Log Transformation

In the previous section, we briefly introduced the notion of taking the logarithm of the count to map the data to exponential-width bins. Let's take a closer look at that now.

The log function is the inverse of the exponential function. It is defined such that $\log_a(a^x) = x$, where a is a positive constant, and x can be any positive number. Since $a^0 = 1$, we have $\log_a(1) = 0$. This means that the log function maps the small range of numbers between (0, 1) to the entire range of negative numbers $(-\infty, 0)$. The function $\log_{10}(x)$ maps the range of [1, 10] to [0, 1], [10, 100] to [1, 2], and so on. In other words, the log function compresses the range of large numbers and expands the range of small numbers. The larger x is, the slower $\log(x)$ increments.

This is easier to digest by looking at a plot of the log function (see Figure 2-6). Note how the horizontal x values from 100 to 1,000 get compressed into just 2.0 to 3.0 in the vertical y range, while the tiny horizontal portion of x values less than 100 are mapped to the rest of the vertical range.

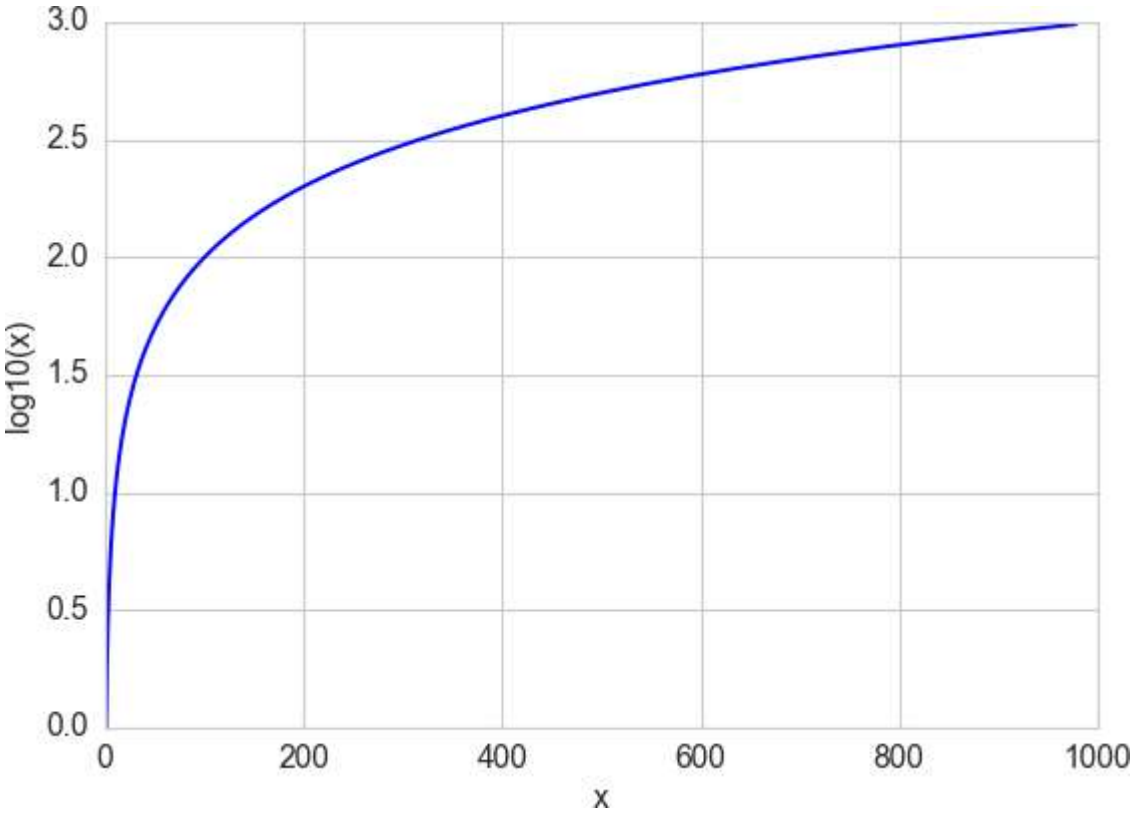

Figure 2-6. The log function compresses the high numeric range and expands the low range

The log transform is a powerful tool for dealing with positive numbers with a heavy-tailed distribution. (A heavy-tailed distribution places more probability mass in the tail range than a Gaussian distribution.) It compresses the long tail in the high end of the distribution into a shorter tail, and expands the low end into a longer head. Figure 2-7 compares the histograms of Yelp business review counts before and after log transformation (see Example 2-6). The y-axes are now both on a normal (linear) scale. The increased bin spacing in the bottom plot between the range of (0.5, 1] is due to there being only 10 possible integer counts between 1

and 10. Notice that the original review counts are very concentrated in the low count region, with outliers stretching out above 4,000. After log transformation, the histogram is less concentrated in the low end and more spread out over the x-axis.

Example 2-6. Visualizing the distribution of review counts before and after log transform

```
>>> fig, (ax1, ax2) = plt.subplots(2,1)
>>> biz_df['review_count'].hist(ax=ax1, bins=100)
>>> ax1.tick_params(labelsize=14)
>>> ax1.set_xlabel('review_count', fontsize=14)
>>> ax1.set_ylabel('Occurrence', fontsize=14)

>>> biz_df['log_review_count'].hist(ax=ax2, bins=100)
>>> ax2.tick_params(labelsize=14)
>>> ax2.set_xlabel('log10(review_count))', fontsize=14)
>>> ax2.set_ylabel('Occurrence', fontsize=14)
```

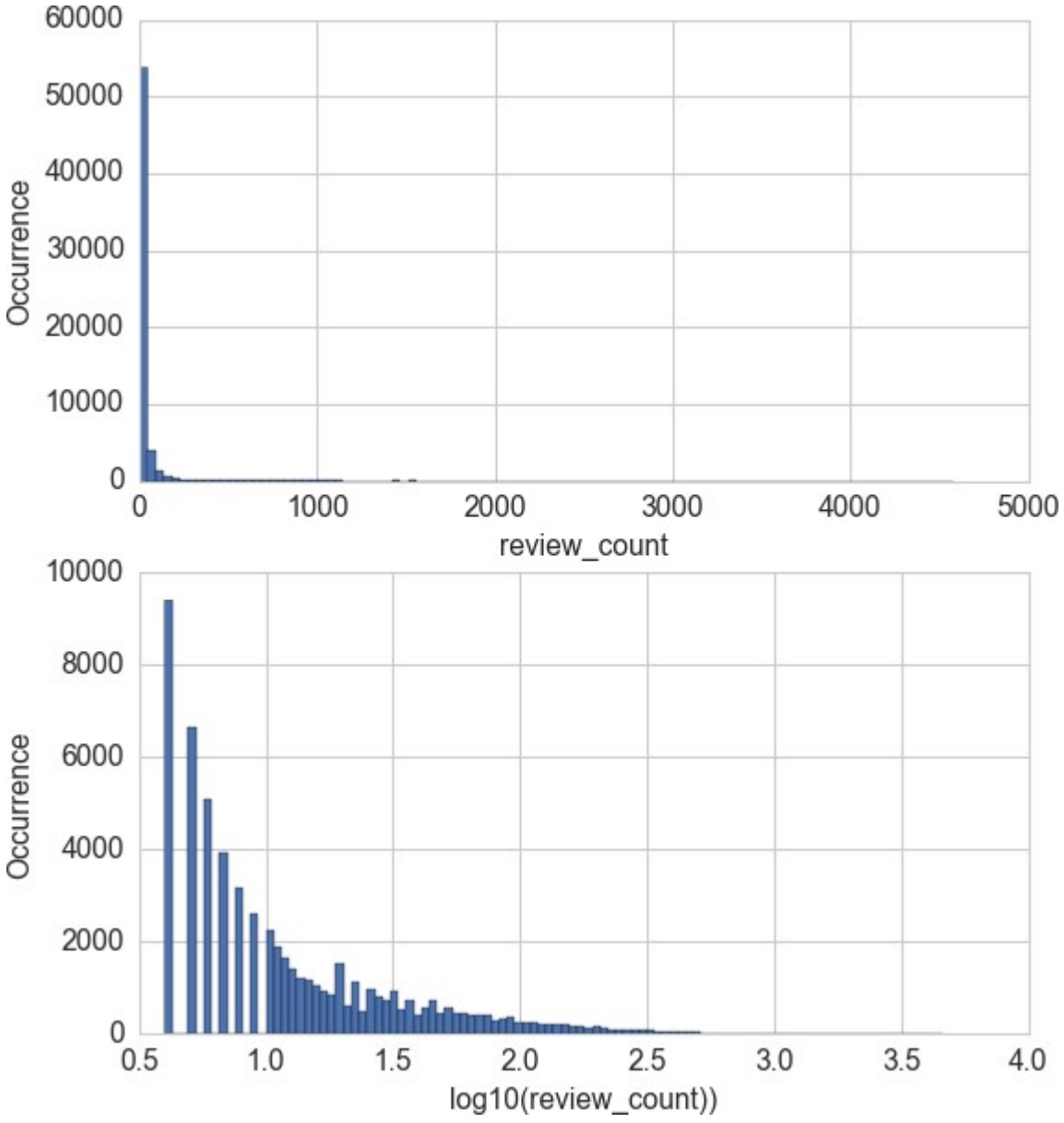

Figure 2-7. Comparison of Yelp business review counts before (top) and after (bottom) log transformation

As another example, let's consider the Online News Popularity dataset from the UC Irvine Machine Learning Repository (Fernandes et al., 2015).

Statistics on the Online News Popularity Dataset

- The dataset includes 60 features of a set of 39,797 news articles published by Mashable over a period of 2 years.

Our goal is to use these features to predict the popularity of the articles in terms of the number of shares on social media. In this example, we'll focus on only one feature—the number of words in the article. Figure 2-8 shows the histograms of the feature before and after log transformation (see Example 2-7). Notice that the distribution looks much more Gaussian after log transformation, with the exception of the burst of number of articles of length zero (no content).

Example 2-7. Visualizing the distribution of news article popularity with and without log transformation

```
>>> fig, (ax1, ax2) = plt.subplots(2,1)
>>> df['n_tokens_content'].hist(ax=ax1, bins=100)
>>> ax1.tick_params(labelsize=14)
>>> ax1.set_xlabel('Number of Words in Article', fontsize=14)
>>> ax1.set_ylabel('Number of Articles', fontsize=14)

>>> df['log_n_tokens_content'].hist(ax=ax2, bins=100)
>>> ax2.tick_params(labelsize=14)
>>> ax2.set_xlabel('Log of Number of Words', fontsize=14)
>>> ax2.set_ylabel('Number of Articles', fontsize=14)
```

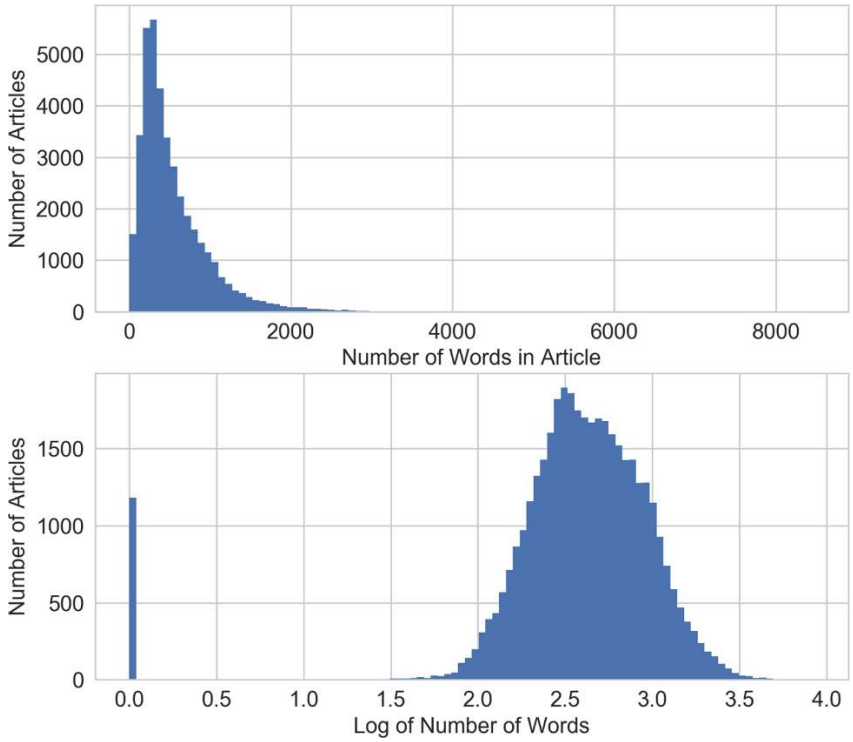

Figure 2-8. Comparison of word counts in Mashable news articles before (top) and after (bottom) log transformation

Log Transform in Action

Let's see how the log transform performs for supervised learning. We'll use both of the previous datasets here. For the Yelp reviews dataset, we'll use the number of reviews to predict the average rating of a business (see Example 2-8). For the Mashable news articles, we'll use the number of words in an article to predict its popularity. Since the outputs are continuous numbers, we'll use simple linear regression as the model. We use scikit-learn to perform 10-fold cross validation of linear regression on the feature with and without log transformation. The models are evaluated by the R-squared score, which measures how well a trained regression model predicts new data. Good models have high R-squared scores. A perfect model gets the maximum score of 1. The score can be negative, and a bad model can get an arbitrarily low negative score. Using cross validation, we obtain not only an estimate of the score but also a variance, which helps us gauge whether the differences between the two models are meaningful.

Example 2-8. Using log transformed Yelp review counts to predict average business rating

```
>>> import pandas as pd
>>> import numpy as np
>>> import json
>>> from sklearn import linear_model
>>> from sklearn.model_selection import cross_val_score

# Using the previously loaded Yelp reviews DataFrame,
# compute the log transform of the Yelp review count.
# Note that we add 1 to the raw count to prevent the logarithm from
# exploding into negative infinity in case the count is zero.
>>> biz_df['log_review_count'] = np.log10(biz_df['review_count'] + 1)

# Train linear regression models to predict the average star rating of a
business,
# using the review_count feature with and without log transformation.
# Compare the 10-fold cross validation score of the two models.
>>> m_orig = linear_model.LinearRegression()
>>> scores_orig = cross_val_score(m_orig, biz_df[['review_count']],
...                               biz_df['stars'], cv=10)
>>> m_log = linear_model.LinearRegression()
>>> scores_log = cross_val_score(m_log, biz_df[['log_review_count']],
...                              biz_df['stars'], cv=10)
>>> print("R-squared score without log transform: %0.5f (+/- %0.5f)"
...       % (scores_orig.mean(), scores_orig.std() * 2))
>>> print("R-squared score with log transform: %0.5f (+/- %0.5f)"
...       % (scores_log.mean(), scores_log.std() * 2))
R-squared score without log transform: -0.03683 (+/- 0.07280)
R-squared score with log transform: -0.03694 (+/- 0.07650)
```

Judging by the output of the experiment, the two simple models (with and without log transform) are equally bad at predicting the target, with the log transformed feature performing slightly worse. How disappointing! It's not surprising that neither of them are very good, given that they both use just one feature, but one would have hoped that the log transformed feature might have performed better.

Now let's look at how the log transform does on the Online News Popularity dataset (Example 2-9).

Example 2-9. Using log transformed word counts in the Online News Popularity dataset to predict article popularity

```
# Download the Online News Popularity dataset from UCI, then use
# Pandas to load the file into a DataFrame.
>>> df = pd.read_csv('OnlineNewsPopularity.csv', delimiter=', ')

# Take the log transform of the 'n_tokens_content' feature, which
# represents the number of words (tokens) in a news article.
>>> df['log_n_tokens_content'] = np.log10(df['n_tokens_content'] + 1)

# Train two linear regression models to predict the number of shares
# of an article, one using the original feature and the other the
# log transformed version.
>>> m_orig = linear_model.LinearRegression()
>>> scores_orig = cross_val_score(m_orig, df[['n_tokens_content']],
...                               df['shares'], cv=10)
>>> m_log = linear_model.LinearRegression()
>>> scores_log = cross_val_score(m_log, df[['log_n_tokens_content']],
...                              df['shares'], cv=10)
>>> print("R-squared score without log transform: %0.5f (+/- %0.5f)"
...       % (scores_orig.mean(), scores_orig.std() * 2))
>>> print("R-squared score with log transform: %0.5f (+/- %0.5f)"
...       % (scores_log.mean(), scores_log.std() * 2))
R-squared score without log transform: -0.00242 (+/- 0.00509)
R-squared score with log transform: -0.00114 (+/- 0.00418)
```

The confidence intervals still overlap, but the model with the log transformed feature is doing better than the one without. Why is the log transform so much more successful on this dataset? We can get a clue by looking at the scatter plots (Example 2-10) of the input feature and target values. As can be seen in the bottom panel of Figure 2-9, the log transform reshaped the x-axis, pulling the articles with large outliers in the target value (>200,000 shares) further out toward the righthand side of the axis. This gives the linear model more "breathing room" on the low end of the input feature space. Without the log transform (top panel), the model is under more pressure to fit very different target values under very small changes in the input.

Example 2-10. Visualizing the correlation between input and output in the news popularity prediction problem

```
>>> fig2, (ax1, ax2) = plt.subplots(2,1)
>>> ax1.scatter(df['n_tokens_content'], df['shares'])
>>> ax1.tick_params(labelsize=14)
>>> ax1.set_xlabel('Number of Words in Article', fontsize=14)
>>> ax1.set_ylabel('Number of Shares', fontsize=14)

>>> ax2.scatter(df['log_n_tokens_content'], df['shares'])
>>> ax2.tick_params(labelsize=14)
>>> ax2.set_xlabel('Log of the Number of Words in Article', fontsize=14)
>>> ax2.set_ylabel('Number of Shares', fontsize=14)
```

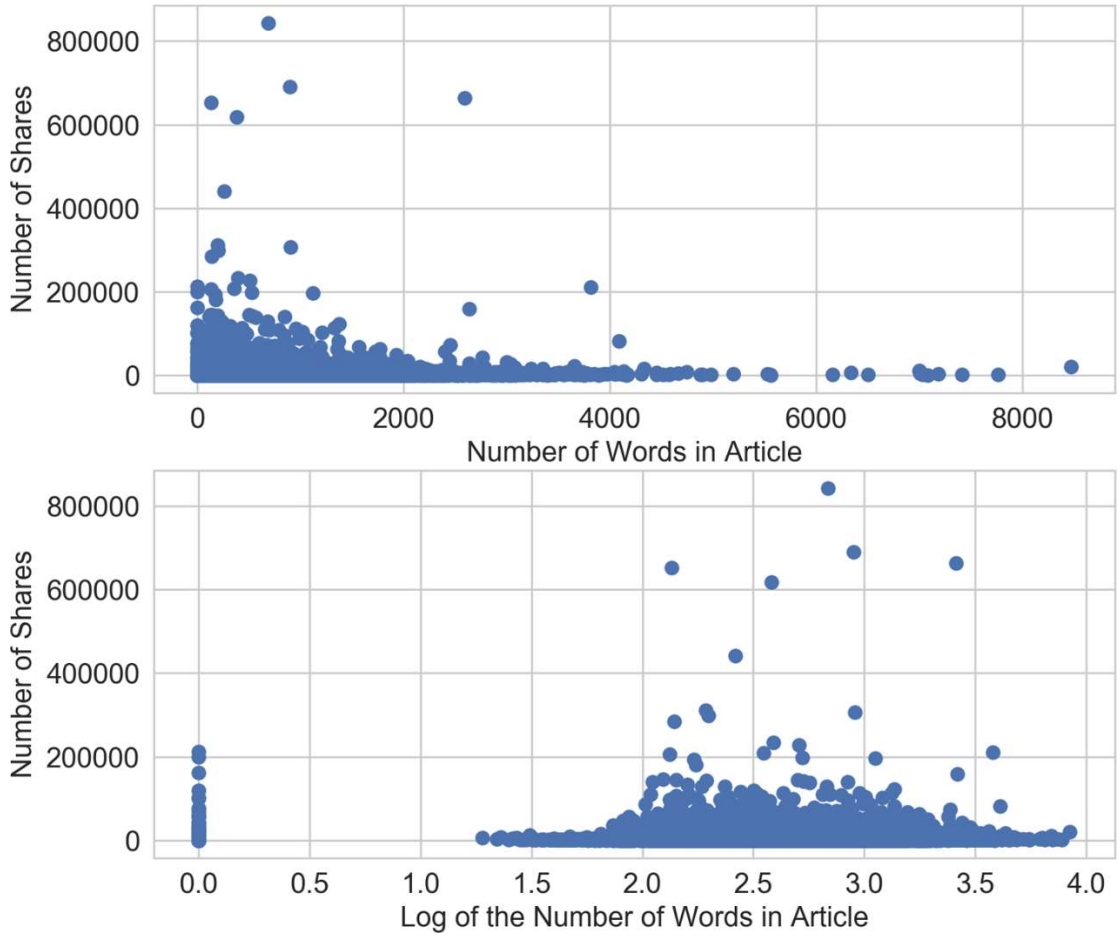

Figure 2-9. Scatter plots of number of words (input) versus number of shares (target) in the Online News Popularity dataset—the top plot visualizes the original feature, and the bottom plot shows the scatter plot after log transformation

Compare this with the same scatter plot applied to the Yelp reviews dataset (Example 2-11). Figure 2-10 looks very different from Figure 2-9. The average star rating is discretized in increments of half-stars ranging from 1 to 5. High review counts (roughly >2,500 reviews) do correlate with higher average star ratings, but the relationship is far from linear. There is no clear way to draw a line to predict the average star rating based on either input. Essentially, the plot shows that review count and its logarithm are both bad linear predictors of average star rating.

Example 2-11. Visualizing the correlation between input and output in Yelp business review prediction

```
>>> fig, (ax1, ax2) = plt.subplots(2,1)
>>> ax1.scatter(biz_df['review_count'], biz_df['stars'])
>>> ax1.tick_params(labelsize=14)
>>> ax1.set_xlabel('Review Count', fontsize=14)
>>> ax1.set_ylabel('Average Star Rating', fontsize=14)

>>> ax2.scatter(biz_df['log_review_count'], biz_df['stars'])
>>> ax2.tick_params(labelsize=14)
>>> ax2.set_xlabel('Log of Review Count', fontsize=14)
>>> ax2.set_ylabel('Average Star Rating', fontsize=14)
```

The Importance of Data Visualization

The comparison of the effect of the log transform on two different datasets illustrates the importance of visualizing the data. Here, we intentionally kept the input and target variables simple so that we can easily visualize the relationship between them. Plots like those in Figure 2-10 immediately reveal that the chosen model (linear) cannot possibly represent the relationship between the chosen input and target. On the other hand, one could convincingly model the distribution of review count *given* the average star rating. When building models, it is a good idea to visually inspect the relationships between input and output, and between different input features.

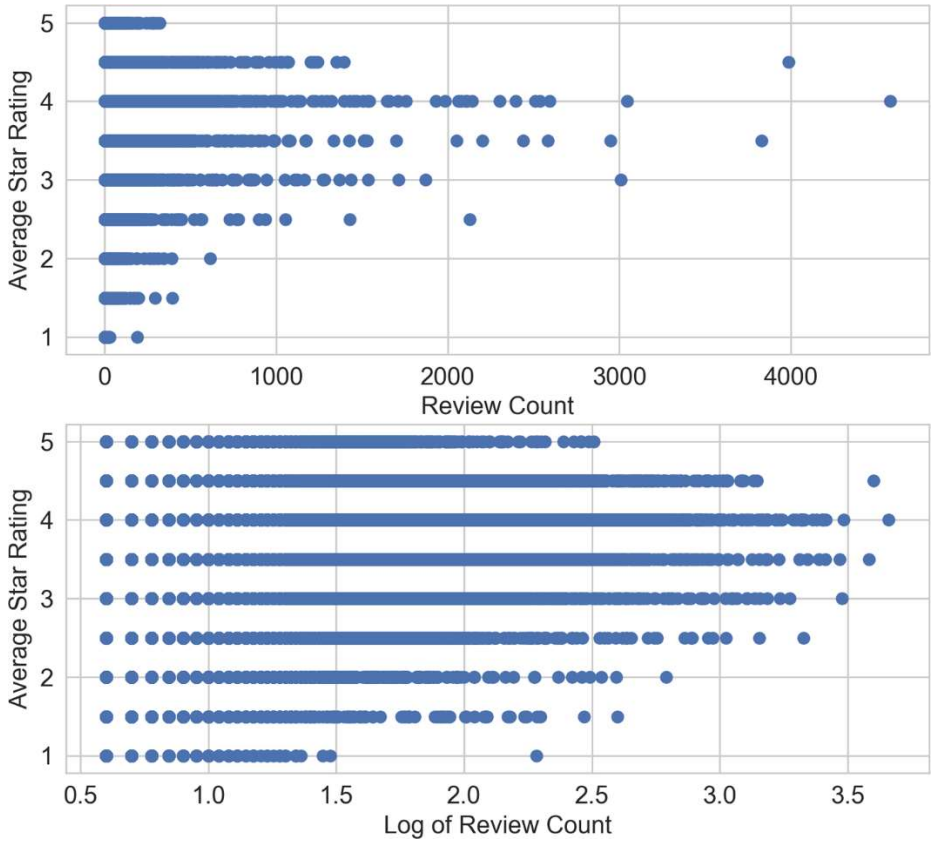

Figure 2-10. Scatter plots of review counts (input) versus average star rating (target) in the Yelp reviews dataset—the top panel plots the original review count, and the bottom panel plots the review count after log transformation

Power Transforms: Generalization of the Log Transform

The log transform is a specific example of a family of transformations known as *power transforms*. In statistical terms, these are *variance-stabilizing transformations*. To understand why variance stabilization is good, consider the Poisson distribution. This is a heavy-tailed distribution with a variance that is equal to its mean: hence, the larger its centre of mass, the larger its variance, and the heavier the tail. Power transforms change the distribution of the variable so that the variance is no longer dependent on the mean. For example, suppose a random variable X has the Poisson distribution. If we transform X by taking its square root, the variance of $\widetilde{X} = \sqrt{X}$ is roughly constant, instead of being equal to the mean.

Figure 2-11 illustrates λ, which represents the mean of the distribution. As λ increases, not only does the mode of the distribution shift to the right, but the mass spreads out and the variance becomes larger.

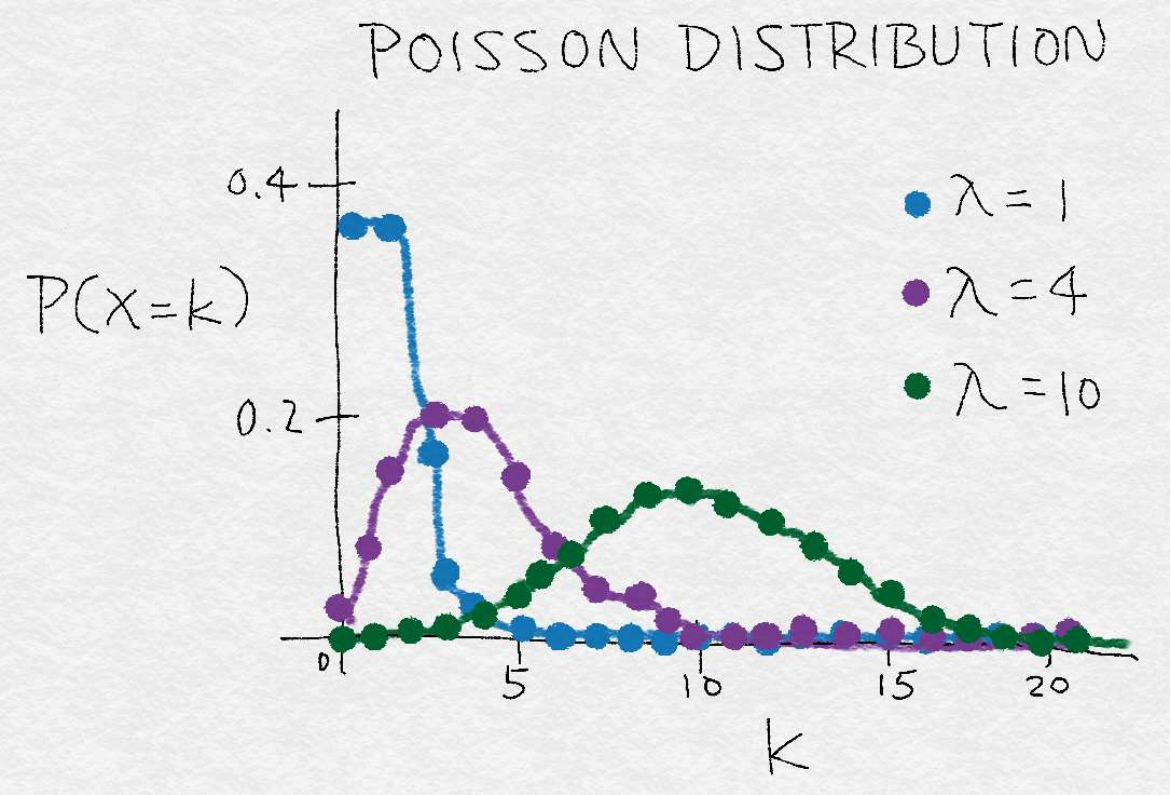

Figure 2-11. A rough illustration of the Poisson distribution, an example distribution where the variance increases along with the mean

A simple generalization of both the square root transform and the log transform is known as the Box-Cox transform:

$$\tilde{x} = \begin{cases} \frac{x^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0, \\ \ln(x) & \text{if } \lambda = 0. \end{cases}$$

Figure 2-12 shows the Box-Cox transform for λ = 0 (the log transform), λ = 0.25, λ = 0.5 (a scaled and shifted version of the square root transform), λ = 0.75, and λ = 1.5. Setting λ to be less than 1 compresses the higher values, and setting λ higher than 1 has the opposite effect.

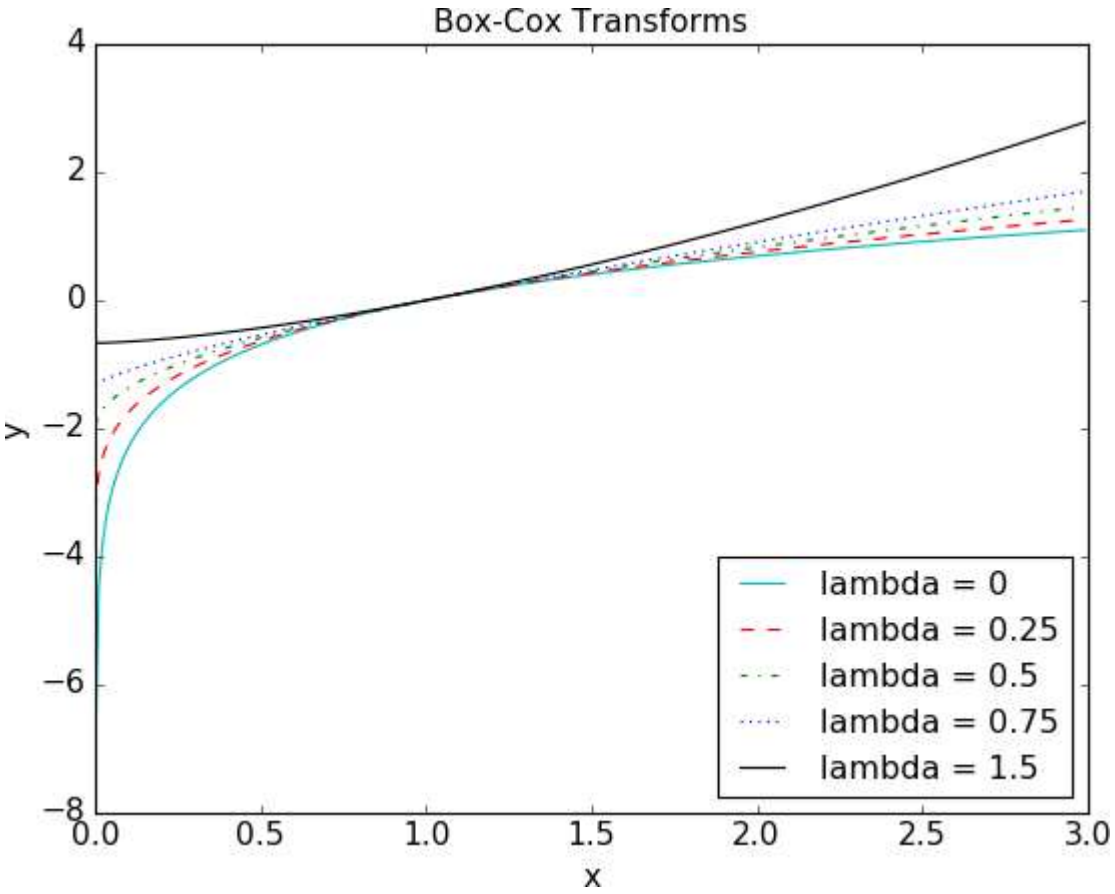

Figure 2-12. Box-Cox transforms for different values of λ

The Box-Cox formulation only works when the data is positive. For nonpositive data, one could shift the values by adding a fixed constant. When applying the Box-Cox transformation or a more general power transform, we have to determine a value for the parameter λ. This may be done via maximum likelihood (finding the λ that maximizes the Gaussian likelihood of the resulting transformed signal) or Bayesian methods. A full treatment of the usage of Box-Cox and general power transforms is outside the scope of this book. Interested readers may find more information on power transforms in *Econometric Methods* by Johnston and DiNardo (1997). Fortunately, SciPy's `stats` package contains an implementation of the Box-Cox transformation that includes finding the optimal transform parameter. Example 2-12 demonstrates its use on the Yelp reviews dataset.

Example 2-12. Box-Cox transformation of Yelp business review counts

```
>>> from scipy import stats

# Continuing from the previous example, assume biz_df contains
# the Yelp business reviews data.
# The Box-Cox transform assumes that input data is positive.
# Check the min to make sure.
>>> biz_df['review_count'].min()
3

# Setting input parameter lmbda to 0 gives us the log transform (without
# constant offset)
>>> rc_log = stats.boxcox(biz_df['review_count'], lmbda=0)
```

```
# By default, the scipy implementation of Box-Cox transform finds the
lambda
# parameter that will make the output the closest to a normal distribution
>>> rc_bc, bc_params = stats.boxcox(biz_df['review_count'])
>>> bc_params
-0.4106510862321085
```

Figure 2-13 provides a visual comparison of the distributions of the original and transformed counts (see Example 2-13).

Example 2-13. Visualizing the histograms of original, log transformed, and Box-Cox transformed counts

```
>>> fig, (ax1, ax2, ax3) = plt.subplots(3,1)
# original review count histogram
>>> biz_df['review_count'].hist(ax=ax1, bins=100)
>>> ax1.set_yscale('log')
>>> ax1.tick_params(labelsize=14)
>>> ax1.set_title('Review Counts Histogram', fontsize=14)
>>> ax1.set_xlabel('')
>>> ax1.set_ylabel('Occurrence', fontsize=14)

# review count after log transform
>>> biz_df['rc_log'].hist(ax=ax2, bins=100)
>>> ax2.set_yscale('log')
>>> ax2.tick_params(labelsize=14)
>>> ax2.set_title('Log Transformed Counts Histogram', fontsize=14)
>>> ax2.set_xlabel('')
>>> ax2.set_ylabel('Occurrence', fontsize=14)

# review count after optimal Box-Cox transform
>>> biz_df['rc_bc'].hist(ax=ax3, bins=100)
>>> ax3.set_yscale('log')
>>> ax3.tick_params(labelsize=14)
>>> ax3.set_title('Box-Cox Transformed Counts Histogram', fontsize=14)
>>> ax3.set_xlabel('')
>>> ax3.set_ylabel('Occurrence', fontsize=14)
```

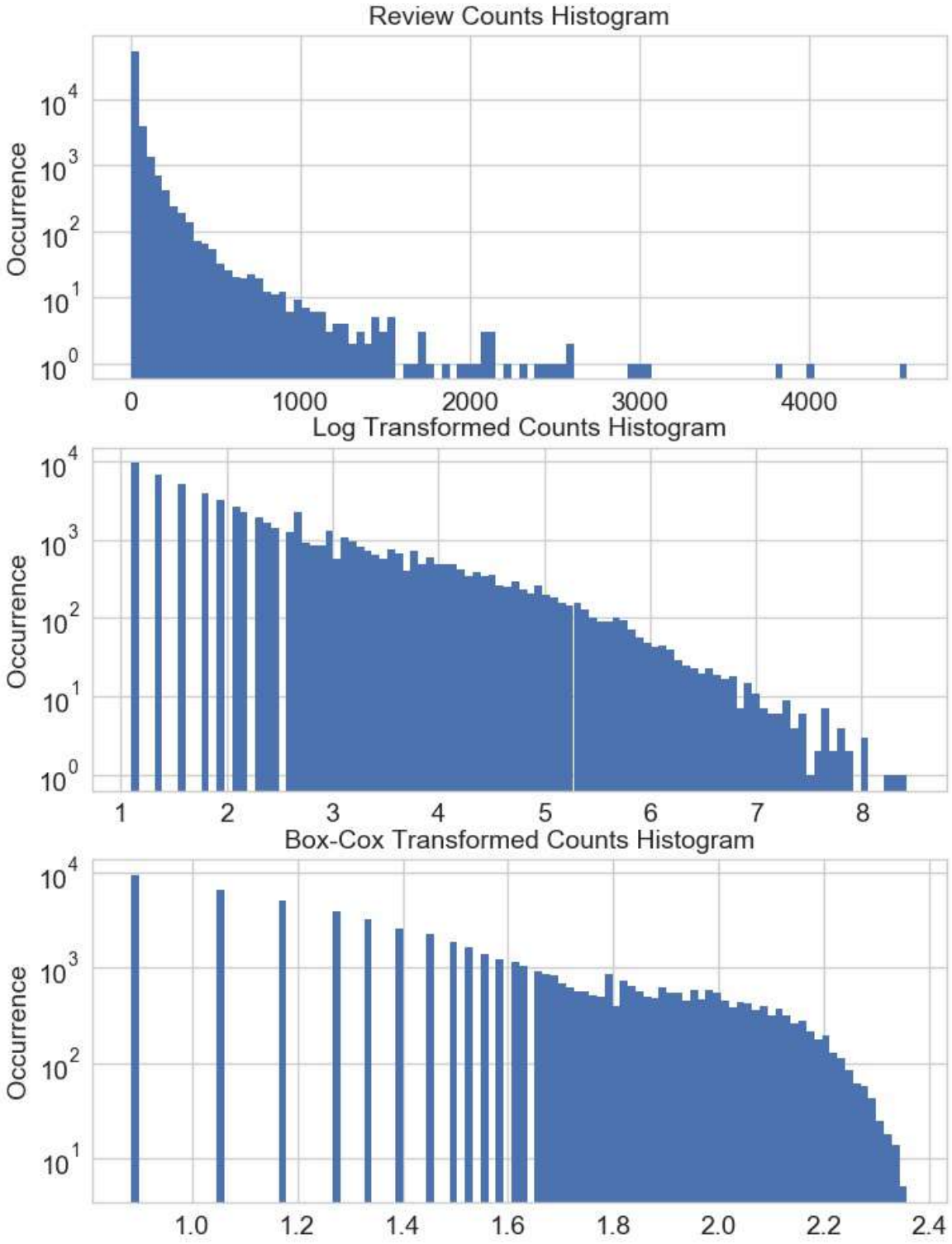

Figure 2-13. Box-Cox transformation of Yelp business review counts (bottom), compared to original (top) and log transformed (middle) histograms

A *probability plot*, or probplot, is an easy way to visually compare an empirical distribution of data against a theoretical distribution. This is essentially a scatter plot of observed versus theoretical quantiles. Figure 2-14 shows the probplots of original and transformed Yelp review counts data against the normal distribution (see Example 2-14). Since the observed data is strictly positive and the Gaussian can be negative, the quantiles could never match up on the negative end. Thus, our focus is on the positive side. On this front, the original counts

are obviously much more heavy-tailed than a normal distribution. (The ordered values go up to 4,000, whereas the theoretical quantiles only stretch to 4.) Both the plain log transform and the optimal Box-Cox transform bring the positive tail closer to normal. The optimal Box-Cox transform deflates the tail more than the log transform, as is evident from the fact that the tail flattens out under the red diagonal equivalence line.

Example 2-14. Probability plots of original and transformed counts against the normal distribution

```
>>> fig2, (ax1, ax2, ax3) = plt.subplots(3,1)
>>> prob1 = stats.probplot(biz_df['review_count'], dist=stats.norm,
plot=ax1)
>>> ax1.set_xlabel('')
>>> ax1.set_title('Probplot against normal distribution')
>>> prob2 = stats.probplot(biz_df['rc_log'], dist=stats.norm, plot=ax2)
>>> ax2.set_xlabel('')
>>> ax2.set_title('Probplot after log transform')
>>> prob3 = stats.probplot(biz_df['rc_bc'], dist=stats.norm, plot=ax3)
>>> ax3.set_xlabel('Theoretical quantiles')
>>> ax3.set_title('Probplot after Box-Cox transform')
```

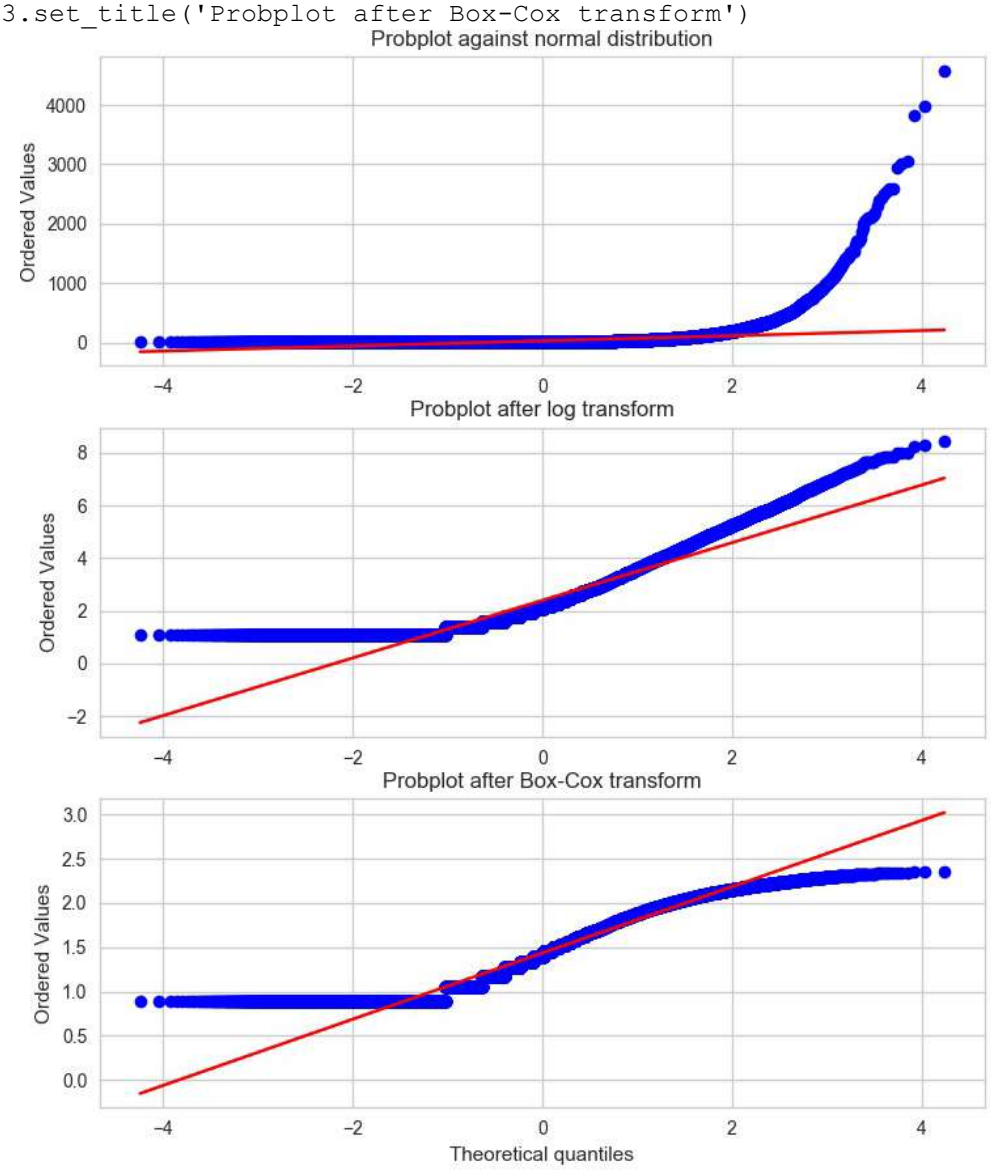

Figure 2-14. Comparing the distribution of raw and transformed review counts against the normal distribution