Airport Analysis

OVERVIEW

This project, sponsored by Northrop Grumman, was created for the Computer Science Senior Capstone at the University of Colorado at Boulder from September 2016 to May 2017. At the beginning of the Fall 2016 semester, our team was assigned the task of creating a piece of functional software to assist in understanding air traffic trends in the United States. Through programmatic analysis, visualization, and machine learning, our team aimed to pinpoint problems in this industry as well as accomplish the goals outlined below. Over the course of the past 9 months our team has accomplished a large number of these goals and has created a functional piece of software that we hope will be used for research in the future.

GOALS

1. Learn and get experience with software that is currently being used in industry and research such as
    a. Python
    b. Javascript
    c. D3.JS
    d. PostgreSQL
    e. Cassandra
    f. Scikit-Learn
2. Get experience with both frontend and backend development on a larger scale project than previous Computer Science classes had allowed for
3. Utilize the Agile methodology in the context of team-based software development
4. Evaluate techniques involved with routing aircraft
5. Produce a baseline product that is easily extensible by a 3rd party

Table of Contents

Method

A chronological overview of the research and development associated with our project.

Configuration

An overview of how this project is configured, and what technologies need to be in place and set up to run it.

Running Instructions

An overview of how to run this project once the machine(s) are configured correctly, including limitations and expected outcomes (speed of responses, accuracy, etc.).

Conclusion

An overview of the results we found by using this system, and how it could be improved to find better results in the future

Team Information

Names and emails of team members.

METHOD

Conception

When tasked with this project, our team spent a large amount of time trying to figure out what we wanted our end product to be as well as how our product would differentiate itself from existing software. While there is undoubtedly a large amount of air traffic and airport data, companies such as Google, Expedia, and Travelocity have already utilized this data to create professional, large scale products that millions of people use on a daily basis. Considering our software development experience and skill levels as a team, we had a very difficult time conceptualizing how our project would be different or better than these existing products. After a number of team discussions we decided that we ultimately would not be able to compete with these companies and would therefore have our project be more oriented toward research as opposed to releasing a commercial product. While this decision was unanimous amongst the team, we were still running into issues as to what aspects of this data we wanted to analyze and incorporate into our project. Therefore, we utilized the Extreme software development methodology to try out different ideas with the intent of switching to the Agile methodology once we had a clearer idea as to what we wanted our project to be. After coming to this conclusion we decided we would first do a basic analysis and visualization on airports and air traffic in the United States.

Finding Data

The first few weeks of our project consisted of finding a cohesive and reliable source of data. Initially, we looked into using System Wide Information Management (SWIM) data from the Federal Aviation Administration (FAA) due to the fact that this government owned agency would likely have the most accurate and the largest quantity of data. After reaching out to a FAA representative, we quickly found out that we would be unable to access this data because of a lack of proper security and equipment to store it. Therefore, we then looked into using alternative sources of data that would have minimal prerequisites to access. Fortunately we found TranStats[1] which would ultimately be our main source of data for the Extreme period of our project. Our team then proceeded to write Python scripts to download, normalize, and store this data into a PostgreSQL database. After gaining access to data, our team quickly transitioned from research to development. We continued using this as our data source for the entirety of the project. For

[1] https://www.transtats.bts.gov/

the routing aspect of our development, we simply used a downloaded CSV file which contained airport latitude and longitude locations.

Frontend

Our frontend development initially consisted of research into what software we'd like to use and would work best for our project. We quickly decided on using Javascript, jQuery, and HTML5 to construct and present our project via webpage since this seems to be the current industry standard. Additionally, we did some research into what visualization software would work best provided the scope of our project as well as the data we had access to. After learning and getting practice with a number of these products such as Plotly, Tableau, and Bokeh, we ultimately decided on using the Google Maps API and D3JS as our visualization software. Finally, we constructed a header on our webpage which allowed a user to input a specific date range, airline carrier, source airport, destination airport, and flight number. These values would then be used to form a query to retrieve data from our database with respect the user's input. This query would result in a set of flights coming from the source airport within the provided date range along with delay information associated with each flight. Once the database transaction completed, we utilized a combination of Python, Javascript, D3JS, and the Google Maps API to plot out the airports and flight paths onto a map. Additionally, we used the retrieved delay information to color the plotted airports corresponding to the amount/severity of delays associated with specific airports. The only other frontend development our team did was creating a similar input-framework for routing and creating About pages which display information about our team and techniques.

Backend

Much like our frontend development, our backend development initially consisted of research into what software we'd like to utilize. Being as though our team had a relatively large amount of experience with Python, we decided on using Python and Tornado as our backend. The development associated with our backend consisted of creating methods to handle the user input, retrieve data from the database, calculate delay statistics, calculate airport lat/lon locations, and more. This data would then be passed to our frontend via AJAX calls and the corresponding visualization outlined above would be displayed. Once our team began development for routing, we were able to reuse many of the methods associated with visualization and plotting.

Database

Apache Cassandra was ultimately chosen as the main database due to its scalability, fault tolerance, and tunable tradeoff between consistency and latency. Initially the project was set up on PostgreSQL on single machines to rapidly explore and prototype data visualization techniques. This allowed us to create multiple prototypes which gave us direction on what to do once Cassandra was finally implemented. This prototyping stage also highlighted a need for large dataset storage and parallel compute capability. Considering that the team was limited to five desktop computers with 500GB hard drives, a database with easy to deploy clustering capability was preferred. PostgreSQL, while adequate for single machine deployments, was deemed too cumbersome to scale to a multi node compute cluster.

Machine Learning

Scikit-Learn was used for training and testing machine learning models, and was used in the final implementation of this system. Originally, Decision Trees were explored as a model for predicting delay time and time of a flight in the air. This approach yielded subpar estimates, so it was abandoned. Next, a K-Nearest Neighbors models was used to predict delay and flight time in air. Results for this model were better, so some time was spent creating a semi-automatic testing suite. This allowed for the testing of different sets of features and different neighbor values to be tested automatically, with results be output to a text file. This approach was primarily focused on predicting delay time, and did not adapt well to predicting time in air for flights. After settling on using machine learning as a heuristic for the A* routing algorithm, focus was shifted to using a Multi-Layer Perception (MLP) neural network to predict time of a flight in the air.

This model was trained and implemented on single month's of data at a time, using the features Month, Day of the Week, Listed Departure Time, and Distance to destination with the label Time Spent in Air. The model was created with an alpha value of 0.1, with a hidden layer configuration of (500, 100). Finding an appropriate feature set, alpha value, and hidden layer composition was done using partially automated testing. Testing was partially automated by having a list of features, alpha values, and hidden layer compositions generated by hand before testing, and training and testing every combination of these lists. Thus approximate configurations for the models could be generated by hand, and all possible combinations could be tested automatically, outputting a list of each configuration along with it's accuracy and time to train and test.

Results that show the output and accuracy of some of the partially automated testing can be found in section B of the appendix.

Routing

Main components of the routing section are a routing algorithm, a heuristic, and a grid of nodes data structure that is used to host the computation of routes. When choosing an algorithm for this application, it was important to consider that the number of nodes in the graph could scale to be quite large as grid resolution increased. A* was chosen for this purpose and implemented through reference to Stanford's A* Comparison[2]. This development was done in a part object oriented approach and a part scripted approach as the flexibility of Python allows for. The heuristic was separated out from the main algorithm in order to easily incorporate more advanced heuristics with an at run time import. The end user can implement additional heuristics by writing them to a python file with a name of their choice that includes a function in the format h(currentNode, parentNode).

Furthermore the parameters considered for the computation at each node are written in an extendable manner such that merely the class need be appended to. All node parameters are available for updating to the heuristic function used.

The algorithm currently uses a gridding method that covers the entire US without consideration of the start and end points. Optimization was taken into consideration, and the optimization strategy attempted can be seen in Appendix D.

CONFIGURATION

General System Requirements

Machine(s) capable of running Python Tornado, Cassandra, and CPU intensive Python processes. If the machine being used for hosting Tornado is not the machine being used to load the front end webpage, internet access and some network configuration may be necessary.

Project was originally developed and tested on multiple machines running CentOS 7.3.1611 each with an i7-2600 processor and 16 GB or RAM.

Operating System

Any machine capable of running the required technologies should suffice. This project was developed and tested on CentOs 7.3.1611, which can be installed using these directions located on the CentOS wiki (https://wiki.centos.org/FrontPage), but it can be configured on most flavors of linux.

Cloning Git Repository

All of our source code is stored via Git. In order to run our application, download or clone the repository from Github[3].

[2] http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

[3] https://github.com/JustinWayneOlson/Air-Traffic-Analysis

Python Environment

A mix of Python 2.7 and Python 3.5 was used in this project. Python 2 is used for the Tornado , and needs the pip package “tornado” installed.

Python 3.5 was used for most machine learning functions, as well as data ingestion and some routing. Python packages to install using pip include: numpy, scipy, scikit-learn, cassandra-driver.

Pip can be installed using the following documentation: https://packaging.python.org/installing/

Cassandra

Install Java: yum install java-1.8.0-openjdk
Download Cassandra: curl -L
http://apache.claz.org/cassandra/3.9/apache-cassandra-3.9-bin.tar.gz | tar xz
Install EPEL repos:
yum -y install epel-release

RUNNING INSTRUCTIONS

Once the operating system, required packages, and necessary technologies are installed, the system can be run using the following directions:

First Time Use

Running the system for the first time requires a few additional steps to populate the database. If you are planning on importing data to the database yourself, you may skip these steps.

1. Obtain data (Transtats portal: https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time, select “Prezipped File” and the desired month. This was previously automated, but Transtats has changed how they serve data files.)
2. Start Cassandra. Navigate to the cassandra bin folder: ./cassandra on each node
3. Set up cassandra Tables. Navigate to the cassandra bin folder: cqlsh -u 'my_username' -p 'my_password' -f Source ‘path to cassandra/cassandrainit.cql’
4. Fill Cassandra: data/cassandra_fill.py <TargetCsvFilePath>
5. Train basic MLP models: data\analysis\a_star_heuristic\MLP_Cassandra_Init_Fill.py

Normal Use

To run the system after completing first time setup, follow these steps:

1. Start Cassandra if it isn't already running. Navigate to the cassandra bin folder: ./cassandra -&
2. Start Tornado: application/src/python app.py
3. Open a web browser and navigate to the IP address of the machine running Tornado

The page should redirect you to "index.html". Here you will see an interface to build a query to filter through the large amount of transtats data pulled. Select the desired parameters, or input a custom JSON string in the textbox and click the "Plot Airports" button. A new tab should appear with various data displayed. Depending on the size of the data requested this could take some time. If the contents of the tab don't refresh after some time click on the tab again and it will refresh. The table can be searched, and filtered by column to view specifics about airports. Clicking on a specific airport will bring up a menu with some selected raw JSON data about the airport (JSON string format can be found in section C of the appendix).

The routing page will have a similar interface for selecting parameters. Much like the index page the routing page has an interface to plot routes as well as compute new routes. For the plotting section one need to merely select the name of the route from the Computed Routes Drop down. With the route selected, click on the plot button. This will display all the nodes from the computed route in a zoomable map overlay. Future versions of this page would include the rest of the information stored with the Node class as an on hover event for each node in the route with a summary of the aggregate node data on the right.

In order to compute new routes the user is given the option of using the front end for basic route computation, or to input a JSON string in order to take advantage of the extendible features of the routing algorithm. The simple features the front end provides are as follows: Sliders for the horizontal and vertical grid resolution. These sliders provide the option of refining how accurate height and latitude and longitude distances are calculated at the expense of computation time. Furthermore a simple job needs to have airport codes for origin and destination specified in a three letter format. Lastly a job name and heuristic file need to be specified. Once all parameters are entered, a push of the compute button will start the compute job. Once the job is finished the job name will be listed in the computed routes drop down.

Computing routes using JSON string entry allows the user to specify additional parameters for route computation. The current list of parameters can be found in appendix A. The parameters need to be specified in the following JSON format: {'Dest': 'DEN', 'Origin': 'SEA', 'gridResPlanar': '100', 'gridResVert': '1000', 'heuristic': '3dDistance', 'jobName': 'SeaDen'}

The above string constitutes an example of the minimum query the algorithm needs to compute a route from Seattle to Denver with a planar grid resolution of 100 miles per grid and altitude levels from 0 to 60,000ft in 1000ft increments. Furthermore this computation will use the 3dDistance.py file as a heuristic and store the computed route under the name SeaDen. It is possible here to specify additional parameters that a 3rd party heuristic may need in order to successfully compute a route. Likewise, one should note that the parameter list may be expanded in routingDriver.py's Node class to accommodate any additional parameters a 3rd party may use.

CONCLUSION

Results

Through a combination of thoughtful design and extended testing, we were unable to find meaningfully better routes or results than what already exists. This could be a limitation of our system, the data we had to work with, or the methods we used. The largest limiting factor is most likely the amount and quality of data we had access to, with the methods we used and our system in general being the next two factors.

Since we were restricted to free and easily accessible data, the data we were able to obtain and use was not complete or very comprehensive. Data from transtats was often incomplete, and only contained information about the flight's departure and arrival, restricting our analysis. Further data limitations included not obtaining specifications for the aircraft that flew each route, meaning our routing algorithm had limited capability and accuracy.

The methods and design choices that were used throughout this project are likely the second largest reason that we did not find better or more efficient routes than what already exists. Design choices that could have limited our results include, but are not limited to: path planning algorithm, path planning heuristic, database design, and data analysis. These all could have been modified or changed to yield better results if we were to continue working on this project, but these choices enabled us to quickly prototype the product, and are still good choices in the context of this project. The system in general functions well and handles storing, plotting and creating routes as intended, though the methods implemented to create routes could have been improved. Improving the system would include using better hardware, and dedicating more time to development operations. Improvements that could be made to development operations would be an automated deployment system, better documentation on setting up and connecting new servers, and better documentation on initializing and running the system.

Despite not finding meaningful or better results, this project still yielded an extensible, customizable, and complete modeling and research tool that could be used to find better results in the future. The product can be modified and improved easily, and can be customized to implement new methods, accommodate new data sets, and run on a variety of systems.

End Product

This project culminated in a full stack application, capable of ingesting and storing flight data, processing data and training neural networks, calculating efficient flight routes, displaying historic and calculated flights paths accurately on a map, and allowing interactive user input for requesting more flight paths. This product is extensible, well documented, and open-source. Potential applications of this product include: research tool for displaying flight paths, analysis of current and potential flight path heuristics, or database integrated machine learning framework.

Future

If we continued development on this project, we would've liked to implement more robust and detailed data ingestion and analysis, improvement on flight path heuristics, and hosting and maintaining this project on a public webserver. Further data ingestion and analysis would include integrating secure and restricted data sets, and integration of more sophisticated modeling techniques. Improvement on flight planning heuristics would improve generated routes, complete faster, and allow more complicated and in depth analysis of flight paths.

TEAM INFORMATION

Justin Olson (Team Lead)	justin.w.olson@colorado.edu
Matt Oakley (Visualization Lead)	matthew.oakley@colorado.edu
Mitchell Zinser (Machine Learning Lead)	mitchell.zinser@colorado.edu
Michael Muehlbradt (Architecture Lead)	michael.muehlbradt@colorado.edu
Kristen Hanslik (Analysis Lead)	kristen.hanslik@colorado.edu

APPENDIX

A. Routing Heuristic parameters

Every node has at minimum the following parameters:

lat
lon
alt
timeVisited
#Aircraft Performance parameters
aircraftType
aircraftWeight
aircraftFuelWeight
aircraftCargoWeight
aircraftFuelUsage
aircraftAirSpeed
aircraftGrndSpeed

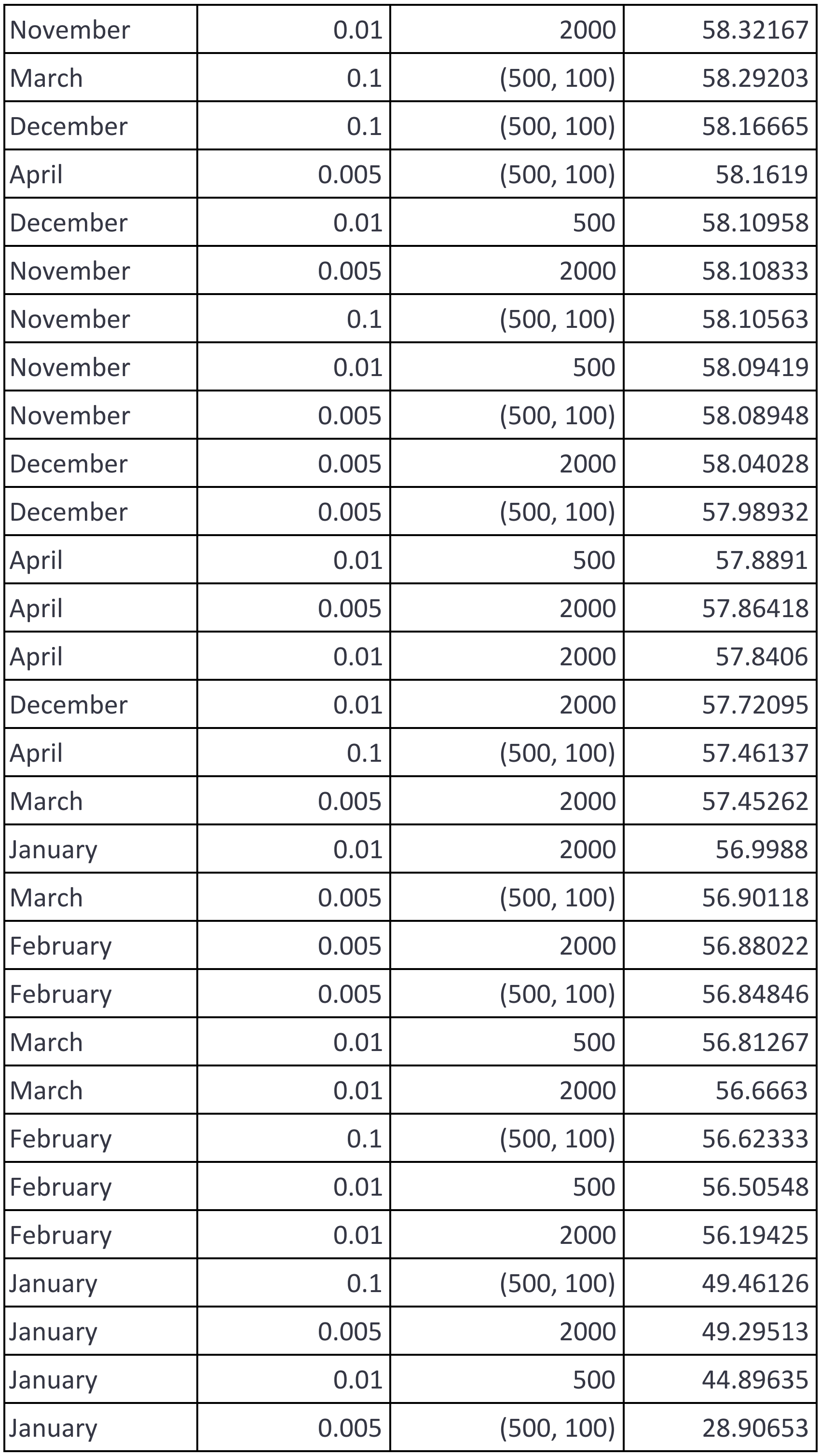
#Weather parameters
windSpeed
windDirection
precipitationChance
precipitationType
precipitationStrength
airTemp
humidity
dewPoint

B. MLP Model Accuracy

The following table show results of training and testing the MLP used in this project over each month in 2015, using the whole month of data as the training set for each model. Features tested:

Month, Day of the Week, Listed Departure Time, Distance. Label tested for: Airtime of Flight. Rounding: 20 minute intervals. Possible MLP configurations for each month (Alpha Value, Hidden Layer composition): [(0.005, (500, 100)), (0.1, (500, 100)), (0.005, 2000), (0.01, 500), (0.01, 2000)]. Rounding was set to 20 minute intervals.

Month	Alpha Value	Hidden Layer Composition	Accuracy (%)
September	0.1	(500, 100)	70.16152
September	0.005	(500, 100)	69.97355
August	0.01	2000	69.66886
September	0.01	500	69.55801
June	0.01	500	69.47578
August	0.005	2000	69.40587
July	0.005	(500, 100)	69.36842
May	0.005	(500, 100)	69.34699
July	0.005	2000	69.29422
July	0.1	(500, 100)	69.14921
July	0.01	2000	69.08648
June	0.005	(500, 100)	68.99374
August	0.01	500	68.96913
September	0.01	2000	68.90079
June	0.01	2000	68.87441
June	0.005	2000	68.8039
July	0.01	500	68.76273
August	0.1	(500, 100)	68.73927
May	0.1	(500, 100)	68.60717
August	0.005	(500, 100)	68.45397
September	0.005	2000	68.15564
May	0.01	500	67.60155
June	0.1	(500, 100)	66.43841
May	0.005	2000	65.24513
October	0.01	500	65.05112
October	0.005	2000	64.8637
October	0.005	(500, 100)	64.71659
October	0.01	2000	64.70786
October	0.1	(500, 100)	64.41834
May	0.01	2000	63.39052



November	0.01	2000	58.32167
March	0.1	(500, 100)	58.29203
December	0.1	(500, 100)	58.16665
April	0.005	(500, 100)	58.1619
December	0.01	500	58.10958
November	0.005	2000	58.10833
November	0.1	(500, 100)	58.10563
November	0.01	500	58.09419
November	0.005	(500, 100)	58.08948
December	0.005	2000	58.04028
December	0.005	(500, 100)	57.98932
April	0.01	500	57.8891
April	0.005	2000	57.86418
April	0.01	2000	57.8406
December	0.01	2000	57.72095
April	0.1	(500, 100)	57.46137
March	0.005	2000	57.45262
January	0.01	2000	56.9988
March	0.005	(500, 100)	56.90118
February	0.005	2000	56.88022
February	0.005	(500, 100)	56.84846
March	0.01	500	56.81267
March	0.01	2000	56.6663
February	0.1	(500, 100)	56.62333
February	0.01	500	56.50548
February	0.01	2000	56.19425
January	0.1	(500, 100)	49.46126
January	0.005	2000	49.29513
January	0.01	500	44.89635
January	0.005	(500, 100)	28.90653

Random guessing accuracy within the max and min of the label in the dataset, averaging over >10 iterations:

Rounding Interval (Minutes)	Accuracy (%)
30	5.0

20	3.3
10	1.7
5	0.82
1	0.17

C. JSON String Format

```
{'Carrier': ['AA'],
 'Dest': ['SEA'],
 'FlightNum': '992',
 'Origin': ['LAS'],
 'date_end': '12/31/2016',
 'date_start': '01/01/2016',
 'path_toggle': True,
 'verbose_toggle': True}
```

D. Routing Optimization Details

For the routing algorithm, an optimization was attempted that would reduce the number of nodes needed to do a routing computation by taking into account those start and end points. The attempted optimization consisted of the following algorithmic structure:

1. Extend the latitude and longitude of each origin and destination away from each other by a tolerance of x degrees.
2. Take these two new points with added tolerance and compute the coefficients that describe the line that can be formed between them. This line can be shifted both up and down by a given tolerance to determine the upper and lower bounds of the map to be formed.
3. Determine whether the dimension in latitude or longitude is longer. Take this longer dimension and divide by the grid resolution size that is specified by the user.
4. The previous step allows for effective iteration over the line in grid-sized chunks. For each chunk value, the max vertical and minimum vertical bounds can be computed using the information computed in step 2.

5. For each chunk of the line, nodes are then generated to fill between the maximum bound and minimum bound calculated.
6. Doing this successively and populating neighbors by utilizing pre-allocation of nodes allows for the generation of a reduced-sized map that can be passed to the routing algorithm.