Prepare Environment

Install/Update Libraries

```
# Update the existing libraries
!pip install -U datasets huggingface_hub fsspec
```

Show hidden output

Import Libraries

```
# Libraries for data loading and preprocessing
from datasets import load_dataset
import copy
import pandas as pd
import numpy as np

# Libraries for model training / evaluation
import torch
import torch.nn as nn
from torch.optim import AdamW
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from torch.optim.lr_scheduler import ReduceLROnPlateau

from sklearn.ensemble import IsolationForest
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, roc_auc_score,confusion_matri
import xgboost as xgb


# Visualization libraries
import matplotlib.pyplot as plt
import seaborn as sns
```

› Set up environment to use GPU if available

[ ] ↳ 1 cell hidden

Overview

Given that a key component of PCs are its hard disk, where data is stored and where read write operation happens, its logs and statistics will be the focus of this project.

Key SMART codes will be used to create new features and train an Autoencoder. This unsupervised learning method will then predict anomalous data and subsequently evaluated.

Additionally, we will only focus on one brand of hard disks from this dataset, with the following assumptions:

1. Dell only mounts hard disks from a handfull of brands. e.g. Dell mounts its proprietary hard disks to its PCs.
2. Disks from different brands would have different benchmarks. e.g. Brand A having a temperature of 80 degree celcius may not be considered high and doesn't contribute to its chances failure. However, the same cannot be the said for Brand B, whereby 80 degree celcius would be considered high and contributes to its chances of failure.
3. Assume that different series of hard disk matter, for example, a 10TB hard disk of different type say SATA or M.2 would also mean different threshold of failure
4. In this example, we selected `WDC` and assume that this brand of hard disk despite its many series, provide hard disk that behave in a similar way due to proprietary software / hardware features

Thus, the goal of this project is to predict if a particular hard disk would fail under the `WDC` brand

⌄ Exploratory Data Analysis & Preprocessing

Loading Data

In this section, we will be loading data from huggingface dataset. We are interested in the backblaze driver stats as it is the Self-Monitoring, Analysis, and Reporting Technology (SMART) report of hard disks of different models and serial numbers across multiple dates.

```
# Define the dataset for 2016 Q1 and Q2
data_files = [
    "zip_csv/2016_Q1.zip",
    "zip_csv/2016_Q2.zip",
]

# Load the dataset for 2016 Q1 and Q2
ds = load_dataset("backblaze/Drive_Stats", data_files=data_files)

# Define the dataset for 2016 Q1 and Q2
data_files_q34 = [
    "zip_csv/2016_Q3.zip",
    "zip_csv/2016_Q4.zip",
]

# Load the dataset for 2016 Q3 and Q4
ds34 = load_dataset("backblaze/Drive_Stats", data_files=data_files_q34)

# save a copy of the original dataframe.
ds1 = copy.deepcopy(ds)

# Retrieve the dataset
ds = ds["train"]
ds34 = ds34['train']
```

Loading dataset shards: 100% 19/19 [00:38<00:00, 1.58s/it]

Loading dataset shards: 100% 21/21 [00:43<00:00, 1.63s/it]

Double-click (or enter) to edit

```
print("Initial dataset metrics:")
ds.info
```

Show hidden output

Data Preparation

Based on research, these are the key SMART features which contributes to disk failures. As such we will extract these columns.

```
# Rename SMART codes with their actual meaning
# Mapping found from:


def select_rename_columns(ds,smart_attributes_to_keep, smart_column_mapping):
  smart_columns_to_keep = [f'smart_{attr}_normalized' for attr in smart_attribu
  other_columns_to_keep = ['date', 'serial_number', 'model', 'capacity_bytes',
  all_columns_to_keep = other_columns_to_keep + smart_columns_to_keep
  ds_cleaned = ds.select_columns(all_columns_to_keep)
  ds_renamed = ds_cleaned.rename_columns(smart_column_mapping)
  return ds_renamed

smart_attributes_to_keep = [1,5,9,10,12,183,184,187,188,189,192,193,194,196,197
smart_column_mapping = {
    'smart_1_normalized': 'smart_read_error_rate',
    'smart_5_normalized': 'smart_reallocated_sector_count',
    'smart_9_normalized': 'smart_power_on_hours',
    'smart_10_normalized': 'smart_spin_retry_count',
    'smart_12_normalized': 'smart_power_cycle_count',
    'smart_183_normalized': 'smart_sata_downshift_error_count',
    'smart_184_normalized': 'smart_end_to_end_error_ioedc',
    'smart_187_normalized': 'smart_reported_uncorrectable_errors',
    "smart_188_normalized": "smart_command_timeout",
    "smart_189_normalized": "smart_high_fly_writes",
    'smart_192_normalized': 'smart_power_off_retract_count',
    'smart_193_normalized': 'smart_load_cycle_count',
    'smart_194_normalized': 'smart_temperature_celsius',
    'smart_196_normalized': 'smart_reallocation_event_count',
    'smart_197_normalized': 'smart_current_pending_sector_count',
    'smart_198_normalized': 'smart_uncorrectable_sector_count',
    "smart_199_normalized":"smart_count_data_transfer_error"
}

ds_renamed = select_rename_columns(ds,smart_attributes_to_keep,smart_column_map
ds_renamed34 = select_rename_columns(ds34,smart_attributes_to_keep,smart_column
```

As mentioned in the Overview, Dell will only mount disks from certain brands. In this project, the `WDC` hard disks is the example.

```
# Retrieve only storage disks from WDC
ds_wdc = ds_renamed.filter(lambda example: 'WDC' in example['model'])
ds_wdc34 = ds_renamed34.filter(lambda example: 'WDC' in example['model'])


# Create categories of hard disks based on it's size to reduce the sparsity of

def categorize_model_family(example):
    model = example['model']
    model_family = 'WDC Other' # Default value
    if not isinstance(model, str):
        model_family = 'WDC Other'
    elif 'WDC WD10' in model:
        model_family = 'WDC 1TB Family'
    elif 'WDC WD20' in model:
        model_family = 'WDC 2TB Family'
    elif 'WDC WD30' in model:
        model_family = 'WDC 3TB Family'
    elif 'WDC WD40' in model:
        model_family = 'WDC 4TB Family'
    elif 'WDC WD50' in model:
        model_family = 'WDC 5TB Family'
    elif 'WDC WD60' in model:
        model_family = 'WDC 6TB Family'
    elif 'WDC WD80' in model:
          model_family = 'WDC 8TB Family'
    elif 'WDC WD100' in model:
          model_family = 'WDC 10TB Family'
    elif 'WDC WD120' in model:
          model_family = 'WDC 12TB Family'
    return {'model_family': model_family}

ds_wdc = ds_wdc.map(categorize_model_family)
ds_wdc34 = ds_wdc34.map(categorize_model_family)



# Convert to pandas data frame for preprocessing operations.
df_wdc = ds_wdc.to_pandas()
df_charts = ds_wdc.to_pandas()
df_wdc34_full = ds_wdc34.to_pandas()
```

```
# Identify columns with missing values
missing_values_per_column = df_wdc.isnull().sum()
columns_with_missing_values = missing_values_per_column[missing_values_per_colu
print("Columns with missing data:")
columns_with_missing_values

# We will fill these columns with 0 later on
```

```
Columns with missing data:
```

	0
smart_sata_downshift_error_count	399549
smart_end_to_end_error_ioedc	399368
smart_reported_uncorrectable_errors	399368
smart_command_timeout	399368
smart_high_fly_writes	399549
smart_power_off_retract_count	968
smart_load_cycle_count	968
smart_temperature_celsius	181

dtype: int64

The `df_wdc34` variable is essential to capture more anomaly examples from the 2016 Q3 & Q4 unseen data. If the model is robust & generalizable, it should be able to differentiate between these examples well

```
# Retrieve only the failure columns from Q3 Q4 as additional validation data
df_wdc34 = df_wdc34_full[df_wdc34_full['failure'] == 1]
```

```
# Convert datetime format from String.
def convert_to_datetime_and_sort(df, date_col):
  df['date_datetime'] = pd.to_datetime(df[date_col])
  df = df.sort_values(by=['serial_number', 'date_datetime'])
  return df

df_wdc = convert_to_datetime_and_sort(df_wdc, 'date')
df_wdc34 = convert_to_datetime_and_sort(df_wdc34, 'date')
```

Show hidden output

```
# Retrieve the age of the drive
def calculate_drive_age(df):
  min_dates = df.groupby('serial_number')['date_datetime'].transform('min')
  df['drive_age_days'] = (df['date_datetime'] - min_dates).dt.days
  return df

df_wdc = calculate_drive_age(df_wdc)
df_wdc34 = calculate_drive_age(df_wdc34)


# Retrieve the maximum temp difference of the next n days.
# e.g.
# data: D1: 0, D2: 1, D3: 3, D4: 2, D5: 1, D6: 1, ...
# result: D1: 3, D2: 2, ...

def calculate_max_temp_difference(df, n = 5):
  df['min_temp'] = df.groupby('serial_number')['smart_temperature_celsius'].tra
  df['max_temp'] = df.groupby('serial_number')['smart_temperature_celsius'].tra
  df['temp_diff'] = df['max_temp'] - df['min_temp']
  df = df.drop(columns=['max_temp', 'min_temp'])
  return df

df_wdc = calculate_max_temp_difference(df_wdc)
df_wdc34 = calculate_max_temp_difference(df_wdc34)
```

```
smart_cols = list(smart_column_mapping.values())

# Create shift in data
def create_lag_feature(series: pd.Series, window: int) -> pd.Series:
  return series.shift(periods=window)

# Retrieve delta values based on the window sizes for look forward and look bac
def create_window_dataframe(df, smart_cols, window_start=1, window_end=6):
  windows = range(window_start, window_end)
  # Create lag and lead features for each SMART column and each window size
  for col in smart_cols:
    # Ensure the column exists before trying to create features
    if col in df.columns:
      print(f"Creating lag and lead features for: {col}")
      for window in windows:
          # Create lag feature
          df[f'{col}_lag_{window}'] = df.groupby('serial_number')[col].transfor
    else:
      print(f"Warning: Column '{col}' not found in DataFrame.")
  df.fillna(0, inplace=True)
  return df

# FILL NAN for the lead lag
df_wdc = create_window_dataframe(df_wdc, smart_cols)
df_wdc34 = create_window_dataframe(df_wdc34, smart_cols)
```

Show hidden output

```
# Data standardization
def dataframe_stats(df):
  print("\nShape of Normal Data (failure == 0):", normal_data.shape)
  print("Shape of Outlier Data (failure == 1):", outlier_data.shape)
  print("\nFirst 5 rows of Normal Data:")
  print(normal_data.head())
  print("\nFirst 5 rows of Outlier Data:")
  print(outlier_data.head())


def data_type_standardization(df):
  df.fillna(0, inplace=True)
  df['failure'] = df['failure'].astype(int)
  df['model_family_code'] = df['model_family'].astype('category').cat.codes
  # Separate data based on the 'failure' column
  normal_data = df[df['failure'] == 0].copy()
  outlier_data = df[df['failure'] == 1].copy()
  return normal_data, outlier_data

normal_data, outlier_data = data_type_standardization(df_wdc)
_, outlier_data34 = data_type_standardization(df_wdc34)

# dataframe_stats(normal_data)
# dataframe_stats(outlier_data)
# dataframe_stats(outlier_data34)
```

Data Insights

Correlation Analysis

The correlation analysis focuses on selecting relevant SMART features for hard drive failure prediction. Due to the large number of available SMART metrics (up to 255), only those identified as important through external research and expert references were chosen. The selection process relied on studies and articles—primarily from Backblaze and other technical resources—that highlight which SMART attributes are most commonly linked to drive failures in commercial environments.

The research reference are from:

1. Black Blaze hard drive dataset schemas
2. Better understanding SMART values
3. Making sense of SMART values
4. SMART values common statistics
5. Full BlackBlaze SMART metric list
6. Smart Metric hardware failure indications

After identifying the top few metrics, the next steps is to involve using these selected metrics to analyze and visualize data for further insights.

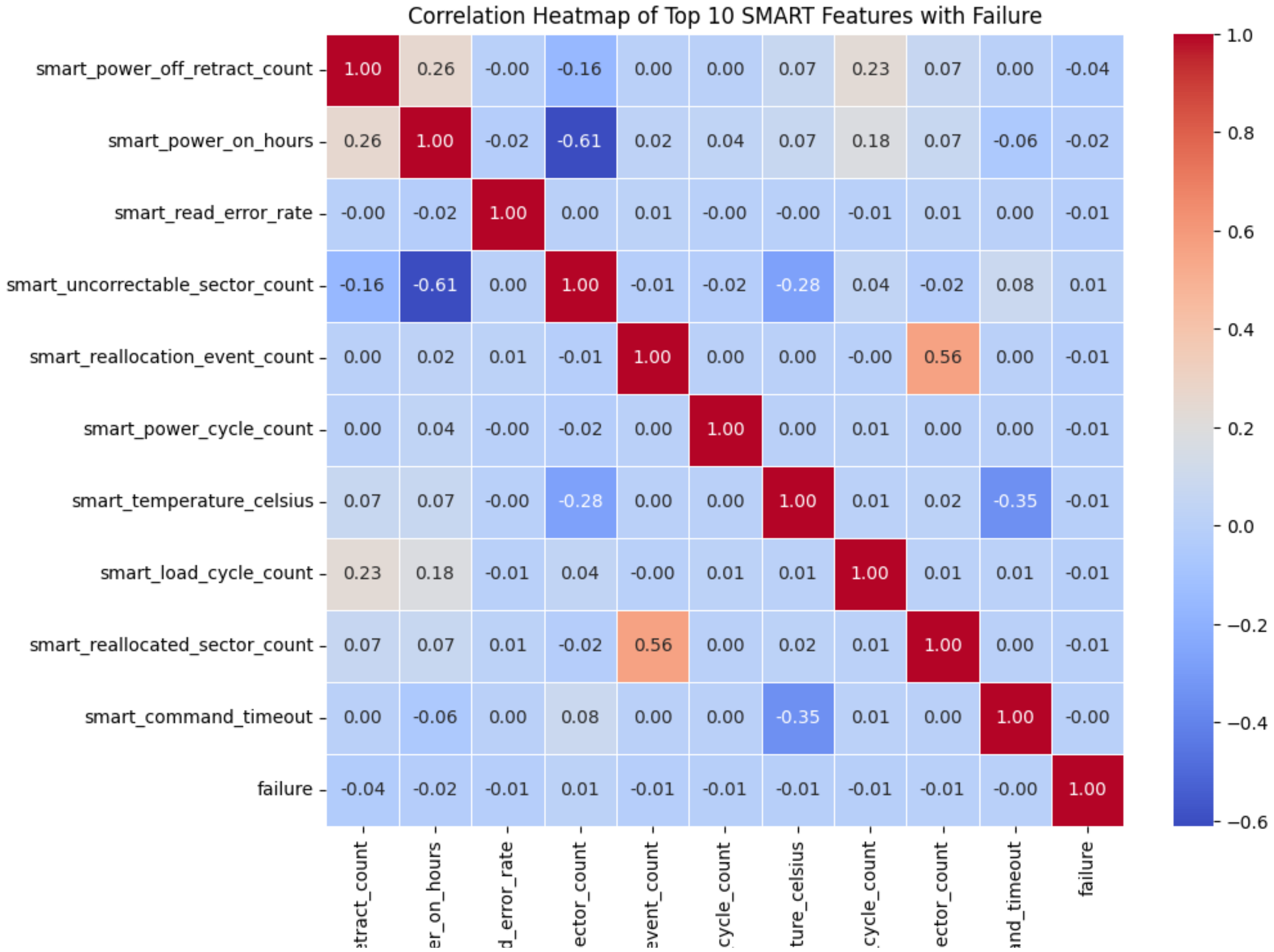
Findings:

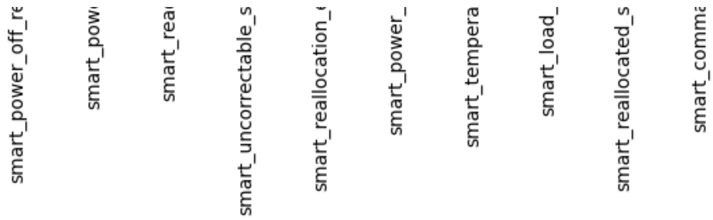
- We observer that the following SMART metrics seen in the list below are correlated to failures. These coincide with some of the findings from point 4 & 6 of the article. Additionally, while not extremely correlated, some of these metrics are also highly correlated to each other as seen in the heat map like power cycle and power retract
    - smart_power_on_hours 0.023456
    - smart_read_error_rate 0.013582
    - smart_uncorrectable_sector_count 0.013230

```
# Select only SMART columns (excluding lag features) and the failure column
smart_features_only = [col for col in df_wdc.columns if col.startswith('smart_'
correlation_cols = smart_features_only + ['failure']
df_correlation = df_wdc[correlation_cols].copy()
# Calculate the correlation matrix
correlation_matrix = df_correlation.corr()
failure_correlation = correlation_matrix['failure'].drop('failure') # Drop self
failure_correlation_sorted = failure_correlation.abs().sort_values(ascending=Fa

# Get the top 10 features correlated to failure
```

```
top_10_features = failure_correlation_sorted.head(10).index.tolist()

print("Top 10 SMART features correlated with failure (by absolute value):")
print(failure_correlation_sorted.head(10))

# Plot a correlation heatmap to show what are the features that best correlate
top_10_features_with_failure = top_10_features + ['failure']
correlation_matrix_top10 = df_correlation[top_10_features_with_failure].corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix_top10, annot=True, cmap='coolwarm', fmt=".2f", l
plt.title('Correlation Heatmap of Top 10 SMART Features with Failure')
plt.show()
```

```
Top 10 SMART features correlated with failure (by absolute value):
smart_power_off_retract_count       0.037007
smart_power_on_hours                0.023456
smart_read_error_rate               0.013582
smart_uncorrectable_sector_count    0.013230
smart_reallocation_event_count      0.013084
smart_power_cycle_count             0.011918
smart_temperature_celsius           0.008847
smart_load_cycle_count              0.007416
smart_reallocated_sector_count      0.005692
smart_command_timeout               0.000316
Name: failure, dtype: float64
```




smart_power_off_re
smart_powe
smart_rea
smart_uncorrectable_s
smart_reallocation_e
smart_power_
smart_tempera
smart_load_
smart_reallocated_s
smart_comma

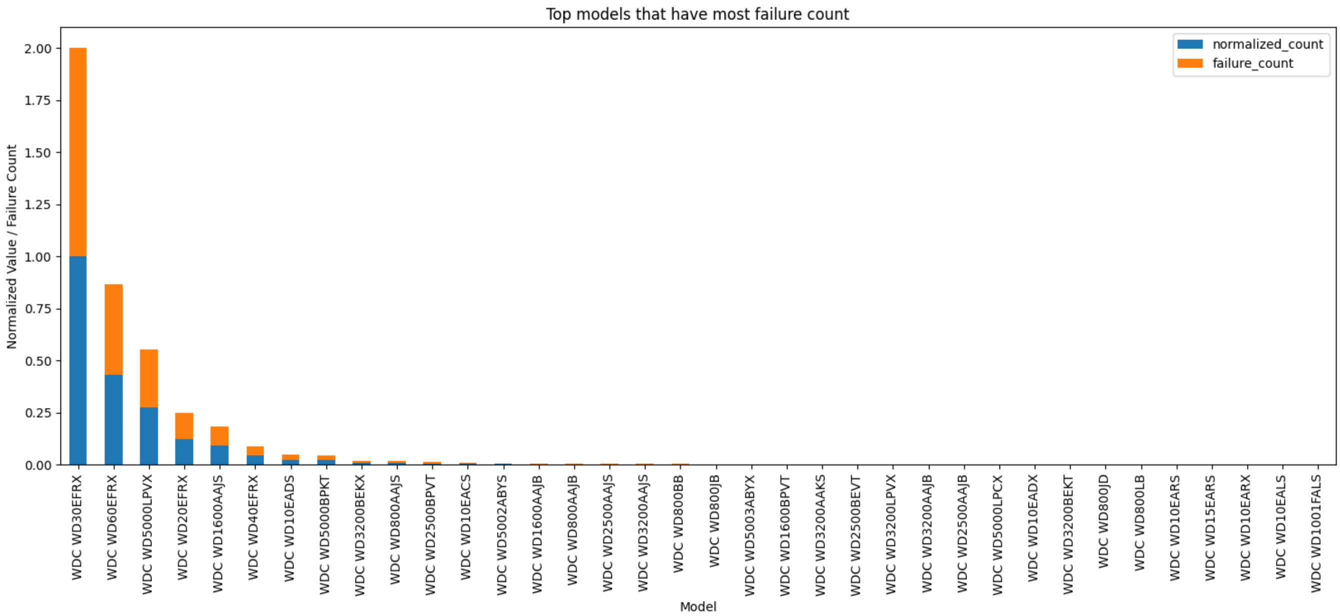
```
grouped_by_model = df_charts.groupby('model').agg(
    failure_count=('failure', 'count'),
    median_capacity=('capacity_bytes', 'median'),
    avg_capacity=('capacity_bytes', 'mean'),
    avg_temperature=('smart_temperature_celsius', 'mean'),
    avg_power_cycles=('smart_power_cycle_count', 'mean'),
    avg_reallocated_sectors=('smart_reallocated_sector_count', 'mean'),
    avg_pending_sectors=('smart_current_pending_sector_count', 'mean'),
    count=('serial_number', 'count') # Count occurrences for each model
)

# Sort by count to focus on models with more data
grouped_by_model = grouped_by_model.sort_values(by='failure_count', ascending=F

print("Top 5 Failure rate models (sorted by count):")
grouped_by_model.head(5)
```

```
Top 5 Failure rate models (sorted by count):
```

model	failure_count	median_capacity	avg_capacity	avg_temperature
WDC WD30EFRX	192107	3.000593e+12	3.000593e+12	124.739046
WDC WD60EFRX	83341	6.001175e+12	6.001175e+12	121.792143
WDC WD5000LPVX	53112	5.001079e+11	5.001079e+11	112.624605
WDC	[illegible]	[illegible]	[illegible]	[illegible]

```
from sklearn.preprocessing import MinMaxScaler

df_plot = grouped_by_model.copy()
scaler = MinMaxScaler()
df_plot['normalized_count'] = scaler.fit_transform(df_plot[['count']])
df_plot['failure_count'] = scaler.fit_transform(df_plot[['failure_count']])
df_plot = df_plot[['normalized_count', 'failure_count']]

df_plot.plot(kind='bar', stacked=True, figsize=(15, 7))

plt.title('Top models that have most failure count')
plt.xlabel('Model')
plt.ylabel('Normalized Value / Failure Count')
plt.xticks(rotation=90)
plt.tight_layout()
```

```
plt.show()
```

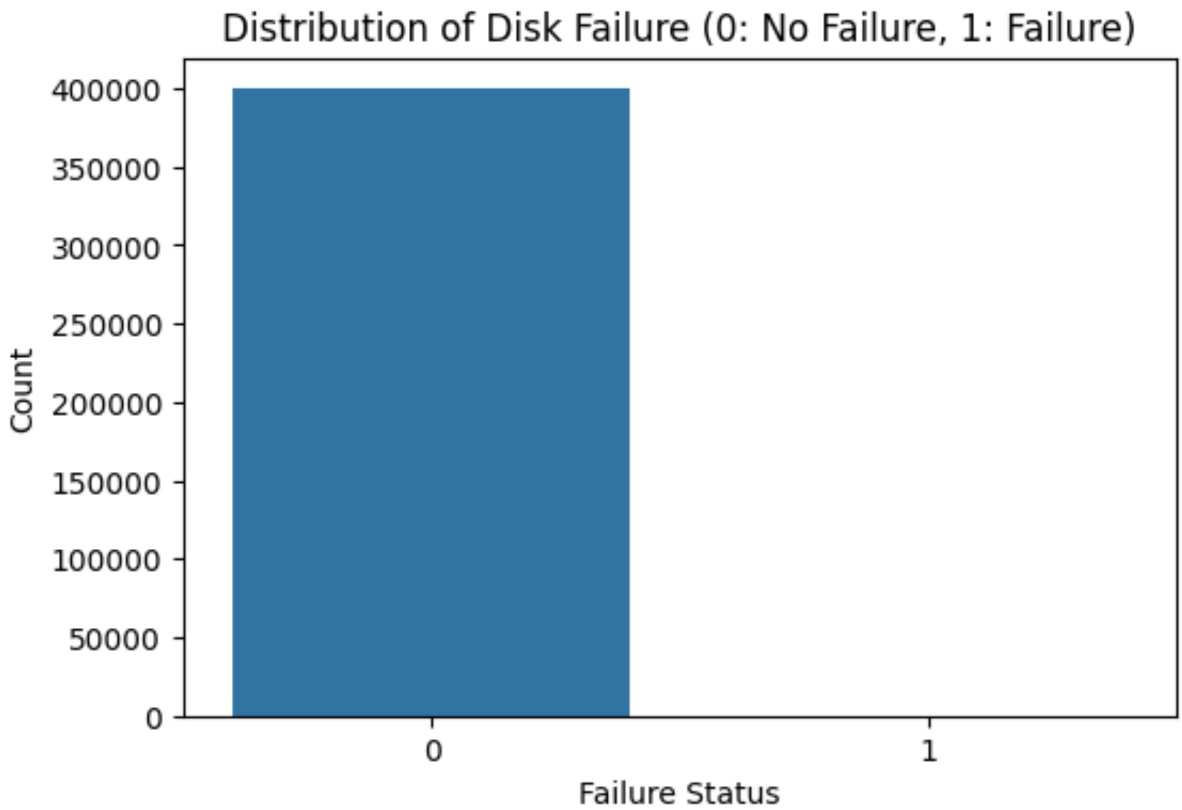


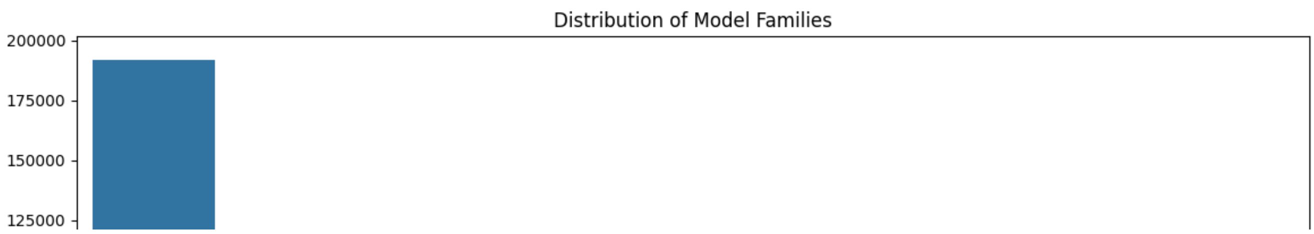

```
list(df_wdc.columns)
```

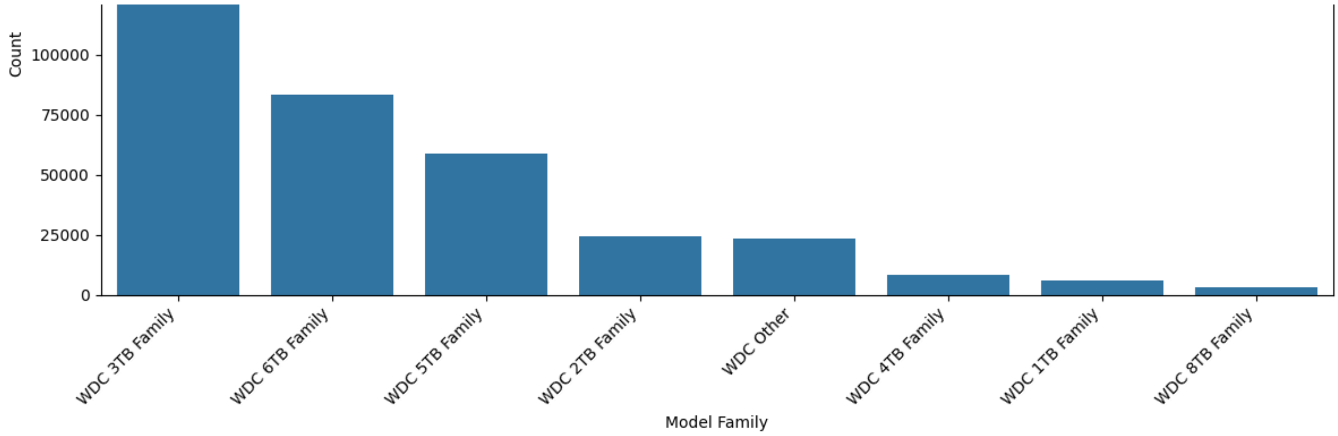
Show hidden output

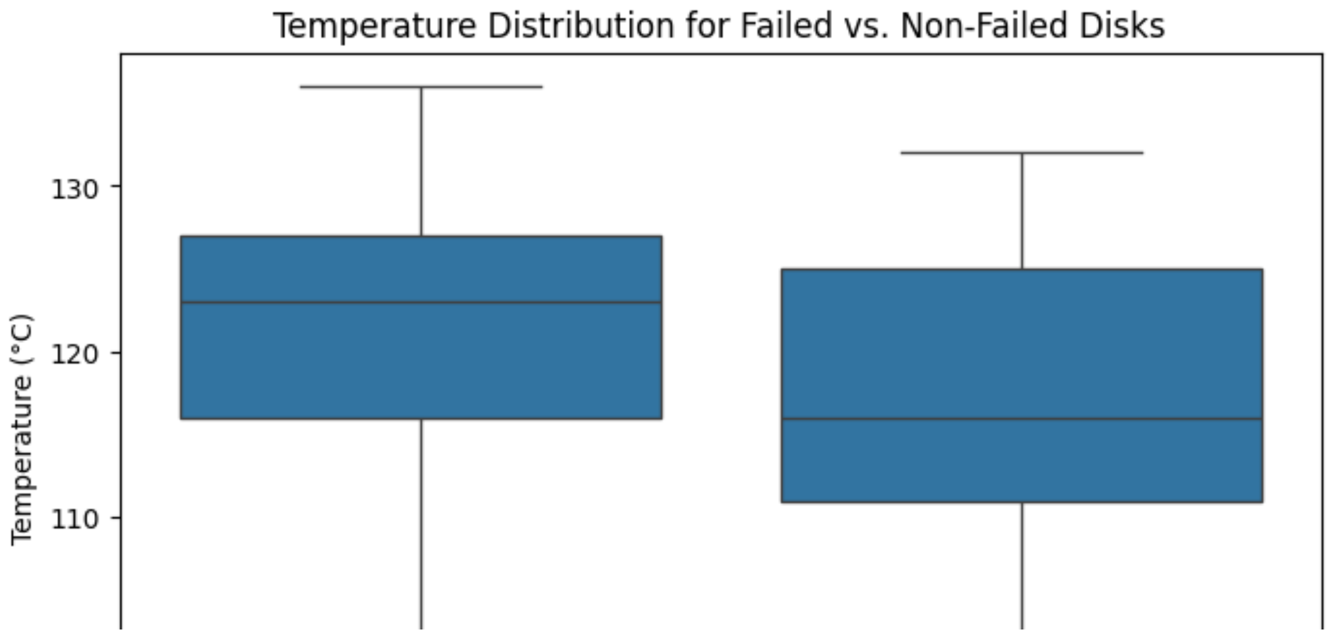
```
# 1. Distribution of 'failure'
plt.figure(figsize=(6, 4))
sns.countplot(x='failure', data=df_charts)
plt.title('Distribution of Disk Failure (0: No Failure, 1: Failure)')
```

```
plt.xlabel('Failure Status')
plt.ylabel('Count')
plt.show()

# 2. Distribution of 'model_family'
plt.figure(figsize=(12, 6))
sns.countplot(x='model_family', data=df_charts, order=df_charts['model_family']
plt.title('Distribution of Model Families')
plt.xlabel('Model Family')
plt.ylabel('Count')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

# 3. Relationship between 'smart_temperature_celsius' and 'failure' (using box
plt.figure(figsize=(8, 5))
sns.boxplot(x='failure', y='smart_temperature_celsius', data=df_charts)
plt.title('Temperature Distribution for Failed vs. Non-Failed Disks')
plt.xlabel('Failure Status (0: No Failure, 1: Failure)')
plt.ylabel('Temperature (°C)')
plt.show()
```



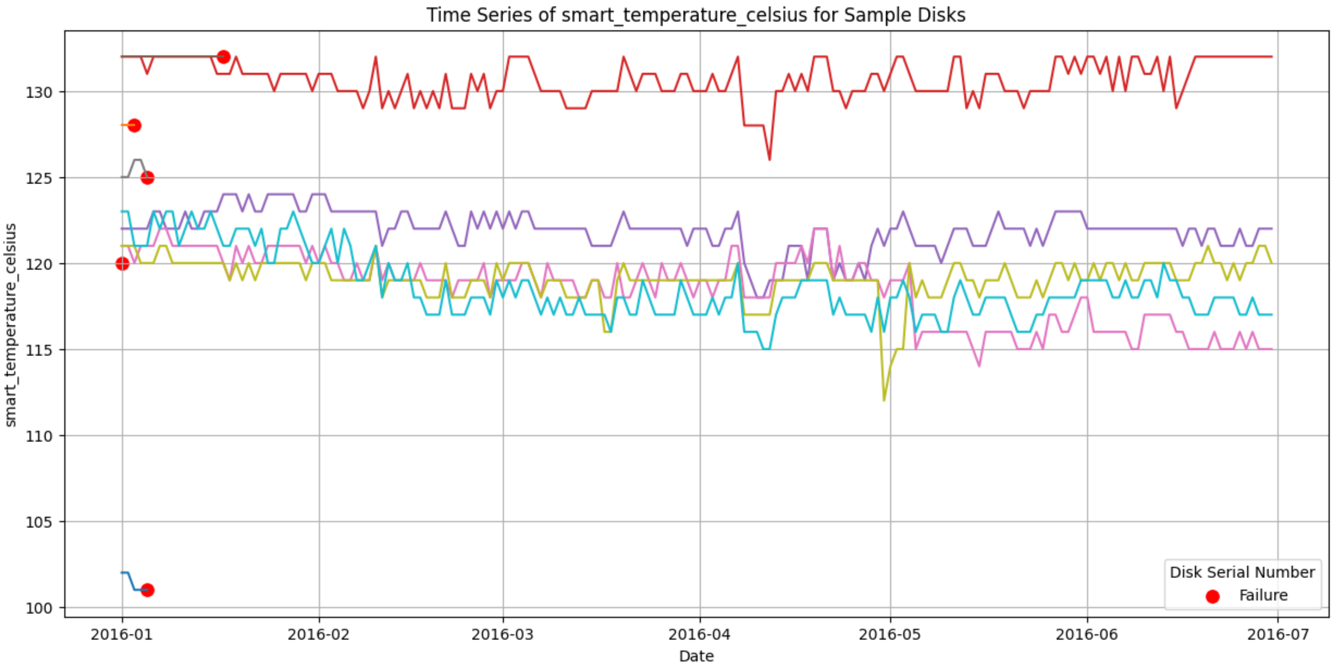


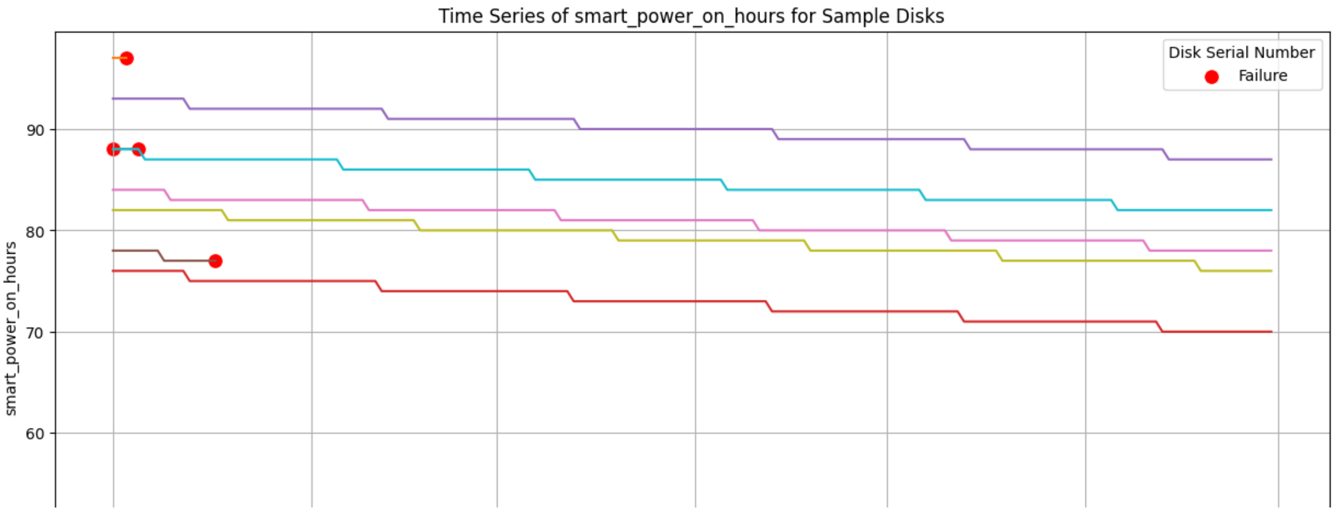


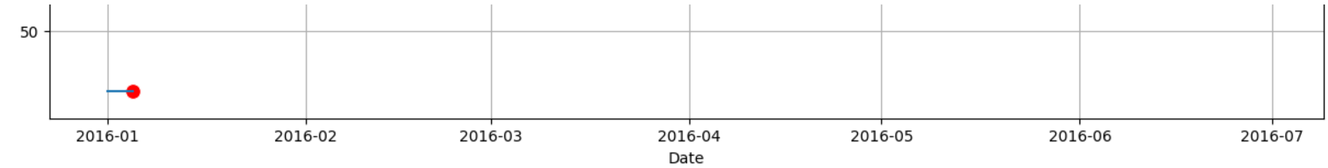


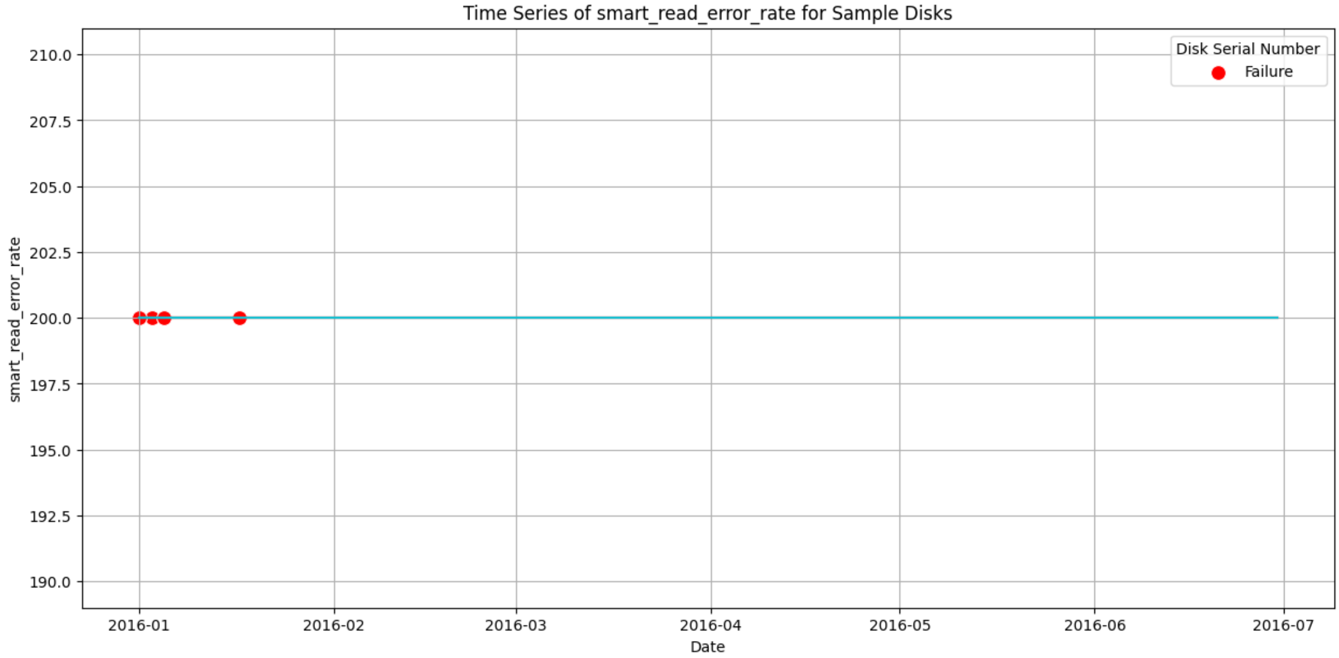

```
# 5 SMART attributes that are relevant for time series analysis of disk failure
smart_time_series_cols = [
    'smart_temperature_celsius',
"smart_power_on_hours",
"smart_read_error_rate" ,
"smart_uncorrectable_sector_count"
]

# Filter data to include only a few sample serial numbers for clearer time seri
# Select 5 serial numbers with failures and 5 without failures (if available)
failed_serials = df_charts[df_charts['failure'] == 1]['serial_number'].unique()
non_failed_serials = df_charts[df_charts['failure'] == 0]['serial_number'].unic

# Take a sample of serial numbers for plotting
num_samples = 5
sample_failed_serials = failed_serials[:num_samples]
sample_non_failed_serials = non_failed_serials[:num_samples]

sample_serials = list(sample_failed_serials) + list(sample_non_failed_serials)
df_plot_ts = df_charts[df_charts['serial_number'].isin(sample_serials)].copy()
```

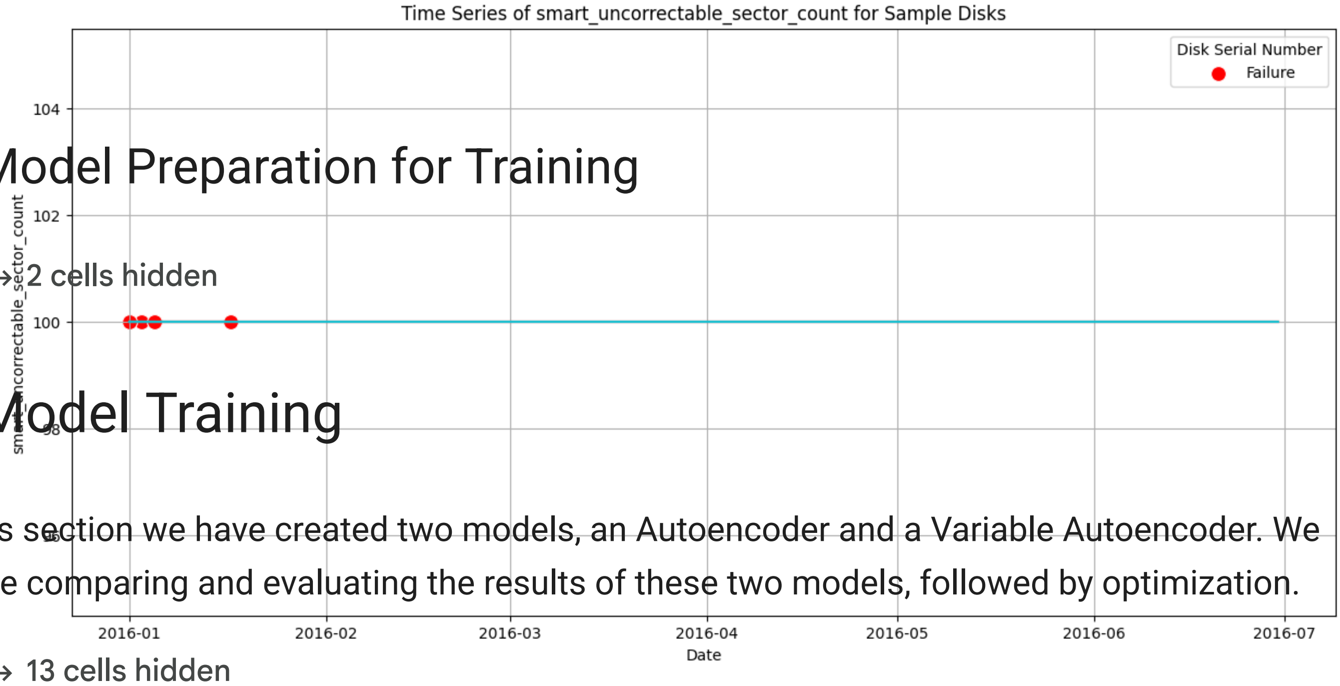
```
# Ensure date is datetime and sort for time series plotting
df_plot_ts['date_datetime'] = pd.to_datetime(df_plot_ts['date'])
df_plot_ts = df_plot_ts.sort_values(by=['serial_number', 'date_datetime'])

# Plot time series for the selected SMART attributes for each sample serial num
for col in smart_time_series_cols:
    plt.figure(figsize=(15, 7))
    sns.lineplot(data=df_plot_ts, x='date_datetime', y=col, hue='serial_number'
    # Highlight failure points
    failure_points = df_plot_ts[df_plot_ts['failure'] == 1]
    sns.scatterplot(data=failure_points, x='date_datetime', y=col, color='red',
    plt.title(f'Time Series of {col} for Sample Disks')
    plt.xlabel('Date')
    plt.ylabel(col)
    plt.legend(title='Disk Serial Number')
    plt.grid(True)
    plt.show()
```











Model Preparation for Training

[ ] ↳ 2 cells hidden

Model Training

In this section we have created two models, an Autoencoder and a Variable Autoencoder. We will be comparing and evaluating the results of these two models, followed by optimization.

[ ] ↳ 13 cells hidden

Model Training Conclusion

Important Metric will be Recall for Anomaly Detection:

In anomaly detection, especially for rare events like hard drive failures, recall for the anomaly class is often a crucial metric. High recall means the model is good at identifying most of the actual anomalies, even if it also flags some normal instances as anomalies (lower precision). Missing a failure (False Negative) is often more costly than a false alarm (False Positive).

Looking at the recall for the 'Anomaly' class:

- VAE: Recall = 0.17 (Only 17% of actual anomalies were detected)
- Vanilla AE: Recall = 0.50 (50% of actual anomalies were detected)

The vanilla AE has significantly higher recall for detecting anomalies compared to the VAE.

The VAE's lower AUC and significantly lower recall for the anomaly class indicate that it is less effective at distinguishing anomalies from normal data in this specific scenario compared to the vanilla AE. This is likely due to the VAE's inherent design goal of learning a structured latent space and distribution, which can sometimes lead to less pronounced reconstruction errors for anomalies compared to a vanilla AE that solely focuses on minimizing reconstruction error for normal data. For this specific anomaly detection problem, the vanilla AE appears to be better at highlighting the difference between normal and anomalous data through reconstruction error.

```
Start coding or generate with AI.
```

Final Custom Model

Known Optimization

1. Leaky ReLU - some negative slope for better activation
2. Batch Norm
3. Dropout rate - avoid over fitting
4. Scheduler - CosineAnnealingLR ("Cosine shaped" scheduling for better training stability and adaptation)
5. Adaptive Learning