

# Winning Space Race with Data Science

Gennaro Rende  
02/11/2022



# Outline

---

- **Executive Summary**
- **Introduction**
- **Methodology**
- **Results**
- **Conclusion**
- **Appendix**

# Executive Summary

## Summary of Methodologies:

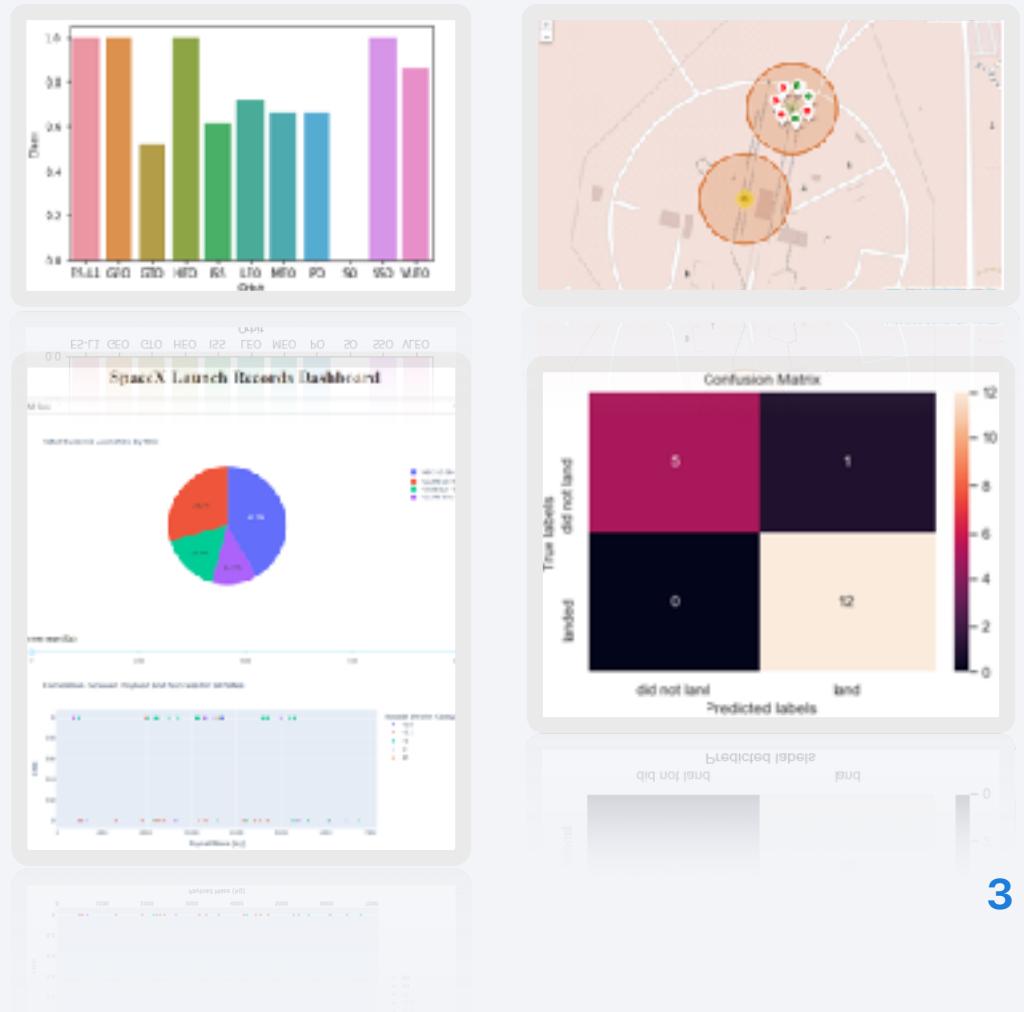
This project follows these steps:

- Data Collection
- Data Wrangling
- Exploratory Data Analysis
- Interactive Visual Analytics
- Predictive Analysis (Classification)

## Summary of Results:

This project produced the following outputs and viz:

1. Exploratory Data Analysis (EDA) results
2. Geospatial analytics
3. Interactive dashboard
4. Predictive analysis of classification models



# Introduction

---

- SpaceX launches Falcon 9 rockets at a cost of around \$62m. This is considerably cheaper than other providers (which usually cost upwards of \$165m), and much of the savings are because SpaceX can land, and then re-use the first stage of the rocket.
- If we can make predictions on whether the first stage will land, we can determine the cost of a launch, and use this information to assess whether or not an alternate company should bid and Space X for a rocket launch.
- This project will ultimately predict if the Space X Falcon 9 first stage will land successfully.



Section 1

# Methodology

# Methodology

---

## 1. Data Collection

- Making GET requests to the SpaceX REST API
- Web Scraping

## 2. Data Wrangling

- Using the `.fillna()` method to remove NaN values
- Using the `.value_counts()` method to determine the following:
  - Number of launches on each site
  - Number and occurrence of each orbit
  - Number and occurrence of mission outcome per orbit type
- Creating a landing outcome label that shows the following:
  - 0 when the booster did not land successfully
  - 1 when the booster did land successfully

## 3. Exploratory Data Analysis

- Using SQL queries to manipulate and evaluate the SpaceX dataset
- Using Pandas and Matplotlib to visualize relationships between variables, and determine patterns

## 4. Interactive Visual Analytics

- Geospatial analytics using Folium
- Creating an interactive dashboard using Plotly Dash

## 5. Data Modelling and Evaluation

- Using Scikit-Learn to:
  - Pre-process (standardize) the data
  - Split the data into training and testing data using `train_test_split`
  - Train different classification models
  - Find hyperparameters using `GridSearchCV`
- Plotting confusion matrices for each classification model
- Assessing the accuracy of each classification model

# Data Collection – SpaceX REST API

[GitHub Link](#)

Using the SpaceX API to retrieve data about launches, including information about the rocket used, payload delivered, launch specifications, landing specifications, and landing outcome.

1

- Make a GET response to the SpaceX REST API
- Convert the response to a .json file then to a Pandas DataFrame

1

```
space_url="https://api.spacexdata.com/v5/launches/post"
```

```
response = requests.get(space_url)
```

```
# Use json_normalize method to convert the json result into a dataframe  
data = pd.json_normalize(response.json())
```

2

- Use custom logic to clean the data (see Appendix)
- Define lists for data to be stored in
- Call custom functions (see Appendix) to retrieve data and fill the lists
- Use these lists as values in a dictionary and construct the dataset

2

```
##Global variables  
BoosterVersion = []  
PayloadMass = []  
Orbit = []  
LaunchSite = []  
Outcome = []  
Flights = []  
GridFlies = []  
Reused = []  
Legs = []  
LandingPad = []  
Block = []  
ReusedCount = []  
Serial = []  
Longitude = []  
Latitude = []
```

```
# Call getBoosterVersion  
getBoosterVersion(data)
```

```
# Call getLaunchSite  
getLaunchSite(data)
```

```
# Call getPayloads  
getPayloads(data)
```

```
# Call getLaunchData  
getLaunchData(data)
```

```
LaunchDict = {} #FlightNumber's list (data['Flight_number']),  
#Date, #Block (data['date']),  
#BoosterVersion (BoosterVersion),  
#PayloadMass (PayloadMass),  
#Orbit (Orbit),  
#LaunchSite (LaunchSite),  
#Outcome (Outcome),  
#Flights (Flights),  
#GridFlies (GridFlies),  
#Reused (Reused),  
#Serial (Serial),  
#Longitude (Longitude),  
#Latitude (Latitude)
```

3

- Create a Pandas DataFrame from the constructed dictionary dataset

3

```
# Create a dict from launch_dict  
df = pd.DataFrame.from_dict(launch_dict)
```

4

- Filter the DataFrame to only include Falcon 9 launches
- Reset the FlightNumber column
- Replace missing values of PayloadMass with the mean PayloadMass value

4

```
data_falcon9 = df[df['BoosterVersion'] == 'Falcon 9']
```

```
data_falcon9['FlightNumber'] = list(range(1, data_falcon9.shape[0]+1))
```

```
# Calculate the mean value of PayloadMass column and replace the missing values with its mean value  
data_falcon9 = data_falcon9.fillna(value={'PayloadMass': data_falcon9['PayloadMass'].mean()})
```

# Data Collection - WebScraping

[GitHub Link](#)

Web scraping to collect Falcon 9 historical launch records from a Wikipedia page titled List of Falcon 9 and Falcon Heavy launches.

- 1
- Request the HTML page from the static URL
  - Assign the response to an object

- 2
- Create a BeautifulSoup object from the HTML response object
  - Find all tables within the HTML page

- 3
- Collect all column header names from the tables found within the HTML page

- 4
- Use the column names as keys in a dictionary
  - Use custom functions and logic to parse all launch tables (see [Appendix](#)) to fill the dictionary values

- 5
- Convert the dictionary to a Pandas DataFrame ready for export

```
1 static_url = "https://en.wikipedia.org/w/index.php?title=List_of_Falcon_9_and_Falcon_Heavy_launches&oldid=1037066912"  
# use requests.get() method with the provided static_url  
response = requests.get(static_url)  
# assign the response to a variable  
data = response.text  
  
2 SDOP = BeautifulSoup(data, 'html5lib')  
html_tables = soup.find_all('table')  
  
3 column_names = []  
  
# apply find_all() function with "th" element on first_launch_table  
# Iterate each th element and apply the provided extract_column_from_header() to get a column name  
# Append the non-empty column name ("if name is not None and len(name) > 0") into a list called column_names  
  
for row in first_launch_table.find_all("th"):  
    name = extract_column_from_header(row)  
    if name is None and len(name) > 0:  
        column_names.append(name)  
  
4 column_dict = {}  
for name in column_names:  
    column_dict[name] = []  
  
Launch_dict = dict.fromkeys(column_names)  
  
# Remove an irrelevant column  
del Launch_dict["Date and time ( )"]  
  
# Let's initial the Launch_dict with each value to be an empty list  
Launch_dict["Flight No."] = []  
Launch_dict["Launch site"] = []  
Launch_dict["Payload"] = []  
Launch_dict["Payload mass"] = []  
Launch_dict["Orbit"] = []  
Launch_dict["Customer"] = []  
Launch_dict["Launch outcome"] = []  
# Added some new columns  
Launch_dict["Version Booster"] = []  
Launch_dict["Hopper Landing"] = []  
Launch_dict["Date"] = []  
Launch_dict["Time"] = []  
  
5 Launch_dict["Date"] = []  
Launch_dict["Time"] = []  
Launch_dict["DF"] = pd.DataFrame(Launch_dict)  
Launch_dict["AllData"] = []
```

# Data Wrangling - Pandas

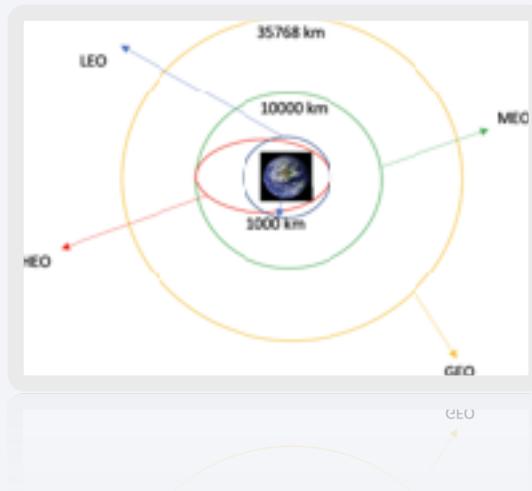
[GitHub Link](#)

## Context:

- The SpaceX dataset contains several Space X launch facilities, and each location is in the LaunchSite column.
- Each launch aims to a dedicated orbit, and some of the common orbit types are shown in the figure below. The orbit type is in the Orbit column.

## Initial Data Exploration:

- Using the .value\_counts() method to determine the following:
  - Number of launches on each site
  - Number and occurrence of each orbit
  - Number and occurrence of landing outcome per orbit type



1  
# Apply .value\_counts() on column launchSite  
df['LaunchSite'].value\_counts()

LaunchSite	Count
CCAFS SLC 40	55
KSC SLC 39A	22
VAFB SLC 4E	13

Name: LaunchSite, dtype: int64

2  
# Apply .value\_counts() on Orbit column  
df['Orbit'].value\_counts()

Orbit	Count
GTO	27
ISS	21
LEO	14
PO	9
MEO	7
SSO	5
HEO	3
ES-L1	2
GEO	1
S0	1
ILO	1

Name: Orbit, dtype: int64

3  
# Landing\_outcomes = values in Outcome column  
landing\_outcomes = df['Outcome'].value\_counts()  
landing\_outcomes

Outcome	Count
True AOGS	61
None None	19
True RTLS	14
False AOGS	6
None RTLS	5
None AOGS	2
False Outcome	2
False RTLS	1

Name: Outcome, dtype: int64

None: Outcomes, dtype: int64

# Data Wrangling - Pandas

[GitHub Link](#)

## Context:

- The landing outcome is shown in the Outcome column:
  - True Ocean – the mission outcome was successfully landed to a specific region of the ocean
  - False Ocean – the mission outcome was unsuccessfully landed to a specific region of the ocean.
  - True RTLS – the mission outcome was successfully landed to a ground pad
  - False RTLS – the mission outcome was unsuccessfully landed to a ground pad.
  - True ASDS – the mission outcome was successfully landed to a drone ship
  - False ASDS – the mission outcome was unsuccessfully landed to a drone ship.
  - None ASDS and None None – these represent a failure to land.

## Data Wrangling:

- To determine whether a booster will successfully land, it is best to have a binary column, i.e., where the value is 1 or 0, representing the success of the landing.
- This is done by:
  - Defining a set of unsuccessful (bad) outcomes, bad\_outcome
  - Creating a list, landing\_class, where the element is 0 if the corresponding row in Outcome is in the set bad\_outcome, otherwise, it's 1.
  - Create a Class column that contains the values from the list landing\_class
  - Export the DataFrame as a .csv file.

1

```
bad_outcomes=set(landing_outcomes.keys()|[1,3,5,6,7])  
bad_outcomes  
{'False ASDS', 'False Ocean', 'False RTLS', 'None ASDS', 'None None'}
```

3

```
# Landing class = 0 if bad outcome  
# Landing_class = 1 otherwise  
  
landing_class = []  
  
for outcome in df['Outcome']:  
    if outcome in bad_outcomes:  
        landing_class.append(0)  
    else:  
        landing_class.append(1)
```

3

```
df['Class']=landing_class
```

4

```
df.to_csv("dataset part\ 2.csv", index=False)
```

0

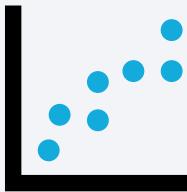
# EDA with Data Visualization

---

## SCATTER CHARTS

Scatter charts were produced to visualize the relationships between:

- Flight Number and Launch Site
- Payload and Launch Site
- Orbit Type and Flight Number
- Payload and Orbit Type



Scatter charts are useful to observe relationships, or correlations, between two numeric variables.

## BAR CHARTS

A bar chart was produced to visualize the relationship between:

- Success Rate and Orbit Type



Bar charts are used to compare a numerical value to a categorical variable. Horizontal or vertical bar charts can be used, depending on the size of the data.

## LINE CHARTS

Line charts were produced to visualize the relationships between:

- Success Rate and Year (i.e. the launch success yearly trend)



Line charts contain numerical values on both axes, and are generally used to show the change of a variable over time.

---

To gather some information about the dataset, some SQL queries were performed.

The SQL queries performed on the data set were used to:

1. Display the names of the unique launch sites in the space mission
2. Display 5 records where launch sites begin with the string 'CCA'
3. Display the total payload mass carried by boosters launched by NASA (CRS)
4. Display the average payload mass carried by booster version F9 v1.1
5. List the date when the first successful landing outcome on a ground pad was achieved
6. List the names of the boosters which had success on a drone ship and a payload mass between 4000 and 6000 kg
7. List the total number of successful and failed mission outcomes
8. List the names of the booster versions which have carried the maximum payload mass
9. List the failed landing outcomes on drone ships, their booster versions, and launch site names for 2015
10. Rank the count of landing outcomes (such as Failure (drone ship) or Success (ground pad)) between the date 2010-06-04 and 2017-03-20, in descending order

# Build an Interactive Map with Folium

[GitHub Link](#)

---

The following steps were taken to visualize the launch data on an interactive map:

## 1. Mark all launch sites on a map

- Initialise the map using a Folium Map object
- Add a folium.Circle and folium.Marker for each launch site on the launch map

## 2. Mark the success/failed launches for each site on a map

- As many launches have the same coordinates, it makes sense to cluster them together.
- Before clustering them, assign a marker colour of successful (class = 1) as green, and failed (class = 0) as red.
- To put the launches into clusters, for each launch, add a folium.Marker to the MarkerCluster() object.
- Create an icon as a text label, assigning the icon\_color as the marker\_colour determined previously.

## 3. Calculate the distances between a launch site to its proximities

- To explore the proximities of launch sites, calculations of distances between points can be made using the Lat and Long values.
- After marking a point using the Lat and Long values, create a folium.Marker object to show the distance.
- To display the distance line between two points, draw a folium.PolyLine and add this to the map.

# Build a Dashboard with Plotly Dash

[GitHub Link](#)

---

The following plots were added to a Plotly Dash dashboard to have an interactive visualisation of the data:

1. Pie chart (px.pie()) showing the total successful launches per site
  - This makes it clear to see which sites are most successful
  - The chart could also be filtered (using a dcc.Dropdown() object) to see the success/failure ratio for an individual site
  
2. Scatter graph (px.scatter()) to show the correlation between outcome (success or not) and payload mass (kg)
  - This could be filtered (using a RangeSlider() object) by ranges of payload masses
  - It could also be filtered by booster version

# Predictive Analysis (Classification)

[GitHub Link](#)

## Model Development



- To prepare the dataset for model development:
  - Load dataset
  - Perform necessary data transformations (standardise and pre-process)
  - Split data into training and test data sets, using `train_test_split()`
  - Decide which type of machine learning algorithms are most appropriate
- For each chosen algorithm:
  - Create a `GridSearchCV` object and a dictionary of parameters
  - Fit the object to the parameters
  - Use the training data set to train the model

## Model Evaluation



- For each chosen algorithm:
  - Using the output `GridSearchCV` object:
    - Check the tuned hyperparameters (`best_params_`)
    - Check the accuracy (`score` and `best_score_`)
  - Plot and examine the Confusion Matrix

## Finding the Best Classification Model

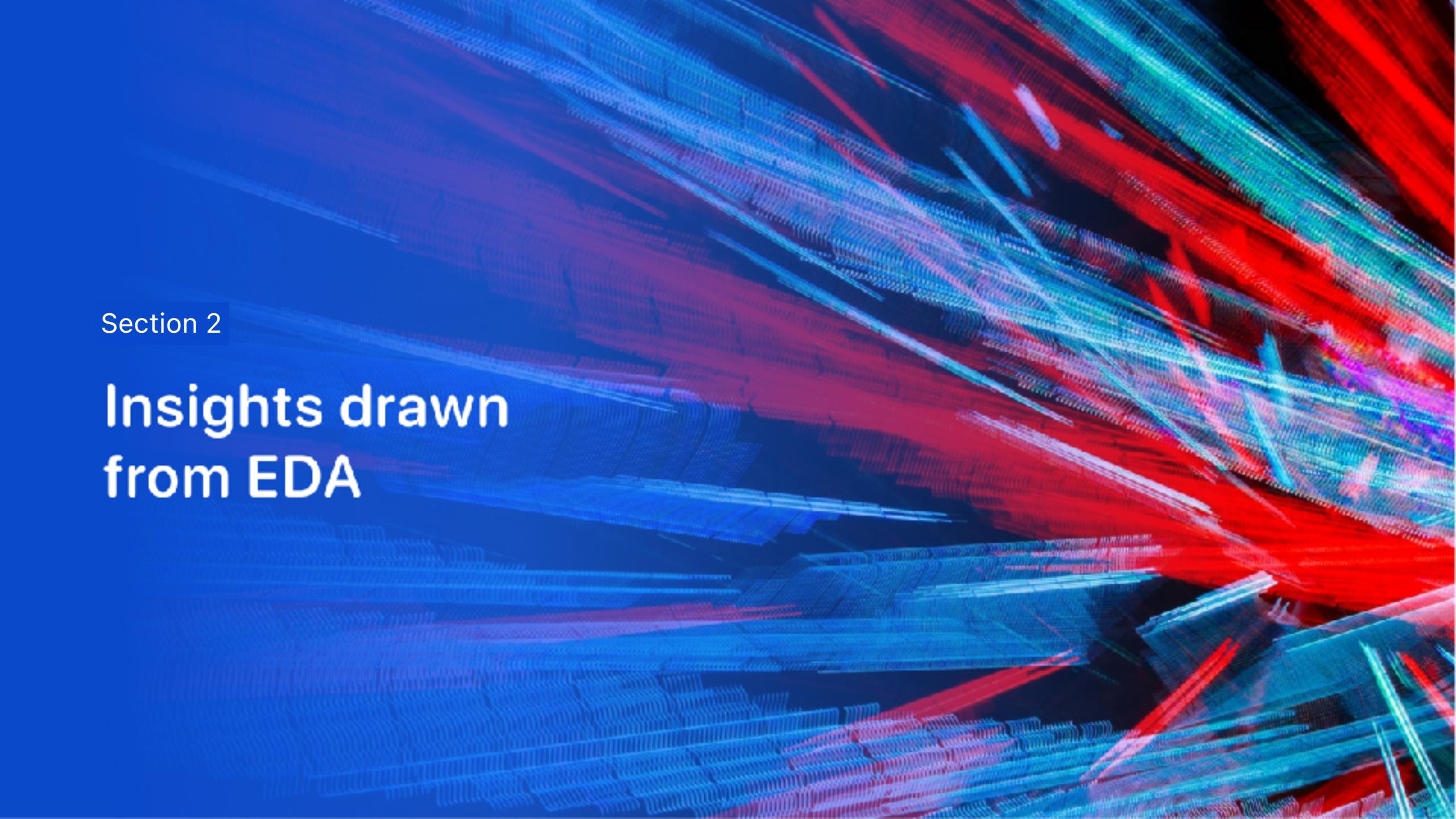


- Review the accuracy scores for all chosen algorithms
- The model with the highest accuracy score is determined as the best performing model

# Results

---

- Exploratory data analysis results
- Interactive analytics demo in screenshots
- Predictive analysis results

The background of the slide features a complex, abstract grid pattern composed of numerous small, glowing particles. The colors of these particles are primarily shades of blue, red, and green, creating a sense of depth and motion. The grid is more dense in the foreground and becomes more sparse towards the background, where it appears as a series of glowing streaks.

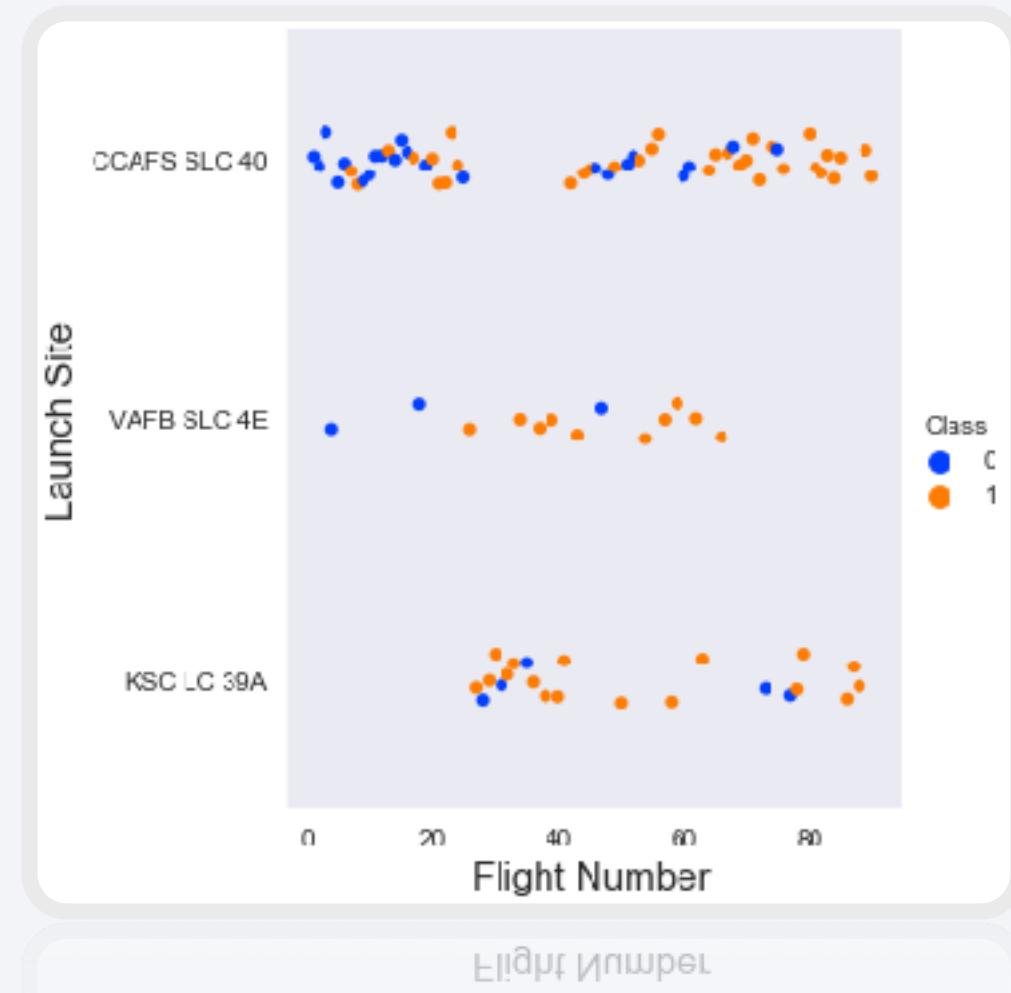
Section 2

## Insights drawn from EDA

# Flight Number vs. Launch Site

The scatter plot of Launch Site vs. Flight Number shows that:

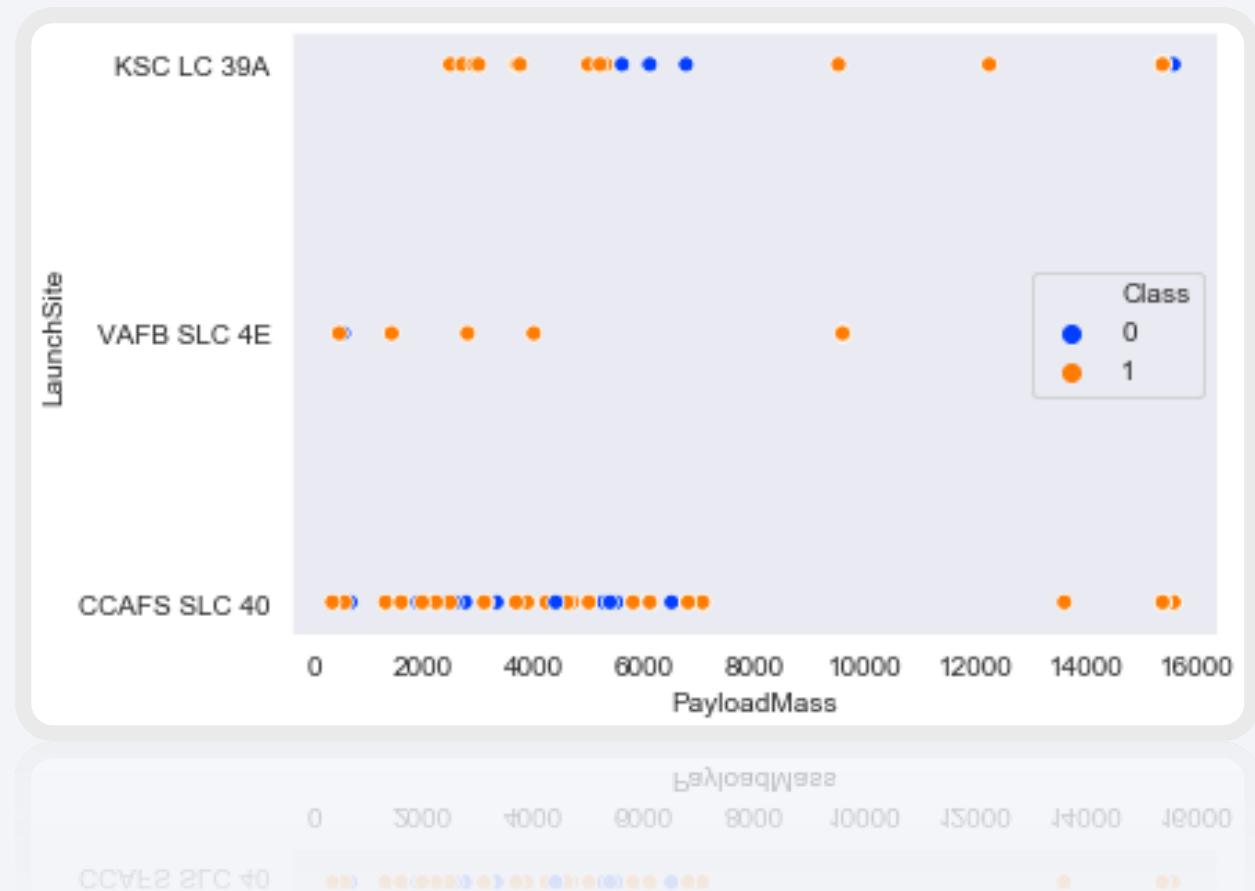
- As the number of flights increases, the rate of success at a launch site increases.
- Most of the early flights (flight numbers < 30) were launched from CCAFS SLC 40, and were generally unsuccessful.
- The flights from VAFB SLC 4E also show this trend, that earlier flights were less successful.
- No early flights were launched from KSC LC 39A, so the launches from this site are more successful.
- Above a flight number of around 30, there are significantly more successful landings (Class = 1).



# Payload vs. Launch Site

The scatter plot of Launch Site vs. Payload Mass shows that:

- Above a payload mass of around 7000 kg, there are very few unsuccessful landings, but there is also far less data for these heavier launches.
- There is no clear correlation between payload mass and success rate for a given launch site.
- All sites launched a variety of payload masses, with most of the launches from CCAFS SLC 40 being comparatively lighter payloads (with some outliers).



# Success Rate vs. Orbit Type

The bar chart of Success Rate vs. Orbit Type shows that the following orbits have the highest (100%) success rate:

- ES-L1 (Earth-Sun First Lagrangian Point)
- GEO (Geostationary Orbit)
- HEO (High Earth Orbit)
- SSO (Sun-synchronous Orbit)

The orbit with the lowest (0%) success rate is:

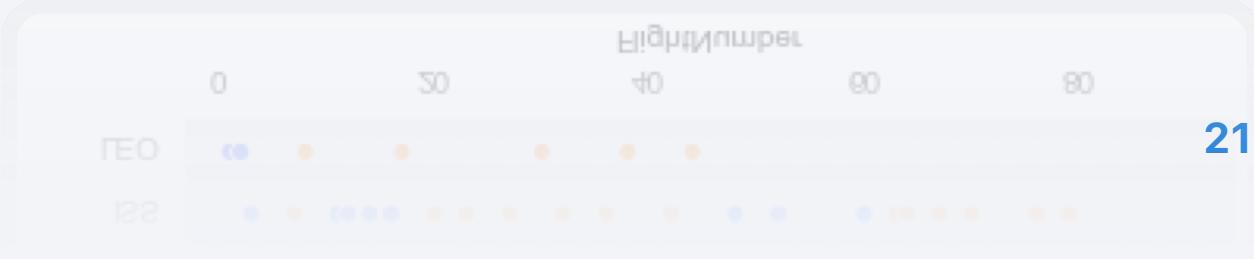
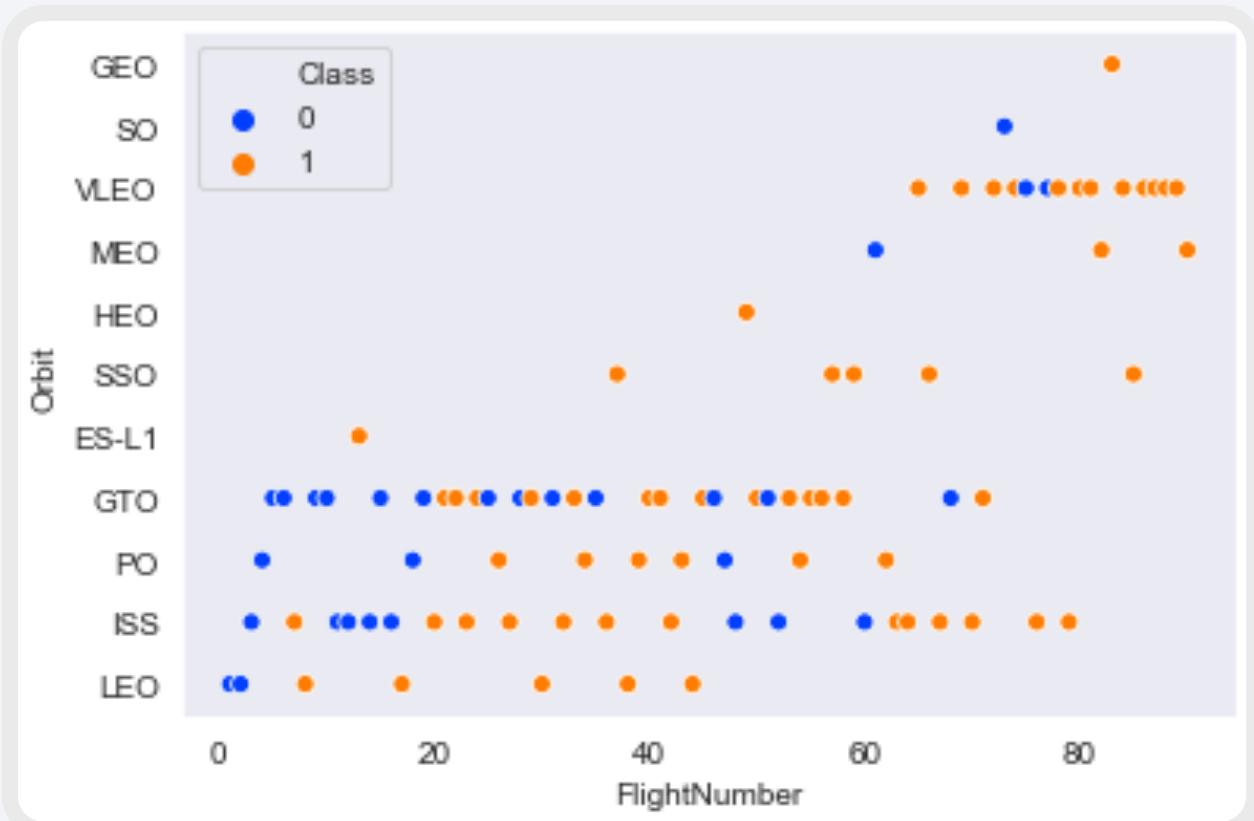
- SO (Heliocentric Orbit)



# Flight Number vs. Orbit Type

This scatter plot of Orbit Type vs. Flight number shows a few useful things that the previous plots did not, such as:

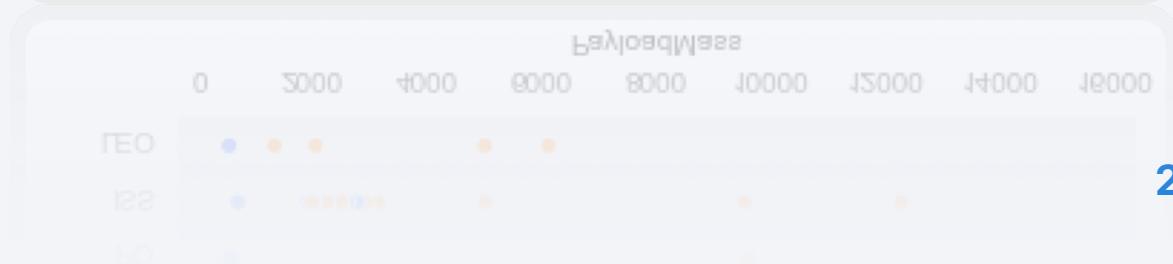
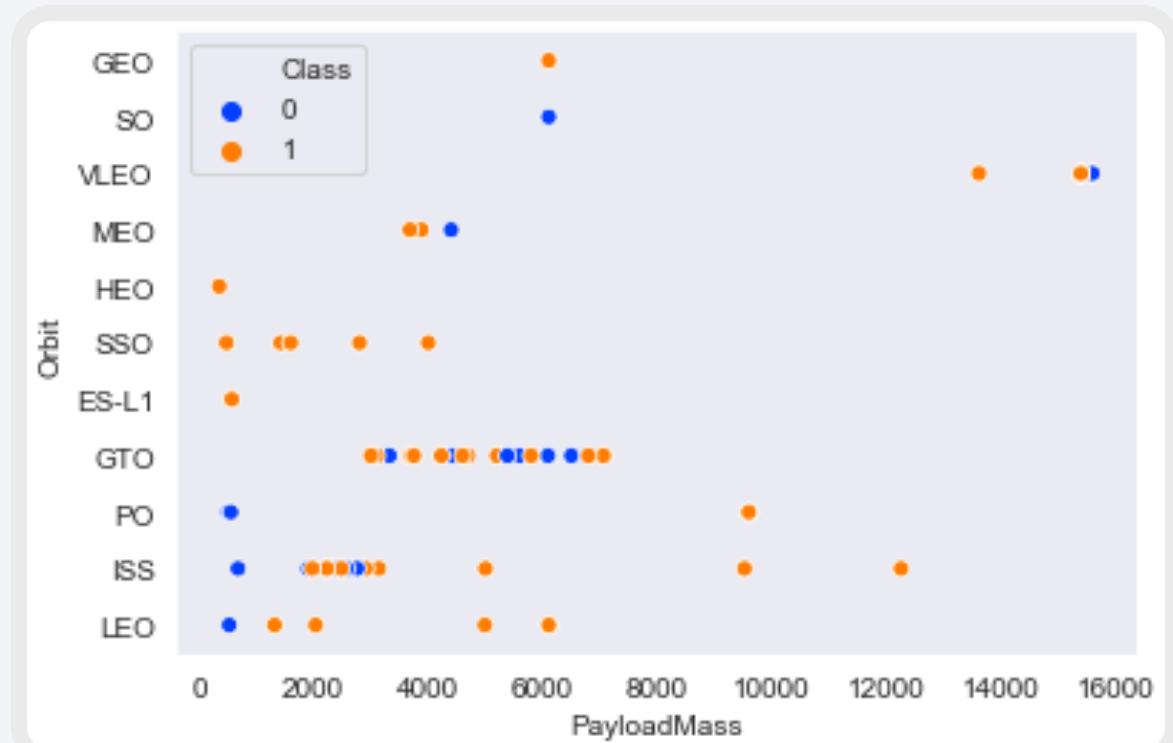
- The 100% success rate of GEO, HEO, and ES-L1 orbits can be explained by only having 1 flight into the respective orbits.
- The 100% success rate in SSO is more impressive, with 5 successful flights.
- There is little relationship between Flight Number and Success Rate for GTO.
- Generally, as Flight Number increases, the success rate increases. This is most extreme for LEO, where unsuccessful landings only occurred for the low flight numbers (early launches).



# Payload vs. Orbit Type

This scatter plot of Orbit Type vs. Payload Mass shows that:

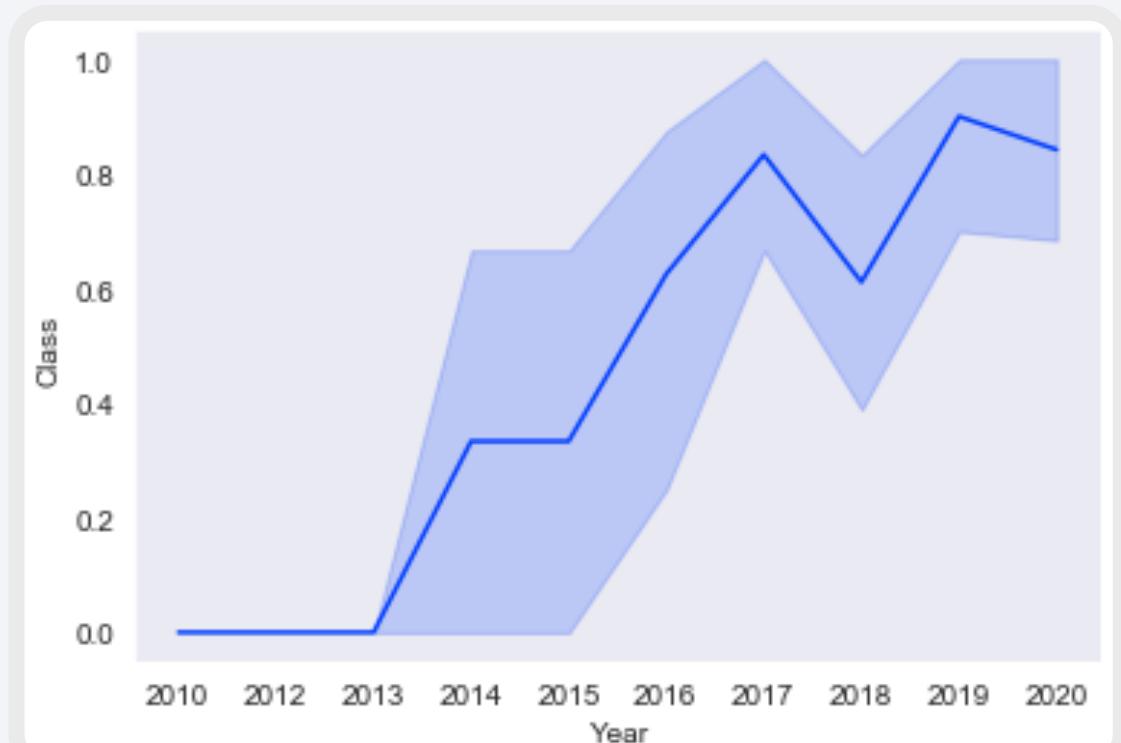
- The following orbit types have more success with heavy payloads:
  - PO (although the number of data points is small)
  - ISS
  - LEO
- For GTO, the relationship between payload mass and success rate is unclear.
- VLEO (Very Low Earth Orbit) launches are associated with heavier payloads, which makes intuitive sense.



# Launch Success Yearly Trend

The line chart of yearly average success rate shows that:

- Between 2010 and 2013, all landings were unsuccessful (as the success rate is 0).
- After 2013, the success rate generally increased, despite small dips in 2018 and 2020.
- After 2016, there was always a greater than 50% chance of success.

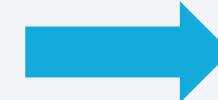


# All Launch Site Names

---

Find the names of the unique launch sites.

```
1 %sql SELECT UNIQUE(LAUNCH_SITE) FROM SPACEXTBL;
```



launch_site
CCAFS LC-40
CCAFS SLC-40
KSC LC-39A
VAFB SLC-4E

The word UNIQUE returns only unique values from the LAUNCH\_SITE column of the SPACEXTBL table.

# Launch Site Names Begin with 'CCA'

Find 5 records where launch sites begin with 'CCA'.

```
1 %sql SELECT LAUNCH_SITE FROM SPACEXTBL WHERE LAUNCH_SITE LIKE 'CCA%' LIMIT 5;
```

A diagram illustrating the execution of a SQL query. On the left, a dark grey rectangular box contains a command-line interface (CLI) session with three colored dots (red, yellow, green) above it. The CLI shows a single line of code: "%sql SELECT LAUNCH\_SITE FROM SPACEXTBL WHERE LAUNCH\_SITE LIKE 'CCA%' LIMIT 5;". A large blue arrow points from this box to the right, leading into a light blue rounded rectangle. Inside this rectangle is a table with a single column labeled "launch\_site". The table contains five identical rows, each showing the value "CCAFS LC-40".

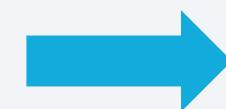
launch_site
CCAFS LC-40

LIMIT 5 fetches only 5 records, and the LIKE keyword is used with the wild card 'CCA%' to retrieve string values beginning with 'CCA'.

# Total Payload Mass

Calculate the total payload carried by boosters from NASA.

```
● ● ●  
1 %sql SELECT SUM(PAYLOAD_MASS__KG_) AS TOTAL_PAYLOAD_MASS FROM SPACEXTBL \  
2 WHERE CUSTOMER = 'NASA (CRS)';
```



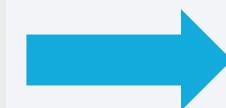
total_payload_mass
45596

The SUM keyword is used to calculate the total of the LAUNCH column, and the SUM keyword (and the associated condition) filters the results to only boosters from NASA (CRS).

# Average Payload Mass by F9 v1.1

Calculate the average payload mass carried by booster version F9 v1.1.

```
● ● ●  
1 %sql SELECT AVG(PAYLOAD_MASS_KG_) AS AVERAGE_PAYLOAD_MASS FROM SPACEXTBL \\  
2 WHERE BOOSTER_VERSION = 'F9 v1.1';
```



average_payload_mass
2928

The AVG keyword is used to calculate the average of the PAYLOAD\_MASS\_KG\_ column, and the WHERE keyword (and the associated condition) filters the results to only the F9 v1.1 booster version.

# First Successful Ground Landing Date

Find the dates of the first successful landing outcome on ground pad.

```
1 %sql SELECT MIN(DATE) AS FIRST_SUCCESSFUL_GROUND_LANDING FROM SPACEXTBL \
2 WHERE LANDING_OUTCOME = 'Success (ground pad)';
```

first successful ground landing  
2015-12-22

The MIN keyword is used to calculate the minimum of the DATE column, i.e. the first date, and the WHERE keyword (and the associated condition) filters the results to only the successful ground pad landings.

## Successful Drone Ship Landing with Payload between 4000 and 6000

List the names of boosters which have successfully landed on drone ship and had payload mass greater than 4000 but less than 6000.

```
1 XSQL SELECT BOOSTER_VERSION FROM SPACEXTBL \n2 WHERE (LANDING_OUTCOME = 'Success (drone ship)') AND (PAYLOAD_MASS_KG_ BETWEEN 4000 AND 6000);
```

A diagram illustrating the execution of a SQL query. On the left, a dark grey rectangular box contains the query code. An orange arrow points from this box to a light blue rounded rectangle on the right. This light blue rectangle contains a table with four rows, each representing a booster version that met the specified criteria.

booster_version
F9 FT B1022
F9 FT B1026
F9 FT B1021.2
F9 FT B1031.2

The WHERE keyword is used to filter the results to include only those that satisfy both conditions in the brackets (as the AND keyword is also used). The BETWEEN keyword allows for  $4000 < x < 6000$  values to be selected.

# Total Number of Successful and Failure Mission Outcomes

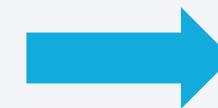
Calculate the total number of successful and failure mission outcome.

A screenshot of a terminal window. The command entered is:

```
i Xsql: SELECT MISSION_OUTCOME, COUNT(MISSION_OUTCOME) AS TOTAL_NUMBER FROM SPACEXTBL GROUP BY MISSION_OUTCOME;
```

The output shows the total number of missions for each outcome category:

mission_outcome	total_number
Failure (in flight)	1
Success	99
Success (payload status unclear)	1



A screenshot of a terminal window. The command entered is:

```
i Xsql: SELECT MISSION_OUTCOME, COUNT(MISSION_OUTCOME) AS TOTAL_NUMBER FROM SPACEXTBL GROUP BY MISSION_OUTCOME;
```

The output shows the total number of missions for each outcome category:

mission_outcome	total_number
Failure (in flight)	1
Success	99
Success (payload status unclear)	1

Below the table, there is a message: "Total number of successful missions is 99."

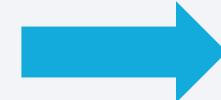
The COUNT keyword is used to calculate the total number of mission outcomes, and the GROUPBY keyword is also used to group these results by the type of mission outcome.

# Boosters Carried Maximum Payload

List the names of the booster which have carried the maximum payload mass.



```
1 %sql SELECT DISTINCT(BOOSTER_VERSION) FROM SPACEXTBL \
2 WHERE PAYLOAD_MASS_KG_ = (SELECT MAX(PAYLOAD_MASS_KG_) FROM SPACEXTBL);
```



booster version

F9 B5 B1048.4

F9 B5 B1048.5

F9 B5 B1049.4

F9 B5 B1049.5

F9 B5 B1049.7

F9 B5 B1051.3

F9 B5 B1051.4

F9 B5 B1051.6

F9 B5 B1056.4

F9 B5 B1058.3

F9 B5 B1060.2

F9 B5 B1060.3

F9 B2 B1020.3

F9 B2 B1020.5

F9 B2 B1028.3

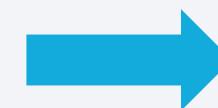
31

A subquery is used here. The SELECT statement within the brackets finds the maximum payload, and this value is used in the WHERE condition. The DISTINCT keyword is then used to retrieve only distinct /unique booster versions.

# 2015 Launch Records

List the failed landing\_outcomes in drone ship, their booster versions, and launch site names for in year 2015.

```
● ● ●  
1 %sql SELECT BOOSTER_VERSION, LAUNCH_SITE FROM SPACEXTBL \n2 WHERE (LANDING_OUTCOME = 'Failure (drone ship)') AND (EXTRACT(YEAR FROM DATE) = '2015');
```



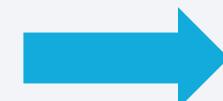
booster_version	launch_site
F9 v1.1 B1012	CCAFS LC-40
F9 v1.1 B1015	CCAFS LC-40

The WHERE keyword is used to filter the results for only failed landing outcomes, AND only for the year of 2015.

## Rank Landing Outcomes Between 2010-06-04 and 2017-03-20

Rank the count of landing outcomes (such as Failure (drone ship) or Success (ground pad)) between the date 2010-06-04 and 2017-03-20, in descending order.

```
● ● ●  
1 %sql SELECT LANDING_OUTCOME, COUNT(LANDING_OUTCOME) AS TOTAL_NUMBER FROM SPACEXTBL \  
2 WHERE DATE BETWEEN '2010-06-04' AND '2017-03-20' \  
3 GROUP BY LANDING_OUTCOME \  
4 ORDER BY TOTAL_NUMBER DESC;
```



landing_outcome	total_number
No attempt	10
Failure (drone ship)	5
Success (drone ship)	5
Controlled (ocean)	3
Success (ground pad)	3
Failure (parachute)	2
Uncontrolled (ocean)	2
Preduded (drone ship)	1

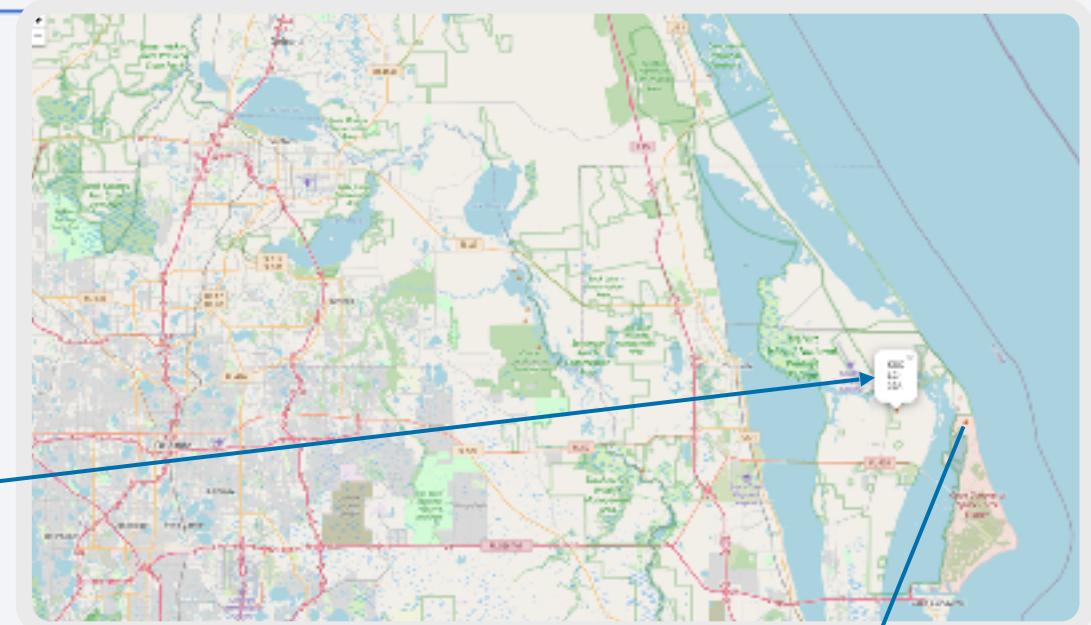
The WHERE keyword is used with the BETWEEN keyword to filter the results to dates only within those specified. The results are then grouped and ordered, using the keywords GROUP BY and ORDER BY, respectively, where DESC is used to specify the descending order.

The background of the slide is a photograph taken from space at night. It shows the curvature of the Earth's horizon against a dark blue sky. City lights are visible as numerous small yellow and white dots, primarily concentrated in the lower right quadrant where a large urban area is shown. In the upper left quadrant, the green and purple glow of the aurora borealis is visible.

Section 3

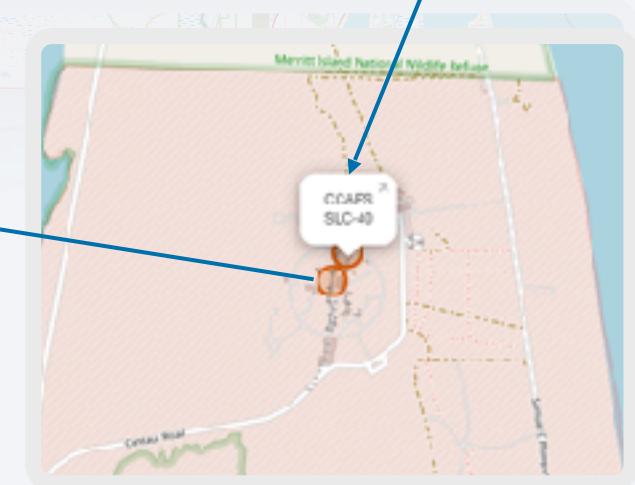
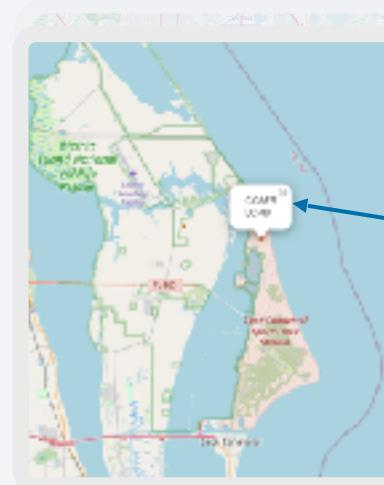
# Launch Sites Proximities Analysis

# ALL LAUNCH SITES ON A MAP



All SpaceX launch sites are on coasts of the United States of America, specifically Florida and California.

- 

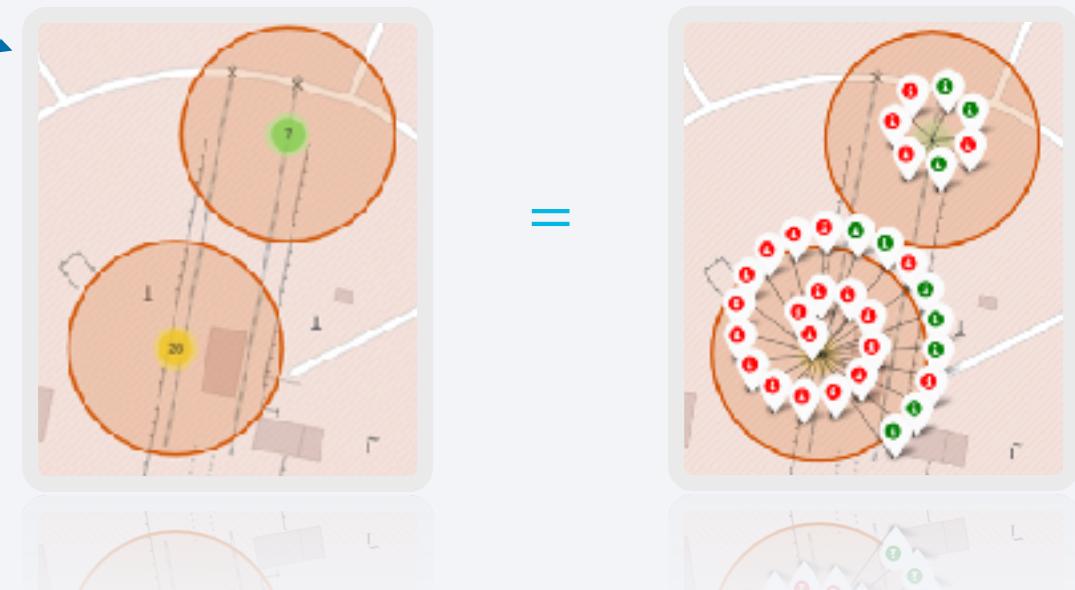


# SUCCESS/FAILED LAUNCHES FOR EACH SITE



- Launches have been grouped into clusters, and annotated with **green icons** for successful launches, and **red icons** for failed launches.

CCAFS SLC-40 and CCAFS LC-40



# PROXIMITY OF LAUNCH SITES TO OTHER POINTS OF INTEREST

Using the CCAFS SLC-40 launch site as an example site, we can understand more about the placement of launch sites.



Are launch sites in close proximity to railways?

- YES. The coastline is only 0.87 km due East.

Are launch sites in close proximity to highways?

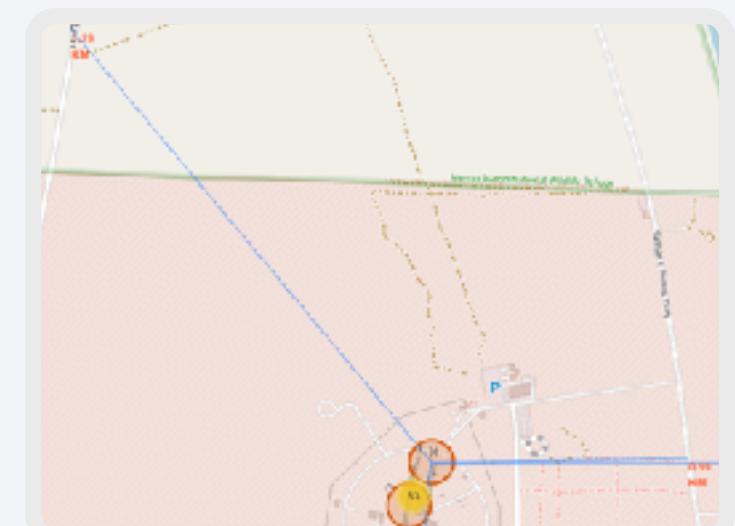
- YES. The nearest highway is only 0.59km away.

Are launch sites in close proximity to railways?

- YES. The nearest railway is only 1.29 km away.

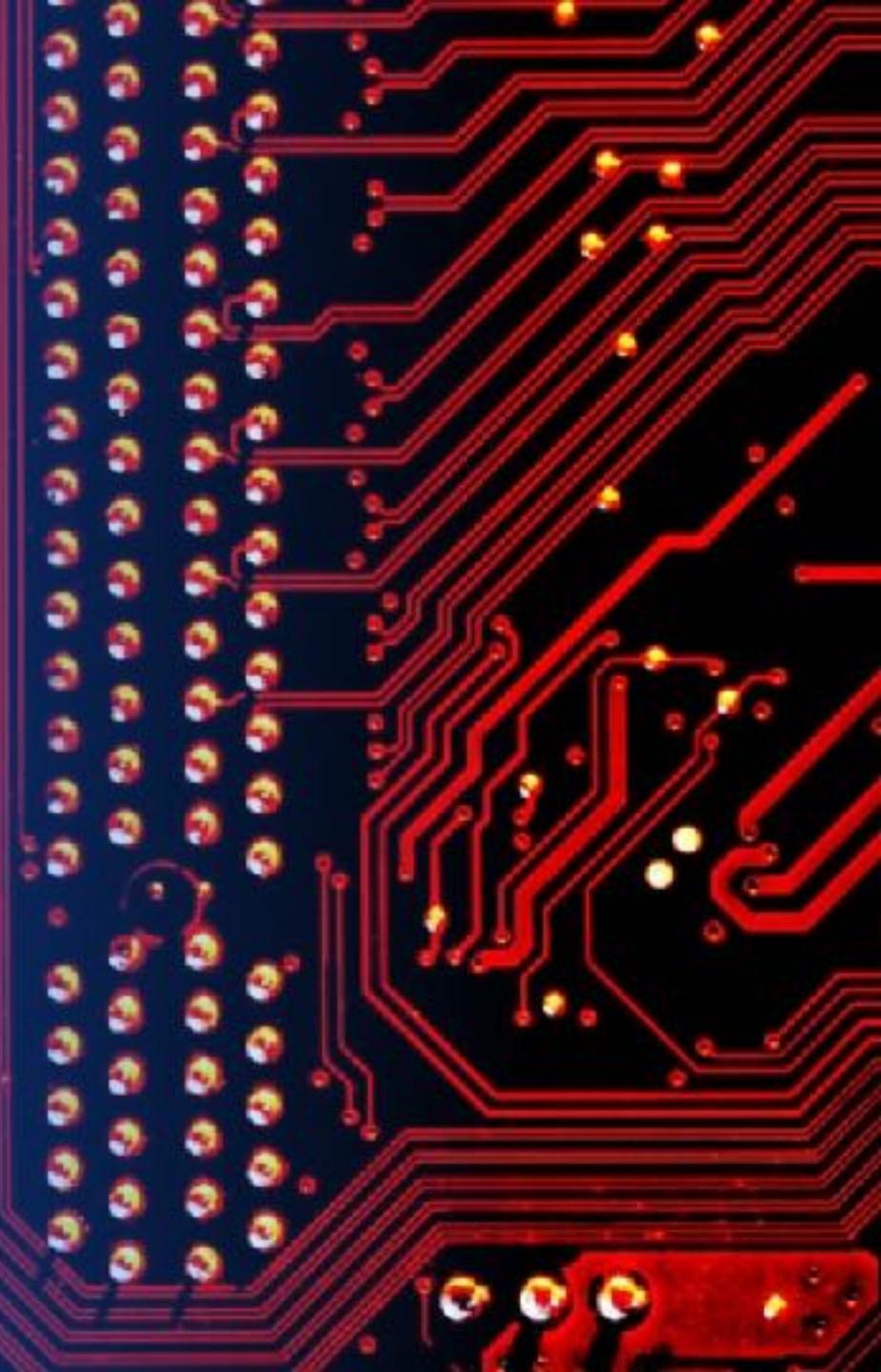
Do launch sites keep certain distance away from cities?

- YES. The nearest city is 51.74 km away.

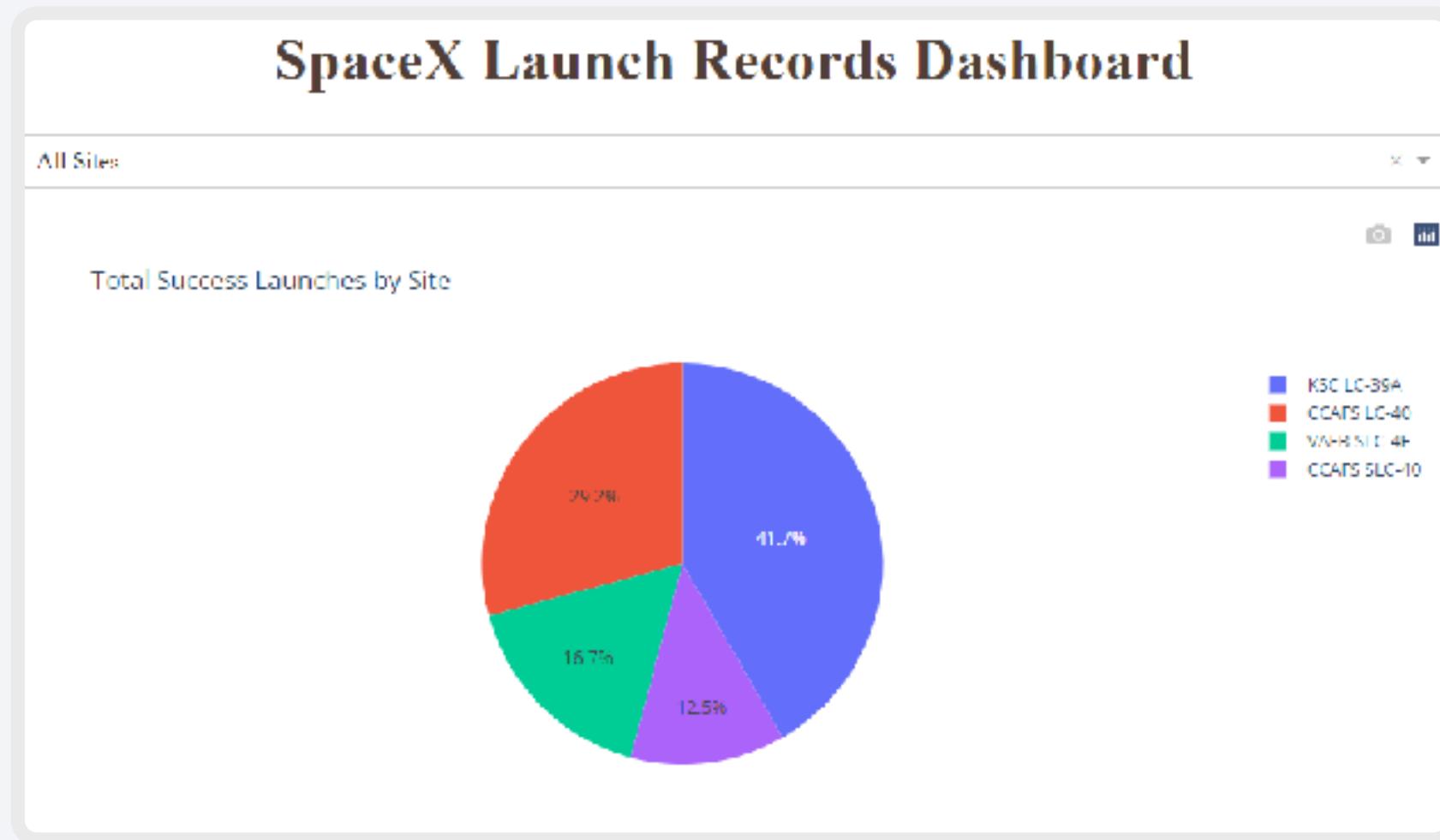


Section 4

## Build a Dashboard with Plotly Dash



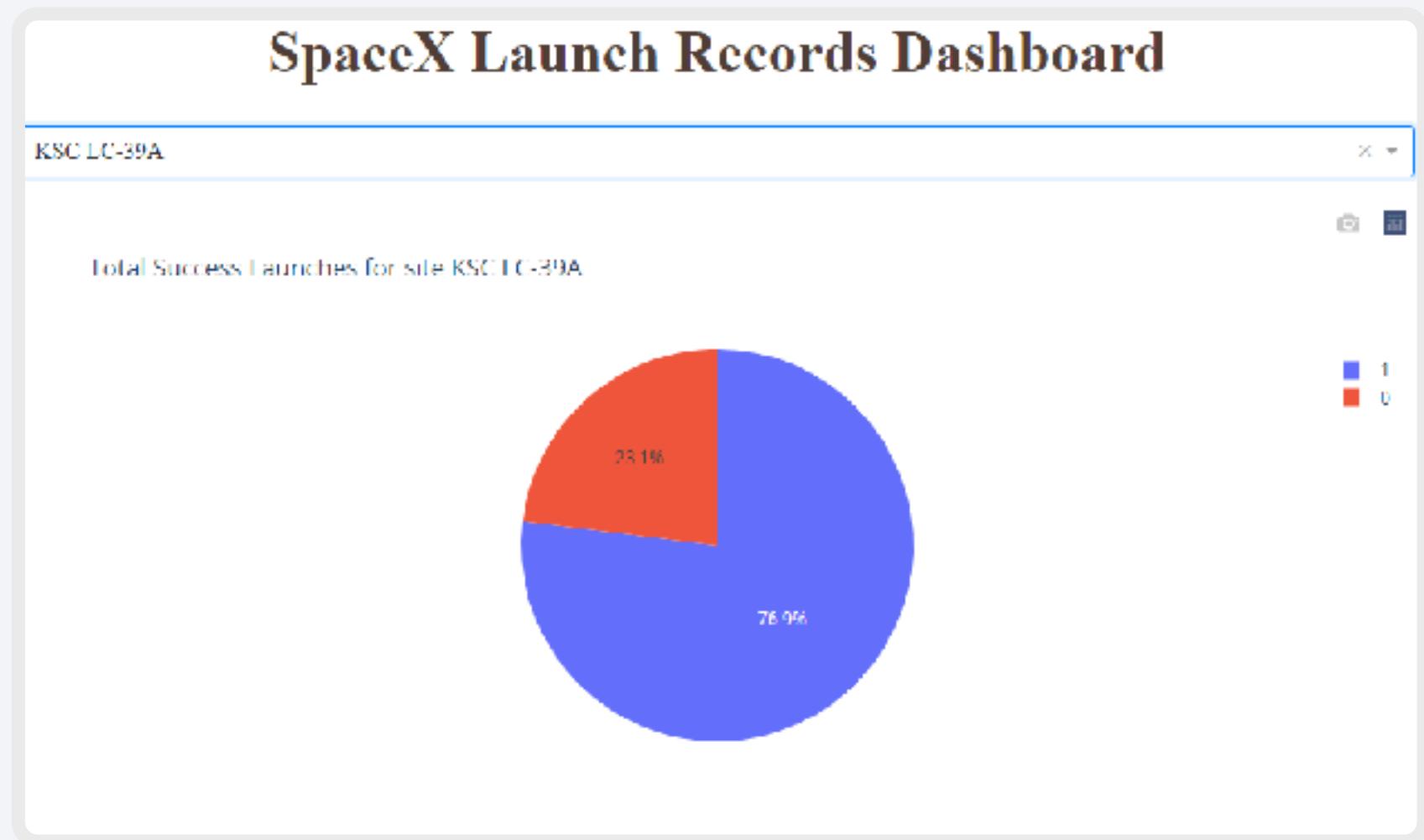
# Launch success count for all sites



- The launch site **KSC LC-39 A** had the most successful launches, with 41.7% of the total successful launches.

## Pie chart for the launch site with highest launch success ratio

Note:  
Class  $\begin{cases} 0, \text{Failure} \\ 1, \text{Success} \end{cases}$



The launch site KSC LC-39 A also had the highest rate of successful launches, with a 76.9% success rate.

# Launch Outcome VS. Payload scatter plot for all sites

Note:  
Class  $\begin{cases} 0, \text{Failure} \\ 1, \text{Success} \end{cases}$



- Plotting the launch outcome vs. payload for all sites shows a gap around 4000 kg, so it makes sense to split the data into 2 ranges:
  - 0 – 4000 kg (low payloads)
  - 4000 – 10000 kg (massive payloads)
- From these 2 plots, it can be shown that the success for massive payloads is lower than that for low payloads.
- It is also worth noting that some booster types (v1.0 and B5) have not been launched with massive payloads.

The background of the slide features a dynamic, abstract design. It consists of several thick, curved lines that transition from a bright yellow at the top right to a deep blue at the bottom left. These lines create a sense of motion and depth, resembling a tunnel or a stylized road. The overall effect is modern and professional.

Section 5

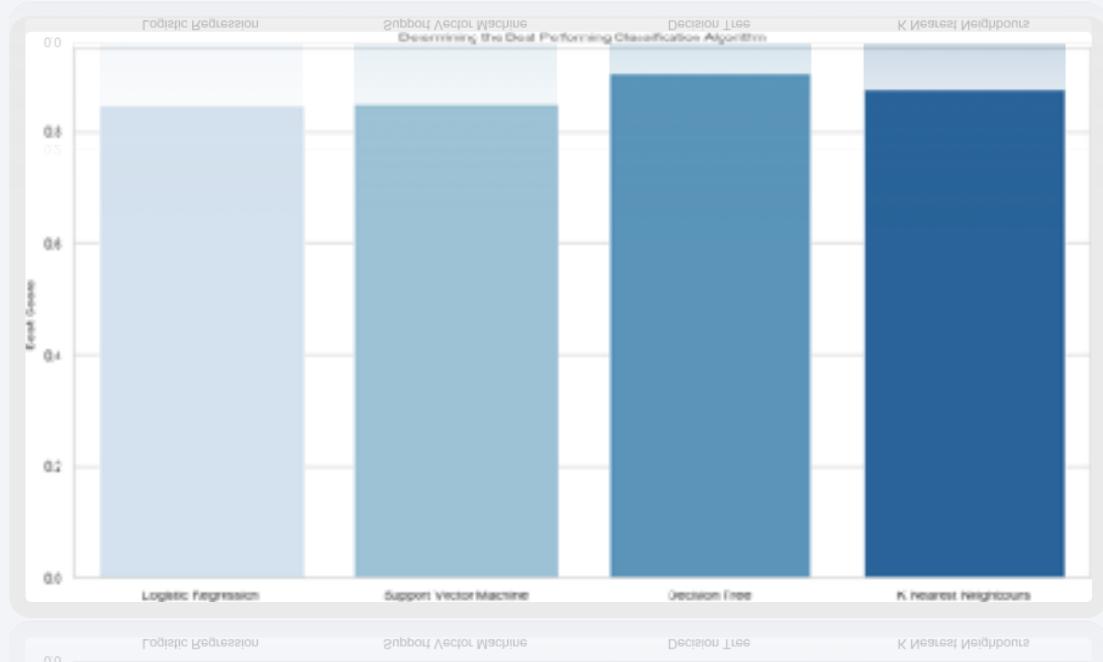
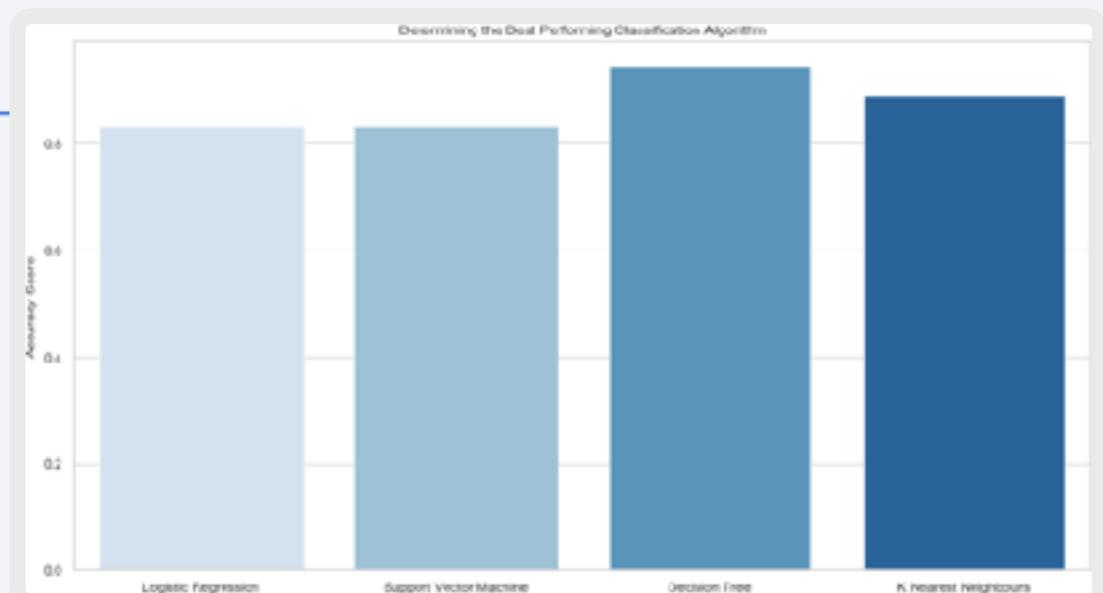
# Predictive Analysis (Classification)

# Classification Accuracy

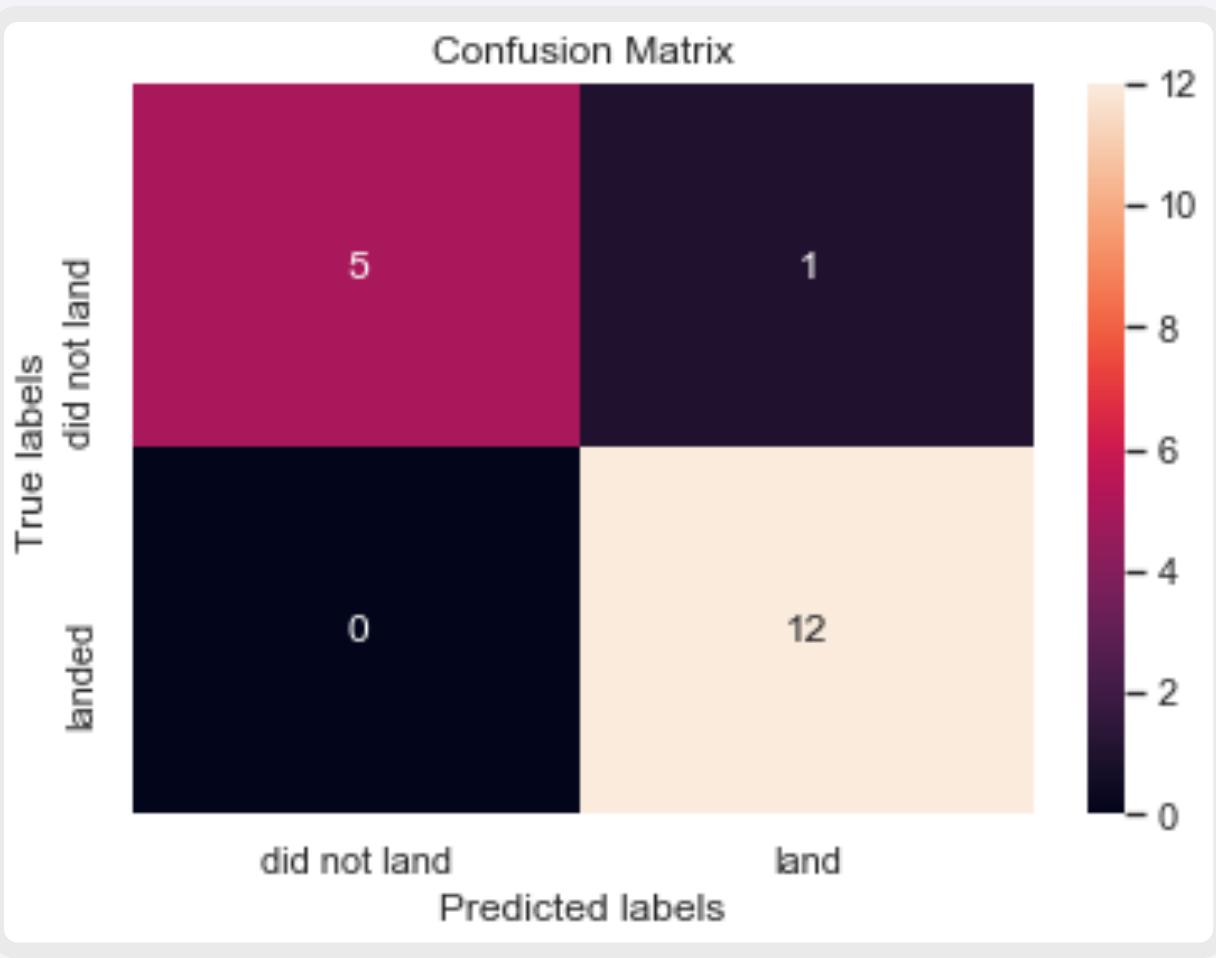
Plotting the Accuracy Score and Best Score for each classification algorithm produces the following result:

- The Decision Tree model has the highest classification accuracy
  - The Accuracy Score is 94.44%
  - The Best Score is 90.36%

Algorithm	Accuracy Score	Best Score
Logistic Regression	0.833333	0.846429
Support Vector Machine	0.000000	0.840214
Decision Tree	0.944444	0.903571
K Nearest Neighbours	0.888889	0.876786
K Nearest Neighbors	0.888889	0.876786
Decision Tree	0.944444	0.903571



# Confusion Matrix



- As shown previously, best performing classification model is the Decision Tree model, with an accuracy of 94.44%.
- This is explained by the confusion matrix, which shows only 1 out of 18 total results classified incorrectly (a false positive, shown in the top-right corner).
- The other 17 results are correctly classified (5 did not land, 12 did land).

# Conclusions

- As the number of flights increases, the rate of success at a launch site increases, with most early flights being unsuccessful. I.e. with more experience, the success rate increases.
  - Between 2010 and 2013, all landings were unsuccessful (as the success rate is 0).
  - After 2013, the success rate generally increased, despite small dips in 2018 and 2020.
  - After 2016, there was always a greater than 50% chance of success.
- Orbit types ES-L1, GEO, HEO, and SSO, have the highest (100%) success rate.
  - The 100% success rate of GEO, HEO, and ES-L1 orbits can be explained by only having 1 flight into the respective orbits.
  - The 100% success rate in SSO is more impressive, with 5 successful flights.
  - The orbit types PO, ISS, and LEO, have more success with heavy payloads:
    - VLEO (Very Low Earth Orbit) launches are associated with heavier payloads, which makes intuitive sense.
- The launch site KSC LC-39 A had the most successful launches, with 41.7% of the total successful launches, and also the highest rate of successful launches, with a 76.9% success rate.
- The success for massive payloads (over 4000kg) is lower than that for low payloads.
- The best performing classification model is the Decision Tree model, with an accuracy of 94.44%.



# Appendix - Data collection SpaceX REST API

- Custom functions to retrieve the required information
- Custom logic to clean the data

```
# We take a subset of our data from keeping only the Customer name and the flight number, and date etc.
sites = sites[["Customer", "FlightNumber", "LaunchSite", "FlightNumber", "FlightDate"]]

# We will remove some additional entries because these are either repeated or 2 entries related to each other
# that have multiple payloads on a single rocket.
sites = sites[sites['FlightNumber'].map(len) == 1]
sites = sites[sites['FlightNumber'].map(len) == 2]

# Since payload and core are lists of size 1 we will also extract the single value in the list and replace the feature.
sites['Payload'] = sites['Payload'].map(lambda x: x[0])
sites['Orbit'] = sites['Payload'].map(lambda x: x[0][0])

# We also need to convert the date into a datetime datatype and then extracting the date from the date
sites['Date'] = pd.to_datetime(sites['FlightDate']).dt.date

# Using the date we will construct the names of the cores
sites['Core'] = sites['Date'].dt.year.map(str).str[-2:] + sites['FlightNumber'].str[-3:]

sites
```

```
from the rocket, column we would like to learn the booster name.

# Takes the dataset and uses the rocket column to call the API and append the data to the list
def getRocketDetails(data):
    for x in data['rocket']:
        response = requests.get("https://api.spacexdata.com/v3/rockets"+x[0]+".json")
        Rocket.append(response.json())

from the launchpad we would like to know the name of the launch site being used, the longitude, and the latitude.

# Takes the dataset and uses the launchpad column to call the API and append the data to the list
def getLaunchPadDetails(data):
    for x in data['launchpad']:
        response = requests.get("https://api.spacexdata.com/v3/launchpads"+x[0]+".json")
        Longitude.append(response['longitude'])
        Latitude.append(response['latitude'])
        LaunchPad.append(response['name'])

from the payload we would like to learn the mass of the payload and the orbit that it is going to.

# Takes the dataset and uses the payloads column to call the API and append the data to the list
def getPayloadDetails(data):
    for load in data['payloads']:
        response = requests.get("https://api.spacexdata.com/v3/payloads"+load[0]+".json")
        PayloadMass.append(response['mass_kg'])
        Orbit.append(response['orbit'])
```

```
from cores we would like to learn the outcome of the landing, the type of the landing, number of flights with the core, whether grilles were used, whether the core is reused; whether legs were used, the landing pad used, the block of the core (which is a number used to separate versions of cores), the number of times this specific core has been reused, and the serial of the core.

# Takes the dataset and uses the cores column to call the API and append the data to the list
def getCoreDetails(data):
    for core in data['cores']:
        if core['core'] != None:
            response = requests.get("https://api.spacexdata.com/v3/cores/"+core['core'][0]+".json")
            Block.append(response['block'])
            Mission.append(response['mission'])
            Serial.append(response['serial'])
        else:
            Block.append(None)
            Mission.append(None)
            Serial.append(None)
            Outcome.append(str(core['landing_outcome'])+ " "+ str(core['landing_type']))
            Flights.append(core['flight'])
            Grilles.append(core['grilles'])
            Reused.append(core['reused'])
            Legs.append(core['legs'])
            LandingPad.append(core['landpad'])
```

```
magnum001.append(core[0].magnum001)
r02sub001.append(core[1].r02sub001)
goerge001.append(core[2].goerge001)
mrgm001.append(core[3].mrgm001)
cris001.append(core[4].cris001)
magnus001.append(core[5].magnus001)
magnus002.append(core[6].magnus002)
magnus003.append(core[7].magnus003)
```

# Appendix - Data collection WebScraping

- Custom functions for web scraping
- Custom logic to fill up the launch\_dict values with values from the launch tables

```
def get_launch_table_data():
    """
    This function extracts the data and time from the HTML table cell
    (td) of the element of a table data cell (td[0]) with class "td".
    Returns: list of tuples containing (td[0].text, td[1].text)
    """
    def extract_td(td):
        """
        This function extracts the header value from the HTML table cell
        (td) of the element of a table data cell (td[0]) with class "td".
        Returns: str
        """
        return td[0].text

    def extract_td(td):
        """
        This function extracts the landing status from the HTML table cell
        (td) of the element of a table data cell (td[0]) with class "td".
        Returns: str
        """
        return td[0].text[4]

    def get_td(td):
        """
        This function extracts the header value from the HTML table cell
        (td) of the element of a table data cell (td[0]) with class "td".
        Returns: str
        """
        if td[0].text == "None":
            return None
        else:
            return td[0].text[4]

    def extract_td(td):
        """
        This function extracts the landing status from the HTML table cell
        (td) of the element of a table data cell (td[0]) with class "td".
        Returns: str
        """
        if td[0].text == "None":
            return None
        else:
            return td[0].text[4]

    def extract_td(td):
        """
        This function extracts the landing status from the HTML table cell
        (td) of the element of a table data cell (td[0]) with class "td".
        Returns: str
        """
        if td[0].text == "None":
            return None
        else:
            return td[0].text[4]

    def extract_td(td):
        """
        This function extracts the landing status from the HTML table cell
        (td) of the element of a table data cell (td[0]) with class "td".
        Returns: str
        """
        if td[0].text == "None":
            return None
        else:
            return td[0].text[4]
```

```
def get_launch_table_data():
    """
    This function extracts the data and time from the HTML table cell
    (td) of the element of a table data cell (td[0]) with class "td".
    Returns: list of tuples containing (td[0].text, td[1].text)
    """
    def extract_td(td):
        """
        This function extracts the header value from the HTML table cell
        (td) of the element of a table data cell (td[0]) with class "td".
        Returns: str
        """
        return td[0].text

    def extract_td(td):
        """
        This function extracts the landing status from the HTML table cell
        (td) of the element of a table data cell (td[0]) with class "td".
        Returns: str
        """
        return td[0].text[4]

    def get_td(td):
        """
        This function extracts the header value from the HTML table cell
        (td) of the element of a table data cell (td[0]) with class "td".
        Returns: str
        """
        if td[0].text == "None":
            return None
        else:
            return td[0].text[4]

    def extract_td(td):
        """
        This function extracts the landing status from the HTML table cell
        (td) of the element of a table data cell (td[0]) with class "td".
        Returns: str
        """
        if td[0].text == "None":
            return None
        else:
            return td[0].text[4]

    def extract_td(td):
        """
        This function extracts the landing status from the HTML table cell
        (td) of the element of a table data cell (td[0]) with class "td".
        Returns: str
        """
        if td[0].text == "None":
            return None
        else:
            return td[0].text[4]
```

Thank you!

