In [1]:
```
# Predict asteroid diameter values using 'Asteroid.csv' dataset from Kaggle co
ntributed by Victor Basu
    # link: https://www.kaggle.com/basu369victor/prediction-of-asteroid-diamet
er
# Model: XGBRegressor
# Notes on data:
    # data is medium size comprising of 839736 entries and 27 columns
    # for a small portion of the data (~ 1/6) the asteroids diameters are know
n -
        # this portion will be used to train and validate the model
    # subsequently the model will be used to predict the diameters for the dat
a in which this information is missing

# Essential updates (6/2020) from previous project version (2/2020)
    # Improvements in data processing and data visualization
    # Comparison between XGBRegressor model and Linear Regression model is dis
carded -->
        # XGBRegressor model optimization via hyperparameter tuning is added i
nstead
    # Statistics of residuals - distribution, mean and standard deviation - re
place absolute error statistics
    # as model performance metrics
```

In [2]:
```
# Import libraries

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.set(style = 'whitegrid', font_scale = 1.5)
```

In [3]:
```
# Ignore warnings

import warnings
warnings.filterwarnings('ignore')
```

In [4]:
```
# Read data

data = pd.read_csv('Asteroid.csv', low_memory = False)
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 839736 entries, 0 to 839735
Data columns (total 27 columns):
 #   Column          Non-Null Count   Dtype  
---  ------          --------------   -----  
 0   full_name       839736 non-null  object 
 1   a               839734 non-null  float64
 2   e               839736 non-null  float64
 3   G               119 non-null     float64
 4   i               839736 non-null  float64
 5   om              839736 non-null  float64
 6   w               839736 non-null  float64
 7   q               839736 non-null  float64
 8   ad              839730 non-null  float64
 9   per_y           839735 non-null  float64
 10  data_arc        823947 non-null  float64
 11  condition_code  838743 non-null  object 
 12  n_obs_used      839736 non-null  int64  
 13  H               837042 non-null  float64
 14  diameter        137681 non-null  object 
 15  extent          18 non-null      object 
 16  albedo          136452 non-null  float64
 17  rot_per         18796 non-null   float64
 18  GM              14 non-null      float64
 19  BV              1021 non-null    float64
 20  UB              979 non-null     float64
 21  IR              1 non-null       float64
 22  spec_B          1666 non-null    object 
 23  spec_T          980 non-null     object 
 24  neo             839730 non-null  object 
 25  pha             822814 non-null  object 
 26  moid            822814 non-null  float64
dtypes: float64(18), int64(1), object(8)
memory usage: 173.0+ MB
```

In [5]:
```
# Print data column names for use in code below

data.columns
```

Out[5]:
```
Index(['full_name', 'a', 'e', 'G', 'i', 'om', 'w', 'q', 'ad', 'per_y',
       'data_arc', 'condition_code', 'n_obs_used', 'H', 'diameter', 'extent',
       'albedo', 'rot_per', 'GM', 'BV', 'UB', 'IR', 'spec_B', 'spec_T', 'ne
o',
       'pha', 'moid'],
      dtype='object')
```

In [6]:
```
# Select only features with meaningful amount of non-null values -->
    # drop 'G', 'extent', 'GM', 'BV', 'UB', 'IR', 'spec_B', and 'spec_T'
# In addition, drop 'full_name' and 'n_obs_used' which are not meaningful for
 the problem
# Place target 'diameter' at the end for easier separation of features, X, and
target, y, later on

data = data[['a', 'e', 'i', 'om', 'w', 'q', 'ad', 'per_y', 'data_arc', 'condit
ion_code',
             'H', 'albedo', 'neo', 'pha', 'moid', 'diameter']]
data.head(10)
```

Out[6]:

	a	e	i	om	w	q	ad	per_y	data_ar
0	2.769165	0.076009	10.594067	80.305532	73.597694	2.558684	2.979647	4.608202	8822.
1	2.772466	0.230337	34.836234	173.080063	310.048857	2.133865	3.411067	4.616444	72318.
2	2.669150	0.256942	12.988919	169.852760	248.138626	1.983332	3.354967	4.360814	72684.
3	2.361418	0.088721	7.141771	103.810804	150.728541	2.151909	2.570926	3.628837	24288.
4	2.574249	0.191095	5.366988	141.576604	358.687608	2.082324	3.066174	4.130323	63431.
5	2.425160	0.203007	14.737901	138.640203	239.807490	1.932835	2.917485	3.776755	62329.
6	2.385334	0.231206	5.523651	259.563231	145.265106	1.833831	2.936837	3.684105	62452.
7	2.201764	0.156499	5.886955	110.889330	285.287462	1.857190	2.546339	3.267115	62655.
8	2.385637	0.123114	5.576816	68.908577	6.417369	2.091931	2.679342	3.684806	61821.
9	3.141539	0.112461	3.831560	283.202167	312.315206	2.788240	3.494839	5.568291	62175.

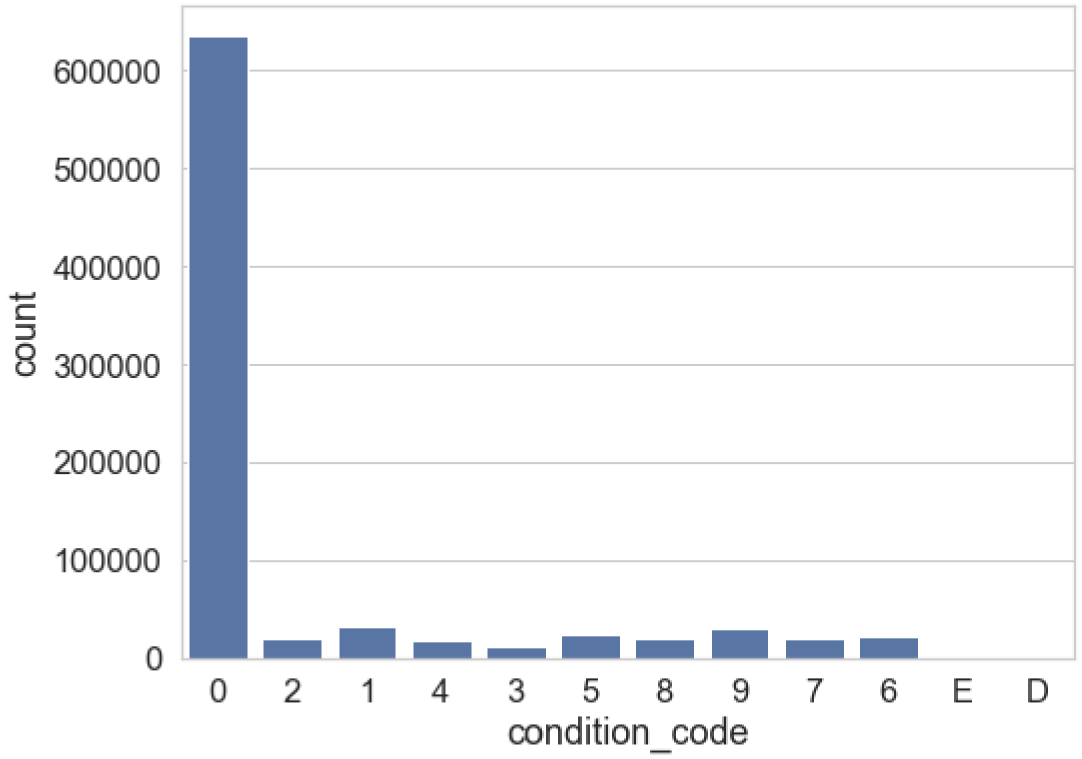
In [7]:
```
# 1) Data Processing and EDA
```

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 839736 entries, 0 to 839735
Data columns (total 16 columns):
 #   Column          Non-Null Count   Dtype  
---  ------          --------------   -----  
 0   a               839734 non-null  float64
 1   e               839736 non-null  float64
 2   i               839736 non-null  float64
 3   om              839736 non-null  float64
 4   w               839736 non-null  float64
 5   q               839736 non-null  float64
 6   ad              839730 non-null  float64
 7   per_y           839735 non-null  float64
 8   data_arc        823947 non-null  float64
 9   condition_code  838743 non-null  object 
 10  H               837042 non-null  float64
 11  albedo          136452 non-null  float64
 12  neo             839730 non-null  object 
 13  pha             822814 non-null  object 
 14  moid            822814 non-null  float64
 15  diameter        137681 non-null  object 
dtypes: float64(12), object(4)
memory usage: 102.5+ MB
```

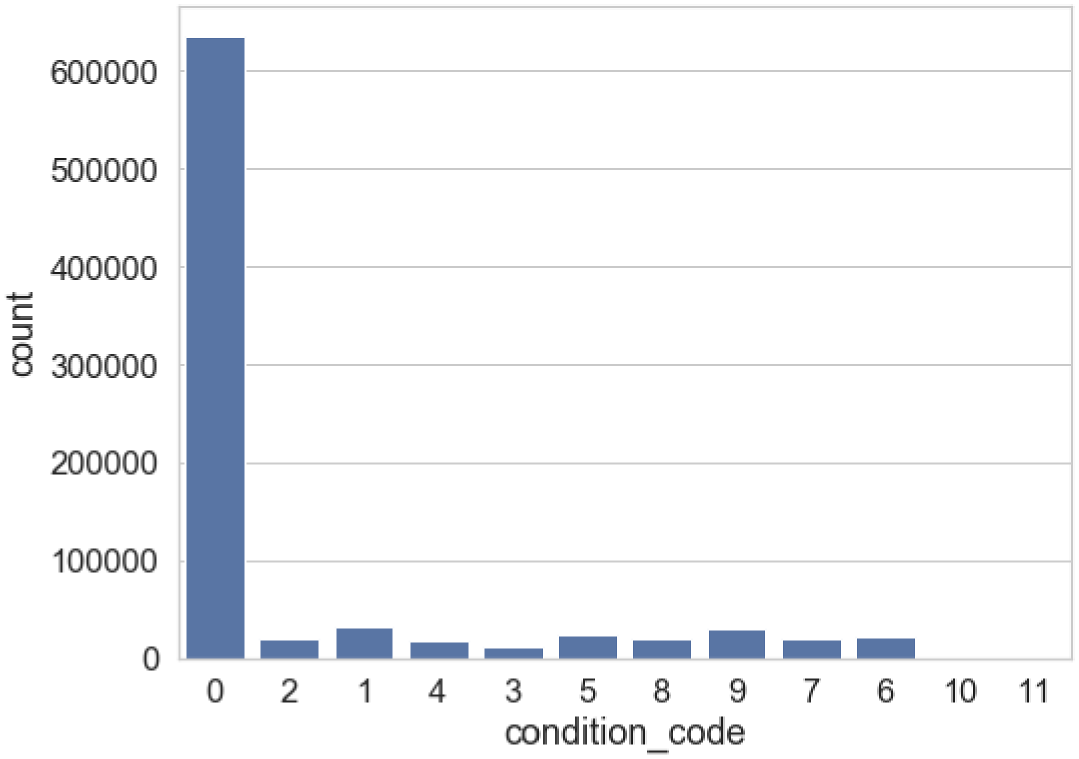
In [9]:
```
# Features 'condition_code', 'neo', and 'pha' appear to be categorical --> exa
mine these features
```

In [10]:
```
# Examine 'condition_code'

plt.figure(figsize = (8, 6))
sns.countplot(data['condition_code'], color = 'b')
plt.show()
```

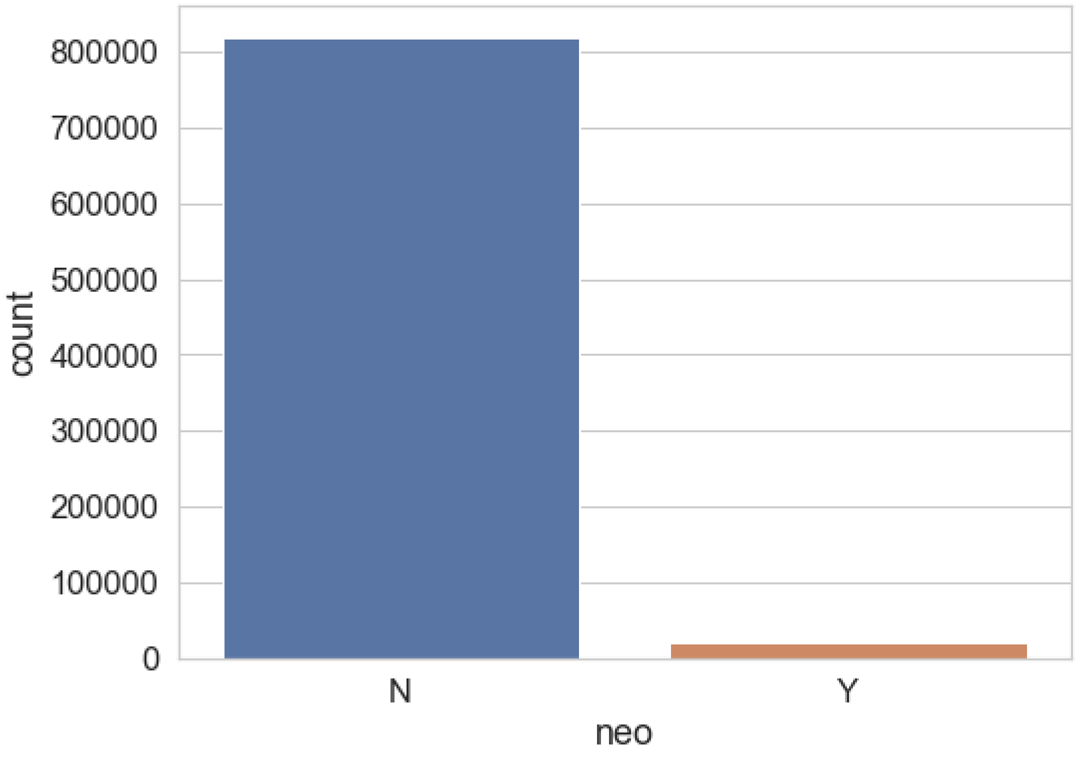


In [11]:
```
# This is a categorical feature with majority of data points having values = 0
# Note that 'condition_code' values includes both numbers and letters
```

In [12]:
```
# Assign numeric values to categorical values 'E' and 'D'

data['condition_code'].replace({'E': 10, 'D': 11}, inplace=True)

plt.figure(figsize = (8, 6))
sns.countplot(data['condition_code'], color = 'b')
plt.show()
```

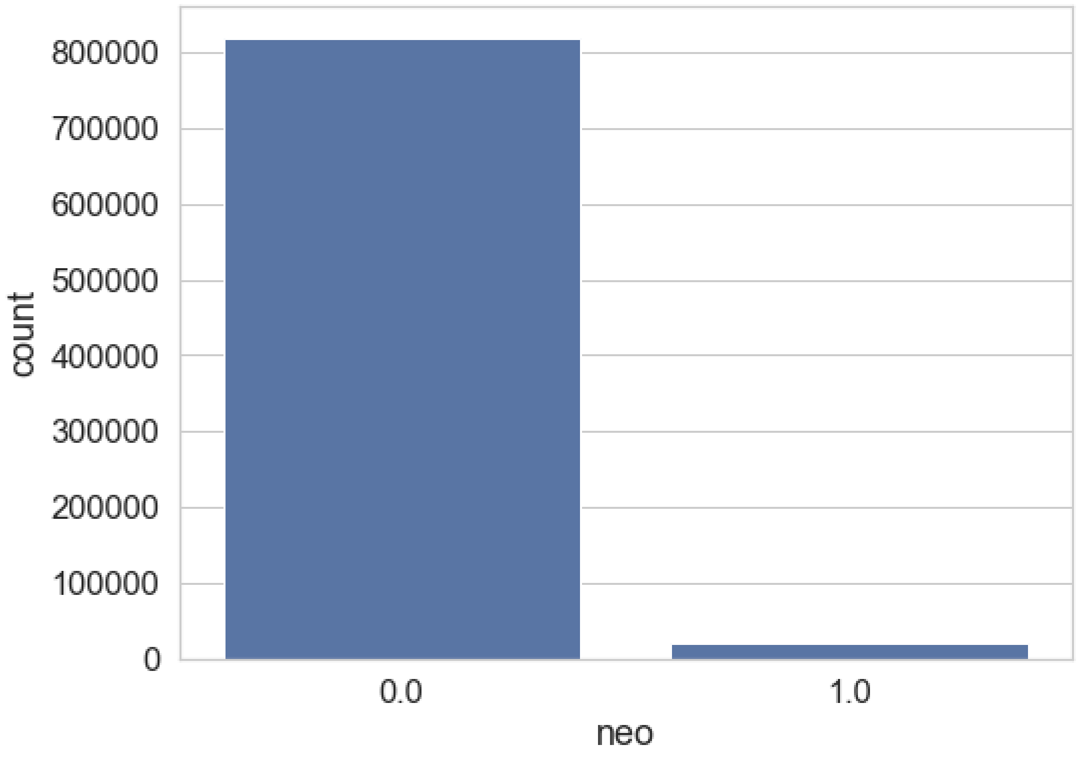


In [13]:
```
# Examine 'neo'

plt.figure(figsize = (8, 6))
sns.countplot(data['neo'])
plt.show()
```

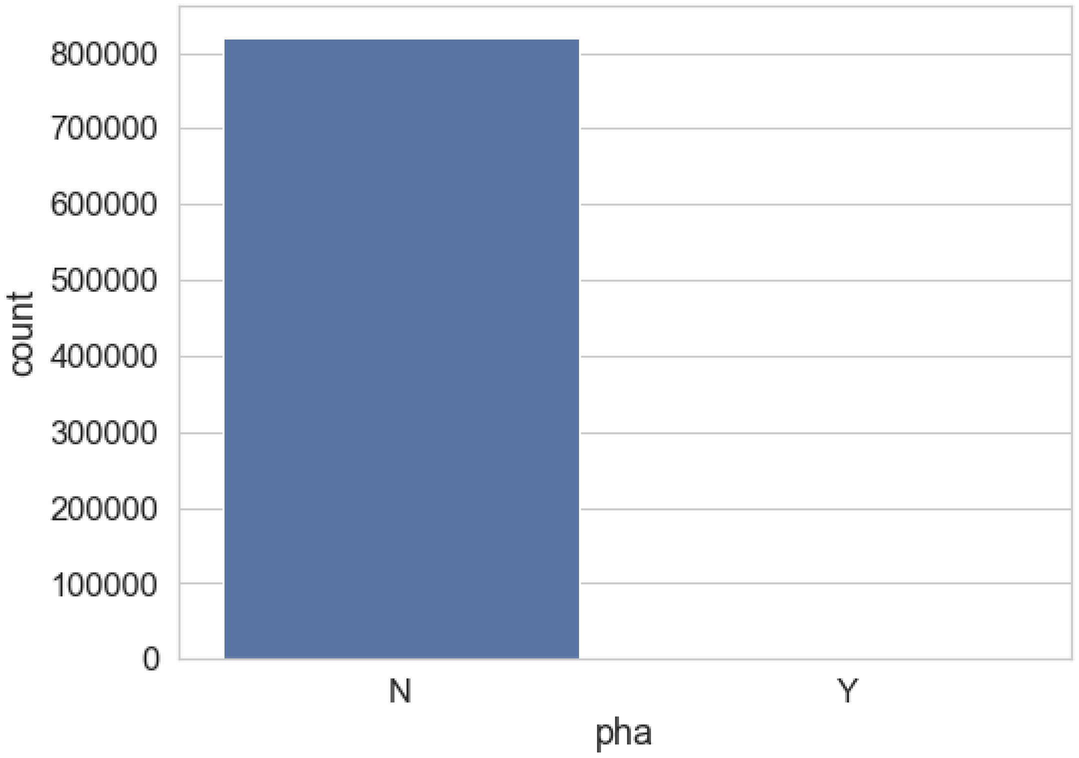


In [14]:
```
# Categorical feature --> majority of data points = N
```

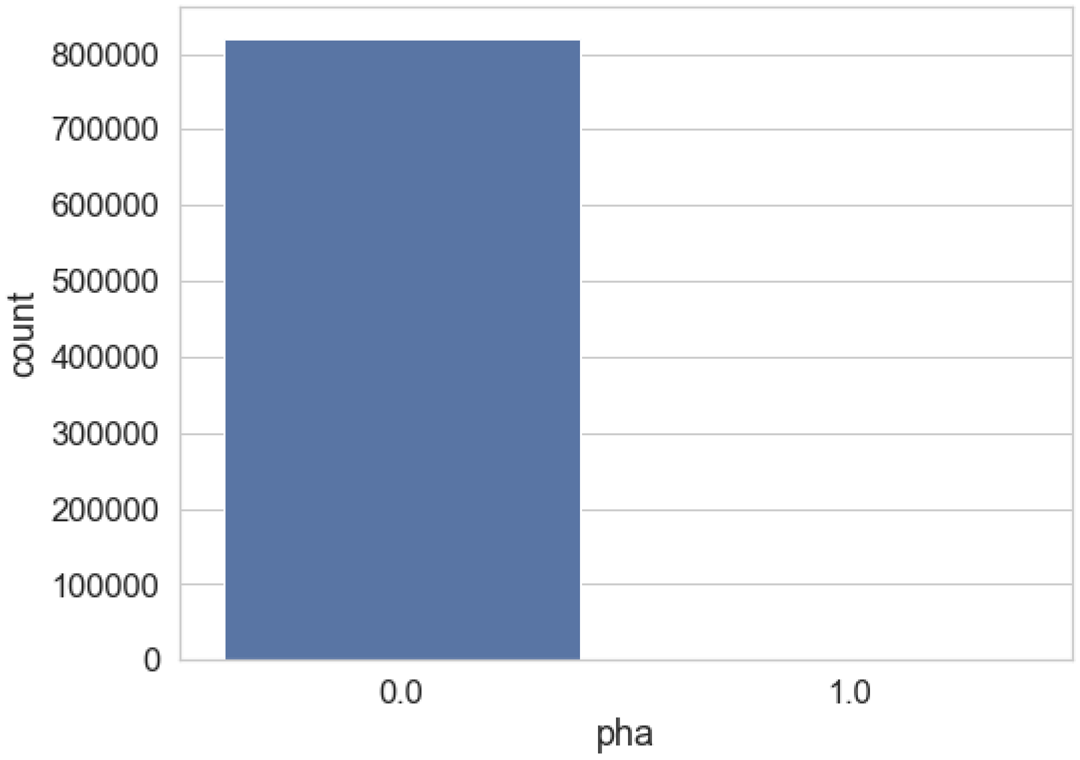
In [15]:
```
# Replace categorical values, 'N' and 'Y', with numerical values of 0 and 1, r
espectively

data['neo'].replace({'N': 0, 'Y': 1}, inplace=True)

plt.figure(figsize = (8, 6))
sns.countplot(data['neo'], color = 'b')
plt.show()
```



In [16]:
```
# Examine 'pha'

plt.figure(figsize = (8, 6))
sns.countplot(data['pha'])
plt.show()
```



In [17]:
```
# Categorical feature --> majority of data points = N

# Replace categorical values, 'N' and 'Y', with numerical values of 0 and 1, r
espectively

data['pha'].replace({'N': 0, 'Y': 1}, inplace=True)

plt.figure(figsize = (8, 6))
sns.countplot(data['pha'], color = 'b')
plt.show()
```



In [18]:
```
# Examine target, 'diameter'

data.head(10)
```

Out[18]:

	a	e	i	om	w	q	ad	per_y	data_ar
0	2.769165	0.076009	10.594067	80.305532	73.597694	2.558684	2.979647	4.608202	8822.
1	2.772466	0.230337	34.836234	173.080063	310.048857	2.133865	3.411067	4.616444	72318.
2	2.669150	0.256942	12.988919	169.852760	248.138626	1.983332	3.354967	4.360814	72684.
3	2.361418	0.088721	7.141771	103.810804	150.728541	2.151909	2.570926	3.628837	24288.
4	2.574249	0.191095	5.366988	141.576604	358.687608	2.082324	3.066174	4.130323	63431.
5	2.425160	0.203007	14.737901	138.640203	239.807490	1.932835	2.917485	3.776755	62329.
6	2.385334	0.231206	5.523651	259.563231	145.265106	1.833831	2.936837	3.684105	62452.
7	2.201764	0.156499	5.886955	110.889330	285.287462	1.857190	2.546339	3.267115	62655.
8	2.385637	0.123114	5.576816	68.908577	6.417369	2.091931	2.679342	3.684806	61821.
9	3.141539	0.112461	3.831560	283.202167	312.315206	2.788240	3.494839	5.568291	62175.

In [19]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 839736 entries, 0 to 839735
Data columns (total 16 columns):
 #   Column          Non-Null Count   Dtype  
---  ------          --------------   -----  
 0   a               839734 non-null  float64
 1   e               839736 non-null  float64
 2   i               839736 non-null  float64
 3   om              839736 non-null  float64
 4   w               839736 non-null  float64
 5   q               839736 non-null  float64
 6   ad              839730 non-null  float64
 7   per_y           839735 non-null  float64
 8   data_arc        823947 non-null  float64
 9   condition_code  838743 non-null  object 
 10  H               837042 non-null  float64
 11  albedo          136452 non-null  float64
 12  neo             839730 non-null  float64
 13  pha             822814 non-null  float64
 14  moid            822814 non-null  float64
 15  diameter        137681 non-null  object 
dtypes: float64(14), object(2)
memory usage: 102.5+ MB
```

In [20]:
```
# Columns 'diameter' and 'albedo' have only about 1/6 of non-null values compa
red to other features
# Although 'diameter' has numerical values in the table, it appears that it is
in string format - data type 'object'
# Convert data to numeric format 'float64'

data = data.astype('float64')

data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 839736 entries, 0 to 839735
Data columns (total 16 columns):
 #   Column          Non-Null Count   Dtype
---  ------          --------------   -----
 0   a               839734 non-null  float64
 1   e               839736 non-null  float64
 2   i               839736 non-null  float64
 3   om              839736 non-null  float64
 4   w               839736 non-null  float64
 5   q               839736 non-null  float64
 6   ad              839730 non-null  float64
 7   per_y           839735 non-null  float64
 8   data_arc        823947 non-null  float64
 9   condition_code  838743 non-null  float64
 10  H               837042 non-null  float64
 11  albedo          136452 non-null  float64
 12  neo             839730 non-null  float64
 13  pha             822814 non-null  float64
 14  moid            822814 non-null  float64
 15  diameter        137681 non-null  float64
dtypes: float64(16)
memory usage: 102.5 MB
```

In [21]:
```
# Replace all missing values with 0 which is the sparse value expected by XGBo
ost

data.fillna(0, inplace = True)

data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 839736 entries, 0 to 839735
Data columns (total 16 columns):
 #   Column          Non-Null Count   Dtype  
---  ------          --------------   -----  
 0   a               839736 non-null  float64
 1   e               839736 non-null  float64
 2   i               839736 non-null  float64
 3   om              839736 non-null  float64
 4   w               839736 non-null  float64
 5   q               839736 non-null  float64
 6   ad              839736 non-null  float64
 7   per_y           839736 non-null  float64
 8   data_arc        839736 non-null  float64
 9   condition_code  839736 non-null  float64
 10  H               839736 non-null  float64
 11  albedo          839736 non-null  float64
 12  neo             839736 non-null  float64
 13  pha             839736 non-null  float64
 14  moid            839736 non-null  float64
 15  diameter        839736 non-null  float64
dtypes: float64(16)
memory usage: 102.5 MB
```

In [22]:
```
# Create dataset, data_1, where diameter is known

data_1 = data[data['diameter'] > 0] # values greater than 0 correspond to data
with known diameter
data_1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 137681 entries, 0 to 810411
Data columns (total 16 columns):
 #   Column          Non-Null Count   Dtype  
---  ------          --------------   -----  
 0   a               137681 non-null  float64
 1   e               137681 non-null  float64
 2   i               137681 non-null  float64
 3   om              137681 non-null  float64
 4   w               137681 non-null  float64
 5   q               137681 non-null  float64
 6   ad              137681 non-null  float64
 7   per_y           137681 non-null  float64
 8   data_arc        137681 non-null  float64
 9   condition_code  137681 non-null  float64
 10  H               137681 non-null  float64
 11  albedo          137681 non-null  float64
 12  neo             137681 non-null  float64
 13  pha             137681 non-null  float64
 14  moid            137681 non-null  float64
 15  diameter        137681 non-null  float64
dtypes: float64(16)
memory usage: 17.9 MB
```

In [23]:
```
# Data with known asteroid diameter have total of 137681 entries
```

In [24]:
```
# Check data_1

data_1.head(10)
```

Out[24]:

	a	e	i	om	w	q	ad	per_y	data_ar
0	2.769165	0.076009	10.594067	80.305532	73.597694	2.558684	2.979647	4.608202	8822.
1	2.772466	0.230337	34.836234	173.080063	310.048857	2.133865	3.411067	4.616444	72318.
2	2.669150	0.256942	12.988919	169.852760	248.138626	1.983332	3.354967	4.360814	72684.
3	2.361418	0.088721	7.141771	103.810804	150.728541	2.151909	2.570926	3.628837	24288.
4	2.574249	0.191095	5.366988	141.576604	358.687608	2.082324	3.066174	4.130323	63431.
5	2.425160	0.203007	14.737901	138.640203	239.807490	1.932835	2.917485	3.776755	62329.
6	2.385334	0.231206	5.523651	259.563231	145.265106	1.833831	2.936837	3.684105	62452.
7	2.201764	0.156499	5.886955	110.889330	285.287462	1.857190	2.546339	3.267115	62655.
8	2.385637	0.123114	5.576816	68.908577	6.417369	2.091931	2.679342	3.684806	61821.
9	3.141539	0.112461	3.831560	283.202167	312.315206	2.788240	3.494839	5.568291	62175.

In [25]:
```
# Create dataset, data_2, where diameter is unknown

data_2 = data[data['diameter'] < data_1['diameter'].min()] # this leaves only
 0s which correspond to unknown diameter
data_2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 702055 entries, 681 to 839735
Data columns (total 16 columns):
 #   Column          Non-Null Count   Dtype  
---  ------          --------------   -----  
 0   a               702055 non-null  float64
 1   e               702055 non-null  float64
 2   i               702055 non-null  float64
 3   om              702055 non-null  float64
 4   w               702055 non-null  float64
 5   q               702055 non-null  float64
 6   ad              702055 non-null  float64
 7   per_y           702055 non-null  float64
 8   data_arc        702055 non-null  float64
 9   condition_code  702055 non-null  float64
 10  H               702055 non-null  float64
 11  albedo          702055 non-null  float64
 12  neo             702055 non-null  float64
 13  pha             702055 non-null  float64
 14  moid            702055 non-null  float64
 15  diameter        702055 non-null  float64
dtypes: float64(16)
memory usage: 91.1 MB
```

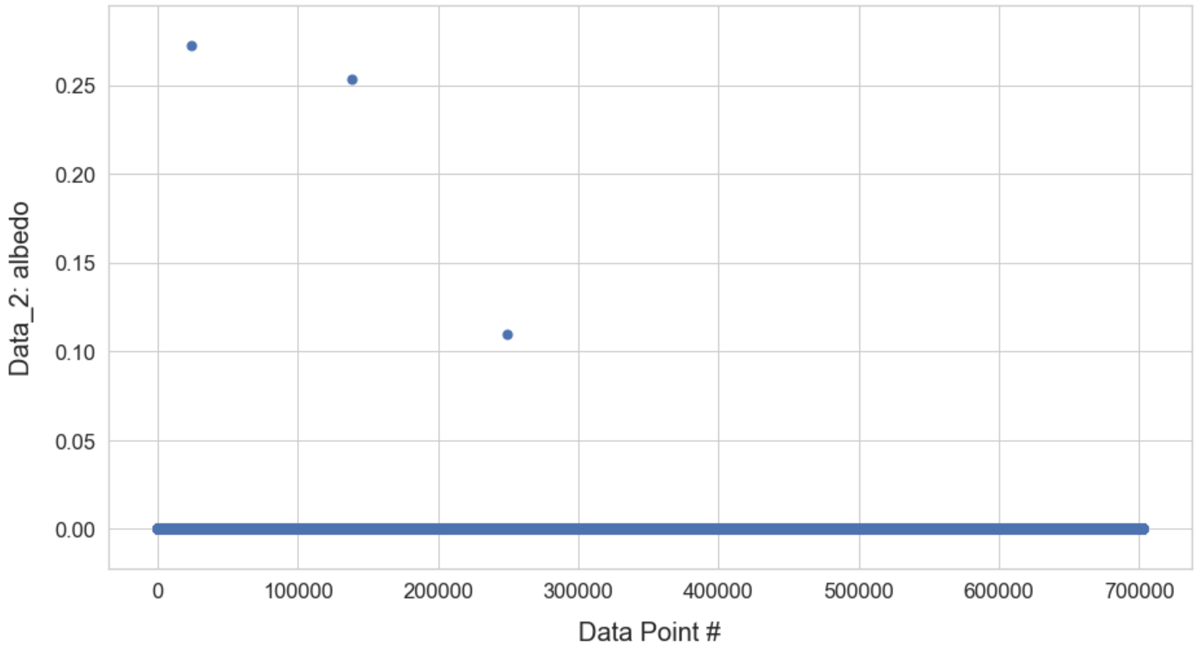
In [26]:
```
# Data with unknown asteroid diameter have total of 702055 entries (more than
 5 x that of data_1)
```

In [27]:
```
# Check data_2

data_2.head(10)
```

Out[27]:

	a	e	i	om	w	q	ad	per_y	data
681	2.654040	0.171983	11.505648	190.799958	104.993826	2.197591	3.110489	4.323837	40
698	2.610998	0.410284	15.299180	242.551766	91.399514	1.539746	3.682249	4.219081	42
718	2.638780	0.546301	11.564845	183.887287	156.163668	1.197212	4.080348	4.286601	39
729	2.243362	0.177505	4.234895	95.073806	123.549777	1.845154	2.641570	3.360139	39
842	2.279598	0.209766	7.997717	4.071363	316.957206	1.801415	2.757780	3.441878	37
961	2.908998	0.097329	2.602636	145.481660	223.473847	2.625868	3.192128	4.961619	37
984	2.299979	0.277462	4.056565	290.307048	59.553605	1.661822	2.938137	3.488142	35
1008	2.625175	0.455500	15.769676	229.461495	186.428747	1.429408	3.820942	4.253492	34
1010	2.391976	0.350864	5.494744	132.525452	353.279770	1.552718	3.231235	3.699504	34
1064	2.360276	0.297141	8.362855	330.324142	353.652287	1.658942	3.061610	3.626205	33

In [28]:
```
# It appears 'albedo' is also unknown in data_2 --> only 0s are shown in table

# Check by plotting data_2['albedo']

plt.figure(figsize = (15, 8))
plt.scatter(np.arange(1, len(data_2) + 1), data_2['albedo'], s = 50, c = 'b')
plt.xlabel('Data Point #', fontsize = 20, labelpad = 15)
plt.ylabel('Data_2: albedo', fontsize = 20, labelpad = 15)
plt.show()
```



In [29]:
```
# Indeed, almost all 'albedo' data points in data_2 are 0s and it cannot be us
ed in predictions -->
    # remove 'albedo' from both data_1 and data_2
```

In [30]:
```
data_1.columns
```

Out[30]:
```
Index(['a', 'e', 'i', 'om', 'w', 'q', 'ad', 'per_y', 'data_arc',
       'condition_code', 'H', 'albedo', 'neo', 'pha', 'moid', 'diameter'],
      dtype='object')
```

In [31]:
```
# Keep all features in data_1 except 'albedo'

data_1 = data_1[['a', 'e', 'i', 'om', 'w', 'q', 'ad', 'per_y', 'data_arc',
                 'condition_code', 'H', 'neo', 'pha', 'moid', 'diameter']]
data_1.head(10)
```

Out[31]:

	a	e	i	om	w	q	ad	per_y	data_ar
0	2.769165	0.076009	10.594067	80.305532	73.597694	2.558684	2.979647	4.608202	8822.
1	2.772466	0.230337	34.836234	173.080063	310.048857	2.133865	3.411067	4.616444	72318.
2	2.669150	0.256942	12.988919	169.852760	248.138626	1.983332	3.354967	4.360814	72684.
3	2.361418	0.088721	7.141771	103.810804	150.728541	2.151909	2.570926	3.628837	24288.
4	2.574249	0.191095	5.366988	141.576604	358.687608	2.082324	3.066174	4.130323	63431.
5	2.425160	0.203007	14.737901	138.640203	239.807490	1.932835	2.917485	3.776755	62329.
6	2.385334	0.231206	5.523651	259.563231	145.265106	1.833831	2.936837	3.684105	62452.
7	2.201764	0.156499	5.886955	110.889330	285.287462	1.857190	2.546339	3.267115	62655.
8	2.385637	0.123114	5.576816	68.908577	6.417369	2.091931	2.679342	3.684806	61821.
9	3.141539	0.112461	3.831560	283.202167	312.315206	2.788240	3.494839	5.568291	62175.

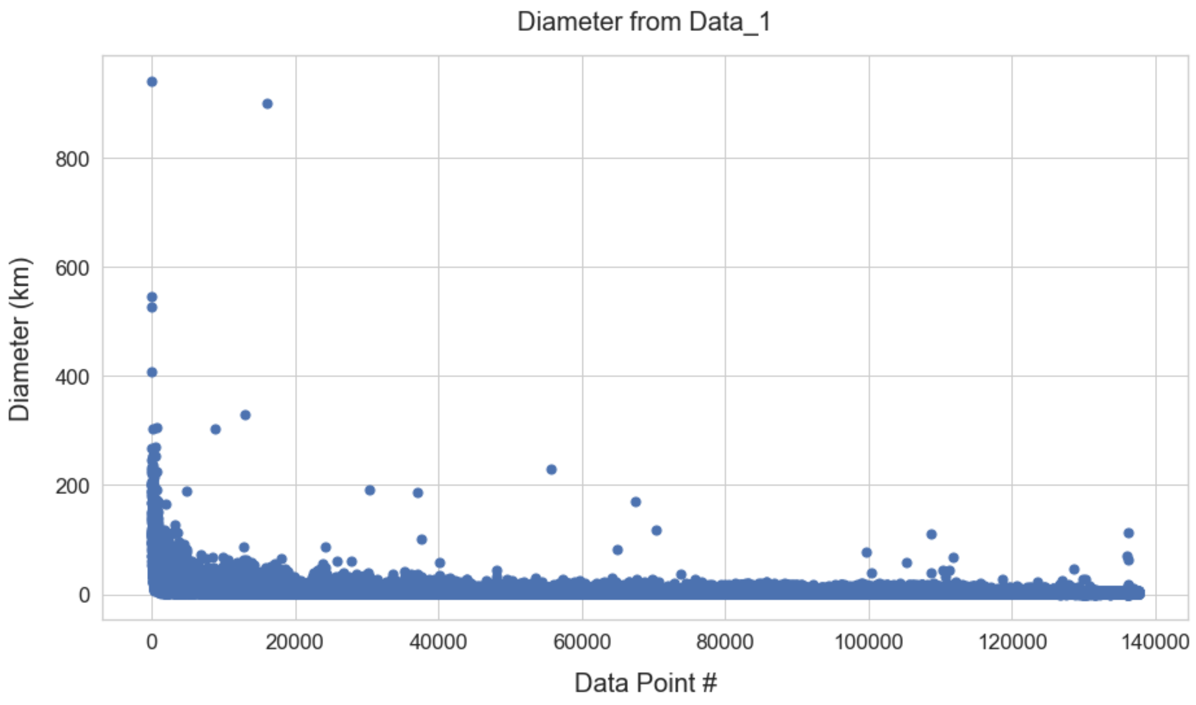
In [32]:
```
# Keep all features in data_2 except 'albedo' and 'diameter' which is unknown

data_2 = data_2[['a', 'e', 'i', 'om', 'w', 'q', 'ad', 'per_y', 'data_arc', 'co
ndition_code', 'H', 'neo', 'pha', 'moid']]

data_2.head(10)
```

Out[32]:

	a	e	i	om	w	q	ad	per_y	data
681	2.654040	0.171983	11.505648	190.799958	104.993826	2.197591	3.110489	4.323837	400
698	2.610998	0.410284	15.299180	242.551766	91.399514	1.539746	3.682249	4.219081	425
718	2.638780	0.546301	11.564845	183.887287	156.163668	1.197212	4.080348	4.286601	394
729	2.243362	0.177505	4.234895	95.073806	123.549777	1.845154	2.641570	3.360139	391
842	2.279598	0.209766	7.997717	4.071363	316.957206	1.801415	2.757780	3.441878	375
961	2.908998	0.097329	2.602636	145.481660	223.473847	2.625868	3.192128	4.961619	374
984	2.299979	0.277462	4.056565	290.307048	59.553605	1.661822	2.938137	3.488142	353
1008	2.625175	0.455500	15.769676	229.461495	186.428747	1.429408	3.820942	4.253492	349
1010	2.391976	0.350864	5.494744	132.525452	353.279770	1.552718	3.231235	3.699504	349
1064	2.360276	0.297141	8.362855	330.324142	353.652287	1.658942	3.061610	3.626205	338

In [33]:
```
# Data_1 has 14 features and target, 'diameter', left
# Data_2 consists of the same 14 features only -- no 'diameter'
```

In [34]:
```
# Visualize 'diameter' from data_1 using scatterplot

plt.figure(figsize = (15, 8))
plt.scatter(np.arange(1, len(data_1) + 1), data_1['diameter'], s = 50, c = 'b'
)
plt.title('Diameter from Data_1', fontsize = 20, pad = 20)
plt.xlabel('Data Point #', fontsize = 20, labelpad = 15)
plt.ylabel('Diameter (km)', fontsize = 20, labelpad = 15)
plt.show()
```



In [35]:
```
# It appears 'diameter' has large number of small values and only few large va
lues
# Get some insights from min, max, median and mean of diameter in data_1
```

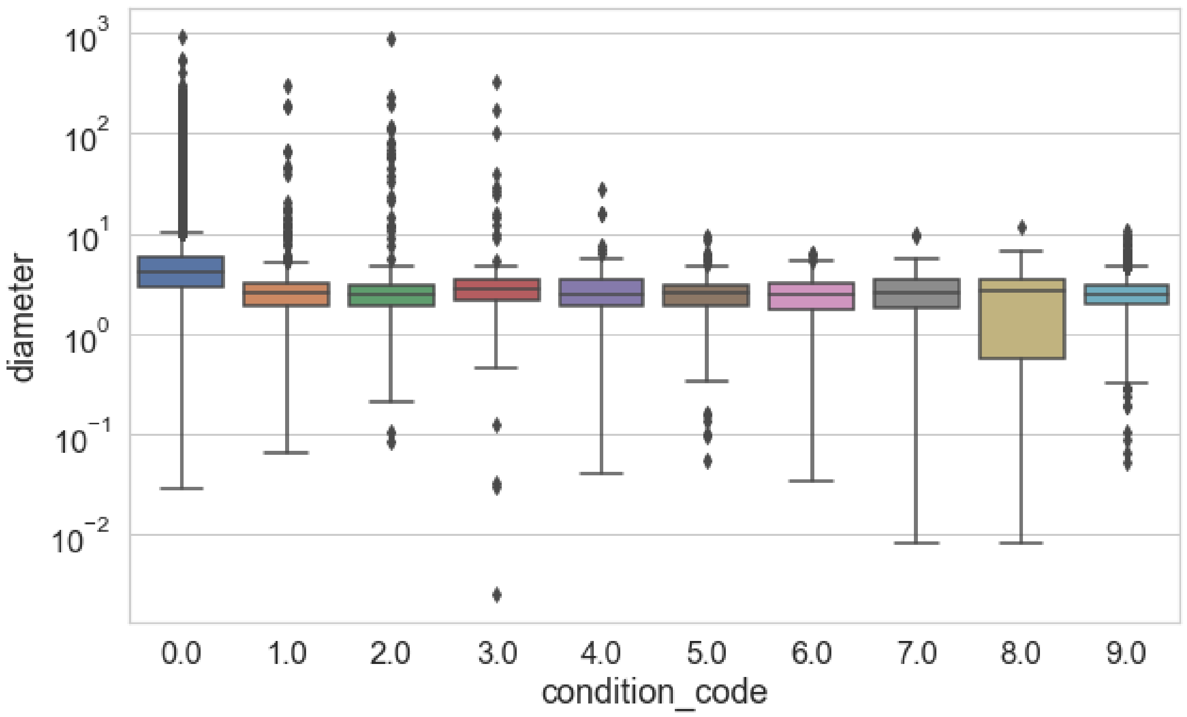
In [36]:
```
# Print min, max, median and mean of diameter in data_1

print("Min diameter in km -->", round(data_1['diameter'].min(), 4))
print("Max diameter in km -->", round(data_1['diameter'].max(), 4))
print("Median diameter in km -->", round(data_1['diameter'].median(), 4))
print("Mean diameter in km -->", round(data_1['diameter'].mean(), 4))
```

```
Min diameter in km --> 0.0025
Max diameter in km --> 939.4
Median diameter in km --> 3.956
Mean diameter in km --> 5.4825
```

In [37]:
```
# Key observations:
    # 1) Max value is much larger than mean (~ 2 orders of magnitude)
    # 2) Despite that, mean and median are very close --> large values are sma
ll portion of the total number of observations
```

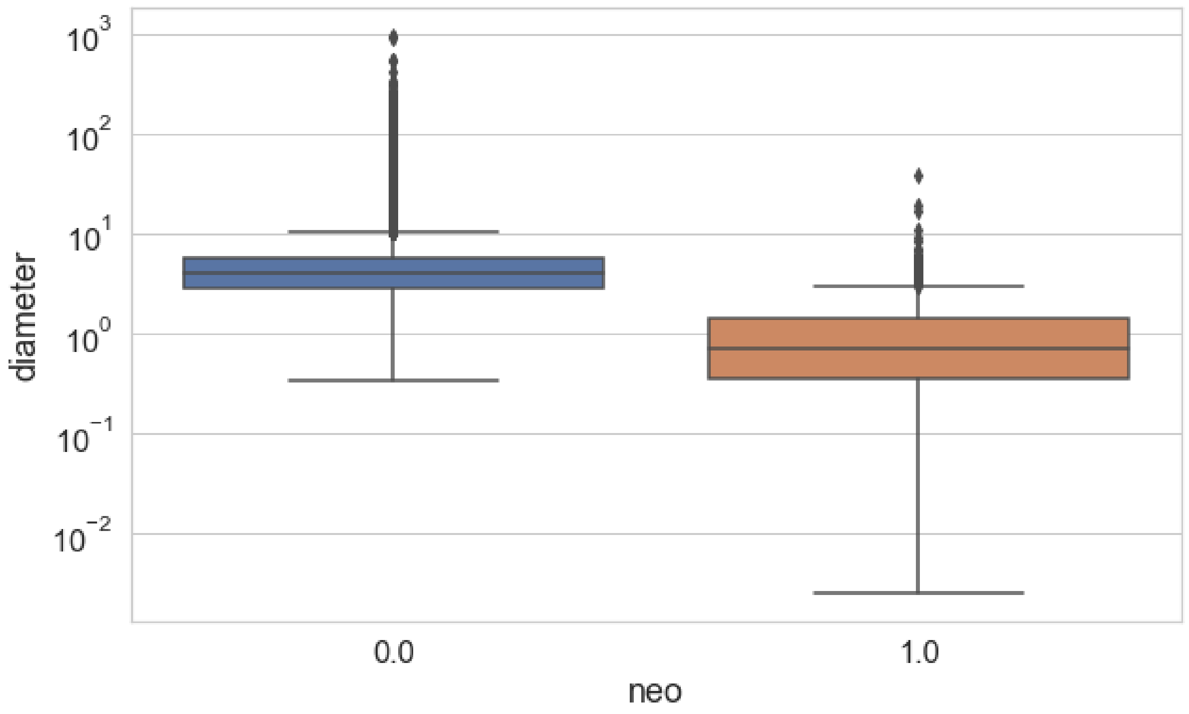
In [38]:
```
# Explore further by using boxplots
# Note: use log scale due to large spread and disparity between the number of
 small (majority) and large diameter values
```

In [39]:
```
# Boxplot of 'diameter' in data_1 vs. 'condition_code' classes

plt.figure(figsize = (10, 6))
sns.boxplot(x = 'condition_code', y = 'diameter', data = data_1)
plt.yscale('log')
plt.show()
```

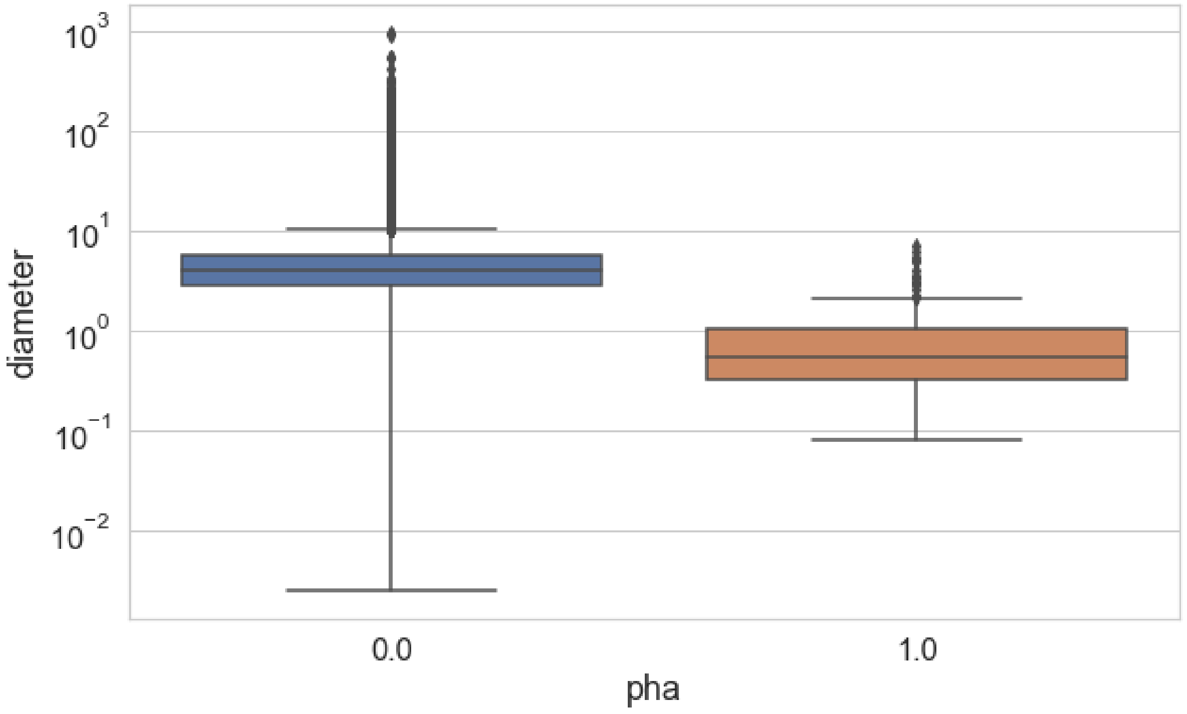


In [40]:
```
# Boxplot confirms that most of the diameter values are small -- between 1 and
10 km
# Values greater than 10 km are considered outliers
```

In [41]:
```
# Boxplot of 'diameter' in data_1 vs. 'neo' classes

plt.figure(figsize = (10, 6))
sns.boxplot(x = 'neo', y = 'diameter', data = data_1)
plt.yscale('log')
plt.show()
```



In [42]:
```
# Boxplot of 'diameter' in data_1 vs. 'pha' classes

plt.figure(figsize = (10, 6))
sns.boxplot(x = 'pha', y = 'diameter', data = data_1)
plt.yscale('log')
plt.show()
```



In [43]:
```
# Distributions of values by 'neo' and 'pha' classes are similar to that by 'c
ondition_code' classes
```

In [44]:
```
# This concludes Data Processing and EDA section
```

In [45]:
```
# 2) Apply XGBRegressor
```

In [46]:
```
# Separate features and target which will be used with XGB model

X_1 = data_1.iloc[:, :-1].values # data_1 features -- all columns, but last
y_1 = data_1.iloc[:, -1].values # data_1 target -- last column

X_2 = data_2.values # data_2 -- features only
```

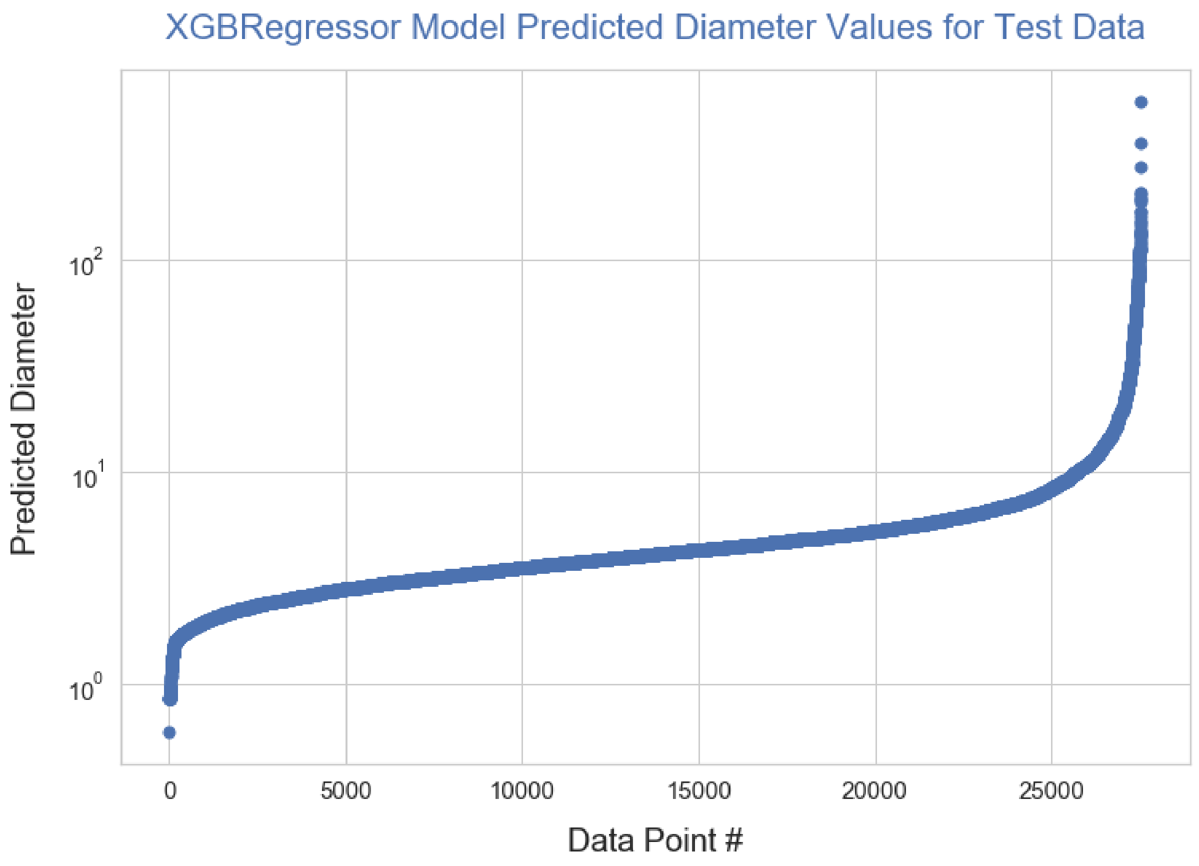
In [47]:
```
# Split X_1 and y_1 in train/test sets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_1, y_1, test_size = 0.2,
random_state = 0)
```

In [48]:
```
# Create XGBRegressor model

from xgboost import XGBRegressor

model_ini = XGBRegressor(objective = 'reg:squarederror') # denote model as'in
i" to distinguish from optimized model later
```

In [49]:
```
# Fit & predict

model_ini.fit(X_train, y_train)

y_pred_1_ini = model_ini.predict(X_test)
```

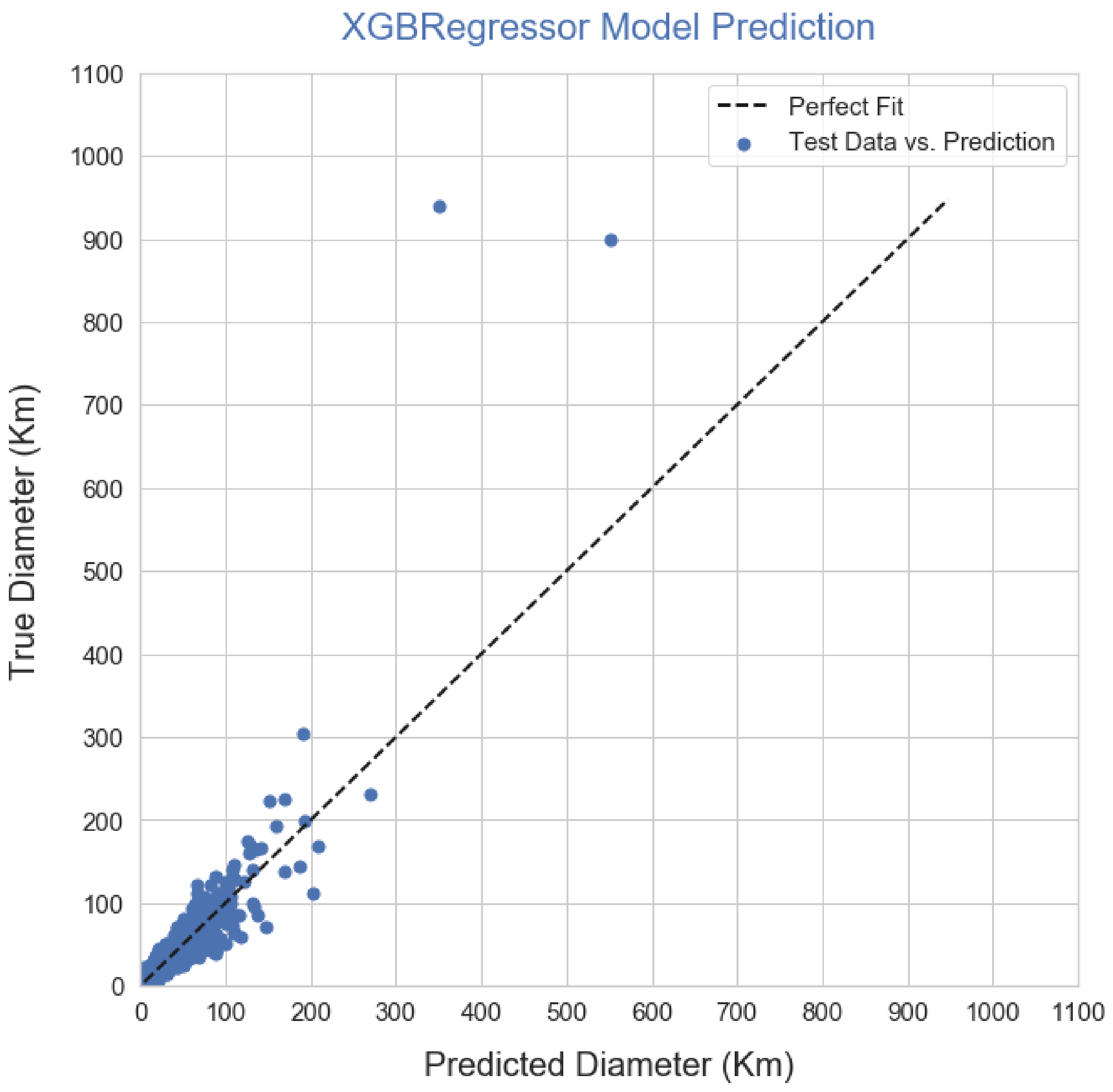
In [50]:
```
# Examine predictions
```

In [51]:
```
# Plot predicted diameter values in ascending order
# Use log scale in order to display well all values

plt.figure(figsize = (12, 8))

plt.scatter(np.arange(1, len(X_test) + 1), np.sort(y_pred_1_ini), s = 50, colo
r = 'b')
plt.yscale('log')
plt.xlabel('Data Point #', fontsize = 20, labelpad = 15)
plt.ylabel('Predicted Diameter', fontsize = 20, labelpad = 15)
plt.title('XGBRegressor Model Predicted Diameter Values for Test Data', fontsi
ze = 22, c = 'b', pad = 20)
plt.tick_params(labelsize = 15)
plt.show()
```



In [52]:
```
# Main observations from plot
    # 1) small portion of predicted values are smaller than 1 km
    # 2) largest predicted value is approximately 500 km (10 ** 2.7)
    # 3) plot shows that vast majority of predicted values fall between 1 and
 10 Km (10 ** 0 and 10 ** 1)
```

In [53]:
```
# Compare predictions, y_pred_1_ini, to test values, y_test, using scatterplot

# create line to represent perfect fit to y_test

y_line = np.arange(int(y_test.min()) - 10, int(y_test.max()) + 10)

# set axes limits - adjust if necessary
x_min = 0
x_max = y_test.max() + 100
d_x = 100

y_min = 0
y_max = y_test.max() + 100
d_y = 100

plt.figure(figsize = (10, 10))
ax = plt.axes()

ax.set_xlim(x_min, x_max)
ax.set_xticks(np.arange(x_min, x_max + d_x, d_x))

ax.set_ylim(y_min, y_max)
ax.set_yticks(np.arange(y_min, y_max + d_y, d_y))

plt.scatter(y_pred_1_ini, y_test, s = 50, c = 'b', label = 'Test Data vs. Pred
iction')
plt.plot(y_line, y_line, 'k--', lw = 2, label = 'Perfect Fit')
plt.xlabel('Predicted Diameter (Km)', fontsize = 20, labelpad = 15)
plt.ylabel('True Diameter (Km)', fontsize = 20, labelpad = 15)
plt.title('XGBRegressor Model Prediction', fontsize = 22, c = 'b', pad = 20)
plt.legend(fontsize = 15)
plt.tick_params(labelsize = 15)
plt.show()
```



In [54]:
```
# Except for two "outliers", predictions are closely grouped around the perfec
t fit line
# Notes:
    # Perhaps the only limitation of XGBoost is that its predictions are cappe
d by the data used for training
    # From the scatter plot of all diameter values in the EDA section and the
 current plot,
    # it is clear that the training data contains only points with diameter sm
aller than 600 km.
    # That's why the predictions with the test data could not capture well the
two points with diameter greater than 800 km
```

In [55]:
```
# Examine model predictions in a more quantitative way --> view statistics of
 residuals
```

In [56]:
```
# Get residuals

residuals_1_ini = y_test - y_pred_1_ini
residuals_1_ini
```

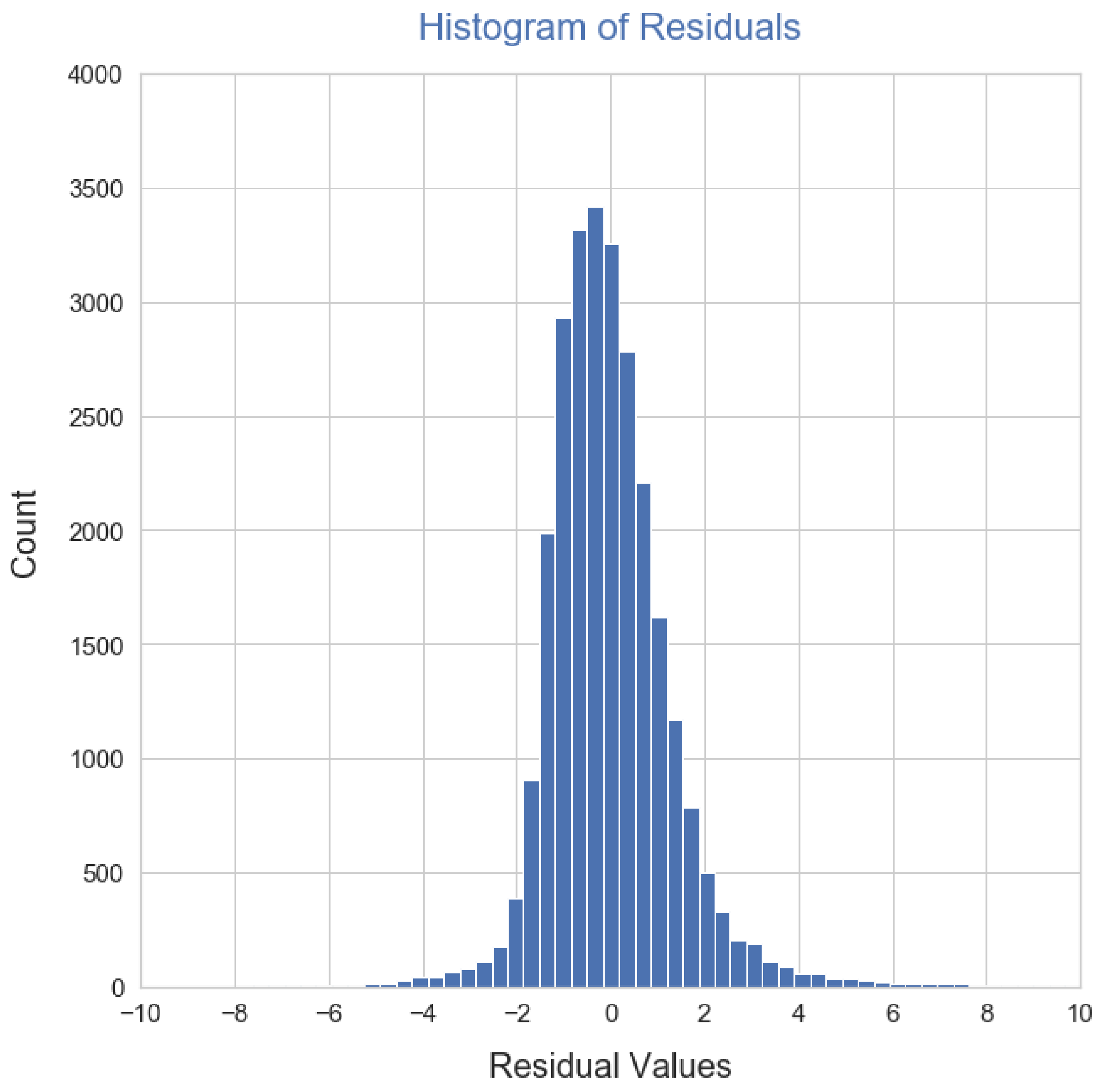
Out[56]:
```
array([-0.91383727, -1.68171666, -0.20006578, ...,  0.16652586,
       -0.47324387,  0.01414502])
```

In [57]:
```
# Get residuals mean and standard deviation, sigma
# Note: residuals sigma is in fact RMSE of predictions

print("Residuals_ini Mean:", round(residuals_1_ini.mean(),4))
print("Residuals_ini Sigma:", round(residuals_1_ini.std(),4))
```

```
Residuals_ini Mean: 0.0187
Residuals_ini Sigma: 4.9317
```

In [58]:
```
# Mean is close to zero and sigma is small compared to test diameter values --
> indicates good model accuracy
```

In [59]:
```
# Examine further: plot the histograms of the residuals -->
    # for better visualization plot histogram only for values within two sigma
s from the mean (~ 95% of all data points)

# Set axes limits - adjust if necessary
x_min = -10
x_max = 10
d_x = 2

y_min = 0
y_max = 4000
d_y = 500

plt.figure(figsize = (10, 10))
ax = plt.axes()

ax.set_xlim(x_min, x_max)
ax.set_xticks(np.arange(x_min, x_max + d_x, d_x))

ax.set_ylim(y_min, y_max)
ax.set_yticks(np.arange(y_min, y_max + d_y, d_y))

plt.hist(residuals_1_ini, bins = 2000, color = 'b')
plt.xlabel('Residual Values', fontsize = 20, labelpad = 15)
plt.ylabel('Count', fontsize = 20, labelpad = 15)
plt.title('Histogram of Residuals', fontsize = 22, c = 'b', pad = 20)
plt.tick_params(labelsize = 15)
plt.show()
```

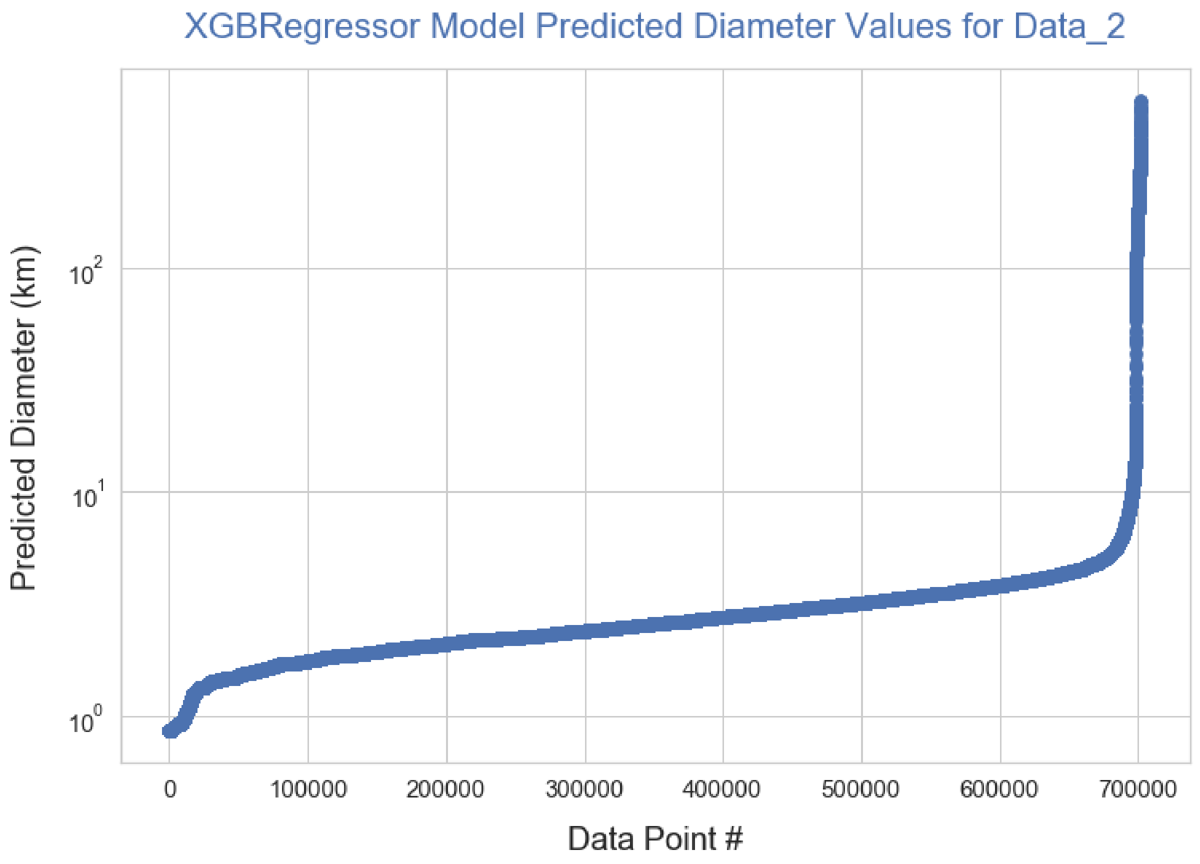


In [60]:
```
# Visually residuals histogram appears close to normal distribution -->
    # it is skewed slightly towards positive values which means model is sligh
tly underevaluating
```

In [61]:
```
# Predict diameter values for asteroids with unknown diameter, data_2

y_pred_2_ini = model_ini.predict(X_2)
```

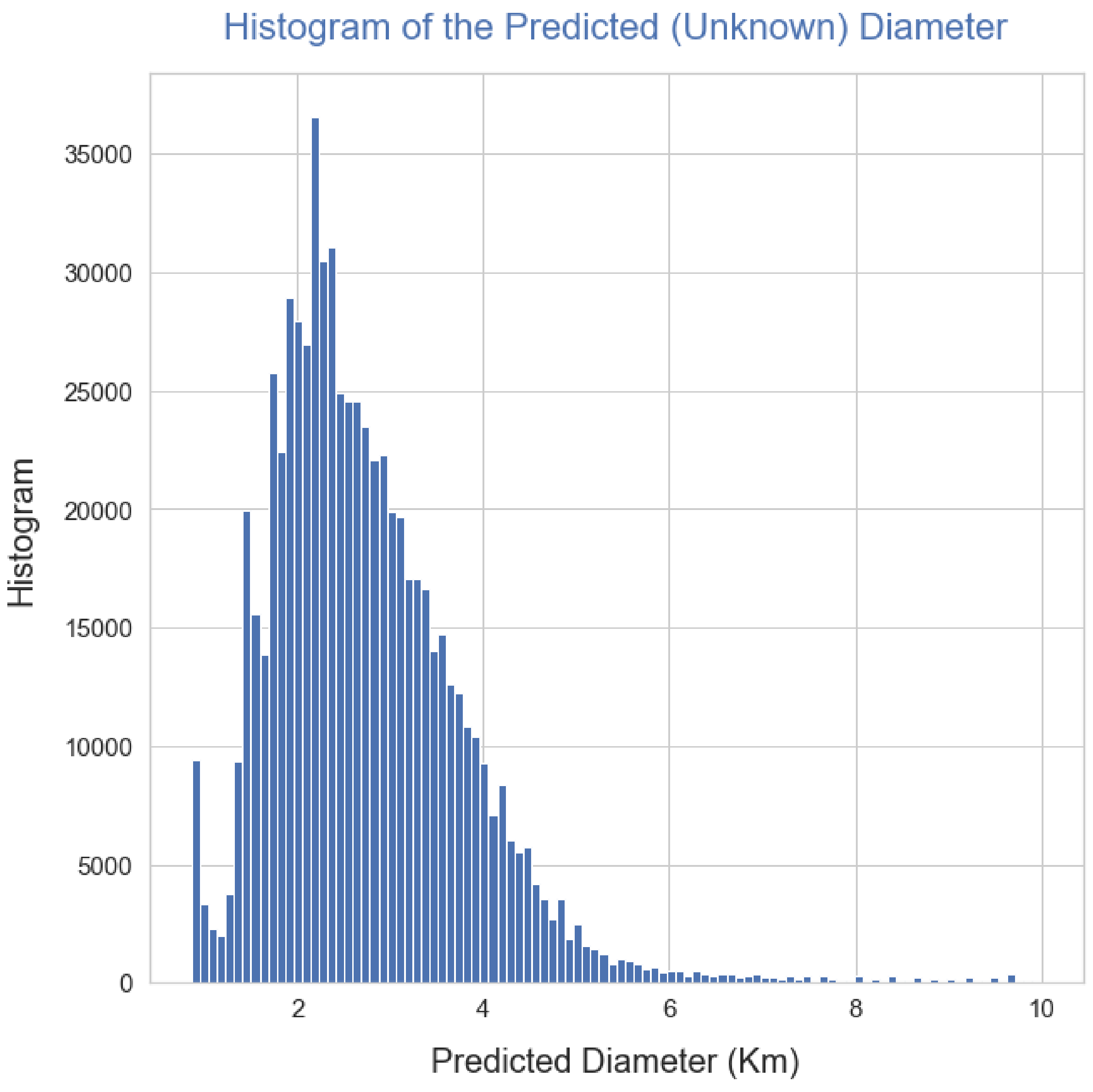
In [62]:
```
# Examine properties of predicted values by creating few simple plots
```

In [63]:
```
# Plot predicted diameter values in ascending order
# Use log scale in order to display well all data points

plt.figure(figsize = (12, 8))

plt.scatter(np.arange(1, len(X_2) + 1), np.sort(y_pred_2_ini), s = 50, color =
'b')
plt.yscale('log')
plt.xlabel('Data Point #', fontsize = 20, labelpad = 15)
plt.ylabel('Predicted Diameter (km)', fontsize = 20, labelpad = 15)
plt.title('XGBRegressor Model Predicted Diameter Values for Data_2', fontsize
= 22, c = 'b', pad = 20)
plt.tick_params(labelsize = 15)
plt.show()
```



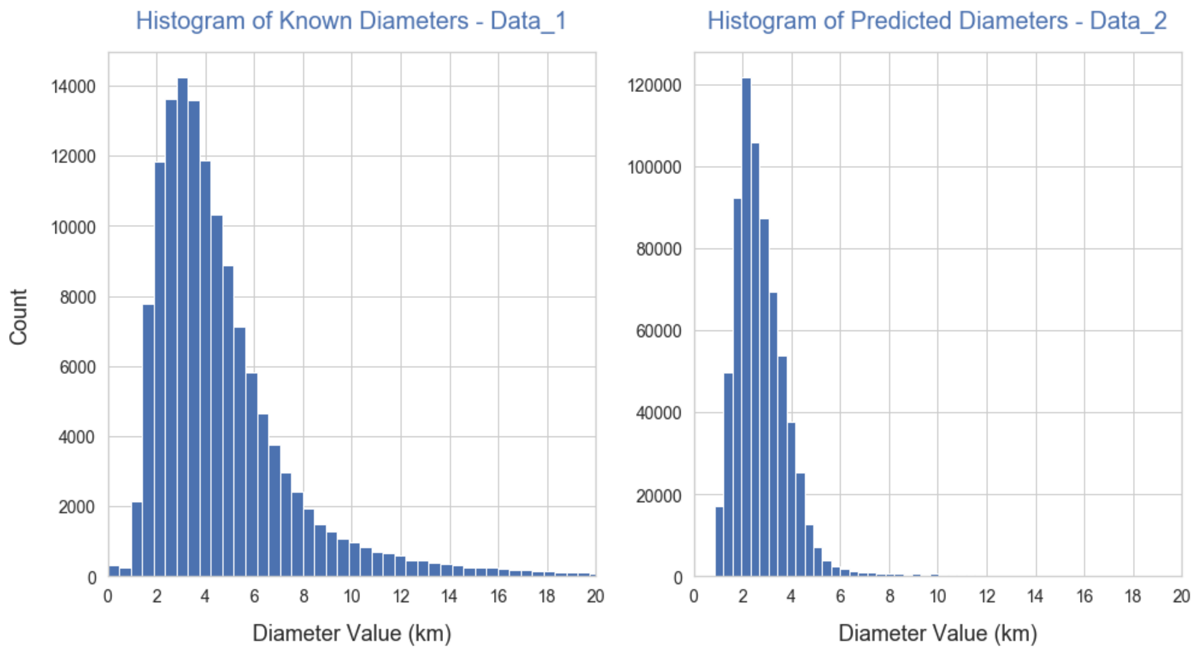
In [64]:
```
# The range of the predicted unknown diameter values is similar to that of the
predicted test values
```

In [65]:
```
# Plot histogram of predicted diameter values

# For better visualization, limit histogram to diameter values smaller than 10
Km
pred_limit = 10

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

plt.hist(y_pred_2_ini[y_pred_2_ini < pred_limit], bins = 100, color = 'b')
plt.xlabel('Predicted Diameter (Km)', fontsize = 20, labelpad = 15)
plt.ylabel('Histogram', fontsize = 20, labelpad = 15)
plt.title('Histogram of the Predicted (Unknown) Diameter', fontsize = 22, c =
'b', pad = 20)
plt.tick_params(labelsize = 15)
plt.show()
```



In [66]:
```
# Predicted values for unknown diameter (data_2) have Poisson-like distributio
n with most of the values between 1.5 and 4 km
```

In [67]:
```
# Examine if distribution is similar to distribution of known diameter values
 -->
    # for adequate comparison set x-axis limit to 20 km

# set axes limits - adjust if necessary
x_min = 0
x_max = 20
d_x = 2

fig, axes = plt.subplots(1, 2, sharey = False, figsize=(16,8))

# known diameter values
axes[0].hist(y_1, bins = 2000, color = 'b')
axes[0].set_title('Histogram of Known Diameters - Data_1', fontsize = 20, c =
'b', pad = 20)
axes[0].set_xlabel('Diameter Value (km)', fontsize = 18, labelpad = 15)
axes[0].set_ylabel('Count', fontsize = 18, labelpad = 15)
axes[0].set_xlim(x_min, x_max)
axes[0].set_xticks(np.arange(x_min, x_max + d_x, d_x))
axes[0].tick_params(labelsize = 14)


# predicted unknown diameter values
axes[1].hist(y_pred_2_ini, bins = 1500, color = 'b')
axes[1].set_title('Histogram of Predicted Diameters - Data_2', fontsize = 20,
c = 'b', pad = 20)
axes[1].set_xlabel('Diameter Value (km)', fontsize = 18, labelpad = 15)
axes[1].set_xlim(x_min, x_max)
axes[1].set_xticks(np.arange(x_min, x_max + d_x, d_x))
axes[1].tick_params(labelsize = 14)

plt.show()
```



In [68]:
```
# It appears that predicted diameter has significantly narrower distribution e
ncompassing smaller values

# Couple of comments regarding this observation:

# 1) We do not know how data with known and unknown asteroid diameter were col
lected -->
    # Thus, comparison between these two histograms may not be fully justified

# 2) However, if we assume that the two datasets are derived from the same ast
ronomical observations and taking into account
# that the number of observation for asteroids with unknown diameter is ~ 5 ti
mes greater than that of known diameter
# one would expect that the predicted values should have similar or even wider
distribution

# Based on #2, the question whether an optimized model would provide different
results arises
```

In [69]:
```
# 3) Model optimization
```

In [70]:
```
# 3.1) First attempt at optimization --> RandomizedSearch
# Note: GridSearch cannot handle large number of combinations --> limit the nu
mber of hyperparameters and their ranges
```

In [71]:
```
# For tuning we use 'max_depth', 'min_child_weight', 'gamma', 'n_estimators',
 'learning_rate', and 'subsample'


grid_random = {'max_depth': [3, 6, 10, 20],
              'min_child_weight': np.arange(1, 10, 1),
              'gamma': np.arange(0, 10, 1),
               'n_estimators': [50, 100, 150],
              'learning_rate': [0.1, 0.2, 0.3],
              'subsample': np.arange(0.5, 1.0, 0.1)}

from sklearn.model_selection import RandomizedSearchCV

model = XGBRegressor(objective = 'reg:squarederror')

model_random = RandomizedSearchCV(estimator = model,
                                 param_distributions = grid_random,
                                 n_iter = 100,
                                 cv = 5,
                                 verbose = 2,
                                 random_state = 42,
                                 n_jobs = -1)
```

In [72]:
```
# Fit model_random with X_train, y_train (using same data as with model_ini)

model_random.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:  1.6min
[Parallel(n_jobs=-1)]: Done 138 tasks      | elapsed:  8.9min
[Parallel(n_jobs=-1)]: Done 341 tasks      | elapsed: 23.9min
[Parallel(n_jobs=-1)]: Done 500 out of 500 | elapsed: 38.0min finished
```

Out[72]:
```
RandomizedSearchCV(cv=5, error_score=nan,
                   estimator=XGBRegressor(base_score=0.5, booster='gbtree',
                                          colsample_bylevel=1,
                                          colsample_bynode=1,
                                          colsample_bytree=1, gamma=0,
                                          importance_type='gain',
                                          learning_rate=0.1, max_delta_step=
0,

                                          max_depth=3, min_child_weight=1,
                                          missing=None, n_estimators=100,
                                          n_jobs=1, nthread=None,
                                          objective='reg:squarederror',
                                          random_state=0, reg_...
                   iid='deprecated', n_iter=100, n_jobs=-1,
                   param_distributions={'gamma': array([0, 1, 2, 3, 4, 5, 6,
7, 8, 9]),
                                        'learning_rate': [0.1, 0.2, 0.3],
                                        'max_depth': [3, 6, 10, 20],
                                        'min_child_weight': array([1, 2, 3,
4, 5, 6, 7, 8, 9]),
                                        'n_estimators': [50, 100, 150],
                                        'subsample': array([0.5, 0.6, 0.7, 0.
8, 0.9])},
                   pre_dispatch='2*n_jobs', random_state=42, refit=True,
                   return_train_score=False, scoring=None, verbose=2)
```

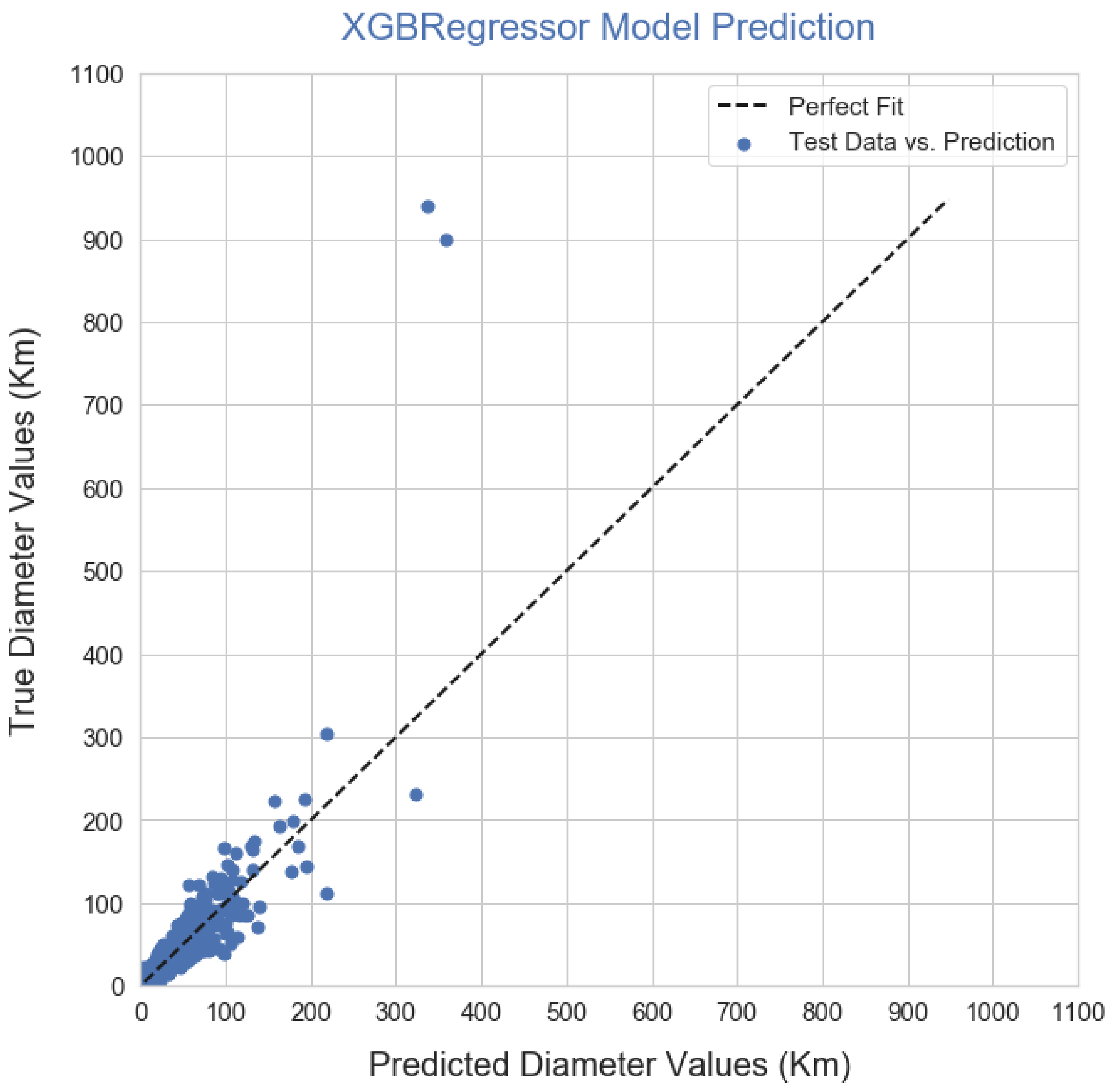
In [73]:
```
# Print best score and best model parameters

print("Best score: %f with %s" % (model_random.best_score_, model_random.best_
params_))
```

```
Best score: 0.867163 with {'subsample': 0.8999999999999999, 'n_estimators': 1
00, 'min_child_weight': 4, 'max_depth': 6, 'learning_rate': 0.1, 'gamma': 0}
```

In [74]:
```
# Get best estimator

model_rand = model_random.best_estimator_
```

In [75]:
```
# Use model_rand to make predictions for X_test and compare with true values,
 y_test

y_pred_1_rand = model_rand.predict(X_test)
```

In [76]:
```
# Compare prediction to test values

# create line to represent perfect fit to data test values, y_test

y_line = np.arange(int(y_test.min()) - 10, int(y_test.max()) + 10)

# set axes limits - adjust if necessary
x_min = 0
x_max = y_test.max() + 100
d_x = 100

y_min = 0
y_max = y_test.max() + 100
d_y = 100

plt.figure(figsize = (10, 10))
ax = plt.axes()

ax.set_xlim(x_min, x_max)
ax.set_xticks(np.arange(x_min, x_max + d_x, d_x))

ax.set_ylim(y_min, y_max)
ax.set_yticks(np.arange(y_min, y_max + d_y, d_y))

plt.scatter(y_pred_1_rand, y_test, s = 50, c = 'b', label = 'Test Data vs. Pre
diction')
plt.plot(y_line, y_line, 'k--', lw = 2, label = 'Perfect Fit')
plt.xlabel('Predicted Diameter Values (Km)', fontsize = 20, labelpad = 15)
plt.ylabel('True Diameter Values (Km)', fontsize = 20, labelpad = 15)
plt.title('XGBRegressor Model Prediction', fontsize = 22, c = 'b', pad = 20)
plt.legend(fontsize = 15)
plt.tick_params(labelsize = 15)
plt.show()
```



In [77]:
```
# Predictions from  RandomizedSearch model have similar behavior to those from
Initial model
```

In [78]:
```
# Get residuals

residuals_1_rand = y_test - y_pred_1_rand
residuals_1_rand
```

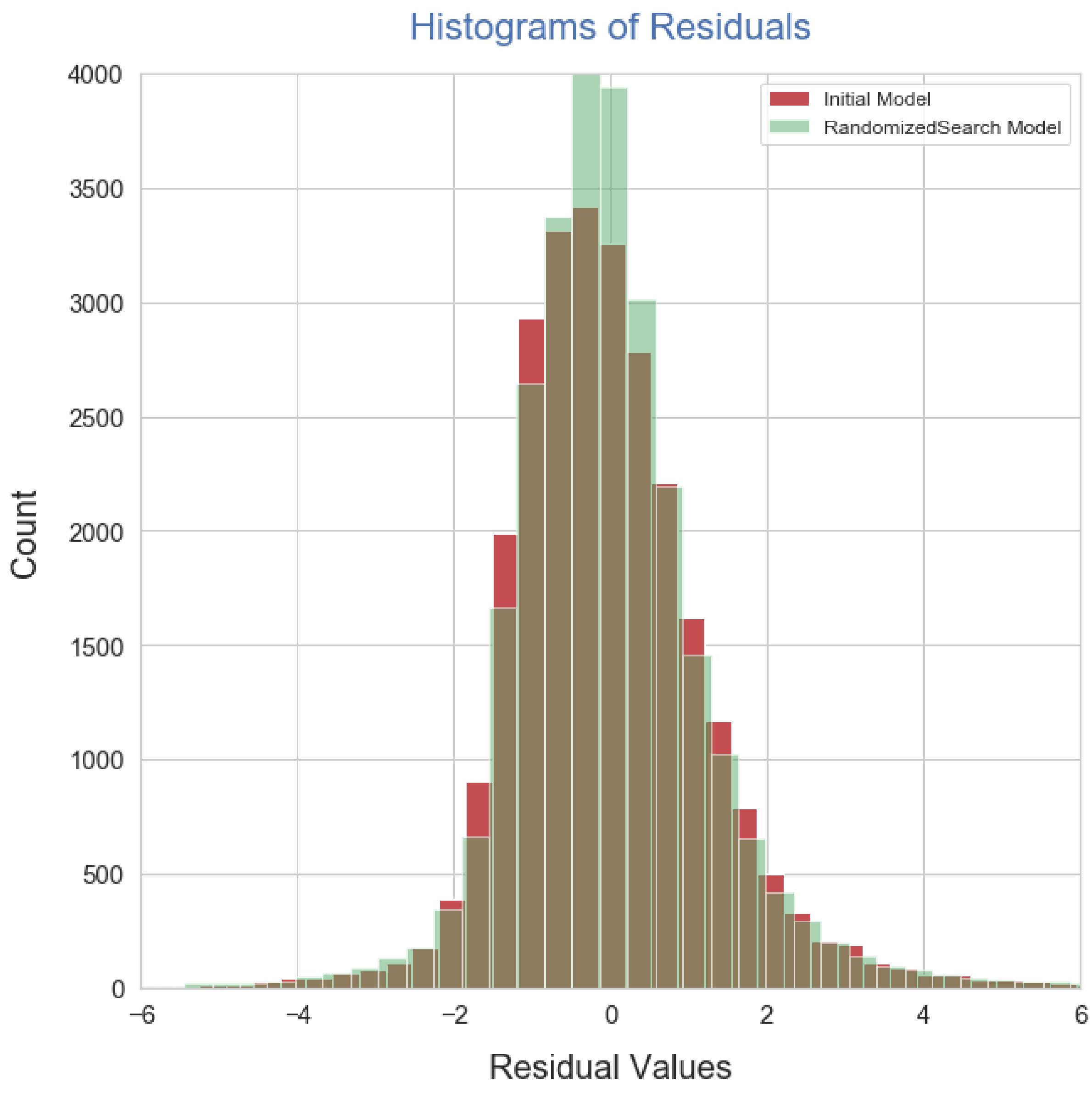
Out[78]:
```
array([-0.74999674, -1.42305062, -0.52065628, ..., -0.1360371 ,
       -0.68275229,  0.09085095])
```

In [79]:
```
# Compare residuals mean and standard deviation, sigma, from Initial and Rando
mizedSearch models

print("Residuals_ini Mean:", round(residuals_1_ini.mean(),4))
print("Residuals_ini Sigma:", round(residuals_1_ini.std(),4))
print('\n')
print("Residuals_rand Mean:", round(residuals_1_rand.mean(),4))
print("Residuals_rand Sigma:", round(residuals_1_rand.std(),4))
```

```
Residuals_ini Mean: 0.0187
Residuals_ini Sigma: 4.9317


Residuals_rand Mean: 0.0379
Residuals_rand Sigma: 5.5739
```

In [80]:
```
# Mean and sigma of residuals from RandomizedSearch are worse than those from
 Initial model!
```

In [81]:
```
# Compare histogram of residuals with Initial model --> for better comparison
 plot histograms on same graph

# Set axes limits - adjust if necessary
x_min = -6
x_max = 6
d_x = 2

y_min = 0
y_max = 4000
d_y = 500

plt.figure(figsize = (10, 10))
ax = plt.axes()

ax.set_xlim(x_min, x_max)
ax.set_xticks(np.arange(x_min, x_max + d_x, d_x))

ax.set_ylim(y_min, y_max)
ax.set_yticks(np.arange(y_min, y_max + d_y, d_y))

plt.hist(residuals_1_ini, bins = 2000, color = 'r', label = 'Initial Model')
plt.hist(residuals_1_rand, bins = 2000, color = 'g', alpha = 0.5, label = 'Ran
domizedSearch Model')
plt.xlabel('Residual Values', fontsize = 20, labelpad = 15)
plt.ylabel('Count', fontsize = 20, labelpad = 15)
plt.title('Histograms of Residuals', fontsize = 22, c = 'b', pad = 20)
plt.tick_params(labelsize = 15)
plt.legend(fontsize = 12)

plt.show()
```



In [82]:
```
# Histograms are similar with no significant improvement
# From these results we can conclude that the RanomizedSearch optimization doe
s not provide better results
```

In [83]:
```
# 3.2) Second attempt at optimization --> Bayesian Optimization using Hyperopt
```

In [84]:

```
# We will use hyperopt with the training set, but we need to have a validation
subset to be able to perform optimization

# Split X_train, y_train into hp_train and hp_valid subsets

X_hp_train, X_hp_valid, y_hp_train, y_hp_valid = train_test_split(X_train, y_t
rain, test_size = 0.2, random_state = 0)


# Hyperopt allows exploring large number of hyperparameters over wide ranges

from hyperopt import fmin, tpe, hp, STATUS_OK, Trials, space_eval
from sklearn import metrics

# Create hyperparameter space to search over
space = {'max_depth': hp.choice('max_depth', np.arange(3, 15, 1, dtype = int
)),
        'n_estimators': hp.choice('n_estimators', np.arange(50, 300, 10, dtype
= int)),
        'colsample_bytree': hp.quniform('colsample_bytree', 0.5, 1.0, 0.1),
        'min_child_weight': hp.choice('min_child_weight', np.arange(0, 10, 1,
dtype = int)),
        'subsample': hp.quniform('subsample', 0.5, 1.0, 0.1),
        'learning_rate': hp.quniform('learning_rate', 0.1, 0.3, 0.1),
         'gamma': hp.choice('gamma', np.arange(0, 20, 0.5, dtype = float)),
         'reg_alpha': hp.choice('reg_alpha', np.arange(0, 20, 0.5, dtype = flo
at)),
         'reg_lambda': hp.choice('reg_lambda', np.arange(0, 20, 0.5, dtype = f
loat)),

        'objective': 'reg:squarederror',

        'eval_metric': 'rmse'}

def score(params):
    model = XGBRegressor(**params)

    model.fit(X_hp_train, y_hp_train,
              eval_set = [(X_hp_train, y_hp_train), (X_hp_valid, y_hp_valid)],
              verbose = False,
              early_stopping_rounds = 10)

    y_pred = model.predict(X_hp_valid)
    score = np.sqrt(metrics.mean_squared_error(y_hp_valid, y_pred))
    print(score)
    return {'loss': score, 'status': STATUS_OK}

def optimize(trials, space):

    best = fmin(score, space, algo = tpe.suggest, max_evals = 200)
    return best

trials = Trials()
best_params = optimize(trials, space)
```

```
3.885478800988722
3.6950735285498086
3.918128602784262
3.929538316645293
3.987241813936812
4.0297437219880194
3.9548526781534834
3.9348234648278853
4.074282296491382
3.384284950863719
4.104720982482018
3.8369764508702953
3.8797335923892513
3.9806804513973715
4.056250601067333
3.912389924309318
3.7509977652897666
3.7802327948957393
3.93824348465134
4.202095152247892
3.9410809772628754
3.8189446904242477
3.991298066170891
3.816198653562568
3.950853179620017
3.9535539658658343
3.8341901752007703
3.8451952625601797
4.041601931775051
3.9476848376732376
3.9172773563323458
3.7415623139062766
3.843493977191749
3.5853263029535727
3.6292247931646195
3.8226003356891676
3.9704346800676507
3.719727656847263
3.871448534904287
4.0003075724294135
4.129836323341933
3.650413236243255
3.9100978529272297
3.7141957348934276
4.038882878846843
4.010370598151539
3.810282155188676
3.7474588001806906
3.9904028987230524
3.882612899112876
4.152505724426132
3.9197409580873885
4.0794903382454635
3.968277155442419
3.921027604295948
3.9463226393582596
3.9177511123379936
```

```
3.8693774995201196
3.730772843739561
3.9098766342358537
4.141284876394967
3.921041452837176
4.063597583877874
4.148030646948843
3.9328941276631917
3.823015811496009
3.989615630353382
3.732663950063289
3.752422532467613
3.5662724154645264
3.5171648296027898
3.403427945946882
3.734930122913362
3.7809271333048535
3.7741119356339787
3.5223746095070676
3.829687819397164
3.7505789652051313
3.7784306023346135
3.8748238507508272
3.979367498846526
3.8921022721763143
3.5281609377836722
3.893294239589885
3.7857441261408455
4.016429635717327
3.9411289494748947
3.8558964512045897
3.5161936547413957
4.024803674833862
4.053310077153744
4.058253564457811
3.710709305471391
3.8704685467247693
3.977933507387382
4.04782643056598
3.8257456195682638
3.9267106361691027
4.02246714508585
3.971907527705438
3.8907514011123276
3.9835052962015602
3.9539009020061746
3.9021799805304798
4.130285086694141
3.867916131520489
3.569458516379916
3.798697666393376
3.8024218687841467
3.7941346124279147
3.5421681990430076
3.9963146922716115
3.707854767227718
3.8879285497442493
```

```
4.037101536093589
3.962405173146065
3.7893657967200918
3.9674934918590608
3.8916843625116178
3.2738423860110375
4.512565372772012
3.4633541560704653
4.099087635755533
3.844791591357427
3.9003312746068737
3.18044556929116
3.974210208386907
3.8429280094322036
3.768283138070977
3.6490395437969783
3.8014240930677357
3.777561223856346
4.034258998443066
3.8115063246457557
4.089718118438315
4.288816260152815
4.130958254539689
3.8501604440942514
4.035192967260811
3.7974484899115994
3.973468035039453
3.9099407353649895
3.8830381998517263
3.887240393068029
4.0244161148913244
3.7892783846611886
3.7984361547791834
3.8162479942293737
3.8087757533964677
3.9363753878983947
3.78567543789966
3.8543162189946596
3.7820999184575688
3.885822816236926
3.5790055883662193
4.002578870491555
3.8129167307405876
4.022329870767796
3.719594160440886
3.660733791361303
3.8041507391746263
3.7928013740030155
3.833928404822651
3.9944172118009815
3.8943057189603567
3.9754254257310544
3.7461470165949473
3.420323270285551
3.8548757417328603
3.636224421271101
3.9414599621675923
```

```
3.719685169842652
3.8853755500536797
3.4882817428551087
4.044326589743707
3.586837217064332
3.8789765708007407
3.8222640437390876
3.993691603139538
3.90482448804304
4.279871509765899
3.924357068147364
3.9066956863653117
3.7031585783134835
3.7819964121605496
3.838808776205403
4.05583342292094
4.103880359198895
4.058263409462786
3.7468123813431222
3.945091399150182
3.891111795598151
3.785357608266322
3.881676853309181
3.9537484732021353
3.8450684133826405
3.794807086134492
3.837764972488475
3.9928953701475907
3.8523619612798963
100%|██████████████████████████████████████████████████| 200/200 [44:05<00:00,
13.23s/trial, best loss: 3.18044556929116]
```

In [85]:

```python
# Get best parameters
space_eval(space, best_params)
```

Out[85]:

```
{'colsample_bytree': 0.6000000000000001,
 'eval_metric': 'rmse',
 'gamma': 18.5,
 'learning_rate': 0.30000000000000004,
 'max_depth': 4,
 'min_child_weight': 1,
 'n_estimators': 100,
 'objective': 'reg:squarederror',
 'reg_alpha': 3.5,
 'reg_lambda': 0.0,
 'subsample': 1.0}
```

In [86]:
```
# Create model with best parameters
model_opt = XGBRegressor(max_depth = 4,
                         n_estimators = 100,
                         learning_rate = 0.3,
                         min_child_weight = 1,
                         subsample = 1.0,
                         colsample_bytree = 0.6,
                         gamma = 18.5,
                         reg_alpha = 3.5,
                         reg_lambda = 0.0,
                         objective = 'reg:squarederror')

# Fit with hp datasets
model_opt.fit(X_hp_train, y_hp_train,
              eval_set = [(X_hp_train, y_hp_train), (X_hp_valid, y_hp_valid)],
              eval_metric = 'rmse',
              verbose = True,
              early_stopping_rounds = 10)
```

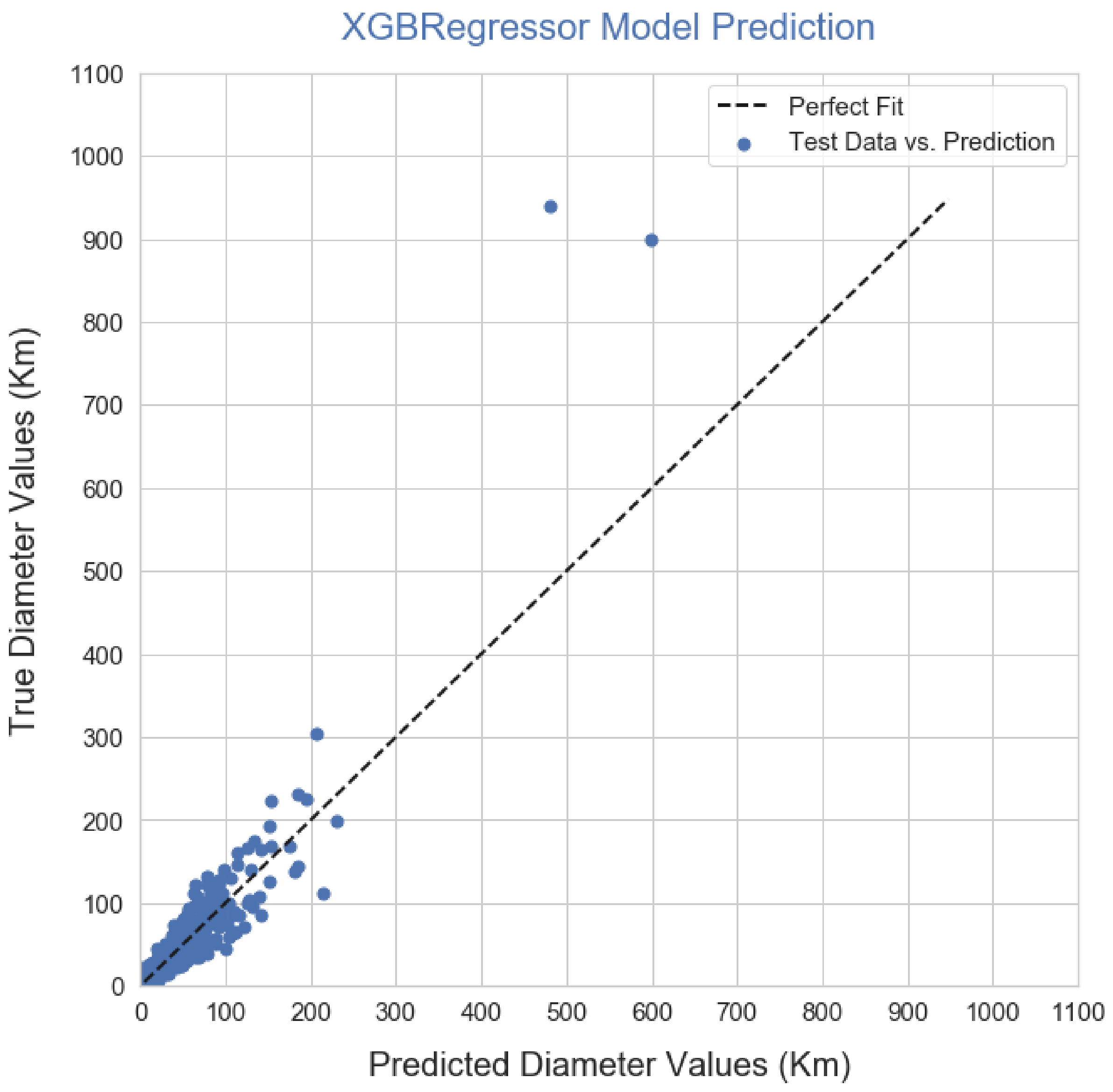
```
[0]	validation_0-rmse:7.37771	validation_1-rmse:8.43978
Multiple eval metrics have been passed: 'validation_1-rmse' will be used for
early stopping.

Will train until validation_1-rmse hasn't improved in 10 rounds.
[1]	validation_0-rmse:5.79572	validation_1-rmse:6.95714
[2]	validation_0-rmse:4.57948	validation_1-rmse:5.59772
[3]	validation_0-rmse:3.80817	validation_1-rmse:4.73038
[4]	validation_0-rmse:3.33314	validation_1-rmse:4.21911
[5]	validation_0-rmse:3.04731	validation_1-rmse:3.93399
[6]	validation_0-rmse:2.87481	validation_1-rmse:3.72508
[7]	validation_0-rmse:2.73631	validation_1-rmse:3.58194
[8]	validation_0-rmse:2.70512	validation_1-rmse:3.56712
[9]	validation_0-rmse:2.63525	validation_1-rmse:3.53271
[10]	validation_0-rmse:2.57363	validation_1-rmse:3.51347
[11]	validation_0-rmse:2.50039	validation_1-rmse:3.38523
[12]	validation_0-rmse:2.48514	validation_1-rmse:3.35766
[13]	validation_0-rmse:2.4475	validation_1-rmse:3.3219
[14]	validation_0-rmse:2.41623	validation_1-rmse:3.31782
[15]	validation_0-rmse:2.37628	validation_1-rmse:3.28033
[16]	validation_0-rmse:2.35888	validation_1-rmse:3.27442
[17]	validation_0-rmse:2.33565	validation_1-rmse:3.25093
[18]	validation_0-rmse:2.31567	validation_1-rmse:3.24554
[19]	validation_0-rmse:2.29801	validation_1-rmse:3.24693
[20]	validation_0-rmse:2.26836	validation_1-rmse:3.23463
[21]	validation_0-rmse:2.22967	validation_1-rmse:3.23472
[22]	validation_0-rmse:2.21134	validation_1-rmse:3.23709
[23]	validation_0-rmse:2.19432	validation_1-rmse:3.2286
[24]	validation_0-rmse:2.19087	validation_1-rmse:3.22758
[25]	validation_0-rmse:2.17581	validation_1-rmse:3.24798
[26]	validation_0-rmse:2.15769	validation_1-rmse:3.21341
[27]	validation_0-rmse:2.13578	validation_1-rmse:3.2131
[28]	validation_0-rmse:2.12261	validation_1-rmse:3.18752
[29]	validation_0-rmse:2.11501	validation_1-rmse:3.18266
[30]	validation_0-rmse:2.09726	validation_1-rmse:3.18045
[31]	validation_0-rmse:2.07411	validation_1-rmse:3.18349
[32]	validation_0-rmse:2.06974	validation_1-rmse:3.18197
[33]	validation_0-rmse:2.04906	validation_1-rmse:3.19454
[34]	validation_0-rmse:2.04579	validation_1-rmse:3.1917
[35]	validation_0-rmse:2.0448	validation_1-rmse:3.19228
[36]	validation_0-rmse:2.02668	validation_1-rmse:3.20543
[37]	validation_0-rmse:2.01003	validation_1-rmse:3.19279
[38]	validation_0-rmse:2.00041	validation_1-rmse:3.19244
[39]	validation_0-rmse:1.99421	validation_1-rmse:3.19558
[40]	validation_0-rmse:1.97943	validation_1-rmse:3.20859
Stopping. Best iteration:
[30]	validation_0-rmse:2.09726	validation_1-rmse:3.18045
```

Out[86]:

```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.6, gamma=18.5,
             importance_type='gain', learning_rate=0.3, max_delta_step=0,
             max_depth=4, min_child_weight=1, missing=None, n_estimators=100,
             n_jobs=1, nthread=None, objective='reg:squarederror',
             random_state=0, reg_alpha=3.5, reg_lambda=0.0, scale_pos_weight=
1,
             seed=None, silent=None, subsample=1.0, verbosity=1)
```

In [87]:
```
# Get predictions from X_test

y_pred_1_opt = model_opt.predict(X_test)
```

In [88]:
```
# Compare predictions, y_pred_1_opt, to test values, y_test

# create line to represent perfect fit to data test values, y_test

y_line = np.arange(int(y_test.min()) - 10, int(y_test.max()) + 10)

# set axes limits - adjust if necessary
x_min = 0
x_max = y_test.max() + 100
d_x = 100

y_min = 0
y_max = y_test.max() + 100
d_y = 100

plt.figure(figsize = (10, 10))
ax = plt.axes()

ax.set_xlim(x_min, x_max)
ax.set_xticks(np.arange(x_min, x_max + d_x, d_x))

ax.set_ylim(y_min, y_max)
ax.set_yticks(np.arange(y_min, y_max + d_y, d_y))

plt.scatter(y_pred_1_opt, y_test, s = 50, c = 'b', label = 'Test Data vs. Pred
iction')
plt.plot(y_line, y_line, 'k--', lw = 2, label = 'Perfect Fit')
plt.xlabel('Predicted Diameter Values (Km)', fontsize = 20, labelpad = 15)
plt.ylabel('True Diameter Values (Km)', fontsize = 20, labelpad = 15)
plt.title('XGBRegressor Model Prediction', fontsize = 22, c = 'b', pad = 20)
plt.legend(fontsize = 15)
plt.tick_params(labelsize = 15)
plt.show()
```



In [89]:
```
# Behavior is similar to that for predictions from Initial model
# However, there is visible improvement of predictions being closer to test va
lues
```

In [90]:
```
# Get residuals

residuals_1_opt = y_test - y_pred_1_opt
residuals_1_opt
```

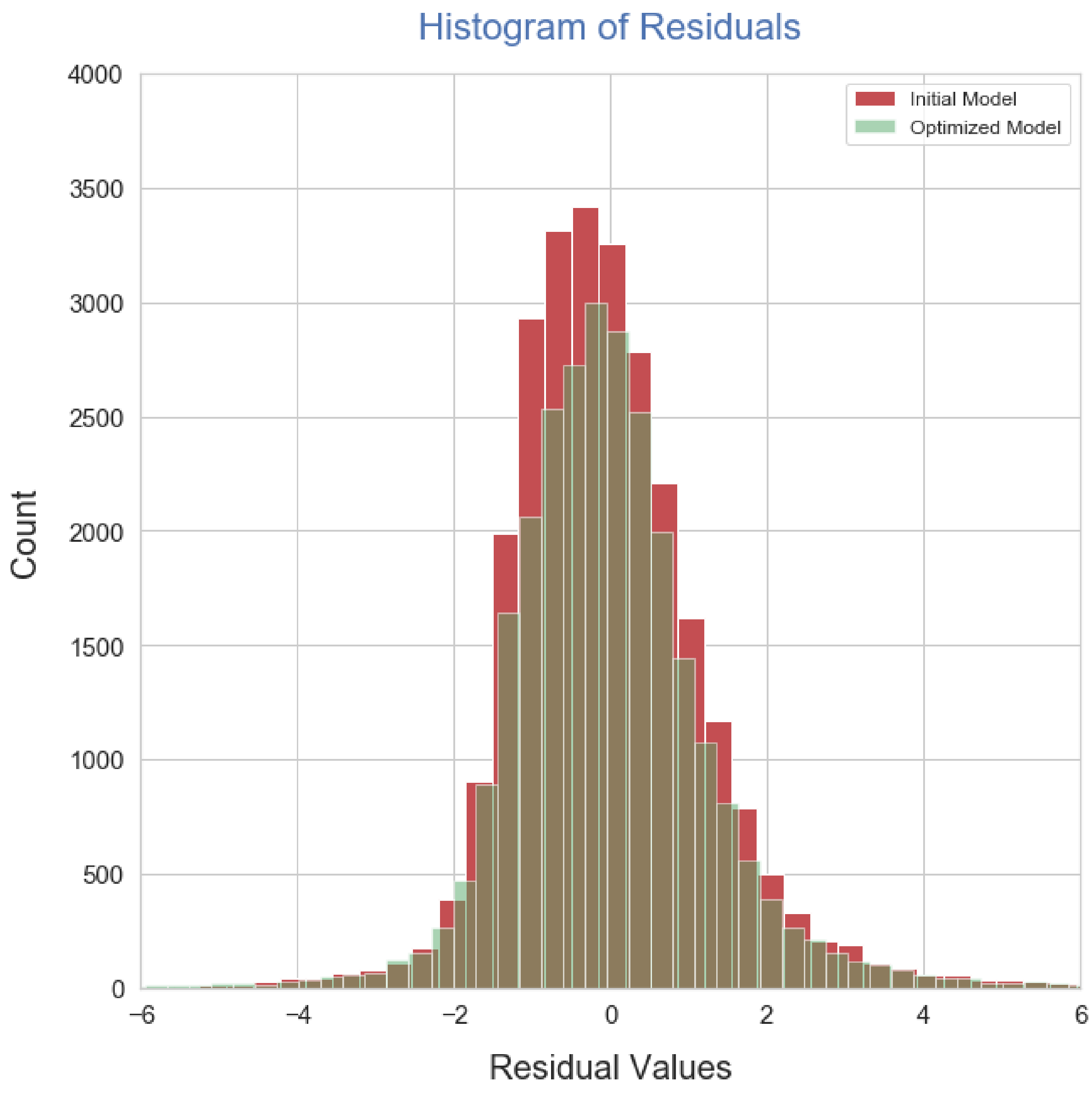
Out[90]:
```
array([-0.74997933, -1.61396144, -0.23879021, ...,  0.04899909,
       -0.55539671, -0.10671389])
```

In [91]:
```
# Compare residuals mean and standard deviation, sigma, from Initial and Optim
ized models

print("Residuals_ini Mean:", round(residuals_1_ini.mean(),4))
print("Residuals_ini Sigma:", round(residuals_1_ini.std(),4))
print('\n')
print("Residuals_opt Mean:", round(residuals_1_opt.mean(),4))
print("Residuals_opt Sigma:", round(residuals_1_opt.std(),4))
```

```
Residuals_ini Mean: 0.0187
Residuals_ini Sigma: 4.9317


Residuals_opt Mean: 0.0321
Residuals_opt Sigma: 4.2539
```

In [92]:
```
# The optimized model has slightly larger mean
# However, sigma is improved by approximatelly 15 %
```

In [93]:
```
# Compare histograms of residuals from Initial and Optimized models --> for be
tter comparison plot histograms on same graph

# Set axes limits - adjust if necessary
x_min = -6
x_max = 6
d_x = 2

y_min = 0
y_max = 4000
d_y = 500

plt.figure(figsize = (10, 10))
ax = plt.axes()

ax.set_xlim(x_min, x_max)
ax.set_xticks(np.arange(x_min, x_max + d_x, d_x))

ax.set_ylim(y_min, y_max)
ax.set_yticks(np.arange(y_min, y_max + d_y, d_y))

plt.hist(residuals_1_ini, bins = 2000, color = 'r', label = 'Initial Model')
plt.hist(residuals_1_opt, bins = 2000, color = 'g', alpha = 0.5, label = 'Opti
mized Model')
plt.xlabel('Residual Values', fontsize = 20, labelpad = 15)
plt.ylabel('Count', fontsize = 20, labelpad = 15)
plt.title('Histogram of Residuals', fontsize = 22, c = 'b', pad = 20)
plt.tick_params(labelsize = 15)
plt.legend(fontsize = 12)
plt.show()
```



```
In [94]: # Histogram from Optimized model is clearly more symmetrical and closer to nor
         mal distribution
         # Also, it is slightly narrower as expected from the smaller sigma
         # These results confirm that Optimized model performance is better than that o
         f Initial model
```

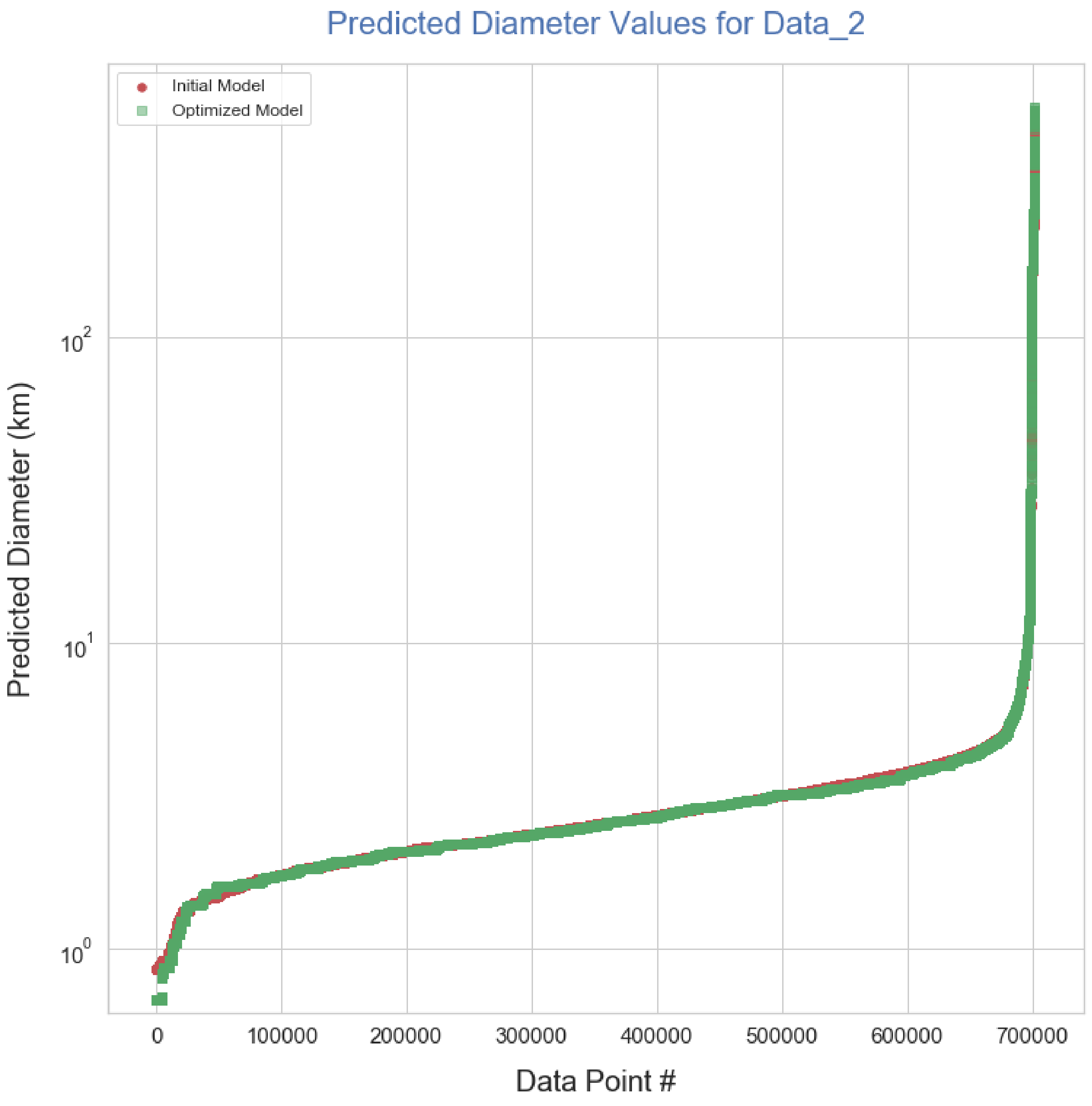
In [95]:
```
# Use Optimized model to predict diameters for dataset with unknown diameter,
 X_2

# Before making predictions we will re-train model using the entire training s
et; test set will be used for evaluation

model_opt.fit(X_train, y_train,
              eval_set = [(X_train, y_train), (X_test, y_test)],
              eval_metric = 'rmse',
              verbose = True,
              early_stopping_rounds = 10)
```

```
[0]     validation_0-rmse:7.5758        validation_1-rmse:9.67777
Multiple eval metrics have been passed: 'validation_1-rmse' will be used for
early stopping.

Will train until validation_1-rmse hasn't improved in 10 rounds.
[1]     validation_0-rmse:5.90844       validation_1-rmse:8.39842
[2]     validation_0-rmse:4.66809       validation_1-rmse:7.00868
[3]     validation_0-rmse:3.88827       validation_1-rmse:6.42491
[4]     validation_0-rmse:3.39238       validation_1-rmse:5.72934
[5]     validation_0-rmse:3.08559       validation_1-rmse:5.46098
[6]     validation_0-rmse:2.91009       validation_1-rmse:5.38548
[7]     validation_0-rmse:2.76386       validation_1-rmse:5.10638
[8]     validation_0-rmse:2.73353       validation_1-rmse:5.02241
[9]     validation_0-rmse:2.66453       validation_1-rmse:4.96718
[10]    validation_0-rmse:2.59948       validation_1-rmse:4.86048
[11]    validation_0-rmse:2.54307       validation_1-rmse:4.83821
[12]    validation_0-rmse:2.52139       validation_1-rmse:4.81831
[13]    validation_0-rmse:2.50635       validation_1-rmse:4.81928
[14]    validation_0-rmse:2.48118       validation_1-rmse:4.84488
[15]    validation_0-rmse:2.45036       validation_1-rmse:4.84349
[16]    validation_0-rmse:2.43351       validation_1-rmse:4.74044
[17]    validation_0-rmse:2.38777       validation_1-rmse:4.75066
[18]    validation_0-rmse:2.37771       validation_1-rmse:4.74841
[19]    validation_0-rmse:2.3619        validation_1-rmse:4.74374
[20]    validation_0-rmse:2.34971       validation_1-rmse:4.7458
[21]    validation_0-rmse:2.31978       validation_1-rmse:4.75435
[22]    validation_0-rmse:2.29915       validation_1-rmse:4.75346
[23]    validation_0-rmse:2.27826       validation_1-rmse:4.75255
[24]    validation_0-rmse:2.26292       validation_1-rmse:4.75264
[25]    validation_0-rmse:2.2481        validation_1-rmse:4.74391
[26]    validation_0-rmse:2.22629       validation_1-rmse:4.75248
Stopping. Best iteration:
[16]    validation_0-rmse:2.43351       validation_1-rmse:4.74044
```

Out[95]:
```
XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
             colsample_bynode=1, colsample_bytree=0.6, gamma=18.5,
             importance_type='gain', learning_rate=0.3, max_delta_step=0,
             max_depth=4, min_child_weight=1, missing=None, n_estimators=100,
             n_jobs=1, nthread=None, objective='reg:squarederror',
             random_state=0, reg_alpha=3.5, reg_lambda=0.0, scale_pos_weight=
1,
             seed=None, silent=None, subsample=1.0, verbosity=1)
```

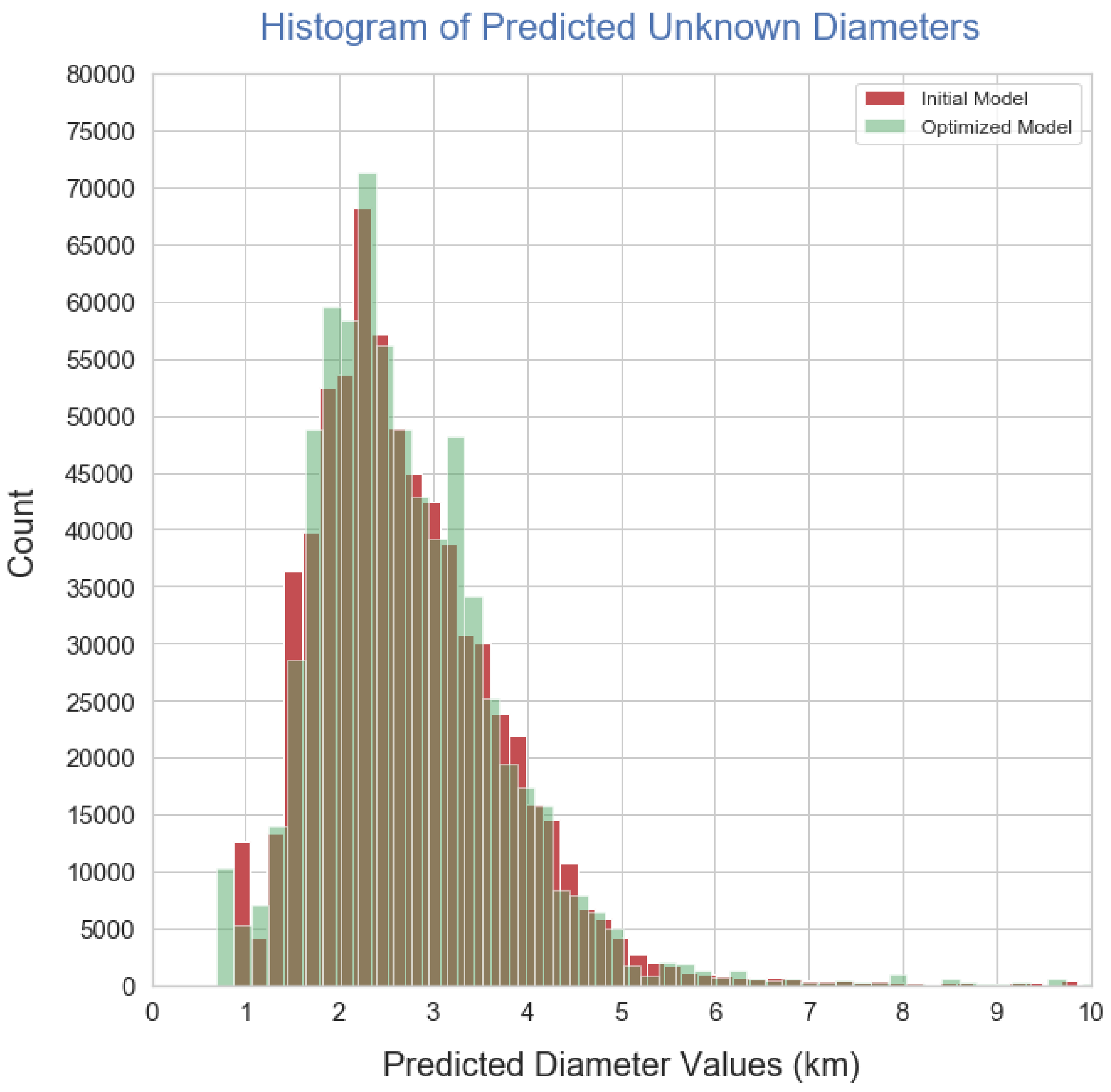
In [96]:
```
# Make predictions using X_2

y_pred_2_opt = model_opt.predict(X_2)
```

In [97]:
```
# Plot predicted diameter values in ascending order and compare with Initial m
odel predictions
# Use log scale

plt.figure(figsize = (12, 12))

plt.scatter(np.arange(1, len(X_2) + 1), np.sort(y_pred_2_ini),
            marker = 'o', color = 'r', s = 30, label = 'Initial Model')
plt.scatter(np.arange(1, len(X_2) + 1), np.sort(y_pred_2_opt),
            marker = 's', color = 'g', s = 30, alpha = 0.5, label = 'Optimized
Model')

plt.yscale('log')

plt.xlabel('Data Point #', fontsize = 20, labelpad = 15)
plt.ylabel('Predicted Diameter (km)', fontsize = 20, labelpad = 15)
plt.title('Predicted Diameter Values for Data_2', fontsize = 22, c = 'b', pad
= 20)
plt.tick_params(labelsize = 15)
plt.legend(fontsize = 12)

plt.show()
```



In [98]:
```
# Both models predictions for the unknown diameter are very close with some de
viation at small diameter values
```

In [99]:
```
# Compare histograms of predicted diameters from Initial and Optimized models
 -->
    # for better comparison plot histograms on same graph

# Set axes limits - adjust if necessary
x_min = 0
x_max = 10
d_x = 1

y_min = 0
y_max = 80000
d_y = 5000

plt.figure(figsize = (10, 10))
ax = plt.axes()

ax.set_xlim(x_min, x_max)
ax.set_xticks(np.arange(x_min, x_max + d_x, d_x))

ax.set_ylim(y_min, y_max)
ax.set_yticks(np.arange(y_min, y_max + d_y, d_y))

plt.hist(y_pred_2_ini, bins = 3000, color = 'r', label = 'Initial Model')
plt.hist(y_pred_2_opt, bins = 3000, color = 'g', alpha = 0.5, label = 'Optimiz
ed Model')
plt.xlabel('Predicted Diameter Values (km)', fontsize = 20, labelpad = 15)
plt.ylabel('Count', fontsize = 20, labelpad = 15)
plt.title('Histogram of Predicted Unknown Diameters', fontsize = 22, c = 'b',
pad = 20)
plt.tick_params(labelsize = 15)
plt.legend(fontsize = 12)
plt.show()
```



In [100]:
```
# As indicated already, predictions from both models are very similar
# Answers question posed earlier:
    # Difference in distributions between known and predicted unknown diameter
s is not an issue of model optimization
# Thus, we have to assume that difference between known and predicted diameter
s is due to different feature values measured
```

In [101]:
```
# Finally, combine the predicted diameter values with features from data_2 to
 complete data as our final delivarable

data_2.head(10)
```

Out[101]:

	a	e	i	om	w	q	ad	per_y	data
681	2.654040	0.171983	11.505648	190.799958	104.993826	2.197591	3.110489	4.323837	40
698	2.610998	0.410284	15.299180	242.551766	91.399514	1.539746	3.682249	4.219081	42
718	2.638780	0.546301	11.564845	183.887287	156.163668	1.197212	4.080348	4.286601	39
729	2.243362	0.177505	4.234895	95.073806	123.549777	1.845154	2.641570	3.360139	39
842	2.279598	0.209766	7.997717	4.071363	316.957206	1.801415	2.757780	3.441878	37
961	2.908998	0.097329	2.602636	145.481660	223.473847	2.625868	3.192128	4.961619	37
984	2.299979	0.277462	4.056565	290.307048	59.553605	1.661822	2.938137	3.488142	35
1008	2.625175	0.455500	15.769676	229.461495	186.428747	1.429408	3.820942	4.253492	34
1010	2.391976	0.350864	5.494744	132.525452	353.279770	1.552718	3.231235	3.699504	34
1064	2.360276	0.297141	8.362855	330.324142	353.652287	1.658942	3.061610	3.626205	33

In [102]:
```
# Reset data_2 indices before adding the predicted diameter values

data_2 = data_2.reset_index(drop = True)

data_2.head(10)
```

Out[102]:

	a	e	i	om	w	q	ad	per_y	data_ar
0	2.654040	0.171983	11.505648	190.799958	104.993826	2.197591	3.110489	4.323837	40087.
1	2.610998	0.410284	15.299180	242.551766	91.399514	1.539746	3.682249	4.219081	42540.
2	2.638780	0.546301	11.564845	183.887287	156.163668	1.197212	4.080348	4.286601	39478.
3	2.243362	0.177505	4.234895	95.073806	123.549777	1.845154	2.641570	3.360139	39112.
4	2.279598	0.209766	7.997717	4.071363	316.957206	1.801415	2.757780	3.441878	37579.
5	2.908998	0.097329	2.602636	145.481660	223.473847	2.625868	3.192128	4.961619	37450.
6	2.299979	0.277462	4.056565	290.307048	59.553605	1.661822	2.938137	3.488142	35366.
7	2.625175	0.455500	15.769676	229.461495	186.428747	1.429408	3.820942	4.253492	34990.
8	2.391976	0.350864	5.494744	132.525452	353.279770	1.552718	3.231235	3.699504	34919.
9	2.360276	0.297141	8.362855	330.324142	353.652287	1.658942	3.061610	3.626205	33882.

In [103]:
```
# Convert predictions array into series with name 'diameter'

y_pred_fin = pd.Series(y_pred_2_opt, name = 'diameter')
y_pred_fin.head(10)
```

Out[103]:
```
0    14.215720
1    15.951775
2     6.792315
3     7.787400
4     7.787400
5    21.908957
6     8.311811
7     8.408096
8     8.742744
9    11.368592
Name: diameter, dtype: float32
```

In [104]:
```
# Combine features with predicted diameter values

data_2 = pd.concat([data_2, y_pred_fin], axis = 1)
data_2.head(10)
```

Out[104]:

	a	e	i	om	w	q	ad	per_y	data_ar
0	2.654040	0.171983	11.505648	190.799958	104.993826	2.197591	3.110489	4.323837	40087.
1	2.610998	0.410284	15.299180	242.551766	91.399514	1.539746	3.682249	4.219081	42540.
2	2.638780	0.546301	11.564845	183.887287	156.163668	1.197212	4.080348	4.286601	39478.
3	2.243362	0.177505	4.234895	95.073806	123.549777	1.845154	2.641570	3.360139	39112.
4	2.279598	0.209766	7.997717	4.071363	316.957206	1.801415	2.757780	3.441878	37579.
5	2.908998	0.097329	2.602636	145.481660	223.473847	2.625868	3.192128	4.961619	37450.
6	2.299979	0.277462	4.056565	290.307048	59.553605	1.661822	2.938137	3.488142	35366.
7	2.625175	0.455500	15.769676	229.461495	186.428747	1.429408	3.820942	4.253492	34990.
8	2.391976	0.350864	5.494744	132.525452	353.279770	1.552718	3.231235	3.699504	34919.
9	2.360276	0.297141	8.362855	330.324142	353.652287	1.658942	3.061610	3.626205	33882.

In [105]:
```
# Data is complete, predicted asteroid diameter values are included --> projec
t's objective achieved
```