Certified Data Science Practitioner Professional Certificate Lab Instructions

How to Use This Document

This document contains instructions for all labs in the Certified Data Science Practitioner (CDSP) Specialization. It also includes lab instructions for projects that have a lab component. To access the lab instructions for the particular lab you're working on, navigate to the appropriate course and module, and look for the name of the relevant lab. If you're not taking every course in the CDSP Specialization, you can skip directly to the labs for the course(s) you are taking.

In the lab instructions, the numbered steps typically give an overview of what you're about to do, whereas the lettered substeps tell you exactly what to do in the lab environment, and provide valuable explanations. Some labs also include discussion questions. You don't need to submit your answers—they're just to get you thinking about the task(s) you're performing. The "Solutions" section at the end of the document gives you some example answers.

Certified Data Science Practitioner Professional Certificate Lab Instructions

Course 2: Extract, Transform, and Load Data

Course Introduction

The following labs are for Course 2: Extract, Transform, and Load Data.

Modules

The labs in this course pertain to the following modules:

- Module 1: Extract Data
- Module 2: Transform Data
- Module 3: Load Data
- Apply What You've Learned

MODULE 1

Extract Data

The following labs are for Module 1: Extract Data.

LAB 2-1
Reading Data from CSV Files

Data Files

~/ETL/Extracting, Transforming, and Loading Data.ipynb

~/ETL/data/consumer_loan_complaints.csv

Scenario

Greene City National Bank (GCNB) has given you access to the sources of data that you've agreed could be useful for achieving its business goals. The data is spread out among different files and databases, and is stored in different formats. From what the data owners at GCNB have told you, there are four main tables of data:

- **Users**: This table includes rows of users and various columns that describe each user's demographics and banking information. This is the largest table that GCNB provided in terms of the sheer volume of data.
- **User devices**: This table tracks what kind of device each user primarily uses when banking electronically with GCNB.
- **User transaction history**: This table includes records of individual financial transactions and the monetary value of each transaction.
- **Consumer loan complaints**: This table includes information about complaints that users have sent to GCNB when going through the loan process.

Your job is to starting pulling all of this data together so that it can eventually be cleaned and consolidated into a single working environment. The consumer loan complaint data is contained within its own comma-separated values (CSV) file, so you'll start by reading that data first.

Note: By default, these lab steps assume you will be typing the code shown in screenshots. Many learners find it easier to understand what code does when they are able to type it themselves. However, if you prefer not to type all of code, the **Solutions** folders contain the finished code for each notebook.

---

1. From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   - The Jupyter Notebook session shows a listing of directories.
   - You can use this listing to navigate to folders that contain notebooks you want to open.
2. Open the notebook.
   a) Select **ETL**.

   The **ETL** directory contains a subdirectory named **data** and a notebook file named **Extracting, Transforming, and Loading Data.ipynb**.
   b) Select **Extracting, Transforming, and Loading Data.ipynb** to open it.
   c) If line numbers aren't visible, select **View→Toggle Line Numbers**.
3. Import the relevant software libraries.

a) View the cell titled **Import software libraries**, and examine the code listing below it.

```
import sys           # Read system parameters.
import pandas as pd  # Manipulate and analyze data.
import sqlite3       # Manage SQL databases.

# Summarize software libraries used.
print('Libraries used in this project:')
print('- Python {}'.format(sys.version))
print('- pandas {}'.format(pd.__version__))
print('- sqlite3 {}'.format(sqlite3.sqlite_version))
```

b) Select the cell that contains the code listing, then select **Run**.

c) Verify that the version of Python is displayed, as are the versions of pandas and sqlite3 that were imported.

```
Libraries used in this project:
- Python 3.8.5 (default, Sep  4 2020, 07:30:14)
[GCC 7.3.0]
- pandas 1.1.3
- sqlite3 3.33.0
```

4. Load a CSV file as a `DataFrame`.

a) Scroll down and view the cell titled **Load a CSV file as a `DataFrame`**, then select the code cell below it.

b) In the code cell, type the following:

```
complaints_data = pd.read_csv('data/consumer_loan_complaints.csv')
```

c) Run the code cell.

This code uses the pandas `read_csv()` function to read the comma-separated values (CSV) file that holds the consumer loan complaints.

Note: Some of the code blocks don't have an output. You can tell that a code block was executed by the presence of a number inside the brackets to the left of the block, like `[2]`.

5. Preview the first three rows of the data.

a) Scroll down and view the cell titled **Preview the first three rows of the data**, then select the code cell below it.

b) In the code cell, type the following:

```
complaints_data.head(n = 3)
```

c) Run the code cell.

d) Examine the output.

	user_id	Date received	Product	Issue	Consumer complaint narrative	State	ZIP code	Submitted via	Date sent to company	Company response to consumer	Timely response?	Consumer disputed?	Complaint ID
0	44fefdad-7045-4be5-890e-12e84ae6fdc9	01/27/2016	Consumer Loan	Account terms and changes	NaN	AL	35180	Phone	01/27/2016	Closed with explanation	Yes	No	1760486
1	c49d5d60-909f-406b-b7ff-51143fcb650b	08/26/2014	Consumer Loan	Account terms and changes	NaN	NC	278XX	Phone	08/29/2014	Closed with non-monetary relief	Yes	No	1001740
2	9b2cd5d2-900e-4052-831f-6489f6d568af	08/22/2012	Consumer Loan	Account terms and changes	NaN	TN	37205	Referral	08/23/2012	Closed with non-monetary relief	Yes	No	140039

Each row in the table is a user that submitted a complaint. The columns are as follows:

- **`user_id`**: An arbitrary hexadecimal string that uniquely identifies the user.
- **`Date received`**: The date the complaint was received by the organization.
- **`Product`**: The product the complaint is about.
- **`Issue`**: The type of issue the complaint is about.
- **`Consumer complaint narrative`**: The text of any written complaints. This field is optional for the complainant to fill out.
- **`State`**: The U.S. state the user resides in.
- **`ZIP code`**: The ZIP code the user resides in.
- **`Submitted via`**: The method the user took to file the complaint.
- **`Date sent to company`**: The date the complaint was sent by the user.
- **`Company response to consumer`**: How the organization handled the complaint.
- **`Timely response?`**: Whether or not the organization's response was, according to some metric, given within an acceptable time period.
- **`Consumer disputed?`**: Whether or not the user disputed the action the organization took in response to the complaint.
- **`Complaint ID`**: A unique identifier for the complaint itself.

6. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 2-2

Extracting Data with Database Queries

Data Files

~/ETL/Extracting, Transforming, and Loading Data.ipynb

~/ETL/data/user_data.db

Scenario

The remaining three tables GCNB provided to you are all contained within an SQL database. You have been given access to this database, but in order to actually retrieve the data, you'll need to execute some SQL queries. You also want to begin shaping the data so that it's in a more workable form. For example, the transaction history database currently records each transaction separately, even transactions made by the same user at different times. Rather than keep it like this, you want to aggregate the transactions for each user so that it's easier to integrate this data with the other tables when the time comes.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **ETL/Extracting, Transforming, and Loading Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Create a connection to the SQLite database** heading, then select **Cell→Run All Above**.

2. Create a connection to the SQLite database.
   a) Scroll down and view the cell titled **Create a connection to the SQLite database**, then select the code cell below it.
   b) In the code cell, type the following:

```
conn = sqlite3.connect('data/user_data.db')
conn
```

   c) Run the code cell.

      You are connecting to an SQLite database, which is just a file on the local file system. In a production environment, you'd likely connect to an SQL server using credentials.
   d) Examine the output.

```
<sqlite3.Connection object at 0x7f2ba83c6c60>
```

      The `conn` object established a connection to the database.

3. Read the `users` data.
   a) Scroll down and view the cell titled **Read the `users` data**, then select the code cell below it.

b) In the code cell, type the following:

```
# Write a query that selects everything from the users table.

query = 'SELECT * FROM users'
```

c) Run the code cell.

You're defining the query that you'll pass to the database in the next code block.

d) Select the next code cell, then type the following:

```
# Read the query into a DataFrame.

users = pd.read_sql(query, conn)

# Preview the data.

users.head()
```

e) Run the code cell.

f) Examine the output.

	user_id	age	job	marital	education	default	housing	loan	contact	duration	campaign	pdays	previous	poutcome	term_deposit
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	58	management	married	tertiary	no	yes	no	None	261	1	-1	0	None	no
1	bb92765a-08de-4963-b432-496524b39157	44	technician	single	secondary	no	yes	no	None	151	1	-1	0	None	no
2	573de577-49ef-42b9-83da-d3cfb817b5c1	33	entrepreneur	married	secondary	no	yes	yes	None	76	1	-1	0	None	no
3	d6b66b9d-7c8f-4257-a682-e136f640b7e3	47	blue-collar	married	None	no	yes	no	None	92	1	-1	0	None	no
4	fade0b20-7594-4d9a-84cd-c02f79b1b526	33	None	single	None	no	no	no	None	198	1	-1	0	None	no

The database file includes a `users` table, the first few rows of which are printed. Like with the consumer complaints file, the users are identified using the same hexadecimal ID numbers. You can consider this the primary key for most of these tables. There are also various columns:

- **`age`**: The age of the user.
- **`job`**: The user's job title.
- **`marital`**: The user's marital status.
- **`education`**: The user's level of education.
- **`default`**: Whether the user has defaulted on a loan.
- **`housing`**: Whether or not the user has a housing loan.
- **`loan`**: Whether or not the user has a personal loan.
- **`contact`**: The method the user and organization use to communicate.
- **`duration`**: The duration of the last contact session with the user, in seconds.
- **`campaign`**: Number of times the user was contacted for the current marketing campaign.
- **`pdays`**: Number of days that passed after the user was contacted from a previous campaign.
- **`previous`**: Number of times the user was contacted prior to the current campaign.
- **`poutcome`**: The result of the previous campaign.
- **`term_deposit`**: Whether or not the client subscribed to a term deposit.
- **`date_joined`**: The date the user signed up for an account.

Note: Throughout the CDSP Specialization, you can reference **data_dictionary.csv** in the data files if you need a reminder of what a column means.

g) Select the next code cell, then type the following:

```
# Check the shape of the data.

users.shape
```

h) Run the code cell.
i) Examine the output.

```
(45216, 16)
```

This table has 45,216 rows and 16 columns.

4. Read the `device` data.

a) Scroll down and view the cell titled **Read the `device` data**, then select the code cell below it.
b) In the code cell, type the following:

```
query = 'SELECT * FROM device'

device = pd.read_sql(query, conn)

device.head()
```

c) Run the code cell.
d) Examine the output.

	user_id	device
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	mobile
1	bb92765a-08de-4963-b432-496524b39157	desktop
2	573de577-49ef-42b9-83da-d3cfb817b5c1	mobile
3	d6b66b9d-7c8f-4257-a682-e136f640b7e3	tablet
4	fade0b20-7594-4d9a-84cd-c02f79b1b526	mobile

The database file also includes a `device` table. Once again, the `user_id` field is present. There's also a `device` column that lists the user's preferred type of device to use when banking electronically.

e) Select the next code cell, then type the following:

```
device.shape
```

f) Run the code cell.
g) Examine the output.

```
(45117, 2)
```

This table has 45,117 rows and 2 columns.

5. Read the `transactions` data.

a) Scroll down and view the cell titled **Read the `transactions` data**, then select the code cell below it.

b) In the code cell, type the following:

```
# Read the user transactions in the last 30 days.

query = 'SELECT * FROM transactions'

transactions = pd.read_sql(query, conn)

transactions.head()
```

c) Run the code cell.

d) Examine the output.

	user_id	transaction_id	amount_usd
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	transaction_5180	1332
1	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	transaction_5607	726
2	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	transaction_6765	85
3	573de577-49ef-42b9-83da-d3cfb817b5c1	transaction_6170	1
4	573de577-49ef-42b9-83da-d3cfb817b5c1	transaction_6090	1

The final table in the database lists transaction information for users. Aside from the familiar `user_id`, there are two other columns:

- **`transaction_id`**: A unique identifier for each transaction.
- **`amount_usd`**: The total amount of the transaction in U.S. dollars. Positive transactions indicate a deposit, whereas negative transactions indicate a withdrawal.

e) Select the next code cell, then type the following:

```
transactions.shape
```

f) Run the code cell.

g) Examine the output.

```
(140034, 3)
```

This table has 140,034 rows and 3 columns. There are so many more rows in this table than the others because of its transactional nature; for example, the first three rows are all transactions by the same user.

6. Aggregate the `transactions` data.

a) Scroll down and view the cell titled **Aggregate the `transactions` data**, then select the code cell below it.

b) In the code cell, type the following:

```
# Aggregate data on the number of transactions and the total amount.

query = '''SELECT user_id,
                  COUNT(*) AS number_transactions,
                  SUM(amount_usd) AS total_amount_usd
           FROM transactions
           GROUP BY user_id'''

transactions_agg = pd.read_sql(query, conn)

transactions_agg.head()
```

The query you'll run can be expressed as follows: From the `transactions` table, for each user ID, count each individual transaction and make it a new column. Also add up the USD of each transaction for a user ID and make it a new column. And, group the rows by each unique user ID.

c) Run the code cell.

d) Examine the output.

	user_id	number_transactions	total_amount_usd
0	0001570d-8aed-465e-b547-8981651084ed	3	792
1	000548ed-aa18-4eef-b8ed-68a9126e33ab	2	1044
2	00069959-4d55-460e-bb76-ae13ddbd80a6	5	0
3	000bab00-aec4-4ee2-81a6-1f897c38726b	19	0
4	000cbac8-212f-46fb-b58f-861dada34284	2	399

Now, you have a table where each user has all of their transactions consolidated into a single row. The number of those transactions and the total amount of those transactions appear as new columns.

e) Select the next code cell, then type the following:

```
transactions_agg.shape
```

f) Run the code cell.

g) Examine the output.

```
(35211, 3)
```

There are now 35,211 rows and 3 columns in this aggregate dataset of user transactions.

7. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 2-3
Consolidating Data from Multiple Sources

Data File

~/ETL/Extracting, Transforming, and Loading Data.ipynb

Scenario

You've extracted all of the relevant data from each source. The data tables currently live in separate objects, but they share a common key: `user_id`. Instead of keeping them all separate, you'll begin consolidating them using this primary key so they end up as a single table. In particular, you'll create a master table that includes user demographics, banking information, device usage, and transaction history. Having this master table will make many forthcoming data science tasks easier.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **ETL/Extracting, Transforming, and Loading Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Merge the `device` table with the `users` table** heading, then select **Cell→Run All Above**.

2. Merge the `device` table with the `users` table.
   a) Scroll down and view the cell titled **Merge the `device` table with the `users` table**, then select the code cell below it.
   b) In the code cell, type the following:

```
# Do a left join, as all users in the users table are of interest.

query = '''SELECT left_table.*,
                  right_table.device
             FROM users AS left_table
             LEFT JOIN device AS right_table
               ON left_table.user_id = right_table.user_id'''

users_w_device = pd.read_sql(query, conn)
```

   You're going to be doing a left join on this data, as all of the users in the `users` table (left) are of interest, and you want to merge any records that match the `user_id` field from the `device` table (right).
   c) Run the code cell.
   d) Select the next code cell, then type the following:

```
users_w_device.head(n = 3)
```

   e) Run the code cell.

f) Examine the output.

d	age	job	marital	education	default	housing	loan	contact	duration	campaign	pdays	previous	poutcome	term_deposit	date_joined	device
5-7-f6	58	management	married	tertiary	no	yes	no	None	261	1	-1	0	None	no	1998-08-23	mobile
a-3-47	44	technician	single	secondary	no	yes	no	None	151	1	-1	0	None	no	2008-07-15	desktop
f-a-21	33	entrepreneur	married	secondary	no	yes	yes	None	76	1	-1	0	None	no	2002-06-04	mobile

If you scroll the table all the way to the right, you'll see the `device` column has been merged for each user.

g) Select the next code cell, then type the following:

```
1 users_w_device.shape
```

h) Run the code cell.

i) Examine the output.

```
(45216, 17)
```

There are 45,216 rows and 17 columns in this new table. The number of rows is equal to the original `users` table, indicating that the join worked as expected.

3. Close the database connection.

a) Scroll down and view the cell titled **Close the database connection**, then select the code cell below it.

b) In the code cell, type the following:

```
1 conn.close()
```

c) Run the code cell.

It's always a good idea to close the connection to a database when you're no longer using it. The next merge you'll perform will use pandas instead of SQL.

4. Merge `users_w_device` with `transactions_agg`.

a) Scroll down and view the cell titled **Merge `users_w_device` with `transactions_agg`**, then select the code cell below it.

b) In the code cell, type the following:

```
1 # Do a right join so users won't be lost.
2
3 users_w_devices_and_transactions = \
4 transactions_agg.merge(users_w_device,
5                        on = 'user_id', how = 'right')
6
7 users_w_devices_and_transactions.head()
```

You're creating a new `DataFrame` that will merge the aggregated transactions data with the combined users and devices table. You're doing a right join so that `users_w_device` (right) won't lose any users that aren't also in `transactions_agg` (left).

c) Run the code cell.

d) Examine the output.

	user_id	number_transactions	total_amount_usd	age	job	marital	education	default	housing	loan	contact	duration	campaign	p
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	3.0	2143.0	58	management	married	tertiary	no	yes	no	None	261	1	
1	bb92765a-08de-4963-b432-496524b39157	NaN	NaN	44	technician	single	secondary	no	yes	no	None	151	1	
2	573de577-49ef-42b9-83da-d3cfb817b5c1	2.0	2.0	33	entrepreneur	married	secondary	no	yes	yes	None	76	1	
3	d6b66b9d-7c8f-4257-a682-e136f640b7e3	NaN	NaN	47	blue-collar	married	None	no	yes	no	None	92	1	
4	fade0b20-7594-4d9a-84cd-c02f79b1b526	1.0	1.0	33	None	single	None	no	no	no	None	198	1	

This new table not only has the main user information including what device they use, but it also includes the number of transactions and the total amount spent on those transactions. Even where transaction data is missing (e.g., the second row), the row is preserved.

e) Select the next code cell, then type the following:

```
# Make sure number of rows is equal to users_w_devices table.

users_w_devices_and_transactions.shape
```

f) Run the code cell.

g) Examine the output.

```
(45216, 19)
```

As expected, the new table has the same number of rows as the `users_w_devices` table (45,216). The table has also grown to include 19 total columns.

5. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

MODULE 2

Transform Data

The following labs are for Module 2: Transform Data.

LAB 2-4
Handling Irregular and Unusable Data

Data File

~/ETL/Extracting, Transforming, and Loading Data.ipynb

Scenario

The tables that GCNB sent over have thousands of records, and it's unlikely that they are in a pristine state. There's a chance they include at least some corrupt or faulty data, whether as a result of data entry errors or something else. In any case, you need to find any unusable data and deal with it so that it doesn't cause issues later on. You have a suspicion that some of the user ages may have been recorded incorrectly, so you'll look for evidence of that and take the appropriate action, if necessary. There may also be other circumstances that could indicate faulty data, so you'll do some more investigation.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **ETL/Extracting, Transforming, and Loading Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Identify data where `age` is greater than 150** heading, then select **Cell→Run All Above**.

2. Identify data where `age` is greater than 150.
   a) Scroll down and view the cell titled **Identify data where `age` is greater than 150**, then select the code cell below it.
   b) In the code cell, type the following:

```
1 users_w_devices_and_transactions[users_w_devices_and_transactions.age > 150]
```

   c) Run the code cell.
   d) Examine the output.

	user_id	number_transactions	total_amount_usd	age	job	marital	education	default	housing	loan	contact	duration	campaign
7228	44fefdad-7045-4be5-890e-12e84ae6fdc9	NaN	NaN	178	blue-collar	married	primary	no	yes	no	None	691	1
10318	9b2cd5d2-900e-4052-831f-6489f6d568af	2.0	3165.0	891	management	married	tertiary	no	yes	no	None	278	2

There are two users whose age is greater than 150, which suggests the data is corrupted. So, you will remove these rows from the dataset.

3. Drop incorrect data.
   a) Scroll down and view the cell titled **Drop incorrect data**, then select the code cell below it.

b) In the code cell, type the following:

```
users_cleaned = \
users_w_devices_and_transactions[users_w_devices_and_transactions.age < 150]

users_cleaned.shape
```

You're creating a new `DataFrame` with all of the same data, except for the two corrupt rows. Dropping entire rows with faulty data isn't the only approach you could take, especially if you believe the values in the other columns are accurate. However, since there are only two affected records out of tens of thousands, it's safe to just remove them.

c) Run the code cell.

d) Examine the output.

```
(45214, 19)
```

There are now 45,214 rows—two fewer than the original.

4. Identify more potentially erroneous data.

a) Scroll down and view the cell titled **Identify more potentially erroneous data**, then select the code cell below it.

b) In the code cell, type the following:

```
# Compare age to device.

pd.crosstab(users_cleaned['age'], users_cleaned['device'])
```

There may be other instances of anomalous data that aren't as obvious, and are only identifiable when placed in a specific context. Here you're using `crosstab()` to generate a frequency table where the number of devices used by each age will be shown.

c) Run the code cell.

d) Examine the output.

device age	desktop	mobile	tablet
18	5	6	1
19	10	22	3
20	11	33	6
21	16	44	19
22	30	87	11
...	...	...	...
90	1	1	0
92	1	1	0
93	0	2	0
94	0	1	0
95	0	1	1

77 rows × 3 columns

The frequency table shows that younger users are much more likely to use electronic devices to do their banking than people who are 90+ years old. Since the number of devices used by these elderly users is so sparse (one to two total per age), it may suggest anomalous data. However, this isn't necessarily the case, so you'll leave the data alone for now. Still, it's worth the effort to at least investigate potential anomalies.

5. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 2-5
Correcting Data Formats

Data File

~/ETL/Extracting, Transforming, and Loading Data.ipynb

Scenario

Thankfully, it seems that most of the data types in the dataset are being cast correctly. However, you'll need to convert string objects to Boolean values where appropriate, since some columns should be `True` or `False`. Also, dates and times can often cause problems when they're pulled into a programming environment, especially since they're often just cast as standard strings. While this isn't necessarily a problem, it's much easier to work with dates and times when they're cast as datetime objects. So, you'll convert the relevant column from a string object to a datetime.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **ETL/Extracting, Transforming, and Loading Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Identify data types that need correcting** heading, then select **Cell→Run All Above**.

2. Identify data types that need correcting.
   a) Scroll down and view the cell titled **Identify data types that need correcting**, then select the code cell below it.
   b) In the code cell, type the following:

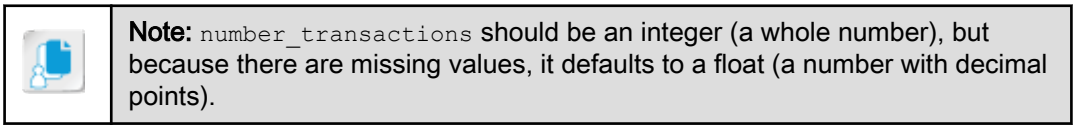
```
1 users_cleaned.info()
```

   c) Run the code cell.

d) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 45214 entries, 0 to 45215
Data columns (total 19 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   user_id              45214 non-null  object
 1   number_transactions  35215 non-null  float64
 2   total_amount_usd     35215 non-null  float64
 3   age                  45214 non-null  int64
 4   job                  44926 non-null  object
 5   marital              45214 non-null  object
 6   education            43357 non-null  object
 7   default              45214 non-null  object
 8   housing              45214 non-null  object
 9   loan                 45214 non-null  object
 10  contact              32196 non-null  object
 11  duration             45214 non-null  int64
 12  campaign             45214 non-null  int64
 13  pdays                45214 non-null  int64
 14  previous             45214 non-null  int64
 15  poutcome             8255 non-null   object
 16  term_deposit         45214 non-null  object
 17  date_joined          45184 non-null  object
 18  device               45120 non-null  object
dtypes: float64(2), int64(5), object(12)
memory usage: 6.9+ MB
```

You can see that `date_joined` has a data type of `object` (a string) instead of `datetime64` (a datetime format). A datetime format will make the column easier to work with. Also, none of the columns are of a Boolean type, and you know that at least some of them should be based on their "yes" and "no" values.



Note: `number_transactions` should be an integer (a whole number), but because there are missing values, it defaults to a float (a number with decimal points).

e) Select the next code cell, then type the following:

```
1 users_cleaned.default.value_counts()
```

This code will print the values for the `default` variable and their frequencies.

f) Run the code cell.

g) Examine the output.

```
no     44398
yes      816
Name: default, dtype: int64
```

The `default` variable has only "yes" or "no" values in string object form. Other variables follow this pattern, including `housing`, `loan`, and `term_deposit`. It would be better if these were cast as Booleans.

3. Convert the relevant variables to a Boolean type.

a) Scroll down and view the cell titled **Convert the relevant variables to a Boolean type**, then select the code cell

b) In the code cell, type the following:

```
users_cleaned_1 = users_cleaned.copy()  # Work with a new object.

users_cleaned_1.default = \
users_cleaned_1.default.map(dict(yes = 1, no = 0)).astype(bool)

users_cleaned_1.default.value_counts()
```

c) Run the code cell.

d) Examine the output.

```
False    44398
True       816
Name: default, dtype: int64
```

The values that were "yes" and "no" are now `True` and `False` for the `default` variable.

e) Select the next code cell, then type the following:

```
# Do the same for the other Boolean variables.

bool_vars = ['housing', 'loan', 'term_deposit']

for var in bool_vars:
    users_cleaned_1[var] = \
    users_cleaned_1[var].map(dict(yes = 1, no = 0)).astype(bool)

    print(f'Converted {var} to Boolean.')
```

Note: The `print()` function should be indented within the `for` loop.

f) Run the code cell.

g) Examine the output.

```
Converted housing to Boolean.
Converted loan to Boolean.
Converted term_deposit to Boolean.
```

The `housing`, `loan`, and `term_deposit` variables have also been converted.

h) Select the next code cell, then type the following:

```
users_cleaned_1.info()
```

i) Run the code cell.

j) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 45214 entries, 0 to 45215
Data columns (total 19 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   user_id              45214 non-null  object
 1   number_transactions  35215 non-null  float64
 2   total_amount_usd     35215 non-null  float64
 3   age                  45214 non-null  int64
 4   job                  44926 non-null  object
 5   marital              45214 non-null  object
 6   education            43357 non-null  object
 7   default              45214 non-null  bool
 8   housing              45214 non-null  bool
 9   loan                 45214 non-null  bool
 10  contact              32196 non-null  object
 11  duration             45214 non-null  int64
 12  campaign             45214 non-null  int64
 13  pdays                45214 non-null  int64
 14  previous             45214 non-null  int64
 15  poutcome             8255 non-null   object
 16  term_deposit         45214 non-null  bool
 17  date_joined          45184 non-null  object
 18  device               45120 non-null  object
dtypes: bool(4), float64(2), int64(5), object(8)
memory usage: 5.7+ MB
```

As you can see, all four variables are now of the Boolean (`bool`) data type.

4. Convert `date_joined` to a datetime format.

a) Scroll down and view the cell titled **Convert `date_joined` to a datetime format**, then select the code cell below it.

b) In the code cell, type the following:

```
users_cleaned_2 = users_cleaned_1.copy()  # Work with a new object.

users_cleaned_2['date_joined'] = \
pd.to_datetime(users_cleaned_2['date_joined'],
               format = '%Y-%m-%d')
```

The `to_datetime()` function will convert the `date_joined` column. The `format` argument specifies that it will follow the YYYY-MM-DD format, which matches how it appeared when it was a string object.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
users_cleaned_2.info()
```

e) Run the code cell.

f) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 45214 entries, 0 to 45215
Data columns (total 19 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   user_id              45214 non-null  object
 1   number_transactions  35215 non-null  float64
 2   total_amount_usd     35215 non-null  float64
 3   age                  45214 non-null  int64
 4   job                  44926 non-null  object
 5   marital              45214 non-null  object
 6   education            43357 non-null  object
 7   default              45214 non-null  bool
 8   housing              45214 non-null  bool
 9   loan                 45214 non-null  bool
 10  contact              32196 non-null  object
 11  duration             45214 non-null  int64
 12  campaign             45214 non-null  int64
 13  pdays                45214 non-null  int64
 14  previous             45214 non-null  int64
 15  poutcome             8255 non-null   object
 16  term_deposit         45214 non-null  bool
 17  date_joined          45184 non-null  datetime64[ns]
 18  device               45120 non-null  object
dtypes: bool(4), datetime64[ns](1), float64(2), int64(5), object(7)
memory usage: 5.7+ MB
```

The `date_joined` column is now in a datetime format.

5. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 2-6
Deduplicating Data

Data File

~/ETL/Extracting, Transforming, and Loading Data.ipynb

Scenario

Another issue that plagues relatively large datasets is the presence of duplicates, and the GCNB data is no different. You'll identify any potentially duplicated rows and then remove them from the dataset. That way, the analysis and modeling you eventually perform won't be skewed by repeated data.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **ETL/Extracting, Transforming, and Loading Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Identify all duplicated data** heading, then select **Cell→Run All Above**.

2. Identify all duplicated data.
   a) Scroll down and view the cell titled **Identify all duplicated data**, then select the code cell below it.
   b) In the code cell, type the following:

```
duplicated_data = \
users_cleaned_2[users_cleaned_2.duplicated(keep = False)]

print('Number of rows with duplicated data:',
      duplicated_data.shape[0])
```

   You're using the `duplicated()` function to find any duplicates in the current dataset.
   c) Run the code cell.
   d) Examine the output.

```
Number of rows with duplicated data: 10
```

   There are 10 rows with duplicated data.
   e) Select the next code cell, then type the following:

```
duplicated_data
```

   f) Run the code cell.

g) Examine the output.

	user_id	number_transactions	total_amount_usd	age	job	marital	education	default	housing	loan	contact	duration
15456	cba59442-af3c-41d7-a39c-0f9bffba0660	2.0	1218.0	57	management	married	tertiary	True	True	False	cellular	317
15457	cba59442-af3c-41d7-a39c-0f9bffba0660	2.0	1218.0	57	management	married	tertiary	True	True	False	cellular	317
22006	1e826721-b38c-41c2-88f4-4c28b335b1e6	4.0	159.0	31	technician	single	secondary	False	False	False	cellular	129
22007	1e826721-b38c-41c2-88f4-4c28b335b1e6	4.0	159.0	31	technician	single	secondary	False	False	False	cellular	129
35415	a2fb8264-d55a-437b-a8e7-9ec4116b76f4	2.0	676.0	34	management	married	tertiary	False	False	False	cellular	156
35416	a2fb8264-d55a-437b-a8e7-9ec4116b76f4	2.0	676.0	34	management	married	tertiary	False	False	False	cellular	156
35623	f49ac08f-b872-4d57-ac82-9b8a9144020d	4.0	117.0	38	blue-collar	married	secondary	False	True	False	cellular	54
35624	f49ac08f-b872-4d57-ac82-9b8a9144020d	4.0	117.0	38	blue-collar	married	secondary	False	True	False	cellular	54
36296	ae3b92a2-cad8-434f-8037-9815e2228839	2.0	426.0	43	admin.	single	secondary	False	True	False	cellular	76
36297	ae3b92a2-cad8-434f-8037-9815e2228839	2.0	426.0	43	admin.	single	secondary	False	True	False	cellular	76

You can see the 10 duplicated rows, where each unique row appears to have one duplicate. So, there are actually five rows that can be removed.

3. Remove the duplicated data.

a) Scroll down and view the cell titled **Remove the duplicated data**, then select the code cell below it.

b) In the code cell, type the following:

```
users_cleaned_final = \
users_cleaned_2[~users_cleaned_2.duplicated()]

users_cleaned_final[users_cleaned_final['user_id'] == \
'cba59442-af3c-41d7-a39c-0f9bffba0660']
```

This will create a new `DataFrame` equal to the existing one, except for the identified duplicates. Also, you'll retrieve a specific user ID that was duplicated before to make sure it only appears once in the new dataset.

c) Run the code cell.

d) Examine the output.

	user_id	number_transactions	total_amount_usd	age	job	marital	education	default	housing	loan	contact	duration	campaign
15456	cba59442-af3c-41d7-a39c-0f9bffba0660	2.0	1218.0	57	management	married	tertiary	True	True	False	cellular	317	6

This particular user ID only appears once, so it seems the duplicates have been successfully removed.

e) Select the next code cell, then type the following:

```
users_cleaned_final.shape
```

f) Run the code cell.

g) Examine the output.

```
(45209, 19)
```

As expected, five rows have been removed from the overall dataset.

4. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.
b) Close the lab browser tab and continue on with the course.

LAB 2-7
Handling Textual Data

Data File

~/Text/Handling Textual Data.ipynb

~/Text/data/consumer_loan_complaints.csv

Scenario

The consumer complaints data that you worked with earlier may still be of value to the project, especially if the team plans to develop natural language processing (NLP) models sometime in the future. However, handling the textual data inside this file requires a much different approach than the numeric and categorical data that you've mostly been working with. You need a way to process the text so that it's more conducive to analysis and machine learning. There are many techniques for doing so, and you'll apply several of them to the complaint data.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Text/Handling Textual Data.ipynb** to open it.

2. Import the relevant software libraries.
   a) View the cell titled **Import software libraries**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.
   c) Verify that the version of Python is displayed, as are the versions of the other libraries that were imported.

3. Read and preview the text data.
   a) Scroll down and view the cell titled **Read and preview the text data**, then select the code cell below it.
   b) In the code cell, type the following:

```
complaints_data = pd.read_csv('data/consumer_loan_complaints.csv')

complaints_data.head()
```

   You'll load the data and get another look at its structure.
   c) Run the code cell.

d) Examine the output.

	user_id	Date received	Product	Issue	Consumer complaint narrative	State	ZIP code	Submitted via	Date sent to company	Company response to consumer	Timely response?	Consumer disputed?	Complaint ID
0	44fefdad-7045-4be5-890e-12e84ae6fdc9	01/27/2016	Consumer Loan	Account terms and changes	NaN	AL	35180	Phone	01/27/2016	Closed with explanation	Yes	No	1760486
1	c49d5d60-909f-406b-b7ff-51143fcb650b	08/26/2014	Consumer Loan	Account terms and changes	NaN	NC	278XX	Phone	08/29/2014	Closed with non-monetary relief	Yes	No	1001740
2	9b2cd5d2-900e-4052-831f-6489f6d568af	08/22/2012	Consumer Loan	Account terms and changes	NaN	TN	37205	Referral	08/23/2012	Closed with non-monetary relief	Yes	No	140039
3	b7e5b324-268e-4502-81a1-1a025673c2a0	05/07/2013	Consumer Loan	Problems when you are unable to pay	NaN	OH	43081	Web	05/08/2013	Closed with explanation	Yes	Yes	401541
4	684eeb4c-c9c3-4a97-8213-f3962a6c0aba	06/15/2016	Consumer Loan	Managing the line of credit	NaN	NC	27216	Phone	09/08/2016	Closed with non-monetary relief	Yes	No	1970341

Recall that each user that filed a complaint is listed in a row, and each column is a different aspect of the user or their complaint.

4. Check the shape of the data.

a) Scroll down and view the cell titled **Check the shape of the data**, then select the code cell below it.

b) In the code cell, type the following:

```
1 complaints_data.shape
```

c) Run the code cell.

d) Examine the output.

```
(1824, 13)
```

The dataset has 1,824 rows and 13 columns.

5. Retrieve information about the data.

a) Scroll down and view the cell titled **Retrieve information about the data**, then select the code cell below it.

b) In the code cell, type the following:

```
1 complaints_data.info()
```

c) Run the code cell.

d) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1824 entries, 0 to 1823
Data columns (total 13 columns):
 #   Column                         Non-Null Count  Dtype
---  ------                         --------------  -----
 0   user_id                        1824 non-null   object
 1   Date received                  1824 non-null   object
 2   Product                        1824 non-null   object
 3   Issue                          1824 non-null   object
 4   Consumer complaint narrative   44 non-null     object
 5   State                          1801 non-null   object
 6   ZIP code                       1789 non-null   object
 7   Submitted via                  1824 non-null   object
 8   Date sent to company           1824 non-null   object
 9   Company response to consumer   1824 non-null   object
 10  Timely response?               1824 non-null   object
 11  Consumer disputed?             1824 non-null   object
 12  Complaint ID                   1824 non-null   int64
dtypes: int64(1), object(12)
memory usage: 185.4+ KB
```

There are only 44 values in the `Consumer complaint narrative` feature that aren't null. In other words, most of this data is missing.

e) Select the next code cell, then type the following:

```
1 complaints_data.Issue.value_counts()
```

f) Run the code cell.

g) Examine the output.

```
Managing the line of credit           806
Account terms and changes             484
Shopping for a line of credit         301
Problems when you are unable to pay   233
Name: Issue, dtype: int64
```

There are several different types of issues, which makes this a good candidate for encoding, so there's no need to perform any textual processing on this feature.

h) Select the next code cell, then type the following:

```
1 complaints_data['Company response to consumer'].value_counts()
```

i) Run the code cell.

j) Examine the output.

```
Closed with explanation            1291
Closed with non-monetary relief     184
Closed with monetary relief         182
Closed without relief                75
Closed                               65
Closed with relief                   19
Untimely response                     8
Name: Company response to consumer, dtype: int64
```

Likewise, this feature can be encoded since it's just categorical in nature. Ultimately, the `Consumer complaint narrative` feature is the only one that needs special handling.

6. Extract a subset of data to consider only consumer complaints.

a) Scroll down and view the cell titled **Extract a subset of data to consider only consumer complaints**, then select the code cell below it.

b) In the code cell, type the following:

```
print('Number of users with no complaints data:',
      complaints_data['Consumer complaint narrative'].isnull().sum())
```

c) Run the code cell.

d) Examine the output.

```
Number of users with no complaints data: 1780
```

As expected, most records weren't filed with the actual text of the user's complaint.

e) Select the next code cell, then type the following:

```
# Remove records with missing complaint narratives.

text_data = complaints_data[~complaints_data \
                            ['Consumer complaint narrative'].isnull()] \
                           [['user_id', 'Consumer complaint narrative']]

text_data.head(n = 3)
```

You just need to focus on the records that actually have a complaint narrative, so you'll reduce the dataset.

Note: The first character within the bracket is a tilde (~).

f) Run the code cell.

g) Examine the output.

	user_id	Consumer complaint narrative
53	1a1448a4-bfe5-455f-bc29-dc79ec5fb2c0	NONE OF YOUR " MY LOAN IS A " below apply to ...
59	5fede48c-096e-4f82-997d-8229007d8318	XX/XX/2014 I received a letter from the IRS st...
65	fd9fc5ff-19bc-424c-880e-c159c110d21f	This was a revolving account in which I paid W...

The reduced dataset has the user's ID and the text of their complaint.

h) Select the next code cell, then type the following:

```
text_data.shape
```

i) Run the code cell.

j) Examine the output.

```
(44, 2)
```

As expected, there are 44 rows and 2 columns.

7. Preview an example of the consumer complaints.

a) Scroll down and view the cell titled **Preview an example of the consumer complaints**, then select the code cell below it.
b) In the code cell, type the following:

```
sample_text = text_data['Consumer complaint narrative'].iloc[0]
sample_text
```

This code will print the first complaint text.
c) Run the code cell.
d) Examine the output.

```
'NONE OF YOUR " MY LOAN IS A \'\' below apply to this situation! This was a car loan but the company is providing
fraudulent information this is damaging my credit! \n\nRE : MidAtlantic Finance Company Account No. XXXX - NOT TO
BE CONFUSED with my current MAF loan MidAtlantic Finance Company has reported several false items to all XXXX cred
it reporting agencies, and continues to do so. It is damaging my credit so much so that I was told I did n\'t qual
ify for a mortgage. \n\nMost recently, I settled this account per agreement on XXXX XXXX, XXXX, yet MAF reported i
t is a payment on the amount claimed owed ( which has been disputed since XXXX XXXX ). But that is just the most r
ecent false information that was reported. It is showing a debt of {$250.00} per month along with XXXX different a
mounts charged off of the {$950.00} ( plus interest ) and another {$5100.00} that INCLUDES the {$950.00}. Please r
efer to the following as I NEVER owed MAF {$8100.00} as it reported. That was the original amount financed in XXXX
XXXX with XXXX XXXX, and I made payments of {$250.00} a month through XXXX XXXX to XXXX XXXX. \n1 ) Car was purcha
sed in XXXX XXXX and was financed IN HOUSE until XXXX XXXX per MAF \'s statement of XXXX XXXX, XXXX, with my first
payment being due to MAF XXXX XXXX. Therefore, I was not responsible for any prior late payments ( MAF recorded my
first delinquency as XXXX XXXX ) and that should never have been on my credit report. I do not know why or HOW it
can be considered owned by MAF since XXXX XXXX . \n2 ) There was NEVER a charge off of {$5100.00} for the following
```

The complaint is full of characters like punctuation, numbers, and common words. You'll need to streamline this text to make it more amenable to analysis.

8. Tokenize the sample text into sentences.

a) Scroll down and view the cell titled **Tokenize the sample text into sentences**, then select the code cell below it.
b) In the code cell, type the following:

```
nlp = spacy.load('/home/jovyan/work/spacy_data/' +
                 'en_core_web_sm/en_core_web_sm-3.0.0/')

document = nlp(sample_text)
```

This code uses a third-party text processing library called spaCy to create a document from the sample text.
c) Run the code cell.
d) Select the next code cell, then type the following:

```
for sentence in document.sents:
    print(sentence)
```

This code will tokenize the entire document text into sentences.
e) Run the code cell.

f) Examine the output.

```
NONE OF YOUR " MY LOAN IS A '' below apply to this situation!
This was a car loan but the company is providing fraudulent information this is damaging my credit!


RE : MidAtlantic Finance Company Account
No.
XXXX - NOT TO BE CONFUSED with my current MAF loan MidAtlantic Finance Company has reported several false items to
all XXXX credit reporting agencies, and continues to do so.
It is damaging my credit so much so that I was told I did n't qualify for a mortgage.


Most recently, I settled this account per agreement on XXXX XXXX, XXXX, yet MAF reported it is a payment on the am
ount claimed owed ( which has been disputed since XXXX XXXX ).
But that is just the most recent false information that was reported.
It is showing a debt of {$250.00} per month along with XXXX different amounts charged off of the {$950.00} ( plus
```

Each sentence identified by spaCy is printed. Note that the line breaks (`\n` characters in the document) are being treated as different sentences.

9. Tokenize the sentences into words.

a) Scroll down and view the cell titled **Tokenize the sentences into words**, then select the code cell below it.

b) In the code cell, examine the following:

```
sentence = nlp('It is showing a debt of {$250.00} per month along ' \
               'with XXXX different amounts charged off of the ' \
               '{$950.00} ( plus interest ) and another {$5100.00} ' \
               'that INCLUDES the {$950.00}.')
```

To demonstrate the tokenization of individual words, this code is taking a single sentence from the overall document.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
for token in sentence:
    print(token.text)
```

e) Run the code cell.

f) Examine the output.

```
It
is
showing
a
debt
of
{
$
250.00
}
per
month
along
with
XXXX
different
```

Each individual "word" (as defined by spaCy) is printed. Note that numbers and punctuation are counted as words.

10. Identify the parts of speech for each token.

a) Scroll down and view the cell titled **Identify the parts of speech for each token**, then select the code cell below it.
b) In the code cell, type the following:

```
pos = []

for token in sentence:
    pos.append({'Word': token,
                'Part of Speech': token.pos_
               })

pd.DataFrame(pos)
```

Text processors like spaCy can actually identify each token's role in a sentence, including non-words like numbers and punctuation.

c) Run the code cell.
d) Examine the output.

	Word	Part of Speech
0	It	PRON
1	is	AUX
2	showing	VERB
3	a	DET
4	debt	NOUN
5	of	ADP
6	{	PUNCT
7	$	SYM
8	250.00	NUM
9	}	PUNCT
10	per	ADP

The table lists what part of speech each token is. For example, `It` is a pronoun, `is` is an auxiliary verb (a verb that affects the tenses and moods of other verbs), `{` is punctuation, `250.00` is a number, and so on.

11. Identify stop words.

a) Scroll down and view the cell titled **Identify stop words**, then select the code cell below it.
b) In the code cell, type the following:

```
stop = []

for token in sentence:
    stop.append({'Word': token,
                 'Stop Word?': token.is_stop
                })

pd.DataFrame(stop)
```

This code will identify whether or not a token qualifies as a stop word, or a word that is so common that it should be excluded from the text so as not to exert undue influence.

c) Run the code cell.

d) Examine the output.

	Word	Stop Word?
0	It	True
1	is	True
2	showing	False
3	a	True
4	debt	False
5	of	True
6	{	False
7	$	False
8	250.00	False
9	}	False

Words like `It` and `is` are considered stop words, whereas `showing` is not.

12. Stem the text.

a) Scroll down and view the cell titled **Stem the text**, then select the code cell below it.

b) In the code cell, examine the following:

```
text = 'This was a car loan but the company is providing ' \
       'fraudulent information this is damaging my credit!'

print(word_tokenize(text))
```

This is another sample sentence from the larger document. You're going to be using the Natural Language Toolkit (NLTK) to perform stemming on this tokenized sample. Stemming obtains the affix of a word to reduce that word to a base form, making it easier to work with.

c) Run the code cell.

d) Examine the output.

```
['This', 'was', 'a', 'car', 'loan', 'but', 'the', 'company', 'is', 'providing', 'fraudulent', 'information', 'this
', 'is', 'damaging', 'my', 'credit', '!']
```

Each word is an item in a list.

e) Select the next code cell, then type the following:

```
stemmer = SnowballStemmer(language = 'english')

for token in word_tokenize(text):
    print(token, '-->' , stemmer.stem(token))
```

There are multiple stemming algorithms in NLTK, and each one can produce different results. In this case you're using `SnowballStemmer()` to see how it does.

f) Run the code cell.

g) Examine the output.

```
This --> this
was --> was
a --> a
car --> car
loan --> loan
but --> but
the --> the
company --> compani
is --> is
providing --> provid
fraudulent --> fraudul
information --> inform
this --> this
is --> is
damaging --> damag
my --> my
credit --> credit
! --> !
```

You can see that some tokens were properly stemmed, but that the resulting stem is not quite a real word. For example, `company` became `compani`, and `damaging` became `damag`. That's why a more advanced technique like lemmatization is usually preferred.

13. Lemmatize the text.

a) Scroll down and view the cell titled **Lemmatize the text**, then select the code cell below it.

b) In the code cell, type the following:

```
parsed_text = nlp(text)

for token in parsed_text:
    print(token, '-->', token.lemma_)
```

This code uses the same sentence sample, but this time retrieves each word's lemma. A lemma is the canonical dictionary form of a word.

c) Run the code cell.

d) Examine the output.

```
This --> this
was --> be
a --> a
car --> car
loan --> loan
but --> but
the --> the
company --> company
is --> be
providing --> provide
fraudulent --> fraudulent
information --> information
this --> this
is --> be
damaging --> damage
my --> my
credit --> credit
! --> !
```

The lemmatization did a better job of obtaining root words. For example, `company` didn't change because it's already in the canonical form, and `damaging` became `damage`, which is the canonical form. Also, verbs like `was` and `is` became `be`.

14. Transform the text.

a) Scroll down and view the cell titled **Transform the text**, then select the code cell below it.

b) In the code cell, examine the following:

```
def spacy_cleaner(original_text):
    """Cleans text data to be processed.
    Removes punctuation, whitespace, numbers, stopwords from the text
    and lemmatizes each token."""

    final_tokens = []
    parsed_text = nlp(original_text)

    for token in parsed_text:
        if token.is_punct or token.is_space or token.like_num or token.is_stop:
            pass
        else:
            if token.lemma_ == '-PRON-':
                final_tokens.append(str(token))  # Keep pronouns as they are.
            else:
                sc_removed = re.sub('[^a-zA-Z]', '', str(token.lemma_))
                if len(sc_removed) > 1:
                    final_tokens.append(sc_removed)
    joined = ' '.join(final_tokens)
    preprocessed_text = re.sub(r'(.)\1+', r'\1\1', joined)

    return preprocessed_text
```

Now it's time to actually transform the text data. This function will:

- Use spaCy to tokenize the text.
- Remove punctuation, spaces, numbers, and stop words.
- Obtain the lemma of each word, except for pronouns, which will be kept as is.
- Replace each word with the lemma using a regular expression.
- Remove any other punctuation like parentheses and backslashes using a regular expression.
- Compile the results in a string.

c) Run the code ce l

d) Select the next code cell, then type the following:

```
# Apply transformation to sample.

spacy_cleaner(sample_text)
```

First you'll try the transformation out on the single complaint text example.

e) Run the code cell.

f) Examine the output.

```
'LOAN apply situation car loan company provide fraudulent information damage credit MidAtlantic Finance Company Ac
count xx confused current MAF loan MidAtlantic Finance Company report false item xx credit reporting agency contin
ue damage credit tell qualify mortgage recently settle account agreement XX XX XX MAF report payment claim owe dis
pute XX XX recent false information report show debt month xx different amount charge plus interest include refer
following owe MAF report original finance XX xx XX XX payment month XX xx xx XX car purchase XX XX finance HOUSE X
X XX MAF statement XX XX XX payment MAF XX XX responsible prior late payment MAF record delinquency xx XX credit r
eport know consider own MAF XX XX charge follow reason car total XX XX XX pay xx payment plus additional interest
fee XX XX XX insurance company pay XX XX XX leave balance MAF dispute XX XX give payoff XX xx expiration date disp
ute month give finally MAF send accounting XX xx support claim payment wrongfully charge amount XX xx XX account p
```

As expected, the text of the review has been streamlined.

g) Select the next code cell, then type the following:

```
# Compare to sample before transformation.

sample_text
```

h) Run the code cell.

i) Examine the output.

```
'NONE OF YOUR " MY LOAN IS A \'\' below apply to this situation! This was a car loan but the company is providing
fraudulent information this is damaging my credit! \n\nRE : MidAtlantic Finance Company Account No. XXXX - NOT TO
BE CONFUSED with my current MAF loan MidAtlantic Finance Company has reported several false items to all XXXX cred
it reporting agencies, and continues to do so. It is damaging my credit so much so that I was told I did n\'t qual
ify for a mortgage. \n\nMost recently, I settled this account per agreement on XXXX XXXX, XXXX, yet MAF reported i
t is a payment on the amount claimed owed ( which has been disputed since XXXX XXXX ). But that is just the most r
ecent false information that was reported. It is showing a debt of {$250.00} per month along with XXXX different a
mounts charged off of the {$950.00} ( plus interest ) and another {$5100.00} that INCLUDES the {$950.00}. Please r
efer to the following as I NEVER owed MAF {$8100.00} as it reported. That was the original amount financed in XXXX
XXXX with XXXX XXXX, and I made payments of {$250.00} a month through XXXX XXXX to XXXX XXXX. \n1 ) Car was purcha
```

You can see significant differences between the original form of the complaint and the complaint after it underwent text processing.

j) Select the next code cell, then type the following:

```
# Apply transformation to entire dataset.

text_data['consumer_complaints_cleaned'] = \
text_data['Consumer complaint narrative'].apply(lambda x: spacy_cleaner(x))

text_data.head(n = 3)
```

You'll apply the transformation to the entire dataset.

k) Run the code cell.

Note: This may take a few moments to execute.

l) Examine the output.

	user_id	Consumer complaint narrative	consumer_complaints_cleaned
53	1a1448a4-bfe5-455f-bc29-dc79ec5fb2c0	NONE OF YOUR " MY LOAN IS A " below apply to ...	LOAN apply situation car loan company provide ...
59	5fede48c-096e-4f82-997d-8229007d8318	XX/XX/2014 I received a letter from the IRS st...	XX XX receive letter IRS state owe agency ask ...
65	fd9fc5ff-19bc-424c-880e-c159c110d21f	This was a revolving account in which I paid W...	revolving account pay Wells Fargo National Ban...

The first three records are printed, as is the original narrative and the transformed narrative for each complaint.

15. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on with the course.

MODULE 3

Load Data

The following labs are for Module 3: Load Data.

LAB 2-8
Loading Data into a Database

Data File

~/ETL/Extracting, Transforming, and Loading Data.ipynb

Scenario

Now that the GCNB data is in a relatively clean state, you want to begin packaging it for the forthcoming analysis and modeling tasks. There are many formats that you can load this data into; instead of choosing just one, you'll try out multiple formats to get a feel for how they differ in terms of storage and integration. First, you'll load the dataset into an SQL database.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **ETL/Extracting, Transforming, and Loading Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Load data into an SQL database** heading, then select **Cell→Run All Above**.

2. Load data into an SQL database.
   a) Scroll down and view the cell titled **Load data into an SQL database**, then select the code cell below it.
   b) In the code cell, type the following:

```
conn = sqlite3.connect('users_data_cleaned.db')

users_cleaned_final.to_sql('users_cleaned_final',
                           conn,
                           if_exists = 'replace',
                           index = False)
```

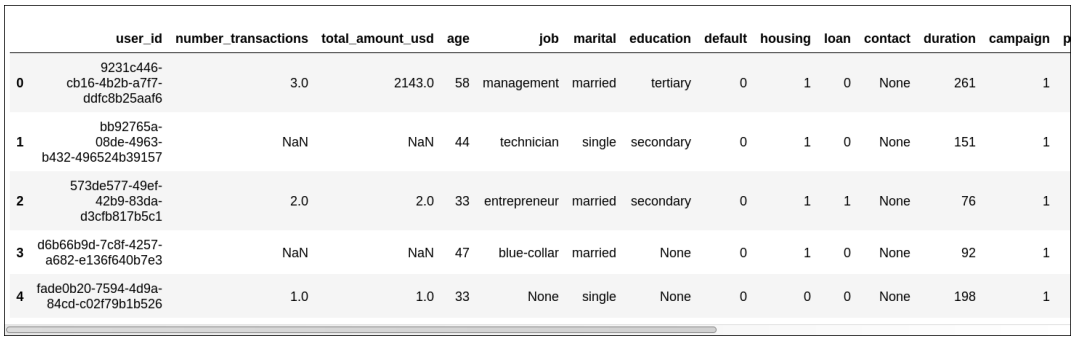
   First, you're connecting to an SQLite database file that doesn't yet exist, but will be the save point for the dataset. Then, the `to_sql()` function actually saves the data to the open database file.
   c) Run the code cell.

3. Confirm that data was loaded into the database.
   a) Scroll down and view the cell titled **Confirm that data was loaded into the database**, then select the code cell below it.
   b) In the code cell, type the following:

```
query = 'SELECT * FROM users_cleaned_final'

pd.read_sql(query, conn).head()
```

   c) Run the code cell.

d) Examine the output.



	user_id	number_transactions	total_amount_usd	age	job	marital	education	default	housing	loan	contact	duration	campaign	p
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	3.0	2143.0	58	management	married	tertiary	0	1	0	None	261	1	
1	bb92765a-08de-4963-b432-496524b39157	NaN	NaN	44	technician	single	secondary	0	1	0	None	151	1	
2	573de577-49ef-42b9-83da-d3cfb817b5c1	2.0	2.0	33	entrepreneur	married	secondary	0	1	1	None	76	1	
3	d6b66b9d-7c8f-4257-a682-e136f640b7e3	NaN	NaN	47	blue-collar	married	None	0	1	0	None	92	1	
4	fade0b20-7594-4d9a-84cd-c02f79b1b526	1.0	1.0	33	None	single	None	0	0	0	None	198	1	

The first few rows of the cleaned dataset appear, indicating that the data was successfully loaded into the SQLite database.

4. Close the database connection.

a) Scroll down and view the cell titled **Close the database connection**, then select the code cell below it.

b) In the code cell, type the following:

```
1 conn.close()
```

c) Run the code cell.

5. Confirm the database file was saved to the local drive.

a) In Firefox, select the **CDSP/ETL/** tab to view the file structure.

b) Verify that a file called **users_data_cleaned.db** exists.

You can't open this file directly, but you already confirmed that you were able to load it into Jupyter Notebook.

6. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 2-9
Loading Data into a DataFrame

Data File

~/ETL/Extracting, Transforming, and Loading Data.ipynb

Scenario

SQL databases are common storage platforms for relational datasets, but they aren't the only option available to you. In fact, if you plan to do a lot of work in a programming language like Python, you may want to preserve your data in a `DataFrame` structure so that it's easier to read from and write to. This is possible by saving the `DataFrame` as a binary pickle file, which you'll do in this lab.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **ETL/Extracting, Transforming, and Loading Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Write the `DataFrame` as a pickle file** heading, then select **Cell→Run All Above**.

2. Write the `DataFrame` as a pickle file.
   a) Scroll down and view the cell titled **Write the `DataFrame` as a pickle file**, then select the code cell below it.
   b) In the code cell, type the following:

```
1 users_cleaned_final.to_pickle('users_data_cleaned.pickle')
```

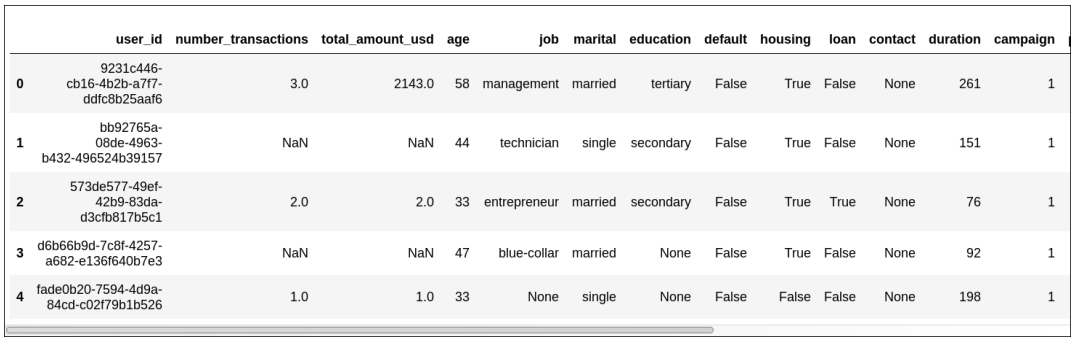
   c) Run the code cell.
   Pickle files are in a binary format and preserve the content and structure of a `DataFrame`.

3. Confirm that the data was written to the pickle file.
   a) Scroll down and view the cell titled **Confirm that the data was written to the pickle file**, then select the code cell below it.
   b) In the code cell, type the following:

```
1 pd.read_pickle('users_data_cleaned.pickle').head()
```

   c) Run the code cell.

d) Examine the output.



	user_id	number_transactions	total_amount_usd	age	job	marital	education	default	housing	loan	contact	duration	campaign
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	3.0	2143.0	58	management	married	tertiary	False	True	False	None	261	1
1	bb92765a-08de-4963-b432-496524b39157	NaN	NaN	44	technician	single	secondary	False	True	False	None	151	1
2	573de577-49ef-42b9-83da-d3cfb817b5c1	2.0	2.0	33	entrepreneur	married	secondary	False	True	True	None	76	1
3	d6b66b9d-7c8f-4257-a682-e136f640b7e3	NaN	NaN	47	blue-collar	married	None	False	True	False	None	92	1
4	fade0b20-7594-4d9a-84cd-c02f79b1b526	1.0	1.0	33	None	single	None	False	False	False	None	198	1

As expected, the `DataFrame` was loaded from the pickle file with its structure intact.

e) Select the next code cell, then type the following:

```
1 pd.read_pickle('users_data_cleaned.pickle').info()
```

f) Run the code cell.

g) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 45209 entries, 0 to 45215
Data columns (total 19 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   user_id              45209 non-null  object
 1   number_transactions  35210 non-null  float64
 2   total_amount_usd     35210 non-null  float64
 3   age                  45209 non-null  int64
 4   job                  44921 non-null  object
 5   marital              45209 non-null  object
 6   education            43352 non-null  object
 7   default              45209 non-null  bool
 8   housing              45209 non-null  bool
 9   loan                 45209 non-null  bool
 10  contact              32191 non-null  object
 11  duration             45209 non-null  int64
 12  campaign             45209 non-null  int64
 13  pdays                45209 non-null  int64
 14  previous             45209 non-null  int64
 15  poutcome             8252 non-null   object
 16  term_deposit         45209 non-null  bool
 17  date_joined          45179 non-null  datetime64[ns]
 18  device               45115 non-null  object
dtypes: bool(4), datetime64[ns](1), float64(2), int64(5), object(7)
memory usage: 5.7+ MB
```

As you can see, the `date_joined` column is still in datetime format. No additional parsing or querying is necessary when loading a `DataFrame` directly from a pickle file.

4. Confirm the pickle file was saved to the local drive.

a) In Firefox, select the **CDSP/ETL/** tab to view the file structure.

b) Verify that a file called **users_data_cleaned.pickle** exists.

You can't open this file directly, but you already confirmed that you were able to load it into Jupyter Notebook.

5. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.
b) Close the lab browser tab and continue on with the course.

LAB 2-10
Exporting Data to a CSV File

Data File

~/ETL/Extracting, Transforming, and Loading Data.ipynb

Scenario

Some of the other team members on the project don't use Python and aren't particularly comfortable with SQL. But they still need a way to work on the same GCNB data that you've been preparing. So, you'll export the dataset as a comma-separated values (CSV) file, which preserves the data and its general structure in a text file. That way they can more easily integrate the data into whatever environments they are using.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **ETL/Extracting, Transforming, and Loading Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Write the data to a CSV file** heading, then select **Cell→Run All Above**.

2. Write the data to a CSV file.
   a) Scroll down and view the cell titled **Write the data to a CSV file**, then select the code cell below it.
   b) In the code cell, type the following:

```
users_cleaned_final.to_csv('users_data_cleaned.csv',
                           index = False)
```

   c) Run the code cell.

   Saving to a CSV will preserve the content of the `DataFrame`, but not the structure. However, this makes the data more extensible, as not every tool can read binary `DataFrame` objects, whereas most can read text-based CSV files.

3. Confirm that the data was written to a CSV file.
   a) Scroll down and view the cell titled **Confirm that the data was written to a CSV file**, then select the code cell below it.
   b) In the code cell, type the following:

```
pd.read_csv('users_data_cleaned.csv').head()
```

   c) Run the code cell.

d) Examine the output.

	user_id	number_transactions	total_amount_usd	age	job	marital	education	default	housing	loan	contact	duration	campaign
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	3.0	2143.0	58	management	married	tertiary	False	True	False	NaN	261	1
1	bb92765a-08de-4963-b432-496524b39157	NaN	NaN	44	technician	single	secondary	False	True	False	NaN	151	1
2	573de577-49ef-42b9-83da-d3cfb817b5c1	2.0	2.0	33	entrepreneur	married	secondary	False	True	True	NaN	76	1
3	d6b66b9d-7c8f-4257-a682-e136f640b7e3	NaN	NaN	47	blue-collar	married	NaN	False	True	False	NaN	92	1
4	fade0b20-7594-4d9a-84cd-c02f79b1b526	1.0	1.0	33	NaN	single	NaN	False	False	False	NaN	198	1

The structure of the `DataFrame` appears to be intact, despite loading from a text file. However, there may be some more subtle changes that will require you to parse the data in order to get it into an ideal state.

e) Select the next code cell, then type the following:

```
1 pd.read_csv('users_data_cleaned.csv').info()
```

f) Run the code cell.

g) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45209 entries, 0 to 45208
Data columns (total 19 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   user_id              45209 non-null  object
 1   number_transactions  35210 non-null  float64
 2   total_amount_usd     35210 non-null  float64
 3   age                  45209 non-null  int64
 4   job                  44921 non-null  object
 5   marital              45209 non-null  object
 6   education            43352 non-null  object
 7   default              45209 non-null  bool
 8   housing              45209 non-null  bool
 9   loan                 45209 non-null  bool
 10  contact              32191 non-null  object
 11  duration             45209 non-null  int64
 12  campaign             45209 non-null  int64
 13  pdays                45209 non-null  int64
 14  previous             45209 non-null  int64
 15  poutcome             8252 non-null   object
 16  term_deposit         45209 non-null  bool
 17  date_joined          45179 non-null  object
 18  device               45115 non-null  object
dtypes: bool(4), float64(2), int64(5), object(8)
memory usage: 5.3+ MB
```

As you can see, the `date_joined` column is now in a plain string format instead of datetime. So, keep in mind that text files may require additional parsing when you load them back into your programming environment.

4. Confirm the CSV file was saved to the local drive.

a) In Firefox, select the **CDSP/ETL/** tab to view the file structure.

b) Verify that a file called **users_data_cleaned.csv** exists.

c) Select the file name to open it.

d) Verify that you can see all of the data in a textual format, separated by commas.
e) When you're done, close this tab and return to the active notebook.

5. Shut down the notebook and close the lab.
   a) From the menu, select **Kernel→Shutdown**.
   b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
   c) Close the lab browser tab and continue on with the course.

MODULE 4

Project

The following lab is for the Course 2 Project.

PROJECT 2

Online Retailer: Extracting, Transforming, and Loading Data

Data Files

~/Project/ETL Project.ipynb

~/Project/data/prod_sample.db

Scenario

You work for an online retailer that sells a wide variety of different items to consumers. Each sale through the online storefront is recorded in a database with various characteristics, including invoice number, product stock code, quantity purchased, sale price, description, etc. As part of your data science business project, your team has been tasked with creating models that can give the business more insights into its customers' behavior, and even predict future purchasing decisions.

Before you can do that, however, you need to go through the preliminary ETL process to make sure your data is in an acceptable state.

	**Note:** There is no "correct" answer for every line of code, as there are many ways to write code to do the same thing. You can write the code however you feel most comfortable, as long as it accomplishes the tasks set out for you.
	**Note:** If you're stuck and need help, you can "borrow" code from existing notebooks, and/or look up documentation on how to use the various Python libraries.
	**Note:** Sometimes, when you run code containing logic errors or bugs, you may corrupt the data contained in variables you created in previous code cells. To clear out such problems, you can select **Kernel→Restart & Clear Output**, then run each code cell again.

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Project/ETL Project.ipynb** to open it.
   c) Observe the notebook.

      Placeholder code cells have been provided in which you can add your own code. Comments provide hints on tasks you might perform in each code cell. The first code cell has already been completely programmed for you.

2. Import software libraries.
   a) Select the code listing under **Import software libraries**.
   b) Run the code and examine the results.

3. Examine the database.
   a) In the first code cell under **Examine the database**, write code to connect to the **prod_sample.db** SQLite database in the **data** folder.
   b) Run the code cell.
   c) In the next code cell, write a query to list all tables in the database.

d) Run the code cell and verify that the `stock_description` and `online_retail_history` tables are listed.

4. Read data from the `online_retail_history` table.

a) In first code cell under **Read data from the `online_retail_history` table**, write and execute a query to select everything from the `online_retail_history` table, then get the first five rows.

b) Run the code cell and verify the first five rows of the table appear.

Note: This dataset comes from the UK, where "stock" is equivalent to "inventory" in American English. The stock code is the item number/identifier and is not to be confused with "stock" in the sense of a financial share of a company.

c) In the next code cell, get the shape of the table.

d) Run the code cell and verify the number of rows and columns in the table.

5. Read data from the `stock_description` table.

a) In first code cell under **Read data from the `stock_description` table**, write and execute a query to select everything from the `stock_description` table, then get the first five rows.

b) Run the code cell and verify the first five rows of the table appear.

c) In the next code cell, get the shape of the table.

d) Run the code cell and verify the number of rows and columns in the table.

6. Aggregate the `online_retail_history` and `stock_description` datasets.

a) In the first code cell under **Aggregate the `online_retail_history` and `stock_description` datasets**, write a query to left join both tables so that the stock descriptions are in the same table as every other column from `online_retail_history`, then get the first five rows.

b) Run the code cell and verify the first five rows of the aggregated table appear.

c) In the next code cell, get the shape of the table.

d) Run the code cell and verify the number of rows and columns in the table.

Note: For now, the number of rows in this aggregate table should exceed the number of rows in `online_retail_history`.

7. Identify and fix corrupt or unusable data.

a) In the first code cell under **Identify and fix corrupt or unusable data**, write code to get the count of all distinct values in the `Description` field.

b) Run the code cell and verify that you are given the counts, and that one of the descriptions is just a question mark (`?`).

c) In the next code cell, write code to drop the rows where the description is just a question mark, then get the shape of the data and the first five rows.

d) Run the code cell and verify that the number of rows now matches the number initially in `online_retail_history`.

8. Identify and remove duplicates.

a) In the first code cell under **Identify and remove duplicates**, write code to identify the number of rows with duplicated data.

b) Run the code cell and verify the number of rows with duplicated data.

c) In the next code cell, print the first five rows of the duplicated data.

d) Run the code cell and verify the duplicated data table is listed.

e) In the next code cell, write code to remove the duplicated data rows, then print the first five rows of the new table.

f) Run the code cell and verify that the new table is printed.

9. Correct date formats.
   a) In the first code cell under **Correct date formats**, write code to get the data types for every column in the table.
   b) Run the code cell and verify that the `InvoiceDate` column is an object (string) type.
   c) In the next code, write code to convert `InvoiceDate` to a datetime object in the format `%Y-%m-%d`.
   d) Run the code cell.
   e) In the next code cell, get the column data types again.
   f) Run the code cell and verify that `InvoiceDate` is now a datetime object.

10. Examine the table before finishing.
   a) In the code cell under **Examine the table before finishing**, write code to print the first five records of the dataset that has undergone preliminary cleaning.
   b) Run the code cell observe the table that's ready to be loaded.

11. Load the dataset into a pickle file.
   a) In the first code cell under **Load the dataset into a pickle file**, write code to save the cleaned `DataFrame` into a pickle file.
   b) Run the code cell.
   c) In the next code cell, close the connection to the SQLite database.
   d) Run the code cell.

12. Save and download the notebook.
   a) From the menu, select **File→Save and Checkpoint**.
   b) Select **File→Download as→Notebook (.ipynb)**.
   c) Save the file to your local drive if it wasn't automatically.
   d) Find the saved file and rename it using the following convention: **FirstnameLastname-ETL-Project.ipynb**.

      For example: **JohnSmith-ETL-Project.ipynb**.

Note: Make sure not to lose this file. This is the project you'll be uploading and sharing to your peers for grading.

13. Shut down the notebook and close the lab.
   a) Back in Jupyter Notebook, select **Kernel→Shutdown**.
   b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
   c) Close the lab browser tab and continue on to the peer review.

Course 3: Analyze Data

Course Introduction

The following labs are for Course 3: Analyze Data.

Modules

The labs in this course pertain to the following modules:

- Module 1: Examine Data
- Module 2: Explore the Underlying Distribution of Data
- Module 3: Use Visualizations to Analyze Data
- Module 4: Preprocess Data
- Apply What You've Learned

MODULE 1

Examine Data

The following labs are for Module 1: Examine Data.

LAB 3-1
Examining Data

Data Files

~/Analysis/Analyzing Data.ipynb

~/Analysis/data/users_data_cleaned.pickle

Scenario

Now that you've gone through the ETL process on the GCNB users data at least once, you can start taking a closer look at the characteristics of that data. There's likely still many more ways the data can be improved before you use it to start building machine learning models. You'll begin your analysis by examining the data from a high level, just to get a sense of what state it's in and what you have to work with.

Note: By default, these lab steps assume you will be typing the code shown in screenshots. Many learners find it easier to understand what code does when they are able to type it themselves. However, if you prefer not to type all of code, the **Solutions** folders contain the finished code for each notebook.

---

1. Open the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Analysis**.
      The **Analysis** directory contains a subdirectory named **data** and a notebook file named **Analyzing Data.ipynb**.
   c) Select **Analyzing Data.ipynb** to open it.

2. Import the relevant software libraries.
   a) View the cell titled **Import software libraries**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.
   c) Verify that the version of Python is displayed, as are the versions of the other libraries that were imported.

3. Load and preview the data.
   a) Scroll down and view the cell titled **Load and preview the data**, then select the code cell below it.
   b) In the code cell, type the following:

```
1 users_data = pd.read_pickle('data/users_data_cleaned.pickle')
2
3 users_data.head(n = 5)
```

      You're loading the pickle file of the cleaned users data table you saved earlier.
   c) Run the code cell.

d) Examine the output.

	user_id	number_transactions	total_amount_usd	age	job	marital	education	default	housing	loan	contact	duration	campaign
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	3.0	2143.0	58	management	married	tertiary	False	True	False	None	261	1
1	bb92765a-08de-4963-b432-496524b39157	NaN	NaN	44	technician	single	secondary	False	True	False	None	151	1
2	573de577-49ef-42b9-83da-d3cfb817b5c1	2.0	2.0	33	entrepreneur	married	secondary	False	True	True	None	76	1
3	d6b66b9d-7c8f-4257-a682-e136f640b7e3	NaN	NaN	47	blue-collar	married	None	False	True	False	None	92	1
4	fade0b20-7594-4d9a-84cd-c02f79b1b526	1.0	1.0	33	None	single	None	False	False	False	None	198	1

This is an overview of all columns in the table, and the first five rows. You can see some of the types of values in each column, including `NaN` (missing data).

4. Check the shape of the data.

a) Scroll down and view the cell titled **Check the shape of the data**, then select the code cell below it.

b) In the code cell, type the following:

```
1 users_data.shape
```

c) Run the code cell.

d) Examine the output.

```
(45209, 19)
```

There are 45,209 rows and 19 columns in the dataset.

5. Check the number of unique users.

a) Scroll down and view the cell titled **Check the number of unique users**, then select the code cell below it.

b) In the code cell, type the following:

```
1 len(np.unique(users_data.user_id))
```

c) Run the code cell.

d) Examine the output.

```
45209
```

As expected, the number of unique users is the same as the number of rows.

6. Check the data types.

a) Scroll down and view the cell titled **Check the data types**, then select the code cell below it.

b) In the code cell, type the following:

```
1 users_data.info()
```

c) Run the code cell.

d) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 45209 entries, 0 to 45215
Data columns (total 19 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   user_id              45209 non-null  object
 1   number_transactions  35210 non-null  float64
 2   total_amount_usd     35210 non-null  float64
 3   age                  45209 non-null  int64
 4   job                  44921 non-null  object
 5   marital              45209 non-null  object
 6   education            43352 non-null  object
 7   default              45209 non-null  bool
 8   housing              45209 non-null  bool
 9   loan                 45209 non-null  bool
 10  contact              32191 non-null  object
 11  duration             45209 non-null  int64
 12  campaign             45209 non-null  int64
 13  pdays                45209 non-null  int64
 14  previous             45209 non-null  int64
 15  poutcome             8252 non-null   object
 16  term_deposit         45209 non-null  bool
 17  date_joined          45179 non-null  datetime64[ns]
 18  device               45115 non-null  object
dtypes: bool(4), datetime64[ns](1), float64(2), int64(5), object(7)
memory usage: 5.7+ MB
```

The data types for each column are as expected.

e) Select the next code cell, then type the following:

```
1 users_data.columns.to_series().groupby(users_data.dtypes).groups
```

f) Run the code cell.

g) Examine the output.

```
{bool: ['default', 'housing', 'loan', 'term_deposit'], int64: ['age', 'duration', 'campaign', 'pdays', 'previous
'], float64: ['number_transactions', 'total_amount_usd'], datetime64[ns]: ['date_joined'], object: ['user_id', 'jo
b', 'marital', 'education', 'contact', 'poutcome', 'device']}
```

Each column is grouped by data type.

7. Check for correlations.

a) Scroll down and view the cell titled **Check for correlations**, then select the code cell below it.

b) In the code cell, type the following:

```
1 users_data.corr().abs()
```

c) Run the code cell.

d) Examine the output.

	number_transactions	total_amount_usd	age	default	housing	loan	duration	campaign	pdays	previous	term_deposit
number_transactions	1.000000	0.163409	0.008813	0.138838	0.030429	0.075319	0.017220	0.026431	0.030751	0.023046	0.053390
total_amount_usd	0.163409	1.000000	0.095839	0.065390	0.066857	0.084526	0.022586	0.017274	0.006435	0.016952	0.050785
age	0.008813	0.095839	1.000000	0.017875	0.185552	0.015641	0.004645	0.004767	0.023745	0.001297	0.025168
default	0.138838	0.065390	0.017875	1.000000	0.006020	0.077232	0.010017	0.016819	0.029982	0.018331	0.022421
housing	0.030429	0.066857	0.185552	0.006020	1.000000	0.041341	0.005041	0.023583	0.124197	0.037087	0.139161
loan	0.075319	0.084526	0.015641	0.077232	0.041341	1.000000	0.012395	0.009972	0.022762	0.011048	0.068193
duration	0.017220	0.022586	0.004645	0.010017	0.005041	0.012395	1.000000	0.084551	0.001549	0.001213	0.394549
campaign	0.026431	0.017274	0.004767	0.016819	0.023583	0.009972	0.084551	1.000000	0.088636	0.032860	0.073179
pdays	0.030751	0.006435	0.023745	0.029982	0.124197	0.022762	0.001549	0.088636	1.000000	0.454817	0.103616
previous	0.023046	0.016952	0.001297	0.018331	0.037087	0.011048	0.001213	0.032860	0.454817	1.000000	0.093232
term_deposit	0.053390	0.050785	0.025168	0.022421	0.139161	0.068193	0.394549	0.073179	0.103616	0.093232	1.000000

This presents a table where the correlation coefficient of each variable pairing is calculated. Most of the coefficients are on the low end, implying weak correlations. It's usually easier to spot correlations when they're plotted visually, so you'll investigate this more later.

8. Correlations aside, take a look at the list of variables and consider their data types, feature types, and what they describe.

   At this point in the process, which of these variables do you think might make good candidates for target features?

9. Save and close the lab.
   a) From the menu, select **File→Save and Checkpoint**.
   b) Close the lab browser tab and continue on with the course.

MODULE 2

Explore the Underlying Distribution of Data

The following labs are for Module 2: Explore the Underlying Distribution of Data.

LAB 3-2
Exploring the Underlying Distribution of Data

Data File

~/Analysis/Analyzing Data.ipynb

Scenario

Now that you've explored the structure and format of the GCNB dataset, you can begin to dive a little deeper into the nature of the values themselves. You'll focus on exploring the distribution of these values, which should give you a sense of how they are spread out and how they can be described using various summary statistics.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Generate summary statistics for all of the data** heading, then select **Cell→Run All Above**.

2. Generate summary statistics for all of the data.
   a) Scroll down and view the cell titled **Generate summary statistics for all of the data**, then select the code cell below it.
   b) In the code cell, type the following:

```
1 users_data.describe(datetime_is_numeric = True, include = 'all')
```

   c) Run the code cell.

d) Examine the output.

	user_id	number_transactions	total_amount_usd	age	job	marital	education	default	housing	loan	contact	du
count	45209	35210.000000	35210.000000	45209.000000	44921	45209	43352	45209	45209	45209	32191	45209.0
unique	45209	NaN	NaN	NaN	11	3	3	2	2	2	2	
top	729b0249-3961-4672-81cd-14b5ec6bdd39	NaN	NaN	NaN	blue-collar	married	secondary	False	True	False	cellular	
freq	1	NaN	NaN	NaN	9731	27212	23202	44394	25128	37965	29285	
mean	NaN	3.977052	1369.417751	40.935853	NaN	NaN	NaN	NaN	NaN	NaN	NaN	258.1
min	NaN	1.000000	-8019.000000	18.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.0
25%	NaN	2.000000	73.000000	33.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	103.0
50%	NaN	3.000000	451.000000	39.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	180.0
75%	NaN	4.000000	1438.000000	48.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	319.0
max	NaN	20.000000	102127.000000	95.000000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4918.0
std	NaN	3.814329	3063.412688	10.618653	NaN	NaN	NaN	NaN	NaN	NaN	NaN	257.5

Various statistics such as mean, frequency, min, max, standard deviation, and more are printed. This can get a little overwhelming, so you'll get more granular with how you retrieve them.

3. Generate summary statistics for numerical data only.

a) Scroll down and view the cell titled **Generate summary statistics for numerical data only**, then select the code cell below it.

b) In the code cell, type the following:

```
1 users_data.describe()
```

c) Run the code cell.

d) Examine the output.

	number_transactions	total_amount_usd	age	duration	campaign	pdays	previous
count	35210.000000	35210.000000	45209.000000	45209.000000	45209.000000	45209.000000	45209.000000
mean	3.977052	1369.417751	40.935853	258.153067	2.763897	40.199651	0.580349
std	3.814329	3063.412688	10.618653	257.525446	3.098076	100.130586	2.303489
min	1.000000	-8019.000000	18.000000	0.000000	1.000000	-1.000000	0.000000
25%	2.000000	73.000000	33.000000	103.000000	1.000000	-1.000000	0.000000
50%	3.000000	451.000000	39.000000	180.000000	2.000000	-1.000000	0.000000
75%	4.000000	1438.000000	48.000000	319.000000	3.000000	-1.000000	0.000000
max	20.000000	102127.000000	95.000000	4918.000000	63.000000	871.000000	275.000000

This output is a little neater, and focuses just on numeric features. Some things to point out include:

- The mean of each variable is on a very different scale. The mean `age` is 40, whereas the mean `total_amount_usd` is around $1,369.
- The standard deviation for each variable is roughly on the same scale as its mean, though `total_amount_usd` has quite a large standard deviation, indicating the values are spread out. This is partially due to the fact that some transactions are deposits (positive) whereas some are withdrawals (negative). The minimum value of $−8,019 confirms this.
- The `50%` statistic is the same as the median. So, it looks like the median age is close to the mean age, which could suggest a normal distribution.

4. Generate modal values for all data.

a) Scroll down and view the cell titled **Generate modal values for all data**, then select the code cell below it.

b) In the code cell, type the following:

```
# Drop user ID since it's unique.

users_data.drop(['user_id'], axis = 1).mode()
```

This will retrieve the mode for each variable, except for `user_id`, which is unique (and not particularly relevant).

c) Run the code cell.

d) Examine the output.

	number_transactions	total_amount_usd	age	job	marital	education	default	housing	loan	contact	duration	campaign	pdays	previous	poutcome
0	2.0	0.0	32.0	blue-collar	married	secondary	False	True	False	cellular	124.0	1.0	-1.0	0.0	failure
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

In this particular dataset:

- The most common age is 32.
- The most common type of job is a blue collar job.
- The most common marital status is married.
- The most common level of education is secondary education.
- Most users didn't default on their loans.
- Most users contact the bank using their mobile phones.
- And so on.

Note: There are two rows because the `date_joined` variable is bimodal. You can see both modes by scrolling all the way to the right.

5. Generate skewness and kurtosis measurements.

a) Scroll down and view the cell titled **Generate skewness and kurtosis measurements**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data.skew()
```

It's easier to interpret skewness and kurtosis visually, which you'll do a bit later. For now, you can just get a quick look at the numeric results.

c) Run the code cell.

d) Examine the output.

```
number_transactions     2.704543
total_amount_usd        8.596128
age                     0.684861
default                 7.245206
housing                -0.224686
loan                    1.852545
duration                3.144556
campaign                4.898555
pdays                   2.615635
previous               41.845672
term_deposit            2.383403
dtype: float64
```

Positive skewness numbers indicate the peak of the distribution is to the left (at the lower end of values) and the tail tapers off to the right; negative implies the opposite. Values closer to zero indicate a lack of skewness. So, it looks like:

- All of the features have a positive skew except for `housing`.
- Features like `age` and `housing` are close to zero, so they are closer to being normalized.
- The `previous` feature (number of times user was contacted prior to the current campaign) has by far the highest skew value.

Note: The `skew()` function takes the skewness measurements of Boolean variables as well as numeric variables. It considers `False` as `0` and `True` as `1`. So, for example, `default` has such a highly positive skew because there are many more `False` (`0`) values (users who didn't default on a loan) than `True` (`1`) values (users who did default on a loan). The peak is therefore shifted to the left, closer to `0`.

e) Select the next code cell, then type the following:

```
1 users_data.kurt()
```

f) Run the code cell.

g) Examine the output.

```
number_transactions       6.659034
total_amount_usd        150.790967
age                       0.319760
default                  50.495241
housing                  -1.949602
loan                      1.431987
duration                 18.155941
campaign                 39.248145
pdays                     6.934713
previous               4506.684640
term_deposit              3.680770
dtype: float64
```

The `kurt()` function in pandas uses excess kurtosis, so zero implies a normal distribution whereas negative values are platykurtic and positive values are leptokurtic. So:

- Most of the features are leptokurtic, implying that there are outliers.
- The `total_amount_usd` and `previous` features have the highest leptokurtosis, so they may have some extreme outliers.
- The `housing` feature is slightly platykurtic, meaning it likely isn't as affected by outliers.

Note: Just like `skew()`, the `kurt()` function considers Boolean values as the numbers `0` and `1`. Since there are really only two numbers to consider, a feature like `default` is highly leptokurtic because one of those numbers is much more frequent than the other, leading to a peak. A feature like `housing` is much more evenly distributed, so it's mostly flat without a significant peak.

6. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

MODULE 3

Use Visualizations to Analyze Data

The following labs are for Module 3: Use Visualizations to Analyze Data.

LAB 3-3
Analyzing Data Using Histograms

Data File

~/Analysis/Analyzing Data.ipynb

Scenario

You want to start exploring the GCNB dataset visually to aid in your interpretation. There are many chart types you can construct from this data, but you'll start with histograms in order to view the distributions of the several features.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Plot histograms for all numerical columns** heading, then select **Cell→Run All Above**.

2. Plot histograms for all numerical columns.
   a) Scroll down and view the cell titled **Plot histograms for all numerical columns**, then select the code cell below it.
   b) In the code cell, type the following:

```
users_data_for_hist = \
users_data.select_dtypes(exclude = ['bool'])
```

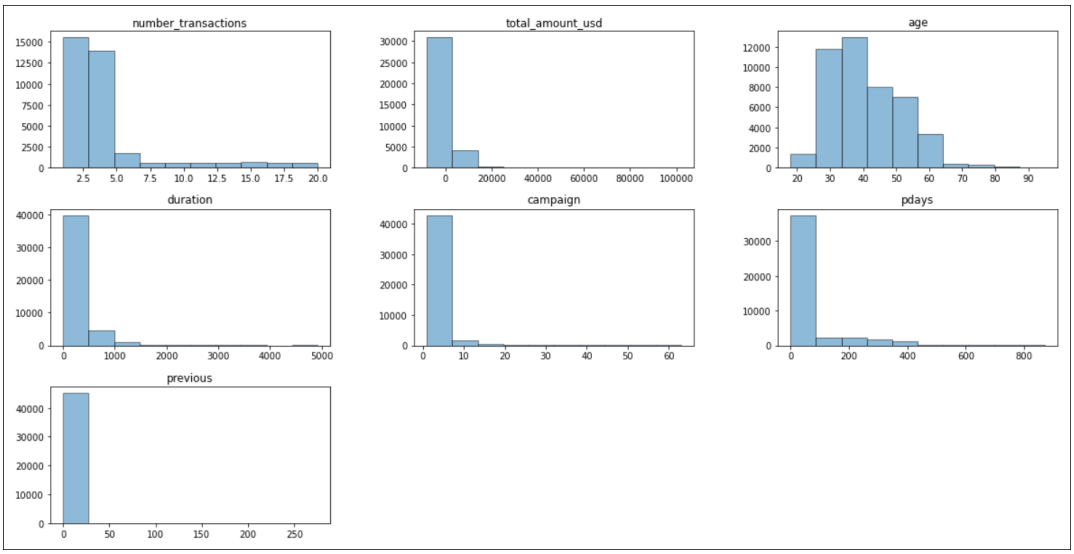
   Before drawing histograms, you'll need to temporarily drop the categorical Boolean data from the dataset, as that data is not continuous.
   c) Run the code cell.
   d) Select the next code cell, then type the following:

```
users_data_for_hist.hist(figsize = (20, 10), alpha = 0.5,
                         edgecolor = 'black', grid = False);
```

Note: In Jupyter Notebook, the semicolon (;) at the end of the last function call in a code cell suppresses that function's text output.

   e) Run the code cell.

f) Examine the output.



These histograms visually confirm the skewness measurements you saw earlier. As you can see, most of the data is heavily skewed to the right (positive). The only variable that comes close to a normal distribution is `age`. You'll do some transformations for these skewed variables later on.

3. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 3-4

Analyzing Data Using Box Plots and Violin Plots

Data File

~/Analysis/Analyzing Data.ipynb

Scenario

Histograms are great for showing general distribution shape, but there are other ways to visualize the spread of values. You'll create box plots and violin plots to help you identify outliers that you may want to remove so that they don't unduly influence any models you plan on building.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Generate a box plot for `age` heading**, then select **Cell→Run All Above**.

2. Generate a box plot for `age`.
   a) Scroll down and view the cell titled **Generate a box plot for `age`**, then select the code cell below it.
   b) In the code cell, type the following:

```
1 users_data['age'].describe()
```

First you'll call up some of the descriptive statistics for the `age` feature, just as a refresher.

   c) Run the code cell.
   d) Examine the output.

```
count    45209.000000
mean        40.935853
std         10.618653
min         18.000000
25%         33.000000
50%         39.000000
75%         48.000000
max         95.000000
Name: age, dtype: float64
```

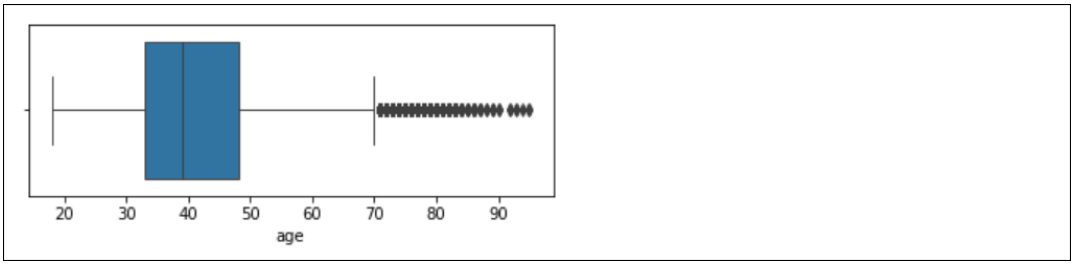
The minimum age is 18 and the maximum is 95. The standard deviation is 10 and the mean is 40, so some of these ages might be outliers.

   e) Select the next code cell, then type the following:

```
1 plt.figure(figsize = (6, 2))
2 sns.boxplot(x = users_data['age'], linewidth = 0.9);
```

   f) Run the code cell.

g) Examine the output.



The result of the box plot shows the minimum and maximum as vertical lines at the end of each whisker. Note these values exclude outliers. In fact, the box plot clearly shows that there are several outliers on the higher end of the distribution. It seems that users past the age of 70 are skewing the distribution. You may want to drop these outliers from the dataset, but for now, you'll leave them be.

3. Generate a violin plot for `age`.

a) Scroll down and view the cell titled **Generate a violin plot for `age`**, then select the code cell below it.

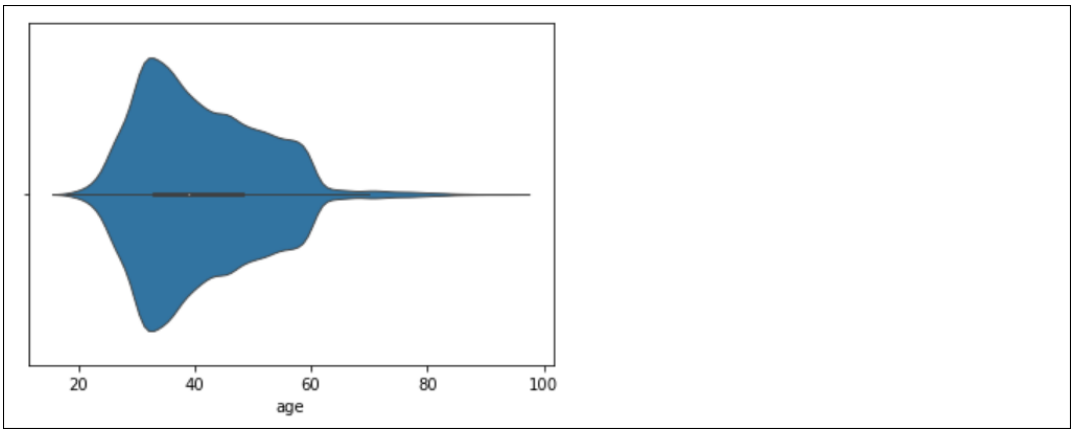
b) In the code cell, type the following:

```
1 sns.violinplot(x = users_data['age'], linewidth = 0.9);
```

A violin plot is another way of showing variable distribution. It uses kernel density estimation (KDE) to map the distribution.

c) Run the code cell.

d) Examine the output.



The wider portion of the violin demonstrates a greater probability of values occurring at the point of the distribution. So ages around 30–35 are pretty likely in this dataset, whereas any age above 60 or below 20 is pretty unlikely.

4. Generate a box plot for `number_transactions`.

a) Scroll down and view the cell titled **Generate a box plot for `number_transactions`**, then select the code cell below it.

b) In the code cell, type the following:

```
1 users_data['number_transactions'].describe()
```

c) Run the code cell.

d) Examine the output.

```
count    35210.000000
mean         3.977052
std          3.814329
min          1.000000
25%          2.000000
50%          3.000000
75%          4.000000
max         20.000000
Name: number_transactions, dtype: float64
```

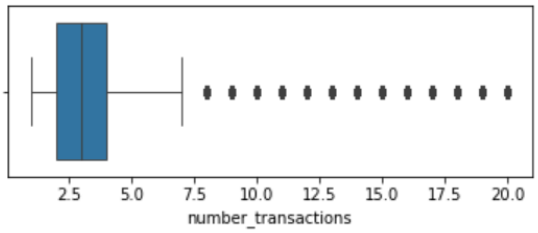
This feature could also be susceptible to outliers, especially at the higher end, so you'll generate a box plot for it as well.

e) Select the next code cell, then type the following:

```
plt.figure(figsize = (6, 2))
sns.boxplot(x = users_data['number_transactions'],
            linewidth = 0.9);
```

f) Run the code cell.

g) Examine the output.



As suspected, there are also outliers at the higher end of this feature's distribution. Some customers had an unusually large number of transactions, which is defined as being more than about 7 transactions.

5. Generate a violin plot for `number_transactions`.

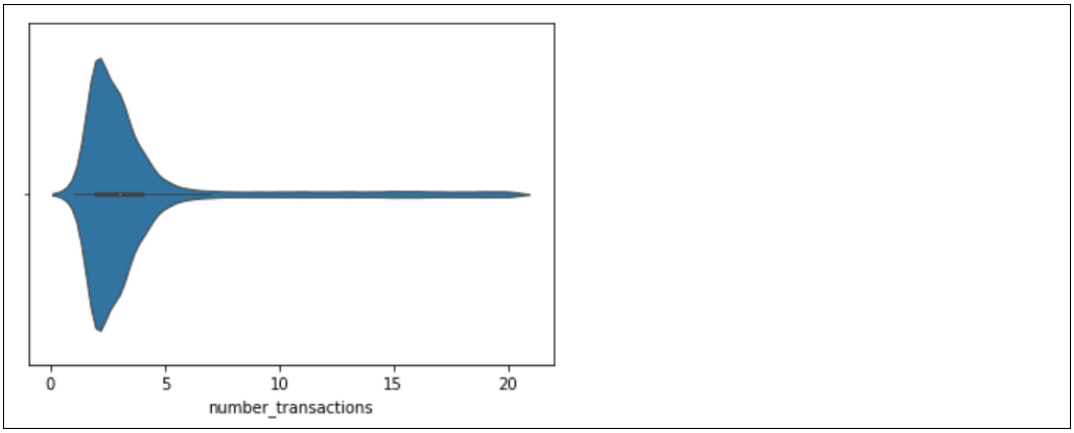
a) Scroll down and view the cell titled **Generate a violin plot for `number_transactions`**, then select the code cell below it.

b) In the code cell, type the following:

```
sns.violinplot(x = users_data['number_transactions'],
               linewidth=0.9);
```

c) Run the code cell.

d) Examine the output.



The probability of the number of transactions being around 2–3 is pretty high, whereas anything above 5 has a low probability.

6. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.
b) Close the lab browser tab and continue on with the course.

LAB 3-5

Analyzing Data Using Scatter Plots and Line Plots

Data File

~/Analysis/Analyzing Data.ipynb

Scenario

Other than using distribution plots, you can also get a sense of how data appears when two or more features are compared to each other. You want to see if the total amount of all of a user's transactions is related to the number of transactions they engage in. Perhaps the more transactions the user makes, the more of a total net positive they have in terms of total amount. Or, perhaps it's the opposite. There may even be no correlation whatsoever between the number of transactions and the total amount. So, you'll generate a scatter plot to find out.

You also want to see how account age is affecting the total amount, if at all. Are long-time users more likely to have a net positive in their account, or not? So, you'll generate a line plot comparing the years users signed up and the average total amount for each user in a sign-up year.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

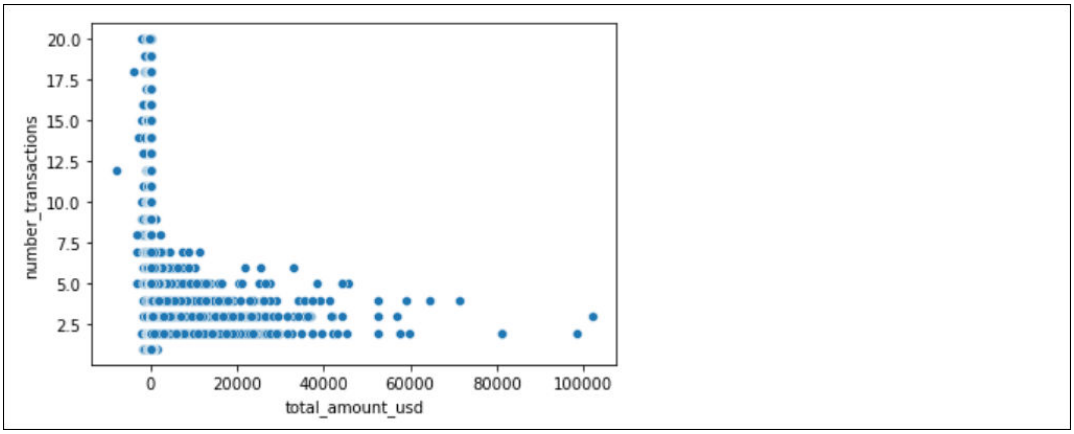
Note: If you encounter errors when running the next code blocks, select the **Generate scatter plots comparing `total_amount_usd` to `number_transactions`** heading, then select **Cell→Run All Above**.

2. Generate scatter plots comparing `total_amount_usd` to `number_transactions`.
   a) Scroll down and view the cell titled **Generate scatter plots comparing `total_amount_usd` to `number_transactions`**, then select the code cell below it.
   b) In the code cell, type the following:

```
sns.scatterplot(data = users_data, x = 'total_amount_usd',
                y = 'number_transactions');
```

   c) Run the code cell.

d) Examine the output.



The scatter plot shows `total_amount_usd` on the x-axis, compared to `number_transactions` on the y-axis. By looking at the scatter plot, you can see that there is not necessarily a strong correlation between either feature. However, you can conclude that most users with a high number of transactions also have a relatively low total transaction amount. This could be due to alternating deposits and withdrawals. Also, the highest transaction amounts seem to be part of a low number of transactions.

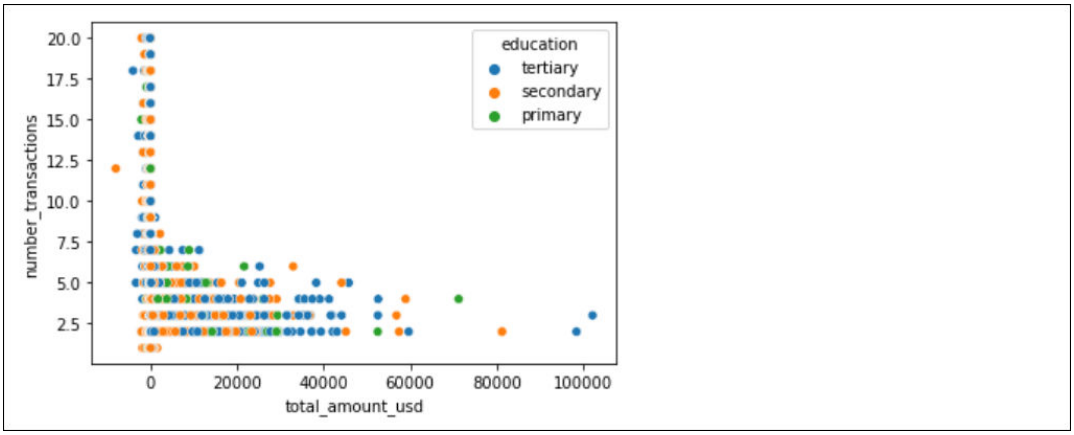
e) Select the next code cell, then type the following:

```
sns.scatterplot(data = users_data, x = 'total_amount_usd',
                y = 'number_transactions', hue = 'education');
```

This code adds an extra dimension to the scatter plot: education level.

f) Run the code cell.

g) Examine the output.



You can now see each education label as a separate color dot. This helps you compare a third feature to the primary two. In this case, a user's education level doesn't necessarily exhibit a pattern when the number of transactions is compared to the total amount. Though, it does appear that the two users with the highest total amounts have a tertiary education.

3. Generate a line plot for `total_amount_usd`.

a) Scroll down and view the cell titled **Generate a line plot for `total_amount_usd`**, then select the code cell below it.

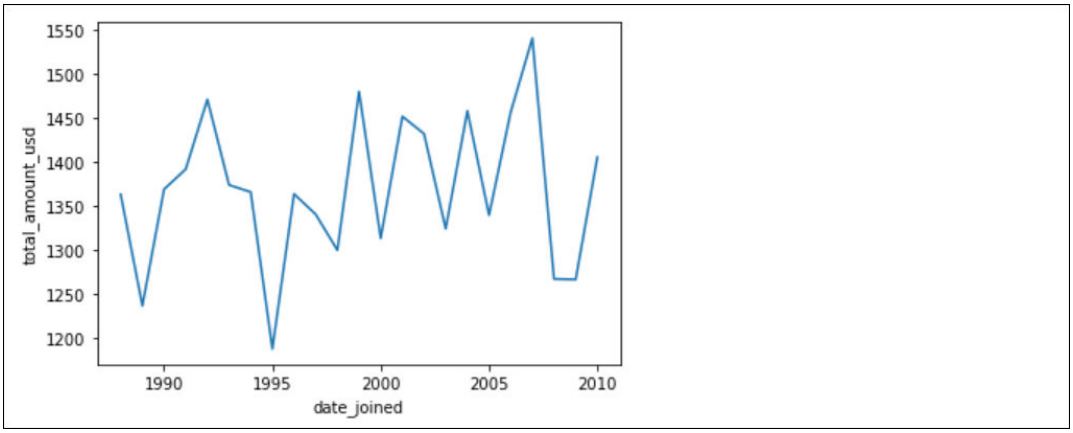
b) In the code cell, type the following:

```
years = users_data['date_joined'].dt.year

sns.lineplot(data = users_data, x = years,
             y = 'total_amount_usd',
             estimator = np.mean);
```

This code takes a slice of `date_joined` so that it only considers the years. These year values are plugged into the plotting function.

c) Run the code cell.

d) Examine the output.



This line plot shows the trend of the mean of total transaction amount for each year that users signed up for an account. For example, for all users who signed up in 1995, the mean of each user's total transaction amount is slightly less than $1,200. The graph indicates that the total transaction amount seems to fluctuate quite a bit between each sign-up year. Some years, like 1995, show an obvious decline, whereas others, like 2007, show a dramatic increase.

Note: This graph is not showing the mean amount of all transactions in each year, but the mean amount for all users who signed up in each year.

4. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 3-6
Analyzing Data Using Bar Charts

Data File

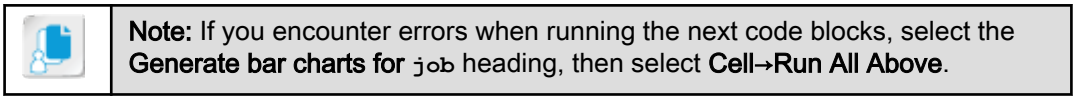
~/Analysis/Analyzing Data.ipynb

Scenario

The GCNB dataset has many categorical features, which are great candidates for bar charts. You want see the frequency of values in each of these features. So, you'll create several bar charts to compare those frequencies.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.



2. Generate bar charts for `job`.
   a) Scroll down and view the cell titled **Generate bar charts for `job`**, then select the code cell below it.
   b) In the code cell, type the following:

```
users_job_dist = \
users_data['job'].value_counts(dropna = False)

users_job_dist
```

Before you plot the bar chart, you'll get a look at the frequencies of each `job` value.

   c) Run the code cell.
   d) Examine the output.

```
blue-collar      9731
management       9457
technician       7597
admin.           5171
services         4154
retired          2264
self-employed    1579
entrepreneur     1487
unemployed       1303
housemaid        1240
student           938
NaN               288
Name: job, dtype: int64
```

The frequencies of each value are in descending order. So, blue collar jobs and management jobs seem to be the most common, whereas students are the least common (besides missing values).

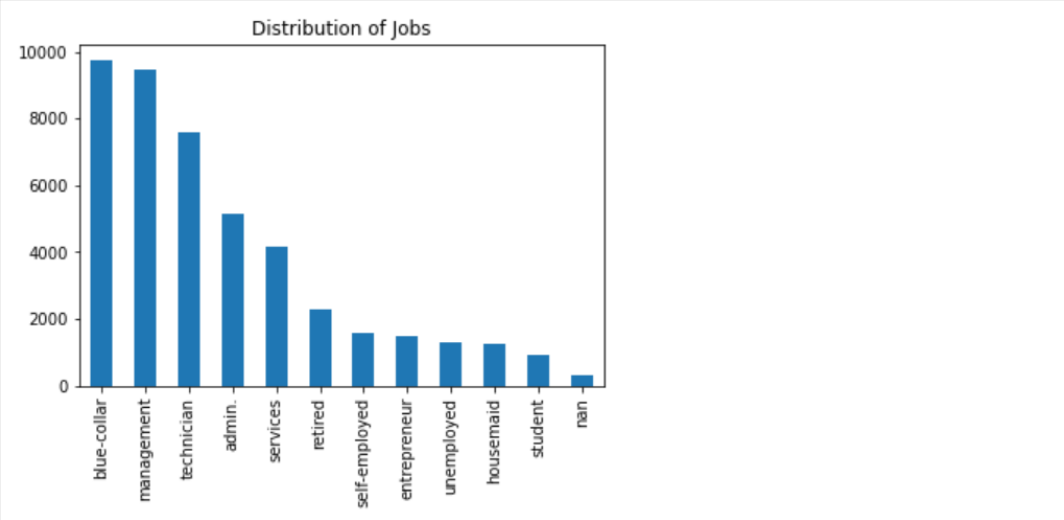
e) Select the next code cell, then type the following:

```
# Vertical bar chart.

users_job_dist.plot(kind = 'bar')
plt.title('Distribution of Jobs');
```

You'll plot a vertical bar chart first.

f) Run the code cell.

g) Examine the output.



This visually confirms the numeric frequencies you just saw. You can also see that the frequencies tend to taper off gradually, rather than there being an abrupt change in frequencies.

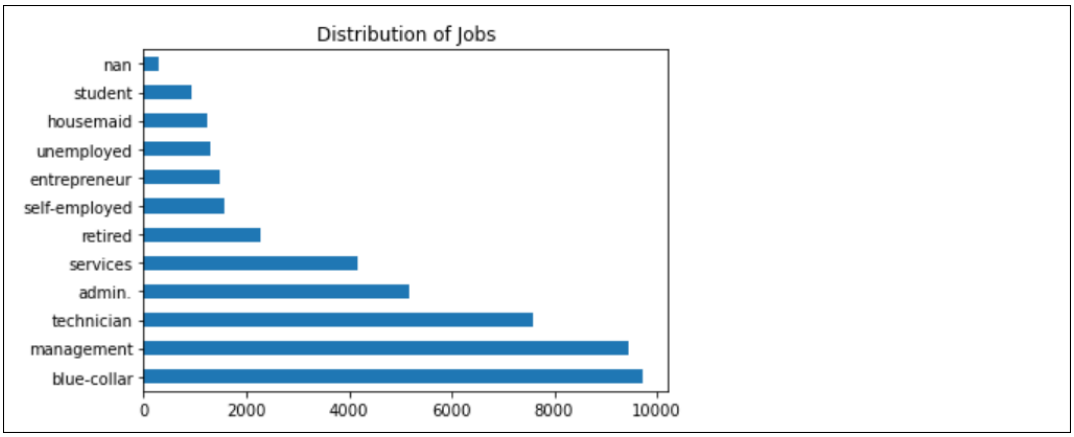
h) Select the next code cell, then type the following:

```
# Horizontal bar chart

users_job_dist.plot(kind = 'barh')
plt.title('Distribution of Jobs');
```

You'll generate the same chart, this time horizontally.

i) Run the code cell.

j) Examine the output.



Sometimes, reorienting a chart can make it easier to read.

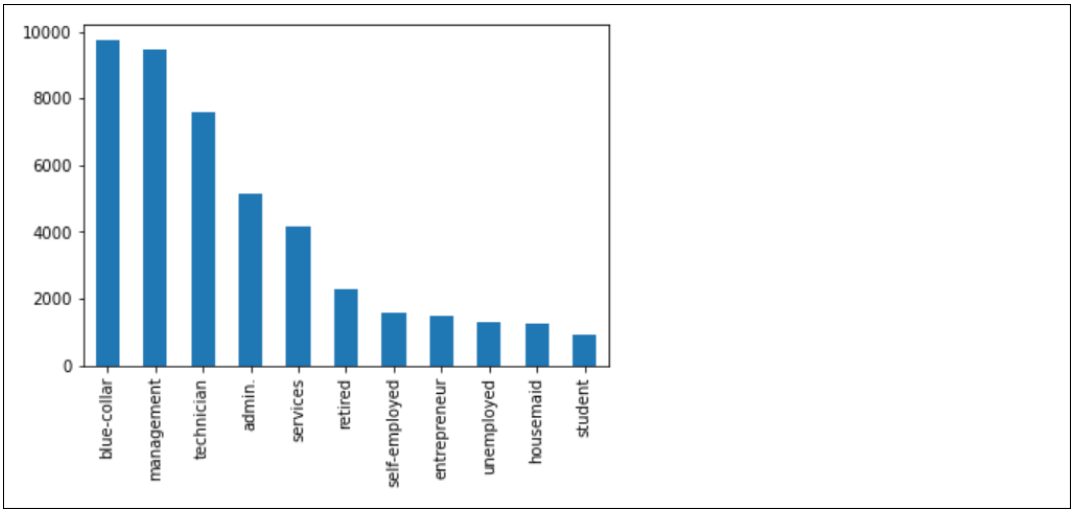
k) Select the next code cell, then type the following:

```
# Exclude missing values.

users_data['job'].value_counts().plot(kind = 'bar');
```

This time you'll plot the same data while excluding missing values.

l) Run the code cell.

m) Examine the output.



You'll deal with these later, but for now it can be helpful to just ignore bad data for analysis purposes.

3. Generate a bar chart for `marital`.

a) Scroll down and view the cell titled **Generate a bar chart for `marital`**, then select the code cell below it.

b) In the code cell, type the following:

```
users_marital_dist = \
users_data['marital'].value_counts(dropna = False)

users_marital_dist
```

As before, you'll get the raw frequencies first.

c) Run the code cell.

d) Examine the output.

```
married     27212
single      12790
divorced     5207
Name: marital, dtype: int64
```

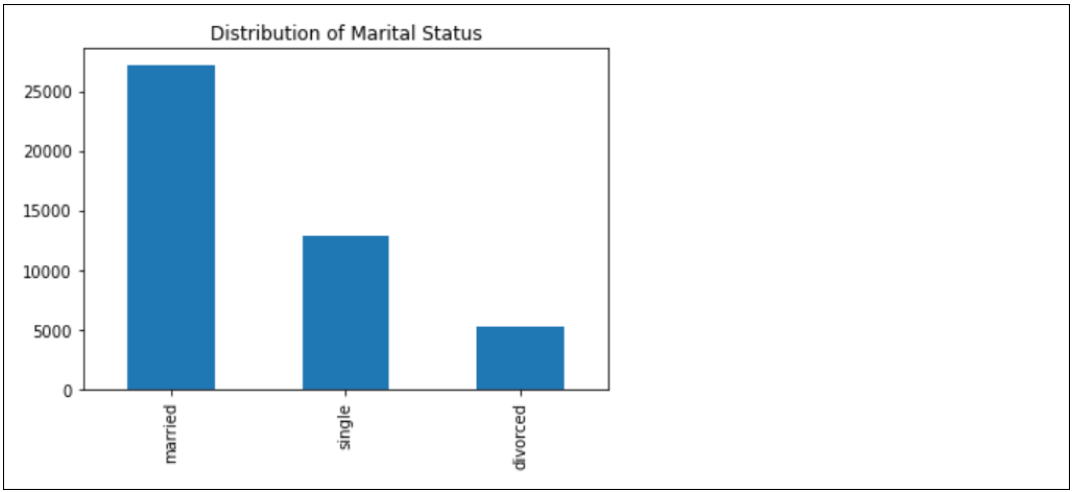
Married users far outweigh single and divorced users.

e) Select the next code cell, then type the following:

```
users_marital_dist.plot(kind = 'bar')
plt.title('Distribution of Marital Status');
```

f) Run the code cell.

g) Examine the output.



This helps to demonstrate just how differently each marital status is represented in the data.

4. Generate a bar chart for `education`.

a) Scroll down and view the cell titled **Generate a bar chart for `education`**, then select the code cell below it.

b) In the code cell, type the following:

```
users_education_dist = \
users_data['education'].value_counts(dropna = False)

users_education_dist
```

c) Run the code cell.

d) Examine the output.

```
secondary    23202
tertiary     13300
primary       6850
NaN           1857
Name: education, dtype: int64
```

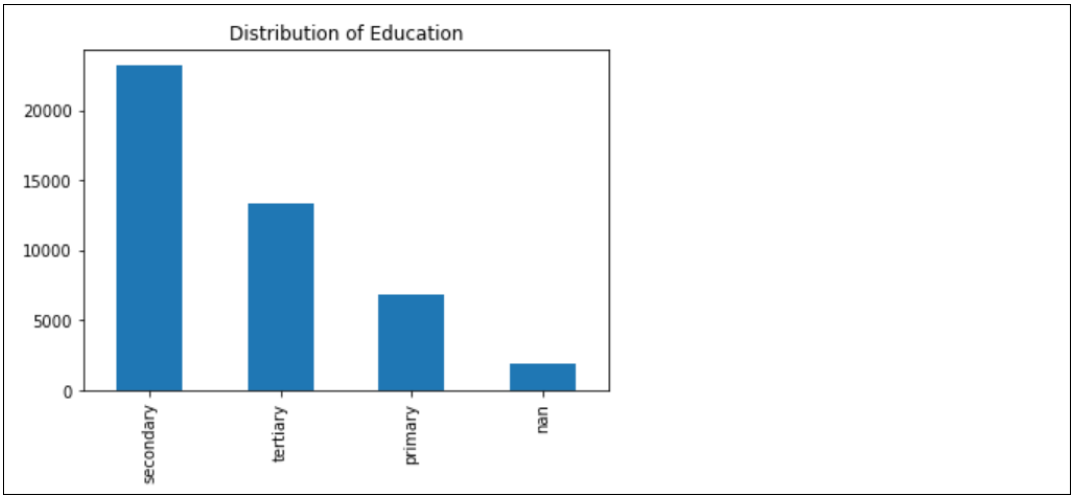
Secondary education seems to be the most common.

e) Select the next code cell, then type the following:

```
users_education_dist.plot(kind = 'bar')
plt.title('Distribution of Education');
```

f) Run the code cell.

g) Examine the output.



Secondary education is indeed the most common, followed by tertiary, then primary.

5. Generate a bar chart for `contact`.

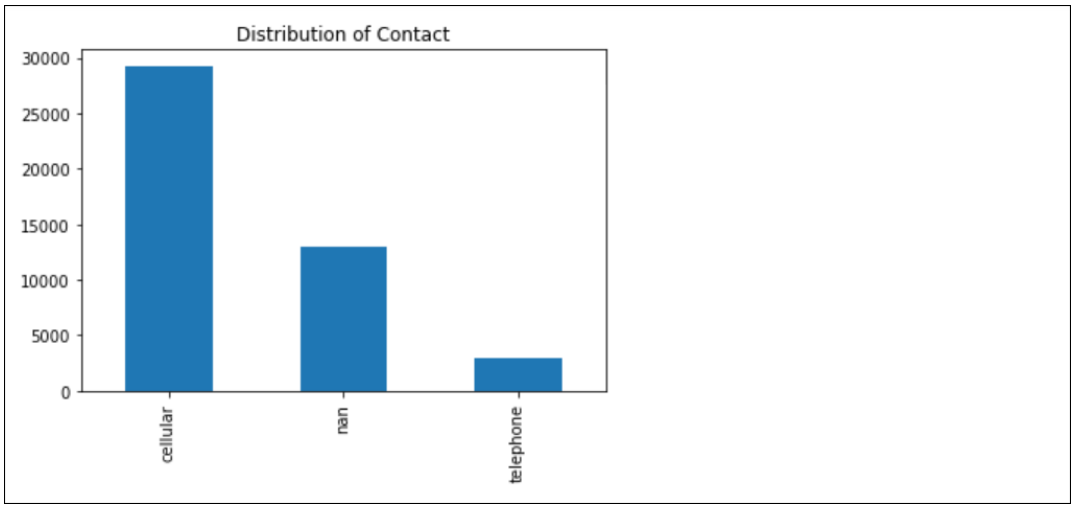
a) Scroll down and view the cell titled **Generate a bar chart for `contact`**, then select the code cell below it.

b) In the code cell, type the following:

```
users_contact_dist = \
users_data['contact'].value_counts(dropna = False)

users_contact_dist.plot(kind = 'bar')
plt.title('Distribution of Contact');
```

c) Run the code cell.

d) Examine the output.



By far, using a mobile phone is the most common way for users to contact the bank. A landline telephone is very uncommon. There are also quite a few missing values for this feature.

6. Generate a bar chart for `poutcome`.

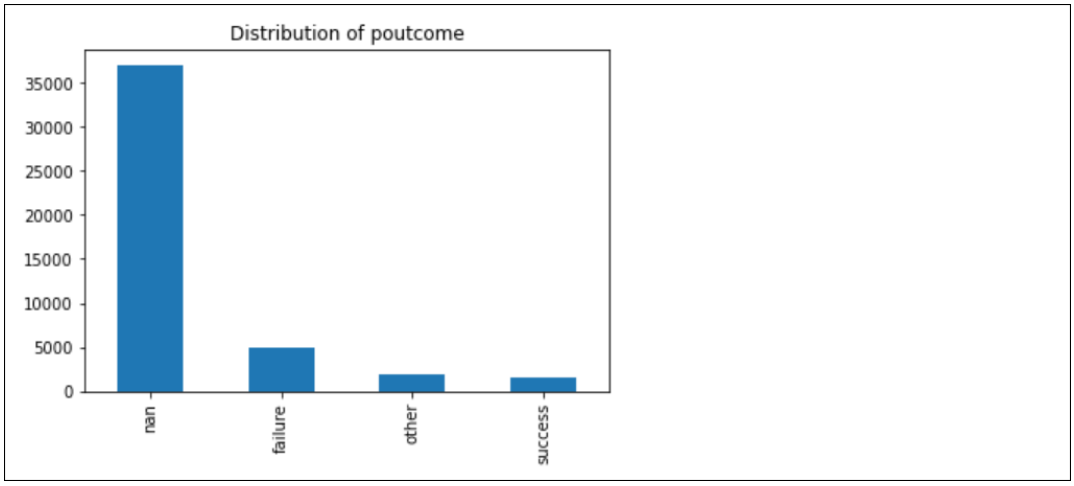
a) Scroll down and view the cell titled **Generate a bar chart for `poutcome`**, then select the code cell below it.

b) In the code cell, type the following:

```
users_poutcome_dist = \
users_data['poutcome'].value_counts(dropna = False)

users_poutcome_dist.plot(kind = 'bar')
plt.title('Distribution of poutcome');
```

c) Run the code cell.

d) Examine the output.



This particular feature (whether or not the previous marketing campaign was successful) has a lot of missing values. You'll handle these soon.

7. Generate a bar chart for `default`.

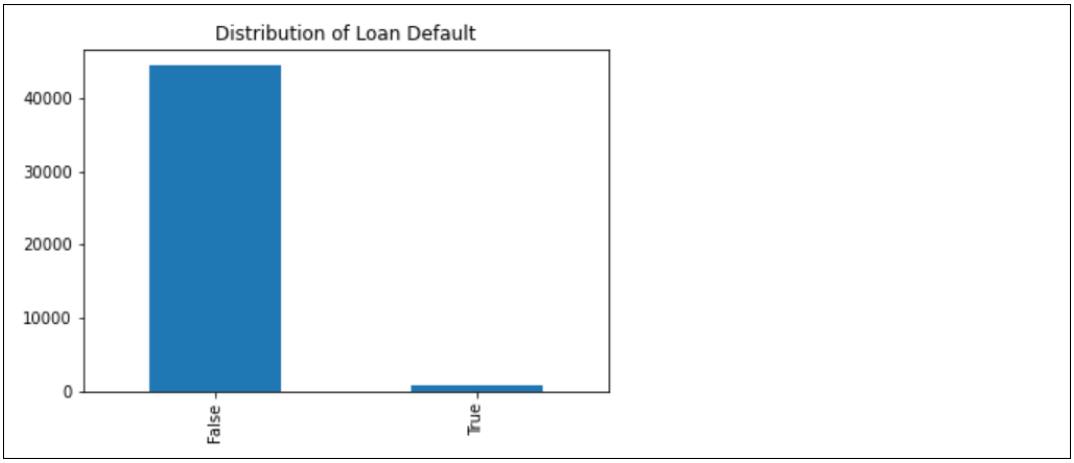
a) Scroll down and view the cell titled **Generate a bar chart for `default`**, then select the code cell below it.

b) In the code cell, type the following:

```
users_device_dist = \
users_data['default'].value_counts(dropna = False)

users_device_dist.plot(kind = 'bar')
plt.title('Distribution of Loan Default');
```

c) Run the code cell.

d) Examine the output.



Many more people did not default on a loan than did. The chart also confirms the high level of positive skewness for this Boolean variable that you identified earlier, as well as the high level of leptokurtosis.

8. Generate a bar chart for `device`.

a) Scroll down and view the cell titled **Generate a bar chart for `device`**, then select the code cell below it.

b) In the code cell, type the following:

```
users_device_dist = \
users_data['device'].value_counts(dropna = False)
```

c) Run the code cell.

d) Select the next code cell, then type the following:

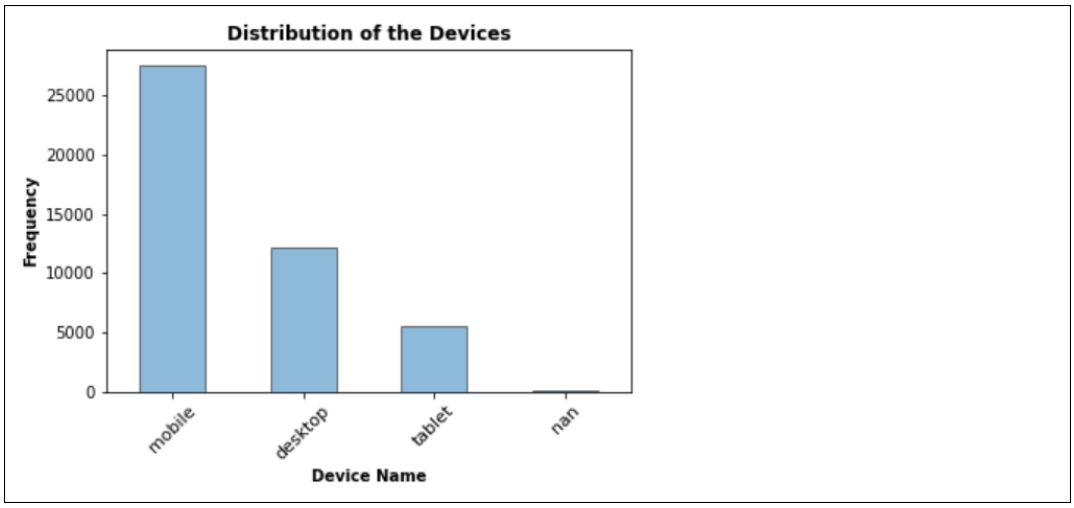
```
users_device_dist.plot(kind = 'bar',
                       alpha = 0.5, edgecolor = 'black')
plt.title('Distribution of the Devices',
          size = 12, weight = 'bold')
plt.xticks(rotation = 45, size = 11)
plt.xlabel('Device Name', size = 10, weight = 'bold')
plt.ylabel('Frequency', size = 10, weight = 'bold')
plt.show();
```

Charts can be much more informative and easier to read when you take the time to format them. Here you're:

- Plotting the chart.
- Adding a title to the chart.
- Changing the tick labels.
- Changing the x-axis and y-axis labels.

e) Run the code cell.

f) Examine the output.



When it comes to using the bank's online services, mobile devices are most popular, while desktops and tablets trail behind. Once again, there are some missing values.

9. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 3-7
Analyzing Data Using Maps

Data File

~/Analysis/Analyzing Data.ipynb

Scenario

You've taken a preliminary look at the feature correlations in the dataset, but you want to dive a little deeper. A correlation heatmap is a good way to visually compare these feature correlations, so you'll create one. The correlations or lack thereof may dictate how you drop or keep certain features.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Generate a heatmap for the feature correlations** heading, then select **Cell→Run All Above**.

2. Generate a heatmap for the feature correlations.
   a) Scroll down and view the cell titled **Generate a heatmap for the feature correlations**, then select the code cell below it.
   b) In the code cell, type the following:

```
corr_matrix = users_data.corr()

corr_matrix
```

   c) Run the code cell.
   d) Examine the output.

	number_transactions	total_amount_usd	age	default	housing	loan	duration	campaign	pdays	previous	term_dep
number_transactions	1.000000	-0.163409	-0.008813	0.138838	0.030429	0.075319	-0.017220	0.026431	-0.030751	-0.023046	-0.053
total_amount_usd	-0.163409	1.000000	0.095839	-0.065390	-0.066857	-0.084526	0.022586	-0.017274	0.006435	0.016952	0.050
age	-0.008813	0.095839	1.000000	-0.017875	-0.185552	-0.015641	-0.004645	0.004767	-0.023745	0.001297	0.025
default	0.138838	-0.065390	-0.017875	1.000000	-0.006020	0.077232	-0.010017	0.016819	-0.029982	-0.018331	-0.022
housing	0.030429	-0.066857	-0.185552	-0.006020	1.000000	0.041341	0.005041	-0.023583	0.124197	0.037087	-0.139
loan	0.075319	-0.084526	-0.015641	0.077232	0.041341	1.000000	-0.012395	0.009972	-0.022762	-0.011048	-0.068
duration	-0.017220	0.022586	-0.004645	-0.010017	0.005041	-0.012395	1.000000	-0.084551	-0.001549	0.001213	0.394
campaign	0.026431	-0.017274	0.004767	0.016819	-0.023583	0.009972	-0.084551	1.000000	-0.088636	-0.032860	-0.073
pdays	-0.030751	0.006435	-0.023745	-0.029982	0.124197	-0.022762	-0.001549	-0.088636	1.000000	0.454817	0.103
previous	-0.023046	0.016952	0.001297	-0.018331	0.037087	-0.011048	0.001213	-0.032860	0.454817	1.000000	0.093
term_deposit	-0.053390	0.050785	0.025168	-0.022421	-0.139161	-0.068193	0.394549	-0.073179	0.103616	0.093232	1.000

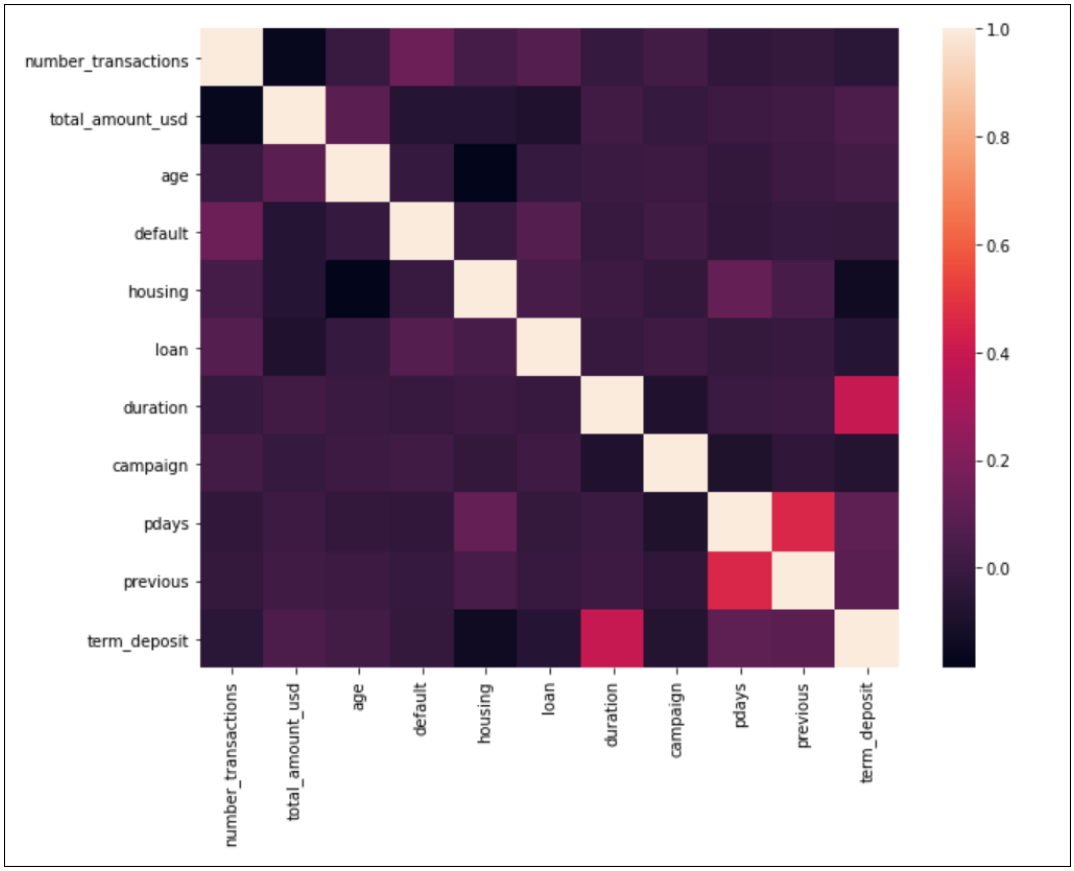
The correlation coefficients are presented as numbers in a table. This can be somewhat difficult to read, so you'll create a heatmap out of this data instead.

e) Select the next code cell, then type the following:

```
fig = plt.figure(figsize = (10, 7.5))

sns.heatmap(corr_matrix);
```

f) Run the code cell.

g) Examine the output.



The heatmap shows correlation strength as a gradation of color. Lighter values show a stronger positive correlation. However, negative values are not represented on an equivalent color gradient, which makes it harder to distinguish weak negative correlations from strong negative correlations. You'll therefore create a more useful heatmap in the next step.

3. Format the heatmap to make it easier to read.

a) Scroll down and view the cell titled **Format the heatmap to make it easier to read**, then select the code cell below it.

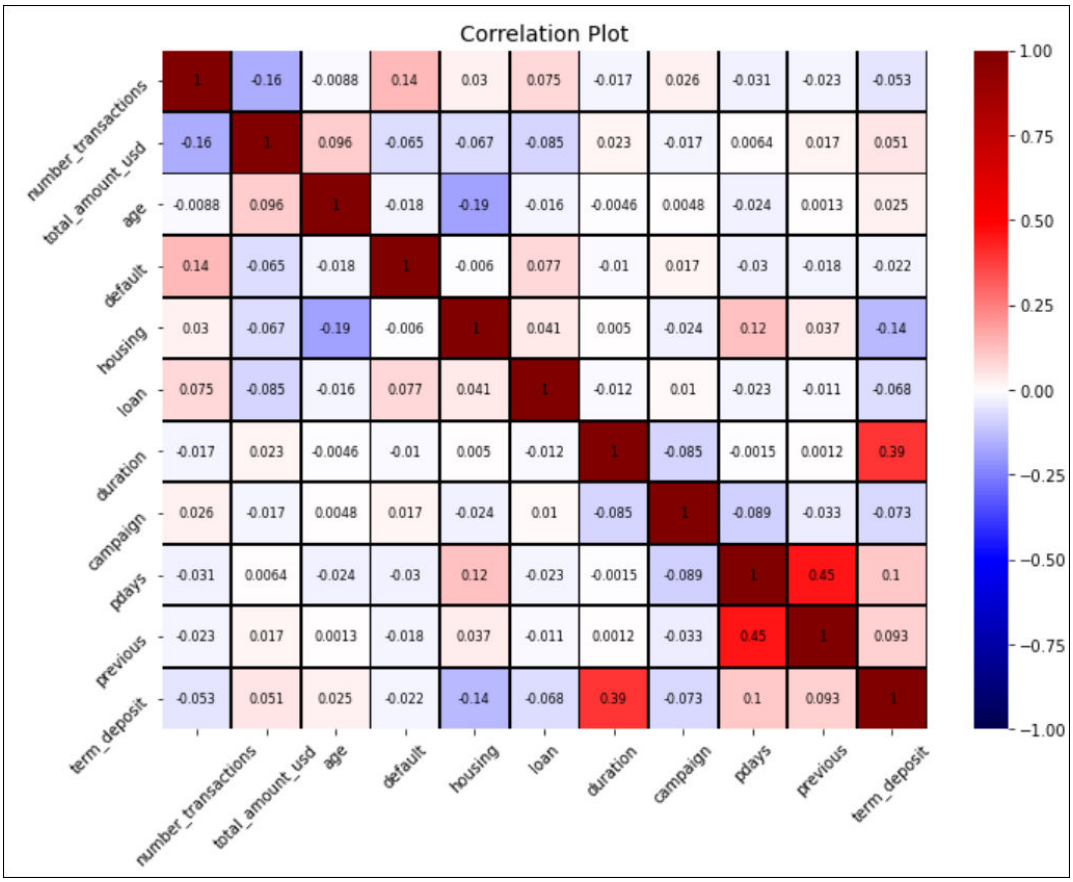
b) In the code cell, type the following:

```
fig = plt.figure(figsize = (11, 8))

sns.heatmap(corr_matrix,
            cmap = 'seismic',
            linewidth = 0.75,
            linecolor = 'black',
            cbar = True,
            vmin = -1,
            vmax = 1,
            annot = True,
            annot_kws = {'size': 8, 'color': 'black'})

plt.tick_params(labelsize = 10, rotation = 45)
plt.title('Correlation Plot', size = 14);
```

This code will show the "heat" as shades of two different diverging colors. Also, this will print the actual correlation values for each cell in the matrix.

c) Run the code cell.

d) Examine the output.



The same data is plotted, but in a more informative way. Darker red cells indicate stronger positive correlations, whereas darker blue cells indicate stronger negative correlations. Lightly colored cells exhibit weak correlation in either direction.

Aside from the main diagonal (variables being compared to themselves), the features in this dataset don't show a lot of correlation. However, the strongest positive correlations seem to be at the bottom right of the map, particularly between `duration` and `term_deposit`, and `pdays` and `previous`. So, the length of the last contact between the user and bank seems to have somewhat of a positive correlation with whether or not the user signed up for a term deposit. Likewise, as more days pass since the last time the user was contacted, the more times the user was likely to have been contacted in previous campaigns. These scenarios make sense intuitively, but an increase in one feature doesn't necessarily cause the other to increase.

There's not a lot of strong negative correlations. The strongest (in a relative sense) are between `total_amount_usd` and `number_transactions`, `housing` and `age`, and `housing` and `term_deposit`. So, for example, this suggests that the older a user is, the slightly less likely it is they'll have a housing loan. Again, you can't definitively say that one feature increasing causes another to decrease.

4. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

MODULE 4

Preprocess Data

The following labs are for Module 4: Preprocess Data.

LAB 3-8
Handling Missing Values

Data File

~/Analysis/Analyzing Data.ipynb

Scenario

As you've probably seen by now, the GCNB dataset has a lot of missing data. You need to handle that missing data before it can be used to build a machine learning model. Rather than remove all of the missing data wholesale, you'll take a more surgical approach and apply different methods to different instances.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Identify missing values** heading, then select **Cell→Run All Above**.

2. Identify missing values.
   a) Scroll down and view the cell titled **Identify missing values**, then select the code cell below it.
   b) In the code cell, type the following:

```
1 users_data.isnull().sum()
```

   This will print the total number of missing values for each column.
   c) Run the code cell.

d) Examine the output.

```
user_id                   0
number_transactions     9999
total_amount_usd        9999
age                        0
job                      288
marital                    0
education               1857
default                    0
housing                    0
loan                       0
contact                13018
duration                   0
campaign                   0
pdays                      0
previous                   0
poutcome               36957
term_deposit               0
date_joined               30
device                    94
dtype: int64
```

It seems that 8 columns contain missing values, some at higher amounts than others.

3. Identify the percentage of missing values for each feature.

a) Scroll down and view the cell titled **Identify the percentage of missing values for each feature**, then select the code cell below it.

b) In the code cell, type the following:

```
1 percent_missing = users_data.isnull().mean()
2
3 percent_missing
```

Another way to look at missing values is to see what percentage of the data in that column is missing.

c) Run the code cell.

d) Examine the output.

```
user_id                0.000000
number_transactions    0.221173
total_amount_usd       0.221173
age                    0.000000
job                    0.006370
marital                0.000000
education              0.041076
default                0.000000
housing                0.000000
loan                   0.000000
contact                0.287952
duration               0.000000
campaign               0.000000
pdays                  0.000000
previous               0.000000
poutcome               0.817470
term_deposit           0.000000
date_joined            0.000664
device                 0.002079
dtype: float64
```

The `poutcome` feature has the highest number of missing values at around 82%, whereas `device` has very few.

4. Generate a missing value report.

a) Scroll down and view the cell titled **Generate a missing value report**, then select the code cell below it.

b) In the code cell, verify the following:

```
def missing_value_pct_df(data):
    """Create a DataFrame to summarize missing values."""

    percent_missing = data.isnull().mean()
    missing_value_df = \
    pd.DataFrame(percent_missing).reset_index()

    missing_value_df = \
    missing_value_df.rename(columns = {'index': 'column_name',
                                       0: 'percent_missing'})

    # Multiply by 100 and round to 4 decimal places.
    missing_value_df['percent_missing'] = \
    missing_value_df['percent_missing']. \
    apply(lambda x: round(x * 100, 2))

    missing_value_df = \
    missing_value_df.sort_values(by = ['percent_missing'],
                                 ascending = False)

    return missing_value_df
```

This code is provided for you. It's a function that creates a `DataFrame` to summarize missing values in a readable and repeatable way.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
missing_value_df = missing_value_pct_df(users_data)

missing_value_df
```

This code calls the function that was just defined on the users dataset.

e) Run the code cell.

f) Examine the output.

	column_name	percent_missing
15	poutcome	81.75
10	contact	28.80
2	total_amount_usd	22.12
1	number_transactions	22.12
6	education	4.11
4	job	0.64
18	device	0.21
17	date_joined	0.07
12	campaign	0.00
16	term_deposit	0.00
14	previous	0.00
13	pdays	0.00
0	user_id	0.00
11	duration	0.00
8	housing	0.00
7	default	0.00
5	marital	0.00
3	age	0.00
9	loan	0.00

The output is a `DataFrame` in which each feature is a row that also has a corresponding percentage of missing values, sorted in descending order by missing percentage. You can see that `contact`, `total_amount_usd`, and `number_transactions` also have a high percentage of missing values.

5. Remove features with a high percentage of missing values.

a) Scroll down and view the cell titled **Remove features with a high percentage of missing values**, then select the code cell below it.

b) In the code cell, type the following:

```
# Threshold above which to drop feature.

threshold = 80

cols_to_drop = \
list(missing_value_df[missing_value_df['percent_missing'] \
                      > threshold]['column_name'])

print('Number of features to drop:',
      missing_value_df[ \
      missing_value_df['percent_missing'] > threshold].shape[0])

print(f'Features with missing values greater than {threshold}%:',
      cols_to_drop)
```

It's not always feasible to fill in missing values when so many occur for a single feature. Rather than keep that feature in the dataset, it should be dropped so that it doesn't negatively influence a model. In this case, the threshold is 80%, so any feature with a percentage of missing values greater than that will be dropped. This is a somewhat arbitrary figure and something that you can adjust for your own purposes, but it's common to drop features that are missing around 70%–80% of values.

c) Run the code cell.

d) Examine the output.

```
Number of features to drop: 1
Features with missing values greater than 80%: ['poutcome']
```

As expected, the `poutcome` feature has too many missing values, and should be dropped from the dataset.

e) Select the next code cell, then type the following:

```
users_data_cleaned = users_data.drop(cols_to_drop, axis = 1)
```

This code actually drops the feature.

f) Run the code cell.

g) Select the next code cell, then type the following:

```
# Confirm feature was dropped.

missing_value_df = missing_value_pct_df(users_data_cleaned)

missing_columns = \
list(missing_value_df[missing_value_df['percent_missing'] \
                      > 0]['column_name'])

print('Number of features with missing values:',
      len(missing_columns))
```

h) Run the code cell.

i) Examine the output.

```
Number of features with missing values: 7
```

As expected, only the one column was dropped.

6. Identify numerical data with missing values.

a) Scroll down and view the cell titled **Identify numerical data with missing values**, then select the code cell below it.
b) In the code cell, type the following:

```
dtypes = ['int64', 'float64']

numerical_columns = \
list(users_data_cleaned.select_dtypes(dtypes).columns)

print('Numerical features with missing values:',
      list(set(numerical_columns).intersection(missing_columns)))
```

There are still features with missing values that you won't drop completely, so you'll start investigating those.
c) Run the code cell.
d) Examine the output.

```
Numerical features with missing values: ['number_transactions', 'total_amount_usd']
```

Both `total_amount_usd` and `number_transactions` have missing values.

7. Impute missing values for `total_amount_usd`.

a) Scroll down and view the cell titled **Impute missing values for `total_amount_usd`**, then select the code cell below it.
b) In the code cell, type the following:

```
# Find a sample user with missing value

sample_user = \
users_data_cleaned[users_data_cleaned['total_amount_usd']. \
                   isnull()].sample(1).user_id

sample_user
```

This will just retrieve a single sample of a user with a missing `total_amount_usd` value.
c) Run the code cell.
d) Examine the output.

```
12481    d7a5acdd-5943-4ca6-9f31-d23ba1a1317c
Name: user_id, dtype: object
```

Note: Since `sample()` takes a random sample from the dataset, your sample will likely be different than what is shown in the screenshot.

e) Select the next code cell, then type the following:

```
# Print mean of total_amount_usd.

print('Mean total_amount_usd:',
      round(users_data_cleaned['total_amount_usd'].mean(), 2))

# Impute missing values for total_amount_usd with mean.

users_data_cleaned['total_amount_usd']. \
fillna(round(users_data_cleaned['total_amount_usd'].mean(), 2),
       inplace = True)
```

There are many ways to impute missing numeric values, but mean is a common approach. So, in this code, the missing values for `total_amount_usd` will be filled in with the mean of all non-missing values.

f) Run the code cell.

g) Examine the output.

```
Mean total_amount_usd: 1369.42
```

The mean used to fill in the missing values is $1,369.42.

h) Select the next code cell, then type the following:

```
users_data_cleaned[users_data_cleaned. \
                   user_id.isin(sample_user)]['total_amount_usd']
```

i) Run the code cell.

j) Examine the output.

```
12481    1369.42
Name: total_amount_usd, dtype: float64
```

The sample user from before now has $1,369.42 as their `total_amount_usd` value, as expected.

Note: The sample's record number should match the random sample you retrieved earlier, which will likely differ from this screenshot.

8. Replace missing values for `number_transactions` with `0`.

a) Scroll down and view the cell titled **Replace missing values for `number_transactions` with `0`**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data_cleaned['number_transactions']. \
fillna(0, inplace = True)
```

It doesn't always make sense to use something like mean imputation to fill in missing numeric values. In this case, it makes more sense to just set the number of transactions to `0`.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
users_data_cleaned[users_data_cleaned. \
                   user_id.isin(sample_user)]['number_transactions']
```

e) Run the code cell.

f) Examine the output.

```
12481    0.0
Name: number_transactions, dtype: float64
```

The sample user's `number_transactions` value is now `0.0`.

9. Identify categorical data with missing values.

a) Scroll down and view the cell titled **Identify categorical data with missing values**, then select the code cell below it.

b) In the code cell, type the following:

```
categorical_columns = \
list(users_data_cleaned.select_dtypes(['object']).columns)

print('Categorical features with missing values:',
      list(set(categorical_columns).intersection(missing_columns)))
```

Now that you've filled in the missing values for the numeric features, you can move on to the categorical features.

c) Run the code cell.

d) Examine the output.

```
Categorical features with missing values: ['job', 'device', 'education', 'contact']
```

The features `job`, `education`, `device`, and `contact` all have missing values.

10. Replace categorical missing values with `'Unknown'`.

a) Scroll down and view the cell titled **Replace categorical missing values with `'Unknown'`**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data_cleaned.device.fillna('Unknown', inplace = True)
users_data_cleaned.education.fillna('Unknown', inplace = True)
users_data_cleaned.contact.fillna('Unknown', inplace = True)
users_data_cleaned.job.fillna('Unknown', inplace = True)
```

For now, you'll simply replace all categorical missing values with the string `'Unknown'`.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
users_data_cleaned.device.value_counts()
```

e) Run the code cell.

f) Examine the output.

```
mobile     27504
desktop    12112
tablet      5499
Unknown       94
Name: device, dtype: int64
```

This confirms that the `device` feature now has 94 values marked as `'Unknown'`. The other three features underwent this same transformation.

11. Check if there are any other missing values.

a) Scroll down and view the cell titled **Check if there are any other missing values**, then select the code cell below it.

b) In the code cell, type the following:

```
missing_value_df = missing_value_pct_df(users_data_cleaned)
missing_columns = \
list(missing_value_df[missing_value_df['percent_missing'] \
                      > 0]['column_name'])

print('Number of features with missing values:', len(missing_columns))
print('Features with missing values:', missing_columns)
```

You're almost done handling missing values, but you need to make sure there aren't any stragglers.

c) Run the code cell.

d) Examine the output.

```
Number of features with missing values: 1
Features with missing values: ['date_joined']
```

There's still one column with missing values that you'll need to address: `date_joined`.

12. Remove all rows where `date_joined` is missing.

a) Scroll down and view the cell titled **Remove all rows where `date_joined` is missing**, then select the code cell below it.

b) In the code cell, type the following:

```
print('Number of users with corrupted data:',
      users_data_cleaned[users_data_cleaned['date_joined']. \
                         isnull()].shape[0])
```

You'd expect all users to have a date joined, so the presence of missing values could indicate data corruption. To be safe, you'll drop any rows that have missing join dates.

c) Run the code cell.

d) Examine the output.

```
Number of users with corrupted data: 30
```

Only 30 users have missing join dates.

e) Select the next code cell, then type the following:

```
# Remove corrupted data.

users_data_cleaned = \
users_data_cleaned[~users_data_cleaned['date_joined'].isnull()]
```

This code will actually remove the corrupted rows.

f) Run the code cell.

g) Select the next code cell, then type the following:

```
# Check to see if any corrupted rows remain.

print('Number of users with corrupted data:',
      users_data_cleaned[users_data_cleaned['date_joined']. \
                         isnull()].shape[0])
```

h) Run the code cell.

i) Examine the output.

```
Number of users with corrupted data: 0
```

All corrupted rows were removed.

13. Perform one last check for missing values.

a) Scroll down and view the cell titled **Perform one last check for missing values**, then select the code cell below it.

b) In the code cell, type the following:

```
missing_value_df = missing_value_pct_df(users_data_cleaned)
missing_columns = \
list(missing_value_df[missing_value_df['percent_missing'] \
                      > 0]['column_name'])

print('Number of features with missing values:',
      len(missing_columns))
```

Just to be sure, you'll do one final check.

c) Run the code cell.

d) Examine the output.

```
Number of features with missing values: 0
```

All of the missing values have been removed from the dataset.

14. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 3-9

Applying Transformation Functions to a Dataset

Data File

~/Analysis/Analyzing Data.ipynb

Scenario

You've seen that a lot of the numeric features in the dataset are skewed. Adjusting this skewness can better facilitate the model training process. These types of adjustments can be dependent on the type of model you're training, so for now, you'll just temporarily transform the `age` feature to get a sense of how such transformations can help mitigate distribution issues. Later, when you start building machine learning models, you'll apply other transformations to the dataset.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

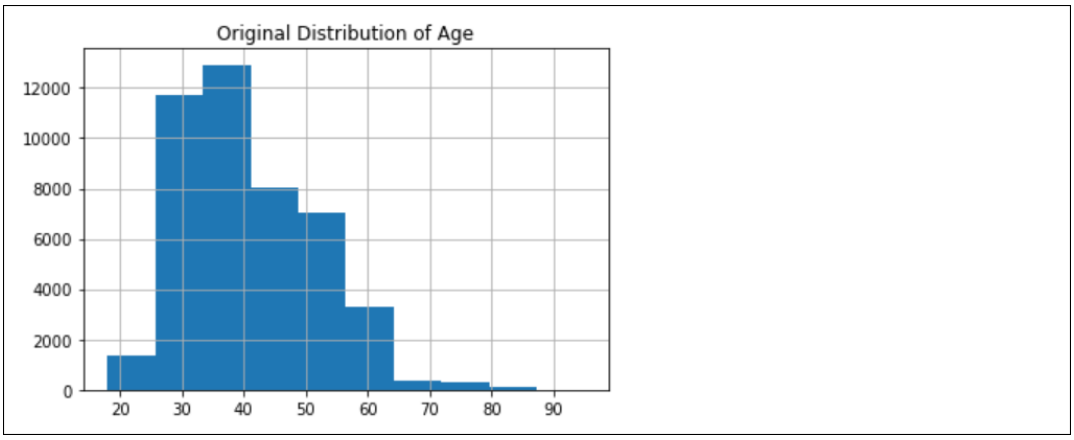
Note: If you encounter errors when running the next code blocks, select the **View the distribution of `age`** heading, then select **Cell→Run All Above**.

2. View the distribution of `age`.
   a) Scroll down and view the cell titled **View the distribution of `age`**, then select the code cell below it.
   b) In the code cell, type the following:

```
users_data_cleaned['age'].hist()
plt.title('Original Distribution of Age');
```

   While `age` is less skewed than a lot of the other features, it still exhibits some degree of skewness than you can address through transformations. This code will show a histogram of `age` as a refresher.
   c) Run the code cell.

d) Examine the output.



This shows a left (positive) skew. Recall that you identified outliers above 70 years of age.

3. Apply a log transformation to `age`.

a) Scroll down and view the cell titled **Apply a log transformation to `age`**, then select the code cell below it.

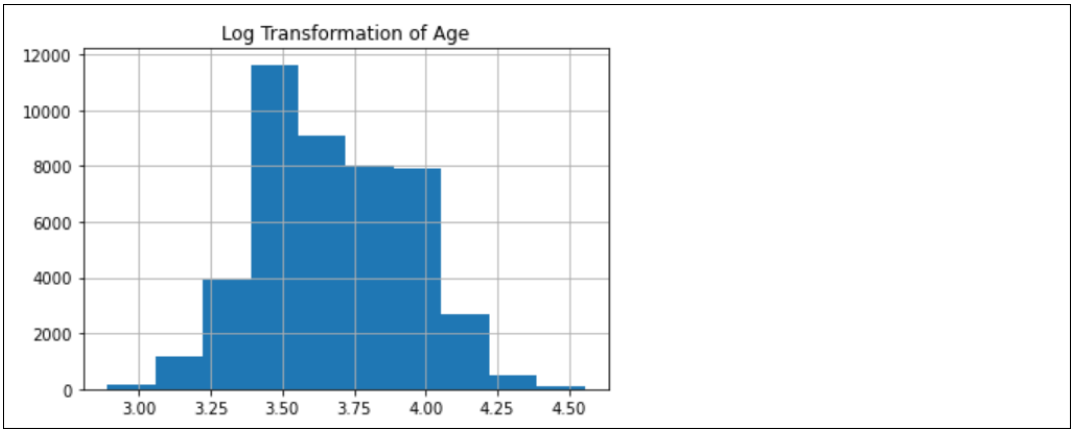
b) In the code cell, type the following:

```
np.log(users_data_cleaned['age']).hist()
plt.title('Log Transformation of Age');
```

First you'll apply a log transformation to the `age` feature.

c) Run the code cell.

d) Examine the output.



As you can see, `age` is now closer to a normal distribution. The distribution no longer trails off toward the high end of values.

4. Apply a Box–Cox transformation to `age`.

a) Scroll down and view the cell titled **Apply a Box–Cox transformation to `age`**, then select the code cell below it.

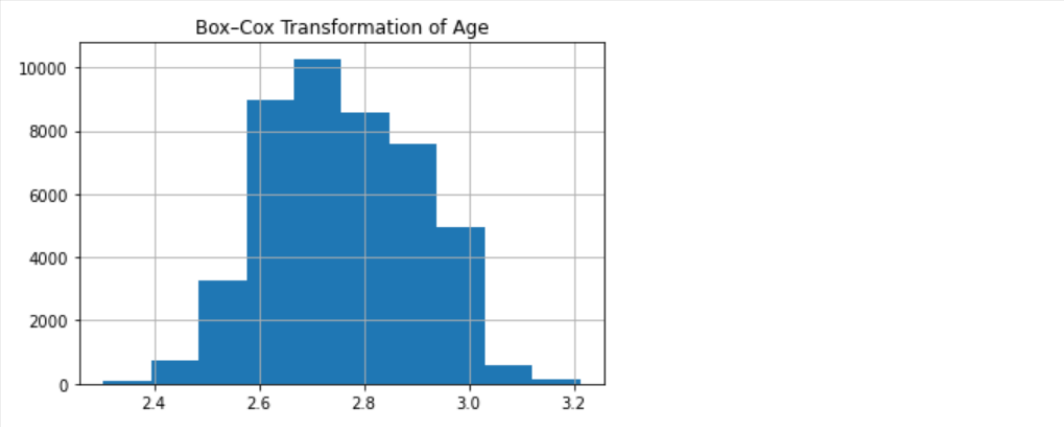
b) In the code cell, type the following:

```
from scipy import stats

pd.Series(stats.boxcox(users_data_cleaned['age'])[0]).hist()
plt.title('Box–Cox Transformation of Age');
```

You'll also apply a Box–Cox transformation to `age` to see how its results differ from the log transformation.

c) Run the code cell.

d) Examine the output.



Like the log transformation, the Box–Cox transformation moved the data closer to a normal distribution. However, there are differences in each technique's outcome. One is not necessarily better than the other in every scenario.

5. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 3-10
Encoding Data

Data File

~/Analysis/Analyzing Data.ipynb

Scenario

Since you plan to input this data into multiple machine learning algorithms, you need to make sure the data is actually in a form that those algorithms can read. Many machine learning algorithms can't deal with categorical features that use string values, so those values need to be encoded as numbers. For the most part, you'll use the common one-hot encoding technique to ensure that the features are properly formatted for machine learning.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Identify categorical features** heading, then select **Cell→Run All Above**.

2. Identify categorical features.
   a) Scroll down and view the cell titled **Identify categorical features**, then select the code cell below it.
   b) In the code cell, type the following:

```
categorical_columns = \
list(users_data_cleaned.select_dtypes(['object']).columns)

print('The number of categorical features:',
      len(categorical_columns))
print('The names of categorical features:',
      categorical_columns)
```

   You'll start by taking stock of your categorical features.
   c) Run the code cell.
   d) Examine the output.

```
The number of categorical features: 6
The names of categorical features: ['user_id', 'job', 'marital', 'education', 'contact', 'device']
```

   There are 6 total categorical features, though there are actually 5, since you can disregard `user_id`.

3. One-hot encode `job`.
   a) Scroll down and view the cell titled **One-hot encode `job`**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data_cleaned.job.value_counts(dropna = True)
```

This code will provide a refresher on the number of values in the `job` feature.

c) Run the code cell.

d) Examine the output.

```
blue-collar      9725
management       9453
technician       7592
admin.           5168
services         4152
retired          2262
self-employed    1577
entrepreneur     1485
unemployed       1301
housemaid        1239
student           937
Unknown           288
Name: job, dtype: int64
```

There are 12 total types of jobs (including unknown).

e) Select the next code cell, then type the following:

```
# Create object for one-hot encoding.

encoder = ce.OneHotEncoder(cols = 'job',
                           return_df = True,
                           use_cat_names = True)
```

The `job` feature has no natural order, so it's a good candidate for one-hot encoding. Each value will be mapped to its own feature, where `0` indicates absence and `1` indicates presence. The `encoder` object will be used to do the actual encoding on the dataset.

f) Run the code cell.

g) Select the next code cell, then type the following:

```
# Fit and transform data.

users_data_encoded = encoder.fit_transform(users_data_cleaned)

# Preview the data.

users_data_encoded.head()
```

This code will use the `encoder` object to do the actual transformation.

h) Run the code cell.

i) Examine the output.

ount_usd	age	job_management	job_technician	job_entrepreneur	job_blue-collar	job_Unknown	job_retired	...	housing	loan	contact	duration	campaign	pda
2143.00	58	1	0	0	0	0	0	...	True	False	Unknown	261	1	
1369.42	44	0	1	0	0	0	0	...	True	False	Unknown	151	1	
2.00	33	0	0	1	0	0	0	...	True	True	Unknown	76	1	
1369.42	47	0	0	0	1	0	0	...	True	False	Unknown	92	1	
1.00	33	0	0	0	0	1	0	...	False	False	Unknown	198	1	

You can see various new columns in the dataset, called `job_x` where `x` is one of the 12 values. Each row has a `0` in every `job_x` column except for one, in which it has a `1` value. For example, the first user in the dataset is a manager.

j) Select the next code cell, then type the following:

```
1 list(users_data_encoded)
```

k) Run the code cell.

l) Examine the output.

```
['user_id',
 'number_transactions',
 'total_amount_usd',
 'age',
 'job_management',
 'job_technician',
 'job_entrepreneur',
 'job_blue-collar',
 'job_Unknown',
 'job_retired',
 'job_admin.',
 'job_services',
 'job_self-employed',
 'job_unemployed',
 'job_housemaid',
 'job_student',
 'marital',
 'education',
 'default',
 'housing',
 'loan',
 'contact',
 'duration',
 'campaign',
 'pdays',
 'previous',
 'term_deposit',
 'date_joined',
 'device']
```

Here's another view of all the new features.

m) Select the next code cell, then type the following:

```
print('Shape of data before encoding:',
      users_data_cleaned.shape)
print('Shape of data after encoding:',
      users_data_encoded.shape)
```

n) Run the code cell.

o) Examine the output.

```
Shape of data before encoding: (45179, 18)
Shape of data after encoding: (45179, 29)
```

As expected, 11 new columns were added. (The original `job` column was removed.)

4. Dummy encode `marital`.

a) Scroll down and view the cell titled **Dummy encode `marital`**, then select the code cell below it.

b) In the code cell, type the following:

```
marital_encoded = \
pd.get_dummies(data = users_data_encoded['marital'],
               drop_first = True)

marital_encoded
```

The `marital` feature will undergo a similar treatment as `job`, except the first values will be removed and will not become a new feature. Some practitioners call this dummy encoding in order to distinguish it from one-hot encoding.

c) Run the code cell.

d) Examine the output.

	married	single
0	1	0
1	0	1
2	1	0
3	1	0
4	0	1
...	...	...
45211	1	0
45212	0	0
45213	1	0
45214	1	0
45215	1	0

45179 rows × 2 columns

Since the first value of `marital` is `divorced`, it did not become a new feature, whereas `married` and `single` did.

e) Select the next code cell, then type the following:

```
# Concatenate the new encoded columns.

users_data_encoded = \
pd.concat([users_data_encoded, marital_encoded], axis = 1)

# Drop the original variable.

users_data_encoded.drop(['marital'], axis = 1, inplace = True)

# Preview the data

users_data_encoded.head()
```

This code adds the new dummy-encoded features to the dataset, while dropping the original `marital` feature.

f) Run the code cell.

g) Examine the output.

job_entrepreneur	job_blue-collar	job_Unknown	job_retired	...	contact	duration	campaign	pdays	previous	term_deposit	date_joined	device	married	single
0	0	0	0	...	Unknown	261	1	-1	0	False	1998-08-23	mobile	1	0
0	0	0	0	...	Unknown	151	1	-1	0	False	2008-07-15	desktop	0	1
1	0	0	0	...	Unknown	76	1	-1	0	False	2002-06-04	mobile	1	0
0	1	0	0	...	Unknown	92	1	-1	0	False	1995-06-29	tablet	1	0
0	0	1	0	...	Unknown	198	1	-1	0	False	1995-08-01	mobile	0	1

At the right side of the table, you can see the two new features.

h) Select the next code cell, then type the following:

```
print('Shape of data after encoding:',
      users_data_encoded.shape)

list(users_data_encoded)
```

i) Run the code cell.

j) Examine the output.

```
Shape of data after encoding: (45179, 30)

['user_id',
 'number_transactions',
 'total_amount_usd',
 'age',
 'job_management',
 'job_technician',
 'job_entrepreneur',
 'job_blue-collar',
 'job_Unknown',
 'job_retired',
 'job_admin.',
 'job_services',
 'job_self-employed',
 'job_unemployed',
 'job_housemaid',
 'job_student',
 'education',
 'default',
 'housing',
 'loan',
 'contact',
 'duration',
 'campaign',
 'pdays',
 'previous',
 'term_deposit',
 'date_joined',
 'device',
 'married',
 'single']
```

There are now a total of 30 columns.

5. One-hot encode the remaining categorical variables.

a) Scroll down and view the cell titled **One-hot encode the remaining categorical variables**, then select the code cell below it.

b) In the code cell, type the following:

```
cols = ['education', 'contact', 'device']

encoder = ce.OneHotEncoder(cols = cols,
                           return_df = True,
                           use_cat_names = True)
```

You're creating another `encoder` object, this time for the remaining three categorical variables.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
# Fit and transform data.

users_data_encoded = encoder.fit_transform(users_data_encoded)

# Preview the data.

users_data_encoded.head()
```

e) Run the code cell.

f) Examine the output.

ue-lar	job_Unknown	job_retired	...	pdays	previous	term_deposit	date_joined	device_mobile	device_desktop	device_tablet	device_Unknown	married	single
0	0	0	...	-1	0	False	1998-08-23	1	0	0	0	1	0
0	0	0	...	-1	0	False	2008-07-15	0	1	0	0	0	1
0	0	0	...	-1	0	False	2002-06-04	1	0	0	0	1	0
1	0	0	...	-1	0	False	1995-06-29	0	0	1	0	1	0
0	1	0	...	-1	0	False	1995-08-01	1	0	0	0	0	1

You can see some of the new `device_x` columns, but since there are so many, they get truncated in output.

g) Select the next code cell, then type the following:

```
print('Shape of data after encoding:',
      users_data_encoded.shape)

list(users_data_encoded)
```

h) Run the code cell.

i) Examine the output.

```
Shape of data after encoding: (45179, 38)

['user_id',
 'number_transactions',
 'total_amount_usd',
 'age',
 'job_management',
 'job_technician',
 'job_entrepreneur',
 'job_blue-collar',
 'job_Unknown',
 'job_retired',
 'job_admin.',
 'job_services',
 'job_self-employed',
 'job_unemployed',
 'job_housemaid',
 'job_student',
 'education_tertiary',
 'education_secondary',
 'education_Unknown',
 'education_primary',
 'default',
 'housing',
 'loan',
 'contact_Unknown',
 'contact_cellular',
 'contact_telephone',
 'duration',
 'campaign'
```

The dataset now has 38 total columns, which is more than double its original size.

6. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.
b) Close the lab browser tab and continue on with the course.

LAB 3-11
Discretizing Variables

Data File

~/Analysis/Analyzing Data.ipynb

Scenario

When you eventually build machine learning models using certain types of algorithms, you'll need to transform your continuous variables into discrete ones. You'll start with `age`, which you'll place into one of several bins. These bins will act as categories to a new feature, which you'll use to replace the original continuous `age` variable.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Discretize `age` into bins** heading, then select **Cell→Run All Above**.

2. Discretize `age` into bins.
   a) Scroll down and view the cell titled **Discretize `age` into bins**, then select the code cell below it.
   b) In the code cell, type the following:

```
1 users_data_encoded.age.describe()
```

You'll get a quick look at the distribution of `age` again to help you determine how to bin it. You could also look at a box plot to get a similar idea.

   c) Run the code cell.
   d) Examine the output.

```
count    45179.000000
mean        40.935103
std         10.618499
min         18.000000
25%         33.000000
50%         39.000000
75%         48.000000
max         95.000000
Name: age, dtype: float64
```

Since the minimum age is 18 and the maximum is 95, it might be a good idea to bin in gaps of 10. This is an example of equal-width binning. You can also place the outliers into a wider bin that captures anyone above 75. There's no objectively "correct" way to bin, so this is just one way you could do it.

e) Select the next code cell, then type the following:

```
# Define age bins and labels.

bins = [18, 25, 35, 45, 55, 65, 75, 110]
labels = ['18–24', '25–34', '35–44',
          '45–54', '55–64', '65–74', '75+']

# Perform binning using bin list.

users_data_encoded['age_group'] = \
pd.cut(users_data_encoded['age'], bins = bins,
       labels = labels, right = False)

# Map bins to integer values.
users_data_encoded['age_group_encoded'] = \
users_data_encoded['age_group'].cat.codes
```

To save you some typing, the `bins` list has already been created for you. There's also a `labels` list that you'll use to make visuals easier to read. The code you'll type includes a call to `cut()`, which does the actual segmentation of the continuous variable into bins. Below that, you're also mapping the bins to integer values, so that the first bin is `0`, the second is `1`, and so on.

f) Run the code cell.

g) Select the next code cell, then type the following:

```
# Verify correct binning.

age_vars = ['age_group_encoded', 'age_group', 'age']

users_data_encoded[age_vars].sample(10)
```

h) Run the code cell.

i) Examine the output.

	age_group_encoded	age_group	age
21005	4	55–64	55
19010	2	35–44	42
30593	4	55–64	64
44973	1	25–34	27
8851	0	18–24	24
26435	2	35–44	37
20102	3	45–54	53
19892	1	25–34	34
20748	3	45–54	49
22036	2	35–44	38

The `age_group_encoded` column has integer values for each bin, which each map to an `age_group`. Everything should appear correct when compared to the actual `age`.

Note: Your random sampling will likely differ from the screenshot, but it should nonetheless help you verify the binning was performed correctly.

3. Plot the new distribution of `age`.

a) Scroll down and view the cell titled **Plot the new distribution of `age`**, then select the code cell below it.

b) In the code cell, type the following:

```
user_age_dist = users_data_encoded.age_group.value_counts()

user_age_dist
```

c) Run the code cell.

d) Examine the output.

```
35–44    14524
25–34    14194
45–54     9951
55–64     4892
18–24      809
65–74      510
75+        299
Name: age_group, dtype: int64
```

Each bin has a set number of values, with the 35–44 bin being the largest, with 25–34 following close behind.

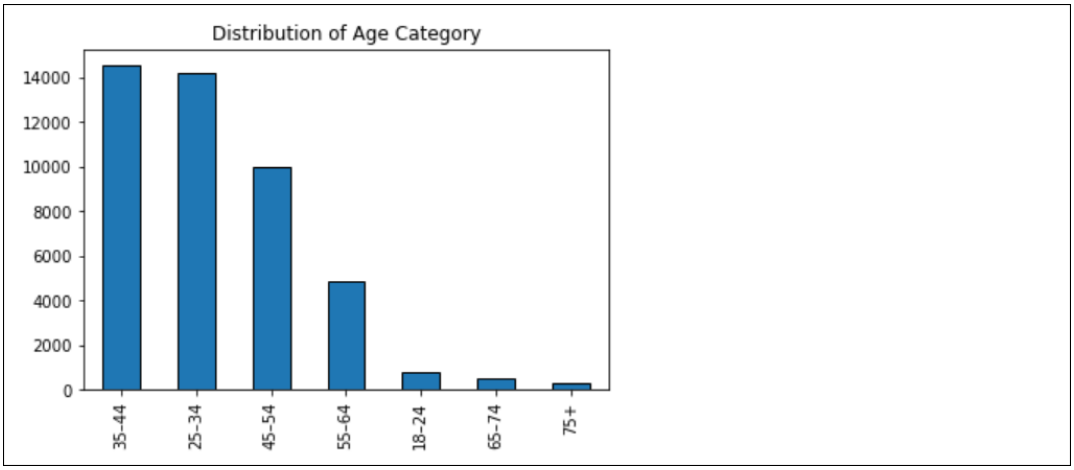
e) Select the next code cell, then type the following:

```
user_age_dist.plot(kind = 'bar', edgecolor = 'black')
plt.title('Distribution of Age Category');
```

To get a better sense of the distribution, you'll plot it visually using a bar chart. Since you're no longer working with a continuous variable, plotting a histogram wouldn't make sense.

f) Run the code cell.

g) Examine the output.



You can see the different age bins that are represented in the dataset. For example, the youngest age range (18–24) seems to have a low amount of representation.

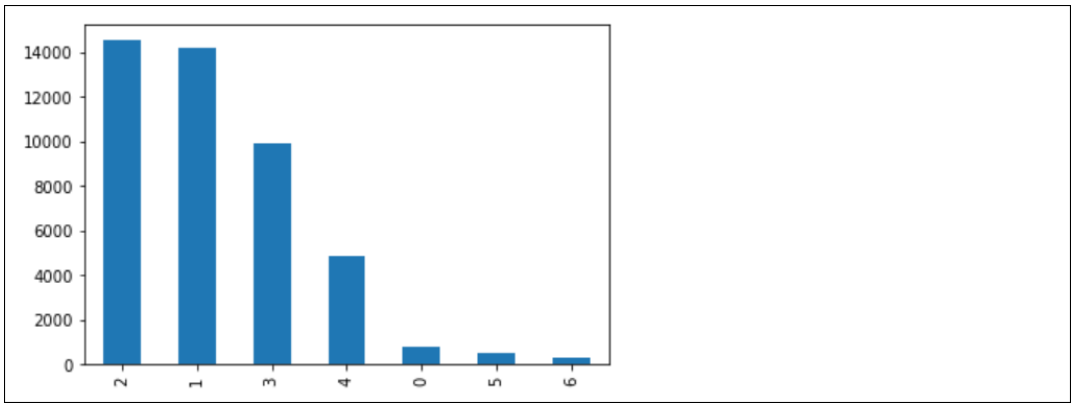
h) Select the next code cell, then type the following:

```
# Check against encoded values.

users_data_encoded.age_group_encoded. \
value_counts().plot(kind = 'bar');
```

Just to make sure the bins are correct, you'll plot the raw integer values for the bins rather than just their labels.

i) Run the code cell.

j) Examine the output.



The integer values should line up as expected. For example, bin `0` (18–24) is still third from the right.

4. Drop the `age` and `age_group` variables.

a) Scroll down and view the cell titled **Drop the `age` and `age_group` variables**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data_encoded.drop(['age', 'age_group'],
                        axis = 1, inplace = True)

list(users_data_encoded)
```

You'll drop the original `age` variable and the label-based `age_group` variable since they're no longer needed. Only the integer-based `age_group_encoded` variable will remain as a feature.

c) Run the code cell.

d) Examine the output.

```
 'loan',
 'contact_Unknown',
 'contact_cellular',
 'contact_telephone',
 'duration',
 'campaign',
 'pdays',
 'previous',
 'term_deposit',
 'date_joined',
 'device_mobile',
 'device_desktop',
 'device_tablet',
 'device_Unknown',
 'married',
 'single',
 'age_group_encoded']
```

You can see `age_group_encoded` at the bottom of the list of columns.

5. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 3-12
Splitting and Removing Features

Data File

~/Analysis/Analyzing Data.ipynb

Scenario

Although the `date_joined` feature is useful, machine learning algorithms don't always work well with complex dates and times. You'll simplify the data by extracting just the months from this feature, and then make that its own feature. This could be useful for identifying things like churn rate, e.g., perhaps there's a higher churn rate in summer months than in winter.

You also want to see what remaining features you can potentially remove to improve the dataset's viability. Earlier you started identifying correlations between pairs of variables, so now you'll actually start dropping one variable in each of those high-correlation pairs. This will hopefully remove redundancy in the dataset. Likewise, you want to identify any features that exhibit low variance and drop them. Features with very low variance have values that are so similar they may not contain any truly useful information.

Lastly, with all of the preprocessing done on this dataset, you'll finally be ready to load it into a data file that you'll use with machine learning.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Create a `month_joined` variable from `date_joined`** heading, then select **Cell→Run All Above**.

2. Create a `month_joined` variable from `date_joined`.
   a) Scroll down and view the cell titled **Create a `month_joined` variable from `date_joined`**, then select the code cell below it.
   b) In the code cell, type the following:

```
users_data_encoded['month_joined'] = \
users_data_encoded.date_joined.dt.month
```

   This code leverages the datetime type to extract just the month values.
   c) Run the code cell.

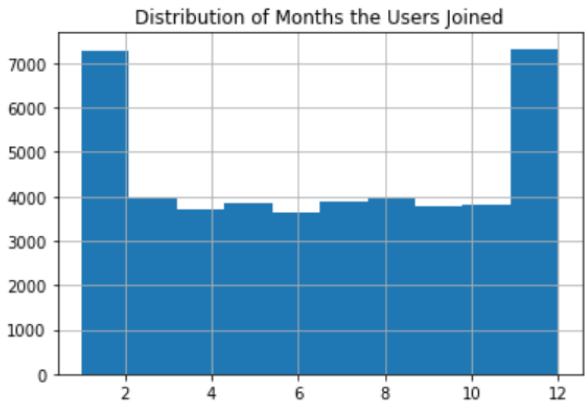
d) Select the next code cell, then type the following:

```
# View the distribution of data.

users_data_encoded['month_joined'].hist()
plt.title('Distribution of Months the Users Joined');
```

You'll take a look at the distribution of the new `month_joined` feature.

e) Run the code cell.

f) Examine the output.



It looks like there's a spike in January and December, indicating that most users seemed to join the bank in those months.

g) Select the next code cell, then type the following:

```
users_data_encoded.drop(['date_joined'],
                        axis = 1, inplace = True)

list(users_data_encoded)
```

Since the original `date_joined` feature is no longer needed, you'll drop it.

h) Run the code cell.

i) Examine the output.

```
'duration',
'campaign',
'pdays',
'previous',
'term_deposit',
'device_mobile',
'device_desktop',
'device_tablet',
'device_Unknown',
'married',
'single',
'age_group_encoded',
'month_joined']
```

The `month_joined` feature is in the list of columns.

3. Remove features with low variance.

a) Scroll down and view the cell titled **Remove features with low variance**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data_encoded.std()
```

You're retrieving the standard deviations of each feature. Remember that standard deviation is just the square root of variance.

c) Run the code cell.

d) Examine the output.

```
number_transactions       3.749994
total_amount_usd       2704.291321
job_management            0.406767
job_technician            0.373908
job_entrepreneur          0.178296
job_blue-collar           0.411004
job_Unknown               0.079587
job_retired               0.218087
job_admin.                0.318287
job_services              0.288889
job_self-employed         0.183543
job_unemployed            0.167236
job_housemaid             0.163318
job_student               0.142513
education_tertiary        0.455691
education_secondary       0.499832
education_Unknown         0.198480
education_primary         0.358591
default                   0.133095
housing                   0.496878
loan                      0.366802
contact_Unknown           0.452851
contact_cellular          0.477695
contact_telephone         0.245250
```

Some features have a high standard deviation, though most seem to have a standard deviation below 1. You want to drop any features with very low values.

e) Select the next code cell, then type the following:

```
# Define standard deviation threshold.

threshold = 0.1

# Identify features below threshold.

cols_to_drop = \
list(users_data_encoded.std()[users_data_encoded.std() \
                              < threshold].index.values)

print('Features with low standard deviation:',
      cols_to_drop)
```

The threshold is used to define features with very low variance; i.e., any feature with a standard deviation less than 0.1. This is a somewhat arbitrary number, as there is no objectively "correct" threshold for every use case. But it's small enough not to drop too many features that could be useful.

f) Run the code cell.

g) Examine the output.

```
Features with low standard deviation: ['job_Unknown', 'device_Unknown']
```

The `job_Unknown` and `device_Unknown` features exhibit low variance. Recall that these are one-hot encoded features from missing categorical values that you filled in with the string `'Unknown'`.

h) Select the next code cell, then type the following:

```
# Drop features below threshold.

users_data_interim = users_data_encoded.drop(cols_to_drop,
                                             axis = 1)

list(users_data_interim)
```

i) Run the code cell.

j) Examine the output.

```
['user_id',
 'number_transactions',
 'total_amount_usd',
 'job_management',
 'job_technician',
 'job_entrepreneur',
 'job_blue-collar',
 'job_retired',
 'job_admin.',
 'job_services',
 'job_self-employed',
 'job_unemployed',
 'job_housemaid',
 'job_student',
 'education_tertiary',
 'education_secondary'
```

Both of the identified columns have been removed.

4. Drop highly correlated features.

a) Scroll down and view the cell titled **Drop highly correlated features**, then select the code cell below it.

b) In the code cell, type the following:

```
# Define correlation threshold.

threshold = 0.75

corr_matrix = users_data_encoded.corr().abs()
high_corr_var = np.where(corr_matrix >= threshold)
high_corr_var = [(corr_matrix.index[x],
                  corr_matrix.columns[y],
                  round(corr_matrix.iloc[x, y], 2))
                  for x, y in zip(*high_corr_var)
                  if x != y and x < y]


high_corr_var
```

You're defining another threshold, this time for correlation coefficient. The `high_corr_var` object will check for pairs of features that exhibit correlation above the threshold.

c) Run the code cell.

d) Examine the output.

```
[('contact_Unknown', 'contact_cellular', 0.86),
 ('device_mobile', 'device_desktop', 0.75),
 ('married', 'single', 0.77)]
```

The results indicate that:

- `contact_Unknown` and `contact_cellular` are highly correlated with a value of 0.86.
- `device_mobile` and `device_desktop` are highly correlated with a value of 0.75.
- `married` and `single` are highly correlated with a value of 0.77.

e) Select the next code cell, then type the following:

```
# Tidy up the output.

record_collinear = pd.DataFrame(high_corr_var). \
rename(columns = {0: 'drop_feature',
                  1: 'corr_feature',
                  2: 'corr_values'})

record_collinear = record_collinear. \
sort_values(by = 'corr_values', ascending = False)

record_collinear = record_collinear.reset_index(drop = True)

record_collinear
```

This code will make the output a little easier to read, helping you identify which feature in the pair to drop, which to keep, and what the correlation coefficient for the pairing is.

f) Run the code cell.

g) Examine the output.

	drop_feature	corr_feature	corr_values
0	contact_Unknown	contact_cellular	0.86
1	married	single	0.77
2	device_mobile	device_desktop	0.75

h) Select the next code cell, then type the following:

```
cols_to_drop = list(record_collinear['drop_feature'])
print(cols_to_drop)
```

i) Run the code cell.

j) Examine the output.

```
['contact_Unknown', 'married', 'device_mobile']
```

The `contact_Unknown`, `married`, and `device_mobile` features will be dropped.

k) Select the next code cell, then type the following:

```
users_data_final = users_data_interim.drop(cols_to_drop,
                                           axis = 1)

list(users_data_final)
```

l) Run the code cell.

m) Examine the output.

```
['user_id',
 'number_transactions',
 'total_amount_usd',
 'job_management',
 'job_technician',
 'job_entrepreneur',
 'job_blue-collar',
 'job_retired',
 'job_admin.',
 'job_services',
 'job_self-employed',
 'job_unemployed',
 'job_housemaid',
 'job_student',
 'education_tertiary',
 'education_secondary',
 'education_Unknown'
```

The three features have been removed.

5. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 3-13
Performing Dimensionality Reduction

Data File

~/Analysis/Analyzing Data.ipynb

Scenario

Your users dataset now has dozens of features, which will likely be useful in many machine learning scenarios. However, some machine learning tasks are more effective when they work with a reduced feature set. Likewise, there are times when you'll want to reduce your dataset's dimensionality to make it more conducive to plotting. There are only so many dimensions you can represent on a scatter plot, for instance.

So, you'll apply principal component analysis (PCA) to a subset of your data—demographics information, in particular—to reduce its dimensionality. Because most of your work will be done on the full users dataset, you'll save this as a separate dataset that you can call up whenever you need it.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Analysis/Analyzing Data.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Filter by demographics data** heading, then select **Cell→Run All Above**.

2. Filter by demographics data.
   a) Scroll down and view the cell titled **Filter by demographics data**, then select the code cell below it.
   b) In the code cell, type the following:

```
users_data_demographics = \
users_data_final.filter(regex = 'education|job|age|single')

users_data_demographics.head(n = 3)
```

To simplify things a bit, you'll filter the data to only include demographic features such as education level, job type, age, and marital status. The regular expression will capture all of these, so you don't need to specify each column name separately.

   c) Run the code cell.
   d) Examine the output.

	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services	job_self-employed	job_unemployed	job_housemaid	job_stude
0	1	0	0	0	0	0	0	0	0	0	
1	0	1	0	0	0	0	0	0	0	0	
2	0	0	1	0	0	0	0	0	0	0	

As expected, the resulting `DataFrame` only includes demographic information.

3. Standardize the demographics data.

a) Scroll down and view the cell titled **Standardize the demographics data**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data_demographics.describe()
```

c) Run the code cell.

d) Examine the output.

	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services	job_self-employed	job_unemployed	job_hous
count	45179.000000	45179.000000	45179.000000	45179.000000	45179.000000	45179.000000	45179.000000	45179.000000	45179.000000	45179.0
mean	0.209234	0.168043	0.032869	0.215255	0.050068	0.114389	0.091901	0.034906	0.028797	0.0
std	0.406767	0.373908	0.178296	0.411004	0.218087	0.318287	0.288889	0.183543	0.167236	0.1
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
75%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.0

All of the variables appear to be on the same scale (ranging from 0 to 1) except for `age_group_encoded`, which ranges from 0 to 6. Remember that these values correspond to the age range bins that you defined earlier. To ensure every feature is on the same scale before you perform PCA, you'll apply standardization.

e) Select the next code cell, then type the following:

```
scaler = StandardScaler()

scaler.fit(users_data_demographics)
users_data_scaled = scaler.transform(users_data_demographics)

print('New standard deviation: ', users_data_scaled.std())
print('New mean:               ', round(users_data_scaled.mean()))
```

This code standardizes the data so that the mean of the distribution is 0 and the standard deviation is 1.

f) Run the code cell.

g) Examine the output.

```
New standard deviation:  1.0
New mean:                0
```

As expected, the data has been scaled using the z-score.

4. Perform PCA to reduce the dimensionality of the demographics dataset.

a) Scroll down and view the cell titled **Perform PCA to reduce the dimensionality of the demographics dataset**, then select the code cell below it.

b) In the code cell, type the following:

```
pca = PCA(n_components = 2, random_state = 1)

pca.fit(users_data_scaled)

reduced = pca.transform(users_data_scaled)
```

You'll perform PCA to derive only 2 total features from the existing demographics data, which has 17 features.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
reduced_df = pd.DataFrame(reduced, columns = ['PCA1', 'PCA2'])

reduced_df
```

Each of the derived features won't necessarily have a qualitative meaning, so you're assigning them the generic labels `PCA1` and `PCA2`.

e) Run the code cell.

f) Examine the output.

	PCA1	PCA2
0	2.557545	1.079613
1	-0.820505	-1.750581
2	-0.576607	-0.551404
3	-0.541647	1.593626
4	0.385598	-0.614768
...	...	...
45174	1.160485	0.073978
45175	-0.660738	4.371075
45176	-1.327676	2.367745
45177	-1.575612	1.040206
45178	-0.632467	-0.134990

45179 rows × 2 columns

All of the rows still represent users, but the columns have changed. The 17 demographics columns have been reduced to only 2. The numeric values in these columns were mathematically derived from the values that were in the 17 demographics columns. The idea is that these two columns now represent an approximation of the initial feature space. The relationships between the records and the features are mostly preserved, but are now in a more compact form. The values may not have much meaning to a human observer, but they can still be meaningful to a machine learning algorithm.

A reduced dataset like this can be applied to a number of different problems. You'll lean on this dataset later, when you start clustering customers based on their demographic information.

5. Load the final dataset.

a) Scroll down and view the cell titled **Load the final dataset**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data_final.info()
```

Now it's time to load all of the main user data you just preprocessed into a file. First you'll do a spot check of the data to ensure the columns all have the correct data type and number of values.

c) Run the code cell.

d) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 45179 entries, 0 to 45215
Data columns (total 33 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   user_id              45179 non-null  object
 1   number_transactions  45179 non-null  float64
 2   total_amount_usd     45179 non-null  float64
 3   job_management       45179 non-null  int64
 4   job_technician       45179 non-null  int64
 5   job_entrepreneur     45179 non-null  int64
 6   job_blue-collar      45179 non-null  int64
 7   job_retired          45179 non-null  int64
 8   job_admin.           45179 non-null  int64
 9   job_services         45179 non-null  int64
 10  job_self-employed    45179 non-null  int64
 11  job_unemployed       45179 non-null  int64
 12  job_housemaid        45179 non-null  int64
 13  job_student          45179 non-null  int64
 14  education_tertiary   45179 non-null  int64
 15  education_secondary  45179 non-null  int64
 16  education_Unknown    45179 non-null  int64
 17  education_primary    45179 non-null  int64
```

All columns apart from `user_id` are either Booleans, integers, or floats. And, all columns have 45,179 values—so no more data is missing.

e) Select the next code cell, then type the following:

```
users_data_final.to_pickle('users_data_final.pickle')
```

You'll save the data as a pickle file to make it easier to import back into Jupyter Notebook.

f) Run the code cell.

6. Load the demographics dataset with PCA applied.

a) Scroll down and view the cell titled **Load the demographics dataset with PCA applied**, then select the code cell below it.

b) In the code cell, type the following:

```
reduced_df.to_pickle('users_data_demo_pca.pickle')
```

You'll save the demographics data with PCA applied as its own file so that it can be retrieved separately when you need it. For the most part, you'll be using the **users_data_final.pickle** file to build your models.

c) Run the code cell.

7. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on with the course.

MODULE 5

Project

The following lab is for the Course 3 Project.

PROJECT 3
Online Retailer: Analyzing Data

Data File

~/Project/Analysis Project.ipynb

~/Project/data/online_history_cleaned.pickle

Scenario

You work for an online retailer that sells a wide variety of different items to consumers. Each sale through the online storefront is recorded in a database with various characteristics, including invoice number, product stock code, quantity purchased, sale price, description, etc. As part of your data science business project, your team has been tasked with creating models that can give the business more insights into its customers' behavior, and even predict future purchasing decisions.

The sales data has already gone through the ETL process. Now it's time to analyze that data to get a more comprehensive look at your data and what it reveals about the business. You'll also perform preprocessing on the data to make sure it's ready to be used to train machine learning models later on.

Note: There is no "correct" answer for every line of code, as there are many ways to write code to do the same thing. You can write the code however you feel most comfortable, as long as it accomplishes the tasks set out for you.

Note: If you're stuck and need help, you can "borrow" code from existing notebooks, and/or look up documentation on how to use the various Python libraries.

Note: Sometimes, when you run code containing logic errors or bugs, you may corrupt the data contained in variables you created in previous code cells. To clear out such problems, you can select **Kernel→Restart & Clear Output**, then run each code cell again.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Project/Analysis Project.ipynb** to open it.
   c) Observe the notebook.

      Placeholder code cells have been provided in which you can add your own code. Comments provide hints on tasks you might perform in each code cell. The first code cell has already been completely programmed for you.

2. Import software libraries.
   a) Select the code listing under **Import software libraries**.
   b) Run the code and examine the results.

3. Read and examine the data.
   a) In the first code cell under **Read and examine the data**, write code to read the **online_history_cleaned.pickle** file in the **data** folder, then get the first five rows.
   b) Run the code cell and verify the first five rows of the dataset appear.

Note: This dataset comes from the UK, where "stock" is equivalent to "inventory" in American English. The stock code is the item number/identifier and is not to be confused with "stock" in the sense of a financial share of a company.

c) In the next code cell, write code to get the shape of the data.
d) Run the code cell and verify the number of rows and columns in the dataset.
e) In the next code cell, write code to get the data types for every column in the dataset.
f) Run the code cell and verify that each column has the proper data type.

`Quantity` should be an integer, `Price` and `TotalAmount` should be floats, `InvoiceDate` should be a datetime, and the rest should be string objects.

4. Generate summary statistics for all of the data.

a) In the code cell under **Generate summary statistics for all of the data**, write code to generate a `DataFrame` with all of the summary statistics, including any non-numeric variables.
b) Run the code cell that verify the number of unique customers, invoices, stock codes, and countries.

5. Plot a bar chart for the average price per item.

a) In the code cell under **Plot a bar chart for the average price per item**, write code to plot a bar chart of average price for each item. Ensure the chart has a title and axis labels, and that whatever axis you put the item descriptions on displays them vertically so they don't overlap.
b) Run the code cell and verify which items have the highest and lowest average prices.

6. Explore the distribution of the numeric variable `Price`.

a) In the first code cell under **Explore the distribution of the numeric variable `Price`**, write code to generate summary statistics for just the numeric variables.
b) Run the code cell and verify the mean, standard deviation, median, etc., for each variable.
c) In the next code cell, write code to generate a violin plot for the `Price` variable.
d) Run the code cell and verify that the distribution of `Price` is highly concentrated at the lower end, but does have a few high outliers that may be skewing the distribution.

7. Visualize correlations between numeric variables.

a) In the first code cell under **Visualize correlations between numeric variables**, write code to generate a correlation matrix in `DataFrame` format.
b) Run the code cell and verify that `Quantity` and `TotalAmount` have strong positive correlation, whereas the other pairings have a low amount of correlation.
c) In the next code cell, write code to visualize the correlation matrix on a heatmap.
d) Run the code cell and verify that you can better see the strength of each correlation through color intensity.

8. Transform skewed variables.

a) In the first code cell under **Transform skewed variables**, write code to generate histograms for all of the numeric variables.
b) Run the code cell and verify that all three numeric variables have a positive (right) skew.
c) In the next code cell, write code to apply a log transformation to the `Price` variable, then plot a histogram of that transformation.
d) Run the code cell and verify that the log transformation has somewhat normalized the `Price` distribution, reducing its skew.
e) Repeat this process for the `Quantity` and `TotalAmount` variables, noting that the former may still exhibit some positive skewness.

9. Analyze time series data.

a) In the first code cell under **Analyze time series data**, write code to obtain and print the number of invoices by each month.

b) Run the code cell and verify that you have a count of each month's invoices.
c) In the next code cell, write code to plot a bar chart of the number of invoices per month.
d) Run the code cell and verify which months have the most and least invoices.

10. Identify and handle missing data.

a) In the first code cell under **Identify and handle missing data**, write code to identify any missing values for each variable.
b) Run the code cell and verify that there are missing values for `Price`, `CustomerID`, and `TotalAmount`.
c) In the next code cell, write code to remove all rows where `CustomerID` is unknown, and print the shape of the data before and after.
d) Run the code cell and verify that several thousand rows were removed.
e) In the next code cell, write code to fill in `N/A` values for `Price` and `TotalAmount` with a value of `0`, then confirm there are no longer any missing values in the dataset.
f) Run the code cell and verify that all missing values in the dataset have been handled.

11. One-hot encode the `Description` variable.

a) In the first code cell under **One-hot encode the `Description` variable**, write code to generate dummy variables for the `Description` values as columns in a new `DataFrame`.
b) Run the code cell and verify that the `DataFrame` shows multiple columns for each description, where each row has a `1` in a single column and a `0` in the rest.
c) In the next code cell, write code to concanetate the dummy variables `DataFrame` with the main dataset `DataFrame`, then drop the original `Description` variable.
d) Run the code cell.
e) In the next code cell, get the first five rows of the dataset.
f) Run the code cell and verify that the dummy variables have been added as columns to the main dataset.

12. Identify and remove columns with low variance.

a) In the first code cell under **Identify and remove columns with low variance**, write code to obtain the standard deviation of every variable (including the dummy variables).
b) Run the code cell and verify the amount of standard deviation in each variable.
c) In the next code cell, write code to define a threshold of `0.26`, then identify any columns that have a standard deviation lower than this threshold.
d) Run the code cell and verify that at least one variable has a standard deviation lower than the threshold.
e) In the next code cell, write code to drop any columns that have low standard deviation, then get the first five rows of the dataset.
f) Run the code cell and verify that the column(s) with low standard deviation have been removed.

13. Generate box plots for each numeric variable.

a) In the code cell under **Generate box plots for each numeric variable**, write code to draw box plots for each numeric variable. Consider placing each box plot as a subplot of an overall figure so that you can easily compare them.
b) Run the code cell and verify that you can see multiple outliers in all three variables, particularly outliers on the higher end.

14. Identify and remove outliers.

a) In the first code cell under **Identify and remove outliers**, observe the code that's been provided for you.

   This function returns the lower and upper bounds of a numeric input variable.
b) Run the code cell.
c) In the next code cell, print the shape of the dataset before removing outliers.
d) Run the code cell and verify the number of rows in the dataset.

e) In the next code cell, call the `calc_outliers()` function iteratively for each numeric variable, dropping any values for that variable that are higher than the upper bounds or values that are lower than the lower bounds. Also print the shape of the dataset after outliers have been removed from each variable.

f) Run the code cell and verify that each variable shows the lower bound and upper bound of the outliers for that variable, and that each time a variable's outliers are removed, the dataset's rows decrease in number.

15. Save the final dataset as a pickle file.

a) In the cell under **Save the final dataset as a pickle file**, write code to save the final `DataFrame` as a pickle file.

b) Run the code cell.

16. Save and download the notebook.

a) From the menu, select **File→Save and Checkpoint**.

b) Select **File→Download as→Notebook (.ipynb)**.

c) Save the file to your local drive if it wasn't automatically.

d) Find the saved file and rename it using the following convention: **FirstnameLastname-Analysis-Project.ipynb**.

For example: **JohnSmith-Analysis-Project.ipynb**.

Note: Make sure not to lose this file. This is the project you'll be uploading and sharing to your peers for grading.

17. Shut down the notebook and close the lab.

a) Back in Jupyter Notebook, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on to the peer review.

Course 4: Train Machine Learning Models

Course Introduction

The following labs are for Course 4: Train Machine Learning Models.

Modules

The labs in this course pertain to the following modules:

- Module 2: Develop Classification Models
- Module 3: Develop Regression Models
- Module 4: Develop Clustering Models
- Apply What You've Learned

MODULE 2

Develop Classification Models

The following labs are for Module 2: Develop Classification Models.

LAB 4-1
Training a Logistic Regression Model

Data Files

~/Classification/Developing Classification Models.ipynb

~/Classification/data/users_data_final.pickle

Scenario

The time has come to start building machine learning models that will help GCNB make intelligent decisions about its data and its business operations. One of the potential target variables you and your team identified earlier was the `term_deposit` variable. The more effective a GCNB marketing campaign, the more term deposits the bank can get from its customers, and the more revenue it can generate. Your task is therefore to determine what customers are most likely to sign up for a term deposit, so that the marketing campaigns can prioritize those customers and seek out new customers who have the features that are most important for predicting a term deposit. This is a good task for classification.

Rather than just build one classification model and call it a day, you'll end up building multiple models from various algorithms to determine which one best meets your performance needs. You'll start by training a logistic regression model. Before you train the model, you'll need to do a bit more prep work on the data to make it more conducive to machine learning.

Note: By default, these lab steps assume you will be typing the code shown in screenshots. Many learners find it easier to understand what code does when they are able to type it themselves. However, if you prefer not to type all of code, the **Solutions** folders contain the finished code for each notebook.

---

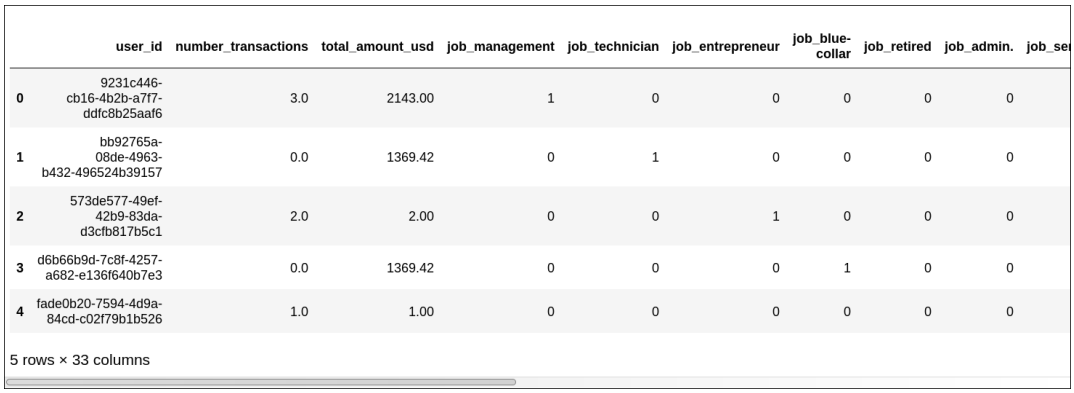
1. Open the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Classification**.
      The **Classification** directory contains a subdirectory named **data** and a notebook file named **Developing Classification Models.ipynb**.
   c) Select **Developing Classification Models.ipynb** to open it.

2. Import the relevant software libraries.
   a) View the cell titled **Import software libraries**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.
   c) Verify that the version of Python is displayed, as are the versions of the other libraries that were imported.

3. Load and preview the data.
   a) Scroll down and view the cell titled **Load and preview the data**, then select the code cell below it.
   b) In the code cell, type the following:

```
1 users_data = pd.read_pickle('data/users_data_final.pickle')
2
3 users_data.head(n = 5)
```

c) Run the code cell.
d) Examine the output.



	user_id	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_se
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	3.0	2143.00	1	0	0	0	0	0	
1	bb92765a-08de-4963-b432-496524b39157	0.0	1369.42	0	1	0	0	0	0	
2	573de577-49ef-42b9-83da-d3cfb817b5c1	2.0	2.00	0	0	1	0	0	0	
3	d6b66b9d-7c8f-4257-a682-e136f640b7e3	0.0	1369.42	0	0	0	1	0	0	
4	fade0b20-7594-4d9a-84cd-c02f79b1b526	1.0	1.00	0	0	0	0	0	0	

5 rows × 33 columns

This is the dataset you cleaned and performed preprocessing on earlier. You'll examine the data to make sure it's in a good state to send to a machine learning algorithm.

4. Check the shape of the data.

a) Scroll down and view the cell titled **Check the shape of the data**, then select the code cell below it.
b) In the code cell, type the following:

```
1 users_data.shape
```

c) Run the code cell.
d) Examine the output.

```
(45179, 33)
```

There are 45,179 records and 33 columns.

5. Check the data types.

a) Scroll down and view the cell titled **Check the data types**, then select the code cell below it.
b) In the code cell, type the following:

```
1 users_data.info()
```

c) Run the code cell.

d) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 45179 entries, 0 to 45215
Data columns (total 33 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   user_id              45179 non-null  object
 1   number_transactions  45179 non-null  float64
 2   total_amount_usd     45179 non-null  float64
 3   job_management       45179 non-null  int64
 4   job_technician       45179 non-null  int64
 5   job_entrepreneur     45179 non-null  int64
 6   job_blue-collar      45179 non-null  int64
 7   job_retired          45179 non-null  int64
 8   job_admin.           45179 non-null  int64
 9   job_services         45179 non-null  int64
 10  job_self-employed    45179 non-null  int64
 11  job_unemployed       45179 non-null  int64
 12  job_housemaid        45179 non-null  int64
 13  job_student          45179 non-null  int64
 14  education_tertiary   45179 non-null  int64
 15  education_secondary  45179 non-null  int64
```

All of the columns are either floats, integers, or Booleans, except for `user_id`. String objects won't work well with machine learning algorithms, and `user_id` isn't particularly useful, so you'll soon drop it from the dataset.

6. Explore the distribution of the target variable.

a) Scroll down and view the cell titled **Explore the distribution of the target variable**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data.term_deposit.value_counts(normalize = True)
```

The target variable is the variable you're interested in learning more about. In this case, you've identified `term_deposit` as a good target variable candidate.

c) Run the code cell.

d) Examine the output.

```
False    0.883021
True     0.116979
Name: term_deposit, dtype: float64
```

As you can see, most people did not sign up for a term deposit (about 88%), which means that the dataset is imbalanced. You'll soon correct this imbalance through oversampling.

7. Split the data into target and features.

a) Scroll down and view the cell titled **Split the data into target and features**, then select the code cell below it.

b) In the code cell, type the following:

```
target_data = users_data.term_deposit
features = users_data.drop(['user_id', 'term_deposit'], axis = 1)
```

You're separating the target variable into its own `DataFrame`, as well as dropping the unnecessary `user_id` column from the main `DataFrame`. This will ensure your models consider `term_deposit` the labeled variable.

c) Run the code cell.

8. Split the data into train and test sets.

a) Scroll down and view the cell titled **Split the data into train and test sets**, then select the code cell below it.

b) In the code cell, type the following:

```
X_train, X_test, y_train, y_test = train_test_split(features,
                                                    target_data,
                                                    test_size = 0.3)
```

This code performs the holdout method by dividing each dataset into training and testing sets.

- `X_train` includes all of the features except the label and will be used to train the model. 70% of the rows are in this set, according to the `test_size` parameter.
- `y_train` includes just the label/target variable and will be used to train the model. Once again, 70% of the rows are in this set.
- `X_test` includes all of the features except the label and will be used to test the model. 30% of the rows are in this set.
- `y_test` includes just the label/target variable and will be used to test the model. 30% of the rows are in this set.

Note: `X_train/test` and `y_train/test` is a conventional naming scheme for dataset splits in machine learning. You could use more descriptive variables instead.

Note: You'll perform more advanced forms of cross-validation later on.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
print('Training data features: ', X_train.shape)
print('Training data target:   ', y_train.shape)
```

e) Run the code cell.

f) Examine the output.

```
Training data features:  (31625, 31)
Training data target:    (31625,)
```

There are 31,625 rows in the training sets, which is 70% of the total `user_data` set.

9. Apply oversampling to the data.

a) Scroll down and view the cell titled **Apply oversampling to the data**, then select the code cell below it.

b) In the code cell, type the following:

```
print('Before oversampling: ', Counter(y_train))
```

c) Run the code cell.

d) Examine the output.

```
Before oversampling:  Counter({False: 27963, True: 3662})
```

At the moment, there are 27,963 `False` values for `term_deposit`, and 3,662 `True` values.

e) Select the next code cell, then type the following:

```
# Define oversampling strategy.

SMOTE = SMOTE()

# Fit and apply the transform.

X_train_SMOTE, y_train_SMOTE = SMOTE.fit_resample(X_train, y_train)
X_train_SMOTE = pd.DataFrame(X_train_SMOTE,
                             columns = X_train.columns)

print('After oversampling: ', Counter(y_train_SMOTE))
```

This code uses the oversampling technique known as SMOTE, which creates synthetic data based on the existing feature space.

f) Run the code cell.

g) Examine the output.

```
After oversampling:  Counter({False: 27963, True: 27963})
```

After performing SMOTE, more samples have been added to the training sets, and both class values for `term_deposit` are now represented equally. In other words, the dataset is now balanced with regard to the target variable.

10. Check the distribution of the test data.

a) Scroll down and view the cell titled **Check the distribution of the test data**, then select the code cell below it.

b) In the code cell, type the following:

```
# Test data should not be oversampled.

print('Test data features: ', X_test.shape)
print('Test data target:   ', y_test.shape)
```

c) Run the code cell.

d) Examine the output.

```
Test data features:  (13554, 31)
Test data target:    (13554,)
```

The test data still has 13,554 rows, which was 30% of the original dataset.

e) Select the next code cell, then type the following:

```
Counter(y_test)
```

f) Run the code cell.

g) Examine the output.

```
Counter({False: 11931, True: 1623})
```

Unlike the training data, the test data is not oversampled, as the model won't be learning from this data.

11. Normalize the data.

a) Scroll down and view the cell titled **Normalize the data**, then select the code cell below it.
b) In the code cell, type the following:

```
norm = MinMaxScaler().fit(X_train_SMOTE)
```

Certain machine learning algorithms, including logistic regression, perform best when the input data is scaled. Here, you're using a function that normalizes the data.

c) Run the code cell.
d) Select the next code cell, then type the following:

```
X_train_norm = norm.transform(X_train_SMOTE)

print('Minimum: ', np.min(X_train_norm))
print('Maximum: ', np.max(X_train_norm))
```

e) Run the code cell.
f) Examine the output.

```
Minimum:  0.0
Maximum:  1.0
```

After the transformation, the minimum value is 0 and the maximum is 1, indicating that the data was normalized. Note that the normalized version of the dataset is in its own variable so that you can retain the non-normalized version for use with algorithms that don't benefit from featuring scaling.

12. Train a logistic regression model.

a) Scroll down and view the cell titled **Train a logistic regression model**, then select the code cell below it.
b) In the code cell, type the following:

```
logreg = LogisticRegression()
logreg.fit(X_train_norm, y_train_SMOTE)
```

This code creates a logistic regression object with the default hyperparameters. Then, the `logreg` model is "fit" with the normalized training data (both the features and the label). This initiates the training process.

c) Run the code cell.

Note: In most cases, the output of creating the model object and calling `fit()` will just be the model's class name.

13. Make predictions using the logistic regression model.

a) Scroll down and view the cell titled **Make predictions using the logistic regression model**, then select the code cell below it.
b) In the code cell, type the following:

```
logreg_y_pred = logreg.predict(X_test)
print(Counter(logreg_y_pred))
```

When `predict()` is called on the model object, it takes the test dataset as the input the model is trying to predict. You can compare the predictions to the actual label values.

c) Run the code cell.

d) Examine the output.

```
Counter({True: 13486, False: 68})
```

The model's predictions include 13,486 `True` values, and 68 `False` values. Remember that this is a prediction of just `X_test`—data that the model is seeing for the first time.

e) Select the next code cell, then type the following:

```
results = pd.concat([y_test.iloc[:5], X_test.iloc[:5]], axis = 1)
results.insert(1, 'term_deposit_pred', logreg_y_pred[:5])
results
```

This code will show the first five records of the test set and compare the actual `term_deposit` labels to the model's predictions.

f) Run the code cell.

g) Examine the output.

	term_deposit	term_deposit_pred	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	jc
43308	False	True	0.0	1369.42	1	0	0	0	0	
32770	False	True	2.0	246.00	0	0	0	1	0	
17440	False	True	0.0	1369.42	0	0	0	0	0	
36164	False	True	0.0	1369.42	0	0	0	1	0	
29218	False	True	0.0	1369.42	0	0	0	0	0	

5 rows × 33 columns

The first column, `term_deposit`, is the ground truth for each record, whereas the `term_deposit_pred` column shows the model's predictions. This logistic regression model incorrectly predicted `True` values for all of the first five records.

14. Obtain the logistic regression model's score.

a) Scroll down and view the cell titled **Obtain the logistic regression model's score**, then select the code cell below it.

b) In the code cell, type the following:

```
accuracy_score(y_test, logreg_y_pred)
```

You can use metrics like accuracy to determine how well the model did in its predictions. Most classifiers in scikit-learn use accuracy as the default scoring metric, which is the fraction of predictions that the model got right. However, this is not the only metric, nor is it necessarily the best for any given scenario. You'll learn more about metrics later—for now, verifying the model's accuracy scores will suffice.

c) Run the code cell.

d) Examine the output.

```
0.1247602183857164
```

The logistic regression model has an accuracy of ~12%. That's very low, and not something you would normally accept. You'll see if you can improve this score later.

15. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 4-2
Training a *k*-NN Model

Data File

~/Classification/Developing Classification Models.ipynb

Scenario

The initial logistic regression model didn't produce very good results. But, there are plenty more classification algorithms to try. So, you'll build another model, this time using k-nearest neighbor (k-NN), a distanced-based algorithm. You'll make predictions using this model and evaluate its accuracy score as well to see how it compares.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Classification/Developing Classification Models.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Train a *k*-nearest neighbor (*k*-NN) model** heading, then select **Cell→Run All Above**.

2. Train a k-nearest neighbor (k-NN) model.
   a) Scroll down and view the cell titled **Train a *k*-nearest neighbor (*k*-NN) model**, then select the code cell below it.
   b) In the code cell, type the following:

```
knn = KNeighborsClassifier()
knn.fit(X_train_norm, y_train_SMOTE)
```

   As with logistic regression, you're creating a model object and fitting the data to it. Since k-NN calculates classifications using distance (i.e., the numeric ranges of features), it's best to use the normalized version of the data.
   c) Run the code cell.

3. Make predictions using the k-NN model.
   a) Scroll down and view the cell titled **Make predictions using the *k*-NN model**, then select the code cell below it.
   b) In the code cell, type the following:

```
knn_y_pred = knn.predict(X_test)
print(Counter(knn_y_pred))
```

   This code will make predictions on the test data, similar to the code for logistic regression.
   c) Run the code cell.

d) Examine the output.

```
Counter({True: 10197, False: 3357})
```

The k-NN model predicted that 10,197 records have `True` for `term_deposit`, whereas 3,357 records show `False`.

e) Select the next code cell, then type the following:

```
results['term_deposit_pred'] = knn_y_pred[:5]
results
```

f) Run the code cell.

g) Examine the output.

	term_deposit	term_deposit_pred	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	jo
43308	False	True	0.0	1369.42	1	0	0	0	0	
32770	False	True	2.0	246.00	0	0	0	1	0	
17440	False	True	0.0	1369.42	0	0	0	0	0	
36164	False	True	0.0	1369.42	0	0	0	1	0	
29218	False	True	0.0	1369.42	0	0	0	0	0	

5 rows × 33 columns

The k-NN model also incorrectly predicted the first five rows.

4. Obtain the k-NN model's score.

a) Scroll down and view the cell titled **Obtain the k-NN model's score**, then select the code cell below it.

b) In the code cell, type the following:

```
accuracy_score(y_test, knn_y_pred)
```

c) Run the code cell.

d) Examine the output.

```
0.2968865279622252
```

The k-NN model's accuracy is ~30%; better than the logistic regression model, but still not great.

e) Select the next code cell, then type the following:

```
print('Number of mislabeled points out of a total %d points: %d'
      % (X_test.shape[0], (y_test != knn_y_pred).sum()))
```

This code will give you another perspective into the model's performance by identifying how many data examples it predicted incorrectly.

f) Run the code cell.

g) Examine the output.

```
Number of mislabeled points out of a total 13554 points: 9530
```

9,530 out of 13,554 examples were predicted incorrectly.

5. Save and close the lab.
   a) From the menu, select **File→Save and Checkpoint**.
   b) Close the lab browser tab and continue on with the course.

LAB 4-3
Training an SVM Classification Model

Data File

~/Classification/Developing Classification Models.ipynb

Scenario

The *k*-NN model did better than the logistic regression model, but there's still a chance to improve your scores and generate a better model. So, you'll try out your data on a support-vector machine (SVM) algorithm. SVMs excel at modeling data with outliers. Although you dealt with outliers earlier in the ETL process, an SVM model might still perform well. So, you'll train one and find out.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Classification/Developing Classification Models.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Train a support-vector machine (SVM) model** heading, then select **Cell→Run All Above**.

2. Train a support-vector machine (SVM) model.
   a) Scroll down and view the cell titled **Train a support-vector machine (SVM) model**, then select the code cell below it.
   b) In the code cell, type the following:

```
svm = SVC()
svm.fit(X_train_norm, y_train_SMOTE)
```

   `SVC()` is a scikit-learn implementation of an SVM classifier. Because SVMs are also based on distance, you'll train the model using the normalized dataset.
   c) Run the code cell.

Note: It will take a minute or two for this model to finish training.

3. Make predictions using the SVM model.
   a) Scroll down and view the cell titled **Make predictions using the SVM model**, then select the code cell below it.
   b) In the code cell, type the following:

```
svm_y_pred = svm.predict(X_test)
print(Counter(svm_y_pred))
```

   c) Run the code cell.

d) Examine the output.

```
Counter({False: 13554})
```

It seems that the SVM model predicted *all* 13,554 examples would be `False` for `term_deposit`. Obviously, this model has some significant limitations.

e) Select the next code cell, then type the following:

```
results['term_deposit_pred'] = svm_y_pred[:5]
results
```

f) Run the code cell.

g) Examine the output.

	term_deposit	term_deposit_pred	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	jc
43308	False	False	0.0	1369.42	1	0	0	0	0	
32770	False	False	2.0	246.00	0	0	0	1	0	
17440	False	False	0.0	1369.42	0	0	0	0	0	
36164	False	False	0.0	1369.42	0	0	0	1	0	
29218	False	False	0.0	1369.42	0	0	0	0	0	

5 rows × 33 columns

The SVM model correctly predicted that the first five records would be `False`.

4. Obtain the SVM model's score.

a) Scroll down and view the cell titled **Obtain the SVM model's score**, then select the code cell below it.

b) In the code cell, type the following:

```
accuracy_score(y_test, svm_y_pred)
```

c) Run the code cell.

d) Examine the output.

```
0.880256750774679
```

The SVM model's accuracy is very high: ~88%. This is a good example of why some metrics, particularly accuracy, can be misleading. The model predicted that every example would be `False`, and since the test set is so imbalanced in favor of `False`, the model ended up making a lot of correct predictions. However, a model that only predicts `False` and nothing else is not particularly useful, since any human could have done that without effort. When you learn about metrics, you'll see some other ways that will make the model's limitations more apparent. For now, consider that not every model you train will end up performing better than the last. This is the unavoidable reality of experimentation.

5. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 4-4

Training a Naïve Bayes Model

Data File

~/Classification/Developing Classification Models.ipynb

Scenario

The next model you'll train is a naïve Bayes model, which calculates classification probabilities based on Bayes' theorem. These types of models assume that the features do not interact with each other, which they almost certainly do in a dataset like this one. Still, it's worth trying out, so you'll build one and see how it scores.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Classification/Developing Classification Models.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Train a naïve Bayes model** heading, then select **Cell→Run All Above**.

2. Train a naïve Bayes model.
   a) Scroll down and view the cell titled **Train a naïve Bayes model**, then select the code cell below it.
   b) In the code cell, type the following:

```
gnb = GaussianNB()
gnb.fit(X_train_SMOTE, y_train_SMOTE)
```

   You'll be training the model on the non-normalized dataset, as naïve Bayes is not as sensitive to differing numerical ranges as logistic regression and distance-based algorithms.
   c) Run the code cell.

3. Make predictions using the naïve Bayes model.
   a) Scroll down and view the cell titled **Make predictions using the naïve Bayes model**, then select the code cell below it.
   b) In the code cell, type the following:

```
gnb_y_pred = gnb.predict(X_test)
print(Counter(gnb_y_pred))
```

   c) Run the code cell.
   d) Examine the output.

```
Counter({False: 9685, True: 3869})
```

   The naïve Bayes model predicted 9,685 `False` values and 3,869 `True` values. Unlike the SVM model, this model is able to discriminate between both values.

e) Select the next code cell, then type the following:

```
results['term_deposit_pred'] = gnb_y_pred[:5]
results
```

f) Run the code cell.

g) Examine the output.

	term_deposit	term_deposit_pred	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	jo
43308	False	False	0.0	1369.42	1	0	0	0	0	
32770	False	True	2.0	246.00	0	0	0	1	0	
17440	False	False	0.0	1369.42	0	0	0	0	0	
36164	False	False	0.0	1369.42	0	0	0	1	0	
29218	False	False	0.0	1369.42	0	0	0	0	0	

5 rows × 33 columns

Out of the first five records, all but the second were predicted correctly.

4. Obtain the naïve Bayes model's score.

a) Scroll down and view the cell titled **Obtain the naïve Bayes model's score**, then select the code cell below it.

b) In the code cell, type the following:

```
accuracy_score(y_test, gnb_y_pred)
```

c) Run the code cell.

d) Examine the output.

```
0.7019330087059171
```

The naïve Bayes model has an accuracy score of ~70%. This is considerably better than the logistic regression and k-NN models.

5. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 4-5

Training Classification Decision Trees and Ensemble Models

Data File

~/Classification/Developing Classification Models.ipynb

Scenario

As you've seen, individually trained models can produce worthwhile results, but you may also get better results with ensemble methods. Ensemble algorithms use an aggregation of multiple decision trees to classify data. So, you'll leverage these algorithms by first training a single decision tree, then training multiple trees within random forests and gradient boosting models. Hopefully, these methods will improve upon the individual models you've built thus far.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Classification/Developing Classification Models.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Train a decision tree model** heading, then select **Cell→Run All Above**.

2. Train a decision tree model.
   a) Scroll down and view the cell titled **Train a decision tree model**, then select the code cell below it.
   b) In the code cell, type the following:

```
clf_tree = DecisionTreeClassifier()
clf_tree.fit(X_train_SMOTE, y_train_SMOTE)
```

   Tree-based algorithms are insensitive to feature scaling, so you'll train these models with the non-normalized data.
   c) Run the code cell.

3. Make predictions using the decision tree model.
   a) Scroll down and view the cell titled **Make predictions using the decision tree model**, then select the code cell below it.
   b) In the code cell, type the following:

```
clf_tree_y_pred = clf_tree.predict(X_test)
print(Counter(clf_tree_y_pred))
```

   c) Run the code cell.

d) Examine the output.

```
Counter({False: 11596, True: 1958})
```

The decision tree predicted 11,596 `False` values and 1,958 `True` values.

e) Select the next code cell, then type the following:

```
results['term_deposit_pred'] = clf_tree_y_pred[:5]
results
```

f) Run the code cell.

g) Examine the output.

	term_deposit	term_deposit_pred	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	jc
43308	False	True	0.0	1369.42	1	0	0	0	0	
32770	False	False	2.0	246.00	0	0	0	1	0	
17440	False	True	0.0	1369.42	0	0	0	0	0	
36164	False	False	0.0	1369.42	0	0	0	1	0	
29218	False	False	0.0	1369.42	0	0	0	0	0	

5 rows × 33 columns

The second, fourth, and fifth records were correctly predicted, but the first and third were not.

4. Obtain the decision tree model's score.

a) Scroll down and view the cell titled **Obtain the decision tree model's score**, then select the code cell below it.

b) In the code cell, type the following:

```
accuracy_score(y_test, clf_tree_y_pred)
```

c) Run the code cell.

d) Examine the output.

```
0.8314150804190645
```

The single decision tree's accuracy is ~83%. This is the highest accuracy score yet, aside from the SVM model.

5. Visualize the decision tree.

a) Scroll down and view the cell titled **Visualize the decision tree**, then select the code cell below it.

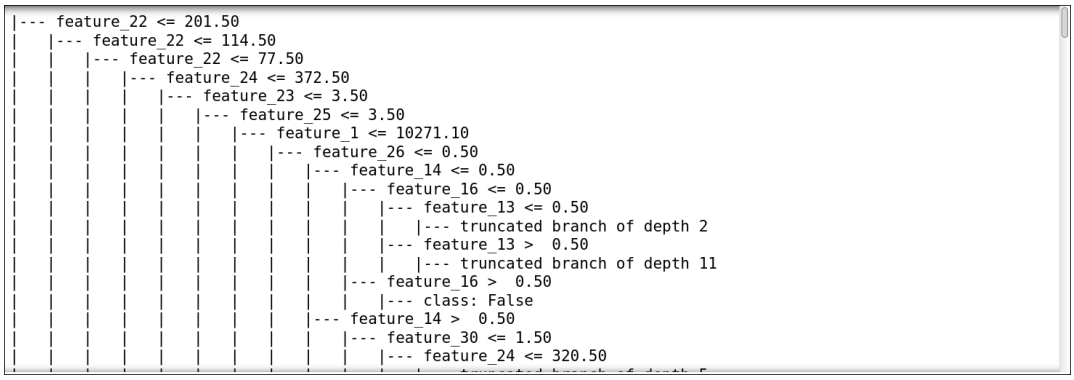
b) In the code cell, type the following:

```
text_representation = tree.export_text(clf_tree)
print(text_representation)
```

Visualizing a decision tree can make it easier to identify specific decisions that lead to classifications.

c) Run the code cell.

d) Examine the output.



```
|--- feature_22 <= 201.50
|   |--- feature_22 <= 114.50
|   |   |--- feature_22 <= 77.50
|   |   |   |--- feature_24 <= 372.50
|   |   |   |   |--- feature_23 <= 3.50
|   |   |   |   |   |--- feature_25 <= 3.50
|   |   |   |   |   |   |--- feature_1 <= 10271.10
|   |   |   |   |   |   |   |--- feature_26 <= 0.50
|   |   |   |   |   |   |   |   |--- feature_14 <= 0.50
|   |   |   |   |   |   |   |   |   |--- feature_16 <= 0.50
|   |   |   |   |   |   |   |   |   |   |--- feature_13 <= 0.50
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 2
|   |   |   |   |   |   |   |   |   |   |--- feature_13 >  0.50
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 11
|   |   |   |   |   |   |   |   |   |--- feature_16 >  0.50
|   |   |   |   |   |   |   |   |   |   |--- class: False
|   |   |   |   |   |   |   |   |--- feature_14 >  0.50
|   |   |   |   |   |   |   |   |   |--- feature_30 <= 1.50
|   |   |   |   |   |   |   |   |   |   |--- feature_24 <= 320.50
```

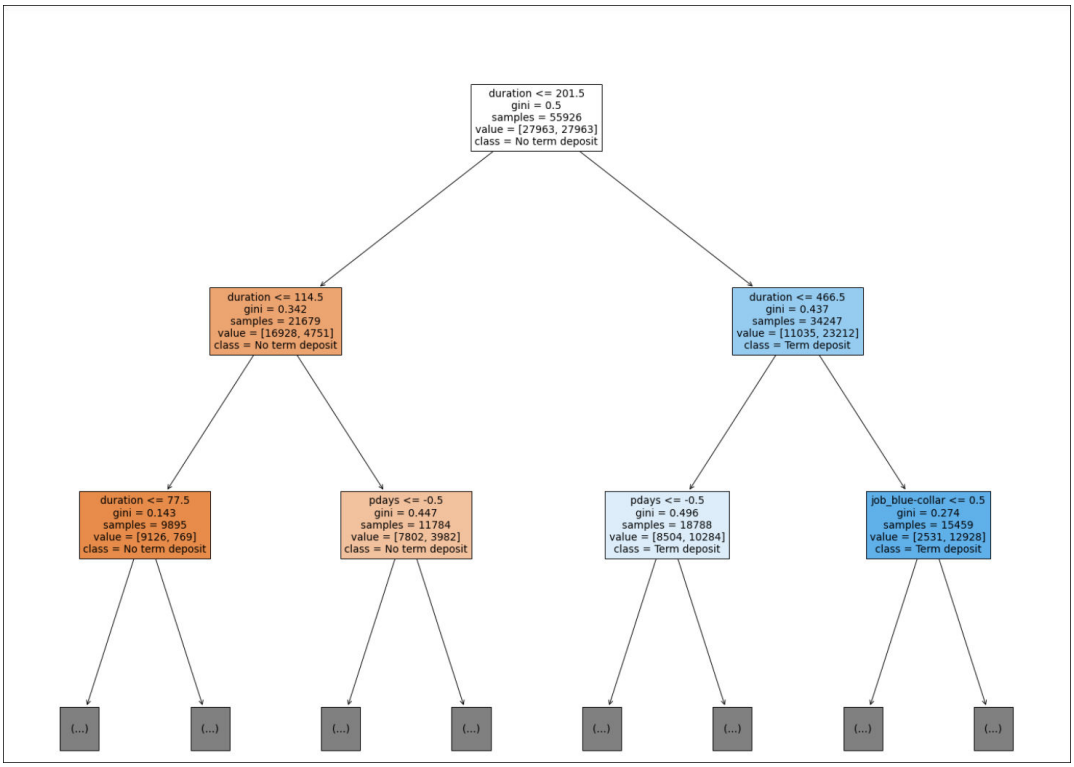
This is a text-based representation of the entire tree. You could trace each branch and leaf, but given the size of the tree, this can take a while. Instead, you'll create a more visual representation of a truncated version of the tree.

e) Select the next code cell, then type the following:

```
fig = plt.figure(figsize = (25, 20))
_ = tree.plot_tree(clf_tree,
                   feature_names = list(X_train.columns),
                   class_names = ['No term deposit', 'Term deposit'],
                   max_depth = 2,
                   filled = True)
```

f) Run the code cell.

g) Examine the output.



This visualization truncates the tree to a depth of two. Each node includes splitting/decision information at that node. For the root decision node at the top:

- The tree starts by evaluating if the user's contact session duration was less than or equal to 201.5 seconds.
- The Gini index at this point is 0.5. An index of 0 represents purity, so the number should ideally be decreasing as you descend the branches.
- The number of samples at this node is 55,926 (the entire oversampled training set).
- The `value` shows that 27,963 examples are true for this decision, and 27,963 are false. This aligns with the oversampling you performed earlier.
- The class prediction at this point is 0, meaning the user didn't sign up for a term deposit. This prediction will get much more useful as you descend the tree.

For the first "True" branch on the left:

- The decision node here is also evaluating the contact duration.
- The Gini index is 0.342, which, as expected, is becoming more pure.
- The number of samples is 21,679.
- 16,928 samples were not less than or equal to 114.5, and 4,751 samples were.
- The class prediction here is still 0 (no term deposit).

For the first "False" branch on the right:

- The decision node here is also evaluating contact duration.
- The Gini index is 0.437.
- The number of samples is 34,247.
- 11,035 samples were not less than or equal to 446.5, and 23,212 samples were.
- The class prediction is 1 (signed up for a term deposit).

If you had printed the entire tree, you could continue to descend the tree until you get to the bottom, where there are multiple leaves, each with their own class predictions based on the branching logic of their parent nodes.

6. Train a random forest model.

a) Scroll down and view the cell titled **Train a random forest model**, then select the code cell below it.
b) In the code cell, type the following:

```
rf = RandomForestClassifier()
rf.fit(X_train_SMOTE, y_train_SMOTE)
```

This random forest algorithm will use bagging with multiple decision trees and make classification decisions based on the popular vote.
c) Run the code cell.

7. Make predictions using the random forest model.

a) Scroll down and view the cell titled **Make predictions using the random forest model**, then select the code cell below it.
b) In the code cell, type the following:

```
rf_y_pred = rf.predict(X_test)
print(Counter(rf_y_pred))
```

c) Run the code cell.
d) Examine the output.

```
Counter({False: 12488, True: 1066})
```

The random forest model predicted 12,488 `False` values and 1,066 `True` values.
e) Select the next code cell, then type the following:

```
results['term_deposit_pred'] = rf_y_pred[:5]
results
```

f) Run the code cell.
g) Examine the output.

	term_deposit	term_deposit_pred	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	jo
43308	False	False	0.0	1369.42	1	0	0	0	0	
32770	False	False	2.0	246.00	0	0	0	1	0	
17440	False	True	0.0	1369.42	0	0	0	0	0	
36164	False	False	0.0	1369.42	0	0	0	1	0	
29218	False	False	0.0	1369.42	0	0	0	0	0	

5 rows × 33 columns

Of the first five records, the random forest correctly predicted all except the third.

8. Obtain the random forest model's score.

a) Scroll down and view the cell titled **Obtain the random forest model's score**, then select the code cell below it.
b) In the code cell, type the following:

```
accuracy_score(y_test, rf_y_pred)
```

c) Run the code cell.

d) Examine the output.

```
0.8826176774383946
```

The random forest achieved an accuracy score of ~88%. This is better than the lone decision tree, and the top-scoring model so far.

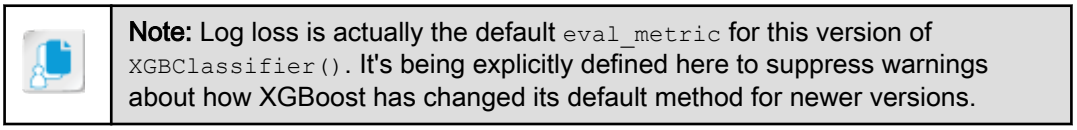
9. Train a gradient boosting model.

a) Scroll down and view the cell titled **Train a gradient boosting model**, then select the code cell below it.

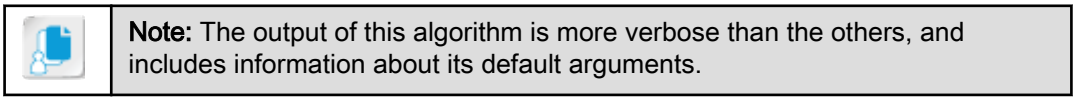
b) In the code cell, type the following:

```
xgb = XGBClassifier(eval_metric = 'logloss', n_jobs = 1)
xgb.fit(X_train_SMOTE, y_train_SMOTE)
```

Gradient boosting is another ensemble method that uses decision trees. It builds the trees sequentially so that past trees can improve the performance of later trees. This particular implementation of gradient boosting is XGBoost. The `eval_metric` argument refers to the method that the algorithm will use to minimize error. Log loss is the same method used by logistic regression to minimize error. The `n_jobs` argument is telling the function to use only a single CPU thread to avoid performance bottlenecks.



c) Run the code cell.



10. Make predictions using the gradient boosting model.

a) Scroll down and view the cell titled **Make predictions using the gradient boosting model**, then select the code cell below it.

b) In the code cell, type the following:

```
xgb_y_pred = xgb.predict(X_test)
print(Counter(xgb_y_pred))
```

c) Run the code cell.

d) Examine the output.

```
Counter({False: 12287, True: 1267})
```

The gradient boosting model predicted 12,287 `False` values and 1,267 `True` values.

e) Select the next code cell, then type the following:

```
results['term_deposit_pred'] = xgb_y_pred[:5]
results
```

f) Run the code cell.

g) Examine the output.

	term_deposit	term_deposit_pred	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	jc
43308	False	False	0.0	1369.42	1	0	0	0	0	
32770	False	False	2.0	246.00	0	0	0	1	0	
17440	False	True	0.0	1369.42	0	0	0	0	0	
36164	False	False	0.0	1369.42	0	0	0	1	0	
29218	False	False	0.0	1369.42	0	0	0	0	0	

5 rows × 33 columns

Of the first five records, the gradient boosting model correctly predicted all except the third.

11. Obtain the gradient boosting model's score.

a) Scroll down and view the cell titled **Obtain the gradient boosting model's score**, then select the code cell below it.

b) In the code cell, type the following:

```
1 accuracy_score(y_test, xgb_y_pred)
```

c) Run the code cell.

d) Examine the output.

```
0.8854950568097979
```

The gradient boosting model's accuracy score is ~89%. This is slightly higher than the random forest, and the highest one yet.

12. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 4-6
Tuning Classification Models

Data File

~/Classification/Developing Classification Models.ipynb

Scenario

You've built several different models to classify the `term_deposit` target variable. Each model is different in how it makes its predictions, as well as its success at doing so. However, effective machine learning employs an iterative tuning process, where such models can get better when they're configured properly. You'll go through this tuning process by determining optimal hyperparameters for two different models: the logistic regression model and the gradient boosting model. The former obviously performed very poorly at first, so it should only get better through tuning. The latter has the highest score thus far, but may be susceptible to overfitting (as tree-based models often are). So, you'll see if you can improve that model as well, while also reducing overfitting through cross-validation.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Classification/Developing Classification Models.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Define the parameter grid used to tune the logistic regression model** heading, then select **Cell→Run All Above**.

2. Define the parameter grid used to tune the logistic regression model.
   a) Scroll down and view the cell titled **Define the parameter grid used to tune the logistic regression model**, then select the code cell below it.
   b) In the code cell, type the following:

```
solvers = ['newton-cg', 'lbfgs', 'liblinear']
penalty = ['l2']
c_values = [10, 1.0, 0.1, 0.01]

param_grid = dict(solver = solvers, penalty = penalty, C = c_values)

print(param_grid)
```

   You'll use hyperparameter searching methods to find the optimal hyperparameters for the logistic regression model. In this code, you'll start by defining a grid of the different hyperparameters to try:
   - `solvers` defines the type of methods that the model will use to minimize errors and find the optimal model parameters.
   - `penalty` is the type of regularization to use, which is a method of minimizing overfitting.
   - `c_values` define different strength values for the regularization.

   c) Run the code cell.

d) Examine the output.

```
{'solver': ['newton-cg', 'lbfgs', 'liblinear'], 'penalty': ['l2'], 'C': [10, 1.0, 0.1, 0.01]}
```

The hyperparameter grid is printed.

3. Perform a randomized search for optimal hyperparameters.

a) Scroll down and view the cell titled **Perform a randomized search for optimal hyperparameters**, then select the code cell below it.

b) In the code cell, type the following:

```
model = LogisticRegression()
random_search = RandomizedSearchCV(estimator = model,
                                   param_distributions = param_grid)
random_search.fit(X_train_norm, y_train_SMOTE)
```

The first search you'll do is a randomized search, in which multiple models are trained on a random distribution of the specified hyperparameters for a fixed number of iterations. The parameter grid you defined in the previous block will be used, and the metric it will attempt to optimize is, by default, accuracy. This will not always determine the best possible hyperparameters, but it usually gets pretty close, and is much faster than an exhaustive search like grid search. The `RandomizedSearchCV()` function in scikit-learn also performs cross-validation by default.

c) Run the code cell.

Note: It will take a few minutes for the search to finish.

d) Select the next code cell, then type the following:

```
# Summarize the results of the randomized search.

print('Best accuracy score:', round(random_search.best_score_, 4))
print('Best parameters:    ', random_search.best_params_)
```

e) Run the code cell.

f) Examine the output.

```
Best accuracy score: 0.8949
Best parameters:     {'solver': 'newton-cg', 'penalty': 'l2', 'C': 1.0}
```

The highest accuracy achieved in the search was a model that produced an accuracy of ~89%. This is much higher than the original logistic regression model, and is comparable to the original gradient boosting model you trained. You can also see which hyperparameters were used in the optimal model:

- `'newton-cg'` as the `solver`.
- `'l2'` as the regularization `penalty`.
- `1.0` as the `C` regularization strength.

Note: `'l2'` uses the lowercase letter "L", not the number 1.

4. Perform a grid search for optimal hyperparameters.

a) Scroll down and view the cell titled **Perform a grid search for optimal hyperparameters**, then select the code cell below it.

b) In the code cell, type the following:

```
model = LogisticRegression()
grid_search = GridSearchCV(estimator = model,
                           param_grid = param_grid)
logreg_fit = grid_search.fit(X_train_norm, y_train_SMOTE)
```

Unlike a randomized search, a grid search will always perform an exhaustive search of the entire parameter grid, so it will always return the optimal hyperparameters from that grid. You'll perform a grid search using the same parameter grid to see if it does any better than the randomized search. Also, `GridSearchCV()` performs cross-validation to minimize overfitting.

c) Run the code cell.

Note: It will take a few minutes for the search to finish.

d) Select the next code cell, then type the following:

```
# Summarize the results of the grid search.

print('Best accuracy score:', round(grid_search.best_score_, 4))
print('Best parameters:    ', grid_search.best_params_)
```

e) Run the code cell.

f) Examine the output.

```
Best accuracy score: 0.8949
Best parameters:     {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cg'}
```

As you can see, the results are the same. The randomized search is therefore sufficient for this particular use case.

5. Tune the gradient boosting model to reduce overfitting.

a) Scroll down and view the cell titled **Tune the gradient boosting model to reduce overfitting**, then select the code cell below it.

b) In the code cell, type the following:

```
model = XGBClassifier(eval_metric = 'logloss', n_jobs = 1)

param_grid = {
    'n_estimators': [10],
    'max_depth': [15, 25],
    'reg_alpha': [1.1, 1.2],
    'reg_lambda': [1.2, 1.3]
}
```

You'll perform another grid search, but this time on the XGBoost model. This can help you select the optimal hyperparameters to use, and the cross-validation will also hopefully reduce overfitting. Since this model uses a different algorithm than logistic regression, it has a different set of hyperparameters, so you need to create a new grid. In this case, the hyperparameters you'll be testing include:

- `n_estimators` is the number of individual decision trees in the model.
- `max_depth` specifies the maximum depth of each tree.
- `reg_alpha` specifies the weights of a certain type of regularization.
- `reg_lambda` specifies the weights of a different type of regularization.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
gs = GridSearchCV(estimator = model,
                  param_grid = param_grid,
                  n_jobs = 1,
                  scoring = 'accuracy',
                  verbose = 2)

fitted_model = gs.fit(X_train_SMOTE, y_train_SMOTE)
```

e) Run the code cell.

Note: It can take a couple minutes for the search to complete.

f) Examine the output.

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
[CV] max_depth=15, n_estimators=10, reg_alpha=1.1, reg_lambda=1.2 ....
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[CV]  max_depth=15, n_estimators=10, reg_alpha=1.1, reg_lambda=1.2, total=   1.8s
[CV] max_depth=15, n_estimators=10, reg_alpha=1.1, reg_lambda=1.2 ....
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    1.8s remaining:    0.0s
```

Since there are 8 different combinations of hyperparameters specified in the grid, and because the default k in the k-fold cross-validation that `GridSearchCV()` uses is 5, the model is fit a total of 8 × 5 = 40 times.

g) Select the next code cell, then type the following:

```
print('Best accuracy score:', round(gs.best_score_, 4))
print('Best parameters:    ', gs.best_params_)
```

h) Run the code cell.

i) Examine the output.

```
Best accuracy score: 0.9092
Best parameters:     {'max_depth': 25, 'n_estimators': 10, 'reg_alpha': 1.1, 'reg_lambda': 1.3}
```

The XGBoost accuracy score for the optimal model is now ~91%. The optimal hyperparameters are:

- `10` as the number of decision trees (`n_estimators`).
- `25` as the `max_depth` of each tree.
- `1.1` as the `reg_alpha` strength.
- `1.3` as the `reg_lambda` strength.

6. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 4-7
Evaluating Classification Models

Data File

~/Classification/Developing Classification Models.ipynb

Scenario

Now that you've trained and tuned your models, you wanted to evaluate their performance using more than just accuracy. There are many useful classification metrics that might be relevant to your data. In particular, because you balanced the dataset through oversampling, and because neither recall nor precision are more valuable in the case of predicting users signing up for a term deposit, you decide to focus on F_1 score as the primary metric. Still, it's important to consider other metrics as well, so you'll get a more comprehensive look at your models.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Classification/Developing Classification Models.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Compare evaluation metrics for each model** heading, then select **Cell→Run All Above**.

2. Compare evaluation metrics for each model.
   a) Scroll down and view the cell titled **Compare evaluation metrics for each model**, then select the code cell below it.

b) In the code cell, examine the following:

```
models = ['Logistic Regression', 'Naïve Bayes', 'SVM', 'k-NN',
          'Decision Tree', 'Random Forest', 'XGBoost', 'Dummy Classifier']

metrics = ['Accuracy', 'Precision', 'Recall', 'F1']

pred_list = ['logreg_y_pred', 'gnb_y_pred', 'svm_y_pred', 'knn_y_pred',
             'clf_tree_y_pred','rf_y_pred', 'xgb_y_pred', 'dummy_y_pred']

# Baseline algorithm.
dummy = DummyClassifier(strategy = 'stratified')
dummy.fit(X_train_SMOTE, y_train_SMOTE)
dummy_y_pred = dummy.predict(X_test)

scores = np.empty((0, 4))

for i in pred_list:
    scores = np.append(scores,
                       np.array([[accuracy_score(y_test, globals()[i]),
                                  precision_score(y_test, globals()[i]),
                                  recall_score(y_test, globals()[i]),
                                  f1_score(y_test, globals()[i])]]),
                       axis = 0)

scores = np.around(scores, 4)

scoring_df = pd.DataFrame(scores, index = models, columns = metrics)
scoring_df.sort_values(by = 'F1', ascending = False)
```

This code, which has been provided for you, constructs a `DataFrame` that will neatly compare the metric scores for each of the models you've trained so far. In addition, a `DummyClassifier()` model is being trained. This model makes classification decisions based on very simplistic rules, and is purely used as a way to ensure that your actual machine learning models are meeting a baseline level of performance. Also, to simplify things, the tuned models have been left out for now.

c) Run the code cell.

d) Examine the output.

	Accuracy	Precision	Recall	F1
XGBoost	0.8855	0.5280	0.4122	0.4630
Random Forest	0.8826	0.5150	0.3383	0.4083
Decision Tree	0.8314	0.3309	0.3993	0.3619
Naïve Bayes	0.7019	0.1876	0.4473	0.2644
Logistic Regression	0.1248	0.1203	1.0000	0.2148
k-NN	0.2969	0.1123	0.7055	0.1937
Dummy Classifier	0.4964	0.1171	0.4904	0.1891
SVM	0.8803	0.0000	0.0000	0.0000

The accuracy, precision, recall, and F_1 scores are listed for every model. The table is sorted by F_1 score. Some conclusions you can draw include:

- The XGBoost model has the highest F_1 score, highest accuracy, and highest precision. For your purposes, you can consider this your "best" model.
- The random forest and SVM models are close behind when it comes to accuracy, but keep in mind that the SVM model was not very helpful since it predicted all `False` values.
- The dummy classifier and k-NN have very low F_1 scores.
- Although logistic regression has very poor accuracy, it has perfect recall. This means that it had no false negatives; i.e., it never predicted that a user wouldn't sign up for a term deposit when they actually would. This is likely due to the fact that it predicted many `True` values and very few `False` values.
- Overall, it seems like the tree-based algorithms, particularly the ensemble methods, performed the best with your data.

3. Generate a confusion matrix.

a) Scroll down and view the cell titled **Generate a confusion matrix**, then select the code cell below it.

b) In the code cell, type the following:

```
1 confusion_matrix(y_test, xgb_y_pred)
```

You'll focus on your best model, the XGBoost model, for the rest of your evaluation efforts. This code will show a confusion matrix of the classification truth values.

c) Run the code cell.

d) Examine the output.

```
array([[11333,   598],
       [  954,   669]])
```

This confusion matrix shows the raw values for each truth assessment. It'll be easier to read when you use a visualization function to plot the matrix, however.

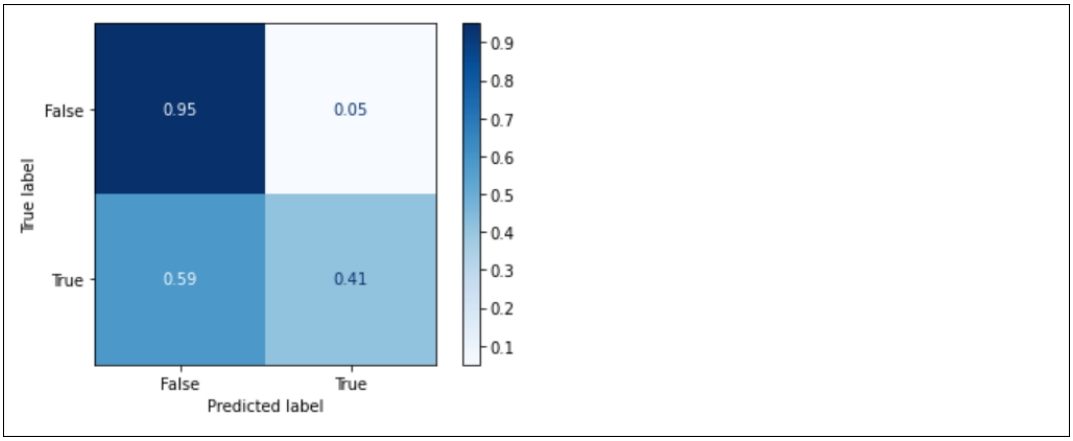
e) Select the next code cell, then type the following:

```
plot_confusion_matrix(xgb,
                      X_test,
                      y_test,
                      cmap = plt.cm.Blues,
                      normalize = 'true')

plt.show();
```

The confusion matrix will be labeled and have a color map to indicate magnitude in each quadrant. Also, the values will be normalized so that they show a percentage of the total row rather than the raw values.

f) Run the code cell.

g) Examine the output.



The results can be interpreted like so:

- In the top-left quadrant, 95% of actual `False` values (i.e., user *did not* sign up for a term deposit) were correctly predicted to be `False` by the model. These are the true negatives.
- In the top-right quadrant, the remaining 5% of actual `False` values were incorrectly predicted to be `True` by the model. These are the false positives.
- In the bottom-left quadrant, 59% of actual `True` values (i.e., user *did* sign up for a term deposit) were incorrectly predicted to be `False` by the model. These are the false negatives.
- In the bottom-right quadrant, the remaining 41% of actual `True` values were correctly predicted to be `True` by the model. These are the true positives.

This supports the scores you saw earlier. The largest "mistake" is in the bottom-left quadrant, which shows the false negatives. As false negatives increase, the recall score decreases.

4. Plot a ROC curve.

a) Scroll down and view the cell titled **Plot a ROC curve**, then select the code cell below it.

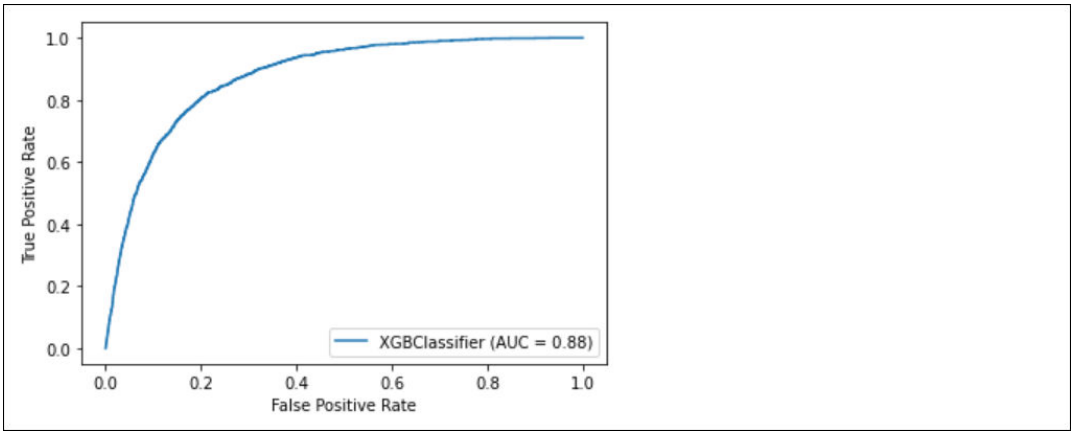
b) In the code cell, type the following:

```
plot_roc_curve(xgb, X_test, y_test)
plt.show()
```

This will plot a ROC curve and show its AUC.

c) Run the code cell.

d) Examine the output.



The ROC curve, since it sweeps up and to the left, has a shape that suggests it is much better than a random guess at distinguishing positive and negative classes. Its high AUC of 0.88 also supports that.

5. Plot a precision–recall curve.

a) Scroll down and view the cell titled **Plot a precision–recall curve**, then select the code cell below it.

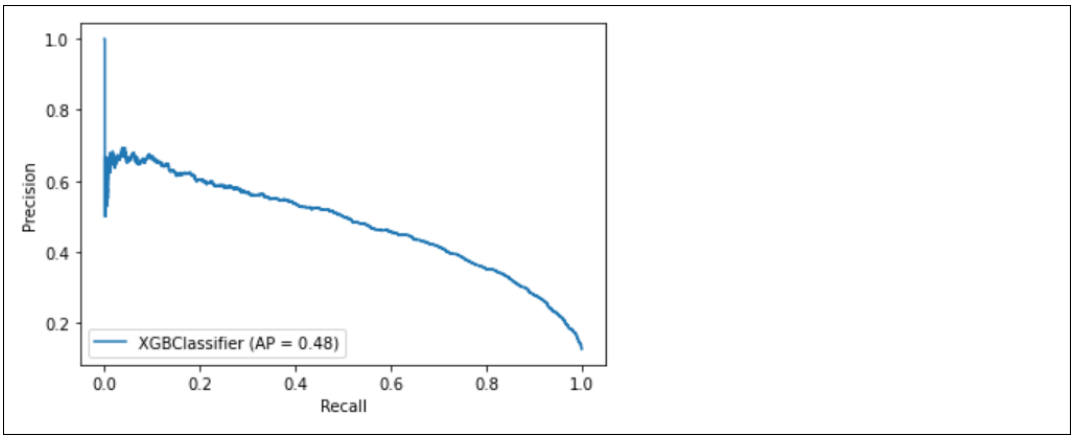
b) In the code cell, type the following:

```
plot_precision_recall_curve(xgb, X_test, y_test)

plt.show();
```

This plots a PRC to show the balance between precision and recall at different thresholds.

c) Run the code cell.

d) Examine the output.



The curve shows that the balance between precision and recall is somewhat low. However, PRCs are most useful in evaluating imbalanced datasets, so this isn't necessarily a problem for the oversampled dataset.

6. Generate a feature importance plot.

a) Scroll down and view the cell titled **Generate a feature importance plot**, then select the code cell below it.

b) In the code cell, examine the following:

```
def feature_importance_plot(model, X_train, n):
    """Plots feature importance. Only works for ensemble learning."""
    plt.figure(figsize = (8, 5))
    feat_importances = pd.Series(model.feature_importances_,
                                 index = X_train.columns)
    feat_importances.nlargest(n).plot(kind = 'barh')
    plt.title(f'Top {n} Features')
    plt.show()
```

This code has been provided for you. It's a function that will plot feature importances for the given model as a bar chart.

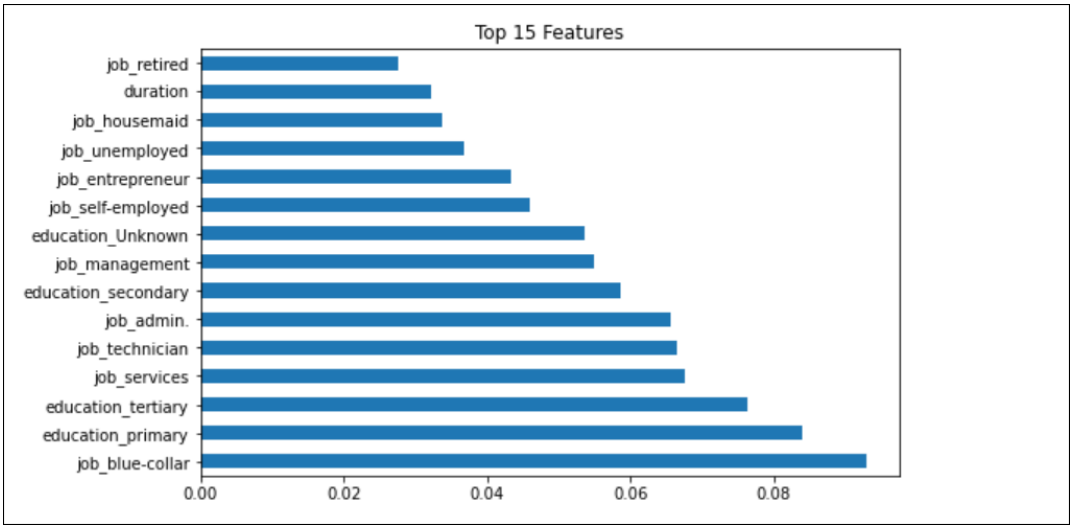
c) Run the code cell.

d) Select the next code cell, then type the following:

```
feature_importance_plot(xgb, X_train_SMOTE, 15)
```

e) Run the code cell.

f) Examine the output.



For the XGBoost model, features like `job_blue-collar`, `education_primary`, and `education_tertiary` contribute the most to the model's predictions. Features with low importance, like `job_retired`, could potentially be removed in order to improve performance.

7. Plot learning curves.

a) Scroll down and view the cell titled **Plot learning curves**, then select the code cell below it.

b) In the code cell, examine the following:

```
def plot_learning_curves(model, X_train, y_train):
    """Plots learning curves for model validation."""
    plt.figure(figsize = (5, 5))
    train_sizes, train_scores, test_scores = \
    learning_curve(model, X_train, y_train, cv = 5,
                   scoring = 'accuracy', n_jobs = 1,
                   shuffle = True,
                   train_sizes = np.linspace(0.01, 1.0, 5))

    # Means of training and test set scores.
    train_mean = np.mean(train_scores, axis = 1)
    test_mean = np.mean(test_scores, axis = 1)

    # Draw lines.
    plt.plot(train_sizes, train_mean, '--',
             color = '#111111', label = 'Training score')
    plt.plot(train_sizes, test_mean,
             color = '#111111', label = 'Cross-validation score')

    # Create plot.
    plt.title('Learning Curves')
    plt.xlabel('Training Set Size')
    plt.ylabel('Accuracy Score')
    plt.legend(loc = 'best')
    plt.tight_layout()

    plt.show()
```

This function, which has been provided for you, generates learning curves for the XGBoost model. Learning curves generate learning performance over time and can be used to identify models that might benefit from more training examples, as well as models that may be overfitting or underfitting.

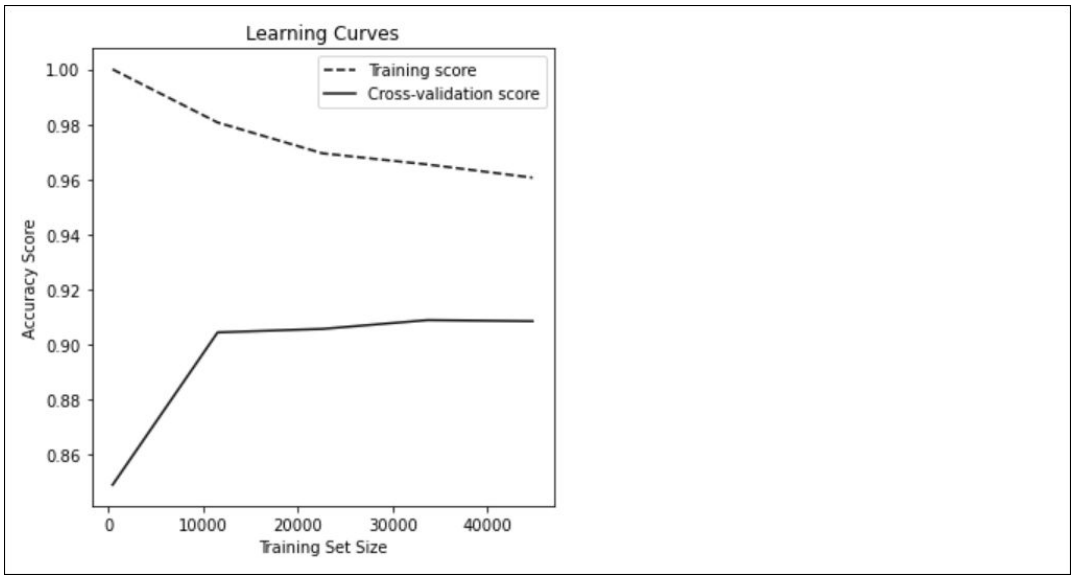
c) Run the code cell.

d) Select the next code cell, then type the following:

```
plot_learning_curves(xgb, X_train_SMOTE, y_train_SMOTE)
```

e) Run the code cell.

f) Examine the output.



Learning curves are plotted for both training and validation datasets. The gap between the two curves indicates that adding more training instances is likely to help, and the cross-validation curve could converge toward the training curve. The gap also indicates that the model suffers from some variance. Although the gap may look large, keep in mind that the scale of the y-axis starts at around 0.85 rather than 0. So, the gap is narrower than it might seem at first, which means that the variance is relatively low. Still, the variance isn't insignificant, so there are some signs of overfitting. Also, because both lines are close to 1 on the y-axis (i.e., they have high accuracy), the bias is also relatively low.

Increasing the number of training instances and applying regularization to the current learning algorithm would likely decrease the variance and increase the bias. You could also consider reducing the number of features to build a more simplified model.

8. Save the best model.

a) Scroll down and view the cell titled **Save the best model**, then select the code cell below it.

b) In the code cell, type the following:

```
pickle.dump(xgb, open('xgboost_classifier.pickle', 'wb'))
```

You'll save the XGBoost model to a binary pickle file so that you can use it later.

c) Run the code cell.

9. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on with the course.

MODULE 3

Develop Regression Models

The following labs are for Module 3: Develop Regression Models.

LAB 4-8

Training a Linear Regression Model

Data Files

~/Regression/Developing Regression Models.ipynb

~/Regression/data/users_data_final.pickle

Scenario

Now that you're done developing classification models, you want to turn your attention to developing models that can estimate continuous numeric values. The `total_amount_usd` that you created in the ETL phase by consolidating transactional data is of particular interest. It represents the total balance of each user's transactions with the bank over the given period of time. So, the `total_amount_usd` reveals either the debit that the user owes to the bank (negative values), or the credit that the bank extends to the user (positive values). You want to build models that can estimate a new customer's debit/credit based on various features.

As with your classification tasks, you won't just build one model, but several. You'll compare these regressors to determine which one best meets your needs. To start with, you'll train a very simple linear model to see how it handles the data.

---

1. Open the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Regression**.
      The **Regression** directory contains a subdirectory named **data** and a notebook file named **Developing Regression Models.ipynb**.
   c) Select **Developing Regression Models.ipynb** to open it.

2. Import the relevant software libraries.
   a) View the cell titled **Import software libraries**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.
   c) Verify that the version of Python is displayed, as are the versions of the other libraries that were imported.

3. Load and preview the data.
   a) Scroll down and view the cell titled **Load and preview the data**, then select the code cell below it.
   b) In the code cell, type the following:

```
users_data = pd.read_pickle('data/users_data_final.pickle')

users_data.head(n = 5)
```

   c) Run the code cell.

d) Examine the output.

	user_id	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_se
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	3.0	2143.00	1	0	0	0	0	0	
1	bb92765a-08de-4963-b432-496524b39157	0.0	1369.42	0	1	0	0	0	0	
2	573de577-49ef-42b9-83da-d3cfb817b5c1	2.0	2.00	0	0	1	0	0	0	
3	d6b66b9d-7c8f-4257-a682-e136f640b7e3	0.0	1369.42	0	0	0	1	0	0	
4	fade0b20-7594-4d9a-84cd-c02f79b1b526	1.0	1.00	0	0	0	0	0	0	

5 rows × 33 columns

This is the same dataset you used in your classification tasks. As a refresher, you'll retrieve some fundamental information about the dataset.

4. Check the shape of data.

a) Scroll down and view the cell titled **Check the shape of data**, then select the code cell below it.

b) In the code cell, type the following:

```
1 users_data.shape
```

c) Run the code cell.

d) Examine the output.

```
(45179, 33)
```

There are 45,179 records and 33 columns.

5. Check the data types.

a) Scroll down and view the cell titled **Check the data types**, then select the code cell below it.

b) In the code cell, type the following:

```
1 users_data.info()
```

c) Run the code cell.

d) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 45179 entries, 0 to 45215
Data columns (total 33 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   user_id              45179 non-null  object
 1   number_transactions  45179 non-null  float64
 2   total_amount_usd     45179 non-null  float64
 3   job_management       45179 non-null  int64
 4   job_technician       45179 non-null  int64
 5   job_entrepreneur     45179 non-null  int64
 6   job_blue-collar      45179 non-null  int64
 7   job_retired          45179 non-null  int64
 8   job_admin.           45179 non-null  int64
 9   job_services         45179 non-null  int64
 10  job_self-employed    45179 non-null  int64
 11  job_unemployed       45179 non-null  int64
 12  job_housemaid        45179 non-null  int64
```

All of the columns are in the proper data formats for machine learning, except for `user_id`, which you'll remove.

6. Explore the distribution of the target variable.

a) Scroll down and view the cell titled **Explore the distribution of the target variable**, then select the code cell below it.

b) In the code cell, type the following:

```
1 users_data.total_amount_usd.describe()
```

The target variable you're interested in studying is `total_amount_usd`.

c) Run the code cell.

d) Examine the output.

```
count     45179.000000
mean       1369.751283
std        2704.291321
min       -8019.000000
25%         160.000000
50%         862.000000
75%        1369.420000
max      102127.000000
Name: total_amount_usd, dtype: float64
```

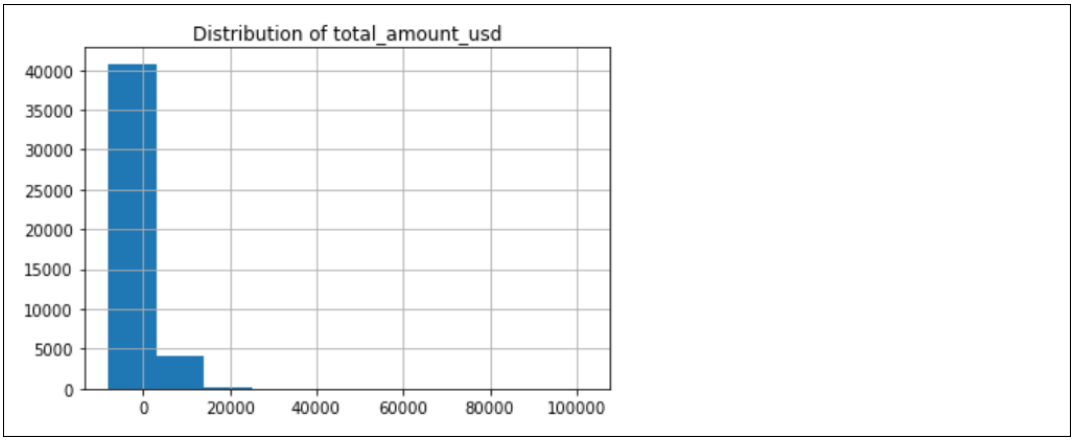
The largest debit is $−8,019, and the largest credit is $102,217.

e) Select the next code cell, then type the following:

```
1 users_data.total_amount_usd.hist()
2 plt.title('Distribution of total_amount_usd');
```

f) Run the code cell.

g) Examine the output.



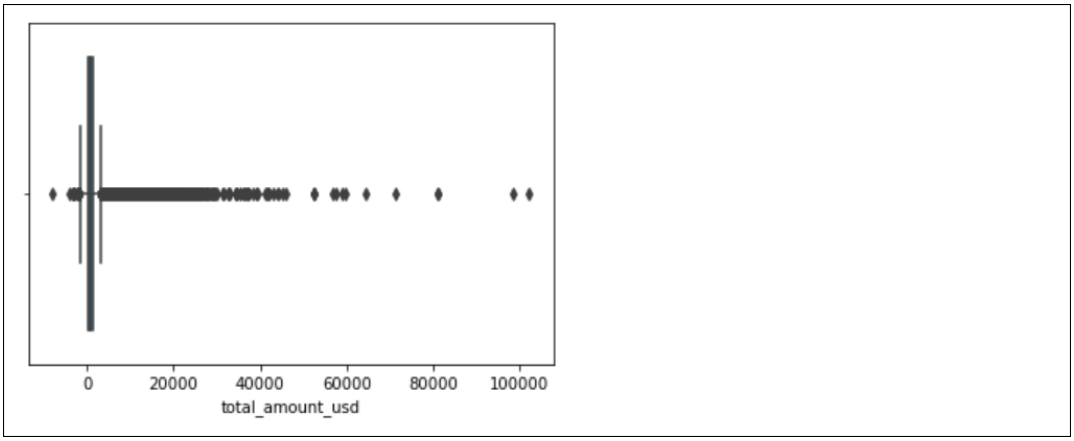
The data has a very heavy positive skew. Most users' total balances are around 0, whereas very few are above $20,000.

h) Select the next code cell, then type the following:

```
1 sns.boxplot(users_data.total_amount_usd);
```

i) Run the code cell.

j) Examine the output.



This box plot confirms that there are several outliers, particularly at the upper bounds of the dataset. You'll remove these outliers so they don't interfere with the model's training.

7. Identify and remove the outliers.

a) Scroll down and view the cell titled **Identify and remove the outliers**, then select the code cell below it.

b) In the code cell, type the following:

```
q1 = np.percentile(users_data.total_amount_usd, 25)
q3 = np.percentile(users_data.total_amount_usd, 75)
iqr = q3 - q1

lb = q1 - 1.5 * iqr
ub = q3 + 1.5 * iqr

print('Lower bound:', round(lb, 2))
print('Upper bound:', round(ub, 2))
```

This code will use the percentiles to determine the lower and upper bounds of the dataset.

c) Run the code cell.

d) Examine the output.

```
Lower bound: -1654.13
Upper bound: 3183.55
```

Any figure lower than $−1,654.13 is an outlier, as is anything above $3,183.55.

e) Select the next code cell, then type the following:

```
print('Number of users with total_amount_usd greater than UB:',
      users_data[(users_data.total_amount_usd >= ub)].shape[0])
print('Number of users with total_amount_usd lower than LB:  ',
      users_data[(users_data.total_amount_usd <= lb)].shape[0])
```

f) Run the code cell.

g) Examine the output.

```
Number of users with total_amount_usd greater than UB: 4110
Number of users with total_amount_usd lower than LB:   26
```

4,110 customers have total balances that are high outliers, whereas 26 have total balances that are low outliers.

h) Select the next code cell, then type the following:

```
users_data_wout_outliers = \
users_data[(users_data.total_amount_usd < ub) \
           & (users_data.total_amount_usd > lb)]

users_data_wout_outliers.shape
```

This code will remove the identified outliers from the dataset.

i) Run the code cell.

j) Examine the output.

```
(41043, 33)
```

There are now 41,043 rows.

8. Explore the dataset with the outliers removed.

a) Scroll down and view the cell titled **Explore the dataset with the outliers removed**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data_wout_outliers.describe()
```

c) Run the code cell.

d) Examine the output.

	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services
count	41043.000000	41043.000000	41043.000000	41043.000000	41043.000000	41043.000000	41043.000000	41043.000000	41043.000000
mean	3.131179	793.578253	0.202032	0.169237	0.033087	0.220257	0.047657	0.116488	0.094316
std	3.916997	761.407759	0.401521	0.374966	0.178867	0.414425	0.213043	0.320813	0.292271
min	0.000000	-1636.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	1.000000	123.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	2.000000	659.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	3.000000	1369.420000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
max	20.000000	3181.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

8 rows × 28 columns

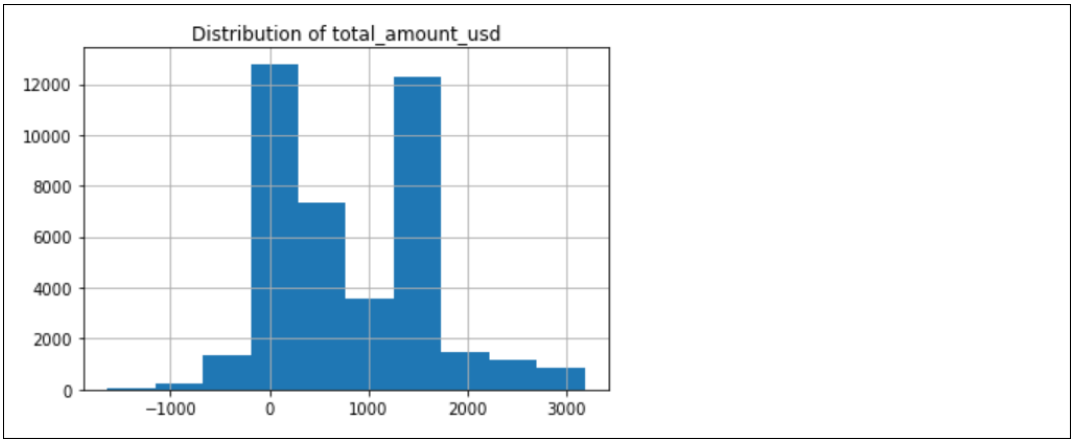
As you can from the `total_amount_usd` column, the minimum and maximum are now at the lower and upper bounds, respectively. Other values, like the mean, have also changed to reflect the removal of outliers.

e) Select the next code cell, then type the following:

```
plt.title('Distribution of total_amount_usd')
users_data_wout_outliers.total_amount_usd.hist();
```

f) Run the code cell.

g) Examine the output.



The distribution of the dataset now exhibits much less skew than before.

9. Split the data into target and features.

a) Scroll down and view the cell titled **Split the data into target and features**, then select the code cell below it.

b) In the code cell, type the following:

```
target_data = users_data_wout_outliers.total_amount_usd
features = users_data_wout_outliers.drop(['user_id', 'total_amount_usd'],
                                         axis = 1)
```

Since you're still performing supervised learning, you'll need to split the dataset into the features and the target variable (label).

c) Run the code cell.

10. Split the data into train and test sets.

a) Scroll down and view the cell titled **Split the data into train and test sets**, then select the code cell below it.

b) In the code cell, type the following:

```
X_train, X_test, y_train, y_test = train_test_split(features,
                                                    target_data,
                                                    test_size = 0.3)
```

You'll perform the holdout method for this dataset in much the same way you did for the classification models.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
print('Training data features: ', X_train.shape)
print('Training data target:   ', y_train.shape)
```

e) Run the code cell.

f) Examine the output.

```
Training data features:  (28730, 31)
Training data target:    (28730,)
```

The training set has 70% of the dataset's examples, or 28,730 rows.

11. Check the distribution of the test data.

a) Scroll down and view the cell titled **Check the distribution of the test data**, then select the code cell below it.

b) In the code cell, type the following:

```
print('Test data features: ', X_test.shape)
print('Test data target:   ', y_test.shape)
```

c) Run the code cell.

d) Examine the output.

```
Test data features:  (12313, 31)
Test data target:    (12313,)
```

The test data has 30% of the data examples, or 12,313 rows.

e) Select the next code cell, then type the following:

```
y_test.describe()
```

f) Run the code cell.

g) Examine the output.

```
count    12313.000000
mean       794.124498
std        765.840040
min      -1629.000000
25%        121.000000
50%        655.000000
75%       1369.420000
max       3181.000000
Name: total_amount_usd, dtype: float64
```

The summary statistics for the test set are as expected. Note that, for these models, you won't be scaling the data. In a production environment, you should consider doing so for certain models that benefit from this scaling. However, since most of the regression models you'll be training will either use simple linear algorithms or tree-based algorithms, scaling isn't as important. Scaling can also have an impact on the interpretability of the results for a numeric target variable, so that's another reason to forego it this time.

12. Train a linear regression model.

a) Scroll down and view the cell titled **Train a linear regression model**, then select the code cell below it.

b) In the code cell, type the following:

```
linreg = LinearRegression()
linreg.fit(X_train, y_train)
```

This code creates a linear regression object that uses the normal equation to determine the optimal model parameters. As with any other supervised algorithm, the model is fit with the input data.

c) Run the code cell.

13. Make predictions using the linear regression model.

a) Scroll down and view the cell titled **Make predictions using the linear regression model**, then select the code cell below it.

b) In the code cell, type the following:

```
linreg_y_pred = linreg.predict(X_test)
```

The model will generate `total_amount_usd` predictions for each customer.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
results = pd.concat([y_test.iloc[:5], X_test.iloc[:5]], axis = 1)
results.insert(1, 'total_pred', linreg_y_pred[:5].round(2))
results
```

This code will show the first five records of the test set and compare the actual `total_amount_usd` labels to the model's predictions.

e) Run the code cell.

f) Examine the output.

	total_amount_usd	total_pred	number_transactions	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services
33025	104.00	856.80	3.0	0	0	0	1	0	0	0
1893	1369.42	1129.94	0.0	0	1	0	0	0	0	0
26956	219.00	994.59	2.0	0	0	0	1	0	0	0
38050	170.00	829.65	2.0	0	0	0	1	0	0	0
19663	53.00	816.78	3.0	1	0	0	0	0	0	0

5 rows × 33 columns

In several cases, the model was far off in its predictions. For example, the model predicted a balance of $856.80 for the first customer, but they actually had a much smaller balance of $104.00. In other cases, like the second customer, the prediction was fairly close.

14. Obtain the linear regression model's score.

a) Scroll down and view the cell titled **Obtain the linear regression model's score**, then select the code cell below it.

b) In the code cell, type the following:

```
1 r2_score(y_test, linreg_y_pred)
```

Whereas accuracy is the default metric for classifiers in scikit-learn, R^2 (coefficient of determination) is the default metric for regressors. You'll learn more about regression metrics later, but for now, just know that the ideal R^2 value is 1.

c) Run the code cell.

d) Examine the output.

```
0.2332152712408806
```

The simple linear model's R^2 score is ~0.23. This isn't great when considered in isolation, but it's most useful when compared to other models.

15. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 4-9

Training Regression Trees and Ensemble Models

Data File

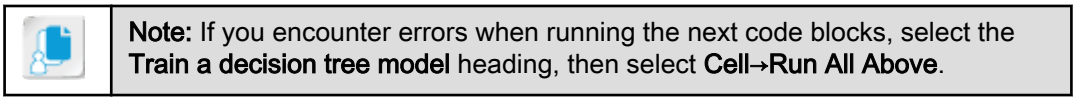
~/Regression/Developing Regression Models.ipynb

Scenario

The simpler linear regression model was able to make predictions on the `total_amount_usd` target variable, but there's definitely room for improvement. Decision trees and ensemble models are not just useful for classification, but regression as well. So you'll train several of these models to see if you can get some better results.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Regression/Developing Regression Models.ipynb**.
   c) Your progress from the previous lab should have been saved.



2. Train a decision tree model.
   a) Scroll down and view the cell titled **Train a decision tree model**, then select the code cell below it.
   b) In the code cell, type the following:

```
reg_tree = DecisionTreeRegressor()
reg_tree.fit(X_train, y_train)
```

   This code builds a decision tree object for regression tasks, then fits the training data.
   c) Run the code cell.

3. Make predictions using the decision tree model.
   a) Scroll down and view the cell titled **Make predictions using the decision tree model**, then select the code cell below it.
   b) In the code cell, type the following:

```
reg_tree_y_pred = reg_tree.predict(X_test)
```

   The decision tree will make predictions on the test set.
   c) Run the code cell.
   d) Select the next code cell, then type the following:

```
results['total_pred'] = reg_tree_y_pred[:5]
results
```

e) Run the code cell.
f) Examine the output.

	total_amount_usd	total_pred	number_transactions	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services
33025	104.00	2958.00	3.0	0	0	0	1	0	0	0
1893	1369.42	1369.42	0.0	0	1	0	0	0	0	0
26956	219.00	2787.00	2.0	0	0	0	1	0	0	0
38050	170.00	59.00	2.0	0	0	0	1	0	0	0
19663	53.00	862.00	3.0	1	0	0	0	0	0	0

5 rows × 33 columns

The predictions for the first five records are different than the linear regression model. Some of them seem to be considerably worse, like the prediction for the first user. The prediction for the second user was exactly on target, however. Keep in mind that this value was the mean you imputed for missing values, so the model may be overfitting to those examples.

4. Obtain the decision tree model's score.

a) Scroll down and view the cell titled **Obtain the decision tree model's score**, then select the code cell below it.
b) In the code cell, type the following:

```
r2_score(y_test, reg_tree_y_pred)
```

c) Run the code cell.
d) Examine the output.

```
-0.40313871451252936
```

The decision tree not only scored significantly worse than the linear regression model, but scored in the negative. This is very likely due to overfitting, which decision trees are prone to. You'll see if you can improve this score during the tuning process.

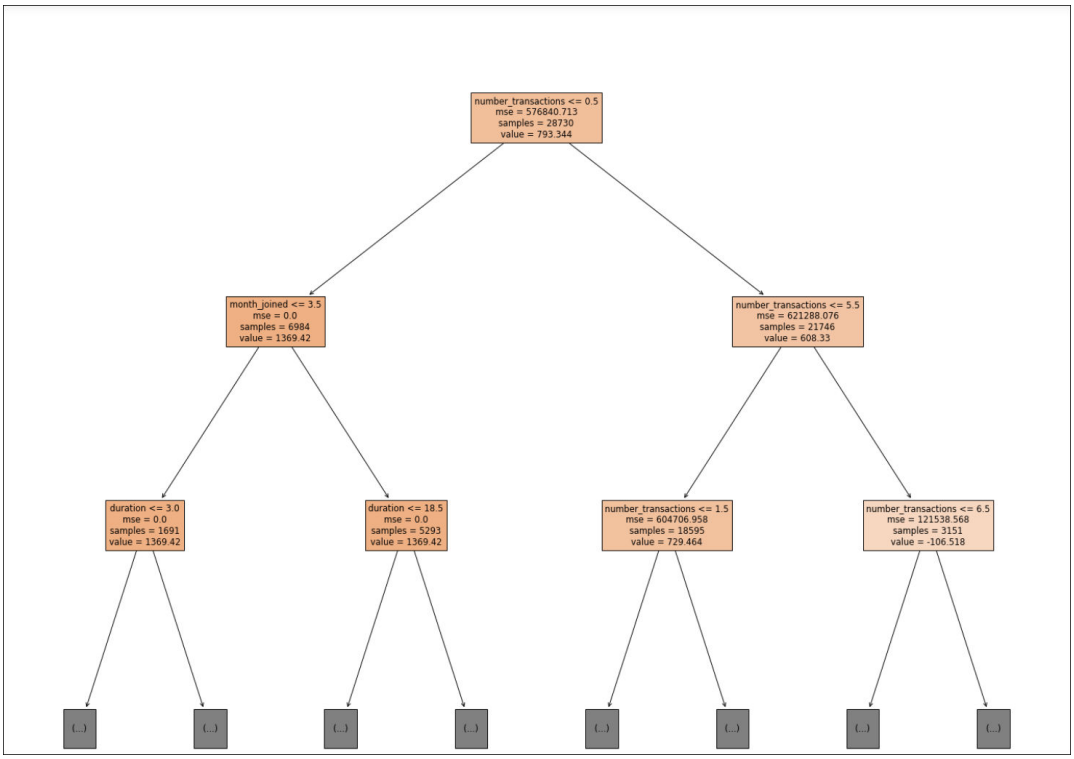
5. Visualize the decision tree.

a) Scroll down and view the cell titled **Visualize the decision tree**, then select the code cell below it.
b) In the code cell, type the following:

```
fig = plt.figure(figsize = (25, 20))
_ = tree.plot_tree(reg_tree,
                   feature_names = list(X_train.columns),
                   max_depth = 2,
                   filled = True)
```

As with classification, it helps to visualize the results of the tree to see how it's making splitting decisions.
c) Run the code cell.

d) Examine the output.



The root node of the tree is using `number_transactions` as the feature to start splitting from. In this case, it's checking to see if there actually *are* any transactions (i.e., less than 0.5 transactions—in other words, 0). If there are 0 transactions, the tree splits to a node that determines whether or not the user joined within the first three months of a year (`month_joined <= 3.5`). If there are more than 0 transactions, the tree splits to a node that considers 5 or fewer transactions—and so on.

6. Train a random forest model.

a) Scroll down and view the cell titled **Train a random forest model**, then select the code cell below it.

b) In the code cell, type the following:

```
rf = RandomForestRegressor()
rf.fit(X_train, y_train)
```

Now you'll aggregate multiple trees into a random forest model to see if the results improve.

c) Run the code cell.

7. Make predictions using the random forest model.

a) Scroll down and view the cell titled **Make predictions using the random forest model**, then select the code cell below it.

b) In the code cell, type the following:

```
rf_y_pred = rf.predict(X_test)
```

c) Run the code cell.

d) Select the next code cell, then type the following:

```
results['total_pred'] = rf_y_pred[:5]
results
```

e) Run the code cell.

f) Examine the output.

	total_amount_usd	total_pred	number_transactions	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services
33025	104.00	1187.61	3.0	0	0	0	1	0	0	0
1893	1369.42	1369.42	0.0	0	1	0	0	0	0	0
26956	219.00	1134.88	2.0	0	0	0	1	0	0	0
38050	170.00	514.93	2.0	0	0	0	1	0	0	0
19663	53.00	969.48	3.0	1	0	0	0	0	0	0

5 rows × 33 columns

Once again, the predicted values are different from the other models.

8. Obtain the random forest model's score.

a) Scroll down and view the cell titled **Obtain the random forest model's score**, then select the code cell below it.

b) In the code cell, type the following:

```
r2_score(y_test, rf_y_pred)
```

c) Run the code cell.

d) Examine the output.

```
0.2955761465520649
```

The random forest's score is much improved over the lone decision tree, and is slightly improved over the linear regression model (~0.30 vs. ~0.23).

9. Train a gradient boosting model.

a) Scroll down and view the cell titled **Train a gradient boosting model**, then select the code cell below it.

b) In the code cell, type the following:

```
xgb = XGBRegressor(objective = 'reg:squarederror', n_jobs = 1)
xgb.fit(X_train, y_train)
```

This code will build a gradient boosting model using the XGBoost algorithm. The `objective` argument refers to the learning objective, or how the model will attempt to minimize error.

Note: As with `eval_metric` in `XGBClassifier()`, the `objective` argument is being explicitly supplied to `XGBRegressor()` so as to suppress warnings.

c) Run the code cell.

10. Make predictions using the gradient boosting model.

a) Scroll down and view the cell titled **Make predictions using the gradient boosting model**, then select the code cell below it.

b) In the code cell, type the following:

```
xgb_y_pred = xgb.predict(X_test)
```

c) Run the code cell.
d) Select the next code cell, then type the following:

```
results['total_pred'] = xgb_y_pred[:5]
results
```

e) Run the code cell.
f) Examine the output.

	total_amount_usd	total_pred	number_transactions	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_servic
33025	104.00	803.491333	3.0	0	0	0	1	0	0	
1893	1369.42	1357.178345	0.0	0	1	0	0	0	0	
26956	219.00	939.850159	2.0	0	0	0	1	0	0	
38050	170.00	425.883087	2.0	0	0	0	1	0	0	
19663	53.00	735.929199	3.0	1	0	0	0	0	0	

5 rows × 33 columns

This model also made a different set of predictions for the target variable.

11. Obtain the gradient boosting model's score.

a) Scroll down and view the cell titled **Obtain the gradient boosting model's score**, then select the code cell below it.
b) In the code cell, type the following:

```
r2_score(y_test, xgb_y_pred)
```

c) Run the code cell.
d) Examine the output.

```
0.31022081348267105
```

The XGBoost model's score is ~0.31, which is only very slightly better than the random forest. Both are far from the ideal. Later, you'll use other metrics to compare the different models.

12. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.
b) Close the lab browser tab and continue on with the course.

LAB 4-10
Tuning Regression Models

Data File

~/Regression/Developing Regression Models.ipynb

Scenario

You've trained some basic regression models, and now you can begin to tune them. You want to tune the linear regression model to see if you can improve its performance. Having a well-performing simple linear model would be beneficial since it's easier to explain and isn't as prone to overfitting as some others. You also want to see if you can reduce the high levels of overfitting in your lone decision tree. Again, having a well-performing simple model has its advantages.

You could also try to improve the XGBoost model, but in the interest of time, you'll just focus on the linear model and the decision tree for now.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Regression/Developing Regression Models.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Define the parameter grid used to tune the linear regression model** heading, then select **Cell→Run All Above**.

2. Define the parameter grid used to tune the linear regression model.
   a) Scroll down and view the cell titled **Define the parameter grid used to tune the linear regression model**, then select the code cell below it.
   b) In the code cell, type the following:

```
param_grid = {'l1_ratio': [0.1, 0.5, 0.9],
              'alpha': [0.0001, 0.01, 0.1],
              'max_iter': [100, 1000, 10000]}

print(param_grid)
```

   This will be the parameter grid used in a grid search.
   c) Run the code cell.

d) Examine the output.

```
{'l1_ratio': [0.1, 0.5, 0.9], 'alpha': [0.0001, 0.01, 0.1], 'max_iter': [100, 1000, 10000]}
```

The parameter grid will be input into a linear model that uses elastic net regression to avoid overfitting. The hyperparameters you've defined in the grid are:

- `l1_ratio`, which determines the weight of the ℓ_1 norm (lasso) or ℓ_2 norm (ridge). In this case, the lower the value, the more the regularization penalty will shift toward ridge regression. The higher the value, the more it will shift toward lasso regression.
- `alpha`, which is the constant that determines the strength of regularization to apply. This is also called the λ (lambda) hyperparameter. Larger values lead to more regularization.
- `max_iter`, which indicates the maximum number of iterations the model will perform using gradient descent to minimize the model's cost.

3. Perform a grid search for optimal elastic net hyperparameters.

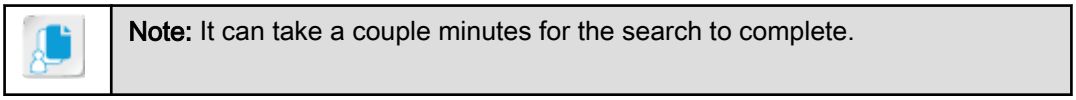
a) Scroll down and view the cell titled **Perform a grid search for optimal elastic net hyperparameters**, then select the code cell below it.

b) In the code cell, type the following:

```
model = ElasticNet()
gs = GridSearchCV(estimator = model,
                  param_grid = param_grid,
                  n_jobs = 1,
                  verbose = 2)

gs.fit(X_train, y_train)
```

In the case of scikit-learn, you don't apply elastic net regression as a hyperparameter to `LinearRegression()`, but as a separate function called `ElasticNet()`. The grid search will use the hyperparameter grid you just defined with the elastic net model.

c) Run the code cell.



d) Select the next code cell, then type the following:

```
print('Best R2 score:   ', round(gs.best_score_, 4))
print('Best parameters: ', gs.best_params_)
```

e) Run the code cell.

f) Examine the output.

```
Best R2 score:    0.2343
Best parameters:  {'alpha': 0.01, 'l1_ratio': 0.9, 'max_iter': 100}
```

The linear model improved, but only barely. It still has a score of ~0.23. Recall that you removed outliers manually, so that might have something to do with it. If you had more time, you could expand the hyperparameter grid to look for more values, but it's possible that linear regression is just not well-suited for this dataset. For now, the optimal hyperparameters are:

- An `alpha` regularization strength of `0.01`, which was in the middle of the grid values.
- An `l1_ratio` of `0.9`, which was the highest value in the grid.
- A `max_iter` of `100`, which was the lowest value in the grid.

4. Define the parameter grid used to tune the decision tree model.

a) Scroll down and view the cell titled **Define the parameter grid used to tune the decision tree model**, then select the code cell below it.
b) In the code cell, type the following:

```
param_grid = {'max_depth': [5, 10, 20],
              'min_samples_split': [10, 100, 1000],
              'min_samples_leaf': [10, 100, 1000]}

print(param_grid)
```

You'll run a new grid search, this time in an attempt to improve the poor performance of the single decision tree.
c) Run the code cell.
d) Examine the output.

```
{'max_depth': [5, 10, 20], 'min_samples_split': [10, 100, 1000], 'min_samples_leaf': [10, 100, 1000]}
```

This parameter grid will be input into the single decision tree algorithm. The hyperparameters you've defined in the grid are:

- `max_depth`, which determines the maximum depth of the tree before it stops splitting.
- `min_samples_split`, which defines how many samples are required in order to split a decision node.
- `min_samples_leaf`, which specifies how many samples are required to be at a leaf node.

5. Perform a grid search for optimal decision tree hyperparameters.

a) Scroll down and view the cell titled **Perform a grid search for optimal decision tree hyperparameters**, then select the code cell below it.
b) In the code cell, type the following:

```
model = DecisionTreeRegressor()
gs = GridSearchCV(estimator = model,
                  param_grid = param_grid,
                  n_jobs = 1,
                  verbose = 2)

gs.fit(X_train, y_train)
```

The grid search is being conducted in the same way as before, but using `DecisionTreeRegressor()` this time.
c) Run the code cell.

Note: This search should be much quicker than the previous one.

d) Select the next code cell, then type the following:

```
print('Best R2 score:   ', round(gs.best_score_, 4))
print('Best parameters: ', gs.best_params_)
```

e) Run the code cell.

f) Examine the output.

```
Best R2 score:    0.3357
Best parameters:  {'max_depth': 10, 'min_samples_leaf': 100, 'min_samples_split': 1000}
```

The decision tree's score is much improved at ~0.34, and the model appears to be better at learning patterns in the training data. The optimal hyperparameters are:

- A `max_depth` of `10`, which was in the middle of the grid values.
- A `min_samples_leaf` of `100`, which was also in the middle of the values.
- A `min_samples_split` of `1000`, which was the highest value in the grid.

6. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 4-11
Evaluating Regression Models

Data File

~/Regression/Developing Regression Models.ipynb

Scenario

Your preliminary training would suggest that some of your regression models are better than others. But, there are more ways to evaluate these models, and you want to be sure you have a more comprehensive understanding of their performance. So, you'll compare each model using multiple metrics, then take a deeper look at one of the models you think is best.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Regression/Developing Regression Models.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Compare evaluation metrics for each model** heading, then select **Cell→Run All Above**.

2. Compare evaluation metrics for each model.
   a) Scroll down and view the cell titled **Compare evaluation metrics for each model**, then select the code cell below it.

b) In the code cell, examine the following:

```
models = ['Linear Regression', 'Decision Tree',
          'Random Forest', 'XGBoost', 'Dummy Regressor']

metrics = ['R2', 'MAE', 'MSE']

pred_list = ['linreg_y_pred', 'reg_tree_y_pred',
             'rf_y_pred', 'xgb_y_pred', 'dummy_y_pred']

# Baseline algorithm.
dummy = DummyRegressor()
dummy.fit(X_train, y_train)
dummy_y_pred = dummy.predict(X_test)

scores = np.empty((0, 3))

for i in pred_list:
    scores = np.append(scores,
                       np.array([[r2_score(y_test, globals()[i]),
                                  mean_absolute_error(y_test, globals()[i]),
                                  mean_squared_error(y_test, globals()[i])]]),
                       axis = 0)

scores = np.around(scores, 4)

scoring_df = pd.DataFrame(scores, index = models, columns = metrics)
scoring_df.sort_values(by = 'MSE', ascending = True)
```

This code should look familiar. It'll output a table that has each trained regression model as a row, as well as a dummy regressor, and each evaluation metric as a column.

c) Run the code cell.

d) Examine the output.

	R2	MAE	MSE
XGBoost	0.3102	431.7623	404530.2009
Random Forest	0.2956	424.5235	413118.7611
Linear Regression	0.2332	531.7977	449691.1279
Dummy Regressor	-0.0000	656.6227	586463.9422
Decision Tree	-0.4031	561.2292	822889.4076

The table shows each model's scores for R^2, the mean absolute error (MAE), and the mean squared error (MSE). Lower values of MSE and MAE are best. The table is sorted by the lowest MSE, which is often the preferred regression metric, though MAE may be similarly useful. Both MSE and MAE tend to be more useful than R^2, which identifies how much of the model's variance can be explained. MSE and MAE don't tell you much intuitively on their own, like a classification metric would, so they're best used as a point of comparison between models.

Some conclusions you can draw include:

- The XGBoost model has the best MSE and R^2, and second best MAE.
- The random forest actually has a better MAE than XGBoost. Either of these models could be good candidates for "best" model, but for now, you'll go with XGBoost again.
- The unoptimized linear regression model is behind the ensemble methods, but is at least better than the dummy regressor baseline.
- The unoptimized decision tree has a better MAE than the dummy model, but worse R^2 and MSE.

3. Plot the residuals.

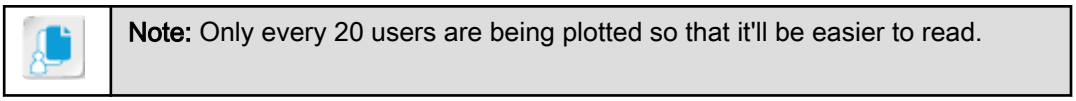
a) Scroll down and view the cell titled **Plot the residuals**, then select the code cell below it.

b) In the code cell, examine the following:

```
# Set up DataFrame for plotting.

resid_df = pd.DataFrame()
resid_df['total_amount_usd'] = y_test
resid_df['total_pred'] = xgb_y_pred
resid_df['residuals'] = resid_df['total_amount_usd'] - resid_df['total_pred']
resid_df = resid_df.sort_values('total_amount_usd')[::20]
resid_df['record_num'] = np.arange(len(resid_df))
```

You're going to build a residual plot from the XGBoost model, but first, you need to set up a `DataFrame` that has the residual values as well as the ground truth values and the predictions. A residual is just the magnitude of difference between the ground truth and the prediction. The closer the residual is to 0, the better. So, this kind of plot can help you visualize how far off the model is as compared to the prediction itself.

The users are being sorted by `total_amount_usd` so that you can more easily track how the increase in user balance does or does not affect the residuals. The sorted users are assigned a `record_num` that starts at 0 (the user with the lowest balance) and increments by one for each user with the next highest balance



c) Run the code cell.

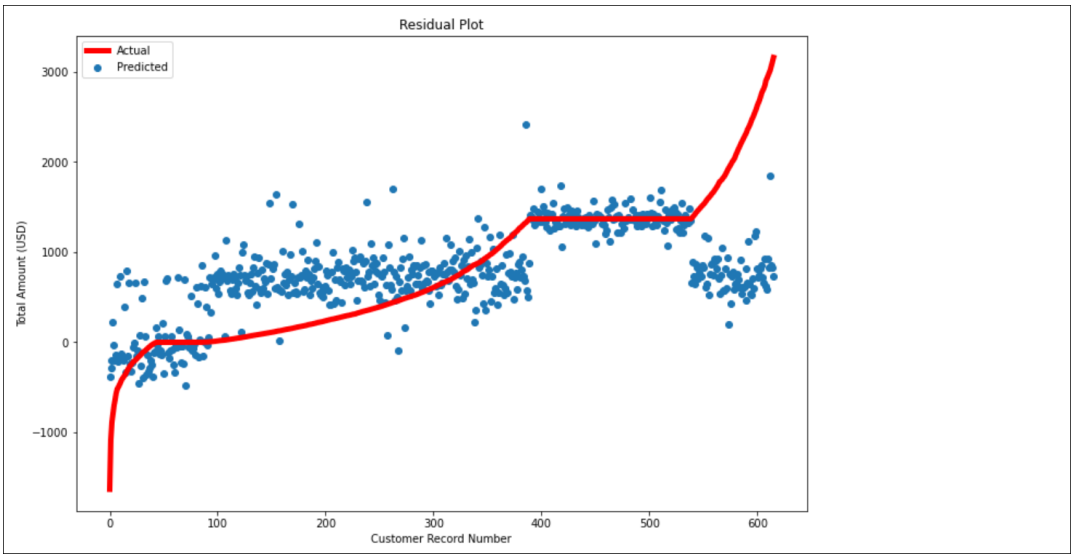
d) Select the next code cell, then type the following:

```
plt.figure(figsize = (12, 8))

plt.plot(resid_df['record_num'], resid_df['total_amount_usd'],
         color = 'red', linewidth = 5)
plt.scatter(resid_df['record_num'], resid_df['total_pred'])

plt.legend(['Actual', 'Predicted'])
plt.title('Residual Plot')
plt.ylabel('Total Amount (USD)')
plt.xlabel('Customer Record Number')
plt.show();
```

This code plots the predictions as a scatter plot, and the ground truth values as a line on that same plot.

e) Run the code cell.

f) Examine the output.



The users are plotted along the x-axis, where an increase in user record number indicates an increase in the ground truth of `total_amount_usd`. The y-axis shows the user's total balance, either predicted or ground truth. The line indicates the ground truth, whereas the dots indicate the model's predictions. The closer a dot is to the line where they both appear on the x-axis, the smaller the residual, and therefore the less error in the prediction.

For user record numbers around 150, the differences between the prediction dot and the ground truth line appears to be largest. In other words, the residuals here are relatively large. The actual balance for these users is around $0, whereas the predictions could go as high as $1,500+. Compare this to record numbers between 400 and 500, where the dots are very close to the flat portion of the line, indicating small residuals. The ground truth line is flat here because the values are the imputed mean of $1,369.42.

Ultimately, you can use a plot like this to detect patterns in your model's effectiveness. The model might be better at predicting certain ranges of the target variable compared to other ranges.

4. Generate a feature importance plot.

a) Scroll down and view the cell titled **Generate a feature importance plot**, then select the code cell below it.

b) In the code cell, examine the following:

```
def feature_importance_plot(model, X_train, n):
    """Plots feature importance. Only works for ensemble learning."""
    plt.figure(figsize = (8, 5))
    feat_importances = pd.Series(model.feature_importances_,
                                 index = X_train.columns)
    feat_importances.nlargest(n).plot(kind = 'barh')
    plt.title(f'Top {n} Features')
    plt.show()
```

This is code you've seen before: it will generate a feature importance plot.

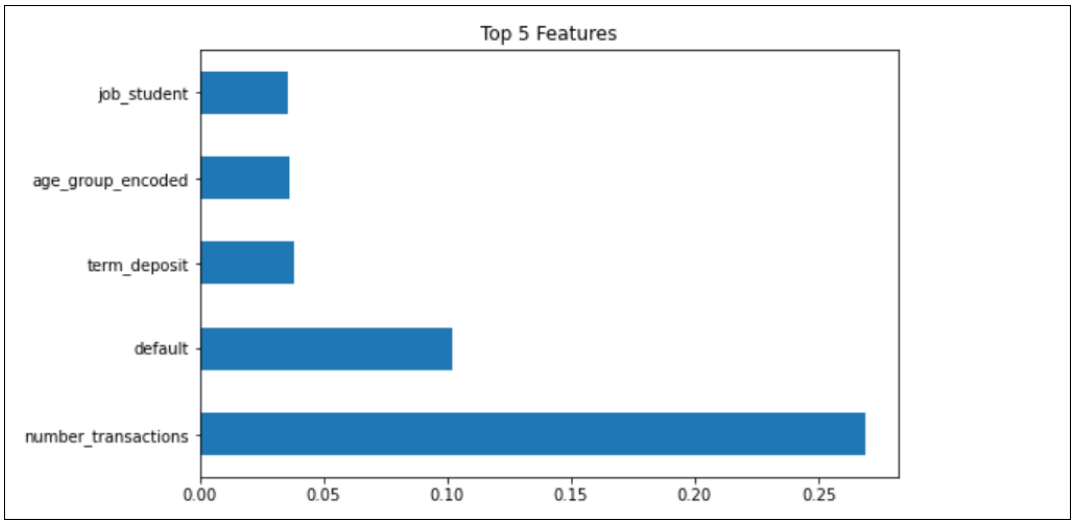
c) Run the code cell.

d) Select the next code cell, then type the following:

```
feature_importance_plot(xgb, X_train, 5)
```

e) Run the code cell.

f) Examine the output.



The most important feature to the XGBoost model was the number of transactions. This makes sense, as the number of transactions would likely impact the total balance a customer has. The second most important feature was whether or not the customer defaulted on a loan. Other features tend to be much less important in predicting the target variable.

5. Plot learning curves.

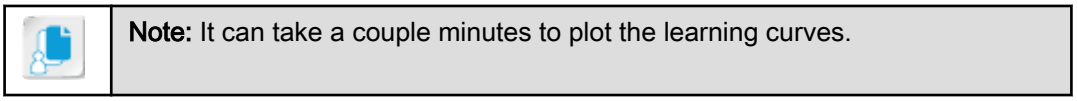
a) Scroll down and view the cell titled **Plot learning curves**, then select the code cell below it.

b) In the code cell, examine the following:

```
def plot_learning_curves(model, X_train, y_train):
    """Plots learning curves for model validation."""
    plt.figure(figsize = (5, 5))
    train_sizes, train_scores, test_scores = \
    learning_curve(model, X_train, y_train, cv = 5,
                   scoring = 'neg_mean_squared_error',
                   n_jobs = 1,
                   shuffle = True,
                   train_sizes = np.linspace(0.01, 1.0, 5))

    # Means of training and test set scores.
    train_mean = np.mean(train_scores, axis = 1)
    test_mean = np.mean(test_scores, axis = 1)

    # Draw lines.
    plt.plot(train_sizes, train_mean, '--',
             color = '#111111', label = 'Training score')
    plt.plot(train_sizes, test_mean,
             color = '#111111', label = 'Cross-validation score')

    # Create plot.
    plt.title('Learning Curves')
    plt.xlabel('Training Set Size')
    plt.ylabel('Negative MSE')
    plt.legend(loc = 'best')
    plt.tight_layout()

    plt.show()
```

This is another block of code that you saw in the classification activities. One important thing to point out is that the plot is using the negative MSE instead of regular MSE. Making MSE negative simply means that higher values are ideal, rather than lower ones. This is useful when plotting, because it makes plots look consistent across other metrics, and the plot will generally be easier to read.

c) Run the code cell.

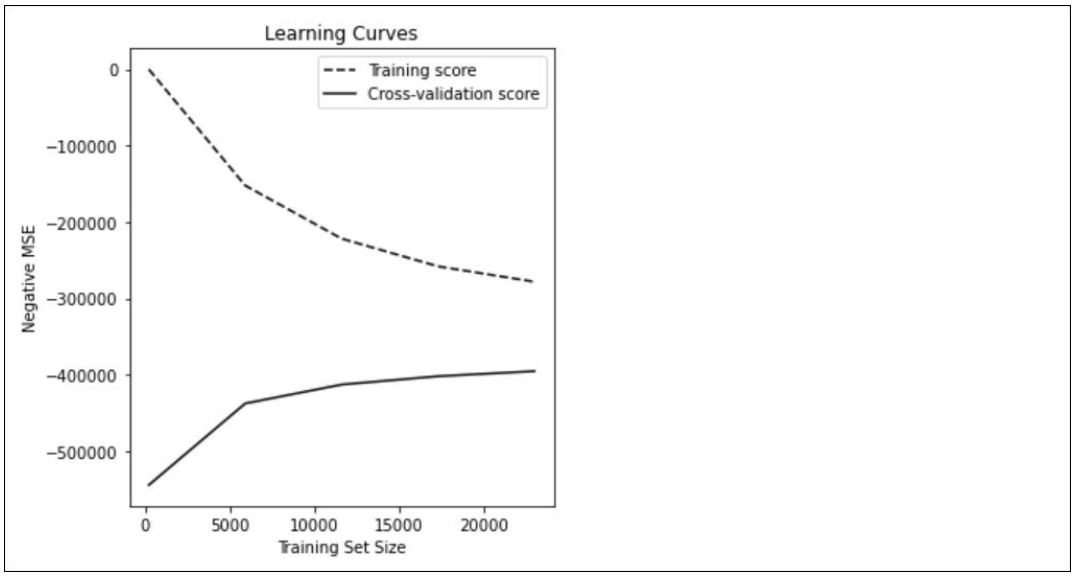


d) Select the next code cell, then type the following:

```
plot_learning_curves(xgb, X_train, y_train)
```

e) Run the code cell.

f) Examine the output.



The training and cross-validation scores haven't yet converged, so the model will probably improve by adding more examples. Looking at the gap and positioning of the curves, the model does appear to be exhibiting some bias and variance issues. This could be improved by tuning the hyperparameters further, so in a production environment, you'd want to keep working on the model. For now, you'll take the model as is.

6. Save the best model.

a) Scroll down and view the cell titled **Save the best model**, then select the code cell below it.

b) In the code cell, type the following:

```
pickle.dump(xgb, open('xgboost_regressor.pickle', 'wb'))
```

The XGBoost regressor will be saved as a binary pickle file.

c) Run the code cell.

7. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on with the course.

MODULE 4

Develop Clustering Models

The following labs are for Module 4: Develop Clustering Models.

LAB 4-12

Training a *k*-Means Clustering Model

Data Files

~/Clustering/Developing Clustering Models.ipynb

~/Clustering/data/users_data_final.pickle

Scenario

Although supervised learning has been very useful to the project so far, there are other ways you can learn more from GCNB's customer data that don't involve labels. The marketing department might shape their campaigns to maximize term deposit subscriptions, but that's just one of many factors that define customers and their behavior. The business would benefit greatly from being able to target specific segments of customers for their marketing campaigns. Targeted marketing can lead to greater customer engagement with many different facets of the business. After all, the bank offers more services than just term deposits.

So, you decide to see how you can segment the bank's customers using cluster analysis. There are a few different methods you can try, but you'll start by building a model that leverages the *k*-means algorithm for generating clusters.

---

1. Open the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Clustering**.
      The **Clustering** directory contains a subdirectory named **data** and a notebook file named **Developing Clustering Models.ipynb**.
   c) Select **Developing Clustering Models.ipynb** to open it.

2. Import the relevant software libraries.
   a) View the cell titled **Import software libraries**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.
   c) Verify that the version of Python is displayed, as are the versions of the other libraries that were imported.

3. Load and preview the data.
   a) Scroll down and view the cell titled **Load and preview the data**, then select the code cell below it.
   b) In the code cell, type the following:

```
users_data = pd.read_pickle('data/users_data_final.pickle')

users_data.head(n = 5)
```

   c) Run the code cell.

d) Examine the output.

	user_id	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_se
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	3.0	2143.00	1	0	0	0	0	0	
1	bb92765a-08de-4963-b432-496524b39157	0.0	1369.42	0	1	0	0	0	0	
2	573de577-49ef-42b9-83da-d3cfb817b5c1	2.0	2.00	0	0	1	0	0	0	
3	d6b66b9d-7c8f-4257-a682-e136f640b7e3	0.0	1369.42	0	0	0	1	0	0	
4	fade0b20-7594-4d9a-84cd-c02f79b1b526	1.0	1.00	0	0	0	0	0	0	

5 rows × 33 columns

This is the same dataset of GCNB customer information that you've been using. You'll want to make sure everything's ready to be used in unsupervised learning.

4. Check the shape of the data.

a) Scroll down and view the cell titled **Check the shape of the data**, then select the code cell below it.
b) In the code cell, type the following:

```
1 users_data.shape
```

c) Run the code cell.
d) Examine the output.

```
(45179, 33)
```

There are 45,179 records and 33 columns.

5. Check the data types.

a) Scroll down and view the cell titled **Check the data types**, then select the code cell below it.
b) In the code cell, type the following:

```
1 users_data.info()
```

c) Run the code cell.

d) Examine the output.

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 45179 entries, 0 to 45215
Data columns (total 33 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   user_id              45179 non-null  object
 1   number_transactions  45179 non-null  float64
 2   total_amount_usd     45179 non-null  float64
 3   job_management       45179 non-null  int64
 4   job_technician       45179 non-null  int64
 5   job_entrepreneur     45179 non-null  int64
 6   job_blue-collar      45179 non-null  int64
 7   job_retired          45179 non-null  int64
 8   job_admin.           45179 non-null  int64
 9   job_services         45179 non-null  int64
 10  job_self-employed    45179 non-null  int64
 11  job_unemployed       45179 non-null  int64
 12  job_housemaid        45179 non-null  int64
 13  job_student          45179 non-null  int64
 14  education_tertiary   45179 non-null  int64
```

All of the columns are in the proper data formats for machine learning, except for `user_id`, which you'll remove.

6. Filter by demographics data.

a) Scroll down and view the cell titled **Filter by demographics data**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data_demographics = \
users_data.filter(regex = 'education|job|age|single')

users_data_demographics.head(n = 3)
```

Since you're interested in clustering customers by their characteristics, rather than their direct interactions with the bank, you'll filter the data to only include demographic features.

c) Run the code cell.

d) Examine the output.

	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services	job_self-employed	job_unemployed	job_housemaid	job_stude
0	1	0	0	0	0	0	0	0	0	0	
1	0	1	0	0	0	0	0	0	0	0	
2	0	0	1	0	0	0	0	0	0	0	

As expected, the resulting `DataFrame` only includes demographic information.

7. Scale the data.

a) Scroll down and view the cell titled **Scale the data**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data_demographics.describe()
```

c) Run the code cell.

d) Examine the output.

	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services	job_self-employed	job_unemployed	job_hous
count	45179.000000	45179.000000	45179.000000	45179.000000	45179.000000	45179.000000	45179.000000	45179.000000	45179.000000	45179.0
mean	0.209234	0.168043	0.032869	0.215255	0.050068	0.114389	0.091901	0.034906	0.028797	0.0
std	0.406767	0.373908	0.178296	0.411004	0.218087	0.318287	0.288889	0.183543	0.167236	0.1
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
75%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.0

Because k-means is a distance-based algorithm, it's a good idea to scale all of these features.

e) Select the next code cell, then type the following:

```
# Standardize the data.

scaler = StandardScaler()

scaler.fit(users_data_demographics)
users_data_scaled = scaler.transform(users_data_demographics)

print('New standard deviation: ', users_data_scaled.std())
print('New mean:               ', round(users_data_scaled.mean()))
```

This code standardizes the data so that the mean of the distribution is 0 and the standard deviation is 1.

f) Run the code cell.

g) Examine the output.

```
New standard deviation:  1.0
New mean:                0
```

As expected, the data has been scaled using the z-score.

8. Train a k-means clustering model.

a) Scroll down and view the cell titled **Train a *k*-means clustering model**, then select the code cell below it.

b) In the code cell, type the following:

```
# Specify initial number clusters.

n_clusters = 5
```

Unfortunately, the problem at hand doesn't really suggest a "good" number of clusters to divide the customers into. Without domain knowledge to guide you, you'll need to start clustering using an arbitrary value for k. You'll be able to tune and evaluate this value later to determine a better number of clusters.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
# Build k-means model.

kmeans = KMeans(n_clusters = n_clusters, random_state = 10)

# Fit scaled data to model.

kmeans.fit(users_data_scaled)
```

This code constructs the k-means model using the scaled data. It'll divide the data into the number of clusters you just specified (5).

e) Run the code cell.

9. Generate the clusters.

a) Scroll down and view the cell titled **Generate the clusters**, then select the code cell below it.

b) In the code cell, type the following:

```
y_kmeans = kmeans.predict(users_data_scaled)

results = pd.DataFrame(users_data_demographics)
results.insert(0, 'cluster', y_kmeans)
results.head()
```

The `predict()` function is used to assign cluster labels to each data example. In this case, you'll assign labels to the existing training set rather than a new set.

c) Run the code cell.

d) Examine the output.

	cluster	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services	job_self-employed	job_unemployed	job_housemaid	j
0	0	1	0	0	0	0	0	0	0	0	0	
1	2	0	1	0	0	0	0	0	0	0	0	
2	2	0	0	1	0	0	0	0	0	0	0	
3	4	0	0	0	1	0	0	0	0	0	0	
4	2	0	0	0	0	0	0	0	0	0	0	

The new `cluster` column indicates what cluster each data example is in. The clusters are simply numbers, as it's up to you to determine what each cluster means qualitatively. The first customer is in cluster `0`, the second, third, and fifth customers are in cluster `2`, and the fourth customer is in cluster `4`. There isn't necessarily a common feature shared among all three examples in cluster `2`, though they appear to be people that are relatively young and have less than a tertiary education. However, focusing on more than just three examples could reveal more obvious patterns.

10. Visualize the number of users in each cluster.

a) Scroll down and view the cell titled **Visualize the number of users in each cluster**, then select the code cell below it.

b) In the code cell, examine the following:

```
def cluster_bar(cluster_labels):
    """Create a bar chart to show number of users in each cluster."""
    pd.DataFrame(Counter(cluster_labels).most_common()). \
    set_index(0).plot.bar(legend = None)

    plt.title('Distribution of Clusters')
    plt.xlabel('Cluster ID')
    plt.xticks(rotation = 0)
    plt.ylabel('Number of users in cluster');
```

This function will plot the number of customers in each cluster.

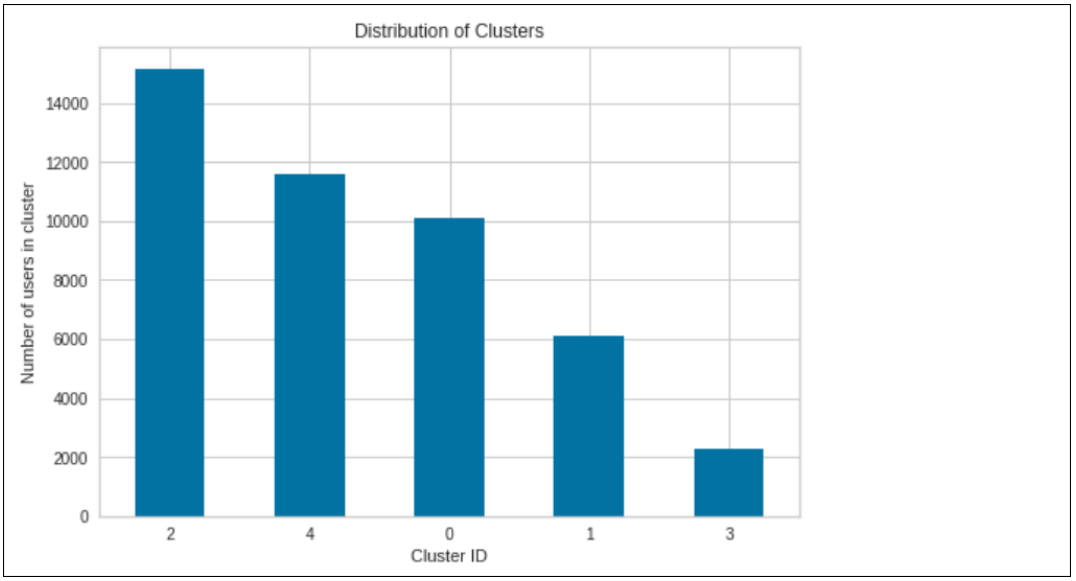
c) Run the code cell.

d) Select the next code cell, then type the following:

```
cluster_bar(y_kmeans)
```

e) Run the code cell.

f) Examine the output.



In this model, cluster `2` includes the most customers. Cluster `3` has the least. The decreasing number of customers in each cluster seems to follow a relatively stable pattern; i.e., no one cluster dominates all the others, nor is one cluster dominated by the others.

11. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 4-13

Training a Hierarchical Clustering Model

Data File

~/Clustering/Developing Clustering Models.ipynb

Scenario

The *k*-means approach is the most common method for cluster analysis in unsupervised learning, and it seems to be a good choice for the GCNB dataset. Hierarchical clustering is less common and has more limited applications, and may not be as suitable for this dataset. Still, like any other type of machine learning, it's always a good idea to try out multiple algorithms when time permits. So, you'll generate a hierarchical clustering model to see how it groups customers.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Clustering/Developing Clustering Models.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Filter the data by education and scale it** heading, then select **Cell→Run All Above**.

2. Filter the data by education and scale it.
   a) Scroll down and view the cell titled **Filter the data by education and scale it**, then select the code cell below it.
   b) In the code cell, type the following:

```
users_data_reduced = users_data.filter(regex = 'education')
users_data_reduced.head(n = 3)
```

Hierarchical clustering might benefit from a smaller feature space, so you'll filter the dataset to only include education level for this module.

   c) Run the code cell.
   d) Examine the output.

	education_tertiary	education_secondary	education_Unknown	education_primary
0	1	0	0	0
1	0	1	0	0
2	0	1	0	0

All four education categories are listed.

e) Select the next code cell, then type the following:

```
# Standardize the data.

scaler = StandardScaler()

scaler.fit(users_data_reduced)
users_data_reduced_scaled = scaler.transform(users_data_reduced)
```

Since all of the data is in binary format, it's not strictly necessary to standardize it, but you'll do it anyway for consistency.

f) Run the code cell.

3. Train a hierarchical clustering model.

a) Scroll down and view the cell titled **Train a hierarchical clustering model**, then select the code cell below it.

b) In the code cell, type the following:

```
agglom = AgglomerativeClustering(n_clusters = n_clusters,
                                 affinity = 'euclidean',
                                 linkage = 'single')

agglom.fit(users_data_reduced_scaled)
y_agglom = agglom.fit_predict(users_data_reduced_scaled)

results['cluster'] = y_agglom
results.head()
```

This code builds a hierarchical agglomerative clustering (HAC) model, which employs a bottom-up approach to generating clusters. Aside from the number of clusters, which is the same as the k-means model (5), the defined hyperparameters are:

- `linkage`, which refers to the linkage criterion. The linkage criterion specifies the distance metric to use when comparing two sets of data examples. In this case, the model is using the `'single'` method, which uses the minimum of the distances between all data examples in the two sets. This enables the model to determine which clusters to merge. Although not as effective as other linkage criteria, it tends to be more efficient on larger datasets.
- `affinity`, which is the method used to compute the linkage. The `'euclidean'` method refers to Euclidean distance, which is the distance of a straight line drawn between two points in Euclidean space. This is the most intuitive measurement of distance, though there are many others.

c) Run the code cell.

d) Examine the output.

	cluster	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services	job_self-employed	job_unemployed	job_housemaid	j
0	3	1	0	0	0	0	0	0	0	0	0	
1	1	0	1	0	0	0	0	0	0	0	0	
2	1	0	0	1	0	0	0	0	0	0	0	
3	0	0	0	0	1	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	0	

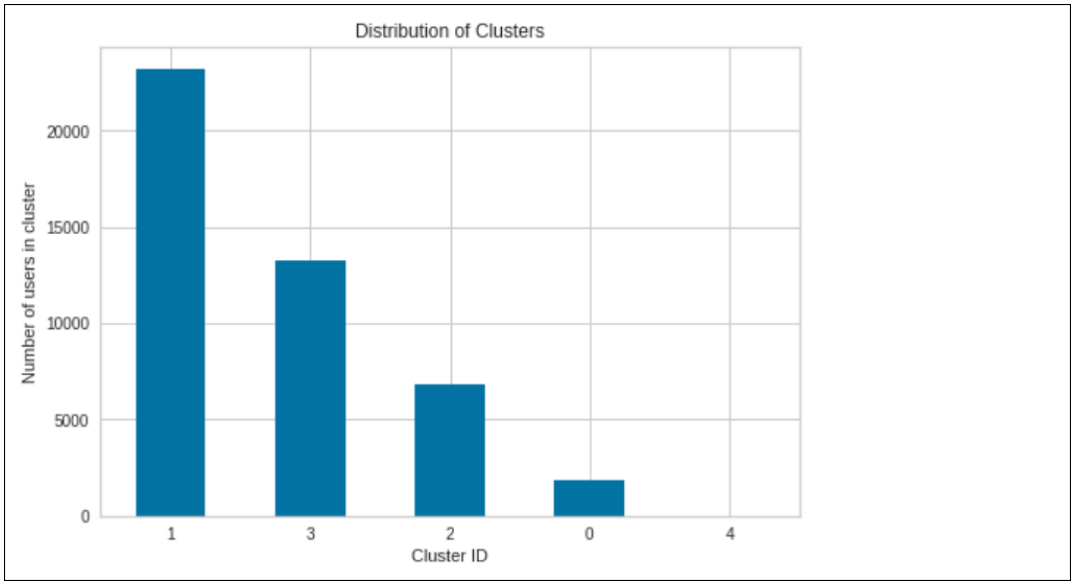
The agglomerative model produced different results for the first five customers as compared to the k-means model. The first customer is in cluster `3`, the second and third customers are in cluster `1`, and the fourth and fifth customers are in cluster `0`.

e) Select the next code cell, then type the following:

```
cluster_bar(y_agglom)
```

f) Run the code cell.

g) Examine the output.



Cluster `1` has the most customers in it, whereas cluster `4` seems to have hardly any. While it's interesting to see the differences between the two models, ultimately, the k-means model is probably the better choice for this data. Later on, you'll focus mostly on tuning and evaluating the k-means model.

4. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 4-14
Tuning Clustering Models

Data Files

~/Clustering/Developing Clustering Models.ipynb

~/Clustering/data/users_data_demo_pca.pickle

Scenario

You developed an initial *k*-means clustering model based on an arbitrary choice of 5 clusters. However, you want to adjust the model to see how it changes in response to a different number of clusters. Also, earlier in the preprocessing phase, you reduced the dimensionality of the user demographics data. So, you'll load this reduced dataset to make the clusters easier to visualize on a chart.

---

1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Clustering/Developing Clustering Models.ipynb**.
   c) Your progress from the previous lab should have been saved.

Note: If you encounter errors when running the next code blocks, select the **Adjust the number of clusters in the *k*-means clustering model** heading, then select **Cell→Run All Above**.

2. Adjust the number of clusters in the *k*-means clustering model.
   a) Scroll down and view the cell titled **Adjust the number of clusters in the *k*-means clustering model**, then select the code cell below it.
   b) In the code cell, type the following:

```
n_clusters = 10

kmeans = KMeans(n_clusters = n_clusters, random_state = 10)

kmeans.fit(users_data_scaled)
```

   This time, the number of clusters the model will output is 10. Experimenting with different numbers of clusters can lead to interesting results.
   c) Run the code cell.

3. Examine the results.
   a) Scroll down and view the cell titled **Examine the results**, then select the code cell below it.
   b) In the code cell, type the following:

```
y_kmeans = kmeans.predict(users_data_scaled)

results['cluster'] = y_kmeans
results.head()
```

c) Run the code cell.

d) Examine the output.

	cluster	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services	job_self-employed	job_unemployed	job_housemaid	j
0	0	1	0	0	0	0	0	0	0	0	0	
1	1	0	1	0	0	0	0	0	0	0	0	
2	8	0	0	1	0	0	0	0	0	0	0	
3	4	0	0	0	1	0	0	0	0	0	0	
4	4	0	0	0	0	0	0	0	0	0	0	

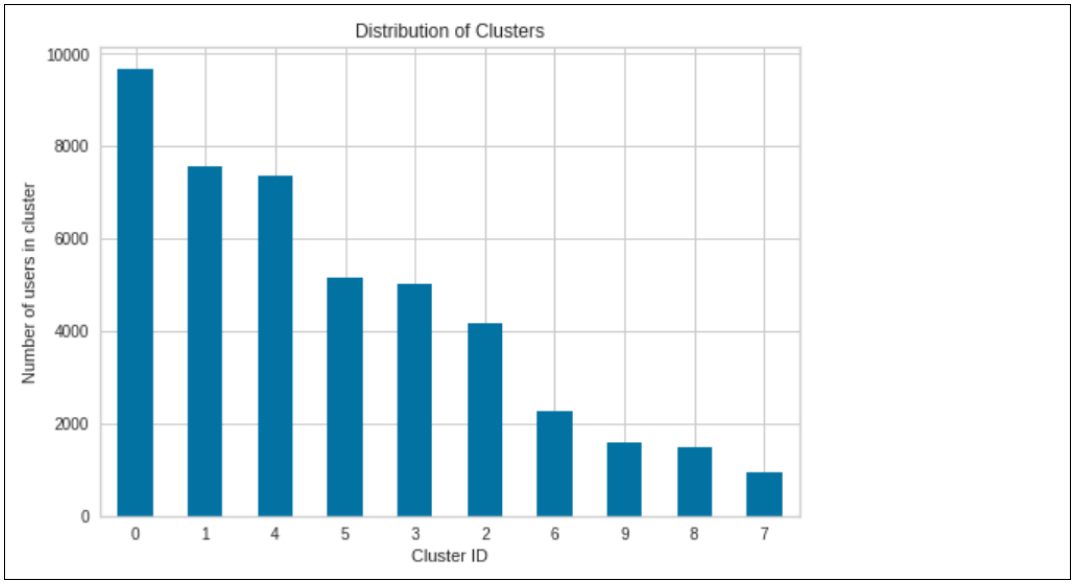
Once again, the results of the first five customers have changed. The first customer is still in cluster `0`, but the second customer is in cluster `1`, the third in cluster `8`, and the fourth and fifth in cluster `4`.

e) Select the next code cell, then type the following:

```
1 cluster_bar(y_kmeans)
```

f) Run the code cell.

g) Examine the output.



Cluster `0` seems to have the most representation among the customers, whereas cluster `7` has the least. As before, the spread of each cluster seems to decrease in a relatively stable way.

4. Load the PCA-reduced demographics data.

a) Scroll down and view the cell titled **Load the PCA-reduced demographics data**, then select the code cell below it.

b) In the code cell, type the following:

```
pca_df = pd.read_pickle('data/users_data_demo_pca.pickle')

pca_df = pd.concat([pca_df, pd.DataFrame(y_kmeans)], axis = 1). \
rename(columns = {0: 'cluster'})

pca_df
```

This code loads the demographics data you reduced to two features using principal component analysis (PCA). The reduced data will make it easier to visualize the clusters on a scatter plot.

c) Run the code cell.

d) Examine the output.

	PCA1	PCA2	cluster
0	2.557545	1.079613	0
1	-0.820505	-1.750581	1
2	-0.576607	-0.551404	8
3	-0.541647	1.593626	4
4	0.385598	-0.614768	4
...	...	...	...
45174	1.160485	0.073978	1
45175	-0.660738	4.371075	6
45176	-1.327676	2.367745	6
45177	-1.575612	1.040206	4
45178	-0.632467	-0.134990	8

45179 rows × 3 columns

As before, the reduced demographics data is represented as two numeric columns. The cluster that each user belongs to is added as a third column.

5. Visualize the clusters on the reduced data.

a) Scroll down and view the cell titled **Visualize the clusters on the reduced data**, then select the code cell below it.

b) In the code cell, type the following:

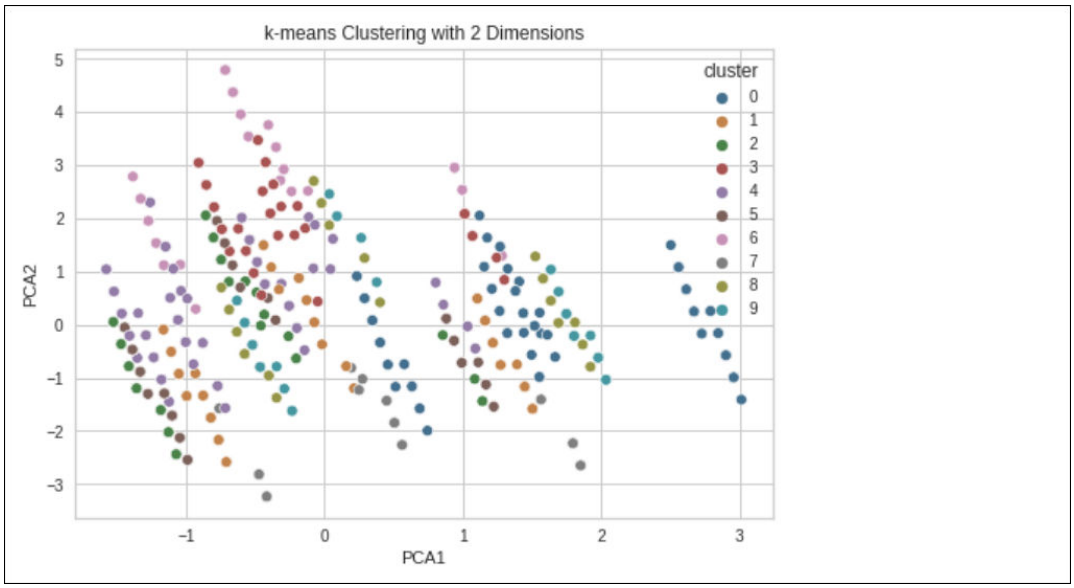
```
cmap = sns.color_palette('tab10', n_colors = n_clusters, desat = .5)

sns.scatterplot(x = 'PCA1', y = 'PCA2',
                hue = 'cluster', data = pca_df[::25],
                palette = cmap, legend = True)

plt.title('k-means Clustering with 2 Dimensions');
```

This code will create a scatter plot of the features while also showing the clusters as different colors.

Note: Only every 25 users will be plotted so that it's easier to read.

c) Run the code cell.

d) Examine the output.



Scatterplots like this can help you visualize how clusters appear within the feature space. For example:

- Cluster `7`, represented by grey dots, appears to include negative values of the `PCA2` feature, but a wide range of `PCA1` values.
- Cluster `0`, represented by dark blue dots, seems to be the only cluster at the highest end of `PCA1` values (from 2.5 to 3), though it also appears at lower (but still positive) values.
- Cluster `6`, represented by pink dots, appears to have all positive values for `PCA2`, and also has the highest `PCA2` values.

6. Save and close the lab.

a) From the menu, select **File→Save and Checkpoint**.

b) Close the lab browser tab and continue on with the course.

LAB 4-15
Evaluating Clustering Models

Data File

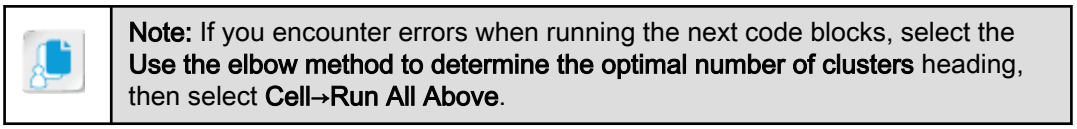
~/Clustering/Developing Clustering Models.ipynb

Scenario

As with supervised learning, you'll need some way to evaluate your clustering models. This is especially important for choosing the optimal number of clusters, as any change to this number will have a significant impact on how GCNB's customers are segmented.

---

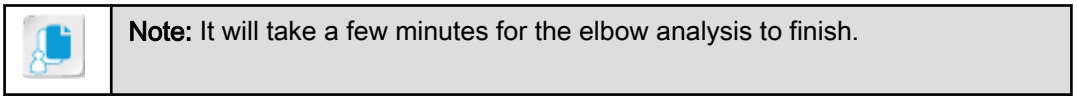
1. Open the lab and return to where you were in the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Open **Clustering/Developing Clustering Models.ipynb**.
   c) Your progress from the previous lab should have been saved.



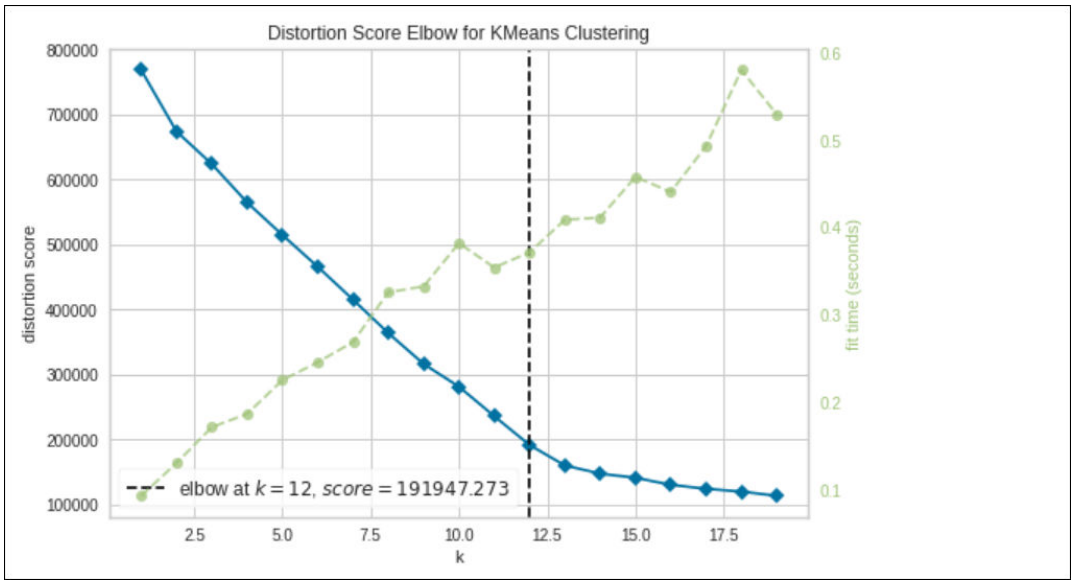
2. Use the elbow method to determine the optimal number of clusters.
   a) Scroll down and view the cell titled **Use the elbow method to determine the optimal number of clusters**, then select the code cell below it.
   b) In the code cell, type the following:

```
elbow = KElbowVisualizer(kmeans, k = (1, 20))
elbow.fit(users_data_scaled)
elbow.poof();
```

   This code uses the Yellowbrick library to automatically generate an elbow point graph of the (non-PCA) k-means model you trained earlier. In this case, the model will be trained on a range of clusters from 1 to 20.
   c) Run the code cell.



d) Examine the output.



The elbow point indicates the point of diminishing returns at around a k of 12. This does not objectively mean that 12 is the optimal number of clusters, only that a model with 12 clusters is worth exploring.

3. Use silhouette analysis to determine the optimal number of clusters.

a) Scroll down and view the cell titled **Use silhouette analysis to determine the optimal number of clusters**, then select the code cell below it.

b) In the code cell, type the following:

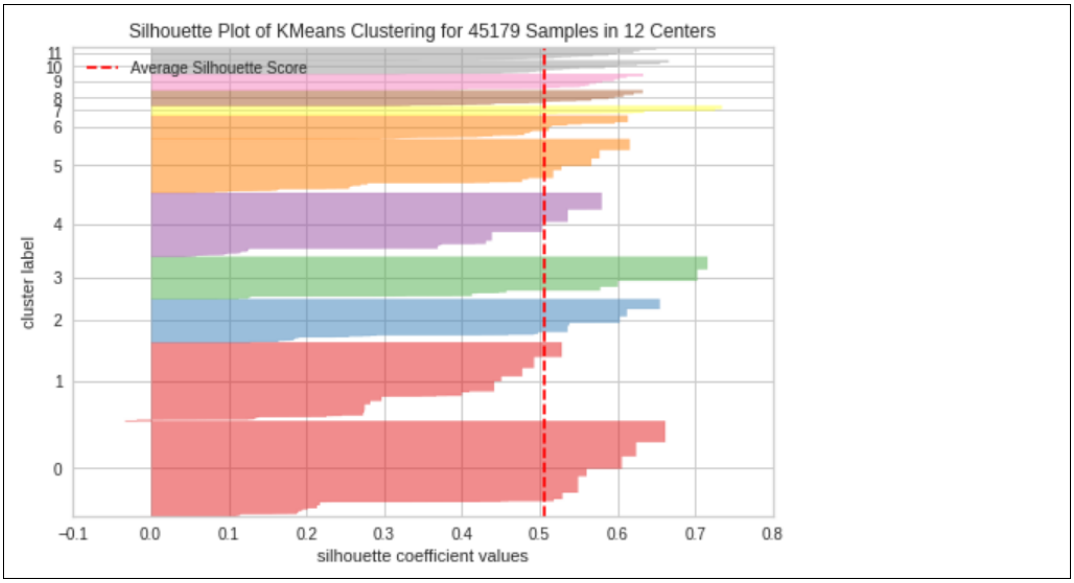
```
silhouette = SilhouetteVisualizer(KMeans(12, random_state = 10))
silhouette.fit(users_data_scaled)
silhouette.poof();
```

An alternative to the elbow point method is the silhouette score, which this code will visualize. Based on the elbow point's conclusions, you'll generate the silhouettes with a k of 12 first.

c) Run the code cell.

Note: It will take a few minutes for the silhouette analysis to finish.

d) Examine the output.



Each of the 12 clusters is plotted as its own silhouette, and the average silhouette score for the clusters is drawn as a vertical red line.

Note: You can see that at least one silhouette score for a data example in cluster `1` is negative (i.e., its silhouette line is plotted to the left of 0.0). The data example is therefore closer to a neighboring cluster than its own, and was probably placed in the wrong cluster.

e) Select the next code cell, then type the following:

```
print('Number of clusters: ', silhouette.n_clusters_)
print('Silhouette score:   ', silhouette.silhouette_score_)
```

f) Run the code cell.

g) Examine the output.

```
Number of clusters:  12
Silhouette score:    0.5051362488084534
```

The silhouette score for 12 clusters is ~0.51. Scores closer to 1 indicate a greater deal of cluster compactness and separation, which are the two main goals of effective clustering. However, a silhouette score for one number of clusters is most useful when it's compared to a different number of clusters.

4. Try a different number of clusters and compare the silhouette scores.

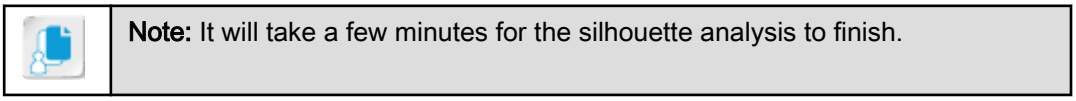
a) Scroll down and view the cell titled **Try a different number of clusters and compare the silhouette scores**, then select the code cell below it.

b) In the code cell, type the following:

```
silhouette = SilhouetteVisualizer(KMeans(15, random_state = 10))
silhouette.fit(users_data_scaled)
silhouette.poof();
```

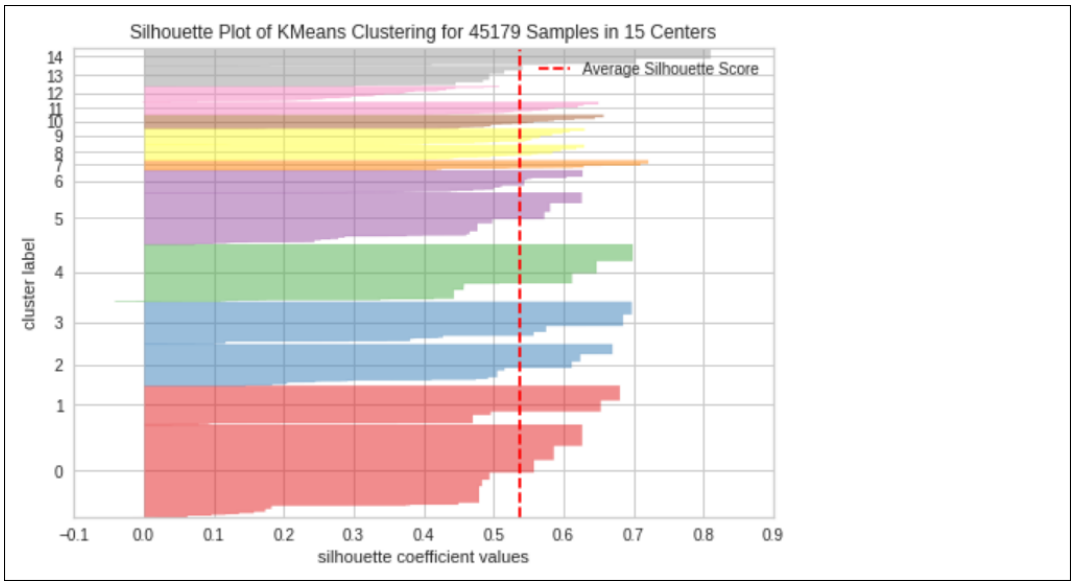
This time, the silhouettes will be calculated for 15 clusters using k-means.

c) Run the code cell.



Note: It will take a few minutes for the silhouette analysis to finish.

d) Examine the output.



The silhouette score appears to have increased somewhat, but you'll retrieve the exact score to make sure.

e) Select the next code cell, then type the following:

```
print('Number of clusters: ', silhouette.n_clusters_)
print('Silhouette score:   ', silhouette.silhouette_score_)
```

f) Run the code cell.

g) Examine the output.

```
Number of clusters:  15
Silhouette score:    0.5370783078641602
```

As expected, the silhouette score did increase somewhat. This suggests that 15 clusters may be better than 12. If you had time, you'd probably want to generate silhouette scores for many more values of k, as there could be some that score even better.

However, the score didn't improve that drastically from the model with 12 clusters. Also, you need to consider how the number of clusters will influence the way in which you apply those customers to the real world. There might be an upper limit to the number of clusters that's imposed by the problem domain. For example, will it be feasible for the marketing department to create and track 15 different market segments? For now, you'll just consider 12 to be the "optimal" number of clusters since it also aligns with what the elbow point concluded.

5. Retrain the k-means model on the optimal number of clusters.

a) Scroll down and view the cell titled **Retrain the k-means model on the optimal number of clusters**, then select the code cell below it.

b) In the code cell, type the following:

```
n_clusters = 12

kmeans = KMeans(n_clusters = n_clusters, random_state = 10)

kmeans.fit(users_data_scaled)
```

The model is being retrained to generate 12 clusters.

c) Run the code cell.

6. Generate the clusters and the number of users in each cluster.

a) Scroll down and view the cell titled **Generate the clusters and the number of users in each cluster**, then select the code cell below it.

b) In the code cell, type the following:

```
y_kmeans = kmeans.predict(users_data_scaled)

results['cluster'] = y_kmeans
results.head()
```

c) Run the code cell.

d) Examine the output.

	cluster	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services	job_self-employed	job_unemployed	job_housemaid	j
0	0	1	0	0	0	0	0	0	0	0	0	
1	1	0	1	0	0	0	0	0	0	0	0	
2	8	0	0	1	0	0	0	0	0	0	0	
3	4	0	0	0	1	0	0	0	0	0	0	
4	4	0	0	0	0	0	0	0	0	0	0	

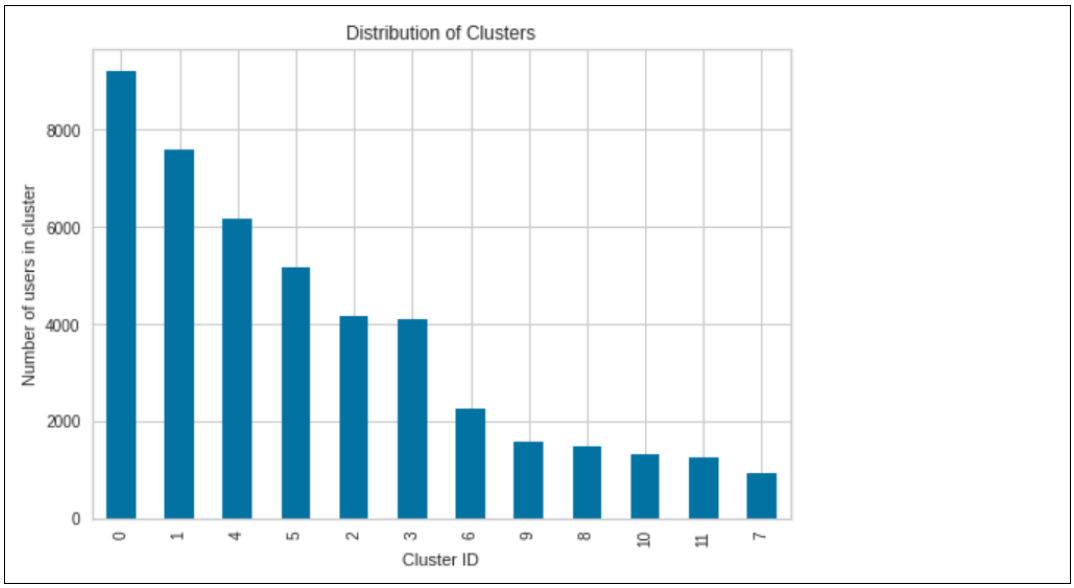
The clusters for the first five customers are the same as they were when you trained the k-means model on 10 clusters.

e) Select the next code cell, then type the following:

```
cluster_bar(y_kmeans)
```

f) Run the code cell.

g) Examine the output.



Cluster `0` still has the most users, and cluster `7` still has the fewest. Once again the decrease in customer count per cluster seems to be gradual.

7. Compare two different clusters.

a) Scroll down and view the cell titled **Compare two different clusters**, then select the code cell below it.

b) In the code cell, type the following:

```
cluster_6 = results[results.cluster == 6]
cluster_6.describe()
```

If you're going to apply these clusters to a practical situation, you need to learn more about them by understanding how the clusters differ. This code will show summary statistics for cluster `6`.

c) Run the code cell.

d) Examine the output.

e-ar	job_retired	job_admin.	job_services	job_self-employed	job_unemployed	job_housemaid	job_student	education_tertiary	education_secondary	education_Unknown
.0	2262.0	2262.0	2262.0	2262.0	2262.0	2262.0	2262.0	2262.000000	2262.000000	2262.000000
.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.161362	0.434571	0.052608
.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.367946	0.495810	0.223300
.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000
.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000
.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000
.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000	1.000000	0.000000
.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.000000	1.000000	1.000000

Looking at these statistics, here are a few interesting conclusions you could draw:

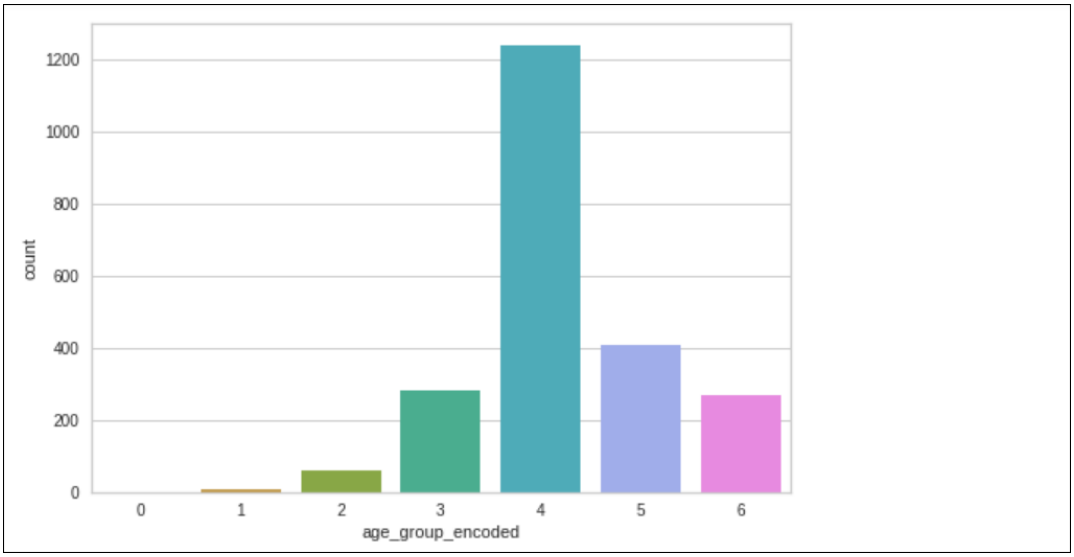
- Every member of this cluster appears to be retired. The `job_retired` feature, like most of the other features, is a one-hot encoded binary value. Its minimum is 1.0, which means that every customer in this cluster has a 1 ("yes") for this value.
- Members of this cluster are most likely to have a secondary level of education. The `education_secondary` feature has the highest mean value of the education features by far.
- Members of this cluster are very likely to have been married at some point (or still are). The mean `single` value is very close to 0; since this is a binary feature, a value of 0 means that the user is *not* single.
- Members of this cluster appear to be relatively older. The `age_group_encoded` feature shows a mean value of ~4.22 out of a total of 6. Recall that bin `4` is the age group 55–64. You'll explore these age values further using a chart.

e) Select the next code cell, then type the following:

```
1 sns.countplot(cluster_6['age_group_encoded']);
```

f) Run the code cell.

g) Examine the output.



Age bin `4` (55–64) has the most representation in this cluster by far, with bins `3` (45–54), `5` (65–74), and `6` (75+) each including about a third of that. Bins `0` (18–24), `1` (25–34), and `2` (35–44) have very little representation.

h) Select the next code cell, then type the following:

```
cluster_7 = results[results.cluster == 7]
cluster_7.describe()
```

This code will print the same summary statistics, but for cluster `7`.

i) Run the code cell.

j) Examine the output.

ob_unemployed	job_housemaid	job_student	education_tertiary	education_secondary	education_Unknown	education_primary	single	age_group_encoded
937.0	937.0	937.0	937.000000	937.000000	937.000000	937.000000	937.000000	937.000000
0.0	0.0	1.0	0.237994	0.542156	0.172892	0.046958	0.935966	0.721451
0.0	0.0	0.0	0.426083	0.498486	0.378356	0.211663	0.244945	0.592239
0.0	0.0	1.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
0.0	0.0	1.0	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000
0.0	0.0	1.0	0.000000	1.000000	0.000000	0.000000	1.000000	1.000000
0.0	0.0	1.0	0.000000	1.000000	0.000000	0.000000	1.000000	1.000000
0.0	0.0	1.0	1.000000	1.000000	1.000000	1.000000	1.000000	3.000000

Some conclusions you can draw from this include:

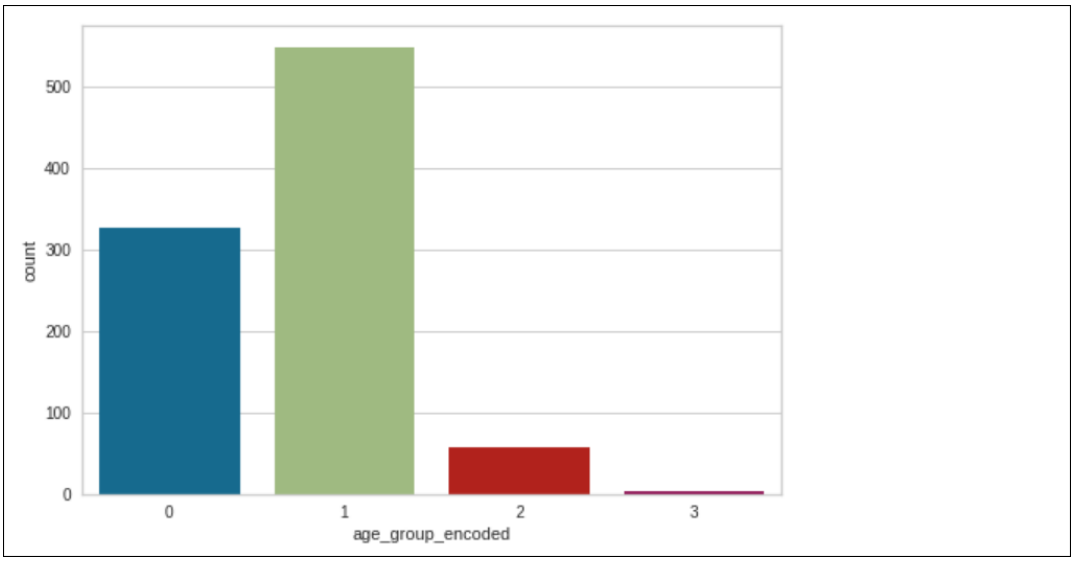
- Every member of this cluster is a student, for the same reason that every customer in cluster `6` is retired.
- Secondary education also seems to be the most common level of education among customers in this cluster.
- Most of the users in this cluster are single. The mean `single` value is very close to 1, meaning a "yes" for single.
- The average age of customers in this cluster is much lower than cluster `6`.

k) Select the next code cell, then type the following:

```
sns.countplot(cluster_7['age_group_encoded']);
```

l) Run the code cell.

m) Examine the output.



Bin `1` has the most customers in this cluster, whereas bin `0` has a decent amount of representation. Bins `2` and `3` have little representation, and no one 55 or above is included in this cluster at all.

So, in comparing these two clusters, you might conclude that cluster `6` represents a segment of the market that can be aimed at older retirees with a secondary education who have been or still are married, whereas cluster `7` represents a segment that can be aimed at young students with a secondary education who have never been married.

8. Save the best model.

a) Scroll down and view the cell titled **Save the best model**, then select the code cell below it.

b) In the code cell, type the following:

```
pickle.dump(kmeans, open('kmeans.pickle', 'wb'))
```

You'll save the k-means model with 12 clusters as the optimal model.

c) Run the code cell.

9. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on with the course.

MODULE 5

Project

The following labs are for the Course 4 Project.

PROJECT 4-1

Online Retailer: Developing Classification Models

Data File

~/Projects/Classification Project.ipynb

~/Projects/data/customer_data.pickle

Scenario

You work for an online retailer that sells a wide variety of different items to consumers. Each sale through the online storefront is recorded in a database. You and your team have spent time cleaning and preprocessing the data to prepare it for machine learning. The version of the dataset ready for machine learning is a table where each row is a customer, and each column is as follows:

- The **frequency**, or the number of times the customer visited the site.
- The **recency**, or how many days it's been since the customer last visited the site.
- The **tenure**, or how many days it's been since the customer first visited the site.
- The **monetary_value**, or how much the customer has spent on purchases.
- The **number_unique_items**, or the number of unique items the customer has purchased.
- The **churned** status, where **True** means the customer didn't return to purchase more items, and **False** means they did.

You've been tasked with building machine learning models that can predict whether or not a new customer will "churn," or fail to do repeat business with the company.

	**Note:** There is no "correct" answer for every line of code, as there are many ways to write code to do the same thing. You can write the code however you feel most comfortable, as long as it accomplishes the tasks set out for you.
	**Note:** If you're stuck and need help, you can "borrow" code from existing notebooks, and/or look up documentation on how to use the various Python libraries.
	**Note:** Sometimes, when you run code containing logic errors or bugs, you may corrupt the data contained in variables you created in previous code cells. To clear out such problems, you can select **Kernel→Restart & Clear Output**, then run each code cell again.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Projects/Classification Project.ipynb** to open it.
   c) Observe the notebook.

      Placeholder code cells have been provided in which you can add your own code. Comments provide hints on tasks you might perform in each code cell. The first code cell has already been completely programmed for you.

2. Import software libraries.
   a) Select the code listing under **Import software libraries**.
   b) Run the code and examine the results.

3. Read and examine the data.
   a) In the first code cell under **Read and examine the data**, write code to read the **customer_data.pickle** file in the **data** folder, then get the first five rows.
   b) Run the code cell and verify the first five rows of the dataset appear.
   c) In the next code cell, write code to get the structure of the data, including its data types and value counts for each column.
   d) Run the code cell and verify that there is no missing data and that each column is in the proper format.

4. Prepare the data.
   a) In the first code cell under **Prepare the data**, write code to define `churned` as the target variable, then get the value counts.
   b) Run the code cell and verify that you see the count of both `True` and `False` values, noting any imbalance.
   c) In the next code cell, write code to split the data into separate sets for the target and the rest of the features.
   d) Run the code cell.
   e) In the next code cell, write code to split the dataset into train and test sets for both the features and the target, using a `test_size` of `0.3`. Also print the shape of the data after this split.
   f) Run the code cell and verify the shape of each split.
   g) In the next code cell, write code to obtain the count of each value in the target variable using the test set.
   h) Run the code cell and verify the `True` and `False` counts.

5. Train a logistic regression model.
   a) In the first code cell under **Train a logistic regression model**, write code to normalize the training data.

   Note: The `MinMaxScaler()` class can do this for you.

   b) Run the code cell.
   c) In the next code cell, write code to fit a logistic regression model on the scaled training data.
   d) Run the code cell.
   e) In the next code cell, write code to make predictions on the test data, then print the count of each predicted value.
   f) Run the code cell and verify that the model predicted a certain number of `True` and `False` values.

6. Perform a quick evaluation of the logistic regression model.
   a) In the code cell under **Perform a quick evaluation of the logistic regression model**, write code to obtain the accuracy of the model's predictions.
   b) Run the code cell and verify the accuracy score.

7. Train a random forest model.
   a) In the first code cell under **Train a random forest model**, write code to fit a random forest model on the non-scaled training data.
   b) Run the code cell.
   c) In the next code cell, write code to make predictions on the test data, then print the count of each predicted value.
   d) Run the code cell and verify that the model predicted a certain number of `True` and `False` values.

8. Perform a quick evaluation of the random forest model.
   a) In the code cell under **Perform a quick evaluation of the random forest model**, write code to obtain the accuracy of the model's predictions.

b) Run the code cell and verify the accuracy score, and note how it compares to the logistic regression model's accuracy score.

9. Compare evaluation metrics for each model.

a) In the first code cell under **Compare evaluation metrics for each model**, observe the code that's been provided for you.

This code generates a list of model objects, including a dummy classifier and a new gradient boosting (XGBoost) model.

b) Run the code cell.

c) In the next code cell, observe the code that's been provided for you.

This code will generate dictionaries that contain various metric scores for each of the models in the list.

d) Run the code cell.

e) In the next code cell, write code to convert the list of dictionaries to a `DataFrame`, then print the `DataFrame` sorted by accuracy (descending).

f) Run the code cell and verify the different scores that each model received, including which model had the highest accuracy.

Note: For this scenario, you'll consider accuracy as the ideal metric. However, in the real world, you could also take any of the others into account.

10. Begin evaluating the best model.

a) In the first code cell under **Begin evaluating the best model**, write code to fit your training data using the "best" algorithm—i.e., the algorithm that produced the highest accuracy from the metrics table you just created.

Note: You can copy the relevant algorithm class object from the append statements in an earlier code block, including any parameters that were passed to the algorithm.

b) Run the code cell.

c) In the next code cell, write code to make predictions on the test data, then print the count of each predicted value.

d) Run the code cell and verify that the model predicted a certain number of `True` and `False` values.

e) In the next code cell, write code to plot a receiver operating characteristic (ROC) curve and its associated area under curve (AUC) value.

f) Run the code cell and verify the model's ROC curve shape and its AUC.

11. Generate a confusion matrix of the best model.

a) In the first code cell under **Generate a confusion matrix of the best model**, write code to generate a confusion matrix of the best model's predictions.

b) Run the code cell and verify you obtained the confusion matrix.

c) In the next code cell, write code to plot the confusion matrix in a visual form, using a color map to show the magnitude of each truth value quadrant.

Note: You can also plot the proportion of values using `normalize = 'true'` if you prefer that to absolute counts.

d) Run the code cell and verify that you have a visual of the confusion matrix.

12. Generate a feature importance plot for the best model.

a) In the first code cell under **Generate a feature importance plot for the best model**, observe the code that's been provided for you.

This function generates a feature importance plot on a bar chart. It takes the model, training features, and number of features to plot as arguments, respectively.

b) Run the code cell.

c) In the next code cell, write code to call the function using your best model, training features, and five features as arguments.
d) Run the code cell and verify that the top five features are displayed in bar chart form.

13. Plot a learning curve for the best model.

a) In the first code cell under **Plot a learning curve for the best model**, observe the code that's been provided for you.

   This function generates and plots a learning curve. It takes the model, training features, and training target as arguments, respectively.
b) Run the code cell.
c) In the next code cell, write code to call the function using your best model, training features, and training target as arguments.
d) Run the code and verify the learning curves that are displayed, noting any potential signs of overfitting.

14. Save the best model.

a) In the code cell under **Save the best model**, write code to save your best model as a pickle file named **best_classification_model.pickle**.
b) Run the code cell.

15. Save and download the notebook.

a) From the menu, select **File→Save and Checkpoint**.
b) Select **File→Download as→Notebook (.ipynb)**.
c) Save the file to your local drive if it wasn't automatically.
d) Find the saved file and rename it using the following convention: **FirstnameLastname-Classification-Project.ipynb**.

   For example: **JohnSmith-Classification-Project.ipynb**.

Note: Make sure not to lose this file. This is the project you'll be uploading and sharing to your peers for grading.

16. Shut down the notebook and close the lab.

a) Back in Jupyter Notebook, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on to the peer review.

PROJECT 4-2
Online Retailer: Developing Regression Models

Data File

~/Projects/Regression Project.ipynb

~/Projects/data/customer_data.pickle

Scenario

You work for an online retailer that sells a wide variety of different items to consumers. Each sale through the online storefront is recorded in a database. You and your team have spent time cleaning and preprocessing the data to prepare it for machine learning. The version of the dataset ready for machine learning is a table where each row is a customer, and each column is as follows:

- The **frequency**, or the number of times the customer visited the site.
- The **recency**, or how many days it's been since the customer last visited the site.
- The **tenure**, or how many days it's been since the customer first visited the site.
- The **monetary_value**, or how much the customer has spent on purchases.
- The **number_unique_items**, or the number of unique items the customer has purchased.
- The **churned** status, where **True** means the customer didn't return to purchase more items, and **False** means they did.

You've been tasked with building machine learning models that can predict how much a new customer will spend on purchases.

Note: There is no "correct" answer for every line of code, as there are many ways to write code to do the same thing. You can write the code however you feel most comfortable, as long as it accomplishes the tasks set out for you.

Note: If you're stuck and need help, you can "borrow" code from existing notebooks, and/or look up documentation on how to use the various Python libraries.

Note: Sometimes, when you run code containing logic errors or bugs, you may corrupt the data contained in variables you created in previous code cells. To clear out such problems, you can select **Kernel→Restart & Clear Output**, then run each code cell again.

---

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Projects/Regression Project.ipynb** to open it.
   c) Observe the notebook.

      Placeholder code cells have been provided in which you can add your own code. Comments provide hints on tasks you might perform in each code cell. The first code cell has already been completely programmed for you.

2. Import software libraries.
   a) Select the code listing under **Import software libraries**.
   b) Run the code and examine the results.

3. Read and examine the data.

a) In the first code cell under **Read and examine the data**, write code to read the **customer_data.pickle** file in the **data** folder, then get the first five rows.
b) Run the code cell and verify the first five rows of the dataset appear.
c) In the next code cell, write code to get the structure of the data, including its data types and value counts for each column.
d) Run the code cell and verify that there is no missing data and that each column is in the proper format.

4. Prepare the data.

a) In the first code cell under **Prepare the data**, write code to define `monetary_value` as the target variable, then get the value counts.

Note: You can also get the proportion of each value using `normalize = 'true'` if you prefer that to absolute counts.

b) Run the code cell and verify that you see the count of each monetary value.
c) In the next code cell, write code to plot a histogram of the target variable's distribution.
d) Run the code cell and verify that the shape of the distribution.
e) In the next code cell, write code to split the data into separate sets for the target and the rest of the features.
f) Run the code cell.
g) In the next code cell, write code to split the dataset into train and test sets for both the features and the target, using a `test_size` of `0.3`. Also print the shape of the data after this split.
h) Run the code cell and verify the shape of each split.
i) In the next code cell, write code to get summary statistics for the target variable in the test set.
j) Run the code cell and verify the target variable's mean, standard deviation, minimum, maximum, etc.

5. Train a logistic regression model.

a) In the first code cell under **Train a linear regression model**, write code to standardize both the training data and the test data.

Note: The `StandardScaler()` class can do this for you.

b) Run the code cell.
c) In the next code cell, write code to fit a linear regression model on the scaled training data.
d) Run the code cell.
e) In the next code cell, write code to make predictions on the test data, then print the first five predicted values.
f) Run the code cell and verify the model's monetary predictions for the first five customers.

6. Perform a quick evaluation of the logistic regression model.

a) In the code cell under **Perform a quick evaluation of the linear regression model**, write code to obtain the R^2 score (coefficient of determination) of the model's predictions.
b) Run the code cell and verify the R^2 score.

7. Train a random forest model.

a) In the first code cell under **Train a random forest model**, write code to fit a random forest model on the non-scaled training data.
b) Run the code cell.
c) In the next code cell, write code to make predictions on the test data, then print the first five predicted values.
d) Run the code cell and verify the model's monetary predictions for the first five customers.

8. Perform a quick evaluation of the random forest model.

a) In the code cell under **Perform a quick evaluation of the random forest model**, write code to obtain the R^2 score of the model's predictions.
b) Run the code cell and verify the R^2 score, and note how it compares to the linear regression model's R^2 score.

9. Compare evaluation metrics for each model.

a) In the first code cell under **Compare evaluation metrics for each model**, observe the code that's been provided for you.

   This code generates a list of model objects, including a dummy classifier and a new gradient boosting (XGBoost) model.
b) Run the code cell.
c) In the next code cell, observe the code that's been provided for you.

   This code will generate dictionaries that contain various metric scores for each of the models in the list.
d) Run the code cell.
e) In the next code cell, write code to convert the list of dictionaries to a `DataFrame`, then print the `DataFrame` sorted by mean squared error (ascending).
f) Run the code cell and verify the different scores that each model received, including which model had the lowest (best) MSE.

Note: For this scenario, you'll consider MSE as the ideal metric. However, in the real world, you could also take any of the others into account.

10. Begin evaluating the best model.

a) In the first code cell under **Begin evaluating the best model**, write code to fit your training data using the "best" algorithm—i.e., the algorithm that produced the lowest MSE from the metrics table you just created.

Note: You can copy the relevant algorithm class object from the append statements in an earlier code block, including any parameters that were passed to the algorithm.

b) Run the code cell.
c) In the next code cell, write code to make predictions on the test data, then print the count of each predicted value.
d) Run the code cell and verify that the model predicted monetary values for the first five customers.
e) In the next code cell, write code to plot the residuals from the predicted values.

Note: Recall that residuals are the differences between the predicted values and the actual values in the test set. A residual plot is a scatter plot that compares residuals on one axis to the predicted values on the other axis.

f) Run the code cell and verify how the residuals change as the predicted values increase and decrease.

11. Generate a feature importance plot for the best model.

a) In the first code cell under **Generate a feature importance plot for the best model**, observe the code that's been provided for you.

   This function generates a feature importance plot on a bar chart. It takes the model, training features, and number of features to plot as arguments, respectively.
b) Run the code cell.
c) In the next code cell, write code to call the function using your best model, training features, and five features as arguments.
d) Run the code cell and verify that the top five features are displayed in bar chart form.

12. Plot a learning curve for the best model.

a) In the first code cell under **Plot a learning curve for the best model**, observe the code that's been provided for you.

   This function generates and plots a learning curve. It takes the model, training features, and training target as arguments, respectively.

b) Run the code cell.

c) In the next code cell, write code to call the function using your best model, training features, and training target as arguments.

d) Run the code and verify the learning curves that are displayed, noting any potential signs of overfitting.

13. Save the best model.

a) In the code cell under **Save the best model**, write code to save your best model as a pickle file named **best_regression_model.pickle**.

b) Run the code cell.

14. Save and download the notebook.

a) From the menu, select **File→Save and Checkpoint**.

b) Select **File→Download as→Notebook (.ipynb)**.

c) Save the file to your local drive if it wasn't automatically.

d) Find the saved file and rename it using the following convention: **FirstnameLastname-Regression-Project.ipynb**.

   For example: **JohnSmith-Regression-Project.ipynb**.

Note: Make sure not to lose this file. This is the project you'll be uploading and sharing to your peers for grading.

15. Shut down the notebook and close the lab.

a) Back in Jupyter Notebook, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on to the peer review.

PROJECT 4-3
Online Retailer: Developing Clustering Models

Data File

~/Projects/Clustering Project.ipynb

~/Projects/data/customer_data.pickle

Scenario

You work for an online retailer that sells a wide variety of different items to consumers. Each sale through the online storefront is recorded in a database. You and your team have spent time cleaning and preprocessing the data to prepare it for machine learning. The version of the dataset ready for machine learning is a table where each row is a customer, and each column is as follows:

- The **frequency**, or the number of times the customer visited the site.
- The **recency**, or how many days it's been since the customer last visited the site.
- The **tenure**, or how many days it's been since the customer first visited the site.
- The **monetary_value**, or how much the customer has spent on purchases.
- The **number_unique_items**, or the number of unique items the customer has purchased.
- The **churned** status, where **True** means the customer didn't return to purchase more items, and **False** means they did.

You've been tasked with building machine learning models that can place customers into overall groups to facilitate more effective and targeted marketing.

	**Note:** There is no "correct" answer for every line of code, as there are many ways to write code to do the same thing. You can write the code however you feel most comfortable, as long as it accomplishes the tasks set out for you.
	**Note:** If you're stuck and need help, you can "borrow" code from existing notebooks, and/or look up documentation on how to use the various Python libraries.
	**Note:** Sometimes, when you run code containing logic errors or bugs, you may corrupt the data contained in variables you created in previous code cells. To clear out such problems, you can select **Kernel→Restart & Clear Output**, then run each code cell again.

1. Open the lab and notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Projects/Clustering Project.ipynb** to open it.
   c) Observe the notebook.

      Placeholder code cells have been provided in which you can add your own code. Comments provide hints on tasks you might perform in each code cell. The first code cell has already been completely programmed for you.

2. Import software libraries.
   a) Select the code listing under **Import software libraries**.
   b) Run the code and examine the results.

3. Read and examine the data.

a) In the first code cell under **Read and examine the data**, write code to read the **customer_data.pickle** file in the **data** folder, then get the first five rows.
b) Run the code cell and verify the first five rows of the dataset appear.
c) In the next code cell, write code to get summary statistics for the dataset.
d) Run the code cell and verify each feature's mean, standard deviation, minimum, maximum, etc.
e) In the next code cell, write code to get the structure of the data, including its data types and value counts for each column.
f) Run the code cell and verify that there is no missing data and that each column is in the proper format.
g) In the next code cell, write code to get the shape of the data.
h) Run the code and verify the number of rows and columns in the data.

4. Train a k-means clustering model.

a) In the first code cell under **Train a *k*-means clustering model**, write code to standardize the data.

Note: The `StandardScaler()` class can do this for you.

b) In the next code cell, observe the code that's been provided for you.
This function will train k-means clustering models for each value of k from 1 to 30. It will then plot an elbow point.
c) Run the code cell.
d) In the next code cell, call the function to cluster the scaled data and generate the elbow point.
e) Run the code cell and verify that you can see the elbow point graph, and note where the elbow point appears to be located.

5. Perform silhouette analysis on the clustering model.

a) In the first code cell under **Perform silhouette analysis on the clustering model**, write code to generate a silhouette plot using `10` as the k value.

Note: The `SilhouetteVisualizer()` class from the Yellowbrick library can easily generate silhouette plots.

b) Run the code cell and verify the silhouette plot for each cluster, as well as the average silhouette score line.
c) In the next code cell, print the number of clusters and the average silhouette score.
d) Run the code cell and verify the silhouette score for a 10-cluster model.
e) In the next code cell, write code to generate a silhouette plot using `5` as the k value.
f) Run the code cell and verify the silhouette plot for each cluster, as well as the average silhouette score line.
g) In the next code cell, print the number of clusters and the average silhouette score.
h) Run the code cell and verify the silhouette score for a 5-cluster model, and note how it compares to the 10-cluster model.

6. Train an optimal clustering model.

a) In the first code cell under **Train an optimal clustering model**, write code to define an optimal number of clusters according to the best silhouette score, then fit a k-means clustering model on the scaled data using this optimal number of clusters.
b) Run the code cell.
c) In the next code cell, write code to determine (predict) the cluster each customer is in, then print a `DataFrame` that has a new column showing what cluster the customer is assigned to. Also print the first five rows of the `DataFrame`.
d) Run the code cell and verify that you see the first five rows of the data, along with cluster assignments for each row (customer).

7. Evaluate the clustering model as a whole.

a) In the first code cell under **Evaluate the clustering model as a whole**, write code to generate a bar chart showing how many users were assigned to each cluster.
b) Run the code cell and verify that the bar char demonstrates the population of each cluster.
c) In the next code cell, write code to get the actual count of customers in each cluster.
d) Run the code cell and verify the number of customers in each cluster.

8. Evaluate summary statistics for individual clusters.

a) In the first code cell under **Evaluate summary statistics for individual clusters**, write code to generate summary statistics for cluster 0.
b) Run the code cell and verify each feature's mean, standard deviation, minimum, maximum, etc. in this cluster.
c) In the next code cell, write code to print the average amount of money (`monetary_value`) spent by each customer in this cluster.
d) Run the code cell and verify the average amount spent.
e) In the next code cell, write code to generate summary statistics for cluster 3.
f) Run the code cell and verify each feature's mean, standard deviation, minimum, maximum, etc. in this cluster.
g) In the next code cell, write code to print the average amount of money (`monetary_value`) spent by each customer in this cluster.
h) Run the code cell and verify the average amount spent.
i) In the next code cell, write code to generate summary statistics for cluster 4.
j) Run the code cell and verify each feature's mean, standard deviation, minimum, maximum, etc. in this cluster.
k) In the next code cell, write code to print the average amount of money (`monetary_value`) spent by each customer in this cluster.
l) Run the code cell and verify the average amount spent, and compare this number to the other clusters' averages.

9. Evaluate the distribution of features for individual clusters.

a) In the first code cell under **Evaluate the distribution of features for individual clusters**, write code to generate violin plots for the distribution of the `recency` feature for each of the clusters.
b) Run the code cell and verify that you can see the `recency` distribution for each of the clusters.
c) In the next code cell, write code to generate violin plots for the distribution of the `frequency` feature for each of the clusters.
d) Run the code cell and verify that you can see the `frequency` distribution for each of the clusters.
e) In the next code cell, write code to generate violin plots for the distribution of the `monetary_value` feature for each of the clusters.
f) Run the code cell and verify that you can see the `monetary_value` distribution for each of the clusters, and compare these distributions to the distributions for the other features.

10. Perform PCA to visualize the clusters in two dimensions.

a) In the first code cell under **Perform PCA to visualize the clusters in two dimensions**, observe the code that's been provided for you.

   This function performs principal component analysis (PCA) on the data by taking the model, scaled data, and number of clusters as arguments, respectively. It also plots the reduced data on a scatter plot.
b) Run the code cell.
c) In the next code cell, call the function to perform PCA and generate the scatter plot using your model, the scaled data, and the optimal number of clusters.
d) Run the code cell and verify that you see a scatter plot showing two distinct clusters, and also note the clusters' separation and compactness, as well as their overall shapes.

11. Save the optimal model.

a) In the code cell under **Save the optimal model**, write code to save your optimal model as a pickle file named **optimal_clustering_model.pickle**.

b) Run the code cell.

12. Save and download the notebook.

a) From the menu, select **File→Save and Checkpoint**.
b) Select **File→Download as→Notebook (.ipynb)**.
c) Save the file to your local drive if it wasn't automatically.
d) Find the saved file and rename it using the following convention: **FirstnameLastname-Clustering-Project.ipynb**.

For example: **JohnSmith-Clustering-Project.ipynb**.

Note: Make sure not to lose this file. This is the project you'll be uploading and sharing to your peers for grading.

13. Shut down the notebook and close the lab.

a) Back in Jupyter Notebook, select **Kernel→Shutdown**.
b) In the **Shutdown kernel?** dialog box, select **Shutdown**.
c) Close the lab browser tab and continue on to the peer review.

Course 5: Finalize a Data Science Project

Course Introduction

The following lab is for Course 5: Finalize a Data Science Project.

Modules

The lab in this course pertains to the following module:

- Module 3: Implement and Test Production Pipelines

MODULE 3

Implement and Test Production Pipelines

The following lab is for Module 3: Implement and Test Production Pipelines.

LAB 5-1
Building an ML Pipeline

Data Files

~/Finalizing/Building an ML Pipeline.ipynb

~/Finalizing/data/users_data_final.pickle

~/Finalizing/data/new_users_data.csv

Scenario

The GCNB marketing project is coming to a close, but that doesn't mean your work is finished. It was good for you to go through the entire data science process, but your approach wasn't necessarily the most efficient. It would help you save time and effort if you were able to automate the process, making it repeatable for future projects. What you need is to set up a data pipeline. Eventually, the team wants to look into cloud-based options for setting up a pipeline that can automate the entire data science process—from data collection all the way to presentation.

For now, however, you want to test out scikit-learn's `Pipeline` module, which focuses on automating the machine learning model training process. The tasks you performed earlier to train, tune, and evaluate models can be condensed and streamlined using this pipeline functionality. This pipeline will be able to take any new set of training data and build a working model from it. However, because the pipeline only focuses on the model training process, it will assume that any new training data fed into it will already have been cleaned.

In this case, you'll implement the pipeline using the familiar GCNB customer dataset, with classification as the goal. After you create the pipeline, you'll test the results on a new set of unlabeled customer data so that you can predict whether or not these new customers will sign up for a term deposit.

Note: By default, these lab steps assume you will be typing the code shown in screenshots. Many learners find it easier to understand what code does when they are able to type it themselves. However, if you prefer not to type all of code, the **Solutions** folders contain the finished code for each notebook.

---

1. Open the notebook.
   a) From the Coursera lab page, select **Open Lab** to open a Jupyter Notebook session in another web browser tab.
   b) Select **Finalizing**.
      The **Finalizing** directory contains a subdirectory named **data** and a notebook file named **Building an ML Pipeline.ipynb**.
   c) Select **Building an ML Pipeline.ipynb** to open it.

2. Import the relevant software libraries.
   a) View the cell titled **Import software libraries**, and examine the code listing below it.
   b) Select the cell that contains the code listing, then select **Run**.
   c) Verify that the version of Python is displayed, as are the versions of the other libraries that were imported.

3. Load and preview the data.
   a) Scroll down and view the cell titled **Load and preview the data**, then select the code cell below it.

b) In the code cell, type the following:

```
users_data = pd.read_pickle('data/users_data_final.pickle')

users_data.head(n = 3)
```

c) Run the code cell.

d) Examine the output.

	user_id	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_ser
0	9231c446-cb16-4b2b-a7f7-ddfc8b25aaf6	3.0	2143.00	1	0	0	0	0	0	
1	bb92765a-08de-4963-b432-496524b39157	0.0	1369.42	0	1	0	0	0	0	
2	573de577-49ef-42b9-83da-d3cfb817b5c1	2.0	2.00	0	0	1	0	0	0	

3 rows × 33 columns

This is the same customer data you've been working with all throughout the CDSP Specialization, so it should look familiar.

4. Split the data into train and test sets.

a) Scroll down and view the cell titled **Split the data into train and test sets**, then select the code cell below it.

b) In the code cell, type the following:

```
target = users_data.term_deposit
features = users_data.drop(['user_id', 'term_deposit'], axis = 1)

X_train, X_test, y_train, y_test = train_test_split(features, target)
```

Some tasks, like splitting the target variable from the features, and performing the holdout method, still need to be done manually.

c) Run the code cell.

5. Define an initial pipeline.

a) Scroll down and view the cell titled **Define an initial pipeline**, then select the code cell below it.

b) In the code cell, type the following:

```
pipe = Pipeline([('scaler', MinMaxScaler()),
                 ('reduce_dim', PCA(n_components = 2)),
                 ('model', DecisionTreeClassifier())])
```

This code builds a pipeline using scikit-learn's `Pipeline` module. Each tuple represents a step in the pipeline process, where each step is executed sequentially. So, the `MinMaxScaler()` function will be applied to the data first, then the `PCA()` function, and finally, the last step is always the algorithm used to train the model (in this case, a decision tree model).

Note: Feature scaling isn't typically necessary with tree-based algorithms, but it's being done here to demonstrate a common component of a scikit-learn pipeline.

c) Run the code cell.

6. Evaluate the initial pipeline.

a) Scroll down and view the cell titled **Evaluate the initial pipeline**, then select the code cell below it.

b) In the code cell, type the following:

```
pipe = pipe.fit(X_train, y_train)
print('Model accuracy on test data:', pipe.score(X_test, y_test))
```

Whenever you call `fit()` on the `Pipeline` object, it will perform all of the transformation steps in the pipeline. The `score()` function will take the default score on the test data. In the case of classification, the default metric is accuracy.

c) Run the code cell.

d) Examine the output.

```
Model accuracy on test data: 0.8110668437361664
```

The decision tree model created by the pipeline has an accuracy of ~0.81.

e) Select the next code cell, then type the following:

```
y_pred = pipe.predict(X_test)
print(Counter(y_pred))
```

f) Run the code cell.

g) Examine the output.

```
Counter({False: 9935, True: 1360})
```

The model predicted 9,935 `False` values and 1,360 `True` values in the test set.

h) Select the next code cell, then type the following:

```
results = pd.concat([y_test.iloc[:5], X_test.iloc[:5]], axis = 1)
results.insert(1, 'term_deposit_pred', y_pred[:5])
results
```

i) Run the code cell.

j) Examine the output.

	term_deposit	term_deposit_pred	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	jc
43308	False	True	0.0	1369.42	1	0	0	0	0	
32770	False	False	2.0	246.00	0	0	0	1	0	
17440	False	False	0.0	1369.42	0	0	0	0	0	
36164	False	False	0.0	1369.42	0	0	0	1	0	
29218	False	False	0.0	1369.42	0	0	0	0	0	

5 rows × 33 columns

The model incorrectly predicted the first record, but correctly predicted the four after that. Like you've seen before, hyperparameter optimization can improve a model. You can do the same thing with pipelines.

7. Tune the pipeline.

a) Scroll down and view the cell titled **Tune the pipeline**, then select the code cell below it.

b) In the code cell, type the following:

```
scalers = [None, StandardScaler(), MinMaxScaler()]
pca_dims = [None, PCA(n_components = 2), PCA(n_components = 5)]
models = [DecisionTreeClassifier(),
          RandomForestClassifier(random_state = 1)]

params = {'scaler': scalers,
          'reduce_dim': pca_dims,
          'model': models}
```

You'll be performing a grid search, only this time, rather than searching for one model's optimal hyperparameters, you'll be searching for the optimal steps to perform in the pipeline. So, your parameter grid will instruct the search to try out two different scalers (or no scaler, as indicated by `None`), two different dimensions for PCA (or no PCA), and two different algorithms (decision tree and random forest).

Note: The seed value at the beginning of the notebook isn't enough to make these grid search results deterministic, so a random seed is being defined explicitly for `RandomForestClassifier()`, and `StratifiedKFold()` is configured to not shuffle the data before splitting it into folds.

c) Run the code cell.

d) Select the next code cell, then type the following:

```
gs = GridSearchCV(pipe, params, n_jobs = -1, verbose = 2,
                  cv = StratifiedKFold(5, shuffle = False)). \
fit(X_train, y_train)
```

Rather than passing in a model to `GridSearchCV()`, you're passing in the `pipe` object. Otherwise, the process is essentially the same.

Note: Remember that `n_jobs = -1` is telling scikit-learn to use all available CPU threads.

e) Run the code cell.

Note: It can take a few minutes for the search to complete.

8. Evaluate the tuned pipeline.

a) Scroll down and view the cell titled **Evaluate the tuned pipeline**, then select the code cell below it.

b) In the code cell, type the following:

```
print('Best accuracy score:', gs.score(X_test, y_test))
print('Best parameters:    ', gs.best_params_)
```

c) Run the code cell.

d) Examine the output.

```
Best accuracy score: 0.888092076139885
Best parameters:     {'model': RandomForestClassifier(random_state=1), 'reduce_dim': None, 'scaler': MinMaxScaler
()}
```

The best score for the tuned pipeline is ~0.89, an improvement over the initial pipeline. The grid search retrieved the following optimal steps:

- The features are standardized.
- No PCA is performed.
- The model is based on the random forest algorithm.

If you had a different set of training data, it might retrieve a different set of pipeline steps. Also, if you had more time, you'd probably want to expand the parameter grid to include more types of scaling and dimensionality reduction options, as well as more types of machine learning algorithms.

e) Select the next code cell, then type the following:

```
y_pred = gs.predict(X_test)
print(Counter(y_pred))
```

f) Run the code cell.

g) Examine the output.

```
Counter({False: 10599, True: 696})
```

The random forest model created from the pipeline predicted 10,599 `False` values and 696 `True` values.

h) Select the next code cell, then type the following:

```
results['term_deposit_pred'] = y_pred[:5]
results
```

i) Run the code cell.

j) Examine the output.

	term_deposit	term_deposit_pred	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	jc
43308	False	False	0.0	1369.42	1	0	0	0	0	
32770	False	False	2.0	246.00	0	0	0	1	0	
17440	False	False	0.0	1369.42	0	0	0	0	0	
36164	False	False	0.0	1369.42	0	0	0	1	0	
29218	False	False	0.0	1369.42	0	0	0	0	0	

5 rows × 33 columns

The model correctly predicted the first five records.

9. Test the model generated by the pipeline on new data.

a) Scroll down and view the cell titled **Test the model generated by the pipeline on new data**, then select the code cell below it.

b) In the code cell, type the following:

```
new_data = pd.read_csv('data/new_users_data.csv')

new_data
```

This is a new set of data that the model has not yet seen.

c) Run the code cell.

d) Examine the output.

	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_services	job_self-employed	...
0	4	237.10	0	0	0	0	1	0	0	0	...
1	2	-43.12	0	0	1	0	0	0	0	0	...
2	1	789.45	0	1	0	0	0	0	0	0	...
3	4	3291.41	0	1	0	0	0	0	0	0	...

4 rows × 31 columns

This new data has four total customer records. It's also not labeled, so it's closer to approximating the real-world application of your classification tasks: predicting whether or not a customer will sign up for a term deposit.

e) Select the next code cell, then type the following:

```
y_pred = gs.predict(new_data)

new_data.insert(0, 'term_deposit_pred', y_pred)
new_data
```

f) Run the code cell.

g) Examine the output.

	term_deposit_pred	number_transactions	total_amount_usd	job_management	job_technician	job_entrepreneur	job_blue-collar	job_retired	job_admin.	job_serv
0	False	4	237.10	0	0	0	0	1	0	
1	False	2	-43.12	0	0	1	0	0	0	
2	False	1	789.45	0	1	0	0	0	0	
3	False	4	3291.41	0	1	0	0	0	0	

4 rows × 32 columns

The model generated from the pipeline predicts that all four of these new customers won't sign up for a term deposit.

10. Shut down the notebook and close the lab.

a) From the menu, select **Kernel→Shutdown**.

b) In the **Shutdown kernel?** dialog box, select **Shutdown**.

c) Close the lab browser tab and continue on with the course.

Solutions

LAB 3-1: Examining Data

8. **At this point in the process, which of these variables do you think might make good candidates for target features?**

   A: Answers will vary, as there can be many potential target features, and the dataset may have new features added to it later. Still, numeric features like `total_amount_usd` and `number_transactions` could be good candidates for a prediction model, whereas categorical features like `default` (whether the user has defaulted on a loan) and `term_deposit` (whether or not the user signed up for a term deposit) could be good candidates for a classification model.