Statistical Analysis of Hard Disk Drive Failure

Philip Matthew Unrue

Western Governors University

Abstract

Hard drive failure is the most common form of data loss, which is one of the most impactful problems that businesses can experience. Being able to predict which hard drives are at the highest risk of failure based on understanding of combinations of routine diagnostics test results is an ideal solution to backup and replace failing drives before the data is lost. Backblaze's 4th quarter data from 2019 is examined to determine what information is most relevant to imminent hard drive failure on the day measured. A logistic regression model and a decision tree are examined alongside PCA to look for important predictors. A random forest ensemble model and neural networks are then created to find the most promising predictive model for detecting impending hard drive failure.

Statistical Analysis of Hard Disk Drive Failure

Data helps businesses solve problems, make better decisions, and understand consumers, but a lot of data needs to be stored and kept available to enable these benefits. Hard drive failure is the most common form of data loss, which is one of the most impactful problems that businesses can experience today as simple drive recovery can cost up to $7,500 per drive (Painchaud, 2018). For data centers, keeping multitudes of businesses' data intact for their own operations is crucial. Being able to predict which hard drives are at the highest risk of failure based on understanding of combinations of routine diagnostics test results is an ideal solution to backup and replace failing drives before the data is lost.

Research Question

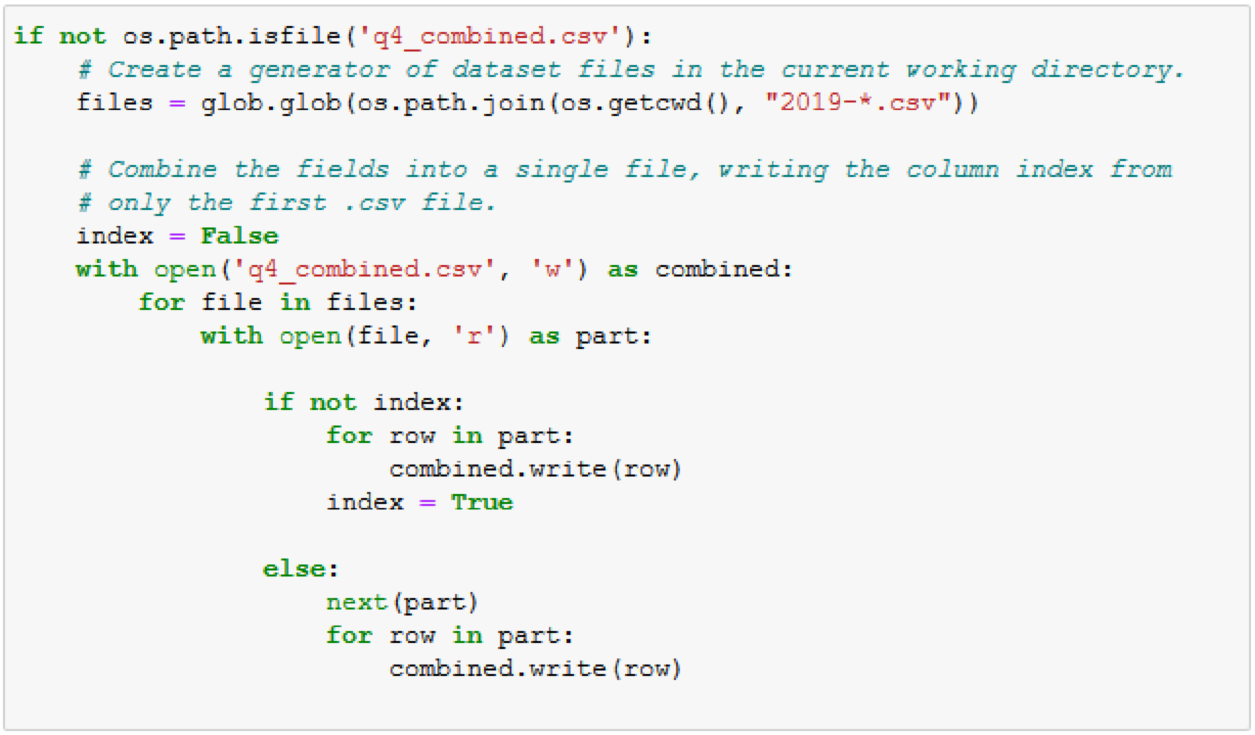
What factors indicate impending hard disk drive (HDD) failure? The null hypothesis is that study factors do not significantly indicate impending hard disk drive failure. The alternative hypothesis is that study factors do significantly indicate impending hard disk drive failure. The study factors examined were sixty-two Self-Monitoring, Analysis and Reporting Technology (SMART) test result values and three values of drive information, namely model number, drive capacity, and failure status. These study factors can easily be gathered daily for each drive in a data center without downtime and then maintained in a database to be leveraged for HDD retirement with as little drive life wasted and as little data lost as possible. Once significance of these factors was shown, logistic regression models, a decision tree, two random forest ensembles, and two Multi-Layer Perceptron (MLP) models were examined to determine a best approach to predicting HDD failure.

Data Collection

The dataset used was Backblaze's 4th quarter data from 2019 (Backblaze, 2020). All of the needed data was contained within the .zip file that Backblaze provides to the public for research and examination, simply downloaded from their web page. The dataset contains .csv files for each day of its corresponding quarter, in this case from 2019-10-01 to 2019-12-31. As an example, the subsection of the dataset for 2019-10-01 contains 115,259 rows of data. However, as this data contains recorded readings from a live data center, the number of hard drives and thus rows, changes daily as failed drives were taken out and new drives were installed. The 129 column attributes of the dataset are Date, Serial Number, Model Number, Capacity, Failure, 62 SMART test result values, and 62 normalized values of the SMART values. The Failure attribute is the dependent variable of this study and is a qualitative binary categorical variable. The Date, Serial Number, and Model are nominal qualitative independent variables. Finally, the SMART value columns are continuous quantitative independent variables. As stated in Backblaze's Hard Drive Data and Stats page (Backblaze, n.d.), this dataset is free for any use as long as Backblaze is cited as the data source, that users accept that they are solely responsible for how the data is used, and that the data cannot be sold to anybody as it publicly available.

Data Extraction and Preparation

In this project, Python and pandas were extensively used for the loading, tidying, and manipulation of the dataset. All of the data handling and analysis were performed in a Jupyter notebook. The dataset is made up of 92 .csv files totaling 3.13GB of text data. As hard drive failure is an extremely rare event, all of these days are needed to be considered together in order to have enough failures to draw conclusions. The project began by combining all parts of the dataset from their .csv files into a single file. Python's glob module was used to create a generator of all dataset files in the project directory and a new dataset file made up of the column indexes from the first file and the rows of each .csv file was written to disk.

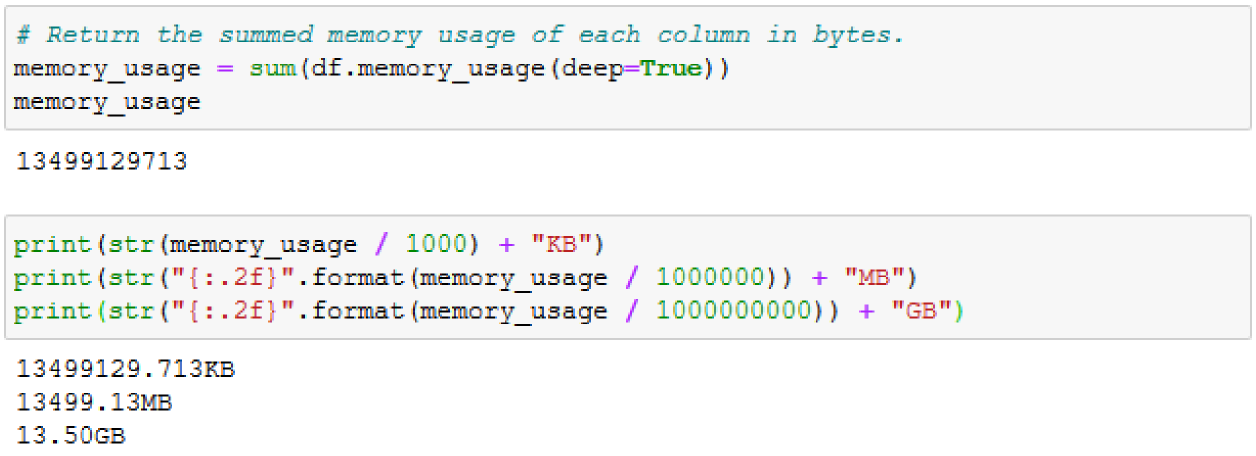


```python
if not os.path.isfile('q4_combined.csv'):
    # Create a generator of dataset files in the current working directory.
    files = glob.glob(os.path.join(os.getcwd(), "2019-*.csv"))

    # Combine the fields into a single file, writing the column index from
    # only the first .csv file.
    index = False
    with open('q4_combined.csv', 'w') as combined:
        for file in files:
            with open(file, 'r') as part:

                if not index:
                    for row in part:
                        combined.write(row)
                    index = True

                else:
                    next(part)
                    for row in part:
                        combined.write(row)
```

Before loading the combined dataset file into working memory, the rows were counted by reading the lines of the file sequentially, finding a total of 10,991,209 rows of data. The combined dataset was loaded into a pandas dataframe as 126 float64 columns, 2 int64 columns, and 3 object (string) columns, for a total of 131 columns of data. The failure column values were counted to find that out of the 10,991,209 hard drive days, there were only 678 failures, which gives a failure rate of 0.006169%.

Weiss (2013) defined the imbalance ratio as the ratio between majority and minority classes with a modestly imbalanced dataset having an imbalance ratio of 10:1, and extremely imbalanced datasets as having an imbalance ratio of 1000:1 or greater (pg. 15). This dataset has an imbalance ratio of approximately 16,210:1 and as such requires very careful cultivation for any predictive model to successfully learn from. The rarity of the positive failure cases was also the reason that the entire 4th quarter dataset was required.

Unfortunately, this combined file required too much memory to load all at once for hardware limitations. It needed 13.5GB for just the data, not including the memory needed for the OS and other software, nor memory for calculations.



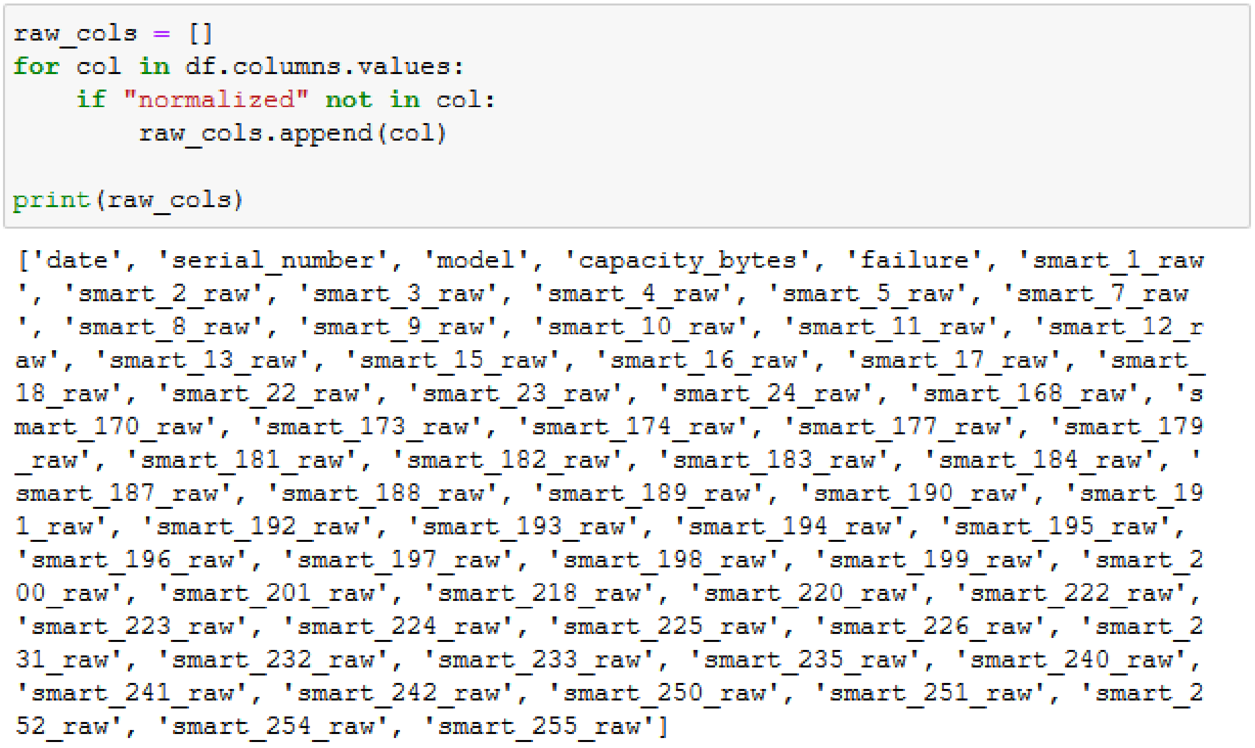
```
# Return the summed memory usage of each column in bytes.
memory_usage = sum(df.memory_usage(deep=True))
memory_usage
```

```
13499129713
```

```
print(str(memory_usage / 1000) + "KB")
print(str("{:.2f}".format(memory_usage / 1000000)) + "MB")
print(str("{:.2f}".format(memory_usage / 1000000000)) + "GB")
```

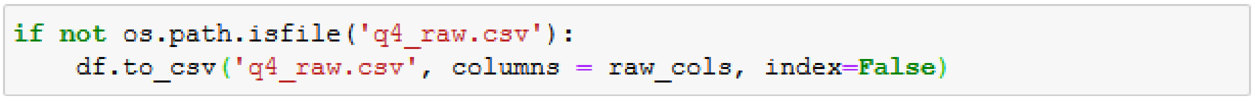
```
13499129.713KB
13499.13MB
13.50GB
```

Memory constraints often affect the available approaches to analyzing datasets. In the base combined form, this dataset was too large to load into memory and perform most analysis algorithms. Out-of-memory approaches could have helped solve this problem but reducing the amount of loaded data allowed for standard in-memory approaches. As this dataset contains both raw and normalized values for all of the SMART values, a simple way to deal with the memory issues was to divide the dataset into a raw form and a normalized form. Two lists were created from the columns, excluding the normalized or raw SMART value columns for each respective file, and were then written to disk. Though preserved, the normalized data was ultimately ignored from that point on, as normalization can be performed after all data tidying and preparation is performed and before predictive models relying on normalization are created.

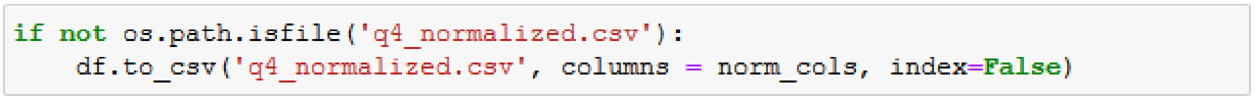


```python
raw_cols = []
for col in df.columns.values:
    if "normalized" not in col:
        raw_cols.append(col)

print(raw_cols)
```

```
['date', 'serial_number', 'model', 'capacity_bytes', 'failure', 'smart_1_raw
', 'smart_2_raw', 'smart_3_raw', 'smart_4_raw', 'smart_5_raw', 'smart_7_raw
', 'smart_8_raw', 'smart_9_raw', 'smart_10_raw', 'smart_11_raw', 'smart_12_r
aw', 'smart_13_raw', 'smart_15_raw', 'smart_16_raw', 'smart_17_raw', 'smart_
18_raw', 'smart_22_raw', 'smart_23_raw', 'smart_24_raw', 'smart_168_raw', 's
mart_170_raw', 'smart_173_raw', 'smart_174_raw', 'smart_177_raw', 'smart_179
_raw', 'smart_181_raw', 'smart_182_raw', 'smart_183_raw', 'smart_184_raw', '
smart_187_raw', 'smart_188_raw', 'smart_189_raw', 'smart_190_raw', 'smart_19
1_raw', 'smart_192_raw', 'smart_193_raw', 'smart_194_raw', 'smart_195_raw',
'smart_196_raw', 'smart_197_raw', 'smart_198_raw', 'smart_199_raw', 'smart_2
00_raw', 'smart_201_raw', 'smart_218_raw', 'smart_220_raw', 'smart_222_raw',
'smart_223_raw', 'smart_224_raw', 'smart_225_raw', 'smart_226_raw', 'smart_2
31_raw', 'smart_232_raw', 'smart_233_raw', 'smart_235_raw', 'smart_240_raw',
'smart_241_raw', 'smart_242_raw', 'smart_250_raw', 'smart_251_raw', 'smart_2
52_raw', 'smart_254_raw', 'smart_255_raw']
```



```python
if not os.path.isfile('q4_raw.csv'):
    df.to_csv('q4_raw.csv', columns = raw_cols, index=False)
```



```python
if not os.path.isfile('q4_normalized.csv'):
    df.to_csv('q4_normalized.csv', columns = norm_cols, index=False)
```

After the dataset was split off into all columns except the normalized columns, the raw value dataset was loaded back into memory for preparation with enough memory leftover for the appropriate analyses. In examining the dataset, all SMART value columns were found to have NaN, pandas’ representation of null, values in them. The model column contains information on the manufacturer of the drive as well as the drive model itself. As SMART implementation varies by manufacturer, that is important information to leverage. Two model values stand out, the “DELLBOSS VD” and the “Seagate SSD” models, pertaining to 60 drives and 96 drives respectively. The first is a RAID controller and contains no SMART values in any row, and the second appears to be a generic catch-all model value for some Seagate drives. Neither have any failures in the dataset and are questionable entries, so these rows were removed.

```
df.loc[(df['model'] == "DELLBOSS VD") &
       (df['failure'] == 1)]
```

```
    date  serial_number  model  capacity_bytes  failure  smart_1_raw  smart_2_raw  smart_3_raw  s

0 rows × 68 columns
```

```
df.loc[(df['model'] == "Seagate SSD") &
       (df['failure'] == 1)]
```

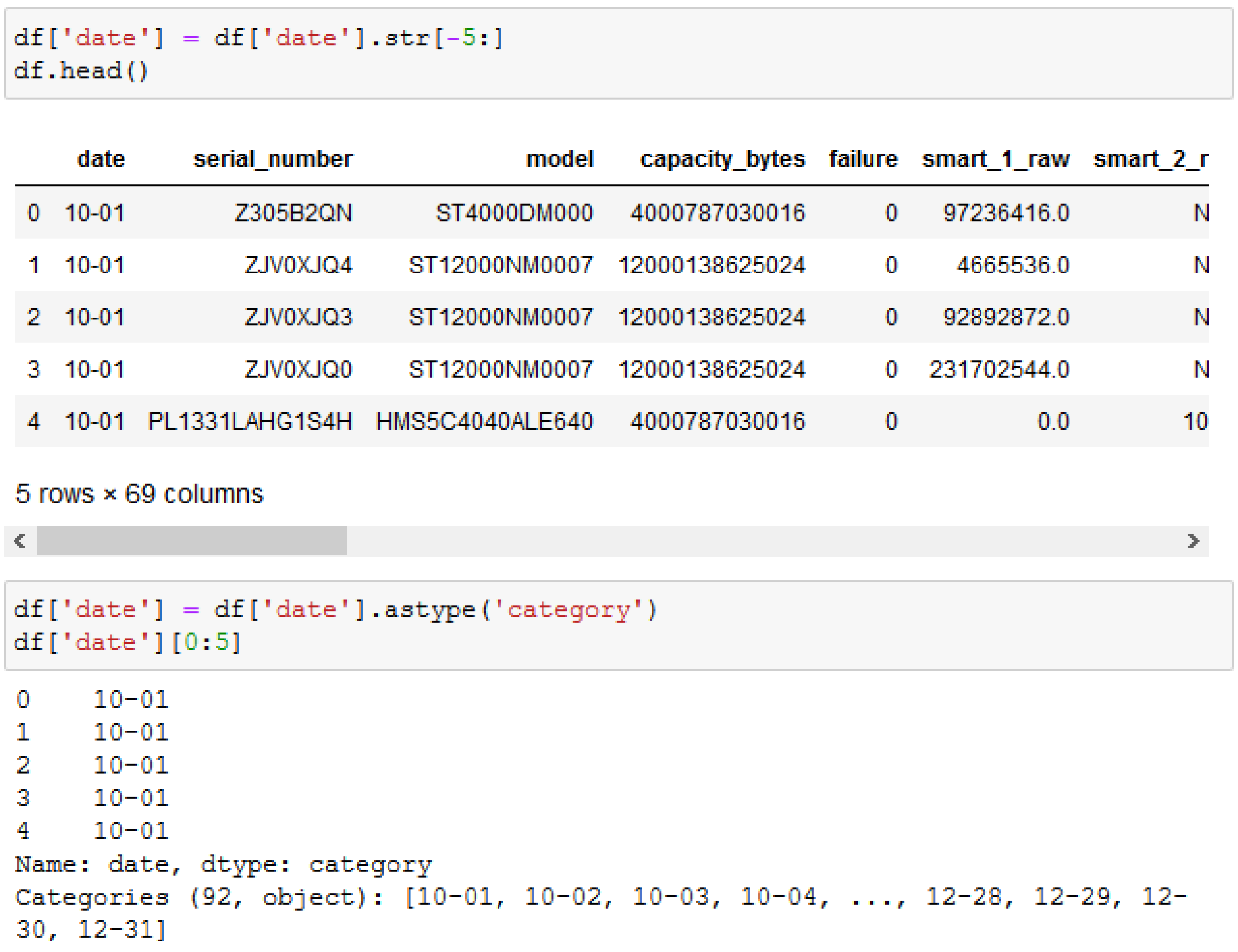
```
    date  serial_number  model  capacity_bytes  failure  smart_1_raw  smart_2_raw  smart_3_raw  s

0 rows × 68 columns
```

```
df.drop(df[(df['model'] == "DELLBOSS VD") | \
           (df['model'] == "Seagate SSD")].index, axis = 0, inplace = True)
```

A dictionary of manufacturer and distinct model value was created for each dataset model value, and the column was divided into a new model column and a manufacturer column.

```
# Change the model column into Manufacturer and Model columns.
df['model_temp'] = df['model']
df['manufacturer'] = ''

df['manufacturer'] = df['model_temp'].map(lambda x: manufacturer_dict[x][0])
df['model'] = df['model_temp'].map(lambda x: manufacturer_dict[x][1])

df.drop(['model_temp'], axis=1, inplace=True)
```

Some columns contain redundant information, like the date column containing characters for the year. To save some memory on disk backups and potentially loading memory, the column was adjusted to only contain the month and day, and its dtype was changed to category.



```
df['date'] = df['date'].str[-5:]
df.head()
```

	date	serial_number	model	capacity_bytes	failure	smart_1_raw	smart_2_r
0	10-01	Z305B2QN	ST4000DM000	4000787030016	0	97236416.0	N
1	10-01	ZJV0XJQ4	ST12000NM0007	12000138625024	0	4665536.0	N
2	10-01	ZJV0XJQ3	ST12000NM0007	12000138625024	0	92892872.0	N
3	10-01	ZJV0XJQ0	ST12000NM0007	12000138625024	0	231702544.0	N
4	10-01	PL1331LAHG1S4H	HMS5C4040ALE640	4000787030016	0	0.0	10

5 rows × 69 columns

```
df['date'] = df['date'].astype('category')
df['date'][0:5]
```

```
0     10-01
1     10-01
2     10-01
3     10-01
4     10-01
Name: date, dtype: category
Categories (92, object): [10-01, 10-02, 10-03, 10-04, ..., 12-28, 12-29, 12-
30, 12-31]
```

The model column was explicitly converted to categorical data and failure was converted to boolean in the same manner.

In examining the capacity_bytes column for values to indicate reason for conversion to categorical data, 1108 drive days distributed across all manufacturers were found to have an error value of -1 rather than their actual capacity. These rows had the potential to be an excellent signal for a failing drive, but ultimately none of the affected drives have a positive failure. Though this was the largest amount of data cut from the dataset at this point, it only makes up an insignificant 0.01% of the data. Given the error and lack of failures, these rows were dropped from the dataset.

```
df.loc[df["capacity_bytes"] == -1]["manufacturer"].value_counts()
```

```
Seagate            759
HGST               299
Toshiba             48
Western Digital      2
Name: manufacturer, dtype: int64
```

```
# Calculate the percentage of the dataset that is affected by this error.
str(np.around(((1008/n_rows) * 100), 2)) + "%"
```

```
'0.01%'
```

```
df.drop(df[(df['capacity_bytes'] == -1)].index, axis = 0, inplace = True)
```

Once the error rows were dropped, the value counts of the capacity_bytes column were examined for further irregularities.

```
df['capacity_bytes'].value_counts()
```

```
12000138625024    4855875
4000787030016     3197457
8001563222016     2309775
14000519643136     232122
500107862016       177166
10000831348736     110993
6001175126016       82595
250059350016         6844
16000900661248       1840
2000398934016         355
1000204886016          91
Name: capacity_bytes, dtype: int64
```

While some capacities are much rarer than others, none are out of place. To save memory and to make the values more readable, the byte values were changed to represent the drives' capacity in terabytes. Peculiarly, the values did not line up with the 1024 kilo base or the rounded 1000 kilo base representations of drive capacity. They were uniform however and were easily rounded to an appropriate value. Once the new capacity column was created, the original was dropped.

```
df['capacity_TB'] = np.around((df['capacity_bytes']/(1000*1000*1000*1000)), \
                              decimals = 2)
df.head()
```

0_raw	smart_251_raw	smart_252_raw	smart_254_raw	smart_255_raw	manufacturer	capacity_TB
NaN	NaN	NaN	NaN	NaN	Seagate	4.0
NaN	NaN	NaN	NaN	NaN	Seagate	12.0
NaN	NaN	NaN	NaN	NaN	Seagate	12.0
NaN	NaN	NaN	NaN	NaN	Seagate	12.0
NaN	NaN	NaN	NaN	NaN	HGST	4.0

```
df['capacity_TB'].value_counts()
```

```
12.00    4855875
4.00     3197457
8.00     2309775
14.00     232122
0.50      177166
10.00     110993
6.00       82595
0.25        6844
16.00       1840
2.00         355
1.00          91
Name: capacity_TB, dtype: int64
```

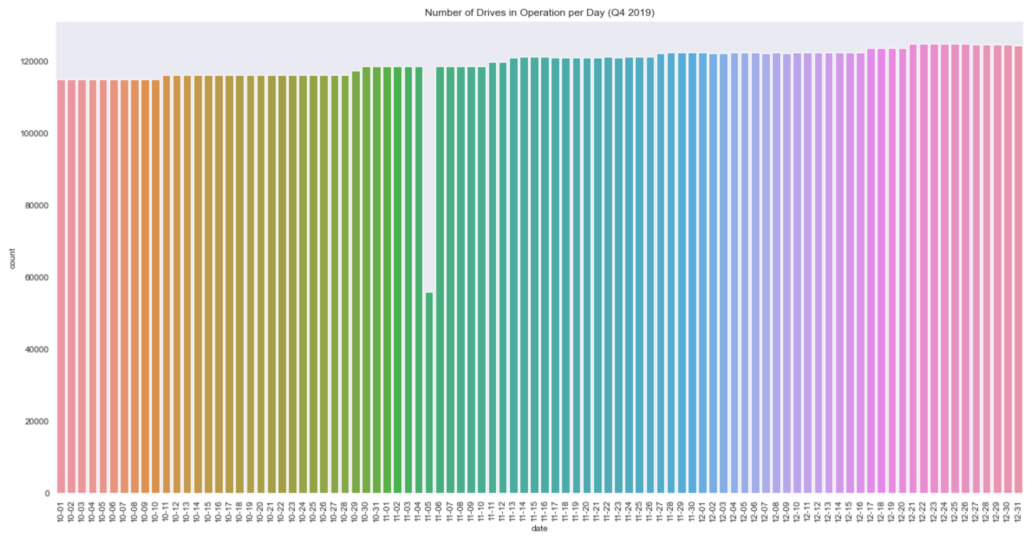
```
df.drop(['capacity_bytes'], axis=1, inplace=True)
df.head()
```

	date	serial_number	model	failure	smart_1_raw	smart_2_raw	smart_3_raw
0	10-01	Z305B2QN	ST4000DM000	False	97236416.0	NaN	0.0
1	10-01	ZJV0XJQ4	ST12000NM0007	False	4665536.0	NaN	0.0
2	10-01	ZJV0XJQ3	ST12000NM0007	False	92892872.0	NaN	0.0
3	10-01	ZJV0XJQ0	ST12000NM0007	False	231702544.0	NaN	0.0
4	10-01	PL1331LAHG1S4H	HMS5C4040ALE640	False	0.0	103.0	436.0

5 rows × 69 columns

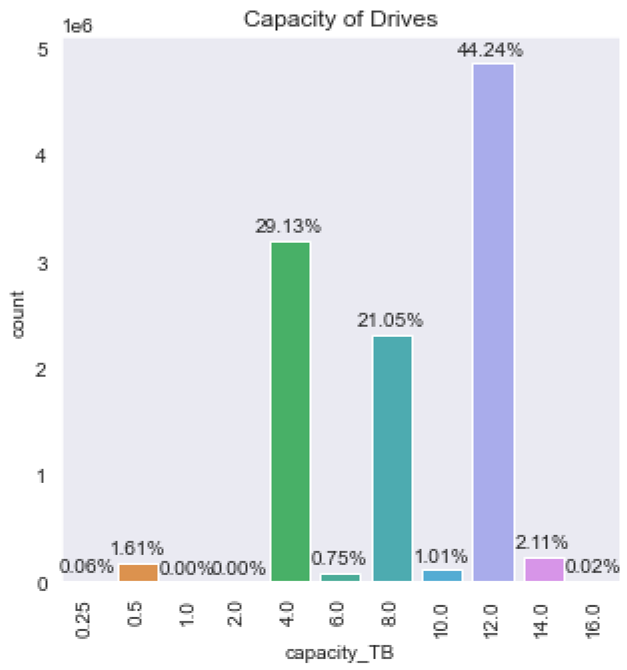
With the general tidying complete, the univariate distributions were examined to gain a better sense of the data. Examining the distribution of the first column, date, showed some sort of testing or operational failure on November 5th.

```
plt.figure(figsize = (20, 10))
plt.title('Number of Drives in Operation per Day (Q4 2019)')
g = sns.countplot(df['date'], data = df)
g.set_xticklabels(g.get_xticklabels(), rotation = 90)
g.figure.savefig("Charts/Date Distribution.png")
g.figure.savefig("Charts/Date Distribution.svg")
```



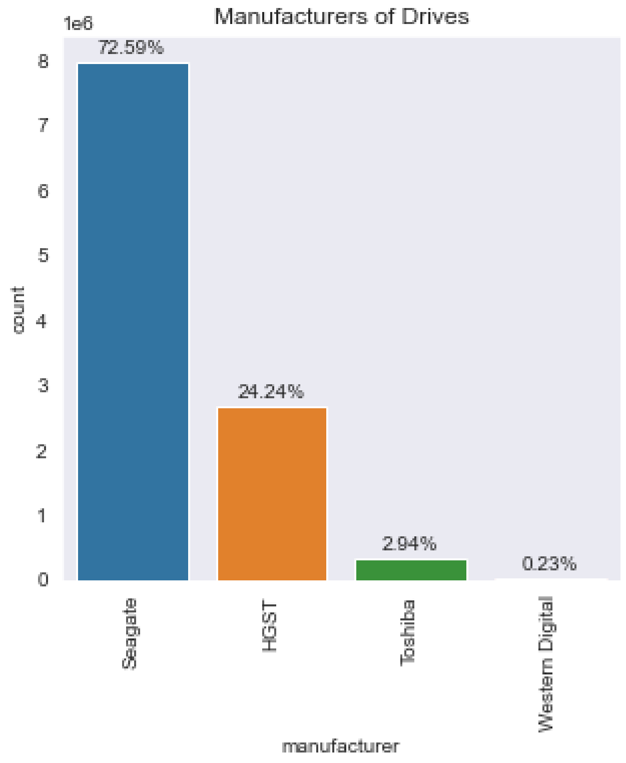
Drive capacities are mostly 4, 8, and 12 TB, likely coinciding with large investments in new drives for the datacenter and possibly alongside the price lowering of specific models.

```
plt.figure(figsize = (5, 5))
plt.title('Capacity of Drives')
g = sns.countplot(df['capacity_TB'], data = df)
g.set_xticklabels(g.get_xticklabels(), rotation = 90)
for p in g.patches:
    percentage = "{0:.2f}".format((p.get_height() / n_rows) * 100) + "%"
    g.annotate(percentage, (p.get_x() + p.get_width() / 2., p.get_height()),
    ha = 'center', va = 'center', xytext = (0, 7), textcoords = 'offset points'

g.figure.savefig("Charts/Capacity Distribution.svg")
g.figure.savefig("Charts/Capacity Distribution.svg")
```



The manufacturer of the most drives in this dataset is Seagate at 72.59%. HGST is the second highest at 24.24%. Western Digital is the least represented manufacturer in the dataset with only 0.23%, but as HGST was acquired by Western Digital in 2012 (Sanders, 2018), the drives in this dataset are likely be quite similar between the two manufacturers given the seven-year timespan between then and the time of dataset recording and creation. Finally, Toshiba is the other manufacturer, with 2.94% of the dataset. This amount is quite low and potentially made it difficult to accurately predict their drives in comparison.

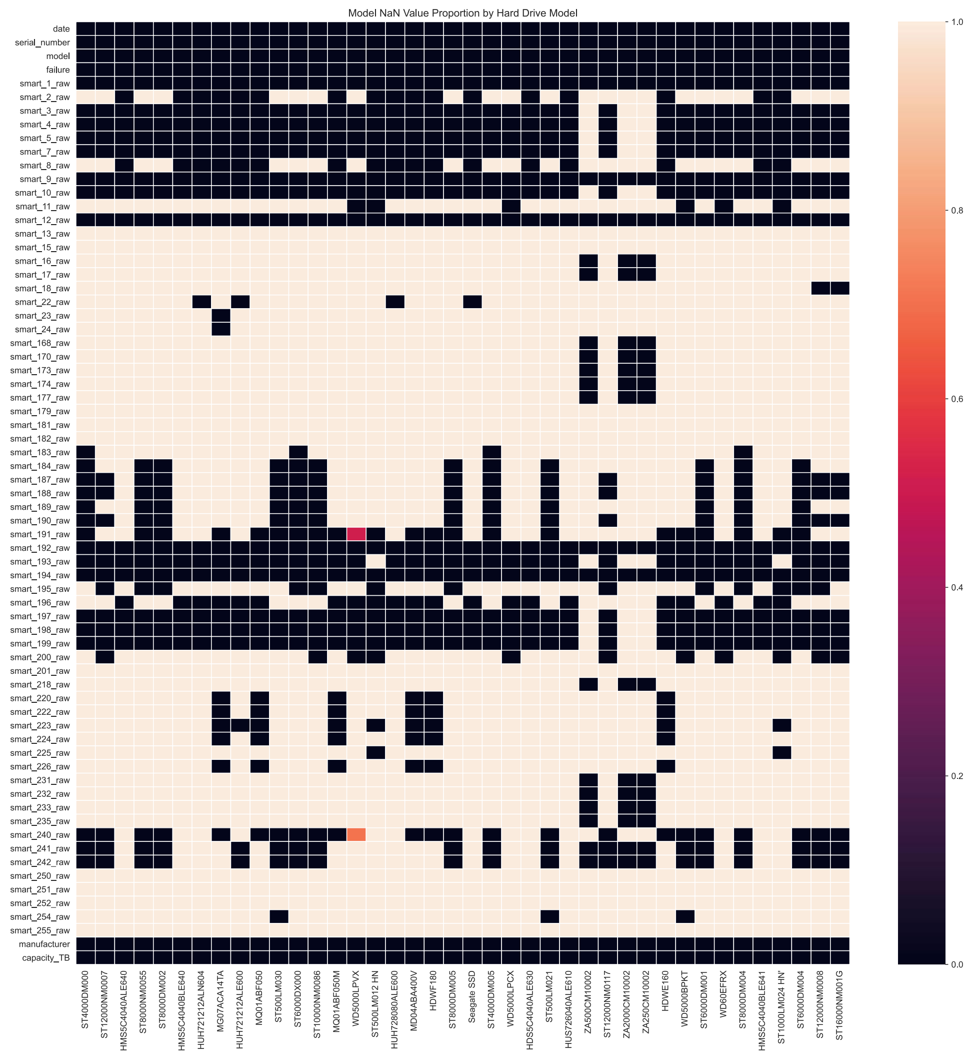
```
plt.figure(figsize = (5, 5))
plt.title('Manufacturers of Drives')
g = sns.countplot(df['manufacturer'], data = df)
g.set_xticklabels(g.get_xticklabels(), rotation = 90)
for p in g.patches:
    percentage = "{0:.2f}".format((p.get_height() / n_rows) * 100) + "%"
    g.annotate(percentage, (p.get_x() + p.get_width() / 2., p.get_height()),
    ha = 'center', va = 'center', xytext = (0, 7), textcoords = 'offset points'

g.figure.savefig("Charts/Manufacturer Distribution.svg")
g.figure.savefig("Charts/Manufacturer Distribution.png")
```



The SMART values vary greatly from the number of different types of drives that exist in this dataset. Before the columns could be graphed appropriately, the NaN values needed to be examined and then interpolated or filled in with summary statistics. The proportion of the models' missing values for each column was calculated and then graphed as a heatmap.

```
model_nan_percent_df = pd.DataFrame()
for model in df['model'].unique():
    model_nan_percent_df[model] = (df.loc[df['model'] == model].isna().sum())\
        /len(df.loc[df['model'] == model])
```

```
plt.figure(figsize = (20, 20))
plt.title('Model NaN Value Proportion by Hard Drive Model')
g = sns.heatmap(model_nan_percent_df, linewidths=0.2)
g.figure.savefig("Charts/Model NaN Heatmap.svg")
g.figure.savefig("Charts/Model NaN Heatmap.png")
```



For specific numbers on the NaN values in each column, a new dataframe was created from the count output row of pandas' describe function, which counts the non-NaN values of the column. The percentage of non-NaN values was then added as an additional column in the new dataframe. To assist in quick recognition of the information, a styling function was created and then applied to the percentage column.

```
count_df = pd.DataFrame()
count_df['count'] = description_df.iloc[0]
count_df
```

```
# Pandas styling function
def highlight_count_nans1(val):
    if val >= 66.6:
        color = 'green'
    elif val >= 33.3 and val < 66.6:
        color = 'yellow'
    else:
        color = 'red'

    return 'color: %s' % color
```

```
count_df['perc_not_nan'] = (count_df['count'] / n_rows) * 100
count_df.style.applymap(highlight_count_nans1, subset = ['perc_not_nan'])
```

	count	perc_not_nan
smart_1_raw	1.09751e+07	100
smart_2_raw	3.02845e+06	27.5938
smart_3_raw	1.09663e+07	99.9199
smart_4_raw	1.09663e+07	99.9199
smart_5_raw	1.09663e+07	99.9199
smart_7_raw	1.09663e+07	99.9199
smart_8_raw	3.02845e+06	27.5938

Between the heatmap and the information in the count_df dataframe, many of the useless columns of the dataset became apparent. From this dataframe, a list was created for the completely empty columns and then another for the columns with less than 80% of filled values.

```
empty_columns = []
columns_to_examine = []

for row in count_df.iterrows():
    if row[1][0] == 0.0:
        empty_columns.append(row[0])

    elif row[1][0] < (0.8 * n_rows):
        columns_to_examine.append(row[0])
```

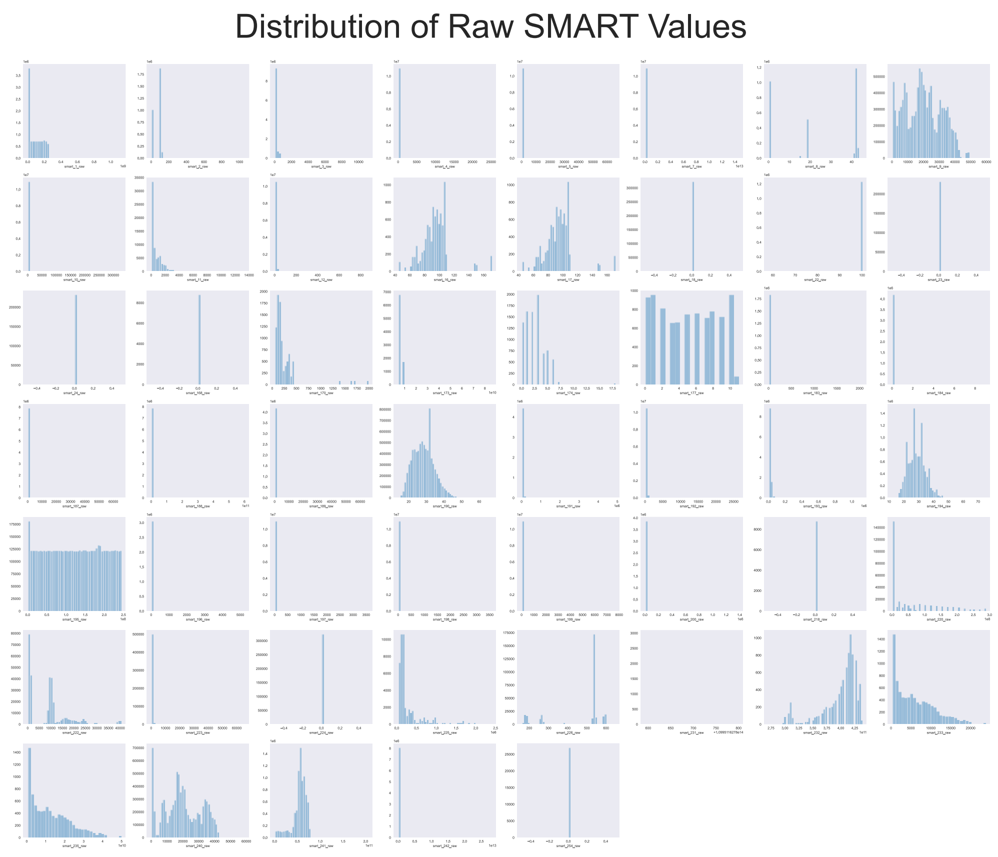
Using this list, all columns made up of exclusively NaN values were dropped.

```
before_mem = df.memory_usage(deep=True).sum()
df.drop(empty_columns, axis=1, inplace=True)
after_mem = df.memory_usage(deep=True).sum()
memory_saved = before_mem - after_mem
print("Memory saved on empty column removal: " + \
      str(np.around((memory_saved / 1024 ** 2), 2)) + "MB")
```

```
Memory saved on empty column removal: 837.33MB
```

In order to determine the appropriate summary statistic to use for filling in the NaN values for each SMART column, the non-NaN values were graphed to examine their distribution.

```
fig, axes = plt.subplots(7, 8, figsize = (50, 40))

row = 0
col = 0
for df_col in ['smart_1_raw', 'smart_2_raw', 'smart_3_raw', 'smart_4_raw',
               'smart_5_raw', 'smart_7_raw', 'smart_8_raw', 'smart_9_raw',
               'smart_10_raw', 'smart_11_raw', 'smart_12_raw', 'smart_16_raw',
               'smart_17_raw', 'smart_18_raw', 'smart_22_raw', 'smart_23_raw',
               'smart_24_raw', 'smart_168_raw', 'smart_170_raw', 'smart_173_raw',
               'smart_174_raw', 'smart_177_raw', 'smart_183_raw', 'smart_184_raw',
               'smart_187_raw', 'smart_188_raw', 'smart_189_raw', 'smart_190_raw',
               'smart_191_raw', 'smart_192_raw', 'smart_193_raw', 'smart_194_raw',
               'smart_195_raw', 'smart_196_raw', 'smart_197_raw', 'smart_198_raw',
               'smart_199_raw', 'smart_200_raw', 'smart_218_raw', 'smart_220_raw',
               'smart_222_raw', 'smart_223_raw', 'smart_224_raw', 'smart_225_raw',
               'smart_226_raw', 'smart_231_raw', 'smart_232_raw', 'smart_233_raw',
               'smart_235_raw', 'smart_240_raw', 'smart_241_raw', 'smart_242_raw',
               'smart_254_raw']:

    if col == 8:
        row += 1
        col = 0

    sns.distplot(df[df_col], ax = axes[row, col], \
                 kde = False, norm_hist = False)

    col += 1


axes[6, 5].set_axis_off()
axes[6, 6].set_axis_off()
axes[6, 7].set_axis_off()

plt.subplots_adjust(top = 0.90)
fig.suptitle("Distribution of Raw SMART Values", fontsize = 96, y = 0.95)
fig.savefig("Charts/SMART Distributions.svg")
fig.savefig("Charts/SMART Distributions.png")
```



The process of filling in the NaN values began with ordering the columns in order from the lowest amount of missing values to the highest amount. Doing so revealed groups of columns with the same amount of missing values made up of the same models of drives. The first five mostly complete columns all had two NaN values, which were the result of two rows that had no raw smart values at all. Both drives failed, making them quite important for predicting future failure. However, the lack of data made them useless for predicting future failure in their current form.

The most likely scenario is that both drives failed just before the diagnostics were collected. As such, these two rows were deleted and their associated row for the date before their marked failures were updated to have failed that day instead. This kept two of the extremely imbalanced minority class rows in the dataset while also making the least amount of assumptions. Presumably, the previous day for the same drives as determined by serial_number and date had the most pertinent information in their SMART values for the failure instances. Once the previous date rows for those drives had their failure value changed, the empty rows were dropped.

```
df.loc[df['smart_1_raw'].isnull() & df['smart_192_raw'].isnull() & \
       df['smart_9_raw'].isnull() & df['smart_12_raw'].isnull() & \
       df['smart_194_raw'].isnull()]
```

	date	serial_number	model	failure	smart_1_raw	smart_2_raw	smart_3_raw
4632946	11-10	ZJV00DR4	ST12000NM0007	1	NaN	NaN	NaN
4797700	11-11	ZHZ3M097	ST12000NM0008	1	NaN	NaN	NaN

2 rows × 59 columns

```
df.loc[(df['serial_number'] == 'ZJV00DR4') & (df['date'] == '11-09')]
```

	date	serial_number	model	failure	smart_1_raw	smart_2_raw	smart_3_raw
4514189	11-09	ZJV00DR4	ST12000NM0007	0	118859320.0	NaN	0.0

1 rows × 59 columns

```
df.at[4514189, 'failure'] = 1
df.iloc[4514189]
df.loc[(df['serial_number'] == 'ZHZ3M097') & (df['date'] == '11-10')]
```

	date	serial_number	model	failure	smart_1_raw	smart_2_raw	smart_3_raw
4678156	11-10	ZHZ3M097	ST12000NM0008	0	196597768.0	NaN	0.0

1 rows × 59 columns

```
df.at[4678156, 'failure'] = 1
df.iloc[4678156]
```

```
df.drop(df.index[[4797700, 4632946]], inplace = True)
```

The second grouping of columns smart_3_raw, smart_4_raw, smart_5_raw, smart_7_raw, smart_10_raw, smart_197_raw, smart_198_raw, and smart_199_raw all had 8794 values missing and were found to be the same 8794 rows. These rows were made up of 3 capacity variations of the same model made by Seagate.

```
df_8794 = df.loc[df['smart_3_raw'].isnull() & df['smart_4_raw'].isnull() & \
                 df['smart_5_raw'].isnull() & df['smart_7_raw'].isnull() & \
                 df['smart_10_raw'].isnull() & df['smart_197_raw'].isnull() & \
                 df['smart_198_raw'].isnull() & df['smart_199_raw'].isnull()]
```

```
df_8794['manufacturer'].value_counts()
```

```
Seagate    8792
Name: manufacturer, dtype: int64
```

```
df_8794['model'].value_counts()
```

```
ZA250CM10002     6844
ZA500CM10002     1593
ZA2000CM10002     355
Name: model, dtype: int64
```

```
df_8794['capacity_TB'].value_counts()
```

```
0.25    6844
0.50    1593
2.00     355
Name: capacity_TB, dtype: int64
```

```
df_8794['failure'].value_counts()
```

```
0    8792
Name: failure, dtype: int64
```

The means and medians, based on the non-missing column distribution, from the same manufacturer and the models' respective capacity_TB categories if available, were used to fill the SMART columns' missing values. For each of the columns smart_3_raw, smart_4_raw, smart_5_raw, smart_7_raw, smart_197_raw, smart_198_raw, and smart_199_raw the availability of each respective drive capacity value within the matching manufacturer value subset was checked. In each case, only the 0.50 capacity_TB value had matching drives to use for a more specialized summary statistic.

```
df.loc[(df['manufacturer'] == "Seagate") & (df['smart_3_raw'].notnull()) & \
       (df['capacity_TB'] == 0.25)]['smart_3_raw']
```

```
Series([], Name: smart_3_raw, dtype: float64)
```

```
df.loc[(df['manufacturer'] == "Seagate") & (df['smart_3_raw'].notnull()) & \
       (df['capacity_TB'] == 0.50)]['smart_3_raw']
```

```
134            0.0
246         2044.0
714            0.0
1006        1989.0
1502        1801.0
             ...
10974512       0.0
10974685       0.0
10974769       0.0
10974840       0.0
10974960       0.0
Name: smart_3_raw, Length: 71163, dtype: float64
```

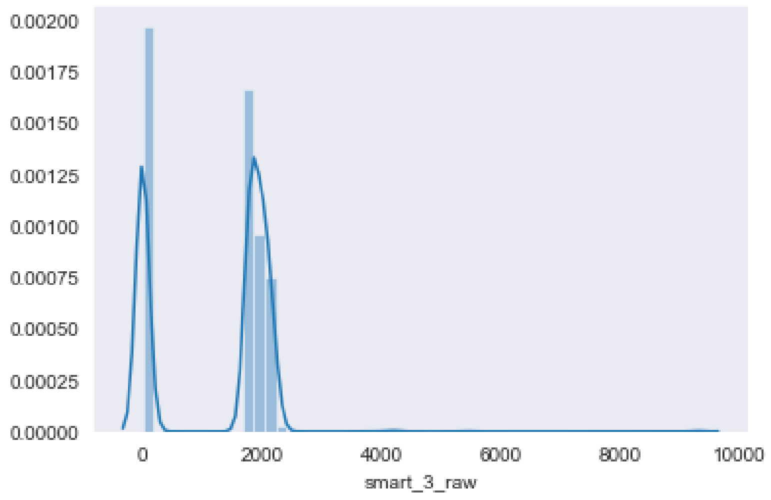
```
df.loc[(df['manufacturer'] == "Seagate") & (df['smart_3_raw'].notnull()) & \
       (df['capacity_TB'] == 2.00)]['smart_3_raw']
```

```
Series([], Name: smart_3_raw, dtype: float64)
```

With this knowledge, the distributions of the available capacity_TB value and of the general matching manufacturer, Seagate, were graphed for each column to determine whether the mean or median should be used to fill the missing values for the two subsets of rows.

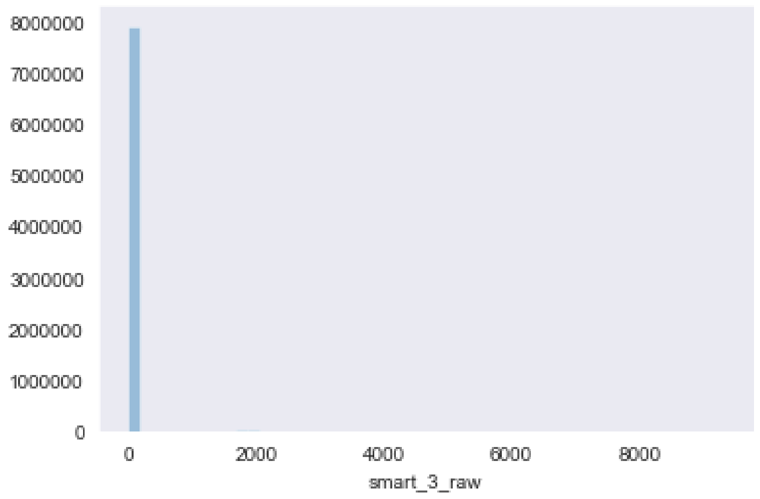
```
sns.distplot(df.loc[(df['manufacturer'] == "Seagate") & \
                    (df['smart_3_raw'].notnull()) & \
                    (df['capacity_TB'] == 0.50)]['smart_3_raw'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x16be98c0448>
```



```
sns.distplot(df.loc[(df['manufacturer'] == "Seagate") & \
                    (df['smart_3_raw'].notnull())]['smart_3_raw'], kde = False)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x16ba74e4dc8>
```



In the cases of smart_3_raw, smart_5_raw, smart_7_raw, smart_197_raw, smart_198_raw, and smart_199_raw, the two distributions were quite non-normal, and the median was chosen to fill missing values with. The two distributions for smart_4_raw were much closer to normal distributions and the mean was chosen for it.

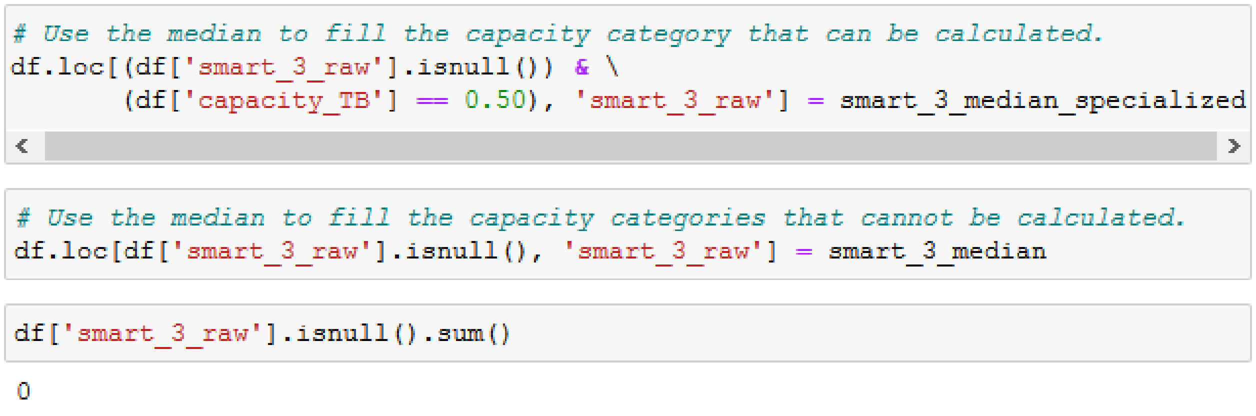
```
smart_3_median_specialized = df.loc[(df['manufacturer'] == "Seagate") & \
                         (df['smart_3_raw'].notnull()) & \
                         (df['capacity_TB'] == 0.50)]['smart_3_raw'].median()
smart_3_median_specialized
```

```
1816.0
```

```
smart_3_median = df.loc[(df['manufacturer'] == "Seagate") & \
                         (df['smart_3_raw'].notnull())]['smart_3_raw'].median()
smart_3_median
```

```
0.0
```

These summary statistics values were then used to fill the missing values in each column, resulting in 0 NaN values.



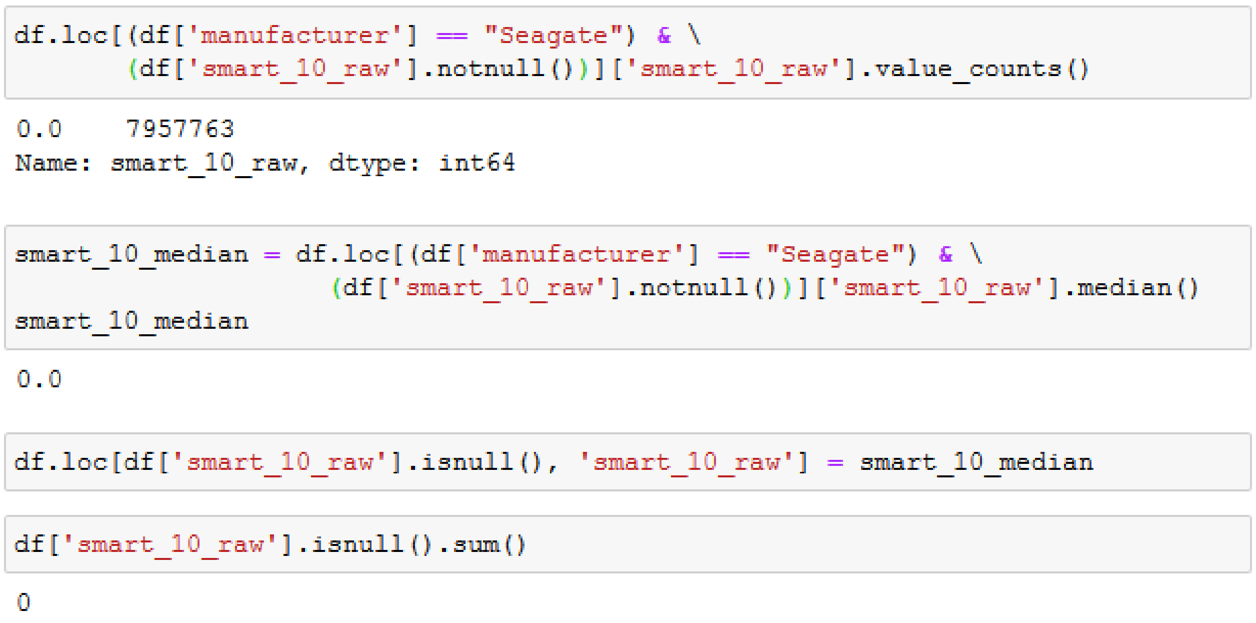
```
# Use the median to fill the capacity category that can be calculated.
df.loc[(df['smart_3_raw'].isnull()) & \
       (df['capacity_TB'] == 0.50), 'smart_3_raw'] = smart_3_median_specialized
```

```
# Use the median to fill the capacity categories that cannot be calculated.
df.loc[df['smart_3_raw'].isnull(), 'smart_3_raw'] = smart_3_median
```

```
df['smart_3_raw'].isnull().sum()
```

```
0
```

For the smart_10_raw missing values, the median of the same manufacturer drives was used for all drive capacity_TB categories as all Seagate drives only had 0.0 as their value for the column. This means that a subset could not be given a more specialized summary statistic for filling its missing values as in the other columns in the same group.



```
df.loc[(df['manufacturer'] == "Seagate") & \
       (df['smart_10_raw'].notnull())]['smart_10_raw'].value_counts()
```

```
0.0    7957763
Name: smart_10_raw, dtype: int64
```

```
smart_10_median = df.loc[(df['manufacturer'] == "Seagate") & \
                         (df['smart_10_raw'].notnull())]['smart_10_raw'].median()
smart_10_median
```

```
0.0
```

```
df.loc[df['smart_10_raw'].isnull(), 'smart_10_raw'] = smart_10_median
```

```
df['smart_10_raw'].isnull().sum()
```

```
0
```

The smart_193_raw column was a different problem than the last group of columns. This group had 53985 rows with NaN values, which was still low enough in this large dataset to fill values without major effects on the statistics of the data. An important note here is that some manufacturers use different SMART attributes to represent the same information. Most Seagate and some Western Digital and Hitachi drives use 225 rather than 193 to store the Load/Unload Cycle Count value (Acronis, Knowledge Base 9128; Acronis, Knowledge Base 9152). In this dataset no row had both 193 and 225 values.

```
df.loc[(df['smart_193_raw'].notnull()) & \
       (df['smart_225_raw'].notnull())][['smart_193_raw', 'smart_225_raw']]
```

smart_193_raw	smart_225_raw

```
df_193 = df.loc[df['smart_193_raw'].isnull()]
```

The only rows that did not have either value were the exact same rows that made up the last group of columns.

```
df_193.loc[(df_193['smart_193_raw'].isnull()) & \
           (df_193['smart_225_raw'].isnull())]['model'].value_counts()
```

```
ZA250CM10002       6844
ZA500CM10002       1593
ZA2000CM10002       355
```

The 45193 other rows were filled by combining the two columns that represent the same information into a new smart_193_225 column.

```
df['smart_193_225'] = df['smart_193_raw']
```

```
df['smart_193_225'].fillna(df['smart_225_raw'], inplace = True)
```

```
df[['smart_193_raw', 'smart_225_raw', 'smart_193_225']].isna().sum()
```

```
smart_193_raw       53985
smart_225_raw    10929918
smart_193_225        8792
dtype: int64
```

```
df.drop(['smart_193_raw', 'smart_225_raw'], axis=1, inplace=True)
```

As the 8792 rows missing from the new smart_193_225 column are the same rows from the first grouping of NaN columns, the same approach was taken to fill their missing values. As the distributions were not normal, the median was chosen for this column. The median of the available 0.50 capacity_TB drives was calculated and the median of all drives of the same manufacturer was calculated for the drives of other capacities.

```
df.loc[(df['manufacturer'] == "Seagate") & \
       (df['smart_193_225'].notnull()) & \
       (df['capacity_TB'] == 0.25)]['smart_193_225']
```

```
Series([], Name: smart_193_225, dtype: float64)
```

```
df.loc[(df['manufacturer'] == "Seagate") & \
       (df['smart_193_225'].notnull()) & \
       (df['capacity_TB'] == 0.50)]['smart_193_225']
```

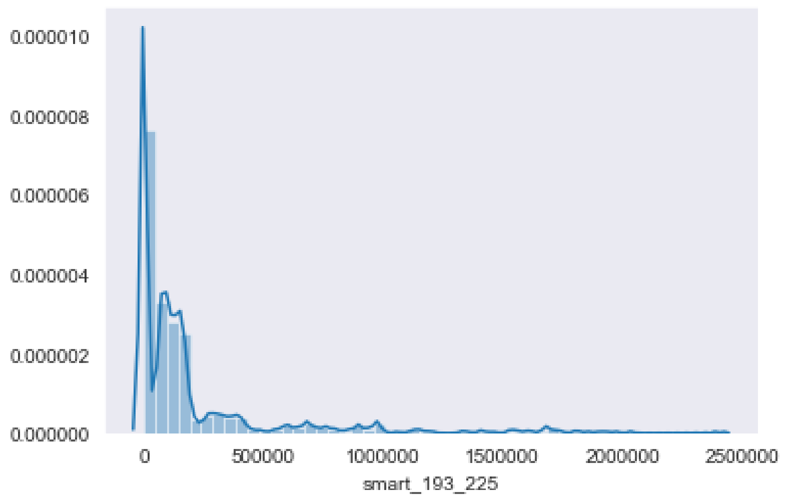
```
134              266.0
246           310513.0
714               62.0
1006           72805.0
1502           72944.0
                 ...
10974512         651.0
10974685          27.0
10974769          27.0
10974840          19.0
10974960          13.0
Name: smart_193_225, Length: 71163, dtype: float64
```

```
df.loc[(df['manufacturer'] == "Seagate") & \
       (df['smart_193_225'].notnull()) & \
       (df['capacity_TB'] == 2.00)]['smart_193_225']
```

```
Series([], Name: smart_193_225, dtype: float64)
```

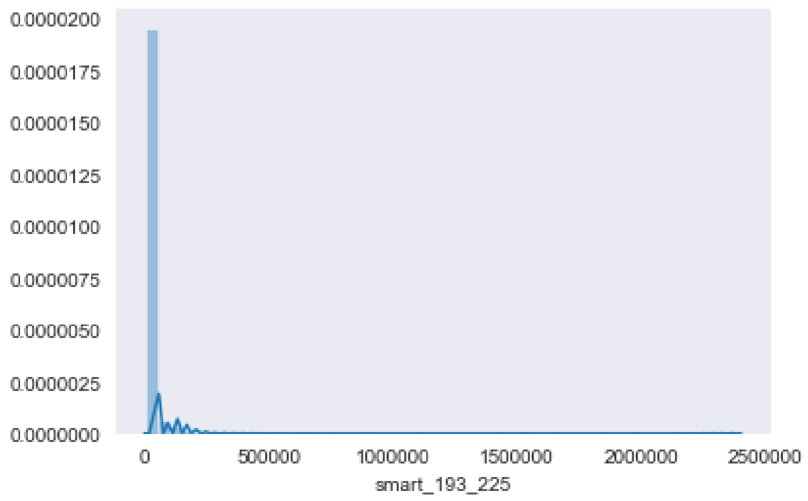
```
sns.distplot(df.loc[(df['manufacturer'] == "Seagate") & \
                    (df['smart_193_225'].notnull()) & \
                    (df['capacity_TB'] == 0.50)]['smart_193_225'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x16c0ba5a488>
```



```
sns.distplot(df.loc[(df['manufacturer'] == "Seagate") & \
                    (df['smart_193_225'].notnull())]['smart_193_225'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x16ab62d0a48>
```



```
smart_193_225_median_specialized = df.loc[(df['manufacturer'] == "Seagate") &\
                        (df['smart_193_225'].notnull()) & \
                        (df['capacity_TB'] == 0.50)]['smart_193_225'].median()
smart_193_225_median_specialized
```

```
90136.0
```

```
smart_193_225_median = df.loc[(df['manufacturer'] == "Seagate") & \
                              (df['smart_193_225'].notnull())]['smart_193_225'].median()
smart_193_225_median
```

```
3694.0
```

The missing values were then filled with the medians for each subset.

```
# Use the median to fill the capacity category that can be calculated.
df.loc[(df['smart_193_225'].isnull()) & \
          (df['capacity_TB'] == 0.50), 'smart_193_225'] = \
           smart_193_225_median_specialized
```

```
# Use the median to fill the capacity categories that cannot be calculated.
df.loc[df['smart_193_225'].isnull(), 'smart_193_225'] = smart_193_225_median
```

```
df['smart_193_225'].isnull().sum()
```

```
0
```

The remaining columns with NaN values each had over 2 million missing values. The columns smart_240_raw, smart_241_raw, smart_242_raw, smart_187_raw, smart_188_raw, and smart_190_raw had over 70% of their values filled. This amount was the decided cutoff for filling missing values with summary statistics.

Notably, none of the HGST drives had a value for the smart_240_raw column, and none of the Toshiba drives had values for the smart_241_raw and smart_242_raw columns. As such, specialized summary statistics could not be calculated for each subset of drives by manufacturer as in the previous cases. The means of all available rows for each column were used to fill the missing values.

```
df.loc[df['smart_240_raw'].notnull()]['manufacturer'].value_counts()
```

```
Seagate            7912570
Toshiba             322722
Western Digital       6567
HGST                     0
Name: manufacturer, dtype: int64
```

```
df.loc[df['smart_241_raw'].notnull()]['manufacturer'].value_counts()
```

```
Seagate            7921362
HGST                143520
Western Digital        912
Toshiba                  0
Name: manufacturer, dtype: int64
```

```
df.loc[df['smart_242_raw'].notnull()]['manufacturer'].value_counts()
```

```
Seagate            7921362
HGST                143520
Western Digital        912
Toshiba                  0
Name: manufacturer, dtype: int64
```

```
smart_240_mean = df.loc[df['smart_240_raw'].notnull()]['smart_240_raw'].mean()
smart_240_mean
```

```
19479.888113349185
```

```
df['smart_240_raw'].fillna(smart_240_mean, inplace = True)
```

```
df['smart_240_raw'].isnull().sum()
```

```
0
```

```
smart_241_mean = df.loc[df['smart_241_raw'].notnull()]['smart_241_raw'].mean()
smart_241_mean
```

```
53260820637.912384
```

```
df['smart_241_raw'].fillna(smart_241_mean, inplace = True)
```

```
df['smart_241_raw'].isnull().sum()
```

```
0
```

```
smart_242_mean = df.loc[df['smart_242_raw'].notnull()]['smart_242_raw'].mean()
smart_242_mean
```

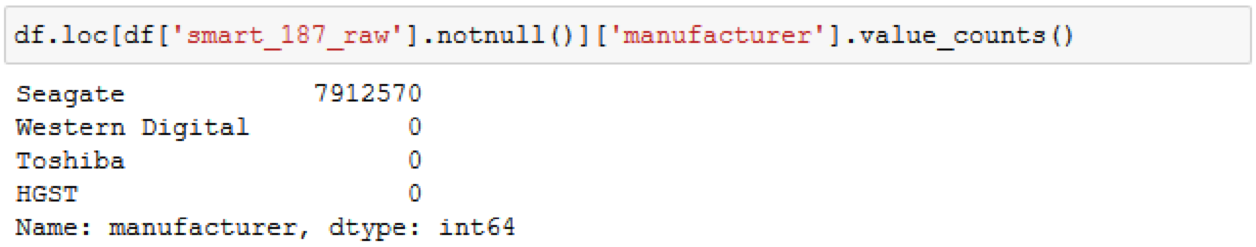
```
140094512785.44415
```

```
df['smart_242_raw'].fillna(smart_242_mean, inplace = True)
```

```
df['smart_242_raw'].isnull().sum()
```

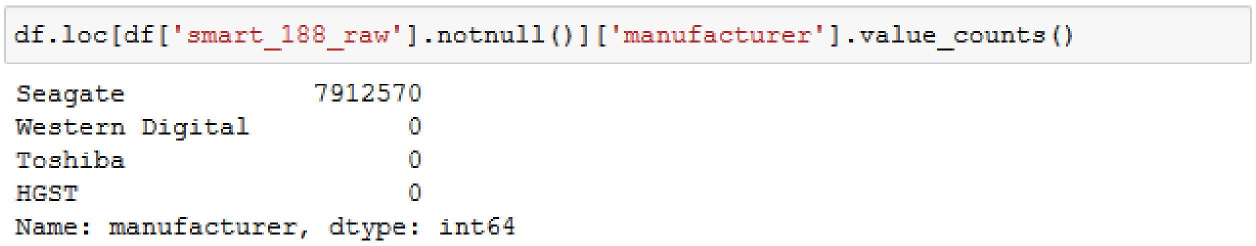
```
0
```

The group of the smart_187_raw, smart_188_raw, and smart_190_raw columns were divided by manufacturer, with all Seagate drives having the values and none of the other drive manufacturers having the values.



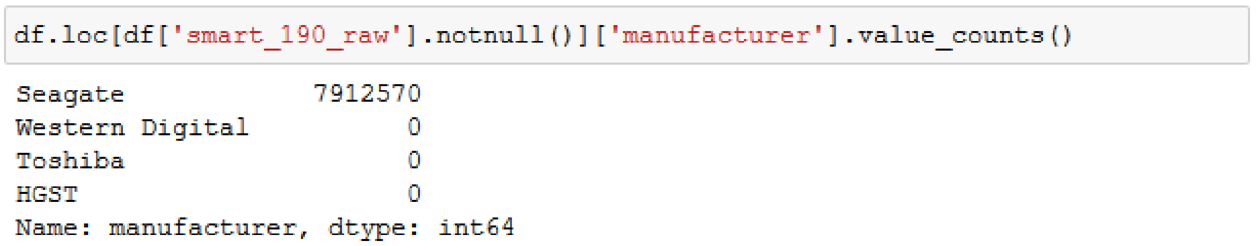
```
df.loc[df['smart_187_raw'].notnull()]['manufacturer'].value_counts()
```

```
Seagate            7912570
Western Digital          0
Toshiba                  0
HGST                     0
Name: manufacturer, dtype: int64
```



```
df.loc[df['smart_188_raw'].notnull()]['manufacturer'].value_counts()
```

```
Seagate            7912570
Western Digital          0
Toshiba                  0
HGST                     0
Name: manufacturer, dtype: int64
```



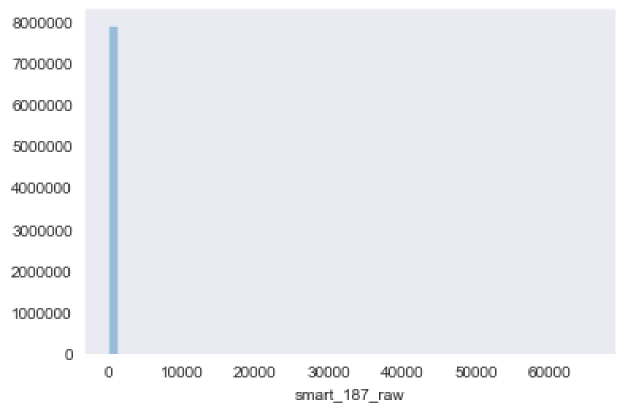
```
df.loc[df['smart_190_raw'].notnull()]['manufacturer'].value_counts()
```

```
Seagate            7912570
Western Digital          0
Toshiba                  0
HGST                     0
Name: manufacturer, dtype: int64
```

Given the split across manufacturers, specialized summary statistics could not be calculated for each subset of drives by manufacturer as in the earlier cases. Based on the distributions of the 3 columns, the median was chosen to fill missing values for smart_187_raw and smart_188_raw, while the mean was selected for the smart_190_raw column.

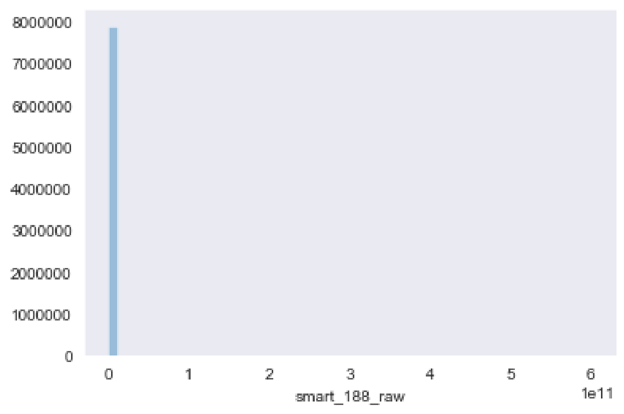
```
sns.distplot(df.loc[df['smart_187_raw'].notnull()]['smart_187_raw'], \
             kde = False)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x16ab6c04988>
```



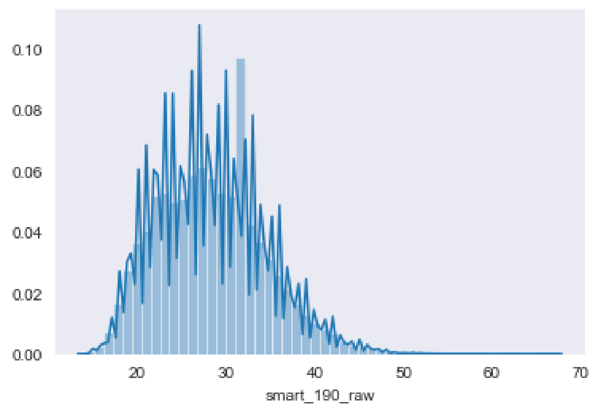
```
sns.distplot(df.loc[df['smart_188_raw'].notnull()]['smart_188_raw'], \
             kde = False)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x16c0bdc4708>
```



```
sns.distplot(df.loc[df['smart_190_raw'].notnull()]['smart_190_raw'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x16c1069ec08>
```



```
smart_187_median = df.loc[df['smart_187_raw'].notnull()]['smart_187_raw'].media
smart_187_median
```

```
0.0
```

```
df['smart_187_raw'].fillna(smart_187_median, inplace = True)
```

```
df['smart_187_raw'].isnull().sum()
```

```
0
```

```
smart_188_median = df.loc[df['smart_188_raw'].notnull()]['smart_188_raw'].media
smart_188_median
```

```
0.0
```

```
df['smart_188_raw'].fillna(smart_188_median, inplace = True)
```

```
df['smart_188_raw'].isnull().sum()
```

```
0
```

```
smart_190_mean = df.loc[df['smart_190_raw'].notnull()]['smart_190_raw'].mean()
smart_190_mean
```

```
28.227229585330683
```

```
df['smart_190_raw'].fillna(smart_190_mean, inplace = True)
```

```
df['smart_190_raw'].isnull().sum()
```

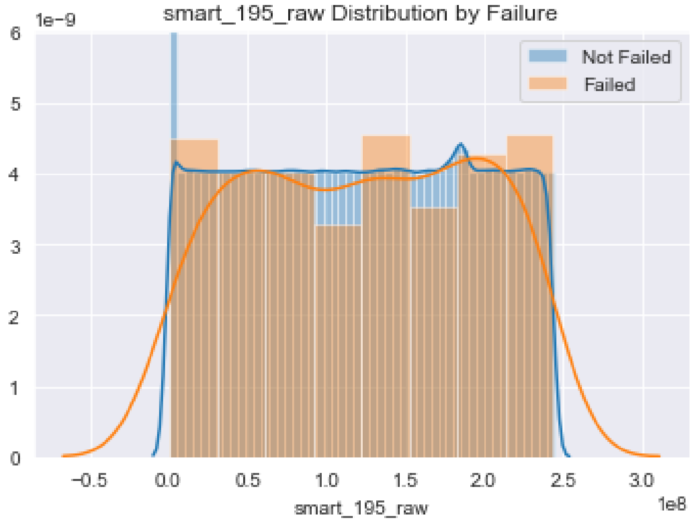
```
0
```

The remaining 32 columns have over 30% of their values missing, and an individualized approach was taken with each of them. In some cases, categories of existing values were used to preserve some of the information with NaN values being their own category. In many cases, there was too little data or variance for the column to be useful in the analysis.

The smart_195_raw, smart_189_raw, and smart_183_raw columns had extremely little difference in distributions between failure and non-failure instances, and only had values on some Seagate drives. To avoid collinearity with the manufacturer column for little predictive benefit, the columns were dropped.

```
sns.distplot(df.loc[df['failure'] == 0]['smart_195_raw'])
sns.distplot(df.loc[df['failure'] == 1]['smart_195_raw'])
plt.grid(True)
plt.title("smart_195_raw Distribution by Failure")
plt.legend(["Not Failed", "Failed"])
```

```
<matplotlib.legend.Legend at 0x17218898f08>
```



```
df.loc[df['smart_195_raw'].notnull()]['manufacturer'].value_counts()
```

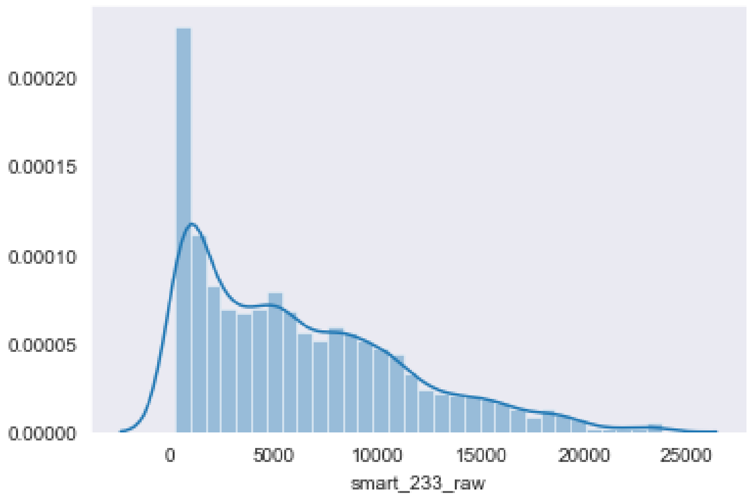
```
Seagate    6168809
Name: manufacturer, dtype: int64
```

```
df.drop(['smart_195_raw'], axis=1, inplace=True)
```

Many of the columns had less than 2% of their values filled in and had no failure instances. Failing to have instances in both classes renders any predictive power the columns may have had useless, and as such these columns were dropped. The columns in this group were smart_233_raw, smart_235_raw, smart_232_raw, smart_168_raw, smart_170_raw, smart_218_raw, smart_174_raw, smart_16_raw, smart_17_raw, smart_173_raw, smart_231_raw, and smart_177_raw.

```
sns.distplot(df.loc[df['smart_233_raw'].notnull()]['smart_233_raw'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1ce86f91f88>
```



```
df.loc[(df['smart_233_raw'].notnull()) & (df['failure'] == 1)]
```

date	serial_number	model	failure	smart_1_raw	smart_3_raw	smart_4_raw	smart_5_raw	sm

0 rows × 48 columns

```
df.loc[df['smart_233_raw'].notnull()]['manufacturer'].value_counts()
```

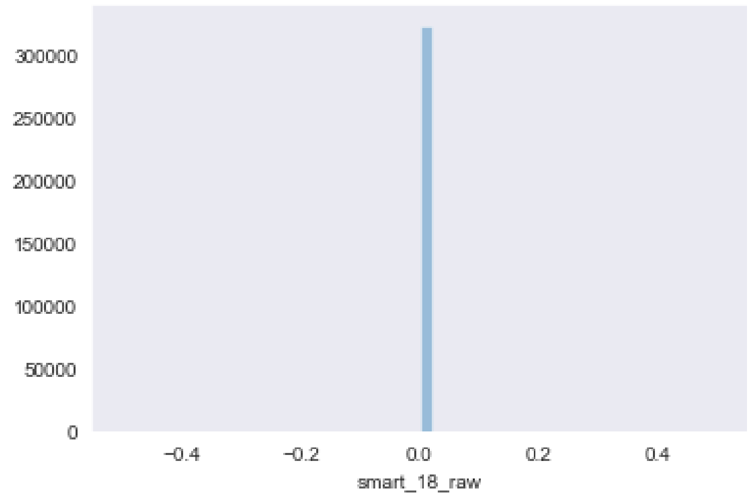
```
Seagate            8792
Western Digital       0
Toshiba               0
HGST                  0
Name: manufacturer, dtype: int64
```

```
df.drop(['smart_233_raw'], axis=1, inplace=True)
```

Several of the columns did have instances of failures and non-failures but had no variance in the values and as such were dropped. Without variance, no distinction exists between failure and non-failure, making the columns useless for analysis and prediction. The columns in this group were smart_18_raw, smart_224_raw, smart_23_raw, smart_24_raw, and smart_254_raw.

```
sns.distplot(df.loc[df['smart_18_raw'].notnull()]['smart_18_raw'], kde = False)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1dd4e11b108>
```



```
df['smart_18_raw'].value_counts()
```

```
0.0    323114
Name: smart_18_raw, dtype: int64
```

```
df.loc[df['failure'] == 1]['smart_18_raw'].value_counts()
```

```
0.0    10
Name: smart_18_raw, dtype: int64
```

```
df.loc[df['smart_18_raw'].notnull()]['manufacturer'].value_counts()
```

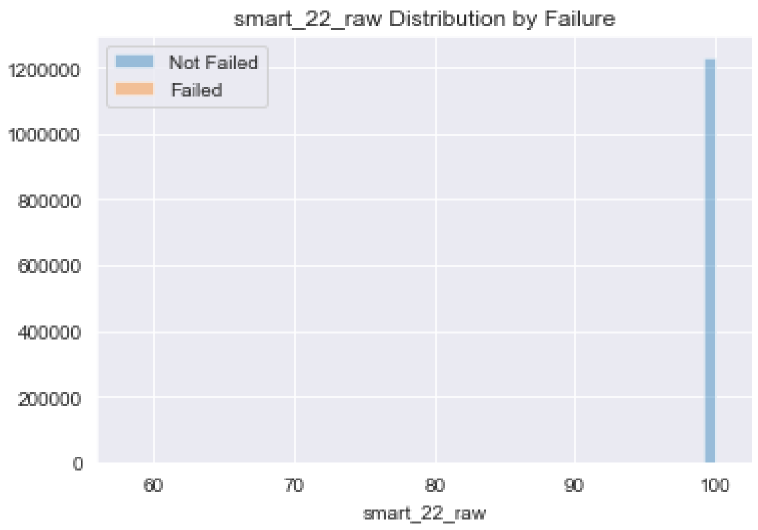
```
Seagate            323114
Western Digital         0
Toshiba                 0
HGST                    0
Name: manufacturer, dtype: int64
```

```
df.drop(['smart_18_raw'], axis=1, inplace=True)
```

The smart_22_raw column is quite different from the other types of SMART values as it is an indication of helium levels encased in certain HGST drives (Klein, 2015). Given this, it would have made no sense to fill this column's NaN values in rows of drives from other manufacturers. Beyond that, the dataset did not have any failures with abnormal levels in this column, making this column potentially a negative impact to the real-world effectiveness of a predictive model. Given this risk, the purpose of the value, and the risk of collinearity with the manufacturer column, this column was dropped from the dataset.

```
sns.distplot(df.loc[df['failure'] == 0]['smart_22_raw'], kde = False)
sns.distplot(df.loc[df['failure'] == 1]['smart_22_raw'], kde = False)
plt.grid(True)
plt.title("smart_22_raw Distribution by Failure")
plt.legend(["Not Failed", "Failed"])
```

```
<matplotlib.legend.Legend at 0x26243e3ea08>
```



```
df.loc[df['failure'] == 1]['smart_22_raw'].value_counts()
```

```
100.0    10
Name: smart_22_raw, dtype: int64
```

```
df.loc[df['smart_22_raw'].notnull()]['manufacturer'].value_counts()
```

```
HGST    1233138
Name: manufacturer, dtype: int64
```

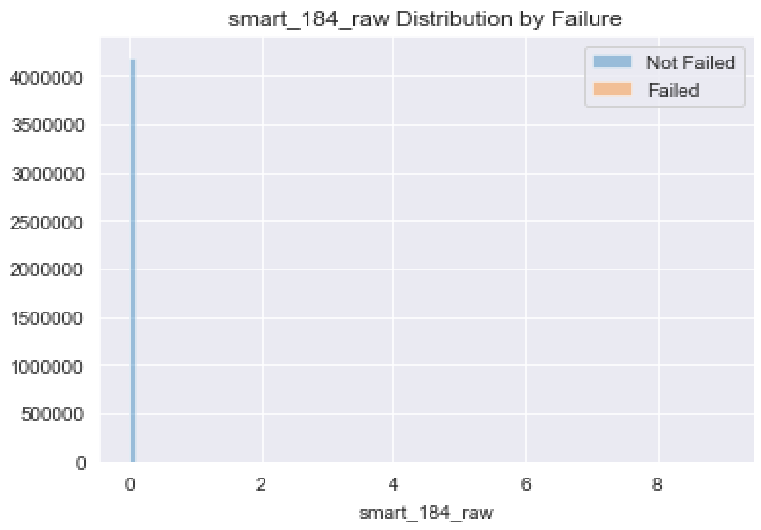
```
df.drop(['smart_22_raw'], axis=1, inplace=True)
```

The smart_184_raw column was another unique column as it had very few values other than zero, but half of the non-zero values were instances of failure. To preserve this information despite most of the values being NaN values, a new Boolean column was created where 0 or

NaN values were false and non-zero values were true. The original column was then dropped from the dataset.

```
sns.distplot(df.loc[df['failure'] == 0]['smart_184_raw'], kde = False)
sns.distplot(df.loc[df['failure'] == 1]['smart_184_raw'], kde = False)
plt.grid(True)
plt.title("smart_184_raw Distribution by Failure")
plt.legend(["Not Failed", "Failed"])
```

```
<matplotlib.legend.Legend at 0x16c10895808>
```



```
df.loc[df['smart_184_raw'].notnull()]['smart_184_raw'].value_counts()
```

```
0.0    4194090
1.0          5
5.0          3
9.0          1
8.0          1
4.0          1
2.0          1
Name: smart_184_raw, dtype: int64
```

```
df.loc[(df['smart_184_raw'] != 0) & \
       (df['smart_184_raw'].notnull())][['smart_184_raw', 'failure']]
```

	smart_184_raw	failure
99758	8.0	True
2613651	9.0	True
4849813	1.0	True
5931498	2.0	True
8943626	4.0	False
9066037	5.0	False
9189214	5.0	True
9836420	1.0	False
9961273	1.0	False
10086127	1.0	True
10771703	1.0	False
10896354	5.0	False

```
df['smart_184_cat'] = 0
```

```
df.loc[(df['smart_184_raw'] > 0), 'smart_184_cat'] = 1
```

```
df['smart_184_cat'] = df['smart_184_cat'].astype('category')
df['smart_184_cat'].dtype
```

```
CategoricalDtype(categories=[0, 1], ordered=False)
```

```
df['smart_184_cat'].value_counts()
```

```
0    10975099
1          12
Name: smart_184_cat, dtype: int64
```

```
df['smart_184_cat'].isnull().sum()
```

```
0
```

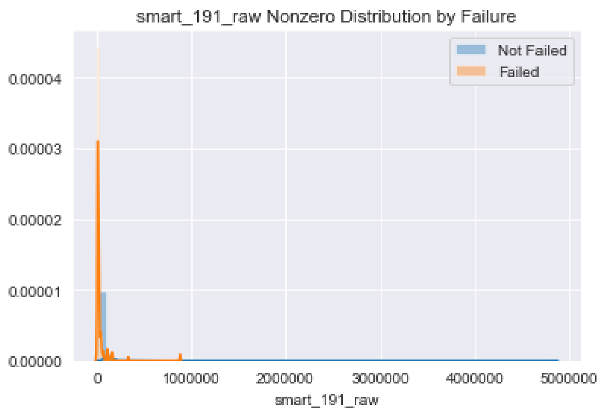
```
df.drop(['smart_184_raw'], axis=1, inplace=True)
```

The ten columns remaining were smart_191_raw, smart_200_raw, smart_196_raw, smart_8_raw, smart_2_raw, smart_223_raw, smart_11_raw, smart_220_raw, smart_222_raw, and smart_226_raw. This group of columns did not have a lack of value variance nor were they split directly on manufacturer lines, but they still had over 30% of their values missing. Filling in that large of a proportion of missing values with summary statistics would likely have skewed the data significantly. To avoid losing all of the information contained in the available data, categorical columns for each original column were created.

Each categorical column was given 3 possible category values. For each categorical column, the value of 0 was given to rows that had NaN values, the value of 1 was given to rows that had a value below the mean of the respective original column, and the value of 2 was given to rows that had a value above the mean of the respective original column, except in the case of smart_220_raw where the median was used for these comparison assignments instead. Once the creation and value assignment of the categorical columns were verified, the original columns were dropped from the dataset.

```
sns.distplot(df.loc[(df['failure'] == 0) & \
                    (df['smart_191_raw'] != 0.0)]['smart_191_raw'])
sns.distplot(df.loc[(df['failure'] == 1) & \
                    (df['smart_191_raw'] != 0.0)]['smart_191_raw'])
plt.grid(True)
plt.title("smart_191_raw Nonzero Distribution by Failure")
plt.legend(["Not Failed", "Failed"])
```

```
<matplotlib.legend.Legend at 0x2638a69f108>
```



```
df.loc[df['smart_191_raw'].notnull()]['manufacturer'].value_counts()
```

```
Seagate            4239295
Toshiba             322722
Western Digital      10249
Name: manufacturer, dtype: int64
```

```
smart_191_mean = df.loc[df['smart_191_raw'].notnull()]['smart_191_raw'].mean()
smart_191_mean
```

```
14090.159100979688
```

```
df['smart_191_cat'] = 0
```

```
df.loc[(df['smart_191_raw'] < smart_191_mean), 'smart_191_cat'] = 1
df.loc[(df['smart_191_raw'] > smart_191_mean), 'smart_191_cat'] = 2
```

```
df['smart_191_cat'] = df['smart_191_cat'].astype('category')
df['smart_191_cat'].dtype
```

```
CategoricalDtype(categories=[0, 1, 2], ordered=False)
```

```
df['smart_191_cat'].value_counts()
```

```
0    6402845
1    3563568
2    1008698
Name: smart_191_cat, dtype: int64
```

```
df['smart_191_cat'].isnull().sum()
```

```
0
```

```
df.drop(['smart_191_raw'], axis=1, inplace=True)
```

Though the decision and process of NaN value management was the same as the other 7 columns, the 3 columns smart_220_raw, smart_222_raw, and smart_226_raw deserve additional explanation. These columns were split entirely along manufacturer lines and had large percentages of missing values but were some of the few predictors available for Toshiba drives. However, despite the Toshiba drives not having the highest rate of failure among the manufacturers, these 3 columns were among the highest column correlations to failure in the entire dataset. These columns were given the same categorical column approach to ensure the enough predictors existed for the Toshiba drives.

```
fail_df = pd.crosstab(df["manufacturer"], df["failure"])
fail_df['Rate'] = fail_df[1] / (fail_df[0] + fail_df[1])
fail_df
```

failure	False	True	Rate
manufacturer			
HGST	2660507	26	0.000010
Seagate	7965951	606	0.000076
Toshiba	322682	40	0.000124
Western Digital	25295	6	0.000237

```
df.loc[df['smart_220_raw'].notnull()]['manufacturer'].value_counts()
```

```
Toshiba            322722
Western Digital         0
Seagate                 0
HGST                    0
Name: manufacturer, dtype: int64
```

```
df[['smart_220_raw', 'failure']].corr()
```

	smart_220_raw	failure
smart_220_raw	1.000000	-0.006208
failure	-0.006208	1.000000

```
df.loc[df['smart_222_raw'].notnull()]['manufacturer'].value_counts()
```

```
Toshiba            322722
Western Digital         0
Seagate                 0
HGST                    0
Name: manufacturer, dtype: int64
```

```
df[['smart_222_raw', 'failure']].corr()
```

	smart_222_raw	failure
smart_222_raw	1.000000	0.010691
failure	0.010691	1.000000

```
df.loc[df['smart_226_raw'].notnull()]['manufacturer'].value_counts()
```

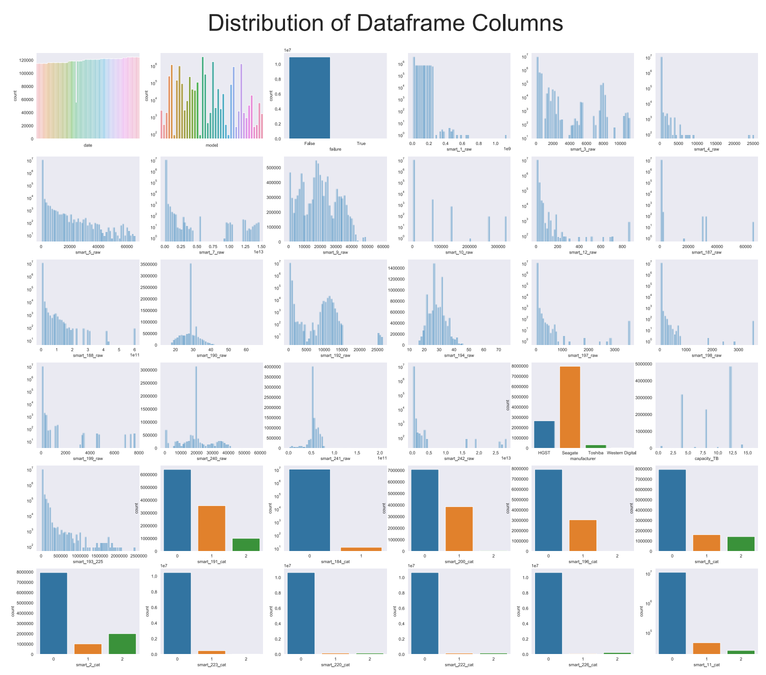
```
Toshiba            322722
Western Digital         0
Seagate                 0
HGST                    0
Name: manufacturer, dtype: int64
```

```
df[['smart_226_raw', 'failure']].corr()
```

	smart_226_raw	failure
smart_226_raw	1.000000	-0.014187
failure	-0.014187	1.000000

With this, the number of dimensions in the dataset was reduced to 36 and all values in all columns were filled. The univariate distributions of all columns were plotted together in the form of histograms for continuous data and countplots for categorical data.

```
fig, axes = plt.subplots(6, 6, figsize = (30, 25))

row = 0
col = 0
for df_col in ['date', 'model', 'failure', 'smart_1_raw',
        'smart_3_raw', 'smart_4_raw', 'smart_5_raw', 'smart_7_raw',
        'smart_9_raw', 'smart_10_raw', 'smart_12_raw', 'smart_187_raw',
        'smart_188_raw', 'smart_190_raw', 'smart_192_raw', 'smart_194_raw',
        'smart_197_raw', 'smart_198_raw', 'smart_199_raw', 'smart_240_raw',
        'smart_241_raw', 'smart_242_raw', 'manufacturer', 'capacity_TB',
        'smart_193_225', 'smart_191_cat', 'smart_184_cat', 'smart_200_cat',
        'smart_196_cat', 'smart_8_cat', 'smart_2_cat', 'smart_223_cat',
        'smart_220_cat', 'smart_222_cat', 'smart_226_cat', 'smart_11_cat']:

    if col == 6:
        row += 1
        col = 0

    # Histograms
    if df[df_col].dtype.name == 'float64':
        if df_col in ['smart_1_raw', 'smart_3_raw', 'smart_4_raw',
                      'smart_5_raw', 'smart_7_raw', 'smart_10_raw',
                      'smart_12_raw', 'smart_187_raw', 'smart_188_raw',
                      'smart_192_raw', 'smart_197_raw', 'smart_198_raw',
                      'smart_199_raw', 'smart_242_raw', 'smart_193_225']:
            ax = sns.distplot(df[df_col], ax = axes[row, col], kde = False)
            ax.set_yscale('log')

        else:
            ax = sns.distplot(df[df_col], ax = axes[row, col], kde = False)
    # Countplots
    elif df[df_col].dtype.name == 'category' or \
                df[df_col].dtype.name == 'bool':
        if df_col == "date":
            ax = sns.countplot(df[df_col], ax = axes[row, col])
            ax.set(xticklabels = [])

        elif df_col == "model":
            ax = sns.countplot(df[df_col], ax = axes[row, col])
            ax.set(xticklabels = [])
            ax.set_yscale('log')

        elif df_col in ['smart_184_cat', 'smart_11_cat']:
            ax = sns.countplot(df[df_col], ax = axes[row, col])
            ax.set_yscale('log')

        else:
            sns.countplot(df[df_col], ax = axes[row, col])

    else:
        print("Unknown column dtype")

    col += 1


plt.subplots_adjust(top = 0.90)
fig.suptitle("Distribution of Dataframe Columns", fontsize = 54, y = 0.95)
fig.savefig("Charts/Dataframe Distributions.svg")
fig.savefig("Charts/Dataframe Distributions.png")
```



At this point in the project, the quantitative column correlation coefficients and the qualitative columns contingency tables were created. Before factor analysis and model creation could occur, one last series of data preparation had to occur. The dataset then prepared through standardization and normalization, as well as the test, train, and validation splits occurring then. Doing these before the PCA ensures that no data is contaminated with the influence of the testing and validation data. Additionally, this must occur before SMOTE or any other oversampling technique can be performed. At this point, the date and serial_number columns were also dropped as identification columns were no longer needed.

```
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
```

```
y_df = df['failure']
x_df = df.drop('failure', axis = 1)
```

```
x_df.drop(['date', 'serial_number'], axis = 1, inplace = True)
```

```
del df
```

The first split is 80% Train and 20% Test, stratified on the y_df / failure series. Using stratified sampling ensures that there is an evenly distributed proportion of minority classes in the training, testing, and validation datasets despite the extreme imbalance of the minority class from the rarity of hard drive failure.

```
x_train, x_test, y_train, y_test = train_test_split(x_df, y_df, \
                                    test_size = 0.2, random_state = 13, stratify = y_df)
```

Verify the stratified splitting.

```
y_train.value_counts()
```

```
False    8779546
True         542
Name: failure, dtype: int64
```

```
y_train.value_counts()[1] / y_train.value_counts()[0]
```

```
6.173439947805957e-05
```

```
y_test.value_counts()
```

```
False    2194887
True         136
Name: failure, dtype: int64
```

The ratios between the minority and majority classes are calculated for each dataset split to ensure that the stratified random sampling functioned properly. Although the ratios are not equal, the sampling selected the closest ratio mathematically possible.

```
y_test.value_counts()[1] / y_test.value_counts()[0]
```

```
6.196218757503233e-05
```

```
(y_test.value_counts()[1] - 1) / y_test.value_counts()[0]
```

```
6.150658325462769e-05
```

```
(y_test.value_counts()[1] + 1) / y_test.value_counts()[0]
```

```
6.241779189543698e-05
```

The second split is 87.5% Train and 12.5% Validation, stratified on the y_df / failure series, to result in 70% Train and 10% Validation overall.

```
x_train, x_valid, y_train, y_valid = train_test_split(x_train, y_train, \
                    test_size = 0.125, random_state = 13, stratify = y_train)
```

```
y_train.value_counts()
```

```
False    7682103
True         474
Name: failure, dtype: int64
```

```
y_train.value_counts()[1] / y_train.value_counts()[0]
```

```
6.170185429692885e-05
```

As in the first split between that training and testing datasets, the class ratios between the training and validation datasets are also examined and found to be accurate.

```
y_valid.value_counts()
```

```
False    1097443
True          68
Name: failure, dtype: int64
```

```
y_valid.value_counts()[1] / y_valid.value_counts()[0]
```

```
6.196221580528556e-05
```

A scaler is created and fit to the training data in order to standardize the quantitative columns for model training. This avoids any contamination of the training data by ensuring that the test and validation datasets do not influence the training data at all, as the mean and standard deviation of the data must be calculated to scale and normalize. The fit scaler can then be used on the testing and validation datasets. Scikit-learn’s StandardScaler() was selected to produce standardized and normalized data in the form that models like neural networks need for smooth training.

```
cont_cols = [
    'smart_1_raw', 'smart_3_raw', 'smart_4_raw', 'smart_5_raw',
    'smart_7_raw', 'smart_9_raw', 'smart_10_raw', 'smart_12_raw',
    'smart_187_raw', 'smart_188_raw', 'smart_190_raw', 'smart_192_raw',
    'smart_194_raw', 'smart_197_raw', 'smart_199_raw', 'smart_240_raw',
    'smart_241_raw', 'smart_242_raw', 'smart_193_225', 'capacity_TB'
]
```

```
scaler = preprocessing.StandardScaler()
```

```
x_train[cont_cols] = scaler.fit_transform(x_train[cont_cols])
```

A mean as close to zero as possible given the dataset and a standard deviation of 1 is a successful standardization.

```
x_train[cont_cols].describe()
```

	smart_1_raw	smart_3_raw	smart_4_raw	smart_5_raw	smart_7_raw	smart_9_raw
count	7.682577e+06	7.682577e+06	7.682577e+06	7.682577e+06	7.682577e+06	7.682577e+06
mean	6.559957e-17	1.272166e-17	-3.058378e-17	-6.428810e-18	-2.783875e-18	-3.501949e-17
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00

Once proper standardization and normalization is confirmed, the scaler is then used to transform the quantitative columns of the testing and validation splits.

```
x_test[cont_cols] = scaler.transform(x_test[cont_cols])
x_valid[cont_cols] = scaler.transform(x_valid[cont_cols])
```

Analysis

With all data tidying and preparation complete, the analysis began with calculating the Pearson correlation coefficients for all quantitative columns.

```
corr_df = df.corr(method = 'pearson')
```

	failure	smart_1_raw	smart_3_raw	smart_4_raw	smart_5_raw	smart_7_raw
failure	1.000000	0.002200	-1.664119e-04	0.001083	0.044413	0.000082
smart_1_raw	0.002200	1.000000	-2.311701e-01	-0.008473	0.014507	0.008715
smart_3_raw	-0.000166	-0.231170	1.000000e+00	0.007011	-0.007325	-0.004563
smart_4_raw	0.001083	-0.008473	7.010971e-03	1.000000	-0.000145	0.000296
smart_5_raw	0.044413	0.014507	-7.325152e-03	-0.000145	1.000000	-0.000127

Preliminary examinations show that smart_197_raw and smart_198_raw have a nearly perfect degree of collinearity with each other and little in comparison with any other column. In order to prevent that from affecting the predictive models, the smart_198_raw column was dropped as it has a lower correlation with the dependent variable failure.

```
df[['smart_197_raw', 'smart_198_raw', 'failure']].corr()
```

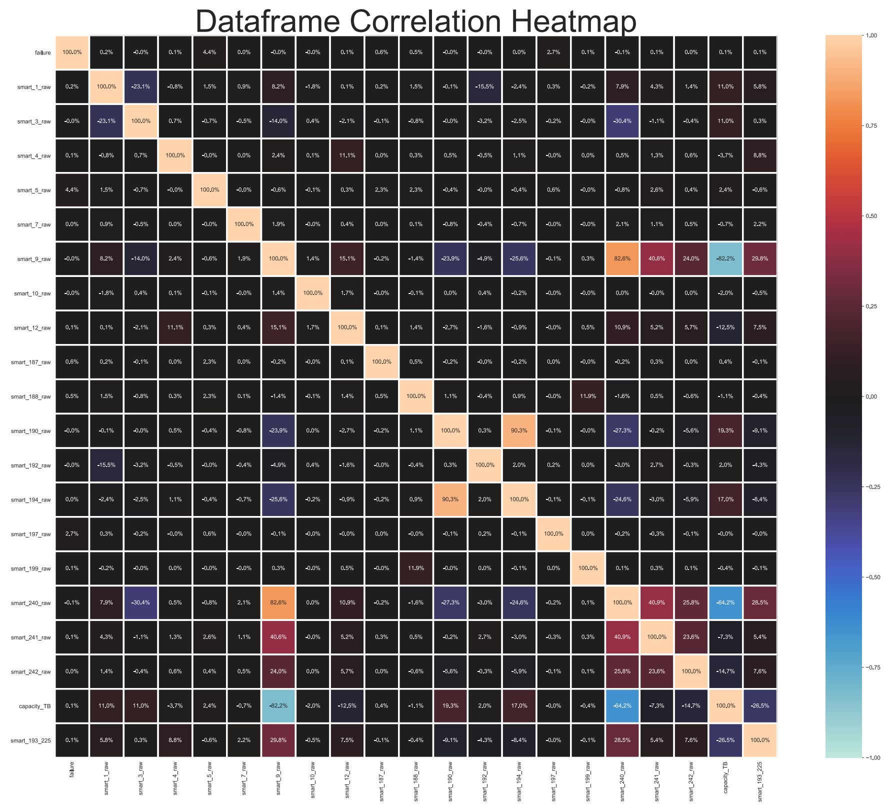
```
              smart_197_raw  smart_198_raw   failure
smart_197_raw      1.000000       0.978249   0.02741
smart_198_raw      0.978249       1.000000   0.02087
      failure      0.027410       0.020870   1.00000
```

```
df.drop('smart_198_raw', axis = 1, inplace = True)
```

```
corr_df = df.corr(method = 'pearson')
```

This dataframe is then used to create a heatmap of the column correlations, with the color scale centered at 0 for both positive and negative correlations.

```
fig, ax = plt.subplots(figsize = (30, 23))

sns.heatmap(
    corr_df,
    ax = ax,
    annot = True,
    fmt = ".1%",
    vmin = -1, vmax = 1, center = 0,
    linewidths = 3,
    linecolor = "white",
    xticklabels = corr_df.columns,
    yticklabels = corr_df.columns,
    square = True,
    cbar = True
)

plt.title("Dataframe Correlation Heatmap", fontsize = 54)
fig.savefig("Charts/Corr Heatmap.svg")
fig.savefig("Charts/Corr Heatmap.png")
```



Examining the column correlations shows several important details. For potential predictors for failure, smart_5_raw and smart_197_raw have the highest positive correlations with failure, at 4.4% and 2.7%. SMART attribute 5 is the reallocated sectors count of drives, which triggers when a read, write, or verification error occurs (Acronis, Knowledge Base 9105). SMART attribute 197 is the current pending sector count, which is the count of unstable sectors that are awaiting remapping (Acronis, Knowledge Base 9133). This value decreases as sectors are remapped, but the value would remain consistently high if these sectors are unable to be remapped. Both columns make complete sense as the highest correlation with failure and will likely be the most important predictor variables for HDD failure.

Another prominent feature is smart_9_raw as the column with the most extreme correlations with other columns, which is understandable given that SMART attribute 9 represents the total count of hours the drive has been in a power-on state (Acronis, Knowledge Base 9109). Most other issues worth measuring are likely correlated with the drive age and amount of operation. This column may also be a powerful predictor within predictive models as an older drive is more likely to wear down to failure suddenly than a newer drive in general even if other values are not present. Even if other predictors of failure are present in an instance, a drive with an average or lower smart_9_raw value may represent a drive that will fail far sooner than the average length of time to failure.

Other features to note are that the smart_240_raw column has quite high correlations with other independent variables and that smart_190_raw and smart_194_raw have a very high degree of collinearity with each other and little in comparison with any other column. The dataset is likely large enough to not need to worry about the multicollinearity affecting the predictive power of the models, but the redundancy of information may skew the results.

After examining the quantitative variables, the qualitative variables were assessed with Fisher’s exact test. Pearson’s chi-squared was calculated on each for comparison, but the results cannot be trusted as the data is not normally distributed. Rpy2 was used to load the R.stats package for Fisher’s exact test, as neither scikit-learn nor scipy have an implementation of Fisher’s exact test on contingency tables with dimensions greater than 2x2. Specifically, a contingency table was created for each column through pandas’ crosstab method, and these contingency tables were passed into a custom function that passes the contingency table into rpy2, which embeds the R.stats code into the Python process and returns the R.stats Fisher_Test with Monte Carlo p-value simulation output. The custom function then takes this output and displays it appropriately in the Jupyter notebook.

```
import rpy2.robjects.numpy2ri
from rpy2.robjects.packages import importr
import rpy2.robjects as ro
rpy2.robjects.numpy2ri.activate()
rstats = importr('stats')
```

```
# Display the formatted results of the R stats Fisher_Test,
# using Monte Carlo Simulation
def r_fisher_output(dataframe):
    results = rstats.fisher_test(dataframe.to_numpy(), \
                                 simulate_p_value = True)

    # Convert the listvector object returned from R stats to
    # a list of string values
    d = [key + "_" + str(results.rx2(key)[0]) for key in results.names]
    d2 = []
    for i in d:
        d2.append("".join(i.replace("\t", "").splitlines()))

    # Replicate the tabluar data formatting
    for line in d2:
        if len(line.split("_")[0]) < 8:
            print(line.replace("_", "\t\t"))
        else:
            print(line.replace("_", "\t"))
```

```
manufacturer_contingency = pd.crosstab(df['manufacturer'], df['failure'])
manufacturer_contingency
```

failure	False	True
manufacturer		
HGST	2660507	26
Seagate	7965949	606
Toshiba	322682	40
Western Digital	25295	6

```
r_fisher_output(manufacturer_contingency)
```

```
p.value          0.0004997501249375312
alternative      two.sided
method           Fisher's Exact Test for Count Data with simulated p-value (b
ased on 2000 replicates)
data.name1       structure(c(2660507L, 7965949L, 322682L, 25295L, 26L, 606L,
40L,
data.name2       6L), .Dim = c(4L, 2L))
```

All columns were found to be significant, but smart_184_cat had the absolute lowest p-value at 5.0214144599400225e-23.

```
r_fisher_output(smart_184_contingency)
```

```
p.value          5.0214144599400225e-23
conf.int         4201.0256410217235
estimate         16382.987859030574
null.value       1.0
alternative      two.sided
method           Fisher's Exact Test for Count Data
data.name        structure(c(10974427L, 6L, 672L, 6L), .Dim = c(2L, 2L))
```

As all were found to be significant, no columns were dropped to reduce dimensionality here. Principal Component Analysis (PCA) is used for dimensionality reduction instead. However, the model column was dropped as its contingency table was shown to be very sparsely filled and it had redundant information with the manufacturer column.

```
df.drop('model', axis = 1, inplace = True)
```

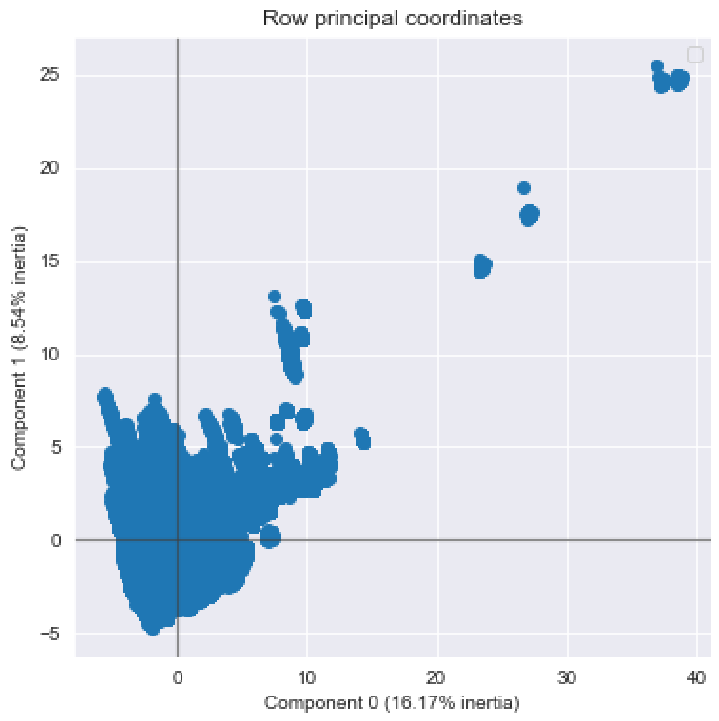
In the same way that the standardization was performed, the PCA was fit to the training data only. PCA as a form of dimensionality reduction ensures that as little information, in the form of inertia, is lost as possible for the given number of dimensions reduced. As this dataset is quite large, any amount of dimensionality reduction greatly affects the speed and chance of proper convergence in predictive models. This first PCA was created with a number of components equal to the number of the quantitative columns to examine the inertia explained in the dataset in order to determine the appropriate number of dimensions to use.

```
import prince
```

```
pca = prince.PCA(
     n_components = len(cont_cols),
     n_iter = 3 ,
     copy = True,
     check_input = True,
     random_state = 13
)
```

```
pca = pca.fit(x_train[cont_cols])
```

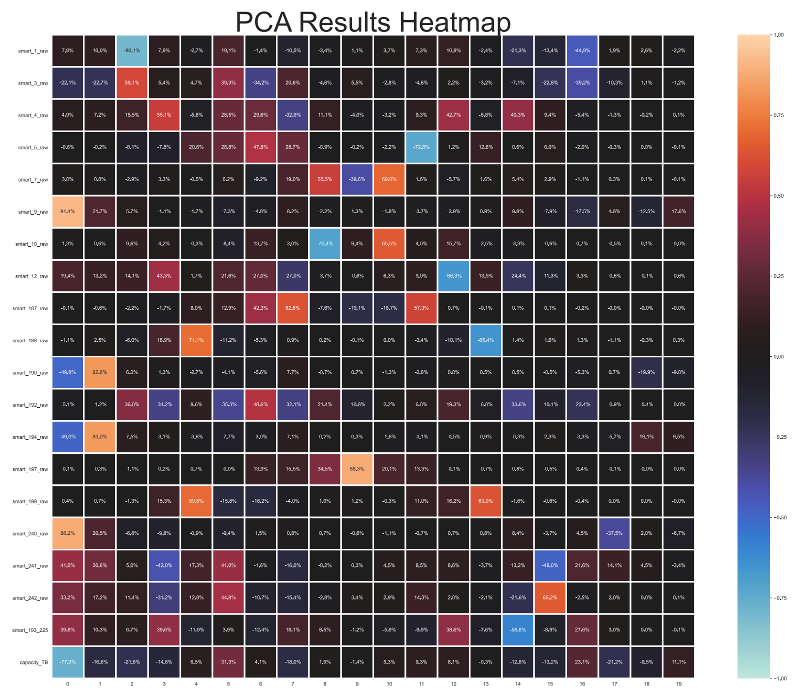
```
ax = pca.plot_row_coordinates(
     x_train[cont_cols],
     ax = None,
     figsize = (6, 6),
     x_component = 0,
     y_component = 1
)

# No .svg file will be saved for this plot as it takes up
# 1.07 GB (1,158,481,389 bytes).
#plt.savefig("Charts/PCA.svg")
plt.savefig("Charts/PCA.png")
```



The influence of each column on each principal component was then examined by creating a heatmap from a dataframe formed on the information.

```
pca_results_df = pca.column_correlations(x_train[cont_cols])
```

```
fig, ax = plt.subplots(figsize = (30, 23))

sns.heatmap(
    pca_results_df,
    ax = ax,
    annot = True,
    fmt = ".1%",
    vmin = -1, vmax = 1, center = 0,
    linewidths = 3,
    linecolor = "white",
    xticklabels = pca_results_df.columns,
    yticklabels = pca_results_df.index,
    square = True,
    cbar = True
)

plt.title("PCA Results Heatmap", fontsize = 54)
plt.savefig("Charts/PCA Heatmap.svg")
plt.savefig("Charts/PCA Heatmap.png")
```



As these correlations are spread out well over the heatmap nothing else needed to be done with it. The eigenvalues and explained inertia were used to create a scree plot, and this plot was used alongside the cumulative inertia to determine that 13 principal components was the appropriate amount of dimensionality reduction to use as these components made up 82.37% of the inertia of the dataset in only 13 out of the 20, or 65%, of the total components.

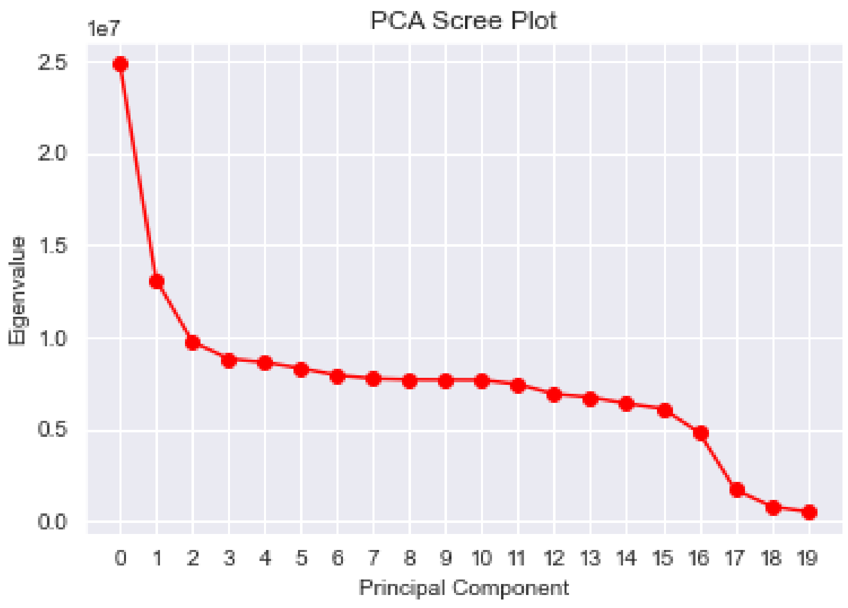
```
pca_eigenvalues = pca.eigenvalues_
pca_eigenvalues
```

```
[24843306.51,
 13129034.19,
 9758841.017,
 8826913.796,
 8626783.65,
 8304944.847,
 7926688.784,
 7744995.316,
 7689628.569,
 7679972.369,
 7671021.412,
 7440170.605,
 6917334.239,
 6736046.078,
 6395419.9,
 6110364.178,
 4819698.087,
 1720733.771,
 797311.7685,
 512330.9103]
```

```
pca.explained_inertia_
```

```
[0.16168602354268868,
 0.08544681158976027,
 0.06351280968257833,
 0.05744761032724548,
 0.05614511673603315,
 0.05405051486588958,
 0.051588736334488725,
 0.05040623293370864,
 0.0500458932548495,
 0.049983048454989215,
 0.04992479354032559,
 0.04842236273611431,
 0.04501962192457163,
 0.04383975635902524,
 0.041622881877629705,
 0.039767672864061826,
 0.03136771741602234,
 0.011198936056775211,
 0.005189090643217338,
 0.0033343688600271061]
```

```
plt.plot(np.arange(len(cont_cols)), pca_eigenvalues, 'ro-')
plt.title("PCA Scree Plot")
plt.xlabel("Principal Component")
plt.ylabel("Eigenvalue")
plt.xticks(range(0, len(cont_cols)))
plt.grid(b = True, which = 'major', color = 'w', linewidth = 1.0)
plt.grid(b = True, which = 'minor', color = 'w', linewidth = 0.5)
plt.savefig("Charts/PCA Scree Plot.svg")
plt.savefig("Charts/PCA Scree Plot.png")
```



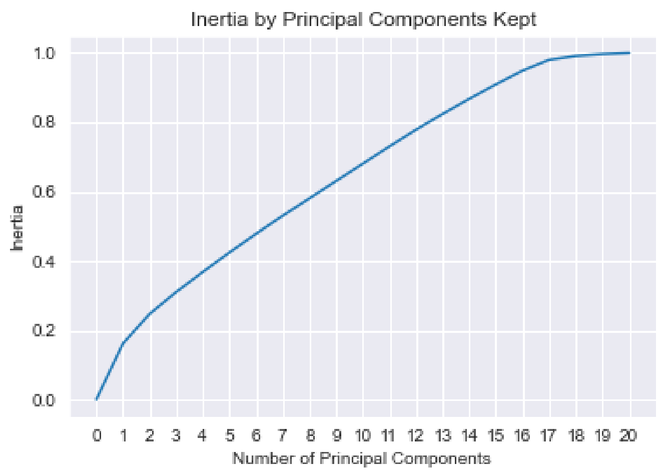
```
cum_inertia = [0]

for i, e in enumerate(pca_eigenvalues):
    cum_inertia.append(sum(pca_eigenvalues[0:i+1]) / sum(pca_eigenvalues))

cum_inertia
```

```
[0,
 0.1616860235,
 0.2471328351,
 0.3106456448,
 0.3680932551,
 0.4242383719,
 0.4782888867,
 0.5298776231,
 0.580283856,
 0.6303297493,
 0.6803127977,
 0.7302375913,
 0.778659954,
 0.8236795759,
 0.8675193323,
 0.9091422142,
 0.948909887,
 0.9802776044,
 0.9914765405,
 0.9966656311,
 1]
```

```
sum(pca_eigenvalues[0:13]) / sum(pca_eigenvalues)
```

```
0.8236795759
```

```
plt.plot(range(0, len(cum_inertia)), cum_inertia)
plt.title("Inertia by Principal Components Kept")
plt.xlabel("Number of Principal Components")
plt.ylabel("Inertia")
plt.xticks(range(0, len(cum_inertia)))
plt.grid(b=True, which = 'major', color = 'w', linewidth = 1.0)
plt.grid(b=True, which = 'minor', color = 'w', linewidth = 0.5)
plt.savefig("Charts/PCA Inertia Plot.svg")
plt.savefig("Charts/PCA Inertia Plot.png")
```



A new PCA is created with the appropriate number of components and then fit to the training data.

```
pca = prince.PCA(
    n_components = 13,
    n_iter = 3,
    copy = True,
    check_input = True,
    random_state = 13
)
```

```
pca = pca.fit(x_train[cont_cols])
```

This PCA is used to transform the quantitative values of the dataframe. The original quantitative columns are then dropped from the dataframe and the transformed values are merged in with a pandas join() method.

```
pca_df = pca.transform(x_train[cont_cols])
pca_df = pca_df.add_prefix('pca_component_')
```

```
pca_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 7682577 entries, 3450686 to 9385528
Data columns (total 13 columns):
pca_component_0     float64
pca_component_1     float64
pca_component_2     float64
pca_component_3     float64
pca_component_4     float64
pca_component_5     float64
pca_component_6     float64
pca_component_7     float64
pca_component_8     float64
pca_component_9     float64
pca_component_10    float64
pca_component_11    float64
pca_component_12    float64
dtypes: float64(13)
memory usage: 820.6 MB
```

```
# Replace the columns that factored in the PCA with
# the reduced-dimension PCA results.
x_train.drop(cont_cols, axis = 1, inplace = True)
x_train = x_train.join(pca_df)
```

This process is then repeated with the test and validation datasets.

```
pca_df = pca.transform(x_test[cont_cols])
pca_df = pca_df.add_prefix('pca_component_')

# Replace the columns that factored in the PCA with
# the reduced-dimension PCA results.
x_test.drop(cont_cols, axis = 1, inplace = True)
x_test = x_test.join(pca_df)
```

```
pca_df = pca.transform(x_valid[cont_cols])
pca_df = pca_df.add_prefix('pca_component_')

# Replace the columns that factored in the PCA with
# the reduced-dimension PCA results.
x_valid.drop(cont_cols, axis = 1, inplace = True)
x_valid = x_valid.join(pca_df)
```

While Factor Analysis of Mixed Data (FAMD) would have been ideal for dimensionality reduction in this mixed data, the current hardware requirements and software availability do not allow for it with such a large dataset.

One last adjustment was needed before training the predictive models as the categorical variables needed converting into Boolean encodings. Pandas get_dummies() method was used on the categorical columns of each dataset split to produce dataframes of encoding columns. These were then joined to the original dataframes of the train, test, and validation datasets after the original categorical columns were dropped.

```
cat_cols = [
    'manufacturer',  'smart_191_cat', 'smart_184_cat',
    'smart_200_cat', 'smart_196_cat', 'smart_8_cat',
    'smart_2_cat', 'smart_223_cat', 'smart_220_cat',
    'smart_222_cat', 'smart_226_cat', 'smart_11_cat'
]
```

```
x_train_cat = pd.get_dummies(x_train[cat_cols], \
                             columns = cat_cols, dtype = bool)
x_test_cat = pd.get_dummies(x_test[cat_cols], \
                            columns = cat_cols, dtype = bool)
x_valid_cat = pd.get_dummies(x_valid[cat_cols], \
                             columns = cat_cols, dtype = bool)
```

```
# Replace the categorical columns with their encoded representation columns.
x_test.drop(cat_cols, axis = 1, inplace = True)
x_test = x_test.join(x_test_cat)
```

```
# Replace the categorical columns with their encoded representation columns.
x_valid.drop(cat_cols, axis = 1, inplace = True)
x_valid = x_valid.join(x_valid_cat)
```

Traditional training would fail as hard drive failure is an extremely rare occurrence. The model would learn to only predict non-failure, making it useless for predicting failure.
As an illustration to this, a logistic regression model using the sag solver in scikit-learn was trained on the extremely imbalanced training set and then scored on the test set.

```
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, precision_score, \
                            classification_report, roc_curve, auc
```

```
regression = LogisticRegression(solver = 'sag', n_jobs = -1)
regression.fit(x_train, y_train.values.ravel())
```

As predicted, the model learned the dataset quite well regarding accuracy, in which it attained a nearly perfect score of the test data.

```
accuracy = regression.score(x_test, y_test)
accuracy
```

```
0.9999375860754078
```

As predicted as well though, it could not have had worse precision, in which it attained the lowest possible score of 0.0.

```
predictions = regression.predict(x_test)
actual = y_test
```

```
confusion = confusion_matrix(actual, predictions)
confusion
```

```
array([[2194886,       1],
       [    136,       0]], dtype=int64)
```

```
precision = precision_score(actual, predictions)
precision
```

```
0.0
```

Therefore, undersampling the non-failures, oversampling the failures, or a combination of both will improve the training and production of the predictive models. These methods are applied to the training data and reduce or eliminate the imbalance ratio for the training period. This project used Synthetic Minority Oversampling Technique (SMOTE) to synthetically create failure instances mathematically similar to the actual failures and did so until the imbalance was eliminated at a 50/50 split between failure and non-failure instances.

```
sm = SMOTE(random_state = 13)
```

```
x_train, y_train = sm.fit_resample(x_train, y_train)
```

```
y_train['failure'].value_counts()
```

```
True     7682103
False    7682103
Name: failure, dtype: int64
```

With the SMOTE training dataset, a new logistic regression model is trained, using the LBFGS solver.

```
regression = LogisticRegression(solver = 'lbfgs', \
                                max_iter = 10000, n_jobs = 1)
```

```
regression.fit(x_train, y_train.values.ravel())
```

```
LogisticRegression(max_iter=10000, n_jobs=1)
```

This model attained a very respectable accuracy but more importantly successfully identified 63.97% of failure instances while only having a false positive rate of 2.75%. SMOTE significantly improved the models training.

```
regression_accuracy = regression.score(x_test, y_test)
regression_accuracy
```

```
0.9731984585127355
```

```
regression_predictions = regression.predict(x_test)
actual = y_test
```

```
regression_confusion = confusion_matrix(actual, regression_predictions)
regression_confusion
```

```
array([[2136106,   58781],
       [     49,      87]], dtype=int64)
```

```
regression_precision = precision_score(actual, regression_predictions)
regression_precision
```

```
0.0014778827206631787
```

```
print(classification_report(actual, regression_predictions))
```

```
              precision    recall  f1-score   support

         0.0       1.00      0.97      0.99   2194887
         1.0       0.00      0.64      0.00       136

    accuracy                           0.97   2195023
   macro avg       0.50      0.81      0.49   2195023
weighted avg       1.00      0.97      0.99   2195023
```

The prediction probabilities, false positive and false negative rates, and the AUC were calculated. By using all of these, an ROC was graphed.

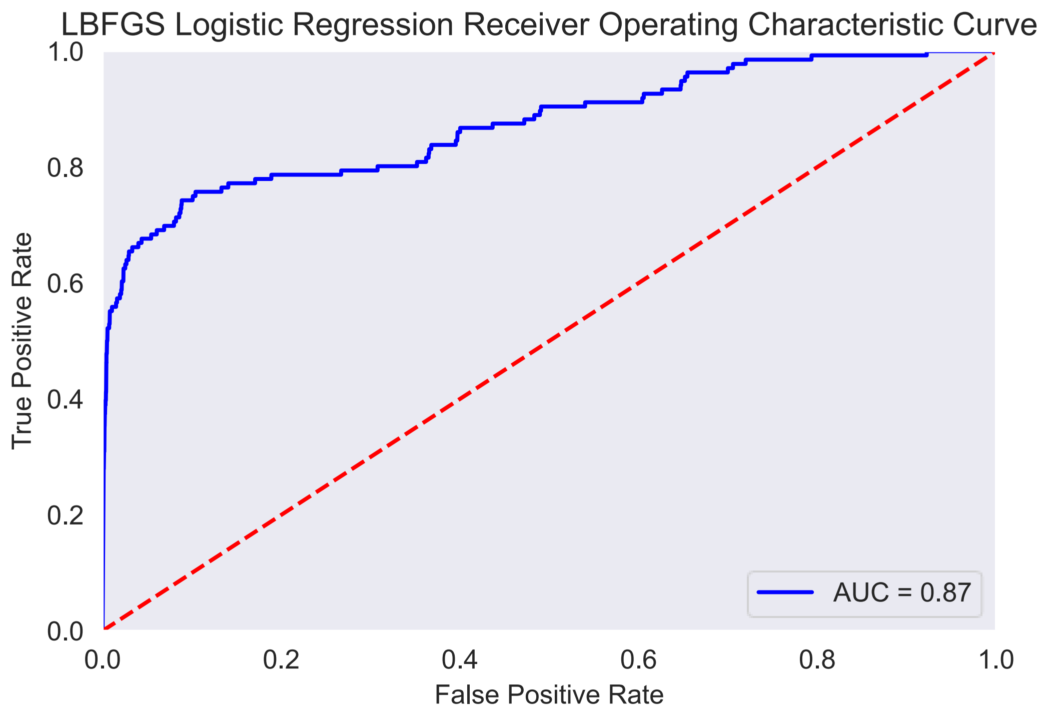
```
regression_probabilities = regression.predict_proba(x_test)
predictions = regression_probabilities[:,1]
```

```
regression_false_positive_rate, regression_true_positive_rate, threshold =\
    roc_curve(y_test, predictions)
```

```
regression_roc_auc = auc(regression_false_positive_rate, \
                         regression_true_positive_rate)
regression_roc_auc
```

```
0.8729115935460594
```

```
plt.title('LBFGS Logistic Regression Receiver Operating Characteristic Curve')
plt.plot(regression_false_positive_rate, regression_true_positive_rate, 'blue',
         label = 'AUC = %0.2f' % regression_roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.savefig('Charts/LBFGS Logistic ROC AUC.svg')
plt.show()
```



Examining the logistic regression's coefficients shows which parameters are the most influential in determining whether a failure is predicted to occur or not.

```
coefs = pd.concat([pd.DataFrame(x_train.columns),
                   pd.DataFrame(np.transpose(regression.coef_))], axis = 1)

coefs.columns = ["Column", "Coefficient"]
coefs
```

The most influential positive coefficients in this regression are smart_2_cat value 2 at 24.94, smart_220_cat value 2 at 15.66, and smart_226_cat value 1 at 15.14. The most influential negative coefficients in this regression are smart_223_cat value 1 at -22.03, smart_184_cat value 0 at -21.49, and manufacturer_Toshiba at -14.98.

The next standard model trained and tested was a decision tree with the maximum depth set to 20. A few other depths were tried, going as high as 100, but this limit amount performed the best out of the attempts.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import plot_tree
```

```
tree = DecisionTreeClassifier(max_depth = 20, splitter = 'best', \
                              random_state = 13)
```

```
tree.fit(x_train, y_train)
```

```
DecisionTreeClassifier(max_depth=20, random_state=13)
```

Once trained, the model was scored in the same fashion as the logistic regression model previously.

```
tree_accuracy = tree.score(x_test, y_test)
tree_accuracy
```

```
0.9690331263043713
```

```
tree_predictions = tree.predict(x_test)
actual = y_test
```

```
tree_confusion = confusion_matrix(actual, tree_predictions)
tree_confusion
```

```
array([[2126990,   67897],
       [     76,      60]], dtype=int64)
```

Unfortunately, the decision tree model did not perform as well as the logistic regression in any way. Its false positive amount was higher, and its true negative count was lower.

```
tree_precision = precision_score(actual, tree_predictions)
tree_precision
```

```
0.0008829112527039157
```

```
print(classification_report(actual, tree_predictions))
```

```
              precision    recall  f1-score   support

       False       1.00      0.97      0.98   2194887
        True       0.00      0.44      0.00       136

    accuracy                           0.97   2195023
   macro avg       0.50      0.71      0.49   2195023
weighted avg       1.00      0.97      0.98   2195023
```

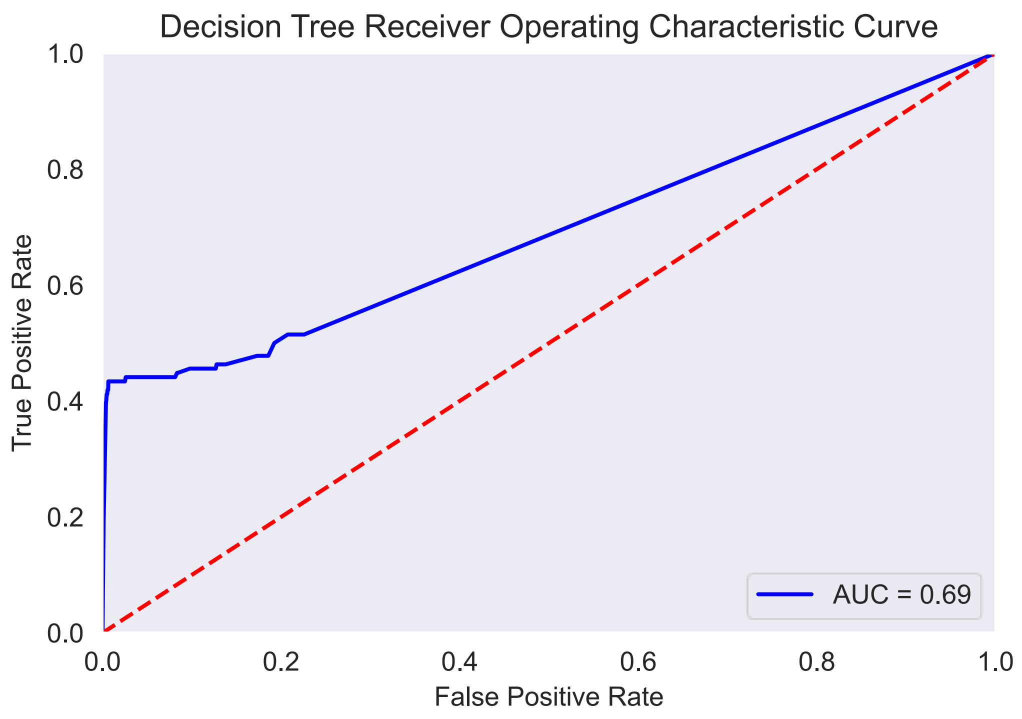
```
tree_probabilities = tree.predict_proba(x_test)
predictions = tree_probabilities[:,1]
```

```
tree_false_positive_rate, tree_true_positive_rate, threshold =\
    roc_curve(y_test, predictions)
```

```
tree_roc_auc = auc(tree_false_positive_rate, tree_true_positive_rate)
tree_roc_auc
```

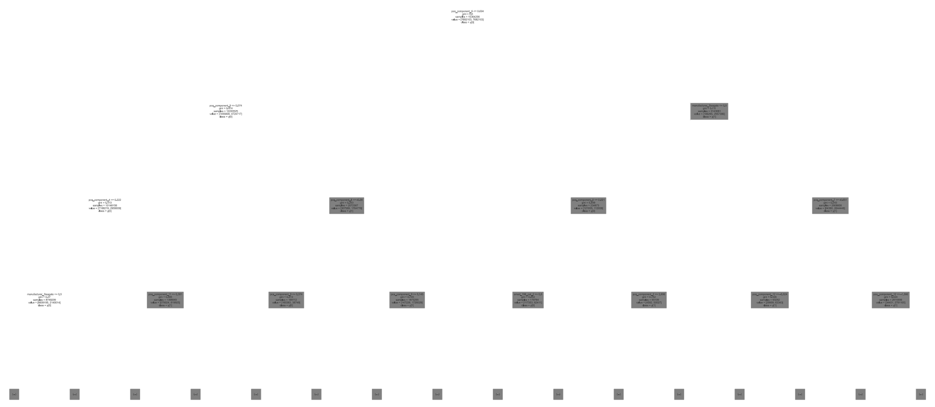
```
0.6900202690992079
```

```
plt.title('Decision Tree Receiver Operating Characteristic Curve')
plt.plot(tree_false_positive_rate, tree_true_positive_rate, 'blue',
         label = 'AUC = %0.2f' % tree_roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.savefig('Charts/Tree ROC AUC.svg')
plt.show()
```



Additionally, the decision tree was mapped out and plotted to examine the decisions and branches.

```
fig, ax = plt.subplots(figsize=(40, 20))
plot_tree(tree, fontsize = 6, max_depth = 3, class_names = True, \
          feature_names = x_train.columns)
plt.savefig('Charts/Decision Tree.svg', dpi=100)
plt.savefig('Charts/Decision Tree.png', dpi=100)
```



Zooming into this image to read the text shows that pca_component_6 <= 0.504 is the first split, followed by pca_component_9 <= 0.074 and pca_component_4 <= 0.222 down the trunk. The tree appears to have learned to use the PCA components to split at first and then only use the categorical encodings to make fine decisions later.

A random forest modeled was trained and then scored the same ways as the previous models next.

```
from sklearn.ensemble import RandomForestClassifier
```

```
forest = RandomForestClassifier(max_depth = 20, verbose = 1, \
                                random_state = 13, n_jobs = -1)
```

```
forest.fit(x_train, y_train.values.ravel())
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 12 concurrent wor
kers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed: 16.5min
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed: 45.8min finished
```

```
RandomForestClassifier(max_depth=20, n_jobs=-1, random_state=13, verbose=1)
```

```
forest_accuracy = forest.score(x_test, y_test)
forest_accuracy
```

```
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent wor
kers.
[Parallel(n_jobs=12)]: Done  26 tasks      | elapsed:    1.9s
[Parallel(n_jobs=12)]: Done 100 out of 100 | elapsed:    5.9s finished
```

```
0.9902342708937446
```

```
forest_predictions = forest.predict(x_test)
actual = y_test
```

```
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent wor
kers.
[Parallel(n_jobs=12)]: Done  26 tasks      | elapsed:    1.8s
[Parallel(n_jobs=12)]: Done 100 out of 100 | elapsed:    5.8s finished
```

```
forest_confusion = confusion_matrix(actual, forest_predictions)
forest_confusion
```

```
array([[2173538,   21349],
       [     87,      49]], dtype=int64)
```

The random forest had the least false positives out of all of the models, but also the least true negatives. This model is likely the safest to come out of the project, but ultimately among the least useful in a high-risk problem like detecting HDD failure.

```
forest_precision = precision_score(actual, forest_predictions)
forest_precision
```

```
0.0022899336386578185
```

```
print(classification_report(actual, forest_predictions))
```

```
              precision    recall  f1-score   support

       False       1.00      0.99      1.00   2194887
        True       0.00      0.36      0.00       136

    accuracy                           0.99   2195023
   macro avg       0.50      0.68      0.50   2195023
weighted avg       1.00      0.99      1.00   2195023
```

```
forest_probabilities = forest.predict_proba(x_test)
predictions = forest_probabilities[:,1]
```

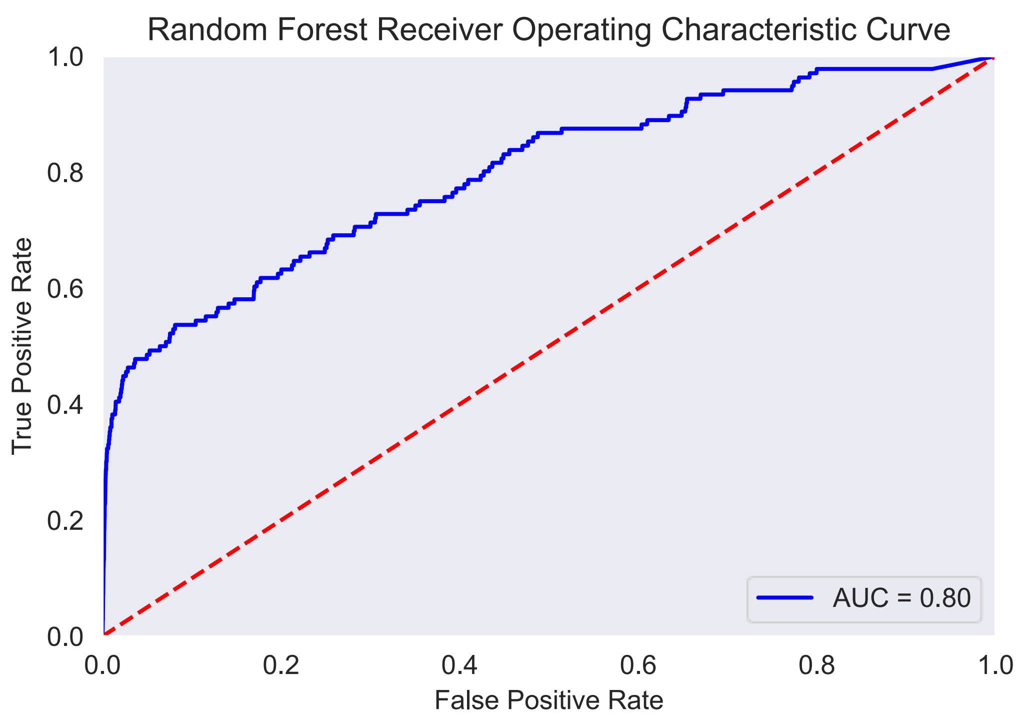
```
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent wor
kers.
[Parallel(n_jobs=12)]: Done  26 tasks      | elapsed:    1.9s
[Parallel(n_jobs=12)]: Done 100 out of 100 | elapsed:    6.2s finished
```

```
forest_false_positive_rate, forest_true_positive_rate, threshold =\
    roc_curve(y_test, predictions)
```

```
forest_roc_auc = auc(forest_false_positive_rate, forest_true_positive_rate)
forest_roc_auc
```

```
0.7974132039599305
```

```
plt.title('Random Forest Receiver Operating Characteristic Curve')
plt.plot(forest_false_positive_rate, forest_true_positive_rate, 'blue',
         label = 'AUC = %0.2f' % forest_roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.savefig('Charts/Forest ROC AUC.png')
plt.savefig('Charts/Forest ROC AUC.svg')
plt.show()
```



To improve the random forest ensemble, a second ensemble was created to prioritize the true negative results by weighting class failures as twice as important as non-failures.

```
weighted_forest = RandomForestClassifier(max_depth = 20, verbose = 1, \
        random_state = 13, n_jobs = -1, class_weight = {0: 1, 1: 2})
```

```
weighted_forest.fit(x_train, y_train.values.ravel())
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 12 concurrent wor
kers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed: 15.7min
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed: 44.5min finished
```

```
RandomForestClassifier(class_weight={0: 1, 1: 2}, max_depth=20, n_jobs=-1,
                       random_state=13, verbose=1)
```

```
weighted_forest_accuracy = weighted_forest.score(x_test, y_test)
weighted_forest_accuracy
```

```
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent wor
kers.
[Parallel(n_jobs=12)]: Done  26 tasks      | elapsed:    1.8s
[Parallel(n_jobs=12)]: Done 100 out of 100 | elapsed:    5.9s finished
```

```
0.9717009798986161
```

```
weighted_forest_predictions = weighted_forest.predict(x_test)
actual = y_test
```

```
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent wor
kers.
[Parallel(n_jobs=12)]: Done  26 tasks      | elapsed:    2.3s
[Parallel(n_jobs=12)]: Done 100 out of 100 | elapsed:    6.5s finished
```

```
weighted_forest_confusion = confusion_matrix(actual, \
                                             weighted_forest_predictions)
weighted_forest_confusion
```

```
array([[2132851,   62036],
       [     81,      55]], dtype=int64)
```

Compared to the unweighted Random Forest ensemble, this weighted ensemble gains another 6 true negative classifications for a true negative rate of 40% rather than 36%, but also gains 40,687 false positive classifications, for 0.028%, instead of 0.0097% false positives.

```
weighted_forest_precision = precision_score(actual, \
                                            weighted_forest_predictions)
weighted_forest_precision
```

```
0.0008857966532991899
```

```
print(classification_report(actual, weighted_forest_predictions))
```

```
              precision    recall  f1-score   support

         0.0       1.00      0.97      0.99   2194887
         1.0       0.00      0.40      0.00       136

    accuracy                           0.97   2195023
   macro avg       0.50      0.69      0.49   2195023
weighted avg       1.00      0.97      0.99   2195023
```

```
weighted_forest_probabilities = weighted_forest.predict_proba(x_test)
weighted_predictions = weighted_forest_probabilities[:,1]
```

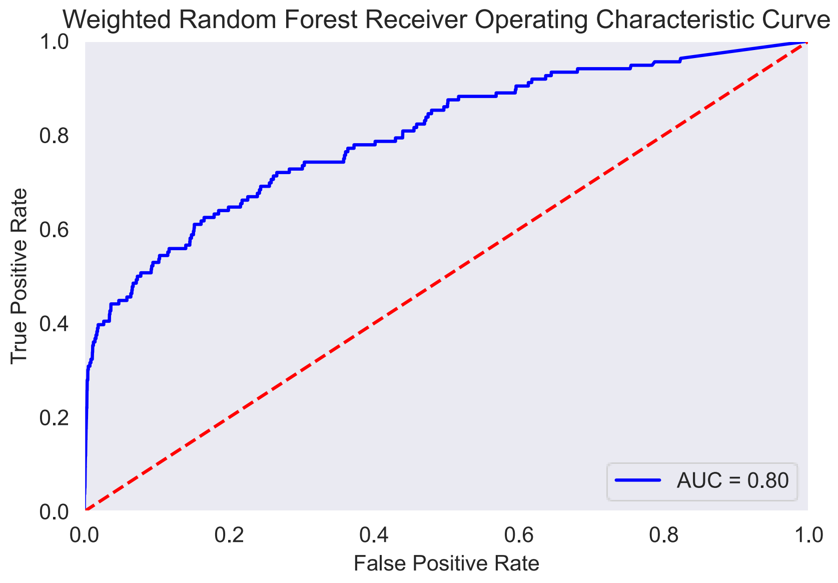
```
[Parallel(n_jobs=12)]: Using backend ThreadingBackend with 12 concurrent wor
kers.
[Parallel(n_jobs=12)]: Done  26 tasks      | elapsed:    1.8s
[Parallel(n_jobs=12)]: Done 100 out of 100 | elapsed:    6.2s finished
```

```
weighted_forest_false_positive_rate, weighted_forest_true_positive_rate, \
                    threshold = roc_curve(y_test, weighted_predictions)
```

```
weighted_forest_roc_auc = auc(weighted_forest_false_positive_rate, \
                              weighted_forest_true_positive_rate)
weighted_forest_roc_auc
```

```
0.799827248576833
```

```
plt.title('Weighted Random Forest Receiver Operating Characteristic Curve')
plt.plot(weighted_forest_false_positive_rate, \
         weighted_forest_true_positive_rate, \
         'blue', label = 'AUC = %0.2f' % weighted_forest_roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.savefig('Charts/Weighted Forest ROC AUC.png')
plt.savefig('Charts/Weighted Forest ROC AUC.svg')
plt.show()
```



Finally, two neural networks were built using PyTorch. The first was a simpler architecture and the second a more complex architecture, but both were deep neural networks (DNN) and implementations of multi-layer perceptrons (MLP). PyTorch requires the boolean values to be converted to floating point data, so the dtypes in all datasets were changed before the neural networks were defined.

```
for col in x_train:
    if x_train[col].dtype == "bool":
        x_train[col] = x_train[col].astype(float)
        x_test[col] = x_test[col].astype(float)
```

```
y_train = y_train.astype(float)
y_test = y_test.astype(float)
```

Once the data was in the appropriate form, the training and testing sets were loaded into tensors and dataloaders were formed from the tensors.

```
train_label = torch.tensor(y_train.values)
trainset = torch.tensor(x_train.values)
train_tensor = data_utils.TensorDataset(trainset, train_label)
trainloader = data_utils.DataLoader(dataset = train_tensor, \
                                    batch_size = 512, shuffle = True)


test_label = torch.tensor(y_test.values)
testset = torch.tensor(x_test.values)
test_tensor = data_utils.TensorDataset(testset, test_label)
testloader = data_utils.DataLoader(dataset = test_tensor, \
                                   batch_size = 512, shuffle = True)
```

GPU-accelerated training via CUDA was used for training these networks and once confirmed available, the GPU was assigned as the device to move tensors to for calculations.

```
torch.backends.cudnn.enabled
```

```
True
```

```
torch.cuda.is_available()
```

```
True
```

```
print(torch.version.cuda)
```

```
10.2
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device
```

```
device(type='cuda', index=0)
```

The first MLP was defined to go from 49 input nodes to 24 hidden nodes to 12 hidden nodes and then to 1 output node with leaky ReLU activation functions on the input and hidden layers. The output activation function was a sigmoid as this was a binary classification task. The criterion used was the Binary Cross Entropy Loss, or BCELoss criterion. The optimizer was the Adam algorithm with a very low learning rate as the dataset was quite large. The MLP’s weights were then initialized with Xavier, or Glorot, initialization.

```
class nn_Classifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(49, 24)
        self.act1 = nn.LeakyReLU()
        self.fc2 = nn.Linear(24, 12)
        self.act2 = nn.LeakyReLU()
        self.fc3 = nn.Linear(12, 1)

    def forward(self, x):
        # make sure input tensor is flattened
        x = x.view(-1, 49)

        x = self.act1(self.fc1(x))
        x = self.act2(self.fc2(x))
        x = torch.sigmoid(self.fc3(x))

        return x
```

```
neural_network = nn_Classifier()
criterion = nn.BCELoss()
optimizer = optim.Adam(neural_network.parameters(), lr = 1e-7, \
                       weight_decay = 1e-5)
```

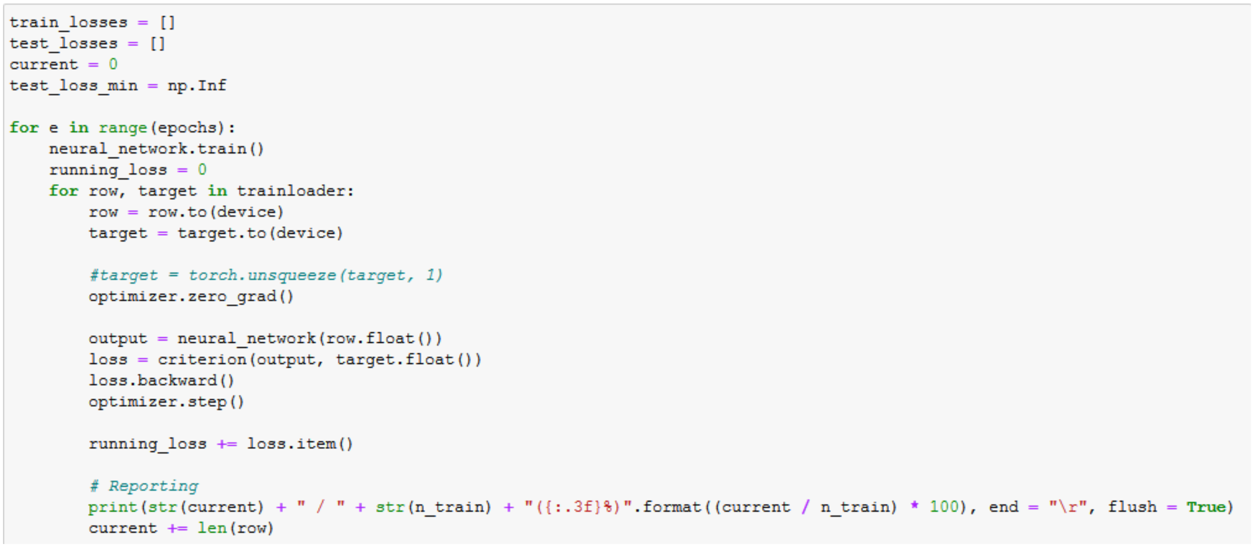
```
def init_weights(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_uniform_(m.weight)
        m.bias.data.fill_(0.01)

neural_network.apply(init_weights)
```

```
nn_Classifier(
  (fc1): Linear(in_features=49, out_features=24, bias=True)
  (act1): LeakyReLU(negative_slope=0.01)
  (fc2): Linear(in_features=24, out_features=12, bias=True)
  (act2): LeakyReLU(negative_slope=0.01)
  (fc3): Linear(in_features=12, out_features=1, bias=True)
)
```

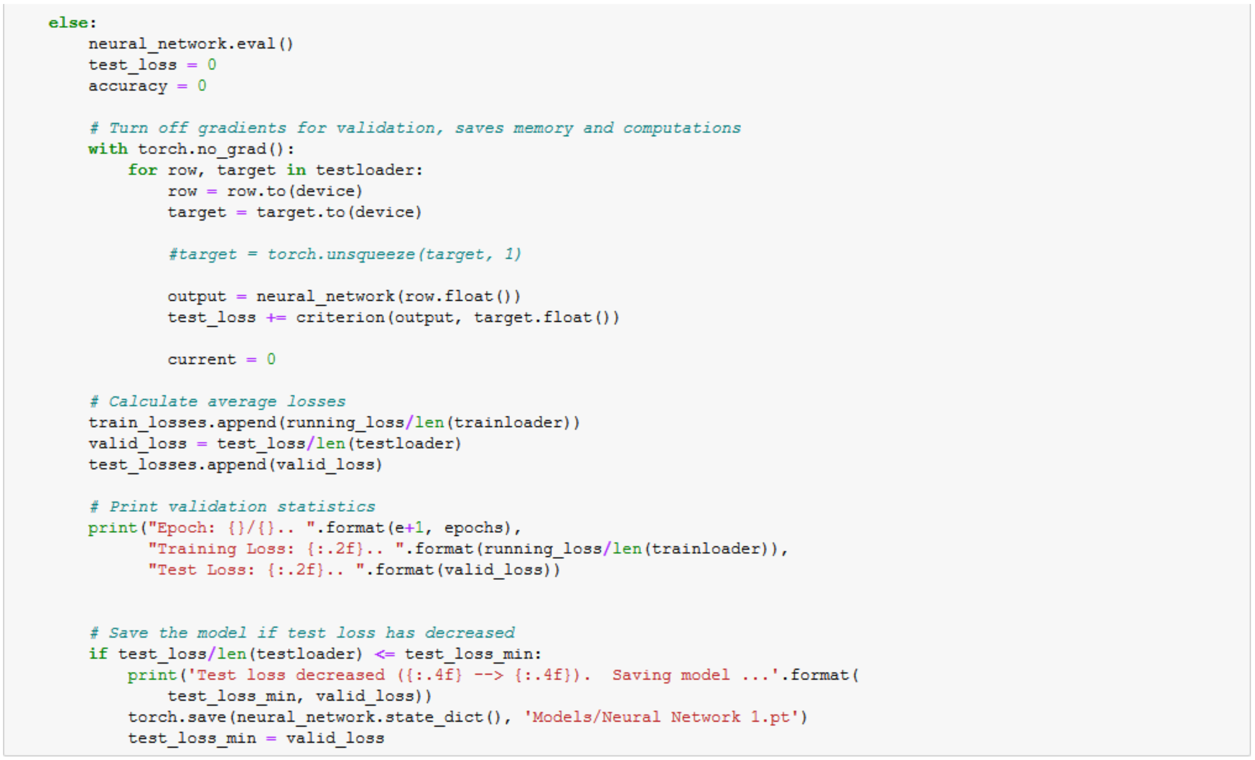
Finally, the number of epochs to train for was set to 10 and the model was moved to GPU memory.

```
n_train = len(x_train)
epochs = 10
neural_network.to(device);
```

The training loop is shown in the two screenshots below.

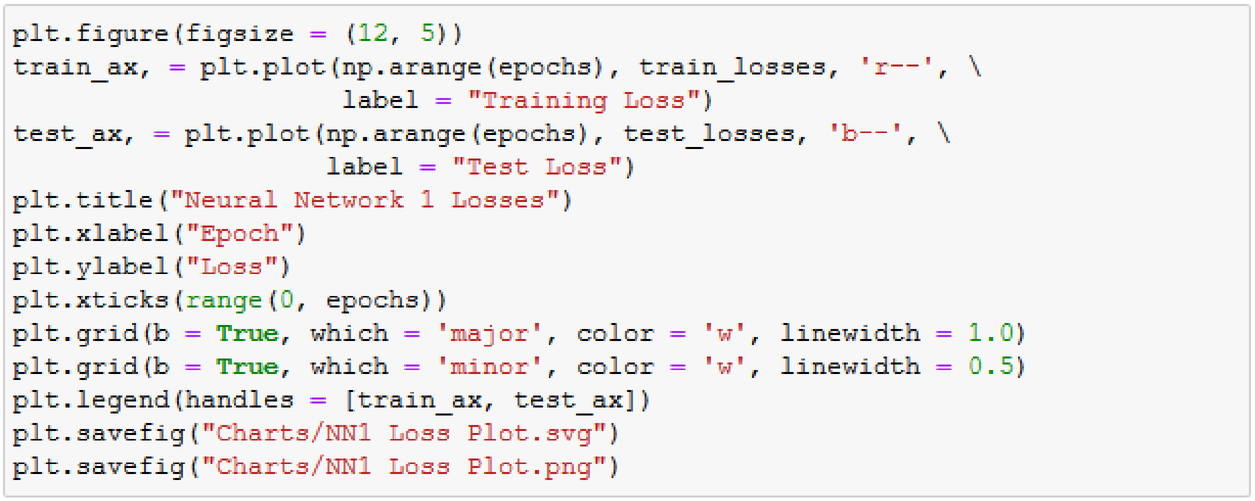


```
train_losses = []
test_losses = []
current = 0
test_loss_min = np.Inf

for e in range(epochs):
    neural_network.train()
    running_loss = 0
    for row, target in trainloader:
        row = row.to(device)
        target = target.to(device)

        #target = torch.unsqueeze(target, 1)
        optimizer.zero_grad()

        output = neural_network(row.float())
        loss = criterion(output, target.float())
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        # Reporting
        print(str(current) + " / " + str(n_train) + "({:.3f}%)".format((current / n_train) * 100), end = "\r", flush = True)
        current += len(row)
```

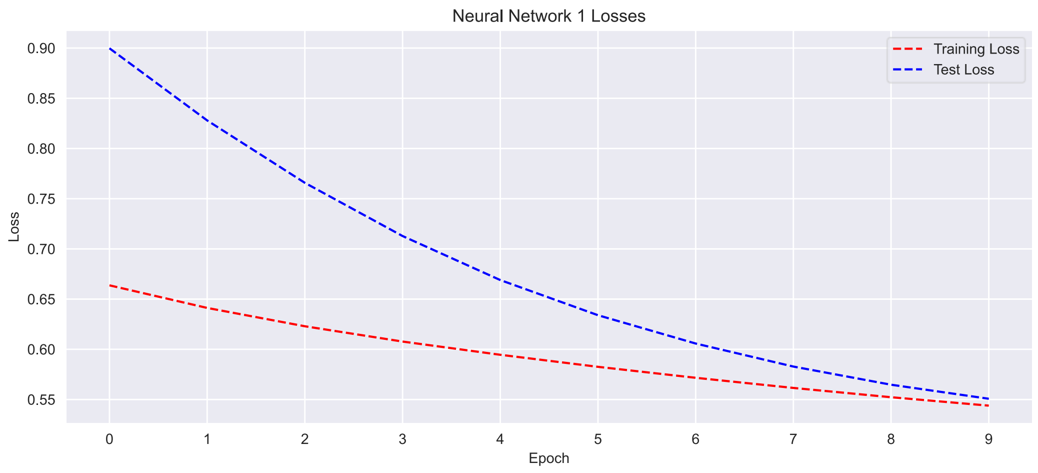


```
    else:
        neural_network.eval()
        test_loss = 0
        accuracy = 0

        # Turn off gradients for validation, saves memory and computations
        with torch.no_grad():
            for row, target in testloader:
                row = row.to(device)
                target = target.to(device)

                #target = torch.unsqueeze(target, 1)

                output = neural_network(row.float())
                test_loss += criterion(output, target.float())

                current = 0

        # Calculate average losses
        train_losses.append(running_loss/len(trainloader))
        valid_loss = test_loss/len(testloader)
        test_losses.append(valid_loss)

        # Print validation statistics
        print("Epoch: {}/{}.. ".format(e+1, epochs),
              "Training Loss: {:.2f}.. ".format(running_loss/len(trainloader)),
              "Test Loss: {:.2f}.. ".format(valid_loss))


        # Save the model if test loss has decreased
        if test_loss/len(testloader) <= test_loss_min:
            print('Test loss decreased ({:.4f} --> {:.4f}).  Saving model ...'.format(
                test_loss_min, valid_loss))
            torch.save(neural_network.state_dict(), 'Models/Neural Network 1.pt')
            test_loss_min = valid_loss
```

Once the network had trained and tested through the 10 epochs, the training and test losses were graphed.



```
plt.figure(figsize = (12, 5))
train_ax, = plt.plot(np.arange(epochs), train_losses, 'r--', \
                     label = "Training Loss")
test_ax, = plt.plot(np.arange(epochs), test_losses, 'b--', \
                    label = "Test Loss")
plt.title("Neural Network 1 Losses")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.xticks(range(0, epochs))
plt.grid(b = True, which = 'major', color = 'w', linewidth = 1.0)
plt.grid(b = True, which = 'minor', color = 'w', linewidth = 0.5)
plt.legend(handles = [train_ax, test_ax])
plt.savefig("Charts/NN1 Loss Plot.svg")
plt.savefig("Charts/NN1 Loss Plot.png")
```



In order to score the MLP in the same way that the scikit-learn models were scored, the model was given the test data and the output recorded.

```
neural_network.eval()
output = []
pred_targets = []

with torch.no_grad():
    for rows, targets in testloader:
        rows = rows.to(device)

        output += neural_network(rows.float())
        pred_targets += targets
```

The output was then adjusted to the binary predictions expected by the scoring functions by setting a false prediction as less than or equal to 0.5 and a true predication as all other output.

```
nn1_predictions = []
actual = []

for i, x in enumerate(output):
    if output[i].item() <= 0.5:
        nn1_predictions.append(0)
    else:
        nn1_predictions.append(1)

    if pred_targets[i].item() == 0.0:
        actual.append(0)
    elif pred_targets[i].item() == 1.0:
        actual.append(1)
```

```
nn1_confusion = confusion_matrix(actual, nn1_predictions)
nn1_confusion
```

```
array([[2016112,  178775],
       [     52,      84]], dtype=int64)
```

The results are relatively good. The 84 true negatives were second only to the logistic regression, but the 178,775 false positives were over double all other models.

```
nn1_precision = precision_score(actual, nn1_predictions)
nn1_precision
```

```
0.0004696436858083742
```

```
print(classification_report(actual, nn1_predictions))
```

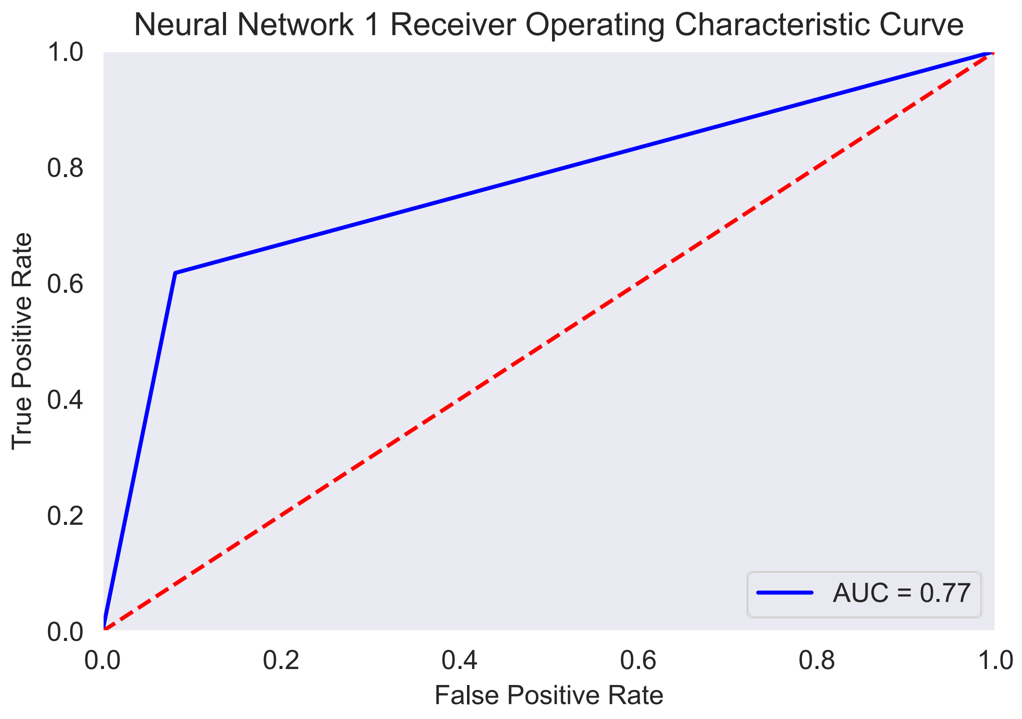
```
              precision    recall  f1-score   support

           0       1.00      0.92      0.96   2194887
           1       0.00      0.62      0.00       136

    accuracy                           0.92   2195023
   macro avg       0.50      0.77      0.48   2195023
weighted avg       1.00      0.92      0.96   2195023
```

```
nn1_false_positive_rate, nn1_true_positive_rate, threshold =\
    roc_curve(actual, nn1_predictions)
```

```
nn1_roc_auc = auc(nn1_false_positive_rate, nn1_true_positive_rate)
nn1_roc_auc
```

```
0.7680981982215941
```

```
plt.title('Neural Network 1 Receiver Operating Characteristic Curve')
plt.plot(nn1_false_positive_rate, nn1_true_positive_rate, 'blue',
         label = 'AUC = %0.2f' % nn1_roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.savefig('Charts/NN1 ROC AUC.png')
plt.savefig('Charts/NN1 ROC AUC.svg')
plt.show()
```

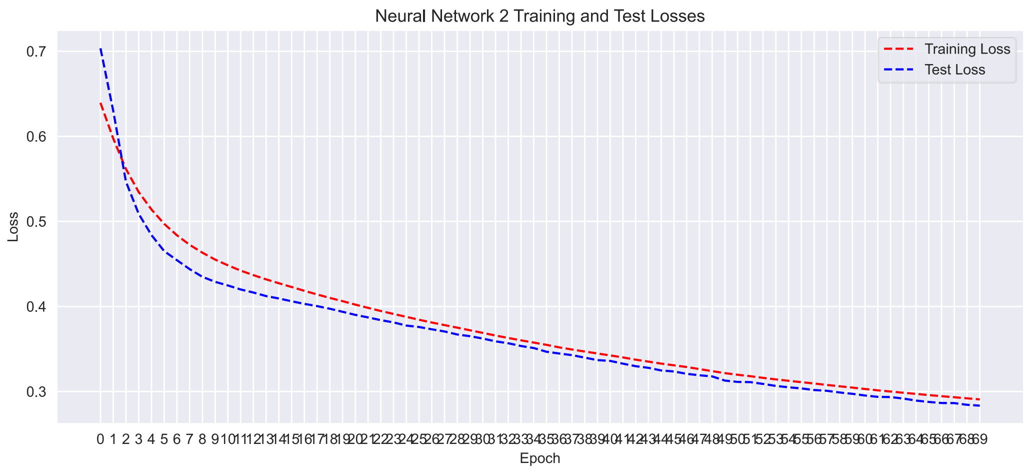


This neural network is a good model, but it is likely too simple of an architecture for this scale of problem. A second MLP was then defined to go from 49 input nodes to 98 hidden nodes to 72 hidden nodes to 36 hidden nodes to 9 hidden nodes and then finally to 1 output node with leaky ReLU activation functions on the input and hidden layers. The output activation function, criterion, optimizer, and weight initialization were the same as the first MLP. The number of epochs for training however differed, instead being set to 70.

```
class nn_Classifier2(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(49, 98)
        self.act1 = nn.LeakyReLU()
        self.fc2 = nn.Linear(98, 72)
        self.act2 = nn.LeakyReLU()
        self.fc3 = nn.Linear(72, 36)
        self.act3 = nn.LeakyReLU()
        self.fc4 = nn.Linear(36, 9)
        self.act4 = nn.LeakyReLU()
        self.fc5 = nn.Linear(9, 1)

    def forward(self, x):
        # make sure input tensor is flattened
        x = x.view(-1, 49)

        x = self.act1(self.fc1(x))
        x = self.act2(self.fc2(x))
        x = self.act3(self.fc3(x))
        x = self.act4(self.fc4(x))
        x = torch.sigmoid(self.fc5(x))

        return x
```

Once defined, the new MLP was trained using the same training and testing loop as the first. The losses were then graphed in the same way. Finally, the same prediction conversion and scoring was performed.

```
plt.figure(figsize = (18, 5))
train_ax, = plt.plot(np.arange(epochs + 20), train_losses, 'r--', \
                     label = "Training Loss")
test_ax, = plt.plot(np.arange(epochs + 20), test_losses, 'b--',\
                    label = "Test Loss")
plt.title("Neural Network 2 Training and Test Losses")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.xticks(range(0, epochs + 20))
plt.grid(b = True, which = 'major', color = 'w', linewidth = 1.0)
plt.grid(b = True, which = 'minor', color = 'w', linewidth = 0.5)
plt.legend(handles = [train_ax, test_ax])
plt.savefig("Charts/NN2 Loss Plot 2.svg")
plt.savefig("Charts/NN2 Loss Plot 2.png")
```



```
nn2_confusion = confusion_matrix(actual, nn2_predictions)
nn2_confusion
```

```
array([[2055329,  139558],
       [     39,      97]], dtype=int64)
```

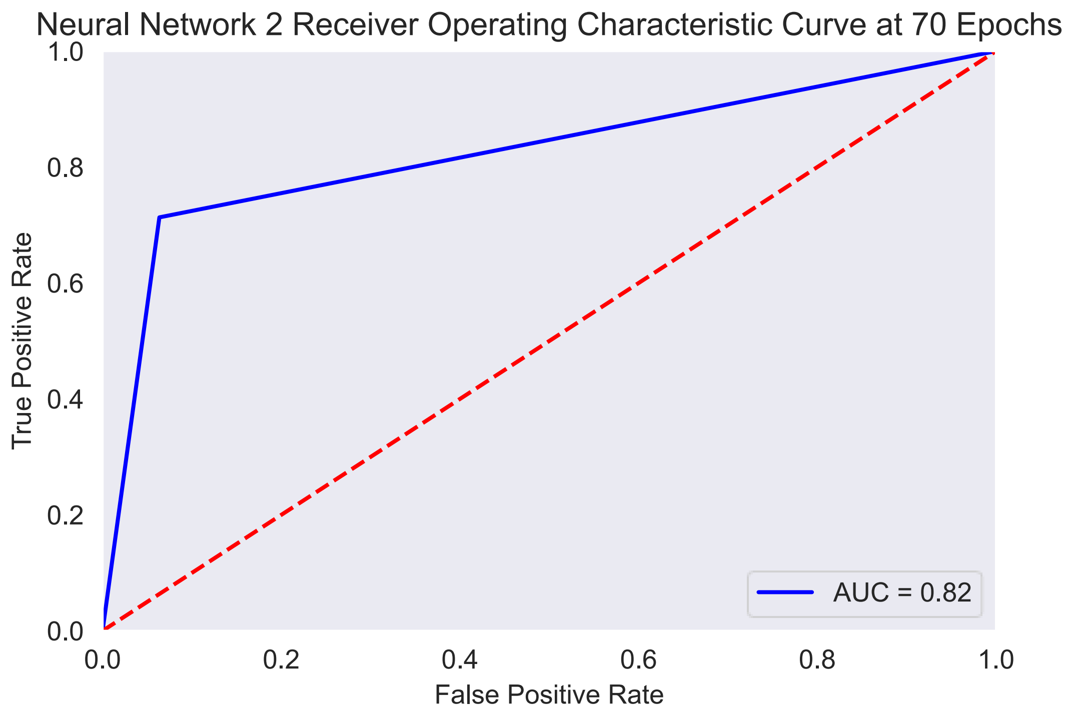
```
nn2_precision = precision_score(actual, nn2_predictions)
nn2_precision
```

```
0.0006945687587268626
```

```
print(classification_report(actual, nn2_predictions))
```

```
              precision    recall  f1-score   support

           0       1.00      0.94      0.97   2194887
           1       0.00      0.71      0.00       136

    accuracy                           0.94   2195023
   macro avg       0.50      0.82      0.48   2195023
weighted avg       1.00      0.94      0.97   2195023
```

The scores for this second neural network are the best for the task out of all the tested models. 71.32% of failures were correctly predicted and using a more complex network architecture reduced the false positive count by 39,217, or 21.93%, from the first MLP.



With the logistic regression model appearing to be the best ratio of true negative predictions to false positive predictions, it is chosen to be test on the validation dataset.

```
regression_accuracy = regression.score(x_valid, y_valid)
regression_accuracy
```

```
0.9732312477961497
```

```
regression_predictions = regression.predict(x_valid)
actual = y_valid
```

```
regression_confusion = confusion_matrix(actual, regression_predictions)
regression_confusion
```

```
array([[1068088,   29355],
       [     24,      44]], dtype=int64)
```

The logistic regression model proved to be an excellent solution for the task of predicting hard drive failure. The false positives are quite low, and the 66.6% of failure instances were correctly predicted.

```
regression_precision = precision_score(actual, regression_predictions)
regression_precision
```

```
0.001496649545902922
```

```
print(classification_report(actual, regression_predictions))
```

```
              precision    recall  f1-score   support

       False       1.00      0.97      0.99   1097443
        True       0.00      0.65      0.00        68

    accuracy                           0.97   1097511
   macro avg       0.50      0.81      0.49   1097511
weighted avg       1.00      0.97      0.99   1097511
```

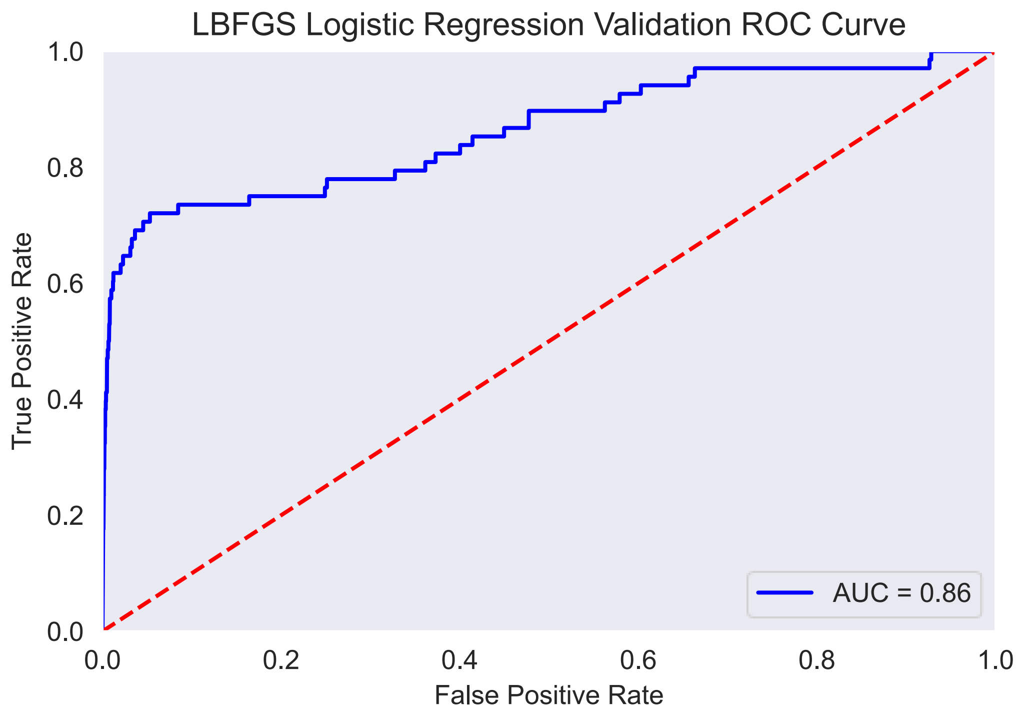
```
regression_probabilities = regression.predict_proba(x_valid)
predictions = regression_probabilities[:,1]
```

```
regression_false_positive_rate, regression_true_positive_rate, threshold =\
    roc_curve(y_valid, predictions)
```

```
regression_roc_auc = auc(regression_false_positive_rate, \
                         regression_true_positive_rate)
regression_roc_auc
```

```
0.8627243456996373
```

```
plt.title('LBFGS Logistic Regression Validation ROC Curve')
plt.plot(regression_false_positive_rate, regression_true_positive_rate, \
         'blue', label = 'AUC = %0.2f' % regression_roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.savefig('Charts/LBFGS Logistic Validation ROC AUC.svg')
plt.savefig('Charts/LBFGS Logistic Validation ROC AUC.png')
plt.show()
```



Data Summary and Implications

Throughout this study, six different models were created and trained for predictive analysis of HDD failure to determine if the study factors significantly indicate impending hard disk drive failure. As the analysis showed, all of the study factors are statistically significant and useful for predicting HDD failure before it occurs. Based on the results of the exploratory data analysis and the predictive analysis, the SMART attributes 5, 197, and 9 are the best predictors for HDD failure. The manufacturer is also very influential when combined with the other data.

The logistic regression model or the more complex MLP neural network are the two best model approaches for attempting to automate a data center’s approach to predicting HDD failure before it happens so that the drive can be backed up and retired before the data is lost. The more complex MLP neural network is the best approach to successfully predicting failure but does have the drawback of a relatively large amount of false positive results. If this is a concern, the logistic regression model is a close second in successful failure prediction with less than half of the false positive rate. It is recommended that either of these models should be added to a production backup pipeline so that they can assist in automating the flagging of drives at risk of failing before the failure occurs.

A few limitations of this project exist. First, a very large amount of the dataset was made up of missing values. A more complete dataset would greatly improve the accuracy and ability of the predictive models and allow for more possibilities of key predictors as many columns of data had to be dropped from the dataset because of their missing values. Another limitation that deserves caution is that the ratios of drives made by each manufacturer in the dataset is very imbalanced. No assumptions about value or reliability of the four manufacturers included in the dataset should be made from this study.

A third limitation is that the dataset was extremely imbalanced in terms of the minority (failure) and majority (non-failure) classes. Though SMOTE succeeded exceptionally well at allowing predictive models to learn from the imbalanced data, it does introduce bias as the synthetically created instances of the minority classes overrepresent their information in the analysis. Though difficult given the rarity of HDD failure, more instances of failure would improve the results and predictive power of this analysis. Finally, working computer memory was a great limitation throughout the project as the dataset is so large. This limitation prevented factor analysis of mixed data from being performed and PCA had to be selected as the alternative.

Several ways to continue and improve this study exist. As computing power was a limiting factor, only a few hyperparameter combinations per model could be tested. Scikitlearns' grid_search_cv would be an excellent way to set up, train, and test combinations of model hyperparameters to further improve each model. This would work especially well to improve the decision tree and random forest models.

Though it would create far more complexity, rerunning the analysis and recreating the predictive models for each of the manufacturers would remove nearly all of the difficulty in filling NaN values, as most missing values are tied to the manufacturer's implementation of SMART. The model column would be able to be kept in its stead, specializing the information more for each drive. Additionally, it would allow for much greater specialization in predictions as a disproportionate amount of unexplainable variance in values came from the differences in drive manufacturer.

For smaller projects, continuations of the study include using different ratios of SMOTE rather than equalizing the failure class ratio, testing a class weighted random forest without or with reduced SMOTE, and testing different MLP neural network architectures with different learning rates.

Finally, the last major proposal for study continuation is to restructure the approach to the dataset and build up to using a recurrent neural network (RNN) in place of the other models and MLP neural networks. RNNs excel at working with time series data. This study's approach took the time series data and flattened it into hard drive days for simplicity of study and analysis rather than treating the data as time progression throughout the quarter. Though it would be substantially more difficult, the results would likely be unmatched by anything this study's current approach can result in.

References

Acronis. *Knowledge Base 9105*. S.M.A.R.T. Attribute: Reallocated Sectors Count | Knowledge Base. https://kb.acronis.com/content/9105.

Acronis. *Knowledge Base 9109*. S.M.A.R.T. Attribute: Power-On Hours (POH) | Knowledge Base. https://kb.acronis.com/content/9109.

Acronis. *Knowledge Base 9128*. S.M.A.R.T. Attribute: Load Cycle Count; Load/Unload Cycle Count | Knowledge Base. https://kb.acronis.com/content/9128.

Acronis. *Knowledge Base 9133*. S.M.A.R.T. Attribute: Current Pending Sector Count | Knowledge Base. https://kb.acronis.com/content/9133.

Acronis. *Knowledge Base 9152*. S.M.A.R.T. Attribute: Load/Unload Cycle Count | Knowledge Base. https://kb.acronis.com/content/9152.

Backblaze. (2020). data_Q4_2019. San Mateo, CA; Backblaze.

Klein, A. (2015, April 16). *SMART Hard Drive Attributes: SMART 22 is a Gas Gas Gas*. Backblaze Blog | Cloud Storage & Cloud Backup. https://www.backblaze.com/blog/smart-22-is-a-gas-gas-gas/.

Painchaud, A. (2018, October 31). *8 Reasons on How Data Loss Can Negatively Impact Your Bussiness*. https://www.sherweb.com/blog/security/statistics-on-data-loss/.

Sanders, J. (2018, November 13). *Western Digital spins down HGST and Tegile brands in hard disk market shuffle*. TechRepublic. https://www.techrepublic.com/article/western-digital-spins-down-hgst-and-tegile-brands-in-hard-disk-market-shuffle/.

Weiss, G. M. (2013). Foundations of Imbalanced Learning. *Imbalanced Learning*, 13–41. https://doi.org/10.1002/9781118646106.ch2