17

AUTOMATED FEATURE ENGINEERING

OVERVIEW

In this chapter, you will be dealing with automated feature engineering using feature tools and analyzing datasets and building business context; creating entities from datasets and mapping the relationships between base datasets and entities; and building classification models based on automated feature engineering. By the end of this chapter, you will be able to use these automated feature engineering techniques.

INTRODUCTION

In the previous chapter, we learned about a utility function called the ML pipeline, which automates various processes, such as scaling, dimensionality reduction, and modeling, within the data science life cycle.

In this chapter, we will learn about another utility that helps in automating feature engineering. We have completed different feature engineering tasks in the previous chapters, such as in *Chapter 3, Binary Classification*, and *Chapter 12, Feature Engineering*. When building features in the previous exercises, you would have realized how tedious this step is when it's done manually.

For instance, in *Chapter 3, Binary Classification*, in *Exercise 3.02*, you implemented different aggregation functions using the traditional manual ways to create a new feature, as shown in the following code snippet:

```
# Aggregation on age
ageTot = bankData.groupby('age')['y'].agg(ageTot='count').reset_index()
ageTot.head()
```

	age	ageTot
0	18	12
1	19	35
2	20	50
3	21	79
4	22	129

Figure 17.1: Different aggregation functions being used

As you may recall, in this operation, you grouped the data frame by the **age** variable and then did an aggregation to find the total count of all the examples in each age group.

Now, imagine you had to perform such operations on hundreds of variables. This would have been mind-boggling. However, with the tools that you will be learning about in this chapter, doing everything manually can be eliminated in favor of automated feature creation.

In addition to alleviating the pain of creating new features manually, there is the difficult task of identifying opportunities for new features.

Would you have thought of a feature such as (duration – balance/duration – previous), which is a ratio of the difference between a variable's duration and balance compared to the difference of duration and previous value? You are probably wishing for a magic wand to alleviate the pain of identifying opportunities for new features such as these. Well, your wish will be coming true in this chapter as we will introduce feature tools that will create features for us.

Let's dig deep to see it in action.

FEATURE ENGINEERING

Feature engineering is the process of creating or transforming features from raw data, as we mentioned in *Chapter 3, Binary Classification*. These features get fed into various machine learning models to generate our desired business outcomes. Feature engineering is one of the most important steps in the data science life cycle and is even more important than the models themselves. The veracity of the models depends on what goes into the models, which are the features you build for the dataset.

Building the best set of features is dependent largely on domain understanding and the intuitions derived from the data during the exploratory data analysis phase. There is a lot of creativity involved in creating and transforming features, and therefore feature engineering can be considered both as an art and a science. However, performing feature engineering manually is quite an arduous and time-consuming process. This is where automated feature engineering plays a significant role in the data science life cycle.

AUTOMATING FEATURE ENGINEERING USING FEATURE TOOLS

Automated feature engineering entails creating features from raw data through a predefined data schema. These new features can be further distilled so that we can select the most suitable candidates for downstream modeling. The automated generation of features is enabled by a library in Python called Featuretools.

Let's look at the inner workings of this utility from the perspective of a business use case.

BUSINESS CONTEXT

The dataset we will be using to learn about different methods of automated feature engineering will be the bank marketing dataset.

> **NOTE**
>
> This dataset is from the UCI machine learning library and can be found at https://archive.ics.uci.edu/ml/machine-learning-databases/00222/.
> [Moro et al., 2014] S. Moro, P. Cortez and P. Rita. *A Data-Driven Approach to Predict the Success of Bank Telemarketing*. Decision Support Systems, Elsevier, 62:22-31, June 2014. UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science.

This dataset pertains to a marketing campaign for a bank. It contains details about various customers, such as the following:

- Demographic information: Age, job, marital status, education, and so on
- Financial details: Housing, loan, bank balance, and so on

The problem statement is to predict whether a particular customer would subscribe to a term deposit or not.

Starting any feature engineering process is done through the definition of the business context. This entails understanding various business factors that influence a problem statement. Let's define the various business factors that influence the problem statement by creating a domain story.

DOMAIN STORY FOR THE PROBLEM STATEMENT

The problem statement for us is to determine which customer is likely to buy a term deposit. Let's assume that, from market research and through discussions with domain experts, we have identified four critical factors that indicate the propensity for term deposits:

- Demographics of the customer
- Asset portfolio of the customer
- Liabilities of the customer
- Financial behavior of the customer

These business factors will drive the formulation of our feature engineering strategy.

After identifying the critical business factors, the next task is to identify those data points that are related to these factors.

From our dataset, the data points that can be mapped to these factors are as follows:

- Customer demographics: The various data points related to demographics are variables such as age, education, and marital status.
- Asset portfolio: An indicator of an asset portfolio in the dataset is whether the customer owns a house or not.
- Liability: The presence of loans for the customer is an indicator of liabilities.
- Financial behavior: A financial behavior is an indicator of whether a customer is credit-worthy or not. One indicator that is present in our dataset that indicates financial behavior is whether the customer has defaulted or not.

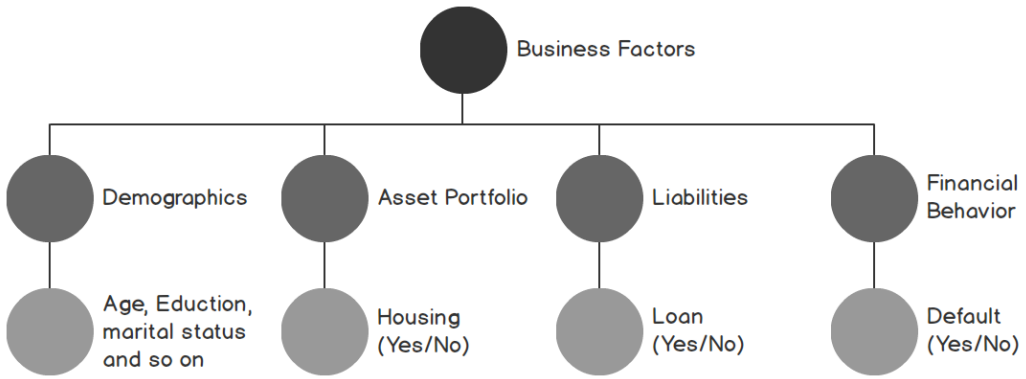

Figure 17.2: Business factors – data point mapping

The preceding diagram shows the mapping of various business factors to the relevant data points in the data and defines the domain story. Once the domain story has been defined, the next task is to implement this structure using feature tools. Let's discuss this next.

FEATURETOOLS – CREATING ENTITIES AND RELATIONSHIPS

Featuretools, which is a library in Python, allows the creation of a comprehensive list of features from the available dataset. A basic building block inside Featuretools is called **entities**. An entity can be thought of as an individual data table that aids in creating features. The first task that's involved when using Featuretools is to create these entities. This is where the domain story that we created previously becomes relevant. The data points related to the four business factors that we defined earlier defined would be the entities in our case, which are the demographics, assets, liability, and financial behavior.

Once the entities have been defined, the next task is to define a relationship so that we can connect the entities. In our context, we have a customer, and this customer will have an asset portfolio, a liability portfolio, and a certain financial behavior.

So, the type of relationship that we can define within our entities is one where the customer demographics act as a parent and the other entities have a child relationship with the entity demographics. The parent entity and its set of child entities, when put together, is called an **entity set**. This can be seen in the following diagram:

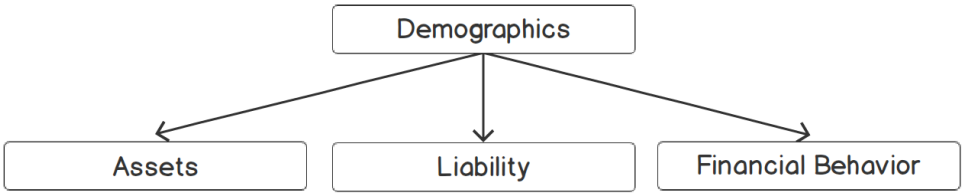

Figure 17.3: Entity set with its relationship

In the preceding diagram, we can see that Demographics is dependent on the Assets, Liability, and Financial Behavior entities.

Now, let's take a look at the dataset and all the variables of the dataset. Then, we'll map the variables to the different entities that we have.

The following screenshot shows the list of variables we have from the dataset and their subsequent mapping to the entities:

```
(['age', 'job', 'marital', 'education', 'default', 'balance', 'housing',
  'loan', 'contact', 'day', 'month', 'duration', 'campaign', 'pdays',
  'previous', 'poutcome', 'y'],
```

Figure 17.4: List of variables from the dataset

Have a look at the following table. The variables have been set alongside their mapped entity:

Variables	Mapped entity
Age, job, marital, education, contact, day, month, duration, campaign, pdays, previous, poutcome	Demographics
Housing	Asset
Loan	Liability
Default	Financial Behavior

Figure 17.5: Variables of the mapped entity

> **NOTE**
>
> Variable **`y`**, which is our target variable, will not be used for feature creation.

One prerequisite of defining relationships between entities is to have some IDs to connect the different entities within the relationship hierarchy. However, our dataset doesn't have any IDs. Therefore, we will have to create IDs for each entity. The creation of IDs will have to be based on the unique set of values each entity has.

The Demographics entity contains the customer information. Each row in the dataset is a unique customer. Therefore, for the Demographics entity, we will create an ID called **`CustID`**, which will be equal to the number of rows in the dataset.

The Asset entity is defined based on the **`housing`** variable. This variable has two values: **`Yes`** and **`No`**. We will map these values to a numerical ID. We'll map **`Yes`** to **`1`** and **`No`** to **`0`**. The variable name for this ID is **`AssetId`**.

Similarly, the Liability entity, which is mapped to the **`loan`** variable, and the Financial Behavior entity, which is mapped to the **`default`** variable, also have values of **`Yes`** and **`No`**. Their values are also mapped to **`1`** and **`0`**, respectively. The variables name for the respective IDs are **`LoanId`** and **`FinbehId`**.

The final data schema, which defines the entities and their relationships, is as follows:

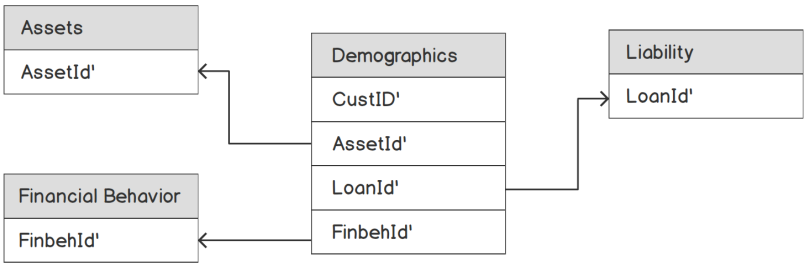

Figure 17.6: Entities relationship schema

Now we will implement these initial tasks using Featuretools in the following exercise.

EXERCISE 17.01: DEFINING ENTITIES AND ESTABLISHING RELATIONSHIPS

In this exercise, we will use the banking dataset. We will define the entities, create IDs, and then establish a relationship between the entities. These are the initial tasks we must complete before we create automated features in the subsequent exercises.

> **NOTE**
>
> The dataset file to be used in this exercise can be found in this book's GitHub repository at https://packt.live/2ZSHGxg.

The following steps will help you complete this exercise:

1. Open a Colab notebook.
2. Define the path of the GitHub repository:

```
 # Defining the path to the
file_url = 'https://raw.githubusercontent.com/PacktWorkshops/The-Data-Science-Workshop/master/Chapter17/Datasets/bank-full.csv'
```

3. Next, load the data using pandas:

```
# Loading data using pandas
import pandas as pd
bankData = pd.read_csv(file_url,sep=";")
bankData.head()
```

You should get the following output:

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	y
0	58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1	0	unknown	no
1	44	technician	single	secondary	no	29	yes	no	unknown	5	may	151	1	-1	0	unknown	no
2	33	entrepreneur	married	secondary	no	2	yes	yes	unknown	5	may	76	1	-1	0	unknown	no
3	47	blue-collar	married	unknown	no	1506	yes	no	unknown	5	may	92	1	-1	0	unknown	no
4	33	unknown	single	unknown	no	1	no	no	unknown	5	may	198	1	-1	0	unknown	no

Figure 17.7: Data loaded using pandas

4. Remove the target variable.

   Since the **`y`** target variable is not required for creating features, we will remove it using the **`.pop()`** function:

```
# Removing the target variable
Y = bankData.pop('y')
```

5. Create an ID for the Demographic entity, as shown in the following code snippet:

```
# Creating the Ids for Demographic Entity
bankData['custID'] = bankData.index.values
bankData['custID'] = 'cust' + bankData['custID'].astype(str)
```

   Since each row pertains to a unique customer, there will be as many customer IDs as there are rows in the dataset. We created the customer ID by taking the index value of each row and attaching a string called **`cust`** to the index values.

   The index values of the rows can be derived by the **`.index.values()`** method. These values are then attached to the **`cust`** string in the second line and the whole ID, which is a string value, is stored in the **`custID`** variable.

6. Create an ID for Assets, as shown in the following code snippet:

```
# Creating AssetId
bankData['AssetId'] = 0
bankData.loc[bankData.housing == 'yes','AssetId']= 1
```

   Here, we created the ID for Assets. As we mentioned previously, Assets is mapped to the **`housing`** variable, which has two values: **`Yes`** and **`No`**. In the first line of the preceding code, we initialized all the values of the **`AssetId`** variable to **`0`**. In the second line, we change the values of **`AssetId`** to **`1`** wherever the housing variable is **`yes`**.

7. Now, create an ID for **Loans**:

```
# Creating LoanId
bankData['LoanId'] = 0
bankData.loc[bankData.loan == 'yes','LoanId']= 1
```

Similar to the step for assets, we created a **LoanId** based on the value of the **loan** variable.

8. Create an ID for Financial Behavior, as shown in the following code snippet:

```
# Creating Financial behaviour ID
bankData['FinbehId'] = 0
bankData.loc[bankData.default == 'yes','FinbehId']= 1
```

Similarly, you create the ID for Financial Behavior based on the value of the **default** variable.

9. Display the data frame after adding the IDs:

```
# Displaying the new data frame after adding the ids
bankData.head()
```

You should get the following output:

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	custID	AssetId	LoanId	FinbehId
0	58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1	0	unknown	cust0	1	0	0
1	44	technician	single	secondary	no	29	yes	no	unknown	5	may	151	1	-1	0	unknown	cust1	1	0	0
2	33	entrepreneur	married	secondary	no	2	yes	yes	unknown	5	may	76	1	-1	0	unknown	cust2	1	1	0
3	47	blue-collar	married	unknown	no	1506	yes	no	unknown	5	may	92	1	-1	0	unknown	cust3	1	0	0
4	33	unknown	single	unknown	no	1	no	no	unknown	5	may	198	1	-1	0	unknown	cust4	0	0	0

Figure 17.8: Dataframe after adding the IDs

From the preceding output, you can see the four IDs that we created.

10. Import the necessary libraries for implementing Featuretools, as shown in the following code snippet:

```
# Importing necessary libraries
import featuretools as ft
import numpy as np
```

> **NOTE**
>
> Feature tools are implemented using the **featuretools** library.

11. Initialize **Entityset**:

```
# creating the entity set 'Bankentities'
Bankentities = ft.EntitySet(id = 'Bank')
```

Here, you initialize the entity set. This is implemented using the **.EntitySet()** method. We provide a string for tracking the entity set as an argument within the method. The string we have given here is **Bank**.

> **NOTE**
>
> The entity set is a combination of different entities, such as Demographics, Assets, Liability, and Financial Behavior.

12. Map the data frame to the entity set to create the parent entity:

```
# Mapping a dataframe to the entityset to form the parent entity
Bankentities.entity_from_dataframe(entity_id = 'Demographic Data'
, dataframe = bankData, index = 'custID')
```

Once the entity set has been initialized, we have to map the bank dataset to the entity set and then create the parent entity, that is, Demographic Data. The parent entity is tracked by **custID**. Mapping the data frame is done using the **.entity_from_dataframe()** method.

You should get the following output:

```
Entityset: Bank
  Entities:
    Demographic Data [Rows: 45211, Columns: 20]
  Relationships:
    No relationships
```

Figure 17.9: Output showing the mapping of the entity set to create the parent set

The various arguments within this method are **entity_id**, which is just a tracking name, then the **bankData** dataset, and most importantly the index, which is the **custID** we created.

From the output, we can see that the first entity, which is the parent entity, **Demographic Data**, has been created. So far, no relationships have been created for this entity, but we will be doing this in the steps ahead.

13. Define the Assets entity and set the relationship.

    Once the parent entity has been created, it is time to define each of the child entities and then create relationships between the parent entity, **Demographic Data**, and the child entities. This is done using the **.normalize_entity()** function. This is implemented as follows:

```
# Mapping Assets and setting the relationship
Bankentities.normalize_entity(base_entity_id='Demographic Data', new_
entity_id='Assets', index = 'AssetId',
additional_variables = ['housing'])
```

    You should get the following output:

```
Entityset: Bank
  Entities:
    Demographic Data [Rows: 45211, Columns: 19]
    Assets [Rows: 2, Columns: 2]
  Relationships:
    Demographic Data.AssetId -> Assets.AssetId
```

    Figure 17.10: Output showing the Assets entity and setting its relationship

    From the implementation, we can see that the various arguments for this method are **base_entity_id**, which is the parent entity, that is, **Demographic Data**; the new entity, which is **Assets**, and its ID, **AssetId**; and also the variable for Assets, which is the **housing** variable.

    From the output, we can see that two entities have been created and that a relationship has been formed between the parent entity and the child entity.

14. Now, map the loan and financial behavior entities.

    Similar to the Asset entity's creation, let's map the other two entities, that is, Loan and Financial Behavior. This is implemented as follows:

```
# Mapping Loans and Financial behavior entities
Bankentities.normalize_entity(base_entity_id='Demographic Data', new_
entity_id='Liability', index = 'LoanId',
additional_variables = ['loan'])


Bankentities.normalize_entity(base_entity_id='Demographic Data', new_
entity_id='FinBehaviour', index = 'FinbehId',
additional_variables = ['default'])
```

You should get the following output:

```
Entityset: Bank
  Entities:
    Demographic Data [Rows: 45211, Columns: 17]
    Assets [Rows: 2, Columns: 2]
    Liability [Rows: 2, Columns: 2]
    FinBehaviour [Rows: 2, Columns: 2]
  Relationships:
    Demographic Data.AssetId -> Assets.AssetId
    Demographic Data.LoanId -> Liability.LoanId
    Demographic Data.FinbehId -> FinBehaviour.FinbehId
```

Figure 17.11: Mapping the Loan and Financial Behavior entities

From the new output, we can see that all the entities have been formed and that their relationships have also been defined.

In this exercise, we were able to set the stage for automated feature engineering using Featuretools. We implemented the domain story by defining the entities and mapping the relationship of different entities with the parent entity.

Now that the entities have been created and the relationships have been set, it is time to commence the feature engineering process.

FEATURE ENGINEERING – BASIC OPERATIONS

Creating new features from raw variables can be summarized into two basic operations:

- Aggregation
- Transformation

An aggregation operation entails changing the value of a variable based on the value of another variable. Let's carry out an aggregation operation on the bank dataset. First, we'll visualize the head of the dataset first:

```
bankData.head()
```

You should get the following output:

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	y
0	58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1	0	unknown	no
1	44	technician	single	secondary	no	29	yes	no	unknown	5	may	151	1	-1	0	unknown	no
2	33	entrepreneur	married	secondary	no	2	yes	yes	unknown	5	may	76	1	-1	0	unknown	no
3	47	blue-collar	married	unknown	no	1506	yes	no	unknown	5	may	92	1	-1	0	unknown	no
4	33	unknown	single	unknown	no	1	no	no	unknown	5	may	198	1	-1	0	unknown	no

Figure 17.12: Data being displayed

Suppose we want to find the mean value of **`balance`** based on the value of housing. In other words, we want to find the mean of balance when housing is **`yes`** and when it is **`no`**. This is an aggregation operation where we are aggregating the value of a **`balance`** variable based on the value of another variable, which in this case is **`housing`**. This can be implemented as follows:

```
# Aggregating based on housing data
agg = bankData.groupby('housing')['balance'].agg('mean')
print(agg)
```

You will get the following output:

```
housing
no     1596.501270
yes    1175.103064
Name: balance, dtype: float64
```

Figure 17.13: Aggregated output

The first step is to aggregate the **`balance`** data based on the **`housing`** variable. In the preceding code snippet, the **`.groupby()`** function is to group the data based on the values of **`housing`** and the **`.agg()`** function is to find the **`mean`** of the **`balance`** data.

From the output, we can see that we have two values of mean corresponding to each value of **`housing`**. The mean of customers who do not own a house, which is denoted by **`no`**, is **`1596.5`** and the mean value for those who own a house, which is denoted by **`yes`**, is **`1175.1`**.

The next step is to merge this aggregation to the main data frame so that we get the aggregated values as part of the data frame. This is implemented as follows:

```
# Merging with the original data frame
bankNew = bankData.merge(agg,left_on = 'housing',right_index=True,how = 'left')
bankNew.head(10)
```

You will get the following output:

	age	job	marital	education	default	balance_x	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	balance_y
0	58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1	0	unknown	1175.103064
1	44	technician	single	secondary	no	29	yes	no	unknown	5	may	151	1	-1	0	unknown	1175.103064
2	33	entrepreneur	married	secondary	no	2	yes	yes	unknown	5	may	76	1	-1	0	unknown	1175.103064
3	47	blue-collar	married	unknown	no	1506	yes	no	unknown	5	may	92	1	-1	0	unknown	1175.103064
4	33	unknown	single	unknown	no	1	no	no	unknown	5	may	198	1	-1	0	unknown	1596.501270
5	35	management	married	tertiary	no	231	yes	no	unknown	5	may	139	1	-1	0	unknown	1175.103064
6	28	management	single	tertiary	no	447	yes	yes	unknown	5	may	217	1	-1	0	unknown	1175.103064
7	42	entrepreneur	divorced	tertiary	yes	2	yes	no	unknown	5	may	380	1	-1	0	unknown	1175.103064
8	58	retired	married	primary	no	121	yes	no	unknown	5	may	50	1	-1	0	unknown	1175.103064
9	43	technician	single	secondary	no	593	yes	no	unknown	5	may	55	1	-1	0	unknown	1175.103064

Figure 17.14: Merging the aggregation with the main dataset

The merging is done by the **`.merge()`** function, where **`bankData`** is merged with **`agg`** data. The **`left_on`** argument is used to ensure that the merge is based on the values of **`housing`**, while the **`how = left`** argument ensures that the resultant data frame is in the same form as the main data frame. This was implemented in *Chapter 12, Feature Engineering*. From the output, we can see that a new variable called **`balance_y`** is created, which displays the mean value corresponding to the value of housing.

These aggregation operations, when done manually, are not very complex; however, they involve multiple steps. Think of a scenario when we have to do some aggregations across 10 different variables or even 100 variables. It would be quite laborious to complete these tasks manually.

The second type of operation, which is the transformation operation, is much simpler than the aggregation operation. In transformation operation, we just effect a change on a variable based on some transformative functions. For example, we can introduce a new variable that's the natural logarithm of another variable or a square root of another variable. In such operations, the change is not dependent on the value of any other variable. The new variable is just a transformative function of the original variable.

To demonstrate this concept, let's do a transformation on the balance variable by taking a natural logarithm:

```
# Transformation operation
import numpy as np
bankData['Tranbalance'] = np.log(bankData['balance'])
bankData.head()
```

You should get the following output:

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	Tranbalance
0	58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1	0	unknown	7.669962
1	44	technician	single	secondary	no	29	yes	no	unknown	5	may	151	1	-1	0	unknown	3.367296
2	33	entrepreneur	married	secondary	no	2	yes	yes	unknown	5	may	76	1	-1	0	unknown	0.693147
3	47	blue-collar	married	unknown	no	1506	yes	no	unknown	5	may	92	1	-1	0	unknown	7.317212
4	33	unknown	single	unknown	no	1	no	no	unknown	5	may	198	1	-1	0	unknown	0.000000

Figure 17.15: Transformation operation performed on the dataset

In the transformation implementation, we took a logarithm of the **`balance`** variable using the **`np.log`** function and then saved it into a new variable called **`Tranbalance`**. You can see the transformed values in the last column. As we can see, a transformation operation is dependent on only one variable that is being transformed.

As shown in the preceding example, these operations are quite tedious when they're executed manually. The work that's involved and the time that's taken to do this increases many times when there are a number of variables involved.

This is where Featuretools adds value. In the next section, we will see how Featuretools can be used to enable automated feature extraction

FEATURETOOLS – AUTOMATED FEATURE ENGINEERING

In the previous section, we look at two basic operations for feature engineering. In the feature tools parlance, these operations are called **feature primitives**. Feature primitives are a list of aggregation and transformation possibilities that have been predefined in Featuretools.

The list of primitives that are available can be obtained by using the following command:

```
import featuretools as ft
ft.primitives.list_primitives()
```

You should get the following output:

	name	type	description
0	percent_true	aggregation	Finds the percent of 'True' values in a boolea...
1	min	aggregation	Finds the minimum non-null value of a numeric ...
2	std	aggregation	Finds the standard deviation of a numeric feat...
3	skew	aggregation	Computes the skewness of a data set.
4	trend	aggregation	Calculates the slope of the linear trend of va...
5	mean	aggregation	Computes the average value of a numeric feature.
6	median	aggregation	Finds the median value of any feature with wel...
7	time_since_last	aggregation	Time since last related instance.
8	any	aggregation	Test if any value is 'True'.
9	mode	aggregation	Finds the most common element in a categorical...
10	avg_time_between	aggregation	Computes the average time between consecutive ...
11	num_unique	aggregation	Returns the number of unique categorical varia...
12	all	aggregation	Test if all values are 'True'.
13	last	aggregation	Returns the last value.
14	max	aggregation	Finds the maximum non-null value of a numeric ...
15	n_most_common	aggregation	Finds the N most common elements in a categori...

Figure 17.16: List of available primitives

In the preceding output, you can see some of the aggregation names, such as **`min`**, **`std`**, **`skew`**, **`mean`**, and **`median`**. For instance, if you want the median from a column data of a dataset, you can use the median option from the aggregation type.

The Featuretools utility automates the process of generating new features through a method called **Deep Feature Synthesis** (**DFS**). Through this method, feature primitive steps such as aggregation and transformations are applied to the raw features to create new features. When implementing deep feature synthesis, there are some default primitives within the method, such as **`sum`**, **`standard deviation (std)`**, **`max`**, **`min`**, and **`skew`**.

In addition, we can also specify the list of primitives we want to implement from the available list of primitives. On top of that, we can also create new primitives by defining custom functions that can then be added to this list.

Now, we will look at how the Deep Feature Synthesis method is implemented to create new features.

EXERCISE 17.02: CREATING NEW FEATURES USING DEEP FEATURE SYNTHESIS

This exercise will build upon what we did in the previous exercise. In this exercise, we will create the entities and relationships and then apply the Deep Feature Synthesis (DFS) method to create new features.

> **NOTE**
>
> The dataset file to be used in this exercise can be found in this book's GitHub repository at https://packt.live/2ZSHGxg.

The following steps will help you complete this exercise:

1. Open a new Colab notebook.
2. Implement all the steps of *Exercise 17.01* until the creation of entities and relationships.
3. Create automated features using DFS.

   Once the entities have been created and the relationships have been defined, the features can be synthesized using the **`ft.dfs()`** function. This can be implemented as follows:

```
# Creating feature sets using Deep Feature Synthesis
feature_set, feature_names = ft.dfs(entityset=Bankentities,
target_entity = 'Demographic Data',
max_depth = 2,
verbose = 1,
n_jobs = 1)
```

You should get the following output:

```
Built 196 features
Elapsed: 00:13 | Remaining: 00:00 | Progress: 100%|██████████| Calculated: 11/11 chunks
```

Figure 17.17: Output showing the automated features

The first two arguments to the function are **`entityset`**, which we created first, and **`target_entity`**, which is the parent entity. The other important argument is **`max_depth`**, which defines the level of stacking of the created features.

In this example, we have set **`max_depth`** to **`2`**. Depending on the level of max_depth, the number of features will vary. It has to be noted that with depth **`1`**, no aggregation primitives will be applied. With depth **`1`**, only transformation primitives will be applied. This concept will be clearer once we see the output of the automated feature engineering.

The verbose argument is used to specify whether we want a notification for the process. A value of **`1`** means we need notification.

The **`n_jobs`** argument is used for specifying whether we want to parallelize the jobs across multiple cores of the system. This is applicable when we have a multi-core machine. For Colab, we have to keep this argument as **`1`**.

There are two outputs from the DFS implementation: feature set and feature names. The first one is the modified data frame after the creation of all the additional features. From the output, we can see that a total of **`196`** features are created by the utility. The second output is the names of all these features.

4. Reset the index of the features.

   Once the feature sets have been created, the indexing will be all jumbled up. We need to reindex them so that the index is similar to the original dataset. This is implemented as follows:

```
# Reindexing the feature_set
feature_set = feature_set.reindex(index=bankData['custID'])
feature_set = feature_set.reset_index()
```

In the first line, reindexing is done using the **`.reindex()`** function. As the argument, we give the target index, based on which the reindexing has to be done. In the argument, we specify that the indexing has to be done based on the order of **`custID`**. Once this line is implemented, the index of the data frame becomes **`cust01`**,**`cust02`** ..., and so on. The second line, **`.reset_index()`**, is used to change this index to **`0`**,**`1`**,**`2`**, and so on.

5. Verify the shape of the new data frame and compare it with the old data frame.

   Let's print the shapes of the new data frame and the old one so that we can compare the shapes:

```
# Verifying the shape of the features and original bank data
print(feature_set.shape)
print(bankData.shape)
```

   You should get the following output:

```
(45211, 197)
(45211, 20)
```

   From the output, we can see that the new data frame has **197** features. After the DFS process, 196 features were generated. The remaining feature is the **CustID**, which makes the total number of features **197**.

6. Print the head of the new data frame:

```
# Printing head of the feature set
feature_set.head()
```

   You should get the following output:

Id	LoanId	FinbehId	Assets.housing	Liability.loan	FinBehaviour.default	Assets.SUM(Demographic Data.age)	Assets.SUM(Demographic Data.balance)	Assets.SUM(Demographic Data.day)	Assets.SUM(Demographic Data.duration)
1	0	0	yes	no	no	984475	29530340	391984	6517000
1	0	0	yes	no	no	984475	29530340	391984	6517000
1	1	0	yes	yes	no	984475	29530340	391984	6517000
1	0	0	yes	no	no	984475	29530340	391984	6517000
0	0	0	no	no	no	866292	32059342	322640	5154811

Figure 17.18: New DataFrame as the output

> **NOTE**
>
> Only a few of the features are being displayed in the preceding screenshot.

From the preceding screenshot, we can see the stacked features, such as **`Assets.Sum(Demographic Data.balance)`**, that were created.

Let's analyze this feature to really identify its mechanics.

In this feature, we are aggregating the Demographics data, which is the **`balance`** variable, with respect to the Asset portfolio. Here, we are summing the balance of customers who have a house and who don't have a house. To execute this manually, use the following code:

```
# Verifying the features for Assets.SUM(Demographic Data.balance)
bankData.groupby('AssetId')['balance'].agg('sum')
```

You should get the following output:

```
AssetId
0    32059342
1    29530340
Name: balance, dtype: int64
```

Figure 17.19: Each feature being analyzed

From the preceding code, we are grouping the **`AssetId`** variable using the **`groupby()`** function. An **`AssetId`** of 1 means the customer has a house, while 0 means no house. After grouping, summing is done on the **`balance`** method using the **`.agg()`** function. From the output, we can see that the value for **`'0'`** is **`32059342`** and that **`'1'`** is **`29530340`**. From the automated output, we can see these values against the **`Asset.housing = no`** and **`Asset.housing = 'yes'`** variables, respectively.

7. Let's print out all the feature names:

```
# Printing the list of all features
feature_names
```

You should get the following output:

```
[<Feature: age>,
 <Feature: job>,
 <Feature: marital>,
 <Feature: education>,
 <Feature: balance>,
 <Feature: contact>,
 <Feature: day>,
 <Feature: month>,
 <Feature: duration>,
 <Feature: campaign>,
 <Feature: pdays>,
 <Feature: previous>,
 <Feature: poutcome>,
 <Feature: AssetId>,
 <Feature: LoanId>,
 <Feature: FinbehId>,
 <Feature: Assets.housing>,
 <Feature: Liability.loan>,
 <Feature: FinBehaviour.default>,
 <Feature: Assets.SUM(Demographic Data.age)>,
 <Feature: Assets.SUM(Demographic Data.balance)>,
 <Feature: Assets.SUM(Demographic Data.day)>,
 <Feature: Assets.SUM(Demographic Data.duration)>,
 <Feature: Assets.SUM(Demographic Data.campaign)>,
 <Feature: Assets.SUM(Demographic Data.pdays)>,
 <Feature: Assets.SUM(Demographic Data.previous)>,
 <Feature: Assets.STD(Demographic Data.age)>,
 <Feature: Assets.STD(Demographic Data.balance)>,
 <Feature: Assets.STD(Demographic Data.day)>,
 <Feature: Assets.STD(Demographic Data.duration)>,
```

Figure 17.20: Output showing all the feature names

> **NOTE**
>
> The preceding output is only the partial list of features.

8. As we described at the beginning of this exercise, we can configure the list of primitives that are used for creating new features. Now, let's define the set of aggregation and transformation primitives we want. This is implemented as a list, as follows:

```
# Creating aggregation and transformation primitives
aggPrimitives=[
        'std', 'min', 'max', 'mean',
        'last', 'count'
```

```
]
tranPrimitives=[
        'percentile',
         'subtract_numeric', 'divide_numeric']
```

As we can see, we are specifying the list of primitives that we want to implement. These are defined as lists and then provided to the DFS function.

9. Create a new set of features with the custom primitive list.

   After defining our list of primitives, we will provide these primitives to the DFS so that it will create a new set of features. The new set of primitives are given as arguments to the DFS function:

```
# Defining the new set of features
feature_set, feature_names = ft.dfs(entityset=Bankentities,
target_entity = 'Demographic Data',
agg_primitives=aggPrimitives,
trans_primitives=tranPrimitives,
max_depth = 2,
verbose = 1,
n_jobs = 1)
```

You should get the following output:

```
Built 3420 features
Elapsed: 01:35 | Remaining: 00:00 | Progress: 100%|██████████| Calculated: 11/11 chunks
```

Figure 17.21: Creating a new set of features

From the output, we can see that the number of features has increased to around **3420**. This substantial increase is because, in the default mode, there are no transformation primitives. The default mode only contains aggregation primitives. So, with the additional transformation primitives we've provided, the DFS method has created a new set of features, because of which the feature list has increased to **3420**.

10. Let's print the head of the new data frame:

```
# Displaying the feature set
feature_set.head()
```

You should get an output similar to the following:

LoanId	FinbehId	PERCENTILE(age)	PERCENTILE(balance)	PERCENTILE(day)	PERCENTILE(duration)	PERCENTILE(campaign)	PERCENTILE(pdays)	PERCENTILE(previous)
0	0	0.935337	0.822919	0.112683	0.671717	0.194035	0.408695	0.408695
0	0	0.640983	0.208190	0.112683	0.413373	0.194035	0.408695	0.408695
0	0	0.560992	0.398863	0.112683	0.598549	0.194035	0.408695	0.408695
0	0	0.640983	0.008847	0.112683	0.664993	0.194035	0.408695	0.408695
0	0	0.718465	0.122172	0.196634	0.454358	0.194035	0.408695	0.408695

Figure 17.22: The new DataFrame

From the output, we can see the new features that were generated by the transformation primitive (**`PERCENTILE`**).

In this exercise, we implemented the DFS method to generate a new set of features.

In the business context, we have a new set of features that will help us predict whether a customer will buy a term deposit or not.

Let's summarize what we have learned in this exercise. To begin with, let's start with the default primitives. As seen from this exercise, when we did not specify any primitives, only default aggregation primitives were applied to generate a new set of features. Only when we specified our custom list were transformation primitives applied to generate a much larger feature set. It should be noted that when we specify a list of primitives, this new list will override the default set of primitives. Another aspect that should be noted is the growth in the number of features generated. The list of features that were generated grows with the number of primitives defined and also with the depth. By varying both of these parameters, we can vary the number of features that are generated.

The most important factor to be noted is the relevance of the features. The list of features that's generated by the utility is not a silver bullet. The Featuretools utility creates a list of features based on the relationship we build, the data types we implement, and the available features. Featuretools allows us to give a very exhaustive list, efficiently reducing our manual effort.

It is the onus of the data scientist to finally screen the list of features and take the most relevant ones.

So far in this chapter, we have learned about various techniques for generating automated features. We started off by putting together a domain story and defined various entities and set the relationship between the entities. Then, we used the entity set to generate a new set of features using deep feature synthesis. The proof of the pudding of feature engineering is in the performance of a model that uses all of these features.

Now, we'll build a logistic regression model using the features we built and then validate the performance of the new features that are generated.

EXERCISE 17.03: CLASSIFICATION MODEL AFTER AUTOMATED FEATURE GENERATION

In this exercise, we will build a logistic regression model based on the bank marketing dataset to predict the propensity for term deposit purchase. We will begin this exercise by fitting a benchmark model of raw features and then note the benchmark metrics. After this, we will generate new features using Featuretools and then build another model on the new dataset. Finally, we will analyze the results to observe the performance of the models we have built.

> **NOTE**
>
> The dataset file to be used in this exercise can be found in this book's GitHub repository at https://packt.live/2SSXPl3.

The following steps will help you complete this exercise:

1. Open a Colab notebook.
2. Define the path to the GitHub repository:

```
# Defining the path to the Github repository
file_url = 'https://raw.githubusercontent.com/PacktWorkshops/The-Data-Science-Workshop/master/Chapter17/Datasets/bank-full.csv'
```

3. Load the data using pandas:

```
# Loading data using pandas
import pandas as pd
bankData = pd.read_csv(file_url,sep=";")
bankData.head()
```

You should get a similar output to the following:

	age	job	marital	education	default	balance	housing	loan	contact	day	month	duration	campaign	pdays	previous	poutcome	y
0	58	management	married	tertiary	no	2143	yes	no	unknown	5	may	261	1	-1	0	unknown	no
1	44	technician	single	secondary	no	29	yes	no	unknown	5	may	151	1	-1	0	unknown	no
2	33	entrepreneur	married	secondary	no	2	yes	yes	unknown	5	may	76	1	-1	0	unknown	no
3	47	blue-collar	married	unknown	no	1506	yes	no	unknown	5	may	92	1	-1	0	unknown	no
4	33	unknown	single	unknown	no	1	no	no	unknown	5	may	198	1	-1	0	unknown	no

Figure 17.23: Loading data using pandas

4. We'll remove the target variable using the **`.pop()`** function:

```
# Removing the target variable
Y = bankData.pop('y')
```

Here, the **`.pop()`** function removes the defined variable from the dataset.

5. Now, we'll split the dataset into train and test sets:

```
from sklearn.model_selection import train_test_split


# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(bankData, Y,
test_size=0.3, random_state=123)
```

6. Then, we will use pipelines to transform the variables.

In this exercise, we will use pipelines to scale numerical variables and create dummy variables from categorical variables. This implementation is similar to the exercises that we completed in *Chapter 16, Machine Learning Pipelines*:

```
#Using pipeline to transform categorical variable and numeric
variables


from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder


categorical_transformer = Pipeline(steps=[('onehot',
OneHotEncoder(handle_unknown='ignore'))])
numeric_transformer = Pipeline(steps=[('scaler', StandardScaler())])
```

First, we define the categorical and numerical transformers, which are one-hot encoding and scaling, respectively.

7. After defining the transformation pipelines, we will define the categorical and numerical data types. This step is similar to what we did in *Chapter 16, Machine Learning Pipelines*:

```
# Defining data types for numeric and categorical features
numeric_features = bankData.select_dtypes(include=['int64',
'float64']).columns
categorical_features = bankData.select_dtypes(include=['object']).
columns
```

8. In this step, we create the preprocessor pipeline using the **`ColumnTransformer()`** function in scikit-learn. Please note that this step is similar to the one we implemented in *Chapter 16, Machine Learning Pipelines*:

```
# Defining preprocessor
from sklearn.compose import ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])
```

9. Again, similar to what we did in *Chapter 16, Machine Learning Pipelines*, we will create the estimator that contains the preprocessor and logistic regression classifier:

```
# Defining the estimator for processing and classification

from sklearn.linear_model import LogisticRegression
estimator = Pipeline(steps=[('preprocessor', preprocessor),
                            ('classifier',LogisticRegression(random_
state=123))])
```

10. Next, we'll fit the estimator on the training set and print the model's score:

```
# Fit the estimator on the training set
estimator.fit(X_train, y_train)
print("model score: %.2f" % estimator.score(X_test, y_test))
```

You should get a similar output to the following:

```
Model score: 0.90
```

11. Now, we'll predict on the test set:

```
# Predict on the test set
pred = estimator.predict(X_test)
```

12. Next, we'll generate the classification report, as follows:

```
# Generating classification report
from sklearn.metrics import classification_report

print(classification_report(pred,y_test))
```

You should get a similar output to the following:

```
                precision    recall  f1-score   support

           no       0.98      0.92      0.95     12765
          yes       0.32      0.64      0.43       799

     accuracy                           0.90     13564
    macro avg       0.65      0.78      0.69     13564
 weighted avg       0.94      0.90      0.92     13564
```

Figure 17.24: Expected classification matrix

Once the benchmark model has been created without feature engineering, we will proceed and create some new features using Featuretools and then fit another model on the new set of features. Now, we'll proceed by defining the entities and their relationships.

13. Create the customer ID for tracking entities:

```
# Creating the Ids for Demographic Entity

bankData['custID'] = bankData.index.values

bankData['custID'] = 'cust' + bankData['custID'].astype(str)
```

Since each row pertains to a unique customer, there will be as many customer IDs as there are rows in the dataset. You created the customer ID by taking the index value of each row and attaching a string called **`cust`** to the index values. The index values of the rows can be derived by the **`.index.values()`** method. These values are then attached to the **`cust`** string in the second line and the whole ID, which is a string value, is stored in the **`custID`** variable.

14. Next, we create the ID for Assets. As we explained earlier, Assets is mapped to the **housing** variable and has two possible values: **yes** or **no**. In the first line of the code, we initialize all the values of the **AssetId** variable to **0**. In the second line, we change the values of **AssetId** to **1** wherever the housing variable is **yes**. This can be implemented as follows:

```
# Creating AssetId
bankData['AssetId'] = 0
bankData.loc[bankData.housing == 'yes','AssetId']= 1
```

15. Similar to the step for assets, we create **LoanId** based on the value of the **loan** variable:

```
# Creating LoanId
bankData['LoanId'] = 0
bankData.loc[bankData.loan == 'yes','LoanId']= 1
```

16. Now, we'll create the ID for Financial Behavior based on the value of the **default** variable:

```
# Creating Financial behaviour ID
bankData['FinbehId'] = 0
bankData.loc[bankData.default == 'yes','FinbehId']= 1
```

17. Import the necessary libraries for Featuretools extraction.

    Featuretools can be implemented using the **featuretools** library:

```
# Importing necessary libraries
import featuretools as ft
import numpy as np
```

18. The first step is to initialize the entity set. This is implemented using the **.EntitySet()** method. We provide a string for tracking the entity set as an argument within the method. The string we have given here is **Bank**:

```
# creating the entity set 'Bankentities'
Bankentities = ft.EntitySet(id ='Bank')
```

19. Map the parent entity to the data frame.

    Once the entity set has been initialized, we have to map the bank dataset to the entity set and then create the parent entity, that is, Demographic Data. The parent entity is tracked by **`custID`**. Mapping the data frame is done using the **`.entity_from_dataframe()`** method. This is implemented as follows:

```
# Mapping a dataframe to the entityset to form the parent entity
Bankentities.entity_from_dataframe(entity_id = 'Demographic Data',
dataframe = bankData, index = 'custID')
```

    You should get the following output:

```
Entityset: Bank
  Entities:
    Demographic Data [Rows: 45211, Columns: 20]
  Relationships:
    No relationships
```

Figure 17.25: Mapping the parent entity

    In the preceding output, we can see that the first entity, which is the parent entity, **`Demographic Data`**, has been created. So far, no relationships have been created for this entity.

20. Map all the entities and set their relationships.

    Once the parent entity has been created, it is time to define each of the child entities and then create relationships between the parent entity, **`Demographic Data`**, and the child entities. This is done using the **`.normalize_entity()`** function. This can be implemented as follows:

```
# Mapping to parent entity and setting the relationship
Bankentities.normalize_entity(base_entity_id='Demographic Data', new_
entity_id='Assets', index = 'AssetId',
additional_variables = ['housing'])


Bankentities.normalize_entity(base_entity_id='Demographic Data', new_
entity_id='Liability', index = 'LoanId',
additional_variables = ['loan'])


Bankentities.normalize_entity(base_entity_id='Demographic Data', new_
entity_id='FinBehaviour', index = 'FinbehId',
additional_variables = ['default'])
```

You should get the following output:

```
Entityset: Bank
  Entities:
    Demographic Data [Rows: 45211, Columns: 17]
    Assets [Rows: 2, Columns: 2]
    Liability [Rows: 2, Columns: 2]
    FinBehaviour [Rows: 2, Columns: 2]
  Relationships:
    Demographic Data.AssetId -> Assets.AssetId
    Demographic Data.LoanId -> Liability.LoanId
    Demographic Data.FinbehId -> FinBehaviour.FinbehId
```

Figure 17.26: Mapping the entities to the relationships

From the preceding output, we can see that all four entities (Demographics, Assets, Liability, and Financial Behavior) are created and a relationship is formed between the parent entity (Demographics) and the child entities.

21. Create aggregation and transformation primitives.

    We can configure the list of primitives that are used for creating new features. Let's define the set of aggregation and transformation primitives we want. This is implemented as a list as follows:

```
# Creating aggregation and transformation primitives
aggPrimitives=[
        'std', 'min', 'max', 'mean',
         'last', 'count'

]
tranPrimitives=[
        'percentile',
         'subtract_numeric', 'divide_numeric']
```

22. In this step, we define the DFS with the created primitives. We set the depth to **2**:

```
# Defining the new set of features

feature_set, feature_names = ft.dfs(entityset=Bankentities,
target_entity = 'Demographic Data',
agg_primitives = aggPrimitives,
trans_primitives = tranPrimitives,
```

```
    max_depth = 2,
    verbose = 1,
    n_jobs = 1)
```

You should get a similar output to the following:

```
Built 3420 features
Elapsed: 01:42 | Remaining: 00:00 | Progress: 100%|██████████| Calculated: 11/11 chunks
```

Figure 17.27: Defining the new features

From the preceding output, we can see that **3420** features have been created.

23. Once the feature sets have been created, the indexing will be all jumbled up. We need to reindex them so that the index is similar to the original dataset. This is implemented as follows:

```
    # Reindexing the feature_set
    feature_set = feature_set.reindex(index=bankData['custID'])
    feature_set = feature_set.reset_index()
```

In the first line, reindexing is done using the **.reindex()** function. As an argument, we pass the target index, based on which the reindexing has to be done. In the argument, we specify that the indexing has to be done based on the order of **custID**. Once this line is implemented, the index of the data frame becomes 'cust01','cust02' ..., and so on. The second line, that is, **.reset_index()**, is used to change this index to **0**, **1**, **2**, and so on.

24. Now that the feature set has been created, we can print the shape of the feature set:

```
    # Displaying the feature set
    feature_set.shape
```

You should get the following output:

```
    (45211, 3421)
```

From the preceding output, we can see that **3421** new features have been created. The number of rows, that is, **45211**, remains the same as the original dataset.

25. Now, drop all the IDs. In the feature set, there are some features that are related to the IDs that we created. When building models, these IDs will not add any value as they are only for tracking purposes. Therefore, we can remove them. We can do this as follows:

```
# Dropping all Ids
X = feature_set[feature_set.columns[~feature_set.columns.str.
contains(
    'custID|AssetId|LoanId|FinbehId')]]
```

In this implementation, the tilde **~** sign means negation. In the preceding code, we are subsetting the feature set with those features that don't contain **`custID`**, **`AssetId`**, **`LoanId`**, or **`FinbehId`**. With this step, we remove all the features related to the IDs we created.

26. Replace all infinity values with **`nan`** values:

```
# Replacing all columns with infinity with nan
X = X.replace([np.inf, -np.inf], np.nan)
```

One after-effect of a transformation primitive such as divide is to create features with infinity values. This happens when there are features that contain **0**. As you know, division by **0** generates an infinity value. These infinity values have to be removed from the data frame because they are not desired during the modeling phase. This is done by replacing all the infinity values with nan values and then later dropping the nan values. The first step is implemented as follows using the **`.replace()`** function.

> **NOTE**
>
> **`np.inf`** and **`-np.inf`** stands for infinity values.

27. Drop all the columns containing **`nan`**. Once the replacement of infinity values with **`nan`** has been done, these columns can be dropped from the dataset. This is implemented using the **`.dropna()`** function:

```
# Dropping all columns with nan
X = X.dropna(axis=1, how='any')
X.shape
```

You should get the following output:

```
(45211, 1046)
```

In the preceding implementation, **`axis = 1`** means along the columns. **`how = 'any'`** means drop any column contacting **`nan`** values. We can see that the number of features drops from **`3421`** to **`1046`** after removing all the redundant columns.

28. Now, let's split the new dataset into train and test sets for modeling:

```
# Splitting train and test sets
from sklearn.model_selection import train_test_split


# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_
size=0.3, random_state=123)
```

29. Like we did during the benchmark model creation step, let's create the processing pipeline. We will use pipelines to scale numerical variables and create dummy variables from categorical variables. This implementation is similar to the implementation we completed in *Chapter 16, Machine Learning Pipelines*:

```
# Creating the preprocessing pipeline


from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder


categorical_transformer = Pipeline(steps=[('onehot',
OneHotEncoder(handle_unknown='ignore'))])

numeric_transformer = Pipeline(steps=[('scaler', StandardScaler())])


numeric_features = X.select_dtypes(include=['int64', 'float64']).
columns
categorical_features = X.select_dtypes(include=['object']).columns


from sklearn.compose import ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])
```

As seen from the preceding implementation, we give the necessary transformations through the **`transformers`** argument. The first transformer is the numerical transformer, which is represented using the **`numeric`** string. Then, we apply the transformer, **`numTransformer`**, which is a scaler function on the numerical features, **`numFeatures`**. Similarly, we define the appropriate transformations on the categorical variables.

30. Let's create the estimator function, which contains the processing step and the classifier layer. After doing this, we'll fit the estimator on the training set and print the scores. This implementation is similar to the pipeline estimator we created in *Chapter 16, Machine Learning Pipelines*

```
# Creating the estimator function and fitting the training set
estimator = Pipeline(steps=[('preprocessor', preprocessor),
                            ('classifier',LogisticRegression(random_
state=123))])
estimator.fit(X_train, y_train)
print("model score: %.2f" %estimator.score(X_test, y_test))
```

You should get the following output:

```
Model score: 0.90
```

As we can see, the estimator is defined using the **`Pipeline()`** function, to which we provide the steps we described previously. Then, we fit the training set on the estimator using the **`.fit()`** function. Finally, the model scores are printed.

As seen from the output, the accuracy level remains the same as the benchmark model. Let's also see what the classification report looks like.

31. Once the fitting has been done, the predictions are generated using the **`.predict()`** function. Prediction is done on the test set:

```
# Predicting on the test set
pred = estimator.predict(X_test)
```

32. Once the predictions have been generated, the classification report is generated:

```
# Generating the classification report
from sklearn.metrics import classification_report

print(classification_report(pred,y_test))
```

You should get the following output:

```
                precision    recall  f1-score   support

           no       0.97      0.92      0.95     12716
          yes       0.35      0.65      0.45       848

     accuracy                           0.90     13564
    macro avg       0.66      0.78      0.70     13564
 weighted avg       0.94      0.90      0.92     13564
```

Figure 17.28: Expected classification matrix

From the preceding output, we can see that the accuracy scores have remained the same. However, there is improvement in the precision, recall, and f1-score of the minority class (yes). All of these values have increased from **`34%`**, **`64%`**, and **`43%`**, respectively. We should remember that this is an extremely unbalanced dataset. However, with the new features we generated, we were able to show marginal improvement in the performance of the minority class.

Having said that, it will be important to note that automated feature engineering is not a silver bullet that will always guarantee improvements in performance. The value this utility adds is in giving us an exhaustive list of features to select from. The real value this method adds is in making the feature engineering process quite efficient and also providing the data scientist with an exhaustive list to choose from. Once equipped with an exhaustive list of features, the onus is upon the data scientist to apply domain understanding, experience, and intuition so that they can select the features that they feel would make a difference.

From a business perspective, this result indicates that out of the total 848 customers who were likely to buy a term deposit, only 65% of them (the recall of yes) were correctly identified as having the propensity to buy a term deposit.

FEATURETOOLS ON A NEW DATASET

In this chapter, we have learned about Featuretools and how to build automated features using it. In the next activity, we will apply what we have learned to a new dataset. This dataset is a modified version of the adult dataset from the UCI Machine Learning Repository, Irvine, CA: University of California, School of Information and Computer Science, which can be found at https://archive.ics.uci.edu/ml/machine-learning-databases/adult/, in the **`adult.data`** file. This dataset has various attributes of a working adult, such as age, occupation, education, and native. The task is to predict whether a particular adult will earn more than **`50,000`** in their yearly salary or not.

The details about the various attributes are available at the preceding link in the **`adult.names`** file. This dataset has a mix of both categorical and numerical data and is a good dataset to try out what you have learned about Featuretools.

ACTIVITY 17.01: BUILDING A CLASSIFICATION MODEL WITH FEATURES THAT HAVE BEEN GENERATED USING FEATURETOOLS

In this activity, you will build a logistic regression model on the adult dataset to predict whether an adult will earn more than 50,000 per year or not. You will begin this activity by fitting a benchmark model on raw features and then note the benchmark metrics. After this, you will generate new features using Featuretools and then build another model on the new dataset. You should analyze the results to observe the performance of the models you've built.

> **NOTE**
>
> The dataset file to be used in this activity can be found in this book's GitHub repository at https://packt.live/39LrZfM.

Follow these steps to complete this activity:

1. Read the data using **`pandas`** from the following GitHub repository:

   https://packt.live/2T7OAO7

2. Drop all **`na`** values using the **`.dropna()`** function.
3. Create the **`Y`** variable using the **`.pop()`** function on the **`label`** variable.
4. Split the dataset into train and test sets.
5. Create the processor pipeline to convert categorical variables into one-hot encoding and numerical variables into scaled variables.
6. Define the estimator function using the data processor and a logistic regression classifier.
7. Fit the estimator on the train set and then print the scores on the test set.
8. Generate predictions on the test set using the estimator function.
9. Print the classification report.

10. Create a parent entity ID called **`parentID`** that's similar to the **`custID`** we created in *Exercise 17.01*.
11. The different child entities can be variables such as education, work class, and occupation. For education, the education-num variable can be used as the ID. For the other two variables, create some unique IDs that depend on the number of unique values in that variable.
12. For the **`workclass`** variable, the unique values are as follows:

    ```
    ' Federal-gov', ' Local-gov', ' Private',' Self-emp-inc',
    ' Self-emp-not-inc', ' State-gov', ' Without-pay'
    ```

13. For the **`Occupation`** variable, the unique values are as follows:

    ```
    ' Adm-clerical', ' Armed-Forces',' Craft-repair', ' Exec-
    managerial', ' Farming-fishing',' Handlers-cleaners', '
    Machine-op-inspct', ' Other-service', ' Priv-house-serv',
    ' Prof-specialty',' Protective-serv',' Sales', ' Tech-
    support', ' Transport-moving'
    ```

14. Create the parent entity and set the relationship with education, workclass, and occupation using their respective IDs.
15. Create the aggregation and transformation primitives.
16. Create the DFS with the defined primitives.
17. Reindex the created data frame.
18. Drop all the variables related to the IDs you've created.
19. Replace all the infinity values with **`na`** and the drop columns with **`na`** using the **`dropna()`** function.
20. Split the dataset into train and test sets.
21. Create the processing pipeline.
22. Create the estimator function and fit the training set on the estimator. Then, generate the scores.
23. Generate predictions on the test set and print the classification report.

You should get a similar output to what's shown here.

The following is the classification report for the benchmark model:

	precision	recall	f1-score	support
0	0.93	0.88	0.91	7189
1	0.62	0.75	0.68	1860
accuracy			0.85	9049
macro avg	0.77	0.81	0.79	9049
weighted avg	0.87	0.85	0.86	9049

Figure 17.29: Expected classification matrix for the benchmark model

The following is the classification report for the feature engineered model:

	precision	recall	f1-score	support
0	0.93	0.89	0.91	7134
1	0.64	0.76	0.69	1915
accuracy			0.86	9049
macro avg	0.79	0.82	0.80	9049
weighted avg	0.87	0.86	0.86	9049

Figure 17.30: Expected classification matrix for the feature engineered model

SUMMARY

In this chapter, we learned how to use Featuretools. As part of your journey as a data scientist, you will come to realize that building features is a very tedious process and involves a lot of effort and time. However, we have seen that by using Featuretools, the generation of new features is very efficient and we get a very exhaustive set of potential features that we can try on our model.

In this chapter, we approached the feature engineering step from a domain/business perspective. We identified some critical business factors and built a domain story by connecting these business factors.

We also learned about some of the critical building blocks for automated feature engineering, such as entities and entity sets. We used the domain story we formulated to create the entities and set the relationships between the parent entity and the child entities.

Once the entity set was defined, we were exposed to the basic operations for feature tools, such as aggregation and transformation, which are called primitives. We also created a new set of features using the deep feature synthesis method in feature tools. Using the newly created feature sets, we built a logistic regression model to predict the propensity of term deposit purchases. We found that the new set of features improved the performance of the minority class. Then, we applied our learning to a new dataset where we predicted whether an adult would earn more than 50,000 per year. We observed that the feature sets we created improved the performance of the logistic regression model we built.

The objective of this chapter was to introduce you to a very potent tool that will help in generating an exhaustive list of features. However, it should be remembered that building an exhaustive set of features is not the end and it is only a means to the end. The onus is on the data scientist to carefully verify the generated list of features and select the most important ones based on domain understanding, intuition, and experience.

Now that we've learned about a very important toolset, we will proceed to the next chapter, where we will be introduced to how our model can be built as a service.

The next chapter will cover how our data science project can be made into a data product using a package called Flask.