CHAPTER 4

Feature Engineering and Selection

Building Machine Learning systems and pipelines take significant effort, which is evident from the knowledge you gained in the previous chapters. In the first chapter, we presented some high-level architecture for building Machine Learning pipelines. The path from data to insights and information is not an easy and direct one. It is tough and also iterative in nature involving data scientists and analysts to reiterate through several steps multiple times to get to the perfect model and derive correct insights. The limitation of Machine Learning algorithms is the fact that they can only understand numerical values as inputs. This is because, at the heart of any algorithm, we usually have multiple mathematical equations, constraints, optimizations and computations. Hence it is almost impossible for us to feed raw data into any algorithm and expect results. This is where features and attributes are extremely helpful in building models on top of our data.

Building machine intelligence is a multi-layered process having multiple facets. In this book, so far, we have already explored how you can retrieve, process, wrangle, and visualize data. Exploratory data analysis and visualizations are the first step toward understanding your data better. Understanding your data involves understanding the complete scope encompassing your data including the domain, constraints, caveats, quality and available attributes. From Chapter 3, you might remember that data is comprised of multiple fields, attributes, or variables. Each attribute by itself is an inherent feature of the data. You can then derive further features from these inherent features and this itself forms a major part of feature engineering. Feature selection is another important task that comes hand in hand with feature engineering, where the data scientist is tasked with selecting the best possible subset of features and attributes that would help in building the right model.

An important point to remember here is that feature engineering and selection is not a one-time process which should be carried out in an ad hoc manner. The nature of building Machine Learning systems is iterative (following the *CRISP-DM* principle) and hence extracting and engineering features from the dataset is not a one-time task. You may need to extract new features and try out multiple selections each time you build a model to get the best and optimal model for your problem. Data processing and feature engineering is often described to be the toughest task or step in building any Machine Learning system by data scientists. With the need of both domain knowledge as well as mathematical transformations, feature engineering is often said to be both an art as well as a science. The obvious complexities involve dealing with diverse types of data and variables. Besides this, each Machine Learning problem or task needs specific features and there is no one solution fits all in the case of feature engineering. This makes feature engineering all the more difficult and complex.

Hence we follow a proper structured approach in this chapter covering the following three major areas in the feature engineering workflow. They are mentioned as follows.

- Feature extraction and engineering
- Feature scaling
- Feature selection

D. Sarkar et al., *Practical Machine Learning with Python*, https://doi.org/10.1007/978-1-4842-3207-1_4

This chapter covers essential concepts for all the three major areas mentioned above. Techniques for feature engineering will be covered in detail for diverse data types including numeric, categorical, temporal, text and image data. We would like to thank our good friend and fellow data scientist, Gabriel Moreira for helping us with some excellent compilations of feature engineering techniques over these diverse data types. We also cover different feature scaling methods typically used as a part of the feature engineering process to normalize values preventing higher valued features from taking unnecessary prominence. Several feature selection techniques like filter, wrapper, and embedded methods will also be covered. Techniques and concepts will be supplemented with sufficient hands-on examples and code snippets. Remember to check out the relevant code under Chapter 4 in the GitHub repository at `https://github.com/dipanjanS/practical-machine-learning-with-python` which contains necessary code, notebooks, and data. This will make things easier to understand, help you gain enough knowledge to know which technique should be used in which scenario and thus help you get started on your own journey toward feature engineering for building Machine Learning models!

Features: Understand Your Data Better

The essence of any Machine Learning model is comprised of two components namely, data and algorithms. You might remember the same from the Machine Learning paradigm which we introduced in Chapter 1. Any Machine Learning algorithm is at essence a combination of mathematical functions, equations and optimizations which are often augmented with business logic as needed. These algorithms are not intelligent enough to usually process raw data and discover latent patterns from the same which would be used to train the system. Hence we need better data representations for building Machine Learning models, which are also known as data features or attributes. Let's look at some important concepts associated with data and features in this section.

Data and Datasets

Data is essential for analytics and Machine Learning. Without data we are literally powerless to implement any intelligent system. The formal definition of data would be a collection or set of qualitative and/or quantitative variables containing values based on observations. Typically data is usually measured and collected from various observations. This is then stored it is raw form which can then be processed further and analyzed as required. Typically in any analytics or Machine Learning system, you might need multiple sources of data and processed data from one component can be fed as raw data to another component for further processing. Data can be structured having definite rows and columns indicating observations and attributes or unstructured like free textual data.

A dataset can be defined as a collection of data. Typically this indicates data present in the form of flat files like CSV files or MS Excel files, relational database tables or views, or even raw data two-dimensional matrices. Sample datasets which are quite popular in Machine Learning are available in the `scikit-learn` package to quickly get started. The `sklearn.datasets` module has these sample datasets readily available and other utilities pertaining to loading and handling datasets. You can find more details in this link `http://scikit-learn.org/stable/datasets/index.html#datasets` to learn more about the toy datasets and best practices for handling and loading data. Another popular resource for Machine Learning based datasets is the UC Irvine Machine Learning repository which can be found here `http://archive.ics.uci.edu/ml/index.php` and this contains a wide variety of datasets from real-world problems, scenarios and devices. In fact the popular Machine Learning and predictive analytics competitive platform Kaggle also features some datasets from UCI and other datasets pertaining to various competitions. Feel free to check out these resources and we will in fact be using some datasets from these resources in this chapter as well as in subsequent chapters.

Features

Raw data is hardly used to build any Machine Learning model, mostly because algorithms can't work with data which is not properly processed and wrangled in a desired format. Features are attributes or properties obtained from raw data. Each feature is a specific representation on top of the raw data. Typically, each feature is an individual measurable attribute which usually is depicted by a column in a two dimensional dataset. Each observation is depicted by a row and each feature will have a specific value for an observation. Thus each row typically indicates a feature vector and the entire set of features across all the observations forms a two-dimensional feature matrix also known as a feature set. Features are extremely important toward building Machine Learning models and each feature represents a specific chunk of representation and information from the data which is used by the model. Both quality as well as quantity of features influences the performance of the model.

Features can be of two major types based on the dataset. Inherent raw features are obtained directly from the dataset with no extra data manipulation or engineering. Derived features are usually what we obtain from feature engineering where we extract features from existing data attributes. A simple example would be creating a new feature *Age* from an employee dataset containing *Birthdate* by just subtracting their birth date from the current date. The next major section covers more details on how to handle, extract, and engineer features based on diverse data types.

Models

Features are better representations of underlying raw data which act as inputs to any Machine Learning model. Typically a model is comprised of data features, optional class labels or numeric responses for supervised learning and a Machine Learning algorithm. The algorithm is chosen based on the type of problem we want to solve after converting it into a specific Machine Learning task. Models are built after training the system on data features iteratively till we get the desired performance. Thus, a model is basically used to represent relationships among the various features of our data.

Typically the process of modeling involves multiple major steps. Model building focuses on training the model on data features. Model tuning and optimization involves tuning specific model parameters, known as hyperparameters and optimizing the model to get the best model. Model evaluation involves using standard performance evaluation metrics like accuracy to evaluate model performance. Model deployment is usually the final step where, once we have selected the most suitable model, we deploy it live in production which usually involves building an entire system around this model based on the CRISP-DM methodology. Chapter 5 will focus on these aspects in further detail.

Revisiting the Machine Learning Pipeline

We covered the standard Machine Learning pipeline in detail in Chapter 1, which was based on the CRISP-DM standard. Let's refresh our memory by looking at Figure 4-1, which depicts our standard generic Machine Learning pipeline with the major components identified with the various building blocks.

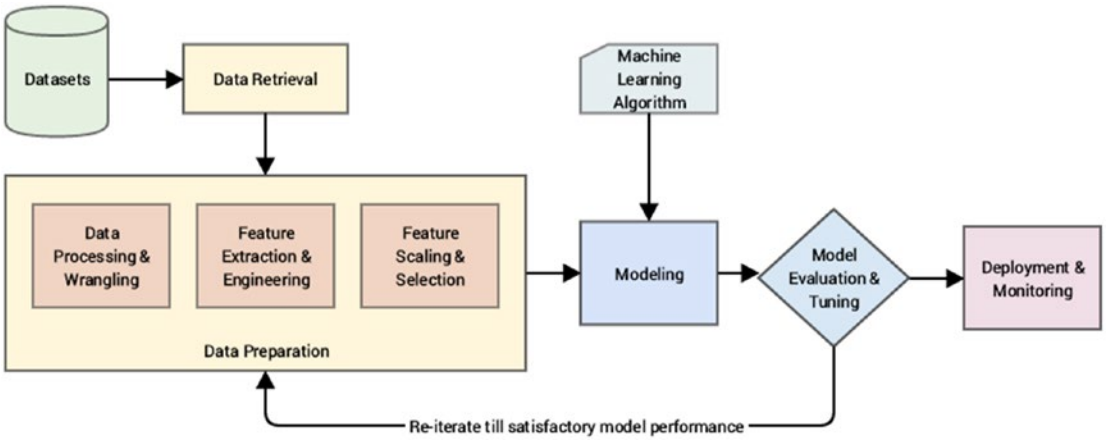

Figure 4-1. *Revisiting our standard Machine Learning pipeline*

The figure clearly depicts the main components in the pipeline, which you should already be well-versed on by now. These components are mentioned once more for ease of understanding.

- Data retrieval
- **Data preparation**
- Modeling
- Model evaluation and tuning
- Model deployment and monitoring

Our area of focus in this chapter falls under the blocks under "Data Preparation". We already covered processing and wrangling data in Chapter 3 in detail. Here, we will be focusing on the three major steps essential toward handling data features. These are mentioned as follows.

1. Feature extraction and engineering
2. Feature scaling
3. Feature selection

These blocks are highlighted in Figure 4-1 and are essential toward the process of transforming processed data into features. By processed, we mean the raw data, after going through necessary pre-processing and wrangling operations. The sequence of steps that are usually followed in the pipeline for transforming processed data into features is depicted in a more detailed view in Figure 4-2.

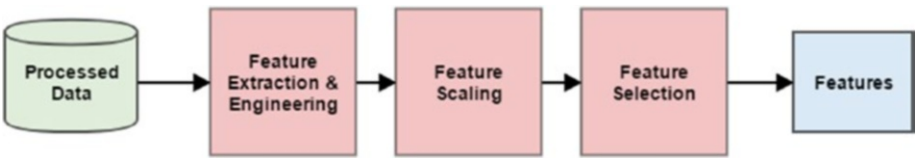

Figure 4-2. *A standard pipeline for feature engineering, scaling, and selection*

It is quite evident that based on the sequence of steps depicted in the figure, features are first crafted and engineering, necessary normalization and scaling is performed and finally the most relevant features are selected to give us the final set of features. We will cover these three components in detail in subsequent sections following the same sequence as depicted in the figure.

Feature Extraction and Engineering

The process of feature extraction and engineering is perhaps the most important one in the entire Machine Learning pipeline. Good features depicting the most suitable representations of the data help in building effective Machine Learning models. In fact, more than often it's not the algorithms but the features that determine the effectiveness of the model. In simple words, good features give good models. A data scientist approximately spends around 70% to 80% of his time in data processing, wrangling, and feature engineering for building any Machine Learning model. Hence it's of paramount importance to understand all aspects pertaining to feature engineering if you want to be proficient in Machine Learning.

Typically feature extraction and feature engineering are synonyms that indicate the process of using a combination of domain knowledge, hand-crafted techniques and mathematical transformations to convert data into features. Henceforth we will be using the term *feature engineering* to refer to all aspects concerning the task of extracting or creating new features from data. While the choice of Machine Learning algorithm is very important when building a model, more than often, the choice and number of features tend to have more impact toward the model performance. In this section, we will be looking to answer some questions such as the why, what, and how of feature engineering to get a more in-depth understanding toward feature engineering.

What Is Feature Engineering?

We already informally explained the core concept behind feature engineering, where we use specific components from domain knowledge and specific techniques to transform data into features. Data in this case is raw data after necessary pre-processing and wrangling, which we have mentioned earlier. This includes dealing with bad data, imputing missing values, transforming specific values, and so on. Features are the final end result from the process of feature engineering, which depicts various representations of the underlying data.

Let's now look at a couple of definitions and quotes relevant to feature engineering from several renowned people in the world of data science! Renowned computer and data scientist Andrew Ng talks about Machine Learning and feature engineering.

> *"Coming up with features is difficult, time-consuming, requires expert knowledge. 'Applied Machine Learning' is basically feature engineering."*
>
> —Prof. Andrew Ng

This basically reinforces what we mentioned earlier about data scientists spending close to 80% of their time in engineering features which is a difficult and time-consuming process, requiring both domain knowledge and mathematical computations. Besides this, practical or applied Machine Learning is mostly feature engineering because the time taken in building and evaluating models is considerably less than the total time spent toward feature engineering. However, this doesn't mean that modeling and evaluation are any less important than feature engineering.

We will now look at a definition of feature engineering by Dr. Jason Brownlee, data scientist and ML practitioner who provides a lot of excellent resources over at http://machinelearningmastery.com with regard to Machine Learning and data science. Dr. Brownlee defines feature engineering as follows.

> *"Feature engineering is the process of transforming* ***raw data*** *into* ***features*** *that better represent* ***the underlying problem*** *to* ***the predictive models****, resulting in improved* ***model accuracy*** *on* ***unseen data****."*
>
> —Dr. Jason Brownlee

Let's spend some more time on this definition of feature engineering. It tells us that the process of feature engineering involves transforming data into features taking into account several aspects pertaining to the problem, model, performance, and data. These aspects are highlighted in this definition and are explained in further detail as follows.

- **Raw data:** This is data in its native form after data retrieval from source. Typically some amount of data processing and wrangling is done before the actual process of feature engineering.
- **Features:** These are specific representations obtained from the raw data after the process of feature engineering.
- **The underlying problem:** This refers to the specific business problem or use-case we want to solve with the help of Machine Learning. The business problem is typically converted into a Machine Learning task.
- **The predictive models:** Typically feature engineering is used for extracting features to build Machine Learning models that learn about the data and the problem to be solved from these features. Supervised predictive models are widely used for solving diverse problems.
- **Model accuracy:** This refers to model performance metrics that are used to evaluate the model.
- **Unseen data:** This is basically new data that was not used previously to build or train the model. The model is expected to learn and generalize well for unseen data based on good quality features.

Thus feature engineering is the process of transforming data into features to act as inputs for Machine Learning models such that good quality features help in improving the overall model performance. Features are also very much dependent on the underlying problem. Thus, even though the Machine Learning task might be same in different scenarios, like classification of e-mails into spam and non-spam or classifying handwritten digits, the features extracted in each scenario will be very different from the other.

By now you must be getting a good grasp on the idea and significance of feature engineering. Always remember that for solving any Machine Learning problem, feature engineering is the key! This in fact is reinforced by Prof. Pedro Domingos from the University of Washington, in his paper titled, "A Few Useful Things to Know about Machine Learning" available at http://homes.cs.washington.edu/~pedrod/papers/cacm12.pdf, which tells us the following.

> *"At the end of the day, some Machine Learning projects succeed and some fail. What makes the difference? Easily the most important factor is the features used."*
>
> —Prof. Pedro Domingos

Feature engineering is indeed both an art and a science to transform data into features for feeding into models. Sometimes you need a combination of domain knowledge, experience, intuition, and mathematical transformations to give you the features you need. By solving more problems over time, you will gain the experience you need to know what features might be best suited for a problem. Hence do not be overwhelmed, practice will make you master feature engineering with time. The following list depicts some examples of engineering features.

- Deriving a person's age from birth date and the current date
- Getting the average and median view count of specific songs and music videos
- Extracting word and phrase occurrence counts from text documents
- Extracting pixel information from raw images
- Tabulating occurrences of various grades obtained by students

The final quote to whet your appetite on feature engineering is from renowned Kaggler, Xavier Conort. Most of you already know that tough Machine Learning problems are often posted on Kaggle regularly which is usually open to everyone. Xavier's thoughts on feature engineering are mentioned as follows.

> *"The algorithms we used are very standard for Kagglers. …We spent most of our efforts in feature engineering. …We were also very careful to discard features likely to expose us to the risk of over-fitting our model."*
>
> —Xavier Conort

This should give you a good idea what is feature engineering, the various aspects surrounding it and a very basic introduction into why do we really need feature engineering. In the following section, we will expand more on why we need feature engineering, its benefits and advantages.

Why Feature Engineering?

We have defined feature engineering in the previous section and also touched upon the basics pertaining to the importance of feature engineering. Let's now look at why we need feature engineering and how can it be an advantage for us when we are building Machine Learning models and working with data.

- **Better representation of data**: Features are basically various representations of the underlying raw data. These representations can be better understood by Machine Learning algorithms. Besides this, we can also often easily visualize these representations. A simple example would be to visualize the frequent word occurrences of a newspaper article as opposed to being totally perplexed as to what to do with the raw text!
- **Better performing models**: The right features tend to give models that outperform other models no matter how complex the algorithm is. In general if you have the right feature set, even a simple model will perform well and give desired results. In short, better features make better models.
- **Essential for model building and evaluation**: We have mentioned this numerous times by now, raw data cannot be used to build Machine Learning models. Get your data, extract features, and start building models! Also on evaluating model performance and tuning the models, you can reiterate over your feature set to choose the right set of features to get the best model.

- **More flexibility on data types:** While is it definitely easier to use numeric data types directly with Machine Learning algorithms with little or no data transformations, the real challenge is to build models on more complex data types like text, images, and even videos. Feature engineering helps us build models on diverse data types by applying necessary transformations and enables us to work even on complex unstructured data.
- **Emphasis on the business and domain:** Data scientists and analysts are usually busy in processing, cleaning data and building models as a part of their day to day tasks. This often creates a gap between the business stakeholders and the technical/analytics team. Feature engineering involves and enables data scientists to take a step back and try to understand the domain and the business better, by taking valuable inputs from the business and subject matter experts. This is necessary to create and select features that might be useful for building the right model to solve the problem. Pure statistical and mathematical knowledge is rarely sufficient to solve a complex real-world problem. Hence feature engineering emphasizes to focus on the business and the domain of the problem when building features.

This list, though not an exhaustive one, gives us a pretty good insight into the importance of feature engineering and how it is an essential aspect of building Machine Learning models. The importance of the problem to be solved and the domain is also pretty important in feature engineering.

How Do You Engineer Features?

There are no fixed rules for engineering features. It involves using a combination of domain knowledge, business constraints, hand-crafted transformations and mathematical transformations to transform the raw data into desired features. Different data types have different techniques for feature extraction. Hence in this chapter, we focus on various feature engineering techniques and strategies for the following major data types.

- Numeric data
- Categorical data
- Text data
- Temporal data
- Image data

Subsequent sections in this chapter focus on dealing with these diverse data types and specific techniques which can be applied to engineer features. You can use them as a reference and guidebook for engineering features from your own datasets in the future.

Another aspect into feature engineering has recently gained prominence. Here, you do not use hand-crafted features but, make the machine itself try to detect patterns and extract useful data representations from the raw data, which can be used as features. This process is also known as auto feature generation. Deep Learning has proved to be extremely effective in this area and neural network architectures like convolutional neural networks (CNNs), recurrent neural networks (RNNs), and Long Short Term Memory networks (LSTMs) are extensively used for auto feature engineering and extraction. Let's dive into the world of feature engineering now with some real-world datasets and examples.

Feature Engineering on Numeric Data

Numeric data, fields, variables, or features typically represent data in the form of scalar information that denotes an observation, recording, or measurement. Of course, numeric data can also be represented as a vector of scalars where each specific entity in the vector is a numeric data point in itself. Integers and floats are the most common and widely used numeric data types. Besides this, numeric data is perhaps the easiest to process and is often used directly by Machine Learning models. If you remember we have talked about numeric data previously in the "Data Description" section in Chapter 3.

Even though numeric data can be directly fed into Machine Learning models, you would still need to engineer features that are relevant to the scenario, problem, and domain before building a model. Hence the need for feature engineering remains. Important aspects of numeric features include feature scale and distribution and you will observe some of these aspects in the examples in this section. In some scenarios, we need to apply specific transformations to change the scale of numeric values and in other scenarios we need to change the overall distribution of the numeric values, like transforming a skewed distribution to a normal distribution.

The code used for this section is available in the code files for this chapter. You can load `feature_engineering_numeric.py` directly and start running the examples or use the jupyter notebook, `Feature Engineering on Numeric Data.ipynb`, for a more interactive experience. Before we begin, let's load the following dependencies and configuration settings.

```
In [1]: import pandas as pd
   ...: import matplotlib.pyplot as plt
   ...: import matplotlib as mpl
   ...: import numpy as np
   ...: import scipy.stats as spstats
   ...:
   ...: %matplotlib inline
   ...: mpl.style.reload_library()
   ...: mpl.style.use('classic')
   ...: mpl.rcParams['figure.facecolor'] = (1, 1, 1, 0)
   ...: mpl.rcParams['figure.figsize'] = [6.0, 4.0]
   ...: mpl.rcParams['figure.dpi'] = 100
```

Now that we have the initial dependencies loaded, let's look at some ways to engineer features from numeric data in the following sections.

Raw Measures

Just like we mentioned earlier, numeric features can be directly fed to Machine Learning models often since they are in a format which can be easily understood, interpreted, and operated on. Raw measures typically indicated using numeric variables directly as features without any form of transformation or engineering. Typically these features can indicate values or counts.

Values

Usually, scalar values in its raw form indicate a specific measurement, metric, or observation belonging to a specific variable or field. The semantics of this field is usually obtained from the field name itself or a data dictionary if present. Let's load a dataset now about Pokémon! This dataset is also available on Kaggle. If you do not know, Pokémon is a huge media franchise surrounding fictional characters called Pokémon which stands for pocket monsters. In short, you can think of them as fictional animals with superpowers! The following snippet gives us an idea about this dataset.

```
In [2]: poke_df = pd.read_csv('datasets/Pokemon.csv', encoding='utf-8')
   ...: poke_df.head()
```

Out[2]:

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	1	False
3	3	VenusaurMega Venusaur	Grass	Poison	625	80	100	123	122	120	80	1	False
4	4	Charmander	Fire	NaN	309	39	52	43	60	50	65	1	False

Figure 4-3. *Raw data from the Pokémon dataset*

If you observe the dataset depicted in Figure 4-3, there are several attributes there which represent numeric raw values which can be used directly. The following snippet depicts some of these features with more emphasis.

```
In [3]: poke_df[['HP', 'Attack', 'Defense']].head()
Out[3]:
   HP  Attack  Defense
0  45      49       49
1  60      62       63
2  80      82       83
3  80     100      123
4  39      52       43
```

You can directly use these attributes as features that are depicted in the previous dataframe. These include each Pokémon's HP (Hit Points), Attack, and Defense stats. In fact, we can also compute some basic statistical measures on these fields using the following code.

```
In [4]: poke_df[['HP', 'Attack', 'Defense']].describe()
Out[4]:
                HP      Attack     Defense
count  800.000000  800.000000  800.000000
mean    69.258750   79.001250   73.842500
std     25.534669   32.457366   31.183501
min      1.000000    5.000000    5.000000
25%     50.000000   55.000000   50.000000
50%     65.000000   75.000000   70.000000
75%     80.000000  100.000000   90.000000
max    255.000000  190.000000  230.000000
```

We can see multiple statistical measures like count, average, standard deviation, and quartiles for each of the numeric features in this output. Try plotting their distributions if possible!

Counts

Raw numeric measures can also indicate counts, frequencies and occurrences of specific attributes. Let's look at a sample of data from the million-song dataset, which depicts counts or frequencies of songs that have been heard by various users.

```
In [5]: popsong_df = pd.read_csv('datasets/song_views.csv', encoding='utf-8')
   ...: popsong_df.head(10)
```

Out[5]:

	user_id	song_id	title	listen_count
0	b6b799f34a204bd928ea014c243ddad6d0be4f8f	SOBONKR12A58A7A7E0	You're The One	2
1	b41ead730ac14f6b6717b9cf8859d5579f3f8d4d	SOBONKR12A58A7A7E0	You're The One	0
2	4c84359a164b161496d05282707cecbd50adbfc4	SOBONKR12A58A7A7E0	You're The One	0
3	779b5908593756abb6ff7586177c966022668b06	SOBONKR12A58A7A7E0	You're The One	0
4	dd88ea94f605a63d9fc37a214127e3f00e85e42d	SOBONKR12A58A7A7E0	You're The One	0
5	68f0359a2f1cedb0d15c98d88017281db79f9bc6	SOBONKR12A58A7A7E0	You're The One	0
6	116a4c95d63623a967edf2f3456c90ebbf964e6f	SOBONKR12A58A7A7E0	You're The One	17
7	45544491ccfcdc0b0803c34f201a6287ed4e30f8	SOBONKR12A58A7A7E0	You're The One	0
8	e701a24d9b6c59f5ac37ab28462ca82470e27cfb	SOBONKR12A58A7A7E0	You're The One	68
9	edc8b7b1fd592a3b69c3d823a742e1a064abec95	SOBONKR12A58A7A7E0	You're The One	0

Figure 4-4. *Song listen counts as a numeric feature*

We can see that the `listen_count` field in the data depicted in Figure 4-4 can be directly used as a count/frequency based numeric feature.

Binarization

Often raw numeric frequencies or counts are not necessary in building models especially with regard to methods applied in building recommender engines. For example if I want to know if a person is interested or has listened to a particular song, I do not need to know the total number of times he/she has listened to the same song. I am more concerned about the various songs he/she has listened to. In this case, a binary feature is preferred as opposed to a count based feature. We can binarize our `listen_count` field from our earlier dataset in the following way.

```
In [6]: watched = np.array(popsong_df['listen_count'])
   ...: watched[watched >= 1] = 1
   ...: popsong_df['watched'] = watched
```

You can also use `scikit-learn`'s `Binarizer` class here from its preprocessing module to perform the same task instead of `numpy` arrays, as depicted in the following code.

```
In [7]: from sklearn.preprocessing import Binarizer
   ...:
   ...: bn = Binarizer(threshold=0.9)
   ...: pd_watched = bn.transform([popsong_df['listen_count']])[0]
   ...: popsong_df['pd_watched'] = pd_watched
   ...: popsong_df.head(11)
```

Out[7]:

	user_id	song_id	title	listen_count	watched	pd_watched
0	b6b799f34a204bd928ea014c243ddad6d0be4f8f	SOBONKR12A58A7A7E0	You're The One	2	1	1
1	b41ead730ac14f6b6717b9cf8859d5579f3f8d4d	SOBONKR12A58A7A7E0	You're The One	0	0	0
2	4c84359a164b161496d05282707cecbd50adbfc4	SOBONKR12A58A7A7E0	You're The One	0	0	0
3	779b5908593756abb6ff7586177c966022668b06	SOBONKR12A58A7A7E0	You're The One	0	0	0
4	dd88ea94f605a63d9fc37a214127e3f00e85e42d	SOBONKR12A58A7A7E0	You're The One	0	0	0
5	68f0359a2f1cedb0d15c98d88017281db79f9bc6	SOBONKR12A58A7A7E0	You're The One	0	0	0
6	116a4c95d63623a967edf2f3456c90ebbf964e6f	SOBONKR12A58A7A7E0	You're The One	17	1	1
7	45544491ccfcdc0b0803c34f201a6287ed4e30f8	SOBONKR12A58A7A7E0	You're The One	0	0	0
8	e701a24d9b6c59f5ac37ab28462ca82470e27cfb	SOBONKR12A58A7A7E0	You're The One	68	1	1
9	edc8b7b1fd592a3b69c3d823a742e1a064abec95	SOBONKR12A58A7A7E0	You're The One	0	0	0
10	fb41d1c374d093ab643ef3bcd70eeb258d479076	SOBONKR12A58A7A7E0	You're The One	1	1	1

Figure 4-5. *Binarizing song counts*

You can clearly see from Figure 4-5 that both the methods have produced the same results depicted in features `watched` and `pd_watched`. Thus, we have the song listen counts as a binarized feature indicating if the song was listened to or not by each user.

Rounding

Often when dealing with numeric attributes like proportions or percentages, we may not need values with a high amount of precision. Hence it makes sense to round off these high precision percentages into numeric integers. These integers can then be directly used as raw numeric values or even as categorical (discrete-class based) features. Let's try applying this concept in a dummy dataset depicting store items and their popularity percentages.

```
In [8]: items_popularity = pd.read_csv('datasets/item_popularity.csv', encoding='utf-8')
   ...: # rounding off percentages
   ...: items_popularity['popularity_scale_10'] =
                    np.array(np.round((items_popularity['pop_percent'] * 10)), dtype='int')
   ...: items_popularity['popularity_scale_100'] =
                    np.array(np.round((items_popularity['pop_percent'] * 100)), dtype='int')
   ...: items_popularity
Out[8]:
    item_id  pop_percent  popularity_scale_10  popularity_scale_100
0  it_01345      0.98324                   10                    98
1  it_03431      0.56123                    6                    56
2  it_04572      0.12098                    1                    12
3  it_98021      0.35476                    4                    35
```

```
4  it_01298      0.92101                  9                  92
5  it_90120      0.81212                  8                  81
6  it_10123      0.56502                  6                  57
```

Thus after our rounding operations, you can see the new features in the data depicted in the previous dataframe. Basically we tried two forms of rounding. The features depict the item popularities now both on a scale of `1-10` and on a scale of `1-100`. You can use these values both as numerical or categorical features based on the scenario and problem.

Interactions

A model is usually built in such a way that we try to model the output responses (discrete classes or continuous values) as a function of the input feature variables. For example, a simple linear regression equation can be depicted as $y = c_1x_1 + c_2x_2 + ... + c_nx_n$ where the input features are depicted by variables $\{x_1, x_2, ... x_n\}$ having weights or coefficients of $\{c_1, c_2, ... c_n\}$ respectively and the goal is the predict response y. In this case, this simple linear model depicts the relationship between the output and inputs, purely based on the individual, separate input features.

However, often in several real-world datasets and scenarios, it makes sense to also try to capture the interactions between these feature variables as a part of the input feature set. A simple depiction of the extension of the above linear regression formulation with interaction features would be $y = c_1x_1 + c_2x_2 + ... + c_nx_n + c_{11}x_1^2 + c_{22}x_2^2 + c_{12}x_1x_2 + ...$ where features like $\{x_1x_2, x_1^2, ...\}$ denote the interaction features. Let's try engineering some interaction features on our Pokémon dataset now.

```
In [9]: atk_def = poke_df[['Attack', 'Defense']]
   ...: atk_def.head()
Out[9]:
   Attack  Defense
0      49       49
1      62       63
2      82       83
3     100      123
4      52       43
```

We can see in this output, the two numeric features depicting Pokémon attack and defense. The following code helps us build interaction features from these two features. We will build features up to the second degree using the `PolynomialFeatures` class from `scikit-learn`'s API.

```
In [10]: from sklearn.preprocessing import PolynomialFeatures
    ...:
    ...: pf = PolynomialFeatures(degree=2, interaction_only=False, include_bias=False)
    ...: res = pf.fit_transform(atk_def)
    ...: res
Out[10]:
array([[    49.,     49.,   2401.,   2401.,   2401.],
       [    62.,     63.,   3844.,   3906.,   3969.],
       [    82.,     83.,   6724.,   6806.,   6889.],
       ...,
       [   110.,     60.,  12100.,   6600.,   3600.],
       [   160.,     60.,  25600.,   9600.,   3600.],
       [   110.,    120.,  12100.,  13200.,  14400.]])
```

We can clearly see from this output that we have a total of five features including the new interaction features. We can see the degree of each feature in the matrix, using the following snippet.

```
In [11]: pd.DataFrame(pf.powers_, columns=['Attack_degree', 'Defense_degree'])
Out[11]:
   Attack_degree  Defense_degree
0              1               0
1              0               1
2              2               0
3              1               1
4              0               2
```

Now that we know what each feature actually represented from the degrees depicted, we can assign a name to each feature as follows to get the updated feature set.

```
In [12]: intr_features = pd.DataFrame(res,
    ...:                              columns=['Attack', 'Defense',
    ...:                                       'Attack^2', 'Attack x Defense', 'Defense^2'])
    ...: intr_features.head(5)
Out[12]:
   Attack  Defense  Attack^2  Attack x Defense  Defense^2
0    49.0     49.0    2401.0            2401.0     2401.0
1    62.0     63.0    3844.0            3906.0     3969.0
2    82.0     83.0    6724.0            6806.0     6889.0
3   100.0    123.0   10000.0           12300.0    15129.0
4    52.0     43.0    2704.0            2236.0     1849.0
```

Thus we can see our original and interaction features in Figure 4-10. The `fit_transform(...)` API function from `scikit-learn` is useful to build a feature engineering representation object on the training data, which can be reused on new data during model predictions by calling on the `transform(...)` function. Let's take some sample new observations for Pokémon attack and defense features and try to transform them using this same mechanism.

```
In [13]: new_df = pd.DataFrame([[95, 75],[121, 120], [77, 60]],
    ...:                        columns=['Attack', 'Defense'])
    ...: new_df
Out[13]:
   Attack  Defense
0      95       75
1     121      120
2      77       60
```

We can now use the `pf` object that we created earlier and transform these input features to give us the interaction features as follows.

```
In [14]: new_res = pf.transform(new_df)
    ...: new_intr_features = pd.DataFrame(new_res,
    ...:                                  columns=['Attack', 'Defense',
    ...:                                           'Attack^2', 'Attack x Defense', 'Defense^2'])
    ...: new_intr_features
Out[14]:
```

```
   Attack  Defense  Attack^2  Attack x Defense  Defense^2
0    95.0     75.0    9025.0            7125.0     5625.0
1   121.0    120.0   14641.0           14520.0    14400.0
2    77.0     60.0    5929.0            4620.0     3600.0
```

Thus you can see that we have successfully obtained the necessary interaction features for the new dataset. Try building interaction features on three or more features now!

Binning

Often when working with numeric data, you might come across features or attributes which depict raw measures such as values or frequencies. In many cases, often the distributions of these attributes are skewed in the sense that some sets of values will occur a lot and some will be very rare. Besides that, there is also the added problem of varying range of these values. Suppose we are talking about song or video view counts. In some cases, the view counts will be abnormally large and in some cases very small. Directly using these features in modeling might cause issues. Metrics like similarity measures, cluster distances, regression coefficients and more might get adversely affected if we use raw numeric features having values which range across multiple orders of magnitude. There are various ways to engineer features from these raw values so we can these issues. These methods include transformations, scaling and binning/quantization.

In this section, we will talk about binning which is also known as quantization. The operation of binning is used for transforming continuous numeric values into discrete ones. These discrete numbers can be thought of as bins into which the raw values or numbers are binned or grouped into. Each bin represents a specific degree of intensity and has a specific range of values which must fall into that bin. There are various ways of binning data which include fixed-width and adaptive binning. Specific techniques can be employed for each binning process. We will use a dataset extracted from the 2016 FreeCodeCamp Developer/Coder survey which talks about various attributes pertaining to coders and software developers. You can check it out yourself at https://github.com/freeCodeCamp/2016-new-coder-survey for more details. Let's load the dataset and take a peek at some interesting attributes.

```
In [15]: fcc_survey_df = pd.read_csv('datasets/fcc_2016_coder_survey_subset.csv',
                                    encoding='utf-8')
    ...: fcc_survey_df[['ID.x', 'EmploymentField', 'Age', 'Income']].head()
```

Out[15]:

	ID.x	EmploymentField	Age	Income
0	cef35615d61b202f1dc794ef2746df14	office and administrative support	28.0	32000.0
1	323e5a113644d18185c743c241407754	food and beverage	22.0	15000.0
2	b29a1027e5cd062e654a63764157461d	finance	19.0	48000.0
3	04a11e4bcb573a1261eb0d9948d32637	arts, entertainment, sports, or media	26.0	43000.0
4	9368291c93d5d5f5c8cdb1a575e18bec	education	20.0	6000.0

Figure 4-6. *Important attributes from the FCC coder survey dataset*

The dataframe depicted in Figure 4-6 shows us some interesting attributes of the coder survey dataset, some of which we will be analyzing in this section. The `ID.x` variable is basically a unique identifier for each coder/developer who took the survey and the other fields are pretty self-explanatory.

Fixed-Width Binning

In fixed-width binning, as the name indicates, we have specific fixed widths for each of the bins, which are usually pre-defined by the user analyzing the data. Each bin has a pre-fixed range of values which should be assigned to that bin on the basis of some business or custom logic, rules, or necessary transformations.

Binning based on rounding is one of the ways, where you can use the rounding operation that we discussed earlier to bin raw values. Let's consider the Age feature from the coder survey dataset. The following code shows the distribution of developer ages who took the survey.

```
In [16]: fig, ax = plt.subplots()
    ...: fcc_survey_df['Age'].hist(color='#A9C5D3')
    ...: ax.set_title('Developer Age Histogram', fontsize=12)
    ...: ax.set_xlabel('Age', fontsize=12)
    ...: ax.set_ylabel('Frequency', fontsize=12)
```

```
Out[16]:
```

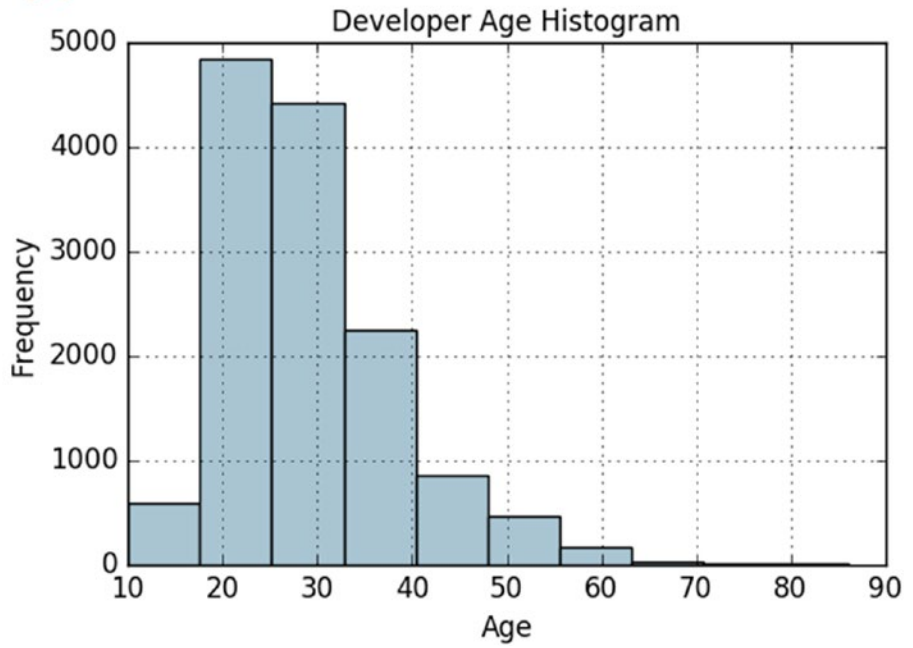

Figure 4-7. *Histogram depicting developer age distribution*

The histogram in Figure 4-7 depicts the distribution of developer ages, which is slightly right skewed as expected. Let's try to assign these raw age values into specific bins based on the following logic.

```
Age Range: Bin
---------------
 0 -  9  : 0
10 - 19  : 1
20 - 29  : 2
30 - 39  : 3
```

```
40 - 49  : 4
50 - 59  : 5
60 - 69  : 6
  ... and so on
```

We can easily do this using what we learned in the "Rounding" section earlier where we round off these raw age values by taking the floor value after dividing it by 10. The following code depicts the same.

```
In [17]: fcc_survey_df['Age_bin_round'] = np.array(np.floor(np.array(fcc_survey_df['Age']) /
                                                                                    10.))
    ...: fcc_survey_df[['ID.x', 'Age', 'Age_bin_round']].iloc[1071:1076]
Out[17]:
                                ID.x   Age  Age_bin_round
1071  6a02aa4618c99fdb3e24de522a099431  17.0            1.0
1072  f0e5e47278c5f248fe861c5f7214c07a  38.0            3.0
1073  6e14f6d0779b7e424fa3fdd9e4bd3bf9  21.0            2.0
1074  c2654c07dc929cdf3dad4d1aec4ffbb3  53.0            5.0
1075  f07449fc9339b2e57703ec7886232523  35.0            3.0
```

We take a specific slice of the dataset (rows 1071-1076) to depict users of varying ages. You can see the corresponding bins for each age have been assigned based on rounding. But what if we need more flexibility? What if I want to decide and fix the bin widths myself?

Binning based on custom ranges is the answer to the all our questions about fixed-width binning, some of which I just mentioned. Let's define some custom age ranges for binning developer ages using the following scheme.

```
Age Range : Bin
---------------
 0 -  15  : 1
16 -  30  : 2
31 -  45  : 3
46 -  60  : 4
61 -  75  : 5
75 - 100  : 6
```

Based on this custom binning scheme, we will now label the bins for each developer age value with the help of the following code. We will store both the bin range as well as the corresponding label.

```
In [18]: bin_ranges = [0, 15, 30, 45, 60, 75, 100]
    ...: bin_names = [1, 2, 3, 4, 5, 6]
    ...: fcc_survey_df['Age_bin_custom_range'] = pd.cut(np.array(fcc_survey_df['Age']),
    ...:                                                bins=bin_ranges)
    ...: fcc_survey_df['Age_bin_custom_label'] = pd.cut(np.array(fcc_survey_df['Age']),
    ...:                                                bins=bin_ranges, labels=bin_names)
    ...: fcc_survey_df[['ID.x', 'Age', 'Age_bin_round',
    ...:                'Age_bin_custom_range', 'Age_bin_custom_label']].iloc[1071:1076]
```

Out[18]:

	ID.x	Age	Age_bin_round	Age_bin_custom_range	Age_bin_custom_label
1071	6a02aa4618c99fdb3e24de522a099431	17.0	1.0	(15, 30]	2
1072	f0e5e47278c5f248fe861c5f7214c07a	38.0	3.0	(30, 45]	3
1073	6e14f6d0779b7e424fa3fdd9e4bd3bf9	21.0	2.0	(15, 30]	2
1074	c2654c07dc929cdf3dad4d1aec4ffbb3	53.0	5.0	(45, 60]	4
1075	f07449fc9339b2e57703ec7886232523	35.0	3.0	(30, 45]	3

Figure 4-8. *Custom age binning for developer ages*

We can see from the dataframe output in Figure 4-8 that the custom bins based on our scheme have been assigned for each developer's age. Try out some of your own binning schemes!

Adaptive Binning

So far, we have decided the bin width and ranges in fixed-width binning. However, this technique can lead to irregular bins that are not uniform based on the number of data points or values which fall in each bin. Some of the bins might be densely populated and some of them might be sparsely populated or even be empty! Adaptive binning is a safer and better approach where we use the data distribution itself to decide what should be the appropriate bins.

Quantile based binning is a good strategy to use for adaptive binning. Quantiles are specific values or cut-points which help in partitioning the continuous valued distribution of a specific numeric field into discrete contiguous bins or intervals. Thus, *q-Quantiles* help in partitioning a numeric attribute into *q* equal partitions. Popular examples of quantiles include the *2-Quantile* known as the median which divides the data distribution into two equal bins, *4-Quantiles* known as the quartiles, which divide the data into four equal bins and *10-Quantiles* also known as the deciles which create 10 equal width bins. Let's now look at a slice of data pertaining to developer income values in our coder survey dataset.

```
In [19]: fcc_survey_df[['ID.x', 'Age', 'Income']].iloc[4:9]
Out[19]:
                              ID.x   Age   Income
4  9368291c93d5d5f5c8cdb1a575e18bec  20.0   6000.0
5  dd0e77eab9270e4b67c19b0d6bbf621b  34.0  40000.0
6  7599c0aa0419b59fd11ffede98a3665d  23.0  32000.0
7  6dff182db452487f07a47596f314bddc  35.0  40000.0
8  9dc233f8ed1c6eb2432672ab4bb39249  33.0  80000.0
```

The slice of data depicted by the dataframe shows us the income values for each developer in our dataset. Let's look at the whole data distribution for this `Income` variable now using the following code.

```
In [20]: fig, ax = plt.subplots()
    ...: fcc_survey_df['Income'].hist(bins=30, color='#A9C5D3')
    ...: ax.set_title('Developer Income Histogram', fontsize=12)
    ...: ax.set_xlabel('Developer Income', fontsize=12)
    ...: ax.set_ylabel('Frequency', fontsize=12)
```

Out[20]:

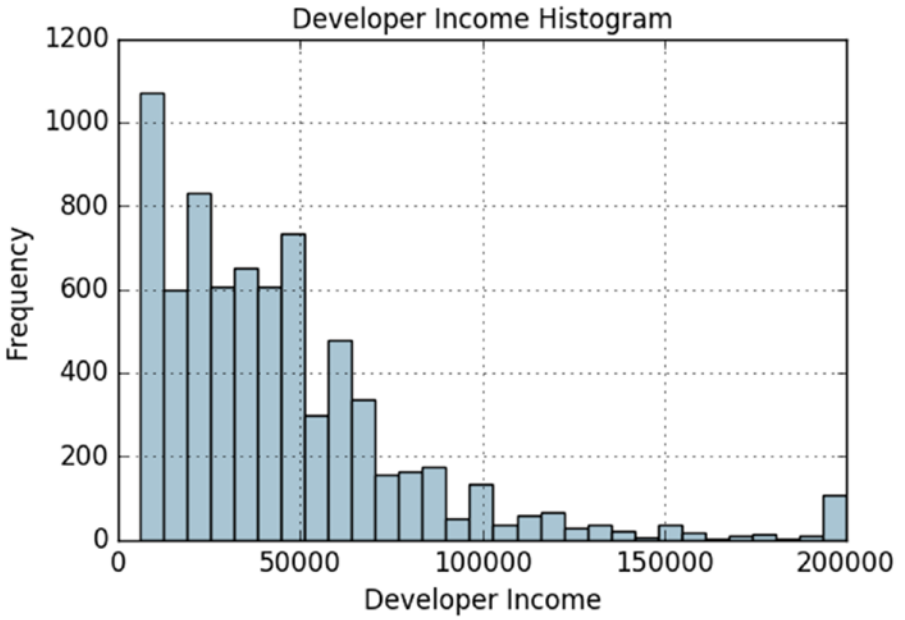

Figure 4-9. *Histogram depicting developer income distribution*

We can see from the distribution depicted in Figure 4-9 that as expected there is a right skew with lesser developers earning more money and vice versa. Let's take a *4-Quantile* or a quartile based adaptive binning scheme. The following snippet helps us obtain the income values that fall on the four quartiles in the distribution.

```
In [21]: quantile_list = [0, .25, .5, .75, 1.]
    ...: quantiles = fcc_survey_df['Income'].quantile(quantile_list)
    ...: quantiles
Out[21]:
0.00      6000.0
0.25     20000.0
0.50     37000.0
0.75     60000.0
1.00    200000.0
```

To visualize the quartiles obtained in this output better, we can plot them in our data distribution using the following code snippet.

```
In [22]: fig, ax = plt.subplots()
    ...: fcc_survey_df['Income'].hist(bins=30, color='#A9C5D3')
    ...:
    ...: for quantile in quantiles:
    ...:     qvl = plt.axvline(quantile, color='r')
    ...: ax.legend([qvl], ['Quantiles'], fontsize=10)
    ...:
    ...: ax.set_title('Developer Income Histogram with Quantiles', fontsize=12)
    ...: ax.set_xlabel('Developer Income', fontsize=12)
    ...: ax.set_ylabel('Frequency', fontsize=12)
```

Out[22]:

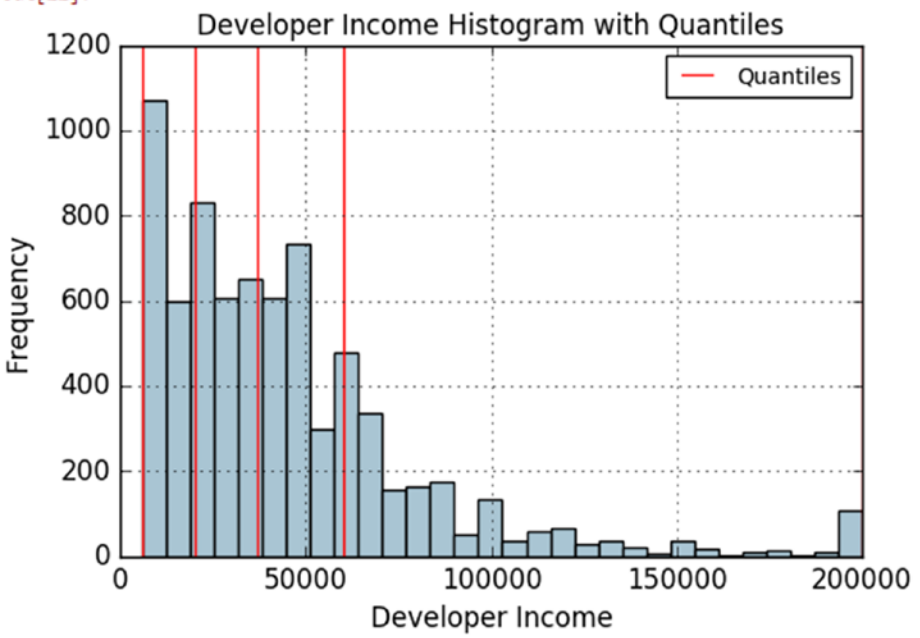

Figure 4-10. *Histogram depicting developer income distribution with quartile values*

The *4-Quantile* values for the income attribute are depicted by red vertical lines in Figure 4-10. Let's now use quantile binning to bin each of the developer income values into specific bins using the following code.

```
In [23]: quantile_labels = ['0-25Q', '25-50Q', '50-75Q', '75-100Q']
    ...: fcc_survey_df['Income_quantile_range'] = pd.qcut(fcc_survey_df['Income'],
    ...:                                                  q=quantile_list)
    ...: fcc_survey_df['Income_quantile_label'] = pd.qcut(fcc_survey_df['Income'],
    ...:                                                  q=quantile_list,
    ...:                                                  labels=quantile_labels)
    ...: fcc_survey_df[['ID.x', 'Age', 'Income',
                        'Income_quantile_range', 'Income_quantile_label']].iloc[4:9]
```

Out[23]:

	ID.x	Age	Income	Income_quantile_range	Income_quantile_label
4	9368291c93d5d5f5c8cdb1a575e18bec	20.0	6000.0	(5999.999, 20000.0]	0-25Q
5	dd0e77eab9270e4b67c19b0d6bbf621b	34.0	40000.0	(37000.0, 60000.0]	50-75Q
6	7599c0aa0419b59fd11ffede98a3665d	23.0	32000.0	(20000.0, 37000.0]	25-50Q
7	6dff182db452487f07a47596f314bddc	35.0	40000.0	(37000.0, 60000.0]	50-75Q
8	9dc233f8ed1c6eb2432672ab4bb39249	33.0	80000.0	(60000.0, 200000.0]	75-100Q

Figure 4-11. *Quantile based bin ranges and labels for developer incomes*

The result dataframe depicted in Figure 4-11 clearly shows the quantile based bin range and corresponding label assigned for each developer income value in the `Income_quantile_range` and `Income_quantile_labels` features, respectively.

Statistical Transformations

Let's look at a different strategy of feature engineering on numerical data by using statistical or mathematical transformations. In this section, we will look at the Log transform as well as the Box-Cox transform. Both of these transform functions belong to the Power Transform family of functions. These functions are typically used to create monotonic data transformations, but their main significance is that they help in stabilizing variance, adhering closely to the normal distribution and making the data independent of the mean based on its distribution. Several transformations are also used as a part of feature scaling, which we cover in a future section.

Log Transform

The log transform belongs to the power transform family of functions. This function can be defined as $y = \log_b(x)$ which reads as log of x to the base b is equal to y. This translates to $b^y = x$, which indicates as to what power must the base b be raised to in order to get x. The natural logarithm uses the base $b = e$ where $e = 2.71828$ popularly known as Euler's number. You can also use base $b = 10$ used popularly in the decimal system. Log transforms are useful when applied to skewed distributions as they tend to expand the values which fall in the range of lower magnitudes and tend to compress or reduce the values which fall in the range of higher magnitudes. This tends to make the skewed distribution as normal-like as possible. Let's use log transform on our developer income feature from our coder survey dataset.

```
In [24]: fcc_survey_df['Income_log'] = np.log((1+ fcc_survey_df['Income']))
    ...: fcc_survey_df[['ID.x', 'Age', 'Income', 'Income_log']].iloc[4:9]
Out[24]:
                              ID.x   Age   Income  Income_log
4  9368291c93d5d5f5c8cdb1a575e18bec  20.0   6000.0    8.699681
5  dd0e77eab9270e4b67c19b0d6bbf621b  34.0  40000.0   10.596660
6  7599c0aa0419b59fd11ffede98a3665d  23.0  32000.0   10.373522
7  6dff182db452487f07a47596f314bddc  35.0  40000.0   10.596660
8  9dc233f8ed1c6eb2432672ab4bb39249  33.0  80000.0   11.289794
```

The dataframe obtained in this output depicts the log transformed income feature in the `Income_log` field. Let's now plot the data distribution of this transformed feature using the following code.

```
In [25]: income_log_mean = np.round(np.mean(fcc_survey_df['Income_log']), 2)
    ...:
    ...: fig, ax = plt.subplots()
    ...: fcc_survey_df['Income_log'].hist(bins=30, color='#A9C5D3')
    ...: plt.axvline(income_log_mean, color='r')
    ...: ax.set_title('Developer Income Histogram after Log Transform', fontsize=12)
    ...: ax.set_xlabel('Developer Income (log scale)', fontsize=12)
    ...: ax.set_ylabel('Frequency', fontsize=12)
    ...: ax.text(11.5, 450, r'$\mu$='+str(income_log_mean), fontsize=10)
```

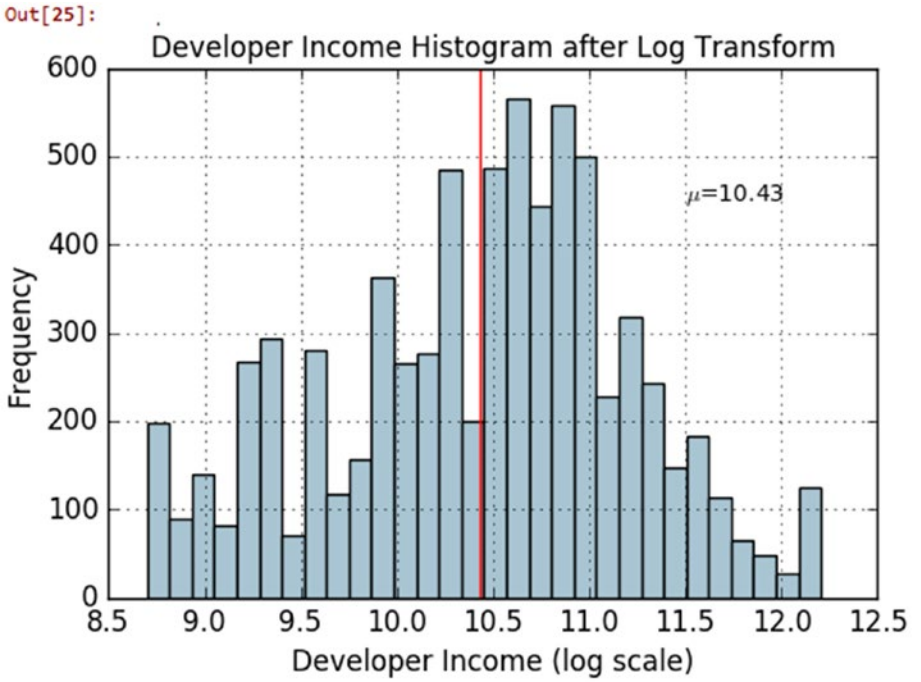

Figure 4-12. *Histogram depicting developer income distribution after log transform*

Thus we can clearly see that the original developer income distribution that was right skewed in Figure 4-10 is more Gaussian or normal-like in Figure 4-12 after applying the log transform.

Box-Cox Transform

Let's now look at the Box-Cox transform, another popular function belonging to the power transform family of functions. This function has a prerequisite that the numeric values to be transformed must be positive (similar to what log transform expects). In case they are negative, shifting using a constant value helps. Mathematically, the Box-Cox transform function can be defined as,

$$y = f(x,\lambda) = x^{\lambda} = \begin{cases} \dfrac{x^{\lambda}-1}{\lambda} & for\ \lambda > 0 \\ \log_e(x)\ for\ \lambda = 0 \end{cases}$$

Such that the resulted transformed output y is a function of input x and transformation parameter λ such that when $\lambda = 0$, the resultant transform is the natural log transform, which we discussed earlier. The optimal value of λ is usually determined using a maximum likelihood or log-likelihood estimation. Let's apply the Box-Cox transform on our developer income feature. To do this, first we get the optimal lambda value from the data distribution by removing the non-null values using the following code.

```
In [26]: # get optimal lambda value from non null income values
    ...: income = np.array(fcc_survey_df['Income'])
    ...: income_clean = income[~np.isnan(income)]
    ...: l, opt_lambda = spstats.boxcox(income_clean)
    ...: print('Optimal lambda value:', opt_lambda)
Optimal lambda value: 0.117991239456
```

Now that we have obtained the optimal λ value, let's use the Box-Cox transform for two values of λ such that $\lambda = 0$ & $\lambda = \lambda_{optimal}$ and transform the raw numeric values pertaining to developer incomes.

```
In [27]: fcc_survey_df['Income_boxcox_lambda_0'] = spstats.boxcox((1+fcc_survey_df['Income']),
    ...:                                                          lmbda=0)
    ...: fcc_survey_df['Income_boxcox_lambda_opt'] = spstats.boxcox(fcc_survey_df['Income'],
    ...:                                                           lmbda=opt_lambda)
    ...: fcc_survey_df[['ID.x', 'Age', 'Income', 'Income_log',
    ...:                'Income_boxcox_lambda_0', 'Income_boxcox_lambda_opt']].iloc[4:9]
```

Out[27]:

	ID.x	Age	Income	Income_log	Income_boxcox_lambda_0	Income_boxcox_lambda_opt
4	9368291c93d5d5f5c8cdb1a575e18bec	20.0	6000.0	8.699681	8.699681	15.180668
5	dd0e77eab9270e4b67c19b0d6bbf621b	34.0	40000.0	10.596660	10.596660	21.115342
6	7599c0aa0419b59fd11ffede98a3665d	23.0	32000.0	10.373522	10.373522	20.346420
7	6dff182db452487f07a47596f314bddc	35.0	40000.0	10.596660	10.596660	21.115342
8	9dc233f8ed1c6eb2432672ab4bb39249	33.0	80000.0	11.289794	11.289794	23.637131

Figure 4-13. *Dataframe depicting developer income distribution after box-cox transform*

The dataframe obtained in the output shown in Figure 4-13 depicts the income feature after applying the Box-Cox transform for $\lambda = 0$ and $\lambda = \lambda_{optimal}$ in the `Income_boxcox_lambda_0` and `Income_boxcox_lambda_opt` fields respectively. Also as expected, the `Income_log` field has the same values as the Box-Cox transform with $\lambda = 0$. Let's now plot the data distribution for the Box-Cox transformed developer values with optimal lambda. See Figure 4-14.

```
In [30]: income_boxcox_mean = np.round(np.mean(fcc_survey_df['Income_boxcox_lambda_opt']), 2)
    ...:
    ...: fig, ax = plt.subplots()
    ...: fcc_survey_df['Income_boxcox_lambda_opt'].hist(bins=30, color='#A9C5D3')
    ...: plt.axvline(income_boxcox_mean, color='r')
    ...: ax.set_title('Developer Income Histogram after Box-Cox Transform', fontsize=12)
    ...: ax.set_xlabel('Developer Income (Box-Cox transform)', fontsize=12)
    ...: ax.set_ylabel('Frequency', fontsize=12)
    ...: ax.text(24, 450, r'$\mu$='+str(income_boxcox_mean), fontsize=10)
```

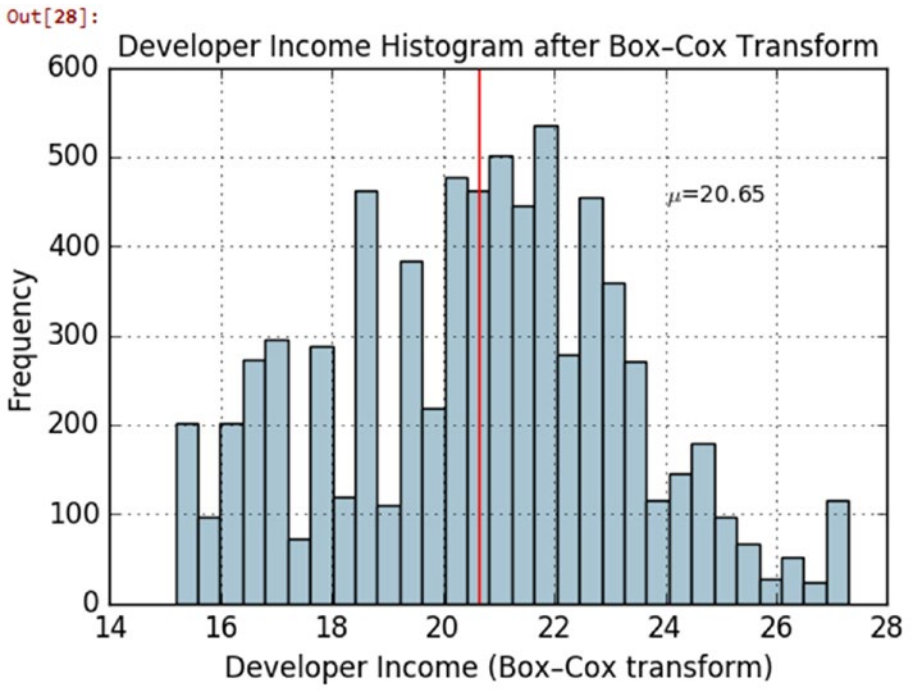

Figure 4-14. *Histogram depicting developer income distribution after box-cox transform* ($\lambda = \lambda_{optimal}$)

The distribution of the transformed numeric values for developer income after the Box-Cox distribution also look similar to the one we had obtained after the Log transform such that it is more normal-like and the extreme right skew that was present in the raw data has been minimized here.

Feature Engineering on Categorical Data

So far, we have been working on continuous numeric data and you have also seen various techniques for engineering features from the same. We will now look at another structured data type, which is categorical data. Any attribute or feature that is categorical in nature represents discrete values that belong to a specific finite set of categories or classes. Category or class labels can be text or numeric in nature. Usually there are two types of categorical variables—nominal and ordinal.

Nominal categorical features are such that there is no concept of ordering among the values, i.e., it does not make sense to sort or order them. Movie or video game genres, weather seasons, and country names are some examples of nominal attributes. Ordinal categorical variables can be ordered and sorted on the basis of their values and hence these values have specific significance such that their order makes sense. Examples of ordinal attributes include clothing size, education level, and so on.

In this section, we look at various strategies and techniques for transforming and encoding categorical features and attributes. The code used for this section is available in the code files for this chapter. You can load `feature_engineering_categorical.py` directly and start running the examples or use the jupyter notebook, `Feature Engineering on Categorical Data.ipynb`, for a more interactive experience. Before we begin, let's load the following dependencies.

```
In [1]: import pandas as pd
   ...: import numpy as np
```

Once you have these dependencies loaded, let's get started and engineer some features from categorical data.

Transforming Nominal Features

Nominal features or attributes are categorical variables that usually have a finite set of distinct discrete values. Often these values are in string or text format and Machine Learning algorithms cannot understand them directly. Hence usually you might need to transform these features into a more representative numeric format. Let's look at a new dataset pertaining to video game sales. This dataset is also available on Kaggle (https://www.kaggle.com/gregorut/videogamesales). We have downloaded a copy of this for your convenience. The following code helps us load this dataset and view some of the attributes of our interest.

```
In [2]: vg_df = pd.read_csv('datasets/vgsales.csv', encoding='utf-8')
   ...: vg_df[['Name', 'Platform', 'Year', 'Genre', 'Publisher']].iloc[1:7]
Out[2]:
                      Name Platform    Year         Genre Publisher
1         Super Mario Bros.      NES  1985.0      Platform  Nintendo
2            Mario Kart Wii      Wii  2008.0        Racing  Nintendo
3         Wii Sports Resort      Wii  2009.0        Sports  Nintendo
4  Pokemon Red/Pokemon Blue       GB  1996.0  Role-Playing  Nintendo
5                    Tetris       GB  1989.0        Puzzle  Nintendo
6     New Super Mario Bros.       DS  2006.0      Platform  Nintendo
```

The dataset depicted in this dataframe shows us various attributes pertaining to video games. Features like `Platform`, `Genre`, and `Publisher` are nominal categorical variables. Let's now try to transform the video game `Genre` feature into a numeric representation. Do note here that this doesn't indicate that the transformed feature will be a numeric feature. It will still be a discrete valued categorical feature with numbers instead of text for each genre. The following code depicts the total distinct genre labels for video games.

```
In [3]: genres = np.unique(vg_df['Genre'])
   ...: genres
Out[3]:
array(['Action', 'Adventure', 'Fighting', 'Misc', 'Platform', 'Puzzle',
       'Racing', 'Role-Playing', 'Shooter', 'Simulation', 'Sports',
       'Strategy'], dtype=object)
```

This output tells us we have 12 distinct video game genres in our dataset. Let's transform this feature now using a mapping scheme in the following code.

```
In [4]: from sklearn.preprocessing import LabelEncoder
   ...:
   ...: gle = LabelEncoder()
   ...: genre_labels = gle.fit_transform(vg_df['Genre'])
   ...: genre_mappings = {index: label for index, label in enumerate(gle.classes_)}
   ...: genre_mappings
Out[4]:
{0: 'Action', 1: 'Adventure', 2: 'Fighting', 3: 'Misc',
 4: 'Platform', 5: 'Puzzle', 6: 'Racing', 7: 'Role-Playing',
 8: 'Shooter', 9: 'Simulation', 10: 'Sports', 11: 'Strategy'}
```

From the output, we can see that a mapping scheme has been generated where each genre value is mapped to a number with the help of the `LabelEncoder` object `gle`. The transformed labels are stored in the `genre_labels` value. Let's write it back to the original dataframe and view the results.

```
In [5]: vg_df['GenreLabel'] = genre_labels
   ...: vg_df[['Name', 'Platform', 'Year', 'Genre', 'GenreLabel']].iloc[1:7]
Out[5]:
                      Name Platform    Year         Genre  GenreLabel
1         Super Mario Bros.      NES  1985.0      Platform           4
2            Mario Kart Wii      Wii  2008.0        Racing           6
3         Wii Sports Resort      Wii  2009.0        Sports          10
4  Pokemon Red/Pokemon Blue       GB  1996.0  Role-Playing           7
5                    Tetris       GB  1989.0        Puzzle           5
6     New Super Mario Bros.       DS  2006.0      Platform           4
```

The `GenreLabel` field depicts the mapped numeric labels for each of the `Genre` labels and we can clearly see that this adheres to the mappings that we generated earlier.

Transforming Ordinal Features

Ordinal features are similar to nominal features except that order matters and is an inherent property with which we can interpret the values of these features. Like nominal features, even ordinal features might be present in text form and you need to map and transform them into their numeric representation. Let's now load our Pokémon dataset that we used earlier and look at the various values of the `Generation` attribute for each Pokémon.

```
In [6]: poke_df = pd.read_csv('datasets/Pokemon.csv', encoding='utf-8')
   ...: poke_df = poke_df.sample(random_state=1, frac=1).reset_index(drop=True)
   ...:
   ...: np.unique(poke_df['Generation'])
Out[6]: array(['Gen 1', 'Gen 2', 'Gen 3', 'Gen 4', 'Gen 5', 'Gen 6'], dtype=object)
```

We resample the dataset in this code just so we can get a good slice of data later on that represents all the distinct values which we are looking for. From this output we can see that there are a total of six generations of Pokémon. This attribute is definitely ordinal because Pokémon belonging to Generation 1 were introduced earlier in the video games and the television shows than Generation 2 and so on. Hence they have a sense of order among them. Unfortunately, since there is a specific logic or set of rules involved in case of each ordinal variable, there is no generic module or function to map and transform these features into numeric representations. Hence we need to hand-craft this using our own logic, which is depicted in the following code snippet.

```
In [7]: gen_ord_map = {'Gen 1': 1, 'Gen 2': 2, 'Gen 3': 3,
   ...:                'Gen 4': 4, 'Gen 5': 5, 'Gen 6': 6}
   ...:
   ...: poke_df['GenerationLabel'] = poke_df['Generation'].map(gen_ord_map)
   ...: poke_df[['Name', 'Generation', 'GenerationLabel']].iloc[4:10]
Out[7]:
                 Name Generation  GenerationLabel
4           Octillery      Gen 2                2
5          Helioptile      Gen 6                6
6              Dialga      Gen 4                4
```

```
7  DeoxysDefense Forme      Gen 3               3
8              Rapidash      Gen 1               1
9                 Swanna     Gen 5               5
```

Thus, you can see that it is really easy to build your own transformation mapping scheme with the help of Python dictionaries and use the `map(...)` function from `pandas` to transform the ordinal feature.

Encoding Categorical Features

We have mentioned several times in the past that Machine Learning algorithms usually work well with numerical values. You might now be wondering we already transformed and mapped the categorical variables into numeric representations in the previous sections so why would we need more levels of encoding again? The answer to this is pretty simple. If we directly fed these transformed numeric representations of categorical features into any algorithm, the model will essentially try to interpret these as raw numeric features and hence the notion of magnitude will be wrongly introduced in the system.

A simple example would be from our previous output dataframe, a model fit on `GenerationLabel` would think that value `6 > 5 > 4` and so on. While order is important in the case of Pokémon generations (ordinal variable), there is no notion of magnitude here. `Generation 6` is not larger than `Generation 5` and `Generation 1` is not smaller than `Generation 6`. Hence models built using these features directly would be sub-optimal and incorrect models. There are several schemes and strategies where dummy features are created for each unique value or label out of all the distinct categories in any feature. In the subsequent sections, we will discuss some of these schemes including one hot encoding, dummy coding, effect coding, and feature hashing schemes.

One Hot Encoding Scheme

Considering we have numeric representation of any categorical feature with m labels, the one hot encoding scheme, encodes or transforms the feature into m binary features, which can only contain a value of 1 or 0. Each observation in the categorical feature is thus converted into a vector of size m with only one of the values as 1 (indicating it as active). Let's take our Pokémon dataset and perform some one hot encoding transformations on some of its categorical features.

```
In [8]: poke_df[['Name', 'Generation', 'Legendary']].iloc[4:10]
Out[8]:
                  Name Generation  Legendary
4            Octillery      Gen 2      False
5           Helioptile      Gen 6      False
6               Dialga      Gen 4       True
7  DeoxysDefense Forme      Gen 3       True
8              Rapidash      Gen 1      False
9                 Swanna     Gen 5      False
```

Considering the dataframe depicted in the output, we have two categorical features, `Generation` and `Legendary`, depicting the Pokémon generations and their legendary status. First, we need to transform these text labels into numeric representations. The following code helps us achieve this.

```
In [9]: from sklearn.preprocessing import OneHotEncoder, LabelEncoder
   ...:
   ...: # transform and map pokemon generations
   ...: gen_le = LabelEncoder()
```

```
   ...: gen_labels = gen_le.fit_transform(poke_df['Generation'])
   ...: poke_df['Gen_Label'] = gen_labels
   ...:
   ...: # transform and map pokemon legendary status
   ...: leg_le = LabelEncoder()
   ...: leg_labels = leg_le.fit_transform(poke_df['Legendary'])
   ...: poke_df['Lgnd_Label'] = leg_labels
   ...:
   ...: poke_df_sub = poke_df[['Name', 'Generation', 'Gen_Label', 'Legendary', 'Lgnd_Label']]
   ...: poke_df_sub.iloc[4:10]
Out[9]:
                 Name Generation  Gen_Label  Legendary  Lgnd_Label
4           Octillery      Gen 2          1      False           0
5          Helioptile      Gen 6          5      False           0
6              Dialga      Gen 4          3       True           1
7  DeoxysDefense Forme      Gen 3          2       True           1
8            Rapidash      Gen 1          0      False           0
9              Swanna      Gen 5          4      False           0
```

The features `Gen_Label` and `Lgnd_Label` now depict the numeric representations of our categorical features. Let's now apply the one hot encoding scheme on these features using the following code.

```
In [10]: # encode generation labels using one-hot encoding scheme
    ...: gen_ohe = OneHotEncoder()
    ...: gen_feature_arr = gen_ohe.fit_transform(poke_df[['Gen_Label']]).toarray()
    ...: gen_feature_labels = list(gen_le.classes_)
    ...: gen_features = pd.DataFrame(gen_feature_arr, columns=gen_feature_labels)
    ...:
    ...: # encode legendary status labels using one-hot encoding scheme
    ...: leg_ohe = OneHotEncoder()
    ...: leg_feature_arr = leg_ohe.fit_transform(poke_df[['Lgnd_Label']]).toarray()
    ...: leg_feature_labels = ['Legendary_'+str(cls_label) for cls_label in leg_le.classes_]
    ...: leg_features = pd.DataFrame(leg_feature_arr, columns=leg_feature_labels)
```

Now, you should remember that you can always encode both the features together using the `fit_transform(...)` function by passing it a two-dimensional array of the two features. But we are depicting this encoding for each feature separately, to make things easier to understand. Besides this, we can also create separate dataframes and label them accordingly. Let's now concatenate these feature frames and see the final result.

```
In [11]: poke_df_ohe = pd.concat([poke_df_sub, gen_features, leg_features], axis=1)
    ...: columns = sum([['Name', 'Generation', 'Gen_Label'],gen_feature_labels,
    ...:                ['Legendary', 'Lgnd_Label'],leg_feature_labels], [])
    ...: poke_df_ohe[columns].iloc[4:10]
```

Out[11]:

	Name	Generation	Gen_Label	Gen 1	Gen 2	Gen 3	Gen 4	Gen 5	Gen 6	Legendary	Lgnd_Label	Legendary_False	Legendary_True
4	Octillery	Gen 2	1	0.0	1.0	0.0	0.0	0.0	0.0	False	0	1.0	0.0
5	Helioptile	Gen 6	5	0.0	0.0	0.0	0.0	0.0	1.0	False	0	1.0	0.0
6	Dialga	Gen 4	3	0.0	0.0	0.0	1.0	0.0	0.0	True	1	0.0	1.0
7	DeoxysDefense Forme	Gen 3	2	0.0	0.0	1.0	0.0	0.0	0.0	True	1	0.0	1.0
8	Rapidash	Gen 1	0	1.0	0.0	0.0	0.0	0.0	0.0	False	0	1.0	0.0
9	Swanna	Gen 5	4	0.0	0.0	0.0	0.0	1.0	0.0	False	0	1.0	0.0

Figure 4-15. *Feature set depicting one hot encoded features for Pokémon generation and legendary status*

From the result feature set depicted in Figure 4-15, we can clearly see the new one hot encoded features for `Gen_Label` and `Lgnd_Label`. Each of these one hot encoded features is binary in nature and if they contain the value `1`, it means that feature is active for the corresponding observation. For example, row 6 indicates the Pokémon Dialga is a `Gen 4` Pokémon having `Gen_Label 3` (mapping starts from `0`) and the corresponding one hot encoded feature `Gen 4` has the value `1` and the remaining one hot encoded features are `0`. Similarly, its `Legendary` status is `True`, corresponding `Lgnd_Label` is `1` and the one hot encoded feature `Legendary_True` is also `1`, indicating it is active.

Suppose we used this data in training and building a model but now we have some new Pokémon data for which we need to engineer the same features before we want to run it by our trained model. We can use the `transform(...)` function for our `LabelEncoder` and `OneHotEncoder` objects, which we have previously constructed to engineer the features from the training data. The following code shows us two dummy data points pertaining to new Pokémon.

```
In [12]: new_poke_df = pd.DataFrame([['PikaZoom', 'Gen 3', True],
    ...:                             ['CharMyToast', 'Gen 4', False]],
    ...:                             columns=['Name', 'Generation', 'Legendary'])
    ...: new_poke_df
Out[12]:
          Name Generation  Legendary
0     PikaZoom      Gen 3       True
1  CharMyToast      Gen 4      False
```

We will follow the same process as before of first converting the text categories into numeric representations using our previously built `LabelEncoder` objects, as depicted in the following code.

```
In [13]: new_gen_labels = gen_le.transform(new_poke_df['Generation'])
    ...: new_poke_df['Gen_Label'] = new_gen_labels
    ...:
    ...: new_leg_labels = leg_le.transform(new_poke_df['Legendary'])
    ...: new_poke_df['Lgnd_Label'] = new_leg_labels
    ...:
    ...: new_poke_df[['Name', 'Generation', 'Gen_Label', 'Legendary', 'Lgnd_Label']]
Out[13]:
          Name Generation  Gen_Label  Legendary  Lgnd_Label
0     PikaZoom      Gen 3          2       True           1
1  CharMyToast      Gen 4          3      False           0
```

We can now use our previously built `LabelEncoder` objects and perform one hot encoding on these new data observations using the following code. See Figure 4-16.

```
In [14]: new_gen_feature_arr = gen_ohe.transform(new_poke_df[['Gen_Label']]).toarray()
    ...: new_gen_features = pd.DataFrame(new_gen_feature_arr, columns=gen_feature_labels)
    ...:
    ...: new_leg_feature_arr = leg_ohe.transform(new_poke_df[['Lgnd_Label']]).toarray()
    ...: new_leg_features = pd.DataFrame(new_leg_feature_arr, columns=leg_feature_labels)
    ...:
    ...: new_poke_ohe = pd.concat([new_poke_df, new_gen_features, new_leg_features], axis=1)
    ...: columns = sum([['Name', 'Generation', 'Gen_Label'], gen_feature_labels,
    ...:                ['Legendary', 'Lgnd_Label'], leg_feature_labels], [])
    ...: new_poke_ohe[columns]
```

Out[14]:

	Name	Generation	Gen_Label	Gen 1	Gen 2	Gen 3	Gen 4	Gen 5	Gen 6	Legendary	Lgnd_Label	Legendary_False	Legendary_True
0	PikaZoom	Gen 3	2	0.0	0.0	1.0	0.0	0.0	0.0	True	1	0.0	1.0
1	CharMyToast	Gen 4	3	0.0	0.0	0.0	1.0	0.0	0.0	False	0	1.0	0.0

Figure 4-16. *Feature set depicting one hot encoded features for new pokemon data points*

Thus, you can see how we used the `fit_transform(...)` functions to engineer features on our dataset and then we were able to use the encoder objects to engineer features on new data using the `transform(...)` function based on the data what it observed previously, specifically the distinct categories and their corresponding labels and one hot encodings. You should always follow this workflow in the future for any type of feature engineering when you deal with training and test datasets when you build models. Pandas also provides a wonderful function called `to_dummies(...)`, which helps us easily perform one hot encoding. The following code depicts how to achieve this.

```
In [15]: gen_onehot_features = pd.get_dummies(poke_df['Generation'])
    ...: pd.concat([poke_df[['Name', 'Generation']], gen_onehot_features], axis=1).
iloc[4:10]
Out[15]:
                  Name Generation  Gen 1  Gen 2  Gen 3  Gen 4  Gen 5  Gen 6
4            Octillery      Gen 2      0      1      0      0      0      0
5           Helioptile      Gen 6      0      0      0      0      0      1
6              Dialga      Gen 4      0      0      0      1      0      0
7  DeoxysDefense Forme      Gen 3      0      0      1      0      0      0
8             Rapidash      Gen 1      1      0      0      0      0      0
9               Swanna      Gen 5      0      0      0      0      1      0
```

The output depicts the one hot encoding scheme for Pokémon generation values similar to what we depicted in our previous analyses.

Dummy Coding Scheme

The dummy coding scheme is similar to the one hot encoding scheme, except in the case of dummy coding scheme, when applied on a categorical feature with m distinct labels, we get m-1 binary features. Thus each value of the categorical variable gets converted into a vector of size m-1. The extra feature is completely disregarded and thus if the category values range from {0, 1, ..., m-1} the 0th or the m-1*th* feature is usually represented by a vector of all zeros (0).

The following code depicts the dummy coding scheme on Pokémon `Generation` by dropping the first level binary encoded feature (`Gen 1`).

```
In [16]: gen_dummy_features = pd.get_dummies(poke_df['Generation'], drop_first=True)
    ...: pd.concat([poke_df[['Name', 'Generation']], gen_dummy_features], axis=1).iloc[4:10]
Out[16]:
                  Name Generation  Gen 2  Gen 3  Gen 4  Gen 5  Gen 6
4           Octillery      Gen 2      1      0      0      0      0
5          Helioptile      Gen 6      0      0      0      0      1
6              Dialga      Gen 4      0      0      1      0      0
7  DeoxysDefense Forme      Gen 3      0      1      0      0      0
8            Rapidash      Gen 1      0      0      0      0      0
9              Swanna      Gen 5      0      0      0      1      0
```

If you want, you can also choose to drop the last level binary encoded feature (`Gen 6`) by using the following code.

```
In [17]: gen_onehot_features = pd.get_dummies(poke_df['Generation'])
    ...: gen_dummy_features = gen_onehot_features.iloc[:,:-1]
    ...: pd.concat([poke_df[['Name', 'Generation']], gen_dummy_features], axis=1).iloc[4:10]
Out[17]:
                  Name Generation  Gen 1  Gen 2  Gen 3  Gen 4  Gen 5
4           Octillery      Gen 2      0      1      0      0      0
5          Helioptile      Gen 6      0      0      0      0      0
6              Dialga      Gen 4      0      0      0      1      0
7  DeoxysDefense Forme      Gen 3      0      0      1      0      0
8            Rapidash      Gen 1      1      0      0      0      0
9              Swanna      Gen 5      0      0      0      0      1
```

Thus from these outputs you can see that based on the encoded level binary feature which we drop, that particular categorical value is represented by a vector/encoded features, which all represent `0`. For example in the previous result feature set, Pokémon Heloptile belongs to `Gen 6` and is represented by all `0`s in the encoded dummy features.

Effect Coding Scheme

The effect coding scheme is very similar to the dummy coding scheme in most aspects. However, the encoded features or feature vector, for the category values that represent all `0`s in the dummy coding scheme, is replaced by `-1`s in the effect coding scheme. The following code depicts the effect coding scheme on the Pokémon `Generation` feature.

```
In [18]: gen_onehot_features = pd.get_dummies(poke_df['Generation'])
    ...: gen_effect_features = gen_onehot_features.iloc[:,:-1]
    ...: gen_effect_features.loc[np.all(gen_effect_features == 0, axis=1)] = -1.
    ...: pd.concat([poke_df[['Name', 'Generation']], gen_effect_features], axis=1).iloc[4:10]
Out[18]:
                  Name Generation  Gen 1  Gen 2  Gen 3  Gen 4  Gen 5
4           Octillery      Gen 2    0.0    1.0    0.0    0.0    0.0
5          Helioptile      Gen 6   -1.0   -1.0   -1.0   -1.0   -1.0
6              Dialga      Gen 4    0.0    0.0    0.0    1.0    0.0
```

```
7  DeoxysDefense Forme      Gen 3    0.0    0.0    1.0    0.0    0.0
8              Rapidash     Gen 1    1.0    0.0    0.0    0.0    0.0
9                Swanna     Gen 5    0.0    0.0    0.0    0.0    1.0
```

We can clearly see from the output feature set that all 0s have been replaced by -1 in case of values which were previously all 0 in the dummy coding scheme.

Bin-Counting Scheme ?

The encoding schemes discovered so far work quite well on categorical data in general, but they start causing problems when the number of distinct categories in any feature becomes very large. Essential for any categorical feature of m distinct labels, you get m separate features. This can easily increase the size of the feature set causing problems like storage issues, model training problems with regard to time, space and memory. Besides this, we also have to deal with what is popularly known as the curse of dimensionality where basically with an enormous number of features and not enough representative samples, model performance starts getting affected. Hence we need to look toward other categorical data feature engineering schemes for features having a large number of possible categories (like IP addresses).

The bin-counting scheme is useful for dealing with categorical variables with many categories. In this scheme, instead of using the actual label values for encoding, we use probability based statistical information about the value and the actual target or response value which we aim to predict in our modeling efforts. A simple example would be based on past historical data for IP addresses and the ones which were used in DDOS attacks; we can build probability values for a DDOS attack being caused by any of the IP addresses. Using this information, we can encode an input feature which depicts that if the same IP address comes in the future, what is the probability value of a DDOS attack being caused. This scheme needs historical data as a pre-requisite and is an elaborate one. Depicting this with a complete example is out of scope of this chapter but there are several resources online that you can refer to.

Feature Hashing Scheme

The feature hashing scheme is another useful feature engineering scheme for dealing with large scale categorical features. In this scheme, a hash function is typically used with the number of encoded features pre-set (as a vector of pre-defined length) such that the hashed values of the features are used as indices in this pre-defined vector and values are updated accordingly. Since a hash function maps a large number of values into a small finite set of values, multiple different values might create the same hash which is termed as collisions. Typically, a signed hash function is used so that the sign of the value obtained from the hash is used as the sign of the value which is stored in the final feature vector at the appropriate index. This should ensure lesser collisions and lesser accumulation of error due to collisions.

Hashing schemes work on strings, numbers and other structures like vectors. You can think of hashed outputs as a finite set of h bins such that when hash function is applied on the same values, they get assigned to the same bin out of the h bins based on the hash value. We can assign the value of h, which becomes the final size of the encoded feature vector for each categorical feature we encode using the feature hashing scheme. Thus even if we have over 1000 distinct categories in a feature and we set $h = 10$, the output feature set will still have only 10 features as compared to 1000 features if we used a one hot encoding scheme.

Let's look at the following code snippet, which shows us the number of distinct genres we have in our video game dataset.

```
In [19]: unique_genres = np.unique(vg_df[['Genre']])
    ...: print("Total game genres:", len(unique_genres))
    ...: print(unique_genres)
```

```
Total game genres: 12
['Action' 'Adventure' 'Fighting' 'Misc' 'Platform' 'Puzzle' 'Racing'
 'Role-Playing' 'Shooter' 'Simulation' 'Sports' 'Strategy']
```

We can clearly see from the output that there are 12 distinct genres and if we used a one hot encoding scheme on the `Genre` feature, we would end up having 12 binary features. Instead, we will now use a feature hashing scheme by leveraging `scikit-learn's FeatureHasher` class, which uses a signed 32-bit version of the `Murmurhash3` hash function. The following code shows us how to use the feature hashing scheme where we will pre-set the feature vector size to be 6 (6 features instead of 12).

```
In [21]: from sklearn.feature_extraction import FeatureHasher
    ...:
    ...: fh = FeatureHasher(n_features=6, input_type='string')
    ...: hashed_features = fh.fit_transform(vg_df['Genre'])
    ...: hashed_features = hashed_features.toarray()
    ...: pd.concat([vg_df[['Name', 'Genre']], pd.DataFrame(hashed_features)], axis=1).
         iloc[1:7]
Out[21]:
                      Name         Genre    0    1    2    3    4    5
1         Super Mario Bros.      Platform  0.0  2.0  2.0 -1.0  1.0  0.0
2            Mario Kart Wii        Racing -1.0  0.0  0.0  0.0  0.0 -1.0
3         Wii Sports Resort        Sports -2.0  2.0  0.0 -2.0  0.0  0.0
4  Pokemon Red/Pokemon Blue  Role-Playing -1.0  1.0  2.0  0.0  1.0 -1.0
5                    Tetris        Puzzle  0.0  1.0  1.0 -2.0  1.0 -1.0
6     New Super Mario Bros.      Platform  0.0  2.0  2.0 -1.0  1.0  0.0
```

Thus we can clearly see from the result feature set that the `Genre` categorical feature has been encoded using the hashing scheme into 6 features instead of 12. We can also see that rows 1 and 6 denote the same genre of games, `Platform` which have been rightly encoded into the same feature vector as expected.

Feature Engineering on Text Data

Dealing with structured data attributes like numeric or categorical variables are usually not as challenging as unstructured attributes like text and images. In case of unstructured data like text documents, the first challenge is dealing with the unpredictable nature of the syntax, format, and content of the documents, which make it a challenge to extract useful information for building models. The second challenge is transforming these textual representations into numeric representations that can be understood by Machine Learning algorithms. There exist various feature engineering techniques employed by data scientists daily to extract numeric feature vectors from unstructured text. In this section, we discuss several of these techniques. Before we get started, you should remember that there are two aspects to execute feature engineering on text data.

- Pre-processing and normalizing text
- Feature extraction and engineering

Without text pre-processing and normalization, the feature engineering techniques will not work at their core efficiency hence it is of paramount importance to pre-process textual documents. You can load `feature_engineering_text.py` directly and start running the examples or use the jupyter notebook, `Feature Engineering on Text Data.ipynb`, for a more interactive experience. Let's load the following necessary dependencies before we start.

```
In [1]: import pandas as pd
   ...: import numpy as np
   ...: import re      regular expressions
   ...: import nltk
```

Let's now load some sample text documents, do some basic pre-processing, and learn about various feature engineering strategies to deal with text data. The following code creates our sample text corpus (a collection of text documents), which we will use in this section.

```
In [2]: corpus = ['The sky is blue and beautiful.',
   ...:           'Love this blue and beautiful sky!',
   ...:           'The quick brown fox jumps over the lazy dog.',
   ...:           'The brown fox is quick and the blue dog is lazy!',
   ...:           'The sky is very blue and the sky is very beautiful today',
   ...:           'The dog is lazy but the brown fox is quick!'
   ...: ]
   ...: labels = ['weather', 'weather', 'animals', 'animals', 'weather', 'animals']
   ...: corpus = np.array(corpus)
   ...: corpus_df = pd.DataFrame({'Document': corpus,
   ...:                           'Category': labels})
   ...: corpus_df = corpus_df[['Document', 'Category']]
   ...: corpus_df
Out[2]:
                                        Document Category
0                    The sky is blue and beautiful.  weather
1                  Love this blue and beautiful sky!  weather
2       The quick brown fox jumps over the lazy dog.  animals
3   The brown fox is quick and the blue dog is lazy!  animals
4  The sky is very blue and the sky is very beaut...  weather
5    The dog is lazy but the brown fox is quick!      animals
```

We can see that we have a total of six documents, where three of them are relevant to weather and the other three talk about animals as depicted by the `Category` class label.

Text Pre-Processing

Before feature engineering, we need to pre-process, clean, and normalize the text like we mentioned before. There are multiple pre-processing techniques, some of which are quite elaborate. We will not be going into a lot of details in this section but we will be covering a lot of them in further detail in a future chapter when we work on text classification and sentiment analysis. Following are some of the popular pre-processing techniques.

- Text tokenization and lower casing
- Removing special characters
- Contraction expansion
- Removing stopwords
- Correcting spellings
- Stemming
- Lemmatization

For more details on these topics, you can jump ahead to Chapter 7 of this book or refer to the section "Text Normalization," Chapter 3, page 115 of *Text Analytics with Python* (Apress; Dipanjan Sarkar, 2016). which covers each of these techniques in detail. We will be normalizing our text here by lowercasing, removing special characters, tokenizing, and removing stopwords. The following code helps us achieve this.

```
In [3]: wpt = nltk.WordPunctTokenizer()
   ...: stop_words = nltk.corpus.stopwords.words('english')
   ...:
   ...: def normalize_document(doc):
   ...:     # lower case and remove special characters\whitespaces
   ...:     doc = re.sub(r'[^a-zA-Z0-9\s]', '', doc, re.I)
   ...:     doc = doc.lower()
   ...:     doc = doc.strip()
   ...:     # tokenize document
   ...:     tokens = wpt.tokenize(doc)
   ...:     # filter stopwords out of document
   ...:     filtered_tokens = [token for token in tokens if token not in stop_words]
   ...:     # re-create document from filtered tokens
   ...:     doc = ' '.join(filtered_tokens)
   ...:     return doc
   ...:
   ...: normalize_corpus = np.vectorize(normalize_document)
```

The `np.vectorize(...)` function helps us run the same function over all elements of a `numpy` array instead of writing a loop. We will now use this function to pre-process our text corpus.

```
In [4]: norm_corpus = normalize_corpus(corpus)
   ...: norm_corpus
Out[4]:
array(['sky blue beautiful', 'love blue beautiful sky',
       'quick brown fox jumps lazy dog', 'brown fox quick blue dog lazy',
       'sky blue sky beautiful today', 'dog lazy brown fox quick'],
      dtype='<U32')
```

You can compare each text document with its original form in our initial dataframe. You will see that each document is in the lowercase, special symbols have been removed and stopwords (words which carry little meaning like articles, pronouns, etc.) have been removed. We can now engineer features from this pre-processed corpus.

Bag of Words Model

This is perhaps one of the simplest yet effective schemes of vectorizing features from unstructured text. The core principle of this model is to convert text documents into numeric vectors. The dimension or size of each vector is *N* where *N* indicates all possible distinct words across the corpus of documents. Each document once transformed is a numeric vector of size *N* where the values or weights in the vector indicate the frequency of each word in that specific document. The following code helps us vectorize the text corpus into numeric feature vectors.

```
In [5]: from sklearn.feature_extraction.text import CountVectorizer
   ...:
   ...: cv = CountVectorizer(min_df=0., max_df=1.)
```

```
   ...: cv_matrix = cv.fit_transform(norm_corpus)
   ...: cv_matrix = cv_matrix.toarray()
   ...: cv_matrix
Out[5]:
array([[1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0],
       [0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0],
       [0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0],
       [1, 1, 0, 0, 0, 0, 0, 0, 0, 2, 1],
       [0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0]], dtype=int64)
```

The output represents a numeric term frequency based feature vector for each document like we mentioned before. To understand it better, we can represent it using the feature names and view it as a dataframe.

```
In [6]: vocab = cv.get_feature_names()
   ...: pd.DataFrame(cv_matrix, columns=vocab)
Out[6]:
   beautiful  blue  brown  dog  fox  jumps  lazy  love  quick  sky  today
0          1     1      0    0    0      0     0     0      0    1      0
1          1     1      0    0    0      0     0     1      0    1      0
2          0     0      1    1    1      1     1     0      1    0      0
3          0     1      1    1    1      0     1     0      1    0      0
4          1     1      0    0    0      0     0     0      0    2      1
5          0     0      1    1    1      0     1     0      1    0      0
```

We can clearly see now that each row of the dataframe depicts the term frequency vector for each text document. Hence the name bag of words because this model represents unstructured text into a bag of words without taking into account word positions, syntax, or semantics.

Bag of N-Grams Model

We have used single word terms as features in the above mentioned bag of words model. But what if we also wanted to take into account phrases or collection of words which occur in a sequence? N-grams help us achieve that. An n-gram is basically a collection of word tokens from a text document such that these tokens are contiguous and occur in a sequence. Bi-grams indicate n-grams of order 2 (two words), Tri-grams indicate n-grams of order 3 (three words), and so on. We can easily extend the bag of words model to use a bag of n-grams model to give us n-gram based feature vectors. The following code computes bi-gram based features on our corpus.

```
In [7]: bv = CountVectorizer(ngram_range=(2,2))
   ...: bv_matrix = bv.fit_transform(norm_corpus)
   ...: bv_matrix = bv_matrix.toarray()
   ...: vocab = bv.get_feature_names()
   ...: pd.DataFrame(bv_matrix, columns=vocab)
```

Out[7]:

	beautiful sky	beautiful today	blue beautiful	blue dog	blue sky	brown fox	dog lazy	fox jumps	fox quick	jumps lazy	lazy brown	lazy dog	love blue	quick blue	quick brown	sky beautiful	sky blue
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
2	0	0	0	0	0	1	0	1	0	1	0	1	0	0	1	0	0
3	0	0	0	1	0	1	1	0	1	0	0	0	0	1	0	0	0
4	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1
5	0	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0

Figure 4-17. *Bi-gram feature vectors for our corpus based on bag of n-grams model*

Figure 4-17 clearly shows our bi-gram feature vectors where each feature is a bi-gram of two contiguous words and the values depict the frequency of that bi-gram in each document. You can use the `ngram_range` parameter to extend the n-gram range to get n-grams of higher orders. Typically n-grams until order three are sufficient for most tasks in Machine Learning and natural language processing.

TF-IDF Model

term frequency - inverse document frequency

There are some potential problems which might arise with the Bag of Words model when it is used on large corpora. Since the feature vectors are based on absolute term frequencies, there might be some terms which occur frequently across all documents and these will tend to overshadow other terms in the feature set. The TF-IDF model tries to combat this issue by using a scaling or normalizing factor in its computation. TF-IDF stands for Term Frequency-Inverse Document Frequency, which uses a combination of two metrics in its computation, namely: term frequency (tf) and inverse document frequency (idf). This technique was developed for ranking results for queries in search engines and now it is an indispensable model in the world of information retrieval and text analytics.

Mathematically, we can define TF-IDF as *tfidf* = *tf* x *idf*, which can be expanded further to be represented as follows.

$$tfidf(w,D) = tf(w,D) \times idf(w,D) = tf(w,D) \times log\left(\frac{C}{df(w)}\right)$$

Here, *tfidf* (*w, D*) is the TF-IDF score for word *w* in document *D*. The term *tf* (*w, D*) represents the term frequency of the word *w* in document *D*, which can be obtained from the Bag of Words model. The term *idf* (*w, D*) is the inverse document frequency for the term *w*, which can be computed as the log transform of the total number of documents in the corpus *C* divided by the document frequency of the word *w*, which is basically the frequency of documents in the corpus where the word *w* occurs. The following code depicts TF-IDF based feature engineering on our corpus.

```
In [8]: from sklearn.feature_extraction.text import TfidfVectorizer
   ...:
   ...: tv = TfidfVectorizer(min_df=0., max_df=1., use_idf=True)
   ...: tv_matrix = tv.fit_transform(norm_corpus)
   ...: tv_matrix = tv_matrix.toarray()
   ...:
   ...: vocab = tv.get_feature_names()
   ...: pd.DataFrame(np.round(tv_matrix, 2), columns=vocab)
```

```
Out[8]:
   beautiful  blue  brown   dog   fox  jumps  lazy  love  quick   sky  today
0       0.60  0.52   0.00  0.00  0.00   0.00  0.00  0.00   0.00  0.60   0.00
1       0.46  0.39   0.00  0.00  0.00   0.00  0.00  0.66   0.00  0.46   0.00
2       0.00  0.00   0.38  0.38  0.38   0.54  0.38  0.00   0.38  0.00   0.00
3       0.00  0.36   0.42  0.42  0.42   0.00  0.42  0.00   0.42  0.00   0.00
4       0.36  0.31   0.00  0.00  0.00   0.00  0.00  0.00   0.00  0.72   0.52
5       0.00  0.00   0.45  0.45  0.45   0.00  0.45  0.00   0.45  0.00   0.00
```

Thus, the preceding output depicts the TF-IDF based feature vectors for each of our text documents. Notice how this is a scaled and normalized version as compared to the raw Bag of Words model. Interested readers who might want to dive into further details of how the internals of this model work can refer to page 181 of *Text Analytics with Python* (Apress; Dipanjan Sarkar, 2016).

Document Similarity

You can even build on top of the tf-idf based features we engineered in the previous section and use them to generate new features which can be useful in multiple applications. An example of this is computing document similarity. This is very useful in domains like search engines, document clustering, and information retrieval. Document similarity is the process of using a distance or similarity based metric that can be used to identify how similar a text document is with another document based on features extracted from the documents like bag of words or tf-idf. Pairwise document similarity in a corpus involves computing document similarity for each pair of documents in a corpus. Thus if you have *C* documents in a corpus, you would end up with a *C* x *C* matrix such that each row and column represents the similarity score for a pair of documents, which represent the indices at the row and column, respectively.

There are several similarity and distance metrics that are used to compute document similarity. These include cosine distance/similarity, BM25 distance, Hellinger-Bhattacharya distance, jaccard distance, and so on. In our analysis, we will be using perhaps the most popular and widely used similarity metric, cosine similarity. Cosine similarity basically gives us a metric representing the cosine of the angle between the feature vector representations of two text documents. Figure 4-18 shows some typical feature vector alignments for text documents.

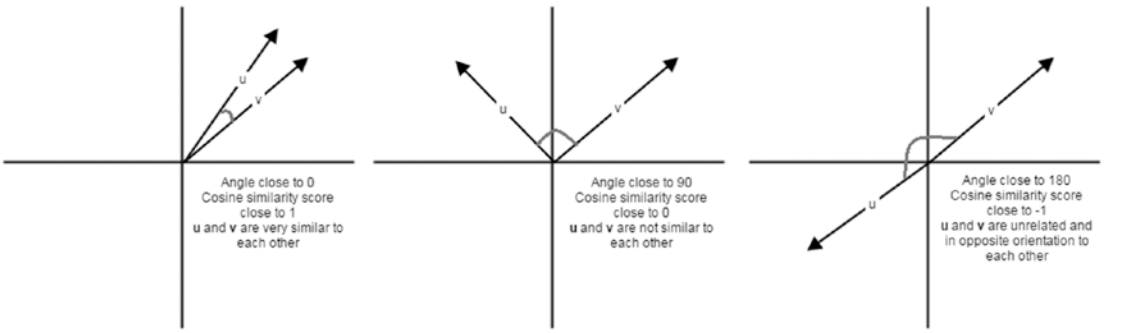

Figure 4-18. *Cosine similarity depictions for text document feature vectors (Source: Text Analytics with Python, Apress)*

From Figure 4-18, we can clearly see that feature vectors having a similar orientation will be very close to one another and the angle between them will be closer to 0° and thus cosine similarity would be cos 0° = 1 When cosine similarity is close to cos 90° = 0, the angle between the documents is closer to 90° indicating they are far apart and hence not very similar. Similarity scores close to -1 indicate the documents have completely opposite orientation as the angle between them would be closer to 180°. The following code helps us compute pairwise cosine similarity for all the documents in our sample corpus.

```
In [9]: from sklearn.metrics.pairwise import cosine_similarity
   ...:
   ...: similarity_matrix = cosine_similarity(tv_matrix)
   ...: similarity_df = pd.DataFrame(similarity_matrix)
   ...: similarity_df
Out[9]:
          0         1         2         3         4         5
0  1.000000  0.753128  0.000000  0.185447  0.807539  0.000000
1  0.753128  1.000000  0.000000  0.139665  0.608181  0.000000
2  0.000000  0.000000  1.000000  0.784362  0.000000  0.839987
3  0.185447  0.139665  0.784362  1.000000  0.109653  0.933779
4  0.807539  0.608181  0.000000  0.109653  1.000000  0.000000
5  0.000000  0.000000  0.839987  0.933779  0.000000  1.000000
```

From the pairwise similarity matrix obtained in the preceding output, we can clearly see that documents 0, 1, and 4 have very strong similarity among one another. Also documents 2, 3, and 5 have strong similarity among themselves. This must indicate they all have some similar features. This is a perfect example of grouping or clustering that can be solved by unsupervised learning.

Let's use K-means clustering to try to use the features to see if we can actually cluster or group these documents based on their feature representations. In K-means clustering, we have an input parameter k, which specifies the number of clusters it will output using the document features. This clustering method is a centroid based clustering method, where it tries to cluster these documents into clusters of equal variance. It tries to create these clusters by minimizing the within-cluster sum of squares measure, also known as inertia. The following snippet builds a clustering model using our similarity features to cluster our text documents.

```
In [10]: from sklearn.cluster import KMeans
    ...:
    ...: km = KMeans(n_clusters=2)
    ...: km.fit_transform(similarity_df)
    ...: cluster_labels = km.labels_
    ...: cluster_labels = pd.DataFrame(cluster_labels, columns=['ClusterLabel'])
    ...: pd.concat([corpus_df, cluster_labels], axis=1)
Out[10]:
                                        Document Category  ClusterLabel
0                    The sky is blue and beautiful.  weather             0
1                Love this blue and beautiful sky!  weather             0
2       The quick brown fox jumps over the lazy dog.  animals             1
3  The brown fox is quick and the blue dog is lazy!  animals             1
4  The sky is very blue and the sky is very beaut...  weather             0
5        The dog is lazy but the brown fox is quick!  animals             1
```

The output obtained clearly shows us that our K-means clustering model has labeled our documents into two clusters with labels 0 and 1. We can also see that these labels are correct where labels with value 0 indicate documents relevant to weather and labels with value 1 indicate documents relevant to animals. Thus you can see how useful these features are in document clustering and categorization!

Topic Models

Besides document terms, phrases and similarities, we can also use some summarization techniques to extract topic or concept based features from text documents. The idea of topic models revolves around the process of extracting key themes or concepts from a corpus of documents which are represented as topics. Each topic can be represented as a bag or collection of words/terms from the document corpus. Together, these terms signify a specific topic, theme or a concept and each topic can be easily distinguished from other topics by virtue of the semantic meaning conveyed by these terms. These concepts can range from simple facts and statements to opinions and outlook. Topic models are extremely useful in summarizing large corpus of text documents to extract and depict key concepts. They are also useful in extracting features from text data that capture latent patterns in the data.

There are various techniques for topic modeling and most of them involve some form of matrix decomposition. Some techniques like Latent Semantic Indexing (LSI) use matrix decomposition operations, more specifically Singular Valued Decomposition (refer back to important mathematical concepts in Chapter 1), to split a term-document matrix (transpose of our TF-IDF document-term feature matrix) into three matrices, U, S & V^T. You can use the left singular vectors in matrix U and multiply it by the singular vectors S to get terms and their weights (signifying importance) per topic. You can use `scikit-learn` or `gensim` to use LSI based topic modeling.

Another technique is Latent Dirichlet Allocation (LDA), which uses a generative probabilistic model where each document consists of a combination of several topics and each term or word can be assigned to a specific topic. This is similar to pLSI based model (probabilistic LSI). Each latent topic contains a Dirichlet prior over them in the case of LDA. The math behind this is pretty involving and it would not be possible to go into details in the current scope. Interested readers can refer to page 241 of *Text Analytics with* Python (Apress; Dipanjan Sarkar, 2016) for further details on LDA. For the purpose of feature engineering, you need to remember that when LDA is applied on a document-term matrix (TF-IDF feature matrix), it gets decomposed into two main components. A document-topic matrix, which would be the feature matrix we are looking for and a topic-term matrix, which helps us in looking at potential topics in the corpus. The following code builds an LDA model to extract features and topics from our sample corpus.

```
In [11]: from sklearn.decomposition import LatentDirichletAllocation
    ...:
    ...: lda = LatentDirichletAllocation(n_topics=2, max_iter=100, random_state=42)
    ...: dt_matrix = lda.fit_transform(tv_matrix)
    ...: features = pd.DataFrame(dt_matrix, columns=['T1', 'T2'])
    ...: features
Out[11]:
         T1        T2
0  0.190615  0.809385
1  0.176860  0.823140
2  0.846148  0.153852
3  0.815229  0.184771
4  0.180563  0.819437
5  0.839140  0.160860
```

Thus, the `dt_matrix` refers to the document-topic matrix giving us two features since we chose number of topics to be 2. You can also use the other matrix obtained from the decomposition, the topic-term matrix to see the topics extracted from our corpus using the LDA model using the following code.

```
In [12]: tt_matrix = lda.components_
    ...: for topic_weights in tt_matrix:
    ...:     topic = [(token, weight) for token, weight in zip(vocab, topic_weights)]
    ...:     topic = sorted(topic, key=lambda x: -x[1])
```

```
    ...:     topic = [item for item in topic if item[1] > 0.6]
    ...:     print(topic)
    ...:     print()
[('fox', 1.7265536238698524), ('quick', 1.7264910761871224), ('dog', 1.7264019823624879),
('brown', 1.7263774760262807), ('lazy', 1.7263567668213813), ('jumps', 1.0326450363521607),
('blue', 0.7770158513472083)]

[('sky', 2.263185143458752), ('beautiful', 1.9057084998062579), ('blue',
1.7954559705805626), ('love', 1.1476805311187976), ('today', 1.0064979209198706)]
```

The preceding output represents each of the two topics as a collection of terms and their importance is depicted by the corresponding weight. It is definitely interesting to see that the two topics are quite distinguishable from each other by looking at the terms. The first topic shows terms relevant to animals and the second topic shows terms relevant to weather. This is reinforced by applying our unsupervised K-means clustering algorithm on our document-topic feature matrix (`dt_matrix`) using the following code snippet.

```
In [13]: km = KMeans(n_clusters=2)
    ...: km.fit_transform(features)
    ...: cluster_labels = km.labels_
    ...: cluster_labels = pd.DataFrame(cluster_labels, columns=['ClusterLabel'])
    ...: pd.concat([corpus_df, cluster_labels], axis=1)
Out[13]:
                                        Document Category  ClusterLabel
0                    The sky is blue and beautiful.  weather             0
1                 Love this blue and beautiful sky!  weather             0
2       The quick brown fox jumps over the lazy dog.  animals             1
3   The brown fox is quick and the blue dog is lazy!  animals             1
4  The sky is very blue and the sky is very beaut...  weather             0
5        The dog is lazy but the brown fox is quick!  animals             1
```

This clearly makes sense and we can see that by just using two topic-model based features, we are still able to cluster our documents efficiently!

Word Embeddings

There are several advanced word vectorization models that have recently gained a lot of prominence. Almost all of them deal with the concept of word embeddings. Basically, word embeddings can be used for feature extraction and language modeling. This representation tries to map each word or phrase into a complete numeric vector such that semantically similar words or terms tend to occur closer to each other and these can be quantified using these embeddings. The `word2vec` model is perhaps one of the most popular neural network based probabilistic language models and can be used to learn distributed representational vectors for words. Word embeddings produced by `word2vec` involve taking in a corpus of text documents, representing words in a large high dimensional vector space such that each word has a corresponding vector in that space and similar words (even semantically) are located close to one another, analogous to what we observed in document similarity earlier.

The `word2vec` model was released by Google in 2013 and uses a neural network based implementation with architectures like continuous Bag of Words and Skip-Grams to learn the distributed vector representations of words in a corpus. We will be using the `gensim` framework to implement the same model on our corpus to extract features. Some of the important parameters in the model are explained briefly as follows.

- **size**: Represents the feature vector size for each word in the corpus when transformed.
- **window**: Sets the context window size specifying the length of the window of words to be taken into account as belonging to a single, similar context when training.
- **min_count**: Specifies the minimum word frequency value needed across the corpus to consider the word as a part of the final vocabulary during training the model.
- **sample**: Used to downsample the effects of words which occur very frequently.

The following snippet builds a word2vec embedding model on the documents of our sample corpus. Remember to tokenize each document before passing it to the model.

```
In [14]: from gensim.models import word2vec
    ...:
    ...: wpt = nltk.WordPunctTokenizer()
    ...: tokenized_corpus = [wpt.tokenize(document) for document in norm_corpus]
    ...:
    ...: # Set values for various parameters
    ...: feature_size = 10    # Word vector dimensionality
    ...: window_context = 10          # Context window size
    ...: min_word_count = 1   # Minimum word count
    ...: sample = 1e-3   # Downsample setting for frequent words
    ...:
    ...: w2v_model = word2vec.Word2Vec(tokenized_corpus, size=feature_size,
    ...:                               window=window_context, min_count = min_word_count,
    ...:                               sample=sample)
Using TensorFlow backend.
```

Each word in the corpus will essentially now be a vector itself of size 10. We can verify the same using the following code.

```
In [15]: w2v_model.wv['sky']
Out[15]:
array([ 0.02626196, -0.02171229, -0.04910386,  0.0194816 ,  0.01649994,
        0.01200452,  0.04641563,  0.01844106,  0.02693636, -0.02992732], dtype=float32)
```

A question might arise in your mind now that so far, we had feature vectors for each complete document, but now we have vectors for each word. How on earth do we represent entire documents now? We can do that using various aggregation and combinations. A simple scheme would be to use an averaged word vector representation, where we simply sum all the word vectors occurring in a document and then divide by the count of word vectors to represent an averaged word vector for the document. The following code enables us to do the same.

```
In [16]: def average_word_vectors(words, model, vocabulary, num_features):
    ...:
    ...:     feature_vector = np.zeros((num_features,),dtype="float64")
    ...:     nwords = 0.
    ...:
    ...:     for word in words:
    ...:         if word in vocabulary:
    ...:             nwords = nwords + 1.
```

```
    ...:             feature_vector = np.add(feature_vector, model[word])
    ...:
    ...:     if nwords:
    ...:         feature_vector = np.divide(feature_vector, nwords)
    ...:
    ...:     return feature_vector
    ...:
    ...:
    ...: def averaged_word_vectorizer(corpus, model, num_features):
    ...:     vocabulary = set(model.wv.index2word)
    ...:     features = [average_word_vectors(tokenized_sentence, model, vocabulary,
                                              num_features)
    ...:                 for tokenized_sentence in corpus]
    ...:     return np.array(features)

In [17]: w2v_feature_array = averaged_word_vectorizer(corpus=tokenized_corpus, model=w2v_model,
    ...:                                              num_features=feature_size)
    ...: pd.DataFrame(w2v_feature_array)
```

Out[17]:

	0	1	2	3	4	5	6	7	8	9
0	-0.010540	-0.015367	0.005373	-0.020741	0.030717	-0.022407	-0.001724	0.004722	0.026881	0.011909
1	-0.017797	-0.013693	-0.003599	-0.015436	0.022831	-0.017905	0.010470	0.001540	0.025658	0.016208
2	-0.020869	-0.018273	-0.019681	-0.004124	-0.010980	0.001654	-0.001310	0.003395	0.003760	0.010851
3	-0.017561	-0.017866	-0.016438	-0.007601	-0.005687	-0.008843	-0.002385	0.001444	0.005643	0.012638
4	0.002371	-0.006731	0.017480	-0.014220	0.022088	-0.014882	0.003067	0.002605	0.021167	0.006461
5	-0.018306	-0.012056	-0.015671	-0.011617	-0.011667	-0.005490	0.005404	-0.003512	-0.003198	0.013306

Figure 4-19. *Averaged word vector feature set for our corpus documents*

Thus, we have our averaged word vector based feature set for all our corpus documents, as depicted by the dataframe in Figure 4-19. Let's use a different clustering algorithm this time known as Affinity Propagation to try to cluster our documents based on these new features. Affinity Propagation is based on the concept of message passing and you do not need to specify the number of clusters beforehand like you did in K-means clustering.

```
In [18]: from sklearn.cluster import AffinityPropagation
    ...:
    ...: ap = AffinityPropagation()
    ...: ap.fit(w2v_feature_array)
    ...: cluster_labels = ap.labels_
    ...: cluster_labels = pd.DataFrame(cluster_labels, columns=['ClusterLabel'])
    ...: pd.concat([corpus_df, cluster_labels], axis=1)
Out[18]:
                                        Document Category  ClusterLabel
0                    The sky is blue and beautiful.  weather             0
1                  Love this blue and beautiful sky!  weather             0
2       The quick brown fox jumps over the lazy dog.  animals             1
```

```
3   The brown fox is quick and the blue dog is lazy!  animals            1
4  The sky is very blue and the sky is very beaut...  weather            0
5        The dog is lazy but the brown fox is quick!  animals            1
```

The preceding output uses the averaged word vectors based on word embeddings to cluster the documents in our corpus and we can clearly see that it has obtained the right clusters! There are several other schemes of aggregating word vectors like using TF-IDF weights along with the word vector representations. Besides this there have been recent advancements in the field of Deep Learning where architectures like RNNs and LSTMs are also used for engineering features from text data.

Feature Engineering on Temporal Data

Temporal data involves datasets that change over a period of time and time-based attributes are of paramount importance in these datasets. Usually temporal attributes include some form of data, time, and timestamp values and often optionally include other metadata like time zones, daylight savings time information, and so on. Temporal data, especially time-series based data is extensively used in multiple domains like stock, commodity, and weather forecasting. You can load `feature_engineering_temporal.py` directly and start running the examples or use the jupyter notebook, `Feature Engineering on Temporal Data.ipynb`, for a more interactive experience. Let's load the following dependencies before we move on to acquiring some temporal data.

```
In [1]: import datetime
   ...: import numpy as np
   ...: import pandas as pd
   ...: from dateutil.parser import parse
   ...: import pytz
```

We will now use some sample time-based data as our source of temporal data by loading the following values in a dataframe.

```
In [2]: time_stamps = ['2015-03-08 10:30:00.360000+00:00', '2017-07-13 15:45:05.755000-07:00',
   ...:                '2012-01-20 22:30:00.254000+05:30', '2016-12-25
00:30:00.000000+10:00']
   ...: df = pd.DataFrame(time_stamps, columns=['Time'])
   ...: df
Out[2]:
                              Time
0  2015-03-08 10:30:00.360000+00:00
1  2017-07-13 15:45:05.755000-07:00
2  2012-01-20 22:30:00.254000+05:30
3  2016-12-25 00:30:00.000000+10:00
```

Of course by default, they are stored as strings or text in the dataframe so we can convert time into `Timestamp` objects by using the following code snippet.

```
In [3]: ts_objs = np.array([pd.Timestamp(item) for item in np.array(df.Time)])
   ...: df['TS_obj'] = ts_objs
   ...: ts_objs
Out[3]:
array([Timestamp('2015-03-08 10:30:00.360000+0000', tz='UTC'),
```

```
        Timestamp('2017-07-13 15:45:05.755000-0700', tz='pytz.FixedOffset(-420)'),
        Timestamp('2012-01-20 22:30:00.254000+0530', tz='pytz.FixedOffset(330)'),
        Timestamp('2016-12-25 00:30:00+1000', tz='pytz.FixedOffset(600)')], dtype=object)
```

You can clearly see from the temporal values that we have multiple components for each `Timestamp` object which include date, time, and even a time based offset, which can be used to identify the time zone also. Of course there is no way we can directly ingest or use these features in any Machine Learning model. Hence we need specific strategies to extract meaningful features from this data. In the following sections, we cover some of these strategies that you can start using on your own temporal data in the future.

Date-Based Features

Each temporal value has a date component that can be used to extract useful information and features pertaining to the date. These include features and components like year, month, day, quarter, day of the week, day name, day and week of the year, and many more. The following code depicts how we can obtain some of these features from our temporal data.

```
In [4]: df['Year'] = df['TS_obj'].apply(lambda d: d.year)
   ...: df['Month'] = df['TS_obj'].apply(lambda d: d.month)
   ...: df['Day'] = df['TS_obj'].apply(lambda d: d.day)
   ...: df['DayOfWeek'] = df['TS_obj'].apply(lambda d: d.dayofweek)
   ...: df['DayName'] = df['TS_obj'].apply(lambda d: d.weekday_name)
   ...: df['DayOfYear'] = df['TS_obj'].apply(lambda d: d.dayofyear)
   ...: df['WeekOfYear'] = df['TS_obj'].apply(lambda d: d.weekofyear)
   ...: df['Quarter'] = df['TS_obj'].apply(lambda d: d.quarter)
   ...:
   ...: df[['Time', 'Year', 'Month', 'Day', 'Quarter',
   ...:     'DayOfWeek', 'DayName', 'DayOfYear', 'WeekOfYear']]
```

Out[4]:

	Time	Year	Month	Day	Quarter	DayOfWeek	DayName	DayOfYear	WeekOfYear
0	2015-03-08 10:30:00.360000+00:00	2015	3	8	1	6	Sunday	67	10
1	2017-07-13 15:45:05.755000-07:00	2017	7	13	3	3	Thursday	194	28
2	2012-01-20 22:30:00.254000+05:30	2012	1	20	1	4	Friday	20	3
3	2016-12-25 00:30:00.000000+10:00	2016	12	25	4	6	Saturday	360	51

Figure 4-20. *Date based features in temporal data*

The features depicted in Figure 4-20 show some of the attributes we talked about earlier and have been derived purely from the date segment of each temporal value. Each of these features can be used as categorical features and further feature engineering can be done like one hot encoding, aggregations, binning, and more.

Time-Based Features

Each temporal value also has a time component that can be used to extract useful information and features pertaining to the time. These include attributes like hour, minute, second, microsecond, UTC offset, and more. The following code snippet extracts some of the previously mentioned time-based features from our temporal data.

```
In [5]: df['Hour'] = df['TS_obj'].apply(lambda d: d.hour)
   ...: df['Minute'] = df['TS_obj'].apply(lambda d: d.minute)
   ...: df['Second'] = df['TS_obj'].apply(lambda d: d.second)
   ...: df['MUsecond'] = df['TS_obj'].apply(lambda d: d.microsecond)
   ...: df['UTC_offset'] = df['TS_obj'].apply(lambda d: d.utcoffset())
   ...:
   ...: df[['Time', 'Hour', 'Minute', 'Second', 'MUsecond', 'UTC_offset']]
```

Out[5]:

	Time	Hour	Minute	Second	MUsecond	UTC_offset
0	2015-03-08 10:30:00.360000+00:00	10	30	0	360000	00:00:00
1	2017-07-13 15:45:05.755000-07:00	15	45	5	755000	-1 days +17:00:00
2	2012-01-20 22:30:00.254000+05:30	22	30	0	254000	05:30:00
3	2016-12-25 00:30:00.000000+10:00	0	30	0	0	10:00:00

Figure 4-21. *Time based features in temporal data*

The features depicted in Figure 4-21 show some of the attributes we talked about earlier which have been derived purely from the time segment of each temporal value. We can further engineer these features based on categorical feature engineering techniques and even derive other features like extracting time zones. Let's try to use binning to bin each temporal value into a specific time of the day by leveraging the Hour feature we just obtained.

```
In [6]: hour_bins = [-1, 5, 11, 16, 21, 23]
   ...: bin_names = ['Late Night', 'Morning', 'Afternoon', 'Evening', 'Night']
   ...: df['TimeOfDayBin'] = pd.cut(df['Hour'],
   ...:                             bins=hour_bins, labels=bin_names)
   ...: df[['Time', 'Hour', 'TimeOfDayBin']]
Out[6]:
                              Time  Hour TimeOfDayBin
0  2015-03-08 10:30:00.360000+00:00    10      Morning
1  2017-07-13 15:45:05.755000-07:00    15    Afternoon
2  2012-01-20 22:30:00.254000+05:30    22        Night
3  2016-12-25 00:30:00.000000+10:00     0   Late Night
```

Thus you can see from the preceding output that based on hour ranges (0-5, 5-11, 11-16, 16-21, 21-23) we have assigned a specific time of the day bin for each temporal value. The UTC offset component of the temporal data is very useful in knowing how far ahead or behind is that time value from the UTC (Coordinated Universal Time), which is the primary time standard that clocks and time are regulated from. This information can also be used to engineer new features like potential time zones from which each temporal value might have been obtained. The following code helps us achieve the same.

```
In [7]: df['TZ_info'] = df['TS_obj'].apply(lambda d: d.tzinfo)
   ...: df['TimeZones'] = df['TS_obj'].apply(lambda d: list({d.astimezone(tz).tzname()
   ...:                                 for tz in map(pytz.timezone,
   ...:                                               pytz.all_timezones_set)
   ...:                                   if d.astimezone(tz).utcoffset() == d.utcoffset()}))
   ...:
   ...: df[['Time', 'UTC_offset', 'TZ_info', 'TimeZones']]
```

Out[7]:

	Time	UTC_offset	TZ_info	TimeZones
0	2015-03-08 10:30:00.360000+00:00	00:00:00	UTC	[WET, UTC, UCT, GMT]
1	2017-07-13 15:45:05.755000-07:00	-1 days +17:00:00	pytz.FixedOffset(-420)	[MST, GMT+7, PDT]
2	2012-01-20 22:30:00.254000+05:30	05:30:00	pytz.FixedOffset(330)	[IST]
3	2016-12-25 00:30:00.000000+10:00	10:00:00	pytz.FixedOffset(600)	[VLAT, ChST, AEST, PGT, DDUT, GMT-10, CHUT]

Figure 4-22. *Time zone relevant features in temporal data*

Thus as we mentioned earlier, the features depicted in Figure 4-22 show some of the attributes pertaining to time zone relevant information for each temporal value. We can also get time components in other formats, like the Epoch, which is basically the number of seconds that have elapsed since January 1, 1970 (midnight UTC) and the Gregorian Ordinal, where January 1st of year 1 is represented as 1 and so on. The following code helps us extract these representations. See Figure 4-23.

```
In [8]: df['TimeUTC'] = df['TS_obj'].apply(lambda d: d.tz_convert(pytz.utc))
   ...: df['Epoch'] = df['TimeUTC'].apply(lambda d: d.timestamp())
   ...: df['GregOrdinal'] = df['TimeUTC'].apply(lambda d: d.toordinal())
   ...:
   ...: df[['Time', 'TimeUTC', 'Epoch', 'GregOrdinal']]
```

Out[8]:

	Time	TimeUTC	Epoch	GregOrdinal
0	2015-03-08 10:30:00.360000+00:00	2015-03-08 10:30:00.360000+00:00	1.425811e+09	735665
1	2017-07-13 15:45:05.755000-07:00	2017-07-13 22:45:05.755000+00:00	1.499986e+09	736523
2	2012-01-20 22:30:00.254000+05:30	2012-01-20 17:00:00.254000+00:00	1.327079e+09	734522
3	2016-12-25 00:30:00.000000+10:00	2016-12-24 14:30:00+00:00	1.482590e+09	736322

Figure 4-23. *Time components depicted in various representations*

Do note we converted each temporal value to UTC before deriving the other features. These alternate representations of time can be further used for easy date arithmetic. The epoch gives us time elapsed in seconds and the Gregorian ordinal gives us time elapsed in days. We can use this to derive further features like time elapsed from the current time or time elapsed from major events of importance based on the problem we are trying to solve. Let's compute the time elapsed for each temporal value since the current time. See Figure 4-24.

```
In [9]: curr_ts = datetime.datetime.now(pytz.utc)
   ...: # compute days elapsed since today
   ...: df['DaysElapsedEpoch'] = (curr_ts.timestamp() - df['Epoch']) / (3600*24)
   ...: df['DaysElapsedOrdinal'] = (curr_ts.toordinal() - df['GregOrdinal'])
   ...:
   ...: df[['Time', 'TimeUTC', 'DaysElapsedEpoch', 'DaysElapsedOrdinal']]
```

Out[9]:

	Time	TimeUTC	DaysElapsedEpoch	DaysElapsedOrdinal
0	2015-03-08 10:30:00.360000+00:00	2015-03-08 10:30:00.360000+00:00	860.207396	860
1	2017-07-13 15:45:05.755000-07:00	2017-07-13 22:45:05.755000+00:00	1.696917	2
2	2012-01-20 22:30:00.254000+05:30	2012-01-20 17:00:00.254000+00:00	2002.936564	2003
3	2016-12-25 00:30:00.000000+10:00	2016-12-24 14:30:00+00:00	203.040734	203

Figure 4-24. *Deriving elapsed time difference from current time*

Based on our computations, each new derived feature should give us the elapsed time difference between the current time and the time value in the `Time` column (actually `TimeUTC` since conversion to UTC is necessary). Both the values are almost equal to one another, which is expected. Thus you can use time and date arithmetic to extract and engineer more features which can help build better models. Alternate time representations enable you to do date time arithmetic directly instead of dealing with specific API methods of `Timestamp` and `datetime` objects from Python. However you can use any method to get to the results you want. It's all about ease of use and efficiency!

Feature Engineering on Image Data

Another very popular format of unstructured data is images. Sound and visual data in the form of images, video, and audio are very popular sources of data which pose a lot of challenge to data scientists in terms of processing, storage, feature extraction and modeling. However their benefits as sources of data are quite rewarding especially in the field of artificial intelligence and computer vision. Due to the unstructured nature of data, it is not possible to directly use images for training models. If you are given a raw image, you might have a hard time trying to think of ways to represent it so that any Machine Learning algorithm can utilize it for model training. There are various strategies and techniques that can be used in this case to engineer the right features from images. One of the core principles to remember when dealing with images is that any image can be represented as a matrix of numeric pixel values. With that thought in mind, let's get started! You can load `feature_engineering_image.py` directly and start running the examples or use the jupyter notebook, `Feature Engineering on Image Data.ipynb`, for a more interactive experience. Let's start by loading the necessary dependencies and configuration settings.

```
In [1]: import skimage
   ...: import numpy as np
   ...: import pandas as pd
   ...: import matplotlib.pyplot as plt
   ...: from skimage import io
   ...:
   ...: %matplotlib inline
```

The `scikit-image` (`skimage`) library is an excellent framework consisting of several useful interfaces and algorithms for image processing and feature extraction. Besides this, we will also leverage the `mahotas` framework, which is useful in computer vision and image processing. Open CV is another useful framework that you can check out if interested in aspects pertaining to computer vision. Let's now look at ways to represent images as useful feature vector representations.

Image Metadata Features

There are tons of useful features obtainable from the image metadata itself without even processing the image. Most of this information can be found from the EXIF data, which is usually recorded for each image by the device when the picture is being taken. Following are some of the popular features that are obtainable from the image EXIF data.

- Image create date and time
- Image dimensions
- Image compression format
- Device make and model
- Image resolution and aspect ratio
- Image artist
- Flash, aperture, focal length, and exposure

For more details on what other data points can be used as features from image EXIF metadata, you can refer to https://sno.phy.queensu.ca/~phil/exiftool/TagNames/EXIF.html, which lists the possible EXIF tags.

Raw Image and Channel Pixels

An image can be represented by the value of each of its pixels as a two dimensional array. We can leverage numpy arrays for this. However, color images usually have three components also known as channels. The `R`, `G`, and `B` channels stand for the red, green, and blue channels, respectively. This can be represented as a three dimensional array `(m, n, c)` where `m` indicates the number of rows in the image, `n` indicates the number of columns. These are determined by the image dimensions. The `c` indicates which channel it represents (`R`, `G` or `B`). Let's load some sample color images now and try to understand their representation.

```
In [2]: cat = io.imread('datasets/cat.png')
   ...: dog = io.imread('datasets/dog.png')
   ...: df = pd.DataFrame(['Cat', 'Dog'], columns=['Image'])
   ...:
   ...: print(cat.shape, dog.shape)
(168, 300, 3) (168, 300, 3)

In [3]: fig = plt.figure(figsize = (8,4))
   ...: ax1 = fig.add_subplot(1,2, 1)
   ...: ax1.imshow(cat)
   ...: ax2 = fig.add_subplot(1,2, 2)
   ...: ax2.imshow(dog)
```

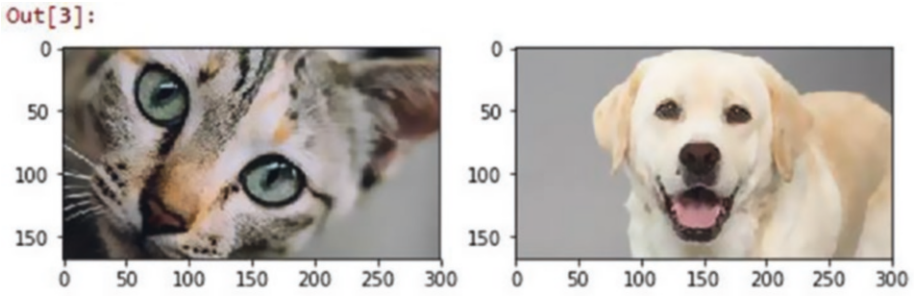

Figure 4-25. *Our two sample color images*

We can clearly see from Figure 4-25 that we have two images of a cat and a dog having dimensions 168x300 pixels where each row and column denotes a specific pixel of the image. The third dimension indicates these are color images having three color channels. Let's now try to use numpy indexing to slice out and extract the three color channels separately for the dog image.

```
In [4]: dog_r = dog.copy() # Red Channel
   ...: dog_r[:,:,1] = dog_r[:,:,2] = 0 # set G,B pixels = 0
   ...: dog_g = dog.copy() # Green Channel
   ...: dog_g[:,:,0] = dog_r[:,:,2] = 0 # set R,B pixels = 0
   ...: dog_b = dog.copy() # Blue Channel
   ...: dog_b[:,:,0] = dog_b[:,:,1] = 0 # set R,G pixels = 0
   ...:
   ...: plot_image = np.concatenate((dog_r, dog_g, dog_b), axis=1)
   ...: plt.figure(figsize = (10,4))
   ...: plt.imshow(plot_image)
```

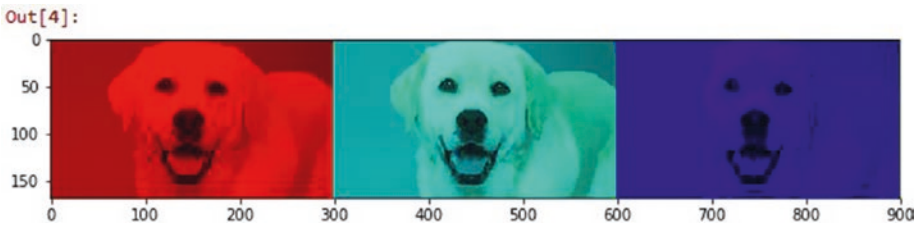

Figure 4-26. *Extracting red, green, and blue channels from our color RGB image*

We can clearly see from Figure 4-26 how we can easily use numpy indexing and extract out the three color channels from the sample image. You can now refer to any of these channel's raw image pixel matrix and even flatten it if needed to form a feature vector.

```
In [5]: dog_r[:,:,0]
Out[5]:
array([[160, 160, 160, ..., 113, 113, 112],
       [160, 160, 160, ..., 113, 113, 112],
       ...,
```

```
       [165, 165, 165, ..., 212, 211, 210],
       [165, 165, 165, ..., 210, 210, 209],
       [164, 164, 164, ..., 209, 209, 209]], dtype=uint8)
```

This image pixel matrix is a two-dimensional matrix so you can extract features from this further or even flatten it to a one-dimensional vector to use as inputs for any Machine Learning algorithm.

Grayscale Image Pixels

If you are dealing with color images, it might get difficult working with multiple channels and three-dimensional arrays. Hence converting images to grayscale is a nice way of keeping the necessary pixel intensity values but getting an easy to process two-dimensional image. Grayscale images usually capture the luminance or intensity of each pixel such that each pixel value can be computed using the equation

$$Y = 0.2125 \text{ x } R + 0.7154 \text{ x } G + 0.0721 \text{ x } B$$

Where *R, G & B* are the pixel values of the three channels and *Y* captures the final pixel intensity information and is usually ranges from `0`(complete intensity absence - black) to `1`(complete intensity presence - white). The following snippet shows us how to convert `RGB` color images to grayscale and extract the raw pixel values, which can be used as features.

```
In [6]: from skimage.color import rgb2gray
   ...:
   ...: cgs = rgb2gray(cat)
   ...: dgs = rgb2gray(dog)
   ...:
   ...: print('Image shape:', cgs.shape, '\n')
   ...:
   ...: # 2D pixel map
   ...: print('2D image pixel map')
   ...: print(np.round(cgs, 2), '\n')
   ...:
   ...: # flattened pixel feature vector
   ...: print('Flattened pixel map:', (np.round(cgs.flatten(), 2)))
Image shape: (168, 300)

2D image pixel map
[[ 0.42  0.41  0.41 ...,  0.5   0.52  0.53]
 [ 0.41  0.41  0.4  ...,  0.51  0.52  0.54]
 ...,
 [ 0.11  0.11  0.1  ...,  0.51  0.51  0.51]
 [ 0.11  0.11  0.1  ...,  0.51  0.51  0.51]]

Flattened pixel map: [ 0.42  0.41  0.41 ...,  0.51  0.51  0.51]
```

Binning Image Intensity Distribution

We already obtained the raw image intensity values for the grayscale images in the previous section. One approach would be to use these raw pixel values themselves as features. Another approach would be to binning the image intensity distribution based on intensity values using a histogram and using the bins as features. The following code snippet shows us how the image intensity distribution looks for the two sample images.

```
In [7]: fig = plt.figure(figsize = (8,4))
   ...: ax1 = fig.add_subplot(2,2, 1)
   ...: ax1.imshow(cgs, cmap="gray")
   ...: ax2 = fig.add_subplot(2,2, 2)
   ...: ax2.imshow(dgs, cmap='gray')
   ...: ax3 = fig.add_subplot(2,2, 3)
   ...: c_freq, c_bins, c_patches = ax3.hist(cgs.flatten(), bins=30)
   ...: ax4 = fig.add_subplot(2,2, 4)
   ...: d_freq, d_bins, d_patches = ax4.hist(dgs.flatten(), bins=30)
```

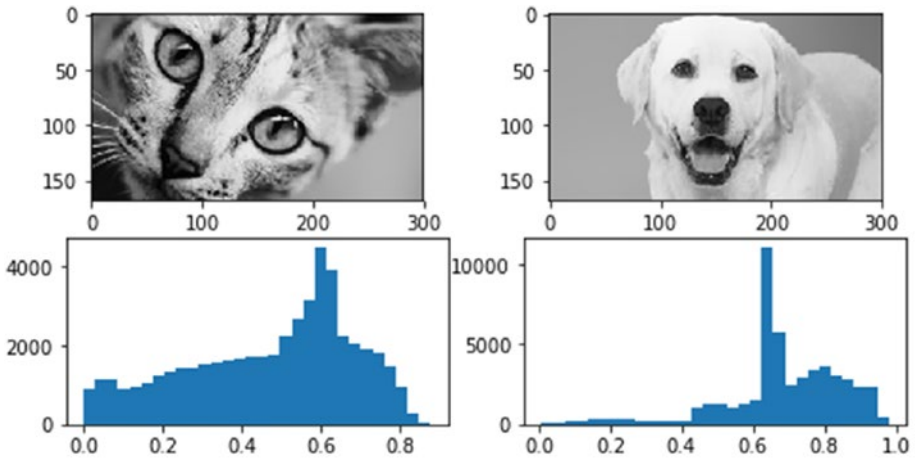

Figure 4-27. *Binning image intensity distributions with histograms*

As we mentioned, image intensity ranges from `0` to `1` and is evident by the x-axes depicted in Figure 4-27. The y-axes depict the frequency of the respective bins. We can clearly see that the dog image has more concentration of the bin frequencies around `0.6 - 0.8` indicating higher intensity and the reason for that being that the Labrador dog is white in color and white has a high intensity value like we mentioned in the previous section. The variables `c_freq`, `c_bins`, and `d_freq`, `d_bins` can be used to get the numeric values pertaining to the bins and used as features.

Image Aggregation Statistics

We already obtained the raw image intensity values for the grayscale images in the previous section. One approach would be to use them as features directly or use some level of aggregations and statistical measures which can be obtained from the pixels and intensity. We already saw an approach of binning intensity values using histograms. In this section, we use descriptive statistical measures and aggregations to compute specific features from the image pixel values.

We can compute RGB ranges for each image by basically subtracting the maximum from the minimum value for pixel values in each channel. The following code helps us achieve this.

```
In [8]: from scipy.stats import describe
   ...:
   ...: cat_rgb = cat.reshape((168*300), 3).T
   ...: dog_rgb = dog.reshape((168*300), 3).T
```

```
   ...:
   ...: cs = describe(cat_rgb, axis=1)
   ...: ds = describe(dog_rgb, axis=1)
   ...:
   ...: cat_rgb_range = cs.minmax[1] - cs.minmax[0]
   ...: dog_rgb_range = ds.minmax[1] - ds.minmax[0]
   ...: rgb_range_df = pd.DataFrame([cat_rgb_range, dog_rgb_range],
   ...:                             columns=['R_range', 'G_range', 'B_range'])
   ...: pd.concat([df, rgb_range_df], axis=1)
Out[8]:
  Image  R_range  G_range  B_range
0   Cat      240      223      235
1   Dog      246      250      246
```

We can then use these range features as specific characteristic attributes of each image. Besides this, we can also compute other metrics like mean, median, variance, skewness, and kurtosis for each image channel as follows.

```
In [9]: cat_stats= np.array([np.round(cs.mean, 2),np.round(cs.variance, 2),
   ...:                      np.round(cs.kurtosis, 2),np.round(cs.skewness, 2),
   ...:                      np.round(np.median(cat_rgb, axis=1), 2)]).flatten()
   ...: dog_stats= np.array([np.round(ds.mean, 2),np.round(ds.variance, 2),
   ...:                        np.round(ds.kurtosis, 2),np.round(ds.skewness, 2),
   ...:                        np.round(np.median(dog_rgb, axis=1), 2)]).flatten()
   ...:
   ...: stats_df = pd.DataFrame([cat_stats, dog_stats],
   ...:                         columns=['R_mean', 'G_mean', 'B_mean', 'R_var', 'G_var',
   ...:                                  'B_var', 'R_kurt', 'G_kurt', 'B_kurt', 'R_skew',
   ...:                                  'G_skew', 'B_skew', 'R_med', 'G_med', 'B_med'])
   ...: pd.concat([df, stats_df], axis=1)
```

Out[9]:

	Image	R_mean	G_mean	B_mean	R_var	G_var	B_var	R_kurt	G_kurt	B_kurt	R_skew	G_skew	B_skew	R_med	G_med	B_med
0	Cat	127.48	118.80	111.94	3054.04	2863.78	3003.05	-0.63	-0.77	-0.94	-0.48	-0.50	-0.25	140.0	132.0	120.0
1	Dog	184.46	173.46	160.77	1887.71	1776.00	1574.73	1.30	2.24	2.32	-0.96	-1.12	-1.09	185.0	169.0	165.0

Figure 4-28. *Image channel aggregation statistical features*

We can observe from the features obtained in Figure 4-28 that the mean, median, and kurtosis values for the various channels for the dog image are mostly greater than corresponding ones in the cat image. Variance and skewness are however more for the cat image.

Edge Detection

One of the more interesting and sophisticated techniques involve detecting edges in an image. Edge detection algorithms can be used to detect sharp intensity and brightness changes in an image and find areas of interest. The canny edge detector algorithm developed by John Canny is one of the most widely used edge detector algorithms today. This algorithm typically involves using a Gaussian distribution with a specific standard deviation σ (sigma) to smoothen and denoise the image. Then we apply a Sobel filter to extract image intensity gradients. Norm value of this gradient is used to determine the edge strength.

Potential edges are thinned down to curves with width of 1 pixel and hysteresis based thresholding is used to label all points above a specific high threshold as edges and then recursively use the low threshold value to label points above the low threshold as edges connected to any of the previously labeled points. The following code applied the canny edge detector to our sample images.

```
In [10]: from skimage.feature import canny
    ...:
    ...: cat_edges = canny(cgs, sigma=3)
    ...: dog_edges = canny(dgs, sigma=3)
    ...:
    ...: fig = plt.figure(figsize = (8,4))
    ...: ax1 = fig.add_subplot(1,2, 1)
    ...: ax1.imshow(cat_edges, cmap='binary')
    ...: ax2 = fig.add_subplot(1,2, 2)
    ...: ax2.imshow(dog_edges, cmap='binary')
```

Out[10]:

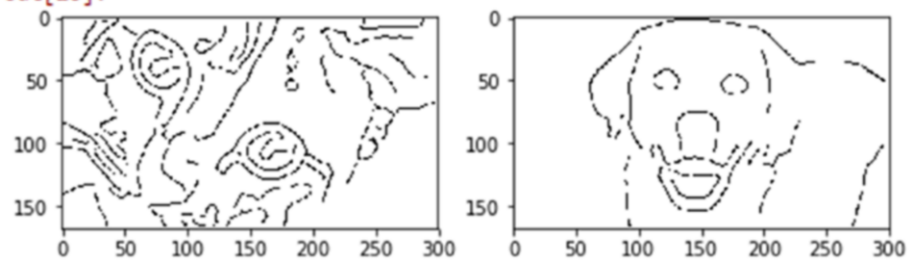

Figure 4-29. *Canny edge detection to extract edge based features*

The image plots based on the edge feature arrays depicted in Figure 4-29 clearly show the prominent edges of our cat and dog. You can use these edge feature arrays (`cat_edges` and `dog_edges`) by flattening them, extracting pixel values and positions pertaining to the edges (non-zero values), or even by aggregating them like finding out the total number of pixels making edges, mean value, and so on.

Object Detection

Another interesting technique in the world of computer vision is object detection where features useful in highlighting specific objects in the image are detected and extracted. The histogram of oriented gradients, also known as HOG, is one of the techniques that's extensively used in object detection. Going into the details of this technique would not be possible in the current scope but for the process of feature engineering, you need to remember that the HOG algorithm works by following a sequence of steps similar to edge detection. The image is normalized and denoised to remove excess illumination effects. First order image gradients are computed to capture image attributes like contour, texture, and so on. Gradient histograms are built on top of these gradients based on specific windows called cells. Finally these cells are normalized and a flattened feature descriptor is obtained, which can be used as a feature vector for our models. The following code shows the HOG object detection technique on our sample images.

```
In [11]: from skimage.feature import hog
    ...: from skimage import exposure
    ...:
```

```
...: fd_cat, cat_hog = hog(cgs, orientations=8, pixels_per_cell=(8, 8),
...:                       cells_per_block=(3, 3), visualise=True)
...: fd_dog, dog_hog = hog(dgs, orientations=8, pixels_per_cell=(8, 8),
...:                       cells_per_block=(3, 3), visualise=True)
...:
...: # rescaling intensity to get better plots
...: cat_hogs = exposure.rescale_intensity(cat_hog, in_range=(0, 0.04))
...: dog_hogs = exposure.rescale_intensity(dog_hog, in_range=(0, 0.04))
...:
...: fig = plt.figure(figsize = (10,4))
...: ax1 = fig.add_subplot(1,2, 1)
...: ax1.imshow(cat_hogs, cmap='binary')
...: ax2 = fig.add_subplot(1,2, 2)
...: ax2.imshow(dog_hogs, cmap='binary')
```

Out[11]:

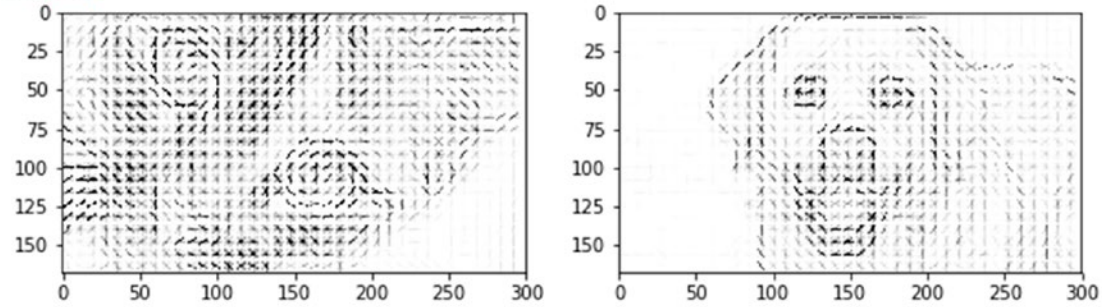

Figure 4-30. *HOG object detector to extract features based on object detection*

The image plots in Figure 4-30 show us how the HOG detector has identified the objects in our sample images. You can also get the flattened feature descriptors as follows.

```
In [12]: print(fd_cat, fd_cat.shape)
[ 0.00288784  0.00301086  0.0255757  ...,  0.          0.          0.        ] (47880,)
```

Localized Feature Extraction

We have talked about aggregating pixel values from two-dimensional image or feature matrices and also flattening them into feature vectors. Localized feature extraction based techniques are slightly better methods which try to detect and extract localized feature descriptors on various small localized regions of our input images. This is hence rightly named localized feature extraction. We will be using the popular and patented SURF algorithm invented by Herbert Bay, et al. SURF stands for Speeded Up Robust Features. The main idea is to get scale invariant local feature descriptors from images which can be used later as image features. This algorithm is similar to the popular SIFT algorithm. There are mainly two major phases in this algorithm. The first phase is to detect points of interest using square shaped filters and hessian matrices. The second phase is to build feature descriptors by extracting localized features around these points of interest. There are usually computed by taking a localized square image region around a point of interest and then aggregating Haar wavelet responses at specific interval based sample points. We use the `mahotas` Python framework for extracting SURF feature descriptors from our sample images.

```
In [13]: from mahotas.features import surf
    ...: import mahotas as mh
    ...:
    ...: cat_mh = mh.colors.rgb2gray(cat)
    ...: dog_mh = mh.colors.rgb2gray(dog)
    ...:
    ...: cat_surf = surf.surf(cat_mh, nr_octaves=8, nr_scales=16, initial_step_size=1,
                              threshold=0.1, max_points=50)
    ...: dog_surf = surf.surf(dog_mh, nr_octaves=8, nr_scales=16, initial_step_size=1,
                              threshold=0.1, max_points=54)
    ...:
    ...: fig = plt.figure(figsize = (10,4))
    ...: ax1 = fig.add_subplot(1,2, 1)
    ...: ax1.imshow(surf.show_surf(cat_mh, cat_surf))
    ...: ax2 = fig.add_subplot(1,2, 2)
    ...: ax2.imshow(surf.show_surf(dog_mh, dog_surf))
```

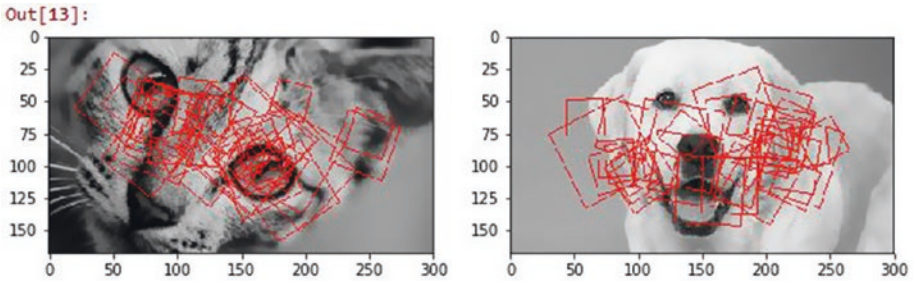

Figure 4-31. *Localized feature extraction with SURF*

The square boxes in the image plots in Figure 4-31 depict the square image regions around the points of interest which were used for localized feature extraction. You can also use the `surf.dense(...)` function to extract uniform dimensional feature descriptors at dense points with regular interval spacing in pixels. The following code depicts how to achieve this.

```
In [14]: cat_surf_fds = surf.dense(cat_mh, spacing=10)
    ...: dog_surf_fds = surf.dense(dog_mh, spacing=10)
    ...: cat_surf_fds.shape
Out[14]: (140, 64)
```

We see from the preceding output that we have obtained 140 feature descriptors of size 64 (elements) each. You can further apply other schemes on this like aggregation, flattening, and so on to derive further features. Another sophisticated technique that you can use to extract features on these SURF feature descriptors is to use the visual bag of words model, which we discuss in the next section.

Visual Bag of Words Model

We have seen the effectiveness of the popular Bag of Words model in extracting meaningful features from unstructured text documents. Bag of words refers to the document being broken down into its constituents, words and computing frequency of occurrences or other measures like tf-idf. Similarly, in case of image raw pixel matrices or derived feature descriptors from other algorithms, we can apply a bag of words principle. However the constituents will not be words in this case but they will be subset of features/pixels extracted from images which are similar to each other.

Imagine you have multiple pictures of octopuses and you were able to extract the `140` dense surf features each having `64` values in each feature vector. You can now use an unsupervised learning algorithm like clustering to extract clusters of similar feature descriptors. Each cluster can be labeled as a *visual word* or a *visual feature*. Subsequently, each feature descriptor can be binned into one of these clusters or visual words. Thus, you end up getting a one-dimensional visual bag of words vector with counts of number of feature descriptors assigned to each of the visual words for the `140x64` feature descriptor matrix. Each feature or visual word tends to capture some portion of the images that are similar to each other like octopus eyes, tentacles, suckers, and so on, as depicted in Figure 4-32.

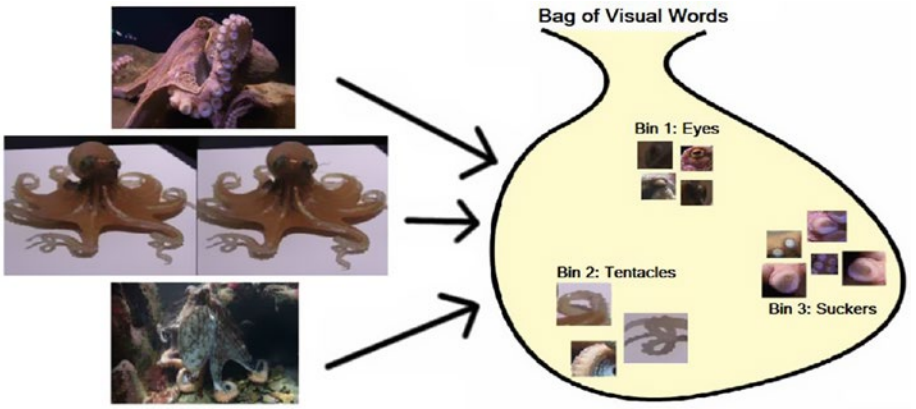

Figure 4-32. *Visual bag of words (Courtesy of Ian London, Image Classification in Python with Visual Bag of Words)*

The basic idea is hence to get a feature descriptor matrix from using any algorithm like SURF, apply an unsupervised algorithm like K-means clustering, and extract out k bins or visual features/words and their counts (based on number of feature descriptors assigned to each bin). Then for each subsequent image, once you extract the feature descriptors, you can use the K-means model to assign each feature descriptor to one of the visual feature clusters and get a one-dimensional vector of counts. This is depicted in Figure 4-33 for a sample octopus image, assuming our VBOW (Visual Bag of Words) model has three bins of eyes, tentacles, and suckers.

Figure 4-33. *Transforming an image into a VBOW vector (Courtesy of Ian London, Image Classification in Pythonwith Visual Bag of Words)*

Thus you can see from Figure 4-33, how a two-dimensional image and its corresponding feature descriptors can be easily transformed into a one-dimensional VBOW vector `[1, 3, 5]`. Going into extensive details of the VBOW model would not be possible in the current scope, but I would like to thank my friend and fellow data scientist, Ian London, for helping me out with providing the two figures on VBOW models. I would also recommend you to check out his wonderful blog article https://ianlondon.github.io/blog/visual-bag-of-words/, which talks about using the VBOW model for image classification.

We will now use our 140x64 SURF feature descriptors for our two sample images and use K-means clustering on them and compute VBOW vectors for each image by assigning each feature descriptor to one of the bins. We will take k=20 in this case. See Figure 4-34.

```
In [15]: from sklearn.cluster import KMeans
    ...:
    ...: k = 20
    ...: km = KMeans(k, n_init=100, max_iter=100)
    ...:
    ...: surf_fd_features = np.array([cat_surf_fds, dog_surf_fds])
    ...: km.fit(np.concatenate(surf_fd_features))
    ...:
    ...: vbow_features = []
    ...: for feature_desc in surf_fd_features:
    ...:     labels = km.predict(feature_desc)
    ...:     vbow = np.bincount(labels, minlength=k)
    ...:     vbow_features.append(vbow)
    ...:
    ...: vbow_df = pd.DataFrame(vbow_features)
    ...: pd.concat([df, vbow_df], axis=1)
```

Out[15]:

	Image	0	1	2	3	4	5	6	7	8	...	10	11	12	13	14	15	16	17	18	19
0	Cat	8	16	11	7	3	0	16	6	0	...	0	13	1	0	1	15	10	2	14	2
1	Dog	3	10	6	16	9	16	9	5	3	...	2	10	3	2	3	7	7	6	7	2

Figure 4-34. *Transforming SURF descriptors into VBOW vectors for sample images*

You can see how easy it is to transform complex two-dimensional SURF feature descriptor matrices into easy to interpret VBOW vectors. Let's now take a new image and think about how we could apply the VBOW pipeline. First we would need to extract the SURF feature descriptors from the image using the following snippet (This is only to depict the localized image subsets used in SURF we will actually use the dense features as before.) See Figure 4-35.

```
In [16]: new_cat = io.imread('datasets/new_cat.png')
    ...: newcat_mh = mh.colors.rgb2gray(new_cat)
    ...: newcat_surf = surf.surf(newcat_mh, nr_octaves=8, nr_scales=16, initial_step_size=1,
                                 threshold=0.1, max_points=50)
    ...:
    ...: fig = plt.figure(figsize = (10,4))
    ...: ax1 = fig.add_subplot(1,2, 1)
    ...: ax1.imshow(surf.show_surf(newcat_mh, newcat_surf))
```

Out[16]:

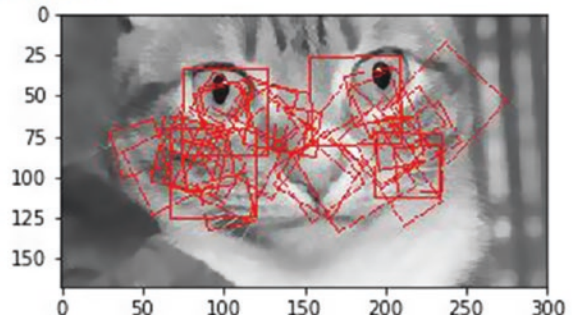

Figure 4-35. *Localized feature extraction with SURF for new image*

Let's now extract the dense SURF features and transform them into a VBOW vector using our previously trained VBOW model. The following code helps us achieve this. See Figure 4-36.

```
In [17]: new_surf_fds = surf.dense(newcat_mh, spacing=10)
    ...:
    ...: labels = km.predict(new_surf_fds)
    ...: new_vbow = np.bincount(labels, minlength=k)
    ...: pd.DataFrame([new_vbow])
```

Out[17]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	9	5	11	0	9	4	19	9	0	16	0	7	3	0	0	7	20	3	16	2

Figure 4-36. *Transforming new image SURF descriptors into a VBOW vector*

Thus you can see the final VBOW feature vector for the new image based on SURF feature descriptors. This is also an example of using an unsupervised Machine Learning model for feature engineering. You can now compare the similarity of this new image with the other two sample images using some similarity metrics.

```
In [18]: from sklearn.metrics.pairwise import euclidean_distances, cosine_similarity
    ...:
    ...: eucdis = euclidean_distances(new_vbow.reshape(1,-1) , vbow_features)
    ...: cossim = cosine_similarity(new_vbow.reshape(1,-1) , vbow_features)
    ...:
    ...: result_df = pd.DataFrame({'EuclideanDistance': eucdis[0],
    ...:                'CosineSimilarity': cossim[0]})
    ...: pd.concat([df, result_df], axis=1)
Out[18]:
  Image  CosineSimilarity  EuclideanDistance
0   Cat          0.871609          21.260292
1   Dog          0.722096          30.000000
```

Based on the distance and similarity metrics, we can see that our new image (of a cat) is definitely closer to the cat image than the dog image. Try this out with a bigger dataset to get better results!

Automated Feature Engineering with Deep Learning

We have used a lot of simple and sophisticated feature engineering techniques so far in this section. Building complex feature engineering systems and pipelines is time consuming and building algorithms for the same is even more tasking. Deep Learning is a novel and new approach toward automating this complex task of feature engineering by making the machine extract features automatically by learning multiple layered and complex representations of the underlying raw data.

Convolutional neural networks or CNNs are extensively used for automated feature extraction in images. We have already covered the basic principles of CNNs in Chapter 1. Go ahead and refresh your memory you heading to the "Important Concepts" sub-section under the "Deep Learning" section in Chapter 1. Just like we mentioned before, the idea of CNNs operate on the principles of convolution and pooling besides your regular activation function layers.

Convolutional layers typically slides or convolves learnable filters (also known as kernels or convolution matrix) across the entire width and height of the input image pixels. Dot products between the input pixels and the filter are computed at each position on sliding the filter. Two-dimensional activation maps for the filter get created and consequently the network is able to learn these filters when it activates on detecting specific features like edges, corners and so on. If we take n filters, we will get n separate two-dimensional activation maps, which can then be stacked along the depth dimension to get the output volume.

Pooling is a kind of aggregation or downsampling layer where typically a non-linear downsampling operation is inserted between convolutional layers. Filters are applied here too. They are slided along the convolution output matrix and, for each sliding operation, also known as a stride, elements in the slice of matrix covered by the pooling filter are either summed (Sum pooling) or averaged (Mean pooling) or the maximum value is selected (Max pooling). More than often max pooling works really well in several real-world scenarios. Pooling helps in reducing feature dimensionality and control model overfitting. Let's now try to use Deep Learning for automated feature extraction on our sample images using CNNs. Load the following dependencies necessary for building deep networks.

```
In [19]: from keras.models import Sequential
    ...: from keras.layers.convolutional import Conv2D
    ...: from keras.layers.convolutional import MaxPooling2D
    ...: from keras import backend as K
Using TensorFlow backend.
```

You can use Theano or Tensorflow as your backend Deep Learning framework for `keras` to work on. I am using `tensorflow` in this scenario. Let's build a basic two-layer CNN now with a Max Pooling layer between them.

```
In [20]: model = Sequential()
    ...: model.add(Conv2D(4, (4, 4), input_shape=(168, 300, 3), activation='relu',
    ...:                  kernel_initializer='glorot_uniform'))
    ...: model.add(MaxPooling2D(pool_size=(2, 2)))
    ...: model.add(Conv2D(4, (4, 4), activation='relu',
    ...:                 kernel_initializer='glorot_uniform'))
```

We can actually visualize this network architecture using the following code snippet to understand the layers that have been used in this network, in a better way.

```
In [21]: from IPython.display import SVG
    ...: from keras.utils.vis_utils import model_to_dot
    ...:
    ...: SVG(model_to_dot(model, show_shapes=True,
    ...:                  show_layer_names=True, rankdir='TB').create(prog='dot', format='svg'))
```

Out[21]:

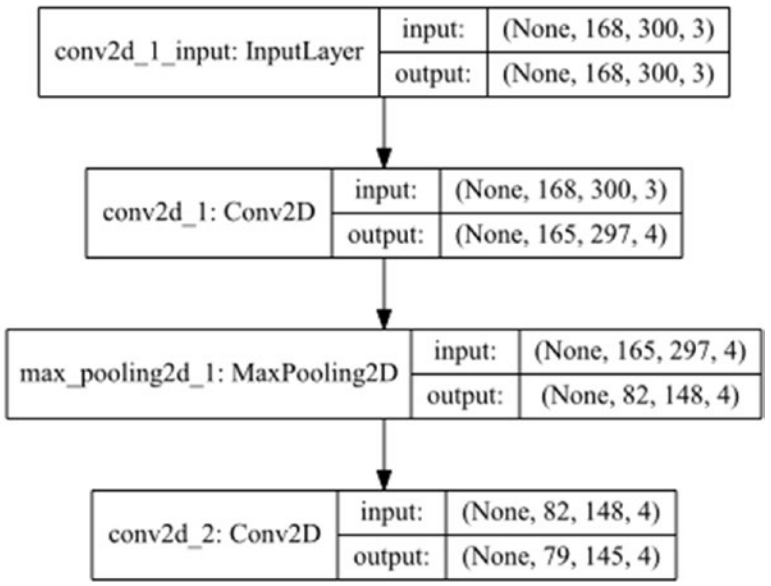

Figure 4-37. *Visualizing our two-layer convolutional neural network architecture*

You can now understand from the depiction in Figure 4-37 that we are using two two-dimensional Convolutional layers containing four `(4x4)` filters. We also have a Max Pool layer between them of size `(2x2)` for some downsampling. Let's now build some functions to extract features from these intermediate network layers.

```
In [22]: first_conv_layer = K.function([model.layers[0].input, K.learning_phase()],
    ...:                               [model.layers[0].output])
    ...: second_conv_layer = K.function([model.layers[0].input, K.learning_phase()],
    ...:                                [model.layers[2].output])
```

Let's now use these functions to extract the feature representations learned in the convolutional layers and visualize these features to see what the network is trying to learn from the images.

```
In [23]: catr = cat.reshape(1, 168, 300,3)
    ...:
    ...: # extract features
    ...: first_conv_features = first_conv_layer([catr])[0][0]
    ...: second_conv_features = second_conv_layer([catr])[0][0]
    ...:
    ...: # view feature representations
    ...: fig = plt.figure(figsize = (14,4))
    ...: ax1 = fig.add_subplot(2,4, 1)
    ...: ax1.imshow(first_conv_features[:,:,0])
    ...: ax2 = fig.add_subplot(2,4, 2)
    ...: ax2.imshow(first_conv_features[:,:,1])
    ...: ax3 = fig.add_subplot(2,4, 3)
    ...: ax3.imshow(first_conv_features[:,:,2])
    ...: ax4 = fig.add_subplot(2,4, 4)
    ...: ax4.imshow(first_conv_features[:,:,3])
    ...:
    ...: ax5 = fig.add_subplot(2,4, 5)
    ...: ax5.imshow(second_conv_features[:,:,0])
    ...: ax6 = fig.add_subplot(2,4, 6)
    ...: ax6.imshow(second_conv_features[:,:,1])
    ...: ax7 = fig.add_subplot(2,4, 7)
    ...: ax7.imshow(second_conv_features[:,:,2])
    ...: ax8 = fig.add_subplot(2,4, 8)
    ...: ax8.imshow(second_conv_features[:,:,3])
```

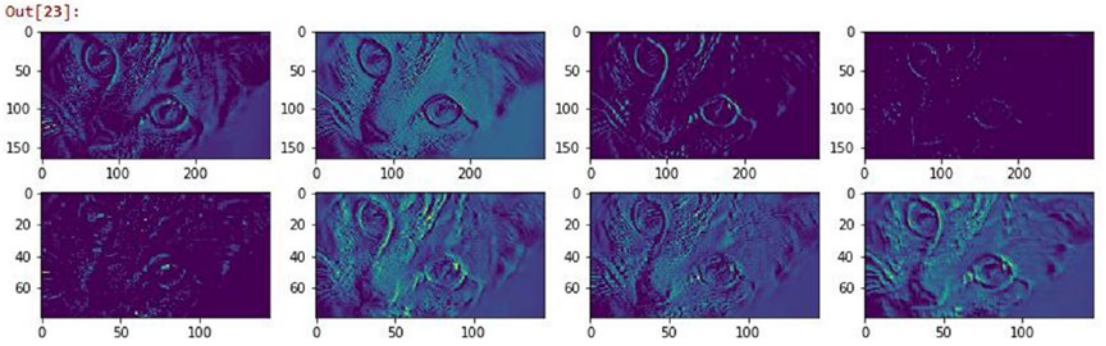

Figure 4-38. *Intermediate feature maps obtained after passing though convolutional Layers*

The feature map visualizations depicted in Figure 4-38 are definitely interesting. You can clearly see that each feature matrix produced by the convolutional neural network is trying to learn something about the image like its texture, corners, edges, illumination, hue, brightness, and so on. This should give you an idea of how these activation feature maps can then be used as features for images. In fact you can stack the output of a CNN, flatten it if needed, and pass it as an input layer to a multi-layer fully connected perceptron neural network and use it to solve the problem of image classification. This should give you a head start on automated feature extraction with the power of Deep Learning!

Don't worry if you did not understand some of the terms mentioned in this section; we will cover Deep Learning and CNNs in more depth in a subsequent chapter. If can't wait to get started with Deep Learning, you can fire up the bonus notebook provided with this chapter, called `Bonus - Classifying handwritten digits using Deep CNNs.ipynb`, for a complete real-world example of applying CNNs and Deep Learning to classify hand-written digits!

Feature Scaling

When dealing with numeric features, we have specific attributes which may be completely unbounded in nature, like view counts of a video or web page hits. Using the raw values as input features might make models biased toward features having really high magnitude values. These models are typically sensitive to the magnitude or scale of features like linear or logistic regression. Other models like tree based methods can still work without feature scaling. However it is still recommended to normalize and scale down the features with feature scaling, especially if you want to try out multiple Machine Learning algorithms on input features. We have already seen some examples of scaling and transforming features using log and box-cox transforms earlier in this chapter. In this section, we look at some popular feature scaling techniques. You can load `feature_scaling.py` directly and start running the examples or use the jupyter notebook, `Feature Scaling.ipynb` for a more interactive experience. Let's start by loading the following dependencies and configurations.

```
In [1]: from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
   ...: import numpy as np
   ...: import pandas as pd
   ...: np.set_printoptions(suppress=True)
```

Let's now load some sample data of user views pertaining to online videos. The following snippet creates this sample dataset.

```
In [2]: views = pd.DataFrame([1295., 25., 19000., 5., 1., 300.], columns=['views'])
   ...: views
Out[2]:
     views
0   1295.0
1     25.0
2  19000.0
3      5.0
4      1.0
5    300.0
```

From the preceding dataframe we can see that we have five videos that have been viewed by users and the total view count for each video is depicted by the feature views. It is quite evident that some videos have been viewed a lot more than the others, giving a rise to values of high scale and magnitude. Let's look at how we can scale this feature using several handy techniques.

Standardized Scaling

The standard scaler tries to standardize each value in a feature column by removing the mean and scaling the variance to be 1 from the values. This is also known as centering and scaling and can be denoted mathematically as

$$SS(X_i) = \frac{X_i - \mu_X}{\sigma_X}$$

where each value in feature X is subtracted by the mean μ_X and the resultant is divided by the standard deviation σ_X. This is also popularly known as Z-score scaling. You can also divide the resultant by the variance instead of the standard deviation if needed. The following snippet helps us achieve this.

```
In [3]: ss = StandardScaler()
   ...: views['zscore'] = ss.fit_transform(views[['views']])
   ...: views
Out[3]:
     views    zscore
0   1295.0 -0.307214
1     25.0 -0.489306
2  19000.0  2.231317
3      5.0 -0.492173
4      1.0 -0.492747
5    300.0 -0.449877
```

We can see the standardized and scaled values in the `zscore` column in the preceding dataframe. In fact, you can manually use the formula we used earlier to compute the same result. The following example computes the z-score mathematically.

```
In [4]: vw = np.array(views['views'])
   ...: (vw[0] - np.mean(vw)) / np.std(vw)
Out[4]: -0.30721413311687235
```

Min-Max Scaling

With min-max scaling, we can transform and scale our feature values such that each value is within the range of [0, 1]. However the `MinMaxScaler` class in `scikit-learn` also allows you to specify your own upper and lower bound in the scaled value range using the `feature_range` variable. Mathematically we can represent this scaler as

$$MMS(X_i) = \frac{X_i - \min(X)}{\max(X) - \min(X)}$$

where we scale each value in the feature X by subtracting it from the minimum value in the feature min (X) and dividing the resultant by the difference between the maximum and minimum values in the feature max(X) - min (X). The following snippet helps us compute this.

```
In [5]: mms = MinMaxScaler()
   ...: views['minmax'] = mms.fit_transform(views[['views']])
```

```
   ...: views
Out[5]:
     views    zscore    minmax
0   1295.0 -0.307214  0.068109
1     25.0 -0.489306  0.001263
2  19000.0  2.231317  1.000000
3      5.0 -0.492173  0.000211
4      1.0 -0.492747  0.000000
5    300.0 -0.449877  0.015738
```

The preceding output shows the min-max scaled values in the `minmax` column and as expected, the maximum viewed video in row index 2 has a value of 1, and the minimum viewed video in row index 4 has a value of 0. You can also compute this mathematically using the following code (sample computation for the first row).

```
In [6]: (vw[0] - np.min(vw)) / (np.max(vw) - np.min(vw))
Out[6]: 0.068108847834096528
```

Robust Scaling

The disadvantage of min-max scaling is that often the presence of outliers affects the scaled values for any feature. Robust scaling tries to use specific statistical measures to scale features without being affected by outliers. Mathematically this scaler can be represented as

$$RS(X_i) = \frac{X_i - median(X)}{IQR_{(1,3)}(X)}$$

where we scale each value of feature X by subtracting the median of X and dividing the resultant by the IQR also known as the Inter-Quartile Range of X which is the range (difference) between the first quartile (25th %ile) and the third quartile (75th %ile). The following code performs robust scaling on our sample feature.

```
In [7]: rs = RobustScaler()
   ...: views['robust'] = rs.fit_transform(views[['views']])
   ...: views
Out[7]:
     views    zscore    minmax     robust
0   1295.0 -0.307214  0.068109   1.092883
1     25.0 -0.489306  0.001263  -0.132690
2  19000.0  2.231317  1.000000  18.178528
3      5.0 -0.492173  0.000211  -0.151990
4      1.0 -0.492747  0.000000  -0.155850
5    300.0 -0.449877  0.015738   0.132690
```

The scaled values are depicted in the `robust` column and you can compare them with the scaled features in the other columns. You can also compute the same using the mathematical equation we formulated for the robust scaler as depicted in the following snippet (for the first row index value).

```
In [8]: quartiles = np.percentile(vw, (25., 75.))
```

```
   ...: iqr = quartiles[1] - quartiles[0]
   ...: (vw[0] - np.median(vw)) / iqr
Out[8]: 1.0928829915560916
```

There are several other techniques for feature scaling and normalization, but these should be sufficient to get you started and are used extensively in building Machine Learning systems. Always remember to check if you need to scale and standardize features whenever you are dealing with numerical features.

Feature Selection

While it is good to try to engineering features that try to capture some latent representations and patterns in the underlying data, it is not always a good thing to deal with feature sets having maybe thousands of features or even more. Dealing with a large number of features bring us to the concept of the curse of dimensionality which we mentioned earlier during the "Bin Counting" section in "Feature Engineering on Categorical Data". More features tend to make models more complex and difficult to interpret. Besides this, it can often lead to models over-fitting on the training data. This basically leads to a very specialized model tuned only to the data which it used for training and hence even if you get a high model performance, it will end up performing very poorly on new, previously unseen data. The ultimate objective is to select an optimal number of features to train and build models that generalize very well on the data and prevent overfitting.

Feature selection strategies can be divided into three main areas based on the type of strategy and techniques employed for the same. They are described briefly as follows.

- **Filter methods**: These techniques select features purely based on metrics like correlation, mutual information and so on. These methods do not depend on results obtained from any model and usually check the relationship of each feature with the response variable to be predicted. Popular methods include threshold based methods and statistical tests.
- **Wrapper methods**: These techniques try to capture interaction between multiple features by using a recursive approach to build multiple models using feature subsets and select the best subset of features giving us the best performing model. Methods like backward selecting and forward elimination are popular wrapper based methods.
- **Embedded methods**: These techniques try to combine the benefits of the other two methods by leveraging Machine Learning models themselves to rank and score feature variables based on their importance. Tree based methods like decision trees and ensemble methods like random forests are popular examples of embedded methods.

The benefits of feature selection include better performing models, less overfitting, more generalized models, less time for computations and model training, and to get a good insight into understanding the importance of various features in your data. In this section, we look at some of the most widely used techniques in feature selection. You can load `feature_selection.py` directly and start running the examples or use the jupyter notebook, `Feature Selection.ipynb` for a more interactive experience. Let's start by loading the following dependencies and configurations.

```
In [1]: import numpy as np
   ...: import pandas as pd
   ...: np.set_printoptions(suppress=True)
   ...: pt = np.get_printoptions()['threshold']
```

We will now look at various ways of selecting features including statistical and model based techniques by using some sample datasets.

Threshold-Based Methods

This is a filter based feature selection strategy, where you can use some form of cut-off or thresholding for limiting the total number of features during feature selection. Thresholds can be of various forms. Some of them can be used during the feature engineering process itself, where you can specify threshold parameters. A simple example of this would be to limit feature terms in the Bag of Words model, which we used for text based feature engineering earlier. The `scikit-learn` framework provides parameters like `min_df` and `max_df` which can be used to specify thresholds for ignoring terms which have document frequency above and below user specified thresholds. The following snippet depicts a way to do this.

```
In [2]: from sklearn.feature_extraction.text import CountVectorizer
   ...:
   ...: cv = CountVectorizer(min_df=0.1, max_df=0.85, max_features=2000)
   ...: cv
Out[2]:
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=0.85, max_features=2000, min_df=0.1,
        ngram_range=(1, 1), preprocessor=None, stop_words=None,
        strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
        tokenizer=None, vocabulary=None)
```

This basically builds a count vectorizer which ignores feature terms which occur in less than 10% of the total corpus and also ignores terms which occur in more than 85% of the total corpus. Besides this we also put a hard limit of 2000 maximum features in the feature set.

Another way of using thresholds is to use variance based thresholding where features having low variance (below a user-specified threshold) are removed. This signifies that we want to remove features that have values that are more or less constant across all the observations in our datasets. We can apply this to our Pokémon dataset, which we used earlier in this chapter. First we convert the `Generation` feature to a categorical feature as follows.

```
In [3]: df = pd.read_csv('datasets/Pokemon.csv')
   ...: poke_gen = pd.get_dummies(df['Generation'])
   ...: poke_gen.head()
Out[3]:
   Gen 1  Gen 2  Gen 3  Gen 4  Gen 5  Gen 6
0      1      0      0      0      0      0
1      1      0      0      0      0      0
2      1      0      0      0      0      0
3      1      0      0      0      0      0
4      1      0      0      0      0      0
```

Next, we want to remove features from the one hot encoded features where the variance is less than `0.15`. We can do this using the following snippet.

```
In [4]: from sklearn.feature_selection import VarianceThreshold
   ...:
   ...: vt = VarianceThreshold(threshold=.15)
   ...: vt.fit(poke_gen)
Out[4]: VarianceThreshold(threshold=0.15)
```

To view the variances as well as which features were finally selected by this algorithm, we can use the `variances_` property and the `get_support(...)` function respectively. The following snippet depicts this clearly in a formatted dataframe.

```
In [5]: pd.DataFrame({'variance': vt.variances_,
   ...:               'select_feature': vt.get_support()},
   ...:              index=poke_gen.columns).T
Out[5]:
                  Gen 1     Gen 2 Gen 3     Gen 4     Gen 5      Gen 6
select_feature     True     False  True     False      True      False
variance       0.164444  0.114944  0.16  0.128373  0.163711  0.0919937
```

We can clearly see which features have been selected based on their `True` values and also their variance being above `0.15`. To get the final subset of selected features, you can use the following code.

```
In [6]: poke_gen_subset = poke_gen.iloc[:,vt.get_support()].head()
   ...: poke_gen_subset
Out[6]:
   Gen 1  Gen 3  Gen 5
0      1      0      0
1      1      0      0
2      1      0      0
3      1      0      0
4      1      0      0
```

The preceding feature subset depicts that features `Gen 1`, `Gen 3`, and `Gen 5` have been finally selected out of the original six features.

Statistical Methods

Another widely used filter based feature selection method, which is slightly more sophisticated, is to select features based on univariate statistical tests. You can use several statistical tests for regression and classification based models including mutual information, ANOVA (analysis of variance) and chi-square tests. Based on scores obtained from these statistical tests, you can select the best features on the basis of their score. Let's load a sample dataset now with 30 features. This dataset is known as the Wisconsin Diagnostic Breast Cancer dataset, which is also available in its native or raw format at https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic), which is the UCI Machine Learning repository. We will use `scikit-learn` to load the data features and the response class variable.

```
In [7]: from sklearn.datasets import load_breast_cancer
   ...:
   ...: bc_data = load_breast_cancer()
   ...: bc_features = pd.DataFrame(bc_data.data, columns=bc_data.feature_names)
   ...: bc_classes = pd.DataFrame(bc_data.target, columns=['IsMalignant'])
   ...:
```

```
   ...: # build featureset and response class labels
   ...: bc_X = np.array(bc_features)
   ...: bc_y = np.array(bc_classes).T[0]
   ...: print('Feature set shape:', bc_X.shape)
   ...: print('Response class shape:', bc_y.shape)
Feature set shape: (569, 30)
Response class shape: (569,)
```

We can clearly see that, as we mentioned before, there are a total of 30 features in this dataset and a total of 569 rows of observations. To get some more detail into the feature names and take a peek at the data points, you can use the following code.

```
In [8]: np.set_printoptions(threshold=30)
   ...: print('Feature set data [shape: '+str(bc_X.shape)+']')
   ...: print(np.round(bc_X, 2), '\n')
   ...: print('Feature names:')
   ...: print(np.array(bc_features.columns), '\n')
   ...: print('Response Class label data [shape: '+str(bc_y.shape)+']')
   ...: print(bc_y, '\n')
   ...: print('Response variable name:', np.array(bc_classes.columns))
   ...: np.set_printoptions(threshold=pt)
Feature set data [shape: (569, 30)]
[[  17.99   10.38  122.8  ...,    0.27    0.46    0.12]
 [  20.57   17.77  132.9  ...,    0.19    0.28    0.09]
 [  19.69   21.25  130.   ...,    0.24    0.36    0.09]
 ...,
 [  16.6    28.08  108.3  ...,    0.14    0.22    0.08]
 [  20.6    29.33  140.1  ...,    0.26    0.41    0.12]
 [   7.76   24.54   47.92 ...,    0.      0.29    0.07]]

Feature names:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']

Response Class label data [shape: (569,)]
[0 0 0 ..., 0 0 1]

Response variable name: ['IsMalignant']
```

This gives us a better perspective on the data we are dealing with. The response class variable is a binary class where 1 indicates the tumor detected was benign and 0 indicates it was malignant. We can also see the 30 features that are real valued numbers that describe characteristics of cell nuclei present in digitized images of breast mass. Let's now use the chi-square test on this feature set and select the top 15 best features out of the 30 features. The following snippet helps us achieve this.

```
In [9]: from sklearn.feature_selection import chi2, SelectKBest
   ...:
   ...: skb = SelectKBest(score_func=chi2, k=15)
   ...: skb.fit(bc_X, bc_y)
Out[9]: SelectKBest(k=15, score_func=<function chi2 at 0x0000018C2BEB7840>)
```

You can see that we have passed our input features (bc_X) and corresponding response class outputs (bc_y) to the fit(...) function when computing the necessary metrics. The chi-square test will compute statistics between each feature and the class variable (univariate tests). Selecting the top K features is more than likely to remove features having a low score and consequently they are most likely to be independent of the class variable and hence not useful in building models. We sort the scores to see the most relevant features using the following code.

```
In [10]: feature_scores = [(item, score) for item, score in zip(bc_data.feature_names,
                                                                skb.scores_)]
    ...: sorted(feature_scores, key=lambda x: -x[1])[:10]
Out[10]:
[('worst area', 112598.43156405364),
 ('mean area', 53991.655923750892),
 ('area error', 8758.5047053344697),
 ('worst perimeter', 3665.0354163405909),
 ('mean perimeter', 2011.1028637679051),
 ('worst radius', 491.68915743332195),
 ('mean radius', 266.10491719517802),
 ('perimeter error', 250.57189635982184),
 ('worst texture', 174.44939960571074),
 ('mean texture', 93.897508098633352)]
```

We can now create a subset of the 15 selected features obtained from our original feature set of 30 features with the help of the chi-square test by using the following code.

```
In [11]: select_features_kbest = skb.get_support()
    ...: feature_names_kbest = bc_data.feature_names[select_features_kbest]
    ...: feature_subset_df = bc_features[feature_names_kbest]
    ...: bc_SX = np.array(feature_subset_df)
    ...: print(bc_SX.shape)
    ...: print(feature_names_kbest)
(569, 15)
['mean radius' 'mean texture' 'mean perimeter' 'mean area' 'mean concavity'
 'radius error' 'perimeter error' 'area error' 'worst radius'
 'worst texture' 'worst perimeter' 'worst area' 'worst compactness'
 'worst concavity' 'worst concave points']
```

Thus from the preceding output, you can see that our new feature subset bc_SX has 569 observations of 15 features instead of 30 and we also printed the names of the selected features for your ease of understanding. To view the new feature set, you can use the following snippet.

```
In [12]: np.round(feature_subset_df.iloc[20:25], 2)
```

Out[12]:

mean radius	mean texture	mean perimeter	mean area	mean concavity	radius error	perimeter error	area error	worst radius	worst texture	worst perimeter	worst area	worst compactness	worst concavity	worst concave points
13.08	15.71	85.63	520.0	0.05	0.19	1.38	14.67	14.50	20.49	96.09	630.5	0.28	0.19	0.07
9.50	12.44	60.34	273.9	0.03	0.28	1.91	15.70	10.23	15.66	65.13	314.9	0.11	0.09	0.06
15.34	14.26	102.50	704.4	0.21	0.44	3.38	44.91	18.07	19.08	125.10	980.9	0.60	0.63	0.24
21.16	23.04	137.20	1404.0	0.11	0.69	4.30	93.99	29.17	35.59	188.00	2615.0	0.26	0.32	0.20
16.65	21.38	110.00	904.6	0.15	0.81	5.46	102.60	26.46	31.56	177.00	2215.0	0.36	0.47	0.21

Figure 4-39. *Selected feature subset of the Wisconsin Diagnostic Breast Cancer dataset using chi-square tests*

The dataframe with the top scoring features is depicted in Figure 4-39. Let's now build a simple classification model using logistic regression on the original feature set of 30 features and compare the model accuracy performance with another model built using our selected 15 features. For model evaluation, we will use the accuracy metric (percent of correct predictions) and use a five-fold cross-validation scheme. We will be covering model evaluation and tuning strategies in detail in Chapter 5, so do not despair if you cannot understand some of the terminology right now. The main idea here is to compare the model prediction performance between models trained on different feature sets.

```
In [13]: from sklearn.linear_model import LogisticRegression
    ...: from sklearn.model_selection import cross_val_score
    ...:
    ...: # build logistic regression model
    ...: lr = LogisticRegression()
    ...:
    ...: # evaluating accuracy for model built on full featureset
    ...: full_feat_acc = np.average(cross_val_score(lr, bc_X, bc_y, scoring='accuracy', cv=5))
    ...: # evaluating accuracy for model built on selected featureset
    ...: sel_feat_acc = np.average(cross_val_score(lr, bc_SX, bc_y, scoring='accuracy', cv=5))
    ...:
    ...: print('Model accuracy statistics with 5-fold cross validation')
    ...: print('Model accuracy with complete feature set', bc_X.shape, ':', full_feat_acc)
    ...: print('Model accuracy with selected feature set', bc_SX.shape, ':', sel_feat_acc)
Model accuracy statistics with 5-fold cross validation
Model accuracy with complete feature set (569, 30) : 0.950904193921
Model accuracy with selected feature set (569, 15) : 0.952643324356
```

The accuracy metrics clearly show us that we actually built a better model having accuracy of 95.26% when trained on the selected 15 feature subset as compared to the model built with the original 30 features which had an accuracy of 95.09%. Try this out on your own datasets! Do you see any improvements?

Recursive Feature Elimination

You can also rank and score features with the help of a Machine Learning based model estimator such that you recursively keep eliminating lower scored features till you arrive at the specific feature subset count. Recursive Feature Elimination, also known as RFE, is a popular wrapper based feature selection technique, which allows you to use this strategy. The basic idea is to start off with a specific Machine Learning estimator like the Logistic Regression algorithm we used for our classification needs. Next we take the entire feature set of 30 features and the corresponding response class variables. RFE aims to assign weights to these features based on the model fit. Features with the smallest weights are pruned out and then a model is fit again on

the remaining features to obtain the new weights or scores. This process is recursively carried out multiple times and each time features with the lowest scores/weights are eliminated, until the pruned feature subset contains the desired number of features that the user wanted to select (this is taken as an input parameter at the start). This strategy is also popularly known as backward elimination. Let's select the top 15 features on our breast cancer dataset now using RFE.

```
In [14]: from sklearn.feature_selection import RFE
    ...:
    ...: lr = LogisticRegression()
    ...: rfe = RFE(estimator=lr, n_features_to_select=15, step=1)
    ...: rfe.fit(bc_X, bc_y)
Out[14]:
RFE(estimator=LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False),
  n_features_to_select=15, step=1, verbose=0)
```

We can now use the `get_support(...)` function to obtain the final 15 selected features. This is depicted in the following snippet.

```
In [15]: select_features_rfe = rfe.get_support()
    ...: feature_names_rfe = bc_data.feature_names[select_features_rfe]
    ...: print(feature_names_rfe)
['mean radius' 'mean texture' 'mean perimeter' 'mean smoothness'
 'mean concavity' 'mean concave points' 'mean symmetry' 'texture error'
 'worst radius' 'worst texture' 'worst smoothness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

Can we compare this feature subset with the one we obtained using statistical tests in the previous section and see which features are common among both these subsets? Of course we can! Let's use set operations to get the list of features that were selected by both these techniques.

```
In [16]: set(feature_names_kbest) & set(feature_names_rfe)
Out[16]:
{'mean concavity', 'mean perimeter', 'mean radius', 'mean texture',
 'worst concave points', 'worst concavity', 'worst radius', 'worst texture'}
```

Thus we can see that 8 out of 15 features are common and have been chosen by both the feature selection techniques, which is definitely interesting!

Model-Based Selection

Tree based models like decision trees and ensemble models like random forests (ensemble of trees) can be utilized not just for modeling alone but for feature selection. These models can be used to compute feature importances when building the model that can in turn be used for selecting the best features and discarding irrelevant features with lower scores. Random forest is an ensemble model. This can be used as an embedded feature selection method, where each decision tree model in the ensemble is built by taking a training sample of data from the entire dataset. This sample is a bootstrap sample (sample taken with replacement). Splits at any node are taken by choosing the best split from a random subset of the features rather than taking all the features into account. This randomness tends to reduce the variance of the model

at the cost of slightly increasing the bias. Overall this produces a better and more generalized model. We will cover the bias-variance tradeoff in more detail in Chapter 5. Let's now use the random forest model to score and rank features based on their importance.

```
In [17]: from sklearn.ensemble import RandomForestClassifier
    ...:
    ...: rfc = RandomForestClassifier()
    ...: rfc.fit(bc_X, bc_y)
Out[17]:
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
            verbose=0, warm_start=False)
```

The following code uses this random forest estimator to score the features based on their importance and we display the top 10 most important features based on this score.

```
In [18]: importance_scores = rfc.feature_importances_
    ...: feature_importances = [(feature, score) for feature, score in zip(bc_data.feature_
names, importance_scores)]
    ...: sorted(feature_importances, key=lambda x: -x[1])[:10]
Out[18]:
[('worst area', 0.25116985146898885),
 ('worst radius', 0.16995187376059454),
 ('worst concavity', 0.1164662504282163),
 ('worst concave points', 0.11253251729478526),
 ('mean concave points', 0.10839170432994949),
 ('mean concavity', 0.063554137255925847),
 ('mean area', 0.023771318604377804),
 ('worst perimeter', 0.020636790800076958),
 ('worst texture', 0.019171556030722112),
 ('mean radius', 0.014908508522792335)]
```

You can now use a threshold based parameter to filter out the top *n* features as needed or you can even make use of the `SelectFromModel` meta-transformer provided by `scikit-learn` by using it as a wrapper on top of this model. Can you find out how many of the higher ranked features from the random forest model are in common with the previous two feature selectors?

Dimensionality Reduction

the difficulties a machine learning algorithm faces when working with data in the higher dimensions, that did not exist in the lower dimensions.

Dealing with a lot of features can lead to issues like model overfitting, complex models, and many more that all roll up to what we have mentioned as the curse of dimensionality. Refer to the section "Dimensionality Reduction" in Chapter 1 to refresh your memory. Dimensionality reduction is the process of reducing the total number of features in our feature set using strategies like feature selection or feature extraction. We have already talked about feature selection extensively in the previous section. We now cover feature extraction where the basic objective is to extract new features from the existing set of features such that the higher-dimensional dataset with many features can be reduced into a lower-dimensional dataset of these newly created features. A very popular technique of linear data transformation from higher to lower dimensions is Principal Component Analysis, also known as PCA. Let's try to understand more about PCA and how we can use it for feature extraction in the following sections.

Feature Extraction with Principal Component Analysis

Principal component analysis, popularly known as PCA, is a statistical method that uses the process of linear, orthogonal transformation to transform a higher-dimensional set of features that could be possibly correlated into a lower-dimensional set of linearly uncorrelated features. These transformed and newly created features are also known as Principal Components or PCs. In any PCA transformation, the total number of PCs is always less than or equal to the initial number of features. The first principal component tries to capture the maximum variance of the original set of features. Each of the succeeding components tries to capture more of the variance such that they are orthogonal to the preceding components. An important point to remember is that PCA is sensitive to feature scaling.

Our main task is to take a set of initial features with dimension let's say D and reduce it to a subset of extracted principal components of a lower dimension LD. The matrix decomposition process of Singular Value Decomposition is extremely useful in helping us obtain the principal components. You can quickly refresh your memory on SVD by referring to the sub-section of "Singular Value Decomposition" under the "Important Concepts" in the "Mathematics" section in Chapter 1 to check out the necessary mathematical formula and concepts. Considering we have a data matrix $F_{(n \times D)}$, where we have n observations and D dimensions (features), we can depict SVD of the feature matrix as $(F_{(n \times D)}) = USV^T$ such that all the principal components are contained in the component V^T, which can be depicted as follows:

$$V^T_{(D \times D)} = \begin{bmatrix} PC_{1(1 \times D)} \\ PC_{2(1 \times D)} \\ \cdots \\ PC_{D(1 \times D)} \end{bmatrix}$$

The principal components are represented by $\{PC_1, PC_2, \ldots PC_D\}$, which are all one-dimensional vectors of dimensions (1 x D). For extracting the first d principal components, we can first transpose this matrix to obtain the following representation.

$$PC_{(D \times D)} = \left(V^T\right)^T = \left[PC_{1(D \times 1)} \middle| PC_{2(D \times 1)} \middle| \ldots \middle| PC_{D(D \times 1)} \right]$$

Now we can extract out the first d principal components such that $d \leq D$ and the reduced principal component set can be depicted as follows.

$$PC_{(D \times D)} = \left(V^T\right)^T = \left[PC_{1(D \times 1)} \middle| PC_{2(D \times 1)} \middle| \ldots \middle| PC_{D(D \times 1)} \right]$$

d x d

Finally, to perform dimensionality reduction, we can get the reduced feature set using the following mathematical transformation $F_{(n \times d)} = F_{(n \times D)} \cdot PC_{(D \times d)}$ where the dot product between the original feature matrix and the reduced subset of principal components gives us a reduced feature set of d features. A very important point to remember here is that you might need to center your initial feature matrix by removing the mean because by default, PCA assumes that your data is centered around the origin.

Let's try to extract the first three principal components now from our breast cancer feature set of 30 features using SVD. We first center our feature matrix and then use SVD and subsetting to extract the first three PCs using the following code.

```
In [19]: # center the feature set
    ...: bc_XC = bc_X - bc_X.mean(axis=0)
    ...:
    ...: # decompose using SVD
    ...: U, S, VT = np.linalg.svd(bc_XC)
    ...:
    ...: # get principal components
    ...: PC = VT.T
    ...:
    ...: # get first 3 principal components
    ...: PC3 = PC[:, 0:3]
    ...: PC3.shape
Out[19]: (30, 3)
```

We can now get the reduced feature set of three features by using the dot product operation we discussed earlier. The following snippet gives us the final reduced feature set that can be used for modeling.

```
# reduce feature set dimensionality
np.round(bc_XC.dot(PC3), 2)
Out[20]:
array([[-1160.14,  -293.92,   -48.58],
       [-1269.12,    15.63,    35.39],
       [ -995.79,    39.16,     1.71],
       ...,
       [ -314.5 ,    47.55,    10.44],
       [-1124.86,    34.13,    19.74],
       [  771.53,   -88.64,   -23.89]])
```

Thus you can see how powerful SVD and PCA can be in helping us reduce dimensionality by extracting necessary features. Of course in Machine Learning systems and pipelines you can use utilities from `scikit-learn` instead of writing unnecessary code and equations. The following code enables us to perform PCA on our breast cancer feature set leveraging `scikit-learn`'s APIs.

```
In [21]: from sklearn.decomposition import PCA
    ...: pca = PCA(n_components=3)
    ...: pca.fit(bc_X)
Out[21]:
PCA(copy=True, iterated_power='auto', n_components=3, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

To understand how much of the variance is explained by each of these principal components, you can use the following code.

```
In [22]: pca.explained_variance_ratio_
Out[22]: array([ 0.98204467,  0.01617649,  0.00155751])
```

From the preceding output, as expected, we can see the maximum variance is explained by the first principal component. To obtain the reduced feature set, we can use the following snippet.

```
In [23]: bc_pca = pca.transform(bc_X)
    ...: np.round(bc_pca, 2)
Out[23]:
array([[ 1160.14,  -293.92,    48.58],
       [ 1269.12,    15.63,   -35.39],
       [  995.79,    39.16,    -1.71],
       ...,
       [  314.5 ,    47.55,   -10.44],
       [ 1124.86,    34.13,   -19.74],
       [ -771.53,   -88.64,    23.89]])
```

If you compare the values of this reduced feature set with the values obtained in our mathematical implementation based code, you will see they are exactly the same except sign inversions in some cases.

The reason for sign inversion in some of the values in principal components is because the direction of these principal components is unstable. The sign indicates direction. Hence even if the principal components point in opposite directions, they should still be on the same plane and hence shouldn't have an effect when modeling with this data.

Let's now quickly build a logistic regression model as before and use model accuracy and five-fold cross validation to evaluate the model quality using these three features.

```
In [24]: np.average(cross_val_score(lr, bc_pca, bc_y, scoring='accuracy', cv=5))
Out[24]: 0.92808003078106949
```

We can see from the preceding output that even though we used only three features derived from the principal components instead of the original 30 features, we still obtained a model accuracy close to 93%, which is quite decent!

Summary

This was a content packed chapter with a lot of hands-on examples based on real-world datasets. The main intent of this chapter is to get you familiarized with essential concepts, tools, techniques, and strategies used for feature extraction, engineering, scaling, and selection. One of the toughest tasks that data scientists face day in and day out is data processing and feature engineering. Hence it is of paramount importance that you understand the various aspects involved with deriving features from raw data. This chapter is intended to be used both as a starting ground as well as a reference guide for understanding what techniques and strategy should be applied when trying to engineer features on your own datasets. We cover the basic concepts of feature engineering, scaling, and selection and also the importance behind each of these processes. Feature engineering techniques are covered extensively for diverse data types including numerical, categorical, text,

temporal and images. Multiple feature scaling techniques are also covered, which are useful to tone down the scale and magnitude of features before modeling. Finally, we cover feature selection techniques in detail with emphasis on the three different strategies of feature selection namely filter, wrapper, and embedded methods. Special sections on dimensionality reduction and automated feature extraction using Deep Learning have also been included since they have gained a lot of prominence in both research as well as the industry. I want to conclude this chapter by leaving you with the following quote by Peter Norvig, renowned computer scientist and director at Google, which should reinforce the importance of feature engineering.

> *"More data beats clever algorithms, but better data beats more data."*
>
> —Peter Norvig