USA Car Accidents Severity Prediction

by Vinay Kumar Moluguri

Nov 29, 2022 (updated on Dec 22, 2022)

0 INTRODUCTION

Motivation

The economic and societal impact of traffic accidents cost U.S. citizens hundreds of billions of dollars every year. And a large part of losses is caused by a small number of serious accidents. Reducing traffic accidents, especially serious accidents, is nevertheless always an important challenge. The proactive approach, one of the two main approaches for dealing with traffic safety problems, focuses on preventing potential unsafe road conditions from occurring in the first place. For the effective implementation of this approach, accident prediction and severity prediciton are critical. If we can identify the patterns of how these serious accidents happen and the key factors, we might be able to implement well-informed actions and better allocate financial and human resources.

Objectives

The first objective of this project is to recognize **key factors affecting the accident severity**. The second one is to develop a model that can **accurately predict accident severity**. To be specific, for a given accident, without any detailed information about itself, like driver attributes or vehicle type, this model is supposed to be able to predict the likelihood of this accident being a severe one. The accident could be the one that just happened and still lack of detailed information, or a potential one predicted by other models. Therefore, with the sophisticated real-time traffic accident prediction solution developed by the creators of the same dataset used in this project, this model might be able to further predict severe accidents in real-time.

Process

Data cleaning was first performed to detect and handle corrupt or missing records. EDA (Exploratory Data Analysis) and feature engineering were then done over most features. Finally, Logistic regression, Random Forest Classifier, and EasyEnsemble were used to develop the predictive model.

It is worth noting that the severity in this project is "**an indication of the effect the accident has on traffic**", rather than the injury severity that has already been thoroughly studied by many articles. Another thing is that the final model is dependent on only **a small range of data attributes** that are **easily achievable** for all regions in the United States and before the accident really happened.

Key Findings

- Country-wide accident severity can be accurately predicted with limited data attributes (location, time, weather, and POI).

- **Minute(frequency-encoding)** is the most useful feature. An accident is more likely to be a serious one when accidents happen less frequently at this time.
- Spatial patterns are also very important. For small areas like **street** and **zipcode**, severe accidents are more likely to happen at places having more accidents while for larger areas like **city** and **airport region**, at places having less accident.
- **Pressure** is top fourth important feature in the random-forest model and there is negative correlation between pressure and severity.
- If an accident happens on **Interstate Highway**, there is a 2% chance that it will be a serious one, which is about 2.3 times of average and higher than any other street type.
- An accident is much less likely to be severe if it happens near **traffic signal** while more likely if near **junction**.

Dataset Overview

US-Accident dataset is a countrywide car accident dataset, which covers **49 states of the United States**. It contains more than **4 million cases** of traffic accidents that took place from **February 2016 to December 2020**. In this project, however, only the data of accidents that happened after **February 2019** and were reported by *MapQuest* was finally used in exploration analysis and modeling so that irrelevant factors can be eliminated to the greatest extent.

Link for kaggle dataset: https://www.kaggle.com/sobhanmoosavi/us-accidents

Acknowledgements

Moosavi, Sobhan, Mohammad Hossein Samavatian, Srinivasan Parthasarathy, and Rajiv Ramnath. "A Countrywide Traffic Accident Dataset.", 2019.

Moosavi, Sobhan, Mohammad Hossein Samavatian, Srinivasan Parthasarathy, Radu Teodorescu, and Rajiv Ramnath. "Accident Risk Prediction based on Heterogeneous Sparse Data: New Dataset and Insights." In proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM, 2019.

Refrences

I found these notebooks really helpful:

USA Accidents Data Analysis

https://www.kaggle.com/sobhanmoosavi/us-accidents/discussion/113055

how Severity the Accidents is ?

Severity Prediction in SFO Bay Area

ML to Predict Accident Severity_PA_Mont

severity and hours wasted

USA Accidents Plotly maps + text classification

Tabel of content

1 OVERVIEW & PREPROCESSING

In [1]:

```
import numpy as np
import pandas as pd
import json
import matplotlib
import matplotlib.pyplot as plt
from matplotlib import cm
from datetime import datetime
import glob
import seaborn as sns
import re
import os
import io
from scipy.stats import boxcox
```

1.1 Overview the dataset

Details about features in the dataset:

Traffic Attributes (12):

- **ID**: This is a unique identifier of the accident record.
- **Source**: Indicates source of the accident report (i.e. the API which reported the accident.).
- **TMC**: A traffic accident may have a Traffic Message Channel (TMC) code which provides more detailed description of the event.
- **Severity**: Shows the severity of the accident, a number between 1 and 4, where 1 indicates the least impact on traffic (i.e., short delay as a result of the accident) and 4 indicates a significant impact on traffic (i.e., long delay).
- **Start_Time**: Shows start time of the accident in local time zone.
- **End_Time**: Shows end time of the accident in local time zone.
- **Start_Lat**: Shows latitude in GPS coordinate of the start point.
- **Start_Lng**: Shows longitude in GPS coordinate of the start point.
- **End_Lat**: Shows latitude in GPS coordinate of the end point.
- **End_Lng**: Shows longitude in GPS coordinate of the end point.

- **Distance(mi)**: The length of the road extent affected by the accident.
- **Description**: Shows natural language description of the accident.

Address Attributes (9):

- **Number**: Shows the street number in address field.
- **Street**: Shows the street name in address field.
- **Side**: Shows the relative side of the street (Right/Left) in address field.
- **City**: Shows the city in address field.
- **County**: Shows the county in address field.
- **State**: Shows the state in address field.
- **Zipcode**: Shows the zipcode in address field.
- **Country**: Shows the country in address field.
- **Timezone**: Shows timezone based on the location of the accident (eastern, central, etc.).

Weather Attributes (11):

- **Airport_Code**: Denotes an airport-based weather station which is the closest one to location of the accident.
- **Weather_Timestamp**: Shows the time-stamp of weather observation record (in local time).
- **Temperature(F)**: Shows the temperature (in Fahrenheit).
- **Wind_Chill(F)**: Shows the wind chill (in Fahrenheit).
- **Humidity(%)**: Shows the humidity (in percentage).
- **Pressure(in)**: Shows the air pressure (in inches).
- **Visibility(mi)**: Shows visibility (in miles).
- **Wind_Direction**: Shows wind direction.
- **Wind_Speed(mph)**: Shows wind speed (in miles per hour).
- **Precipitation(in)**: Shows precipitation amount in inches, if there is any.
- **Weather_Condition**: Shows the weather condition (rain, snow, thunderstorm, fog, etc.).

POI Attributes (13):

- **Amenity**: A Point-Of-Interest (POI) annotation which indicates presence of amenity in a nearby location.
- **Bump**: A POI annotation which indicates presence of speed bump or hump in a nearby location.
- **Crossing**: A POI annotation which indicates presence of crossing in a nearby location.

- **Give_Way**: A POI annotation which indicates presence of give_way sign in a nearby location.
- **Junction**: A POI annotation which indicates presence of junction in a nearby location.
- **No_Exit**: A POI annotation which indicates presence of no_exit sign in a nearby location.
- **Railway**: A POI annotation which indicates presence of railway in a nearby location.
- **Roundabout**: A POI annotation which indicates presence of roundabout in a nearby location.
- **Station**: A POI annotation which indicates presence of station (bus, train, etc.) in a nearby location.
- **Stop**: A POI annotation which indicates presence of stop sign in a nearby location.
- **Traffic_Calming**: A POI annotation which indicates presence of traffic_calming means in a nearby location.
- **Traffic_Signal**: A POI annotation which indicates presence of traffic_signal in a nearby location.
- **Turning_Loop**: A POI annotation which indicates presence of turning_loop in a nearby location.

Period-of-Day (4):

- **Sunrise_Sunset**: Shows the period of day (i.e. day or night) based on sunrise/sunset.
- **Civil_Twilight**: Shows the period of day (i.e. day or night) based on civil twilight.
- **Nautical_Twilight**: Shows the period of day (i.e. day or night) based on nautical twilight.
- **Astronomical_Twilight**: Shows the period of day (i.e. day or night) based on astronomical twilight.

In [2]:

```
df = pd.read_csv('../input/us-accidents/US_Accidents_Dec20.csv')
print("The shape of data is:",(df.shape))
display(df.head(3))
```

```
The shape of data is: (4232541, 49)
```

	ID	Source	TMC	Severity	Start_Time	End_Time	Start_Lat	Start_Lng	End_Lat	End_Lng	...	Roundabout	Station	Stop	Traffic_Calming	Traffic_Signal	Turning_Loop	Sunrise_Sunset	Civil_Twilight	Nautical_Twilight	Astronomical_Twilight
0	A-1	MapQuest	201.0	3	2016-02-08 05:46:00	2016-02-08 11:00:00	39.865147	-84.058723	NaN	NaN	...	False	False	False	False	False	False	Night	Night	Night	Night
1	A-2	MapQuest	201.0	2	2016-02-08 06:07:59	2016-02-08 06:37:59	39.928059	-82.831184	NaN	NaN	...	False	False	False	False	False	False	Night	Night	Night	Day
2	A-3	MapQuest	201.0	2	2016-02-08 06:49:27	2016-02-08 07:19:27	39.063148	-84.032608	NaN	NaN	...	False	False	False	False	True	False	Night	Night	Day	Day

3 rows × 49 columns

linkcode

1.2 Reporting Sources

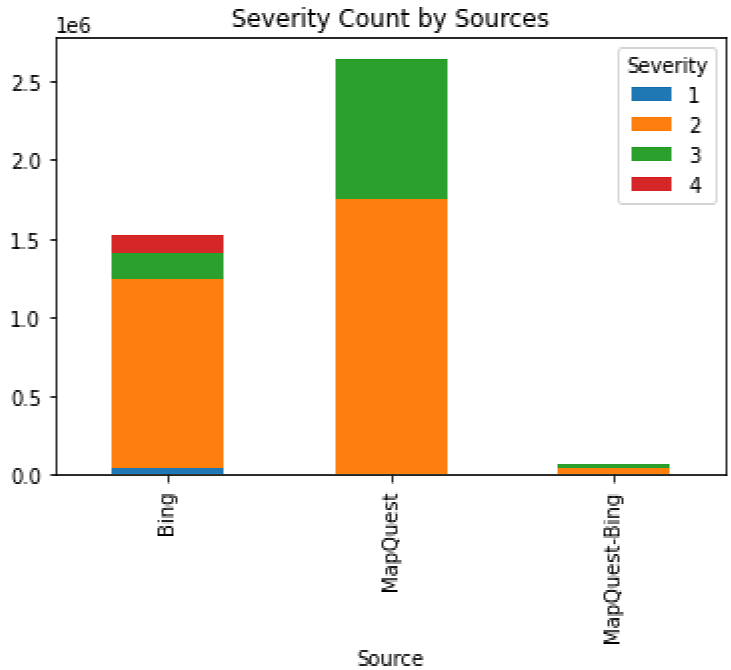
These data came from two sources, *MapQuest* and *Bing*, both of which report severity level but in a different way. Bing has 4 levels while MapQuest has 5. And according to dataset creator, there is no way to do a 1:1 mapping between them. Since severity is what we really care about in this project, I think it is crucial to figure out the difference.

In [3]:

```
df_source = df.groupby(['Severity','Source']).size().reset_index().pivot(\
    columns='Severity', index='Source', values=0)
df_source.plot(kind='bar', stacked=True, title='Severity Count by Sources')
```

Out[3]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1a24cf6210>
```



The stacked bar chart shows that two data providers reported totally different proportions of accidents of each level. *MapQuest* reported so rare accidents with severity level 4 which can not even be seen in the plot, whereas *Bing* reported almost the same number of level 4 accidents as level 2. Meanwhile, *MapQuest* reported much more level 3 accidents than *Bing* in terms of proportion. These differences may be due to the different kinds of accidents they tend to collect or the different definitions of severity level, or the combination of them. If the latter is the case, I don't think we can use the data from both of them at the same time. To check it out, we can examine the distribution of accidents with different severity levels across two main measures, **Impacted Distance** and **Duration**.

In [4]:

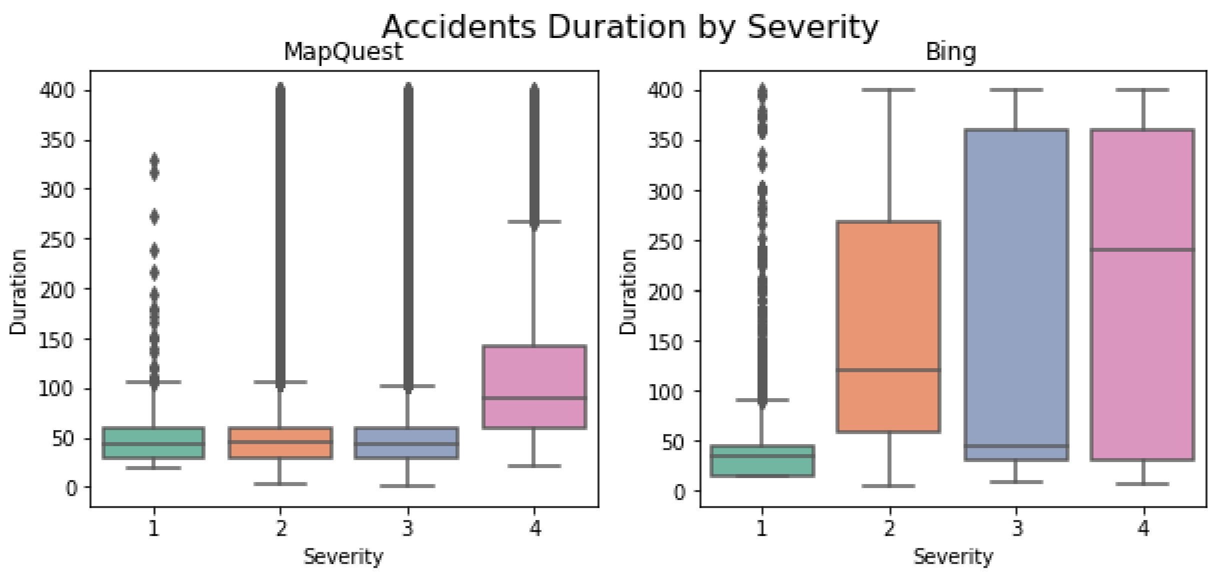
```
# fix datetime type
df['Start_Time'] = pd.to_datetime(df['Start_Time'])
df['End_Time'] = pd.to_datetime(df['End_Time'])
df['Weather_Timestamp'] = pd.to_datetime(df['Weather_Timestamp'])

# calculate duration as the difference between end time and start time in minut
e
df['Duration'] = df.End_Time - df.Start_Time
df['Duration'] = df['Duration'].apply(lambda x:round(x.total_seconds() / 60)
)
```

```
print("The overall mean duration is: ", (round(df['Duration'].mean(),3)), 'mi
n')
```

```
The overall mean duration is:  134.661 min
```

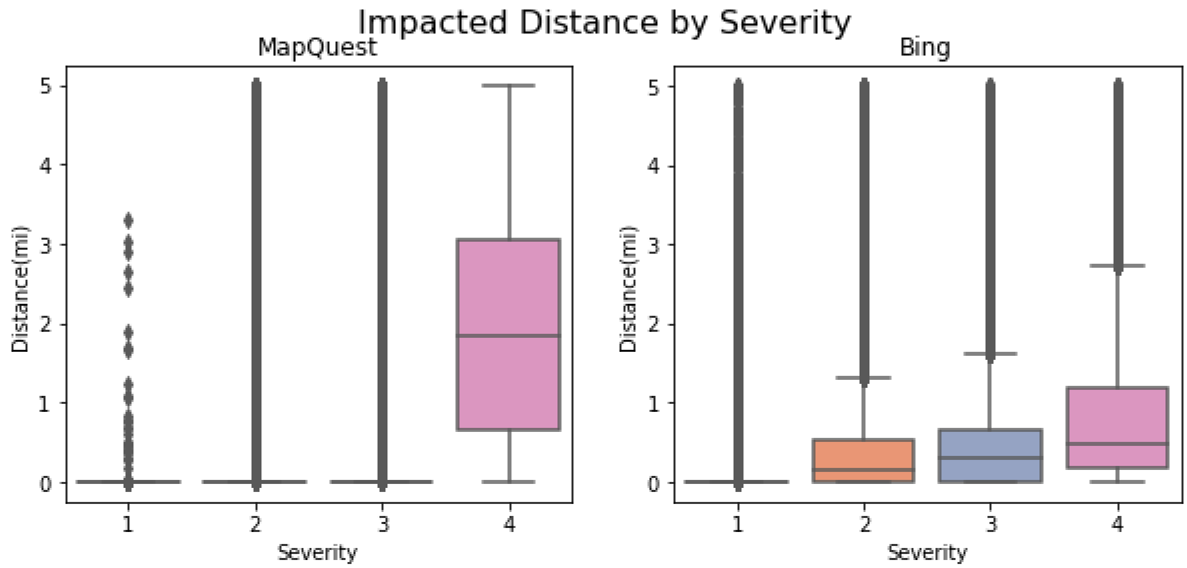
In [5]:

```
fig, axs = plt.subplots(ncols=2, figsize=(10, 4))
sns.boxplot(x="Severity", y="Duration",
            data=df.loc[(df['Source']=="MapQuest") & (df['Duration']<400),],
palette="Set2", ax=axs[0])
axs[0].set_title('MapQuest')
fig.suptitle('Accidents Duration by Severity', fontsize=16)
sns.boxplot(x="Severity", y="Duration",
            data=df.loc[(df['Source']=="Bing") & (df['Duration']<400),], pale
tte="Set2", ax=axs[1])
axs[1].set_title('Bing')
plt.show()
```



In [6]:

```
fig, axs = plt.subplots(ncols=2, figsize=(10, 4))
sns.boxplot(x="Severity", y="Distance(mi)",
            data=df.loc[(df['Source']=="MapQuest") & (df['Distance(mi)']<5),]
, palette="Set2", ax=axs[0])
axs[0].set_title('MapQuest')
fig.suptitle('Impacted Distance by Severity', fontsize=16)
sns.boxplot(x="Severity", y="Distance(mi)",
            data=df.loc[(df['Source']=="Bing") & (df['Distance(mi)']<5),], pa
lette="Set2", ax=axs[1])
axs[1].set_title('Bing')
plt.show()
```



Two differences are obvious in the above plots. The first is that the overall duration and impacted distance of accidents reported by *Bing* are much longer than those by *MapQuest*. Second, same severity level holds different meanings for *MapQuest* and *Bing*. *MapQuest* seems to have a clear and strict threshold for severity level 4, cases of which nevertheless only account for a tiny part of the whole dataset. *Bing*, on the other hand, doesn't seem to have a clear-cut threshold, especially regards duration, but the data is more balanced.

It is hard to choose one and we definitely can't use both. I decided to select *MapQuest* because serious accidents are we really care about and the sparse data of such accidents is the reality we have to confront.

Finally, drop data reported from *Bing* and 'Source' column.

In [7]:

```
df = df.loc[df['Source']=="MapQuest",]
df = df.drop(['Source'], axis=1)
print("The shape of data is:",(df.shape))
```

```
The shape of data is: (2651861, 49)
```

1.3 Useless Features

Features 'ID' doesn't provide any useful information about accidents themselves. 'TMC', 'Distance(mi)', 'End_Time' (we have start time), 'Duration', 'End_Lat', and 'End_Lng'(we have start location) can be collected only after the accident has already happened and hence cannot be predictors for serious accident prediction. For 'Description', the POI features have already been extracted from it by dataset creators. Let's get rid of these features first.

In [8]:

```
df = df.drop(['ID','TMC','Description','Distance(mi)', 'End_Time', 'Duration'
,
              'End_Lat', 'End_Lng'], axis=1)
```

Check out some categorical features.

In [9]:

```
cat_names = ['Side', 'Country', 'Timezone', 'Amenity', 'Bump', 'Crossing',
             'Give_Way', 'Junction', 'No_Exit', 'Railway', 'Roundabout', 'Sta
tion',
             'Stop', 'Traffic_Calming', 'Traffic_Signal', 'Turning_Loop', 'Su
nrise_Sunset',
             'Civil_Twilight', 'Nautical_Twilight', 'Astronomical_Twilight']
print("Unique count of categorical features:")
for i in cat_names:
  print(i,df[i].unique().size)
```

```
Unique count of categorical features:
Side 3
Country 1
Timezone 5
Amenity 2
Bump 2
Crossing 2
Give_Way 2
Junction 2
No_Exit 2
Railway 2
Roundabout 2
Station 2
Stop 2
Traffic_Calming 2
Traffic_Signal 2
Turning_Loop 1
Sunrise_Sunset 3
Civil_Twilight 3
Nautical_Twilight 3
Astronomical_Twilight 3
```

Drop 'Country' and 'Turning_Loop' for they have only one class.

In [10]:

```
df = df.drop(['Country','Turning_Loop'], axis=1)
```

1.4 Clean Up Categorical Features

If we look at categorical features closely, we will find some chaos in 'Wind_Direction' and 'Weather_Condition'. It is necessary to clean them up first.

Wind Direction

In [11]:

```
print("Wind Direction: ", df['Wind_Direction'].unique())
```

```
Wind Direction:  ['Calm' 'SW' 'SSW' 'WSW' 'WNW' 'NW' 'West' 'NNW' 'NNE' 'South' 'North'
 'Variable' 'SE' 'SSE' 'ESE' 'East' 'NE' 'ENE' 'E' 'W' nan 'S' 'VAR'
 'CALM' 'N']
```

Simplify wind direction

In [12]:

```
df.loc[df['Wind_Direction']=='Calm','Wind_Direction'] = 'CALM'
df.loc[(df['Wind_Direction']=='West')|(df['Wind_Direction']=='WSW')|(df['Wind_Direction']=='WNW'),'Wind_Direction'] = 'W'
df.loc[(df['Wind_Direction']=='South')|(df['Wind_Direction']=='SSW')|(df['Wind_Direction']=='SSE'),'Wind_Direction'] = 'S'
df.loc[(df['Wind_Direction']=='North')|(df['Wind_Direction']=='NNW')|(df['Wind_Direction']=='NNE'),'Wind_Direction'] = 'N'
df.loc[(df['Wind_Direction']=='East')|(df['Wind_Direction']=='ESE')|(df['Wind_Direction']=='ENE'),'Wind_Direction'] = 'E'
df.loc[df['Wind_Direction']=='Variable','Wind_Direction'] = 'VAR'
print("Wind Direction after simplification: ", df['Wind_Direction'].unique())
```

```
Wind Direction after simplification:  ['CALM' 'SW' 'S' 'W' 'NW' 'N' 'VAR'
'SE' 'E' 'NE' nan]
```

Weather Condition

Weather-related vehicle accidents kill more people annually than large-scale weather disasters(source: weather.com). According to Road Weather Management Program, most weather-related crashes happen on wet-pavement and during rainfall. Winter-condition and fog are another two main reasons for weather-related accidents. To extract these three weather conditions, we first look at what we have in 'Weather_Condition' Feature.

In [13]:

```
# show distinctive weather conditions
weather ='!'.join(df['Weather_Condition'].dropna().unique().tolist())
weather = np.unique(np.array(re.split(
    "!|\s/\s|\sand\s|\swith\s|Partly\s|Mostly\s|Blowing\s|Freezing\s", weather))).tolist()
print("Weather Conditions: ", weather)
```

```
Weather Conditions:  ['', 'Clear', 'Cloudy', 'Drizzle', 'Dust', 'Dust Whirlwinds', 'Fair', 'Fog', 'Funnel Cloud', 'Hail', 'Haze', 'Heavy ', 'Heavy Drizzle', 'Heavy Ice Pellets', 'Heavy Rain', 'Heavy Rain Showers', 'Heavy Sleet', 'Heavy Smoke', 'Heavy Snow', 'Heavy T-Storm', 'Heavy Thunderstorms', 'Ice Pellets', 'Light ', 'Light Drizzle', 'Light Fog', 'Light Hail', 'Light Haze', 'Light Ice Pellets', 'Light Rain', 'Light Rain Shower', 'Light Rain Showers', 'Light Sleet', 'Light Snow', 'Light Snow Grains', 'Light Snow Shower', 'Light Snow Showers', 'Light Thunderstorm', 'Light Thunderstorms', 'Low Drifting Snow', 'Mist', 'N/A Precipitation', 'Overcast', 'Partial Fog', 'Patches of Fog', 'Rain', 'Rain Shower', 'Rain Showers', 'Sand', 'Scattered Clouds', 'Shallow Fog', 'Showers in the Vicinity', 'Sleet', 'Small Hail', 'Smoke', 'Snow', 'Snow Grains', 'Snow Showers', 'Squalls', 'T-St
```

```
orm', 'Thunder', 'Thunder in the Vicinity', 'Thunderstorm', 'Thunderstorms
', 'Tornado', 'Volcanic Ash', 'Widespread Dust', 'Windy', 'Wintry Mix']
```

Create features for some common weather conditions and drop 'Weather_Condition' then.

In [16]:

```
df['Clear'] = np.where(df['Weather_Condition'].str.contains('Clear', case=Fal
se, na = False), True, False)
df['Cloud'] = np.where(df['Weather_Condition'].str.contains('Cloud|Overcast',
case=False, na = False), True, False)
df['Rain'] = np.where(df['Weather_Condition'].str.contains('Rain|storm', case
=False, na = False), True, False)
df['Heavy_Rain'] = np.where(df['Weather_Condition'].str.contains('Heavy Rain|
Rain Shower|Heavy T-Storm|Heavy Thunderstorms', case=False, na = False), True
, False)
df['Snow'] = np.where(df['Weather_Condition'].str.contains('Snow|Sleet|Ice',
case=False, na = False), True, False)
df['Heavy_Snow'] = np.where(df['Weather_Condition'].str.contains('Heavy Snow|
Heavy Sleet|Heavy Ice Pellets|Snow Showers|Squalls', case=False, na = False),
True, False)
df['Fog'] = np.where(df['Weather_Condition'].str.contains('Fog', case=False,
na = False), True, False)
```

In [19]:

```
# Assign NA to created weather features where 'Weather_Condition' is null.
weather = ['Clear','Cloud','Rain','Heavy_Rain','Snow','Heavy_Snow','Fog']
for i in weather:
    df.loc[df['Weather_Condition'].isnull(),i] = df.loc[df['Weather_Condition
'].isnull(),'Weather_Condition']
    df[i] = df[i].astype('bool')

df.loc[:,['Weather_Condition'] + weather]

df = df.drop(['Weather_Condition'], axis=1)
```

1.5 Fix Datetime Format

In [20]:

```
# average difference between weather time and start time
print("Mean difference between 'Start_Time' and 'Weather_Timestamp': ",
(df.Weather_Timestamp - df.Start_Time).mean())
```

```
Mean difference between 'Start_Time' and 'Weather_Timestamp':  0 days 00:0
0:33.122457
```

Since the 'Weather_Timestamp' is almost as same as 'Start_Time', we can just keep 'Start_Time'.
Then map 'Start_Time' to 'Year', 'Month', 'Weekday', 'Day' (in a year), 'Hour', and 'Minute' (in a day).

In [21]:

```
df = df.drop(["Weather_Timestamp"], axis=1)
```

```
df['Year'] = df['Start_Time'].dt.year

nmonth = df['Start_Time'].dt.month
df['Month'] = nmonth

df['Weekday']= df['Start_Time'].dt.weekday

days_each_month = np.cumsum(np.array([0,31,28,31,30,31,30,31,31,30,31,30,31])
)
nday = [days_each_month[arg-1] for arg in nmonth.values]
nday = nday + df["Start_Time"].dt.day.values
df['Day'] = nday

df['Hour'] = df['Start_Time'].dt.hour

df['Minute']=df['Hour']*60.0+df["Start_Time"].dt.minute

df.loc[:4,['Start_Time', 'Year', 'Month', 'Weekday', 'Day', 'Hour', 'Minute']
]
```

Out[21]:

	Start_Time	Year	Month	Weekday	Day	Hour	Minute
0	2016-02-08 05:46:00	2016	2	0	39	5	346.0
1	2016-02-08 06:07:59	2016	2	0	39	6	367.0
2	2016-02-08 06:49:27	2016	2	0	39	6	409.0
3	2016-02-08 07:23:34	2016	2	0	39	7	443.0
4	2016-02-08 07:39:07	2016	2	0	39	7	459.0

2 HANDLING MISSING DATA

2.1 Drop Features

As seen from below, many columns have missing values.

In [22]:

```
missing = pd.DataFrame(df.isnull().sum()).reset_index()
missing.columns = ['Feature', 'Missing_Percent(%)']
missing['Missing_Percent(%)'] = missing['Missing_Percent(%)'].apply(lambda x:
x / df.shape[0] * 100)
missing.loc[missing['Missing_Percent(%)']>0,:]
```

Out[22]:

	Feature	Missing_Percent(%)
4	Number	60.026676
7	City	0.001999
10	Zipcode	0.013462
11	Timezone	0.087222
12	Airport_Code	0.177649
13	Temperature(F)	1.708008
14	Wind_Chill(F)	53.724535
15	Humidity(%)	1.821589
16	Pressure(in)	1.465273
17	Visibility(mi)	1.998672

	Feature	Missing_Percent(%)
18	Wind_Direction	1.517463
19	Wind_Speed(mph)	12.992687
20	Precipitation(in)	57.669350
33	Sunrise_Sunset	0.002149
34	Civil_Twilight	0.002149
35	Nautical_Twilight	0.002149
36	Astronomical_Twilight	0.002149
37	Clear	1.995995
38	Cloud	1.995995
39	Rain	1.995995
40	Heavy_Rain	1.995995
41	Snow	1.995995

	Feature	Missing_Percent(%)
42	Heavy_Snow	1.995995
43	Fog	1.995995

More than 60% percent of 'Number', 'Wind_Chill(F)', and 'Precipitation(in)' is missing. Drop na and value imputation wouldn't work for these features. 'Number' and 'Wind_Chill(F)' will be dropped because they are not highly related to severity according to previous research, whereas 'Precipitation(in)' could be a useful predictor and hence can be handled by separating feature.

Drop these features:

1. 'Number'
2. 'Wind_Chill(F)'

In [23]:

```
df = df.drop(['Number','Wind_Chill(F)'], axis=1)
```

2.2 Separate Featrue

Add a new feature for missing values in 'Precipitation(in)' and replace missing values with median.

In [24]:

```
df['Precipitation_NA'] = 0
df.loc[df['Precipitation(in)'].isnull(),'Precipitation_NA'] = 1
df['Precipitation(in)'] = df['Precipitation(in)'].fillna(df['Precipitation(in
)'].median())
df.loc[:5,['Precipitation(in)','Precipitation_NA']]
```

Out[24]:

	Precipitation(in)	Precipitation_NA
0	0.02	0
1	0.00	0

	Precipitation(in)	Precipitation_NA
2	0.00	1
3	0.00	1
4	0.00	1
5	0.03	0

2.3 Drop NaN

The counts of missing values in some features are much smaller compared to the total sample. It is convenient to drop rows with missing values in these columns.

Drop NAs by these features:

1. 'City'
2. 'Zipcode'
3. 'Airport_Code'
4. 'Sunrise_Sunset'
5. 'Civil_Twilight'
6. 'Nautical_Twilight'
7. 'Astronomical_Twilight'

In [25]:

```
df = df.dropna(subset=['City','Zipcode','Airport_Code',
                       'Sunrise_Sunset','Civil_Twilight','Nautical_Twilight',
'Astronomical_Twilight'])
```

2.4 Value Imputation

Most of the rest columns only have small missing part that can be filled. (It is not absolutely necessary though, we can also just drop na)

Continuous Weather Data

Continuous weather features with missing values:

1. Temperature(F)
2. Humidity(%)
3. Pressure(in)
4. Visibility(mi)
5. Wind_Speed(mph)

Before imputation, weather features will be grouped by location and time first, to which weather is naturally related. 'Airport_Code' is selected as location feature because the sources of weather data are airport-based weather stations. Then the data will be grouped by 'Start_Month' rather than 'Start_Hour' because using the former is computationally cheaper and remains less missing values. Finally, missing values will be replaced by median value of each group.

In [26]:

```
# group data by 'Airport_Code' and 'Start_Month' then fill NAs with median valu
e
Weather_data=['Temperature(F)','Humidity(%)','Pressure(in)','Visibility(mi)',
'Wind_Speed(mph)']
print("The number of remaining missing values: ")
for i in Weather_data:
  df[i] = df.groupby(['Airport_Code','Month'])[i].apply(lambda x: x.fillna(x.
median()))
  print( i + " : " + df[i].isnull().sum().astype(str))
```

```
The number of remaining missing values:
Temperature(F) : 5176
Humidity(%) : 5200
Pressure(in) : 5146
Visibility(mi) : 11901
Wind_Speed(mph) : 11934
```

There still are some missing values but much less. Just dropna by these features for the sake of simplicity.

In [27]:

```
df = df.dropna(subset=Weather_data)
```

Categorical Weather Features

For categorical weather features, majority rather than median will be used to replace missing values.

In [28]:

```
# group data by 'Airport_Code' and 'Start_Month' then fill NAs with majority va
lue
from collections import Counter
weather_cat = ['Wind_Direction'] + weather
print("Count of missing values that will be dropped: ")
for i in weather_cat:
  df[i] = df.groupby(['Airport_Code','Month'])[i].apply(lambda x: x.fillna(Co
unter(x).most_common()[0][0]) if all(x.isnull())==False else x)
  print(i + " : " + df[i].isnull().sum().astype(str))
```

```
# drop na
df = df.dropna(subset=weather_cat)
```

```
Count of missing values that will be dropped:
Wind_Direction : 9767
Clear : 10372
Cloud : 12480
Rain : 10601
Heavy_Rain : 9358
Snow : 9480
Heavy_Snow : 9349
Fog : 9635
```

3 EXPLORATION & ENGINEERING

3.1 Resampling

Based on the exploration we did in 1.2, the accidents with severity level 4 are much more serious than accidents of other levels, between which the division is far from clear-cut. Therefore, I decided to focus on level 4 accidents and regroup the levels of severity into level 4 versus other levels.

In [29]:

```
df['Severity4'] = 0
df.loc[df['Severity'] == 4, 'Severity4'] = 1
df = df.drop(['Severity'], axis = 1)
df.Severity4.value_counts()
```

Out[29]:

```
0    2609942
1       9047
Name: Severity4, dtype: int64
```

As seen from above, the data is so unbalanced that we can hardly do exploratory analysis. To address this issue, the combination of over- and under-sampling will be used since the dataset is large enough. level 4 will be randomly oversampled to 50000 and other levels will be randomly undersampled to 50000.

In [30]:

```
def resample(dat, col, n):
    return pd.concat([dat[dat[col]==1].sample(n, replace = True),
                      dat[dat[col]==0].sample(n)], axis=0)
```

In [31]:

```
df_bl = resample(df, 'Severity4', 50000)
print('resampled data:', df_bl.Severity4.value_counts())
```

```
resampled data: 1    50000
0    50000
Name: Severity4, dtype: int64
```

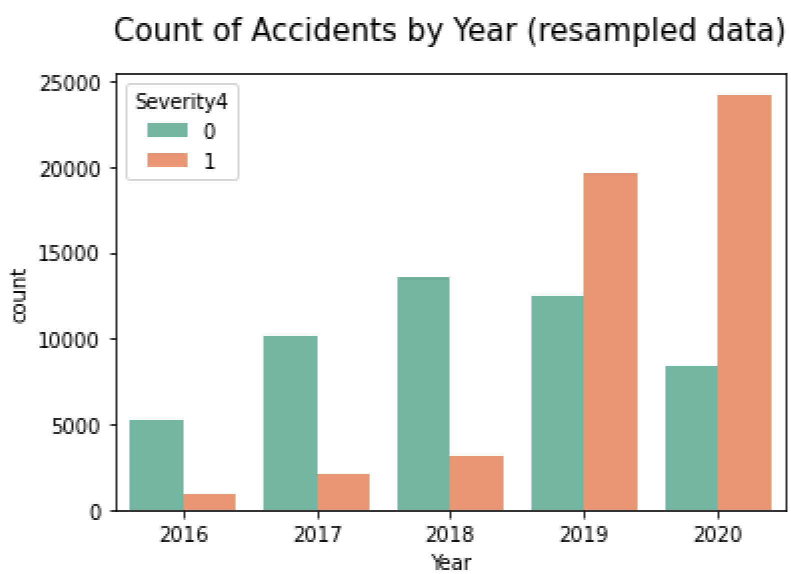
Then we can do some exploratoty analysis on resampled data.

3.2 Time Features

Year

In [32]:

```
df_bl.Year = df_bl.Year.astype(str)
sns.countplot(x='Year', hue='Severity4', data=df_bl ,palette="Set2")
plt.title('Count of Accidents by Year (resampled data)', size=15, y=1.05)
plt.show()
```



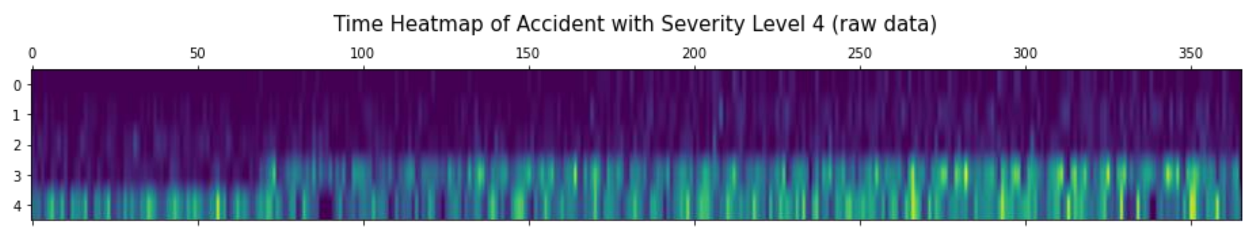
There must be something wrong. It is impossible that the number of accidents with severity level 4 after 2018 is 5 times more than the number before 2018 while the number of other levels accidents is less. Let's back to raw data to have a look.

I created a heatmap of accidents with severity level 4 from 2016 to 2020, seeing how they actually distributed.

In [33]:

```
# create a dataframe used to plot heatmap
df_date = df.loc[:,['Start_Time','Severity4']]         # create a new datefram
e only containing time and severity
df_date['date'] = df_date['Start_Time'].dt.normalize() # keep only the date pa
rt of start time
df_date = df_date.drop(['Start_Time'], axis = 1)
df_date = df_date.groupby('date').sum()                # sum the number of acc
idents with severity level 4 by date
df_date = df_date.reset_index().drop_duplicates()

# join the dataframe with full range of date from 2016 to 2020
full_date = pd.DataFrame(pd.date_range(start="2016-01-02",end="2020-12-31"))
```

```
df_date = full_date.merge(df_date, how = 'left',left_on = 0, right_on = 'date
')
df_date['date'] = df_date.iloc[:,0]
df_date = df_date.fillna(0)
df_date = df_date.iloc[:,1:].set_index('date')

# group by date
groups = df_date['Severity4'].groupby(pd.Grouper(freq='A'))
years = pd.DataFrame()
for name, group in groups:
    if name.year != 2020:
        years[name.year] = np.append(group.values,0)
    else:
        years[name.year] = group.values


# plot
years = years.T
plt.matshow(years, interpolation=None, aspect='auto')
plt.title('Time Heatmap of Accident with Severity Level 4 (raw data)', y=1.2,
fontsize=15)
plt.show()
```



The heatmap indicates that something changed after Feb 2019. Maybe it is the way that *MapQuest* defines severity or the way they collect data. Anyway, we have to narrow down our data again. Since the data after Feb 2019 is less imbalanced and the data in the future is more likely to look like this, dropping the data before Mar 2019 may be the best choice.

In [34]:

```
df = df.loc[df['Start_Time'] > "2019-03-10",:]
df = df.drop(['Year', 'Start_Time'], axis=1)
df['Severity4'].value_counts()
```

Out[34]:

```
0    967693
1      7862
Name: Severity4, dtype: int64
```

Month

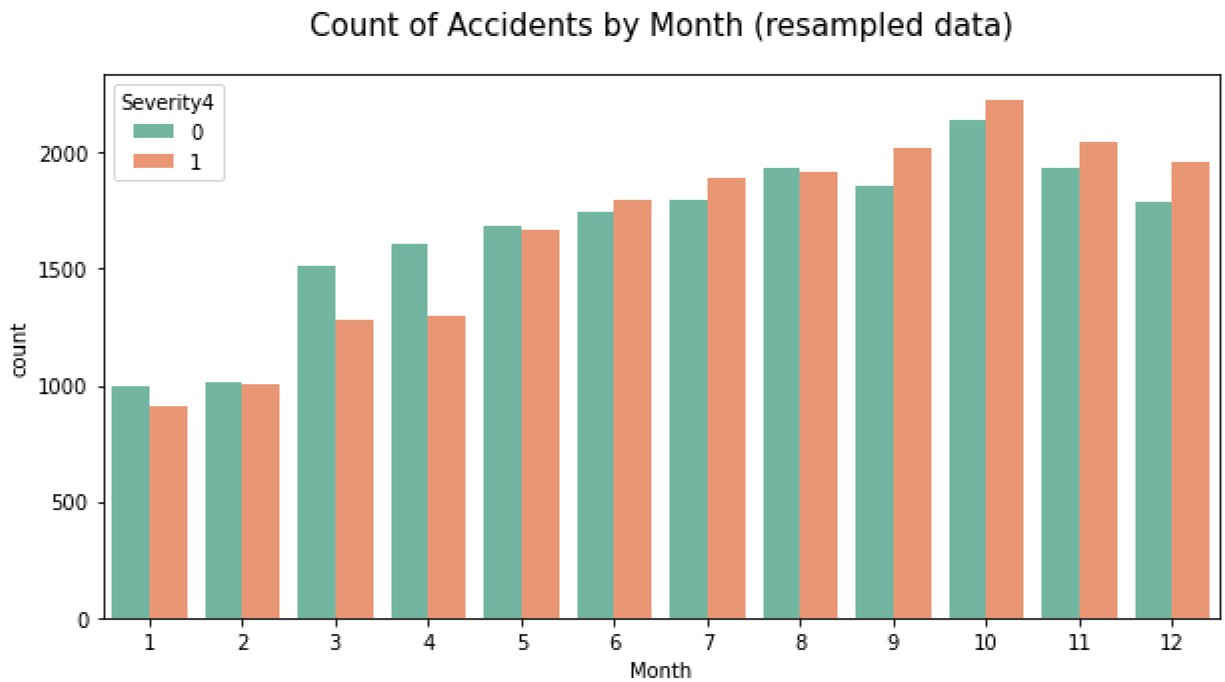
It's quite interesting that the count of other levels accidents is mostly consistent from March to December, whereas the number of level 4 accidents rapidly increased from March to May and remained stable until September then increased again from October.

In [35]:

```
df_bl = resample(df, 'Severity4', 20000)
```

In [36]:

```
plt.figure(figsize=(10,5))
sns.countplot(x='Month', hue='Severity4', data=df_bl ,palette="Set2")
plt.title('Count of Accidents by Month (resampled data)', size=15, y=1.05)
plt.show()
```

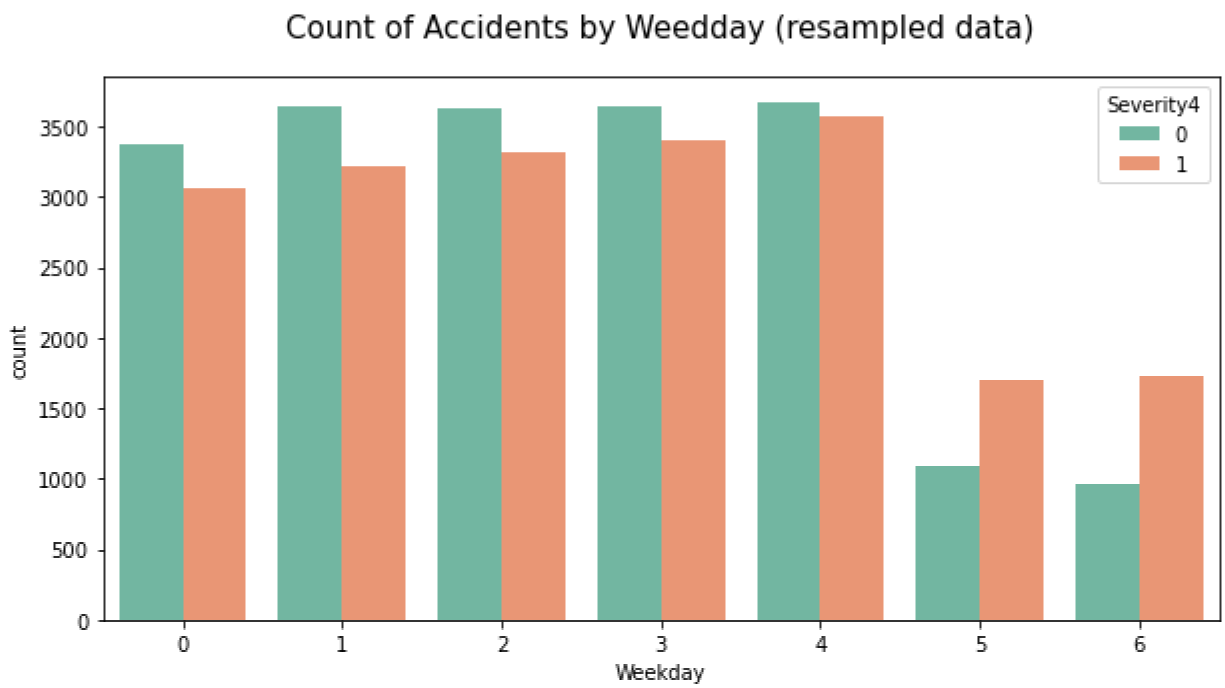


Weekday

The number of accidents was much less on weekends while the proportion of level 4 accidents was higher.

In [37]:

```
plt.figure(figsize=(10,5))
sns.countplot(x='Weekday', hue='Severity4', data=df_bl ,palette="Set2")
plt.title('Count of Accidents by Weedday (resampled data)', size=15, y=1.05)
plt.show()
```

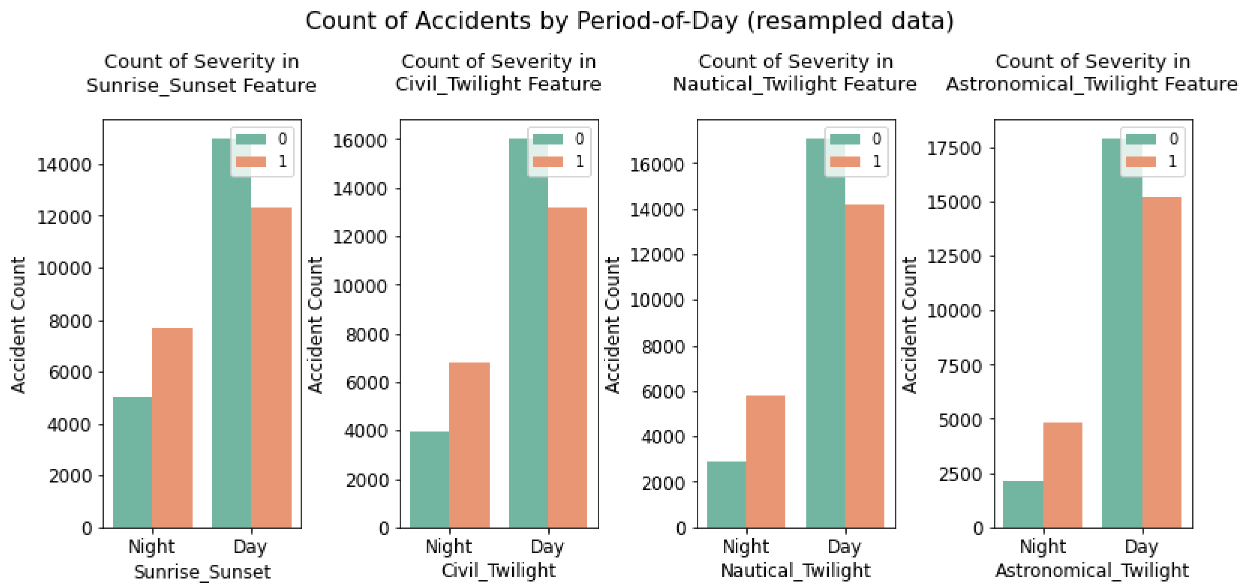


Period-of-Day

Accidents were less during the night but were more likely to be serious.

In [38]:

```
period_features = ['Sunrise_Sunset','Civil_Twilight','Nautical_Twilight','Ast
ronomical_Twilight']
fig, axs = plt.subplots(ncols=1, nrows=4, figsize=(13, 5))

plt.subplots_adjust(wspace = 0.5)
for i, feature in enumerate(period_features, 1):
    plt.subplot(1, 4, i)
    sns.countplot(x=feature, hue='Severity4', data=df_bl ,palette="Set2")

    plt.xlabel('{}'.format(feature), size=12, labelpad=3)
    plt.ylabel('Accident Count', size=12, labelpad=3)
    plt.tick_params(axis='x', labelsize=12)
    plt.tick_params(axis='y', labelsize=12)

    plt.legend(['0', '1'], loc='upper right', prop={'size': 10})
    plt.title('Count of Severity in\n{} Feature'.format(feature), size=13, y=
1.05)
fig.suptitle('Count of Accidents by Period-of-Day (resampled data)',y=1.08, f
ontsize=16)
plt.show()
```

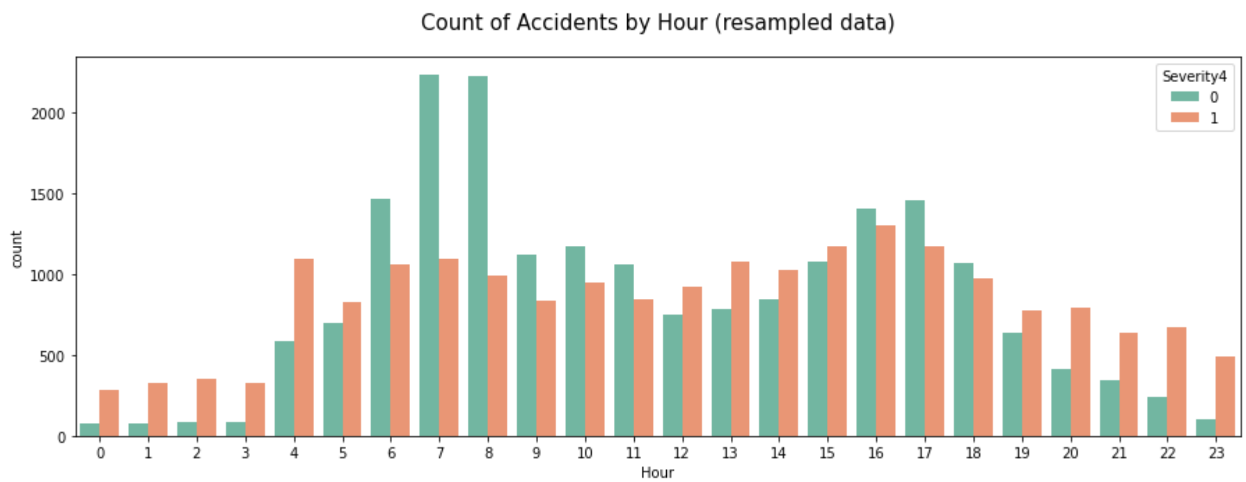


Hour

Most accidents happened during the daytime, especially AM peak and PM peak. When it comes to night, accidents were far less but more likely to be serious.

In [39]:

```
plt.figure(figsize=(15,5))
sns.countplot(x='Hour', hue='Severity4', data=df_bl ,palette="Set2")
plt.title('Count of Accidents by Hour (resampled data)', size=15, y=1.05)
plt.show()
```



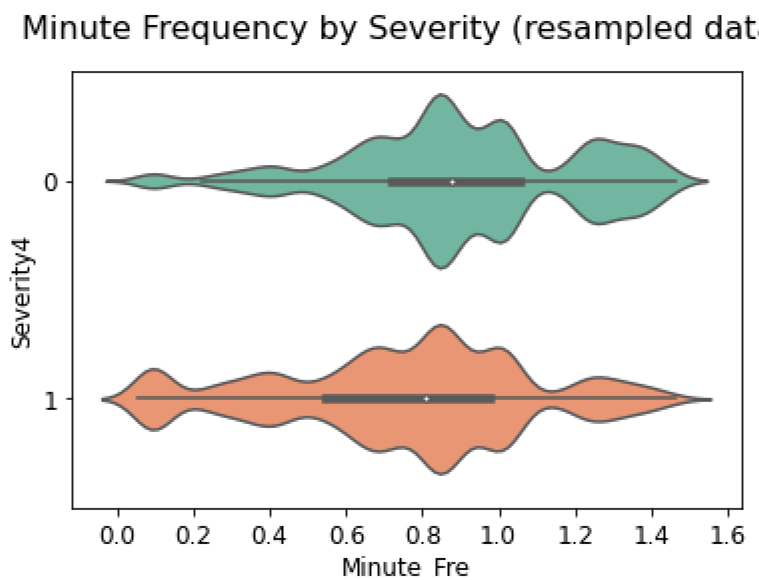
Frequence Encoding (Minute)

As seen in the plot of 'Hour', 'Minute' may also be an important predictor. But directly using it would produce an overabundance of dummy variables. Therefore, the frequency of 'Minute' was utilized as labels, rather than 'Minute' itself. To normalize the distribution, the frequency was also transformed by log.

In [40]:

```
# frequence encoding and log-transform
```

```
df['Minute_Freq'] = df.groupby(['Minute'])['Minute'].transform('count')
df['Minute_Freq'] = df['Minute_Freq']/df.shape[0]*24*60
df['Minute_Freq'] = df['Minute_Freq'].apply(lambda x: np.log(x+1))

# resampling
df_bl = resample(df, 'Severity4', 20000)

# plot
df_bl['Severity4'] = df_bl['Severity4'].astype('category')
sns.violinplot(x='Minute_Freq', y="Severity4", data=df_bl, palette="Set2")
plt.xlabel('Minute_Fre', size=12, labelpad=3)
plt.ylabel('Severity4', size=12, labelpad=3)
plt.tick_params(axis='x', labelsize=12)
plt.tick_params(axis='y', labelsize=12)
plt.title('Minute Frequency by Severity (resampled data)', size=16, y=1.05)
plt.show()
```



The violin plot shows that the overall minute frequency of accidents with severity level 4 is less than other levels. In other words, an accident is more likely to be a serious one when accidents happen less frequently.

3.3 Address Features

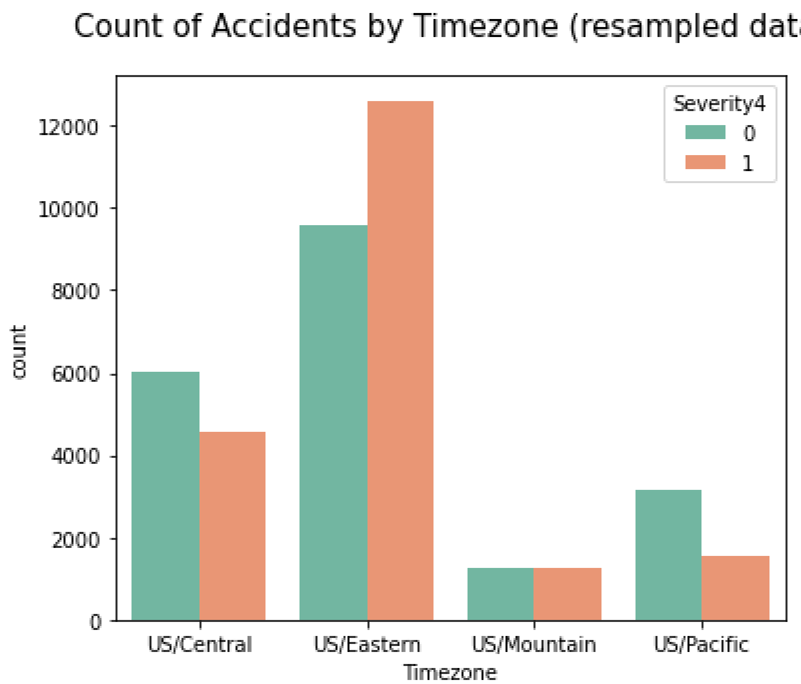
Timezone

Eastern time zone is the most dangeous one.

In [41]:

```
plt.figure(figsize=(6,5))
chart = sns.countplot(x='Timezone', hue='Severity4', data=df_bl ,palette="Set
2")
```

```
plt.title("Count of Accidents by Timezone (resampled data)", size=15, y=1.05)
plt.show()
```

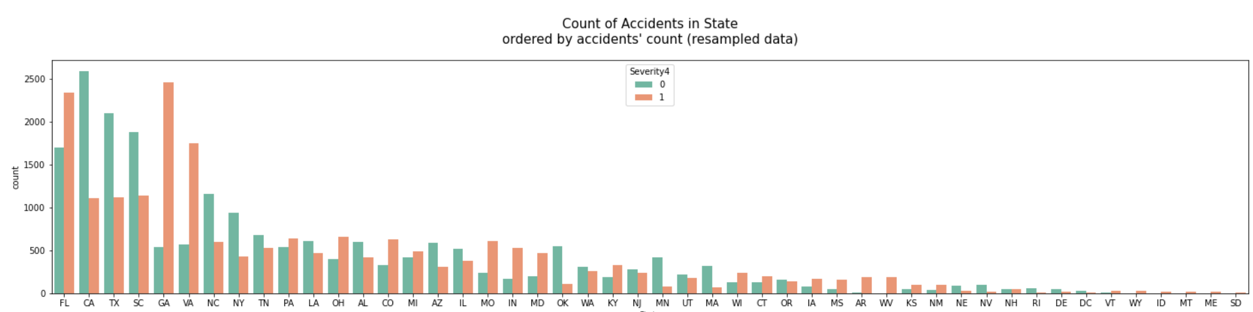


State

FL, CA, and TX are the top 3 states with the most accidents.

In [42]:

```
plt.figure(figsize=(25,5))
chart = sns.countplot(x='State', hue='Severity4',
                      data=df_bl ,palette="Set2", order=df_bl['State'].value_
counts().index)
plt.title("Count of Accidents in State\nordered by accidents' count (resample
d data)", size=15, y=1.05)
plt.show()
```

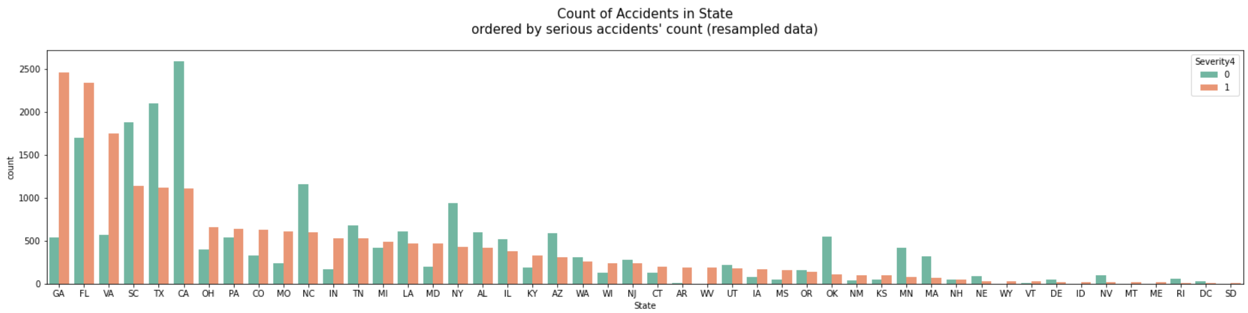


It is a different story if we order the plot by the count of accidents with severity of level 4. FL is still the top one but the next two are GA and VA.

In [43]:

```
plt.figure(figsize=(25,5))
```

```
chart = sns.countplot(x='State', hue='Severity4', data=df_bl ,palette="Set2",
order=df_bl[df_bl['Severity4']==1]['State'].value_counts().index)
plt.title("Count of Accidents in State\nordered by serious accidents' count (
resampled data)", size=15, y=1.05)
plt.show()
```



County

There are too many counties that we cannot visualize them as we did for states. But we do can incorporate census data for them.

Several basic variables, like total population, percent of commuters who drive, take transit or walk to work, and median household income, for all counties were downloaded from ACS 5-year estimates 2018. Then, counties' names were isolated.

In [44]:

```
!pip install -q censusdata
import censusdata

# download data
county = censusdata.download('acs5', 2018, censusdata.censusgeo([('county', '
*')]),
                                   ['DP05_0001E',  'DP03_0019PE','DP03_0021PE
','DP03_0022PE','DP03_0062E'],
                                   tabletype='profile')
# rename columns
county.columns = ['Population_County','Drive_County','Transit_County','Walk_C
ounty','MedianHouseholdIncome_County']
county = county.reset_index()
# extract county name and state name
county['County_y'] = county['index'].apply(lambda x : x.name.split(' County')
[0].split(',')[0]).str.lower()
county['State_y'] = county['index'].apply(lambda x : x.name.split(':')[0].spl
it(', ')[1])
```

unfold_moreShow hidden output

In [45]:

```
us_state_abbrev = {
    'Alabama': 'AL',
    'Alaska': 'AK',
    'American Samoa': 'AS',
```

```
'Arizona': 'AZ',
'Arkansas': 'AR',
'California': 'CA',
'Colorado': 'CO',
'Connecticut': 'CT',
'Delaware': 'DE',
'District of Columbia': 'DC',
'Florida': 'FL',
'Georgia': 'GA',
'Guam': 'GU',
'Hawaii': 'HI',
'Idaho': 'ID',
'Illinois': 'IL',
'Indiana': 'IN',
'Iowa': 'IA',
'Kansas': 'KS',
'Kentucky': 'KY',
'Louisiana': 'LA',
'Maine': 'ME',
'Maryland': 'MD',
'Massachusetts': 'MA',
'Michigan': 'MI',
'Minnesota': 'MN',
'Mississippi': 'MS',
'Missouri': 'MO',
'Montana': 'MT',
'Nebraska': 'NE',
'Nevada': 'NV',
'New Hampshire': 'NH',
'New Jersey': 'NJ',
'New Mexico': 'NM',
'New York': 'NY',
'North Carolina': 'NC',
'North Dakota': 'ND',
'Northern Mariana Islands':'MP',
'Ohio': 'OH',
'Oklahoma': 'OK',
'Oregon': 'OR',
'Pennsylvania': 'PA',
'Puerto Rico': 'PR',
'Rhode Island': 'RI',
'South Carolina': 'SC',
'South Dakota': 'SD',
'Tennessee': 'TN',
'Texas': 'TX',
'Utah': 'UT',
'Vermont': 'VT',
'Virgin Islands': 'VI',
'Virginia': 'VA',
```

```
    'Washington': 'WA',
    'West Virginia': 'WV',
    'Wisconsin': 'WI',
    'Wyoming': 'WY'
}
county['State_y'] = county['State_y'].replace(us_state_abbrev)
```

In [46]:

```
county.head()
```

Out[46]:

	index	Population_County	Drive_County	Transit_County	Walk_County	MedianHouseholdIncome_County	County_y	State_y
0	Washington County, Mississippi: Summary level:...	47086	86.4	0.0	1.3	30834	washington	MS
1	Perry County, Mississippi: Summary level: 050,...	12028	85.8	0.0	1.8	39007	perry	MS
2	Choctaw County, Mississippi: Summary level: 05...	8321	85.6	0.3	1.1	37203	choctaw	MS
3	Itawamba County, Mississippi: Summar	23480	82.4	0.2	0.7	40510	itawamba	MS

	index	Population_County	Drive_County	Transit_County	Walk_County	MedianHouseholdIncome_County	County_y	State_y
	y level: 0...							
4	Carroll County, Mississippi: Summary level: 05...	10129	90.0	0.0	1.4	43060	carroll	MS

Counties' names turned out to be very tricky. Converting all of them into lowercase is not enough. Some counties name in USA-accidents omit "city" or "parish", and hence can't be matched with names in census data. We need to manually put them back and rejoin them.

In [47]:

```
df.shape
```

Out[47]:

```
(975555, 48)
```

In [48]:

```
# convert all county name to lowercase
df['County'] = df['County'].str.lower()

# left join df with census data
df = df.merge(county, left_on = ['County','State'], right_on=['County_y','Sta
te_y'],how = 'left').drop(['County_y','State_y'], axis = 1)
join_var = county.columns.to_list()[:-2]

# check how many miss match we got
print('Count of missing values before: ', df[join_var].isnull().sum())

# add "city" and match again
df_city = df[df['Walk_County'].isnull()].drop(join_var, axis=1)
df_city['County_city'] = df_city['County'].apply(lambda x : x + ' city')
df_city = df_city.merge(county,left_on= ['County_city','State'],right_on = ['
County_y','State_y'], how = 'left').drop(['County_city','County_y','State_y']
, axis=1)
df = pd.concat((df[df['Walk_County'].isnull()==False], df_city), axis=0)

# add "parish" and match again
df_parish = df[df['Walk_County'].isnull()].drop(join_var, axis=1)
df_parish['County_parish'] = df_parish['County'].apply(lambda x : x + ' paris
h')
```

```
df_parish = df_parish.merge(county,left_on= ['County_parish','State'],right_o
n = ['County_y','State_y'], how = 'left').drop(['County_parish','County_y','S
tate_y'], axis=1)
df = pd.concat((df[df['Walk_County'].isnull()==False], df_parish), axis=0)
print('Count of missing values after: ', df[join_var].isnull().sum())
```

```
Count of missing values before:  index                          41514
Population_County                41514
Drive_County                     41514
Transit_County                   41514
Walk_County                      41514
MedianHouseholdIncome_County     41514
dtype: int64
Count of missing values after:  index                          9248
Population_County                9248
Drive_County                     9248
Transit_County                   9248
Walk_County                      9248
MedianHouseholdIncome_County     9248
dtype: int64
```

Drop na and use Logit transformation on some variables having extremly skewed distribution.

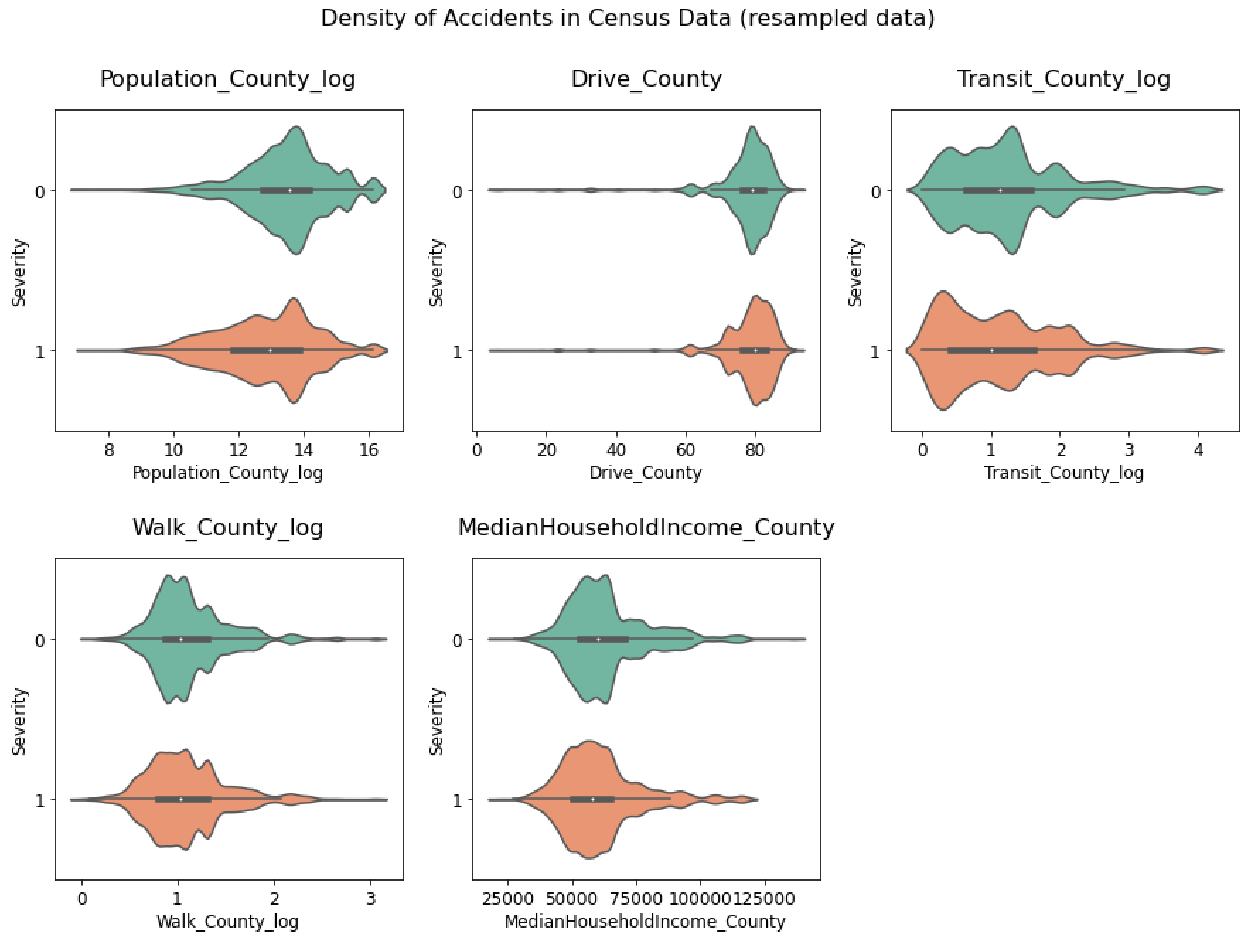
In [49]:

```
# drop na
df = df.drop('index', axis = 1).dropna()

# log-transform
for i in ['Population_County','Transit_County','Walk_County']:
    df[i + '_log'] = df[i].apply(lambda x: np.log(x+1))
df = df.drop(['Population_County','Transit_County','Walk_County'], axis = 1)
```

In [50]:

```
# resample again
df_bl = resample(df, 'Severity4', 20000)

# plot
df_bl['Severity4'] = df_bl['Severity4'].astype('category')
census_features = ['Population_County_log','Drive_County','Transit_County_log
','Walk_County_log','MedianHouseholdIncome_County']
fig, axs = plt.subplots(ncols=2, nrows=3, figsize=(15, 10))
plt.subplots_adjust(hspace=0.4,wspace = 0.2)
for i, feature in enumerate(census_features, 1):
    plt.subplot(2, 3, i)
    sns.violinplot(x=feature, y="Severity4", data=df_bl, palette="Set2")

    plt.xlabel('{}'.format(feature), size=12, labelpad=3)
    plt.ylabel('Severity', size=12, labelpad=3)
    plt.tick_params(axis='x', labelsize=12)
    plt.tick_params(axis='y', labelsize=12)
```

```
    plt.title('{}'.format(feature), size=16, y=1.05)
fig.suptitle('Density of Accidents in Census Data (resampled data)', fontsize
=16)
plt.show()
```



Percent of people taking transit to commute seems to related to severity. Level 4 accidents happened more frequently in those counties with a lower usage rate of transit.

Street

There are more and more studies found that higher speed limits were associated with an increased likelihood of crashes and deaths. (https://www.cga.ct.gov/2013/rpt/2013-R-0074.htm) And speed limits are highly related to street type. Street type hence can be a good predictor of serious accidents. There is no feature about street type in the original dataset though, we can extract it from the street name.

The top 40 most common words in street names were selected. This list contains not only street types but also some common words widely used in street names.

In [51]:

```
# create a list of top 40 most common words in street name
st_type =' '.join(df['Street'].unique().tolist()) # flat the array of street n
ame
```

```
st_type = re.split(" |-", st_type) # split the long string by space and hyphen
st_type = [x[0] for x in Counter(st_type).most_common(40)] # select the 40 mos
t common words
print('the 40 most common words')
print(*st_type, sep = ", ")
```

```
the 40 most common words
Rd, Dr, St, Ave, N, S, E, W, Blvd, Ln, Highway, Way, Pkwy, Hwy, Ct, SW, NE
, Pl, NW, State, Old, SE, Road, Cir, US, Creek, County, Hill, Park, Route,
Lake, Trl, I, Valley, Ridge, Mill, River, Oak, Pike, Loop
```

Remove some irrelevant words and add spaces and hyphen back

In [52]:

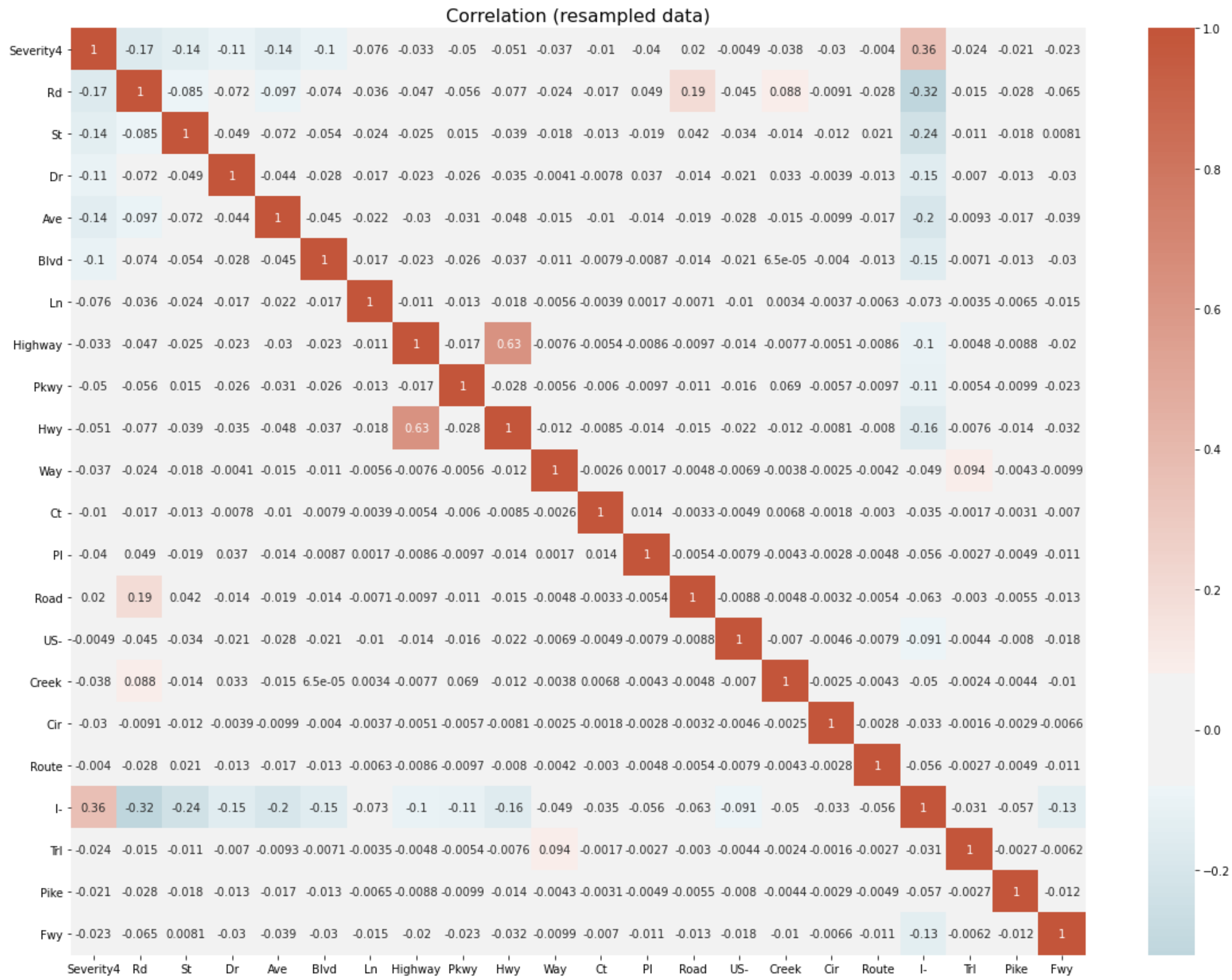
```
# Remove some irrelevant words and add spaces and hyphen back
st_type= [' Rd', ' St', ' Dr', ' Ave', ' Blvd', ' Ln', ' Highway', ' Pkwy', '
Hwy',
          ' Way', ' Ct', 'Pl', ' Road', 'US-', 'Creek', ' Cir',  'Route',
          'I-', 'Trl', 'Pike', ' Fwy']
print(*st_type, sep = ", ")
```

```
 Rd,  St,  Dr,  Ave,  Blvd,  Ln,  Highway,  Pkwy,  Hwy,  Way,  Ct, Pl,  Ro
ad, US-, Creek,  Cir, Route, I-, Trl, Pike,  Fwy
```

Create a dummy variable for each word in the list and plot the correlation between these key words and severity.

In [53]:

```
# for each word create a boolean column
for i in st_type:
  df[i.strip()] = np.where(df['Street'].str.contains(i, case=True, na = False
), True, False)
df.loc[df['Road']==1,'Rd'] = True
df.loc[df['Highway']==1,'Hwy'] = True

# resample again
df_bl = resample(df, 'Severity4', 20000)

# plot correlation
df_bl['Severity4'] = df_bl['Severity4'].astype(int)
street_corr  = df_bl.loc[:,['Severity4']+[x.strip() for x in st_type]].corr()
plt.figure(figsize=(20,15))
cmap = sns.diverging_palette(220, 20, sep=20, as_cmap=True)
sns.heatmap(street_corr, annot=True, cmap=cmap, center=0).set_title("Correlat
ion (resampled data)", fontsize=16)
plt.show()
```



Interstate Highway turns out to be the most dangerous street. Other roads like basic road, street, drive, and avenue are relatively safe. Let's just keep these five features.

In [54]:

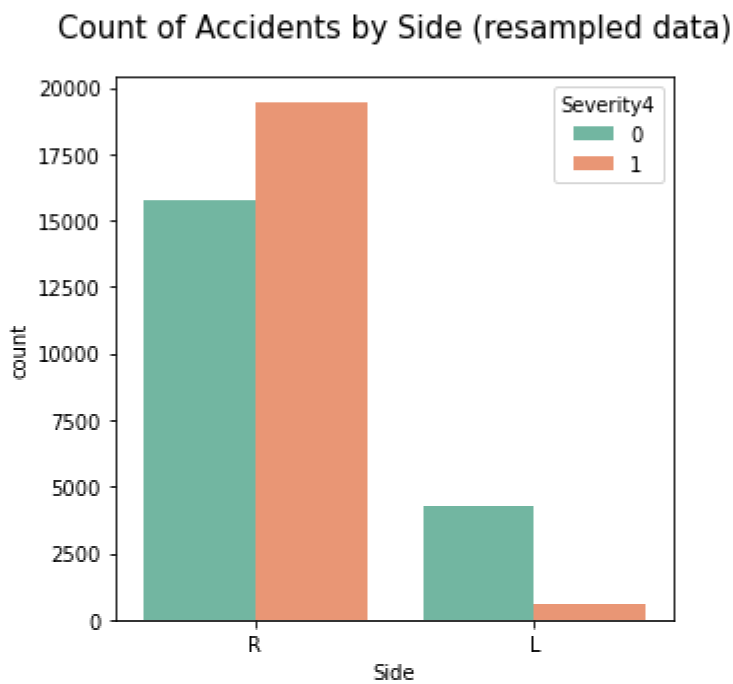
```
drop_list = street_corr.index[street_corr['Severity4'].abs()<0.1].to_list()
df = df.drop(drop_list, axis=1)

# resample again
df_bl = resample(df, 'Severity4', 20000)
```

Side

Right side of the line is much more dangerous than left side.

In [55]:

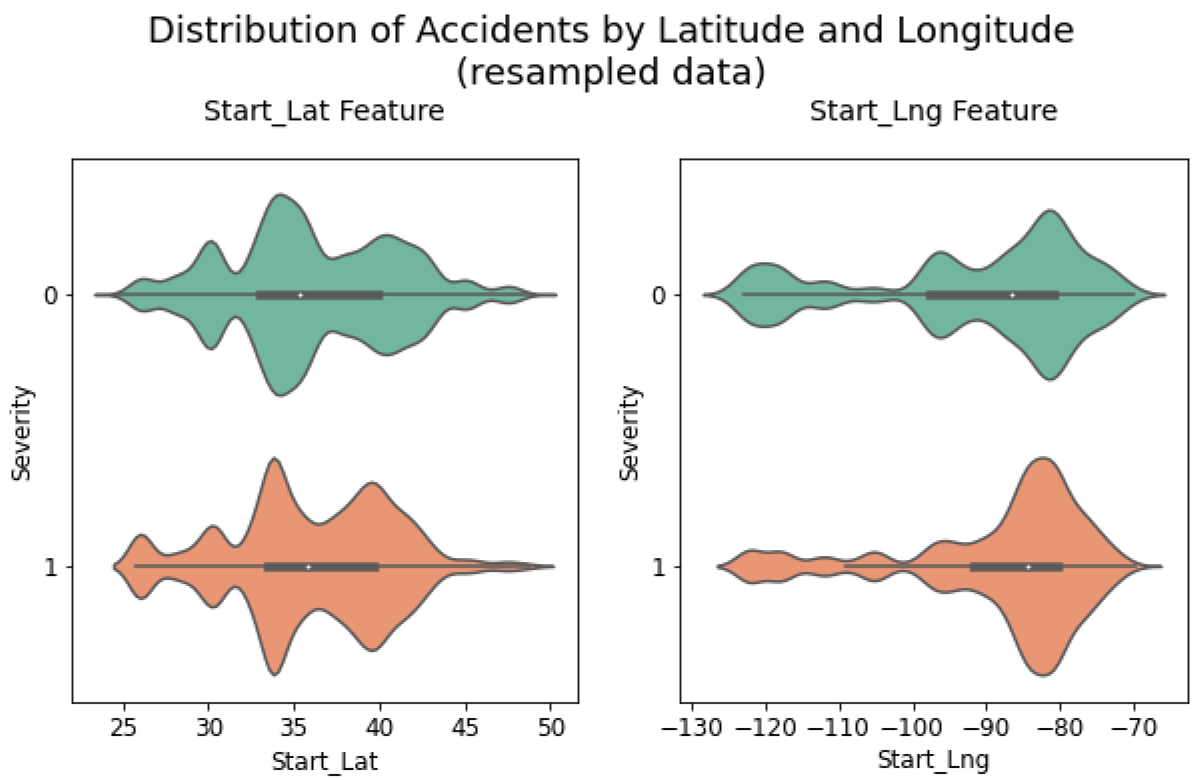
```
plt.figure(figsize=(5,5))
chart = sns.countplot(x='Side', hue='Severity4', data=df_bl ,palette="Set2")
plt.title("Count of Accidents by Side (resampled data)", size=15, y=1.05)
plt.show()
```



Latitude and Longitude

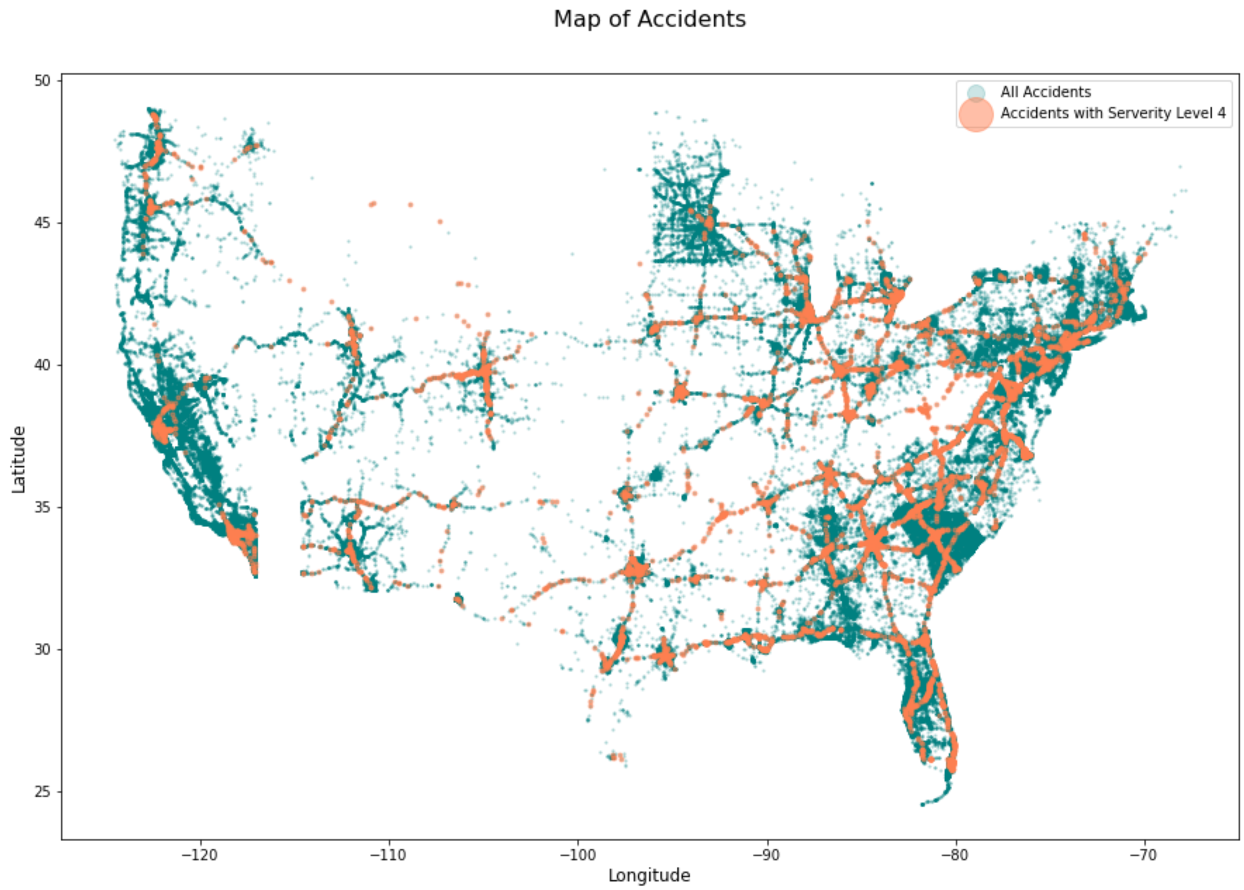
In [56]:

```
df_bl['Severity4'] = df_bl['Severity4'].astype('category')
num_features = ['Start_Lat', 'Start_Lng']
fig, axs = plt.subplots(ncols=1, nrows=2, figsize=(10, 5))
plt.subplots_adjust(hspace=0.4,wspace = 0.2)
for i, feature in enumerate(num_features, 1):
    plt.subplot(1, 2, i)
    sns.violinplot(x=feature, y="Severity4", data=df_bl, palette="Set2")

    plt.xlabel('{}'.format(feature), size=12, labelpad=3)
    plt.ylabel('Severity', size=12, labelpad=3)
    plt.tick_params(axis='x', labelsize=12)
    plt.tick_params(axis='y', labelsize=12)

    plt.title('{} Feature'.format(feature), size=14, y=1.05)
fig.suptitle('Distribution of Accidents by Latitude and Longitude\n(resampled
data)', fontsize=18,y=1.08)
plt.show()
```



In [57]:

```
df_4 = df[df['Severity4']==1]

plt.figure(figsize=(15,10))

plt.plot( 'Start_Lng', 'Start_Lat', data=df, linestyle='', marker='o', marker
size=1.5, color="teal", alpha=0.2, label='All Accidents')
plt.plot( 'Start_Lng', 'Start_Lat', data=df_4, linestyle='', marker='o', mark
ersize=3, color="coral", alpha=0.5, label='Accidents with Serverity Level 4')
plt.legend(markerscale=8)
plt.xlabel('Longitude', size=12, labelpad=3)
plt.ylabel('Latitude', size=12, labelpad=3)
plt.title('Map of Accidents', size=16, y=1.05)
plt.show()
```



Frequency Encoding

Similar to 'Minute', some location features like 'City' and 'Zipcode' that have too many unique values can be labeled by their frequency. Frequency encoding and log-transform:

1. 'Street'
2. 'City'
3. 'County'
4. 'Zipcode'
5. 'Airport_Code'

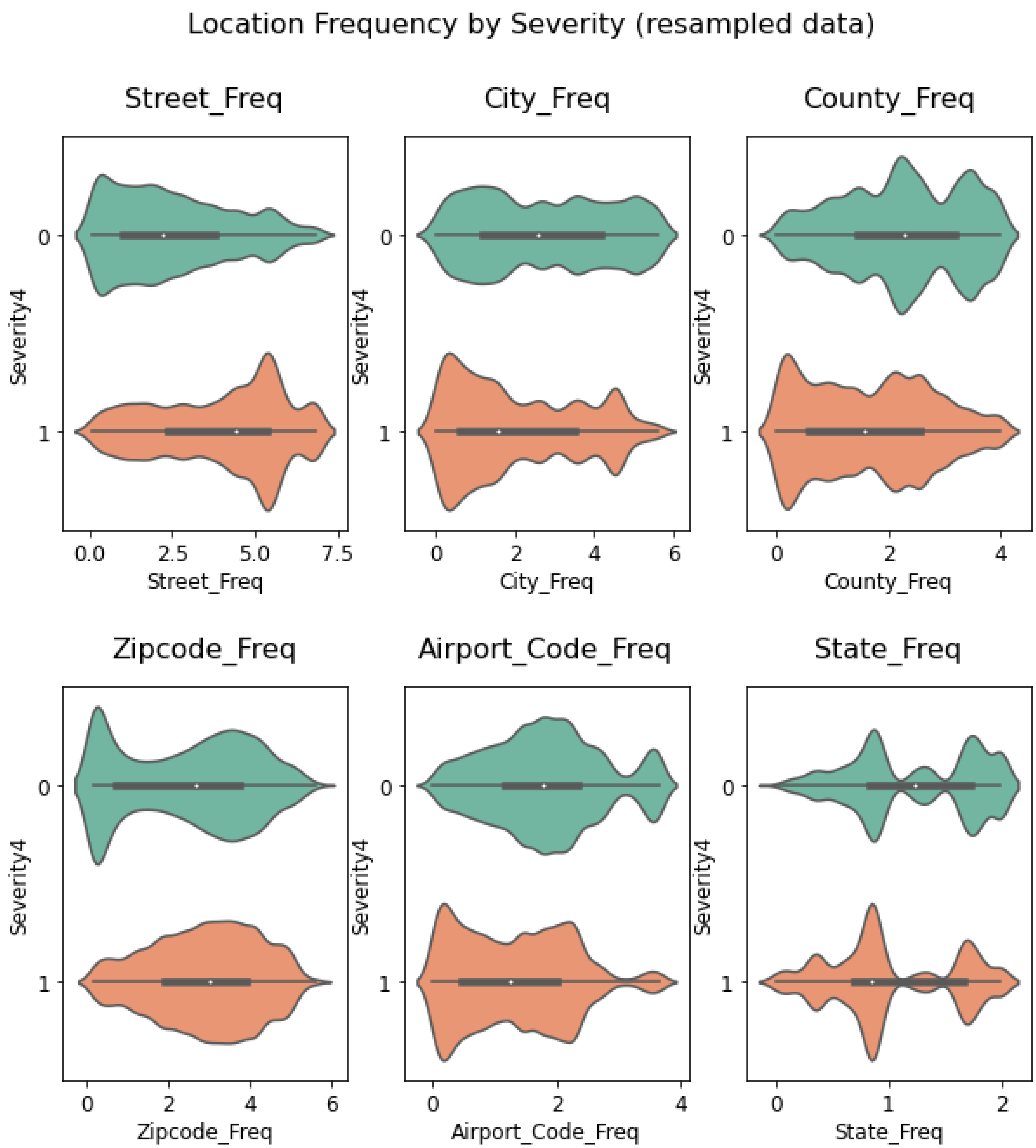
In [58]:

```
fre_list = ['Street', 'City', 'County', 'Zipcode', 'Airport_Code','State']
for i in fre_list:
  newname = i + '_Freq'
  df[newname] = df.groupby([i])[i].transform('count')
  df[newname] = df[newname]/df.shape[0]*df[i].unique().size
  df[newname] = df[newname].apply(lambda x: np.log(x+1))
```

In [59]:

```
# resample again
df_bl = resample(df, 'Severity4', 20000)

df_bl['Severity4'] = df_bl['Severity4'].astype('category')
```

```python
fig, axs = plt.subplots(ncols=2, nrows=3, figsize=(10, 10))
plt.subplots_adjust(hspace=0.4,wspace = 0.2)
fig.suptitle('Location Frequency by Severity (resampled data)', fontsize=16)
for i, feature in enumerate(fre_list, 1):
    feature = feature + '_Freq'
    plt.subplot(2, 3, i)
    sns.violinplot(x=feature, y="Severity4", data=df_bl, palette="Set2")

    plt.xlabel('{}'.format(feature), size=12, labelpad=3)
    plt.ylabel('Severity4', size=12, labelpad=3)
    plt.tick_params(axis='x', labelsize=12)
    plt.tick_params(axis='y', labelsize=12)

    plt.title('{}'.format(feature), size=16, y=1.05)
plt.show()
```



Two opposite patterns can be identified in these plots. For 'Street' and 'Zipcode', higher frequency means higher likelihood of being a serious accident. In contrast with these smaller regions, for 'City' and 'Airport_Code' instead, higher frequency means less likelihood of being a serious accident. Get rid of features we don't need anymore.

In [60]:

```
df = df.drop(fre_list, axis  = 1)
```

3.4 Weather Features

Continuous Weather Features

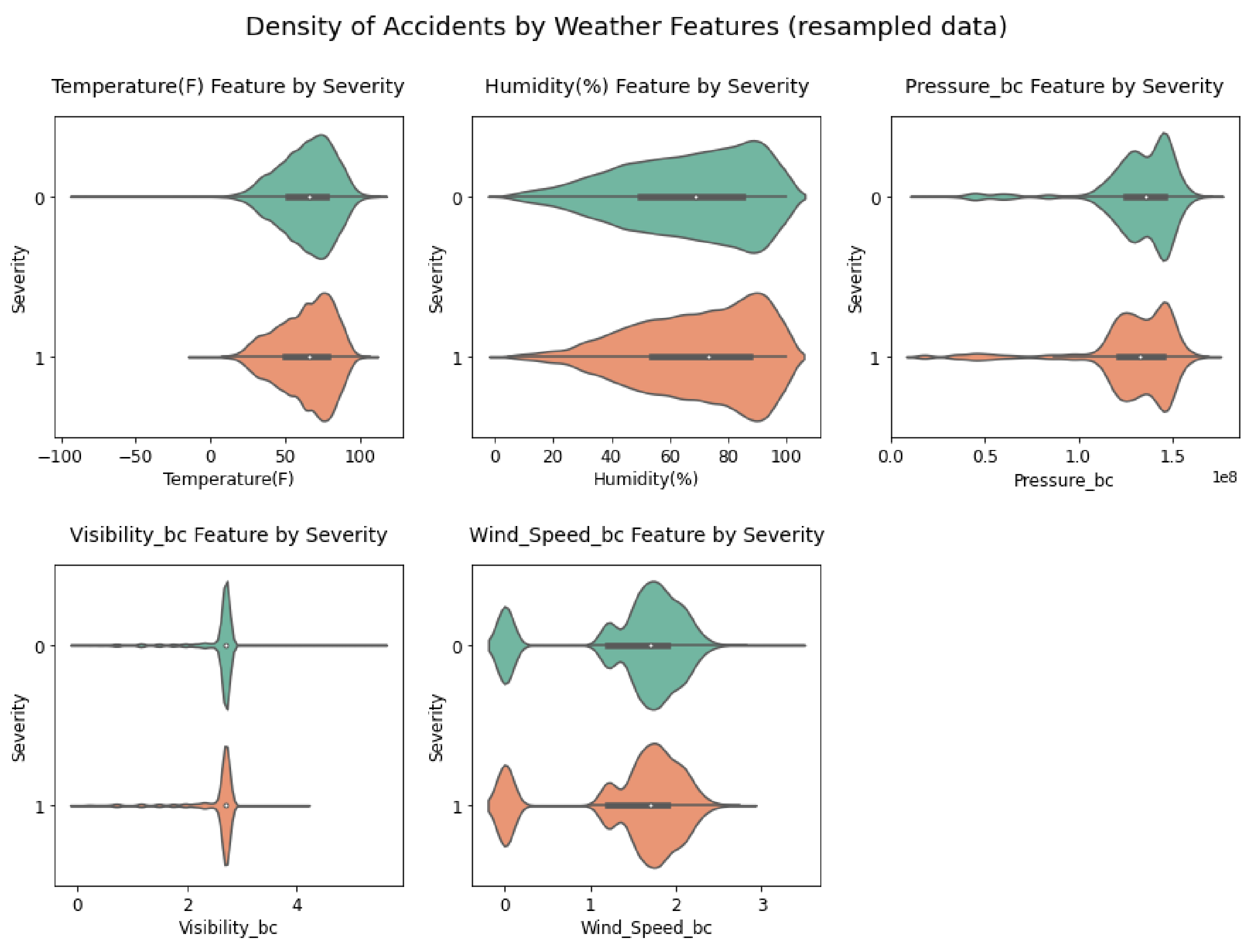
Normalize features with extreamly skewed distribution first.

In [61]:

```
df['Pressure_bc']= boxcox(df['Pressure(in)'].apply(lambda x: x+1),lmbda=6)
df['Visibility_bc']= boxcox(df['Visibility(mi)'].apply(lambda x: x+1),lmbda = 0.1)
df['Wind_Speed_bc']= boxcox(df['Wind_Speed(mph)'].apply(lambda x: x+1),lmbda=-0.2)
df = df.drop(['Pressure(in)','Visibility(mi)','Wind_Speed(mph)'], axis=1)
```

In [62]:

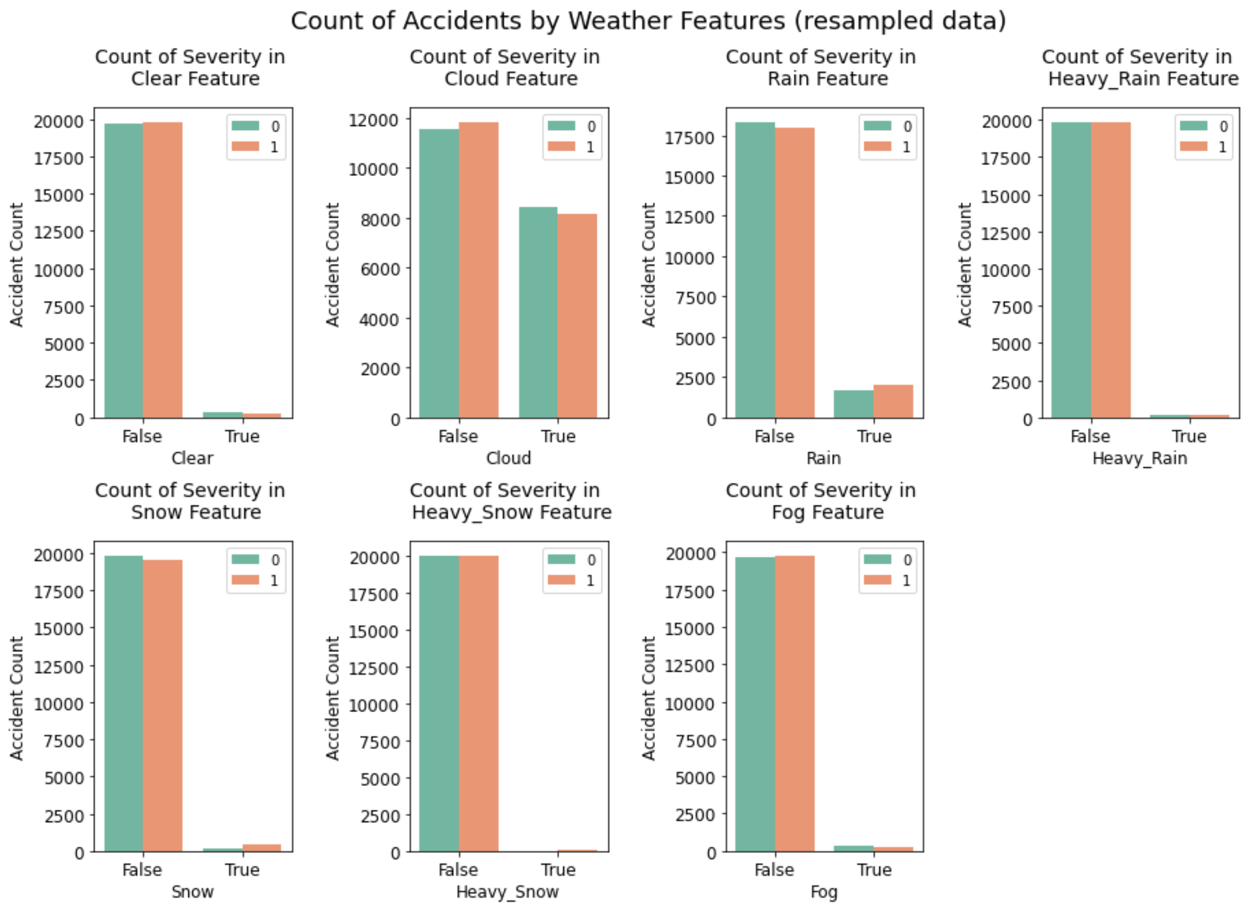
```
# resample again
df_bl = resample(df, 'Severity4', 20000)

df_bl['Severity4'] = df_bl['Severity4'].astype('category')
num_features = ['Temperature(F)', 'Humidity(%)', 'Pressure_bc', 'Visibility_bc', 'Wind_Speed_bc']
fig, axs = plt.subplots(ncols=2, nrows=3, figsize=(15, 10))
plt.subplots_adjust(hspace=0.4,wspace = 0.2)
for i, feature in enumerate(num_features, 1):
    plt.subplot(2, 3, i)
    sns.violinplot(x=feature, y="Severity4", data=df_bl, palette="Set2")

    plt.xlabel('{}'.format(feature), size=12, labelpad=3)
    plt.ylabel('Severity', size=12, labelpad=3)
    plt.tick_params(axis='x', labelsize=12)
    plt.tick_params(axis='y', labelsize=12)

    plt.title('{} Feature by Severity'.format(feature), size=14, y=1.05)
fig.suptitle('Density of Accidents by Weather Features (resampled data)', fontsize=18)
plt.show()
```



Weather Conditions

In [63]:

```
fig, axs = plt.subplots(ncols=2, nrows=4, figsize=(15, 10))
plt.subplots_adjust(hspace=0.4,wspace = 0.6)
for i, feature in enumerate(weather, 1):
    plt.subplot(2, 4, i)
    sns.countplot(x=feature, hue='Severity4', data=df_bl ,palette="Set2")

    plt.xlabel('{}'.format(feature), size=12, labelpad=3)
    plt.ylabel('Accident Count', size=12, labelpad=3)
    plt.tick_params(axis='x', labelsize=12)
    plt.tick_params(axis='y', labelsize=12)

    plt.legend(['0', '1'], loc='upper right', prop={'size': 10})
    plt.title('Count of Severity in \n {} Feature'.format(feature), size=14,
y=1.05)
fig.suptitle('Count of Accidents by Weather Features (resampled data)', fonts
ize=18)
plt.show()
```



As seen from above, accidents are little more likely to be serious during rain or snow while less likely on a cloudy day.

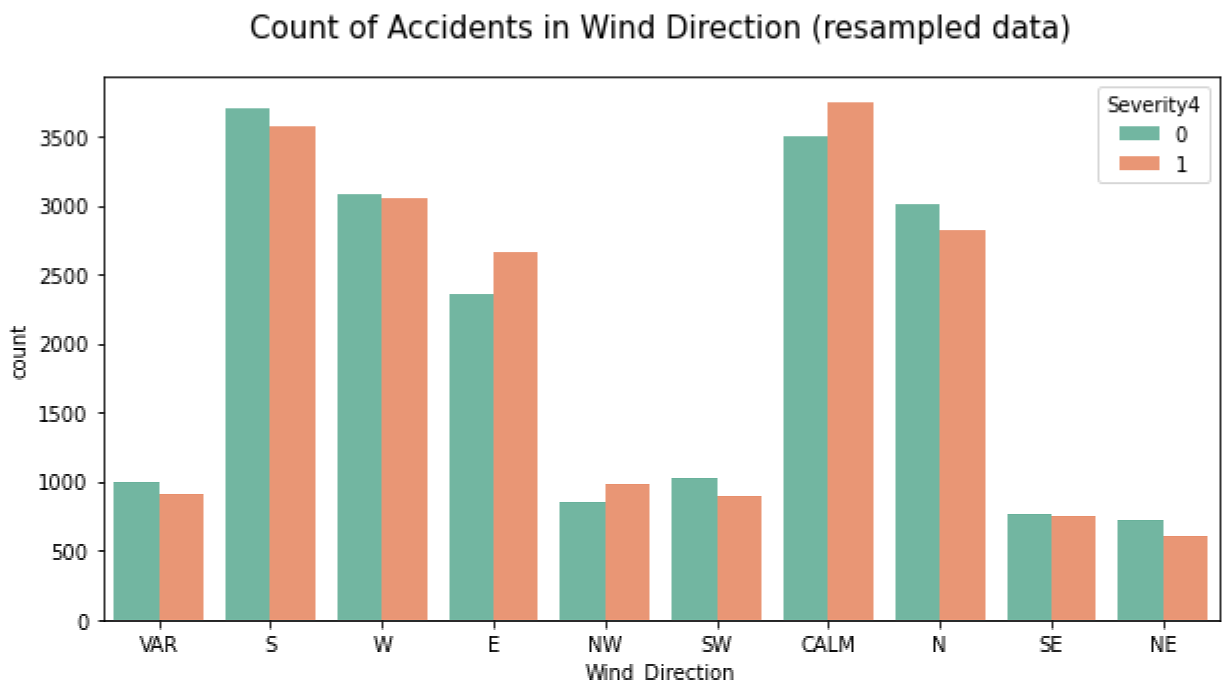
In [65]:

```
df = df.drop(['Heavy_Rain','Heavy_Snow','Fog'], axis  = 1)
```

Wind Direction

In [66]:

```
plt.figure(figsize=(10,5))
chart = sns.countplot(x='Wind_Direction', hue='Severity4', data=df_bl ,palett
e="Set2")
plt.title("Count of Accidents in Wind Direction (resampled data)", size=15, y
=1.05)
plt.show()
```



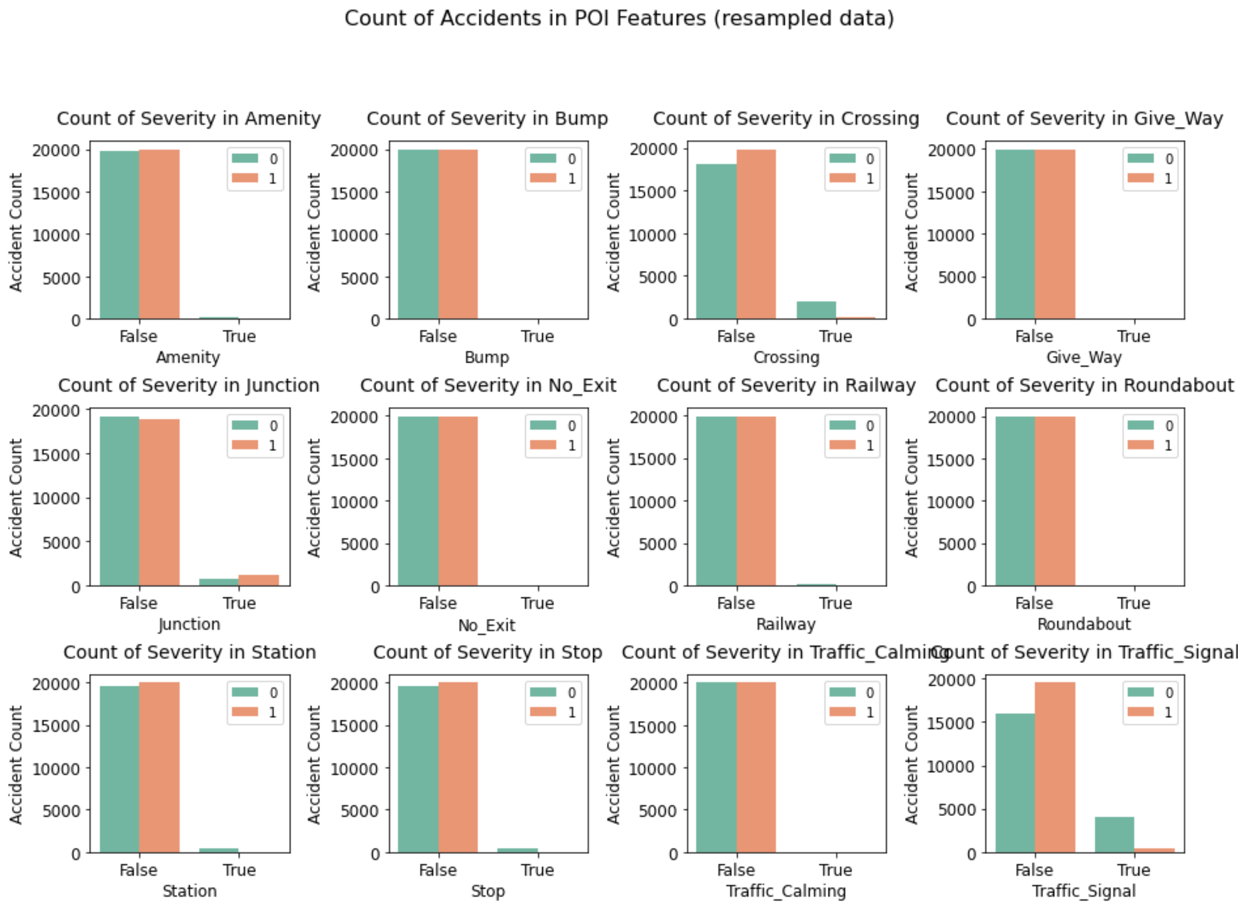
In [67]:

```
df = df.drop(['Wind_Direction'], axis=1)
```

3.5 POI Features

In [68]:

```
POI_features = ['Amenity','Bump','Crossing','Give_Way','Junction','No_Exit','
Railway','Roundabout','Station','Stop','Traffic_Calming','Traffic_Signal']

fig, axs = plt.subplots(ncols=3, nrows=4, figsize=(15, 10))

plt.subplots_adjust(hspace=0.5,wspace = 0.5)
for i, feature in enumerate(POI_features, 1):
    plt.subplot(3, 4, i)
    sns.countplot(x=feature, hue='Severity4', data=df_bl ,palette="Set2")

    plt.xlabel('{}'.format(feature), size=12, labelpad=3)
    plt.ylabel('Accident Count', size=12, labelpad=3)
    plt.tick_params(axis='x', labelsize=12)
    plt.tick_params(axis='y', labelsize=12)

    plt.legend(['0', '1'], loc='upper right', prop={'size': 10})
    plt.title('Count of Severity in {}'.format(feature), size=14, y=1.05)
fig.suptitle('Count of Accidents in POI Features (resampled data)',y=1.02, fo
ntsize=16)
plt.show()
```



Accidents near traffic signal and crossing are much less likely to be serious accidents while little more likely to be serious if they are near the junction. Maybe it is because people usually slow down in front of crossing and traffic signal but junction and severity are highly related to speed. Other POI features are so unbalanced that it is hard to tell their relation with severity from plots.

Drop some features:

1. 'Bump'
2. 'Give_Way'
3. 'No_Exit'
4. 'Roundabout'
5. 'Traffic_Calming'

In [69]:

```
df= df.drop(['Amenity','Bump','Give_Way','No_Exit','Roundabout','Traffic_Calm
ing'], axis=1)
```

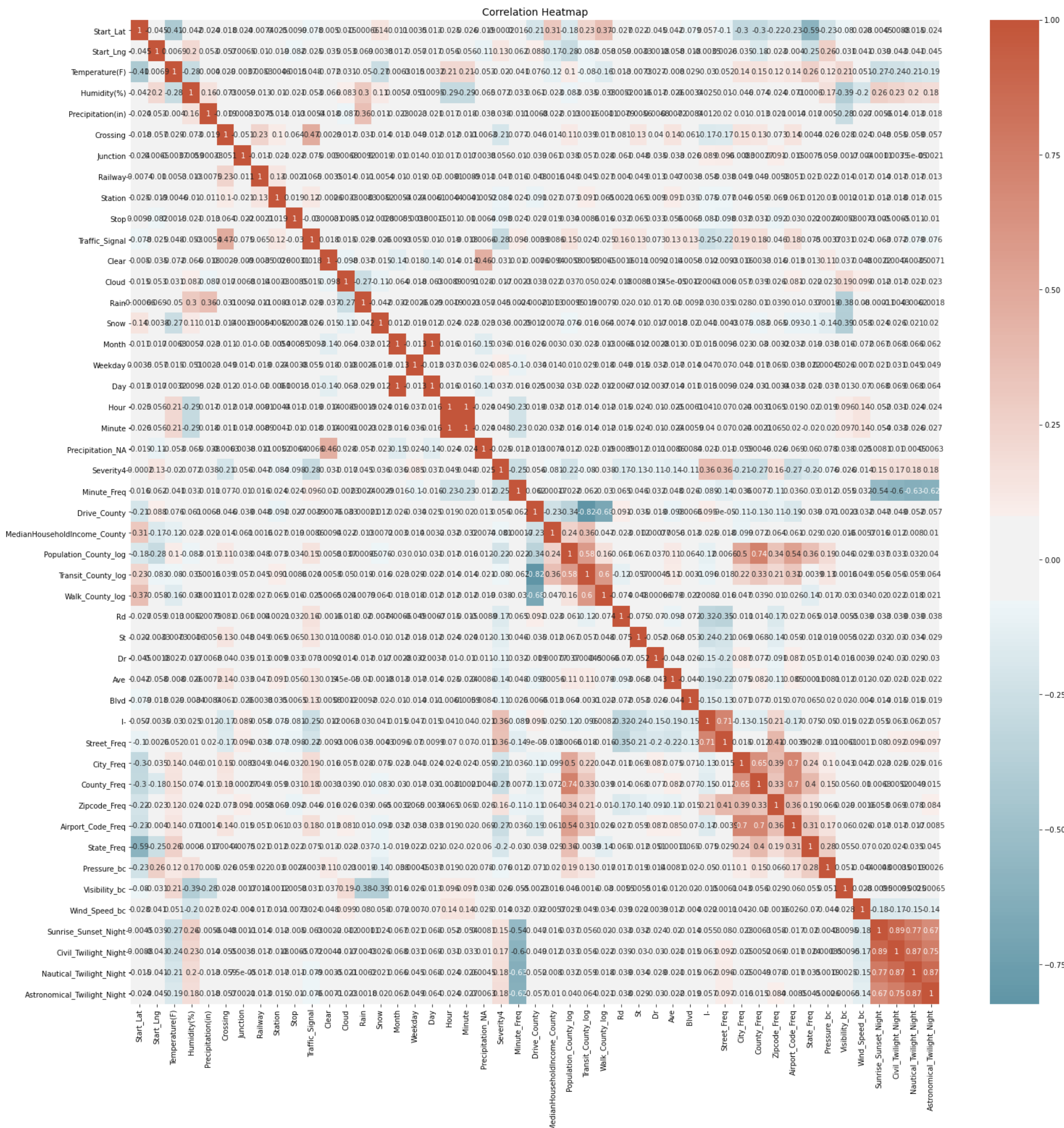
3.6 Correlation

In [ ]:

```
# one-hot encoding
df[period_features] = df[period_features].astype('category')
df = pd.get_dummies(df, columns=period_features, drop_first=True)
```

In [74]:

```
# resample again
df_bl = resample(df, 'Severity4', 20000)

# plot correlation
df_bl['Severity4'] = df_bl['Severity4'].astype(int)
plt.figure(figsize=(25,25))
cmap = sns.diverging_palette(220, 20, sep=20, as_cmap=True)
sns.heatmap(df_bl.corr(), annot=True,cmap=cmap, center=0).set_title("Correlation Heatmap", fontsize=14)
plt.show()
```

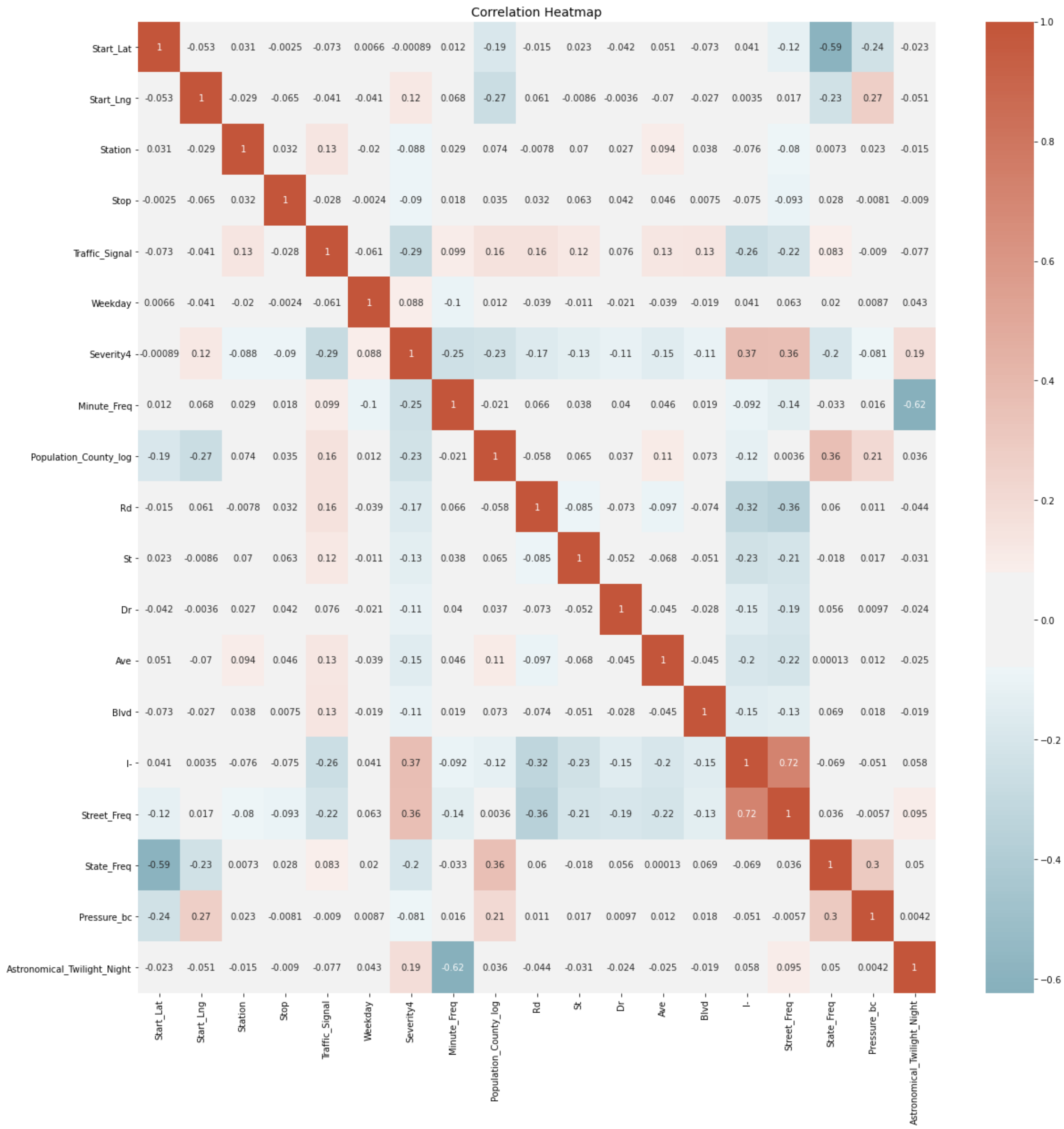


In [75]:

```
df = df.drop(['Temperature(F)', 'Humidity(%)', 'Precipitation(in)', 'Precipitation_NA','Visibility_bc', 'Wind_Speed_bc',
              'Clear','Cloud','Snow','Crossing','Junction','Railway','Month',
              'Hour', 'Day','Minute','MedianHouseholdIncome_County', 'Transit_County_log',
              'Walk_County_log','Drive_County', 'City_Freq','County_Freq','Airport_Code_Freq','Zipcode_Freq',
              'Sunrise_Sunset_Night', 'Civil_Twilight_Night', 'Nautical_Twilight_Night'], axis=1)
```

In [80]:

```
# resample again
df_bl = resample(df, 'Severity4', 20000)

# plot correlation
df_bl['Severity4'] = df_bl['Severity4'].astype(int)
plt.figure(figsize=(20,20))
cmap = sns.diverging_palette(220, 20, sep=20, as_cmap=True)
sns.heatmap(df_bl.corr(), annot=True,cmap=cmap, center=0).set_title("Correlation Heatmap", fontsize=14)
plt.show()
```



3.7 One-hot Encoding

One-hot encode categorical features.

In [81]:

```
df = df.replace([True, False], [1,0])

cat = ['Side','Timezone','Weekday']
df[cat] = df[cat].astype('category')
df = pd.get_dummies(df, columns=cat, drop_first=True)
```

```
df_int = df.select_dtypes(include=['int']).apply(pd.to_numeric,downcast='unsi
gned')
df_float = df.select_dtypes(include=['float']).apply(pd.to_numeric,downcast='
float')
df = pd.concat([df.select_dtypes(include=['uint8']),df_int,df_float],axis=1)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 966307 entries, 0 to 36464
Data columns (total 28 columns):
 #   Column                        Non-Null Count   Dtype  
---  ------                        --------------   -----  
 0   Side_R                        966307 non-null  uint8  
 1   Timezone_US/Eastern           966307 non-null  uint8  
 2   Timezone_US/Mountain          966307 non-null  uint8  
 3   Timezone_US/Pacific           966307 non-null  uint8  
 4   Weekday_1                     966307 non-null  uint8  
 5   Weekday_2                     966307 non-null  uint8  
 6   Weekday_3                     966307 non-null  uint8  
 7   Weekday_4                     966307 non-null  uint8  
 8   Weekday_5                     966307 non-null  uint8  
 9   Weekday_6                     966307 non-null  uint8  
 10  Station                       966307 non-null  uint8  
 11  Stop                          966307 non-null  uint8  
 12  Traffic_Signal                966307 non-null  uint8  
 13  Severity4                     966307 non-null  uint8  
 14  Rd                            966307 non-null  uint8  
 15  St                            966307 non-null  uint8  
 16  Dr                            966307 non-null  uint8  
 17  Ave                           966307 non-null  uint8  
 18  Blvd                          966307 non-null  uint8  
 19  I-                            966307 non-null  uint8  
 20  Astronomical_Twilight_Night   966307 non-null  uint8  
 21  Start_Lat                     966307 non-null  float32
 22  Start_Lng                     966307 non-null  float32
 23  Minute_Freq                   966307 non-null  float32
 24  Population_County_log         966307 non-null  float32
 25  Street_Freq                   966307 non-null  float32
 26  State_Freq                    966307 non-null  float32
 27  Pressure_bc                   966307 non-null  float32
dtypes: float32(7), uint8(21)
memory usage: 92.5 MB
```

4 Model

Imbalance ratio of this dataset is about 100, which is the key problem we need to deal with. There are several ways to handle it:

1. **under-sampling** (I didn't use over-sampling because this dataset is large enough and over-sampling is very likely to casue overfitting)
2. **modify the loss function**
3. **ensemble methods**
    - EasyEnsemble
    - BalanceCascade

References:

X. Y. Liu, J. Wu and Z. H. Zhou, "Exploratory Undersampling forClass-Imbalance Learning," in IEEE Transactions on Systems, Man, andCybernetics, Part B (Cybernetics), vol. 39, no. 2, pp. 539-550,April 2009.

Ajinkya More | *Resampling techniques and other strategies*

In [82]:

```
from sklearn.model_selection import GridSearchCV, KFold, train_test_split, cr
oss_val_predict
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import AdaBoostClassifier, RandomForestClassifier
from imblearn.under_sampling import RandomUnderSampler
from collections import Counter
```

```
Using TensorFlow backend.
```

4.1 Train Test Split

In [83]:

```
# split X, y
X = df.drop('Severity4', axis=1)
y= df['Severity4']

# split train, test
X_train, X_test, y_train, y_test = train_test_split(\
  X, y, test_size=0.30, random_state=42)
```

4.2 Logistic regression with balanced class weights

under-sampling + modify the loss function

In [84]:

```
# Randomly undersample majority class to about 10 times of minority class
rus = RandomUnderSampler(sampling_strategy = 0.1, random_state=42)
X_train_res, y_train_res = rus.fit_sample(X_train, y_train)
```

```
print ("Distribution of class labels before resampling {}".format(Counter(y_t
rain)))
print ("Distribution of class labels after resampling {}".format(Counter(y_tr
ain_res)))
```

```
Distribution of class labels before resampling Counter({0: 671001, 1: 5413
})
Distribution of class labels after resampling Counter({0: 54130, 1: 5413})
```

In [85]:

```
clf_base = LogisticRegression()
grid = {'C': 10.0 ** np.arange(-2, 3),
        'penalty': ['l1', 'l2'],
        'class_weight': ['balanced']}
clf_lr = GridSearchCV(clf_base, grid, cv=5, n_jobs=8, scoring='f1_macro')

clf_lr.fit(X_train_res, y_train_res)

coef = clf_lr.best_estimator_.coef_
intercept = clf_lr.best_estimator_.intercept_
print (classification_report(y_test, clf_lr.predict(X_test)))
```

```
/opt/conda/lib/python3.7/site-packages/sklearn/metrics/_classification.py:
1272: UndefinedMetricWarning: Precision and F-score are ill-defined and be
ing set to 0.0 in labels with no predicted samples. Use `zero_division` pa
rameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
              precision    recall  f1-score   support

           0       0.99      1.00      1.00    287610
           1       0.00      0.00      0.00      2283

    accuracy                           0.99    289893
   macro avg       0.50      0.50      0.50    289893
weighted avg       0.98      0.99      0.99    289893
```

4.3 Random Forest

under-sampling

In [86]:

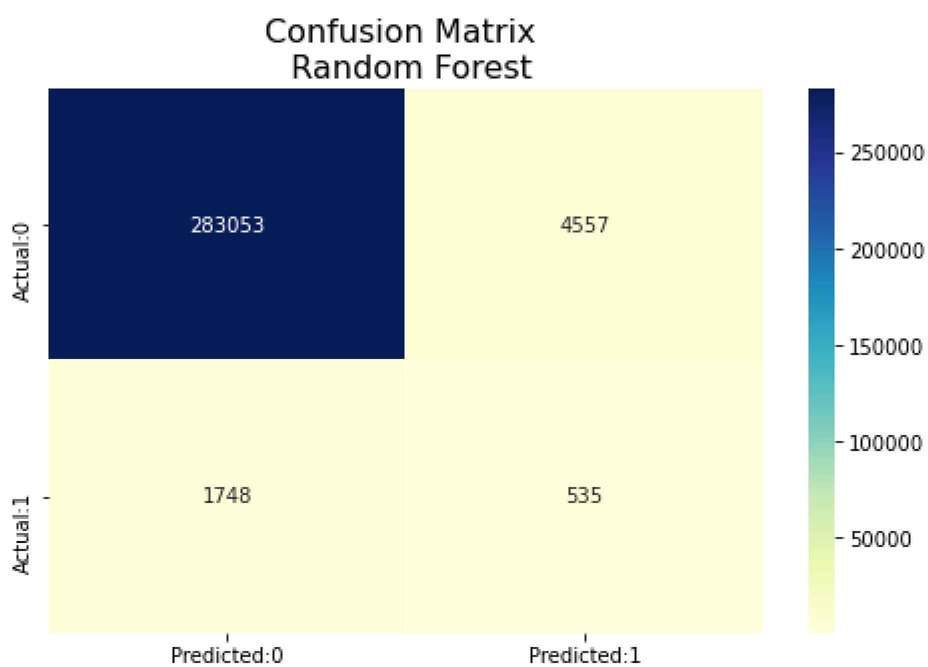
```
clf_base = RandomForestClassifier()
grid = {'n_estimators': [10, 50, 100],
        'max_features': ['auto','sqrt']}
clf_rf = GridSearchCV(clf_base, grid, cv=5, n_jobs=8, scoring='f1_macro')

clf_rf.fit(X_train_res, y_train_res)
y_pred = clf_rf.predict(X_test)
```

```
print (classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.99      0.98      0.99    287610
           1       0.11      0.23      0.15      2283

    accuracy                           0.98    289893
   macro avg       0.55      0.61      0.57    289893
weighted avg       0.99      0.98      0.98    289893
```

In [87]:

```
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)

conf_matrix = pd.DataFrame(data=confmat,
                           columns=['Predicted:0','Predicted:1'],index=['Actu
al:0','Actual:1'])
plt.figure(figsize = (8,5))
sns.heatmap(conf_matrix, annot=True,fmt='d',cmap="YlGnBu").set_title(
    "Confusion Matrix \n Random Forest", fontsize=16)
plt.show()
```



Try a different ratio.

In [96]:

```
# Randomly undersample majority class to about 20 times of minority class
rus = RandomUnderSampler(sampling_strategy = 0.05, random_state=42)
X_train_res, y_train_res = rus.fit_sample(X_train, y_train)
```

```
print ("Distribution of class labels before resampling {}".format(Counter(y_t
rain)))
print ("Distribution of class labels after resampling {}".format(Counter(y_tr
ain_res)))
```

```
Distribution of class labels before resampling Counter({0: 671001, 1: 5413
})
Distribution of class labels after resampling Counter({0: 108260, 1: 5413}
)
```

In [97]:

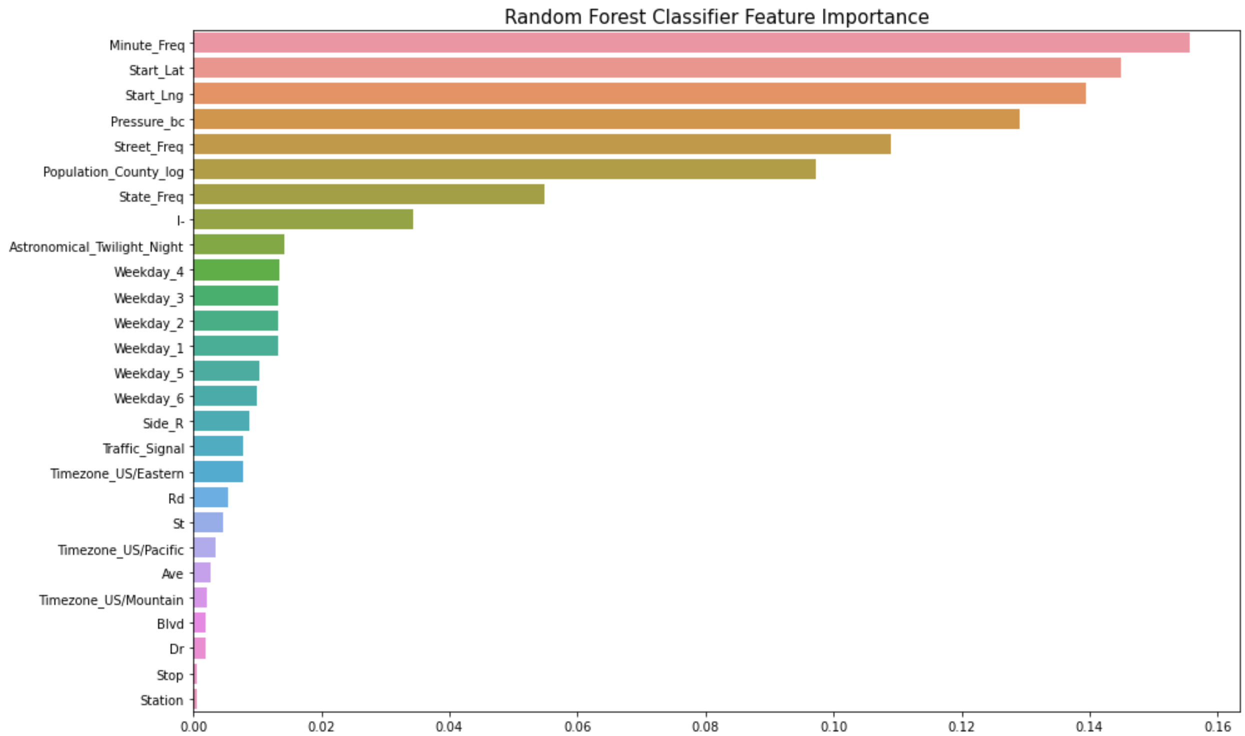
```
clf_base = RandomForestClassifier()
grid = {'n_estimators': [10, 50, 100],
        'max_features': ['auto','sqrt']}
clf_rf = GridSearchCV(clf_base, grid, cv=5, n_jobs=8, scoring='f1_macro')

clf_rf.fit(X_train_res, y_train_res)
y_pred = clf_rf.predict(X_test)

print (classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.99      1.00      0.99    287610
           1       0.16      0.12      0.14      2283

    accuracy                           0.99    289893
   macro avg       0.58      0.56      0.57    289893
weighted avg       0.99      0.99      0.99    289893
```

More data doesn't lead to better result.

In [88]:

```
importances = pd.DataFrame(np.zeros((X_train_res.shape[1], 1)), columns=['imp
ortance'], index=df.drop('Severity4',axis=1).columns)

importances.iloc[:,0] = clf_rf.best_estimator_.feature_importances_

importances.sort_values(by='importance', inplace=True, ascending=False)
importances30 = importances.head(30)

plt.figure(figsize=(15, 10))
sns.barplot(x='importance', y=importances30.index, data=importances30)

plt.xlabel('')
plt.tick_params(axis='x', labelsize=10)
plt.tick_params(axis='y', labelsize=10)
plt.title('Random Forest Classifier Feature Importance', size=15)

plt.show()
```



The feature importance plot shows that high-resolution spatio-temporal patterns of accidents are the most useful features to predict severity. Apart from that, pressure, population, road type are also critical.

4.4 EASYENSEMBLE

In [89]:

```
# n folds random under-sampling
def multi_rus(X, y, n_folds, ratio):
    X_res = [None] * n_folds
    y_res = [None] * n_folds
    rus = RandomUnderSampler(sampling_strategy = ratio, random_state=42)
    for i in range(n_folds):
        X_res[i], y_res[i] = rus.fit_sample(X, y)

    return X_res, y_res
```

In [90]:

```
X_train_res, y_train_res = multi_rus(X_train, y_train, 3, 0.1)
y_pred_proba = np.zeros(len(y_test))
for i in range(len(y_train_res)):
    clf = RandomForestClassifier(n_estimators=100, max_features='auto')
    clf.fit(X_train_res[i], y_train_res[i])
    y_pred_proba += clf.predict(X_test)

y_pred_proba = y_pred_proba/len(y_train_res)
y_pred = (y_pred_proba > 0.5).astype(int)
print (classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.99      0.98      0.99    287610
           1       0.11      0.24      0.15      2283

    accuracy                           0.98    289893
   macro avg       0.55      0.61      0.57    289893
weighted avg       0.99      0.98      0.98    289893
```

In [92]:

```
X_train_res, y_train_res = multi_rus(X_train, y_train, 9, 0.2)
y_pred_proba = np.zeros(len(y_test))
for i in range(len(y_train_res)):
    clf = RandomForestClassifier(n_estimators=100, max_features='auto')
    clf.fit(X_train_res[i], y_train_res[i])
    y_pred_proba += clf.predict(X_test)

y_pred_proba = y_pred_proba/len(y_train_res)
y_pred = (y_pred_proba > 0.5).astype(int)
print (classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.96      0.97    287610
           1       0.07      0.40      0.11      2283

    accuracy                           0.95    289893
   macro avg       0.53      0.68      0.54    289893
weighted avg       0.99      0.95      0.97    289893
```

In [94]:

```
X_train_res, y_train_res = multi_rus(X_train, y_train, 3, 0.1)
y_pred_proba = np.zeros(len(y_test))
for i in range(len(y_train_res)):
    clf_base = AdaBoostClassifier()
    grid = {'n_estimators': [10, 50, 100]}

    clf = GridSearchCV(clf_base, grid, cv=3, n_jobs=8, scoring='f1_macro')
    clf.fit(X_train_res[i], y_train_res[i])
    y_pred_proba += clf.predict(X_test)

y_pred_proba = y_pred_proba/len(y_train_res)
y_pred = (y_pred_proba > 0.5).astype(int)
print (classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99    287610
```

```
           1       0.10      0.15      0.12      2283

    accuracy                           0.98    289893
   macro avg       0.55      0.57      0.55    289893
weighted avg       0.99      0.98      0.98    289893
```

EasyEnsemble didn't improve the result very much.

4.5 BalanceCascade

```python
def BalanceCascadeSample(X,
                         y,
                         estimator=AdaBoostClassifier(),
                         random_state = 42,
                         n_max_subset = 10
                        ):
    """Resample the dataset.

    Parameters
    ----------
    estimator : object, optional (default=AdaBoostClassifier())
        An estimator inherited from :class:`sklearn.base.ClassifierMixin` and
        having an attribute :func:`predict_proba`.

    X : ndarray, shape (n_samples, n_features)
        Matrix containing the data which have to be sampled.

    y : ndarray, shape (n_samples, )
        Corresponding label for each sample in X.

    random_state : int, RandomState instance or None, optional (default=42)
        If int, ``random_state`` is the seed used by the random number
        generator; If ``RandomState`` instance, random_state is the random
        number generator; If ``None``, the random number generator is the
        ``RandomState`` instance used by ``np.random``.

    n_max_subset : int or None, optional (default=10)
        Maximum number of subsets to generate. By default, all data from
        the training will be selected that could lead to a large number of
        subsets. We can probably deduce this number empirically.

    Returns
    -------
    X_resampled : ndarray, shape (n_subset, n_samples_new, n_features)
        The array containing the resampled data.

    y_resampled : ndarray, shape (n_subset, n_samples_new)
```

```
        The corresponding label of `X_resampled`

    idx_under : ndarray, shape (n_subset, n_samples, )
        If `return_indices` is `True`, a boolean array will be returned
        containing the which samples have been selected.

    """
    # array to know which samples are available to be taken
    samples_mask = np.ones(y.shape, dtype=bool)

    # where the different set will be stored
    X_resampled = []
    y_resampled = []
    idx_under = []

    n_subsets = 0
    b_subset_search = True
    while b_subset_search:
        target_stats = Counter(y[samples_mask])
        # build the data set to be classified
        X_subset = np.empty((0, X.shape[1]), dtype=X.dtype)
        y_subset = np.empty((0, ), dtype=y.dtype)
        # store the index of the data to under-sample
        index_under_sample = np.empty((0, ), dtype=y.dtype)
        # value which will be picked at each round
        X_constant = np.empty((0, X.shape[1]), dtype=X.dtype)
        y_constant = np.empty((0, ), dtype=y.dtype)
        index_constant = np.empty((0, ), dtype=y.dtype)
        for target_class in target_stats.keys():
            X_constant = np.concatenate((X_constant,
                                          X[y == target_class]),
                                         axis=0)
            y_constant = np.concatenate((y_constant,
                                         y[y == target_class]),
                                         axis=0)
            index_constant = np.concatenate(
                (index_constant,
                 np.flatnonzero(y == target_class)),
                axis=0)

        # store the set created
        n_subsets += 1
        X_resampled.append(np.concatenate((X_subset, X_constant),
                                           axis=0))
        y_resampled.append(np.concatenate((y_subset, y_constant),
                                           axis=0))
        idx_under.append(np.concatenate((index_under_sample,
                                          index_constant),
                                         axis=0))
```

```
        # fit and predict using cross validation
        pred = cross_val_predict(estimator,
                                 np.concatenate((X_subset, X_constant),
                                                axis=0),
                                 np.concatenate((y_subset, y_constant),
                                                axis=0))
        # extract the prediction about the targeted classes only
        pred_target = pred[:y_subset.size]
        index_classified = index_under_sample[pred_target == y_subset]
        samples_mask[index_classified] = False

        # check the stopping criterion
        if n_subsets == n_max_subset:
            b_subset_search = False

    return np.array(X_resampled), np.array(y_resampled)
```

In [102]:

```
rus = RandomUnderSampler(sampling_strategy = 0.1, random_state=42)
X_train_res, y_train_res = rus.fit_sample(X_train, y_train)
X_train_res, y_train_res = BalanceCascadeSample(X = X_train_res.to_numpy(),
                                                y = y_train_res.to_numpy(),
                                                estimator=RandomForestClassif
ier(n_estimators=100, max_features='auto'),
                                                n_max_subset = 5)
```

In [103]:

```
y_pred_proba = np.zeros(len(y_test))
for i in range(len(y_train_res)):
    clf = RandomForestClassifier(n_estimators=100, max_features='auto')
    clf.fit(X_train_res[i], y_train_res[i])
    y_pred_proba += clf.predict(X_test)

y_pred_proba = y_pred_proba/len(y_train_res)
y_pred = (y_pred_proba > 0.5).astype(int)
print (classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.99      0.98      0.99    287610
           1       0.11      0.24      0.15      2283

    accuracy                           0.98    289893
   macro avg       0.55      0.61      0.57    289893
weighted avg       0.99      0.98      0.98    289893
```

Similar result as EasyEnsemble.

In [ ]:

```
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)

conf_matrix = pd.DataFrame(data=confmat,
                           columns=['Predicted:0','Predicted:1'],index=['Actu
al:0','Actual:1'])
plt.figure(figsize = (8,5))
sns.heatmap(conf_matrix, annot=True,fmt='d',cmap="YlGnBu").set_title(
    "Confusion Matrix \n Random Forest", fontsize=16)
plt.show()
```

5 Future Work

1. Find a better way to handle class imbalance.
2. Incorporate this model in a real-time accident risk prediction model or develop a new real-time severe accident risk prediction on grid cells.
3. Detailed relations between some key factors and accident severity can be further studied.
4. Policy implications of this project can be explored.