Using Supervised Machine Learning to Predict Flight Delays

By Jose Gallegos

1/23/2018

General Assembly


Will My Flight
Be Delayed?

Why are we interested in this data?

- Go to Document

Measuring Out Models

Metrics for Binary Classification
Binary Classification is arguably the most common and conceptually simple application fo machine learning in practice. However, there are still anumber of caveats in evaluating this simple task.

We are looking for the positive class.
We are not looking for the negative class.

Accuracy is not a good measure of predictive performance, since the number of mistakes one makes does not contain all the information that interests the data scientist.

Think of testing flight data for lateness. If the test is negative, then the observation is labelled "on time". But our predictive model might make mistakes…

It could classify a "delayed" flight as "on time." This incorrect positive prediction ("false positive") leads to some business costs involving travel. They could be as expensive as a few days stay in a hotel. Or it could have business consequences such as missing a timely opportunity.

The other possible mistake is that an "on time" flight could be classified as "delayed." This incorrect negative prediction is called a "false negative."

On the other hand, given similar flight data, we could also generalize a model that would predict airline catastrophes… But that is out of the realm of this presentation.

Reasons for using metrics:
Using a metric to assess a model ensures that it weeds out nonsense predictions.
Can easily spot balance or imbalanced classes.

Accuracy metrics for Logistic Regression:
Precision, Recall, f-score

Conclusions
Knowing the causes that contribute to airline delays creates better-informed consumers. Knowing specifically what types of delays are possible, when to expect delays, and what airlines and airports to associate with delays would be great information to have. This knowledge could empower the cutomer to choose between a list of the best performing airlines and airports given a set of conditions. Alternatively, the same customer could decide when to fly, expecially if his home airport is computed to be one of the main culprits. This knowledge could also empower the airlines so that they may monitor hencefoth or may research specific causes of

delays. Another business benefit could be saving time and capital expenditure (preventing loss). Another result could be attracting customers (increasing profit).

Because there is no association with life or death, we can say that it is not abundantly clear which consequence is more drastic, missed opportunites, travel expenditure over budget, profit/loss, etc. It all depends on the business decisions that have to be made based on the data. As such, we would have to assign dollar value to each kind of mistake, which would allow measuring data in actual money instead of abstract probability.

Where do we get the data?

- https://www.transtats.bts.gov/OT_Delay/OT_DelayCause1.asp

Cleaning the data

- Drop columns with no information
- Remove leading space
- Drop NaNs and ensure NaN strategy is solid
- Go to document

2018.01.08

In [1]:
```
import numpy as np
import pandas as pd
import re
import seaborn as sns
import matplotlib.pyplot as plt

from pygeocoder import Geocoder
from mpl_toolkits.basemap import Basemap

%matplotlib inline
```

Import the csv

In [2]:
```
# where this data came from: https://www.transtats.bts.gov/OT_Delay/OT_DelayCause1.asp?pn=1
orig_data = pd.read_csv('./project_datasets/682464398_102017_1651_airline_delay_causes.csv', skipi
nitialspace=True)
```

drop the extraneous column that has no information

In [3]:
```
data_right_columns = orig_data.drop('Unnamed: 21', axis=1)
```

Remove redundant columns

Of the four columns in `col_categorical`, only two are relevant since the other two contain the same information

In [29]:
```
# the `data2` variable holds the `col_categorical` because when we color the redundant columns, the
# programming that does the coloring disables other DataFrame methods and functions we will need for analysis
data2 = data[col_categorical]
# function examines value's length, and returns red if its too long
def longstring_red(val):
    val = len(str(val)); color = 'red' if val >3 else 'black'; return 'color: %s' % color
# apply the function
redundant_red = data2[:5].style.applymap(longstring_red)
redundant_red
```

Out[29]:

	carrier	carrier_name	airport	airport_name
0	AA	American Airlines Inc.	DFW	Dallas/Fort Worth, TX: Dallas/Fort Worth International
1	AA	American Airlines Inc.	DTW	Detroit, MI: Detroit Metro Wayne County
2	AA	American Airlines Inc.	SEA	Seattle, WA: Seattle/Tacoma International
3	AA	American Airlines Inc.	JFK	New York, NY: John F. Kennedy International
4	AA	American Airlines Inc.	SJC	San Jose, CA: Norman Y. Mineta San Jose International

In [30]:
```
# remove redundant columns
# `col_categorical` is the second variable we see below in the Quick Summary__________ second [2nd]
col_categorical = col_categorical[0::2]
col_categorical
```

Out[30]:
```
['carrier', 'airport']
```

In [31]:
```
# This is much tider and can be used easily for stats and modeling
data[col_categorical].head()
```

Out[31]:

	carrier	airport
0	AA	DFW
1	AA	DTW
2	AA	SEA
3	AA	JFK

Data Description

- 21 features: 'year', 'month', 'carrier', 'carrier_name', 'airport', 'airport_name','arr_flights', 'arr_del15', 'carrier_ct', 'weather_ct', 'nas_ct', 'security_ct', 'late_aircraft_ct','arr_cancelled', 'arr_diverted', 'arr_delay', 'carrier_delay', 'weather_delay', 'nas_delay','security_delay', and 'late_aircraft_delay’
- 11173 Observations
- Dummied-out to 345 features…

Some of the column names have initial space, so we use a regular expression

to make them easier to work with programmatically

In [4]:
```
columns = []
for i, col in enumerate(data_right_columns.columns):
    global columns
    col = re.sub(r'\s+', '', col)
    columns.append(col)
data_right_columns.columns = columns
data_right_columns.columns
```

Out[4]:
```
Index([u'year', u'month', u'carrier', u'carrier_name', u'airport',
       u'airport_name', u'arr_flights', u'arr_del15', u'carrier_ct',
       u'weather_ct', u'nas_ct', u'security_ct', u'late_aircraft_ct',
       u'arr_cancelled', u'arr_diverted', u'arr_delay', u'carrier_delay',
       u'weather_delay', u'nas_delay', u'security_delay',
       u'late_aircraft_delay'],
      dtype='object')
```

these next two cells work with dot notation and bracket notation

... but no need to run them except for sanity check

In [5]:
```
# data_right_columns['weather_ct'].head()
```

In [6]:
```
# data_right_columns.weather_ct.head()
```

Dropping the NaN rows as invalid data, it is a small percentage of the entire dataset

In [7]:
```
data_right_columns.shape
```

Out[7]:
```
(11193, 21)
```

In [8]:
```
data_right_columns.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11193 entries, 0 to 11192
Data columns (total 21 columns):
year                  11193 non-null int64
month                 11193 non-null int64
carrier               11193 non-null object
carrier_name          11193 non-null object
airport               11193 non-null object
airport_name          11193 non-null object
arr_flights           11175 non-null float64
arr_del15             11173 non-null float64
carrier_ct            11175 non-null float64
weather_ct            11175 non-null float64
nas_ct                11175 non-null float64
security_ct           11175 non-null float64
late_aircraft_ct      11175 non-null float64
arr_cancelled         11175 non-null float64
arr_diverted          11175 non-null float64
arr_delay             11175 non-null float64
carrier_delay         11175 non-null float64
weather_delay         11175 non-null float64
nas_delay             11175 non-null float64
security_delay        11175 non-null float64
late_aircraft_delay   11175 non-null float64
dtypes: float64(15), int64(2), object(4)
memory usage: 1.8+ MB
```

In [9]:
```
(11193 - 11173) / 11193.0
```

Out[9]:
```
0.001786831055123738
```

In [10]:
```
data_NoNaN = data_right_columns.dropna(axis=0)
```

In [11]:
```
# while the .info() function prints out a by-column summary of non- null values,
# best practices would be to double-verify the absence of NaNs with by calling
# `print(dataset.isnull().sum())`
print(data_NoNaN.isnull().sum())
# print(data_filled_mean.isnull().sum())
# print(data_filled_median.isnull().sum())
# print(data_filled_mode.isnull().sum())
```

```
year                  0
month                 0
carrier               0
carrier_name          0
airport               0
airport_name          0
arr_flights           0
arr_del15             0
carrier_ct            0
weather_ct            0
nas_ct                0
security_ct           0
late_aircraft_ct      0
arr_cancelled         0
arr_diverted          0
arr_delay             0
carrier_delay         0
weather_delay         0
nas_delay             0
security_delay        0
late_aircraft_delay   0
dtype: int64
```

In [12]:
```
data_NoNaN.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 11173 entries, 0 to 11192
Data columns (total 21 columns):
year                  11173 non-null int64
month                 11173 non-null int64
carrier               11173 non-null object
carrier_name          11173 non-null object
airport               11173 non-null object
airport_name          11173 non-null object
arr_flights           11173 non-null float64
arr_del15             11173 non-null float64
carrier_ct            11173 non-null float64
weather_ct            11173 non-null float64
nas_ct                11173 non-null float64
security_ct           11173 non-null float64
late_aircraft_ct      11173 non-null float64
arr_cancelled         11173 non-null float64
arr_diverted          11173 non-null float64
arr_delay             11173 non-null float64
carrier_delay         11173 non-null float64
weather_delay         11173 non-null float64
nas_delay             11173 non-null float64
security_delay        11173 non-null float64
late_aircraft_delay   11173 non-null float64
dtypes: float64(15), int64(2), object(4)
memory usage: 1.9+ MB
```

So, the Nans have been dropped...

Another approach would be to fill the NaN values with mean, median or mode...

we'd have to start with the `data_right_columns` variable

In [13]:
```
data_filled_mean = data_right_columns.fillna((data_right_columns.mean()))
# data_filled_mean.info()
```

In [14]:
```
data_filled_median = data_right_columns.fillna((data_right_columns.median()))
# data_filled_median.info()
```

In [15]:
```
data_filled_mode = data_right_columns.fillna((data_right_columns.quantile(.5)))
# data_filled_mode.info()
```

Now we want to look at the entire datasets' statitstics

In [16]:
```
data_NoNaN.describe()
```

Out[16]:

	year	month	arr_flights	arr_del15	carrier_ct	weather_ct	nas_ct	security_ct	late_
count	11173.0	11173.000000	11173.000000	11173.000000	11173.000000	11173.000000	11173.000000	11173.000000	1117
mean	2016.0	6.025239	461.532892	77.029088	22.628786	2.278168	24.216326	0.146415	27.7
std	0.0	3.157616	1170.892758	187.033010	51.024876	6.806785	66.990992	0.578833	72.3
min	2016.0	1.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00
25%	2016.0	3.000000	60.000000	8.000000	2.720000	0.000000	1.440000	0.000000	1.95
50%	2016.0	6.000000	133.000000	21.000000	7.520000	0.270000	4.770000	0.000000	6.31
75%	2016.0	9.000000	343.000000	56.000000	19.300000	1.740000	14.010000	0.000000	18.5
max	2016.0	11.000000	21977.000000	3368.000000	1242.160000	168.440000	1092.620000	11.730000	1320

In [17]:
```
# data_filled_mean.describe()
```

In [18]:
```
# data_filled_median.describe()
```

In [19]:
```
# data_filled_mode.describe()
```

Working with NaN values

We are working with NaN values only in the float variables, so filling categorical with statistical data is not an issue

There are advantages and disadvantages to the four Nan handilng strategies

- **Drop** NaN values (+): easy, clear-cut, , simplest strategy, works because it is a small percentage of the data, increases computational speed and is convenient
- **Drop** NaN values (-): aggressively eliminates data, imbalances small datasets
- variable: `data_NoNaN`

- Fill NaN with **mean** (+): balances a dataset with probable values, artful
- Fill NaN with **mean** (-): fabricates data, imbalances small datasets
- variable: `data_filled_mean`

- Fill NaN with **median** (+): balances a dataset with probable values, artful
- Fill NaN with **median** (-): fabricates data, imbalances small datasets
- variable: `data_filled_median`

- Fill NaN with **mode** (+): balances a dataset with probable values, artful
- Fill NaN with **mode** (-): fabricates data, imbalances small datasets
- variable: `data_filled_mode`

At this point, we don't know what the use will be of imputing or just dropping the values, but we have each dataset variable-ized, so we can begin to create subsets... It might be that we have to impute a different statistic for each variable, so this global handling of NaNs will have to be addressed.

Once NaNs are addressed we carry forward knowing that we may have to re-strategize our handling of them. Many machine Learning Algorithms will not work if a dataset contains NaNs. Its datatype has to be consistent.

Imputing Nans

Imputation won't work if it has to work on categorical columns. So we subtract those features and the target from the dataset before using the imputer function from sklearn:

- `from sklearn.preprocessing import Imputer`

Then, depending on our data problem, we might need to re-join the imputed values, or just work with them

categorical columns that we won't include in the imputer:

- 'year', 'month', 'carrier', 'carrier_name', 'airport', 'airport_name'

target column not included in the imputer dataset:

- 'arr_del15'

So, in data.iloc[] notation, the imputer takes in:

- `data_right_columns.iloc[:,[6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]]`

Group the Data

- Categorical
- Numeric
- Time Data

Stores variables 1 - 6 :

data, col_year_month, col_categorical, col_target, col_numeric, combined_list

In [12]:
```
%store data
```

```
Stored 'data' (DataFrame)
```

Group by Categorical / Numeric

In [13]:
```
col_numeric = data.select_dtypes(include=[np.number]).columns.tolist()
col_categorical = data.select_dtypes(include=[np.object]).columns.tolist()
```

`select_dtypes()` selected `year` and `month` as numeric. They are now separated so that pure numeric columns may be seen graphically

Use `year` and `month` separately in the `col_year_month` variable

In [14]:
```
# `col_year_month` is the first variable we see below in the Quick Summary__________ first [1st]
col_year_month = col_numeric[0:2]
col_year_month
```

Out[14]:
```
['year', 'month']
```

In [15]:
```
%store col_year_month
```

```
Stored 'col_year_month' (list)
```

Remove redundant columns

Of the four columns in `col_categorical`, only two are relevant since the other two contain the same information

In [16]:
```
# remove redundant columns
# `col_categorical` is the second variable we see below in the Quick Summary__________ second [2n
d]
col_categorical = col_categorical[0::2]
col_categorical
```

Out[16]:
```
['carrier', 'airport']
```

In [17]:
```
%store col_categorical
```

```
Stored 'col_categorical' (list)
```

Separate the target so it can be treated separately

In [18]:
```
# `col_target` is the third variable we see below in the Quick Summary__________ third [3rd]
col_target = col_numeric[16:17]
col_target
```

Out[18]:
```
['target']
```

In [19]:
```
%store col_target
```

```
Stored 'col_target' (list)
```

Numeric Columns

In [20]:
```
# rename `col_numeric` so it does not include 'year' and 'month'
# `col_numeric` is the fourth variable we see below in the Quick Summary__________ fourth [4th]
col_numeric = col_numeric[2:16]
# col_numeric
```

In [21]:
```
%store col_numeric
```

```
Stored 'col_numeric' (list)
```

Combine the lists

In [22]:
```
pre_combine = [col_year_month, col_categorical, col_numeric, col_target]
combined_list = [item for sublist in pre_combine for item in sublist]
# combined_list
# ['year', 'month', 'carrier', 'airport', 'arr_flights', 'carrier_ct', 'weather_ct',
#  'nas_ct', 'security_ct', 'late_aircraft_ct', 'arr_cancelled', 'arr_diverted',
#  'arr_delay', 'carrier_delay', 'weather_delay', 'nas_delay', 'security_delay',
#  'late_aircraft_delay', 'target']
```

In [23]:
```
%store combined_list
```

```
Stored 'combined_list' (list)
```

In [2]:
```
%store -r Denver_grouping
```

In [4]:
```
Denver_grouping.sum()
```

Out[4]:

			carrier_ct	target
airport	carrier	carrier_name		
DEN	AA	American Airlines Inc.	718.90	11
	AS	Alaska Airlines Inc.	71.55	6
	B6	JetBlue Airways	93.23	9
	DL	Delta Air Lines Inc.	427.88	11
	F9	Frontier Airlines Inc.	763.01	11
	NK	Spirit Air Lines	124.74	11
	OO	SkyWest Airlines Inc.	1414.07	11
	UA	United Air Lines Inc.	2077.87	11
	VX	Virgin America	23.79	5
	WN	Southwest Airlines Co.	2469.61	11

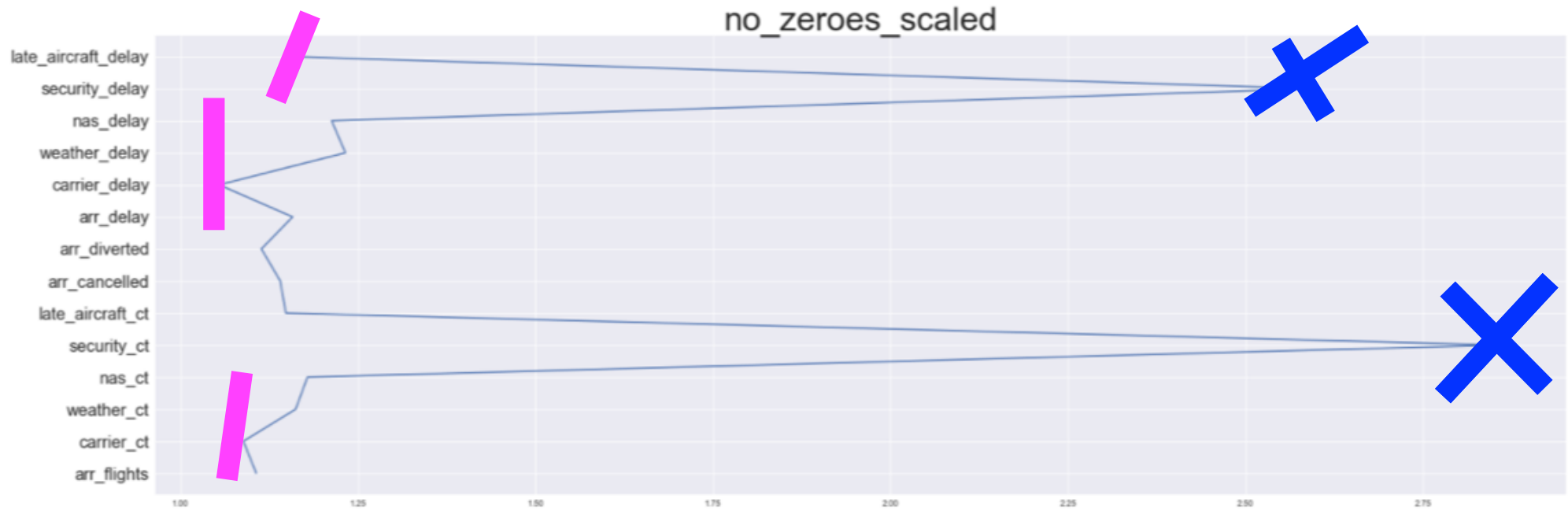
What is Logistic Regression

- Go to document

- https://github.com/josezilla/DAT-DEN-03/blob/master/projects/FlightDelaysProject/ProjectFolder%20copy/2018.01.21_What_is_Logistic_Regression.ipynb

Plots

```
Out[45]: [<matplotlib.lines.Line2D at 0x1220c0350>]
```



In [2]:
```
%store -r Denver_grouping
```

In [4]:
```
Denver_grouping.sum()
```

Out[4]:

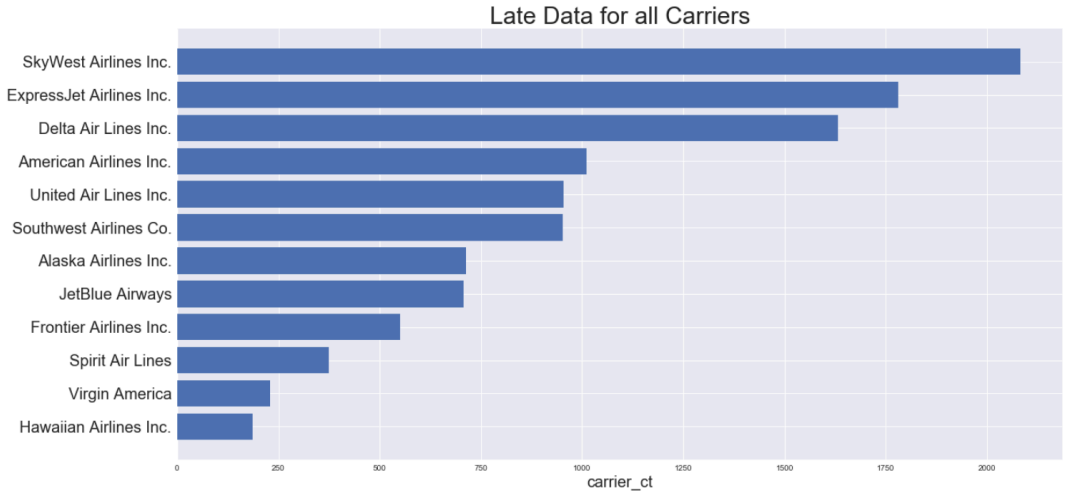
			carrier_ct	target
airport	**carrier**	**carrier_name**		
DEN	**AA**	**American Airlines Inc.**	718.90	11
	AS	**Alaska Airlines Inc.**	71.55	6
	B6	**JetBlue Airways**	93.23	9
	DL	**Delta Air Lines Inc.**	427.88	11
	F9	**Frontier Airlines Inc.**	763.01	11
	NK	**Spirit Air Lines**	124.74	11
	OO	**SkyWest Airlines Inc.**	1414.07	11
	UA	**United Air Lines Inc.**	2077.87	11
	VX	**Virgin America**	23.79	5
	WN	**Southwest Airlines Co.**	2469.61	11

Plots

Barplots I

Barplots for categorical data - carrier

In [26]:
```
# Have to use `data[col_categorical].carrier` and `data.carrier_name`
height = data[col_categorical].carrier.value_counts().values
bars = data.carrier_name.value_counts().keys()
# use set slicing to show the list from most to least
y_pos = np.arange(len(bars))[::-1]
# increase the figure size
plt.figure(figsize=(20,10))
plt.barh(y_pos, height)
plt.yticks(y_pos, bars, size=20)
plt.title('Late Data for all Carriers', size = 30)
plt.xlabel("carrier_ct", size=20)
plt.show()
```



Barplots II

Barplots for categorical data - Airline

Graphing this entire dataset is messy since there are many categories.

We will have to use some sorted data to look at specific groups of airports

In [27]:
```
# this grouping focuses on Denver Airport, is indexed by carrier and shows the total number of del
ayed flights
# referenced by the carrier_ct
# we could later calculate a percentage
Denver_grouping = data[data.airport == 'DEN'].iloc[:, [2, 3,4,7, 20]].groupby(by=['airport', 'carr
ier', 'carrier_name'])
Denver_grouping.sum().sort_values('carrier_ct')
```

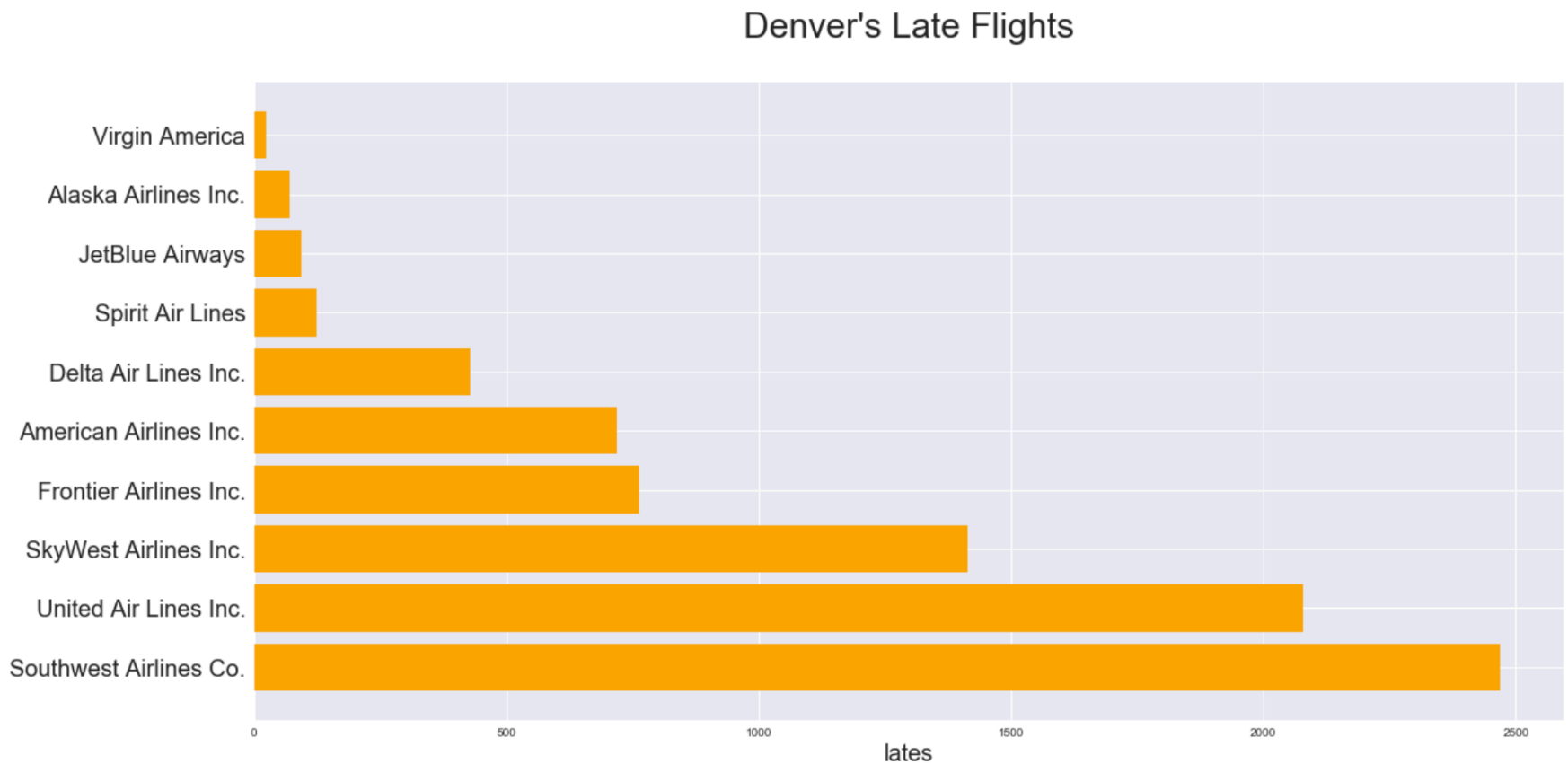
Out[27]:

			carrier_ct	target
airport	**carrier**	**carrier_name**		
DEN	**VX**	**Virgin America**	23.79	5
	AS	**Alaska Airlines Inc.**	71.55	6
	B6	**JetBlue Airways**	93.23	9
	NK	**Spirit Air Lines**	124.74	11
	DL	**Delta Air Lines Inc.**	427.88	11
	AA	**American Airlines Inc.**	718.90	11
	F9	**Frontier Airlines Inc.**	763.01	11
	OO	**SkyWest Airlines Inc.**	1414.07	11
	UA	**United Air Lines Inc.**	2077.87	11
	WN	**Southwest Airlines Co.**	2469.61	11

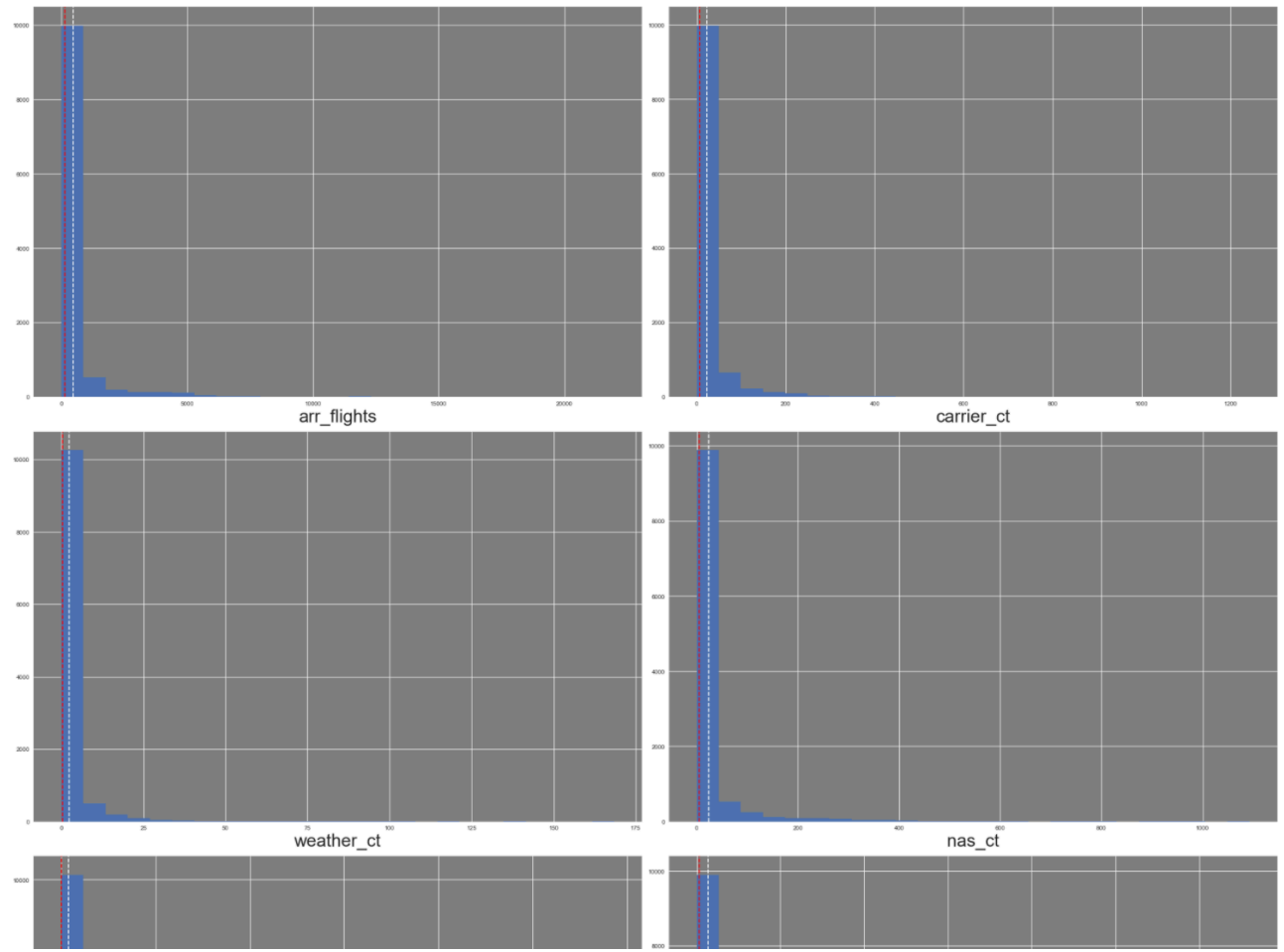
In [28]:
```
Denver_Airport, DEN_carriers_, DEN_carriers_full = zip(*Denver_grouping.sum().sort_values('carrier
_ct').index.values)
print(DEN_carriers_full)
```

```
('Virgin America', 'Alaska Airlines Inc.', 'JetBlue Airways', 'Spirit Air Lines', 'Delta Air Lines
Inc.', 'American Airlines Inc.', 'Frontier Airlines Inc.', 'SkyWest Airlines Inc.', 'United Air Li
nes Inc.', 'Southwest Airlines Co.')
```

In [29]:
```
height = Denver_grouping.sum().sort_values('carrier_ct').carrier_ct
bars = DEN_carriers_full
# use set slicing to show the list from most to least
y_pos = np.arange(len(bars))[::-1]
# increase the figure size
plt.figure(figsize=(20,10))
plt.barh(y_pos, height, color = "orange")
plt.yticks(y_pos, bars, size=20)
plt.title("Denver's Late Flights\n", size = 30)
plt.xlabel("lates", size=20)
plt.show()
```



In [31]:
```
rows = len(data[col_numeric].columns)//2
fig = plt.figure(figsize=(30, 10*rows))

for i, col in enumerate(data[col_numeric].columns):
    fig.add_subplot(rows, 2, i+1, facecolor='grey')
    plt.axvline(np.mean(data[col_numeric][col]), linestyle='--', color='w')
    plt.axvline((data[col_numeric][col]).median(), linestyle='--', color='r')
    data[col_numeric][col].hist(bins=25)
    plt.xlabel(col, size=30)
    plt.tight_layout()
```



Scale and Remove Outliers

Hack a scipy function to remove the outliers

```
from scipy import stats
df[(np.abs(stats.zscore(df)) < 3).all(axis=1)]
```

In [32]:
```
from scipy import stats
```

In [33]:
```
no_outliers = data[col_numeric][(np.abs(stats.zscore(data[col_numeric])) < 3).all(axis=1)]
```

In [34]:
```
no_outliers.head()
```

Out[34]:

	arr_flights	carrier_ct	weather_ct	nas_ct	security_ct	late_aircraft_ct	arr_cancelled	arr_diverted	arr_delay	carrier_delay
1	588.0	34.71	4.71	19.77	0.32	38.49	13.0	2.0	5170.0	1754.0
2	607.0	35.52	5.40	23.55	0.00	27.52	12.0	2.0	4485.0	1857.0
4	327.0	23.67	1.47	14.50	0.06	19.29	0.0	0.0	2276.0	985.0
7	661.0	50.08	7.39	15.91	0.00	35.61	8.0	1.0	6142.0	2707.0
9	1639.0	143.71	13.08	158.40	1.09	92.72	55.0	2.0	19038.0	8099.0

Scale

In [35]:
```
from sklearn.preprocessing import scale
no_outliers_scaled = pd.DataFrame(scale(no_outliers))
no_outliers_scaled.columns = no_outliers.columns
# no_outliers_scaled
```

In [36]:
```
# no_zeroes
no_zeroes_scaled = no_outliers_scaled[no_outliers_scaled > 0]
```

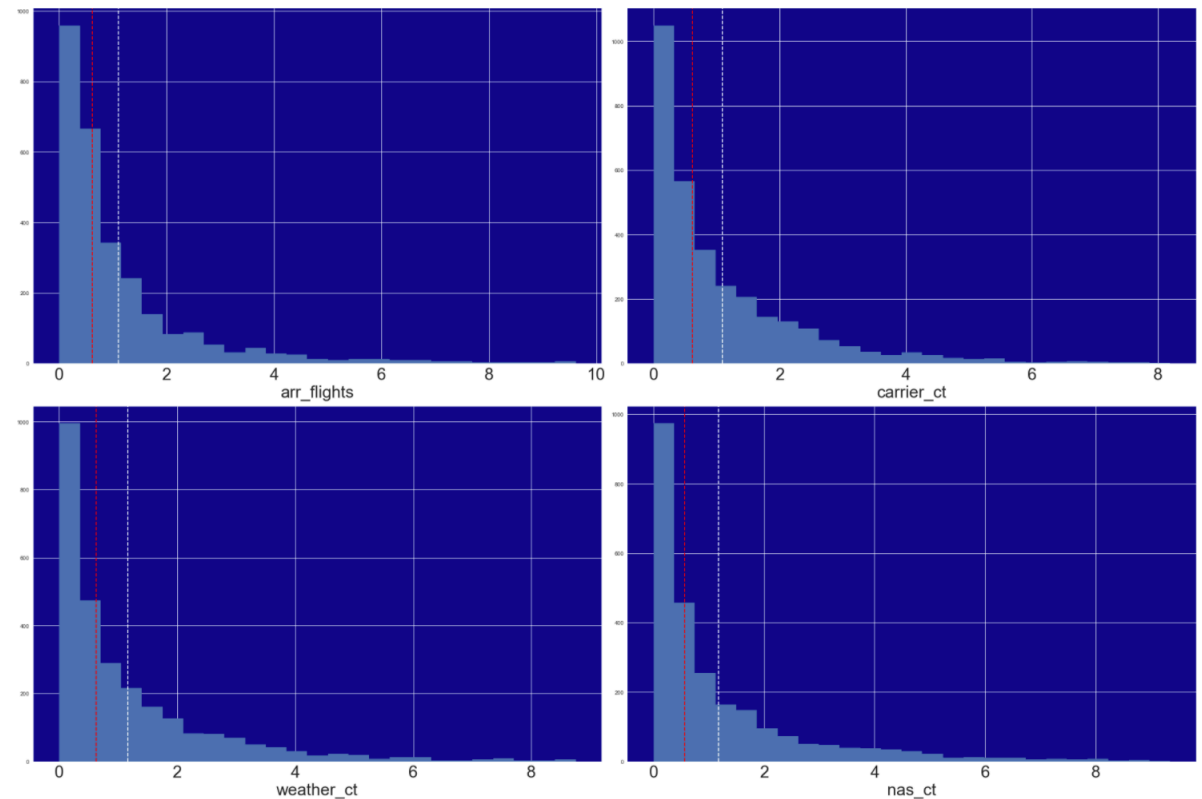
In [37]:
```
# it wold also be useful to have this variable not scaled, so we can see the true values in a plo
t.
data_no_outliers = no_outliers[no_outliers > 0]
```

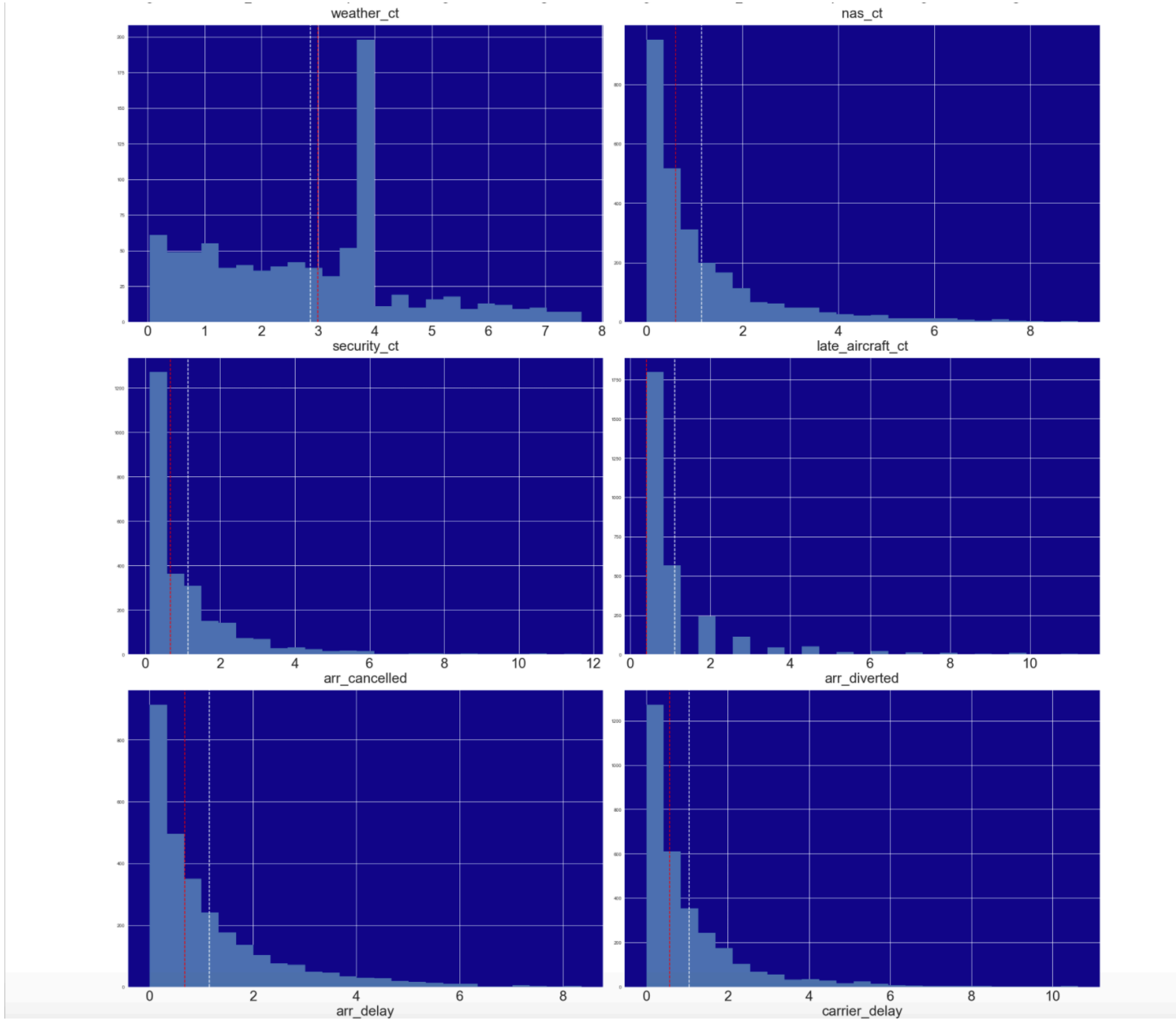
Re-plot Histograms

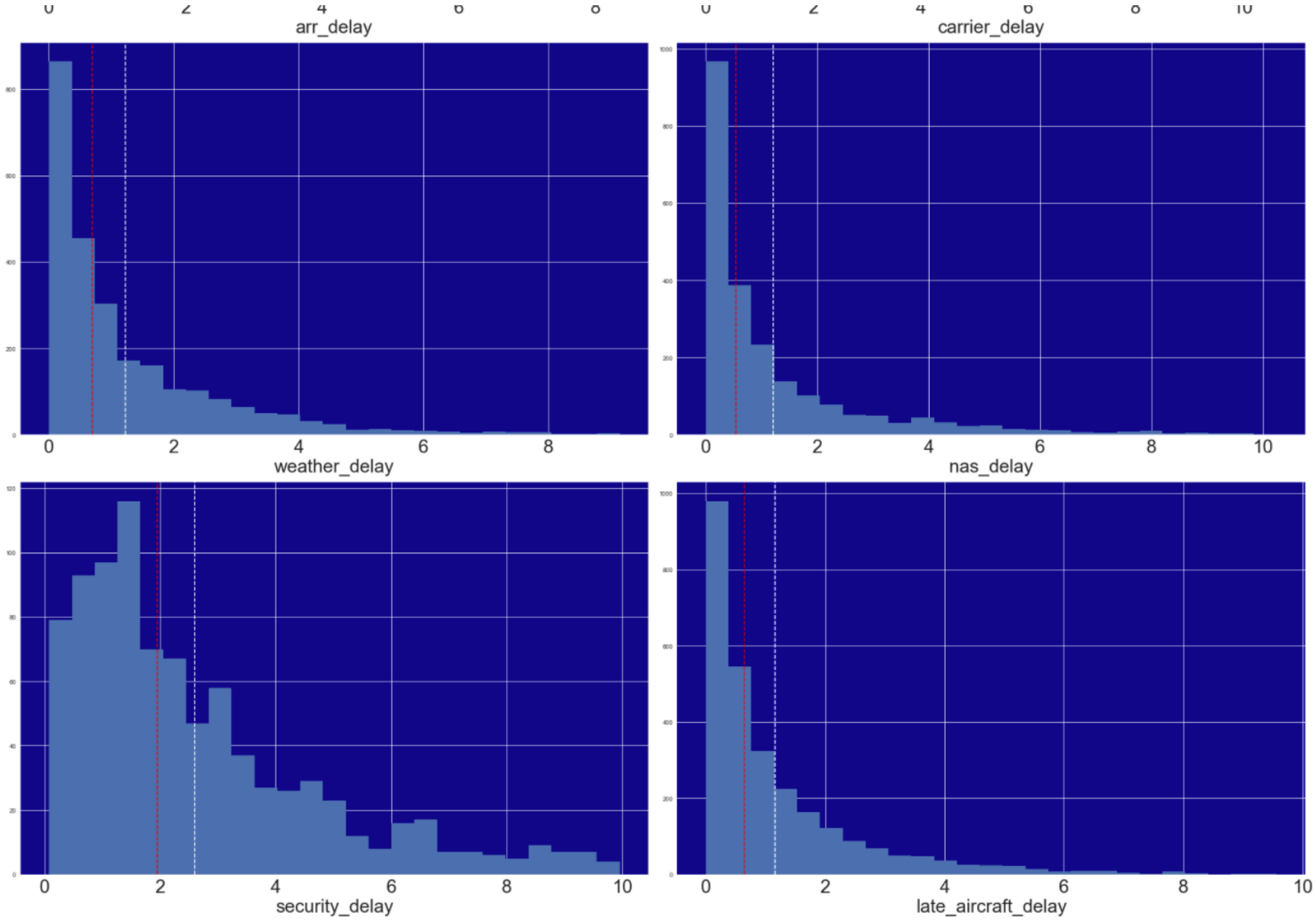
mean: white line

median: red line

In [38]:
```
rows = len(no_zeroes_scaled.columns)//2
fig = plt.figure(figsize=(30, 10*rows))

for i, col in enumerate(no_zeroes_scaled.columns):
    fig.add_subplot(rows, 2, i+1, facecolor='darkblue')
    plt.axvline(np.mean(no_zeroes_scaled[col]), linestyle='--', color='w')
    plt.axvline((no_zeroes_scaled[col]).median(), linestyle='--', color='r')
    no_zeroes_scaled[col].hist(bins=25, normed=False)
    plt.xticks(size = 30)
    plt.xlabel(col, size=30)
    plt.tight_layout()
```




weather_ct
nas_ct
security_ct
late_aircraft_ct
arr_cancelled
arr_diverted
arr_delay
carrier_delay


arr_delay
carrier_delay
weather_delay
nas_delay
security_delay
late_aircraft_delay

What the histograms tell us now is based on their mean and median

In [40]:
```
for i, col in enumerate(data_no_outliers.columns):
    h = data_no_outliers[col].median()
    print ("%s: %f" %(data_no_outliers.columns[i], h))
```

```
arr_flights: 120.000000
carrier_ct: 7.510000
weather_ct: 1.230000
nas_ct: 4.950000
security_ct: 0.740000
late_aircraft_ct: 6.740000
arr_cancelled: 3.000000
arr_diverted: 1.000000
arr_delay: 1104.000000
carrier_delay: 451.000000
weather_delay: 94.000000
nas_delay: 180.000000
security_delay: 20.000000
late_aircraft_delay: 427.000000
```

In [41]:
```
for i, col in enumerate(data_no_outliers.columns):
    h = data_no_outliers[col].mean()
    print ("%s: %f" %(data_no_outliers.columns[i], h))
```

```
arr_flights: 250.819761
carrier_ct: 14.393441
weather_ct: 2.365284
nas_ct: 12.766646
security_ct: 0.713855
late_aircraft_ct: 15.994281
arr_cancelled: 4.980853
arr_diverted: 1.852850
arr_delay: 2505.348665
carrier_delay: 916.270884
weather_delay: 202.176000
nas_delay: 561.991328
security_delay: 26.265127
late_aircraft_delay: 1050.251560
```

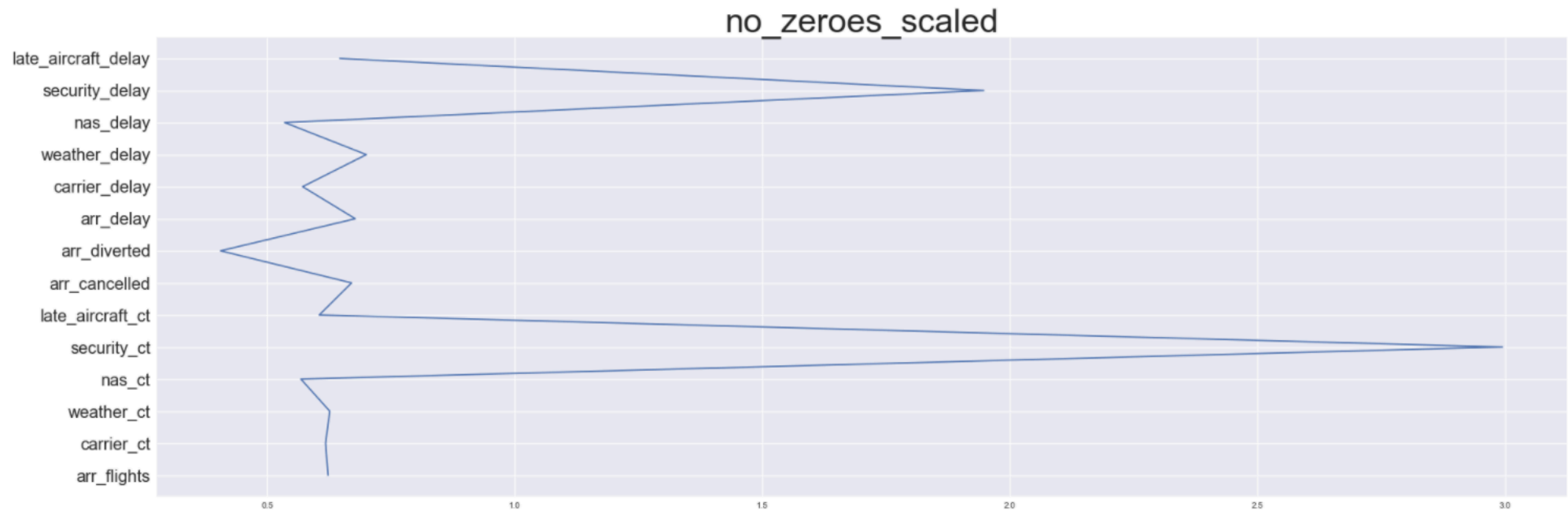
Plot the range of medians

In [42]:
```
# Graph shows range of the medians of the numeric features
plt.figure(figsize=(30,10))
y = np.array(data_no_outliers.median())
x = np.array(range(len(data_no_outliers.columns)))
my_yticks = data_no_outliers.columns
plt.xticks(y, size=15)
plt.yticks(x, my_yticks, size=20)
plt.title('Medians of data_no_outliers', size=40)

plt.plot(y, x)
```

Out[42]:
```
[<matplotlib.lines.Line2D at 0x1257448d0>]
```

In [43]:
```
# no_zeroes_scaled
# Graph shows range of the medians of the numeric features
plt.figure(figsize=(30,10))
y = np.array(no_zeroes_scaled.median())
x = np.array(range(len(no_zeroes_scaled.columns)))
my_yticks = no_zeroes_scaled.columns
# plt.xticks(y, size=15)
plt.yticks(x, my_yticks, size=20)
plt.title('no_zeroes_scaled', size=40)
plt.plot(y, x)
```

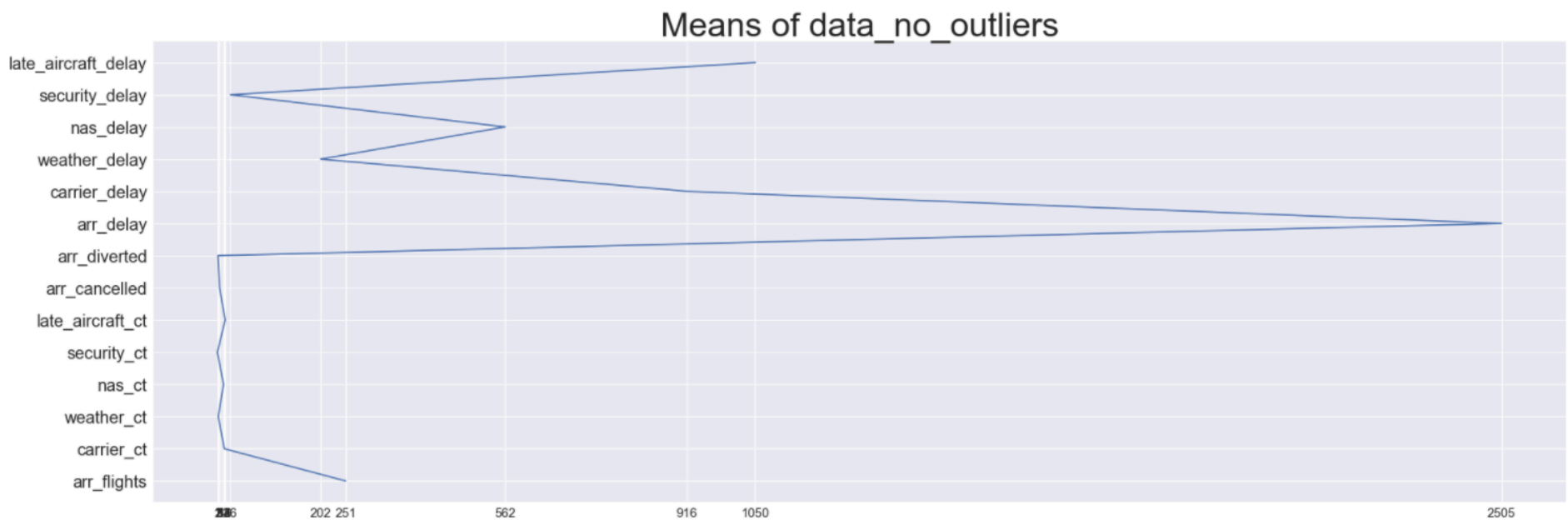
Out[43]:
```
[<matplotlib.lines.Line2D at 0x11bd5ea50>]
```



Plot the range of means

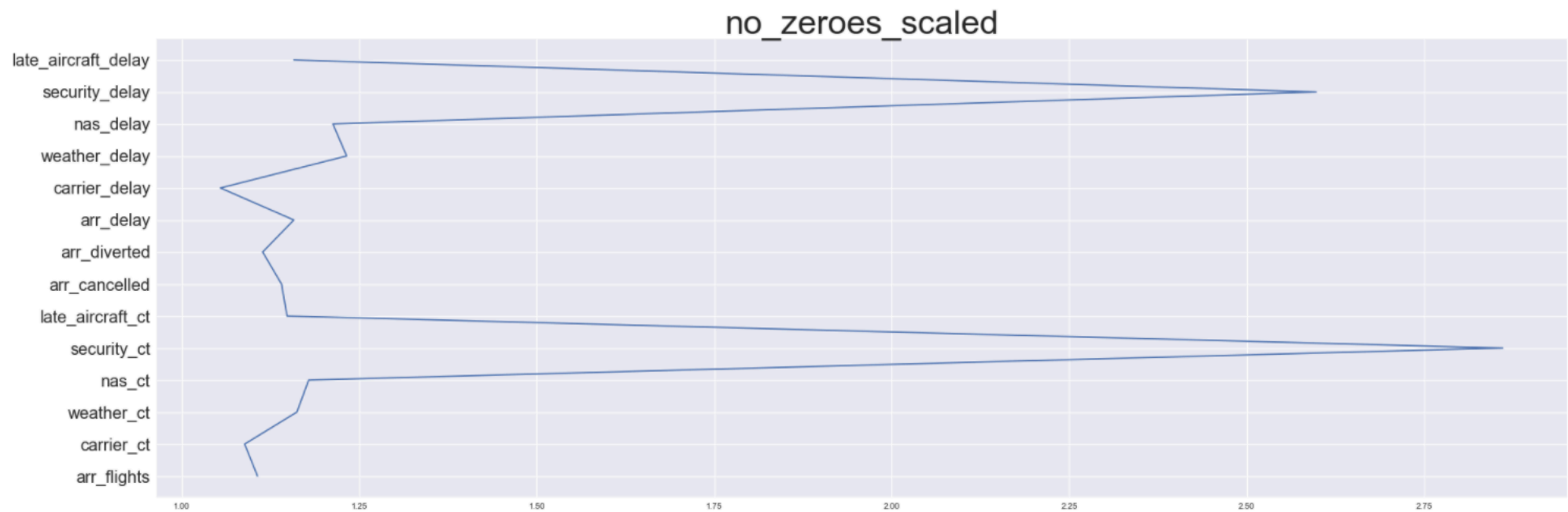
In [44]:
```
# Graph shows range of the medians of the numeric features
plt.figure(figsize=(30,10))
y = np.array(data_no_outliers.mean())
x = np.array(range(len(data_no_outliers.columns)))
my_yticks = data_no_outliers.columns
plt.xticks(y, size=15)
plt.yticks(x, my_yticks, size=20)
plt.title('Means of data_no_outliers', size=40)

plt.plot(y, x)
```

Out[44]:
```
[<matplotlib.lines.Line2D at 0x122bb8550>]
```



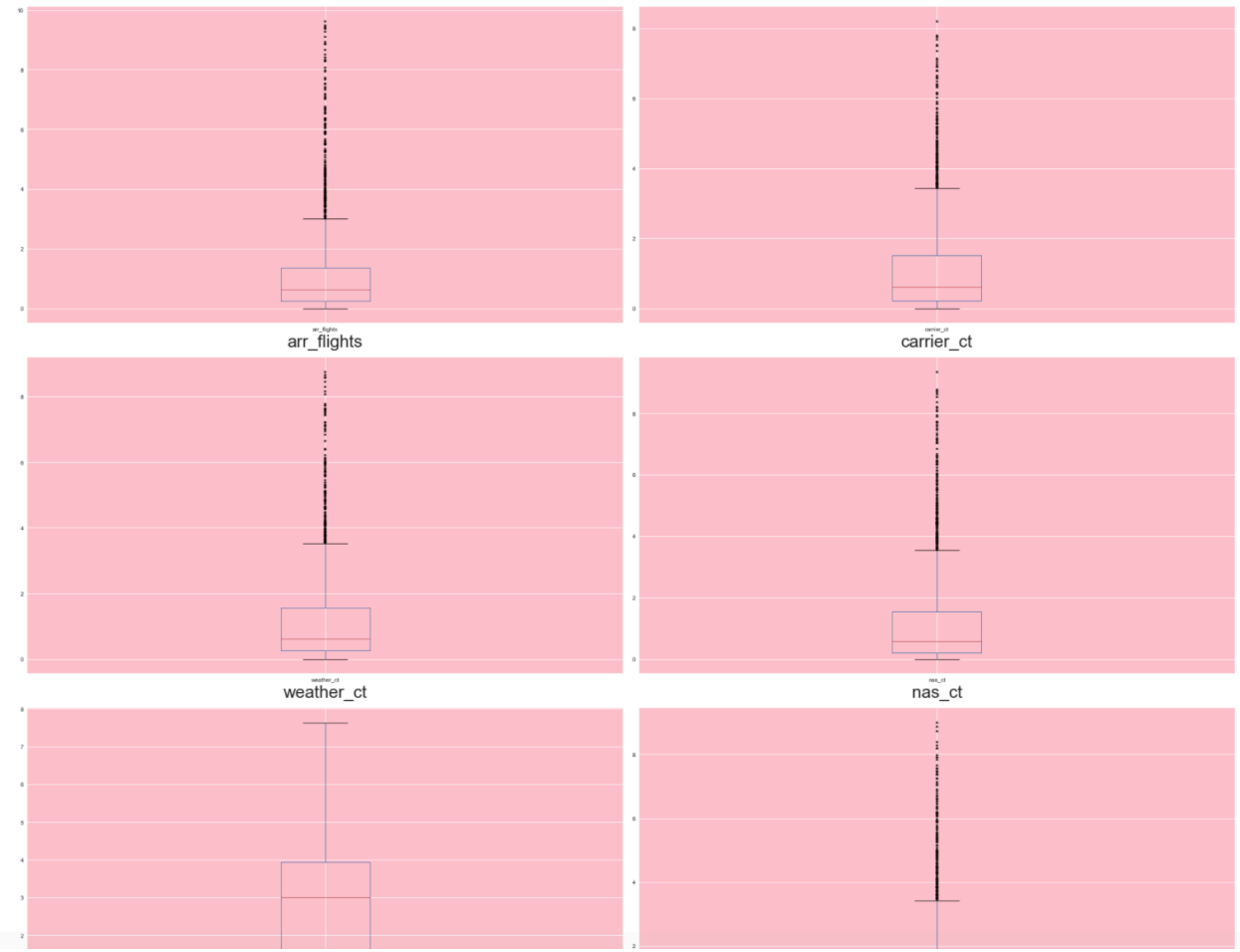
In [45]:
```
# no_zeroes_scaled
# Graph shows range of the medians of the numeric features
plt.figure(figsize=(30,10))
y = np.array(no_zeroes_scaled.mean())
x = np.array(range(len(no_zeroes_scaled.columns)))
my_yticks = no_zeroes_scaled.columns
# plt.xticks(y, size=15)
plt.yticks(x, my_yticks, size=20)
plt.title('no_zeroes_scaled', size=40)
plt.plot(y, x)
```

Out[45]:
```
[<matplotlib.lines.Line2D at 0x1220c0350>]
```



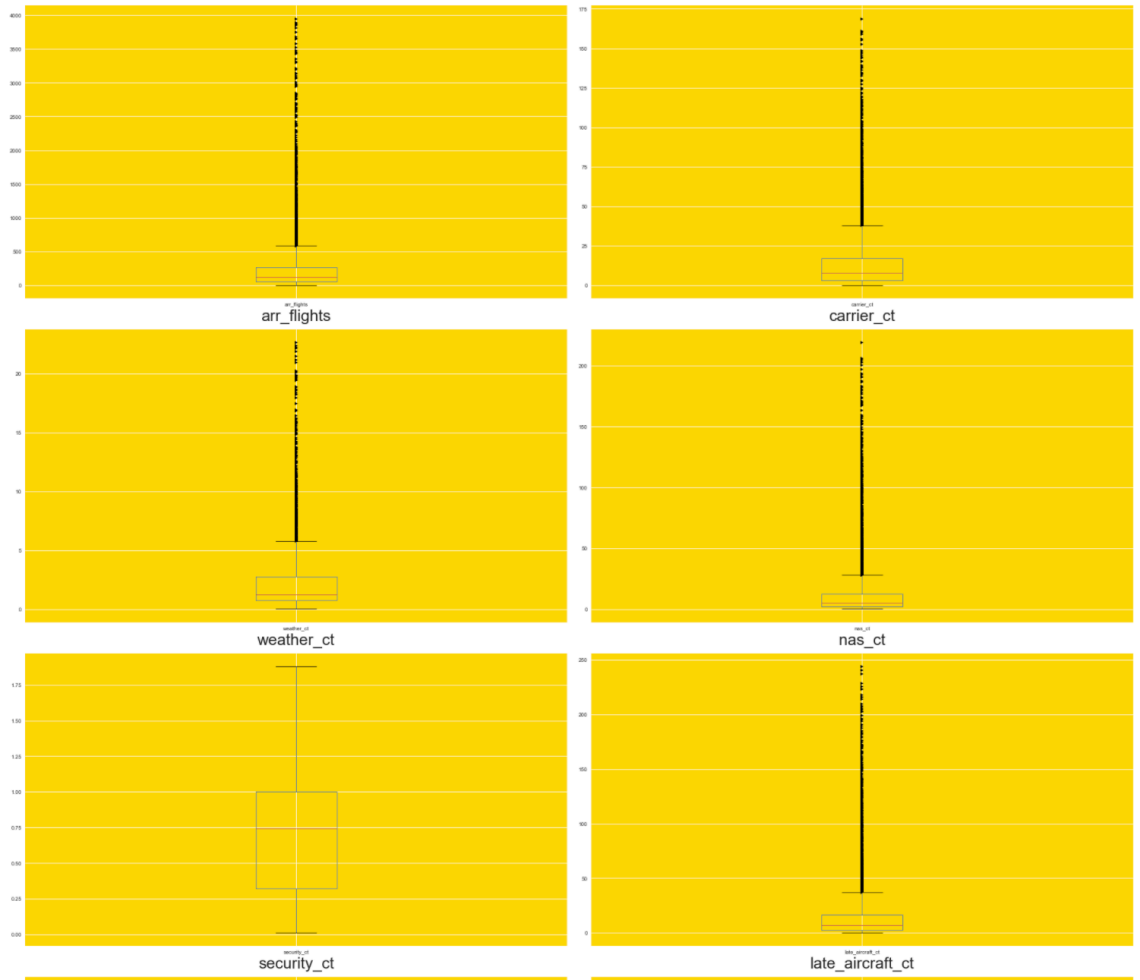
Boxplots I

In [49]:
```
box_rows = len(no_zeroes_scaled.columns)//2
fig = plt.figure(figsize=(30, 60))

for i, col in enumerate(no_zeroes_scaled):
    fig.add_subplot(box_rows, 2, i+1, facecolor="pink")
    no_zeroes_scaled[col].plot.box(sym='k*')
    plt.xlabel(col, size=30)
    plt.tight_layout()
```



Boxplots II

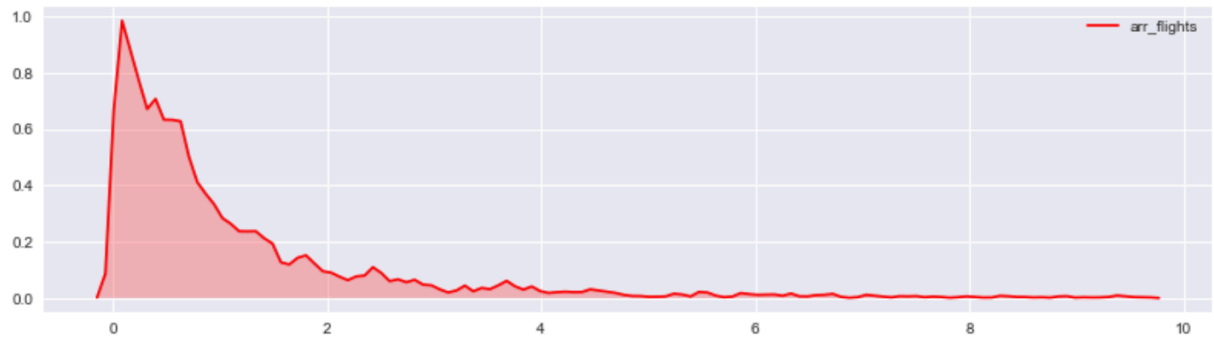
In [52]:
```
box_rows = len(data_no_outliers.columns)//2
fig = plt.figure(figsize=(30, 60))

for i, col in enumerate(data_no_outliers):
    fig.add_subplot(box_rows, 2, i+1, facecolor='gold')
    data_no_outliers[col].plot.box(sym='k>')
    plt.xlabel(col, size=30)
    plt.tight_layout()
```

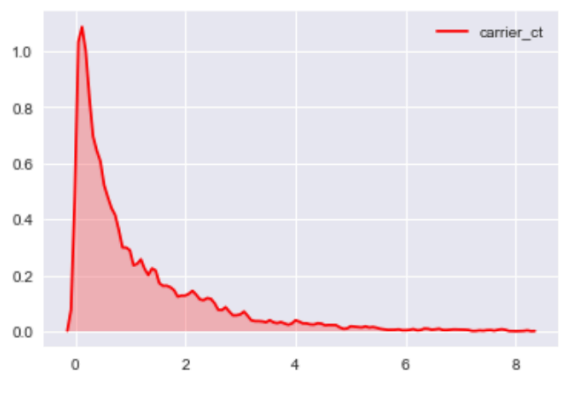


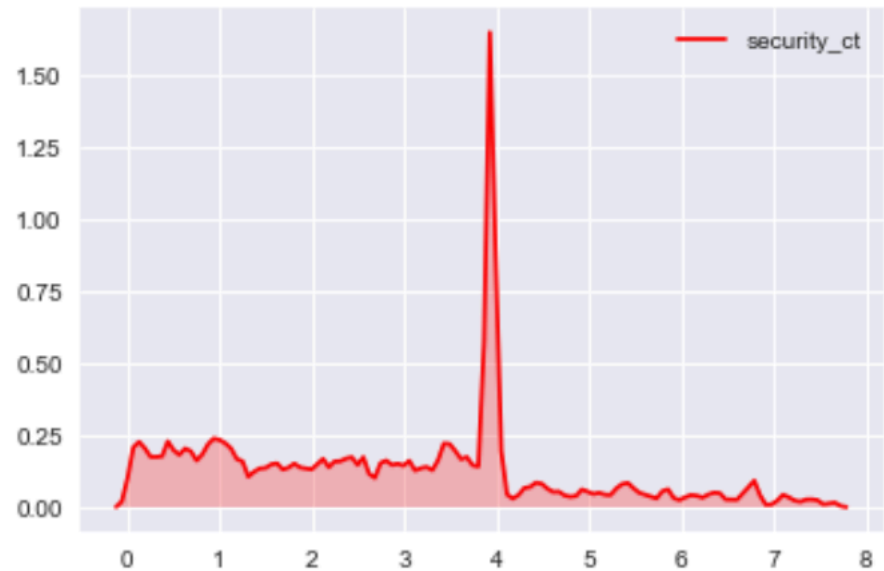
Kernel Density Plots

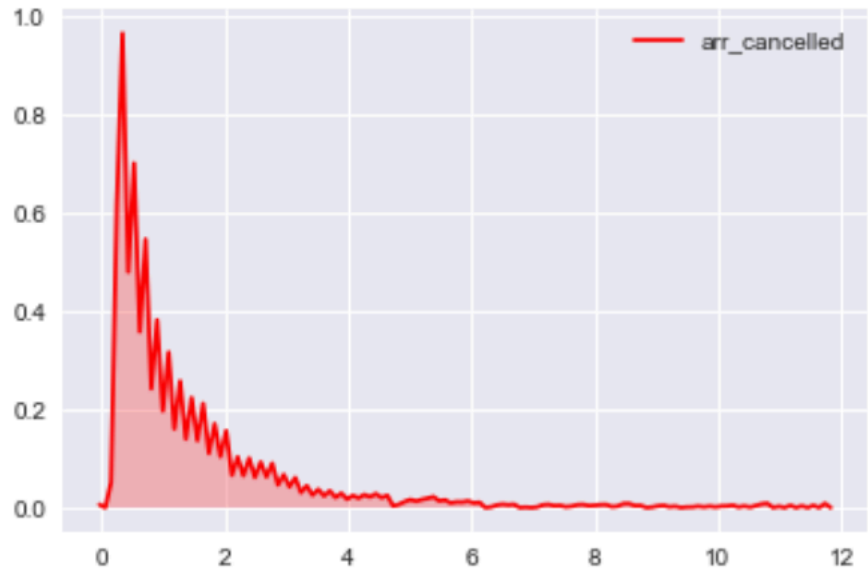
In [53]:
```
box_rows = len(no_zeroes_scaled.columns)//2
fig = plt.figure(figsize=(30, 30))

# ax.set_aspect("equal")
for i, dd in enumerate(no_zeroes_scaled.columns):
    fig.add_subplot(box_rows, 2, i+1)
    sns.kdeplot(no_zeroes_scaled[dd], shade=True, bw=.05, color="red")
    plt.show()
```

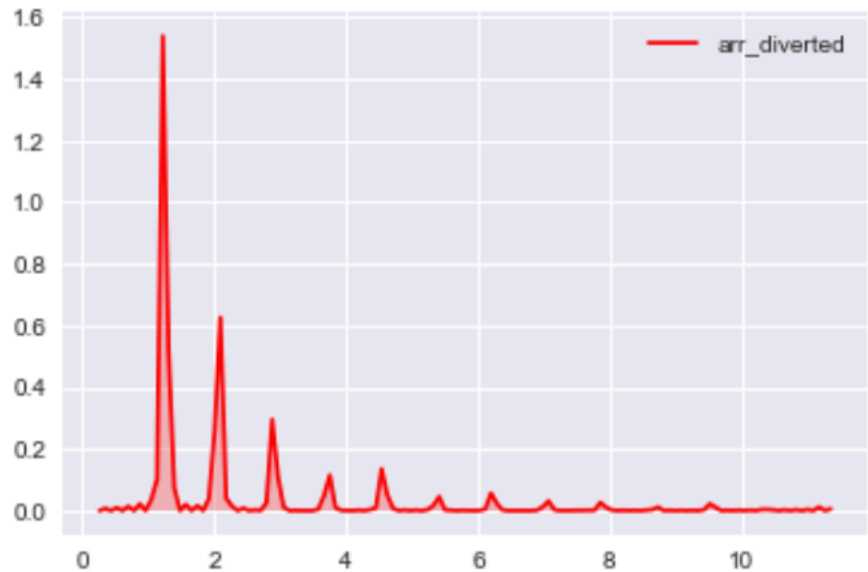
```
/Users/jg/anaconda/lib/python2.7/site-packages/statsmodels/nonparametric/kde.py:454: RuntimeWarnin
g: invalid value encountered in greater
  X = X[np.logical_and(X>clip[0], X<clip[1])] # won't work for two columns.
/Users/jg/anaconda/lib/python2.7/site-packages/statsmodels/nonparametric/kde.py:454: RuntimeWarnin
g: invalid value encountered in less
  X = X[np.logical_and(X>clip[0], X<clip[1])] # won't work for two columns.
```





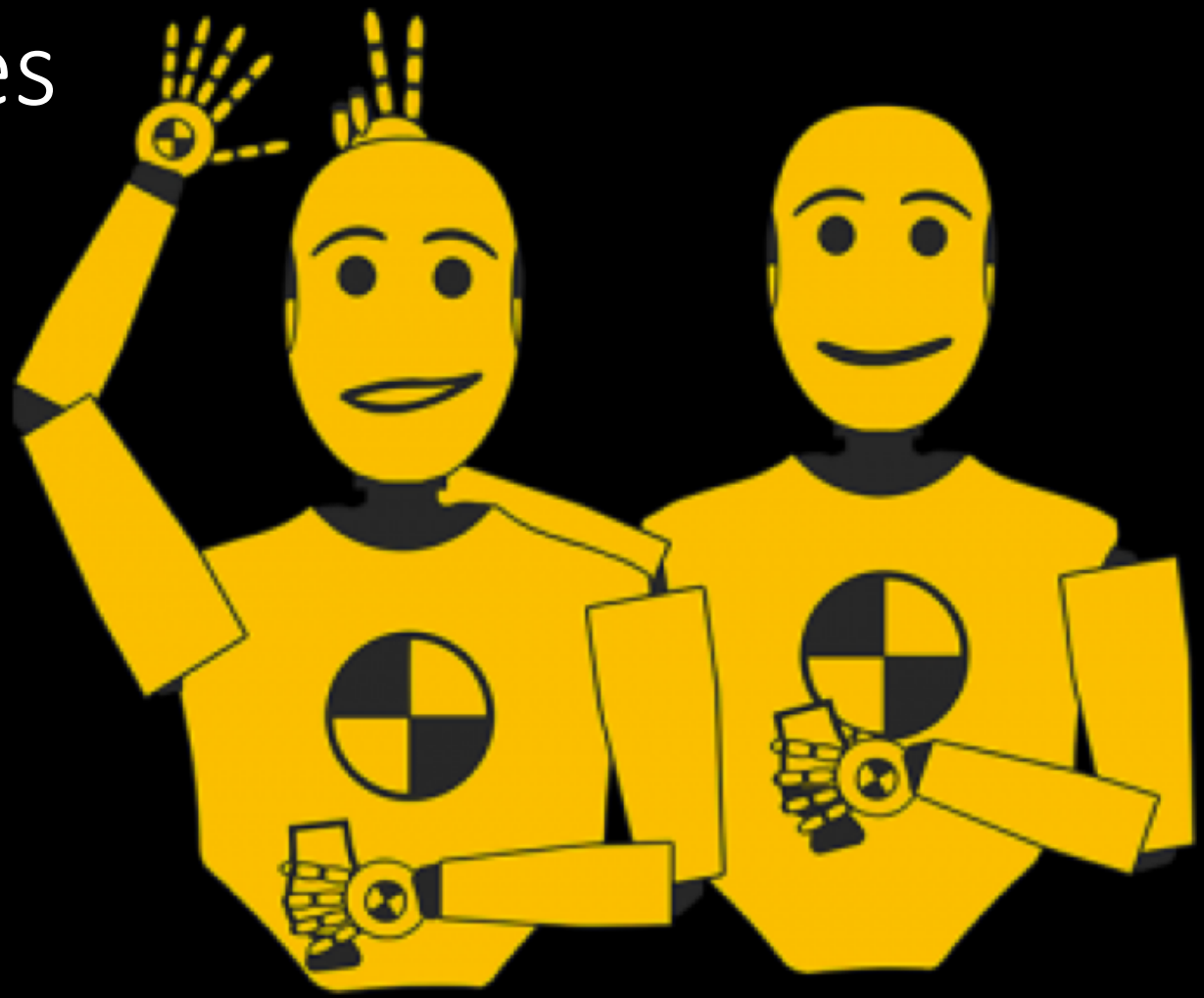

security_ct
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
1
2
3
4
5
6
7
8


arr_cancelled
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10
12


arr_diverted
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10


security_delay
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
2
4
6
8
10

Dummies



Display the columns to be transformed into dummies in violet

In [5]:
```
# the `data2` variable holds the `combined_list` because when we color the redundant columns, the
# programming that does the coloring disables other DataFrame methods and functions we will need for
# further programming... need to do %store -r data; %store -r combined_list;
# then run this cell
data_for_dummies = data[combined_list].sample(5)
# function examines value's length, and returns red if its too long
def if_str_violet(val):
    val = val; color = 'darkviolet' if type(val) == str else 'black'; return 'color: %s' % color
# apply the function
red_dummies = data_for_dummies.style.applymap(if_str_violet)
red_dummies
```

Out[5]:

	year	month	carrier	airport	arr_flights	carrier_ct	weather_ct	nas_ct	security_ct	late_aircraft_ct	arr_cancelled	arr_diverted	arr_delay	carrier_delay
8019	2016	8	VX	IAD	172	10.71	2.74	5.59	0.84	13.12	1	0	1403	397
5326	2016	6	DL	MOB	30	5.22	0	1.28	0	1.5	0	0	467	300
1102	2016	2	AS	SEA	4207	107.21	7.27	188.88	3.73	110.9	17	7	21141	6169
632	2016	1	OO	TUS	631	40.12	1.71	18.06	1.76	44.34	2	0	6336	3107
6997	2016	7	WN	ABQ	1039	98.87	1.46	23.86	0.07	184.75	18	2	19030	5589

In [6]:
```
%store data_for_dummies
```

```
Stored 'data_for_dummies' (DataFrame)
```

Dummies for months

In [22]:
```
# since 1 is first month without calling drop_first=True, it is not present for concatenation
# wiht the datset used for supervised machine learning algorithms, however,
# we may use the commented out code in order to include 1 (January)
# months_dummies = pd.get_dummies(data[combined_list].month, prefix='months', prefix_sep ='_')
months_dummies = pd.get_dummies(data[combined_list].month, prefix='months', prefix_sep ='_', drop_first=True)
months_dummies.head()
```

Out[22]:

	months_2	months_3	months_4	months_5	months_6	months_7	months_8	months_9	months_10	months_11
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0

In [23]:
```
%store months_dummies
```

```
Stored 'months_dummies' (DataFrame)
```

Use pandas to get dummies for `data.airport`

In [11]:
```
# since ABE is first without calling drop_first=True, it is not present for concatenation
# wiht the datset used for supervised machine learning algorithms, however,
# we may use the commented out code in order to include ABE
# airport_dummies = pd.get_dummies(data.airport, prefix='airport', prefix_sep ='_')

airport_dummies = pd.get_dummies(data[combined_list].airport, prefix='airport', prefix_sep ='_', drop_first=True)
airport_dummies.columns
```

Out[11]:
```
Index([u'airport_ABI', u'airport_ABQ', u'airport_ABR', u'airport_ABY',
       u'airport_ACK', u'airport_ACT', u'airport_ACV', u'airport_ACY',
       u'airport_ADK', u'airport_ADQ',
       ...
       u'airport_TYR', u'airport_TYS', u'airport_UST', u'airport_VLD',
       u'airport_VPS', u'airport_WRG', u'airport_WYS', u'airport_XNA',
       u'airport_YAK', u'airport_YUM'],
      dtype='object', length=309)
```

Use pandas to get dummies for `data.carrier`

In [9]:
```
# since AA is first without caling drop_first=True, it is not present for concatenation
# with the dataset used for supervised machine learning algorithms, however,
# we may use the commented out code in order to include AA
# carrier_dummies = pd.get_dummies(data.carrier, prefix='carrier', prefix_sep='_')

carrier_dummies = pd.get_dummies(data[combined_list].carrier, prefix='carrier', prefix_sep='_', drop_first=True)
carrier_dummies.columns
```

Out[9]:
```
Index([u'carrier_AS', u'carrier_B6', u'carrier_DL', u'carrier_EV',
       u'carrier_F9', u'carrier_HA', u'carrier_NK', u'carrier_OO',
       u'carrier_UA', u'carrier_VX', u'carrier_WN'],
      dtype='object')
```

In [26]:
```
len(data_full_02.columns)
```

Out[26]: 349

In [27]:
```
data_for_models = data_full_02.drop(['year','carrier', 'airport', 'month'], axis=1)
```

In [28]:
```
len(data_for_models.columns)
```

Out[28]: 345

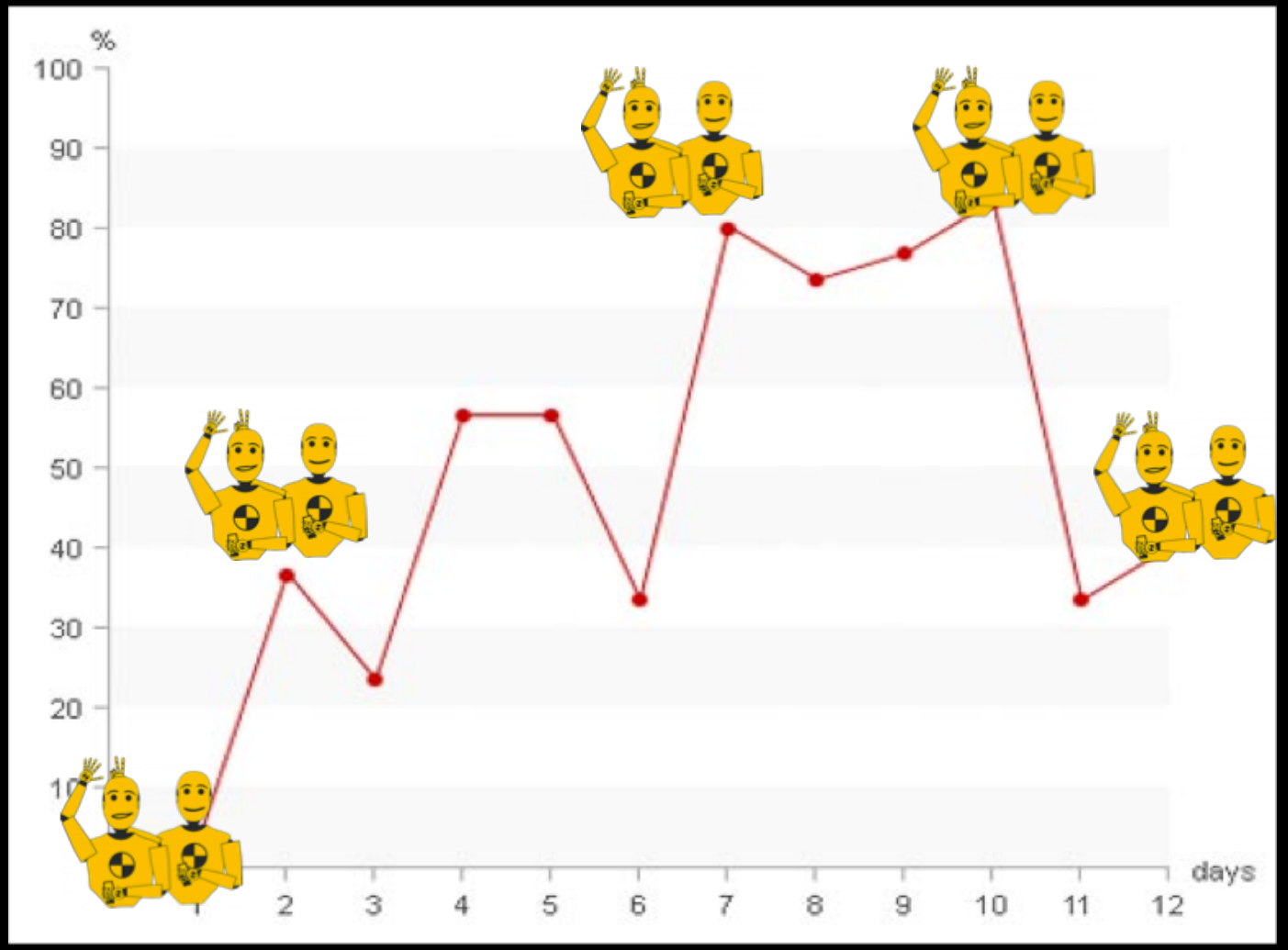
Store the variables

In [29]:
```
%store data_for_models
```

```
Stored 'data_for_models' (DataFrame)
```

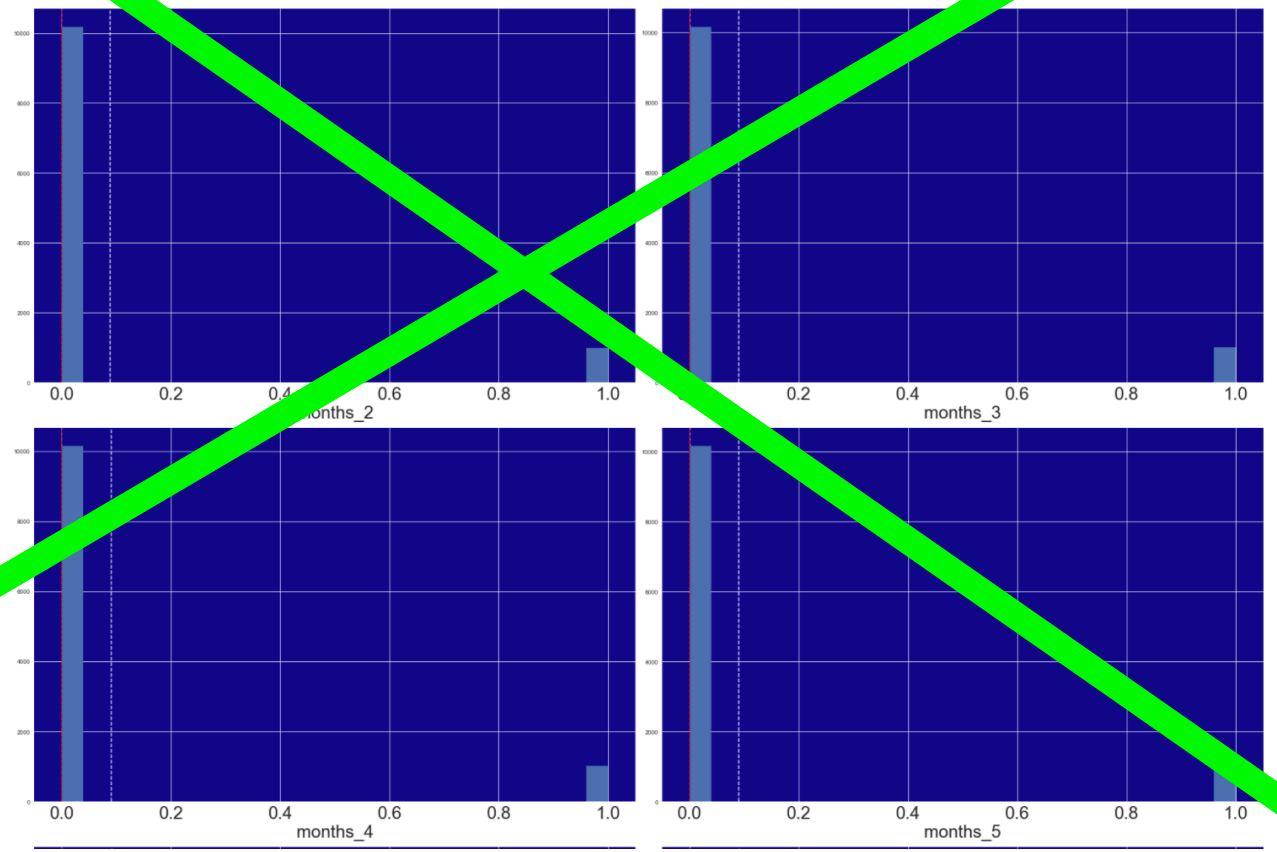
In [ ]:
```

```

Re-do Plots with Dummies



In [21]:
```
%store -r months_dummies
```

In [23]:
```
rows = len(months_dummies.columns)//2
fig = plt.figure(figsize=(30, 10*rows))

for i, col in enumerate(months_dummies.columns):
    fig.add_subplot(rows, 2, i+1, facecolor='darkblue')
    plt.axvline(np.mean(months_dummies[col]), linestyle='--', color='w')
    plt.axvline((months_dummies[col]).median(), linestyle='--', color='r')
    months_dummies[col].hist(bins=25, normed=False)
    plt.xticks(size = 30)
    plt.xlabel(col, size=30)
    plt.tight_layout()
plt.s
```



Gradient Boost

- Raw Data
- Dummies data – filtered
- Dummies data -- unfiltered

Use some raw data to get a first look

Combine `col_numeric` and `col_target` to get a datset this modelling can work with

In [4]:
```
pre_combine_00 = [col_numeric, col_target]
combined_list_for_xgboost = [item for sublist in pre_combine_00 for item in sublist]
# combined_list_for_xgboost
```

In [5]:
```
len(data[combined_list_for_xgboost].columns)
```

Out[5]: 15

`data[combined_list_for_xgboost]` is our clean dataset, since it contains only numeric data and the target

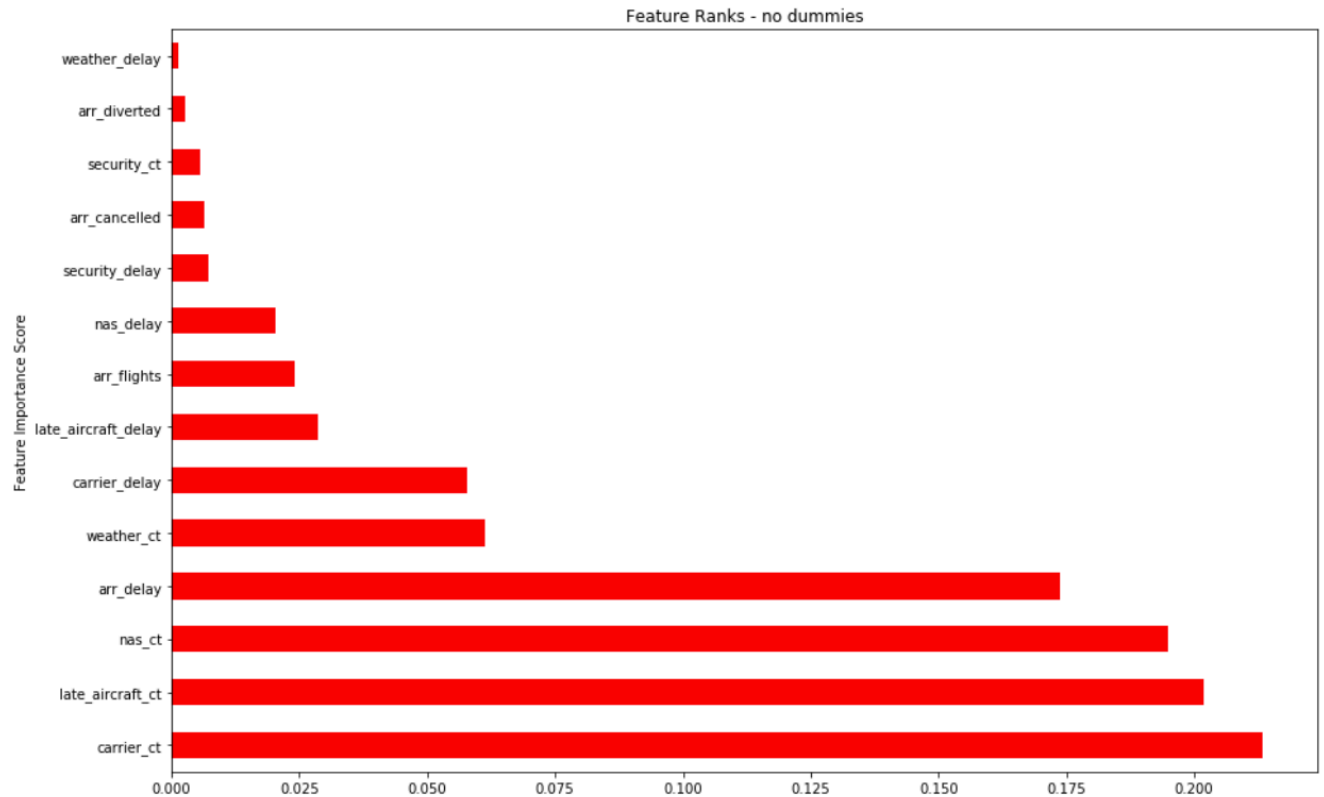
In [7]:
```
data[combined_list_for_xgboost].head()
```

Out[7]:

	arr_flights	carrier_ct	weather_ct	nas_ct	security_ct	late_aircraft_ct	arr_cancelled	arr_diverted	arr_delay	carrier_del
0	11956.0	507.33	39.83	452.65	4.72	529.47	201.0	9.0	106950.0	50027.0
1	588.0	34.71	4.71	19.77	0.32	38.49	13.0	2.0	5170.0	1754.0
2	607.0	35.52	5.40	23.55	0.00	27.52	12.0	2.0	4485.0	1857.0
3	1595.0	117.00	10.26	117.66	0.83	89.25	137.0	6.0	23698.0	9681.0
4	327.0	23.67	1.47	14.50	0.06	19.29	0.0	0.0	2276.0	985.0

In [10]:
```
#Choose all predictors except target & IDcols
predictors = [x for x in train.columns if x not in [target, IDcol]]
gbm0 = GradientBoostingClassifier(random_state=10)
modelfit(gbm0, train, predictors)
```

```
Model Report
Accuracy : 0.9975
AUC Score (Train): 0.999975
CV Score : Mean - 0.9994707 | Std - 0.0001631409 | Min - 0.9992015 | Max - 0.9996357
```



In [11]:
```
# get the list of features
print(predictors)
```

```
['arr_flights', 'carrier_ct', 'weather_ct', 'nas_ct', 'security_ct', 'late_aircraft_ct', 'arr_canc
elled', 'arr_diverted', 'arr_delay', 'carrier_delay', 'weather_delay', 'nas_delay', 'security_dela
y', 'late_aircraft_delay']
```

Now let's try this on the dataset that has all the dummies

Retrieve the stored variable `data_for_models`

In [12]:
```
%store -r data_for_models
```

Remove the `year` column so it won't affect the model

In [13]:
```
# run this only once or it throws an error
data_for_models = data_for_models.drop('year', axis=1)
data_for_models.head()
```

Out[13]:

	arr_flights	carrier_ct	weather_ct	nas_ct	security_ct	late_aircraft_ct	arr_cancelled	arr_diverted	arr_delay	carrier_del
0	11956.0	507.33	39.83	452.65	4.72	529.47	201.0	9.0	106950.0	50027.0
1	588.0	34.71	4.71	19.77	0.32	38.49	13.0	2.0	5170.0	1754.0
2	607.0	35.52	5.40	23.55	0.00	27.52	12.0	2.0	4485.0	1857.0
3	1595.0	117.00	10.26	117.66	0.83	89.25	137.0	6.0	23698.0	9681.0
4	327.0	23.67	1.47	14.50	0.06	19.29	0.0	0.0	2276.0	985.0

5 rows × 345 columns

In [14]:
```
len(data_for_models.columns)
```

Out[14]:
```
345
```

Function defines XGBoost parameters - II

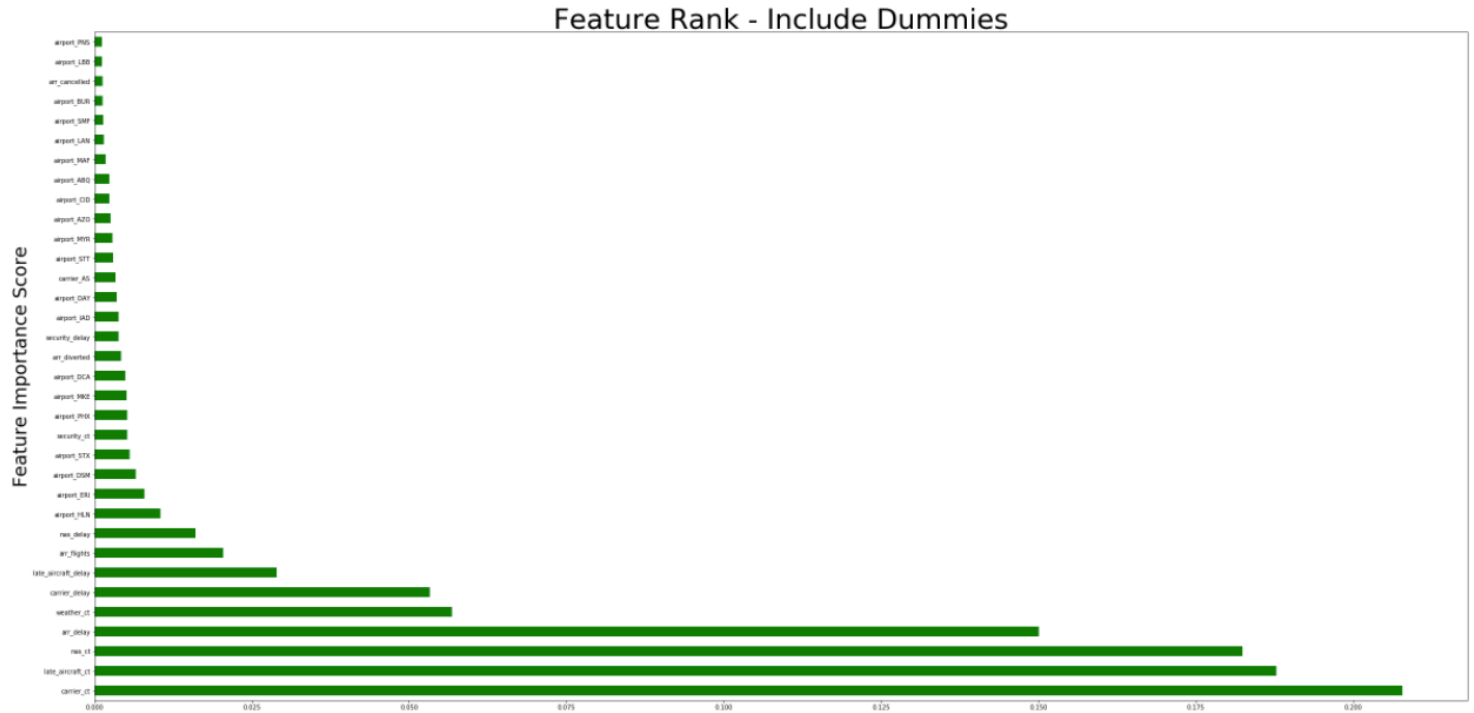
In [17]:
```
def modelfit2(alg, dtrain, predictors, performCV=True, printFeatureImportance=True, cv_folds=5):
    #Fit the algorithm on the data
    alg.fit(dtrain[predictors], dtrain.target)

    #Predict training set:
    dtrain_predictions = alg.predict(dtrain[predictors])
    dtrain_predprob = alg.predict_proba(dtrain[predictors])[:,1]

    #Perform cross-validation:
    if performCV:
        cv_score = cross_validation.cross_val_score(alg, dtrain[predictors], dtrain.target, cv=cv_
folds, scoring='roc_auc')

    #Print model report:
    print "\nModel Report"
    print "Accuracy : %.4g" % metrics.accuracy_score(dtrain.target.values, dtrain_predictions)
    print "AUC Score (Train): %f" % metrics.roc_auc_score(dtrain.target, dtrain_predprob)

    if performCV:
        print "CV Score : Mean - %.7g | Std - %.7g | Min - %.7g | Max - %.7g" % (np.mean(cv_score
),np.std(cv_score),np.min(cv_score),np.max(cv_score))

    #Print Feature Importance:
    if printFeatureImportance:
        feat_imp = pd.Series(alg.feature_importances_, predictors).sort_values(ascending=False)
        feature_catcher_xx.append(feat_imp)
        feat_imp.nlargest(len(train2.columns)//10).plot(kind='barh', color='green')
        plt.title('Feature Rank - Include Dummies', size=45)
        plt.ylabel('Feature Importance Score', size=30 )
```

Adjust the Display Size, since there are many features...

In [18]:
```
rcParams['figure.figsize'] = 40,20
```

In [19]:
```
#Choose all predictors except target & IDcols
predictors2 = [x for x in train2.columns if x not in [target2, IDcol2]]
gbm2 = GradientBoostingClassifier(random_state=10)
modelfit2(gbm2, train2, predictors2)
```

```
Model Report
Accuracy : 0.998
AUC Score (Train): 0.999978
CV Score : Mean - 0.9994289 | Std - 0.0001499955 | Min - 0.9992124 | Max - 0.9996124
```



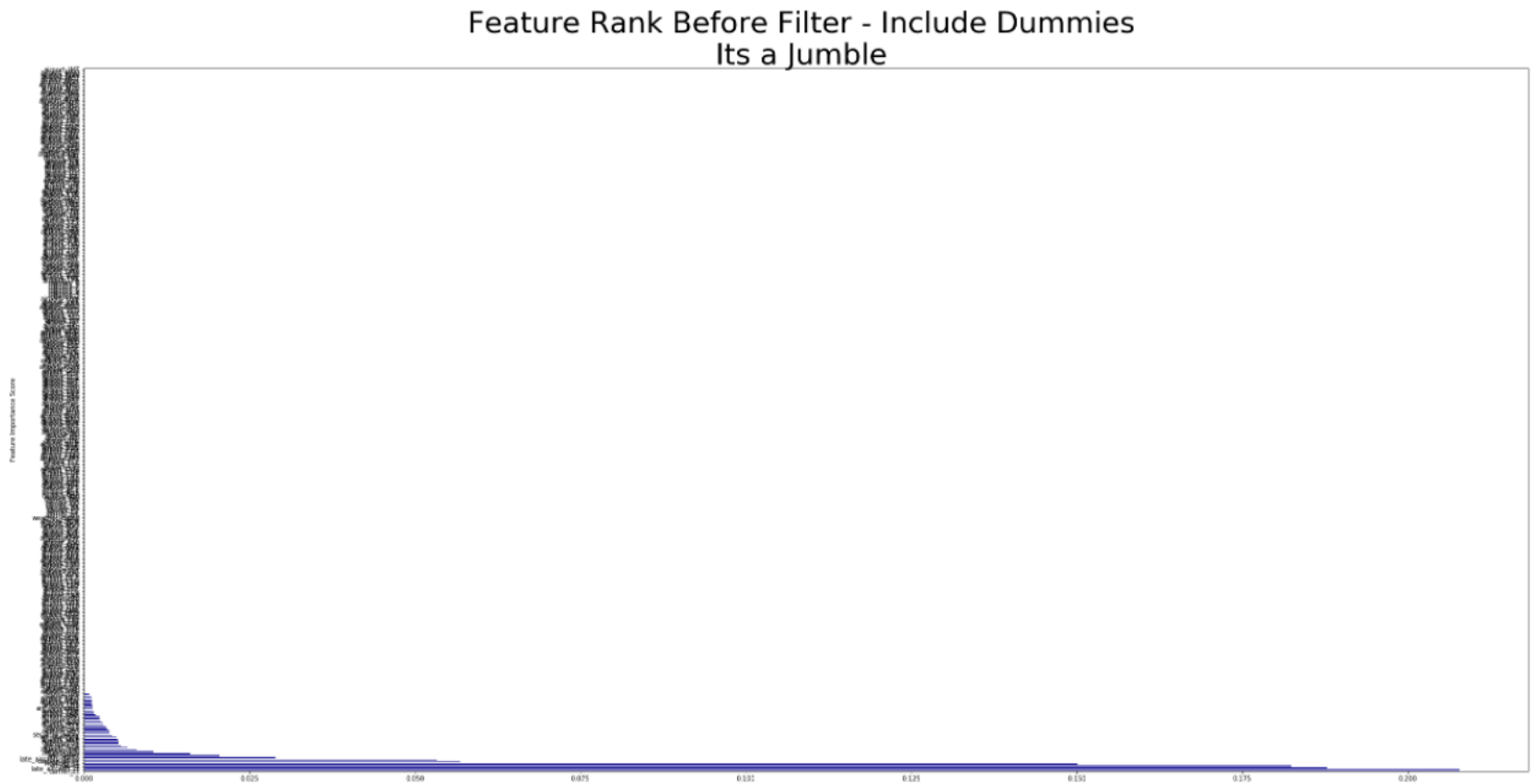
Some commented out code to inspect the output

In [20]:
```
# get the list of features
# print(predictors2)
```

In [21]:
```
# feature_catcher_xx
```

```
#
#Choose all predictors except target & IDcols
predictors3 = [x for x in train3.columns if x not in [target3, IDcol3]]
gbm3 = GradientBoostingClassifier(random_state=10)
modelfit3(gbm3, train3, predictors3)
```

```
Model Report
Accuracy : 0.998
AUC Score (Train): 0.999978
CV Score : Mean - 0.9994289 | Std - 0.0001499955 | Min - 0.9992124 | Max - 0.9996124
```



Fun with Graphing

Airports and Airport Codes

lists of airports for plotting on map

In [2]:
```
import re
import pandas as pd
import numpy as np
from pygeocoder import Geocoder

from mpl_toolkits.basemap import Basemap
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf', 'svg')
%matplotlib inline
```

In [6]:
```
# this stupid display always changes sizes
# use the RC params to change it
plt.rcParams["figure.figsize"] = [22, 10]
```

In [7]:
```
# set the map range of coordinates to display mUSA
mUSA = Basemap(projection='mill', llcrnrlat = 22.5, llcrnrlon = -128,
               urcrnrlat = 52.0, urcrnrlon = -67, resolution='l')
```

In [8]:
```
# all the stuff that runs through get_airport_code()
codes_gotten = []
```

In [9]:
```
def get_airport_code(airport_name):
    """function takes in a loosely typed name of city and or airport and
    returns the airports' three digit code, full name, and coordinates
    It's good to specify the airport if a city has two, such as Dallas Love Field
    and Dallas Fort Worth """
    g = Geocoder.geocode(airport_name)
    proto_code = g.formatted_address.encode('utf-8')
#     airport_code = re.search('\(([^)]+)', proto_code).group(1)
#     x_map_coords, y_map_coords = mUSA(g.coordinates[0], g.coordinates[1])
    x_map_coords, y_map_coords = mUSA(g.coordinates[1], g.coordinates[0])
    codes_gotten.append((str(airport_name), (x_map_coords, y_map_coords)))
    return str(g.airport), g.coordinates, str(airport_name), (x_map_coords, y_map_coords)
```

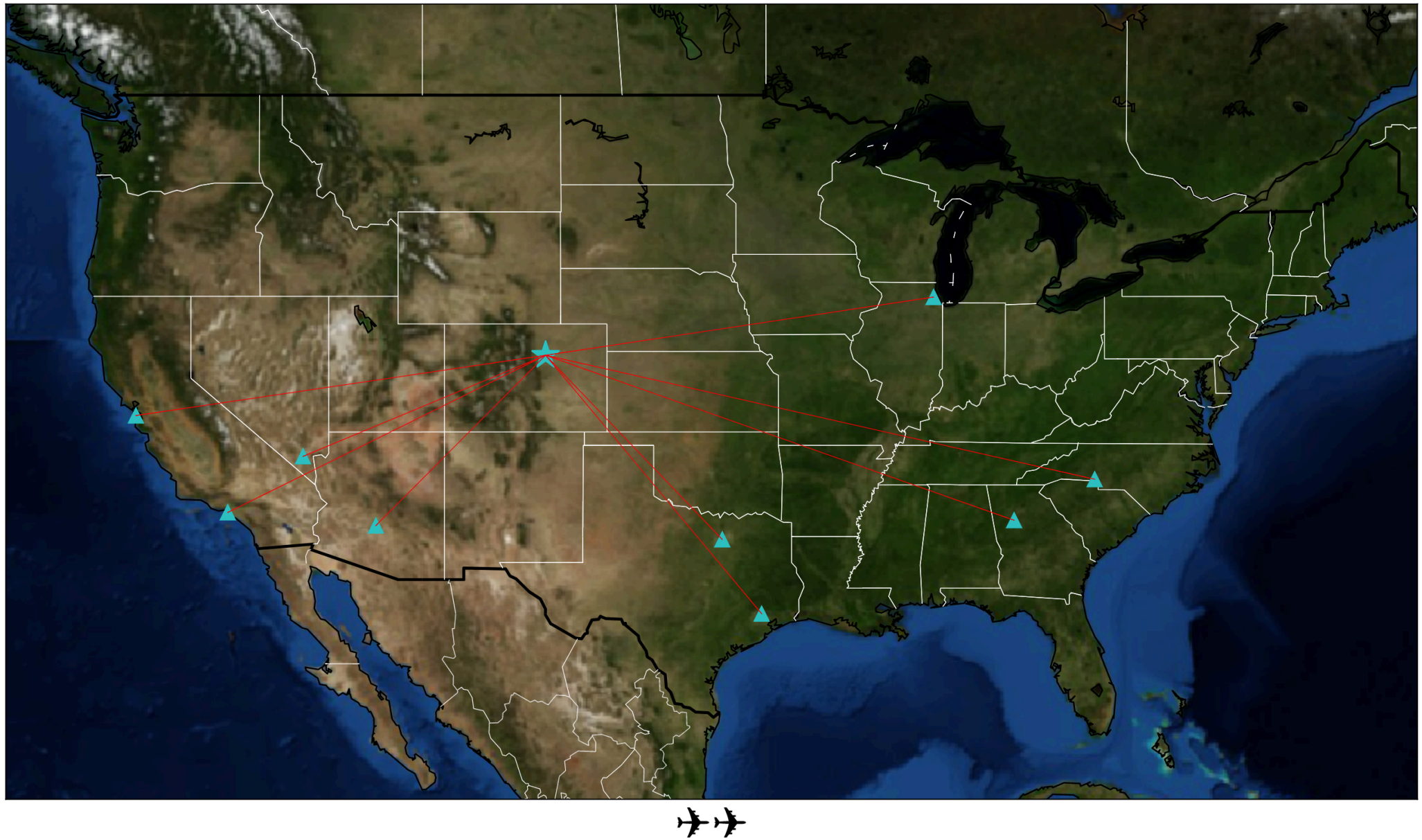
In [10]:
```
codes_gotten = []
for xxxxx in worst_airports:
    get_airport_code(xxxxx)
codes_gotten_dict = dict(codes_gotten)
```

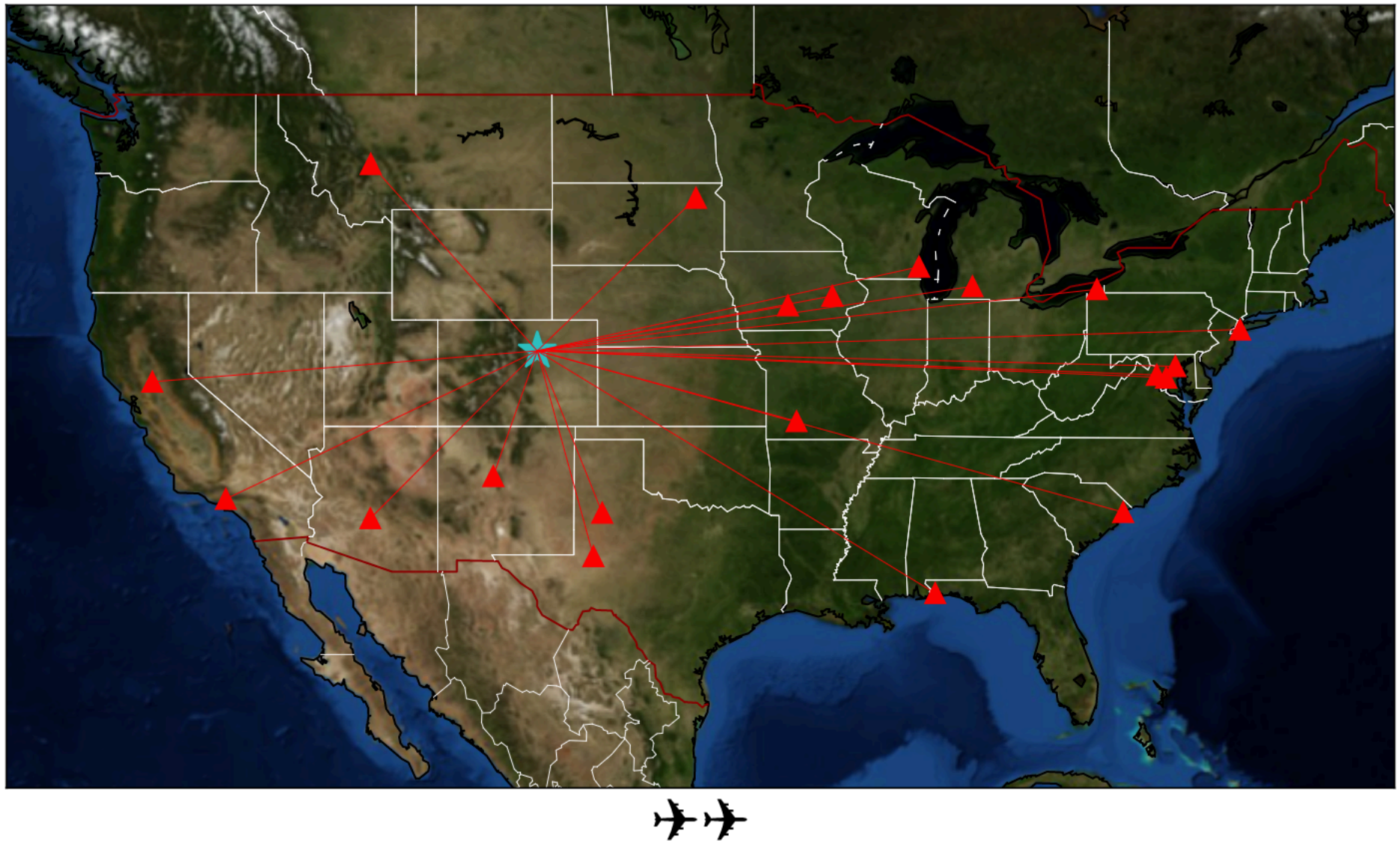
```
In [14]: codes_gotten_dict
```

```
Out[14]: {'ATL': (4845016.409674927, 1342252.968623687),
          'CLT': (5232017.437809654, 1539608.9499379043),
          'DFW': (3442556.035331869, 1250100.3674997683),
          'IAH': (3631982.3414067235, 892632.3678245954),
          'LAS': (1428438.388042691, 1649658.5204071011),
          'LAX': (1066522.300855291, 1379810.5044003548),
          'ORD': (4458100.356663127, 2415925.8138660314),
          'PHX': (1778251.9917544825, 1316899.771714727),
          'SFO': (625031.3476455866, 1846034.0456910953)}
```

```
In [15]: codes_gotten
```

```
Out[15]: [('ORD', (4458100.356663127, 2415925.8138660314)),
          ('ATL', (4845016.409674927, 1342252.968623687)),
          ('LAX', (1066522.300855291, 1379810.5044003548)),
          ('DFW', (3442556.035331869, 1250100.3674997683)),
          ('CLT', (5232017.437809654, 1539608.9499379043)),
          ('LAS', (1428438.388042691, 1649658.5204071011)),
          ('IAH', (3631982.3414067235, 892632.3678245954)),
          ('SFO', (625031.3476455866, 1846034.0456910953)),
          ('PHX', (1778251.9917544825, 1316899.771714727))]
```





In [19]:
```
# codes_gotten_2 = []
# for xxxxx in list_to_use:
#     get_airport_code_2(xxxxx)
# codes_gotten_dict_2 = dict(codes_gotten_2)
```

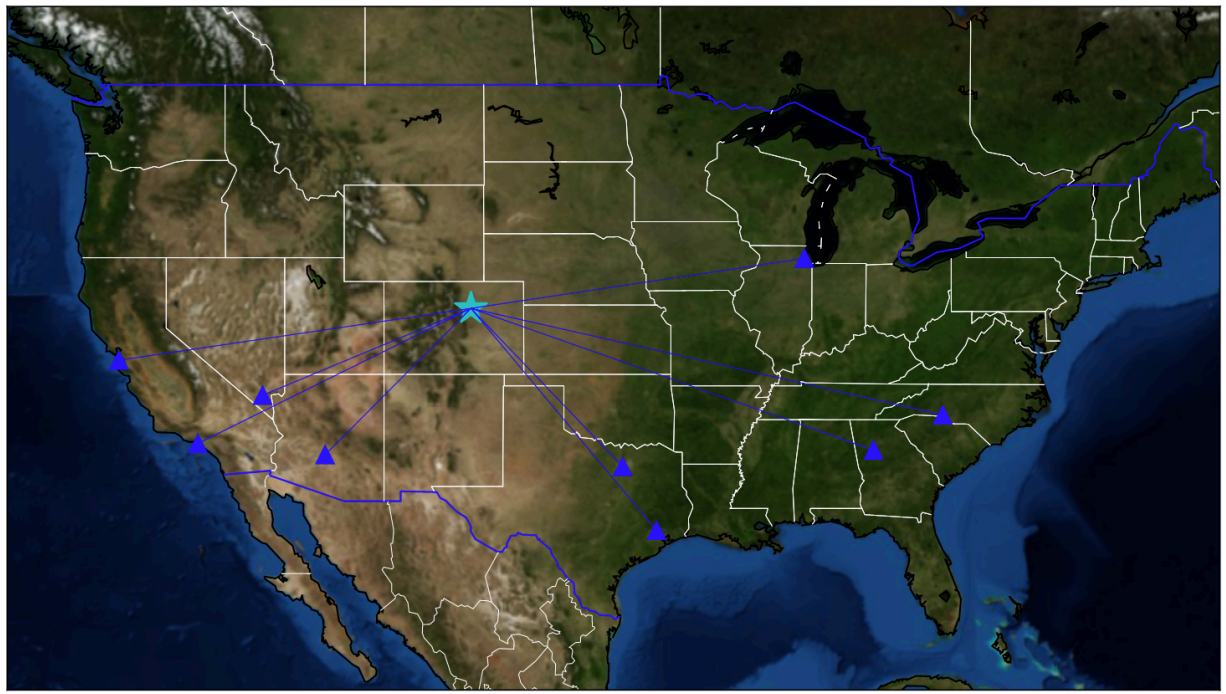
```
codes_gotten_dict_2 = {'ATL': (4845016.409674927, 1342252.968623687),
 'CLT': (5232017.437809654, 1539608.9499379043),
 'DFW': (3442556.035331869, 1250100.3674997683),
 'IAH': (3631982.3414067235, 892632.3678245954),
 'LAS': (1428438.388042691, 1649658.5204071011),
 'LAX': (1066522.300855291, 1379810.5044003548),
 'ORD': (4458100.356663127, 2415925.8138660314),
 'PHX': (1778251.9917544825, 1316899.771714727),
 'SFO': (625031.3476455866, 1846034.0456910953)}
```

In [20]:
```
codes_gotten_dict_2 = {'ATL': (4845016.409674927, 1342252.968623687),
 'CLT': (5232017.437809654, 1539608.9499379043),
 'DFW': (3442556.035331869, 1250100.3674997683),
 'IAH': (3631982.3414067235, 892632.3678245954),
 'LAS': (1428438.388042691, 1649658.5204071011),
 'LAX': (1066522.300855291, 1379810.5044003548),
 'ORD': (4458100.356663127, 2415925.8138660314),
 'PHX': (1778251.9917544825, 1316899.771714727),
 'SFO': (625031.3476455866, 1846034.0456910953)}
```

In [21]:
```
codes_gotten_2 = [('ORD', (4458100.356663127, 2415925.8138660314)),
 ('ATL', (4845016.409674927, 1342252.968623687)),
 ('LAX', (1066522.300855291, 1379810.5044003548)),
 ('DFW', (3442556.035331869, 1250100.3674997683)),
 ('CLT', (5232017.437809654, 1539608.9499379043)),
 ('LAS', (1428438.388042691, 1649658.5204071011)),
 ('IAH', (3631982.3414067235, 892632.3678245954)),
 ('SFO', (625031.3476455866, 1846034.0456910953)),
 ('PHX', (1778251.9917544825, 1316899.771714727))]
```

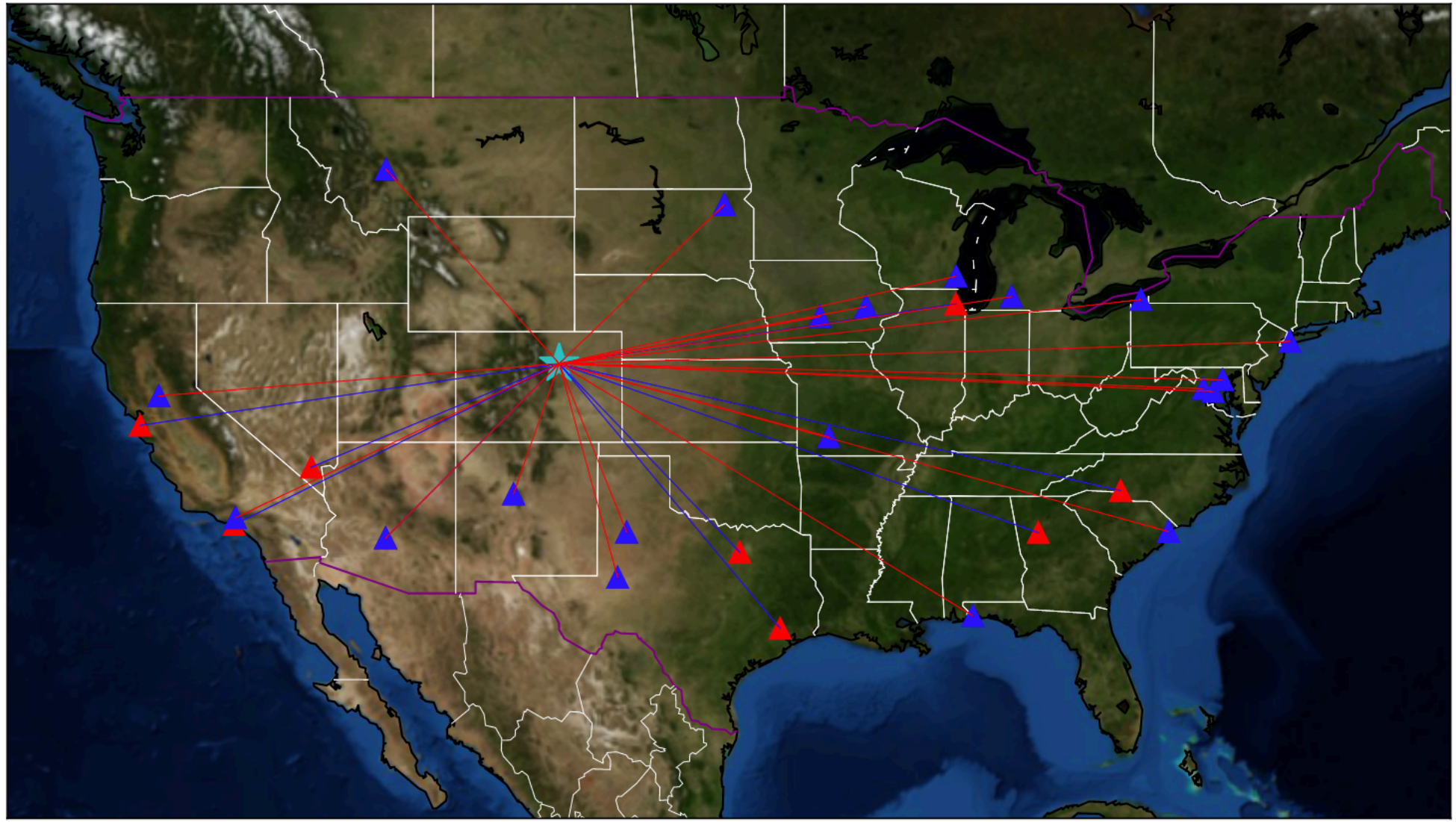
In [22]:

```
# mark DENVER (origin) with a cyan star
mUSA.plot(2593760.815099486, 2135931.09593892, 'c*', markersize=20)
# loop through worst_airport_codes for destination markers
for i, xs in codes_gotten_2:
    mUSA.plot(codes_gotten_dict_2[i][0], codes_gotten_dict_2[i][1], 'b^', markersize=10)
# loop through destination codes and origin codes, combine them and draw
# lines between origin and destination
for i2, xs2 in codes_gotten_2:
    mUSA.plot( [2593760.815099486, codes_gotten_dict_2[i2][0]],
               [2135931.09593892, codes_gotten_dict_2[i2][1]],
               color='b', linewidth=.5, label='flight' )

# Draw the macro details
mUSA.drawcoastlines()
mUSA.drawcountries(linewidth=1, color='blue')
mUSA.drawstates(color='w')
mUSA.bluemarble(scale=0.75)

plt.show()
```



Overlapping Maps

In [23]:
```
# mark DENVER (origin) with a cyan star
mUSA.plot(2593760.815099486, 2135931.09593892, 'c*', markersize=20)
# loop through worst_airport_codes for destination markers
for i, xs in codes_gotten_2:
    mUSA.plot(codes_gotten_dict_2[i][0], codes_gotten_dict_2[i][1], 'r^', markersize=10)
for i2, xs2 in codes_gotten_2:
    mUSA.plot( [2593760.815099486, codes_gotten_dict_2[i2][0]],
               [2135931.09593892, codes_gotten_dict_2[i2][1]],
               color='b', linewidth=.5, label='flight' )

for i, xs in codes_gotten:
    mUSA.plot(codes_gotten_dict[i][0], codes_gotten_dict[i][1], 'b^', markersize=10)
for i2, xs2 in codes_gotten:
    mUSA.plot( [2593760.815099486, codes_gotten_dict[i2][0]],
               [2135931.09593892, codes_gotten_dict[i2][1]],
               color='r', linewidth=.5, label='flight' )

# Draw the macro details
mUSA.drawcoastlines()
mUSA.drawcountries(linewidth=1, color='purple')
mUSA.drawstates(color='w')
mUSA.bluemarble(scale=0.75)

plt.show()
```



Algorithm Comparison

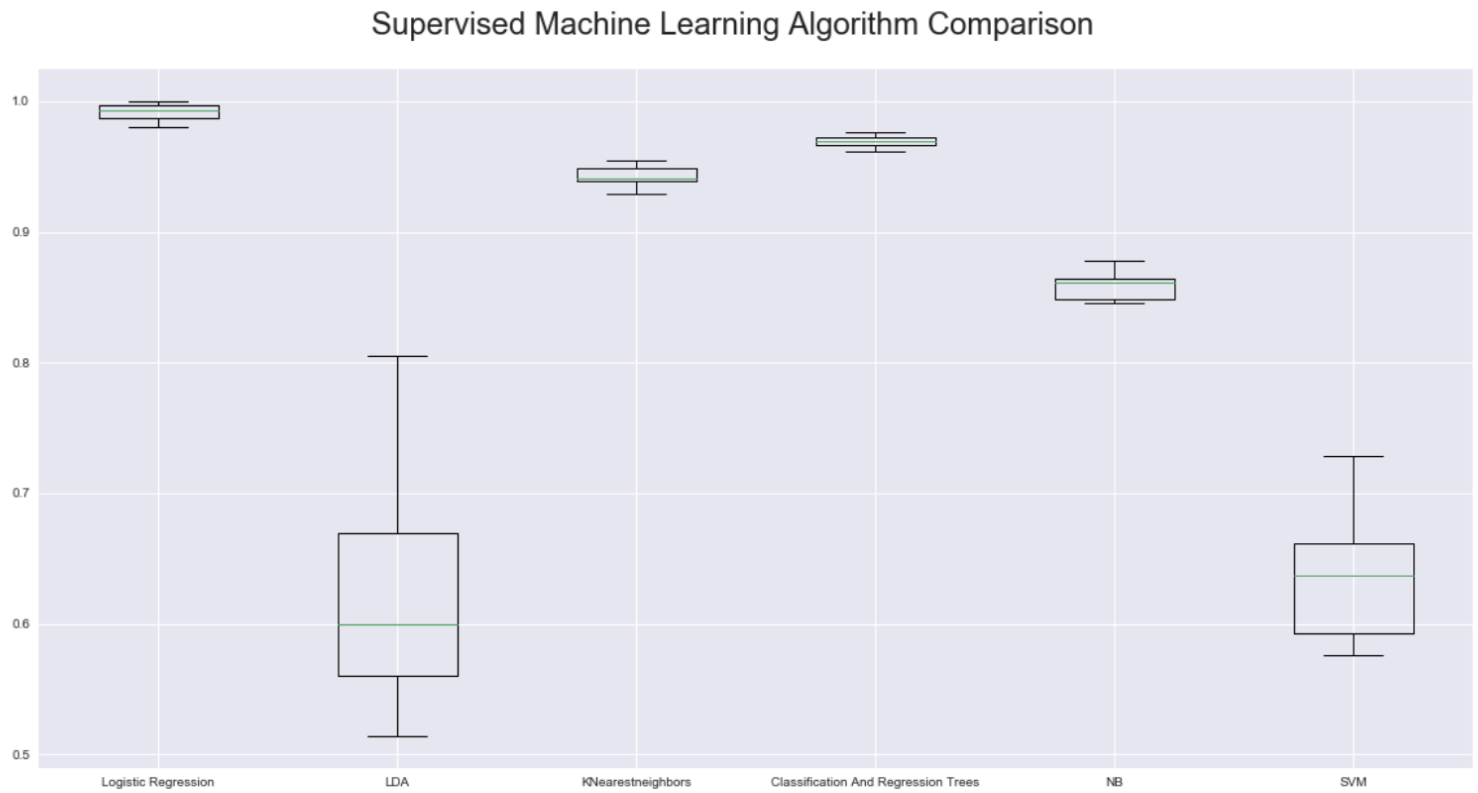
- See slides

Comparing Algorithms

In [61]:
```
# this cell takes about 90 seconds to run,... mostly bc of SVM
# Compare Algorithms
import pandas
import matplotlib.pyplot as plt
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC

seed = 42
# prepare list for models
models = []
models.append(('Logistic Regression', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNearestneighbors', KNeighborsClassifier()))
models.append(('Classification And Regression Trees', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
# models.append(('RFRg', RandomForestRegressor(n_estimators = 100,oob_score = True, n_jobs = -1, r
andom_state=seed)))
models.append(('SVM', SVC()))
# evaluate each model in turn
results = []
names = []
scoring = 'accuracy'
for name, model in models:
    kfold = model_selection.KFold(n_splits=10, random_state=seed)
    cv_results = model_selection.cross_val_score(model, X, y, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

```
Logistic Regression: 0.991945 (0.006417)
/Users/jg/anaconda/lib/python2.7/site-packages/sklearn/discriminant_analysis.py:387: UserWarning:
Variables are collinear.
  warnings.warn("Variables are collinear.")
LDA: 0.626157 (0.092132)
KNearestneighbors: 0.942091 (0.007824)
Classification And Regression Trees: 0.969300 (0.004674)
NB: 0.857067 (0.014922)
SVM: 0.640382 (0.050608)
```

In [68]:
```
# boxplot algorithm comparison
fig = plt.figure(figsize=(20,10))
fig.suptitle('\nSupervised Machine Learning Algorithm Comparison', size =24)
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```



Random Forest model

- oob_score(out of bag)=True shows model performance with or without using Cross-Validation
- Generally the target follows a certain distribution and few observations, then using CV will not give much improvement.

Metrics

- Confusion Matrix
- AUC
- ROC
- f1 score
- https://github.com/josezilla/DAT-DEN-03/blob/master/projects/FlightDelaysProject/ProjectFolder%20copy/2018.01.20_Modeling_dataset.ipynb