Introduction

When deciding health or life insurance packages, the most important factor for most is the pricing rate. The intention of the project is to gather information that compares health and life insurance rates, how demographics may have an effect in pricing, and we will form a machine learning model to try and predict life expectancy by a demographic group. The goal is to compare groups if specific demographics are seen to be charged a higher/lower rate for the predicted life expectancy value.

Data Sources

Life Insurance Data

https://www.bestliferates.org/statistics

This source includes various facts and stats about life insurance. The data we scraped for our project included total market penetration, ownership gap, why consumers purchase life insurance, and why consumers don't purchase life insurance.

https://www.investopedia.com/articles/personal-finance/022615/how-age-affects-life-insurance-rates.asp

This source is an article on how life insurance rates work as well as how life insurance rates rise with age. The data we scraped for our project was the "Life Insurance Rates by Age" table which showed the rates for each life insurance plan for different ages separated by gender.

https://www.iii.org/table-archive/22403

This source had tables that showed life insurance benefits and claims payouts for 5-year stretches. The data we scraped was the current table which was Life/Annuity Insurance Benefits And Claims, 2016-2020.

https://www.cdc.gov/nchs/pressroom/sosmap/life_expectancy/life_expectancy.htm

Displayed the life expectancy at birth for every state, downloaded the data as a csv to use in Machine learning model

Census Data

https://data.census.gov/cedsci/

Search for the following tables:

- S1901 (ACS 5-year estimates 2019&2018)
- S2701 (ACS 5-year estimates 2019&2018)

Used to provide general state by state demographics to be manipulated as independent variables in the predictive model.

Construct an API call using the site:

https://api.census.gov/data/2020/acs/acs5/subject/variables.html

S2701_C01_001E, S2701_C01_014E, S2701_C01_015E, S2701_C01_016E, S2701_C01_017E, S2701_C01_018E, S2701_C01_019E, S2701_C01_020E, S2701_C01_021E, S2701_C01_022E, S2701_C01_023E, S2701_C03_001E, S1901_C01_012E, NAME chosen by the state geography.

https://www.census.gov/data/tables/time-series/demo/health-insurance/acs-hi.html

HI-05_ACS excel file for appropriate years Used to provide percent of state population with health insurance

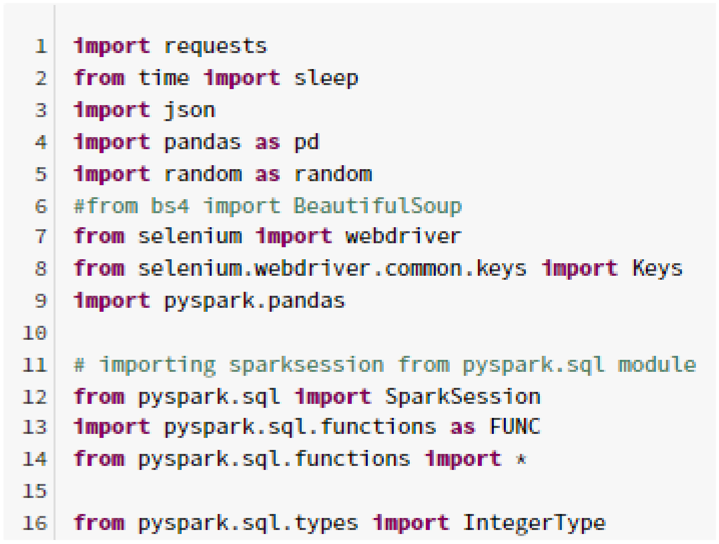
Abbreviation Table

https://worldpopulationreview.com/states/state-abbreviations

Used for the purposes of merging tables which represent states using two letter abbreviations and those using the full name.

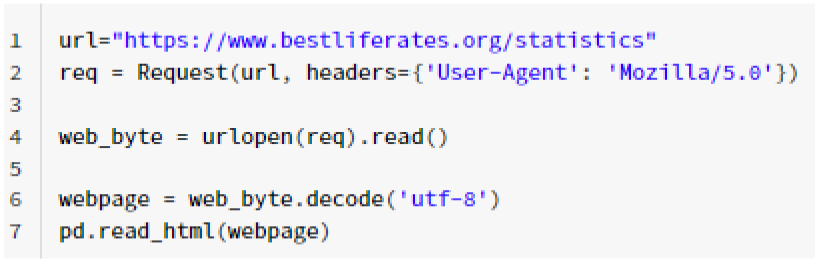
ETL in General

For the website data, these are the imports that we used



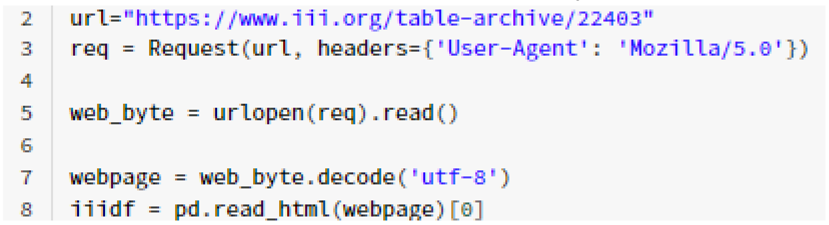
```python
import requests
from time import sleep
import json
import pandas as pd
import random as random
#from bs4 import BeautifulSoup
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import pyspark.pandas

# importing sparksession from pyspark.sql module
from pyspark.sql import SparkSession
import pyspark.sql.functions as FUNC
from pyspark.sql.functions import *

from pyspark.sql.types import IntegerType
```

All the used websites were accessed with the following python code in Azure databricks, where the only difference was the url to distinguish various sites.



```python
url="https://www.bestliferates.org/statistics"
req = Request(url, headers={'User-Agent': 'Mozilla/5.0'})

web_byte = urlopen(req).read()

webpage = web_byte.decode('utf-8')
pd.read_html(webpage)
```

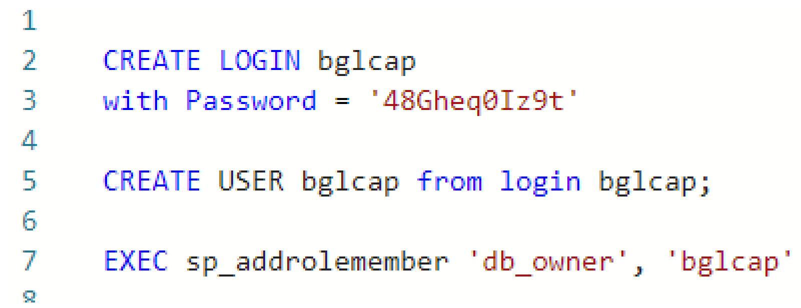
This gives a list of the tables on a given website, from which we can get a specific table by index and set it to a pandas data frame as seen below for the first website, the insurance information institute, where our desired table was the first one, Life/Annuity Insurance Benefits and Claims, 2016-2020.



```
url="https://www.iii.org/table-archive/22403"
req = Request(url, headers={'User-Agent': 'Mozilla/5.0'})

web_byte = urlopen(req).read()

webpage = web_byte.decode('utf-8')
iiidf = pd.read_html(webpage)[0]
```

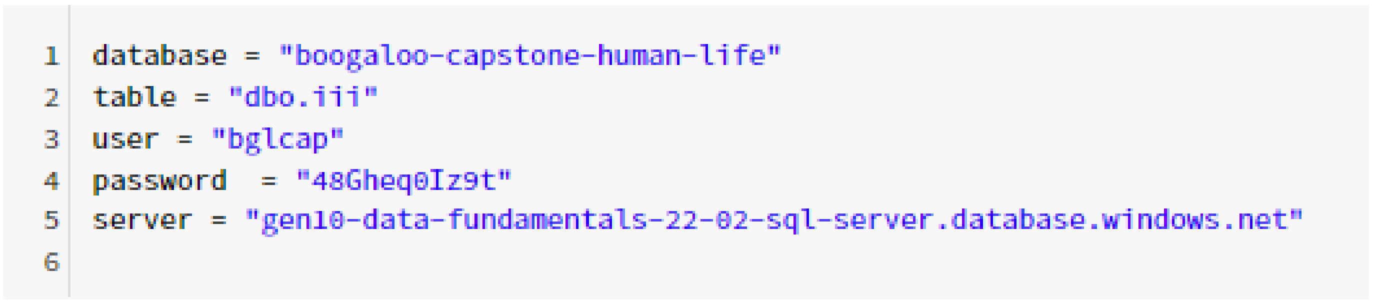
The transformations depended on the tables themselves, and while we tried to convert them to spark data frames where possible, some things just weren't doable using spark data frames rather than pandas. However, all tables were eventually converted into spark data frames for ease of writing to database

The azure sql database was accessed with the following code, and various tables were created depending on the data to allow for the data to be loaded into the database



```

CREATE LOGIN bglcap
with Password = '48Gheq0Iz9t'

CREATE USER bglcap from login bglcap;

EXEC sp_addrolemember 'db_owner', 'bglcap'
```

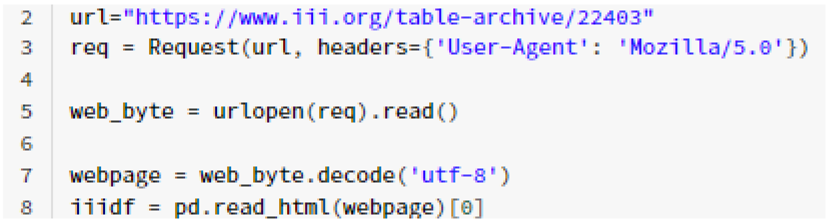
We connected to the database using this code based on the created login credentials, with only the table variable changing to connect to different tables



```
database = "boogaloo-capstone-human-life"
table = "dbo.iii"
user = "bglcap"
password  = "48Gheq0Iz9t"
server = "gen10-data-fundamentals-22-02-sql-server.database.windows.net"

```

iii data

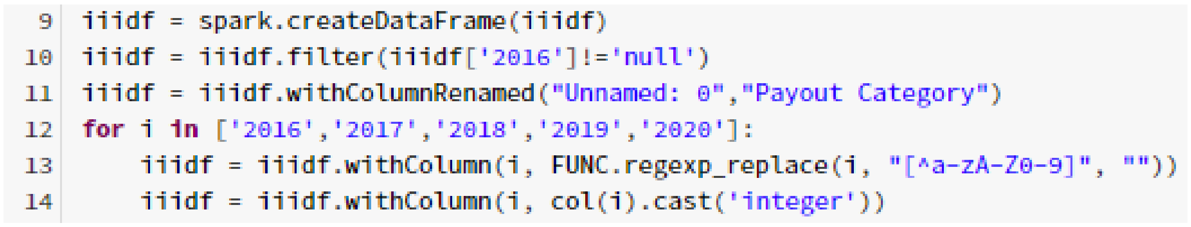
This code gets the Life/Annuity Insurance Benefits and Claims, 2016-2020 table from the insurance information institute website shown in the link, where it was the first table in the page



```python
url="https://www.iii.org/table-archive/22403"
req = Request(url, headers={'User-Agent': 'Mozilla/5.0'})

web_byte = urlopen(req).read()

webpage = web_byte.decode('utf-8')
iiidf = pd.read_html(webpage)[0]
```

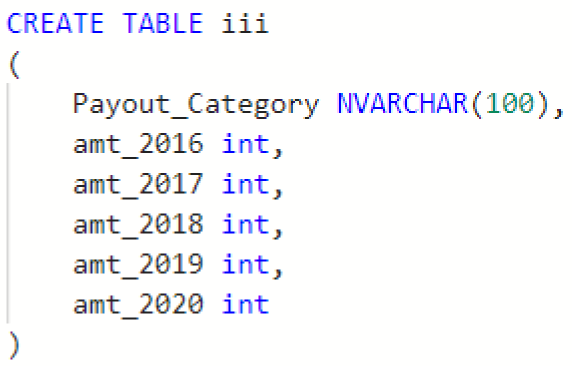
We initially convert the pandas data frame to a spark data frame as per specification suggestion. In line 10 we used a filter to remove an excess row filled with nulls and then on line 11 we renamed the previously unlabeled column to 'Payout Category'.

Afterwards, we loop through the numeric data in the years columns to replace the dollar signs, periods, and commas so that the data is in numeric form and convert them into integers



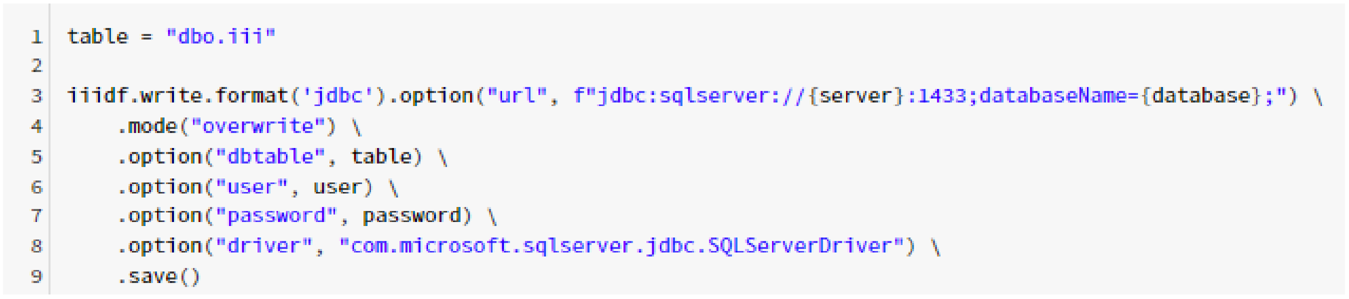
```python
 9  iiidf = spark.createDataFrame(iiidf)
10  iiidf = iiidf.filter(iiidf['2016']!='null')
11  iiidf = iiidf.withColumnRenamed("Unnamed: 0","Payout Category")
12  for i in ['2016','2017','2018','2019','2020']:
13      iiidf = iiidf.withColumn(i, FUNC.regexp_replace(i, "[^a-zA-Z0-9]", ""))
14      iiidf = iiidf.withColumn(i, col(i).cast('integer'))
```

The database table for this data was created with



```sql
CREATE TABLE iii
(
    Payout_Category NVARCHAR(100),
    amt_2016 int,
    amt_2017 int,
    amt_2018 int,
    amt_2019 int,
    amt_2020 int
)
```

And the data was written with



```python
table = "dbo.iii"

iiidf.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .mode("overwrite") \
    .option("dbtable", table) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
```

Investopedia data

This code gets the Life Insurance Rates by Age table from the website shown in the link, and as there was only one table on this page, it was naturally the first one

```
url="https://www.investopedia.com/articles/personal-finance/022615/how-age-affects-life-insurance-rates.asp"
req = Request(url, headers={'User-Agent': 'Mozilla/5.0'})
web_byte = urlopen(req).read()
webpage = web_byte.decode('utf-8')

rates = pd.read_html(webpage)[0]
```

Line 7 converts the pandas data frame into a spark data frame
Line 9 fills the NA's in the Age category with the age previously stated
Line 11 we looked through the other columns not labeled 'Gender' and replaced the dollar sign so it would just be the integer value.
Line 12 just converts the values to numeric and checks for errors

```
 7  rates  = pd.DataFrame(rates.values[1:], columns=rates.iloc[0] )
 8  # fill forward, replace nan in age with above age
 9  rates = rates.fillna(method='ffill')
10
11  for i in rates.loc[:, rates.columns != "Gender"]:
12      rates[i] = rates[i].str.replace("$","", regex=True)
13      rates[i] = pd.to_numeric(rates[i], errors = 'coerce')
14  rates = spark.createDataFrame(rates)
```

The database table for this data was created with

```
CREATE TABLE rates
(
    gender NVARCHAR(20),
    age int,
    lowest float,
    low float,
    high float,
    highest float
)
```

And the data was written with

```
table = "dbo.rates"

rates.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .mode("overwrite") \
    .option("dbtable", table) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
```

Best Life Rates data

This code shown at the beginning gets the list of tables from the website shown, of which there were many. From this list, we got the 1st, 2nd, 10th, and 11th tables corresponding to total market penetration parts one and two, why consumers purchase life insurance, and why consumers don't purchase life insurance, respectively. The total market penetration part one shows how many US adults were covered by life insurance by year, and part 2 shows the ownership gap per year, which is the difference between the number of people who believe they should have life insurance and those that do.

```
url="https://www.bestliferates.org/statistics"
req = Request(url, headers={'User-Agent': 'Mozilla/5.0'})

web_byte = urlopen(req).read()

webpage = web_byte.decode('utf-8')
pd.read_html(webpage)
```

```
life_insured = pd.read_html(webpage)[0]
```

```
want_insured = pd.read_html(webpage)[1]
```

```
why_insured = pd.read_html(webpage)[9]
```

```
why_not_insured = pd.read_html(webpage)[10]
```

The life insurance coverage data and ownership gap were transformed in the same way, with the only cleaning being removing the percentage signs from the percentage column and then converting the column to numeric data

```
life_insured = spark.createDataFrame(life_insured)
life_insured = life_insured.withColumn('Percentage', FUNC.regexp_replace('Percentage', "[^a-zA-Z0-9]", ""))
life_insured = life_insured.withColumn('Percentage', col('Percentage').cast('integer'))
```

```
want_insured = spark.createDataFrame(want_insured)
want_insured = want_insured.withColumn('Percentage', FUNC.regexp_replace('Percentage', "[^a-zA-Z0-9]", ""))
want_insured = want_insured.withColumn('Percentage', col('Percentage').cast('integer'))
```

They then had tables created and were imported into the database as seen below

```
CREATE TABLE life_insured
(
    year int,
    portion int
)

CREATE TABLE ownership_gap
(
    year int,
    portion int
)
```

```
table = "dbo.life_insured"

life_insured.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .mode("overwrite") \
    .option("dbtable", table) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
```

```
table = "dbo.ownership_gap"

want_insured.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .mode("overwrite") \
    .option("dbtable", table) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
```

The data for why people were insured had to be cleaned very specifically to remove the percentage signs but not the negative signs on the change column as seen in line 5, but otherwise all that was done was renaming an unlabeled column and converting the numeric data into numeric form

```
2  why_insured  = pd.DataFrame(why_insured.values[1:], columns=why_insured.iloc[0] )
3  why_insured.rename(columns={ why_insured.columns[0]: "Reason to hold life insurance" }, inplace = True)
4  for i in why_insured.loc[:, why_insured.columns != "Reason to hold life insurance"]:
5      why_insured[i] = why_insured[i].str.replace("%","", regex=True)
6      why_insured[i] = pd.to_numeric(why_insured[i], errors = 'coerce')
7
8  why_insured = spark.createDataFrame(why_insured)
```

Its table was created and then the data was imported into the database the same way as previous tables

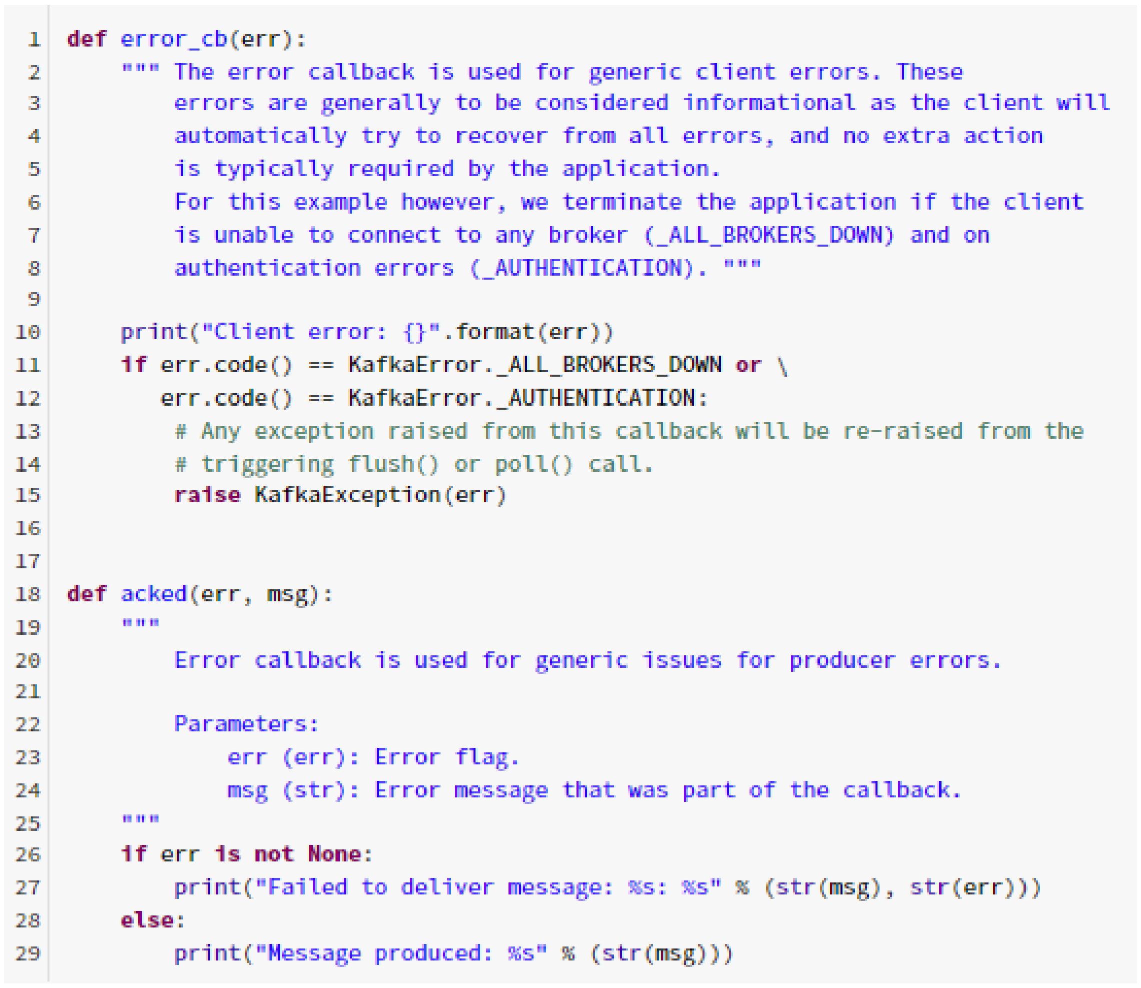
```sql
CREATE TABLE why_insured
(
    reason NVARCHAR(100),
    year1 int,
    year2 int,
    change int
)
```

```python
table = "dbo.why_insured"

why_insured.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};")
    .mode("overwrite") \
    .option("dbtable", table) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
```

The table for why people are not life insured was what we used to demonstrate a run of the Kafka consumer producer dataflow, and as such it is different from all the other ETL noted thus far, even if it begins similarly to the previous best life rates tables by renaming an unlabeled column, removing percent signs and converting columns to numeric type as seen below

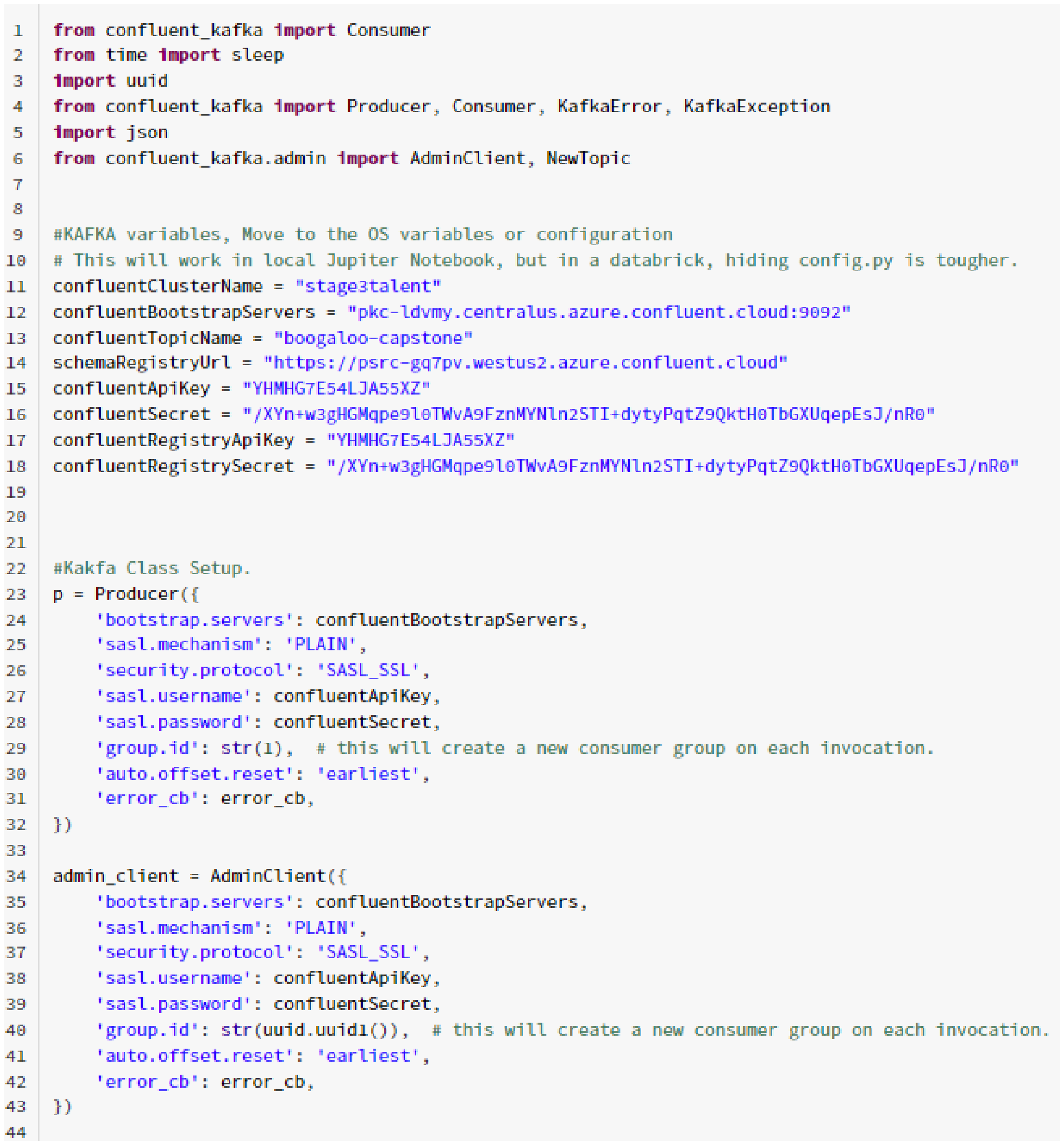
```python
why_not_insured.rename(columns={ why_not_insured.columns[0]: "Reason to not hold life insurance", why_not_insured.columns[1]: "Percentage" }, inplace = True)
why_not_insured = spark.createDataFrame(why_not_insured)
why_not_insured = why_not_insured.withColumn('Percentage', FUNC.regexp_replace('Percentage', "[^a-zA-Z0-9]", ""))
why_not_insured = why_not_insured.withColumn('Percentage', col('Percentage').cast('integer'))
```

The following code isn't strictly necessary, as all it does is create error messages to aid in debugging and a slight change to later code could render this unnecessary, but for the sake of completeness it's included.



```python
def error_cb(err):
    """ The error callback is used for generic client errors. These
        errors are generally to be considered informational as the client will
        automatically try to recover from all errors, and no extra action
        is typically required by the application.
        For this example however, we terminate the application if the client
        is unable to connect to any broker (_ALL_BROKERS_DOWN) and on
        authentication errors (_AUTHENTICATION). """

    print("Client error: {}".format(err))
    if err.code() == KafkaError._ALL_BROKERS_DOWN or \
       err.code() == KafkaError._AUTHENTICATION:
        # Any exception raised from this callback will be re-raised from the
        # triggering flush() or poll() call.
        raise KafkaException(err)


def acked(err, msg):
    """
        Error callback is used for generic issues for producer errors.

        Parameters:
            err (err): Error flag.
            msg (str): Error message that was part of the callback.
    """
    if err is not None:
        print("Failed to deliver message: %s: %s" % (str(msg), str(err)))
    else:
        print("Message produced: %s" % (str(msg)))
```

This code imports some packages needed for the Kafka producer, as well as creating some variables to be used to perform the production message flow such as names of locations and access credentials. Furthermore, it creates the producer itself using some of these variables and an administrator client to create the topic where we will be sending our produced and consumed messages



```python
from confluent_kafka import Consumer
from time import sleep
import uuid
from confluent_kafka import Producer, Consumer, KafkaError, KafkaException
import json
from confluent_kafka.admin import AdminClient, NewTopic


#KAFKA variables, Move to the OS variables or configuration
# This will work in local Jupiter Notebook, but in a databrick, hiding config.py is tougher.
confluentClusterName = "stage3talent"
confluentBootstrapServers = "pkc-ldvmy.centralus.azure.confluent.cloud:9092"
confluentTopicName = "boogaloo-capstone"
schemaRegistryUrl = "https://psrc-gq7pv.westus2.azure.confluent.cloud"
confluentApiKey = "YHMHG7E54LJA55XZ"
confluentSecret = "/XYn+w3gHGMqpe9l0TWvA9FznMYNln2STI+dytyPqtZ9QktH0TbGXUqepEsJ/nR0"
confluentRegistryApiKey = "YHMHG7E54LJA55XZ"
confluentRegistrySecret = "/XYn+w3gHGMqpe9l0TWvA9FznMYNln2STI+dytyPqtZ9QktH0TbGXUqepEsJ/nR0"



#Kakfa Class Setup.
p = Producer({
    'bootstrap.servers': confluentBootstrapServers,
    'sasl.mechanism': 'PLAIN',
    'security.protocol': 'SASL_SSL',
    'sasl.username': confluentApiKey,
    'sasl.password': confluentSecret,
    'group.id': str(1),  # this will create a new consumer group on each invocation.
    'auto.offset.reset': 'earliest',
    'error_cb': error_cb,
})

admin_client = AdminClient({
    'bootstrap.servers': confluentBootstrapServers,
    'sasl.mechanism': 'PLAIN',
    'security.protocol': 'SASL_SSL',
    'sasl.username': confluentApiKey,
    'sasl.password': confluentSecret,
    'group.id': str(uuid.uuid1()),  # this will create a new consumer group on each invocation.
    'auto.offset.reset': 'earliest',
    'error_cb': error_cb,
})

```

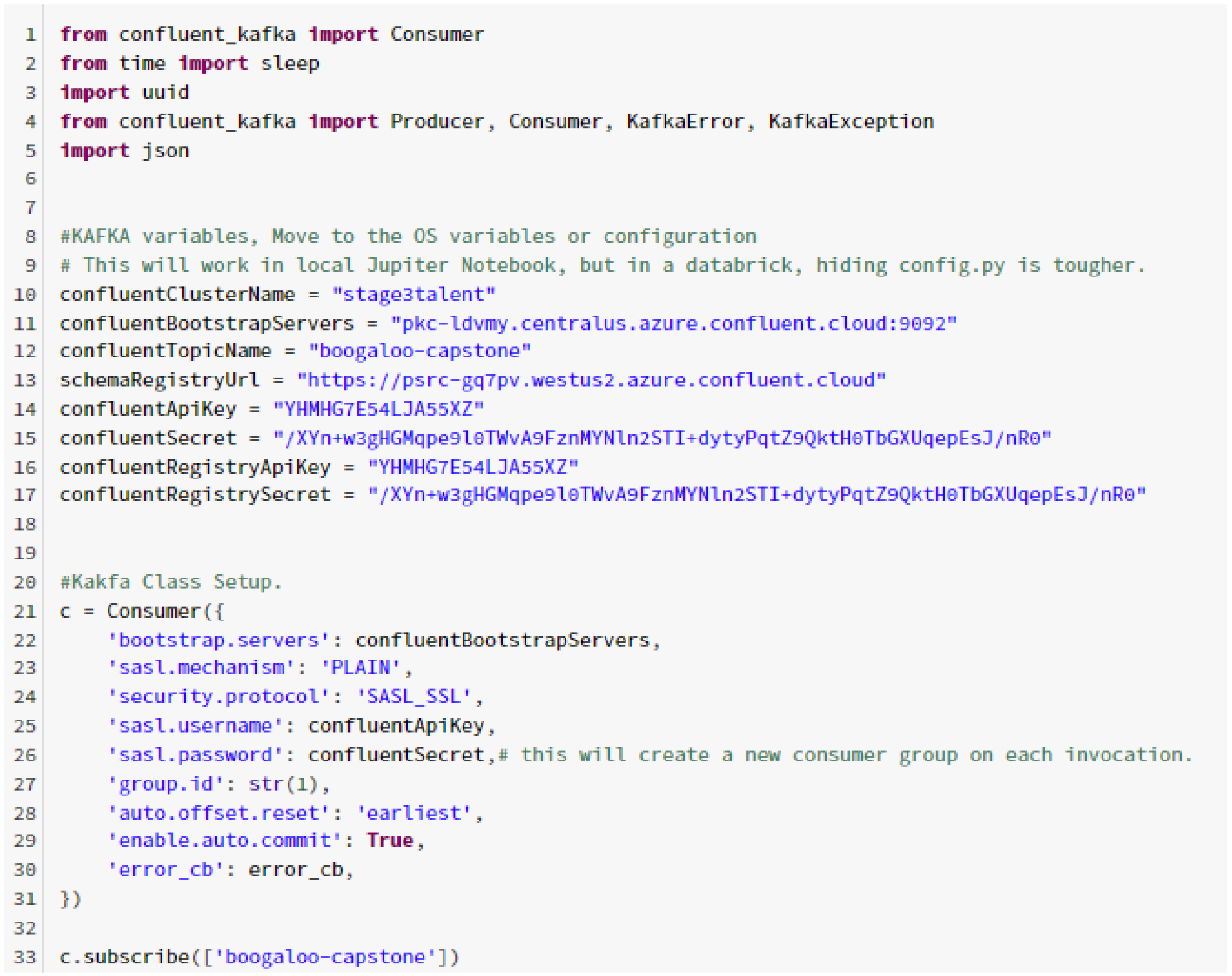
This code creates the topic that we will be sending and consuming messages from, and tests if the topic was created successfully. This code was only run once, and while we could make a try-except block for the first 5 lines shown, we decided to just comment out this entire data brick cell for future runs

```python
topic_list = []

topic_list.append(NewTopic("boogaloo-capstone", 1, 3))
admin_client.create_topics(topic_list)
futures = admin_client.create_topics(topic_list)


try:
    record_metadata = []
    for k, future in futures.items():
        # f = i.get(timeout=10)
        print(f"type(k): {type(k)}")
        print(f"type(v): {type(future)}")
        print(future.result())

except KafkaError:
    # Decide what to do if produce request failed...
    print(traceback.format_exc())
    result = 'Fail'
finally:
    print("finally")
```

This code iterates over the rows of data in the table for why Americans aren't life insured and produces messages to be sent to the previously created topic as python dictionaries, clears the producer queue so that it's empty for the next message, and then waits 5 seconds before producing the next message

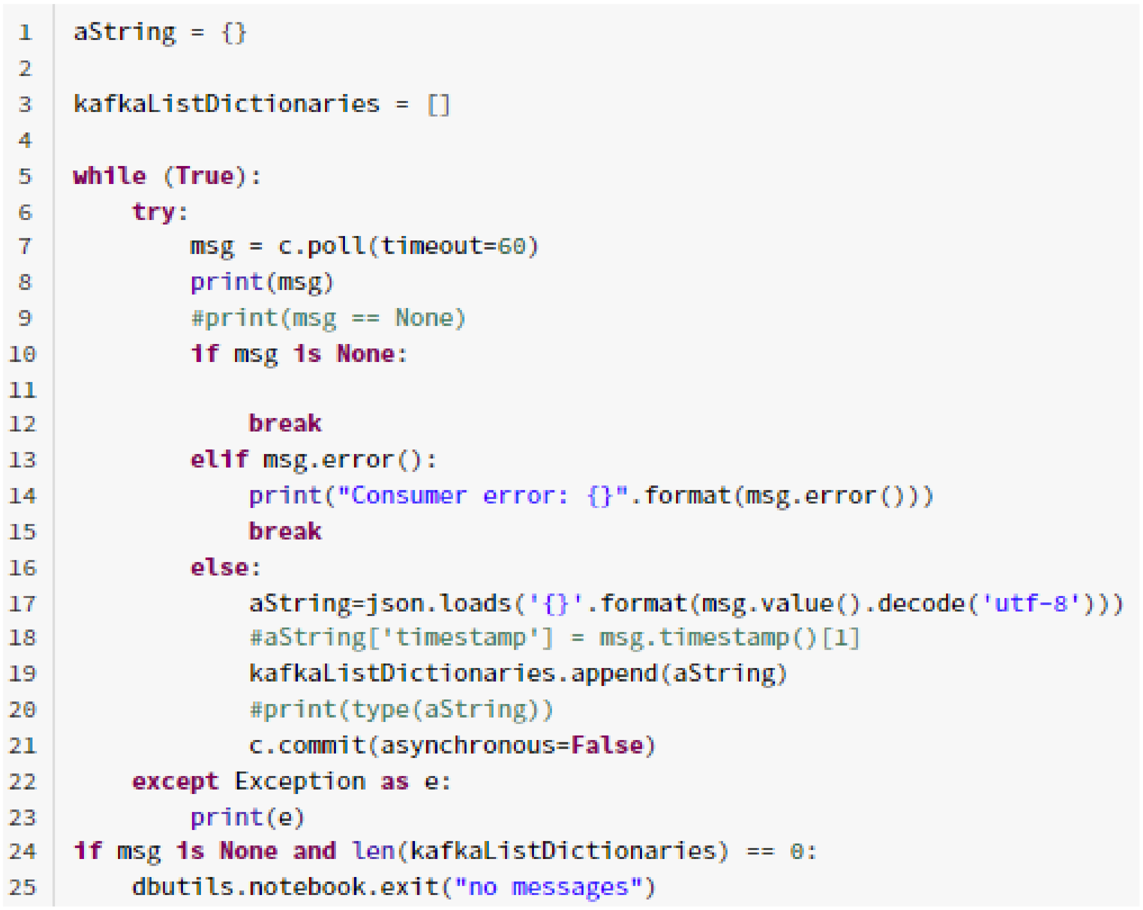
```python
for row in why_not_insured.rdd.collect():
    p.produce(confluentTopicName,json.dumps( row.asDict() ))
    p.flush()
    print(row.asDict())
    sleep(5)
```

This code imports some packages needed for the Kafka consumer, as well as creating some variables to be used to consume the produced messages such as names of locations and access credentials. Furthermore, it creates the producer itself using some of these variables and sets the consumer to connect to our previously created topic



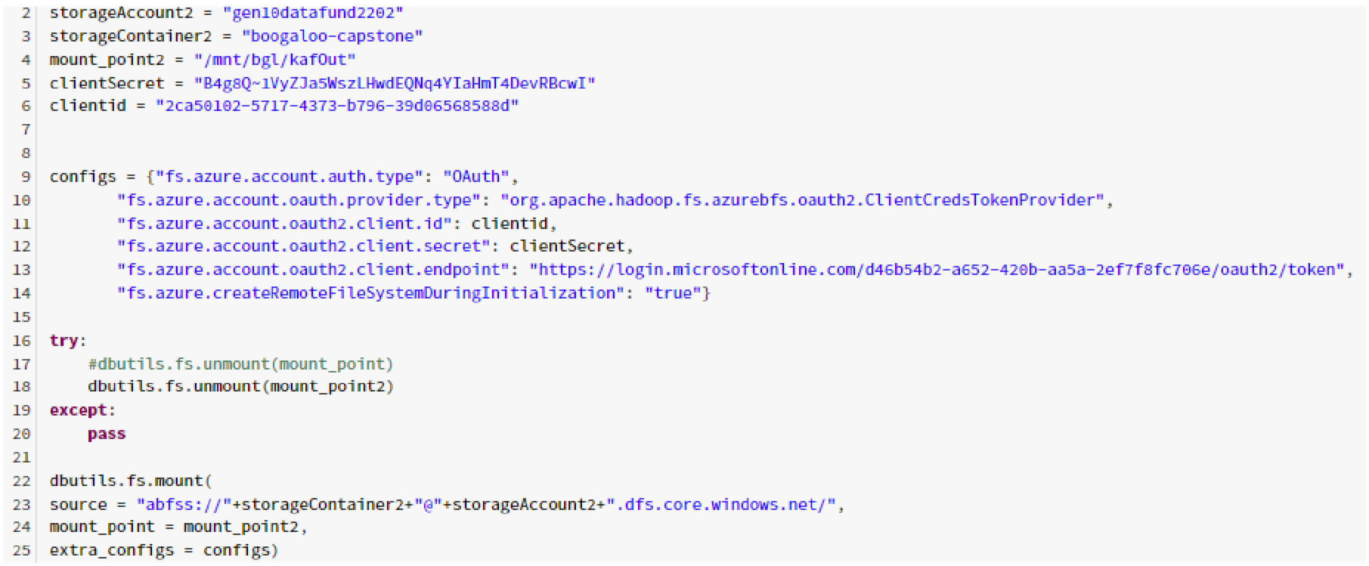
```python
from confluent_kafka import Consumer
from time import sleep
import uuid
from confluent_kafka import Producer, Consumer, KafkaError, KafkaException
import json


#KAFKA variables, Move to the OS variables or configuration
# This will work in local Jupiter Notebook, but in a databrick, hiding config.py is tougher.
confluentClusterName = "stage3talent"
confluentBootstrapServers = "pkc-ldvmy.centralus.azure.confluent.cloud:9092"
confluentTopicName = "boogaloo-capstone"
schemaRegistryUrl = "https://psrc-gq7pv.westus2.azure.confluent.cloud"
confluentApiKey = "YHMHG7E54LJA55XZ"
confluentSecret = "/XYn+w3gHGMqpe9l0TWvA9FznMYNln2STI+dytyPqtZ9QktH0TbGXUqepEsJ/nR0"
confluentRegistryApiKey = "YHMHG7E54LJA55XZ"
confluentRegistrySecret = "/XYn+w3gHGMqpe9l0TWvA9FznMYNln2STI+dytyPqtZ9QktH0TbGXUqepEsJ/nR0"


#Kakfa Class Setup.
c = Consumer({
    'bootstrap.servers': confluentBootstrapServers,
    'sasl.mechanism': 'PLAIN',
    'security.protocol': 'SASL_SSL',
    'sasl.username': confluentApiKey,
    'sasl.password': confluentSecret,# this will create a new consumer group on each invocation.
    'group.id': str(1),
    'auto.offset.reset': 'earliest',
    'enable.auto.commit': True,
    'error_cb': error_cb,
})

c.subscribe(['boogaloo-capstone'])
```

This code runs and consumes messages until the message queue is empty or it doesn't find a message after running for a minute. If a message is found, it's appended to a list of dictionaries of the other messages. If no messages are found, the databrick is exited



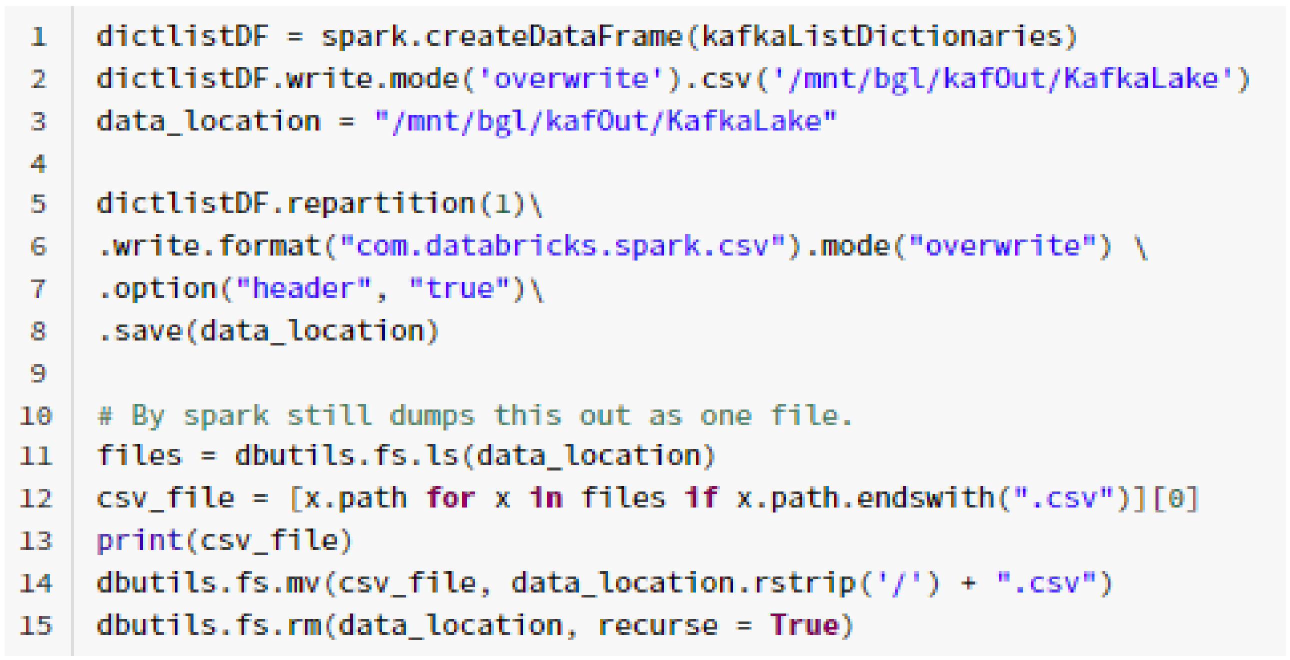
```python
aString = {}

kafkaListDictionaries = []

while (True):
    try:
        msg = c.poll(timeout=60)
        print(msg)
        #print(msg == None)
        if msg is None:

            break
        elif msg.error():
            print("Consumer error: {}".format(msg.error()))
            break
        else:
            aString=json.loads('{}'.format(msg.value().decode('utf-8')))
            #aString['timestamp'] = msg.timestamp()[1]
            kafkaListDictionaries.append(aString)
            #print(type(aString))
            c.commit(asynchronous=False)
    except Exception as e:
        print(e)
if msg is None and len(kafkaListDictionaries) == 0:
    dbutils.notebook.exit("no messages")
```

This code creates a connection point to an Azure data lake using various names and credentials as variables



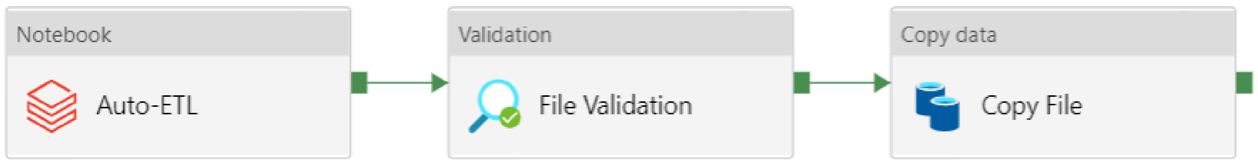
```python
storageAccount2 = "gen10datafund2202"
storageContainer2 = "boogaloo-capstone"
mount_point2 = "/mnt/bgl/kafOut"
clientSecret = "B4g8Q~1VyZJa5WszLHwdEQNq4YIaHmT4DevRBcwI"
clientid = "2ca50102-5717-4373-b796-39d06568588d"


configs = {"fs.azure.account.auth.type": "OAuth",
       "fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",
       "fs.azure.account.oauth2.client.id": clientid,
       "fs.azure.account.oauth2.client.secret": clientSecret,
       "fs.azure.account.oauth2.client.endpoint": "https://login.microsoftonline.com/d46b54b2-a652-420b-aa5a-2ef7f8fc706e/oauth2/token",
       "fs.azure.createRemoteFileSystemDuringInitialization": "true"}

try:
    #dbutils.fs.unmount(mount_point)
    dbutils.fs.unmount(mount_point2)
except:
    pass

dbutils.fs.mount(
source = "abfss://"+storageContainer2+"@"+storageAccount2+".dfs.core.windows.net/",
mount_point = mount_point2,
extra_configs = configs)
```

This code creates a spark data frame from the list of dictionaries made from the consumed messages and writes it to the data lake from the connection point created previously as a csv. It then combines the files written in parallel due to the cloud process as one file so that the data factory only needs to confirm one file later on



```
dictlistDF = spark.createDataFrame(kafkaListDictionaries)
dictlistDF.write.mode('overwrite').csv('/mnt/bgl/kafOut/KafkaLake')
data_location = "/mnt/bgl/kafOut/KafkaLake"

dictlistDF.repartition(1)\
.write.format("com.databricks.spark.csv").mode("overwrite") \
.option("header", "true")\
.save(data_location)

# By spark still dumps this out as one file.
files = dbutils.fs.ls(data_location)
csv_file = [x.path for x in files if x.path.endswith(".csv")][0]
print(csv_file)
dbutils.fs.mv(csv_file, data_location.rstrip('/') + ".csv")
dbutils.fs.rm(data_location, recurse = True)
```

This shows the pipeline in Azure data factory which can automate this entire ETL process for the web scraped data. It first runs the data brick with all the code shown above, then after confirming that the data lake contains the csv file for the table for why people aren't insured, it copies it over to the SQL database



Census Data ETL:

Once loaded, the columns were renamed things relevant to the data, using a format similar to how they appear on the Census Bureau Data Explorer.

After renaming the columns, these tables were saved to the SQL Server.

HI_05 Data ETL:

From the links provided in the Healthcare and Insurance portion of the Census Datasets section of the Capstone: Research Industry-Related Data page of the M11-Capstone module we found several useful tables by these steps:

1. following the 1st link: Health Insurance Data from the Census
2. following the 2nd link: Current Population Survey Tables for Health Insurance
3. following the 6th link: Health Insurance: HI-05

4. following the link pointing to other year's data

which landed us at the following website: https://www.census.gov/data/tables/time-series/demo/health-insurance/acs-hi.html from which we downloaded the HI-05 data for each of the years 2019-2011 (2011 and 2012 from the previous page before the change to other years)

For every year before 2019's data, change the downloaded file from .xls to .xlsx. For the years 2011-2015 load that data into an excel power query upon which preform the following transformations:

1. Restrict the rows kept containing state name and % with any health insurance
2. Transpose the table
3. Change column names
4. Change the state names to lowercase

These transformations were preformed to make the datasets similar to those of other years.

Load all of these files to the DataLake and preform transformations in the data factory power query which add a year column (fixed for each dataset), filter the data to All Persons (All People in the 2019 case), create a State:Year column by merging the appropriate columns of each table, remove unnecessary columns so that the final dataset is just State:Year and % with Health Insurance, and load the transformed tables back into the DataLake as CSVs.

For the years 2011-2015 this reloading step accidentally introduced a number of extra leading commas attached to each row of the CSVs, which would clog up the workflow, so since the files were small enough, going into each to remove the unnecessary additional commas, while minorly time consuming, isn't overly difficult. Once these changes have been made at the blob level, proceed to load the files in a DataBrick and save them to the SQL server.

State Abbreviations:

One of the datasets we found online used State Abbreviations instead of full state names, so a table containing all the state abbreviations was downloaded from https://worldpopulationreview.com/states/state-abbreviations and uploaded to the DataLake from which it was loaded into a DataBrick and saved to the SQL server.

Model Data:

Load all of the Census Data (S1901s, S2791s and HI-05s) along with the Life Expectancy and State Abbreviations tables from the SQL Server into a DataBrick, perform some transformations on the Census Data (convert # to %, Add Year Columns, create State:Year merge points), join the Life Expectancy and State Abbreviations tables to then create a State:Year column, and finally join all the tables together on the State:Year columns (some column renaming may be necessary for ease of code flow). Once all the data is appropriately merged, save the end result to the SQL database (note that all of the operations in this step can be done in SQL, they were originally done in the DataBrick before the initial saves to the SQL Server, and for convenience that code was minorly repurposed instead of writing new SQL to accomplish the same task)

Conclusion

Wrap it up here.

Overall