Feature Scaling or Normalization

Some features, such as latitude or longitude, are bounded in value. Other numeric features, such as counts, may increase without bound. Models that are smooth functions of the input, such as linear regression, logistic regression, or anything that involves a matrix, are affected by the scale of the input. Tree-based models, on the other hand, couldn't care less. If your model is sensitive to the scale of input features, feature scaling could help. As the name suggests, feature scaling changes the scale of the feature. Sometimes people also call it *feature normalization*. Feature scaling is usually done individually to each feature. Next, we will discuss several types of common scaling operations, each resulting in a different distribution of feature values.

Min-Max Scaling

Let x be an individual feature value (i.e., a value of the feature in some data point), and $\min(x)$ and $\max(x)$, respectively, be the minimum and maximum values of this feature over the entire dataset. Min-max scaling squeezes (or stretches) all feature values to be within the range of [0, 1]. Figure 2-15 demonstrates this concept. The formula for min-max scaling is:

$$\tilde{x} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

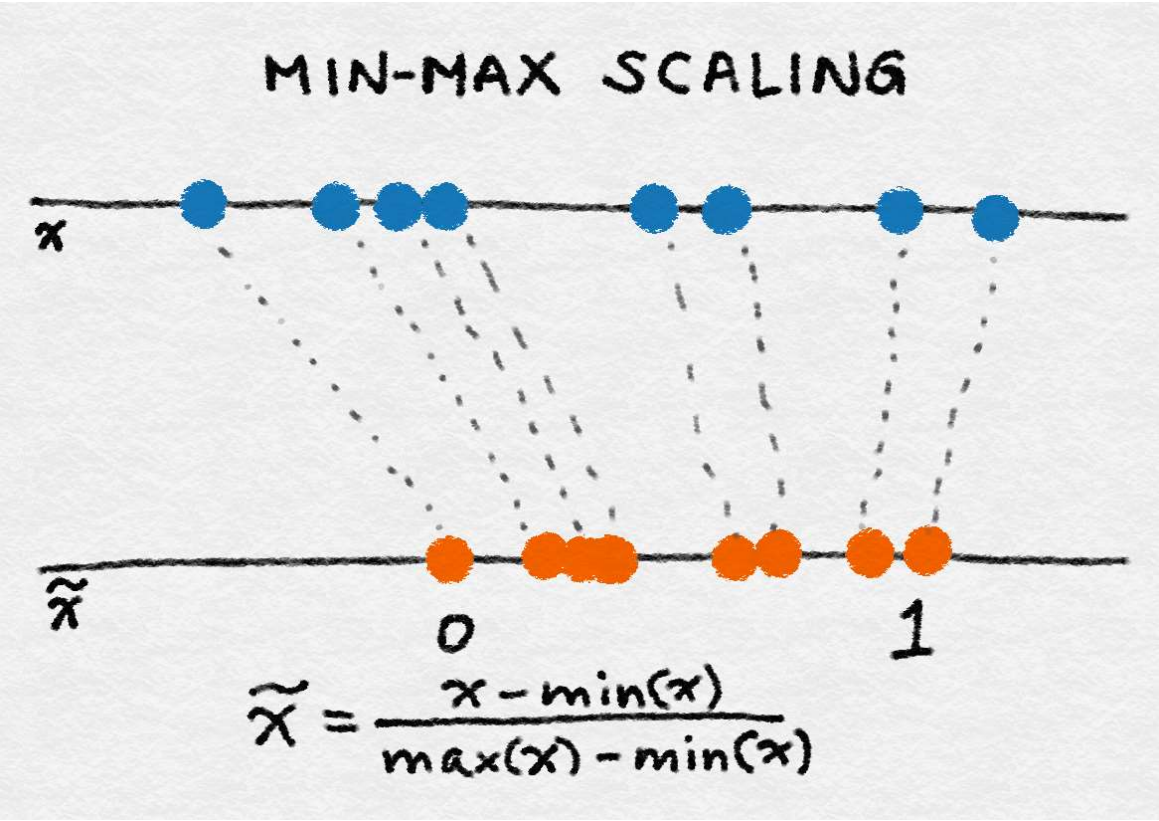

Figure 2-15. Illustration of min-max scaling

Standardization (Variance Scaling)

Feature standardization is defined as:

$$x\tilde{}=x-\text{mean}(x)\text{sqrt}(\text{var}(x))$$

It subtracts off the mean of the feature (over all data points) and divides by the variance. Hence, it can also be called *variance scaling*. The resulting scaled feature has a mean of 0 and a variance of 1. If the original feature has a Gaussian distribution, then the scaled feature does too. Figure 2-16 is an illustration of standardization.

$$\tilde{x} = \frac{x - \text{mean}(x)}{\text{sqrt}(\text{var}(x))}$$

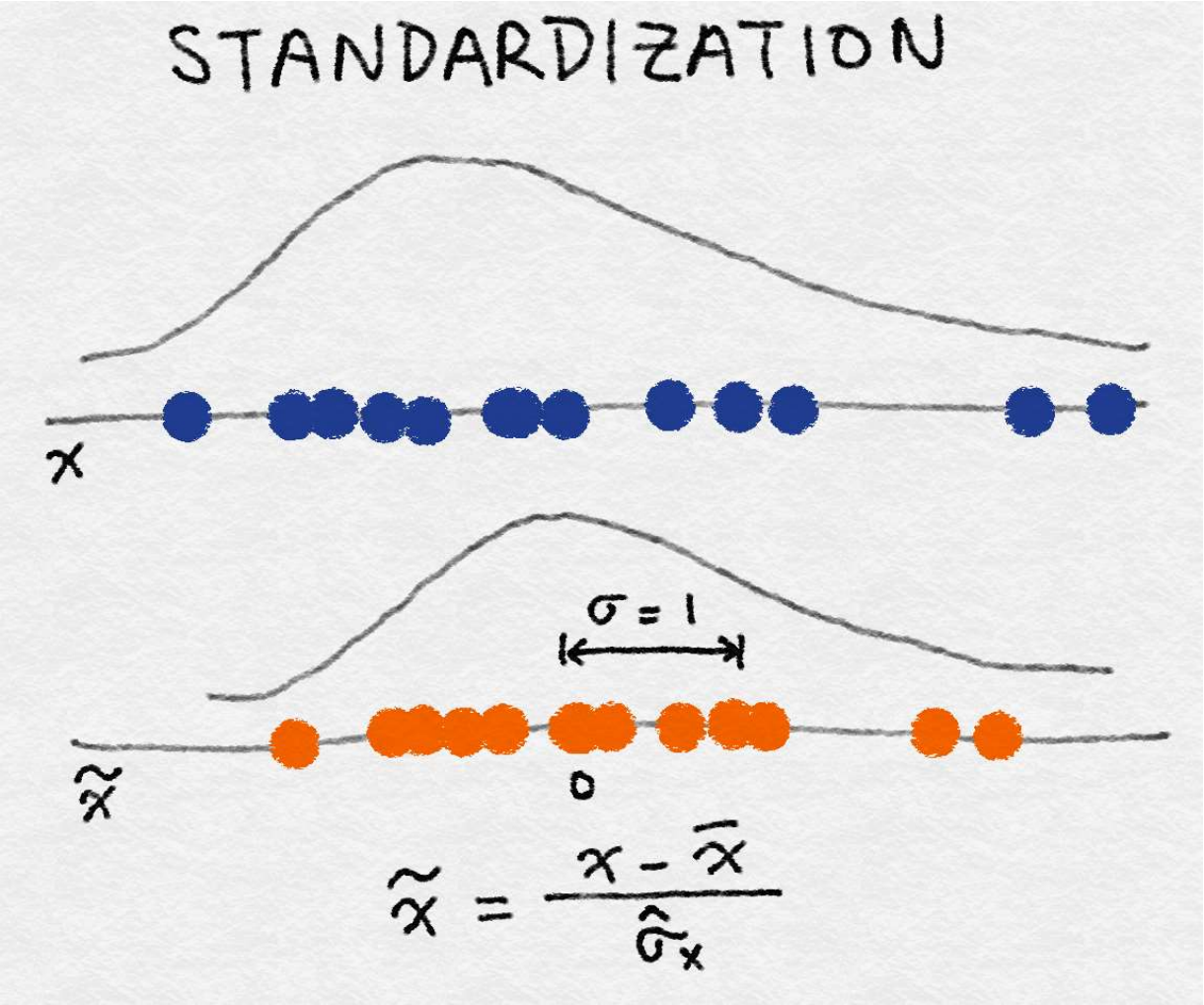

Figure 2-16. Illustration of feature standardization

Don’t “Centre” Sparse Data!

Use caution when performing min-max scaling and standardization on sparse features. Both subtract a quantity from the original feature value. For min-max scaling, the shift is the minimum over all values of the current feature; for standardization, it is the mean. If the shift is not zero, then these two transforms can turn a sparse feature vector where most values are zero into a dense one. This in turn could create a huge computational burden for the classifier, depending on how it is implemented (not to mention that it would be horrendous if the representation now included every word that didn’t appear in a document!). Bag-of-words is a sparse representation, and most classification libraries optimize for sparse inputs.

ℓ^2 Normalization

This technique normalizes (divides) the original feature value by what’s known as the ℓ^2 norm, also known as the Euclidean norm. It’s defined as follows:

$$\tilde{x} = \frac{x}{\|x\|_2}$$

The ℓ^2 norm measures the length of the vector in coordinate space. The definition can be derived from the well-known Pythagorean theorem that gives us the length of the hypotenuse of a right triangle given the lengths of the sides:

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_m^2}$$

The ℓ^2 norm sums the squares of the values of the features across data points, then takes the square root. After ℓ^2 normalization, the feature column has norm 1. This is also sometimes called ℓ^2 scaling. (Loosely speaking, *scaling* means multiplying by a constant, whereas *normalization* could involve a number of operations.) Figure 2-17 illustrates ℓ^2 normalization.

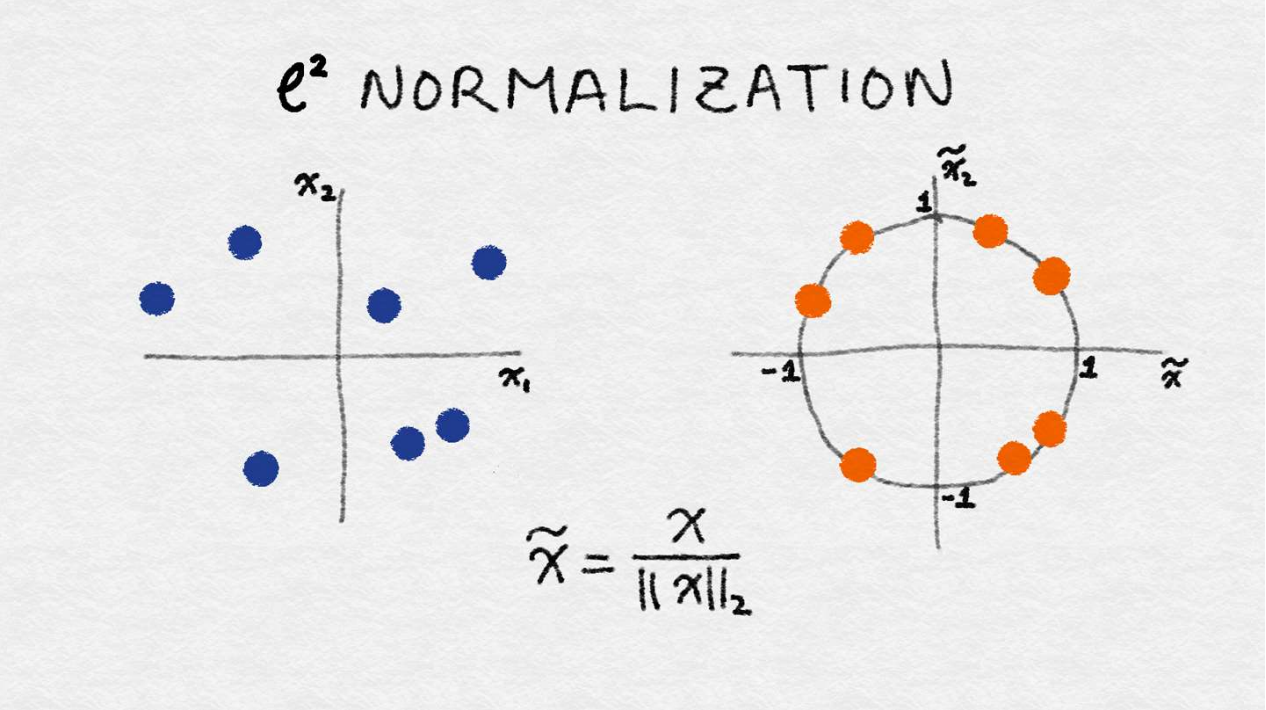

Figure 2-17. Illustration of ℓ^2 feature normalization

Data Space Versus Feature Space

Note that the illustration in Figure 2-17 is in data space, not feature space. One can also do ℓ^2 normalization for the data point instead of the feature, which will result in data vectors with unit norm (norm of 1). See the discussion in "Bag-of-Words" about the complementary nature of data vectors and feature vectors.

No matter the scaling method, feature scaling always divides the feature by a constant (known as the *normalization constant*). Therefore, it does not change the shape of the single-feature distribution. We'll illustrate this with the online news article token counts (see Example 2-15).

Example 2-15. Feature scaling example

```
>>> import pandas as pd
>>> import sklearn.preprocessing as preproc

# Load the Online News Popularity dataset
>>> df = pd.read_csv('OnlineNewsPopularity.csv', delimiter=', ')

# Look at the original data - the number of words in an article
>>> df['n_tokens_content'].as_matrix()
array([ 219.,  255.,  211., ...,  442.,  682.,  157.])

# Min-max scaling
>>> df['minmax'] = preproc.minmax_scale(df[['n_tokens_content']])
>>> df['minmax'].as_matrix()
array([ 0.02584376,  0.03009205,  0.02489969, ...,  0.05215955,
        0.08048147,  0.01852726])
```

```
# Standardization - note that by definition, some outputs will be negative
>>> df['standardized'] =
preproc.StandardScaler().fit_transform(df[['n_tokens_content']])
>>> df['standardized'].as_matrix()
array([-0.69521045, -0.61879381, -0.71219192, ..., -0.2218518 ,
        0.28759248, -0.82681689])

# L2-normalization
>>> df['l2_normalized'] = preproc.normalize(df[['n_tokens_content']],
axis=0)
>>> df['l2_normalized'].as_matrix()
array([ 0.00152439,  0.00177498,  0.00146871, ...,  0.00307663,
        0.0047472 ,  0.00109283])
```

We can also visualize the distribution of data with different feature scaling methods (Figure 2-18). As Example 2-16 shows, unlike the log transform, feature scaling doesn't change the shape of the distribution; only the scale of the data changes.

Example 2-16. Plotting the histograms of original and scaled data

```
>>> fig, (ax1, ax2, ax3, ax4) = plt.subplots(4,1)
>>> fig.tight_layout()
>>> df['n_tokens_content'].hist(ax=ax1, bins=100)
>>> ax1.tick_params(labelsize=14)
>>> ax1.set_xlabel('Article word count', fontsize=14)
>>> ax1.set_ylabel('Number of articles', fontsize=14)

>>> df['minmax'].hist(ax=ax2, bins=100)
>>> ax2.tick_params(labelsize=14)
>>> ax2.set_xlabel('Min-max scaled word count', fontsize=14)
>>> ax2.set_ylabel('Number of articles', fontsize=14)

>>> df['standardized'].hist(ax=ax3, bins=100)
>>> ax3.tick_params(labelsize=14)
>>> ax3.set_xlabel('Standardized word count', fontsize=14)
>>> ax3.set_ylabel('Number of articles', fontsize=14)

>>> df['l2_normalized'].hist(ax=ax4, bins=100)
>>> ax4.tick_params(labelsize=14)
>>> ax4.set_xlabel('L2-normalized word count', fontsize=14)
>>> ax4.set_ylabel('Number of articles', fontsize=14)
```

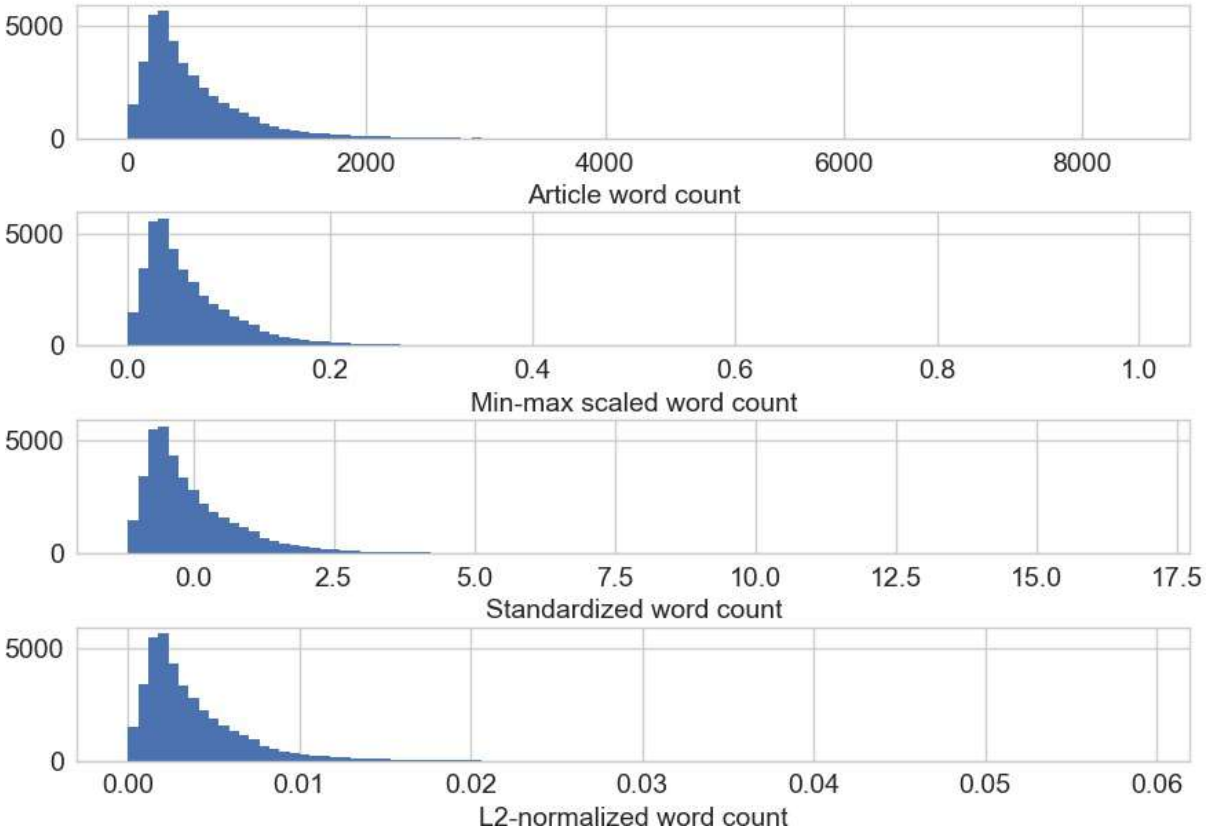

Figure 2-18. Original and scaled news article word counts—note that only the scale of the x-axis changes; the shape of the distribution stays the same with feature scaling

Feature scaling is useful in situations where a set of input features differs wildly in scale. For instance, the number of daily visitors to a popular ecommerce site might be a hundred thousand, while the actual number of sales might be in the thousands. If both of those features are thrown into a model, then the model will need to balance its scale while figuring out what to do. Drastically varying scale in input features can lead to numeric stability issues for the model training algorithm. In those situations, it's a good idea to standardize the features.

Interaction Features

A simple pairwise *interaction feature* is the product of two features. The analogy is the logical AND. It expresses the outcome in terms of pairs of conditions: "the purchase is coming from zip code 98121" AND "the user's age is between 18 and 35." Decision tree–based models get this for free, but generalized linear models often find interaction features very helpful.

A simple linear model uses a linear combination of the individual input features x_1, x_2, ... x_n to predict the outcome y:

$$y = w_1x_1 + w_2x_2 + \ldots + w_nx_n$$

An easy way to extend the linear model is to include combinations of pairs of input features, like so:

$y = w_1x_1 + w_2x_2 + ... + w_nx_n + w_{1,1}x_1x_1 + w_{1,2}x_1x_2 + w_{1,3}x_1x_3 + ...$

This allows us to capture interactions between features, and hence these pairs are called *interaction features*. If x_1 and x_2 are binary, then their product x_1x_2 is the logical function x_1 AND x_2. Suppose the problem is to predict a customer's preference based on their profile information. In our example, instead of making predictions based solely on the age or location of the user, interaction features allow the model to make predictions based on the user being of a certain age AND at a particular location.

In Example 2-17, we use pairwise interaction features from the UCI Online News Popularity dataset to predict the number of shares for each news article. As the results show, interaction features result in some lift in accuracy above singleton features. Both perform better than Example 2-9, which used as a single predictor the number of words in the body of the article (with or without a log transform).

Example 2-17. Example of interaction features in prediction

```
>>> from sklearn import linear_model
>>> from sklearn.model_selection import train_test_split
>>> import sklearn.preprocessing as preproc

# Assume df is a Pandas DataFrame containing the UCI Online News Popularity
dataset
>>> df.columns
Index(['url', 'timedelta', 'n_tokens_title', 'n_tokens_content',
       'n_unique_tokens', 'n_non_stop_words', 'n_non_stop_unique_tokens',
       'num_hrefs', 'num_self_hrefs', 'num_imgs', 'num_videos',
       'average_token_length', 'num_keywords', 'data_channel_is_lifestyle',
       'data_channel_is_entertainment', 'data_channel_is_bus',
       'data_channel_is_socmed', 'data_channel_is_tech',
       'data_channel_is_world', 'kw_min_min', 'kw_max_min', 'kw_avg_min',
       'kw_min_max', 'kw_max_max', 'kw_avg_max', 'kw_min_avg',
'kw_max_avg',
       'kw_avg_avg', 'self_reference_min_shares',
'self_reference_max_shares',
       'self_reference_avg_sharess', 'weekday_is_monday',
'weekday_is_tuesday',
       'weekday_is_wednesday', 'weekday_is_thursday', 'weekday_is_friday',
       'weekday_is_saturday', 'weekday_is_sunday', 'is_weekend', 'LDA_00',
       'LDA_01', 'LDA_02', 'LDA_03', 'LDA_04', 'global_subjectivity',
       'global_sentiment_polarity', 'global_rate_positive_words',
       'global_rate_negative_words', 'rate_positive_words',
       'rate_negative_words', 'avg_positive_polarity',
'min_positive_polarity',
       'max_positive_polarity', 'avg_negative_polarity',
       'min_negative_polarity', 'max_negative_polarity',
'title_subjectivity',
       'title_sentiment_polarity', 'abs_title_subjectivity',
       'abs_title_sentiment_polarity', 'shares'],
      dtype='object')

# Select the content-based features as singleton features in the model,
# skipping over the derived features
>>> features = ['n_tokens_title', 'n_tokens_content',
```

```
...             'n_unique_tokens', 'n_non_stop_words',
'n_non_stop_unique_tokens',
...             'num_hrefs', 'num_self_hrefs', 'num_imgs', 'num_videos',
...             'average_token_length', 'num_keywords',
'data_channel_is_lifestyle',
...             'data_channel_is_entertainment', 'data_channel_is_bus',
...             'data_channel_is_socmed', 'data_channel_is_tech',
...             'data_channel_is_world']

>>> X = df[features]
>>> y = df[['shares']]

# Create pairwise interaction features, skipping the constant bias term
>>> X2 = preproc.PolynomialFeatures(include_bias=False).fit_transform(X)
>>> X2.shape
(39644, 170)

# Create train/test sets for both feature sets
>>> X1_train, X1_test, X2_train, X2_test, y_train, y_test = \
...     train_test_split(X, X2, y, test_size=0.3, random_state=123)

>>> def evaluate_feature(X_train, X_test, y_train, y_test):
...     """Fit a linear regression model on the training set and
...     score on the test set"""
...     model = linear_model.LinearRegression().fit(X_train, y_train)
...     r_score = model.score(X_test, y_test)
...     return (model, r_score)

# Train models and compare score on the two feature sets
>>> (m1, r1) = evaluate_feature(X1_train, X1_test, y_train, y_test)
>>> (m2, r2) = evaluate_feature(X2_train, X2_test, y_train, y_test)
>>> print("R-squared score with singleton features: %0.5f" % r1)
>>> print("R-squared score with pairwise features: %0.10f" % r2)
R-squared score with singleton features: 0.00924
R-squared score with pairwise features: 0.0113276523
```

Interaction features are very simple to formulate, but they are expensive to use. The training and scoring time of a linear model with pairwise interaction features would go from $O(n)$ to $O(n^2)$, where n is the number of singleton features.

There are a few ways around the computational expense of higher-order interaction features. One could perform feature selection on top of all of the interaction features. Alternatively, one could more carefully craft a smaller number of complex features.

Both strategies have their advantages and disadvantages. Feature selection employs computational means to select the best features for a problem. (This technique is not limited to interaction features.) However, some feature selection techniques still require training multiple models with a large number of features.

Handcrafted complex features can be expressive enough that only a small number of them are needed, which reduces the training time of the model—but the features themselves may be expensive to compute, which increases the computational cost of the model scoring stage. Good examples of handcrafted (or machine-learned) complex features may be found in Chapter 8. Let’s now look at some feature selection techniques.

Feature Selection

Feature selection techniques prune away nonuseful features in order to reduce the complexity of the resulting model. The end goal is a parsimonious model that is quicker to compute, with little or no degradation in predictive accuracy. In order to arrive at such a model, some feature selection techniques require training more than one candidate model. In other words, feature selection is not about reducing training time—in fact, some techniques *increase* overall training time—but about reducing model scoring time.

Roughly speaking, feature selection techniques fall into three classes:

Filtering

> Filtering techniques preprocess features to remove ones that are unlikely to be useful for the model. For example, one could compute the correlation or mutual information between each feature and the response variable, and filter out the features that fall below a threshold. Chapter 3 discusses examples of these techniques for text features. Filtering techniques are much cheaper than the wrapper techniques described next, but they do not take into account the model being employed. Hence, they may not be able to select the right features for the model. It is best to do prefiltering conservatively, so as not to inadvertently eliminate useful features before they even make it to the model training step.

Wrapper methods

> These techniques are expensive, but they allow you to try out subsets of features, which means you won't accidentally prune away features that are uninformative by themselves but useful when taken in combination. The wrapper method treats the model as a black box that provides a quality score of a proposed subset for features. There is a separate method that iteratively refines the subset.

Embedded methods

> These methods perform feature selection as part of the model training process. For example, a decision tree inherently performs feature selection because it selects one feature on which to split the tree at each training step. Another example is the ℓ_1 regularizer, which can be added to the training objective of any linear model. The ℓ_1 regularizer encourages models that use a few features as opposed to a lot of features, so it's also known as a sparsity constraint on the model. Embedded methods incorporate feature selection as part of the model training process. They are not as powerful as wrapper methods, but they are nowhere near as expensive. Compared to filtering, embedded methods select features that are specific to the model. In this sense, embedded methods strike a balance between computational expense and quality of results.

Summary

We have discussed a number of common numeric feature engineering techniques, such as quantization, scaling (a.k.a. normalization), log transforms (a type of power transform), and interaction features, and gave a brief summary of feature selection techniques, necessary for handling large quantities of interaction features. In statistical machine learning, all data eventually boils down to numeric features. Therefore, all roads lead to some kind of numeric feature engineering technique at the end. Keep these tools handy for the end game of feature engineering!