UNIT V

PREDICTIVE ANALYTICS

Predictive analytics is a branch of advanced analytics that makes predictions about future outcomes using historical data combined with statistical modeling, data mining techniques and machine learning. Companies employ predictive analytics to find patterns in this data to identify risks and opportunities.

Predictive analytics models may be able to identify correlations between sensor readings. For example, if the temperature reading on a machine correlates to the length of time it runs on high power, those two combined readings may put the machine at risk of downtime.

There are three types of predictive analytics techniques: predictive models, descriptive models, and decision models.

Linear least Squares

Correlation coefficients measure the strength and sign of a relationship, but not the slope. There are several ways to estimate the slope; the most common is a linear least squares fit. A “linear fit” is a line intended to model the relationship between variables. A “least squares” fit is one that minimizes the mean squared error (MSE) between the line and the data.

Suppose we have a sequence of points, ys, that we want to express as a function of another sequence xs. If there is a

linear relationship between xs and ys with intercept inter and slope slope, we expect each y[i] to be inter + slope * x[i].

But unless the correlation is perfect, this prediction is only approximate. The vertical deviation from the line, or residual, is

```
res = ys - (inter + slope * xs)
```

The residuals might be due to random factors like measurement error, or nonrandom factors that are unknown. For example, if we are trying to predict weight as a function of height, unknown factors might include diet, exercise, and body type.

If we get the parameters inter and slope wrong, the residuals get bigger, so it makes intuitive sense that the parameters we want are the ones that minimize the residuals.

We might try to minimize the absolute value of the residuals, or their squares, or their cubes; but the most common choice is to minimize the sum of squared residuals, sum(res**2)).

Why? There are three good reasons and one less important one:

- Squaring has the feature of treating positive and negative residuals the same, which is usually what we want.
- Squaring gives more weight to large residuals, but not so much weight that the largest residual always dominates.

- If the residuals are uncorrelated and normally distributed with mean 0 and constant (but unknown) variance, then the least squares fit is also the maximum likelihood estimator of inter and slope.
- The values of inter and slope that minimize the squared residuals can be computed efficiently.

The last reason made sense when computational efficiency was more important than choosing the method most appropriate to the problem at hand. That's no longer the case, so it is worth considering whether squared residuals are the right thing to mini-mize.

For example, if you are using xs to predict values of ys, guessing too high might be better (or worse) than guessing too low. In that case you might want to compute some cost function for each residual, and minimize total cost, sum(cost(res)). However, computing a least squares fit is quick, easy and often good enough

Implementation

thinkstats2 provides simple functions that demonstrate linear least squares:

```
def LeastSquares(xs, ys):
    meanx, varx = MeanVar(xs)
    meany = Mean(ys)
```

```
    slope = Cov(xs, ys, meanx, meany) / varx
    inter = meany - slope * meanx

    return inter, slope
```

LeastSquares takes sequences xs and ys and returns the estimated parameters inter and slope. For details on how it works, see Wikipedia.

thinkstats2 also provides FitLine, which takes inter and slope and returns the fitted line for a sequence of xs.

```
def FitLine(xs, inter, slope):
    fit_xs = np.sort(xs)
    fit_ys = inter + slope * fit_xs
    return fit_xs, fit_ys
```

We can use these functions to compute the least squares fit for birth weight as a function of mother's age.

```
live, firsts, others = first.MakeFrames()
live = live.dropna(subset=['agepreg', 'totalwgt_lb'])
ages = live.agepreg
weights = live.totalwgt_lb

inter, slope = thinkstats2.LeastSquares(ages, weights)
```

```
fit_xs, fit_ys = thinkstats2.FitLine(ages, inter, slope)
```

The estimated intercept and slope are 6.8 lbs and 0.017 lbs per year. These values are hard to interpret in this form: the intercept is the expected weight of a baby whose mother is 0 years old, which doesn't make sense in context, and the slope is too small to grasp easily.

Instead of presenting the intercept at x = 0, it is often helpful to present the intercept at the mean of x. In this case the mean age is about 25 years and the mean baby weight for a 25-year-old mother is 7.3 pounds. The slope is 0.27 ounces per year, or 0.17 pounds per decade.

Figure 10-1 shows a scatter plot of birth weight and age along with the fitted line. It's a good idea to look at a figure like this to assess whether the relationship is linear and whether the fitted line seems like a good model of the relationship.

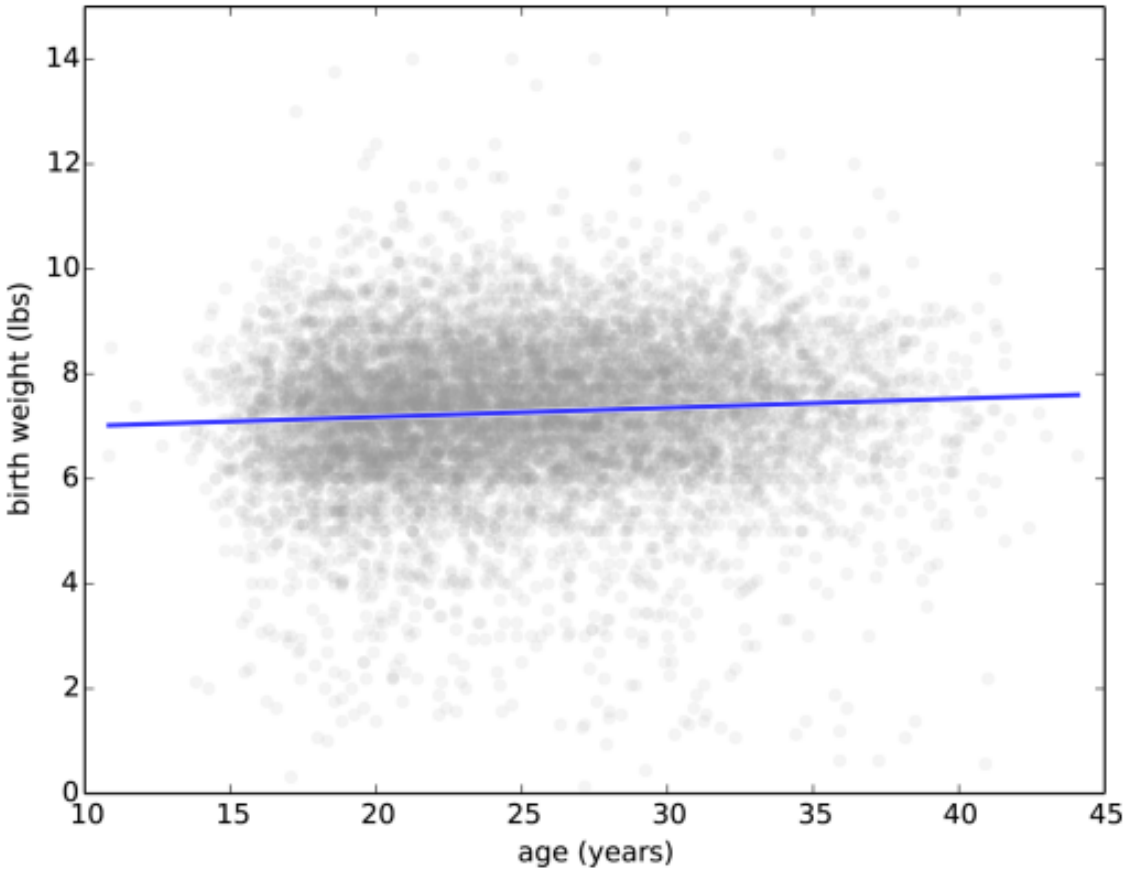

Figure 10-1. Scatter plot of birth weight and mother's age with a linear fit

Residuals

Another useful test is to plot the residuals. thinkstats2 provides a function that computes residuals:

```
def  Residuals(xs, ys, inter, slope):
        xs = np.asarray(xs)
        ys = np.asarray(ys)
        res = ys - (inter + slope * xs)
        return res
```

Residuals takes sequences xs and ys and estimated parameters inter and slope. It returns the differences between the actual values and the fitted line.

Figure 10-2 shows the 25th, 50th and 75th percentiles of the residuals for each age group. The median is near zero, as expected, and the interquartile range is about 2 pounds. So if we know the mother's age, we can guess the baby's weight within a pound, about 50% of the time.

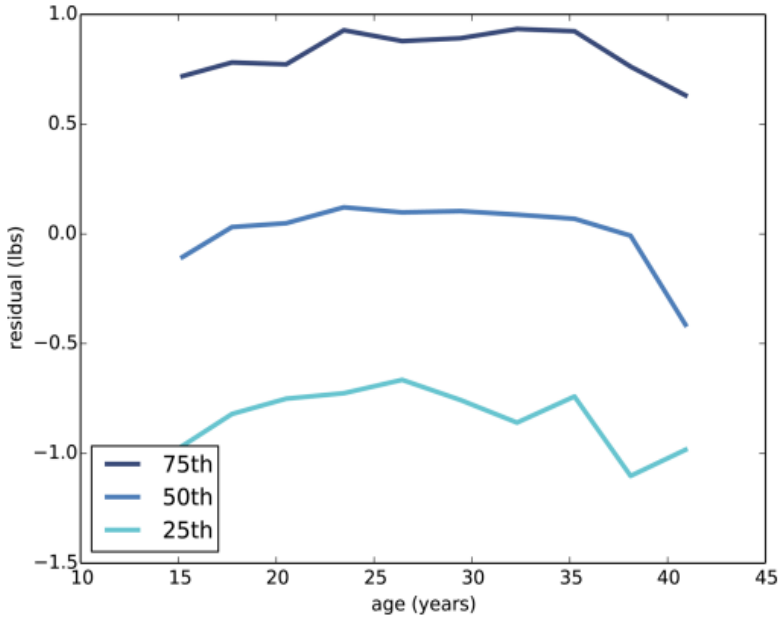

Figure 10-2. Residuals of the linear fit

Ideally these lines should be flat, indicating that the residuals are random, and parallel, indicating that the variance of the residuals is the same for all age groups. In fact, the lines are close to parallel, so that's good; but they have some curvature, indicating that the relationship is non-linear. Nevertheless, the linear fit is a simple model that is probably good enough for some purposes.

Goodness of Fit

There are several ways to measure the quality of a linear model, or goodness of fit. One of the simplest is the standard deviation of the residuals.

If you use a linear model to make predictions, Std(res) is the root mean squared error (RMSE) of your predictions. For example, if you use mother's age to guess birth weight,the RMSE of your guess would be 1.40 lbs.

If you guess birth weight without knowing the mother's age, the RMSE of your guess is Std(ys), which is 1.41 lbs. So in this example, knowing a mother's age does not improve the predictions substantially.

Another way to measure goodness of fit is the coefficient of determination, usually denoted R^2 and called "R-squared":

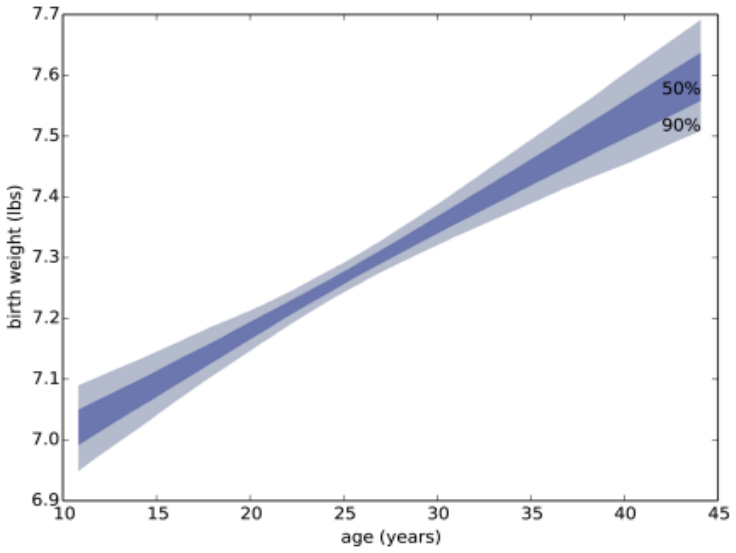

Figure 10-3. 50% and 90% confidence intervals showing variability in the fitted line due to sampling error of inter and slope

```
def CoefDetermination(ys, res):
```

return 1 - Var(res) / Var(ys)

Var(res) is the MSE of your guesses using the model, Var(ys) is the MSE without it. So their ratio is the fraction of MSE that remains if you use the model, and R 2 is the fraction of MSE the model eliminates.

For birth weight and mother's age, R^2 is 0.0047, which means that mother's age predicts about half of 1% of variance in birth weight.

There is a simple relationship between the coefficient of determination and Pearson's coefficient of correlation: $R^2 = \rho^2$. For example, if ρ is 0.8 or -0.8, R 2 = 0.64.

Although ρ and R^2 are often used to quantify the strength of a relationship, they are not easy to interpret in terms of predictive power. In my opinion, Std(res) is the best representation of the quality of prediction, especially if it is presented in relation to Std(ys).

For example, when people talk about the validity of the SAT (a standardized test used for college admission in the U.S.) they often talk about correlations between SAT scores and other measures of intelligence.

According to one study, there is a Pearson correlation of ρ = 0.72 between total SAT scores and IQ scores, which sounds like a strong correlation. But $R^2 = \rho^2 = 0.52$, so SAT scores account for only 52% of variance in IQ.

IQ scores are normalized with Std(ys) = 15, so

```
>>> var_ys = 15**2
>>> rho = 0.72
>>> r2 = rho**2
>>> var_res = (1 - r2) * var_ys
>>> std_res = math.sqrt(var_res)
10.4096
```

So using SAT score to predict IQ reduces RMSE from 15 points to 10.4 points. A correlation of 0.72 yields a reduction in RMSE of only 31%.

If you see a correlation that looks impressive, remember that R^2 is a better indicator of reduction in MSE, and reduction in RMSE is a better indicator of predictive power.

Testing a Linear Model

The effect of mother's age on birth weight is small, and has little predictive power. So is it possible that the apparent relationship is due to chance? There are several ways we might test the results of a linear fit.

One option is to test whether the apparent reduction in MSE is due to chance. In that case, the test statistic is R^2 and the null hypothesis is that there is no relationship between the variables. We can simulate the null hypothesis by permutation, as in "Testing a Correlation", when we tested the correlation between mother's age and birth weight. In fact, because $R^2 = \rho^2$, a one-sided test of R^2 is equivalent to a

two-sided test of ρ. We've already done that test, and found $p < 0.001$, so we conclude that the apparent relationship between mother's age and birth weight is statistically significant.

Another approach is to test whether the apparent slope is due to chance. The null hypothesis is that the slope is actually zero; in that case we can model the birth weights as random variations around their mean. Here's a HypothesisTest for this model:

```
class SlopeTest(thinkstats2.HypothesisTest):

    def TestStatistic(self, data):

        ages, weights = data

        _, slope = thinkstats2.LeastSquares(ages, weights)

        return slope

    def MakeModel(self):

        _, weights = self.data

        self.ybar = weights.mean()

        self.res = weights - self.ybar

    def RunModel(self):

        ages, _ = self.data

        weights = self.ybar + np.random.permutation(self.res)

        return ages, weights
```

The data are represented as sequences of ages and weights. The test statistic is the slope estimated by LeastSquares. The model of the null hypothesis is represented by the mean weight of all babies and the deviations from the mean. To

generate simulated data, we permute the deviations and add them to the mean.

Here's the code that runs the hypothesis test:

```
live, firsts, others = first.MakeFrames()
live = live.dropna(subset=['agepreg', 'totalwgt_lb'])
ht = SlopeTest((live.agepreg, live.totalwgt_lb))
pvalue = ht.PValue()
```

The p-value is less than 0.001, so although the estimated slope is small, it is unlikely to be due to chance.

Estimating the p-value by simulating the null hypothesis is strictly correct, but there is a simpler alternative. Remember that we already computed the sampling distribution of the slope, in "Estimation". To do that, we assumed that the observed slope was correct and simulated experiments by resampling.

Figure 10-4 shows the sampling distribution of the slope, from "Estimation", and the distribution of slopes generated under the null hypothesis. The sampling distribution is centered about the estimated slope, 0.017 lbs/year, and the slopes under the null hypothesis are centered around 0; but other than that, the distributions are identical. The

distributions are also symmetric, for reasons we will see in "Central Limit Theorem".

So we could estimate the p-value two ways:

- Compute the probability that the slope under the null hypothesis exceeds the observed slope.

- Compute the probability that the slope in the sampling distribution falls below 0.(If the estimated slope were negative, we would compute the probability that the slope in the sampling distribution exceeds 0.)

The second option is easier because we normally want to compute the sampling distribution of the parameters anyway. And it is a good approximation unless the sample size is small and the distribution of residuals is skewed. Even then, it is usually good enough,because p-values don't have to be precise.

Here's the code that estimates the p-value of the slope using the sampling distribution:

```
inters, slopes = SamplingDistributions(live, iters=1001)

slope_cdf = thinkstats2.Cdf(slopes)

pvalue = slope_cdf[0]
```

Again, we find $p < 0.001$.

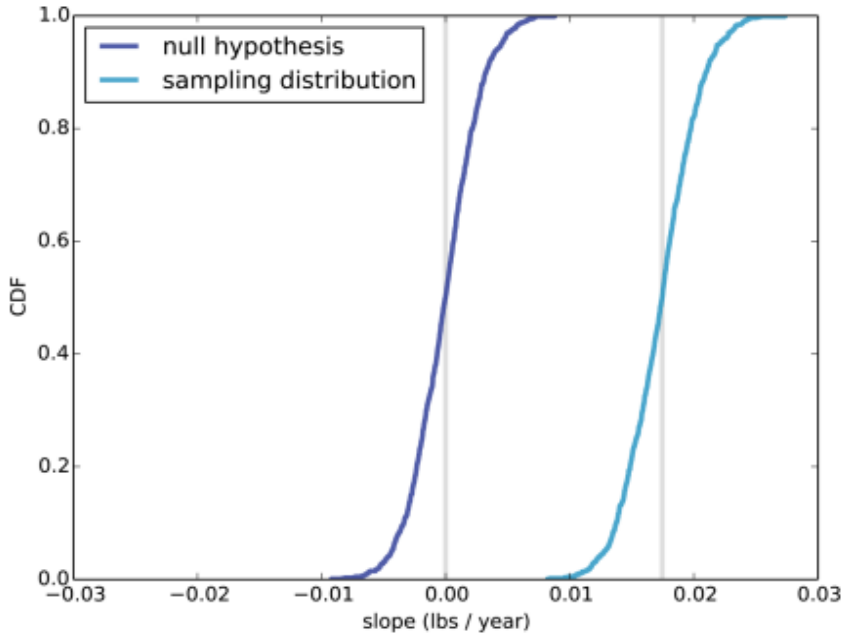

Figure 10-4. The sampling distribution of the estimated slope and the distribution of slopes generated under the null hypothesis. The vertical lines are at 0 and the observed slope, 0.017 lbs/year.

Weighted Resampling

So far we have treated the NSFG data as if it were a representative sample, but as I mentioned in “The National Survey of Family Growth” it is not. The survey deliberately oversamples several groups in order to improve the chance of getting statistically significant results; that is, in order to improve the power of tests involving these groups.

This survey design is useful for many purposes, but it means that we cannot use the sample to estimate values for the

general population without accounting for the sampling process.

For each respondent, the NSFG data includes a variable called finalwgt, which is the number of people in the general population the respondent represents. This value is called a **sampling weight**, or just "weight."

As an example, if you survey 100,000 people in a country of 300 million, each respondent represents 3,000 people. If you oversample one group by a factor of 2, each person in the oversampled group would have a lower weight, about 1500.

To correct for oversampling, we can use resampling; that is, we can draw samples from the survey using probabilities proportional to sampling weights. Then, for any quantity we want to estimate, we can generate sampling distributions, standard errors, and confidence intervals. As an example, I will estimate mean birth weight with and without sampling weights.

In "Estimation", we saw ResampleRows, which chooses rows from a DataFrame, giving each row the same probability. Now we need to do the same thing using probabilities proportional to sampling weights. ResampleRowsWeighted takes a DataFrame, resamples rows according to the weights in finalwgt, and returns a DataFrame containing the resampled rows:

```
def ResampleRowsWeighted(df):
    weights = df.finalwgt
    pmf = thinkstats2.Pmf(weights.iteritems())
```

```
    cdf = pmf.MakeCdf()
    indices = cdf.Sample(len(weights))
    sample = df.loc[indices]
    return sample
```

pmf maps from each row index to its normalized weight. Converting to a Cdf makes the sampling process faster. Indices is a sequence of row indices; sample is a DataFrame that contains the selected rows. Since we sample with replacement, the same row might appear more than once.

Now we can compare the effect of resampling with and without weights. Without weights, we generate the sampling distribution like this:

```
estimates = [ResampleRows(live).totalwgt_lb.mean()
                for _ in range(iters)]
```

With weights, it looks like this:

```
estimates=[ResampleRowsWeighted(live).totalwgt_lb.mean()
            for _ in range(iters)]
```

The following table summarizes the results:

	Mean birth weight (lbs)	Standard error	90% CI
Unweighted	7.27	0.014	(7.24, 7.29)
Weighted	7.35	0.014	(7.32, 7.37)

In this example, the effect of weighting is small but non-negligible. The difference in estimated means, with and without weighting, is about 0.08 pounds, or 1.3 ounces. This difference is substantially larger than the standard error of the estimate, 0.014 pounds, which implies that the difference is not due to chance.

Regression

The linear least squares fit in the previous topic is an example of **regression**, which is the more general problem of fitting any kind of model to any kind of data. This use of the term "regression" is a historical accident; it is only indirectly related to the original meaning of the word.

The goal of regression analysis is to describe the relationship between one set of variables, called the **dependent variables**, and another set of variables, called independent or **explanatory variables**.

In the previous chapter we used mother's age as an explanatory variable to predict birth weight as a dependent variable. When there is only one dependent and one explanatory variable, that's **simple regression**. In this chapter, we move on to **multiple regression**, with more than one explanatory variable. If there is more than one dependent variable, that's multivariate regression.

If the relationship between the dependent and explanatory variable is linear, that's **linear regression**. For example, if the dependent variable is y and the explanatory variables are x_1 and x_2 ,we would write the following linear regression model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$$

where β0 is the intercept, β_1 is the parameter associated with x_1 , β_2 is the parameter associated with x_2 and ε is the residual due to random variation or other unknown factors.

Given a sequence of values for y and sequences for x_1 and x_2, we can find the parameters, β_0, β_1, and β_2, that minimize the sum of ε^2. This process is called **ordinary least squares**. The computation is similar to that of thinkstats2. LeastSquare, but generalized to deal with more than one explanatory variable.

The code for this is in regression.py.

StatsModels

In the previous chapter I presented thinkstats2.LeastSquares, an implementation of simple linear regression intended to be easy to read. For multiple regression we'll switch to StatsModels, a Python package that provides several forms of regression and other analyses. If you are using Anaconda, you already have StatsModels; otherwise you might have to install it.

As an example, I'll run the model from the previous chapter with StatsModels:

```
import statsmodels.formula.api as smf


live, firsts, others = first.MakeFrames()
formula = 'totalwgt_lb ~ agepreg'
model = smf.ols(formula, data=live)
```

```
results = model.fit()
```

statsmodels provides two interfaces (APIs); the “formula” API uses strings to identify the dependent and explanatory variables. It uses a syntax called patsy; in this example, the ~ operator separates the dependent variable on the left from the explanatory variables on the right.

smf.ols takes the formula string and the DataFrame, live, and returns an OLS object that represents the model. The name ols stands for “ordinary least squares.”

The fit method fits the model to the data and returns a RegressionResults object that contains the results.

The results are also available as attributes. params is a Series that maps from variable names to their parameters, so we can get the intercept and slope like this:

```
inter = results.params['Intercept']

slope = results.params['agepreg']
```

The estimated parameters are 6.83 and 0.0175, the same as from LeastSquares.

pvalues is a Series that maps from variable names to the associated p-values, so we can check whether the estimated slope is statistically significant:

```
slope_pvalue = results.pvalues['agepreg']
```

The p-value associated with agepreg is 5.7e-11, which is less than 0.001, as expected.

results.rsquared contains R^2, which is 0.0047. results also provides f_pvalue, which is the p-value associated with the

model as a whole, similar to testing whether R^2 is statistically significant.

And results provides resid, a sequence of residuals, and fittedvalues, a sequence of fitted values corresponding to agepreg.

The results object provides summary(), which represents the results in a readable format.

```
print(results.summary())
```

But it prints a lot of information that is not relevant (yet), so I use a simpler function called SummarizeResults. Here are the results of this model:

```
Intercept       6.83 (0)
agepreg         0.0175 (5.72e-11)
R^2 0.004738
Std(ys) 1.408
Std(res) 1.405
```

Std(ys) is the standard deviation of the dependent variable, which is the RMSE if you have to guess birth weights without the benefit of any explanatory variables. Std(res) is the standard deviation of the residuals, which is the RMSE if your guesses are informed by the mother's age. As we have already seen, knowing the mother's age provides no substantial improvement to the predictions.

Multiple Regression

In “Comparing CDFs”, we saw that first babies tend to be lighter than others, and this effect is statistically significant. But it is a strange result because there is no obvious mechanism that would cause first babies to be lighter. So we might wonder whether this relationship is **spurious**.

In fact, there is a possible explanation for this effect. We have seen that birth weight depends on mother’s age, and we might expect that mothers of first babies are younger than others.

With a few calculations we can check whether this explanation is plausible. Then we’ll use multiple regression to investigate more carefully. First, let’s see how big the difference in weight is:

```
diff_weight = firsts.totalwgt_lb.mean() - others.totalwgt_lb.mean()
```

First babies are 0.125 lbs lighter, or 2 ounces. And the difference in ages:

```
diff_age = firsts.agepreg.mean() - others.agepreg.mean()
```

The mothers of first babies are 3.59 years younger. Running the linear model again, we get the change in birth weight as a function of age:

```
results = smf.ols('totalwgt_lb ~ agepreg', data=live).fit()
slope = results.params['agepreg']
```

The slope is 0.175 pounds per year. If we multiply the slope by the difference in ages, we get the expected difference in birth weight for first babies and others, due to mother’s age:

```
slope * diff_age
```

The result is 0.063, just about half of the observed difference. So we conclude, tentatively, that the observed difference in birth weight can be partly explained by the difference in mother's age.

Using multiple regression, we can explore these relationships more systematically.

```
live['isfirst'] = live.birthord == 1
formula = 'totalwgt_lb ~ isfirst'
results = smf.ols(formula, data=live).fit()
```

The first line creates a new column named isfirst that is True for first babies and false otherwise. Then we fit a model using isfirst as an explanatory variable.

Here are the results:

```
Intercept      7.33 (0)
isfirst[T.True]   -0.125 (2.55e-05)
R^2 0.00196
```

Because isfirst is a boolean, ols treats it as a **categorical variable**, which means that the values fall into categories, like True and False, and should not be treated as numbers. The estimated parameter is the effect on birth weight when isfirst is true, so the result, -0.125 lbs, is the difference in birth weight between first babies and others.

The slope and the intercept are statistically significant, which means that they were unlikely to occur by chance, but the R^2 value for this model is small, which means that isfirst doesn't account for a substantial part of the variation in birth weight.

The results are similar with agepreg:

```
Intercept       6.83 (0)
agepreg       0.0175 (5.72e-11)
R^2 0.004738
```

Again, the parameters are statistically significant, but R^2 is low.

These models confirm results we have already seen. But now we can fit a single model that includes both variables. With the formula totalwgt_lb ~ isfirst + agepreg, we get:

```
Intercept           6.91 (0)
isfirst[T.True]     -0.0698 (0.0253)
agepreg             0.0154 (3.93e-08)
R^2 0.005289
```

In the combined model, the parameter for isfirst is smaller by about half, which means that part of the apparent effect of isfirst is actually accounted for by agepreg. And the p-value for isfirst is about 2.5%, which is on the border of statistical significance.

R^2 for this model is a little higher, which indicates that the two variables together account for more variation in birth weight than either alone (but not by much).

Nonlinear Relationships

Remembering that the contribution of agepreg might be non-linear, we might consider adding a variable to capture more of this relationship. One option is to create a column, agepreg2, that contains the squares of the ages:

```
live['agepreg2'] = live.agepreg**2
formula = 'totalwgt_lb ~ isfirst + agepreg + agepreg2'
```

Now by estimating parameters for agepreg and agepreg2, we are effectively fitting a parabola:

Intercept	5.69	(1.38e-86)
isfirst[T.True]	-0.0504	(0.109)
agepreg	0.112	(3.23e-07)
agepreg2	-0.00185	(8.8e-06)
R^2 0.007462		

The parameter of agepreg2 is negative, so the parabola curves downward, which is consistent with the shape of the lines in Figure 10-2.

The quadratic model of agepreg accounts for more of the variability in birth weight; the parameter for isfirst is smaller in this model, and no longer statistically significant.

Using computed variables like agepreg2 is a common way to fit polynomials and other functions to data. This process is still considered linear regression, because the dependent variable is a linear function of the explanatory variables, regardless of whether some variables are non-linear functions of others.

The following table summarizes the results of these regressions:

	isfirst	agepreg	agepreg2	R^2
Model 1	-0.125 *	–	–	0.002
Model 2	–	0.0175 *	–	0.0047
Model 3	-0.0698 (0.025)	0.0154 *	–	0.0053
Model 4	-0.0504 (0.11)	0.112 *	-0.00185 *	0.0075

The columns in this table are the explanatory variables and the coefficient of determination, R^2. Each entry is an estimated parameter and either a p-value in parentheses or an asterisk to indicate a p-value less that 0.001.

We conclude that the apparent difference in birth weight is explained, at least in part, by the difference in mother's age. When we include mother's age in the model, the effect of isfirst gets smaller, and the remaining effect might be due to chance.

In this example, mother's age acts as a **control variable**; including agepreg in the model "controls for" the difference in age between first-time mothers and others, making it possible to isolate the effect (if any) of isfirst.

Logistic Regression

In the previous examples, some of the explanatory variables were numerical and some categorical (including boolean). But the dependent variable was always numerical.

Linear regression can be generalized to handle other kinds of dependent variables. If the dependent variable is boolean, the generalized model is called **logistic regression**. If the dependent variable is an integer count, it's called **Poisson regression**.

As an example of logistic regression, let's consider a variation on the office pool scenario. Suppose a friend of yours is pregnant and you want to predict whether the baby is a boy or a girl. You could use data from the NSFG to find factors that affect the "sex ratio", which is conventionally defined to be the probability of having a boy.

If you encode the dependent variable numerically, for example 0 for a girl and 1 for a boy, you could apply ordinary least squares, but there would be problems. The linear model might be something like this:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$$

Where y is the dependent variable, and x_1 and x_2 are explanatory variables. Then we could find the parameters that minimize the residuals.

The problem with this approach is that it produces predictions that are hard to interpret. Given estimated

parameters and values for x_1 and x_2, the model might predict y = 0.5, but the only meaningful values of y are 0 and 1.

It is tempting to interpret a result like that as a probability; for example, we might say that a respondent with particular values of x_1 and x_2 has a 50% chance of having a boy. But it is also possible for this model to predict y = 1.1 or y = - 0.1, and those are not valid probabilities.

Logistic regression avoids this problem by expressing predictions in terms of **odds** rather than probabilities. If you are not familiar with odds, "odds in favor" of an event is the ratio of the probability it will occur to the probability that it will not.

So if I think my team has a 75% chance of winning, I would say that the odds in their favor are three to one, because the chance of winning is three times the chance of losing.

Odds and probabilities are different representations of the same information. Given a probability, you can compute the odds like this:

$$o = p / (1-p)$$

Given odds in favor, you can convert to probability like this:

$$p = o / (o+1)$$

Logistic regression is based on the following model:

$$\log o = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$$

Where o is the odds in favor of a particular outcome; in the example, o would be the odds of having a boy.

Suppose we have estimated the parameters β_0, β_1, and β_2. And suppose we are given values for x_1 and x_2. We can compute the predicted value of log o, and then convert to a probability:

```
o = np.exp(log_o)
p = o / (o+1)
```

So in the office pool scenario we could compute the predictive probability of having a boy. But how do we estimate the parameters?

Estimating Parameters

Unlike linear regression, logistic regression does not have a closed form solution, so it is solved by guessing an initial solution and improving it iteratively.

The usual goal is to find the maximum-likelihood estimate (MLE), which is the set of parameters that maximizes the likelihood of the data. For example, suppose we have the following data:

```
>>> y = np.array([0, 1, 0, 1])
>>> x1 = np.array([0, 0, 0, 1])
>>> x2 = np.array([0, 1, 1, 1])
```

And we start with the initial guesses $\beta_0 = -1.5$, $\beta_1 = 2.8$, and $\beta_2 = 1.1$:

```
>>> beta = [-1.5, 2.8, 1.1]
```

Then for each row we can compute log_o:

```
>>> log_o = beta[0] + beta[1] * x1 + beta[2] * x2
```

```
[-1.5 -0.4 -0.4 2.4]
```

And convert from log odds to probabilities:

```
>>> o = np.exp(log_o)
[ 0.223 0.670 0.670 11.02 ]

>>> p = o / (o+1)
[ 0.182 0.401 0.401 0.916 ]
```

Notice that when log_o is greater than 0, o is greater than 1 and p is greater than 0.5.

The likelihood of an outcome is p when y==1 and 1-p when y==0. For example, if we think the probability of a boy is 0.8 and the outcome is a boy, the likelihood is 0.8; if the outcome is a girl, the likelihood is 0.2. We can compute that like this:

```
>>> likes = y * p + (1-y) * (1-p)
[ 0.817 0.401 0.598 0.916 ]
```

The overall likelihood of the data is the product of likes:

```
>>> like = np.prod(likes)
0.18
```

For these values of beta, the likelihood of the data is 0.18. The goal of logistic regression is to find parameters that maximize this likelihood. To do that, most statistics packages use an iterative solver like Newton's method.

Accuracy

In the office pool scenario, we are most interested in the accuracy of the model: the number of successful predictions, compared with what we would expect by chance.

In the NSFG data, there are more boys than girls, so the baseline strategy is to guess “boy” every time. The accuracy of this strategy is just the fraction of boys:

```
actual = endog['boy']
baseline = actual.mean()
```

Since actual is encoded in binary integers, the mean is the fraction of boys, which is 0.507.

Here’s how we compute the accuracy of the model:

```
predict = (results.predict() >= 0.5)
true_pos = predict * actual
true_neg = (1 - predict) * (1 - actual)
```

results.predict returns a NumPy array of probabilities, which we round off to 0 or 1. Multiplying by actual yields 1 if we predict a boy and get it right, 0 otherwise. So, true_pos indicates “true positives”.

Similarly, true_neg indicates the cases where we guess “girl” and get it right. Accuracy is the fraction of correct guesses:

```
acc = (sum(true_pos) + sum(true_neg)) / len(actual)
```

The result is 0.512, slightly better than the baseline, 0.507. But, you should not take this result too seriously. We used the same data to build and test the model, so the model may not have predictive power on new data.

Nevertheless, let’s use the model to make a prediction for the office pool. Suppose your friend is 35 years old and white, her husband is 39, and they are expecting their third child:

```
columns = ['agepreg', 'hpagelb', 'birthord', 'race']
new = pandas.DataFrame([[35, 39, 3, 2]], columns=columns)
```

```
y = results.predict(new)
```

To invoke results.predict for a new case, you have to construct a DataFrame with a column for each variable in the model. The result in this case is 0.52, so you should guess “boy.” But if the model improves your chances of winning, the difference is very small.

Time Series Analysis

A **time series** is a sequence of measurements from a system that varies in time. One famous example is the “hockey stick graph” that shows global average temperature over time.

The example I work with in this chapter comes from Zachary M. Jones, a researcher in political science who studies the black market for cannabis in the US. He collected data from a website called Price of Weed that crowdsources market information by asking participants to report the price, quantity, quality, and location of cannabis transactions. The goal of his project is to investigate the effect of policy decisions, like legalization, on markets. I find this project appealing because it is an example that uses data to address important political questions, like drug policy.

I hope you will find this chapter interesting, but I’ll take this opportunity to reiterate the importance of maintaining a professional attitude to data analysis. Whether and which drugs should be illegal are important and difficult public policy questions; our decisions should be informed by accurate data reported honestly.

The code for this topic is in timeseries.py.

Moving Averages

Most time series analysis is based on the modeling assumption that the observed series is the sum of three components:

Trend

A smooth function that captures persistent changes

Seasonality

Periodic variation, possibly including daily, weekly, monthly, or yearly cycles

Noise

Random variation around the long-term trend

Regression is one way to extract the trend from a series, as we saw in the previous section. But if the trend is not a simple function, a good alternative is a **moving average**. A moving average divides the series into overlapping regions, called **windows**, and computes the average of the values in each window.

One of the simplest moving averages is the **rolling mean**, which computes the mean of the values in each window. For example, if the window size is 3, the rolling mean computes the mean of values 0 through 2, 1 through 3, 2 through 4, etc.

pandas provides rolling_mean, which takes a Series and a window size and returns a new Series.

```
>>> series = np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> pandas.rolling_mean(series, 3)
array([ nan, nan, 1, 2, 3, 4, 5, 6, 7, 8])
```

The first two values are nan; the next value is the mean of the first three elements, 0, 1, and 2. The next value is the mean of 1, 2, and 3. And so on.

Before we can apply rolling_mean to the cannabis data, we have to deal with missing values. There are a few days in the observed interval with no reported transactions for one or more quality categories, and a period in 2013 when data collection was not active.

In the DataFrames we have used so far, these dates are absent; the index skips days with no data. For the analysis that follows, we need to represent this missing data explicitly. We can do that by "reindexing" the DataFrame:

```
dates = pandas.date_range(daily.index.min(), daily.index.max())
reindexed = daily.reindex(dates)
```

The first line computes a date range that includes every day from the beginning to the end of the observed interval. The second line creates a new DataFrame with all of the data from daily, but including rows for all dates, filled with nan.

Now we can plot the rolling mean like this:

```
roll_mean = pandas.rolling_mean(reindexed.ppg, 30)
thinkplot.Plot(roll_mean.index, roll_mean)
```

The window size is 30, so each value in roll_mean is the mean of 30 values from reindexed.ppg.

Figure 12-3 (left) shows the result. The rolling mean seems to do a good job of smoothing out the noise and extracting the trend. The first 29 values are nan, and wherever there's a missing value, it's followed by another 29 nans. There are ways to fill in these gaps, but they are a minor nuisance.

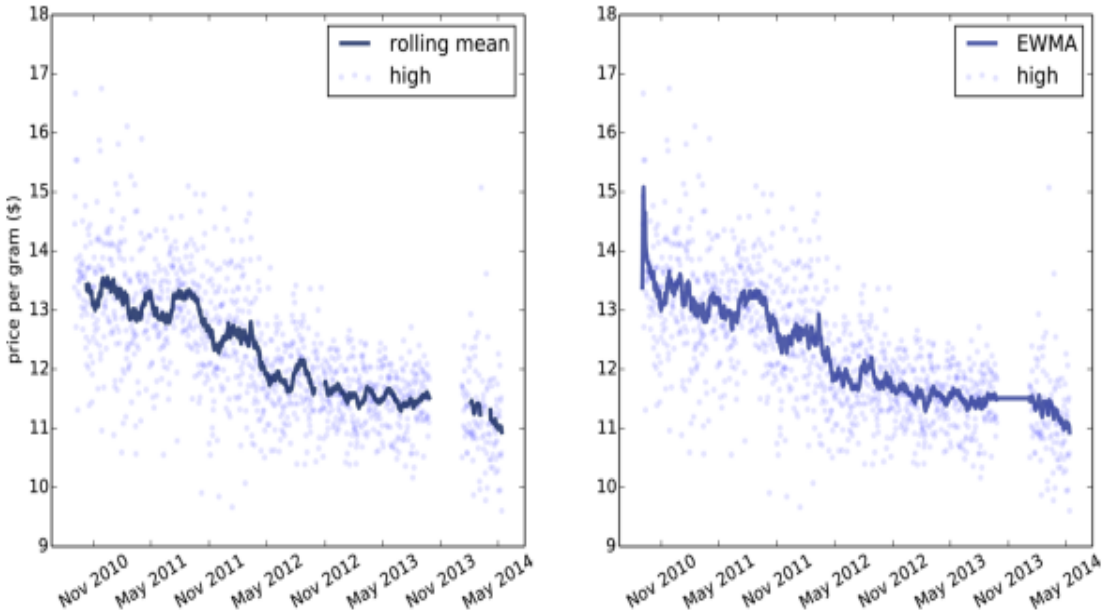

Figure 12-3. Daily price and a rolling mean (left) and exponentially-weighted moving average (right)

An alternative is the **exponentially-weighted moving average** (EWMA), which has two advantages. First, as the name suggests, it computes a weighted average where the most recent value has the highest weight and the weights for previous values drop off exponentially. Second, the pandas implementation of EWMA handles missing values better.

```
ewma = pandas.ewma(reindexed.ppg, span=30)
thinkplot.Plot(ewma.index, ewma)
```

The **span** parameter corresponds roughly to the window size of a moving average; it controls how fast the weights drop off, so it determines the number of points that make a non-negligible contribution to each average.

Figure 12-3 (right) shows the EWMA for the same data. It is similar to the rolling mean, where they are both defined, but it has no missing values, which makes it easier to work with. The values are noisy at the beginning of the time series, because they are based on fewer data points.

Missing Values

Now that we have characterized the trend of the time series, the next step is to investigate seasonality, which is periodic behavior. Time series data based on human behavior often exhibits daily, weekly, monthly, or yearly cycles. In the next section I present methods to test for seasonality, but they don't work well with missing data, so we have to solve that problem first.

A simple and common way to fill missing data is to use a moving average. The Series method `fillna` does just what we want:

```
reindexed.ppg.fillna(ewma, inplace=True)
```

Wherever `reindexed.ppg` is nan, `fillna` replaces it with the corresponding value from `ewma`. The `inplace` flag tells `fillna` to modify the existing Series rather than create a new one.

A drawback of this method is that it understates the noise in the series. We can solve that problem by adding in resampled residuals:

```
resid = (reindexed.ppg - ewma).dropna()
fake_data = ewma + thinkstats2.Resample(resid, len(reindexed))
reindexed.ppg.fillna(fake_data, inplace=True)
```

resid contains the residual values, not including days when ppg is nan. fake_data contains the sum of the moving average and a random sample of residuals. Finally, fillna replaces nan with values from fake_data.

Figure 12-4 shows the result. The filled data is visually similar to the actual values. Since the resampled residuals are random, the results are different every time; later we'll see how to characterize the error created by missing values.

Serial Correlation

As prices vary from day to day, you might expect to see patterns. If the price is high on Monday, you might expect it to be high for a few more days; and if it's low, you might expect it to stay low. A pattern like this is called **serial correlation**, because each value is correlated with the next one in the series.

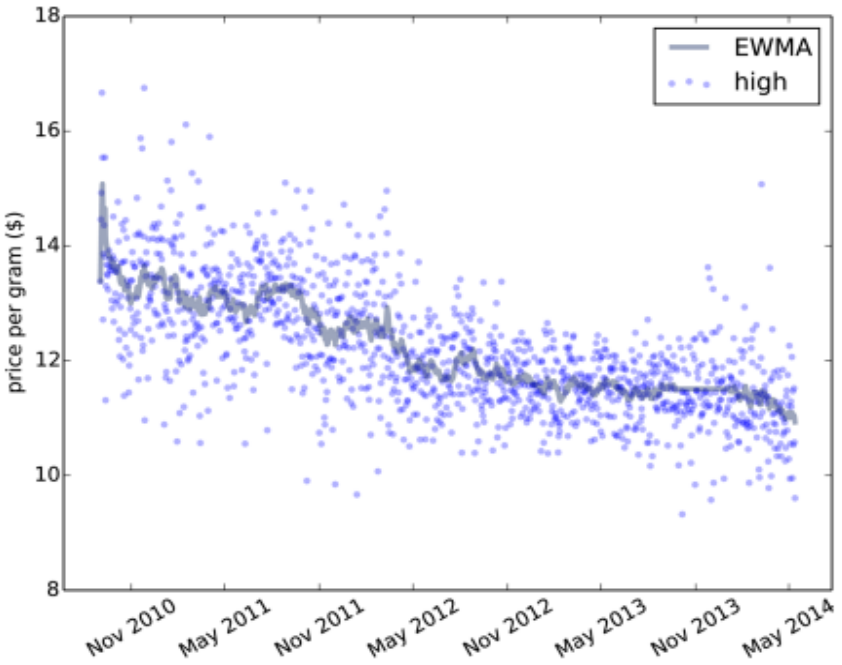

Figure 12-4. Daily price with filled data

To compute serial correlation, we can shift the time series by an interval called a **lag,** and then compute the correlation of the shifted series with the original:

```
def SerialCorr(series, lag=1):
    xs = series[lag:]
    ys = series.shift(lag)[lag:]
    corr = thinkstats2.Corr(xs, ys)
    return corr
```

After the shift, the first `lag` values are `nan`, so I use a slice to remove them before computing `Corr`.

If we apply `SerialCorr` to the raw price data with lag 1, we find serial correlation 0.48 for the high quality category, 0.16 for medium and 0.10 for low. In any time series with a long-term trend, we expect to see strong serial correlations; for example, if prices are falling, we expect to see values above

the mean in the first half of the series and values below the mean in the second half.

It is more interesting to see if the correlation persists if you subtract away the trend. For example, we can compute the residual of the EWMA and then compute its serial correlation:

```
ewma = pandas.ewma(reindexed.ppg, span=30)
resid = reindexed.ppg - ewma
corr = SerialCorr(resid, 1)
```

With lag=1, the serial correlations for the de-trended data are -0.022 for high quality, -0.015 for medium, and 0.036 for low. These values are small, indicating that there is little or no one-day serial correlation in this series.

To check for weekly, monthly, and yearly seasonality, I ran the analysis again with different lags. Here are the results:

Lag	High	Medium	Low
1	-0.029	-0.014	0.034
7	0.02	-0.042	-0.0097
30	0.014	-0.0064	-0.013
365	0.045	0.015	0.033

In the next section, we'll test whether these correlations are statistically significant (they are not), but at this point we can

tentatively conclude that there are no substantial seasonal patterns in these series, at least not with these lags.

Autocorrelation

If you think a series might have some serial correlation, but you don't know which lags to test, you can test them all! The **autocorrelation function** is a function that maps from lag to the serial correlation with the given lag. "Autocorrelation" is another name for serial correlation, used more often when the lag is not 1.

StatsModels, which we used for linear regression in "StatsModels" also provides functions for time series analysis, including acf, which computes the auto-correlation function:

```
import statsmodels.tsa.stattools as smtsa

acf = smtsa.acf(filled.resid, nlags=365, unbiased=True)
```

acf computes serial correlations with lags from 0 through nlags. The unbiased flag tells acf to correct the estimates for the sample size. The result is an array of correlations. If we select daily prices for high quality, and extract correlations for lags 1, 7, 30, and 365, we can confirm that acf and SerialCorr yield approximately the same results:

```
>>> acf[0], acf[1], acf[7], acf[30], acf[365]
1.000, -0.029, 0.020, 0.014, 0.044
```

With lag=0, acf computes the correlation of the series with itself, which is always 1.

Figure 12-5 (left) shows autocorrelation functions for the three quality categories, with nlags=40. The gray region shows

the normal variability we would expect if there is no actual autocorrelation; anything that falls outside this range is statistically significant, with a p-value less than 5%. Since the false positive rate is 5%, and we are computing 120 correlations (40 lags for each of 3 times series), we expect to see about 6 points outside this region. In fact, there are 7. We conclude that there are no autocorrelations in these series that could not be explained by chance.

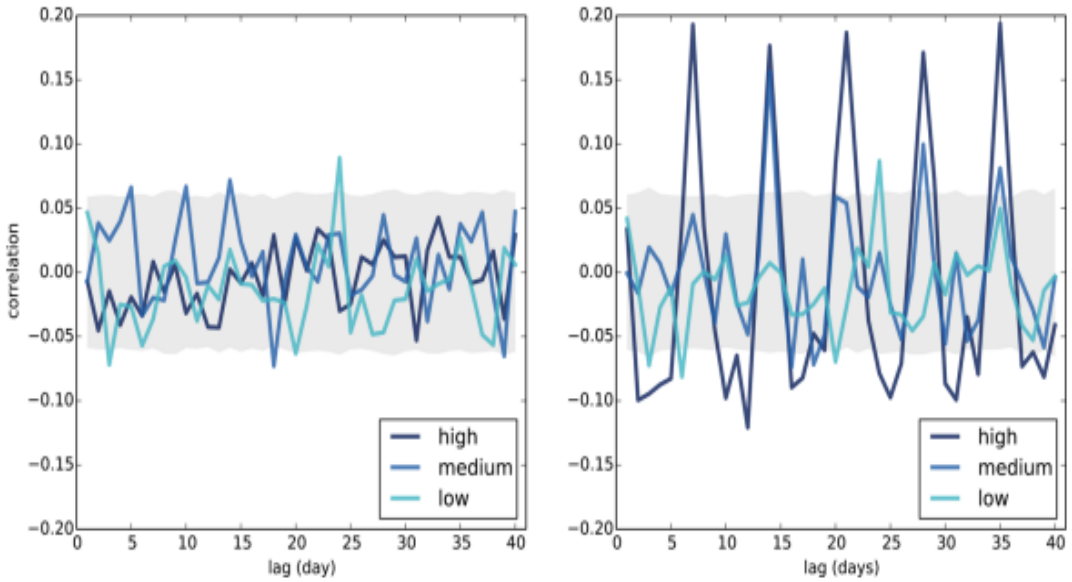

Figure 12-5. Autocorrelation function for daily prices (left), and daily prices with a simulated weekly seasonality (right)

I computed the gray regions by resampling the residuals. You can see my code in timeseries.py; the function is called SimulateAutocorrelation.

To see what the autocorrelation function looks like when there is a seasonal component, I generated simulated data by adding a weekly cycle. Assuming that demand for cannabis is higher on weekends, we might expect the price to be higher. To simulate this effect, I select dates that fall on Friday or

Saturday and add a random amount to the price, chosen from a uniform distribution from $0 to $2.

```
def  AddWeeklySeasonality(daily):
        frisat = (daily.index.dayofweek==4) | (daily.index.dayofweek==5)
        fake = daily.copy()
        fake.ppg[frisat] += np.random.uniform(0, 2, frisat.sum())
        return fake
```

frisat is a boolean Series, True if the day of the week is Friday or Saturday. fake is a new DataFrame, initially a copy of daily, which we modify by adding random values to ppg. frisat.sum() is the total number of Fridays and Saturdays, which is the number of random values we have to generate.

Figure 12-5 (right) shows autocorrelation functions for prices with this simulated seasonality. As expected, the correlations are highest when the lag is a multiple of 7. For high and medium quality, the new correlations are statistically significant. For low quality they are not, because residuals in this category are large; the effect would have to be bigger to be visible through the noise.

Introduction to Survival Analysis

Survival analysis is a way to describe how long things last. It is often used to study human lifetimes, but it also applies to "survival" of mechanical and electronic components, or more generally to intervals in time before an event.

If someone you know has been diagnosed with a life-threatening disease, you might have seen a “5-year survival rate,” which is the probability of surviving five years after diagnosis. That estimate and related statistics are the result of survival analysis.

The code in this topic is in survival.py.

Survival Curves

The fundamental concept in survival analysis is the survival curve, S(t), which is a function that maps from a duration, t, to the probability of surviving longer than t. If you know the distribution of durations, or “lifetimes”, finding the survival curve is easy; it’s just the complement of the CDF:

$$S(t) = 1 - CDF(t)$$

where CDF (t) is the probability of a lifetime less than or equal to t.

For example, in the NSFG dataset, we know the duration of 11189 complete pregnancies. We can read this data and compute the CDF:

```
preg = nsfg.ReadFemPreg()
complete = preg.query('outcome in [1, 3, 4]').prglngth
cdf = thinkstats2.Cdf(complete, label='cdf')
```

The outcome codes 1, 3, 4 indicate live birth, stillbirth, and miscarriage. For this analysis I am excluding induced abortions, ectopic pregnancies, and pregnancies that were in progress when the respondent was interviewed.

The DataFrame method query takes a boolean expression and evaluates it for each row, selecting the rows that yield True.

Figure 13-1 (top) shows the CDF of pregnancy length and its complement, the survival function. To represent the survival function, I define an object that wraps a Cdf and adapts the interface:

```
class SurvivalFunction(object):
    def __init__(self, cdf, label=''):
        self.cdf = cdf
        self.label = label or cdf.label
    @property
    def ts(self):
        return self.cdf.xs
    @property
    def ss(self):
        return 1 - self.cdf.ps
```

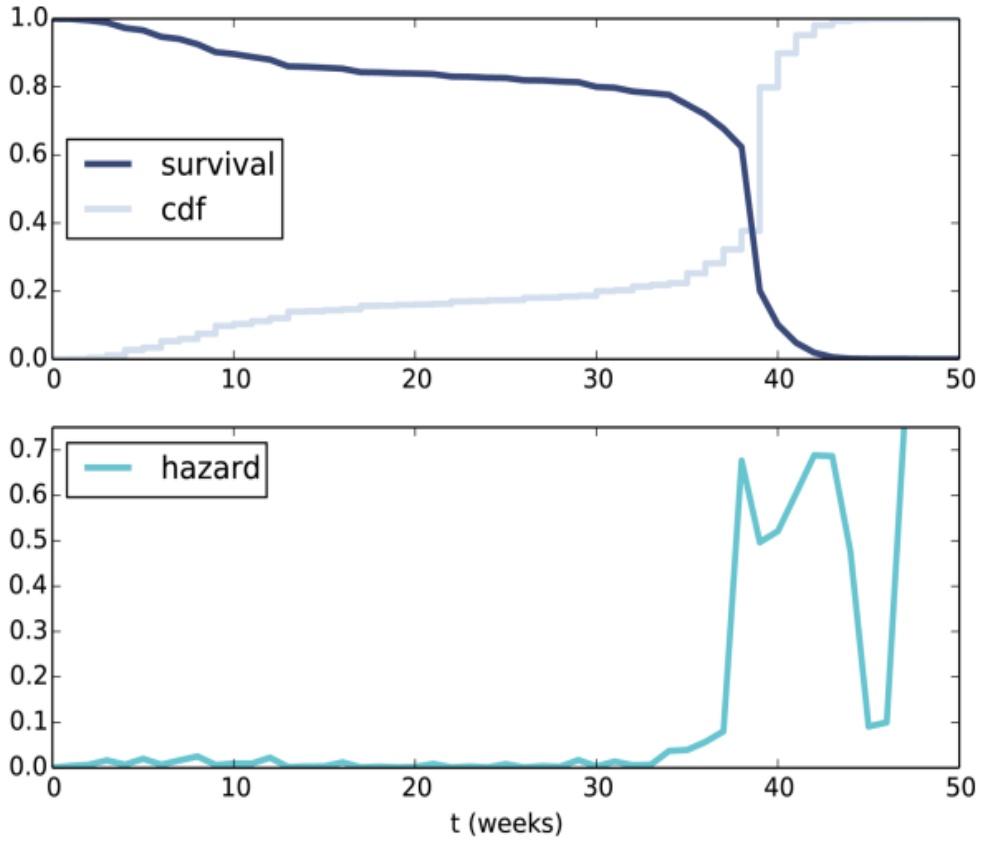

Figure 13-1. Cdf and survival function for pregnancy length (top), hazard function(bottom)

SurvivalFunction provides two properties: ts, which is the sequence of lifetimes, and ss, which is the survival function. In Python, a "property" is a method that can be invoked as if it were a variable.

We can instantiate a SurvivalFunction by passing the CDF of lifetimes:

```
sf = SurvivalFunction(cdf)
```

SurvivalFunction also provides __getitem__ and Prob, which evaluate the survival function:

```
# class SurvivalFunction
        def __getitem__(self, t):
```

```
        return self.Prob(t)

    def Prob(self, t):
        return 1 - self.cdf.Prob(t)
```

For example, sf[13] is the fraction of pregnancies that proceed past the first trimester:

```
>>> sf[13]
0.86022
>>> cdf[13]
0.13978
```

About 86% of pregnancies proceed past the first trimester; about 14% do not.

SurvivalFunction provides Render, so we can plot sf using the functions in thinkplot:

```
thinkplot.Plot(sf)
```

Figure 13-1 (top) shows the result. The curve is nearly flat between 13 and 26 weeks, which shows that few pregnancies end in the second trimester. And the curve is steepest around 39 weeks, which is the most common pregnancy length.