Chapter 4. Handling Numerical Data

4.0 Introduction

Quantitative data is the measurement of something—whether class size, monthly sales, or student scores. The natural way to represent these quantities is numerically (e.g., 29 students, $529,392 in sales). In this chapter, we will cover numerous strategies for transforming raw numerical data into features purpose-built for machine learning algorithms.

4.1 Rescaling a Feature

Problem

You need to rescale the values of a numerical feature to be between two values.

Solution

Use scikit-learn's `MinMaxScaler` to rescale a feature array:

```
# Load libraries
import numpy as np
from sklearn import preprocessing

# Create feature
feature = np.array([[-500.5],
                    [-100.1],
                    [0],
                    [100.1],
                    [900.9]])

# Create scaler
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))

# Scale feature
scaled_feature = minmax_scale.fit_transform(feature)

# Show feature
scaled_feature
array([[ 0.        ],
       [ 0.28571429],
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

Discussion

Rescaling is a common preprocessing task in machine learning. Many of the algorithms

described later in this book will assume all features are on the same scale, typically 0 to 1 or –1 to 1. There are a number of rescaling techniques, but one of the simplest is called *min-max scaling*. Min-max scaling uses the minimum and maximum values of a feature to rescale values to within a range. Specifically, min-max calculates:

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

where x is the feature vector, x_i is an individual element of feature x, and x'_i is the rescaled element. In our example, we can see from the outputted array that the feature has been successfully rescaled to between 0 and 1:

```
array([[ 0.        ],
       [ 0.28571429],
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

scikit-learn's `MinMaxScaler` offers two options to rescale a feature. One option is to use `fit` to calculate the minimum and maximum values of the feature, and then use `transform` to rescale the feature. The second option is to use `fit_transform` to do both operations at once. There is no mathematical difference between the two options, but there is sometimes a practical benefit to keeping the operations separate because it allows us to apply the same transformation to different *sets* of the data.

See Also

- Feature scaling, Wikipedia
- About Feature Scaling and Normalization, Sebastian Raschka

4.2 Standardizing a Feature

Problem

You want to transform a feature to have a mean of 0 and a standard deviation of 1.

Solution

scikit-learn's `StandardScaler` performs both transformations:

```
# Load libraries
import numpy as np
from sklearn import preprocessing

# Create feature
```

```
x = np.array([[-1000.1],
              [-200.2],
              [500.5],
              [600.6],
              [9000.9]])

# Create scaler
scaler = preprocessing.StandardScaler()

# Transform the feature
standardized = scaler.fit_transform(x)

# Show feature
standardized
array([[-0.76058269],
       [-0.54177196],
       [-0.35009716],
       [-0.32271504],
       [ 1.97516685]])
```

Discussion

A common alternative to the min-max scaling discussed in Recipe 4.1 is rescaling of features to be approximately standard normally distributed. To achieve this, we use standardization to transform the data such that it has a mean, $\bar{x}$, of 0 and a standard deviation, σ, of 1. Specifically, each element in the feature is transformed so that:

$$x'_i = \frac{x_i - \bar{x}}{\sigma}$$

where x'_i is our standardized form of x_i. The transformed feature represents the number of standard deviations of the original value from the feature's mean value (also called a *z-score* in statistics).

Standardization is a common go-to scaling method for machine learning preprocessing and, in my experience, is used more often than min-max scaling. However, it depends on the learning algorithm. For example, principal component analysis often works better using standardization, while min-max scaling is often recommended for neural networks (both algorithms are discussed later in this book). As a general rule, I'd recommend defaulting to standardization unless you have a specific reason to use an alternative.

We can see the effect of standardization by looking at the mean and standard deviation of our solution's output:

```
# Print mean and standard deviation
print("Mean:", round(standardized.mean()))
print("Standard deviation:", standardized.std())
Mean: 0.0
Standard deviation: 1.0
```

If our data has significant outliers, it can negatively impact our standardization by affecting the feature's mean and variance. In this scenario, it is often helpful to instead rescale the feature

using the median and quartile range. In scikit-learn, we do this using the `RobustScaler` method:

```
# Create scaler
robust_scaler = preprocessing.RobustScaler()

# Transform feature
robust_scaler.fit_transform(x)
array([[ -1.87387612],
       [ -0.875     ],
       [  0.        ],
       [  0.125     ],
       [ 10.61488511]])
```

4.3 Normalizing Observations

Problem

You want to rescale the feature values of observations to have unit norm (a total length of 1).

Solution

Use `Normalizer` with a `norm` argument:

```
# Load libraries
import numpy as np
from sklearn.preprocessing import Normalizer

# Create feature matrix
features = np.array([[0.5, 0.5],
                     [1.1, 3.4],
                     [1.5, 20.2],
                     [1.63, 34.4],
                     [10.9, 3.3]])

# Create normalizer
normalizer = Normalizer(norm="l2")

# Transform feature matrix
normalizer.transform(features)
array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

Discussion

Many rescaling methods (e.g., min-max scaling and standardization) operate on features; however, we can also rescale across individual observations. `Normalizer` rescales the values on individual observations to have unit norm (the sum of their lengths is 1). This type of

rescaling is often used when we have many equivalent features (e.g., text classification when every word or *n*-word group is a feature).

`Normalizer` provides three norm options with Euclidean norm (often called L2) being the default argument:

$$\|x\|_2 = \sqrt{{x_1}^2 + {x_2}^2 + \cdots + {x_n}^2}$$

where x is an individual observation and x_n is that observation's value for the nth feature.

```
# Transform feature matrix
features_l2_norm = Normalizer(norm="l2").transform(features)

# Show feature matrix
features_l2_norm
array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

Alternatively, we can specify Manhattan norm (L1):

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|.$$

```
# Transform feature matrix
features_l1_norm = Normalizer(norm="l1").transform(features)

# Show feature matrix
features_l1_norm
array([[ 0.5       ,  0.5       ],
       [ 0.24444444,  0.75555556],
       [ 0.06912442,  0.93087558],
       [ 0.04524008,  0.95475992],
       [ 0.76760563,  0.23239437]])
```

Intuitively, L2 norm can be thought of as the distance between two points in New York for a bird (i.e., a straight line), while L1 can be thought of as the distance for a human walking on the street (walk north one block, east one block, north one block, east one block, etc.), which is why it is called "Manhattan norm" or "Taxicab norm."

Practically, notice that `norm="l1"` rescales an observation's values so they sum to 1, which can sometimes be a desirable quality:

```
# Print sum
print("Sum of the first observation\'s values:",
   features_l1_norm[0, 0] + features_l1_norm[0, 1])
Sum of the first observation's values: 1.0
```

4.4 Generating Polynomial and Interaction Features

Problem

You want to create polynomial and interaction features.

Solution

Even though some choose to create polynomial and interaction features manually, scikit-learn offers a built-in method:

```
# Load libraries
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Create feature matrix
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# Create PolynomialFeatures object
polynomial_interaction = PolynomialFeatures(degree=2, include_bias=False)

# Create polynomial features
polynomial_interaction.fit_transform(features)
array([[ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.]])
```

The `degree` parameter determines the maximum degree of the polynomial. For example, `degree=2` will create new features raised to the second power:

$$x_1,\ x_2,\ {x_1}^2,\ {x_1}^2,\ {x_2}^2$$

while `degree=3` will create new features raised to the second and third power:

$$x_1,\ x_2,\ {x_1}^2,\ {x_2}^2,\ {x_1}^3,\ {x_2}^3,\ {x_1}^2,\ {x_1}^3,\ {x_2}^3$$

Furthermore, by default `PolynomialFeatures` includes interaction features:

$$x_1 x_2$$

We can restrict the features created to only interaction features by setting `interaction_only` to `True`:

```
interaction = PolynomialFeatures(degree=2,
              interaction_only=True, include_bias=False)

interaction.fit_transform(features)
array([[ 2.,  3.,  6.],
       [ 2.,  3.,  6.],
```

```
       [ 2.,  3.,  6.]])
```

Discussion

Polynomial features are often created when we want to include the notion that there exists a nonlinear relationship between the features and the target. For example, we might suspect that the effect of age on the probability of having a major medical condition is not constant over time but increases as age increases. We can encode that nonconstant effect in a feature, x, by generating that feature's higher-order forms (x^2, x^3, etc.).

Additionally, often we run into situations where the effect of one feature is dependent on another feature. A simple example would be if we were trying to predict whether or not our coffee was sweet, and we had two features: (1) whether or not the coffee was stirred, and (2) whether or not we added sugar. Individually, each feature does not predict coffee sweetness, but the combination of their effects does. That is, a coffee would only be sweet if the coffee had sugar and was stirred. The effects of each feature on the target (sweetness) are dependent on each other. We can encode that relationship by including an interaction feature that is the product of the individual features.

4.5 Transforming Features

Problem

You want to make a custom transformation to one or more features.

Solution

In scikit-learn, use `FunctionTransformer` to apply a function to a set of features:

```
# Load libraries
import numpy as np
from sklearn.preprocessing import FunctionTransformer

# Create feature matrix
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# Define a simple function
def add_ten(x: int) -> int:
    return x + 10

# Create transformer
ten_transformer = FunctionTransformer(add_ten)

# Transform feature matrix
ten_transformer.transform(features)
array([[12, 13],
       [12, 13],
```

```
       [12, 13]])
```

We can create the same transformation in pandas using `apply`:

```
# Load library
import pandas as pd

# Create DataFrame
df = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# Apply function
df.apply(add_ten)
```

	feature_1	feature_2
0	12	13
1	12	13
2	12	13

Discussion

It is common to want to make some custom transformations to one or more features. For example, we might want to create a feature that is the natural log of the values of a different feature. We can do this by creating a function and then mapping it to features using either scikit-learn's `FunctionTransformer` or pandas' `apply`. In the solution we created a very simple function, `add_ten`, which added 10 to each input, but there is no reason we could not define a much more complex function.

4.6 Detecting Outliers

Problem

You want to identify extreme observations.

Solution

Detecting outliers is unfortunately more of an art than a science. However, a common method is to assume the data is normally distributed and, based on that assumption, "draw" an ellipse around the data, classifying any observation inside the ellipse as an inlier (labeled as `1`) and any observation outside the ellipse as an outlier (labeled as `-1`):

```
# Load libraries
import numpy as np
from sklearn.covariance import EllipticEnvelope
```

```
from sklearn.datasets import make_blobs

# Create simulated data
features, _ = make_blobs(n_samples = 10,
                         n_features = 2,
                         centers = 1,
                         random_state = 1)

# Replace the first observation's values with extreme values
features[0,0] = 10000
features[0,1] = 10000

# Create detector
outlier_detector = EllipticEnvelope(contamination=.1)

# Fit detector
outlier_detector.fit(features)

# Predict outliers
outlier_detector.predict(features)
array([-1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

In these arrays, values of -1 refer to outliers whereas values of 1 refer to inliers. A major limitation of this approach is the need to specify a `contamination` parameter, which is the proportion of observations that are outliers—a value that we don't know. Think of `contamination` as our estimate of the cleanliness of our data. If we expect our data to have few outliers, we can set `contamination` to something small. However, if we believe that the data is likely to have outliers, we can set it to a higher value.

Instead of looking at observations as a whole, we can instead look at individual features and identify extreme values in those features using interquartile range (IQR):

```
# Create one feature
feature = features[:,0]

# Create a function to return index of outliers
def indicies_of_outliers(x: int) -> np.array(int):
    q1, q3 = np.percentile(x, [25, 75])
    iqr = q3 - q1
    lower_bound = q1 - (iqr * 1.5)
    upper_bound = q3 + (iqr * 1.5)
    return np.where((x > upper_bound) | (x < lower_bound))

# Run function
indicies_of_outliers(feature)
(array([0]),)
```

IQR is the difference between the first and third quartile of a set of data. You can think of IQR as the spread of the bulk of the data, with outliers being observations far from the main concentration of data. Outliers are commonly defined as any value 1.5 IQRs less than the first quartile, or 1.5 IQRs greater than the third quartile.

Discussion

There is no single best technique for detecting outliers. Instead, we have a collection of techniques all with their own advantages and disadvantages. Our best strategy is often trying multiple techniques (e.g., both `EllipticEnvelope` and IQR-based detection) and looking at the results as a whole.

If at all possible, we should look at observations we detect as outliers and try to understand them. For example, if we have a dataset of houses and one feature is number of rooms, is an outlier with 100 rooms really a house or is it actually a hotel that has been misclassified?

See Also

- Three Ways to Detect Outliers (and the source of the IQR function used in this recipe)

4.7 Handling Outliers

Problem

You have outliers in your data that you want to identify and then reduce their impact on the data distribution.

Solution

Typically we can use three strategies to handle outliers. First, we can drop them:

```
# Load library
import pandas as pd

# Create DataFrame
houses = pd.DataFrame()
houses['Price'] = [534433, 392333, 293222, 4322032]
houses['Bathrooms'] = [2, 3.5, 2, 116]
houses['Square_Feet'] = [1500, 2500, 1500, 48000]

# Filter observations
houses[houses['Bathrooms'] < 20]
```

	Price	Bathrooms	Square_Feet
0	534433	2.0	1500
1	392333	3.5	2500
2	293222	2.0	1500

Second, we can mark them as outliers and include “Outlier” as a feature:

```
# Load library
import numpy as np

# Create feature based on boolean condition
houses["Outlier"] = np.where(houses["Bathrooms"] < 20, 0, 1)

# Show data
houses
```

	Price	Bathrooms	Square_Feet	Outlier
0	534433	2.0	1500	0
1	392333	3.5	2500	0
2	293222	2.0	1500	0
3	4322032	116.0	48000	1

Finally, we can transform the feature to dampen the effect of the outlier:

```
# Log feature
houses["Log_Of_Square_Feet"] = [np.log(x) for x in houses["Square_Feet"]]

# Show data
houses
```

	Price	Bathrooms	Square_Feet	Outlier	Log_Of
0	534433	2.0	1500	0	7.31322(
1	392333	3.5	2500	0	7.824046
2	293222	2.0	1500	0	7.31322(
3	4322032	116.0	48000	1	10.7789[

Discussion

Similar to detecting outliers, there is no hard-and-fast rule for handling them. How we handle them should be based on two aspects. First, we should consider what makes them outliers. If we believe they are errors in the data, such as from a broken sensor or a miscoded value, then we might drop the observation or replace outlier values with `NaN` since we can't trust those values. However, if we believe the outliers are genuine extreme values (e.g., a house [mansion] with 200 bathrooms), then marking them as outliers or transforming their values is more appropriate.

Second, how we handle outliers should be based on our goal for machine learning. For example,

if we want to predict house prices based on features of the house, we might reasonably assume the price for mansions with over 100 bathrooms is driven by a different dynamic than regular family homes. Furthermore, if we are training a model to use as part of an online home loan web application, we might assume that our potential users will not include billionaires looking to buy a mansion.

So what should we do if we have outliers? Think about why they are outliers, have an end goal in mind for the data, and, most importantly, remember that not making a decision to address outliers is itself a decision with implications.

One additional point: if you do have outliers, standardization might not be appropriate because the mean and variance might be highly influenced by the outliers. In this case, use a rescaling method more robust against outliers, like `RobustScaler`.

See Also

- `RobustScaler` documentation

4.8 Discretizating Features

Problem

You have a numerical feature and want to break it up into discrete bins.

Solution

Depending on how we want to break up the data, there are two techniques we can use. First, we can binarize the feature according to some threshold:

```
# Load libraries
import numpy as np
from sklearn.preprocessing import Binarizer

# Create feature
age = np.array([[6],
                [12],
                [20],
                [36],
                [65]])

# Create binarizer
binarizer = Binarizer(threshold=18)

# Transform feature
binarizer.fit_transform(age)
array([[0],
       [0],
       [1],
       [1],
       [1]])
```

Second, we can break up numerical features according to multiple thresholds:

```
# Bin feature
np.digitize(age, bins=[20,30,64])
array([[0],
       [0],
       [1],
       [2],
       [3]])
```

Note that the arguments for the `bins` parameter denote the left edge of each bin. For example, the `20` argument does not include the element with the value of 20, only the two values smaller than 20. We can switch this behavior by setting the parameter `right` to `True`:

```
# Bin feature
np.digitize(age, bins=[20,30,64], right=True)
array([[0],
       [0],
       [0],
       [2],
       [3]])
```

Discussion

Discretization can be a fruitful strategy when we have reason to believe that a numerical feature should behave more like a categorical feature. For example, we might believe there is very little difference in the spending habits of 19- and 20-year-olds, but a significant difference between 20- and 21-year-olds (the age in the United States when young adults can consume alcohol). In that example, it could be useful to break up individuals in our data into those who can drink alcohol and those who cannot. Similarly, in other cases it might be useful to discretize our data into three or more bins.

In the solution, we saw two methods of discretization—scikit-learn's `Binarizer` for two bins and NumPy's `digitize` for three or more bins—however, we can also use `digitize` to binarize features like `Binarizer` by specifying only a single threshold:

```
# Bin feature
np.digitize(age, bins=[18])
array([[0],
       [0],
       [1],
       [1],
       [1]])
```

See Also

- `digitize` documentation

4.9 Grouping Observations Using Clustering

Problem

You want to cluster observations so that similar observations are grouped together.

Solution

If you know that you have *k* groups, you can use k-means clustering to group similar observations and output a new feature containing each observation's group membership:

```
# Load libraries
import pandas as pd
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# Make simulated feature matrix
features, _ = make_blobs(n_samples = 50,
                         n_features = 2,
                         centers = 3,
                         random_state = 1)

# Create DataFrame
dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# Make k-means clusterer
clusterer = KMeans(3, random_state=0)

# Fit clusterer
clusterer.fit(features)

# Predict values
dataframe["group"] = clusterer.predict(features)

# View first few observations
dataframe.head(5)
```

	feature_1	feature_2	group
0	–9.877554	–3.336145	0
1	–7.287210	–8.353986	2
2	–6.943061	–7.023744	2
3	–7.440167	–8.791959	2
4	–6.641388	–8.075888	2

Discussion

We are jumping ahead of ourselves a bit and will go into much more depth about clustering algorithms later in the book. However, I wanted to point out that we can use clustering as a preprocessing step. Specifically, we use unsupervised learning algorithms like k-means to cluster observations into groups. The result is a categorical feature with similar observations being members of the same group.

Don't worry if you did not understand all of that: just file away the idea that clustering can be used in preprocessing. And if you really can't wait, feel free to flip to Chapter 19 now.

4.10 Deleting Observations with Missing Values

Problem

You need to delete observations containing missing values.

Solution

Deleting observations with missing values is easy with a clever line of NumPy:

```
# Load library
import numpy as np

# Create feature matrix
features = np.array([[1.1, 11.1],
                     [2.2, 22.2],
                     [3.3, 33.3],
                     [4.4, 44.4],
                     [np.nan, 55]])

# Keep only observations that are not (denoted by ~) missing
features[~np.isnan(features).any(axis=1)]
array([[  1.1,  11.1],
       [  2.2,  22.2],
       [  3.3,  33.3],
       [  4.4,  44.4]])
```

Alternatively, we can drop missing observations using pandas:

```
# Load library
import pandas as pd

# Load data
dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# Remove observations with missing values
dataframe.dropna()
```

	feature_1	feature_2
0	1.1	11.1

1	2.2	22.2
2	3.3	33.3
3	4.4	44.4

Discussion

Most machine learning algorithms cannot handle any missing values in the target and feature arrays. For this reason, we cannot ignore missing values in our data and must address the issue during preprocessing.

The simplest solution is to delete every observation that contains one or more missing values, a task quickly and easily accomplished using NumPy or pandas.

That said, we should be very reluctant to delete observations with missing values. Deleting them is the nuclear option, since our algorithm loses access to the information contained in the observation's nonmissing values.

Just as important, depending on the cause of the missing values, deleting observations can introduce bias into our data. There are three types of missing data:

Missing completely at random (MCAR)

> The probability that a value is missing is independent of everything. For example, a survey respondent rolls a die before answering a question: if she rolls a six, she skips that question.

Missing at random (MAR)

> The probability that a value is missing is not completely random but depends on the information captured in other features. For example, a survey asks about gender identity and annual salary, and women are more likely to skip the salary question; however, their nonresponse depends only on information we have captured in our gender identity feature.

Missing not at random (MNAR)

> The probability that a value is missing is not random and depends on information not captured in our features. For example, a survey asks about annual salary, and women are more likely to skip the salary question, and we do not have a gender identity feature in our data.

It is sometimes acceptable to delete observations if they are MCAR or MAR. However, if the value is MNAR, the fact that a value is missing is itself information. Deleting MNAR observations can inject bias into our data because we are removing observations produced by some unobserved systematic effect.

See Also

- Identifying the 3 Types of Missing Data
- Missing-Data Imputation

4.11 Imputing Missing Values

Problem

You have missing values in your data and want to impute them via a generic method or prediction.

Solution

You can impute missing values using k-nearest neighbors (KNN) or the scikit-learn `SimpleImputer` class. If you have a small amount of data, predict and impute the missing values using k-nearest neighbors:

```
# Load libraries
import numpy as np
from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

# Make a simulated feature matrix
features, _ = make_blobs(n_samples = 1000,
                         n_features = 2,
                         random_state = 1)

# Standardize the features
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)

# Replace the first feature's first value with a missing value
true_value = standardized_features[0,0]
standardized_features[0,0] = np.nan

# Predict the missing values in the feature matrix
knn_imputer = KNNImputer(n_neighbors=5)
features_knn_imputed = knn_imputer.fit_transform(standardized_features)

# Compare true and imputed values
print("True Value:", true_value)
print("Imputed Value:", features_knn_imputed[0,0])
True Value: 0.8730186114
Imputed Value: 1.09553327131
```

Alternatively, we can use scikit-learn's `SimpleImputer` class from the `imputer` module to fill in missing values with the feature's mean, median, or most frequent value. However, we will typically get worse results than with KNN:

```
# Load libraries
import numpy as np
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

# Make a simulated feature matrix
features, _ = make_blobs(n_samples = 1000,
                         n_features = 2,
                         random_state = 1)

# Standardize the features
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)

# Replace the first feature's first value with a missing value
true_value = standardized_features[0,0]
standardized_features[0,0] = np.nan

# Create imputer using the "mean" strategy
mean_imputer = SimpleImputer(strategy="mean")

# Impute values
features_mean_imputed = mean_imputer.fit_transform(features)

# Compare true and imputed values
print("True Value:", true_value)
print("Imputed Value:", features_mean_imputed[0,0])
True Value: 0.8730186114
Imputed Value: -3.05837272461
```

Discussion

There are two main strategies for replacing missing data with substitute values, each of which has strengths and weaknesses. First, we can use machine learning to predict the values of the missing data. To do this we treat the feature with missing values as a target vector and use the remaining subset of features to predict missing values. While we can use a wide range of machine learning algorithms to impute values, a popular choice is KNN. KNN is addressed in depth in Chapter 15, but the short explanation is that the algorithm uses the *k* nearest observations (according to some distance metric) to predict the missing value. In our solution we predicted the missing value using the five closest observations.

The downside to KNN is that in order to know which observations are the closest to the missing value, it needs to calculate the distance between the missing value and every single observation. This is reasonable in smaller datasets but quickly becomes problematic if a dataset has millions of observations. In such cases, approximate nearest neighbors (ANN) is a more feasible approach. We will discuss ANN in Recipe 15.5.

An alternative and more scalable strategy than KNN is to fill in the missing values of numerical data with the mean, median, or mode. For example, in our solution we used scikit-learn to fill in missing values with a feature's mean value. The imputed value is often not as close to the true value as when we used KNN, but we can scale mean-filling to data containing millions of observations more easily.

If we use imputation, it is a good idea to create a binary feature indicating whether the observation contains an imputed value.

See Also

- scikit-learn documentation: Imputation of Missing Values
- A Study of K-Nearest Neighbour as an Imputation Method