ETL REPORT

Contributors: Alex Mora, Yansong Tang, Isaac Lee, Will Stearns

TABLE OF CONTENTS

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

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 preforms 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

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 DATA

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

```
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 data bricks, where the only difference was the url to distinguish various sites.

```
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

```
1
2    CREATE LOGIN bglcap
3    with Password = '48Gheq0Iz9t'
4
5    CREATE USER bglcap from login bglcap;
6
7    EXEC sp_addrolemember 'db_owner', 'bglcap'
8
```

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

```
1  database = "boogaloo-capstone-human-life"
2  table = "dbo.iii"
3  user = "bglcap"
4  password  = "48Gheq0Iz9t"
5  server = "gen10-data-fundamentals-22-02-sql-server.database.windows.net"
6
```

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

```
2  url="https://www.iii.org/table-archive/22403"
3  req = Request(url, headers={'User-Agent': 'Mozilla/5.0'})
4
5  web_byte = urlopen(req).read()
6
7  webpage = web_byte.decode('utf-8')
8  iiidf = pd.read_html(webpage)[0]
```

We initially converted 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

```
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:

```
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:

```
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

```
rates  = pd.DataFrame(rates.values[1:], columns=rates.iloc[0] )
# fill forward, replace nan in age with above age
rates = rates.fillna(method='ffill')

for i in rates.loc[:, rates.columns != "Gender"]:
    rates[i] = rates[i].str.replace("$","", regex=True)
    rates[i] = pd.to_numeric(rates[i], errors = 'coerce')
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

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

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

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

```
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

```
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.

```
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

```
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

```
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 data brick is exited

```
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

```
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 AND SUPPORTING DATA

To begin we must set up a mount point and set up the correct credentials, configurations, and additional information to pass the mount point.

```
1  ###### Mount Point 1 through Oauth security.
2  storageAccount = "gen10datafund2202"
3  storageContainer = "boogaloo-capstone"
4  clientSecret = "B4g8Q~1VyZJa5WszLHwdEQNq4YIaHmT4DevRBcwI"
5  clientid = "2ca50102-5717-4373-b796-39d06568588d"
6  mount_point = "/mnt/willstearns/capstone"
7  #20200906-20201006/Detroit911-20200906-20201006.csv
8
9
10 configs = {"fs.azure.account.auth.type": "OAuth",
11        "fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",
12        "fs.azure.account.oauth2.client.id": clientid,
13        "fs.azure.account.oauth2.client.secret": clientSecret,
14        "fs.azure.account.oauth2.client.endpoint": "https://login.microsoftonline.com/d46b54b2-a652-420b-aa5a-2ef7f8fc706e/oauth2/token",
15        "fs.azure.createRemoteFileSystemDuringInitialization": "true"}
16
17 try:
18     dbutils.fs.unmount(mount_point)
19 except:
20     pass
21
22 dbutils.fs.mount(
23 source = "abfss://"+storageContainer+"@"+storageAccount+".dfs.core.windows.net/",
24 mount_point = mount_point,
25 extra_configs = configs)
```

Once a connection is made, the ability to call data is available.

1. From the link below we will get the HI_05 Data in the form of CSV's.
   Then we will transfer the files into the Storage Account so they will be available to manipulate within the Data Brick.
   https://www.census.gov/data/tables/time-series/demo/health-insurance/acs-hi.html
2. Following the same steps, we will be adding a website that will convert the numeric State values from the dataset.
   Locate and download the CSV file.
   https://worldpopulationreview.com/states/state-abbreviations
3. Once the CSV's are downloaded locate the Storage Account 'gen10datafund2202' and upload the CSV files.

We will now get the HI_05 data First step is to pull CSV's from the Data Lake.

1. Name the function of calling the HI_05 CSV. *(line 2)*
2. Call the State Abbreviations CSV with the spark.read.option function to declare the first line as headers *(line 3)*

Modifications to the CSV's will now be made.

1. Alter Insured_Rates by renaming columns.
   a. Change '_c0' to 'State:Year' *(line 4)*
   b. Change '_c1' to 'Percent Insured' *(line 5)*
2. Change the 'Percent Insured' to be displayed in decimal. *(line 6)*

```
#get dataframes from DataLake CSVs, change column names
Insured_Rates = spark.read.csv('/mnt/willstearns/capstone/Cleaned HI_05 Data/')
State_Abbrs = spark.read.option("header",True).csv('/mnt/willstearns/capstone/csvData.csv')
Insured_Rates=Insured_Rates.withColumnRenamed("_c0","State:Year")
Insured_Rates=Insured_Rates.withColumnRenamed("_c1","Percent Insured")
Insured_Rates=Insured_Rates.withColumn("Percent Insured",Insured_Rates["Percent Insured"]/100.0)
```

Next, we will request to call the census data using the get HTML function

1. Import the libraries needed to be able to use the required functions
    a. import json *(line 1)*
    b. import request *(line 2)*
2. Set the call function
    a. Define the get HTML call function *(line 5)*
3. Get various tables from the census API (**IMPORTANT**: request to get a Census API Key)
    a. Use the getHTML function and insert the url to call the Census API *(for lines 9, 13, 18, 21)*
    b. Following the call, transform the data into json format *(for lines 11, 16, 19, 23)*

Listed below are the urls used to call the data and then turned into JSON format.

```
import json
import requests

#html call function
def getHTML(url):
    response = requests.get(url)
    return response.text
#get various tables from census API, if repeating this please get and use your own CensusAPI key.
S1901_18_html = getHTML('''https://api.census.gov/data/2018/acs/acs5/subject?get=S1901_C01_012E,
NAME&for=state:*&key=4bd66954c86bd0a6ba524b9766beb04e32b367f6''')
S1901_18_json_data = json.loads(S1901_18_html)

S2701_18_html = getHTML('''https://api.census.gov/data/2018/acs/acs5/subject?get=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,
NAME&for=state:*&key=4bd66954c86bd0a6ba524b9766beb04e32b367f6''')
S2701_18_json_data = json.loads(S2701_18_html)

S1901_19_html = getHTML("https://api.census.gov/data/2019/acs/acs5/subject?get=S1901_C01_012E,NAME&for=state:*&key=4bd66954c86bd0a6ba524b9766beb04e32b367f6")
S1901_19_json_data = json.loads(S1901_19_html)

S2701_19_html = getHTML('''https://api.census.gov/data/2019/acs/acs5/subject?get=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,NAME&for=state:*&key=4bd66954c86bd0a6ba524b9766beb04e32b367f6''')
S2701_19_json_data = json.loads(S2701_19_html)
```

Following the formatting to json, we will now create data frames from the api calls with renamed columns to descriptions of the data.

1. Call upon the pyspark.sql.types to import IntergerType
2. Make the tables with the Median Household Income into data frames; select specific sections of the data *(line 3)*
3. Rename the columns *(line 4)*
4. Create a for loop for the specified columns to convert into integers *(line 5 – 6)*
    a. Repeat steps 2 – 4 for 2019 table *(line 8 – 11)*
5. Make the demographic tables into data frames; select the required columns *(line 13 –15)*
6. Create a loop to for the specified columns to convert into integers *(line 16 – 19)*
    a. Repeat steps 5 – 6 for the 2019 table of demographics *(line 21 – 26)*

```
from pyspark.sql.types import IntegerType
#Create dataframes from the API calls with renamed columns from the Census codes to English descriptions of column data
S1901_18_df=spark.createDataFrame(S1901_18_json_data[1:],S1901_18_json_data[0])
S1901_18_df=S1901_18_df.withColumnRenamed("S1901_C01_012E","Median Household Income")
for column in {"Median Household Income","state"}:
    S1901_18_df=S1901_18_df.withColumn(f"{column}",S1901_18_df[column].cast(IntegerType()))

S1901_19_df=spark.createDataFrame(S1901_19_json_data[1:],S1901_19_json_data[0])
S1901_19_df=S1901_19_df.withColumnRenamed("S1901_C01_012E","Median Household Income")
for column in {"Median Household Income","state"}:
    S1901_19_df=S1901_19_df.withColumn(f"{column}",S1901_19_df[column].cast(IntegerType()))

S2701_18_df=spark.createDataFrame(S2701_18_json_data[1:],["Total Population","Male","Female","White",
"Black or African American","American Indian and Alaska Native","Asian","Native Hawaiian and other Pacific Islander",
"Other","Multiracial","Hispanic or Latino","State","state_num"])
for column in {"Total Population","Male","Female","White","Black or African American",
"American Indian and Alaska Native","Asian","Native Hawaiian and other Pacific Islander","Other",
"Multiracial","Hispanic or Latino","state_num"}:
S2701_18_df=S2701_18_df.withColumn(f"{column}",S2701_18_df[column].cast(IntegerType()))

S2701_19_df=spark.createDataFrame(S2701_19_json_data[1:],["Total Population","Male","Female","White","Black or African American",
"American Indian and Alaska Native","Asian","Native Hawaiian and other Pacific Islander","Other","Multiracial",
"Hispanic or Latino","State","state_num"])
for column in {"Total Population","Male","Female","White","Black or African American","American Indian and Alaska Native",
"Asian","Native Hawaiian and other Pacific Islander","Other","Multiracial","Hispanic or Latino","state_num"}:
S2701_19_df=S2701_19_df.withColumn(f"{column}",S2701_19_df[column].cast(IntegerType()))
```

The data we used were formatted into tables and stored into our database “boogaloo-capstone-human-life." Renamed the tables to a similar title.

1. Set database
2. Insert tables that will be going into the database
3. Inset credentials
4. Set connections to server

```
#Necessary SQL connection pieces
database = "boogaloo-capstone-human-life"
tableIR="dbo.health_insurance_rates"
tableSA="dbo.state_abbrs"
table18_S1901="dbo.S1901_2018"
table19_S1901="dbo.S1901_2019"
table18_S2701="dbo.S2701_2018"
table19_S2701="dbo.S2701_2019"
tableLE="dbo.cdc_state"
tableLE_All="dbo.Model_Data"
user = "bglcap"
password  = "48Gheq0Iz9t"
server = "gen10-data-fundamentals-22-02-sql-server.database.windows.net"
```

To save the tables we will need to call the database using our credentials and write to the database. (Repeatable)

1. Begin with a data frame to write to the SQL server *(line 2)*
    a. We use the format(‘jdbc’) so that it will be able to be used within the database *(line 2)*
    b. .mode(“overwrite”) overwrites any table that is already made, in case of conflict *(line 3)*
    c. .option sets up the credentials *(line 4 – 7)*
    d. .save() saves the dataframe to the SQL server *(line 8)*
2. Repeat step 1 for the remaining tables

```
#Write created dataframes to SQL Server
Insured_Rates.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .mode("overwrite") \
    .option("dbtable", tableIR) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
State_Abbrs.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .mode("overwrite") \
    .option("dbtable", tableSA) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
S1901_18_df.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .mode("overwrite") \
    .option("dbtable", table18_S1901) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
S1901_19_df.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .mode("overwrite") \
    .option("dbtable", table19_S1901) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
S2701_18_df.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .mode("overwrite") \
    .option("dbtable", table18_S2701) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
S2701_19_df.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .mode("overwrite") \
    .option("dbtable", table19_S2701) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .save()
```

After the API call and loading are completed, the data is now available to call from the database.

In a new page, set up the necessary SQL connections

1. Set up database
2. Call tables
3. Credentials
4. Input server

```
#Necessary SQL connection pieces
database = "boogaloo-capstone-human-life"
tableIR="dbo.health_insurance_rates"
tableSA="dbo.state_abbrs"
table18_S1901="dbo.S1901_2018"
table19_S1901="dbo.S1901_2019"
table18_S2701="dbo.S2701_2018"
table19_S2701="dbo.S2701_2019"
tableLE="dbo.cdc_state"
tableLE_All="dbo.Model_Data"
user = "bglcap"
password  = "48Gheq0Iz9t"
server = "gen10-data-fundamentals-22-02-sql-server.database.windows.net"
```

Similar to saving the tables into the SQL server, we can now load the tables from the server.

1. Begin with a data frame to read to the SQL server *(line 2)*

a. We use the format('jdbc') so that it will be able to be used within the database *(line 2)*
b. .option sets up the credentials *(line 3 – 6)*
c. .load() allows for the table to be brought into the desired location *(line 7)*

2. Repeat step 1 for the remaining tables

```
1  #Read in data from SQL Server
2  Insured_Rates=spark.read.format("jdbc").option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
3      .option("dbtable", tableIR) \
4      .option("user", user) \
5      .option("password", password) \
6      .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
7      .load()
8  State_Abbrs=spark.read.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
9      .option("dbtable", tableSA) \
10     .option("user", user) \
11     .option("password", password) \
12     .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
13     .load()
14 S1901_18_df=spark.read.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
15     .option("dbtable", table18_S1901) \
16     .option("user", user) \
17     .option("password", password) \
18     .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
19     .load()
20 S1901_19_df=spark.read.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
21     .option("dbtable", table19_S1901) \
22     .option("user", user) \
23     .option("password", password) \
24     .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
25     .load()
26 S2701_18_df=spark.read.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
27     .option("dbtable", table18_S2701) \
28     .option("user", user) \
29     .option("password", password) \
30     .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
31     .load()
32 S2701_19_df=spark.read.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
33     .option("dbtable", table19_S2701) \
34     .option("user", user) \
35     .option("password", password) \
36     .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
37     .load()
38 Life_Expectancy = spark.read.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
39     .option("dbtable", tableLE) \
40     .option("user", user) \
41     .option("password", password) \
42     .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
43     .load()
```

Now that they are loaded in, we are able to manipulate and clean our data.

1. Call pyspark functions *(line 1 – 2)*
   a. from pyspark.sql.functions import lit
   b. from pyspark.sql.functions import concat_ws
   c. The 'lit' function will make a literal variable and concat_ws will merge two string inputs
2. We will first look at the *S1901_18_df* and *S1901_19_df* tables.
3. Add a *year* column to the dataset *(line 4)*
4. Make a *concatenation column* of NAME and YEAR labelled *State:Year (line 5)*
5. Convert *Median Household Income* into a *'double' (line 6)*
6. Reduce data frame to relevant columns (line 6)

```
from pyspark.sql.functions import lit
from pyspark.sql.functions import concat_ws
#Add a Year column (all 2018), make a concatenated column of NAME and Year called State:Year,
S1901_18_df_clean=S1901_18_df.withColumn("Year", lit(2018))
S1901_18_df_clean=S1901_18_df_clean.withColumn("State:Year",concat_ws(":","NAME",'Year'))
S1901_18_df_clean=S1901_18_df_clean[['State:Year', 'Median Household Income']]


#Same as cleaning of S1901_18_df just for different year(2019)
S1901_19_df_clean=S1901_19_df.withColumn("Year", lit(2019))
S1901_19_df_clean=S1901_19_df_clean.withColumn("State:Year",concat_ws(":","NAME",'Year'))
S1901_19_df_clean=S1901_19_df_clean[['State:Year', 'Median Household Income']]
```

Clean the Life Expectancy data set.

1. Change the column name for future join *(line 16 – 17)*

```
#Change column name for future join
Life_Expectancy_clean=Life_Expectancy.withColumnRenamed("STATE","Code")
Life_Expectancy_clean=Life_Expectancy_clean.withColumnRenamed("RATE","Life Expectancy")
```

Clean and transform the demographic tables.

1. First table to clean will be the S2701_18_P_df
2. We will be going through each category and dividing it by the 'Total Population' to get the percentage displayed in decimal form
    a. Repeat step 2 for the remaining categories *(line 20 – 29)*
3. Make 'Year' into a constant variable
4. Convert the individual categories of 'State' and 'Year' to 'State:Year'
    a. Using the concat_ws function
5. Reduce the data frame to relevant columns *(line 32 – 33)*

```
S2701_18_P_df=S2701_18_df.withColumn("Male",S2701_18_df["Male"]/S2701_18_df["Total Population"])
S2701_18_P_df=S2701_18_P_df.withColumn("Female",S2701_18_P_df["Female"]/S2701_18_P_df["Total Population"])
S2701_18_P_df=S2701_18_P_df.withColumn("White",S2701_18_P_df["White"]/S2701_18_P_df["Total Population"])
S2701_18_P_df=S2701_18_P_df.withColumn("Black or African American",S2701_18_P_df["Black or African American"]/S2701_18_P_df["Total Population"])
S2701_18_P_df=S2701_18_P_df.withColumn("American Indian and Alaska Native",S2701_18_P_df["American Indian and Alaska Native"]/S2701_18_P_df["Total Population"])
S2701_18_P_df=S2701_18_P_df.withColumn("Asian",S2701_18_P_df["Asian"]/S2701_18_P_df["Total Population"])
S2701_18_P_df=S2701_18_P_df.withColumn("Native Hawaiian and other Pacific Islander",S2701_18_P_df["Native Hawaiian and other Pacific Islander"]/S2701_18_P_df["Total Population"])
S2701_18_P_df=S2701_18_P_df.withColumn("Other",S2701_18_P_df["Other"]/S2701_18_P_df["Total Population"])
S2701_18_P_df=S2701_18_P_df.withColumn("Multiracial",S2701_18_P_df["Multiracial"]/S2701_18_P_df["Total Population"])
S2701_18_P_df=S2701_18_P_df.withColumn("Hispanic or Latino",S2701_18_P_df["Hispanic or Latino"]/S2701_18_P_df["Total Population"])
S2701_18_P_df=S2701_18_P_df.withColumn("Year", lit(2018))
S2701_18_P_df=S2701_18_P_df.withColumn("State:Year",concat_ws(":","State",'Year'))
S2701_18_P_df=S2701_18_P_df[['State:Year',"Male","Female","White","Black or African American","American Indian and Alaska Native",
"Asian","Native Hawaiian and other Pacific Islander","Other","Multiracial","Hispanic or Latino"]]
```

Continue cleaning and transforming demographic tables.

1. Repeat steps 1 – 5 from above, but for table S2701_19_P_df

```
S2701_19_P_df=S2701_19_df.withColumn("Male",S2701_19_df["Male"]/S2701_19_df["Total Population"])
S2701_19_P_df=S2701_19_P_df.withColumn("Female",S2701_19_P_df["Female"]/S2701_19_P_df["Total Population"])
S2701_19_P_df=S2701_19_P_df.withColumn("White",S2701_19_P_df["White"]/S2701_19_P_df["Total Population"])
S2701_19_P_df=S2701_19_P_df.withColumn("Black or African American",S2701_19_P_df["Black or African American"]/S2701_19_P_df["Total Population"])
S2701_19_P_df=S2701_19_P_df.withColumn("American Indian and Alaska Native",S2701_19_P_df["American Indian and Alaska Native"]/S2701_19_P_df["Total Population"])
S2701_19_P_df=S2701_19_P_df.withColumn("Asian",S2701_19_P_df["Asian"]/S2701_19_P_df["Total Population"])
S2701_19_P_df=S2701_19_P_df.withColumn("Native Hawaiian and other Pacific Islander",S2701_19_P_df["Native Hawaiian and other Pacific Islander"]/S2701_19_P_df["Total Population"])
S2701_19_P_df=S2701_19_P_df.withColumn("Other",S2701_19_P_df["Other"]/S2701_19_P_df["Total Population"])
S2701_19_P_df=S2701_19_P_df.withColumn("Multiracial",S2701_19_P_df["Multiracial"]/S2701_19_P_df["Total Population"])
S2701_19_P_df=S2701_19_P_df.withColumn("Hispanic or Latino",S2701_19_P_df["Hispanic or Latino"]/S2701_19_P_df["Total Population"])
S2701_19_P_df=S2701_19_P_df.withColumn("Year", lit(2019))
S2701_19_P_df=S2701_19_P_df.withColumn("State:Year",concat_ws(":","State",'Year'))
S2701_19_P_df=S2701_19_P_df[['State:Year',"Male","Female","White","Black or African American","American Indian and Alaska Native","Asian",
"Native Hawaiian and other Pacific Islander","Other","Multiracial","Hispanic or Latino"]]
```

Joining tables.

1. Join 'Life_Expectancy_clean' with 'State_Abbrs' to get full state names *(line 2)*
2. Create 'State:Year_LE' as for previous tables *(line 3)*

3. Reduce to relevant columns *(line 4)*

```
2  LE_States=Life_Expectancy_clean.join(State_Abbrs,Life_Expectancy_clean["Code"] == State_Abbrs["Code"],"inner")
3  LE_States=LE_States.withColumn("State:Year_LE",concat_ws(":","State",'YEAR'))
4  LE_States=LE_States[['State:Year_LE',"Life Expectancy"]]
```

1. Join previously created DF with Insured_Rates on the State:Year columns (line 8)
2. Change % insured to a decimal value (line 9)
3. Reduce to relevant columns (line 10)

```
8  LE_States_with_Insurance=LE_States.join(Insured_Rates,LE_States["State:Year_LE"]==Insured_Rates["State:Year"],"inner")
9  LE_States_with_Insurance=LE_States_with_Insurance.withColumn("Percent Insured",LE_States_with_Insurance["Percent Insured"]/100.0)
10 LE_States_with_Insurance=LE_States_with_Insurance[['State:Year',"Life Expectancy","Percent Insured"]]
```

1. Combine the 'S1901 DFs' then 'S2701 DFs'
2. Rename columns to avoid future join confusion

```
12 #combine the S1901 DFs and rename columns to avoid future join confusion
13 S1901_ALL_df=S1901_18_df_clean.union(S1901_19_df_clean)
14 S1901_ALL_df=S1901_ALL_df.withColumnRenamed("State:Year","State:Year_to_kill_first")
15
16 #combine the S2701 DFs and rename columns to avoid future join confusion
17 S2701_ALL_df=S2701_18_P_df.union(S2701_19_P_df)
18 S2701_ALL_df=S2701_ALL_df.withColumnRenamed("State:Year","State:Year_to_kill_second")
```

1. Join 'S1901_ALL_df' and 'S2701_ALL_df' on the 'State:Year' columns *(line 21)*
2. Reduce to relevant columns *(line 22 – 23)*

```
20 #Join S1901_ALL_df and S2701_ALL_df on the State:Year columns and reduce to relevant columns
21 All_Census_Tables=S1901_ALL_df.join(S2701_ALL_df,S1901_ALL_df["State:Year_to_kill_first"] == S2701_ALL_df["State:Year_to_kill_second"],"outer")
22 All_Census_Tables_cleaned=All_Census_Tables[['State:Year_to_kill_second',"Male","Female","White","Black or African American",
23 "American Indian and Alaska Native","Asian","Native Hawaiian and other Pacific Islander","Other","Multiracial","Hispanic or Latino","Median Household Income"]]
```

1. Join 'All_Census_Tables' with 'LE_States_with_Insurance' on the 'State:Year' columns *(line 26)*
2. Reduce to relevant columns *(line 27 – 28)*

```
26 LE_All_Data=LE_States_with_Insurance.join(All_Census_Tables_cleaned,LE_States_with_Insurance["State:Year"]==All_Census_Tables_cleaned["State:Year_to_kill_second"],"inner")
27 LE_All_Data_cleaned=LE_All_Data[["Male","Female","White","Black or African American","American Indian and Alaska Native","Asian",
28 "Native Hawaiian and other Pacific Islander","Other","Multiracial","Hispanic or Latino","Median Household Income","Percent Insured","Life Expectancy"]]
```

1. Write cleaned and joined data for the ML model to the SQL server

```
1 #Write cleaned and joined data for the ML model to the SQL server
2 LE_All_Data_cleaned.write.format('jdbc').option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
3     .mode("overwrite") \
4     .option("dbtable", tableLE_All) \
5     .option("user", user) \
6     .option("password", password) \
7     .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
8     .save()
```

MODEL DATA

Make a connection to the database

1. Enter credentials

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

Call the merged table

1. Use the spark.read function to call the table
2. Rename the variable Model_Data

```
Model_Data=spark.read.format("jdbc").option("url", f"jdbc:sqlserver://{server}:1433;databaseName={database};") \
    .option("dbtable", tableLE_All) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
    .load()
```

Convert Model_Data to pandas

1. Import pandas from the library
    a. Import pandas as pd
2. Call Model_Data and convert to pandas
3. Grab the dependant column 'Life Expectancy'
4. Grab the rest of the columns by dropping 'Life Expectancy'

```
LE_All_Data_cleaned_pandas=Model_Data.toPandas()
LE_dep=LE_All_Data_cleaned_pandas["Life Expectancy"]
LE_ind=LE_All_Data_cleaned_pandas.drop(columns={"Life Expectancy"},inplace=False)
```

Set up the model variables

1. Import pandas as pd
2. Add 'from sklearn.preprocessing import StandardScaler'
3. Set ss for the StandardScaler()
4. Set the zscore from the independent columns ss.fit_transform(LE_ind)
5. Normalize the independent data frame

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
zscore = ss.fit_transform(LE_ind)
LE_ind_normalized = pd.DataFrame(zscore, columns=LE_ind.columns.tolist())
```

Set up the training models

1. Add the 'train_test_split' function

2. Set up the training and testing variables with the independent and dependent variables *(line 2)*
3. Add the Linear Regression Function
4. Set up the linear regression variable as 'reg'
5. Test the score for the variables

```
1 from sklearn.model_selection import train_test_split
2 X_train, X_test, y_train, y_test = train_test_split(LE_ind, LE_dep,random_state=42)
3
4 from sklearn.linear_model import LinearRegression
5 reg = LinearRegression().fit(X_train, y_train)
6
7 reg.score(X_test,y_test)
```

Testing the normalized data frame

1. Set up the training and testing variables with the normalized independent and dependent variables *(line 1)*
2. Add the Linear Regression Function
3. Test the score for the normalized variables

```
1 X_train_normal, X_test_normal, y_train_normal, y_test_normal = train_test_split(LE_ind_normalized, LE_dep,random_state=42)
2
3 reg_normal = LinearRegression().fit(X_train_normal, y_train_normal)
4
5 reg_normal.score(X_test_normal,y_test_normal)
```

Get the prediction variables

1. Get the prediction for the standard data set *(line 1)*
2. Get the prediction for the normalized data set *(line 2)*

```
1 predictions=reg.predict(X_test)
2 predictions_normal=reg_normal.predict(X_test_normal)
```

Plug in the data to display a scatter plot

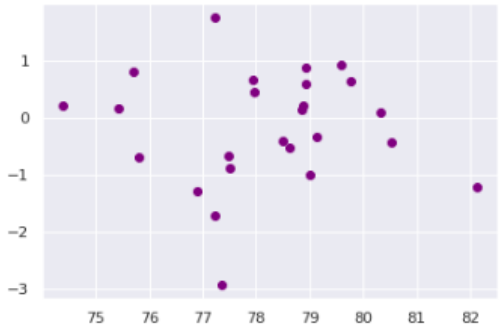
1. Import seaborn as sns; will allow for display customization
2. Import matplotlib.pyplot as plt; will allow to create plot diagrams
3. Import scipy.stats; brings up a library of statstical functions and ability
4. Call sns to set a theme *(line 4)*
5. Plot the scatter plot *(line 6)*
6. Plt.show() displays the diagram

```
import seaborn as sns
import matplotlib.pyplot as plt
import scipy.stats
sns.set_theme(style="darkgrid", palette='pastel')

ax=plt.scatter(predictions,y_test-predictions,c='purple')

plt.show()
```

Similar to above we plot the normalized data set

1. Repeat steps 5 – 6 from above to plot the diagram
    a. Input the correct variables (normalized)

```
ax_normal=plt.scatter(predictions_normal,y_test_normal-predictions_normal,c='purple')

plt.show()
```



Get coefficients

1. Call the coefficient function for the regression line
2. Call the coefficient function for the normalized regression line
3. Print the lists to display the values

```
coefficients_list=reg.coef_
coefficients_normal_list=reg_normal.coef_
print(coefficients_list)
print(coefficients_normal_list)
LE_ind.info()
```

Attempt to gain higher accuracy

1. We use SVR from sklearn.svm library to gain more accurate results
2. Input the training variables *(line2)*
3. Call the variable with the trained variables to test through SVR *(line 3)*
4. Check the score *(line 4)*

```
from sklearn.svm import SVR
svr = SVR().fit(X_train_normal, y_train_normal)
svr_predictions=svr.predict(X_test_normal)
svr.score(X_test_normal,y_test_normal)
```

Setting up the SVR model

1. Import GridSearchCV from sklearn.model_selection
2. Import svm from sklearn
3. Set variable ml from svm.SVR()
4. Set param_grid; acts as a dictionary with parameters names (str) as keys and lists of parameter settings to try as values, or a list of such dictionaries, in which case the grids spanned by each dictionary in the list are explored. This enables searching over any sequence of parameter settings.
5. Set grid to have it search set variables (line 9)
6. Fit the model for grid searching *(line 12)*
7. Print to display grid search the best parameters *(line 13)*

```
from sklearn.model_selection import GridSearchCV
from sklearn import svm
ml = svm.SVR()
param_grid = {'C': [ 5,6,7,8,8.5,9.5,9,10,10.5,11,11.5,12,13,14,15],
              'gamma': [.2,.25,.6,.5,.35,.4,.3],
              'kernel': ['rbf'],
              'epsilon':[.1,.15,.2,.3,.05]}

grid = GridSearchCV(ml, param_grid, refit = True, verbose = 1,cv=5)

# fitting the model for grid search
grid_search=grid.fit(X_train_normal, y_train_normal)
print(grid_search.best_params_)
```

Set SVR variables to a scatter plot

1. Input the svr prediction variables to the scatter plot function *(line1)*

2. Show diagram *(line 3)*

```
1  ax_svr=plt.scatter(svr_predictions1,y_test_normal-svr_predictions1,c='purple')
2
3  plt.show()
```

Restructuring variables

1. Dropping columns and calling a new set *(line 5)*
2. Transforming Median Household Income to format into decimal *(line 7)*
3. Set dependent variable on new data set *(line 8)*
4. Set independent variable on new data set *(line 9)*

```
5  LE_restricted_Data_pandas=LE_All_Data_cleaned_pandas[["Male","White","Black or African American","American Indian and Alaska Native",
6  "Asian","Native Hawaiian and other Pacific Islander","Other","Multiracial","Hispanic or Latino","Median Household Income","Percent Insured","Life Expectancy"]]
7  LE_restricted_Data_pandas["Median Household Income"]=LE_restricted_Data_pandas["Median Household Income"]/100000.0
8  LE_restricted_dep=LE_restricted_Data_pandas["Life Expectancy"]
9  LE_restricted_ind=LE_restricted_Data_pandas.drop(columns={"Life Expectancy"},inplace=False)
```

Set up zscore for new data set

1. Transform the independent variable
2. Normalize the dataframe

Second section of code

3. Set training and testing variables *(line 1)*
4. Input the linear regression to the trained variables *(line 3)*
5. Test restricted linear regression score *(line 5)*

Third section of code

6. Set training and testing variables for normalized data *(line 1)*
7. Input the linear regression to the trained variables for normalized data *(line 3)*
8. Test restricted linear regression score for normalized data *(line 5)*

```
zscore_restricted = ss.fit_transform(LE_restricted_ind)
LE_restricted_ind_normalized = pd.DataFrame(zscore_restricted, columns=LE_restricted_ind.columns.tolist())
```

Command took 0.01 seconds -- by wstearns@dev-10.com at 5/17/2022, 1:39:34 PM on bootcamp

nd 17

```
X_restricted_train, X_restricted_test, y_restricted_train, y_restricted_test = train_test_split(LE_restricted_ind, LE_restricted_dep,random_state=0)

restricted_reg = LinearRegression().fit(X_restricted_train, y_restricted_train)

restricted_reg.score(X_restricted_test,y_restricted_test)
```

Out[17]: 0.7779725688033589

Command took 0.02 seconds -- by wstearns@dev-10.com at 5/17/2022, 1:39:34 PM on bootcamp

nd 18

```
X_restricted_normal_train, X_restricted_normal_test, y_restricted_normal_train, y_restricted_normal_test = train_test_split(LE_restricted_ind_normalized, LE_restricted_dep,random_state=42)

restricted_normal_reg = LinearRegression().fit(X_restricted_normal_train, y_restricted_normal_train)

restricted_normal_reg.score(X_restricted_normal_test,y_restricted_normal_test)
```

Out[18]: 0.7582572816494836

Get new coefficients

4. Call the coefficient function for the regression line
5. Call the coefficient function for the normalized regression line
6. Print the lists to display the values

```
restricted_coefficients_list=restricted_reg.coef_
print(restricted_coefficients_list)
restricted_normal_coefficients_list=restricted_normal_reg.coef_
print(restricted_normal_coefficients_list)
LE_restricted_ind.info()
```

View correlation table

```
LE_restricted_Data_pandas.corr()
```

Out[20]:

	Male	White	Black or African American	American Indian and Alaska Native	Asian	Native Hawaiian and other Pacific Islander	Other	Multiracial	Hispanic or Latino	Median Household Income	Percent Insured	Life Expectancy
Male	1.000000	0.421586	-0.758323	0.542446	-0.036162	0.061100	-0.041662	0.125291	0.020933	0.174721	-0.060589	0.300603
White	0.421586	1.000000	-0.524718	0.012532	-0.725393	-0.604654	-0.301828	-0.631029	-0.235524	-0.265217	0.118463	-0.024882
Black or African American	-0.758323	-0.524718	1.000000	-0.328295	-0.107826	-0.179416	-0.046143	-0.233528	-0.133683	-0.212975	-0.240819	-0.464772
American Indian and Alaska Native	0.542446	0.012532	-0.328295	1.000000	-0.099909	0.003158	0.006392	0.171748	0.137504	-0.025325	-0.326930	-0.075283
Asian	-0.036162	-0.725393	-0.107826	-0.099909	1.000000	0.892455	0.260345	0.892877	0.213100	0.535867	0.244566	0.456682
Native Hawaiian and other Pacific Islander	0.061100	-0.604654	-0.179416	0.003158	0.892455	1.000000	-0.062094	0.948859	0.006673	0.298637	0.161594	0.248467
Other	-0.041662	-0.301828	-0.046143	0.006392	0.260345	-0.062094	1.000000	0.030805	0.850279	0.288388	-0.080866	0.379231
Multiracial	0.125291	-0.631029	-0.233528	0.171748	0.892877	0.948859	0.030805	1.000000	0.086110	0.370191	0.098174	0.282104
Hispanic or Latino	0.020933	-0.235524	-0.133683	0.137504	0.213100	0.006673	0.850279	0.086110	1.000000	0.154219	-0.305603	0.310126
Median Household Income	0.174721	-0.265217	-0.212975	-0.025325	0.535867	0.298637	0.288388	0.370191	0.154219	1.000000	0.387855	0.765060
Percent Insured	-0.060589	0.118463	-0.240819	-0.326930	0.244566	0.161594	-0.080866	0.098174	-0.305603	0.387855	1.000000	0.405851
Life Expectancy	0.300603	-0.024882	-0.464772	-0.075283	0.456682	0.248467	0.379231	0.282104	0.310126	0.765060	0.405851	1.000000

Test the variables under SVR

1. Set up parameters for new data set
2. Set up grid search for the variables
3. Fit the model for grid search

Second block of code

4. Test new variables *(line 2)*

```
svr_restricted_param_grid = {'C': [ 5,6,7,8,8.5,9.5,9,10,10.5,11,11.5,12,13,14,15],
              'gamma': [.2,.25,.6,.5,.35,.4,.3],
              'kernel': ['rbf'],
              'epsilon':[.1,.15,.2,.3,.05]}

svr_restricted_grid = GridSearchCV(ml, svr_restricted_param_grid, refit = True, verbose = 1,cv=5)

# fitting the model for grid search
grid_search_restricted=svr_restricted_grid.fit(X_restricted_normal_train, y_restricted_normal_train)
print(grid_search_restricted.best_params_)
```

```
Fitting 5 folds for each of 525 candidates, totalling 2625 fits
{'C': 15, 'epsilon': 0.2, 'gamma': 0.25, 'kernel': 'rbf'}
```

Command took 12.26 seconds -- by wstearns@dev-10.com at 5/17/2022, 1:39:34 PM on bootcamp

Cmd 22

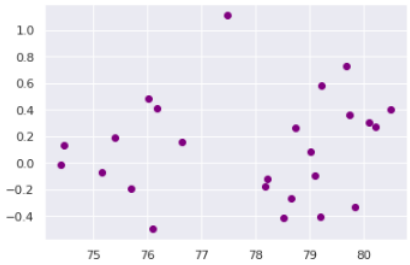
```
svr_restricted_predictions=grid_search_restricted.predict(X_restricted_normal_test)
grid_search_restricted.score(X_restricted_normal_test,y_restricted_normal_test)
```

```
Out[22]: 0.9584617435482025
```

Plot the new model

1. Insert the prediction variables
2. Display the scatter plot

```
ax_svr_restricted=plt.scatter(svr_restricted_predictions,y_restricted_normal_test-svr_restricted_predictions,c='purple')

plt.show()
```



Trying for higher accuracy (3rd attempt)

Modifying the demographic columns in the data frame

```
LE_demographic_Data_pandas=LE_All_Data_cleaned_pandas[["Male","White","Black or African American",
"American Indian and Alaska Native","Asian","Native Hawaiian and other Pacific Islander",
"Other","Multiracial","Hispanic or Latino","Life Expectancy"]]
LE_demographic_dep=LE_demographic_Data_pandas["Life Expectancy"]
LE_demographic_ind=LE_demographic_Data_pandas.drop(columns={"Life Expectancy"},inplace=False)
```

Similar to previous code, input new data set into required fields

```
zscore_demographic = ss.fit_transform(LE_demographic_ind)
LE_demographic_ind_normalized = pd.DataFrame(zscore_demographic, columns=LE_demographic_ind.columns.tolist())

X_demographic_train, X_demographic_test, y_demographic_train, y_demographic_test = train_test_split(LE_demographic_ind, LE_demographic_dep,random_state=42)

demographic_reg = LinearRegression().fit(X_demographic_train, y_demographic_train)

demo_score=demographic_reg.score(X_demographic_test,y_demographic_test)

X_demographic_normal_train, X_demographic_normal_test, y_demographic_normal_train, y_demographic_normal_test = train_test_split(LE_demographic_ind_normalized, LE_demographic_dep,random_state=0)

demographic_normal_reg = LinearRegression().fit(X_demographic_normal_train, y_demographic_normal_train)

normal_demo_score=demographic_normal_reg.score(X_demographic_normal_test,y_demographic_normal_test)
print(demo_score)
print(normal_demo_score)
demographic_coefficients_list=demographic_reg.coef_
print(demographic_coefficients_list)
demographic_normal_coefficients_list=demographic_normal_reg.coef_
print(demographic_normal_coefficients_list)
LE_demographic_ind.info()
```

Set the SVR for the 3rd data set

1. Follow previous steps to similar code, input new data frame

```
svr_demographic_param_grid = {'C': [ 5,6,7,8,8.5,9.5,9,10,10.5,11,11.5,12,13,14,15],
              'gamma': [.2,.25,.6,.5,.35,.4,.3],
              'kernel': ['rbf'],
              'epsilon':[.1,.15,.2,.3,.05]}

svr_demographic_grid = GridSearchCV(ml, svr_demographic_param_grid, refit = True, verbose = 1,cv=5)

# fitting the model for grid search
grid_search_demographic=svr_demographic_grid.fit(X_demographic_normal_train, y_demographic_normal_train)
print(grid_search_demographic.best_params_)
```

```
Fitting 5 folds for each of 525 candidates, totalling 2625 fits
{'C': 15, 'epsilon': 0.1, 'gamma': 0.6, 'kernel': 'rbf'}
```

Command took 12.40 seconds -- by wstearns@dev-10.com at 5/17/2022, 1:39:34 PM on bootcamp

md 27

```
svr_demographic_predictions=grid_search_demographic.predict(X_demographic_normal_test)
grid_search_demographic.score(X_demographic_normal_test,y_demographic_normal_test)
```

```
Out[27]: 0.9672290735486474
```

Final scatter plot with highest accuracy score

```
ax_svr_demo=plt.scatter(svr_demographic_predictions,y_demographic_normal_test-svr_demographic_predictions,c='purple')

plt.show()
```

CONCLUSION

Overall, we can see that there isn't a direct correlation with demographics with life expectancy as we had hypothesized. Instead, we noticed a trend with Median Household Income. What we saw was that the correlation between Median Household Income and Life Expectancy was the highest correlating column. This led us to believe that the higher the income an individual has access to the likely hood of living a longer life is expected. Our ML Model did not have the result we had predicted but we did find another interesting concept to which we would be intrigued into studying further.