

# Simple Data Analysis SQLAlchemy ORM Project

by: Nadiyah Williams

## ✓ A page explaining the Technology and Tools I used throughout my Project

Throughout my project, I used various technologies and libraries to develop my final project, a Jupyter notebook capable of analyzing tweets from airline companies, counting and visualizing the number of negative reviews for each airline, and constructing a model to classify customer tweets into specific groups. One technology that I used was Jupyter Notebook. Unlike a traditional Python script, Jupyter Notebook is extremely versatile by allowing you to write code but also use Markdown for titles and Raw text for sentences and paragraphs. Since my final project consists of me writing a paper and code, I felt as though Jupyter Notebook would be the perfect tool to use. Additionally, when working with Jupyter Notebook, it is easier to collaborate and share your notebook. You even have the option to save it as a PDF, which makes it easier to work on the go.

For my final project, I was tasked with working with a database and using SQLAlchemy Object-Relational Mapping (ORM) to pull data from the database. In the past, I have previously worked with SQLAlchemy Core, which is a lower-level abstraction for database interaction. SQLAlchemy ORM is a more advanced high-level abstraction for database interaction. When using ORM, it allows you to map Python classes to database tables. ORM has an object-oriented approach to database interaction, which makes code easier to read and understand, and also makes it easy to maintain the data. SQLAlchemy ORM was a better choice for my final project because it allowed me to use higher-level abstractions, making my code clearer.

In my project, you would see bar graphs as well as a heatmap. For the bar graph, I created this graph to showcase the count of negative reasons for each airline. In visualizing the count of negative reasons for each airline, I decided to use the Python library Plotly. Plotly is a Python library that is known for its interactive graphs and nice visualizations. I decided to use Plotly because of the hover toolkit. Typically, when I am working with large amounts of data and I

create a graph using Seaborn or Matplotlib, my graph tends to look messy, with all the bars on top of each other, and the count is usually all messed up, not making the graphs visually appealing. The great thing about the hover tool is that instead of the count number being permanently added to the graph, all you have to do is hover over a bar, and you are able to receive the count that way. Not only does this make the graphs more visually appealing, but if someone else were to look at my code, using the interactive graph could possibly make my project more interesting to the person who is viewing my notebook.

Lastly, in my final project, I also constructed a logistic regression model. For this model, I relied on several tools from the Scikit-learn library, including the `train_test_split` and `TfidfVectorizer` tools. The `train_test_split` function helps simplify the task of having to separate my data into test and train data. This could have been completed manually, but using the tool allowed me to save time and also eliminated any user errors in my code. The only con about using the `train_test_split` tool is that sometimes this tool can introduce randomness to your model. I also used the `TfidfVectorizer`, which allowed me to change text data into numbers so that my machine learning algorithm could better understand the data. I used `TfidfVectorizer` over `CountVectorizer` because `TfidfVectorizer` considers both the frequency and inverse frequency. It also prioritizes important words while focusing less on non-important words, which can help with the overall accuracy of the machine learning model.

Another technology I used in my project was GitHub. GitHub is a developer platform that allows creators to store, manage, and collaborate on code. In the beginning of my project, I was tasked with creating a branch from a repository. While working on my project, I created an issue, and I also made regular commits to GitHub. A commit is when you push anything new you have done with your project into Git with a message so that you know what you pushed and so that you can come back later and look at previous commits. GitHub is popular because it allows you to safely store your notebook or script. This is great because if you are working on a project and you accidentally deleted something but you are not sure what you deleted, you can log into GitHub, and Git will tell you the changes made compared to the previous commit. Another reason why Git is a great tool that I used during my project is because it allows for people to review your work and also work together on the same project. There is another platform called Google Colab which allows you to work on notebooks, but Git is a better tool because you can work on notebooks or scripts with others. Also, Google Colab will not be able to tell you what you deleted and show you the difference between your notebook from the past to now like Git can. Overall, GitHub is a powerful tool, and for my project, it was important to use this technology so that my

professor can see my project, but also so that I am able to review any changes I made to my notebook or data.

## ✓ Page Explaining How Docker was created and How I used it

Throughout my project, I had to create a Docker container. Docker is an "open platform for developing, shipping, and running applications," as stated on Docker.com. Docker allows you to manage your project by creating containers, which are a "standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from lightweight, standalone, executable package of software that includes everything needed to run an application code," also stated on Docker.com.

I decided to create my Docker container within my terminal. The steps I took to complete this task were first downloading Docker Desktop onto my Mac. Next, I needed to navigate to my project path. Once I was in my project folder within my terminal, I ran "nano Dockerfile," which allowed me to create a text file called Dockerfile where I added specific information for my Docker and container. Next, I ran "nano docker.compose.yml," which allowed me to create another file needed for my Docker and container. Once those files were created, I ran "docker build," which allowed me to actually build the Docker.

As mentioned in the paragraph above, I created a Dockerfile. This text file holds the foundation of the container, which is Python 3 since that is what I am using to program for my project. Next, in my Dockerfile, I uploaded system dependencies. System dependencies are the tools I will use throughout my project, so in this file, I added SQLite 3 because I knew that I was going to connect to a SQLite database within my notebook. In my project, I knew that I will be using several libraries for modeling, visualizations, cleaning the data, and connecting to the database. Under Python libraries in the text file, I added all the libraries that I will be using such as "Pandas," "Matplotlib," "Seaborn," and others. Next, in my Dockerfile, I added my project files. I knew I would be working on a Jupyter notebook so I added my notebook within the file and then I set my working directory for my notebook. Next, I created a port and made it "8888," which is the direct port for a Jupyter Notebook, allowing me to create my Docker and work in my notebook. Lastly, I set my command to run when my Docker container starts. My container should be able to launch Jupyter with my specific configurations once the Docker is running.

The next file I created was the docker.compose.yml file. This file is used to define and manage

container applications within Docker. In my compose file, I have stated the service configuration which is Jupyter. Next, I built the docker image using my current directory. I specified the port which is "8888," the port used for Jupyter Notebook. Because that is what I am using in my project, that was the only port I needed to define. Next, in my compose file, I mounted two volumes. The first one was my current directory to my notebook allowing the container to have access to the project files. The next volume I defined is database.sqlite which came from the notebooks/database.sqlite host, allowing my container to have access to the database that I will be using throughout my project.

For my project, I created a single container which is built from my Dockerfile and runs the Jupyter Notebook server. My container communicates through the ports I defined internally and through the volumes I have chosen and defined. By creating my container with the python libraries, Python version, and Jupyter notebook, it allows me to work on my project and keep everything consistent and reduce the number of tools I would need to define while working on my project.

There are many ways I can use my Docker. When I first created my Dockerfile in my terminal, I was given two links. Those links when pasted into my web browser URL take me directly to the notebook within my Docker. Another way that I can start my container is by simply opening up Docker Desktop and pushing run so that my Container will start. I also have the option of just typing "docker run" in my terminal; this will start my Docker. Since I was using GitHub in my project and I typically had to be in my branch, the way I ran my Docker is by entering the path for my GitHub branch, starting the docker within Docker Desktop, and by running the command "jupyter notebook" in my terminal which took me to my notebooks within my project folder. Other times if I was just use the URL given to me or work within my visual studios since my Docker was connected to my project folder and visual studios will always automatically start my docker.

## ✓ Explaining How To Run the Container and Different ways to Run a Container From Research

As a data scientist, it's important for us to understand the significance of Docker and the creation of containers within it. In the next page, I will explain the process of Docker creation, as well as provided a explanation into how to run a container. Now, I will go step by step and explain how I executed my container for various scenarios.

During the beginning of my project, I worked within Visual Studio. Visual Studio is an integrated development environment (IDE) created by Microsoft, used for project development. Within Visual Studio, an extension for Docker called "Docker Tools" allowed me to run and debug my container for the final project. This package streamlined the management of my notebook, data, and libraries across different environments. Initially, when creating my Docker, I encountered several issues, but having the Docker Tools extension in Visual Studio allowed me to identify and fix the problem (which was the port mismatch to Jupyter Notebook), ensuring the smooth operation of my Docker and container. Using Docker within Visual Studio made it faster and easier to work on my project.

Running my container while working within Visual Studio was straightforward. Firstly, I opened my final project folder and accessed my notebook named "data\_analysis\_orm." The first time I ran my container, I ensured that I added "Docker Support" in Visual Studio, which integrated my Docker files into the project. Then, I simply clicked "Run Docker Image." This prompt started my Docker Container, allowing me to work on my project. Then I opened Docker Desktop to confirm that my container is running. After starting my Container for the first time the next runs of Docker in Visual Studio became easier; I only had to navigate to my Project and click "Run Docker" within Visual Studio to start my container.

While my Visual Studio environment served me well in the beginning, I noticed the performance started to be slow. I decided to transition to working on my project in Jupyter Notebook. Jupyter Notebook is an open-source web application facilitating the creation and sharing of notebooks, data, and visualizations. Docker containers are integral when working with Jupyter Notebook as they consolidate all project components, simplifying usage and enhancing flexibility. Another advantage of using a Docker container within Jupyter Notebook is the enhanced project reviewability and collaboration.

Running a container within Jupyter Notebook can be accomplished in different ways. I will mention the approaches I used before explaining different ways you can run the Container. The first way I ran my container was a long approach I navigated through my Macbook terminal to access my project directory. Once I was in the directory I ran "Run docker" in the terminal. However, as I became more efficient, I decided to just open Docker Desktop directly. When I created my container, I was provided two URLs. When I paste one of the URLs into my web browser it took me to my final project. This method, being the most efficient, only took around five seconds compared to the earlier, more time-consuming approach.

So far I mentioned three distinct methods of running containers and working within projects.

Now, I'll mention some other approaches I learned from research. First approach is utilizing a Docker API, Jupyter Notebook offers a Docker API wrapper allowing interaction with containers built in Docker. This API facilitates starting, stopping, managing, and monitoring Docker containers via Python code, as detailed on Docker.com. Another option is installing the docker-py library, a Python client for Docker, which enables control of Docker containers and services through Python scripts. As stated on the PyPi.org website under "Docker-py 1.10.6," this library performs all Docker command functions within Python, enhancing container management directly within the notebook environment.

Overall Docker's flexibility allows several unique approaches to starting Docker and running containers. Regardless of the chosen method, each approach allows users to initiate, stop, and execute commands within Docker.

## ✓ Describing Exactly What I have Done in my Notebook

For my final project, I was assigned the task of creating a Python script or notebook along with a Docker container. In the notebook or script, I was required to analyze and visualize data from a relational database using SQLAlchemy Object-Relational Mapping (ORM). This involved retrieving data from database tables, performing basic data analysis such as data filtering or using aggregate functions, and utilizing SQLAlchemy for data retrieval, and Matplotlib or Seaborn for visualizations. I successfully completed this task, and in the following pages, I'll elaborate on how I achieved it.

To complete my final project, I decided to create a Jupyter notebook. Jupyter notebook is an excellent tool for this purpose because it allowed me to incorporate both code and textual explanations which enhanced the presentation of my project. Also, I created a Docker container. I accomplished this by downloading Docker Desktop and creating two essential files: "Dockerfile" and "docker.compose.yml." These files contained important information including libraries, notebook directory, Jupyter notebook ports, and database information.

Throughout my project I decided to work with a SQLite database. The first database I choose was too large to be pushed into GitHub, which was important for my project because my final needed to be pushed into Git. So I decided to find a database on Kaggle.com and found a database containing negative reviews of airline companies, along with tweets from customers about their negative experiences with airlines. Below, I've included a mermaid graph representing

the structure of the database:

erDiagram

```
TWEETS {  
    tweet_id INT (primary key)  
    airline_sentiment VARCHAR  
    airline_sentiment_confidence FLOAT  
    negativereason VARCHAR  
    negativereason_confidence FLOAT  
    airline VARCHAR  
    airline_sentiment_gold VARCHAR  
    name VARCHAR  
    negativereason_gold VARCHAR  
    retweet_count INT  
    text TEXT  
    tweet_coord VARCHAR  
    tweet_created DATETIME  
    tweet_location VARCHAR  
    user_timezone VARCHAR  
}
```

This representation visualizes the structure of the database I utilized.

Having secured my database, my next step was to establish a connection using SQLAlchemy ORM. I began by importing necessary libraries, starting with "create\_engine" from SQLAlchemy for establishing a connection to the SQLite database. Next I imported "sessionmaker" from "sqlalchemy.orm" to create a session class and bind it to the engine that I created earlier. Additionally, I employed "declarative\_base" from "sqlalchemy.ext.declarative" to create a base class for my ORM, enabling the definition of the table structure. Finally, I specified the structure of each column using "Column" from "sqlalchemy." Once connected to the database, I executed a simple query to retrieve all data from the "Tweets" table and stored the results in a pandas dataframe named "airline\_tweets\_df."

In the next pages, I will talk about the data cleaning techniques applied, discuss insights gained from my database and explain the Logistic Regression model I created within my notebook.

When dealing with data as a Data Scientist, it's essential to assume that the data might be messy, even if it appears clean at first glance. My initial step was to check if there were any missing data within my "Tweets" table, and I aimed to determine the count of missing data for each specific column. Initially, I attempted to use Pandas' built-in methods such as "isnull()" and "isna()" but quickly realized they wouldn't work for my case. In my "Tweets" table, missing data was represented as empty strings. To address this, I used a different approach. Instead of relying on Pandas' built-in methods, I opted to create an empty dictionary named "null\_count." Within a for loop, I iterated through each column within the "Tweets" table, and using an expression, I counted each empty string in each column and added the total. I then added the count of missing values to the dictionary. This allowed me to obtain a count of missing data for each column within the table. Next, I iterated over the items within the "null\_count" dictionary to print the column name along with the count of missing data for each column. This revealed that eight columns had no missing values, while seven columns were missing more than 4,000 entries. Although there are various methods to handle missing data, I decided not to fill in the missing data for my project. However, I considered some approaches for categorical and numerical columns, such as using the main response for categorical data or using aggregate functions like average or mean for numerical data. Ultimately, I determined that filling in the missing data was unnecessary for my project's objectives. Once I understood the extent of missing data in each column, I proceeded to answer the question: "What are the negative review counts for each category for each airline?" Utilizing the "negativereason" and "airline" columns from the "Tweets" table, I made a Pandas' "groupby()" function to aggregate and summarize the frequency of negative reasons across all airline companies. This allowed me to see trends in customer dissatisfaction for each negative reason across different airlines. Then, I visualized the data by creating a bar graph for each airline separately using Plotly, an interactive tool supporting hover functionality. This enabled me to identify trends and weaknesses in customer dissatisfaction for each airline. For instance, I discovered that except for Delta, the main reason for negative reviews across airlines was customer service issues. Delta, on the other hand, received a higher count of negative reviews for late flights. This graph provided valuable insights into each airline's weaknesses and the main categories for negative reviews. In conclusion, I learned that customer service issues were a significant concern for most airlines, except for Delta, where late flights were the primary issue driving negative reviews.

In the final phase of my analysis, I decided to make a predictive model to showcase the negative



sentiment reasons and associate customer tweets with those sentiments. To initiate this process, I used the "train\_test\_split" function from the Sklearn.model library to separate my data into training and testing sets, opting for a 50% split to help lower my model robustness.

Next, I utilized the "TfidfVectorizer" to convert my categorical data into numerical features, making it easier for the machine learning model to read the data. This transformation enabled the model to look for important words within each tweet while ignoring the less relevant words. Following the conversion of text into numerical representations, I constructed a logistic regression model and conducted training with an iteration count of 1000 to address convergence issues.

Next I went into fitting the model to the training dataset, I started to generate predictions using the test dataset. Using the trained classifier, I created a classification report to evaluate the model's performance metrics. The report showed an accuracy rate of 63%, indicating that a portion of instances were classified correctly while others were not. Also I was able to see precision scores for each sentiment category which, providing insights into the proportion of correctly predicted instances within each sentiment.

Next I looked into the recall percentages. Recall indicated the proportion of instances with negative sentiments correctly identified and assigned to the correct corresponding group. Additionally, F1-scores were analyzed to understand the model's overall performance. To visually represent the outcomes of the logistic regression model, I turned the classification report in a dataframe and used the seaborn library to generate a heatmap.

Overall, the model showed a 63% accuracy rate, with different performances from each negative sentiment group. Categories such as "Bad Flight" and "Canceled Flight" demonstrated high precision and recall scores, suggesting correct identification of instances related to these categories. On the other hand, categories like "Damaged Luggage" and "Flight Attendant Complaints" exhibited lower precision and recall scores, indicating a challenge in accurately predicting instances for these categories.

Although I did not try to enhance the accuracy score in my project, I researched several strategies that could have been used to achieve improving the accuracy score. First idea was Addressing the issue of empty strings within my categorical columns which could have potentially boost the accuracy score. I also could have expanded the training data to provide the model with more instances to learn from. These are just a couple of examples of strategies that could have enhanced my model performance, and I possibly will use these ideas in the future.

In conclusion I was able to put together a nice developed notebook which shows some data cleaning skills, predictive modeling skills while also showcasing visualizations step by step for each part of my project.

## ✓ Citation Page

"Docker-Py." PyPI, [pypi.org/project/docker-py/](https://pypi.org/project/docker-py/). Accessed 10 May 2024.

"Examples Using the Docker Engine Sdks and Docker API." Docker Documentation, 9 Apr. 2024, [docs.docker.com/engine/api/sdk/examples/](https://docs.docker.com/engine/api/sdk/examples/).

"What Is a Container?" Docker, 26 Mar. 2024, [www.docker.com/resources/what-container/](https://www.docker.com/resources/what-container/).

"What Is the Difference between SQLAlchemy Core and ORM?" GeeksforGeeks, GeeksforGeeks, 21 Mar. 2023, [www.geeksforgeeks.org/what-is-the-difference-between-sqlalchemy-core-and-orm/](https://www.geeksforgeeks.org/what-is-the-difference-between-sqlalchemy-core-and-orm/).

```
#Importing necessary libraries
#Database
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, Float, DateTime
#Cleaning Visuals
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px
#Model
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

# engine to connect to tweets database
engine = create_engine('sqlite:///Dataset/database.sqlite')

# session to interact with tweets database
Sess = sessionmaker(bind=engine)
session = Sess()
```

```

#defining base class for my ORM model
base = declarative_base()


# defining tweet class and explaining the columns within the tweets table
class Tweet(base):
    __tablename__ = 'Tweets'

    tweet_id = Column(Integer, primary_key=True)
    airline_sentiment = Column(String)
    airline_sentiment_confidence = Column(Float)
    negativereason = Column(String)
    negativereason_confidence = Column(Float)
    airline = Column(String)
    airline_sentiment_gold = Column(String)
    name = Column(String)
    negativereason_gold = Column(String)
    retweet_count = Column(Integer)
    text = Column(String)
    tweet_coord = Column(String)
    tweet_created = Column(DateTime)
    tweet_location = Column(String)
    user_timezone = Column(String)

#creating tables from ORM model
base.metadata.create_all(engine)

#commit changes and close my session.
session.commit()
session.close()

```

 /tmp/ipykernel\_1508/3515088453.py:9: MovedIn20Warning: The ``declarative\_base()`` function is deprecated. Use ``sqlalchemy.orm.declarative\_base()`` instead.  
 base = declarative\_base()


```

# Writting SQL query
sql_query = """
    SELECT *
    FROM Tweets
    """

```

```
# Grabbing results from SQL
airline_tweets_df = pd.read_sql(sql_query, engine)

# Printing games data
airline_tweets_df.head(50)
```



	tweet_id	airline_sentiment	airline_sentiment_confidence	negativ
0	567588278875213824	neutral	1.0000	
1	567590027375702016	negative	1.0000	
2	567591480085463040	negative	1.0000	l
3	567592368451248130	negative	1.0000	l
4	567594449874587648	negative	1.0000	Custom
5	567594579310825473	negative	1.0000	
6	567595670463205376	negative	1.0000	l
7	567614049425555457	negative	1.0000	Custom

8	567617081336950784	negative	1.0000	Custom
9	567617486703853568	negative	1.0000	Custom
10	567623209026334720	negative	0.6337	Flight
11	567627253991735296	negative	1.0000	
12	567630296783155203	negative	1.0000	Custom
13	567634106058821632	neutral	1.0000	
14	567643252753694721	neutral	1.0000	
15	567655489119326209	positive	1.0000	
16	567663136082513920	negative	1.0000	Los
17	567663504102940672	negative	1.0000	I
18	567667301067915264	neutral	1.0000	

19	567670985403285504	negative	1.0000	Custom
20	567671602280923136	positive	1.0000	
21	567676400933416960	negative	1.0000	
22	567676626855419904	negative	1.0000	Custom
23	567679487383699456	negative	1.0000	Custom
24	567680108002291712	positive	0.6645	
25	567686758708817921	neutral	0.6890	
26	567686845903826947	neutral	0.6579	
27	567688325276770306	neutral	0.6957	
28	567688411289755648	negative	1.0000	Cance
29	567690417265975296	neutral	0.6739	

29	5676917720070200	neutral	0.0700	
30	567692251954827265	negative	1.0000	Flight C
31	567692504397803520	negative	1.0000	Flight
32	567695310860730369	negative	1.0000	I
33	567696188602712064	negative	1.0000	I
34	567698031081160704	negative	1.0000	Custom
35	567701830805618688	positive	1.0000	
36	567702414157824000	negative	1.0000	Cance
37	567703258425081857	negative	1.0000	I
38	567704339448209409	negative	1.0000	
39	567710245053407232	negative	1.0000	Custom

40	567711860938772480	neutral	1.0000	
41	567712600772050945	negative	0.6716	I
42	567713338873118722	positive	0.6803	
43	567713868747902976	negative	1.0000	Custom
44	567713896627470336	negative	1.0000	Flight C
45	567714192980201472	negative	1.0000	
46	567715682918227970	negative	0.6701	Custom
47	567715917983776772	negative	1.0000	Custom
48	567716378681933825	neutral	1.0000	
49	567716403117957120	negative	1.0000	



## ✓ Cleaning Data

#Counting the amount of null values in each column

```
null_count = {}  
#iterate through each column and count the empty string values  
for c in airline_tweets_df.columns:  
    count = sum(airline_tweets_df[c] == "")  
    null_count[c] = count  
#print the missing values in each column  
print("The number of missing values in each column:")  
for c, count in null_count.items():  
    print(f"{c}: {count}")
```

➞ The number of missing values in each column:

```
tweet_id: 0  
airline_sentiment: 0  
airline_sentiment_confidence: 0  
negativereason: 5403  
negativereason_confidence: 4069  
airline: 0  
airline_sentiment_gold: 14445  
name: 0  
negativereason_gold: 14453  
retweet_count: 0  
text: 0  
tweet_coord: 13478  
tweet_created: 0  
tweet_location: 4687  
user_timezone: 4775
```

## ✓ Looking into Negative sentiments Finding the Overall Main Reason for Negative Reviews

```
# writting sql query
sql_query = """
    SELECT airline_sentiment,airline, negativereason
    FROM Tweets
    WHERE airline_sentiment = 'negative';
    """
```

```
# Grabbing results from SQL
airline_tweets_df = pd.read_sql(sql_query, engine)
```

```
# Printing games data
airline_tweets_df.head(200)
```



	airline_sentiment	airline	negativereason
0	negative	Delta	Can't Tell
1	negative	United	Late Flight
2	negative	United	Late Flight
3	negative	Southwest	Customer Service Issue
4	negative	United	Bad Flight
...	...	...	...
195	negative	Southwest	Late Flight
196	negative	Southwest	Customer Service Issue
197	negative	United	Late Flight
198	negative	Delta	Customer Service Issue
199	negative	Southwest	Customer Service Issue

200 rows x 3 columns

✓ Counting the amount of Bad reviews for each reason based on each airline

```
airline_data = airline_tweets_df.groupby(['airline', 'negativereason']).size().re
```

```
color = 'pink'
for category in airline_data['airline'].unique():
    category_data = airline_data[airline_data['airline'] == category]
    fig = px.bar(category_data, x='negativereason', y='count', title=f'Counts of I
    fig.show()
```










```
# writting sql query
sql_query = """
    SELECT *
    FROM Tweets
    """

# Grabbing results from SQL
airline_tweets_df = pd.read_sql(sql_query, engine)

# Printing games data
airline_tweets_df.head(1)
```



	tweet_id	airline_sentiment	airline_sentiment_confidence	negative
0	567588278875213824	neutral	1.0	

## ✓ Creating a Algorithm to Classify Tweets in Groups

```
#assign the target and feature variable
X = airline_tweets_df['text']
y = airline_tweets_df['negativereason']

#splitting the dataset in test and train sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.5, random

#vectorize the text data transforming text to num
vectorize = TfidfVectorizer()
X_train_v = vectorize.fit_transform(X_train)
X_test_v = vectorize.transform(X_test)

#Training logitsic regression model
classifier = LogisticRegression(max_iter = 1000)
classifier.fit(X_train_v, y_train)

#prediciting the reason for a negative sentiment
y_pred = classifier.predict(X_test_v)
```

```
#printing out classification report
report = classification_report(y_test, y_pred, output_dict=True)
print(classification_report(y_test,y_pred))
```



	precision	recall	f1-score	support
	0.65	0.89	0.75	2730
Bad Flight	0.67	0.10	0.18	281
Can't Tell	0.45	0.14	0.21	555
Cancelled Flight	0.72	0.60	0.66	419
Customer Service Issue	0.56	0.69	0.62	1438
Damaged Luggage	0.00	0.00	0.00	42
Flight Attendant Complaints	0.69	0.10	0.17	231
Flight Booking Problems	0.70	0.05	0.10	260
Late Flight	0.67	0.64	0.65	828
Lost Luggage	0.74	0.53	0.62	374
longlines	0.00	0.00	0.00	85
accuracy			0.63	7243
macro avg	0.53	0.34	0.36	7243
weighted avg	0.62	0.63	0.58	7243

```
/usr/local/lib/python3.12/site-packages/sklearn/metrics/_classification.py:156
Precision is ill-defined and being set to 0.0 in labels with no predicted samples
/usr/local/lib/python3.12/site-packages/sklearn/metrics/_classification.py:156
Precision is ill-defined and being set to 0.0 in labels with no predicted samples
/usr/local/lib/python3.12/site-packages/sklearn/metrics/_classification.py:156
Precision is ill-defined and being set to 0.0 in labels with no predicted samples
/usr/local/lib/python3.12/site-packages/sklearn/metrics/_classification.py:156
Precision is ill-defined and being set to 0.0 in labels with no predicted samples
/usr/local/lib/python3.12/site-packages/sklearn/metrics/_classification.py:156
Precision is ill-defined and being set to 0.0 in labels with no predicted samples
/usr/local/lib/python3.12/site-packages/sklearn/metrics/_classification.py:156
Precision is ill-defined and being set to 0.0 in labels with no predicted samples
```



```
#creating a dataframe from the classification report
report_df = pd.DataFrame(report).transpose()
report_df.drop(columns = ['support'], inplace = True)

#creating a heatmap from classification report.
plt.figure(figsize =(10,8))
sns.heatmap(report_df, annot = True, cmap = 'plasma')
plt.title('Classification Report')
plt.xlabel('Metrics')
plt.ylabel('Categories')
plt.show()
```

