm tall. That's about 5' 3" tall. Let's assume that heights are normally distributed with a coefficient of variation of about 10%. That gives us a range of heights. We'll pick an average leg length of 45% with 2.5% points as the standard deviation which will give us 95% of the data between 40-50%. We'll say that's the longer leg.

The shorter leg is generally 1.1 cm shorter than the longer leg and even then 30% of the people don't have noticeably different leg lengths. This is a mixture model: a test to see if a leg length is shorter and then generating a difference. The difference is generally less than 1.1 cm. We can pick uniformly because it doesn't really matter:

```
[4]: np.random.seed(47383484)
```

```
[5]: data = {}
     data["height"] = stats.norm.rvs(161.8, 16.2, 100)
     data["long_leg"] = data['height'] * stats.norm.rvs(0.45, 0.025, 100)
     data = pd.DataFrame(data)
     data["short_leg"] = data["long_leg"].apply(lambda x: x if stats.uniform.rvs() < 0.3␣
     ↪else x - stats.uniform.rvs(0.0, 1.1))
```

Let's see what we've generated:

```
[6]: data.describe()
```

```
[6]:            height    long_leg   short_leg
     count  100.000000  100.000000  100.000000
     mean   160.131744   72.152456   71.759224
     std     15.344076    8.446286    8.441024
     min    128.082617   55.541141   55.337981
     25%    149.066740   65.251453   65.092133
     50%    158.561970   71.523243   71.125217
     75%    170.632393   77.945643   77.789251
     max    202.151122   98.053558   97.670528
```

So that looks good but we need to get rid of one artifact we created when generating the data. We don't generally collect data as "short" and "long" leg but "left" and "right" leg. Let's shuffle those:

```
[7]: def random_pick(x1, x2):
         return x1 if stats.uniform.rvs() < 0.5 else x2
```

```
[8]: data["total_leg"] = data["long_leg"] + data["short_leg"]
     data["left_leg"] = data[["long_leg", "short_leg"]].apply(lambda x: random_pick(*x),␣
     ↪axis=1)
     data["right_leg"] = data["total_leg"] - data["left_leg"]
```

Before we run the code to generate the linear regression model for `height ~ left_leg + right_leg`, what do you expect the coefficients (β_1 and β_2) on each leg to be? It seems reasonable to think that they would both be positive:

```
[9]: result = models.bootstrap_linear_regression("height ~ left_leg + right_leg", data)
```

```
[10]: print(models.describe_bootstrap_lr(result))
```

```
Model: height ~ left_leg + right_leg
--------  ------  ---  -----  ---
                                95% BCI
Coefficients              Mean    Lo        Hi
              $\beta_0$  45.50   36.08     57.50
left_leg      $\beta_1$  0.68    -2.69     3.14
right_leg     $\beta_2$  0.91    -1.59     4.22

Metrics       Mean        Lo      Hi
$\sigma$      7.47        6.24    8.48
$R^2$         0.77        0.65    0.84
--------  ------  ---  -----  ---
```

The R^2 looks pretty good. Our model captures about 77% of the variability of height. And the coefficients seem to be positive. However, if we look at the credible intervals, there's a strong probability that either or both of the coefficients are negative as well. Based on the data, we don't know what sign the coefficients should be. That seems strange.

This phenomenon goes by the name of *multicollinearity*. So we cannot simply include all of the variables we have even if domain knowledge suggests that they may all be relevant. One way to detect the possibility of such problems is to check for correlations between your features:

```
[11]: stats.pearsonr(data.left_leg, data.right_leg)[0]
```

```
[11]: 0.9979106583969305
```

You need only check Pearson's Correlation Coefficient because the correlation between the two variables must be linear. Note that checking all pairs of (numerical) variables lands us in the combinatorial explosion again but it's not quite as bad. We can use domain knowledge to prune the pairs we need to check. And we can also check coefficients to see if they have unexpected signs or low credibility for the expected sign.

What should we do?

A general approach is to:

1. include the variable that correlates most highly with the target variable.
2. construct a new variable that combines the information of the other, correlated variables by taking a sum, average, min, max, etc.

Let's try an average leg length:

```
[12]: data["average_leg"] = data["total_leg"] / 2
```

```
[13]: result = models.bootstrap_linear_regression("height ~ average_leg", data)
```

```
[14]: print(models.describe_bootstrap_lr(result))
```

```
Model: height ~ average_leg
--------  ------  ---  -----  ---
                                95% BCI
Coefficients              Mean    Lo        Hi
```

(continues on next page)

(continued from previous page)

```
               $\beta_0$  45.51  35.24    59.74
average_leg   $\beta_1$  1.59   1.39     1.73

Metrics       Mean       Lo     Hi
$\sigma$      7.43       6.34   8.39
$R^2$         0.77       0.70   0.85
--------  ------  ---  -----  ---
```

Now our model is interpretable. Notice that the R^2 and σ are unaffected. Multicollinearity is very often a concern only if you are building an explanatory model and not simply a predictive one. It's worth remembering the exchange between Blomberg and Gelman from the Introduction:

Simon Blomberg said,

> machine learning is statistics minus any checking of models and assumptions – Brian D. Ripley Two Cultures

To which Bayesian statistician Andrew Gelman responded,

> In that case, maybe we should get rid of checking of models and assumptions more often. Then maybe we'd be able to solve some of the problems that the machine learning people can solve but we can't!

There are a few things that matter greatly to the typical statistician that do not concern a machine learning engineer at all. For example, Andrew Ng teaches linear regression in his online Machine Learning course but he never once mentions multicollinearity. That doesn't mean we can ignore it, too. You have to know what the purpose of your model is. And even if it is only estimation, that doesn't mean somewhere down the road, a regulatory body isn't going to ask you to explain what it does. I present Gelman's advice throughout because I think it's a good middle ground between the two extremes. We concentrate on differences that *make* a difference when the difference is important.

We will now modify our steps for building linear models:

1. Start with all features as suggested by domain knowledge, but...
2. Analyze features for correlations and, of those groups, consider picking the best variable, an average of the variables, or some other transformation (sum, min, max, etc).

Think back to the EDA chapter and the King's County Housing Example. Do you think that sqft_living, sqft_above, sqft_below, sqft_living15 are all correlated? What about sqft_lot and sqft_lot15?

So now we have an extra goal for EDA. If we know that we are going to be building a linear model, we should investigate correlations between variables that we plan to include in our linear model (and not just correlations between the target variables–price in this case–and the features). We add this now because if we're using a different kind of model, the problem of multicollinearity might not even arise.

2.1.3 Multiple Binary Features

As we have just seen, although linear models are perfectly general, but there can be problems. If there is multicollinearity between two or more features, we will have some difficulty getting good estimates of their coefficients. The values of the coefficients are important if we wish to explain the relationship between the target variable and the features.

Although it's more difficult to introduce multicollinearity between binary features, it's not impossible. Consider the canonical binary feature (that's not entirely binary): gender. Suppose we have $gender$ in the linear model below:

$\hat{y} = \beta_0 + \beta_1 gender + \beta_2 income$

Our first observation is that when it comes to binary variables, we should name them whatever category is "1". So if it's 0 if female and 1 if male, we should just name the variable "male". This helps immeasurably with interpretation:

$\hat{y} = \beta_0 + \beta_1 male + \beta_2 income$

With this model, β_1 becomes the differential effect being male. Where's the female effect? It's in β_0.

But why can't we we have a variable, $female$ too and have β_0 be zero?

$\hat{y} = \beta_0 + \beta_1 male + \beta_2 income + \beta_3 female$

Well, if we think about it a second, this would mean that the data looks like this:

id	male	female	income
1	1	0	34000
2	0	1	38000
3	1	0	42000
4	1	0	40000
5	0	1	29000

$male$ and $female$ are just the opposite of each other so there's no new information to be gained from including $female$ as a variable. And if male is 1 and female is 0, what is β_0? This doesn't make sense. All of this seems obvious because we're used to "natural" binary variables like purchased, voted, etc. `gender` isn't necessarily binary, although we can often treat it as such. You would do well be aware if this makes a different for your business.

What happens if we have a new variable, politics, that has three values: {left, center, right}. Our first thought might be to code these as {1, 2, 3} but that doesn't make sense for linear models: left + center = right? Um, no. Instead we can do a "one hot encoding": left {0, 1}, center: {0, 1}, right: {0, 1}. Now, here's the question. If we want to include these variables in the linear model,

$\hat{y} = \beta_0 + \beta_1 male + \beta_2 income + \beta_3 left + \beta_4 center + \beta_5 right$

does it look like the above? No and for the same reason that we didn't include both male and female in the model. Here's the rule:

> Every non-binary variable must be converted to a one hot encoding using the label as the feature name if possible. This will create m new binary variables if the original variable has m labels or outcomes. However, you can only include $m - 1$ of these new features in your linear model. The missing variable gets pushed into the interpretation of β_0, the intercept.

Let's see how this works. We're going to take out "left":

$\hat{y} = \beta_0 + \beta_1 male + \beta_2 income + \beta_3 center + \beta_4 right$

We can see that the male effect is β_1, the center effect is β_3, and the right effect is β_4. Where is the female effect? β_0. Where is the left effect? β_0. In fact, β_0 is now "left females".

Where is the effect for right females? $\beta_0 + \beta_4$. Left males? $\beta_0 + \beta_1$.

This makes a strong assumption that politics and gender are independent (no pun intended) and this may not be the case. We may want to create interaction terms. However,

> the more specific the terms in your model, the more data you need to estimate it

If you don't have a lot of Pagan, right, low income males in your data set. . . you're not going to get good estimates of the coefficients. Binary variables are always partitioning your data set into subpopulations. As we have seen, their presence in a model doesn't have the same effect on the model as a numeric feature.

It follows that of the m one hot encodings, which one do we push into β_0? It is often said that it doesn't matter but I disagree. You need to take the relative frequency of the subpopulations into account.

Consider what happens if you drop the binary variable with the fewest observations. With fewer observations, β_0 will have a higher variance than the incremental effect β_1 (for example). This might not be good if we need a solid estimate of the base effects.

However, if we're interested *only* in the differential effect (how much does it change in the presence of the category), dropping the encoding out of m with the fewest observations might be satisfactory. Conversely, if we *do* want good base rate estimates, then we want to drop the encoding out of m with the most observations. Another option is to combine categories in meaningful ways to increase the number of observations they cover.

No matter what, at least for primary effects, one of the one hot encodings for each such transformed categorical variable needs to be left out. The value will be pushed into the interpretation of β_0.

We will now modify our steps for building linear models:

1. Start with all features as suggested by domain knowledge, but…
2. Analyze features for correlations and, of those groups, consider picking the best variable, an average of the variables, or some other transformation (sum, min, max, etc).
3. Transform all categorical variables into one hot encodings but leave one encoding out of the model for each variable. The intercept β_0 represent all the outcomes that are excluded explicitly. Which one you leave out might depend on the number of observations for each and what you want to do with the model.

Most data libraries have functions to create "dummy variables" ("dummy" in the sense of "standing in for something real" as in a "crash test dummy"). Pandas is no exception:

```
[15]: politics = {"politics": np.random.choice(["left", "center", "right"], size=10)}
      data = pd.DataFrame(politics)
      data.head()
```

```
[15]:   politics
      0   center
      1   center
      2   center
      3   center
      4     left
```

```
[16]: data = pd.concat([data, pd.get_dummies(data["politics"])], axis=1)
      data.head()
```

```
[16]:   politics  center  left  right
      0   center       1     0      0
      1   center       1     0      0
      2   center       1     0      0
      3   center       1     0      0
      4     left       0     1      0
```

Now what? We estimate our model. For which we'll go back to Child IQs.

2.1.4 Child IQs

We've already seen this data but let's quickly review all of it.

```
[17]: child_iq = pd.read_csv("resources/child_iq.tsv", sep="\t")
```

```
[18]: child_iq.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 434 entries, 0 to 433
Data columns (total 5 columns):
child_iq    434 non-null int64
mom_hs      434 non-null int64
mom_iq      434 non-null float64
```

(continues on next page)

(continued from previous page)

```
mom_work     434 non-null int64
mom_age      434 non-null int64
dtypes: float64(1), int64(4)
memory usage: 17.0 KB
```

Our target variable is child_iq. Our possible features are mom_hs, mom_iq, mom_work (did the mom work during the preschool years), and the mother's age, mom_age. We can guess that mom_iq and mom_age are numerical but what about the others? Let's see some data:

[19]:
```
child_iq.head()
```

[19]:
```
   child_iq  mom_hs      mom_iq  mom_work  mom_age
0        65       1  121.117529         4       27
1        98       1   89.361882         4       25
2        85       1  115.443165         4       27
3        83       1   99.449639         3       25
4       115       1   92.745710         4       27
```

Both mom_hs and mom_work seem to encodings of some kind. Let's see how many values they have:

[20]:
```
child_iq.mom_hs.value_counts()
```

[20]:
```
1    341
0     93
Name: mom_hs, dtype: int64
```

As we already knew, mom_hs is a binary variable. What about mom_work?

[21]:
```
child_iq.mom_work.value_counts()
```

[21]:
```
4    205
2     96
1     77
3     56
Name: mom_work, dtype: int64
```

mom_work has four possible values. What are these values? According to the data dictionary:

1. mother did not work for the first 3 years of the child's life.
2. mother worked in second or third year of child's life.
3. mother worked part-time during first year of child's life.
4. mother worked full-time during first year of child's life.

As we can see, the most common outcome is "4".

We'll need to change this variable into a set of dummy variables. Because the labels are numbers, we have two options:

1. change the values into labels in a new variable and then create the dummy variables.
2. specify a prefix for the dummy values.

For this variable, for the first option, we might have "no_work", "some_work", "part_time", and "full_time". For the second option, we can just use "mom_worked" as the prefix and it'll create dummies for "mom_worked_1", "mom_worked_2", "mom_worked_3" and "mom_worked_4". We're going to take the easy way out:

[22]:
```
child_iq = pd.concat([child_iq, pd.get_dummies(child_iq["mom_work"], prefix="mom_
↪worked")], axis=1)
```

```
[23]: child_iq.head()
```

```
[23]:    child_iq  mom_hs      mom_iq  mom_work  mom_age  mom_worked_1  \
      0        65       1  121.117529         4       27             0
      1        98       1   89.361882         4       25             0
      2        85       1  115.443165         4       27             0
      3        83       1   99.449639         3       25             0
      4       115       1   92.745710         4       27             0

         mom_worked_2  mom_worked_3  mom_worked_4
      0             0             0             1
      1             0             0             1
      2             0             0             1
      3             0             1             0
      4             0             0             1
```

We now have mom_iq, mom_age, mom_hs, mom_worked_1 (no work), mom_work_2 (some work), mom_worked_3 (part time), and mom_worked_4 (full time). Before continuing, what do you think the *sign* of each coefficient (β_i) is going to be?

- `mom_hs`: positive. This could be a proxy for a number of things including income, demographics, intelligence outside general intelligence testing.
- `mom_iq`: positive. It's likely there's a positive relationship between the mother's IQ and the child's.
- `mom_age`: positive. Older mothers might be better educated or more experienced. This could be a proxy for other things including income, demographics, etc.
- `mom_worked_1`: excluded - this the most common outcome for the mom_worked variable so we'll excluded it from explicit inclusion in the model.
- `mom_worked_2`: negative - relative to the baseline, this should show a small decrease in IQ although overall this may be a signifier for demographics and income (if you can afford to take off the first year, you may have good maternity leave or support from the husband's income).
- `mom_worked_3`: negative but more so than mom_worked_2 because it's part-time work throughout the 3 years.
- `mom_worked_4`: negative but more so than mom_worked_3 because it's full-time work throughout the 3 years.

It's interesting to think about the overall implications of the validity assumption here. Depending on the source of the data, this model might not be applicable to minorities, for example. The validity assumption is broader than *just* including relevant variables but also making relevant estimates that are not "out of sample".

Now let's estimate the model: "child_iq ~ mom_hs + mom_iq + mom_age + mom_worked_2 + mom_worked_3 + mom_worked_4"

```
[24]: model = "child_iq ~ mom_hs + mom_iq + mom_age + mom_worked_2 + mom_worked_3 + mom_
      ↪worked_4"
      result = models.bootstrap_linear_regression(model, data=child_iq)
```

```
[25]: print(models.describe_bootstrap_lr(result))
```

```
Model: child_iq ~ mom_hs + mom_iq + mom_age + mom_worked_2 + mom_worked_3 +␣
↪mom_worked_4
--------  ------  ---  -----  ---
                              95% BCI
Coefficients              Mean    Lo        Hi
```

(continues on next page)

(continued from previous page)

```
              $\beta_0$  20.27  -0.82    39.32
mom_hs        $\beta_1$  5.43   0.83     9.64
mom_iq        $\beta_2$  0.55   0.43     0.67
mom_age       $\beta_3$  0.22   -0.57    0.85
mom_worked_2  $\beta_4$  2.98   -2.23    9.33
mom_worked_3  $\beta_5$  5.49   0.14     10.18
mom_worked_4  $\beta_6$  1.42   -2.91    5.66

Metrics       Mean       Lo     Hi
$\sigma$      18.14      17.06  19.24
$R^2$         0.22       0.15   0.30
--------   ------   ---   -----  ---
```

2.1.5 Interpreting the Results

First, we observe that the R^2 is 0.22 or 22%. The model explains 22% of the variation in child_iq. However, we note that the Coefficient of Determination is affected by the number of features in your model. That is, other things being equal, as you go from one feature to two to three to four, etc., the R^2 will not go down. It may stay the same or increase but it won't go down. That is, R^2 is a non-decreasing monotonic function of the number of variables. It is thus prudent to evaluate the *Adjusted* R^2 or $\bar{R}^2$ which is:

$\bar{R}^2 = 1 - (1 - R^2)\frac{n-1}{n-p-1}$

where n is the number of observations and p is the number of features in the model. We can estimate ours here:

```
[26]: def adjusted_r_squared(result):
          adjustment = (result["n"] - 1)/(result["n"] - len(result["coefficients"]) - 1 - 1)
          return 1 - (1 - result["r_squared"]) * adjustment
```

```
[27]: adjusted_r_squared(result)
```

```
[27]: 0.20661898066713968
```

which isn't hugely different. We have 434 observations and 6 features. Once you get into big data territory, the difference will diminish although "big data" is relative. If you have 10,000 observations but 1,000 features, your data isn't really bigger than having 1,000 observations but 10 features.

Second, we note that the error of the regression (σ) is 18.14 IQ points. What does that error really mean? Answering that question requires us to go on a bit of a tangent.

Estimating with Linear Regression

The mean value of mom_iq and mom_age are, respectively:

```
[28]: child_iq.mom_iq.mean()
```

```
[28]: 99.99999999999999
```

```
[29]: child_iq.mom_age.mean()
```

```
[29]: 22.785714285714285
```

Let's say that's an IQ of 100 (which makes sense!) and an age of 23. If we consider the group of mothers with IQs of 100, who graduated high school, who was 23 years of age, and did not work during the first three years of the child's life we have a feature *vector* of [1, 100.0, 23, 0, 0, 0]. However, given the way the `models` module is set up, we need to include $x_0 = 1$ for the intercept *explicitly* so we have: [**1**, 1, 100.0, 23, 0, 0, 0].

Using that feature vector in the `predict` function of the `LinearRegression` model, we have the following $\hat{child\_iq}$:

```
[30]: predicted = result["model"].predict(np.array([[1, 1, 100.0, 23.0, 0, 0, 0]]))[0][0]
      predicted
```

```
[30]: 85.96983127074486
```

Note: the double array in the function call to `predict` is because predict is set up to take more than one feature vector and return an *array* of predictions.

The predicted *average* IQ of children whose mothers have those characteristics is about 86. And this is where σ or the error of the regression comes in. Remember that σ is a standard deviation. If these errors are indeed normally distributed, we expect the true value of the child's IQ to be within 1 standard deviation of the prediction 68% of the time; 2 standard deviations, 95% of the time, and 3 standard deviations, 99.7% of the time (the exact figures are 68.27%, 95.45%, and 99.73%. . . I don't know why we only show the decimal on the last one).

We can do a quick estimate of the 95% bounds on our estimate is:

```
[31]: print("({0:.2f}, {1:.2f})".format(predicted - 2 * result["sigma"], predicted + 2 *␣
      ↪result["sigma"]))
```

```
(49.69, 122.24)
```

Is that better than the "null" model or just using the mean child IQ. The mean and standard deviation of child IQ's are:

```
[32]: mean_child_iq = child_iq.child_iq.mean()
      std_child_iq = child_iq.child_iq.std()
      print("child_iq = {0:.2f} ({1:.2f})".format(mean_child_iq, std_child_iq))
```

```
child_iq = 86.80 (20.41)
```

So for any child, regardless of the characteristics of the mothers, we would predict an IQ of 86.8 and the 95% bounds would be:

```
[33]: print("({0:.2f}, {1:.2f})".format(mean_child_iq - 2 * std_child_iq, mean_child_iq + 2␣
      ↪* std_child_iq))
```

```
(45.98, 127.62)
```

So we've gotten a *little* better?

I know what you're saying. . . wait. . . confidence intervals? Well, let's punt on that for now. Given what we now know, it doesn't hurt to do some quick calculations assuming some mathematical distribution (Normal) and giving them a Bayesian interpretation. Let's see what happens if we do it right. . .

In the result map, we have stored all the Bootstrap estimates for the coefficients and therefore enough information to do bootstrap estimates of the posterior distribution of a prediction. We can write a function for that:

```
[34]: def bootstrap_prediction(result, feature_values):
          # resampled_coefficients is a DataFrame in result.
          # each *row* is a different bootstrap model.
          # we use a Dict instead of Vector because the order may not be preserved.
          # we should modify this to use the model to pick the values and possibly
          # convert interactions terms.
          results = []
          for coefficients in result["resampled_coefficients"].itertuples():
```

(continues on next page)

(continued from previous page)

```
        estimate = 0
        for feature in feature_values.keys():
            estimate += feature_values[feature] * getattr(coefficients, feature)
        results.append(estimate)
    return np.array(results)
```

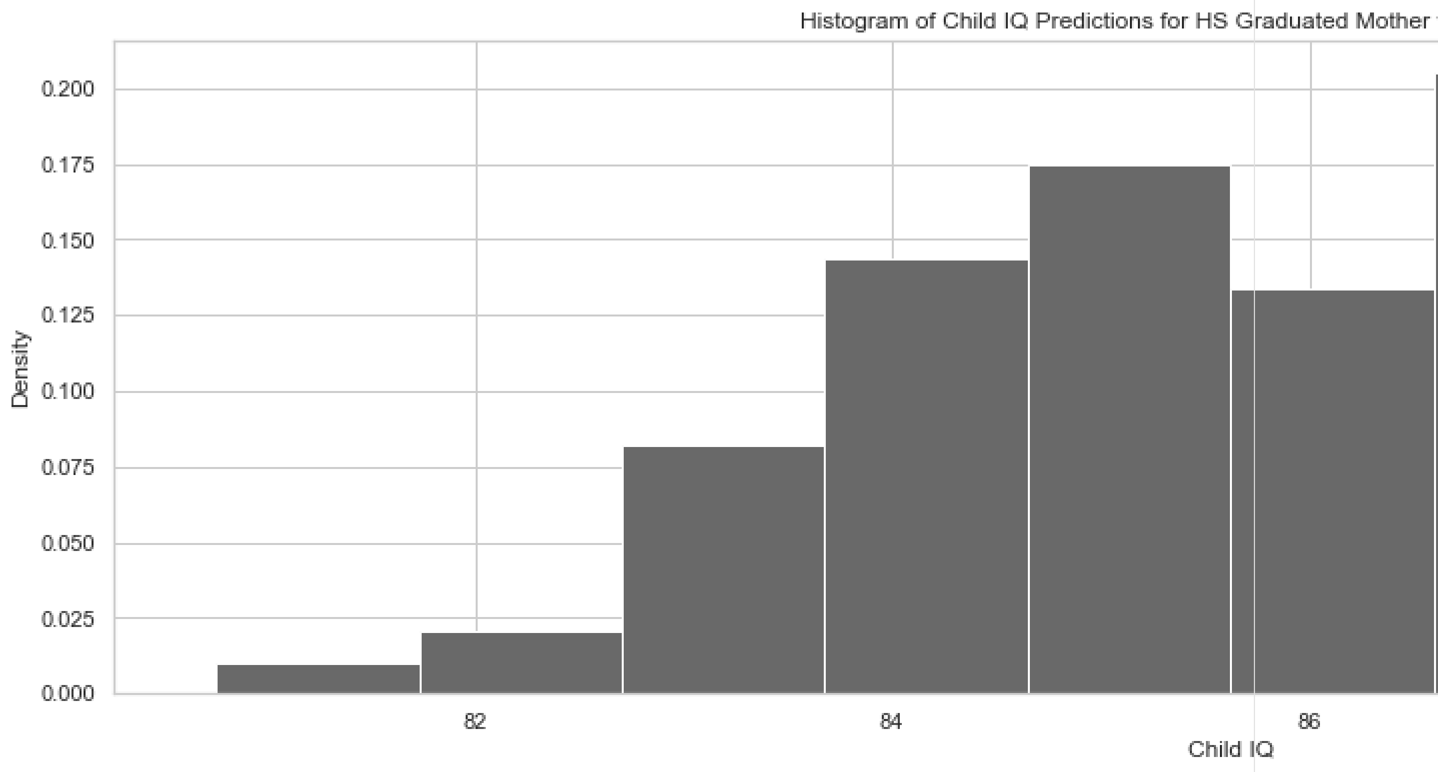
The mean of all of our predictions is:

```
[35]: feature_values = {"intercept": 1, "mom_hs": 1, "mom_iq": 100, "mom_age": 23}
      posterior_prediction = bootstrap_prediction(result, feature_values)
      posterior_prediction.mean()
```

```
[35]: 86.21523981347433
```

The distribution looks like:

```
[36]: figure = plt.figure(figsize=(20, 6))

      axes = figure.add_subplot(1, 1, 1)
      axes.hist(posterior_prediction, color="DimGray", density=True)
      axes.set_xlabel( "Child IQ")
      axes.set_ylabel( "Density")
      axes.set_title("Histogram of Child IQ Predictions for HS Graduated Mother with IQ 100,
      ↪ Aged 23")

      plt.show()
      plt.close()
```



Oh, snap. The distribution is grossly normal–that may be an artifact of the number of bootstrap samples we took; we should consider increasing them for the final model. For now, our 95% credible interval (or Bayesian confidence interval) is:

```
[37]: print("95% BCI", stats.mstats.mquantiles( posterior_prediction, [0.025, 0.975]))
```

```
95% BCI [82.59661173 90.26274411]
```

Note that this bound is a lot tighter. Why? Because our feature values are at their means and σ is for the *entire* range of each feature. The model will be its most accurate at the mean values of the features.

Coefficients

Now that we can interpret the R^2 and σ of the regression, continue with the coefficients which we repeat below:

```
[38]: print(models.describe_bootstrap_lr(result))
```

```
Model: child_iq ~ mom_hs + mom_iq + mom_age + mom_worked_2 + mom_worked_3 +␣
↪mom_worked_4
--------  ------  ---  -----  ---
                                 95% BCI
Coefficients                Mean    Lo        Hi
              $\beta_0$  20.27   -0.82     39.32
mom_hs        $\beta_1$  5.43    0.83      9.64
mom_iq        $\beta_2$  0.55    0.43      0.67
mom_age       $\beta_3$  0.22    -0.57     0.85
mom_worked_2  $\beta_4$  2.98    -2.23     9.33
mom_worked_3  $\beta_5$  5.49    0.14      10.18
mom_worked_4  $\beta_6$  1.42    -2.91     5.66

Metrics       Mean       Lo      Hi
$\sigma$      18.14      17.06   19.24
$R^2$         0.22       0.15    0.30
--------  ------  ---  -----  ---
```

Remember our predictions before we ran the model? Let's write a quick function that compares our predictions based on domain knowledge and the actual values we estimate, in terms of probabilities. Note that this is better than asking if something is "statistically significant".

```
[39]: # {"var1": "+", "var2": "-"}
def evaluate_coefficient_predictions(predictions, result):
    coefficients = result["resampled_coefficients"].columns
    for coefficient in coefficients:
        if coefficient == 'intercept':
            continue
        if predictions[coefficient] == '+':
            print("{0} P(>0)={1:.3f}".format(coefficient, np.mean(result["resampled_
↪coefficients"][coefficient] > 0)))
        else:
            print("{0} P(<0)={1:.3f}".format(coefficient, np.mean(result["resampled_
↪coefficients"][coefficient] < 0)))
```

```
[40]: predictions = {
    "mom_hs": '+',
    "mom_iq": '+',
    "mom_age": '+',
    "mom_worked_2": '-',
    "mom_worked_3": '-',
    "mom_worked_4": '-'}
evaluate_coefficient_predictions(predictions, result)
```

```
mom_hs P(>0)=0.990
mom_iq P(>0)=1.000
mom_age P(>0)=0.780
mom_worked_2 P(<0)=0.150
mom_worked_3 P(<0)=0.010
mom_worked_4 P(<0)=0.300
```

From this we see that mom_hs has a very high probability of being positive as does mom_iq. mom_age might be positive; there's a 78% chance, based on the data, that it is. However, for the other three variables that we thought would be negative, there's a very low probability that they are: 15%, 1%, and 30% respectively. What are we to do?

2.1.6 Guidelines for Evaluating Coefficients

The general guidelines for evaluating coefficients when working with posterior distributions and credible intervals instead of confidence intervals changes a little bit. When we have a confidence interval of (-0.23, 5.97) we note that the coefficient *might* be zero and we just don't know. However, if we have a credible interval with the same bounds and the posterior distribution, we might say there's an 83% probability that the coefficient is positive (or a 17% probability that it's negative).

Case	Sign	Credible Interval	Action
1	expected	does not include 0	Keep
2	expected	does include 0	Keep
3	unexpected	does not include 0	Re-examine
4	unexpected	does include 0	Remove

Basically, for this interpretation, we're treating a 95% *credible* interval that contains 0 as some evidence that the coefficient might be (however slim) the oppose sign. Alternatively, we might interpret ranges of the posterior distribution as [0, 33%) "weak" evidence, [33, 66%), "mixed" evidence, and [66%, 100%] "strong" evidence for a particular sign. *Don't get hung up on it*. Use your judgment and experience. If you aren't sure, go with the interval interpretation until you get experience.

So, why are these rules of thumb the way they are and why they might be reasonable?

In all cases, we can think of "the expected sign" as an informal prior probability of at least the *direction* of influence by the variable, positive or negative. In actual Bayesian modeling it is possible to include this information as an actual prior probability on β_i but we're doing this informally here.

Case 1

So in the first case, say we have an informal prior of "positive", an actual positive coefficient and the confidence interval that does not include zero. This implies that the confidence interval includes only positive values. Since our informal prior was only about the *sign*, we should keep the variable.

Case 2

In the second case, say we have an informal prior of "positive", an actual positive coefficient but the confidence interval includes 0. Remember that the confidence interval is really a section of the posterior probability of the β_i in question. If the coefficient is positive and the confidence interval is symmetric, then there must be a greater than 50% probability that the coefficient is positive. This is why we keep it. But since we have the posterior probability, we might be able to find out if the probability really *is* 50/50 and adjust accordingly.

Case 3

In the third case, say we have an informal prior of "positive", an actual negative coefficient and a confidence interval that does not include zero. This means that we are pretty confident that, given the data, the probability of the value being negative is 95%. This means that we need to re-evaluate our beliefs about the variables.

For example, if the variable is incumbency. In the US, this is often positively correlated with re-election but in India, incumbency is negatively correlated. The unexpected sign might also mean that some unknown variable is missing. Yes, this can happen, too.

Case 4

In the fourth case, say we have an informal prior of "positive", an actual negative coefficient and a confidence interval that includes zero. Because it includes zero and the coefficient is negative, there is a greater than 50% chance the coefficient is actually negative but the evidence is not overwhelming. We could also consider this under the third case if the confidence interval is really skewed towards negative when we expected positive. That is, we can bring in our ideas of "weak", "mixed" and "strong" evidence.

Of course, the reasoning works if we make all the necessary substitutions starting with "say we have an informal prior of 'negative'". We can also think of this in terms of ROPE (region of practical significance), we think of the base line as being zero or not and things "near zero" might be practically zero.

This method of evaluation means thinking ahead of time (remember *validity*?) about the expected sign of the coefficients. This emerges implicitly from our domain knowledge or explicitly through Causal Loop Diagrams.

2.1.7 Interpreting Coefficients (Again)

There's one last thing we haven't addressed about coefficients.

We already talked about the predictive and the causal interpretations of coefficients. The predictive interpretation is not quite as easy to describe. Basically, linear models are predicting means of some kind. If we look at a set of specific values for a set of features then we're trying to predict the mean of the target variable for that set of feature values. Using child_iq again, if we think of mothers with IQs of 100, then we're predicting the mean IQ of their children. And the $beta_i$ of mom_iq, predicts the change in the mean when we look at mothers with IQs of 101.

The causal interpretation is closer to the "slope" idea of a coefficient. If mother's IQ increases by 1 point, then we expect the child's IQ to change by β_i, on average. This is also called the *counterfactual* interpretation.

But there's a question that is often asked about linear models that we haven't yet addressed. When you have a model of 10 features, which coefficient, β_i, is more important? or strongest? And it turns out that this is not an easy question to answer.

The main reason is that the β_i coefficients have *two* jobs to do. First, they measure the *effect*. But, second, they *scale* one metric into another. While mom_iq points and child_iq points are in the same units, mom_age and child_iq are not. They are very nearly not even of the same magnitudes and consider a coefficient that must convert from bedrooms to prices in hundreds of thousands of dollars.

Because of this, a large coefficient may be both converting a small number into a larger number's domain and representing a small effect while a smaller coefficient may be converting numbers into the same ranges but represents a (relatively) large effect. It is because of this that it's generally difficult to talk about which coefficients are more important.

We will talk about ways of working around this when we talk about transformations.

But suppose our model isn't very good. Can we make it better? Where should we look? We'll cover that in the remaining sections.

```
[1]: import warnings
     warnings.filterwarnings('ignore')
```

```
[2]: %matplotlib inline
```

```
[3]: import numpy as np
import scipy.stats as stats
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import random
import patsy
import sklearn.linear_model as linear

sns.set(style="whitegrid")

import sys
sys.path.append('resources')
import models
```

2.2 Residuals

When building regression models of all kinds, we are often concerned with improving the model. There are many ways to do this. Many of them involve domain knowledge, did we leave any important variables out? These kinds of questions can be answered by looking at the R^2 or σ (error of the regression). If the R^2 is low or the σ is high, it may not be because of missing variables but because of not enough data. This is one of the advantages of many of the *machine learning* methods for evaluation models because we can thinking of our models in terms under or overfitting and see where we might be under or overfitting. We'll look at *general* model evaluation in the next chapter.

A more typical approach for linear regression, however, is to look at the *residuals*. Remember that our regression model is a linear function of the features:

$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + N(0, \sigma)$

We learn this model from a set of observations, (X, y). This means, for every observation, we can use the model to estimate what we think y would be ($\hat{y}$) and we have the actual y to compare to. The difference is called the *residual* short for "residual error of the model$:

$r_i = y_i - \hat{y}_i$

where i is some observation. If r_i is positive, we underestimated y and if r_i is negative, we overestimated y. When we look at residuals, we plot them against a numerical feature that has been sorted. We should *not* see any pattern other than, essentially, ϵ. Remember that ϵ is normally distributed with mean 0 and some σ.

Because the feature x_j is sorted, we will readily be able to see many patterns if they exist in the data. Put a different way, we should *not* see any patterns if our model is sufficient.

What patterns might we see?

1. The distribution of residuals may not be distributed normally, $N(0, \sigma)$.
2. The residuals may increase as x_j increases.
3. The residuals may decrease as x_j decreases.
4. There may be some non-linear change in residuals as x_j changes (high residuals at low and high values of x_j but low residuals at x_j).

While you will often hear about the first one, this is often the least interesting of the bunch. While a large divergence from normality may affect our ability to generate accurate predictions, it does not impair our ability to study effects.

Let's look at an example.

2.2.1 Noiseless Data

We're going to start with the dead simplest generated data and regression: one with $\epsilon = 0$:

$y = 1.00 + 2.5x_1 + N(0, 0)$

Note that the *generating* function generates y and not $\hat{y}$. The generating function is the real deal. We're going to estimate the following model:

$\hat{y} = \beta_0 + \beta_1 x_1 + \epsilon$

Without loss of generality, I'm going to make x_1 have a domain of about $(0, 10)$. This makes the β_0 term "meaningful" because it can actually occur in the domain of x_1 and we can side step all of *those* problems.

```
[4]: np.random.seed(6734745)
```

```
[5]: x1 = np.array( [10*np.random.random() for _ in range( 100)])
     ys = 1.00 + 2.5 * x1
     data = pd.DataFrame({"y": ys, "x1": x1})
```

```
[6]: data.describe()
```

```
[6]:                  y          x1
     count  100.000000  100.000000
     mean    14.361599    5.344639
     std      7.921989    3.168796
     min      1.238822    0.095529
     25%      7.408161    2.563264
     50%     14.683931    5.473572
     75%     22.504152    8.601661
     max     25.866052    9.946421
```

Let's run the regression model. We're going to use the simple one so we can concentrate on the residuals.

```
[7]: result = models.linear_regression("y ~ x1", data)
     print(models.simple_describe_lr(result))
```

```
Model: y ~ x1
--------  ------  ---
Coefficients              Value
              $\beta_0$   1.00
x1            $\beta_1$   2.50

Metrics       Value
$\sigma$      0.00
$R^2$         1.00
--------  ------  ---
```

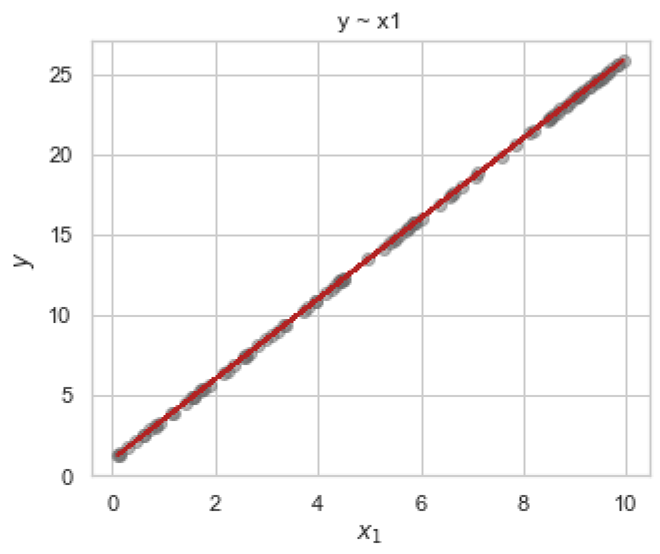
This is unsurprising. There's no noise in the model. . . it's all signal. The estimates of the coefficients are nearly exact, there's infinitesimal error, and the R^2 is 100%. If we chart the data and the line, we see:

```
[8]: figure = plt.figure(figsize=(5,4))

     axes = figure.add_subplot(1, 1, 1)

     axes.scatter(data.x1, data.y, color="dimgray", alpha=0.5)
     beta = result["coefficients"]
     axes.plot(data.x1, [beta[ 0] + beta[ 1] * x for x in data.x1], '-', color="firebrick")
     axes.set_title(result["formula"])
```

(continues on next page)

(continued from previous page)

```
axes.set_xlabel(r"$x_1$")
axes.set_ylabel(r"$y$")

plt.show()
plt.close()
```



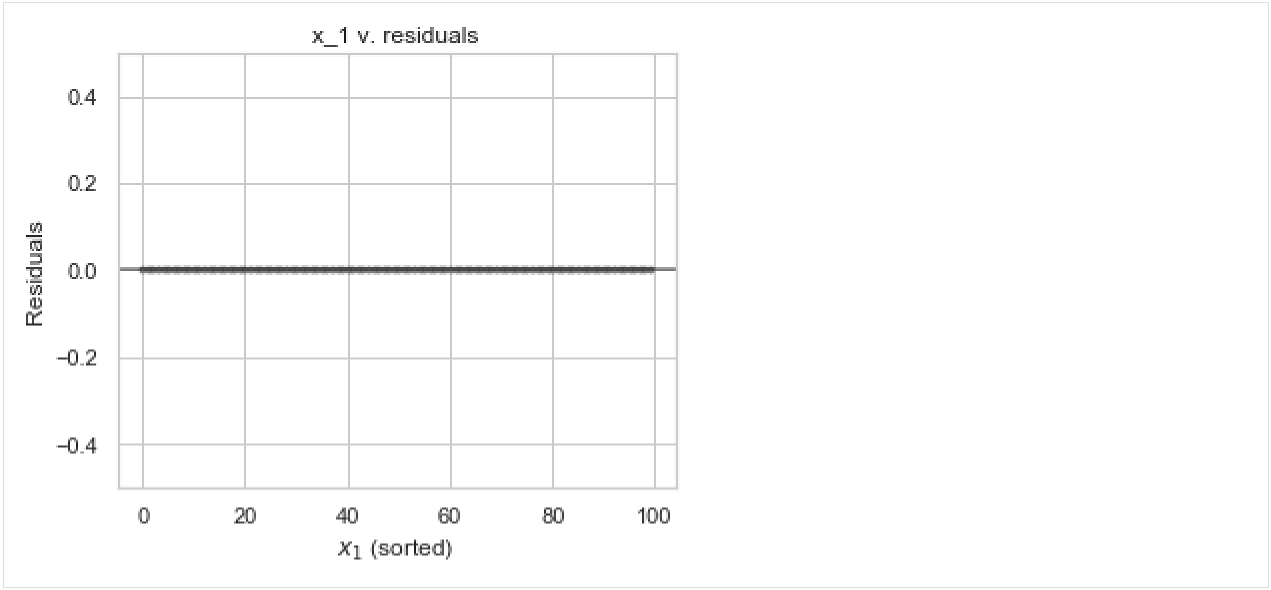
We can also plot the residuals. It's often handy to calculate their mean and standard deviation as well. I'm going to limit each to 4 decimal places to avoid showing the underflow.

[9]:
```
print("mean = ", "%.4f" % np.mean(result["residuals"]))
print("std  = ", "%.4f" % np.std(result["residuals"]))
```

```
mean =  -0.0000
std  =  0.0000
```

And here's the residual plot. Note that I had to do something similar with the y axis because of the underflow. Otherwise, you see a pattern that is entirely do to floating point error.

[10]:
```
figure = plt.figure(figsize=(5,4))

axes = figure.add_subplot(1, 1, 1)

# sorted
keyed_values = sorted(zip( data.x1, result["residuals"]), key=lambda x: x[ 0])

residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( "x_1 v. residuals")
axes.set_ylim((-0.5, 0.5))
axes.set_ylabel("Residuals")
axes.set_xlabel(r"$x_1$ (sorted)")

plt.show()
plt.close()
```



There's nothing to see here but that's good. We're just setting up the remaining examples. At first we're going to concentrate on looking at patterns in data that have no "noise". We'll introduce noise at the end so that the data looks more realistic.

2.2.2 A Squared Term

So far, so good. We have a perfect model and perfect residuals. What I want to do now is show you what the residuals would look like–in the absence of noise–if the true function were:

$y = 1.00 + 2.5x_1^2 + N(0, 0)$

but we only estimated:

$\hat{y} = \beta_0 + \beta_1 x_1 + \epsilon$

First, we need the data. We'll call this new "y", "y1" and add "x1_sqrd".

[11]:
```
data[ "x1_sqrd"] = data.x1 ** 2
data["y1"] = 1.00 + 2.5 * data.x1_sqrd

data.describe()
```

[11]:
```
                y          x1     x1_sqrd          y1
count  100.000000  100.000000  100.000000  100.000000
mean    14.361599    5.344639   38.506023   97.265058
std      7.921989    3.168796   34.046488   85.116221
min      1.238822    0.095529    0.009126    1.022814
25%      7.408161    2.563264    6.570634   17.426584
50%     14.683931    5.473572   29.960363   75.900909
75%     22.504152    8.601661   73.989326  185.973315
max     25.866052    9.946421   98.931284  248.328209
```

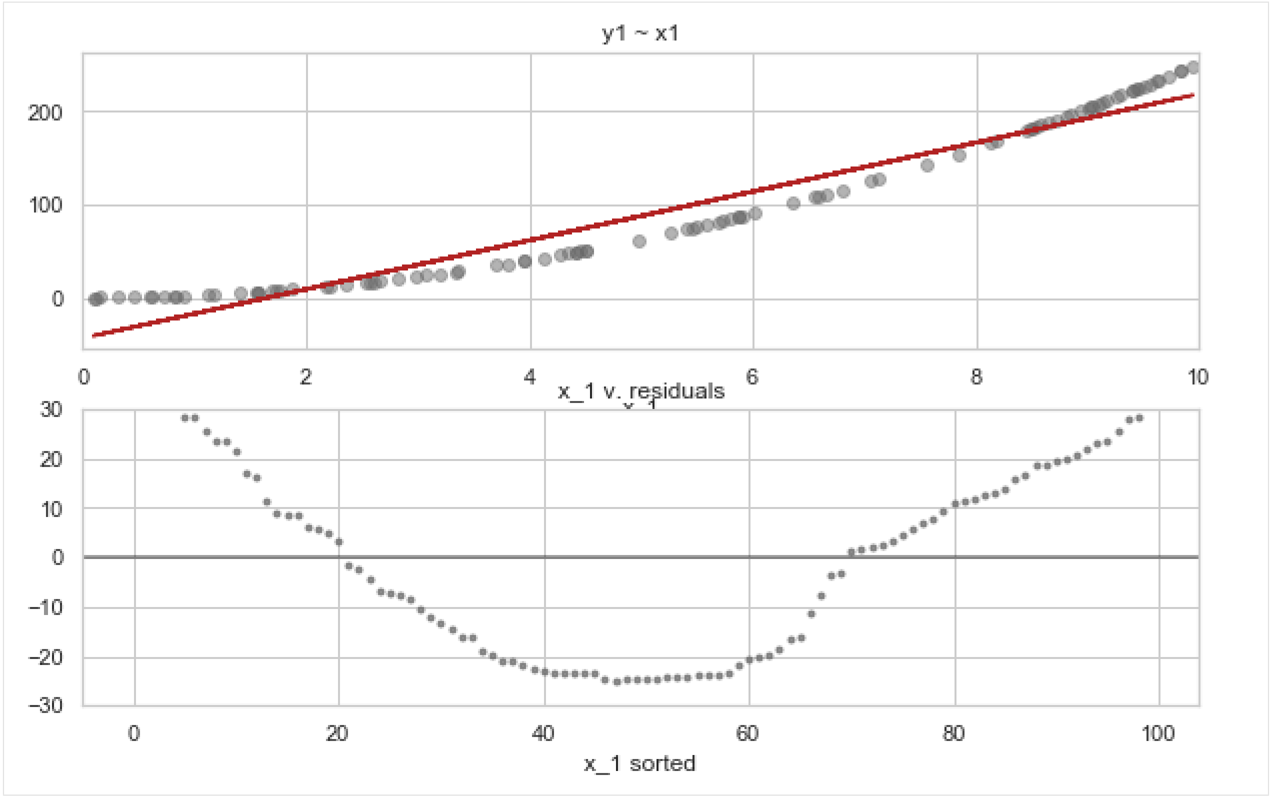
Let's fit the model:

[12]:
```
result = models.linear_regression("y1 ~ x1", data)
print(models.simple_describe_lr(result))
```

```
Model: y1 ~ x1
---------  ------  ----
Coefficients             Value
               $\beta_0$   -42.42
x1             $\beta_1$   26.14

Metrics        Value
$\sigma$       19.75
$R^2$          0.95
---------  ------  ----
```

Wait. There's no noise. Why is the R^2 less than 100%? And the error is now 19.7 units. Let's look at a scatter plot of the data, the model, and the residuals:

```
[13]: figure = plt.figure(figsize=(10,6))

axes = figure.add_subplot(2, 1, 1)

axes.scatter(data.x1, data.y1, color="dimgray", alpha=0.5)
beta = result["coefficients"]
axes.plot(data.x1, [beta[ 0] + beta[ 1] * x for x in data.x1], '-', color="firebrick")
axes.set_title(result[ "formula"])
axes.set_xlim((0, 10))
axes.set_xlabel( "x_1")

axes = figure.add_subplot(2, 1, 2)

keyed_values = sorted(zip(data.x1, result["residuals"]), key=lambda x: x[ 0])

residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( "x_1 v. residuals")
axes.set_xlabel( "x_1 sorted")
axes.set_ylim((-30.0, 30.0))

plt.show()
plt.close()
```



When plotting residuals, it's always a good idea to have a symmetric y scale. This way you can see skewed residuals more easily. If you let the automatic scaling do its thing, sometimes it will look normal-ish when it's not.

These residuals clearly have a pattern they should not have. Notice also it occurs in the absence of "noise". What we can see here is a misspecified model in its pure form. There is at least one caveat. The real data was generated by the function:

$y = 1.00 + 2.5x_1^2 + N(0, 0)$

Notice the relationship of the coefficient 2.5 to the intercept (β_0) 1.00? What if it were smaller? Let's make it 0.025 instead of 2.5.

```
[14]: data["y1"] = 1.00 + 0.025 * data.x1_sqrd

      data.describe()
```

```
[14]:                 y          x1      x1_sqrd          y1
      count  100.000000  100.000000  100.000000  100.000000
      mean    14.361599    5.344639   38.506023    1.962651
      std      7.921989    3.168796   34.046488    0.851162
      min      1.238822    0.095529    0.009126    1.000228
      25%      7.408161    2.563264    6.570634    1.164266
      50%     14.683931    5.473572   29.960363    1.749009
      75%     22.504152    8.601661   73.989326    2.849733
      max     25.866052    9.946421   98.931284    3.473282
```

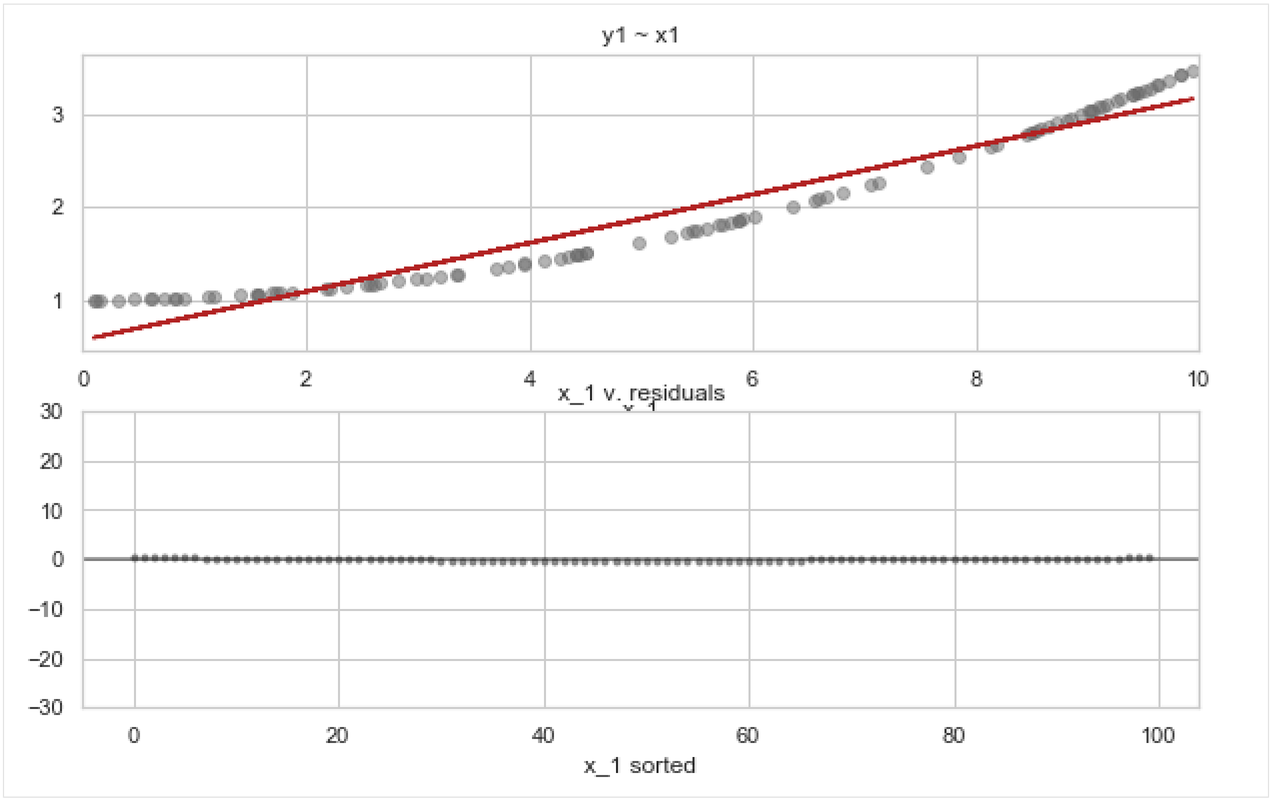
And let's estimate the model:

```
[15]: result = models.linear_regression("y1 ~ x1", data)
      print(models.simple_describe_lr(result))
```

```
Model: y1 ~ x1
---------  ------  ---
Coefficients              Value
               $\beta_0$  0.57
x1             $\beta_1$  0.26

Metrics        Value
$\sigma$       0.20
$R^2$          0.95
---------  ------  ---
```

And plot the data, model, and residuals:

[16]:
```
figure = plt.figure(figsize=(10,6))

axes = figure.add_subplot(2, 1, 1)

axes.scatter(data.x1, data.y1, color="dimgray", alpha=0.5)
beta = result["coefficients"]
axes.plot(data.x1, [beta[ 0] + beta[ 1] * x for x in data.x1], '-', color="firebrick")
axes.set_title(result[ "formula"])
axes.set_xlim((0, 10))
axes.set_xlabel( "x_1")

axes = figure.add_subplot(2, 1, 2)

keyed_values = sorted(zip(data.x1, result["residuals"]), key=lambda x: x[ 0])

residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( "x_1 v. residuals")
axes.set_xlabel( "x_1 sorted")
axes.set_ylim((-30.0, 30.0))

plt.show()
plt.close()
```



If we compare this with the original model, our R^2 is the same but the σ is significantly smaller. Although the true data was generated by x_1^2, because it has such a small effect on y, not knowing this creates less of a problem, generates a smaller error. We'll be able to see later that such small effects can easily get swamped by larger ones elsewhere.

These experiments also emphasize the importance of working with synthetic data. The only way to really discover all the possibilities is to explore them. Working with synthetic data gives you a safety net. The fun part is when you generate data and it doesn't have the pattern you expected.

So based on the pattern in the residuals, we conclude that the real model includes a squared term. We'll see in the next section how to handle that. Let's continue with our exploration of residuals.

Let's put y_1 back:

2.2.3 A Square Root Term

Let's look at another one. What if the true generating function is:

$y = 1.00 + 2.5\sqrt{x_1} + N(0, 0)$

and we estimate:

$\hat{y} = 1.00 + 2.5 x_1 + \epsilon$

?

[17]:
```
data["x1_sqrt"] = np.sqrt( data.x1)
data["y1"] = 1.00 + 2.5 * data.x1_sqrt

data.describe()
```

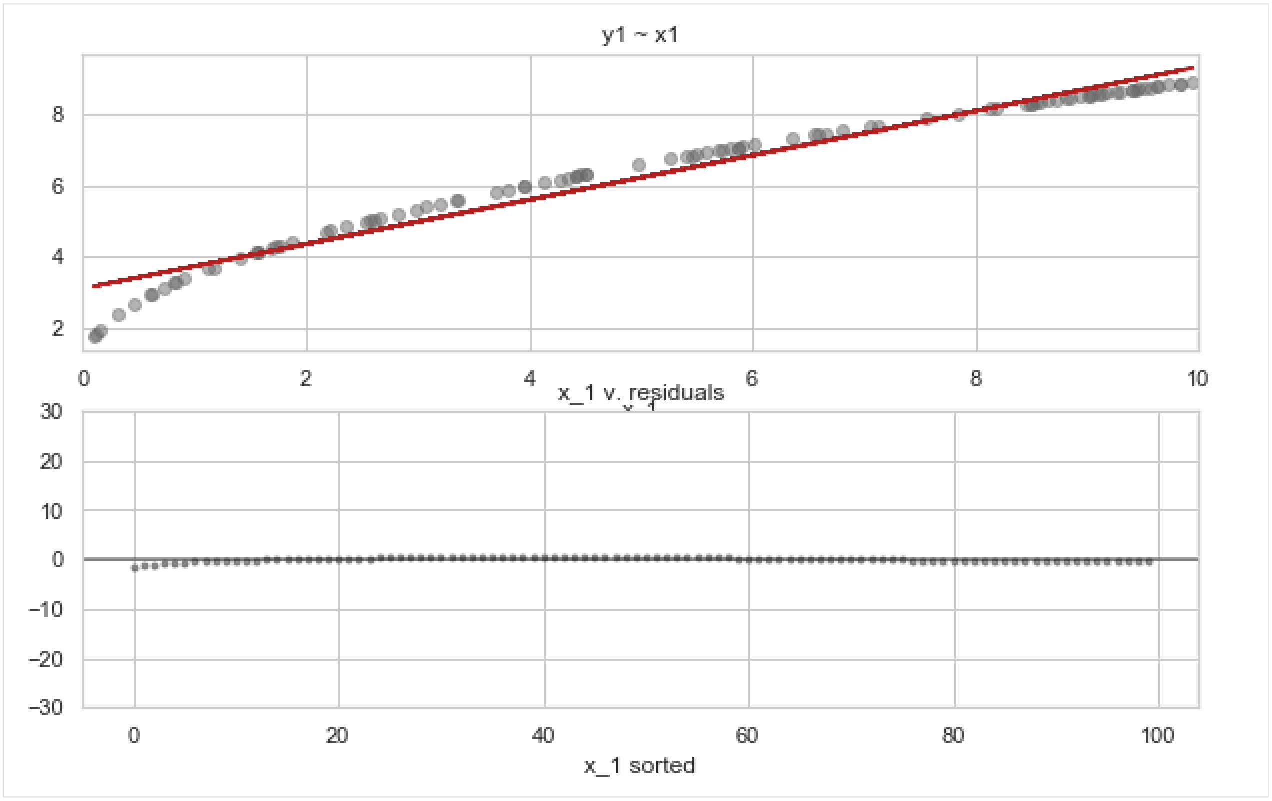
[17]:
```
                y          x1      x1_sqrd          y1      x1_sqrt
count  100.000000  100.000000  100.000000  100.000000  100.000000
mean    14.361599    5.344639   38.506023    6.426707    2.170683
std      7.921989    3.168796   34.046488    1.998699    0.799480
min      1.238822    0.095529    0.009126    1.772694    0.309078
25%      7.408161    2.563264    6.570634    5.002526    1.601010
50%     14.683931    5.473572   29.960363    6.848908    2.339563
75%     22.504152    8.601661   73.989326    8.332138    2.932855
max     25.866052    9.946421   98.931284    8.884487    3.153795
```

Again, we don't the true generating function so we start with x_1:

[18]:
```
result = models.linear_regression("y1 ~ x1", data)
print(models.simple_describe_lr(result))
```

```
Model: y1 ~ x1
--------  ------  ---
Coefficients              Value
              $\beta_0$   3.12
x1            $\beta_1$   0.62

Metrics       Value
$\sigma$      0.39
$R^2$         0.96
--------  ------  ---
```

[19]:
```
figure = plt.figure(figsize=(10,6))

axes = figure.add_subplot(2, 1, 1)

axes.scatter(data.x1, data.y1, color="dimgray", alpha=0.5)
beta = result["coefficients"]
axes.plot(data.x1, [beta[ 0] + beta[ 1] * x for x in data.x1], '-', color="firebrick")
axes.set_title(result[ "formula"])
axes.set_xlim((0, 10))
axes.set_xlabel( "x_1")

axes = figure.add_subplot(2, 1, 2)

keyed_values = sorted(zip(data.x1, result["residuals"]), key=lambda x: x[ 0])

residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( "x_1 v. residuals")
axes.set_xlabel( "x_1 sorted")
axes.set_ylim((-30.0, 30.0))

plt.show()
plt.close()
```



We have the opposite pattern.

We can go in a lot of different directions from here. For example, all of the models so far are noiseless. . . there are no “unknown unknowns”. Additionally, x_1 is from a uniform distribution. . . and not one of the myriad other distributions that x_i could be from.

2.2.4 Noise

When we start thinking about generating functions for fake data, we need to think a bit about how the pieces all fit together. Oddly enough, this also allows us to interpret the various parameters (β_i, σ) of real models better.

Remember that coefficients ($\beta_i, i > 0$) are both the *effect* of a variable and the *scaling* or *translation* of it into the units of y. Consider the following model:

$y = 10.0 + 2.5x_1 - 3.5x_2 + 1.7x_3 - 2.9x_4 + N(0, 2.5)$

We can read this as starting with the value “10.0”, adding 2.5 units of y for each unit of x_1, subtracting 3.5 units of y for each unit of x_2, adding 1.7 units of y per unit of x_3 and subtracting 2.9 units of y for each unit of x_4. Finally, we add a bit of noise to represent the “(un)known unknowns”.

Where this all ends up depends on the domains of the variables x_1, x_2, x_3, x_4. We do know that if and when all the x’s are equal to zero, y is effectively:

$y = N(10.0, 2.5)$

if you remember from the earliest module, the coefficient of variation, v, is $\frac{std}{mean}$ or 2.5/10 = 0.25 (25.0%). That may be a lot of noise depending on the domains of x. This is the key point I want you take away. These can lead to very different results. Here is the current model we’ve been working with using ϵ = 0.25, 2.5, 5.0 and the resulting models:

```
[20]: e = stats.norm.rvs(0, 0.25, 100)
      data[ "y"] = 1.00 + 2.5 * x1 + e

      data.describe()
```

```
[20]:                 y          x1     x1_sqrd          y1     x1_sqrt
      count  100.000000  100.000000  100.000000  100.000000  100.000000
      mean    14.353545    5.344639   38.506023    6.426707    2.170683
      std      7.923677    3.168796   34.046488    1.998699    0.799480
      min      1.133556    0.095529    0.009126    1.772694    0.309078
      25%      7.349998    2.563264    6.570634    5.002526    1.601010
      50%     14.852596    5.473572   29.960363    6.848908    2.339563
      75%     22.448807    8.601661   73.989326    8.332138    2.932855
      max     25.367600    9.946421   98.931284    8.884487    3.153795
```

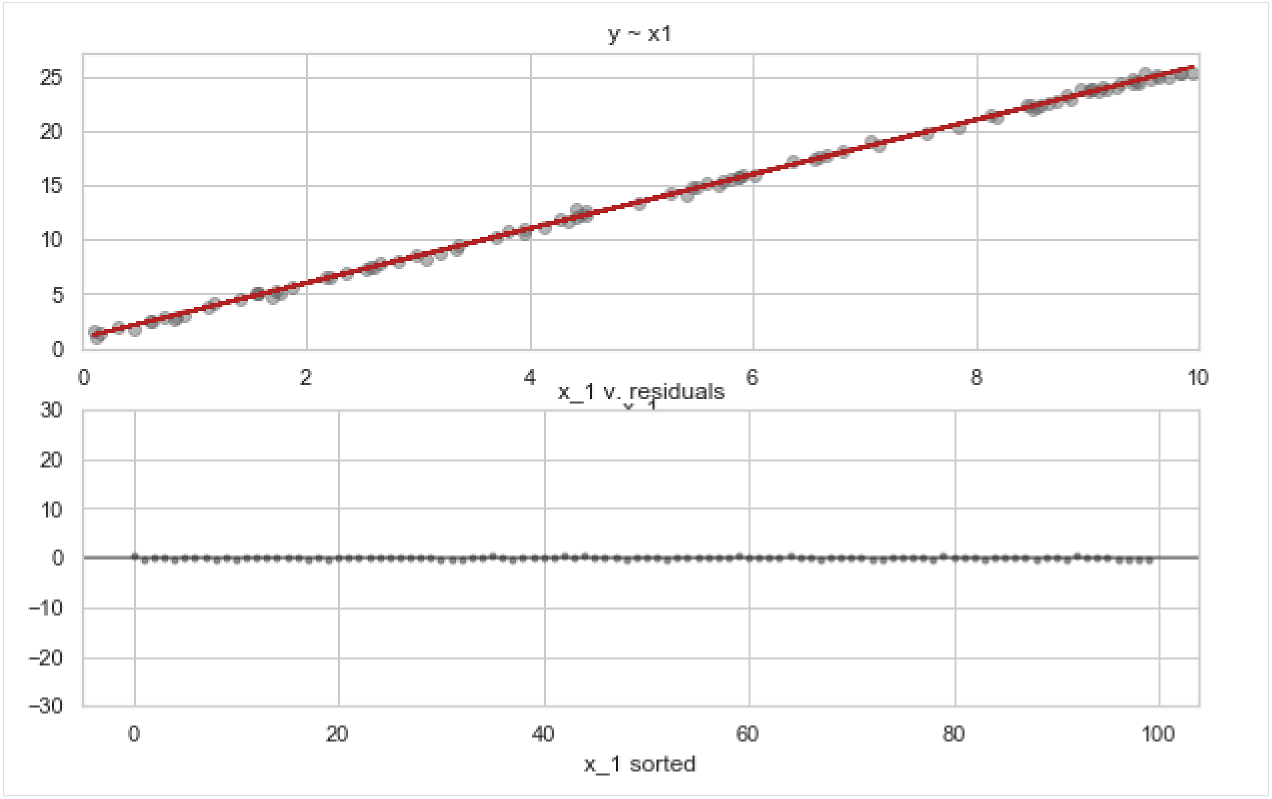
And, you guessed it, let's estimate the model:

```
[21]: result = models.linear_regression("y ~ x1", data)
      print(models.simple_describe_lr(result))
```

```
Model: y ~ x1
--------  ------  ---
Coefficients             Value
              $\beta_0$   0.99
x1            $\beta_1$   2.50

Metrics       Value
$\sigma$      0.22
$R^2$         1.00
--------  ------  ---
```

This isn't a lot of noise but it is some:

```
[22]: figure = plt.figure(figsize=(10,6))

      axes = figure.add_subplot(2, 1, 1)

      axes.scatter(data.x1, data.y, color="dimgray", alpha=0.5)
      beta = result["coefficients"]
      axes.plot(data.x1, [beta[ 0] + beta[ 1] * x for x in data.x1], '-', color="firebrick")
      axes.set_title(result[ "formula"])
      axes.set_xlim((0, 10))
      axes.set_xlabel( "x_1")

      axes = figure.add_subplot(2, 1, 2)

      keyed_values = sorted(zip(data.x1, result["residuals"]), key=lambda x: x[ 0])

      residuals = [x[ 1][ 0] for x in keyed_values]

      axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
      axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
      axes.set_title( "x_1 v. residuals")
      axes.set_xlabel( "x_1 sorted")
      axes.set_ylim((-30.0, 30.0))

      plt.show()
      plt.close()
```



Let’s add more noise.

```
[23]: e = stats.norm.rvs(0, 2.5, 100)
      data[ "y"] = 1.00 + 2.5 * x1 + e
```

We’ll estimate the model again:

```
[24]: result = models.linear_regression("y ~ x1", data)
      print(models.simple_describe_lr(result))
```

```
Model: y ~ x1
--------  ------  ---
Coefficients              Value
              $\beta_0$   1.17
x1            $\beta_1$   2.40

Metrics       Value
$\sigma$      2.12
$R^2$         0.93
--------  ------  ---
```

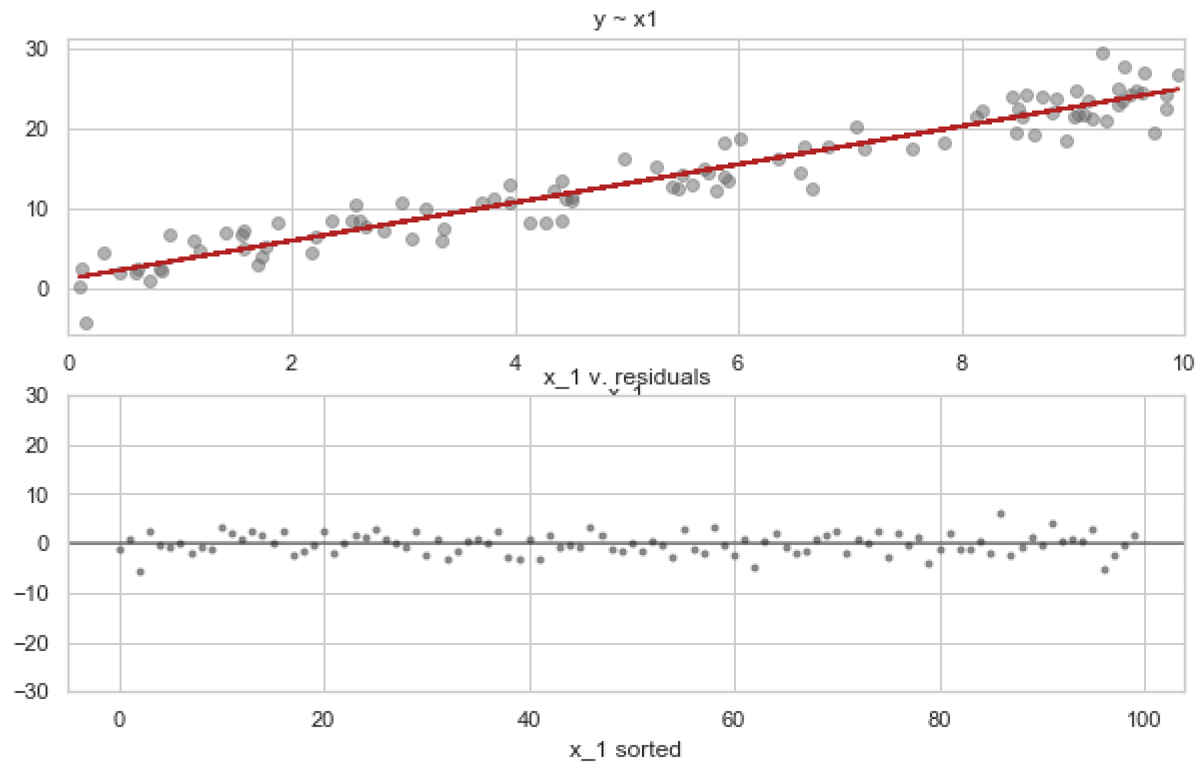
And plot the data, model, and residuals:

```
[25]: figure = plt.figure(figsize=(10,6))

      axes = figure.add_subplot(2, 1, 1)

      axes.scatter(data.x1, data.y, color="dimgray", alpha=0.5)
      beta = result["coefficients"]
      axes.plot(data.x1, [beta[ 0] + beta[ 1] * x for x in data.x1], '-', color="firebrick")
```

(continues on next page)

(continued from previous page)

```
axes.set_title(result[ "formula"])
axes.set_xlim((0, 10))
axes.set_xlabel( "x_1")

axes = figure.add_subplot(2, 1, 2)

keyed_values = sorted(zip(data.x1, result["residuals"]), key=lambda x: x[ 0])

residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( "x_1 v. residuals")
axes.set_xlabel( "x_1 sorted")
axes.set_ylim((-30.0, 30.0))

plt.show()
plt.close()
```



This is closer to the residuals we might start to see. Let's increase the noise one more time:

```
[26]: e = stats.norm.rvs(0, 10.0, 100)
      data[ "y"] = 1.00 + 2.5 * x1 + e
```

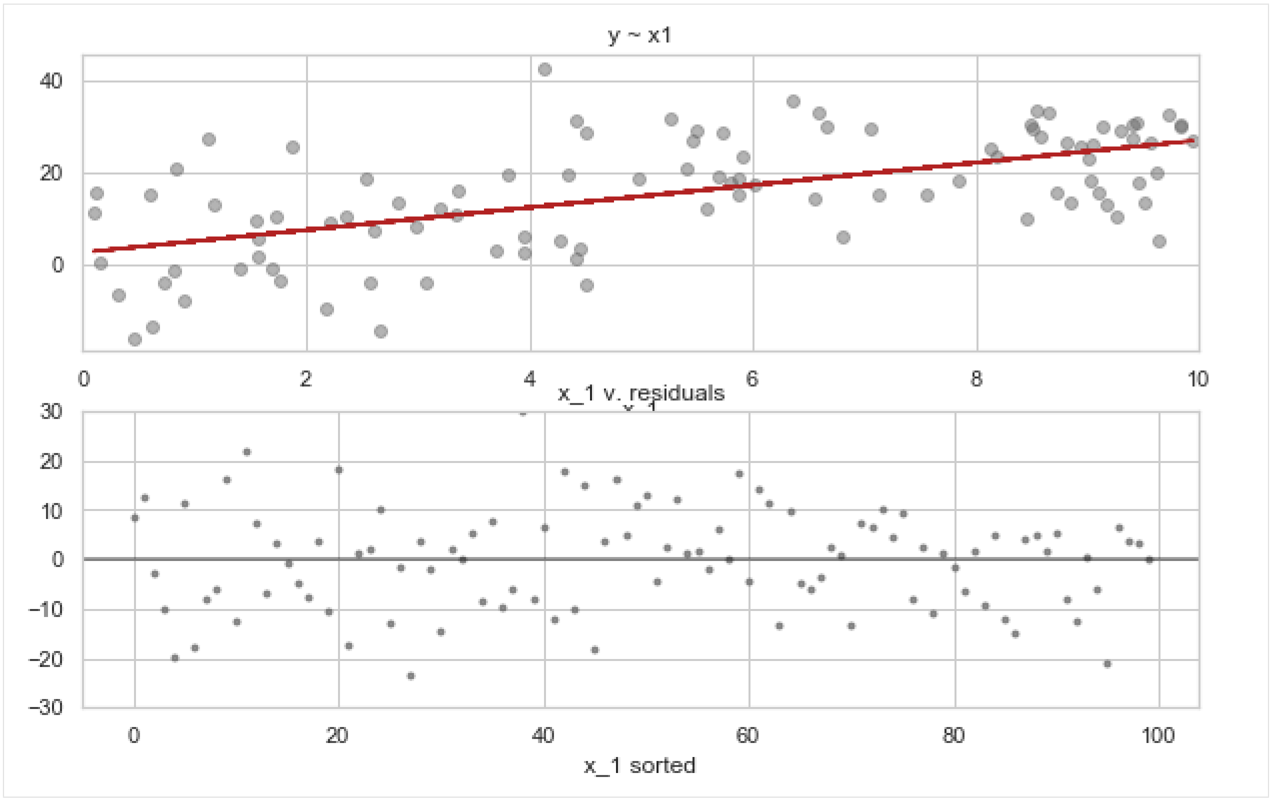
Fit our model:

```
[27]: result = models.linear_regression("y ~ x1", data)
      print(models.simple_describe_lr(result))
```

```
Model: y ~ x1
---------  ------  ---
Coefficients             Value
              $\beta_0$  2.63
x1            $\beta_1$  2.45

Metrics       Value
$\sigma$      10.32
$R^2$         0.36
---------  ------  ---
```

And chart the data, model, and residuals:

[28]:
```
figure = plt.figure(figsize=(10,6))

axes = figure.add_subplot(2, 1, 1)

axes.scatter(data.x1, data.y, color="dimgray", alpha=0.5)
beta = result["coefficients"]
axes.plot(data.x1, [beta[ 0] + beta[ 1] * x for x in data.x1], '-', color="firebrick")
axes.set_title(result[ "formula"])
axes.set_xlim((0, 10))
axes.set_xlabel( "x_1")

axes = figure.add_subplot(2, 1, 2)

keyed_values = sorted(zip(data.x1, result["residuals"]), key=lambda x: x[ 0])

residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( "x_1 v. residuals")
axes.set_xlabel( "x_1 sorted")
axes.set_ylim((-30.0, 30.0))

plt.show()
plt.close()
```



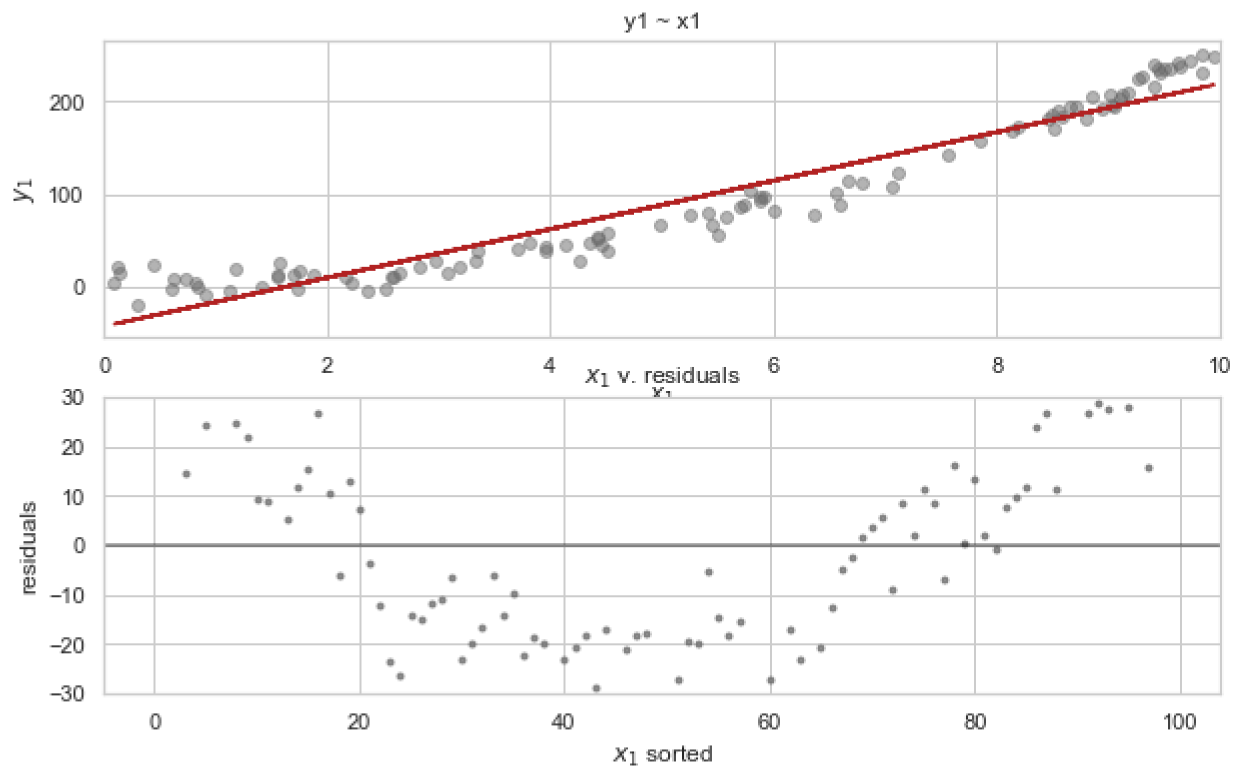
This looks a bit more realistic but what if we revisit one of our earlier models with a squared term? We can look all in one fell swoop.

```
[29]: e = stats.norm.rvs(0, 10.0, 100)
data[ "y1"] = 1.00 + 2.5 * data.x1_sqrd + e

result = models.linear_regression("y1 ~ x1", data)
models.simple_describe_lr(result)

figure = plt.figure(figsize=(10,6))

axes = figure.add_subplot(2, 1, 1)

axes.scatter(data.x1, data.y1, color="dimgray", alpha=0.5)
beta = result["coefficients"]
axes.plot(data.x1, [beta[ 0] + beta[ 1] * x for x in data.x1], '-', color="firebrick")
axes.set_title(result[ "formula"])
axes.set_xlim((0, 10))
axes.set_xlabel(r"$x_1$")
axes.set_ylabel(r"$y_1$")
axes = figure.add_subplot(2, 1, 2)

keyed_values = sorted(zip(data.x1, result["residuals"]), key=lambda x: x[ 0])

residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( r"$x_1$ v. residuals")
axes.set_xlabel( r"$x_1$ sorted")
```

(continues on next page)

(continued from previous page)

```
axes.set_ylabel("residuals")
axes.set_ylim((-30.0, 30.0))

plt.show()
plt.close()
```



2.2.5 Full Multivariate Model

You need to look at the residuals for all of your numerical variables. Let's look at the multivariate model from above:

$y = 10.0 + 2.5x_1 - 3.5x_2 + 1.7x_3 - 2.9x_4 + N(0, 5.0)$

we can we can use as a data generating model. To make things interesting, we're only going to use the first two variables, x_1 and x_2:

$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon$

We'll generate the variables we need from Normal distributions:

```
[30]: data["x2"] = stats.norm.rvs(5, 0.5, 100)
data["x3"] = stats.norm.rvs(7, 0.2, 100)
data["x4"] = stats.norm.rvs(8, 0.3, 100)
data["x5"] = stats.norm.rvs(3, 0.3, 100)
data["y"] = 10.0 + 2.5 * data.x1 - 3.5 * data.x2 + 1.7 * data.x3 - 2.9 * data.x4 +␣
↪stats.norm.rvs(0, 5,100)
```

And describe it:

[31]:
```
data.describe()
```

[31]:
```
                 y          x1     x1_sqrd          y1     x1_sqrt          x2  \
count   100.000000  100.000000  100.000000  100.000000  100.000000  100.000000
mean     -6.342219    5.344639   38.506023   97.683711    2.170683    5.028293
std       8.379301    3.168796   34.046488   86.167871    0.799480    0.517574
min     -26.411007    0.095529    0.009126  -19.351237    0.309078    3.927467
25%     -12.851496    2.563264    6.570634   19.135166    1.601010    4.716104
50%      -4.895713    5.473572   29.960363   77.022920    2.339563    5.020235
75%      -0.553659    8.601661   73.989326  186.761802    2.932855    5.371569
max      12.850611    9.946421   98.931284  250.967081    3.153795    6.151773

                x3          x4          x5
count   100.000000  100.000000  100.000000
mean      6.977789    8.074251    3.017084
std       0.201689    0.320094    0.325489
min       6.511310    7.109811    2.099009
25%       6.834748    7.889215    2.805623
50%       6.980163    8.058954    3.029154
75%       7.134441    8.276449    3.232930
max       7.404249    8.899653    3.767994
```

Let's estimate the first model:

[32]:
```
result = models.linear_regression("y ~ x1 + x2", data)
print(models.simple_describe_lr(result))
```

```
Model: y ~ x1 + x2
--------  ------  ---
Coefficients             Value
              $\beta_0$  -6.10
x1            $\beta_1$  2.13
x2            $\beta_2$  -2.32

Metrics       Value
$\sigma$      4.74
$R^2$         0.69
--------  ------  ---
```

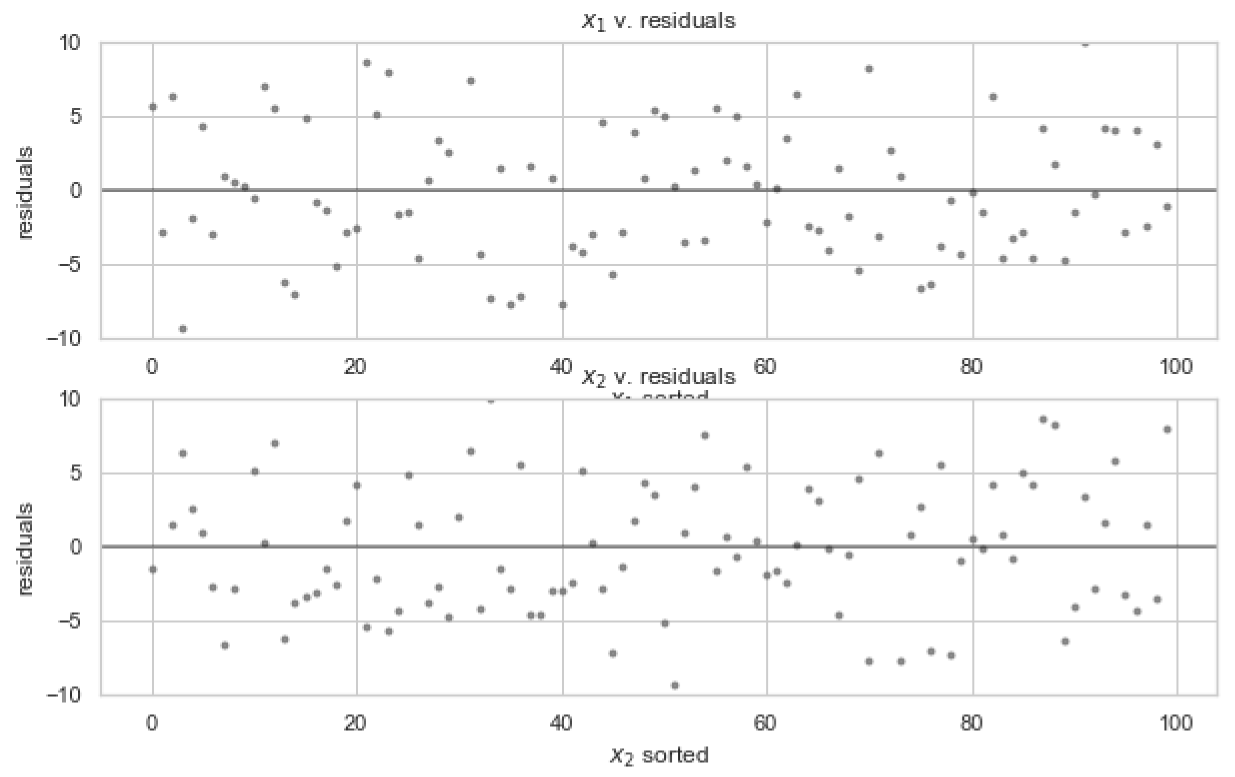
The R^2 isn't bad but there's clearly room for improvement. We've included the obvious features. We have a few more variables we didn't include but which ones should we pick? Let's look at the residuals:

[33]:
```
figure = plt.figure(figsize=(10,6))

axes = figure.add_subplot(2, 1, 1)

keyed_values = sorted(zip(data.x1, result["residuals"]), key=lambda x: x[ 0])
residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( r"$x_1$ v. residuals")
axes.set_xlabel( r"$x_1$ sorted")
axes.set_ylabel("residuals")
axes.set_ylim((-10.0, 10.0))

axes = figure.add_subplot(2, 1, 2)
```

(continues on next page)

(continued from previous page)

```
keyed_values = sorted(zip(data.x2, result["residuals"]), key=lambda x: x[ 0])
residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( r"$x_2$ v. residuals")
axes.set_xlabel( r"$x_2$ sorted")
axes.set_ylabel("residuals")
axes.set_ylim((-10.0, 10.0))

plt.show()
plt.close()
```



The residuals don't show any patterns. The residuals are in a weird format, let's make them a regular array:

```
[34]: residuals = [r[0] for r in result["residuals"]]
```

And let's think about what residuals are. Residuals are the difference between the observed y and the estimated y, $\hat{y}$. $\hat{y}$ is based on a linear equation. If the equation were missing a term, $\beta_3 x_3$, for example, then the residuals must be missing it, too. Because the β_3 is a constant, the correlation must exist between the residuals and the variable. Does it?

```
[35]: print("x_1 = {0:.2f}".format(stats.pearsonr(residuals, data.x1)[0]))
```

```
x_1 = 0.00
```

We're not missing x_1. x_2?

```
[36]: print("x_2 = {0:.2f}".format(stats.pearsonr(residuals, data.x2)[0]))
```

```
x_2 = 0.00
```

We're not missing x_2 x_3?

```
[37]: print("x_3 = {0:.2f}".format(stats.pearsonr(residuals, data.x3)[0]))
```

```
x_3 = 0.20
```

We might be missing x_3. x_4?

```
[38]: print("x_4 = {0:.2f}".format(stats.pearsonr(residuals, data.x4)[0]))
```

```
x_4 = -0.08
```

We might be missing x_4. x_5?

```
[39]: print("x_5 = {0:.2f}".format(stats.pearsonr(residuals, data.x5)[0]))
```

```
x_5 = 0.03
```

I added in x_5 so that you could see that this is not foolproof. We know that we are not missing x_5. We also know that we are missing both x_3 and x_4 but the correlations with the residuals are not huge. This will depend on the noise in the data, $N(0, \sigma)$, which we cannot know.

If there is a clear domain reason for including this values, then we should do it. Of course, this is science, so we might consider if there *must* be a reason:

```
[40]: result = models.linear_regression("y ~ x1 + x2 + x3 + x4 + x5", data)
      print(models.simple_describe_lr(result))
```

```
Model: y ~ x1 + x2 + x3 + x4 + x5
--------  ------  ----
Coefficients          Value
              $\beta_0$  -32.05
x1            $\beta_1$  2.16
x2            $\beta_2$  -2.62
x3            $\beta_3$  5.09
x4            $\beta_4$  -1.26
x5            $\beta_5$  0.66

Metrics       Value
$\sigma$      4.69
$R^2$         0.70
--------  ------  ----
```

These are all a bit off their true values:

$y = 10.0 + 2.5x_1 - 3.5x_2 + 1.7x_3 - 2.9x_4 + N(0, 5.0)$

versus

$\hat{y} = -32.1 + 2.2x_1 - 2.6x_2 + 5.1x_3 - 1.3x_4 + 0.66x_5 + N(0, 4.7)$

Can the Bootstrap help?

```
[41]: result = models.bootstrap_linear_regression("y ~ x1 + x2 + x3 + x4 + x5", data)
      print(models.describe_bootstrap_lr(result))
```

```
Model: y ~ x1 + x2 + x3 + x4 + x5
--------  ------  ----  -----  ---
                                 95% BCI
```

(continues on next page)

(continued from previous page)

```
Coefficients               Mean     Lo        Hi
              $\beta_0$  -32.05   -75.57    13.23
x1            $\beta_1$  2.16     1.89      2.53
x2            $\beta_2$  -2.62    -5.11     -0.59
x3            $\beta_3$  5.09     1.06      11.37
x4            $\beta_4$  -1.26    -5.17     1.59
x5            $\beta_5$  0.66     -2.89     3.54

Metrics       Mean        Lo       Hi
$\sigma$      4.69        3.83     5.03
$R^2$         0.70        0.64     0.81
--------  ------  ----  -----  ---
```

If we follow the advice from the previous section, all the coefficients have their expected signs except β_5 which we should remove. x_4 is a bit dodgey but we're assuming that "omniscience" wins out in this case. Still, it's interesting to see if you don't have strong domain knowledge for including a variable in the model, estimation alone may not give you a reason to keep it in.

[42]:
```
result = models.bootstrap_linear_regression("y ~ x1 + x2 + x3 + x4", data)
print(models.describe_bootstrap_lr(result))
```

```
Model: y ~ x1 + x2 + x3 + x4
--------  ------  ----  -----  ---
                                 95% BCI
Coefficients               Mean     Lo        Hi
              $\beta_0$  -28.51   -79.89    10.44
x1            $\beta_1$  2.16     1.84      2.52
x2            $\beta_2$  -2.60    -4.82     -0.78
x3            $\beta_3$  4.95     0.92      9.89
x4            $\beta_4$  -1.34    -5.23     1.80

Metrics       Mean        Lo       Hi
$\sigma$      4.67        3.86     5.02
$R^2$         0.70        0.64     0.80
--------  ------  ----  -----  ---
```

It isn't perfect but it is closer. We *know* that x_4 is in the actual data but we can't seem to recover that. Perhaps the effect is too small relative to the noise or we need more data (try to simulate getting more data). We would, of course, look at the residuals again.

2.2.6 Unequal Variance

One of the assumptions for linear regression is that the noise has an equal variance. If you happen to have unequal variance, it looks something like this:

[43]:
```
data["e"] = data.x1.apply(lambda x: stats.norm.rvs(0, 0.5 * x, 1)[0])
data["y1"] = 10.0 + 2.5 * data.x1 + data.e
data.describe()
```

[43]:
```
                y          x1      x1_sqrd          y1      x1_sqrt          x2  \
count  100.000000  100.000000  100.000000  100.000000  100.000000  100.000000
mean    -6.342219    5.344639   38.506023   23.478040    2.170683    5.028293
std      8.379301    3.168796   34.046488    8.525640    0.799480    0.517574
min    -26.411007    0.095529    0.009126   10.291584    0.309078    3.927467
25%    -12.851496    2.563264    6.570634   15.942443    1.601010    4.716104
50%     -4.895713    5.473572   29.960363   22.624271    2.339563    5.020235
```

(continues on next page)

(continued from previous page)

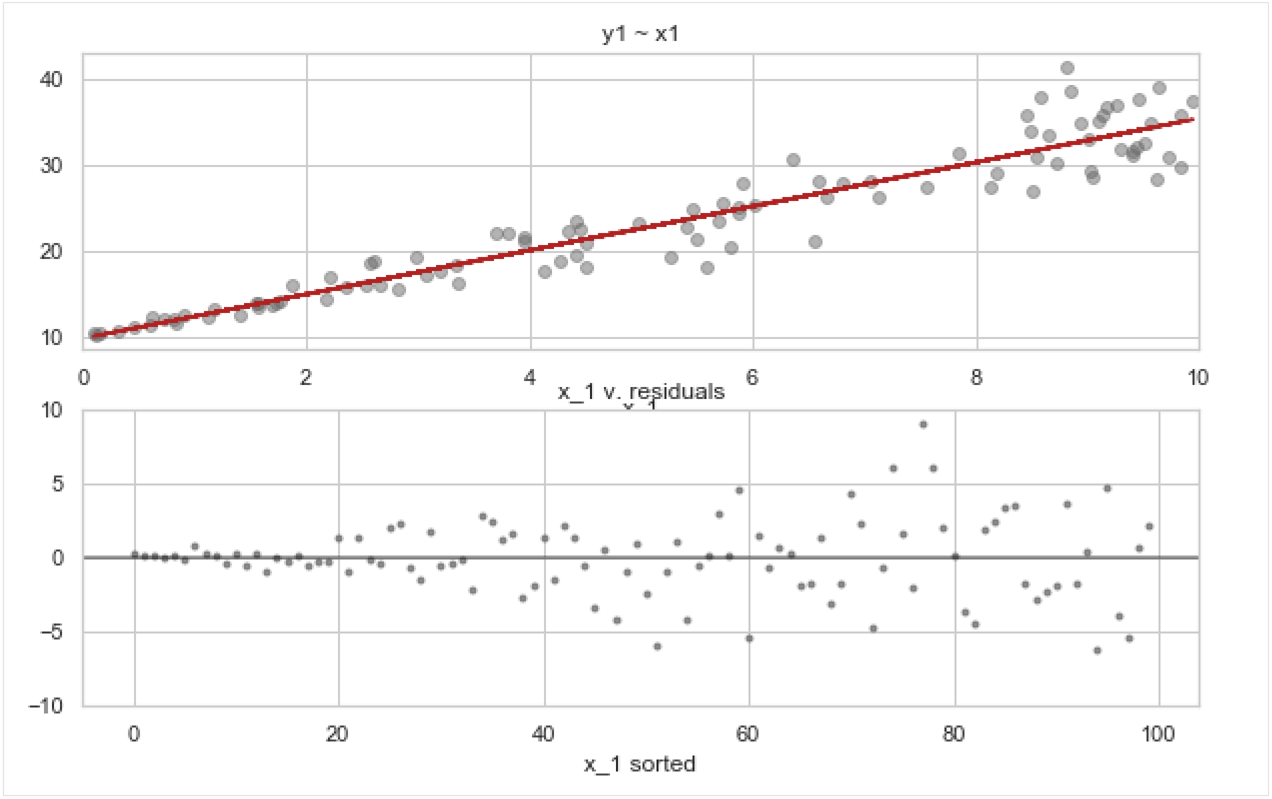
```
75%     -0.553659    8.601661   73.989326   30.726388    2.932855    5.371569
max     12.850611    9.946421   98.931284   41.381870    3.153795    6.151773

               x3          x4          x5           e
count  100.000000  100.000000  100.000000  100.000000
mean     6.977789    8.074251    3.017084    0.116441
std      0.201689    0.320094    0.325489    2.604933
min      6.511310    7.109811    2.099009   -5.807186
25%      6.834748    7.889215    2.805623   -1.406709
50%      6.980163    8.058954    3.029154   -0.066891
75%      7.134441    8.276449    3.232930    1.476717
max      7.404249    8.899653    3.767994    9.371082
```

This data doesn't look much different than any of the other synthetic data we've created before except that I've done something a little sneaky. If you look at the calculation of e, I've made the $sigma$ a function of x_1. Let's see what this does:

[44]:
```
result = models.linear_regression("y1 ~ x1", data)
print(models.simple_describe_lr(result))
```

```
Model: y1 ~ x1
--------  ------  ---
Coefficients              Value
              $\beta_0$   9.78
x1            $\beta_1$   2.56

Metrics       Value
$\sigma$      2.61
$R^2$         0.91
--------  ------  ---
```

[45]:
```
figure = plt.figure(figsize=(10,6))

axes = figure.add_subplot(2, 1, 1)

axes.scatter(data.x1, data.y1, color="dimgray", alpha=0.5)
beta = result["coefficients"]
axes.plot(data.x1, [beta[ 0] + beta[ 1] * x for x in data.x1], '-', color="firebrick")
axes.set_title(result[ "formula"])
axes.set_xlim((0, 10))
axes.set_xlabel( "x_1")

axes = figure.add_subplot(2, 1, 2)

keyed_values = sorted(zip(data.x1, result["residuals"]), key=lambda x: x[ 0])

residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( "x_1 v. residuals")
axes.set_xlabel( "x_1 sorted")
axes.set_ylim((-10.0, 10.0))

plt.show()
plt.close()
```



Notice that the residuals get larger as x_1 gets larger. This violates the assumption of equal variance. This doesn't generally affect the estimation of the coefficients but it can affect the overall performance of the model. This is just something to be aware of.

It's worth noting that EDA will often uncover many of these relationships before you even get to look at residuals.

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%matplotlib inline
```

[3]:
```
import numpy as np
import scipy.stats as stats
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import random
import patsy
import sklearn.linear_model as linear

sns.set(style="whitegrid")

import sys
sys.path.append('resources')
import models
```

2.3 Transformations

At several points in the previous discussions we have hinted at the possibility that transformations might improve the performance and or interpretability of a linear model. In this section, we'll talk about a variety of transformations that accomplish these goals.

2.3.1 Scaling

Let's review the two jobs that a single β does.

Imagine we want to predict some y that has a range of 27 to 89 and we want to model it with an x with the range 345 to 763. Our first observation is that if we have an equation of the form $y = \beta_0 + \beta_1 x$ then β_1 must *at least* be on the order of 0.# (1/10th) because y and x differ in magnitude by that much. Therefore, before before β_1 does anything else, it must *scale* the value of x to be of the same magnitude of y.

It follows that if we start adding features of different magnitudes, some of them in the 000's and some in the 0.000's, then the β's must all adjust to the scales of their respective features. This makes it difficult to determine the relative contributions of each predictor (as measured by β_i).

Additionally, we have the problem of interpreting β_0, the intercept, when all x_i are zero. It doesn't make any sense for someone to have zero IQ, zero height, zero weight, etc.

Therefore, in order to make coefficients more interpreterable, it is often desirable to transform the variables ahead of time to a common scale. There are two such transformations:

1. subtract each value x_i from its mean $\bar{x_i}$. This creates mean *centered* feature.
2. Take mean centered data and divide through by x_i's standard deviation, σ_{x_i}. This creates a mean *scaled* feature.

We only apply this transformation to *numerical* features and not dummy variables.

Mean scaling accomplishes two things:

1. Zero is a meaningful value for each x_i, the mean value of x_i and thus the intercept β_0 is interpretable as the value when all x_i are at their mean.
2. Each variable is projected into the standard normal distribution (mean of 0 and standard deviation of 1) so that they're all on the same scale. A unit change is a change of 1 standard deviation in the standard normal distribution.

Mean *centering* accomplishes only the first thing.

However, if you use mean scaling for a *logistic* regression, then "+1" is an entire standard deviation of data. This means that "Divide by 4" won't work and you'll need to figure out what "+1" in real units is in standard deviations. Often it is better to just use mean *centered* data for logistic regression.

Finally, mean scaling puts the units in "standard units" and mean centering leaves the units in the "natural" units such as feet, square feet, IQ points, etc. This is often desirable.

```
[4]: def mean_scale( df, variable):
         x_bar = df[ variable].mean()
         std = df[ variable].std()
         scaled_variable = (df[ variable] - x_bar) / (2.0 * std) # suggested by Gelman
         df[ variable + "_scaled"] = scaled_variable

     def mean_center( df, variable):
         x_bar = df[ variable].mean()
         scaled_variable = df[ variable] - x_bar
         df[ variable + "_centered"] = scaled_variable
```

Back to child IQs. . .

```
[5]: child_iq = pd.read_csv( "resources/child_iq.tsv", sep="\t")
```

```
[6]: mean_scale(child_iq, "child_iq")
     mean_scale(child_iq, "mom_iq")
     mean_scale(child_iq, "mom_age")
```

```
[7]: child_iq.head()
```

```
[7]:    child_iq  mom_hs      mom_iq  mom_work  mom_age  child_iq_scaled  \
     0        65       1  121.117529         4       27        -0.533966
     1        98       1   89.361882         4       25         0.274434
     2        85       1  115.443165         4       27        -0.044027
     3        83       1   99.449639         3       25        -0.093021
     4       115       1   92.745710         4       27         0.690882

        mom_iq_scaled  mom_age_scaled
     0       0.703918        0.780114
     1      -0.354604        0.409891
     2       0.514772        0.780114
     3      -0.018345        0.409891
     4      -0.241810        0.780114
```

Here's the unscaled model:

```
[8]: model = "child_iq ~ mom_iq + mom_age"
     result = models.bootstrap_linear_regression(model, data=child_iq)
     print(models.describe_bootstrap_lr(result))
```

```
Model: child_iq ~ mom_iq + mom_age
--------  ------  ---  -----  ---
                               95% BCI
Coefficients             Mean    Lo       Hi
              $\beta_0$  17.60   2.07     37.50
mom_iq        $\beta_1$  0.60    0.51     0.71
mom_age       $\beta_2$  0.39    -0.30    0.94

Metrics       Mean       Lo     Hi
$\sigma$      18.26      17.01  19.21
$R^2$         0.20       0.15   0.27
--------  ------  ---  -----  ---
```

Although mom_iq is in the same exact units as child_iq, mom_age and child_iq do not share the same units and they are even of slightly different magnitudes. We know that neither mom_iq nor mom_age can be zero so what is β_0? This is the same model and the same problem as before.

Here's a completely scaled model:

```
[9]: model = "child_iq_scaled ~ mom_iq_scaled + mom_age_scaled"
     result = models.bootstrap_linear_regression(model, data=child_iq)
     print(models.describe_bootstrap_lr(result))
```

```
Model: child_iq_scaled ~ mom_iq_scaled + mom_age_scaled
---------  -------  ---  -----  ---
                                 95% BCI
Coefficients               Mean    Lo       Hi
                $\beta_0$  -0.00   -0.04    0.05
mom_iq_scaled   $\beta_1$  0.44    0.37     0.52
```

(continues on next page)

(continued from previous page)

```
mom_age_scaled  $\beta_2$  0.05   -0.04    0.12

Metrics         Mean       Lo     Hi
$\sigma$        0.45       0.41   0.47
$R^2$           0.20       0.15   0.28
---------  ------  ---  -----  ---
```

How do we interpret this model? At the mean value of mom_iq (=0) and mom_age (=0), child_iq is at *its* mean. If β_0 had been 6.39 then we would say, "at the mean values of mom_iq and mom_age, we add 6.39 to mean of the child IQ". This gets are the heart of what we've been saying all along: linear models are about estimating means.

One of the strange side effects of mean scaled (and mean centered) models is that you need to calculate and keep all the means around so you know what they are.

It might make a bit more sense to *not* scale the target variable:

[10]:
```
model = "child_iq ~ mom_iq_scaled + mom_age_scaled"
result = models.bootstrap_linear_regression(model, data=child_iq)
print(models.describe_bootstrap_lr(result))
```

```
Model: child_iq ~ mom_iq_scaled + mom_age_scaled
---------  ------  ---  -----  ---
                                   95% BCI
Coefficients               Mean    Lo       Hi
                $\beta_0$  86.80   85.56    88.42
mom_iq_scaled   $\beta_1$  18.11   14.70    21.55
mom_age_scaled  $\beta_2$  2.10    -1.42    5.59

Metrics         Mean       Lo     Hi
$\sigma$        18.26      16.86  19.64
$R^2$           0.20       0.13   0.29
---------  ------  ---  -----  ---
```

This is easier to interpret. When mom_iq and mom_age are at their mean values, the mean child IQ is 86.8. Nice.

It's worth noting that this doesn't improve the *performance* of the model, only the interpretability. This is because we're only considering linear transformations of x_i. More about *that* later.

2.3.2 Natural Baselines

Sometimes it's more natural to center the data against a natural baseline instead of the mean. That is, instead of subtracting the data from the mean, we subtract from a baseline. For example, in the case of IQ, there already exists a baseline: the average IQ is defined as an IQ of 100. You might want to define your coefficients in terms of a baseline IQ of 100. Similarly, if 30 MPG were a Federal mandate or goal, you might want to define a regression dealing with gas mileage in terms of 30 MPG by subtracting it from all the values in the data set.

Let's see.

[11]:
```
child_iq["child_iq100"] = child_iq.child_iq - 100
child_iq["mom_iq100"] = child_iq.mom_iq - 100
```

[12]:
```
model = "child_iq100 ~ mom_iq100 + mom_age_scaled"
result = models.bootstrap_linear_regression(model, data=child_iq)
print(models.describe_bootstrap_lr(result))
```

```
Model: child_iq100 ~ mom_iq100 + mom_age_scaled
---------  ------  ----  -----  ----
                                95% BCI
Coefficients              Mean     Lo        Hi
                $\beta_0$  -13.20   -14.83    -11.35
mom_iq100       $\beta_1$  0.60     0.50      0.72
mom_age_scaled  $\beta_2$  2.10     -0.58     5.93

Metrics         Mean       Lo       Hi
$\sigma$        18.26      16.75    19.33
$R^2$           0.20       0.14     0.28
---------  ------  ----  -----  ----
```

How do we interpret *this* model? If mom_iq is at the baseline (mom_iq = 0) and mom_age is at the mean (mom_age = 0), then child_iq will be 13.20 points *below* the baseline:

$100 - 13.20 = 86.8$

which is the same number we've gotten before. Again, this just changes the interpretability.

2.3.3 Save the Parameters

There is one caveat with creating interpretable models using these methods. If you create a regression with transformed variables, the coefficients are now defined in terms of *transformed* variables and you must transform any variables you use the regression equation on. For example, if a new mother came up and wanted us to predict the IQ of her child, we'd need to scale her IQ of, say, 112, with the same mean and standard deviation we used to build the model. Whenever you do any kind of transformation that involves parameters, you should save those parameter values because you will need to use them to make predictions from future data.

2.3.4 Transformations of Variables

There are other transformations that can be applied to the raw data that don't just improve interpretability: they improve performance. We have already seen the problems we might discover by looking at the residuals of a linear regression. Let's see how we might fix them.

Remember our noiseless synthetic data from the Residuals discussion:

```
[13]: np.random.seed(6734745)
```

When making the synthetic data, we leave out the intermediate calculation of x_1^2 to emphasize that we don't know that it exists.

```
[14]: x1 = np.array([10*np.random.random() for _ in range( 100)])
      ys = 1.00 + 2.5 * x1 ** 2 + stats.norm.rvs(0, 30, 100)
      data = pd.DataFrame({"y": ys, "x1": x1})
```

```
[15]: result = models.bootstrap_linear_regression("y ~ x1", data)
      print(models.describe_bootstrap_lr(result))
```

```
Model: y ~ x1
--------  ------  ----  -----  ----
                              95% BCI
Coefficients            Mean     Lo        Hi
              $\beta_0$  -43.12   -55.88    -31.60
x1            $\beta_1$  26.09    24.16     28.17
```

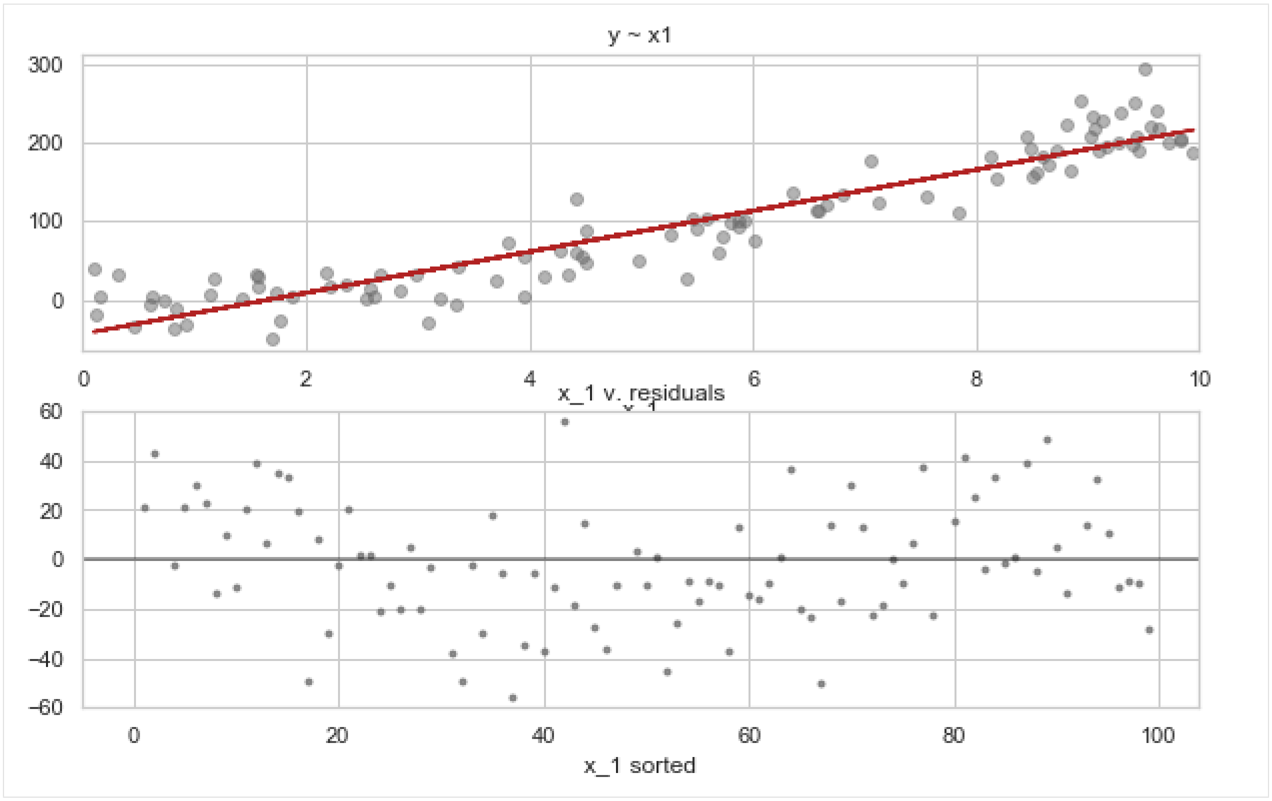
(continues on next page)

(continued from previous page)

```
Metrics        Mean       Lo      Hi
$\sigma$       30.00      24.74   33.72
$R^2$          0.88       0.84    0.91
--------  -------  ----  -----  ----
```

The R^2 is decent (88%) but the error is high. Let's plot the data:

[16]:
```
figure = plt.figure(figsize=(10,6))

axes = figure.add_subplot(2, 1, 1)

axes.scatter(data.x1, data.y, color="dimgray", alpha=0.5)
beta = result["coefficients"]
axes.plot(data.x1, [beta[ 0] + beta[ 1] * x for x in data.x1], '-', color="firebrick")
axes.set_title(result[ "formula"])
axes.set_xlim((0, 10))
axes.set_xlabel( "x_1")

axes = figure.add_subplot(2, 1, 2)

keyed_values = sorted(zip(data.x1, result["residuals"]), key=lambda x: x[ 0])

residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( "x_1 v. residuals")
axes.set_xlabel( "x_1 sorted")
axes.set_ylim((-60.0, 60.0))

plt.show()
plt.close()
```



There's definitely a "bend" to the residuals which indicates that we're over underestimating at low and high values of x_1 and overestimating at medium values of x_1. This suggests that a transformation is in order. . . *which* transformation comes from experience of looking at residuals and at histograms of features.

Since we know what the right answer is, let's see what happens when we create a new feature $x_2 = x_1^2$ and use it in our model:

```
[17]: data["x2"] = data.x1 ** 2
```

```
[18]: result = models.bootstrap_linear_regression("y ~ x2", data)
      print(models.describe_bootstrap_lr(result))
```

```
Model: y ~ x2
---------  ------  ---  -----  ---
                               95% BCI
Coefficients              Mean    Lo        Hi
              $\beta_0$  1.41    -5.25     8.81
x2            $\beta_1$  2.46    2.29      2.62

Metrics       Mean        Lo      Hi
$\sigma$      26.30       22.62   29.80
$R^2$         0.91        0.87    0.94
---------  ------  ---  -----  ---
```

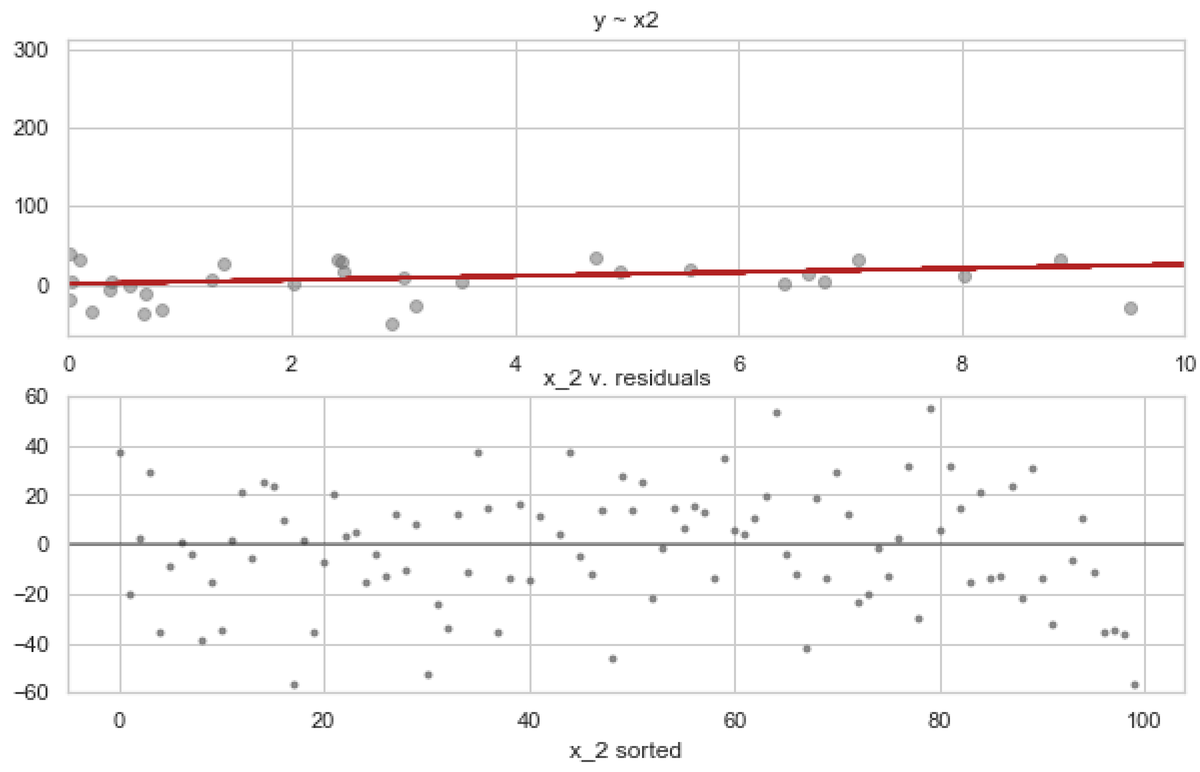
This model is much better. The error is smaller, the R^2 is larger, and the β_1 coefficient is almost the "true" value. Let's plot the data and residuals:

```
[19]: figure = plt.figure(figsize=(10,6))

      axes = figure.add_subplot(2, 1, 1)
```

(continues on next page)

(continued from previous page)

```
axes.scatter(data.x2, data.y, color="dimgray", alpha=0.5)
beta = result["coefficients"]
axes.plot(data.x2, [beta[ 0] + beta[ 1] * x for x in data.x2], '-', color="firebrick")
axes.set_title(result[ "formula"])
axes.set_xlim((0, 10))

axes = figure.add_subplot(2, 1, 2)

keyed_values = sorted(zip(data.x2, result["residuals"]), key=lambda x: x[ 0])

residuals = [x[ 1][ 0] for x in keyed_values]

axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.75)
axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="black", alpha=0.5)
axes.set_title( "x_2 v. residuals")
axes.set_xlabel( "x_2 sorted")
axes.set_ylim((-60.0, 60.0))

plt.show()
plt.close()
```



Perhaps the most surprising thing here is that the model is still linear. "Linear" doesn't mean that you can't have higher degree polynomials:

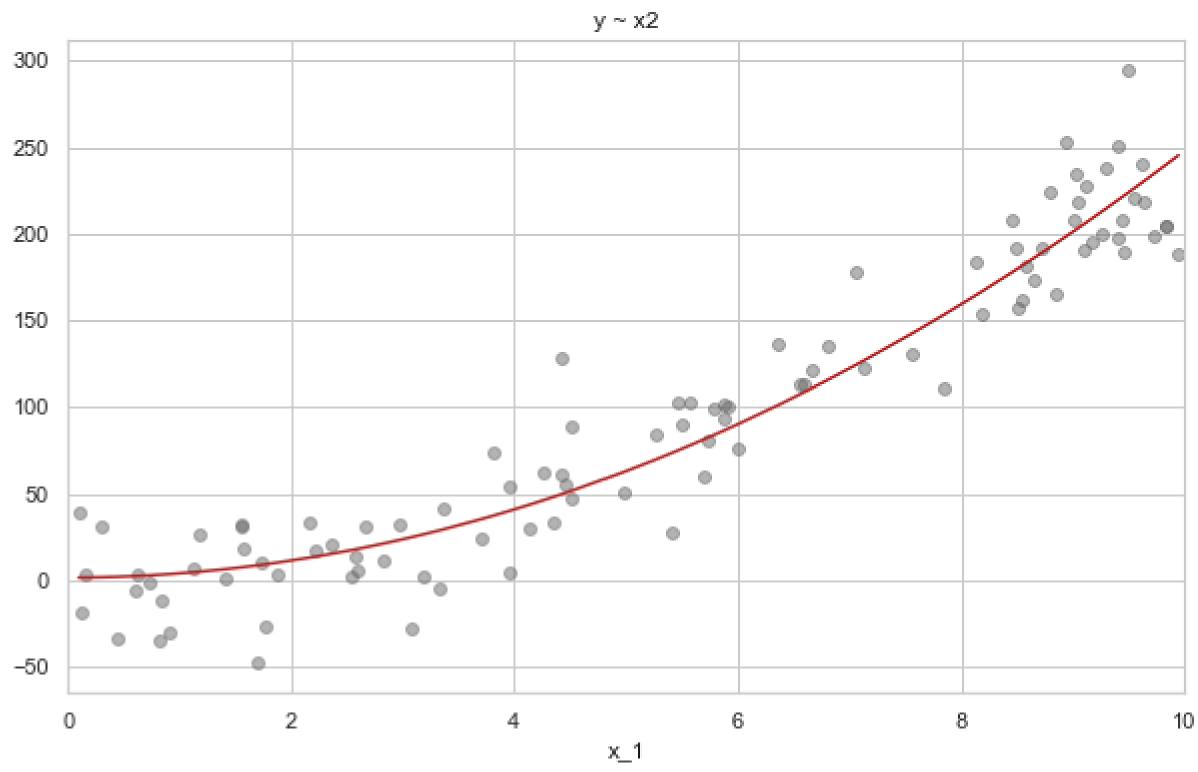
$y = \beta_0 + \beta_1 x_1^2 + \beta_2 log(x_2) + \beta_3 \sqrt{x_3} + \epsilon$

It means you can't have:

$y = \beta_0 + \beta_1^{x_1}$

And even the last one isn't completely impossible as we will see. To drive the point home, let's show the model in $y - x_1$-space:

```
[20]: figure = plt.figure(figsize=(10,6))

      axes = figure.add_subplot(1, 1, 1)

      axes.scatter(data.x1, data.y, color="dimgray", alpha=0.5)
      beta = result["coefficients"]

      data.sort_values("x1", inplace=True)

      axes.plot(data.x1, [beta[ 0] + beta[ 1] * x for x in data.x2], '-', color="firebrick")
      axes.set_title(result[ "formula"])
      axes.set_xlim((0, 10))
      axes.set_xlabel( "x_1")

      plt.show()
      plt.close()
```



The challenge will be determining what transformation to use. There are really a fairly limited number and they tend to be related to the mathematical distribution of the data or the process under consideration. Think about what it might mean that the effect is on the *square* of the variable, say, age. Similarly, because earnings or income are often exponentially distributed, it is common to use the square-root, reciprocal, or log of earnings instead of raw earnings. The kind of transformation you try may tie back to your EDA of the variables.

Below is a table of the x and y on a linear effect, $\beta = 2$, multiplied by either x^2 or $x^{1/2}$:

x	x^2	y	$x^{1/2}$	y
0	0	0	0	0
1	1	2	1	2
2	4	8	1.4	2.8
3	9	18	1.7	3.4

This demonstrates how we can have non-linear effects in x that are really linear in some transformation of x. The problem is identifying the transformation of x to use in your model.

In the case of x^2, we can see that an increase of x from 0 to 1, leads to a y of 2. However, if we increase x from 1 to 2, the effect on y is to increase to 8 which is substantial more than 2 (6 to be exact). In x, this is "increasing returns". If we increase x from 2 to 3, the effect on y is to increase to 18… by 10. This is a non-linear effect in x and y but linear in x^2. In order to model it, we square x.

We can see the opposite in the case of $x^{1/2}$. As x increases from 0 to 1, the increase in y is 2 but when we increase x from 1 to 2, y goes from 2 to 2.8… an increase of only 0.8. In x, this is "decreasing returns". Similarly, when x increases from 2 to 3, there's an increase in y to 3.4 which is an increase of only 0.6. In order to model it, we take the square root of x.

2.3.5 Numerical to Categorical

When it comes to transformations between numeric and categorical variables, the suggested practice is to deal with numeric variables if possible because you can extract more information from them. This is often at the level of measurement though (when you are recording data) rather than at the point of data transformation. For example, it is more informative and your model may be better if you measure handedness on a scale from 0 to 10 rather than as a binary "left or right". It is usually better to model something as percent correct/successful/voted than pass/fail.

However, there are times when only a categorical variable (or factor) will do. If you coded the US States and the District of Columbia as an integer from 1 to 51, what would a coefficient for this variable even mean? And even if you have a continuous variable sometimes a discretization permits you to model non-linearities.

One exception is that you may need to discretize a variable in order to handle some non-linearities. For example, imagine something depends in a non-linear way on `age`: younger and older people are "for" something ($\beta_{age} > 0$) but middle aged people are against it ($\beta_{age} < 0$).

If you used numeric `age` in your model, you would get a single value for the coefficient… perhaps even something near zero. If you used categorical age variables (a dummy for each category), you could get a positive coefficient on `age19-29`, a negative one on `age29-49` and a positive one on `age49+`.

2.3.6 Interaction Terms

We have already mentioned the possibility of including interaction terms in models. When you do you include them? The usual answer is domain knowledge. Sometimes interaction terms fall naturally out of the problem such as when you have height, width, and length. The interaction of these terms is volume.

Otherwise, you can identify variables with "large" effects and then include interaction terms. Although domain theory may indicate that terms with smaller effects interact with each other, it's unlikely that you'll be able to model it. An interaction effect can't be stronger than the main effect. Although it *might* make sense to model only the interaction effect.

Categorical variables are also a source of interaction terms but you need to be aware of the underlying support for the interaction. If you don't have enough observations of that particular–more specific–case, the estimate of the coefficient will probably not be any "good".

2.3.7 Linear Models of Non-Linear Data

Linear models also imply a additivity/linearity assumption has been violated. In these cases, we can often perform a transformation that results in data that satisfies the assumptions. Taking logarithms generally permit you to model non-linear relationships.

Consider the following:

$log(y) = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n$

if you exponentiate this equation you get:

$y = e^{\beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n}$

$y = B_0 + B_1^{X_1} + ...B_n^{X_n}$

where $B_0 = e^{\beta_0}$. In this case, each B_i can be interpreted (if you use natural log) as the percent difference in $log(y)$ between groups that differ by one unit of x_i.

If you start the other way:

$y = \beta_1^{x_1} \times \cdots \times \beta_n^{x_n}$

then taking the log of both sides yields:

$log(y) = log(x_0) + \beta_1 log(x_1) + \cdots + \beta_n log(x_n)$.

In this case, the various β_i are interpretable as *elasticities* or the percent change in y that results from a 1% change in x_i.

2.3.8 Building Linear Models

Where does this all leave us? What are the steps to building a linear model?

1. Start with all features as suggested by domain knowledge, but…
2. Analyze features for correlations and, of those groups, consider picking the best variable, an average of the variables, or some other transformation (sum, min, max, etc).
3. Transform all categorical variables into one hot encodings but leave one encoding out of the model for each variable. The intercept β_0 represent all the outcomes that are excluded explicitly. Which one you leave out might depend on the number of observations for each and what you want to do with the model.
4. Examine the residuals and EDA of the features and refer back to domain knowledge to see if any transformations are warranted including converting numerical variables into discrete, categorical variables.

2.3.9 Step-wise Regression and Data Dredging

As we already mentioned, finding the best subset of features out of a set of variables is NP-hard. Nevertheless, there are heuristics for so-called "Step-wise Regression". To a certain extent, we have been describing a heuristic driven form of manual Step-wise regression that starts with a "all variables in" initial model and then seeks to improve the model by

Step-wise regression is generally frowned up the statistical *cognoscenti* and is associated with "data dredging" and "p-value hacking". The concerns are these:

1. You have collected a bunch of data from an experiment. Your *theory* should tell you the specification for your model. If that model doesn't work, you shouldn't try to tweak it, looking for *some* relationship that fits your theory.

2. As you iterate over models, you are constantly doing statistical significance testing. It's not clear what any of the statistical significance tests for your final model really mean.

We've dodged the last point by using Bayesian inference. We don't have a "multiple comparisons" problem. The evidence supports what the evidence supports to the degree it supports it, based on your prior. As long as you don't go hog wild and use domain knowledge as a guide, you should be alright within the standards of commerce if not science.

And that's really germane to the first point. This is data science but it's not really science *per se*. We're not trying to build and validate a large body of knowledge. We want to know what our customer life time value is. This goes to the Blomberg and Gelman discussion.

It's up to you to not mislead yourself or organization by making tenuous connections in your data, leading to the loss of revenue at best and lives at worst. It behooves all of us to use the tools wisely or someone will take them away from us.

2.3.10 Assumptions of the Linear Model

We've talked about the assumptions of the linear model but let's bring them all into one place. Classical linear regression has a number of assumptions. We follow Gelman's advice and list them in their order of importance:

1. Validity
2. Additivity and Linearity
3. Independence of errors
4. Equal variance of errors
5. Normality of errors

Validity is the most important factor. Reasonably, it is the most important factor in any model building. Since it is possible to find spurious correlations between any number of variables so each variable should have some reason for being included in the model.

Validity is slightly broader than this, however. It also means that if we want to apply our model to underprivileged youths, the original model should not be constructed from data for all youths. Similarly, if we want to apply a model of health and fitness to the general population, it should probably not be built upon data from patients at risk for heart disease.

Additivity is important because, well, we are using an additive model. The same is true of *linearity*. But we can apply transformations to get us back into compliance.

Independence of errors assumes that errors in prediction are not correlated. We have mentioned this before, what we are saying is that our data should be either independently and identically distributed or, a slightly weaker condition, they are exchangeable. It's easy to think how this might be violated in our child IQ problem: what if more than one child from the same mother is included?

Equal variance of errors (residuals) is the assumption that the variance, $\epsilon_i \sim N(0, \sigma)$, is the same for all i. The technical term for this is *homoskedacity*. The opposite (and undesirable) condition is *heteroskedacity*. We saw this in the Residuals section. Gelman dismisses this as not a very big problem because it does not affect the estimation of β.

Normality of errors (residuals) is something that people often wring their hands about and Gelman says this is the least important. While it might affect our ability to predict (and here it might actually be important to someone), it doesn't affect our ability to form a model, to discover the effects that are β.

What does a failure of normality mean? It means ability to calculate error bounds is compromised but we have the Bootstrap to estimate credible intervals for any estimate we want to make.

```
[1]: import warnings
     warnings.filterwarnings('ignore')
```

[2]:
```
%matplotlib inline
```

[3]:
```
import numpy as np
import scipy.stats as stats
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import random
import patsy
import sklearn.linear_model as linear

sns.set(style="whitegrid")

import sys
sys.path.append('resources')
import models
```

2.4 Example

This extended example looks at Mesquite yields of leaves. It is basically a problem in biology/horticulture/agriculture.

[4]:
```
mesquite = pd.read_csv( "resources/mesquite.tsv", sep="\t")
```

We don't have *specific* domain knowledge about this problem but we do have *general* domain knowledge. We'll see how that comes into play later in the example. For now, we know that we *should* have done EDA on this data long before we got to this point. In the interests of brevity, we will simply look at the contents of the data and the first rows:

[5]:
```
mesquite.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 46 entries, 0 to 45
Data columns (total 8 columns):
obs                46 non-null int64
group              46 non-null object
diam1              46 non-null float64
diam2              46 non-null float64
total_height       46 non-null float64
canopy_height      46 non-null float64
density            46 non-null int64
weight             46 non-null float64
dtypes: float64(5), int64(2), object(1)
memory usage: 3.0+ KB
```

This is not a lot of data, really. There are only 46 observations.

[6]:
```
mesquite.head()
```

[6]:
```
   obs group  diam1  diam2  total_height  canopy_height  density  weight
0    1   MCD    1.8   1.15          1.30           1.00        1   401.3
1    2   MCD    1.7   1.35          1.35           1.33        1   513.7
2    3   MCD    2.8   2.55          2.16           0.60        1  1179.2
3    4   MCD    1.3   0.85          1.80           1.20        1   308.0
4    5   MCD    3.3   1.90          1.55           1.05        1   855.2
```

We have several variables. The observation number (which is just an index and not a feature), the group, "diam1" and "diam2" (ostensibly to avoid trying to figure out which one was width and which was length), total_height of the tree, canopy_height, density, and the weight of the leaves. The weight is our target variable. Density is interesting. Let's see a bit more about it:

```
[7]: mesquite.density.describe()
```

```
[7]: count    46.000000
     mean      1.673913
     std       1.592403
     min       1.000000
     25%       1.000000
     50%       1.000000
     75%       2.000000
     max       9.000000
     Name: density, dtype: float64
```

The most common value is 1 (by quite a long shot). It might be better to look at this as a discrete variable with a limited domain:

```
[8]: mesquite.density.value_counts(normalize=True)
```

```
[8]: 1    0.717391
     2    0.152174
     3    0.065217
     9    0.021739
     7    0.021739
     5    0.021739
     Name: density, dtype: float64
```

71.7% of the observations of a density of 1 followed by 15.2% with a density of 2.

Let's look at group as well:

```
[9]: mesquite.group.value_counts(normalize=True)
```

```
[9]: MCD    0.565217
     ALS    0.434783
     Name: group, dtype: float64
```

One of the libraries automatically converts the categorical variable into an encoding but we won't know which one is which so we should do it ourselves. Following our advice, we name the variable "mcd" for group=mcd or mcd=1 in which case als will be 0.

```
[10]: mesquite["mcd"] = mesquite.group.apply(lambda x: 1 if x == 'MCD' else 0)
      mesquite.mcd.value_counts(normalize=True)
```

```
[10]: 1    0.565217
      0    0.434783
      Name: mcd, dtype: float64
```

It's always good to make sure your transformations did what you expected. In any case, the two groups are almost exactly 50/50.

What do we think about the variables and their contributions to weight (of leaves)?

- **mcd** - we don't know what the groups are so it could be positive or negative.
- **diam1** - positive: more diameter is more leaves.
- **diam2** - positive: more diameter is more leaves.
- **total_height** - ??: a tall tree may just be tall with the canopy just higher off the ground.
- **canopy_height** - positive: more canopy, more leaves.

- **density** - positive: more leaves per unit, more leaves.

Let's fit a "all variables in" model:

[11]:
```
model =  "weight ~ mcd + diam1 + diam2 + total_height + canopy_height + density"
result = models.bootstrap_linear_regression(model, data=mesquite)
print(models.describe_bootstrap_lr(result))
```

```
Model: weight ~ mcd + diam1 + diam2 + total_height + canopy_height + density
---------  -------  -----  -----  -----
                                    95% BCI
Coefficients                Mean      Lo        Hi
               $\beta_0$  -1091.89  -1614.68  -524.47
mcd            $\beta_1$  363.30    175.13    597.17
diam1          $\beta_2$  189.67    9.49      382.64
diam2          $\beta_3$  371.46    137.39    546.60
total_height   $\beta_4$  -101.73   -706.76   220.11
canopy_height  $\beta_5$  355.67    -61.22    1314.72
density        $\beta_6$  131.25    -18.98    300.57

Metrics        Mean       Lo        Hi
$\sigma$       268.96     93.37     307.93
$R^2$          0.85       0.78      0.95
---------  -------  -----  -----  -----
```

The R^2 is 85% which makes this a fairly good model in terms of explanation. We explain 85% of the variability in weight. We don't know the range of weight so we don't know if the σ is any good. We would have normally found this out through EDA but let's look now:

[12]:
```
mesquite.weight.describe()
```

[12]:
```
count      46.00000
mean      559.66087
std       642.83481
min        60.20000
25%       219.62500
50%       361.85000
75%       688.72500
max      4052.00000
Name: weight, dtype: float64
```

We cut the error of our estimates from 642.8 to 268.96 with our model which is a pretty good increase in accuracy. Still, based on the actual application, it might not be enough. We might be interested in a good prediction of leaf weight or we might be interested in better explanatory for the model.

The coefficients are all over the place. Only diam2 is clearly positive. For the others, we have:

[13]:
```
predictions = {"mcd": "+", "diam1": "+", "diam2": "+", "total_height": "+", "canopy_
↪height": "+", "density": "+"}
models.evaluate_coefficient_predictions(predictions, result)
```

```
mcd P(>0)=1.000 (strong)
diam1 P(>0)=0.990 (strong)
diam2 P(>0)=0.990 (strong)
total_height P(>0)=0.370 (mixed)
canopy_height P(>0)=0.920 (strong)
density P(>0)=0.950 (strong)
```

First, all of the coefficients have the expected signs except total_height. We didn't even know what to expect with total_height. Second, many of the coefficients pass the bounds test except density. However, if we look at the posterior probabilities for all the coefficients, the evidence is strongly in favor of the observed signs. The only mixed result is for total_height.

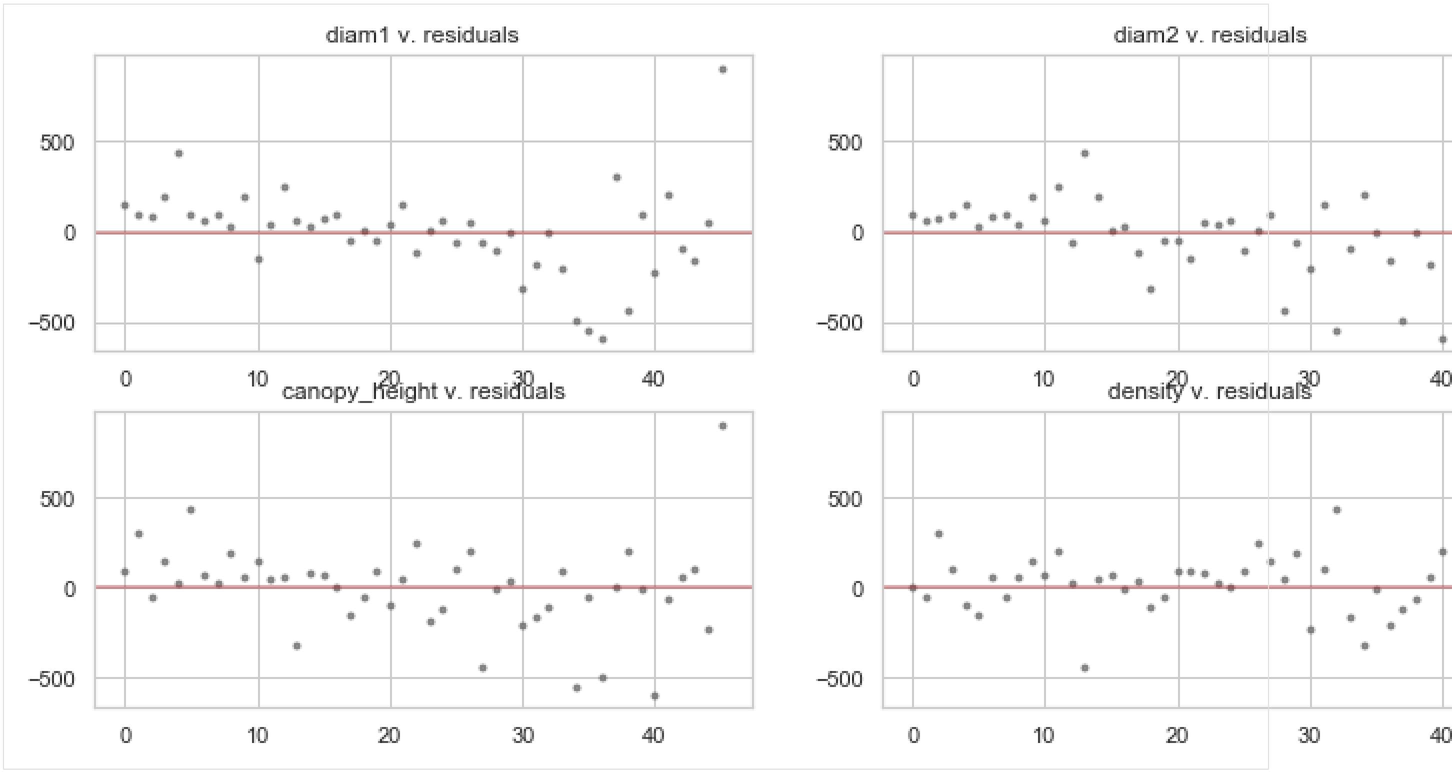
Still, it seems like we have ample chances for problems here including multicollinearity. Trees don't just grow wide, staying 1 foot think (at least not usually). And total_height and canopy_height are surely correlated. Let's check a few of these:

```
[14]: print("diam1 v diam2 = {0:.2f}".format(stats.pearsonr(mesquite.diam1, mesquite.
      ↪diam2)[0]))
      print("total_height v canopy_height = {0:.2f}".format(stats.pearsonr(mesquite.total_
      ↪height, mesquite.canopy_height)[0]))
```

```
diam1 v diam2 = 0.89
total_height v canopy_height = 0.84
```

Lot's of correlation on both accounts. We know that that will mess with our β_i estimates. What about the residuals?

```
[15]: figure = plt.figure(figsize=(20,6))

      variables = ["diam1", "diam2", "total_height", "canopy_height", "density"]

      plots = len( variables)
      rows = (plots // 3) + 1

      for i, variable in enumerate( variables):
          axes = figure.add_subplot(rows, 3, i + 1)

          keyed_values = sorted( zip( mesquite[ variable].values, result[ "residuals"]),␣
      ↪key=lambda x: x[ 0])
          residuals = [x[ 1][ 0] for x in keyed_values]

          axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.
      ↪75)
          axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="firebrick", alpha=0.5)
          axes.set_title( variable + " v. residuals")

      plt.show()
      plt.close()
```



We have two problems evident in all the residual plots. First, there's a pattern to the residuals. At low values, the errors are smaller and as the variable increases, the errors get larger. Second, we seem to be underestimating at low values for all the variables ($y - \hat{y} > 0$) and overestimating at high values ($y - \hat{y} < 0$).

This is where our *general* domain knowledge comes in. Is there really an *additive* relationship between diam(s), height, and weight? Aren't we really talking about *volume*? And if are talking about volume, isn't that a multiplicative model?

$y = diam1 \times diam2 \times canopy\_height$

We know from the transformation section that if we have a multiplicative model, we can transform it into an additive model by taking logs of all the variables:

[16]:
```
def log( x):
    return np.log(float( x))

for column in ["diam1", "diam2", "total_height", "canopy_height", "weight", "density
↪"]:
    mesquite[ "log_" + column] = mesquite[ column].apply( log)
```

[17]:
```
mesquite[["log_diam1", "log_diam2", "log_total_height", "log_canopy_height", "log_
↪weight", "log_density"]].describe()
```

[17]:
```
       log_diam1  log_diam2  log_total_height  log_canopy_height  log_weight  \
count  46.000000  46.000000         46.000000          46.000000   46.000000
mean    0.664373   0.325520          0.340767           0.048187    5.919543
std     0.393820   0.523814          0.337371           0.344195    0.913676
min    -0.223144  -0.916291         -0.430783          -0.693147    4.097672
25%     0.336472   0.000000          0.182322          -0.148229    5.391783
50%     0.667501   0.421860          0.405465           0.095310    5.890144
75%     0.906085   0.641854          0.530628           0.262364    6.534769
```

(continues on next page)

(continued from previous page)

```
max     1.648659   1.386294          1.098612            0.916291     8.306966

       log_density
count    46.000000
mean      0.302183
std       0.557538
min       0.000000
25%       0.000000
50%       0.000000
75%       0.693147
max       2.197225
```

Before estimating a new model, let's check the *adjusted* R^2:

[18]:
```
models.adjusted_r_squared(result)
```

[18]:
```
0.8154810264106127
```

Let's estimate the new model using just these new log() variables:

[19]:
```
model =  "log_weight ~ log_diam1 + log_diam2 + log_total_height + log_canopy_height +␣
↪log_density"
result = models.bootstrap_linear_regression(model, data=mesquite)
print(models.describe_bootstrap_lr(result))
```

```
Model: log_weight ~ log_diam1 + log_diam2 + log_total_height + log_canopy_height +␣
↪log_density
-----------  ------  ---  -----  ---
                                    95% BCI
Coefficients                 Mean   Lo      Hi
                   $\beta_0$ 4.97   4.59    5.33
log_diam1          $\beta_1$ 0.73   0.11    1.26
log_diam2          $\beta_2$ 0.79   0.28    1.22
log_total_height   $\beta_3$ 0.70   -0.18   1.39
log_canopy_height  $\beta_4$ 0.01   -0.65   0.81
log_density        $\beta_5$ -0.11  -0.45   0.06

Metrics            Mean       Lo     Hi
$\sigma$           0.40       0.29   0.46
$R^2$              0.83       0.73   0.91
-----------  ------  ---  -----  ---
```

How do we interpret the β_is in such an model? They're *elasticities*: a 1% increase in x_i leads to a β_i% increase in y.

The R^2 has dipped and the intervals have gotten sketchier. What about $\bar{R}^2$?

[20]:
```
models.adjusted_r_squared(result)
```

[20]:
```
0.7961538611828038
```

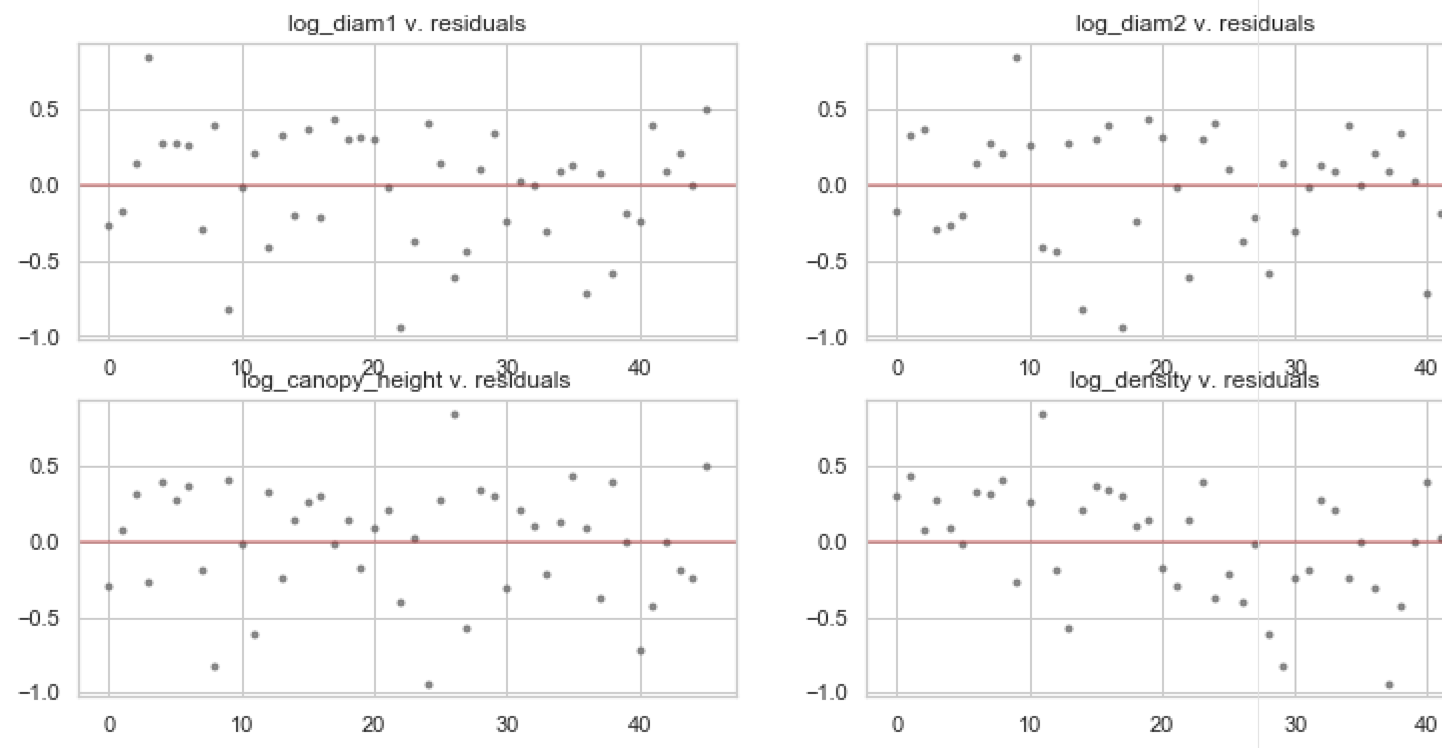
The adjusted R^2 fell as well. And what about our residuals?

[21]:
```
figure = plt.figure(figsize=(20,6))

variables = ["log_diam1", "log_diam2", "log_total_height", "log_canopy_height", "log_
↪density"]

plots = len( variables)
```

(continues on next page)

(continued from previous page)

```
rows = (plots // 3) + 1

for i, variable in enumerate( variables):
    axes = figure.add_subplot(rows, 3, i + 1)

    keyed_values = sorted( zip( mesquite[ variable].values, result[ "residuals"]),␣
↪key=lambda x: x[ 0])
    residuals = [x[ 1][ 0] for x in keyed_values]

    axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.
↪75)
    axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="firebrick", alpha=0.5)
    axes.set_title( variable + " v. residuals")

plt.show()
plt.close()
```



Most of the residuals are fixed; density still looks a little heteroskedastic.

Perhaps there's still just too much multicollinearity. Volume is the product of all the dimensions and for an arbitrary three dimensional shape, one of those might vary significantly but probably not for trees. Maybe we should just use a *volume* feature. In "log space", log_volume is the *sum* of the three dimensions:

```
[22]: mesquite[ "log_canopy_volume"] = mesquite[ "log_diam1"] + mesquite[ "log_diam2"] +␣
↪mesquite[ "log_canopy_height"]
```

```
[23]: model =  "log_weight ~ log_canopy_volume"
result = models.bootstrap_linear_regression(model, data=mesquite)
print(models.describe_bootstrap_lr(result))
```

```
Model: log_weight ~ log_canopy_volume
-----------  ------  ---  -----  ---
                                95% BCI
Coefficients                Mean  Lo       Hi
                   $\beta_0$  5.17  5.01     5.30
log_canopy_volume  $\beta_1$  0.72  0.64     0.81

Metrics            Mean       Lo    Hi
$\sigma$           0.41       0.33  0.49
$R^2$              0.80       0.66  0.88
-----------  ------  ---  -----  ---
```

Almost all of the variation in weight comes from volume. Still, our R^2 isn't as high as it was (which may be because of the number of variables used in the different models). What about adjusted R^2?

```
[24]: models.adjusted_r_squared(result)
```

```
[24]: 0.7848634245562538
```

It's lower than what we've seen but this is just *one* variable in the model. Are we missing anything else? There are two possible additional features:

$area = diam1 \times diam2$

$shape = diam1 \div diam2$

With our log representations, these are addition and subtraction:

```
[25]: mesquite[ "log_canopy_area"] = mesquite[ "log_diam1"] + mesquite[ "log_diam2"]
      mesquite[ "log_canopy_shape"] = mesquite[ "log_diam1"] - mesquite[ "log_diam2"]
```

We expect both of these to be positively related to weight:

```
[26]: model =  "log_weight ~ log_canopy_volume + log_canopy_area + log_canopy_shape"
      result = models.bootstrap_linear_regression(model, data=mesquite)
      print(models.describe_bootstrap_lr(result))
```

```
Model: log_weight ~ log_canopy_volume + log_canopy_area + log_canopy_shape
-----------  ------  ---  -----  ---
                                  95% BCI
Coefficients                Mean   Lo       Hi
                   $\beta_0$  5.12   4.83     5.49
log_canopy_volume  $\beta_1$  0.40   -0.04    1.00
log_canopy_area    $\beta_2$  0.42   -0.36    1.00
log_canopy_shape   $\beta_3$  -0.08  -0.55    0.34

Metrics            Mean       Lo    Hi
$\sigma$           0.41       0.31  0.49
$R^2$              0.81       0.68  0.90
-----------  ------  ---  -----  ---
```

```
[27]: models.adjusted_r_squared(result)
```

```
[27]: 0.7865195478356122
```

Still not as good as the original model,

```
[28]: predictions = {"log_canopy_volume": "+", "log_canopy_area": "+", "log_canopy_shape":
       ↪"+"}
      models.evaluate_coefficient_predictions(predictions, result)
```

```
log_canopy_volume P(>0)=0.960 (strong)
log_canopy_area P(>0)=0.790 (strong)
log_canopy_shape P(>0)=0.370 (mixed)
```

and the evidence for our new features is mixed in one case but strong in the others. However, the model makes *sense* so let's add in the other features:

[29]:
```
model =  "log_weight ~ log_canopy_volume + log_canopy_area + log_canopy_shape + mcd +␣
↪log_total_height + log_density"
result = models.bootstrap_linear_regression(model, data=mesquite)
print(models.describe_bootstrap_lr(result))
```

```
Model: log_weight ~ log_canopy_volume + log_canopy_area + log_canopy_shape + mcd +␣
↪log_total_height + log_density
-----------  ------  ---  -----  ---
                                     95% BCI
Coefficients                   Mean  Lo        Hi
                   $\beta_0$  4.77  4.38      5.08
log_canopy_volume  $\beta_1$  0.37  0.15      1.41
log_canopy_area    $\beta_2$  0.40  -0.68     0.61
log_canopy_shape   $\beta_3$  -0.38 -0.89     -0.05
mcd                $\beta_4$  0.58  0.46      0.87
log_total_height   $\beta_5$  0.39  -0.64     0.85
log_density        $\beta_6$  0.11  -0.09     0.29

Metrics            Mean       Lo     Hi
$\sigma$           0.33       0.21   0.35
$R^2$              0.89       0.85   0.96
-----------  ------  ---  -----  ---
```

[30]:
```
models.adjusted_r_squared(result)
```

[30]:
```
0.8629155301645439
```

This is our best result so far judging by the adjusted R^2. The signs on almost all of the coefficients are what we expect. What about the evidence?

[31]:
```
predictions = {"log_canopy_volume": "+", "log_canopy_area": "+", "log_canopy_shape":
↪"+", "mcd": "+",
               "log_total_height": "+", "log_density": "+"}
models.evaluate_coefficient_predictions(predictions, result)
```

```
log_canopy_volume P(>0)=0.990 (strong)
log_canopy_area P(>0)=0.700 (strong)
log_canopy_shape P(>0)=0.020 (weak)
mcd P(>0)=1.000 (strong)
log_total_height P(>0)=0.750 (strong)
log_density P(>0)=0.900 (strong)
```

The evidence for all the coefficients is strong except log_canopy_shape. However, the evidence is strong for log_canopy_shape being negative *is* strong so we will need to revisit our theory about what this is measuring.

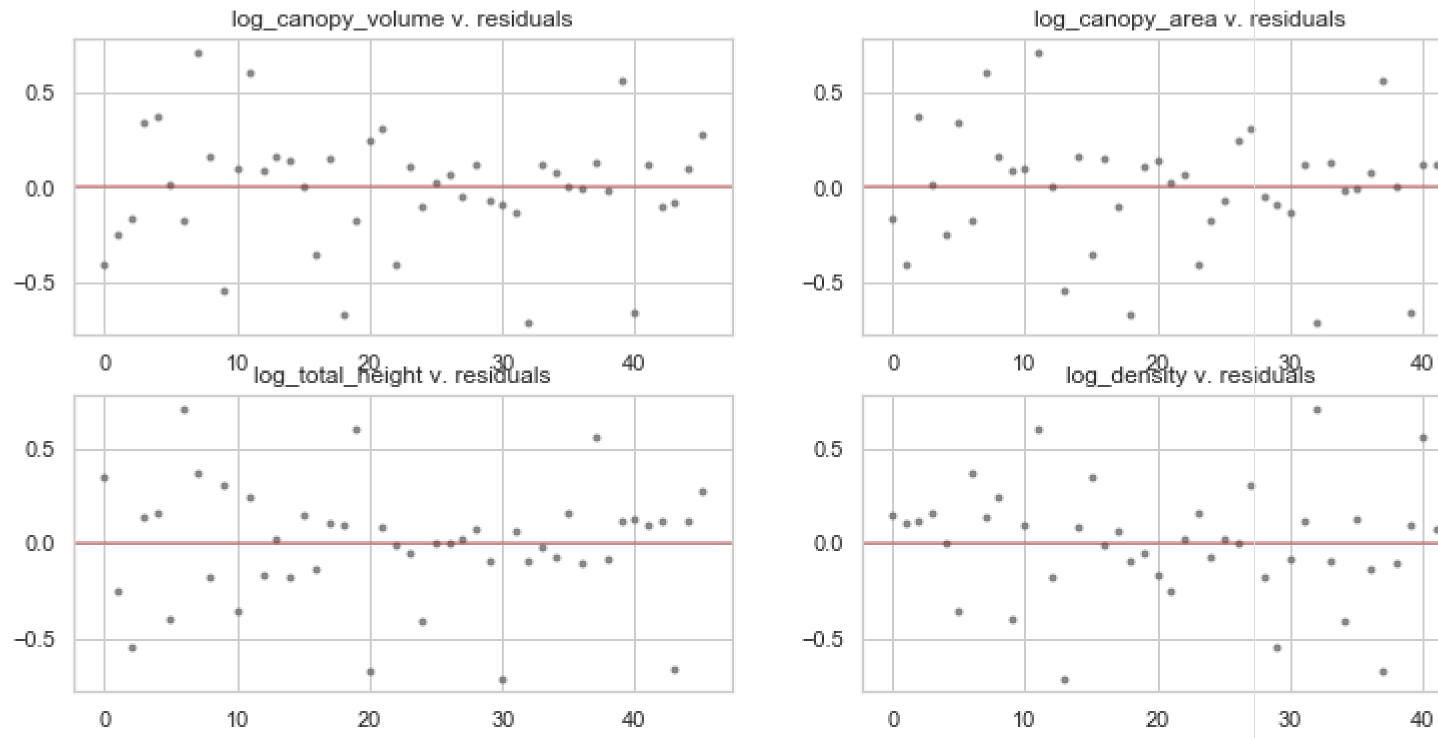
Finally, we need to look at the residuals:

[32]:
```
figure = plt.figure(figsize=(20,6))

variables = ["log_canopy_volume", "log_canopy_area", "log_canopy_shape", "log_total_
↪height", "log_density"]
```

(continues on next page)

(continued from previous page)

```
plots = len( variables)
rows = (plots // 3) + 1

for i, variable in enumerate( variables):
    axes = figure.add_subplot(rows, 3, i + 1)

    keyed_values = sorted( zip( mesquite[ variable].values, result[ "residuals"]),␣
↪key=lambda x: x[ 0])
    residuals = [x[ 1][ 0] for x in keyed_values]

    axes.plot(list(range(0, result[ "n"])), residuals, '.', color="dimgray", alpha=0.
↪75)
    axes.axhline(y=0.0, xmin=0, xmax=result[ "n"], c="firebrick", alpha=0.5)
    axes.set_title( variable + " v. residuals")

plt.show()
plt.close()
```



Our residuals seem to be pretty good.

When you are building a regression based on domain knowledge for the purpose of explanation (you are interested in the coefficient values) or prediction (you're just interested in the σ and R^2), this is the basic approach you take. If this is the ultimate goal, this will inform your EDA a bit. You will start to look for broad patterns in variables:

1. Is this variable approximately Normally distributed?
2. If this variable doesn't look Normally distributed, are there transformations I can apply to make it normally distributed?
3. Is there domain knowledge that suggests there might be increasing or decreasing returns from this variable? Should I then look at squared or square-root transformations of the variable?

2.5 Conclusion

In this chapter we have extended the basic concepts of linear models to the multivariate case. As a result, we have talked about model building. The basic guidelines to model building are:

1. Start with all features as suggested by domain knowledge, but…
2. Analyze features for correlations and, of those groups, consider picking the best variable, an average of the variables, or some other transformation (sum, min, max, etc).
3. Transform all categorical variables into one hot encodings but leave one encoding out of the model for each variable. The intercept β_0 represent all the outcomes that are excluded explicitly. Which one you leave out might depend on the number of observations for each and what you want to do with the model.
4. Examine the residuals and EDA of the features and refer back to domain knowledge to see if any transformations are warranted including converting numerical variables into discrete, categorical variables.

Our rule of thumb for examining coefficients and their credible bounds is:

Case	Sign	Credible Interval	Action
1	expected	does not include 0	Keep
2	expected	does include 0	Keep
3	unexpected	does not include 0	Re-examine
4	unexpected	does include 0	Remove

which presumes that we think about our problem domain before running the models.

There are many, many variations for linear models and we'll talk about a few more variations in the next chapter.

2.5.1 Review

1. What are our guidelines for building linear models?
2. Explain how to use a linear regression for estimation and the role of σ.
3. What is the Adjusted R^2 and why do we need it?
4. When examining a coefficient, what are the four (4) possibilities and what should you do in each case?
5. What is the difference between *scaling* and *mean centering*? How do they help with the interpretation of coefficients? What are the pros and cons of each? Why do we save the parameters?
6. Why might we use a baseline to center our data instead of the mean?
7. How do we handle categorical variables of more than two possibilities? Why can't we just use numeric codes?
8. What are residuals? What is a residual plot?
9. Explain how increasing and decreasing returns might suggest a variable transformation. What kinds of transformation? How might we spot the necessity of such a transformation in a residual plot?
10. Why might the discretization of a numerical variable be a relevant variable transformation?
11. Why might we use log transformations to improve a linear regression? What assumption about linear regression would be violated and why does a log transformation solve it? How do we interpret the coefficients?
12. What are the assumptions of classical linear regression?
13. If the errors are not constant, what is this called?
14. How does the Two Leg Example illustrate the problem with multicollinearity?

CHAPTER

THREE

MODEL EVALUATION

We've discussed two models: Linear Regression and Logistic Regression. Both of these models have a long history in statistics and when talking about them we emphasized mostly the *explanatory* use of those models: interpreting the *importance* of coefficients and trying to determine if the values were *credible* based on the evidence (data). We talked a little bit about the *predictive* function of those models when we looked at linear regression and talked about the σ and R^2 of the regressions. But the point of view was mostly the statistician's point of view.

In this chapter, we're going to focus more broadly on model evaluation and improvement. This encompasses two general and important questions in model development: how good is my model and can I make it better? Note that when we talked about the *explanatory* function of linear regression, we never asked ourself, "could getting more data make the model better?". Now we're going to start to address these kinds of questions.

We'll start with model evaluation from the machine learning point of view which focuses on cross-validation. Cross-validation is a form of *backtesting* that allows you to get a general idea of the performance of your model on new data. It focuses on prediction as the main goal of a model (although it need not exclusively be so).

When we're done with cross-validation, we may have a model with an estimated 9% error rate. We can then ask ourselves, can we make it better? The next section will focus on this question. If we have a model with a certain performance, what should we focus on? The answer is not always, "get more data".

Finally, we'll start to move into a discussion of live testing or A/B testing and the problems that arise with that process both in terms of experimental design and engineering infrastructure.

Obviously, some of these topics bleed back and forth into each other and the process is very iterative. For example, in linear regression we talked about adding key variables, evaluating the model, adding interaction terms, evaluating the model, and so on. Cross validation can certainly fit into this process as it is closely related to bootstrapping.

3.1 Two Cultures

Although previously we looked at the problem of model evaluation from the statistician's point of view, we're now going to look at the problem from the machine learning researcher's ("machine learning") point of view. As Data Scientists, we must be familiar with both and know when each is appropriate, even using one when a "true believer" might use another. One of the larger failings of current Data Science is to exclude the statistical view entirely.

I don't want to overstate the differences, however. Many of the techniques from Machine Learning have found their way into the Statistician's toolbox. The lines have definitely started to blur in recent years. However, they were remarkably different for a long time and as late as 2014, Hal Varian talked about what Econometrics and Machine Learning could learn from each other. These points of view don't change overnight. This goes back to the exchange we have already quoted between Bloomberg and Gelman. Although Gelman is a statistician and his views often straddle the worlds, we're going to move full on into the Machine Learning approach starting with this chapter.

Take the case of linear regression. Whereas a statistician might ask:

1. What is the R^2 (or null deviance or whatever)? What is σ?

2. Are the coefficients statistically significant (or credible)?
3. Do we need to worry about multicollinearity? heteroskedascity?
4. What do the residuals look like?
5. Does the domain offer any suggestions about additional or missing variables, transformations, interaction terms?

A machine learning engineer will ask:

1. What is the likely error of my model when I apply it to *new* data (generalization performance/error)?
2. How can I make my model better? Where "better" might mean an improvement in some metric and the "how" might mean:
    1. Get more observations (this doesn't always work).
    2. Get more variables (you might actually need fewer variables) or variable transformations.
    3. Apply regularization (or remove it).
    4. Set a different value for the model's *metaparameters* (parameters of the model not set by using data).
    5. Switch to a different model.
3. Once my model is deployed, how can I evaluate it "in the wild"?

Although we won't leave the Statistician's view entirely behind, it is with this latter set of questions that we will concern ourselves in this chapter. Let's start at the start.

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%matplotlib inline
```

[3]:
```
import numpy as np
import scipy.stats as stats
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import random
import patsy

sns.set(style="whitegrid")
```

[4]:
```
import sys
sys.path.append('resources')
import models
```

3.2 Loss and Evaluation Metrics

If we are interested solely in prediction, what does it mean for a model to be good or bad? We talked about this before in terms of the general linearized model but the question–and possible answers–is more general.

For example, we could simply predict y by using some constant value of y, $\breve{y}$. What $\breve{y}$ should we use? This depends on how much it costs us to be wrong. What "wrong" means depends on the value we're trying to predict or, more generally, if we have a regression problem (predicting a numeric value) or a classification problem (predicting a categorical value).

3.2.1 Regression

We talked about loss and regression before. If we define loss (or error) as $y - \hat{y}$ and we wish to summarize this loss over some training set, then we have the following "usual suspects":

- **squared loss**

$L(y, \hat{y}) = (y - \hat{y})^2$

- **absolute loss**

$L(y, \hat{y}) = |y - \hat{y}|$

- **0/1 loss**

$L(y, \hat{y}) = 1.0$ if $y = \hat{y}$ else 0.0

All of these can be turned into a normalized summary statistic by taking the sum and averaging:

$\frac{1}{n} \sum L(y, \hat{y})$

But these aren't the only possibilities. You can use other metrics such as σ or R^2. Additionally, you can calculate your own Loss function. For example, imagine that you want to penalize over-estimation *more* than underestimation:

$$L(y, \hat{y}) = \begin{cases} (y - \hat{y})^2, & \hat{y} > y \\ |y - \hat{y}|, & \text{otherwise} \end{cases}$$

This last approach can be a bit tricky. The algorithm to calculate the coefficients of the typical linear regression finds coefficients that minimize *mean squared error* or MSE. If you have your own metric you wish to minimize, you may need to implement your own algorithm using, for example, stochastic gradient techniques.

Most of the time when talking about regression problems (and not just linear regression) we will be using mean squared error, σ, or R^2 as our evaluation metrics. However, I don't want you to get the impression that these are the only possibilities.

3.2.2 Classification

Like regression, classification has a baseline for predicting the class (or class *label*). We talked about this Null model as the relative frequency of the most common class. Technically, this would be p if $p > 50\%$ or $1 - p = q$ if $q > 50\%$. For example, suppose that for a binary problem, the relative frequency of the most common class is 87.3%. Then this is the Null model for classification. If you guess "1" for any observation you see then you are right 87.3% of the time and wrong 12.7% of the time. This is the Null model for binary classification.

For a multi-class problem, we pick the p_i with the highest value. Although we called this the Null model in the previous chapters, in machine learning this is known as *OneR*. We estimate OneR simply by calculating the relative frequencies of the class labels and picking the label with the highest relative frequency.

It is possible to formulate a variety of loss functions for the classification task. Taking the *binary* case, *cross entropy* is one such function and the one used to find the βs in logistic regression.

$L(y, \hat{y}) = ylog(\hat{y}) + (1 - y)log(1 - \hat{y})$

and there are others. However, I never see these functions used to *evaluate* classification models. They are normally used by the *algorithms*, $g(X, y)$, to learn the classification models, $f(X)$, from data.

Instead, there are a variety of classification *metrics* that are used to evaluate how good or bad a classification model is. The reason for this plethora of metrics is that there appear to be a number of ways that a classification prediction can go right or wrong and different ways to summarize these outcomes. For a binary classification task, where "1" is taken to mean "positive" or "in the class" and "0" is taken to be "negative" or "not in the class", the possible cases for a classification model are:

1. The true class can be "1" and the model can predict "1". This is a *true positive* or TP.
2. The true class can be "1" and the model can predict "0". This is a *false negative* or FN.
3. The true class can be "0" and the model can predict "1". This is a *false positive* or FP.
4. The true class can be "0" and the model can predict "0". This is a *true negative* or TN.

We can summarize these results in a table called a *confusion matrix*… it summarizes how confused (or not) our model is:

Actual 1	TP	FN
Actual 0	FP	TN

Since N is simply the number of observations TP + FP + FN + TN, then we have the following:

accuracy = $\frac{TP+TN}{N}$ = 1 - error rate

error rate = $\frac{FP+FN}{N}$ = 1 - accuracy

which are the two most common metrics for evaluating classification models. We used this in previous chapters to evaluate logistic regression.

However, they may not be sufficient. As we see above, there are two types of errors: classifying something that's "1" as "0" and classifying something that's "0" as "1". The impact of these errors may not be symmetric or equally costly. We saw this when talking about Bayes Theorem. A test may be good at telling you there's a problem if there really is a problem but they also tell you when there's a problem when there really *isn't* a problem, i.e., false alarms.

These various possibilities each have their own names (sometimes several) and can be discussed in terms of the confusion matrix. We're going to look at the three more common ones.

The first is **sensitivity**. This is basically the "disease" case: if you have the disease, how good is the model (test) at detecting it? It is also called **true positive rate**, **hit rate**, and **recall**. It is defined only in terms of the positive observations, both those the model predicted correctly and those it did not.

$sensitivity = \frac{TP}{TP+FN}$ (1 - sensitivity is the **false negative rate**)

The next one is **specificity**. This is the other case. If you do not have the disease, how good is the model (test) at determining that (and not telling you that you do have it!)?

$specificity = \frac{TN}{TN+FP}$ (1 - specificity is the **false positive rate**)

The final case is **precision** or **positive predictive value**. Basically, of the positive ("1") predictions that we made, how many were right?

$precision = \frac{TP}{TP+FP}$

When we're not looking at accuracy/error rate, we generally look at sensitivity and precision. In fact, there's a harmonic mean of the two called $F1$:

$F1 = \frac{2TP}{2TP+FP+FN}$

3.2.3 Example

Let's look at our "switching" logistic regression and evaluate it in terms of these new metrics. Let's load the data:

```
[5]: wells = pd.read_csv( "resources/arsenic.wells.tsv", sep=" ")
```

```
[6]: wells[ "dist10"] = wells[ "dist"].apply( lambda x: int( round( x / 10, 0)))
```

One possible model might have been:

```
[7]: result1 = models.logistic_regression( "switch ~ dist10 + arsenic + assoc + educ",␣
     ↪data = wells)
     print(models.simple_describe_lgr(result1))
```

```
Model: switch ~ dist10 + arsenic + assoc + educ
---------  ------  ---
Coefficients               Value
               $\beta_0$   -0.16
dist10         $\beta_1$   -0.09
arsenic        $\beta_2$   0.47
assoc          $\beta_3$   -0.13
educ           $\beta_4$   0.04

Metrics        Value
Error ($\%$)   38.41
Efron's $R^2$  0.07
---------  ------  ---
```

The error rate here was calculated to be 38.41%. The `logistic regression` function has been enriched to return both the y and $\hat{y}$ values to us. Let's use them to generate a confusion matrix:

```
[8]: from tabulate import tabulate
```

```
[9]: def binary_confusion_matrix(result):
         tp = 0; tn = 0; fn = 0; fp = 0
         for y, y_hat in zip(result["y"], result["y_hat"]):
             if y == 0 and y_hat == 0:
                 tn += 1
             elif y == 0 and y_hat == 1:
                 fp += 1
             elif y == 1 and y_hat == 1:
                 tp += 1
             else:
                 fn += 1
         return tabulate([["Predicted 1", tp, fp], ["Predicted 0", fn, tn]], headers=["",
     ↪"Actual 1", "Actual 0"])
```

```
[10]: print(binary_confusion_matrix(result1))
```

```
               Actual 1    Actual 0
-------  --------  -------
Predicted 1        1391         814
Predicted 0         346         469
```

Here we can see that our true positives were 1391. Our true negatives were 469. For our errors, our false positives were 814 and our false negatives were 346.

- Of all the actual negatives, how many did we predict to be positive?

This the *false positive rate*: FP/(TN + FP) = 814/(814+469) = 63.4%.

- Of all the actual positives, how many did we predict to be negative?

The false negatives were 346 and the *false negative rate* was FN/(TP + FN) = 346/(1391+346) = 19.9%.

By comparing these two rates, we can see that our model errs more often by predicting someone will switch when they didn't rather than predicting that someone will stay when they switched.

- Of all the positive predictions, how many were correct?

Finally, we can look at precision: TP/(TP+FP) = 1391/(1391 + 814) = 63.1%. Of our positive predictions (predicting switch), we were 63.1% correct.

As you can see, this gives a more nuanced view of the model than just *error rate*. The main problem here is the false positive rate. We are much more likely to predict a switch than is actually reflected in the data on people switching. For something like arsenic poisoning, this is probably more important than underestimating those who do switch (false negative rate). If this were a government program, you'd need a two pronged approach. A better model of understanding the behavior that leads to switching but monitoring of actual switches to know if the program is successful.

Confusion matrices work even for multi-class problems. Although the language isn't quite the same (you can't have true "positives" with three or more classes), there is still the sense of:

1. Of actual class i, how many did we predict to be something else?
2. Of predicted class i, how many were correct?

3.3 Cross Validation

We start with the fundamental assumption of machine learning:

> The data that you train your model on must come from the same distribution as the data you hope to apply the model to.

And there's is no way around this. One million observations of the wrong thing is not going to help you build a model for the thing you want. This is just the Machine Learning version of the idea that a sample must be representative. Assuming you have the right thing, though, how do you calculate a metric for it that gives you an idea of how it will work on unseen data?

To a certain extent, we have already been wrestling with this problem. We have a *sample* of data from a process but we don't know for sure all the values the process can or will ever generate ("population"). And while we need to assume that, in general, the process is stable for us to even think about modeling it, the problem of inference remains. We have so far looked at the problem of inference as, "given that the data is potentially consistent with a large number of models, which model(s) should I find the most credible?" This is the Bayesian perspective.

The Machine Learning perspective is, "how will the model most likely perform on previously unseen data?". What do we mean by unseen data? Do we mean that the value 7.3 never appeared in the training data? No. We generally mean, the model is not evaluated on the same data that was used to train it. As we noted previously, the data still has to come from the same distribution. Where can we get this unseen data? We've already noted that getting more data (additional samples) can be difficult, costly, or impossible.

One very simple way would be to split the data we have into a train(ing) set and a test set. We could train our model (say, linear regression) on the train set and evaluated it (calculate MSE) on the test set. We then get an estimate of MSE on unseen data for our model.

Unfortunately, that only gets use a single estimate and we'd like to get a sense of the overall variation in performance. We could switch the roles of train and test sets, train on the test set and evaluated on the train set and get another estimate of MSE but two isn't quite enough either. Enter *cross validation*.

Borrowing ideas from bootstrapping, there's another option. You could divide your data into ten sections or *folds* and then loop over each fold, using it as the test set and the others as the training set. So, for the first iteration, you have Fold 1 as the test set and Folds 2-10 (combined) as the training set. The second iteration sees Fold 2 as the test set and Folds 1, 3-10 (combined) as the training set. And so on.

What this looks like is as follows:

1	2	3	4	5	6	7	8	9	10
Test	Train	Train	Train	Train	Train	Train	Train	Train	Train

1	2	3	4	5	6	7	8	9	10
Train	**Test**	Train	Train	Train	Train	Train	Train	Train	Train

1	2	3	4	5	6	7	8	9	10
Train	Train	**Test**	Train	Train	Train	Train	Train	Train	Train

And finally:

1	2	3	4	5	6	7	8	9	10
Train	Train	Train	Train	Train	Train	Train	Train	Train	**Test**

When you're done, you would then end up with 10 estimates of MSE. With ten estimates of MSE we're back into familiar statistical inference territory, we can estimate the mean MSE (which sounds funny if you say "mean Mean Squared Error") and the variance.

There are other variants of cross-validation such as 3 fold and 5 fold. The *main* consideration on the number of folds is this: is each fold representative of the distribution you're trying to model? This often depends on:

1. How many observations do you have? 300 or 300 million?
2. How many variables do you have? The more variables you have, the "smaller" your data is from a machine learning perspective.
3. How many values can each categorical variable take? Each possible value of a categorical variable is a partitioning of your data. If you have a lot of categorical variables which can each take on many values, your 300 million observations may actually *not* be enough data.

There has been quite a bit of research on cross validation in general and how well it works. From my own vantage point of seeing students apply 10 fold cross validation to the same problem, I would suggest doing successive rounds of cross validation, creating new folds each time. You should do as many rounds as needed to reach at least 30 estimates of your metric.

For 10 fold cross validation, that's three (3) rounds of 10 fold-cross validation. This is nice because because with 30 observations, you can do Bayesian bootstrap inference on your results, and estimate the posterior distribution of the mean of your evaluation metric.

Why not just do bootstrap sampling directly by taking a random 10% sample as the test set, 100s of times? I often do this because it's easier than setting up folds. However, the literature gives this approach mixed reviews.

Which leads us to one final world of advice: *randomize*. Never use your data just as you found it. Make sure you randomize your data each time before establishing your folds (in other words, *shuffle* your data each time before you designate folds).

3.3.1 Implementation

It's not difficult to implement N-Fold Cross Validation on your own. You can select the indices of your dataframe (or data), randomize them, "chunk" them into N batches, and iterate. However, Scikit Learn does implement N-Fold Cross Validation. You will have to work the Scikit Learn's classes directly instead of the `models` module.

```
[1]: import warnings
     warnings.filterwarnings('ignore')
```

```
[2]: %matplotlib inline
```

[3]:
```
import numpy as np
import scipy.stats as stats
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import patsy
import sklearn.linear_model as linear
import numpy.random as random

sns.set(style="whitegrid")
```

3.4 Bias/Variance Trade Off

When you're done with N-Fold Cross Validation, you have an estimate of the performance of your model ("generalization error" although you might not have calculated *error*) and a credible interval for it. Let's say the mean value of a regression model has a standard error (square root of the mean MSE) of 23.7. Now, whether and by much a given model is good or bad depends entirely on the real world problem. Plus or minus 23.7 would be great if the base value is 1,305 but maybe not so great if the base value is 139.0. I can't tell you how good–in absolute terms–your model is. The answer to that question is, is it better than what you do now (guessing, estimate, or current model)?

But let's say the standard error is 23.7 and that's not as good as you'd hoped. How can we make the model better? In previous chapters, we've taken a domain knowledge centric approach to some of these potential solutions. Does your "theory" indicate you're missing a variable? Does your theory/EDA/residuals indicate that a variable should be transformed?

The *usual* suspects are things like:

1. get more observations
2. get more features or transform them.
3. apply regularization
4. change metaparameters
5. change models

Now what if I told you that getting more observations doesn't always work, you may need to *remove* features, you shouldn't transform those features, or you might even need to remove regularization (which we'll talk about in a bit. . . just think "smoothing")?

In order to get a good indication of what we're going to do, we're going to need a framework. That framework is the Bias/Variance Trade Off. But first, some Computational Learning Theory. . .

3.4.1 Just Enough Theory

Because this is not a machine learning class, I'm only going to give you just enough Computational Learning Theory to get you by. However, the basics are not complicated.

In the abstract, we have some data (X, y) and we want to model y with X. We assert that there is some true relationship, g, between X and y but we cannot observe $y = g(X)$ directly. We do not even know its shape. This the problem of inference in the context of Machine Learning.

We can represent this true relationship using a model $y = f(X)$ where f becomes an approximation of g. But since we don't know g and have no idea what g looks like, we have to pick *some* starting place for f. Do we pick a constant value (like the mean or median), a line (linear regression, SVM), an artificial neural network, or a decision tree?

Picking a particular *form* for f is known as *representational bias*. In this case, *bias* is a good thing. It was proven a long time ago, that if you don't want to just memorize your data, you need to pick a representation of some kind.

Now that we have picked a form for f, we need to estimate it. This requires an algorithm as well which we will call F, $f = F(X, y)$. F finds f. It is in this sense that Herbert Simon quipped, "all of AI is search". The algorithm, F, to find f introduces another bias, the *inductive bias*. A good example of this comes from Decision Trees where a decision tree f can be learned using different algorithms such as CART (Classification and Regression Trees), CHAID (Chi-Square Automatic Interaction Detection), C4.5, and others, all different versions of F. Variations on F can be introduced by using different loss functions as well.

For linear regression, our representation bias is the form of f or a straight line and the inductive bias is mean squared error. You may have heard that linear regression ("Ordinary Least Squares") is "BLUE" (Best Linear Unbiased Estimator) and you'll wonder why I've now said it is biased. This is because "bias" means different things (albeit related) in statistics than machine learning. But in this context, "bias" in statistics and BLUE means how well it estimates a population parameter from a sample. A little bit of that is in the Machine Learning difference. You can think of g as being the population relationship and f as being the estimate from a sample. Even so, in machine learning, bias can never be eliminated (and it is undesirable to do so).

It is interesting to note that Scikit Learn is set up this way in general:

```
1. linear = LinearRegression() # F
2. model = linear.fit(X, y)    # F -> f
3. y_hat = model.predict(X)    # f
```

In line 1, we create a class for F, possibly supplying "meta parameters" (a meta parameter or hyperparameter is a parameter that we pick instead of one that is learned from the data. They usually apply to F but can apply to f as well). In line 2, we fit the model itself. That is, we take F, X and y and generate f. In line 3, we can apply f to a new batch of data, X, and get estimates, $\hat{y}$.

3.4.2 Prejudice

It's very important that your machine learning models do not incorporate *prejudice* by copying such prejudicial and discriminatory actions that may appear your data. This kind of *social* bias in machine learning has received a lot of well deserved attention. We have to be clear, however, that *bias* is a technical term in our field as well. When I say "machine learning models must always be biased", I mean that representational bias and inductive bias cannot be eliminated. They shouldn't be *discriminatory*, however.

3.5 The Tradeoff

We are now in a position to talk about the general framework for discussing predictive modeling: the bias/variance tradeoff. We will take the case of regression because it is a bit easier to visualize. However, the framework can be formulated for classification problems as well as models other than linear regression. If we take our most basic predictive model for regression:

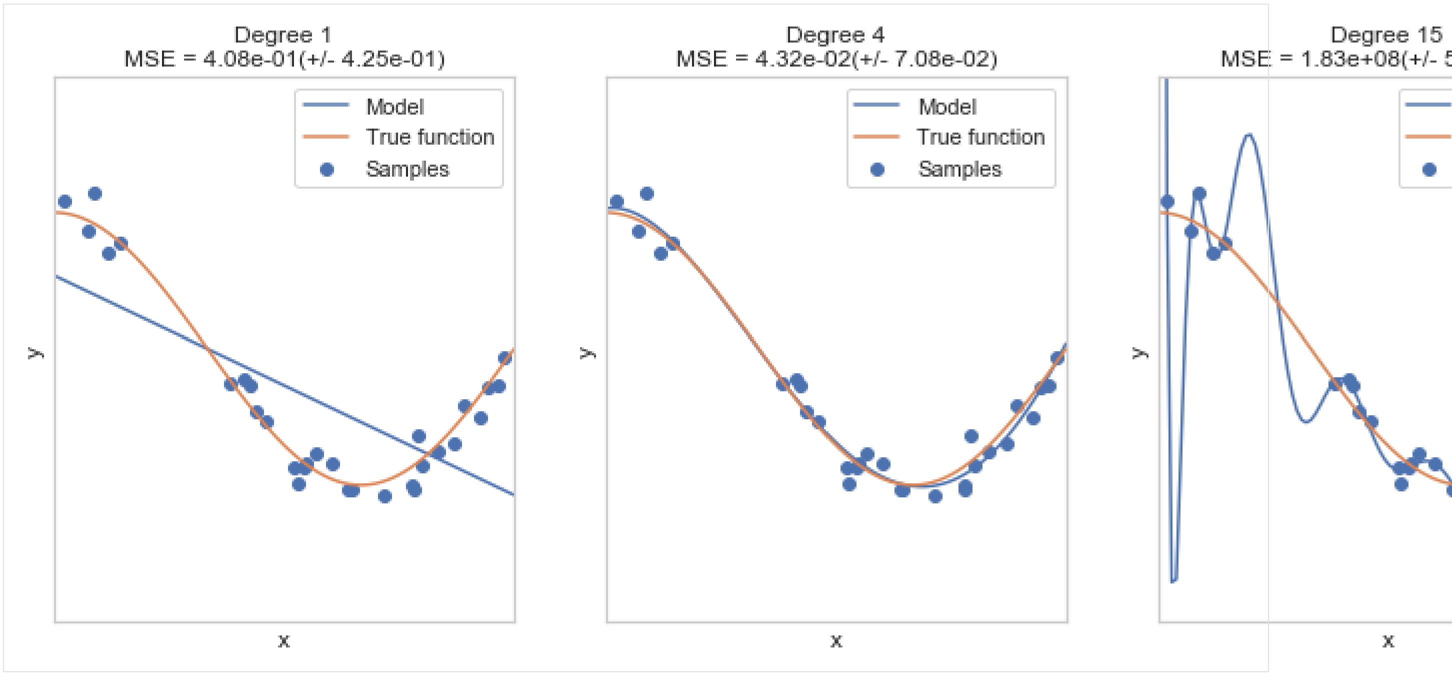
$\hat{y} = \beta_0 + \epsilon$

we can ask ourselves, where does our error $y - \hat{y}$ come from? Conceptually, there are three components of error:

1. **bias** (error) comes from our selection of the model itself, β_0, which may *underfit* the data.
2. **variance** (error) comes from the sensitivity of β_0 to outliers which may *overfit* the data.
3. **irreducible error** comes from ϵ often called "noise", a term I do not like... you can think of it as coming from Rumsfeld's "known unknowns" and "unknown unknowns".

For the most part, we do not have control over **irreducible** error but we may have control over **bias** and **variance**. The problem is that there is generally a *tradeoff*; descreasing one increases the other. Consider a regression problem such as the following:

```
[4]: from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
import sklearn

np.random.seed(0)

n_samples = 30
degrees = [1, 4, 15]

true_fun = lambda X: np.cos(1.5 * np.pi * X)
X = np.sort(np.random.rand(n_samples))
y = true_fun(X) + np.random.randn(n_samples) * 0.1

plt.figure(figsize=(14, 5))
for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)
    plt.setp(ax, xticks=(), yticks=())

    polynomial_features = PolynomialFeatures(degree=degrees[i],
                                             include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([("polynomial_features", polynomial_features),
                         ("linear_regression", linear_regression)])
    pipeline.fit(X[:, np.newaxis], y)

    # Evaluate the models using crossvalidation
    scores = cross_val_score(pipeline,
        X[:, np.newaxis], y, scoring="neg_mean_squared_error", cv=10)

    X_test = np.linspace(0, 1, 100)
    plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="Model")
    plt.plot(X_test, true_fun(X_test), label="True function")
    plt.scatter(X, y, label="Samples")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim((0, 1))
    plt.ylim((-2, 2))
    plt.legend(loc="best")
    plt.title("Degree {}\nMSE = {:.2e}(+/- {:.2e})".format(
        degrees[i], -scores.mean(), scores.std()))

plt.show()
plt.close()
```



source

In each panel, we show a sampling of data (dots) from a true relationship (green line, g) and the resulting model (blue line, f). The first model is:

$\hat{y} = \beta_0 + \beta_1 x_1 + \epsilon$

The results for this model are shown in the first panel. This model has a mean squared error or MSE of 4.08e-01 on average. Ignoring ϵ you can think of *bias* as the difference in the blue dots and the blue line (our data and our model) and *variance* as the difference in the green line and the blue line (the real relationship and our model).

We see that both bias and variance are pretty high but as new data points are observed (somewhere along the green line), the model is going to be pretty consistent in its predictions (along the blue line). We generally call this the "high bias/low variance" case. What we're really saying is that our model was very "opinionated" about the kinds of predictions it would make and took less of the shape of the data into account than we might have liked (representational bias from above). In essence, we "underfit" the data. This is going to *bias* the error of the predictions.

The third model is:

$\hat{y} = \beta_0 + \beta_1 (x_1^{15}) + \epsilon$

The results for this model are shown in the third panel. We see that bias is quite low. . . the blue line goes through most of (not all of) the blue dots. But the *variance* is going to be quite high. If you look where there was no data sampled along the green line, the blue line (prediction) is going to be quite wrong in a lot of spots. This is the high variance/low bias case. In a sense, the model was not opinionated enough. We "overfit" the data. There is going to be large *variance* in the error of the predictions.

The second model is:

$\hat{y} = \beta_0 + \beta_1 (x_1^{4}) + \epsilon$

The results for this model are in the middle panel. Just like Goldilocks, the middle panel shows the case of bias/variance balance. We have not eliminated both types of error. There are cases where the blue dots are quite far from the blue line and there are cases where the blue and green lines diverge. Although it is idealized, it illustrates the concept of what we're aiming for. The problem, of course, is that we can never observe the green line, g.

The question of improving a model based on the five solutions we asked earlier,

1. get more observations
2. get more features or transform them.
3. apply regularization
4. change metaparameters
5. change models

becomes, how do you know if you're in panel 1, 2 or 3 and what do you do about it? As a result, the five solutions become:

1. get more observations *or not*
2. get more features or transform them *or not*
3. apply regularization *or not*
4. change metaparameters *which way?*
5. change models *what kind?*

```
[1]: import warnings
     warnings.filterwarnings('ignore')
```

```
[2]: %matplotlib inline
```

```
[3]: import numpy as np
     import scipy.stats as stats
     import seaborn as sns
     import matplotlib.pyplot as plt
     import pandas as pd
     import patsy
     import sklearn.linear_model as linear
     import numpy.random as random
     from collections import defaultdict

     import sys
     sys.path.append('resources')
     import models


     sns.set(style="whitegrid")
```

3.6 Model Improvement

So we now have all the pieces we need: metrics, cross-validation and an awareness of the bias/variance trade off in order to think how we might go about improving our model's performance. One can reasonably ask, what are the options?

1. **More observations** This is often the first thing that people think about when a model is not doing well, "we need more data." "More data" can mean two different things: either we need more observations (of the variables we have) or we need more/new/different variables (and values for them). The second case is covered below.
2. **Number of Inputs** There are sort of two cases here and we need to distinguish once again between data we have (variables) and data we are using (inputs to the model). The first case assumes we have the variables but we need to change the number of inputs by either decreasing or increasing the number of variables we're using as inputs in the model. This might also include adding or removing transformations of those variables. The second

case involves getting more data. We've used all the variables and transformations we can think of but we believe there is some feature that our model needs that our variables do not capture. Or perhaps some *proxy* variable is not working as well as we would like.

3. **Regularization** Many algorithms have the ability to fine tune the "fit" of a model (as in underfitting and overfitting). This is often called *regularization*. If you are using regularization, you may want to turn it up or down.
4. **Metaparameters** Many algorithms and models use metaparameters. A metaparameter is a parameter that isn't set using data. For example, the threshold value for a logistic regression is a metaparameter.
5. **Model** Sometimes we just need to change our model either from a Poisson Regression to a Linear Regression or from a Logistic Regression to a Random Forest. This will almost always involve revisting everything. On the other hand, this may not be an option if our engineering infrastructure is optimized for Decision Trees.

The first thing we need to do is establish where we appear to be in terms of our algorithm, its parameters, the evaluation metric(s) and our data in terms of the bias/variance trade-off. We can do this with learning curves. Learning curves simulate acquiring more data. If getting more data doesn't improve our metrics, we're likely already overfitting.

3.6.1 Learning Curves

With learning curves we take the algorithm we have with the existing inputs and metaparameters and simulate what happens when more data is obtained. This becomes our baseline.

There are many variations on Learning Curves and this is but one. I find it's the simplest.

```
1. split your data into a training/test set. You use one set of folds from 10 fold␣
↪cross validation.
2. create subsets of your training data of ever increasing size, n% to 100%.
3.     train the model using the training subset.
4.     calculate your metric on the training subset and keep it
5.     calculate your metric on the test set and save it.
6. plot the saved metrics as the "train curve" and the "test curve".
```

The Line 1. is probably the most mysterious. You want to simulate what would happen if you got more observations so you take random subsets from your data of ever increasing size…5%, 10%, 15%…all the way up to 100%. The size of these chunks will depend on how much data you have and how complicated F and f are.

Here's a concrete excample. Suppose were calculating MSE for a linear regression. First, we split our data in 10 folds. Pick Fold 1 as the test set. Combine Folds 2-10. Suppose we have enough data that we can pick 5% increments and have enough to train our model. Pick 5% of your training set at random, calculate the model, and calculate MSE *on the training set*. We don't normally evaluate performance on the training set. Everyone tells you not to do this. For learning curves, we need a baseline of comparison so we do it. Save the value of MSE so you can plot it later.

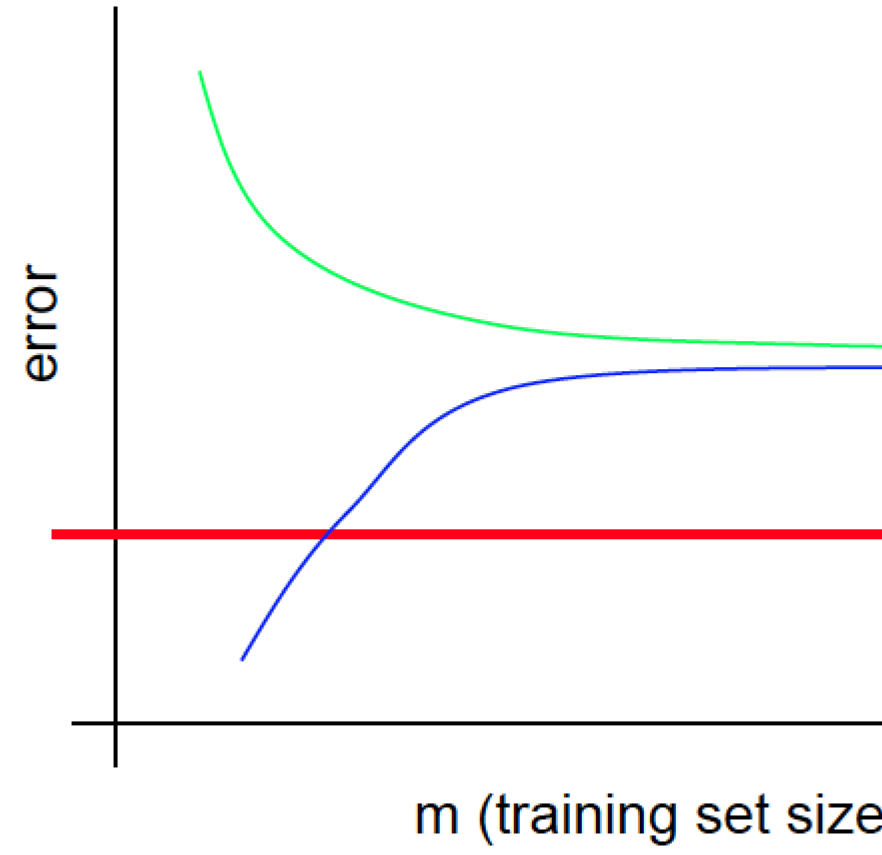
Now apply the model to the test set and calculate MSE. Save it so you can plot it later. Repeat the same thing for 10% then 15% then 20% of your training data. Then plot the two curves.

You should be able to see that by estimating models with first 5% of your training data, then 10% of your training data, then 15%, etc., you are simulating getting more data.

3.6.2 Textbook Learning Curves

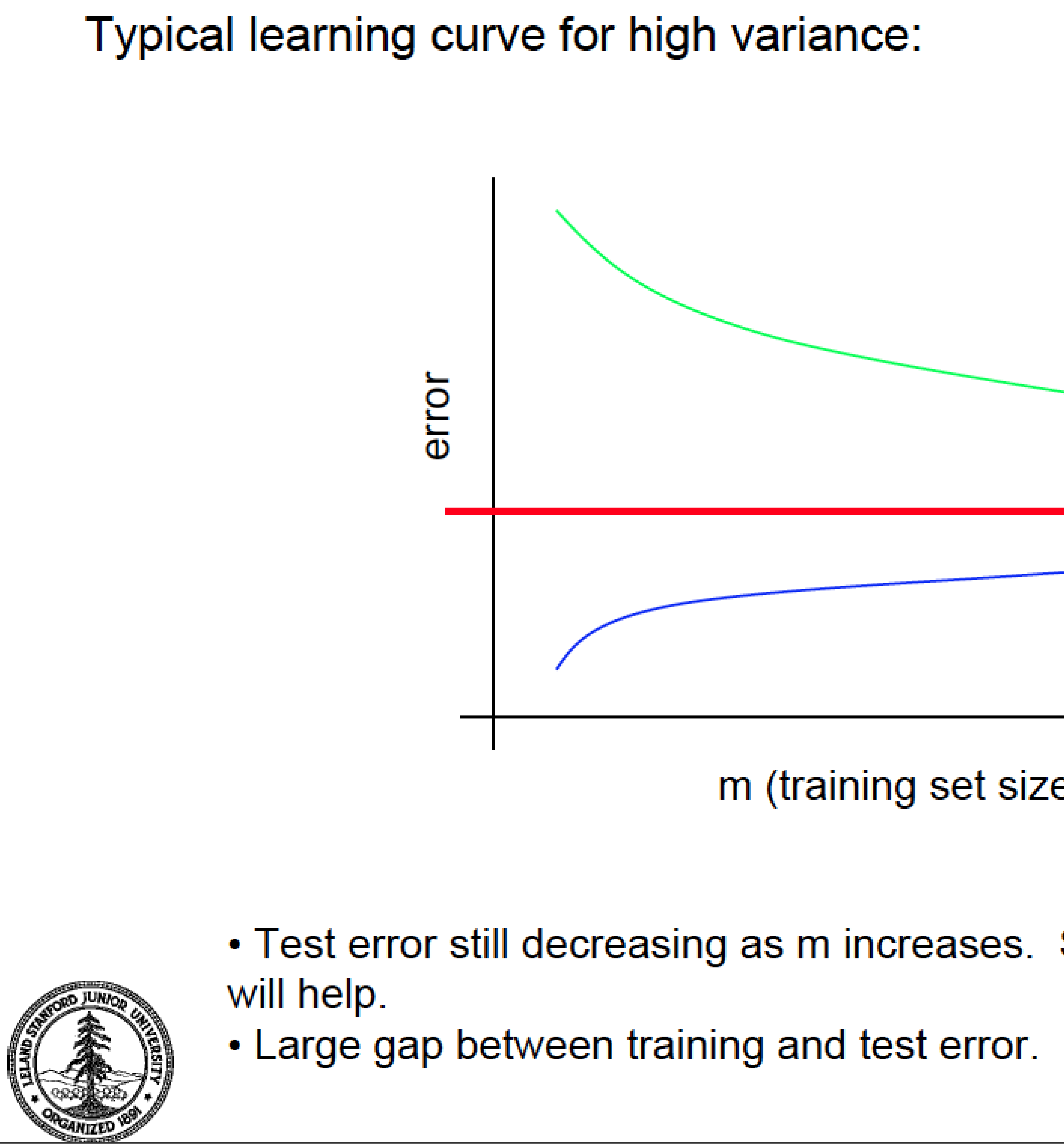
Here are a few textbook examples of learning curves. Do not expect your real world curves to behave as well. The main point here is to be able to identify if you are a "high bias" or "high variance" situation and remember that these are all relative to you current data and model.

Typical learning curve for high bias:



- Even training error is unacceptably high.
- Small gap between training and test error.

Of course, if the *maximum* acceptable error/desired performance is higher than the error to which the curves has converged, go with it!



Typical learning curve for high variance:

• Test error still decreasing as m increases.
will help.
• Large gap between training and test error.

And again, whether or not this is a problem depends on your desired performance. Of course, in this case, it would appear that even if the desired performance were lower (acceptable error higher), you could still improve the model by getting more data.

3.6.3 Example with Linear Regression

Consider the following (unobserved) relationship:

$y = 3.89 + 2.48 \times x_1 - 5.58 \times x_2 + 1.98 \times x_3 + \epsilon(0, 2.75)$

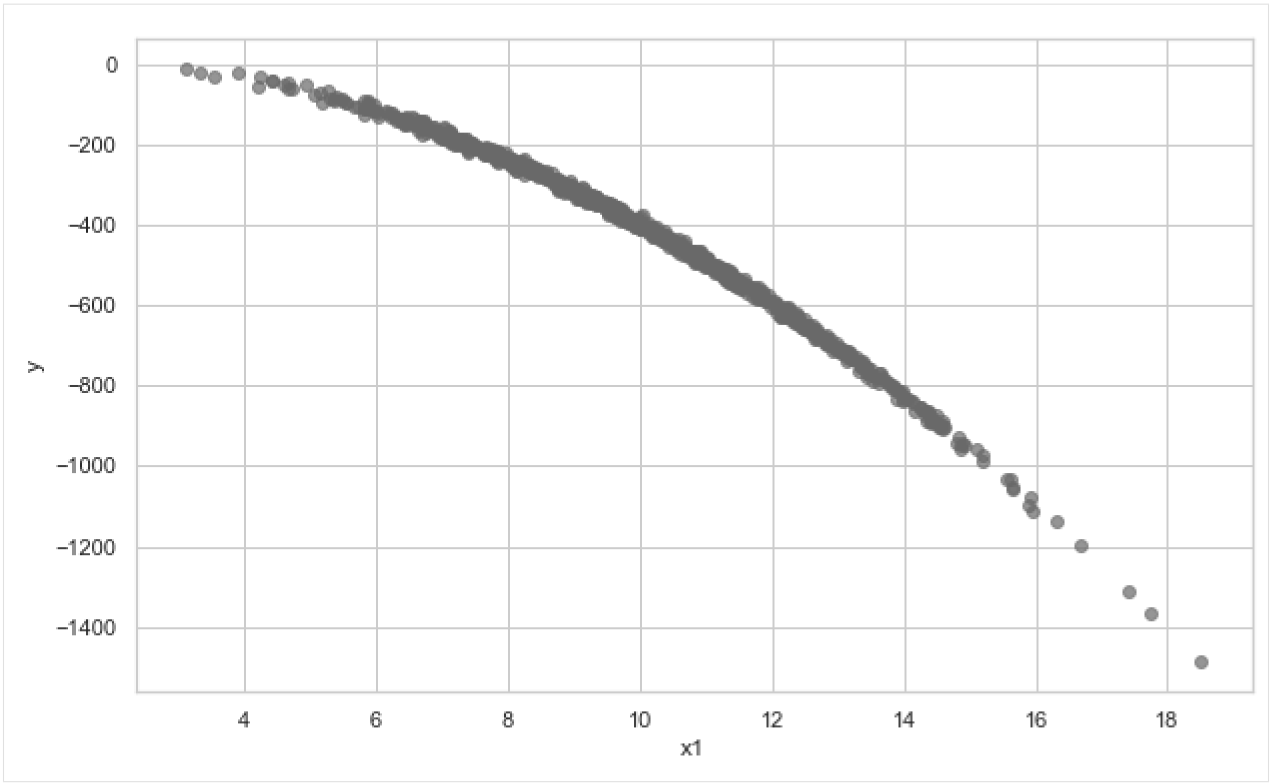
where $x_2 = x_1^2$.

[4]:
```
data = {}
data[ "x1"] = stats.norm.rvs(10.0, 2.5, 1000) #np.array( [random.uniform(-1.0, 1.0)␣
↪for i in range( 0, 1000)])
data[ "x2"] = data[ "x1"]**2
data[ "x3"] = stats.norm.rvs(15.0, 3.8, 1000) # np.array( [random.uniform(-1.0, 1.0)␣
↪for i in range( 0, 1000)])
epsilon = stats.norm.rvs(0, 2.45, 1000) # np.array([ random.normal( 0.0, 1.45) for i␣
↪in range( 0, 1000)])

data[ "y"] = 3.89 + 2.48 * data[ "x1"] - 4.58 * data[ "x2"] + 1.98 * data[ "x3"] +␣
↪epsilon
data = pd.DataFrame( data)
print(data.info())
print(data.head())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 4 columns):
x1     1000 non-null float64
x2     1000 non-null float64
x3     1000 non-null float64
y      1000 non-null float64
dtypes: float64(4)
memory usage: 31.3 KB
None
          x1          x2         x3           y
0   8.059001   64.947505  16.965037 -241.217214
1   8.370450   70.064426  13.835974 -271.650010
2  12.276526  150.713091  15.646462 -624.551057
3  14.397016  207.274071  13.116884 -881.344104
4  13.331793  177.736704  12.257666 -751.405218
```

What does this look like?

[5]:
```
figure = plt.figure(figsize=(10, 6))

axes = figure.add_subplot(1, 1, 1)
axes.scatter( data[ "x1"], data["y"], marker="o", color="dimgray", alpha=0.7)
axes.set_ylabel( "y")
axes.set_xlabel( "x1")

plt.show()
plt.close()
```



Here's our first model and how we've looked at it so far:

[6]:
```
result1 = models.bootstrap_linear_regression("y ~ x1", data=data)
print(models.describe_bootstrap_lr(result1))
```

```
Model: y ~ x1
--------  ------  ----  -----  ----
                               95% BCI
Coefficients                  Mean     Lo        Hi
              $\beta_0$  480.63   457.67    494.50
x1            $\beta_1$  -90.62   -92.10    -88.35

Metrics        Mean         Lo        Hi
$\sigma$       36.59        32.12     40.05
$R^2$          0.97         0.97      0.98
--------  ------  ----  -----  ----
```

For convenience, let's do it a more low level way, using the classes directly:

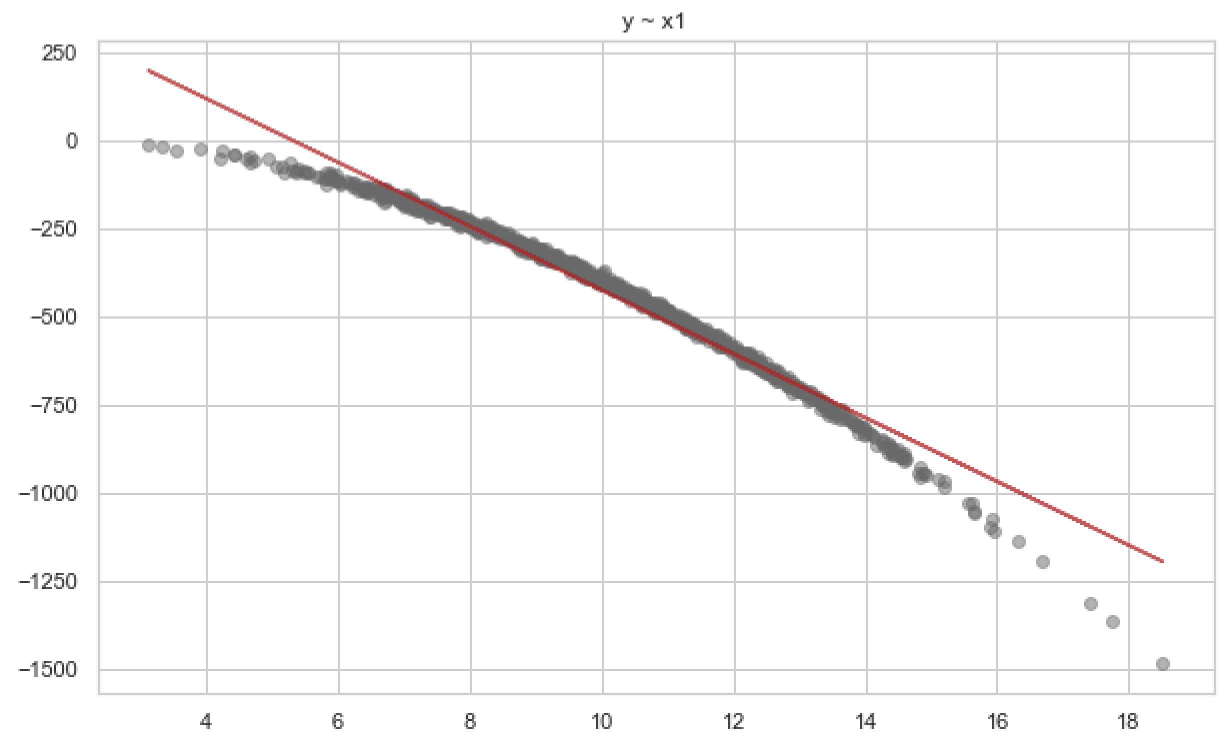
and visualize the result:

[7]:
```
figure = plt.figure(figsize=(10,6))

axes = figure.add_subplot(1, 1, 1)

xs = data[ "x1"]
ys = data[ "y"]
axes.scatter( xs, ys, color="dimgray", alpha=0.5)
betas = result1[ "coefficients"]
axes.plot(xs, [betas[ 0] + betas[ 1] * x for x in xs], '-', color="firebrick",␣
↪alpha=0.75)
```

(continues on next page)

(continued from previous page)

```
axes.set_title( result1[ "formula"])

plt.show()
plt.close()
```



```
[8]: def sse(results):
         errors = results['residuals']
         n = len( errors)
         squared_error = np.sum( [e**2 for e in errors])
         return np.sqrt((1.0/n) * squared_error)
```

```
[9]: print(sse(result1))
```

```
36.551314648178995
```

Instead of MSE, I'm using standard error (the square root) and they're in the same units as y.

```
[10]: data['y'].describe()
```

```
[10]: count     1000.000000
      mean      -424.541687
      std        219.595051
      min      -1488.462383
      25%       -547.012490
      50%       -391.522806
      75%       -255.124823
      max        -11.092131
      Name: y, dtype: float64
```

Looking at y, we see it ranges from -1322 to 45 so a standard error of 42-43, isn't horrible. But, will getting more data help?

Let's follow the formula for learning curves above. Basically, we need two functions. The first function does cross-validation. It's written to work with the functional way we interact with Scikit learn and it will repeat N Fold Cross Validation (often abbreviated as CV, for example, 5CV or 10CV) any number of times.

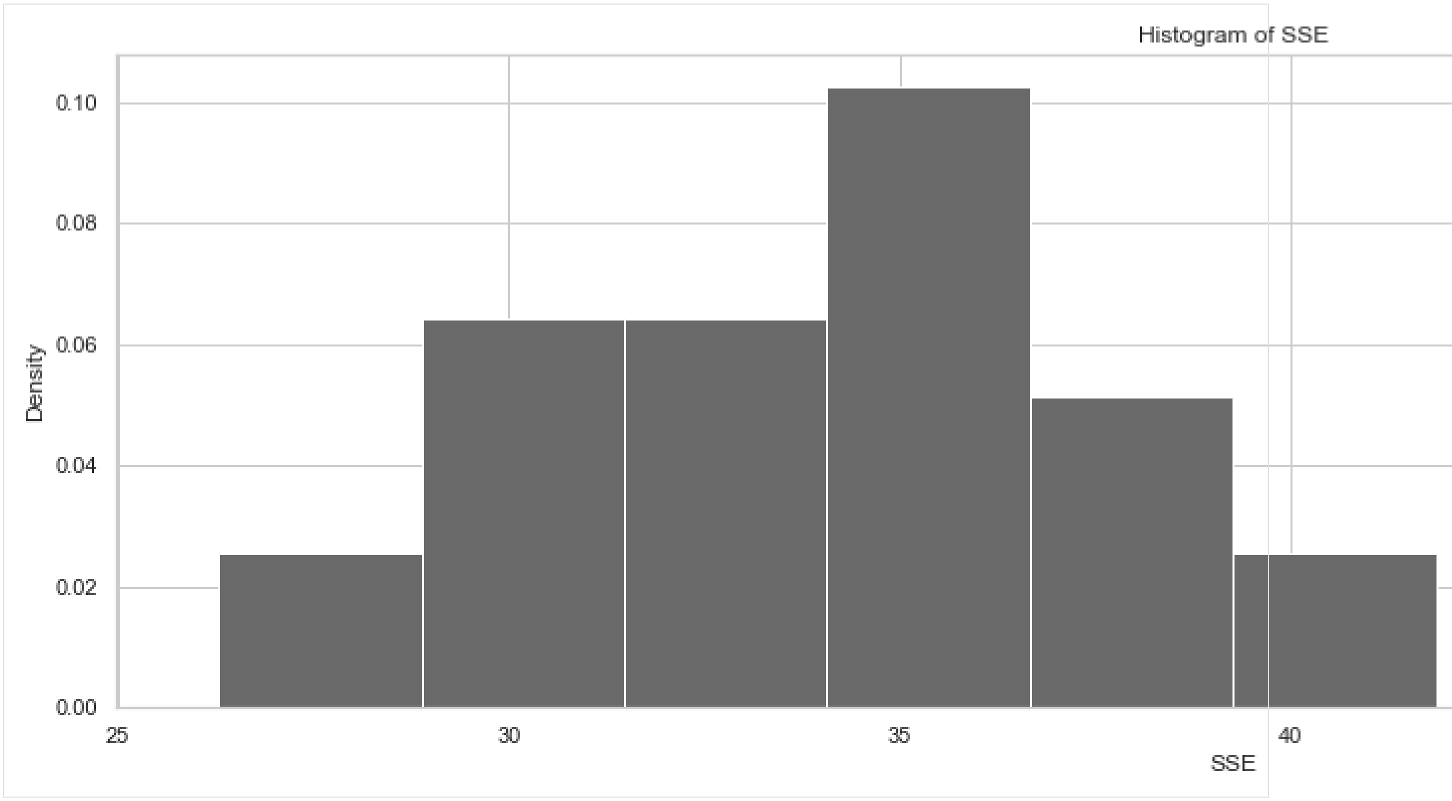
```
[11]: def chunk(xs, n):
          k, m = divmod(len(xs), n)
          return [xs[i * k + min(i, m):(i + 1) * k + min(i + 1, m)] for i in range(n)]
```

```
[12]: def cross_validation(algorithm, formula, data, evaluate, fold_count=10,␣
      ↪repetitions=3):
          indices = list(range(len( data)))
          metrics = []
          for _ in range(repetitions):
              random.shuffle(indices)
              folds = chunk(indices, fold_count)
              for fold in folds:
                  test_data = data.iloc[fold]
                  train_indices = [idx not in fold for idx in indices]
                  train_data = data.iloc[train_indices]
                  result = algorithm(formula, data=train_data)
                  model = result["model"]
                  y, X = patsy.dmatrices(formula, test_data, return_type="matrix")
                  # y = np.ravel( y) # might need for logistic regression
                  results = models.summarize(formula, X, y, model)
                  metric = evaluate(results)
                  metrics.append(metric)
          return metrics
```

```
[13]: results = cross_validation(models.linear_regression, "y ~ x1", data, sse)
```

What do we hope to see here? This is the cross validation evaluation of the model. We can compare it to the Boostrap evaluation we did earlier:

```
[14]: figure = plt.figure(figsize=(20, 6))

      axes = figure.add_subplot(1, 1, 1)
      axes.hist(results, color="DimGray", density=True)
      axes.set_xlabel("SSE")
      axes.set_ylabel("Density")
      axes.set_title("Histogram of SSE")

      plt.show()
      plt.close()
```



This is the histogram of the (raw) Standard Error. The mean Standard Error is:

```
[15]: np.mean(results)
```

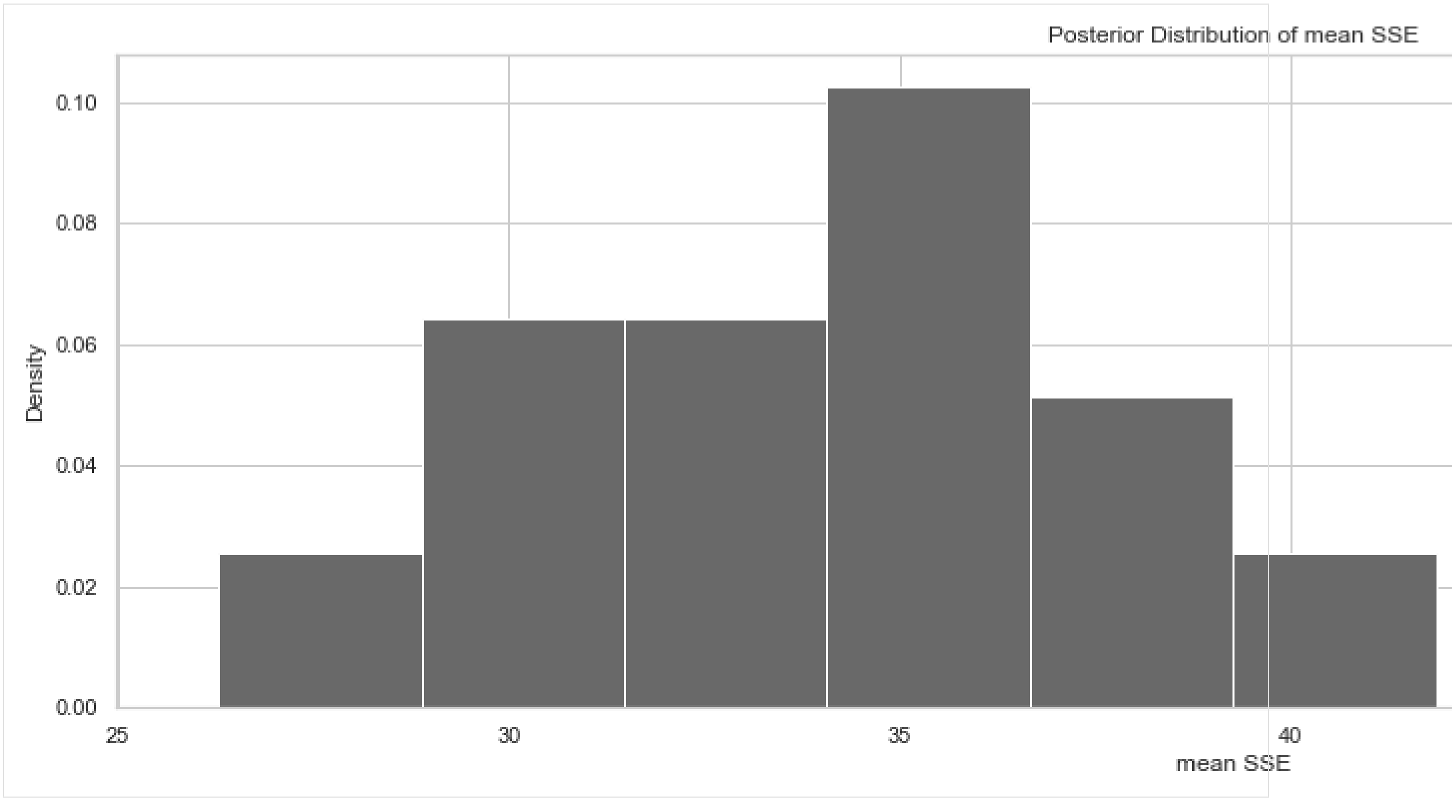
```
[15]: 36.06459702906318
```

which is very nearly what we got above using the Bootstrap. In order to get intervals, as with any other parameters, we need to estimate the posterior distribution using Bootstrap sampling.

```
[16]: def resample(data):
          n = len(data)
          return [data[ i] for i in [stats.randint.rvs(0, n - 1) for _ in range( 0, n)]]
```

```
[17]: posterior_sse = np.array([np.mean(s) for s in [resample(results) for i in range( 0,␣
      ↪1000)]])
```

```
[18]: figure = plt.figure(figsize=(20, 6))

      axes = figure.add_subplot(1, 1, 1)
      axes.hist(results, color="DimGray", density=True)
      axes.set_xlabel("mean SSE")
      axes.set_ylabel("Density")
      axes.set_title("Posterior Distribution of mean SSE")

      plt.show()
      plt.close()
```



What is the 95% credible interval?

```
[19]: print(r"95% CI for *mean* SSE:", stats.mstats.mquantiles(posterior_sse, [0.025, 0.
      ↪975]))
```

```
95% CI for *mean* SSE: [33.86591605 38.53372901]
```

This bound is a bit tighter than the (37, 46) above. I wouldn't expect it to work out like this all time. In fact, most of the time, the error in cross validation can be different (larger or more variable) than the one estimated directly against all of the data.

But let's get to the real question, will getting more data help? For that we need to estimate actual learning curves.

3.6.4 Learning Curves

Scikit Learn has classes and functions for learning curves. However, I like to do a slight variation on the usual learning curve recipe in that I like to give the train and test curves *credible intervals*. This helps me interpret the curves because learning curves in real life rarely look exactly the way they do in the textbooks.

```
[20]: def data_collection():
          result = dict()
          result[ "train"] = defaultdict( list)
          result[ "test"] = defaultdict( list)
          return result
```

```
[21]:  def learning_curves(algorithm, formula, data, evaluate, fold_count=10, repetitions=3,
      ↪ increment=1):
          indices = list(range(len( data)))
```

(continues on next page)

(continued from previous page)

```
    results = data_collection()
    for _ in range(repetitions):
        random.shuffle(indices)
        folds = chunk(indices, fold_count)
        for fold in folds:
            test_data = data.iloc[ fold]
            train_indices = [idx for idx in indices if idx not in fold]
            train_data = data.iloc[train_indices]
            for i in list(range(increment, 100, increment)) + [100]: # ensures 100%␣
↪is always picked.
                # the indices are already shuffled so we only need to take ever␣
↪increasing chunks
                train_chunk_size = int( np.ceil((i/100)*len( train_indices)))
                train_data_chunk = data.iloc[train_indices[0:train_chunk_size]]
                # we calculate the model
                result = algorithm(formula, data=train_data_chunk)
                model = result["model"]
                # we calculate the results for the training data subset
                y, X = patsy.dmatrices( formula, train_data_chunk, return_type="matrix
↪")
                result = models.summarize(formula, X, y, model)
                metric = evaluate(result)
                results["train"][i].append( metric)

                # we calculate the results for the test data.
                y, X = patsy.dmatrices( formula, test_data, return_type="matrix")
                result = models.summarize(formula, X, y, model)
                metric = evaluate(result)
                results["test"][i].append( metric)
            #
        #
    # process results
    # Rely on the CLT...
    statistics = {}
    for k, v in results["train"].items():
        statistics[ k] = (np.mean(v), np.std(v))
    results["train"] = statistics
    statistics = {}
    for k, v in results["test"].items():
        statistics[ k] = (np.mean(v), np.std(v))
    results["test"] = statistics
    return results
#
```

Because of this, the calculation of the learning curves may take a bit longer than usual. When you're first exploring your model, if you take the approach of adding credible intervals, you might want to use a larger increment than the default of 1%. Ironically, the more data you have the better a smaller increment is and the longer it takes. And then you have to add 3 repetitions of 10 fold cross validation on top of that for each interval.

[22]:
```
results = learning_curves(models.linear_regression, "y ~ x1", data, sse)
```

Now that we have results, we can plot them. Let's create a few more helper functions:

[23]:
```
def results_to_curves( curve, results):
    all_statistics = results[ curve]
    keys = list( all_statistics.keys())
```

(continues on next page)

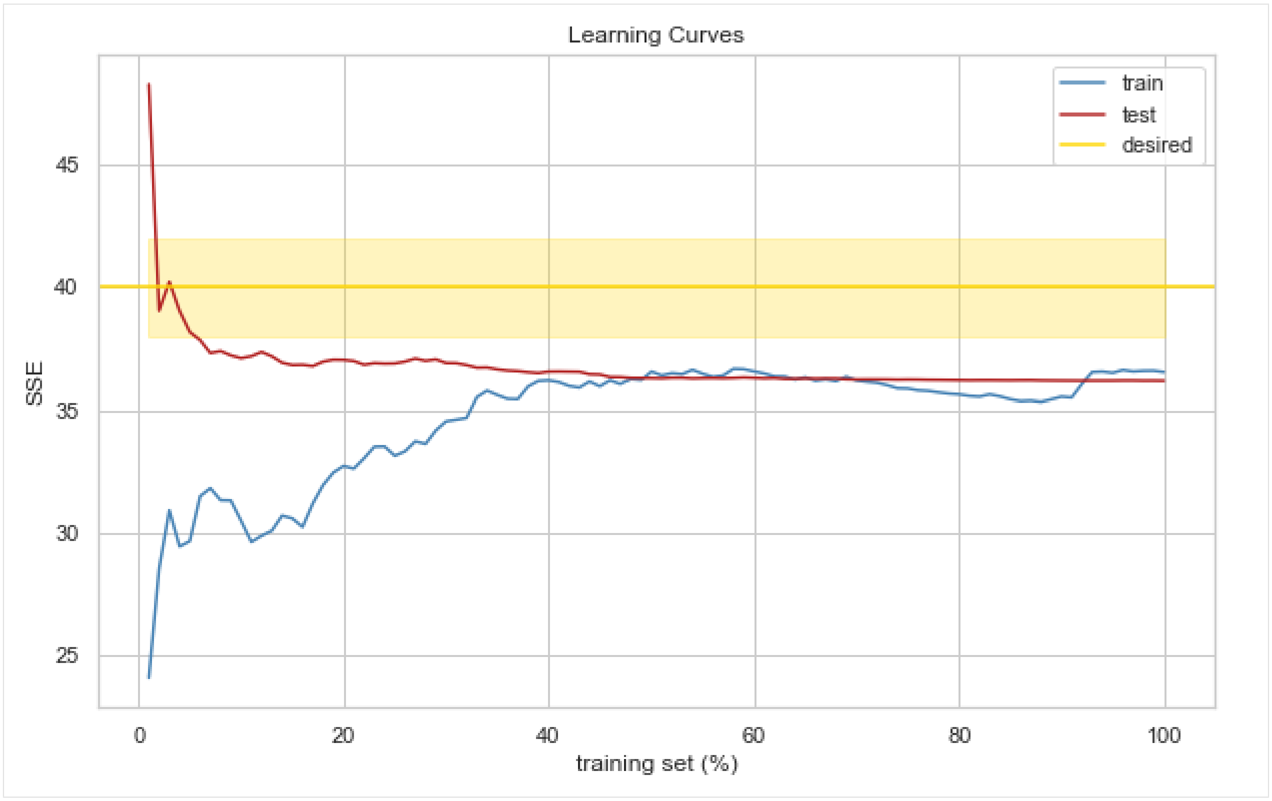
(continued from previous page)

```
    keys.sort()
    mean = []
    upper = []
    lower = []
    for k in keys:
        m, s = all_statistics[ k]
        mean.append( m)
        upper.append( m + 2 * s)
        lower.append( m - 2 * s)
    return keys, lower, mean, upper
```

[24]:
```
def plot_learning_curves( results, metric, desired=None, zoom=False, credible=True):
    figure = plt.figure(figsize=(10,6))

    axes = figure.add_subplot(1, 1, 1)

    xs, train_lower, train_mean, train_upper = results_to_curves( "train", results)
    _, test_lower, test_mean, test_upper = results_to_curves( "test", results)

    axes.plot( xs, train_mean, color="steelblue", label="train")
    axes.plot( xs, test_mean, color="firebrick", label="test")
    if credible:
        axes.fill_between( xs, train_upper, train_lower, color="steelblue", alpha=0.
↪25)
        axes.fill_between( xs, test_upper, test_lower, color="firebrick", alpha=0.25)

    if desired:
        if type(desired) is tuple:
            axes.axhline((desired[0] + desired[1])/2.0, color="gold", label="desired")
            axes.fill_between( xs, desired[1], desired[0], color="gold", alpha=0.25)
        else:
            axes.axhline( desired, color="gold", label="desired")

    axes.legend()
    axes.set_xlabel( "training set (%)")
    axes.set_ylabel( metric)
    axes.set_title("Learning Curves")

    if zoom:
        y_lower = int( 0.9 * np.amin([train_lower[-1], test_lower[-1]]))
        y_upper = int( 1.1 * np.amax([train_upper[-1], test_upper[-1]]))
        axes.set_ylim((y_lower, y_upper))

    plt.show()
    plt.close()
#
```

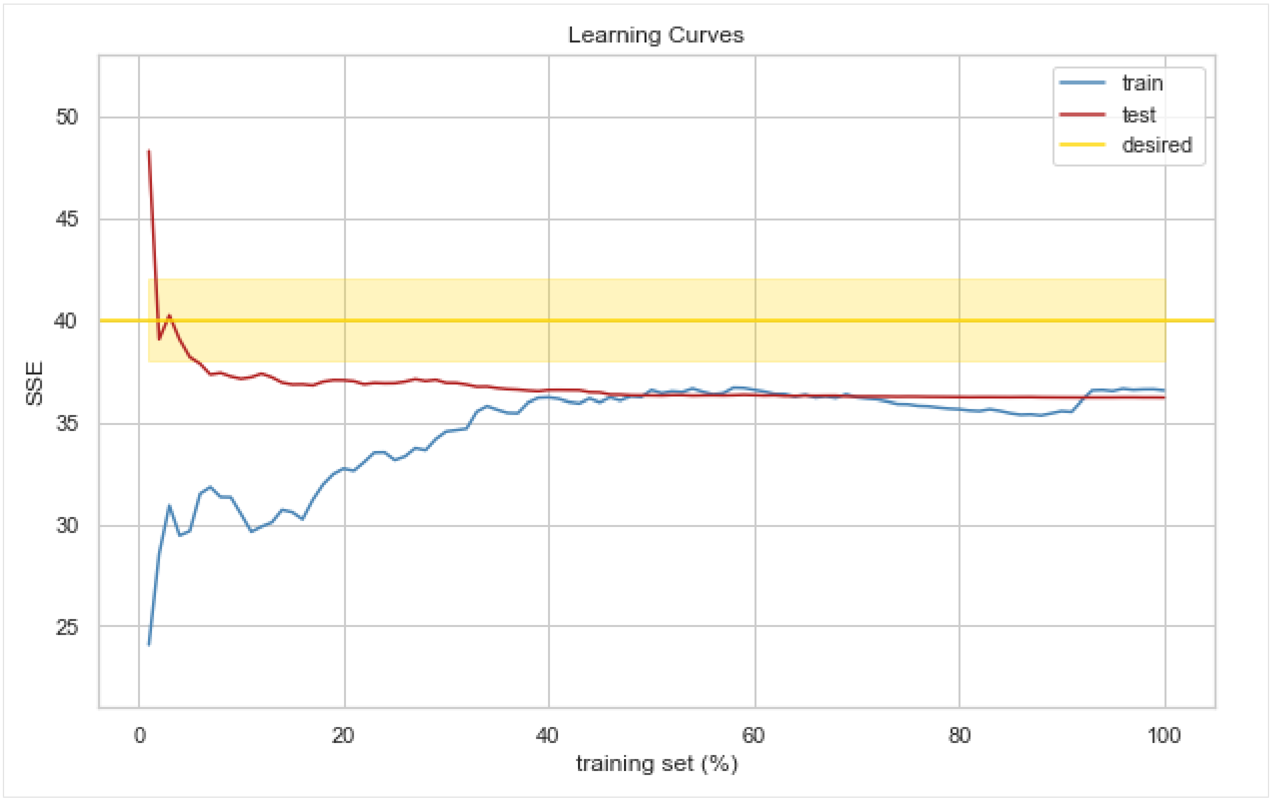
First, let's look at the learning curves without credible intervals (the textbook version). Let's also assume that we have a desired performance of about 40 (38-42) which we can take as our ROPE. As we expect, the model's performance is quite unstable when only using some of our data, about 1% to 40%. However, once we are using 40% of our data, the results are pretty consistent and the metric for both the training and the test data appear to have converged.

[25]:
```
plot_learning_curves(results, r"SSE", desired=(38, 42), credible=False)
```



The problem, however, is that because of the scale of errors at the start, the difference in errors (metrics) at the *end* might be deceptively small. We can “zoom” in, however.

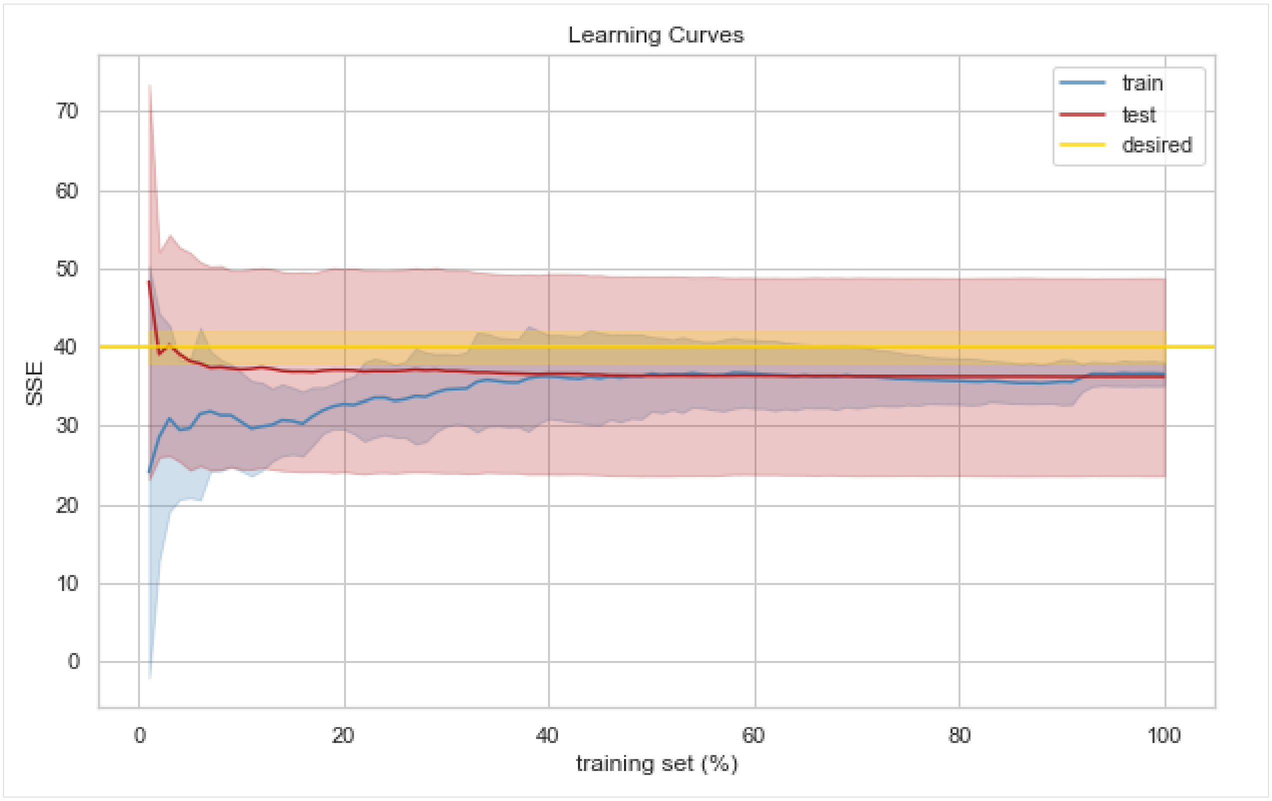
```
[26]: plot_learning_curves(results, r"SSE", desired=(38, 42), zoom=True, credible=False)
```



The errors are really just 1-2 units away so this has definitely converged. **Getting more data will not increase the performance of this model.** Additionally, we are above our desired performance and converged, which means we are in a high bias situation. Remember that "high bias" or "high variance" is always defined relative to the model, the data, the evaluation metric. Bias or variance cannot be absolutely high.

We can also look at the Learning Curves with credible intervals. This can help when a Learning Curve plot is particularly noisy. We need to be reminded if the differences we observe are both *important* and if they are also credible.

```
[27]: plot_learning_curves(results, r"SSE", desired=(38, 42))
```



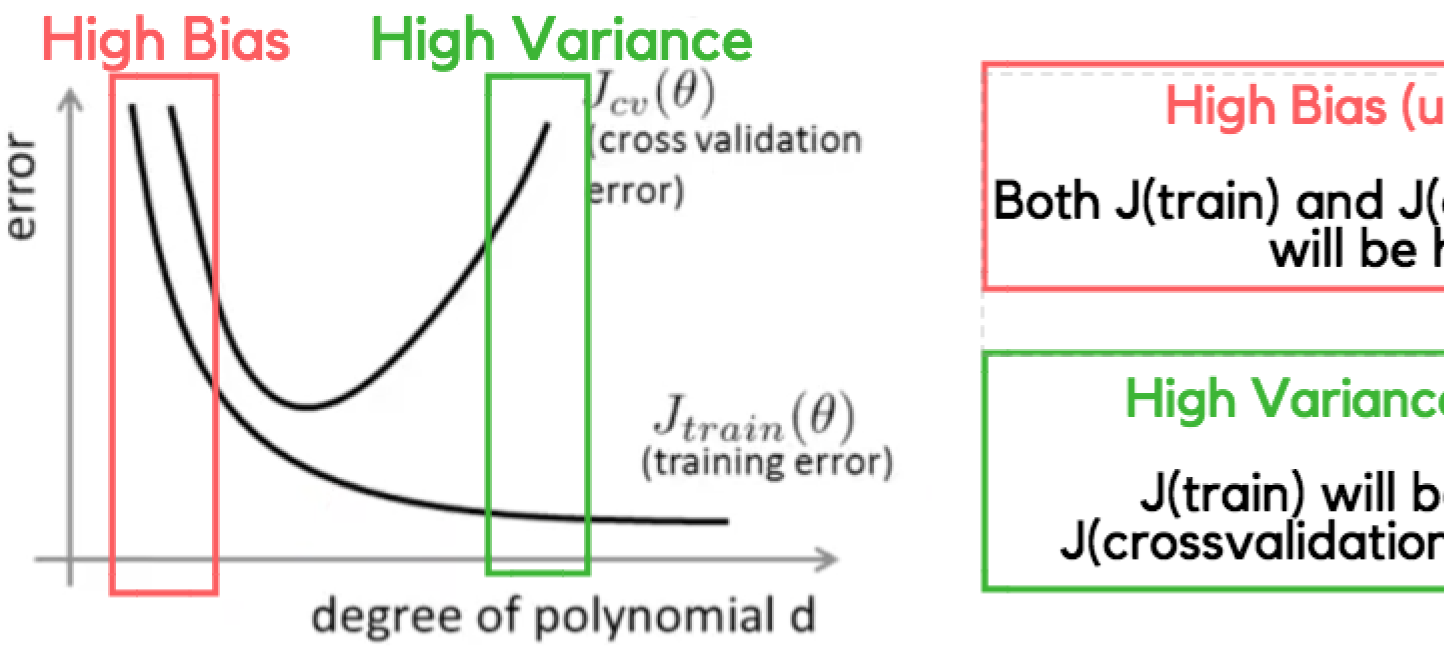
So it is worth reminding ourselves that because of the size of the test set, the variance in the error metric (estimated posterior distribution) is so large that the observed difference *may* just be because of sampling.

3.6.5 Validation Curves

We still have a few more questions to answer in terms of "how can I improve my model". For questions that surround features, their transformations, and metaparameters, we turn to *validation curves*.

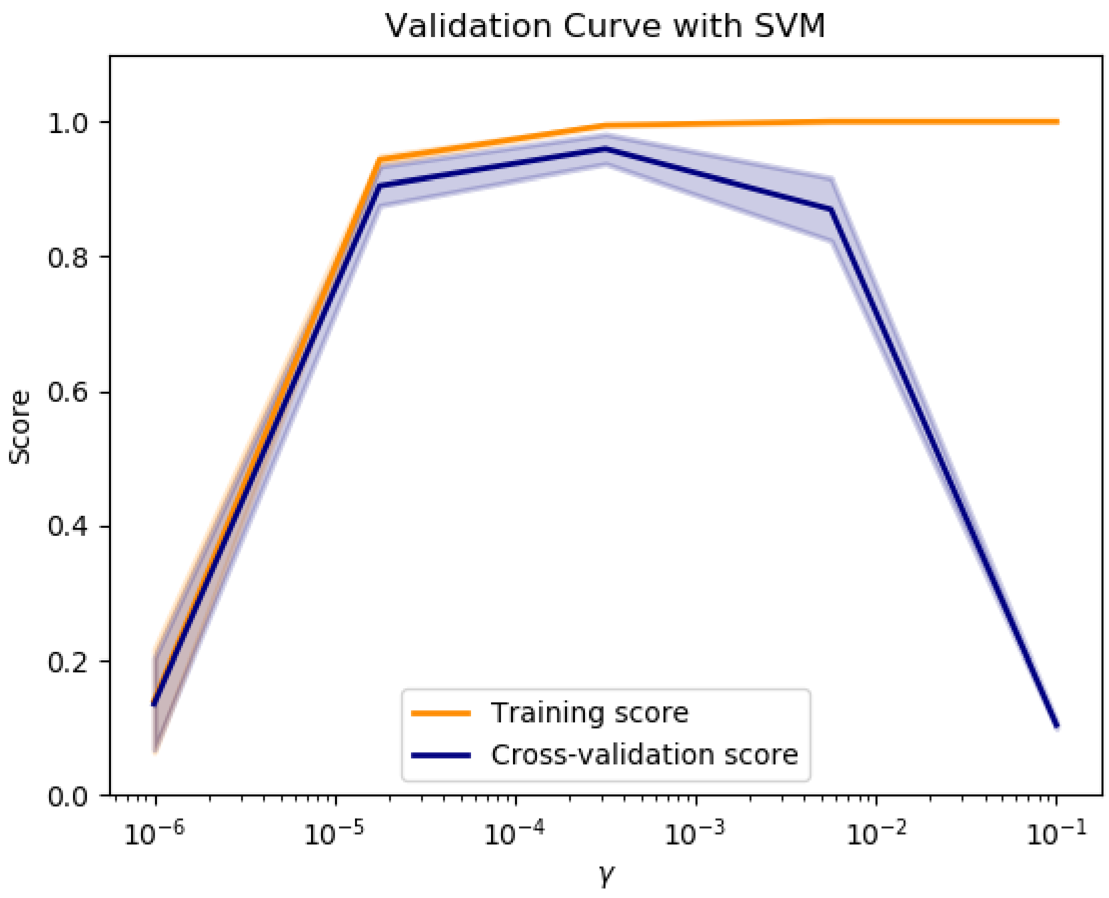
The basic idea behind validation curves is the same as learning curves. We vary something (regularization, training epochs, features, transformations) and see how our performance metric does on the training set compared to the test set. It helps to know that the test set is also called the validation set. . . hence, *validation* curves.

As with Learning Curves, Validation Curves can help you determine if which parameter values result in either a high bias or high variance situation.



As the graphic above illustrates, for the low values of d, you would be in a high bias zone of the parameter. This is also known as "underfitting". Why? For "bad" values of d, the error of the model is high on both the training data and the test data (cross validation set). For the high values of d, you would be in the high variance zone of the parameter value. This is also known as "overfitting". Why? For "bad" values of "d", the error of the model is low on the training data (you have overfit it) and high on the test data (because you overfit the training data). The ideal value of the parameter that balances under and overfitting is the one where the test curve (cross validation curve) has its smallest value.

Please note that if you are measuring something good (like accuracy) instead of bad (like error), then the curves will be upside down!



We already saw how, if we plot Learning Curves, and we're in a *high variance* situation, getting more observations will help (but not if we're in *high bias*). What about for Validation curves?

- High Bias (underfitting)

1. Increase features or add transformations.
2. Decrease regularization

- High Variance (overfitting)

1. Remove features
2. Increase regularization

3.6.6 Grid Search

Validation Curves assume that the item you're varying can be ordered. If you have something that can't be ordered (say, using two different sets of features), you can still look at the bias/variance trade off by using "grid search".

Grid search basically says "try every value of the parameter" (for example, every subset of variables you're interested in) and calculate the training and test error. Pick the parameter value with the lowest *test* error. Any value where the training and test errors are both high is a "high bias" situation and any where the training error is low and the test error is high is a "high variance" situation.

3.6.7 Example of Validation Curves

Of course, one big difference is that because we're not trying to simulate getting more data, we will use all of it for the validation curves. Let's look at the case where we investigate different polynomial transformations of x_1.

This is only an example with an easy visual interpretation This is not the most common use of validation curves. In fact, I would generally stick with EDA and looking at residuals for determining things like the polynomial degree of transformation for a feature.

With the warning out of the way, let's add $x_1^2 ... x_1^5$ to the `data` remembering that we don't know that x_2 *is* x_1^2. We're also going to re-add x_1^1 using the same notation:

```
[28]: data[ "x_1"] = data[ "x1"]**1
      data[ "x_2"] = data[ "x1"]**2
      data[ "x_3"] = data[ "x1"]**3
      data[ "x_4"] = data[ "x1"]**4
      data[ "x_5"] = data[ "x1"]**5
```

We want to do the same basic `cross_validation` function except we want to use a different model each time with successively larger polynomial values of x_1:

$\hat{y} = \beta_0 + \beta_1 x_1$

$\hat{y} = \beta_0 + \beta_1 x_1^2$

$\hat{y} = \beta_0 + \beta_1 x_1^3$

$\hat{y} = \beta_0 + \beta_1 x_1^4$

$\hat{y} = \beta_0 + \beta_1 x_1^5$

We want to find the highest order term that improves the performance of the model. We could also, theoretically, start testing lower order terms after that but we'll still with the highest for now.

```
[29]: def validation_curves(algorithm, formulas, data, values, evaluate, fold_count=10,␣
      ↪repetitions=3, increment=1):
          indices = list(range(len( data)))
          results = data_collection()
          for _ in range(repetitions):
              random.shuffle(indices)
              folds = chunk(indices, fold_count)
              for fold in folds:
                  test_data = data.iloc[ fold]
                  train_indices = [idx for idx in indices if idx not in fold]
                  train_data = data.iloc[train_indices]
                  for i, p in enumerate(zip(formulas, values)):
                      f, v = p
                      # it's ok to resuse the folds for each v of values
                      # we calculate the model
                      result = algorithm(f, train_data, v)
                      model = result["model"]
                      # we calculate the results for the training data subset
                      y, X = patsy.dmatrices(f, train_data, return_type="matrix")
                      result = models.summarize(f, X, y, model)
                      metric = evaluate(result)
                      results["train"][i].append( metric)

                      # we calculate the results for the test data.
                      y, X = patsy.dmatrices(f, test_data, return_type="matrix")
                      result = models.summarize(f, X, y, model)
```

(continues on next page)

(continued from previous page)

```
                metric = evaluate(result)
                results["test"][i].append( metric)
            #
        #
    # process results
    # Rely on the CLT...
    statistics = {}
    for k, v in results["train"].items():
        statistics[ k] = (np.mean(v), np.std(v))
    results["train"] = statistics
    statistics = {}
    for k, v in results["test"].items():
        statistics[ k] = (np.mean(v), np.std(v))
    results["test"] = statistics
    return results
#
```

The function above is set up to take lists of algorithms, formulas, and the values of a parameter to try:

[30]:
```
values = [1, 2, 3, 4, 5]

formulas = []
formula = "y ~ "
for v in values:
    f = f"y ~ x_{str(v).replace('.', '_')}"
    formulas.append(f)

def f(formula, data, v):
    return models.linear_regression(formula, data, style="linear")

result = validation_curves(f, formulas, data, values, lambda r: r["sigma"])
```

We also need a function to plot validation curves:

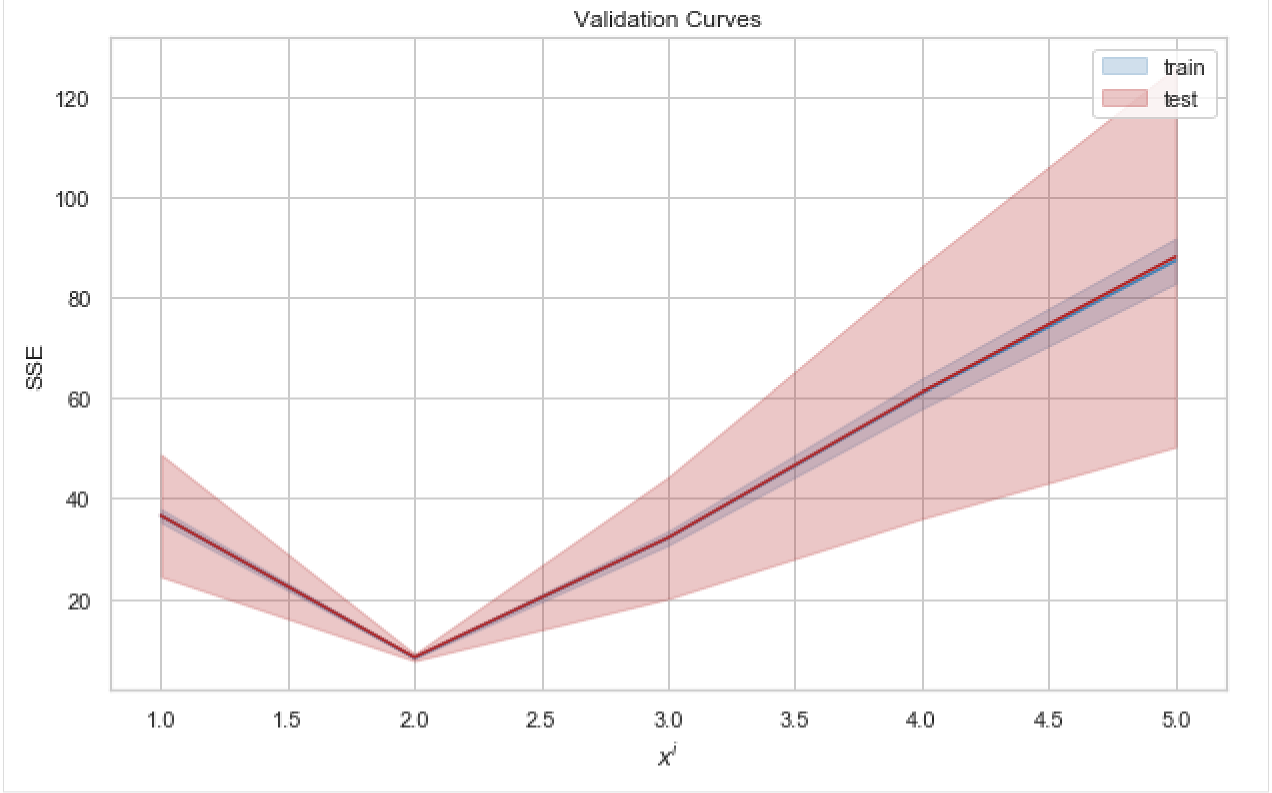
[31]:
```
def plot_validation_curves( results, metric, parameter, values, zoom=False):
    figure = plt.figure(figsize=(10,6))

    axes = figure.add_subplot(1, 1, 1)

    xs, train_lower, train_mean, train_upper = results_to_curves( "train", results)
    _, test_lower, test_mean, test_upper = results_to_curves( "test", results)

    axes.plot( values, train_mean, color="steelblue")
    axes.fill_between( values, train_upper, train_lower, color="steelblue", alpha=0.
↪25, label="train")
    axes.plot( values, test_mean, color="firebrick")
    axes.fill_between( values, test_upper, test_lower, color="firebrick", alpha=0.25,␣
↪label="test")
    axes.legend()
    axes.set_xlabel( parameter)
    axes.set_ylabel( metric)
    axes.set_title("Validation Curves")

    if zoom:
        y_lower = int( 0.9 * np.amin([train_lower[-1], test_lower[-1]]))
        y_upper = int( 1.1 * np.amax([train_upper[-1], test_upper[-1]]))
        axes.set_ylim((y_lower, y_upper))
```

(continues on next page)

(continued from previous page)

```
    plt.show()
    plt.close()
#
```

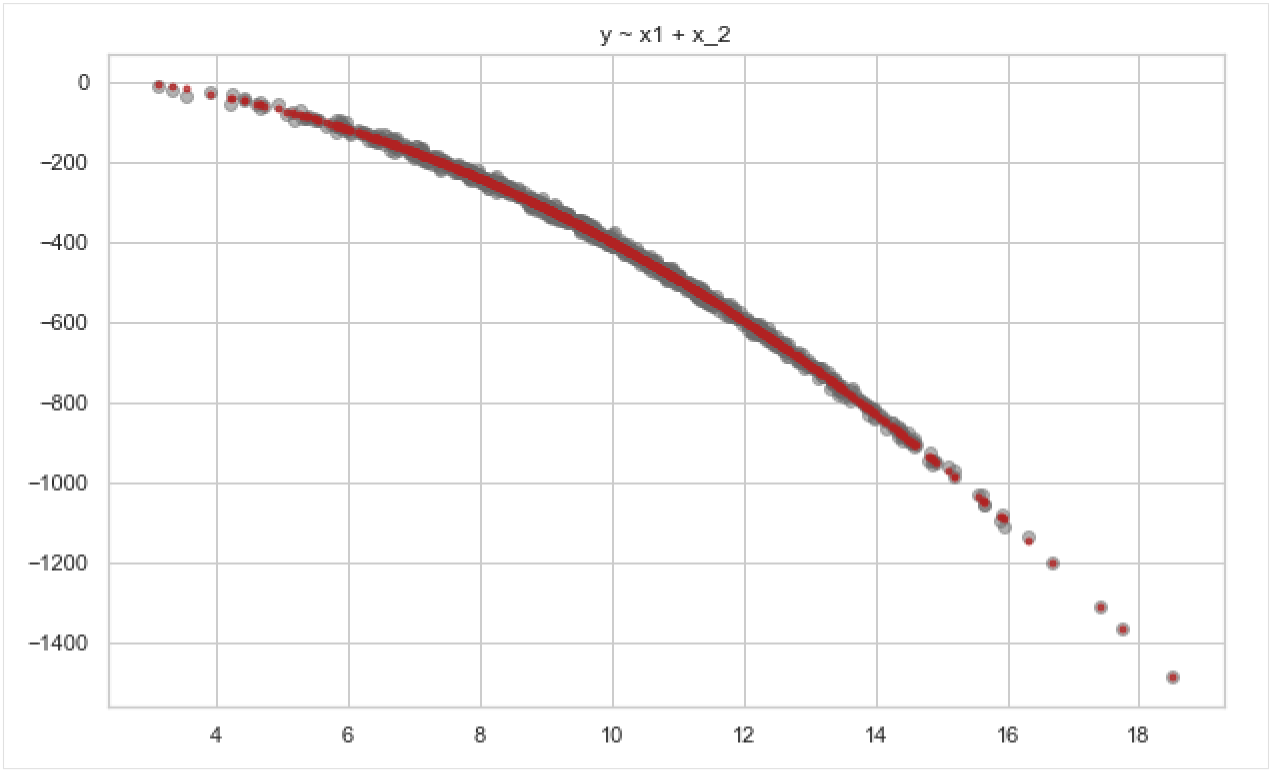
```
[32]: plot_validation_curves(result, "SSE", r"$x^i$", values)
```



You are rarely going to see Validation Curves that look that good. Most of the time they're very noisy. What you are looking for is the value of the parameter where the metric is optimized (lowest for error, highest for accuracy) on the *test* curve.

In any case, let's apply our new found knowledge to our model:

```
[33]: result = models.linear_regression( "y ~ x1 + x_2", data=data)

figure = plt.figure(figsize=(10,6))

axes = figure.add_subplot(1, 1, 1)

xs = data[ "x1"]
ys = data[ "y"]
axes.scatter( xs, ys, color="dimgray", alpha=0.5)
betas = result[ "coefficients"]
axes.plot(xs, [betas[ 0] + betas[ 1] * x + betas[ 2] * x**2 for x in xs], '.', color=
↪"firebrick", alpha=0.75)
axes.set_title( result[ "formula"])

plt.show()
plt.close()
```



Of course, with *real* data it isn't going to be this easy or nice. Ideally, there will be a cycle between learning curves, validation curves, cross validation as you build models, learn what to do, evaluate them, improve them, repeat.

3.7 Model Comparison

There is one possibility that we haven't talked about: switching models. "Switching models" is kind of ambiguous. What do I mean?

1. Switching from $\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$ to $\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$ or
2. Switching from linear regression to a neural network.

The answer is *either*. The first we can call "changing inputs" and the second we'll call "changing models".

3.7.1 Changing Inputs

From a statistical point of view, when we looked at linear regression, we saw that we had R^2 and σ to guide us in terms of better models. From a machine learning point of view, though, we have cross validation and a variety of evaluation metrics. Since we are only interested in predictive power, the better model is the one with a *difference* in MSE (for example) that is larger, *meaningful* and *credible*. Sound familiar?

We've already seen how to approach a problem like this. We just use the cross-validation data to perform statistical inference using the non-parameteric Bayesian bootstrap. You can also use Grid Search to determine which of the different sets of inputs might be "high bias" or "high variance".

3.7.2 Changing Models

This is also ambiguous. We could either change F or change f, although the latter usually implies the former. For example, we may find that we really want a Decision Tree (f) but perhaps we should use C4.5 instead of CART (use a different F).

How you generally proceed will depend on a number of factors.

First, not all models can accept all variable types. Some algorithms need to have categorical variables encoded as binary values and some do not. Otherwise need numeric variables. These will be additional transformations aside any re-expression you might want to do later. This may put a new load on ETL infrastructure, timelines, etc. Patterns that exist in under one transformation (numeric) may not exist under another transformation (ordered categories). You will probably need to repeat your EDA.

Second, how is the model being used? If you're just running the model yourself, either on-demand or using a cron job, you are probably free to do as you like and to add whatever libraries you like. However, if you are building models that need to be "put into production" you may need to coordinate with other engineers to see if you can deploy the model you plan to build. This can get messy. Is there code to generate run algorithm X in language Y? If not, does your engineering department know how to support R code in production? Can you write code to evaluate the model in language Y and simply export the parameters from R? The answers to these kinds of questions fall in the general purview of "data products".

Third, you may very well find yourself in the same boat with respect to performance after you've taken the time translate all your data into a form another learning algorithm can handle. Generally speaking, your time may often be better spent looking for better features than different algorithms.

How all of these factors play–and whether you take them into account from the start–determine your options and process. If you are limited in the kinds of models you can use, you'll follow the flow described here more closely. Changing models only when absolutely necessary. If you have a lot of modeling options and plan ahead, you may already have variables and transformations of variables available to try a wide variety of models.

Taking all of this into account, you may still feel that a regression tree is better than linear regression for your problem, how do you decide which is better?

Assuming you've made all the necessary transformations and performed cross-validation, we're back to a familiar problem: is the difference meaningful and credible given the data? We already know how to answer that. Whether or not the difference is meaningful depends on your particular situation and the input of domain experts and stakeholders (all of which might be you but don't presume they're you).

For example, you might have two models with an apparent difference in MSE is worth changing ETL scripts, retooling and asking engineers to spend four weeks re-writing, testing and deploying code and *means something to the organization* usually by increasing efficiency, increasing revenue or decreasing costs. Alternatively, the result might just end up in a report.

In either case, there is a statistic of interest (the difference in mean squared error) that you want to infer and you already have the tools to solve that problem whether it is the difference in mean squared error (difference of two values) or difference in accuracy (difference of two proportions). Use the Bayes, Luke.

3.8 A/B Testing

There's only so much you can do with cross validation and learning curves to test the generalization performance of your latest "superfantastic" model. Additionally, there are some approaches for which backtesting is either infeasible or impossible. At some point you're going to have to let the model out into the wild and see what it can really do. But should you just shelve your current model? Should you unleash this superfantastic model on all your customers? Probably not. Instead you should conduct an A/B test.

An A/B test is not unlike the process used in medicine for testing new treatments. In the typical medical situation, there's a current treatment and standards of care. A company might come up with a new drug. Doctors don't simply start prescribing the new drug. Instead, there are randomized clinical trials. A drug company will recruit a number of patients and randomly assign them to two groups: the "control" group and the "treatment" group. Control will get the existing treatment (or a placebo if there is no current treatment) and Treatment will get the new drug. At the end of the trial, the statistics are calculated (you do have an evaluation metric, right?) and then we're back in familiar territory: do we have a *meaningful* difference (in medicine, defined as a clinically significant result) and do we have a believable difference–we do statistical inference.

We can do the same thing for, really, any process change. Suppose we send out a marketing letter every year to raise funds for our non-profit. This year we want to see if adding a hand written note increases the response rate and the contribution amount. We can take our donor list, randomize it and send one-half the regular fundraising letter and the other half the fundraising letter with the hand written note. We can tally the results and see if the differences are meaningful and believable.

I bring up this entirely non-computer example because this might be a situation that you run into when doing data science. The actual process may be entirely manual, old-school snail mail but you will be asked to structure and evaluate the experiment.

In general, however, we are more likely to be working with new and improved models and A/B testing these models often requires a complicated infrastructure. Suppose we are send our customers emails every month to let them know what's on sale. We currently pick the 20 most popular items but we have a good idea that if we could personalize the email to cover the kinds of things each customer is more likely to buy, we will generate more revenue. We spend a few months developing a personalization model and decide we want to A/B test it.

Before we can even do that, we need an infrastructure that:

1. keeps track of every email sent and whether or not that person opened it (email services can do this; you may need to download the information through their API).
2. assigns each customer into either control or treatment (A or B) and keeps track of it.
3. can tie an email address to a customer purchase (easy on a website, not as easy in a brick-and-mortar store).
4. generates the personalized content and keeps track of it (either in advance or on-the-fly depending on the scale and resources).
5. monitor our services so that we know that emails are going out and that personalized content is being generated and used.
6. collects results.

From an experimental design perspective, we have a few complications and options. If we are applying Frequentist inference, we have quite a few guidelines that we *must* follow for the test to be accurate.

1. We must, of course, set a level of significance ahead of time.
2. We must determine the differences in purchase rates and revenue that are meaningful to us; generally, differences that make the development, deployment and maintenance of the model and infrastructure cost effective.
3. Given the minimum difference in purchase rates ("lift") we want to detect, select an appropriate sample size: how many emails do we need to send out before we can detect our minimum lift?
4. We should not, generally, peek at the results (we should not do a test of statistical significance) before the experiment is over. We may need to monitor our test, though, to make sure emails are being sent, etc.
5. We should not, generally, stop the test early because we see favorable results.

There is a "generally" in the last few points because it is possible to build the ability to peek and stop early into your experiment when using Frequentist inference. However, you *must* do it ahead of time. That is, if you find you need 10,000 samples to detect a minimum 5% lift and that it will take four weeks to obtain that many samples, you must build into your test plan the fact that you will “peek” at one week, two weeks, three weeks and, finally, a four weeks. This peeking generally includes having a stronger criteria for statistical significance known as “early stopping”.

Why? If you peek at your data at one week and it is statistically significant, how do you know that the data wouldn’t turn the other way in another week?

This is actually a problem in clinical trials as well. What if everyone in Group B is healed completely and everyone in Group A (the original treatment) is still dying? Do you wait until the test is over? In general, there are two ways that medical research handle this problem. First, they include “early stopping” criteria in their experimental design or they use Bayesian inference from the start.

And that brings us to Bayesian inference. Under Bayesian inference, you still need to establish meaningful differences (lift) but you can use the estimate of the Posterior distribution at any time to evaluate the believability of your results. If you stop at 10 observations or 10,000 observations that is simply your responsibility. This is kind of daunting actually.

One very effective practice is to be running A/A tests all the time for the situations where you might apply models. An A/A test is pretty straight forward. You do assign people at random to two groups–but you give them both the control! As you analyze this data as if it were an A/B test, you can test your infrastructure for correctness and you can also assess the size of the lift you generally see in that situation. You can then use this as a prior.

A/B Testing brings in a whole new world of concerns.

One of these is data leakages. Your current data on people buying from emails is *biased* based on the way the current email is constructed. For example, if your email includes the 20 most popular items for your store, then the items people see and the purchases they make from the email are biased by that. If you go to use this data for *backtesting* this has an inherent bias towards popularity. If you tried to develop and backtest a model that was based on personal preferences, it might not work as well simply because it has to work against the bias in the data.

The only way around this sort of bias is to pick a small segment of your users and send them a *random* product listing in their email and only use this data for backtesting. (Don’t keep those customers in the “random” group from month to month if you want to keep them).

You also have problems with *attribution*. If you send a weekly email and I see Product A in the email for week 1, and I see it in the email for week 2, and I buy it in week 3. . . which email caused me to buy Product A? Maybe it was both emails? There isn’t usually a right or wrong answer, but you must be consistent.

You need to also worry about overlapping and interacting tests. It is usually not a problem if someone is in an email test group and a site test group. . . but if someone is in a green/blue button test group *and* a dark/light theme test group, that might be a problem.

You also are up against something like the placebo effect called “novelty effects”. A treatment might have a significant lift simply because it is new. If you need to run a four week A/B test, and you find that B is better after four weeks, you should consider running a second test.

Finally, you must make sure that your groups do not contain biases. Consider what happens if Group A contains all women and Group B contains all men? Or if Group A contains established clients and Group B contains new acquisitions? As R. A. Fisher was fond of repeating, the single most important thing you can do is “Randomize! Randomize! Randomize!”.

In recent years, *Bandit Optimization* has arisen as a possible alternative to A/B testing. In Bandit optimization (which is related to *reinforcement learning*), you assign a model (control or treatment) to a user based on the probability of the model is any good, subject to a minimum. How does that work?

First, let’s start with control and treatment. We assign them to users with a 50/50 probability. As results come in, we can adjust those probabilities based on the purchases. If there are more purchases with B, the probability of assigning B increases and the probability of assigning A decreases. If B starts to lose against A, the probability of assigning B starts to decrease. There is usually a floor of some kind, say, five percent.

Now, when we introduce a new model, we renormalize our probabilities to include the new model at, say, 10% so we have A with a 85% probability, B with a 5% probability and C with a 10% probability. If C is really better than A or B, the probability of assigning C will go up.

What does this give you? If it turns out that some fluctuating circumstance made B look better than A, as that circumstance passes, the probability of picking B will go down because it will stop being better. This takes care of the novelty effects. Additionally, there might be seasonal effects. Maybe B did well in the Spring and when Spring ends, the probability of picking B goes down and when Spring returns, the probability of picking B goes up again. All without human intervention.

The downsides to Bandit Optimization is that you need a fairly large amount of traffic for it to work and it doesn't necessarily pick a clear winner. If you need a clear winner, use A/B testing.

3.9 Conclusion

Model evaluation takes on an entirely new dimension for the Data Scientist. At some point, someone will say to you, "we want to increase engagement" and that's fine, except that we need to agree on what that means in a very concrete way. Does it mean time on the page or number of pages visited? You need to see if that data is being collected and if it can be calculated. This should have been part of the original CoNVO or subsequent improvements to the understanding of the problem. It may have been missed.

Let us take an example from the very first module, you have a news site and you want people to read the articles. One way you think you can do that is to build a model that scores each story for each individual, producing a propensity to read the story. This is a complex classification/information retrieval problem. So when building the model, you might conduct some cross validation studies to see how well the model performs.

But "reading propensity" may not be the metric you're trying to optimize. So in the A/B Test, the performance metric diverges from the loss function and performance metric of the model. Your organization may have a whole slate of engagement scores. Your model was not built to optimize all of those and in a dynamic context.

Put differently, there's a difference between the estimation of response variables (and associated loss functions) and evaluation metrics. The response variable of your model may be the probability that someone buys a certain product. The evaluation metric of the model may be purchase rates. These are not the same thing.

A/B Testing may ultimately be used to test what is more important and only validates the model in a very indirect sort of way.

3.9.1 Summary

Here are a few take away points:

1. High Bias/High Variance are not absolutes. There is no algorithm that produces models that are always high bias. High bias or variance exist in context with the inputs, metaparameters, and data.
2. There is a lot of trial and error in "debugging" a machine learning model. It's a very large search space of possibilities. Use EDA to inform your search. We knew to try a quadratic transformation on x_1 because we'd looked at the data. Note that some companies, DataRobot for example, are trying to automate the process by automatically searching the grid of possibilities using huge clusters in the cloud. You might be able to accomplish something similar on your own.
3. Real life curves–learning or validation–rarely look as neat as the textbook versions.

3.9.2 Review

1. Describe the two cultures of model evaluation. How are they different?
2. What is the most typical evaluation metric for regression (value prediction) problems? What is the formula?
3. What is a confusion matrix? What are each of the elements in the matrix?

4. Define and provide the formula for:
5. accuracy
6. error rate
7. sensitivity/true positive rate
8. specificity
9. precision
10. How does 10 fold cross validation work? How would you apply it to linear or logistic regression?
11. What is the bias/variance tradeoff?
12. How does bias relate to underfitting?
13. How does variance relate to overfitting?
14. What are the five general ways in which a “model” can be improved?
15. Describe how to calculate learning curves. What do they tell you about your model?
16. What do learning curves look like when your current situation involves high bias?
17. What do learning curves look like when your current situation involves high variance?
18. If your learning curves indicate high bias, what does that suggest about getting more data? What should you do?
19. If your learning curves indicate high variance, what does that suggest about getting more data? What should you do?
20. What are validation curves used for? How do you interpret them?
21. What are some other ways you can improve your model under high bias/high variance?
22. What is regularization?
23. What is the difference between “backtesting” (cross validation) and A/B testing?