Sensor Component Failure Prediction

YashwanthGoduguchintha

(15thApril2023)

Abstract

The proposed project aims to develop a personalized travel planner that generates customized travel itineraries for users based on their interests and the duration of their trip. The platform will utilize data on various destinations, attractions to create a comprehensive travel plan for the user. Users will input their travel preferences, including their travel dates, and the algorithm will generate a travel itinerary that optimizes their travel experience. The platform will provide users with a user-friendly interface, making the process of travel planning more efficient and enjoyable. The proposed travel planner has the potential to meet the needs of travelers looking for a more efficient approach to travel planning and could benefit the travel industry stakeholders by generating more revenue for their businesses.

Problem Statement

The Air Pressure System (APS) is a critical component of a heavy-duty vehicle that uses compressed air to force a piston to provide pressure to the brake pads, slowing the vehicle down. The benefits of using an APS instead of a hydraulic system are the easy availability and long-term sustainability of natural air.

This is a Binary Classification problem, in which the affirmative class indicates that the failure was caused by a certain component of the APS, while the negative class indicates that the failure was caused by something else.

Solution Proposed

In this project, the system in focus is the Air Pressure system (APS) which generates pressurized air that are utilized in various functions in a truck, such as braking and gear changes. The datasets positive class corresponds to component failures for a specific component of the APS system. The negative class corresponds to trucks with failures for components not related to the APS system.

The problem is to reduce the cost due to unnecessary repairs. So it is required to minimize the false predictions.

Tech Stack Used

1. **Python**
2. **FastAPI**
3. **Machine learning algorithms**
4. **Docker**
5. **MongoDB**

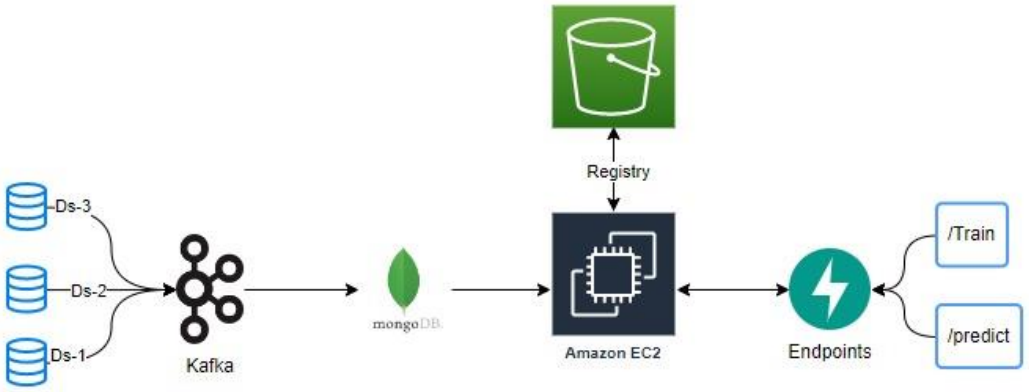
Infrastructure Required.

1. **AWS S3**
2. **AWS EC2**
3. **AWS ECR**
4. **Git Actions**
5. **Terraform**

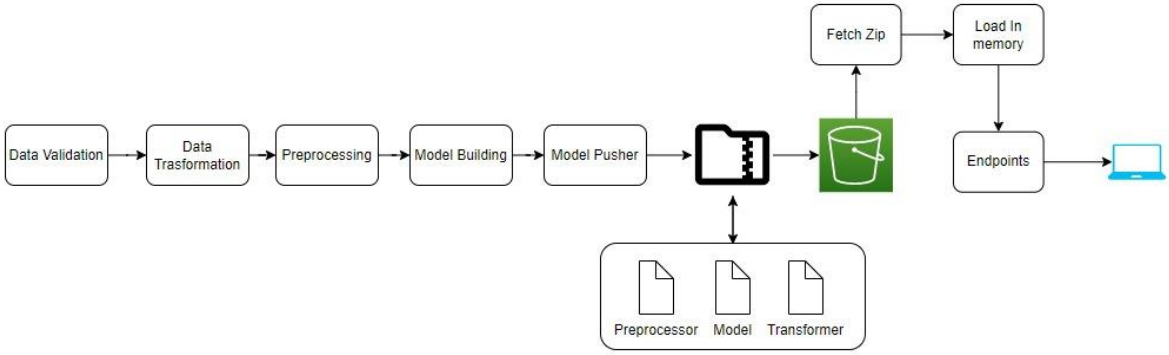
How to run?

Before we run the project, make sure that you are having MongoDB in your local system, with Compass since we are using MongoDB for data storage. You also need AWS account to access the service like S3, ECR and EC2 instances.

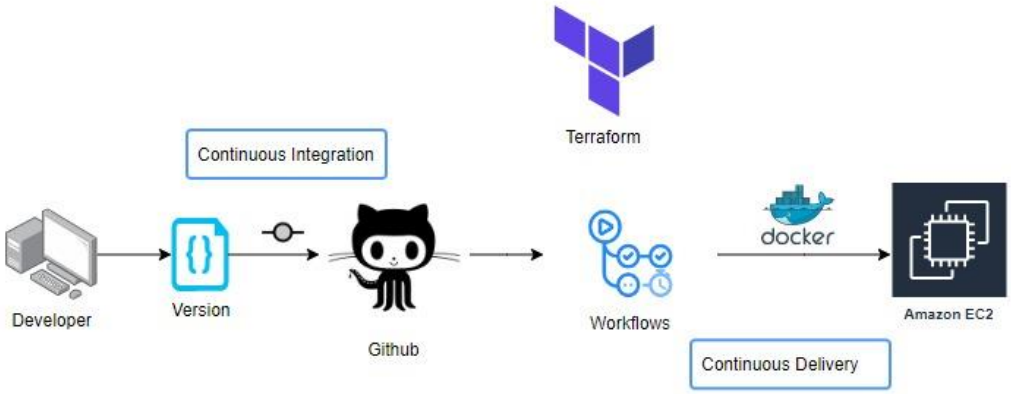
Data Collections



Project Archietecture



Deployment Archietecture



How to Get Market/Customer/Business Need Assessment

Sensor-fault-detection using machine learning can be a valuable solution for businesses that rely on sensors to monitor and control critical processes, such as manufacturing, energy production, transportation, and more. Here is a market/customer/business need assessment for sensor-fault-detection ml:

Market Assessment:

The global market for machine learning in manufacturing is projected to reach USD 4.1 billion by 2026, with a CAGR of 43.2% from 2021 to 2026.

Other industries that can benefit from sensor-fault-detection using machine learning include automotive, aerospace, oil and gas, and healthcare.

There is an increasing demand for efficient and accurate fault detection systems that can minimize downtime, reduce maintenance costs, and improve productivity.

Customer Assessment:

Customers who rely on sensors for critical processes are likely to value a solution that can detect and diagnose sensor faults before they cause significant disruptions or damage.

Small and medium-sized businesses with limited resources may not have the expertise or budget to develop their own sensor-fault-detection system, making them potential customers for a pre-built solution.

Customers are likely to prefer a solution that can integrate with their existing systems and provide real-time alerts and insights to facilitate quick decision-making.

Business Need Assessment:

Businesses that rely on sensors for critical processes can face significant risks and costs associated with sensor failures, including downtime, maintenance, repair, and replacement.

By implementing a sensor-fault-detection system using machine learning, businesses can reduce the risk of sensor failures, minimize downtime, improve productivity, and reduce maintenance costs.

A reliable sensor-fault-detection system can also help businesses maintain compliance with industry regulations and standards.

In summary, there is a growing market and customer need for sensor-fault-detection using machine learning, particularly in industries that rely on sensors for critical processes. By providing an efficient and accurate solution, businesses can reduce downtime, improve productivity, and minimize maintenance costs, ultimately leading to increased profitability and customer satisfaction.

Business Model:-

There are several monetization ideas that could be used for a sensor-fault-detection system using machine learning:

1)Subscription-based model: The company can offer a subscription-based model to customers, where they pay a monthly or annual fee for access to the sensor-fault-detection system. The subscription could be customized based on the number of sensors being monitored and the frequency of sensor data updates.

2)Pay-per-use model: The company can charge customers for each use of the sensor-fault-detection system, based on the number of sensors being monitored and the frequency of sensor data updates. This model would be particularly suitable for customers who only require occasional use of the system.

3)Commission-based model: The company could offer the sensor-fault-detection system as a value-added service to customers who purchase or lease sensors from the company. The company could charge a commission on each sale or lease of sensors that includes the sensor-fault-detection system.

4)Consulting model: The company can offer consulting services to customers who require assistance in developing customized sensor-fault-detection solutions. The consulting fees can be charged on an hourly basis or as a fixed fee based on the complexity of the project.

5)Data analytics model: The company can use the sensor data collected from customers to develop insights and predictive analytics models. These models could be sold to customers as a separate service, generating additional revenue for the company.

In summary, the monetization model for a sensor-fault-detection system using machine learning can be based on subscription, pay-per-use, commission, consulting, or data analytics models. The choice of monetization model will depend on the target customers, the competitive landscape, and the value proposition offered by the company.

Target Specifications:

When designing a sensor-fault-detection system using machine learning, the following target specifications should be considered:

1)Data Sources: The system should be able to integrate with various data sources, such as sensors, programmable logic controllers (PLCs), and human-machine interfaces (HMIs), to collect real-time sensor data and system logs.

2)Data Preprocessing: The system should have a pre-processing pipeline that includes data cleaning, feature selection, feature extraction, and normalization to prepare the data for analysis.

3)Machine Learning Algorithms: The system should incorporate machine learning algorithms that are capable of detecting sensor faults in real-time, such as decision trees, random forests, support vector machines, neural networks, or deep learning algorithms.

4)Model Training and Validation: The system should have a training and validation pipeline that includes data splitting, cross-validation, hyperparameter tuning, and model evaluation to ensure accurate and reliable predictions.

5)Real-time Alerting: The system should provide real-time alerting mechanisms, such as emails, text messages, or visual displays, to notify operators and maintenance personnel of potential sensor faults.

6)Dashboard and Reporting: The system should provide a dashboard and reporting interface that displays real-time and historical sensor data, fault detection rates, and maintenance trends, to facilitate decision-making and performance monitoring.

7)Scalability and Flexibility: The system should be scalable and flexible, allowing for easy integration with existing systems, customization of alerting mechanisms and reporting formats, and expansion to accommodate additional sensors or data sources.

8)In summary, the target specifications for a sensor-fault-detection system using machine learning should include data sources, preprocessing, machine learning algorithms, model training and validation, real-time alerting, dashboard and reporting, and scalability and flexibility.

External Search:

"Sensor Fault Detection in Industrial Processes Using Machine Learning Techniques: A Review" by Muhammad Jawad Khan and Imran Hayee, published in IEEE Access. This paper provides an overview of machine learning techniques for sensor fault detection in industrial processes and their applications.

"Machine Learning-based Sensor Fault Detection and Diagnosis for Heating, Ventilating, and Air Conditioning Systems" by Zhe Zhang et al., published in Energy and Buildings. This paper presents a machine learning-based sensor fault detection and diagnosis method for heating, ventilating, and air conditioning systems.

"Sensor Fault Detection and Isolation Using Machine Learning for a Three-Phase Inverter" by Ibrahim M. Elamin and Murtaza Farsadi, published in IEEE Transactions on Industrial Electronics. This paper proposes a machine learning-based sensor fault detection and isolation method for a three-phase inverter.

"A Machine Learning Approach for Sensor Fault Detection and Diagnosis in Autonomous Vehicles" by Yiren Yang et al., published in IEEE Transactions on Intelligent Transportation Systems. This paper presents a machine learning-based sensor fault detection and diagnosis method for autonomous vehicles.

"Anomaly Detection in Sensor Data Using Machine Learning for Industrial IoT Applications" by Niklas Kühl and Lasse Klingbeil, published in IEEE Transactions on Industrial Informatics. This paper proposes a machine learning-based anomaly detection method for sensor data in industrial IoT applications.

These sources provide valuable insights into the use of machine learning for sensor fault detection in various applications, including industrial processes, HVAC systems, three-phase inverters, autonomous vehicles, and industrial IoT.

Bench marking alternate products:

When benchmarking alternate products for sensor-fault-detection using machine learning, the following factors should be considered:

1)Accuracy: The accuracy of the product in detecting sensor faults is an important factor to consider. The product should be able to accurately detect sensor faults in real-time and provide accurate alerting mechanisms.

2)Speed: The speed of the product in processing sensor data and detecting faults is another important factor. The product should be able to process data in real-time and provide alerts within a few seconds of detecting a fault.

3)Integration: The ease of integration with existing systems and data sources is also an important consideration. The product should be able to integrate with various data sources, such as sensors, PLCs, and HMIs, and provide a seamless user experience.

4)Scalability: The scalability of the product in handling large volumes of sensor data and expanding to accommodate additional sensors or data sources is also important.

5)Cost: The cost of the product in terms of upfront costs, maintenance costs, and total cost of ownership should also be considered.

Some of the alternate products that can be benchmarked against a sensor-fault-detection using machine learning system include:

1)Traditional rule-based systems for sensor fault detection.

2)Statistical process control (SPC) methods.

3)Other machine learning-based sensor fault detection systems.

4)Customized solutions developed in-house.

5)Off-the-shelf software solutions that provide sensor fault detection features.

By benchmarking these alternate products against a sensor-fault-detection using machine learning system, it is possible to identify the strengths and weaknesses of each product and determine the best fit for a particular use case.

Applicable Regulations

When developing a sensor-fault-detection system using machine learning, there are several applicable regulations and standards that should be considered, depending on the industry and location. Here are some examples:

1. **General Data Protection Regulation (GDPR): GDPR is a regulation in the European Union that governs the processing and storage of personal data. If the sensor-fault-detection system processes personal data, GDPR compliance is necessary.**

2. **International Electrotechnical Commission (IEC) 61508: IEC 61508 is a standard that provides guidelines for the functional safety of electrical and electronic systems, including machine learning-based systems.**

3. **International Organization for Standardization (ISO) 26262: ISO 26262 is a standard that provides guidelines for the functional safety of automotive systems, including machine learning-based systems.**

4. **Food and Drug Administration (FDA) Guidance for Industry and FDA Staff: Software as a Medical Device (SaMD): This guidance provides recommendations for the development and validation of machine learning-based software as a medical device.**

5. **Health Insurance Portability and Accountability Act (HIPAA): HIPAA is a regulation in the United States that governs the privacy and security of personal health information. If the sensor-fault-detection system processes personal health information, HIPAA compliance is necessary.**

6. **Occupational Safety and Health Administration (OSHA): OSHA is a regulatory agency in the United States that sets standards for workplace safety. If the sensor-fault-detection system is used in an industrial or manufacturing setting, OSHA compliance is necessary.**

In addition to these regulations and standards, there may be other industry-specific regulations and standards that should be considered when developing a sensor-fault-detection system using machine learning. It is important to consult with legal and regulatory experts to ensure compliance with all applicable regulations and standards.

Applicable Constraints

When developing a sensor-fault-detection system using machine learning, there are several constraints that should be considered to ensure the system's effectiveness and feasibility. Here are some examples:

1. **Data Availability: The sensor-fault-detection system requires a large amount of data to train the machine learning model effectively. In some cases, data may be scarce, leading to challenges in developing a reliable model.**

2. **Data Quality: The quality of the data used to train the machine learning model is crucial. Poor-quality data can lead to inaccurate or unreliable results. Therefore, it is essential to ensure that the data is accurate, consistent, and reliable.**

3. **Computational Resources: Machine learning models require significant computational resources to train and operate effectively. Therefore, the hardware and software infrastructure should be capable of handling the computational demands of the system.**

4. **Cost: The cost of developing and deploying the sensor-fault-detection system can be significant. The cost includes hardware, software, data acquisition, and ongoing maintenance and support costs.**

5. **Regulatory Compliance: The sensor-fault-detection system may need to comply with various regulations and standards, as discussed in the previous response. Compliance with these regulations may require additional resources and expertise.**

6. **Operational Constraints: The sensor-fault-detection system should be designed to operate in various operational environments and handle different types of sensor data. The system must be scalable and flexible enough to adapt to changing operational needs.**

7. **User Experience: The sensor-fault-detection system should be easy to use and understand for operators and maintenance personnel. The system should provide clear and actionable alerts when faults are detected and provide the necessary information to address the issue.**

By considering these constraints, it is possible to design a sensor-fault-detection system using machine learning that is effective, feasible, and meets the needs of the intended users.

Concept Generation:

Here are some concepts for a sensor-fault-detection system using machine learning:

1. **Anomaly Detection: This concept involves using machine learning algorithms to identify abnormal sensor readings and alert operators or maintenance personnel when a fault is detected. This approach is effective in detecting previously unknown faults and can be used to detect a wide range of faults.**

2. **Pattern Recognition: This concept involves using machine learning algorithms to recognize patterns in sensor data and identify trends that indicate a fault. This approach is effective in identifying recurring faults and can be used to create predictive models that can anticipate future faults.**

3. **Fault Signature Analysis: This concept involves using machine learning algorithms to analyze the unique signature of a fault and detect it in real-time. This approach is effective in detecting known faults and can be used to create a library of fault signatures that can be used to identify faults quickly.**

4. **Multi-Sensor Fusion: This concept involves using machine learning algorithms to fuse data from multiple sensors and detect faults that cannot be detected by a single sensor. This approach is effective in identifying complex faults and can be used to create a more comprehensive fault detection system.**

5. **Predictive Maintenance: This concept involves using machine learning algorithms to predict when a fault is likely to occur based on sensor data and maintenance history. This approach is effective in preventing failures before they occur and can be used to create a proactive maintenance program.**

These are just a few concepts for a sensor-fault-detection system using machine learning. The choice of concept depends on the specific application, the available data, and the operational constraints.

Concept Development

Here is a step-by-step process for developing a concept for a sensor-fault-detection system using machine learning:

1. **Define the Problem: Start by defining the problem that the sensor-fault-detection system is intended to solve. Identify the sensors that will be used, the types of faults that need to be detected, and the consequences of failure.**

2. **Identify the Data Sources: Identify the data sources that will be used to train the machine learning model. This may include historical sensor data, fault data, maintenance records, and other relevant data.**

3. **Select the Machine Learning Algorithm: Select the machine learning algorithm that is most appropriate for the problem. Consider factors such as the complexity of the problem, the size of the data set, and the available computational resources.**

4. **Train the Machine Learning Model: Train the machine learning model using the selected algorithm and the available data. This may involve feature engineering, data cleaning, and hyperparameter tuning to optimize the model's performance.**

5. **Validate the Model: Validate the machine learning model to ensure that it is accurate and reliable. This may involve testing the model on a separate data set or using cross-validation techniques.**

6. **Develop the Fault Detection System: Develop the fault detection system using the machine learning model. This may involve integrating the model into a larger software system, developing user interfaces for operators, and defining the alerting mechanism.**

7. **Test and Deploy the System: Test the fault detection system to ensure that it is effective and reliable. Deploy the system in the operational environment and monitor its performance over time.**

8. **Maintain and Update the System: Maintain and update the fault detection system over time to ensure that it continues to perform effectively. This may involve retraining the machine learning model, updating the software, and incorporating new data sources.**

By following this process, it is possible to develop a concept for a sensor-fault-detection system using machine learning that is effective, reliable, and meets the needs of the intended users.

Code Implementation:

Sensor Component Failure Prediction

In [ ]:

1) Problem statement.

Data: Sensor Data

Problem statement :

- The system in focus is the Air Pressure system (APS) which generates pressurized air that are utilized in various functions in a truck, such as braking and gear changes. The datasets positive class corresponds to component failures for a specific component of the APS system. The negative class corresponds to trucks with failures for components not related to the APS system.

- The problem is to reduce the cost due to unnecessary repairs. So it is required to minimize the false predictions.

Cost 1 = 10 and Cost 2 = 500

- The total cost of a prediction model the sum of Cost_1 multiplied by the number of Instances with type 1 failure and Cost_2 with the number of instances with type 2 failure, resulting in a Total_cost. In this case Cost_1 refers to the cost that an unnessecary check needs to be done by an mechanic at an workshop, while Cost_2 refer to the cost of missing a faulty truck, which may cause a breakdown.

- Total_cost = Cost_1 * No_Instances + Cost_2 * No_Instances.

- From the above problem statement we could observe that, we have to reduce false positives and false negatives. More importantly we have to **reduce false negatives, since cost incurred due to false negative is 50 times higher than the false positives.**

True class	Positive	Negative
Predicted class		
Positive	-	cost_1
Negative	cost_2	

-

Challenges and other objectives

- Need to Handle many Null values in almost all columns
- No low-latency requirement.
- Interpretability is not important.
- misclassification leads the unecessary repair costs.

2) Import required libraries

In [21]:

```
import pandas as pd
import seaborn as sns
import numpy as np
from statistics import mean
import matplotlib.pyplot as plt
import warnings
from sklearn.preprocessing import PowerTransformer
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.pipeline import Pipeline
from sklearn.utils import resample

from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier,
GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score,
classification_report,ConfusionMatrixDisplay, \
                            precision_score, recall_score, f1_score,
roc_auc_score,roc_curve,confusion_matrix


from sklearn import metrics
from sklearn.model_selection import  train_test_split,
RepeatedStratifiedKFold, cross_val_score
from sklearn.preprocessing import OneHotEncoder, MinMaxScaler
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer, KNNImputer
from xgboost import XGBClassifier
from sklearn.preprocessing import StandardScaler, MinMaxScaler,RobustScaler
from sklearn.compose import ColumnTransformer
from catboost import CatBoostClassifier

warnings.filterwarnings("ignore")
%matplotlib inline
```

Read Data

In [2]:

```
# Load csv file
df = pd.read_csv('aps_failure_training_reduced.csv', na_values="na")
```

In [3]:

```
# check rows and columns of the dataset
df.shape
```

Out[3]:

```
(36188, 171)
```

In [4]:

```
# Check unique values of target varaible
df['class'].value_counts()
```

Out[4]:

```
neg    35188
pos     1000
Name: class, dtype: int64
```

In [5]:

```
# define numerical & categorical columns
numeric_features = [feature for feature in df.columns if df[feature].dtype !=
'O']
categorical_features = [feature for feature in df.columns if
df[feature].dtype == 'O']

# print columns
print('We have {} numerical features : {}'.format(len(numeric_features),
numeric_features))
print('\nWe have {} categorical features :
{}'.format(len(categorical_features), categorical_features))
We have 170 numerical features : ['aa_000', 'ab_000', 'ac_000', 'ad_000', 'ae
_000', 'af_000', 'ag_000', 'ag_001', 'ag_002', 'ag_003', 'ag_004', 'ag_005',
'ag_006', 'ag_007', 'ag_008', 'ag_009', 'ah_000', 'ai_000', 'aj_000', 'ak_000
', 'al_000', 'am_0', 'an_000', 'ao_000', 'ap_000', 'aq_000', 'ar_000', 'as_00
0', 'at_000', 'au_000', 'av_000', 'ax_000', 'ay_000', 'ay_001', 'ay_002', 'ay
_003', 'ay_004', 'ay_005', 'ay_006', 'ay_007', 'ay_008', 'ay_009', 'az_000',
'az_001', 'az_002', 'az_003', 'az_004', 'az_005', 'az_006', 'az_007', 'az_008
', 'az_009', 'ba_000', 'ba_001', 'ba_002', 'ba_003', 'ba_004', 'ba_005', 'ba_
006', 'ba_007', 'ba_008', 'ba_009', 'bb_000', 'bc_000', 'bd_000', 'be_000', '
bf_000', 'bg_000', 'bh_000', 'bi_000', 'bj_000', 'bk_000', 'bl_000', 'bm_000'
, 'bn_000', 'bo_000', 'bp_000', 'bq_000', 'br_000', 'bs_000', 'bt_000', 'bu_0
00', 'bv_000', 'bx_000', 'by_000', 'bz_000', 'ca_000', 'cb_000', 'cc_000', 'c
d_000', 'ce_000', 'cf_000', 'cg_000', 'ch_000', 'ci_000', 'cj_000', 'ck_000',
'cl_000', 'cm_000', 'cn_000', 'cn_001', 'cn_002', 'cn_003', 'cn_004', 'cn_005
', 'cn_006', 'cn_007', 'cn_008', 'cn_009', 'co_000', 'cp_000', 'cq_000', 'cr_
000', 'cs_000', 'cs_001', 'cs_002', 'cs_003', 'cs_004', 'cs_005', 'cs_006', '
cs_007', 'cs_008', 'cs_009', 'ct_000', 'cu_000', 'cv_000', 'cx_000', 'cy_000'
, 'cz_000', 'da_000', 'db_000', 'dc_000', 'dd_000', 'de_000', 'df_000', 'dg_0
00', 'dh_000', 'di_000', 'dj_000', 'dk_000', 'dl_000', 'dm_000', 'dn_000', 'd
o_000', 'dp_000', 'dq_000', 'dr_000', 'ds_000', 'dt_000', 'du_000', 'dv_000',
```

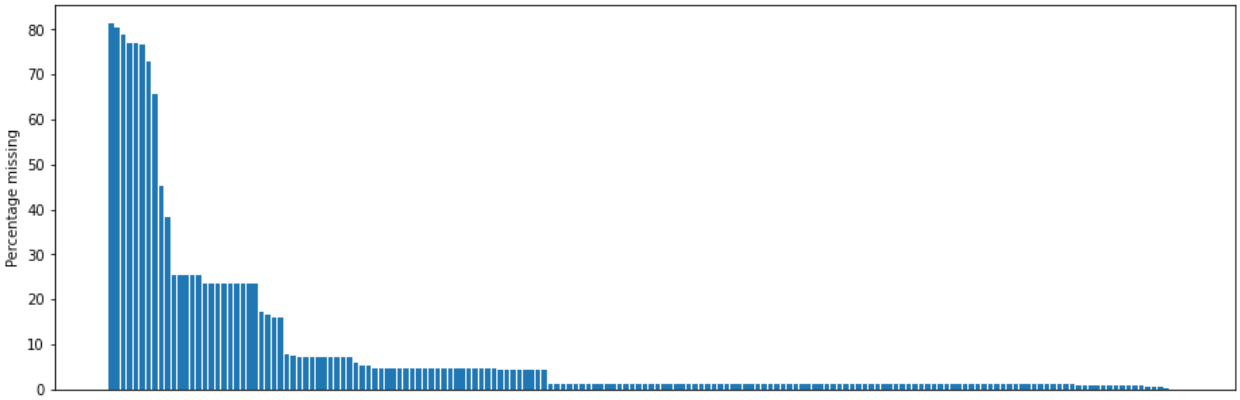
```
'dx_000', 'dy_000', 'dz_000', 'ea_000', 'eb_000', 'ec_00', 'ed_000', 'ee_000'
, 'ee_001', 'ee_002', 'ee_003', 'ee_004', 'ee_005', 'ee_006', 'ee_007', 'ee_0
08', 'ee_009', 'ef_000', 'eg_000']

We have 1 categorical features : ['class']
```

As this is a Sensor data. Interpretation of the data is not required

Checking missing values

In [6]:

```
# Plotting Missing values count for each column
fig, ax = plt.subplots(figsize=(15,5))

missing =
df.isna().sum().div(df.shape[0]).mul(100).to_frame().sort_values(by=0,
ascending = False)

ax.bar(missing.index, missing.values.T[0])
plt.xticks([])
plt.ylabel("Percentage missing")
plt.show()
```



Dropping Columns which has more than 70% of missing values.

In [7]:

```
## Dropping columns which has more than 70% of missing values
dropcols = missing[missing[0]>70]
dropcols
```

Out[7]:

	0
br_000	81.410965

	0
bq_000	80.501824
bp_000	78.794075
ab_000	77.086327
cr_000	77.086327
bo_000	76.533658
bn_000	72.761689

In [8]:

```
df.drop(list(dropcols.index), axis=1, inplace=True)
```

In [9]:

```
# Check shape of the dataset after dropping columns
df.shape
```

Out[9]:

```
(36188, 164)
```

Check the total percentage of missing values of full dataset after dropping columns with more than 70% of missing values

In [10]:

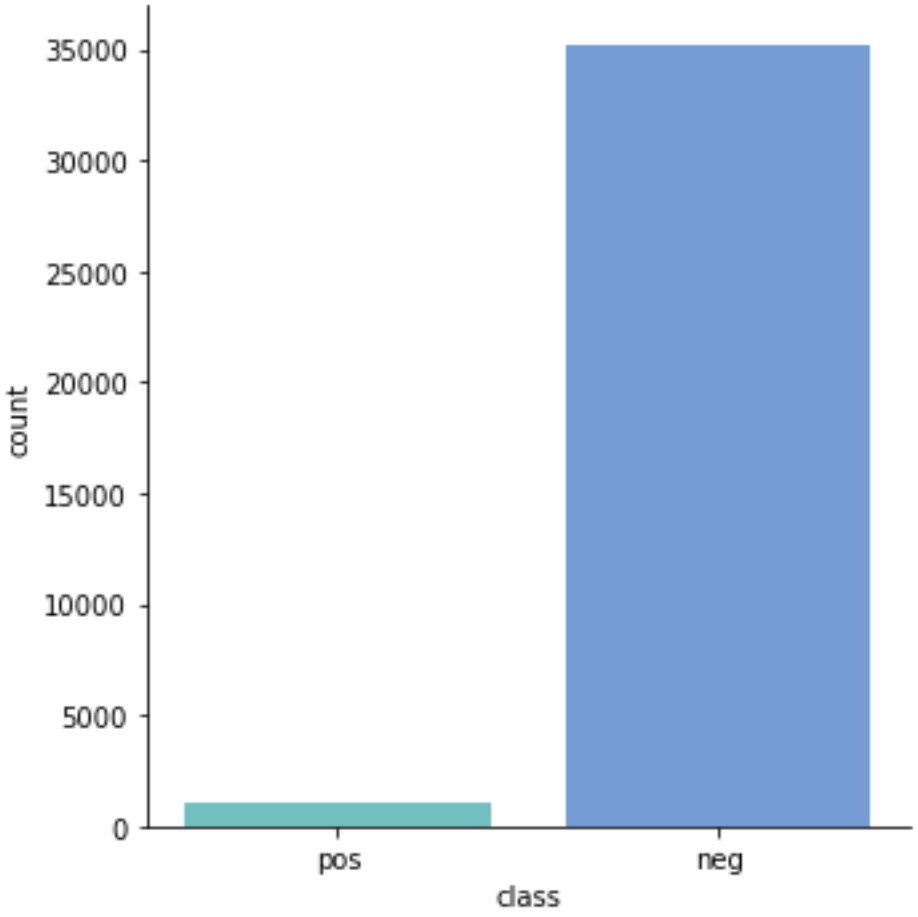
```
missing_values_count= df.isnull().sum()
total_cells = np.product(df.shape)
total_missing = missing_values_count.sum()

# percent of data that is missing
print(f"Percentage of total missing cells in the data
{(total_missing/total_cells) * 100}%")
Percentage of total missing cells in the data 5.37059852747306%
```

Visualization of unique values in Target variable

In [11]:

```
pos = df[df['class']=='pos'].shape[0]
neg = df[df['class']=='neg'].shape[0]
```

```
print("Positive: " + str(pos) + ", Negative: " + str(neg))
sns.catplot(data=df, x="class", kind="count", palette="winter_r", alpha=.6)
plt.show()
Positive: 1000, Negative: 35188
```



Report

- The target classes are highly imbalanced
- Class imbalance is a scenario that arises when we have unequal distribution of class in a dataset i.e. the no. of data points in the negative class (majority class) very large compared to that of the positive class (minority class)
- If the imbalanced data is not treated beforehand, then this will degrade the performance of the classifier model.
- Hence we should handle imbalanced data with certain methods.

How to handle Imbalance Data ?

- Resampling data is one of the most commonly preferred approaches to deal with an imbalanced dataset. There are broadly two types of methods for this i) Undersampling ii) Oversampling. In most cases, oversampling is preferred over undersampling techniques. The reason being, in undersampling we tend to remove instances from data that may be carrying some important information.
- **SMOTE:** Synthetic Minority Oversampling Technique

- SMOTE is an oversampling technique where the synthetic samples are generated for the minority class.
- Hybridization techniques involve combining both undersampling and oversampling techniques. This is done to optimize the performance of classifier models for the samples created as part of these techniques.
- It only duplicates the data and it won't add and new information. Hence we look at some different techniques.

Create Functions for model training and evaluation

In [1]:

```
def evaluate_clf(true, predicted):
    '''
    This function takes in true values and predicted values
    Returns: Accuracy, F1-Score, Precision, Recall, Roc-auc Score
    '''
    acc = accuracy_score(true, predicted) # Calculate Accuracy
    f1 = f1_score(true, predicted) # Calculate F1-score
    precision = precision_score(true, predicted) # Calculate Precision
    recall = recall_score(true, predicted)  # Calculate Recall
    roc_auc = roc_auc_score(true, predicted) #Calculate Roc
    return acc, f1 , precision, recall, roc_auc
```

In [13]:

```
# Create cost of the model as per data description
def total_cost(y_true, y_pred):
    '''
    This function takes y_ture, y_predicted, and prints Total cost due to
misclassification

    '''
    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
    cost = 10*fp + 500*fn
    return cost
```

In [14]:

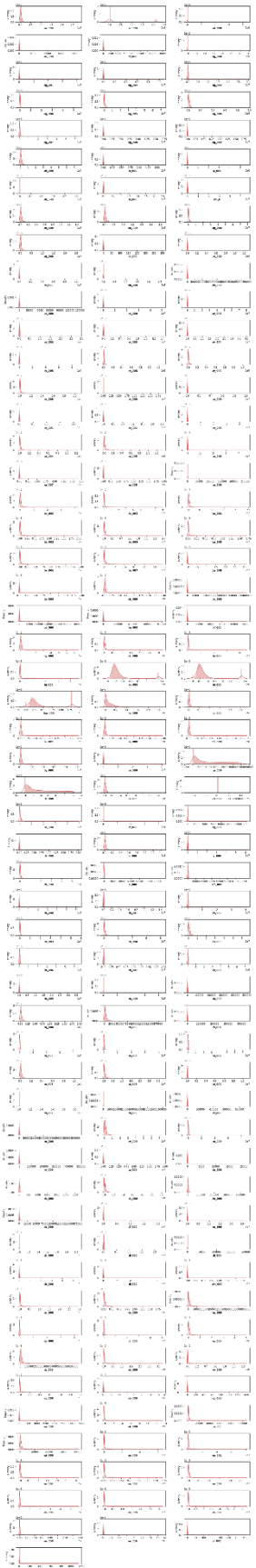
```
# Create a function which can evaluate models and return a report
def evaluate_models(X, y, models):
    '''
    This function takes in X and y and models dictionary as input
    It splits the data into Train Test split
    Iterates through the given model dictionary and evaluates the metrics
    Returns: Dataframe which contains report of all models metrics with cost
    '''
    # separate dataset into train and test
    X_train, X_test, y_train, y_test =
train_test_split(X,y,test_size=0.2,random_state=42)

    cost_list=[]
    models_list = []
    accuracy_list = []
```

```
    for i in range(len(list(models))):
        model = list(models.values())[i]
        model.fit(X_train, y_train) # Train model

        # Make predictions
        y_train_pred = model.predict(X_train)
        y_test_pred = model.predict(X_test)

        # Training set performance
        model_train_accuracy, model_train_f1,model_train_precision,\
        model_train_recall,model_train_rocauc_score=evaluate_clf(y_train
,y_train_pred)
        train_cost = total_cost(y_train, y_train_pred)


        # Test set performance
        model_test_accuracy,model_test_f1,model_test_precision,\
        model_test_recall,model_test_rocauc_score=evaluate_clf(y_test,
y_test_pred)
        test_cost = total_cost(y_test, y_test_pred)

        print(list(models.keys())[i])
        models_list.append(list(models.keys())[i])

        print('Model performance for Training set')
        print("- Accuracy: {:.4f}".format(model_train_accuracy))
        print('- F1 score: {:.4f}'.format(model_train_f1))
        print('- Precision: {:.4f}'.format(model_train_precision))
        print('- Recall: {:.4f}'.format(model_train_recall))
        print('- Roc Auc Score: {:.4f}'.format(model_train_rocauc_score))
        print(f'- COST: {train_cost}.')

        print('----------------------------------')

        print('Model performance for Test set')
        print('- Accuracy: {:.4f}'.format(model_test_accuracy))
        print('- F1 score: {:.4f}'.format(model_test_f1))
        print('- Precision: {:.4f}'.format(model_test_precision))
        print('- Recall: {:.4f}'.format(model_test_recall))
        print('- Roc Auc Score: {:.4f}'.format(model_test_rocauc_score))
        print(f'- COST: {test_cost}.')
        cost_list.append(test_cost)
        print('='*35)
        print('\n')

    report=pd.DataFrame(list(zip(models_list, cost_list)), columns=['Model
Name', 'Cost']).sort_values(by=["Cost"])

    return report
```

Plot distribution of all Independent Numerical variables

In [15]:

```
numeric_features = [feature for feature in df.columns if df[feature].dtype !=
'O']

plt.figure(figsize=(15, 100))
for i, col in enumerate(numeric_features):
    plt.subplot(60, 3, i+1)
    sns.distplot(x=df[col], color='indianred')
    plt.xlabel(col, weight='bold')
    plt.tight_layout()
```



Report

- As per the above plot most of the features are not normally distributed.
- Transformation of data is not of prime importance since it is a classification problem.
- Interpreting each and every column is not necessary as this is sensor data.

Evaluate Model on Different experiments

In [16]:

```
# Splitting X and y for all Experiments
X= df.drop('class', axis=1)
y = df['class']
```

- **Manually Encoding Target Variable**

In [17]:

```
y= y.replace({'pos': 1, 'neg': 0})
```

Experiment: 1 = KNN Imputer for Null values

Why Robust scaler and not Standard scaler?

- Scaling the data using Robust scaler
- Since most of the independent variables are not normally distributed we cannot use Standardscaler

Why Robust Scaler and not Minmax?

- because most of the feature has outliers. So Minmax will scale data according to Max values which is outlier.
- This Scaler removes the median and scales the data according to the quantile range (defaults to IQR: Interquartile Range). The IQR is the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile).

In [22]:

```
# Fit with robust scaler for KNN best K-selection experminet
robustscaler = RobustScaler()
X1 = robustscaler.fit_transform(X)
```

Why KNN Imputer?

- KNNImputer by scikit-learn is a widely used method to impute missing values. It is widely being observed as a replacement for traditional imputation techniques.
- KNNImputer helps to impute missing values present in the observations by finding the nearest neighbors with the Euclidean distance matrix.
- Here we Iterates through different K values and get accuracy and choose best K values.

Finding the optimal n_neighbour value for KNN imputer

In [23]:

```
results=[]
# define imputer
imputer = KNNImputer(n_neighbors=5, weights='uniform',
metric='nan_euclidean')
strategies = [str(i) for i in [1,3,5,7,9]]
for s in strategies:
    pipeline = Pipeline(steps=[('i', KNNImputer(n_neighbors=int(s))), ('m',
LogisticRegression())])
    scores = cross_val_score(pipeline, X1, y, scoring='accuracy', cv=2,
n_jobs=-1)
    results.append(scores)
    print('n_neighbors= %s || accuracy (%.4f)' % (s , mean(scores)))
n_neighbors= 1 || accuracy (0.7317)
n_neighbors= 3 || accuracy (0.7555)
n_neighbors= 5 || accuracy (0.7076)
n_neighbors= 7 || accuracy (0.6810)
n_neighbors= 9 || accuracy (0.6966)
```

We can observe n_neighbors=3 able to produce highest accuracy

Pipeline for KNN imputer

In [24]:

```
num_features = X.select_dtypes(exclude="object").columns

# Fit the KNN imputer with selected K-value
knn_pipeline = Pipeline(steps=[
    ('imputer', KNNImputer(n_neighbors=3)),
    ('RobustScaler', RobustScaler())
])
```

In [25]:

```
X_knn =knn_pipeline.fit_transform(X)
```

Handling Imbalanced data

SMOTE+TOMEK is one of such a hybrid technique that aims to clean overlapping data points for each of the classes distributed in sample space.

- This method combines the SMOTE ability to generate synthetic data for minority class and Tomek Links ability to remove the data that are identified as Tomek links from the majority class

- To add new data of minority class

1. Choose random data from the minority class.
2. Calculate the distance between the random data and its k nearest neighbors.
3. Multiply the difference with a random number between 0 and 1, then add the result to the minority class as a synthetic sample.

4. Repeat step number 2–3 until the desired proportion of minority class is met.

- To remove the tomek links of the majority class

1. Choose random data from the majority class.
2. If the random data's nearest neighbor is the data from the minority class (i.e. create the Tomek Link), then remove the Tomek Link.

- This is method instead of adding duplicate data it synthesises the new data based on the already avalialble classes. Hence we choose this as our imputer method for this problem.

In [26]:

```
from imblearn.combine import SMOTETomek

# Resampling the minority class. The strategy can be changed as required.
smt = SMOTETomek(random_state=42,sampling_strategy='minority',n_jobs=-1)
# Fit the model to generate the data.
X_res, y_res = smt.fit_resample(X_knn, y)
```

Initialize Default Models in a dictionary

In [27]:

```
# Dictionary which contains models for experiment
models = {
    "Random Forest": RandomForestClassifier(),
    "Decision Tree": DecisionTreeClassifier(),
    "Gradient Boosting": GradientBoostingClassifier(),
    "Logistic Regression": LogisticRegression(),
     "K-Neighbors Classifier": KNeighborsClassifier(),
    "XGBClassifier": XGBClassifier(),
     "CatBoosting Classifier": CatBoostClassifier(verbose=False),
    "AdaBoost Classifier": AdaBoostClassifier()
}
```

Fit KNN imputed data for models in dictionary

In [28]:

```
report_knn = evaluate_models(X_res, y_res, models)
Random Forest
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.
----------------------------------
Model performance for Test set
- Accuracy: 0.9924
```

- F1 score: 0.9925
- Precision: 0.9883
- Recall: 0.9967
- Roc Auc Score: 0.9925
- COST: 12330.
==========

Decision Tree
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.

Model performance for Test set
- Accuracy: 0.9865
- F1 score: 0.9866
- Precision: 0.9815
- Recall: 0.9917
- Roc Auc Score: 0.9865
- COST: 30310.
==========

Gradient Boosting
Model performance for Training set
- Accuracy: 0.9839
- F1 score: 0.9840
- Precision: 0.9803
- Recall: 0.9876
- Roc Auc Score: 0.9839
- COST: 179060.

Model performance for Test set
- Accuracy: 0.9844
- F1 score: 0.9844
- Precision: 0.9813
- Recall: 0.9876
- Roc Auc Score: 0.9844
- COST: 44820.
==========

Logistic Regression
Model performance for Training set
- Accuracy: 0.5886
- F1 score: 0.6932
- Precision: 0.5528
- Recall: 0.9291
- Roc Auc Score: 0.5885

- COST: 1205980.

Model performance for Test set
- Accuracy: 0.5820
- F1 score: 0.6896
- Precision: 0.5481
- Recall: 0.9297
- Roc Auc Score: 0.5823
- COST: 300230.
==========

K-Neighbors Classifier
Model performance for Training set
- Accuracy: 0.9814
- F1 score: 0.9816
- Precision: 0.9707
- Recall: 0.9927
- Roc Auc Score: 0.9814
- COST: 110900.

Model performance for Test set
- Accuracy: 0.9744
- F1 score: 0.9748
- Precision: 0.9605
- Recall: 0.9894
- Roc Auc Score: 0.9744
- COST: 39850.
==========

XGBClassifier
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.

Model performance for Test set
- Accuracy: 0.9962
- F1 score: 0.9962
- Precision: 0.9935
- Recall: 0.9989
- Roc Auc Score: 0.9962
- COST: 4460.
==========

CatBoosting Classifier
Model performance for Training set
- Accuracy: 0.9992

```
- F1 score: 0.9992
- Precision: 0.9989
- Recall: 0.9994
- Roc Auc Score: 0.9992
- COST: 8800.
----------------------------------
Model performance for Test set
- Accuracy: 0.9949
- F1 score: 0.9949
- Precision: 0.9922
- Recall: 0.9976
- Roc Auc Score: 0.9949
- COST: 9050.
==================================


AdaBoost Classifier
Model performance for Training set
- Accuracy: 0.9749
- F1 score: 0.9750
- Precision: 0.9736
- Recall: 0.9763
- Roc Auc Score: 0.9749
- COST: 339420.
----------------------------------
Model performance for Test set
- Accuracy: 0.9756
- F1 score: 0.9756
- Precision: 0.9745
- Recall: 0.9767
- Roc Auc Score: 0.9756
- COST: 83290.
==================================
```

Report for KNN Imputed data

In [29]:

```
report_knn
```

Out[29]:

	Model Name	Cost
5	XGBClassifier	4460
6	CatBoosting Classifier	9050

	Model Name	Cost
0	Random Forest	12330
1	Decision Tree	30310
4	K-Neighbors Classifier	39850
2	Gradient Boosting	44820
7	AdaBoost Classifier	83290
3	Logistic Regression	300230

Insights

- For the Experiment 1: Knn imputer has XGBoost classifier as the best Model
- Proceeding with further experiments

Experiment: 2 = Simple Imputer with Strategy Median

- SimpleImputer is a class in the sklearn.impute module that can be used to replace missing values in a dataset, using a variety of input strategies.
- Here we use SimpleImputer can also be used to impute multiple columns at once by passing in a list of column names. SimpleImputer will then replace missing values in all of the specified columns.

In [30]:

```
num_features = X.select_dtypes(exclude="object").columns

# Fit the Simple imputer with strategy median
median_pipeline = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('RobustScaler', RobustScaler())
])
```

In [31]:

```
# Fit X with median_pipeline
X_median = median_pipeline.fit_transform(X)
```

In [32]:

```
# Resampling the minority class. The strategy can be changed as required.
smt = SMOTETomek(random_state=42,sampling_strategy='minority')
# Fit the model to generate the data.
X_res, y_res = smt.fit_resample(X_median, y)
```

In [33]:

```
# Training the models
report_median = evaluate_models(X_res, y_res, models)
Random Forest
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.
----------------------------------
Model performance for Test set
- Accuracy: 0.9917
- F1 score: 0.9918
- Precision: 0.9866
- Recall: 0.9972
- Roc Auc Score: 0.9917
- COST: 10960.
==================================


Decision Tree
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.
----------------------------------
Model performance for Test set
- Accuracy: 0.9852
- F1 score: 0.9854
- Precision: 0.9796
- Recall: 0.9912
- Roc Auc Score: 0.9851
- COST: 32460.
==================================


Gradient Boosting
Model performance for Training set
- Accuracy: 0.9840
- F1 score: 0.9840
- Precision: 0.9813
- Recall: 0.9867
- Roc Auc Score: 0.9840
- COST: 190760.
----------------------------------
Model performance for Test set
- Accuracy: 0.9810
```

- F1 score: 0.9812
- Precision: 0.9757
- Recall: 0.9868
- Roc Auc Score: 0.9809
- COST: 48240.
==========

Logistic Regression
Model performance for Training set
- Accuracy: 0.6300
- F1 score: 0.7147
- Precision: 0.5807
- Recall: 0.9289
- Roc Auc Score: 0.6306
- COST: 1181600.

Model performance for Test set
- Accuracy: 0.6264
- F1 score: 0.7145
- Precision: 0.5813
- Recall: 0.9269
- Roc Auc Score: 0.6237
- COST: 305700.
==========

K-Neighbors Classifier
Model performance for Training set
- Accuracy: 0.9789
- F1 score: 0.9791
- Precision: 0.9683
- Recall: 0.9902
- Roc Auc Score: 0.9789
- COST: 146570.

Model performance for Test set
- Accuracy: 0.9718
- F1 score: 0.9724
- Precision: 0.9596
- Recall: 0.9854
- Roc Auc Score: 0.9716
- COST: 54430.
==========

XGBClassifier
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000

```
- COST: 500.
----------------------------------
Model performance for Test set
- Accuracy: 0.9951
- F1 score: 0.9951
- Precision: 0.9920
- Recall: 0.9983
- Roc Auc Score: 0.9950
- COST: 6570.
==================================


CatBoosting Classifier
Model performance for Training set
- Accuracy: 0.9996
- F1 score: 0.9996
- Precision: 0.9996
- Recall: 0.9995
- Roc Auc Score: 0.9996
- COST: 7110.
----------------------------------
Model performance for Test set
- Accuracy: 0.9939
- F1 score: 0.9939
- Precision: 0.9894
- Recall: 0.9986
- Roc Auc Score: 0.9938
- COST: 5760.
==================================


AdaBoost Classifier
Model performance for Training set
- Accuracy: 0.9749
- F1 score: 0.9749
- Precision: 0.9740
- Recall: 0.9757
- Roc Auc Score: 0.9749
- COST: 346790.
----------------------------------
Model performance for Test set
- Accuracy: 0.9727
- F1 score: 0.9731
- Precision: 0.9702
- Recall: 0.9760
- Roc Auc Score: 0.9727
- COST: 87120.
==================================
```

Report for Simple Imputer with median strategy

In [34]:

```
report_median
```

Out[34]:

	Model Name	Cost
6	CatBoosting Classifier	5760
5	XGBClassifier	6570
0	Random Forest	10960
1	Decision Tree	32460
2	Gradient Boosting	48240
4	K-Neighbors Classifier	54430
7	AdaBoost Classifier	87120
3	Logistic Regression	305700

Insights

- For the Experiment 2: Simple imputer with median strategy has Catboost classifier as the best Model
- Proceeding with further experiments

Experiment: 3 = MICE for Imputing Null values

- MICE stands for Multivariate Imputation By Chained Equations algorithm
- This technique by which we can effortlessly impute missing values in a dataset by looking at data from other columns and trying to estimate the best prediction for each missing value.
- ImputationKernel Creates a kernel dataset. This dataset can perform MICE on itself, and impute new data from models obtained during MICE.

In [35]:

```
import miceforest as mf

X_mice = X.copy()
kernel = mf.ImputationKernel(
  X_mice,
```

```
  save_all_iterations=True,
  random_state=1989
)# Run the MICE algorithm for 3 iterations kernel.mice(3)
```

In [36]:

```
X_mice = kernel.complete_data()
```

In [37]:

```
# fit robust scaler
mice_pipeline = Pipeline(steps=[
    ('RobustScaler', RobustScaler())
])
```

In [38]:

```
# Fit X with Mice imputer
X_mice= mice_pipeline.fit_transform(X_mice)
```

In [39]:

```
# Resampling the minority class. The strategy can be changed as required.
smt = SMOTETomek(random_state=42,sampling_strategy='minority', n_jobs=-1 )
# Fit the model to generate the data.
X_res, y_res = smt.fit_resample(X_mice, y)
```

In [40]:

```
# Training the models
report_mice = evaluate_models(X_res, y_res, models)
Random Forest
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.
----------------------------------
Model performance for Test set
- Accuracy: 0.9920
- F1 score: 0.9921
- Precision: 0.9880
- Recall: 0.9962
- Roc Auc Score: 0.9920
- COST: 14350.
==================================


Decision Tree
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.
```

```
---------------------------------
Model performance for Test set
- Accuracy: 0.9822
- F1 score: 0.9824
- Precision: 0.9783
- Recall: 0.9865
- Roc Auc Score: 0.9822
- COST: 49040.
=================================


Gradient Boosting
Model performance for Training set
- Accuracy: 0.9854
- F1 score: 0.9854
- Precision: 0.9825
- Recall: 0.9884
- Roc Auc Score: 0.9854
- COST: 166940.
---------------------------------
Model performance for Test set
- Accuracy: 0.9813
- F1 score: 0.9815
- Precision: 0.9772
- Recall: 0.9858
- Roc Auc Score: 0.9813
- COST: 51620.
=================================


Logistic Regression
Model performance for Training set
- Accuracy: 0.6272
- F1 score: 0.7191
- Precision: 0.5766
- Recall: 0.9551
- Roc Auc Score: 0.6276
- COST: 826010.
---------------------------------
Model performance for Test set
- Accuracy: 0.6200
- F1 score: 0.7160
- Precision: 0.5729
- Recall: 0.9545
- Roc Auc Score: 0.6187
- COST: 210090.
=================================


K-Neighbors Classifier
Model performance for Training set
- Accuracy: 0.9792
- F1 score: 0.9795
```

- Precision: 0.9677
- Recall: 0.9915
- Roc Auc Score: 0.9792
- COST: 128270.

Model performance for Test set
- Accuracy: 0.9704
- F1 score: 0.9710
- Precision: 0.9566
- Recall: 0.9858
- Roc Auc Score: 0.9703
- COST: 53150.
==========

XGBClassifier
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.

Model performance for Test set
- Accuracy: 0.9959
- F1 score: 0.9960
- Precision: 0.9928
- Recall: 0.9991
- Roc Auc Score: 0.9959
- COST: 3510.
==========

CatBoosting Classifier
Model performance for Training set
- Accuracy: 0.9996
- F1 score: 0.9996
- Precision: 0.9997
- Recall: 0.9995
- Roc Auc Score: 0.9996
- COST: 6580.

Model performance for Test set
- Accuracy: 0.9940
- F1 score: 0.9941
- Precision: 0.9903
- Recall: 0.9979
- Roc Auc Score: 0.9940
- COST: 8190.
==========

```
AdaBoost Classifier
Model performance for Training set
- Accuracy: 0.9758
- F1 score: 0.9758
- Precision: 0.9769
- Recall: 0.9746
- Roc Auc Score: 0.9758
- COST: 361950.
----------------------------------
Model performance for Test set
- Accuracy: 0.9742
- F1 score: 0.9743
- Precision: 0.9746
- Recall: 0.9740
- Roc Auc Score: 0.9742
- COST: 93290.
==================================
```

Report for MICE Imputer algorithm

In [41]:

```
report_mice
```

Out[41]:

	Model Name	Cost
5	XGBClassifier	3510
6	CatBoosting Classifier	8190
0	Random Forest	14350
1	Decision Tree	49040
2	Gradient Boosting	51620
4	K-Neighbors Classifier	53150
7	AdaBoost Classifier	93290

	Model Name	Cost
3	Logistic Regression	210090

Insights

- For the Experiment 3: Mice imputer has XGBoost classifier as the best Model
- Proceeding with further experiments

Experiment: 4 = Simple Imputer with Strategy Constant

- Another strategy which can be used is replacing missing values with a fixed (constant) value.
- To do this, specify "constant" for strategy and specify the fill value using the fill_value parameter

In [42]:

```
# Create a pipeline with simple imputer with strategy constant and fill value
0
constant_pipeline = Pipeline(steps=[
    ('Imputer', SimpleImputer(strategy='constant', fill_value=0)),
    ('RobustScaler', RobustScaler())
])
```

In [44]:

```
X_const =constant_pipeline.fit_transform(X)
```

In [45]:

```
# Resampling the minority class. The strategy can be changed as required.
smt = SMOTETomek(random_state=42,sampling_strategy='minority', n_jobs=-1 )
# Fit the model to generate the data.
X_res, y_res = smt.fit_resample(X_const, y)
```

In [46]:

```
# training the models
report_const = evaluate_models(X_res, y_res, models)
Random Forest
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.
----------------------------------
Model performance for Test set
- Accuracy: 0.9933
- F1 score: 0.9933
- Precision: 0.9894
- Recall: 0.9973
- Roc Auc Score: 0.9933
- COST: 10250.
```

```
==================================


Decision Tree
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.
----------------------------------
Model performance for Test set
- Accuracy: 0.9882
- F1 score: 0.9883
- Precision: 0.9831
- Recall: 0.9936
- Roc Auc Score: 0.9882
- COST: 23700.
==================================


Gradient Boosting
Model performance for Training set
- Accuracy: 0.9829
- F1 score: 0.9829
- Precision: 0.9799
- Recall: 0.9860
- Roc Auc Score: 0.9829
- COST: 201670.
----------------------------------
Model performance for Test set
- Accuracy: 0.9806
- F1 score: 0.9807
- Precision: 0.9774
- Recall: 0.9841
- Roc Auc Score: 0.9806
- COST: 57600.
==================================


Logistic Regression
Model performance for Training set
- Accuracy: 0.6710
- F1 score: 0.7482
- Precision: 0.6057
- Recall: 0.9784
- Roc Auc Score: 0.6712
- COST: 481530.
----------------------------------
Model performance for Test set
- Accuracy: 0.6702
- F1 score: 0.7487
```

- Precision: 0.6058
- Recall: 0.9800
- Roc Auc Score: 0.6693
- COST: 115350.
==========

K-Neighbors Classifier
Model performance for Training set
- Accuracy: 0.9803
- F1 score: 0.9805
- Precision: 0.9682
- Recall: 0.9932
- Roc Auc Score: 0.9803
- COST: 105130.

Model performance for Test set
- Accuracy: 0.9750
- F1 score: 0.9754
- Precision: 0.9626
- Recall: 0.9885
- Roc Auc Score: 0.9749
- COST: 43200.
==========

XGBClassifier
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 500.

Model performance for Test set
- Accuracy: 0.9964
- F1 score: 0.9965
- Precision: 0.9936
- Recall: 0.9993
- Roc Auc Score: 0.9964
- COST: 2950.
==========

CatBoosting Classifier
Model performance for Training set
- Accuracy: 0.9991
- F1 score: 0.9991
- Precision: 0.9991
- Recall: 0.9992
- Roc Auc Score: 0.9991
- COST: 11750.

```
---------------------------------
Model performance for Test set
- Accuracy: 0.9961
- F1 score: 0.9961
- Precision: 0.9934
- Recall: 0.9989
- Roc Auc Score: 0.9961
- COST: 4470.
==================================


AdaBoost Classifier
Model performance for Training set
- Accuracy: 0.9735
- F1 score: 0.9735
- Precision: 0.9731
- Recall: 0.9739
- Roc Auc Score: 0.9735
- COST: 374050.
---------------------------------
Model performance for Test set
- Accuracy: 0.9734
- F1 score: 0.9735
- Precision: 0.9745
- Recall: 0.9724
- Roc Auc Score: 0.9734
- COST: 98790.
==================================
```

Report for Simple Imputer with Constant strategy

In [47]:

```
report_const
```

Out[47]:

	Model Name	Cost
5	XGBClassifier	2950
6	CatBoosting Classifier	4470
0	Random Forest	10250

	Model Name	Cost
1	Decision Tree	23700
4	K-Neighbors Classifier	43200
2	Gradient Boosting	57600
7	AdaBoost Classifier	98790
3	Logistic Regression	115350

Insights

- For the Experiment 4: Simple imputer with constant strategy has XGBoost classifier as the best Model
- Proceeding with further experiments

Experiment: 5 = Simple Imputer with Strategy Mean

- Another strategy which can be used is replacing missing values with mean
- Here we replace the missing values with the mean of the column

In [49]:

```
# Create a pipeline with Simple imputer with strategy mean
mean_pipeline = Pipeline(steps=[
    ('Imputer', SimpleImputer(strategy='mean')),
    ('RobustScaler', RobustScaler())
])
```

In [50]:

```
X_mean = mean_pipeline.fit_transform(X)
```

In [51]:

```
# Resampling the minority class. The strategy can be changed as required.
smt = SMOTETomek(random_state=42,sampling_strategy='minority' , n_jobs=-1)
# Fit the model to generate the data.
X_res, y_res = smt.fit_resample(X_mean, y)
```

In [52]:

```
# Training all models
report_mean = evaluate_models(X_res, y_res, models)
Random Forest
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
```

- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.

Model performance for Test set
- Accuracy: 0.9945
- F1 score: 0.9945
- Precision: 0.9910
- Recall: 0.9980
- Roc Auc Score: 0.9944
- COST: 7640.
==========

Decision Tree
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 0.

Model performance for Test set
- Accuracy: 0.9893
- F1 score: 0.9894
- Precision: 0.9867
- Recall: 0.9921
- Roc Auc Score: 0.9893
- COST: 28940.
==========

Gradient Boosting
Model performance for Training set
- Accuracy: 0.9839
- F1 score: 0.9839
- Precision: 0.9810
- Recall: 0.9868
- Roc Auc Score: 0.9839
- COST: 190870.

Model performance for Test set
- Accuracy: 0.9812
- F1 score: 0.9813
- Precision: 0.9771
- Recall: 0.9855
- Roc Auc Score: 0.9812
- COST: 52630.
==========

Logistic Regression
Model performance for Training set
- Accuracy: 0.9533
- F1 score: 0.9524
- Precision: 0.9711
- Recall: 0.9344
- Roc Auc Score: 0.9533
- COST: 930810.

Model performance for Test set
- Accuracy: 0.9539
- F1 score: 0.9532
- Precision: 0.9694
- Recall: 0.9376
- Roc Auc Score: 0.9539
- COST: 222090.
==================================

K-Neighbors Classifier
Model performance for Training set
- Accuracy: 0.9868
- F1 score: 0.9869
- Precision: 0.9754
- Recall: 0.9988
- Roc Auc Score: 0.9868
- COST: 24580.

Model performance for Test set
- Accuracy: 0.9837
- F1 score: 0.9840
- Precision: 0.9701
- Recall: 0.9983
- Roc Auc Score: 0.9837
- COST: 8170.
==================================

XGBClassifier
Model performance for Training set
- Accuracy: 1.0000
- F1 score: 1.0000
- Precision: 1.0000
- Recall: 1.0000
- Roc Auc Score: 1.0000
- COST: 500.

Model performance for Test set
- Accuracy: 0.9962
- F1 score: 0.9962
- Precision: 0.9936
- Recall: 0.9987
- Roc Auc Score: 0.9962

```
- COST: 4950.
==================================


CatBoosting Classifier
Model performance for Training set
- Accuracy: 0.9993
- F1 score: 0.9993
- Precision: 0.9991
- Recall: 0.9994
- Roc Auc Score: 0.9993
- COST: 9240.
----------------------------------
Model performance for Test set
- Accuracy: 0.9947
- F1 score: 0.9948
- Precision: 0.9915
- Recall: 0.9980
- Roc Auc Score: 0.9947
- COST: 7600.
==================================


AdaBoost Classifier
Model performance for Training set
- Accuracy: 0.9746
- F1 score: 0.9746
- Precision: 0.9754
- Recall: 0.9737
- Roc Auc Score: 0.9746
- COST: 376900.
----------------------------------
Model performance for Test set
- Accuracy: 0.9746
- F1 score: 0.9746
- Precision: 0.9745
- Recall: 0.9748
- Roc Auc Score: 0.9746
- COST: 90800.
==================================
```

Report for Simple imputer with strategy mean

In [53]:

```
report_mean
```

Out[53]:

	Model Name	Cost
5	XGBClassifier	4950
6	CatBoosting Classifier	7600
0	Random Forest	7640
4	K-Neighbors Classifier	8170
1	Decision Tree	28940
2	Gradient Boosting	52630
7	AdaBoost Classifier	90800
3	Logistic Regression	222090

Experiment: 5 = Principle component analysis with imputing median

- Principal component analysis is a technique for feature extraction — so it combines our input variables in a specific way, then we can drop the "least important" variables while still retaining the most valuable parts of all of the variables!
- As the dataset has 164 columns we can try PCA and check our metrics Cost

In [152]:

```
from sklearn.decomposition import PCA
```

In [153]:

```
pca_pipeline = Pipeline(steps=[
    ('Imputer', SimpleImputer(strategy='constant', fill_value=0)),
    ('RobustScaler', RobustScaler())
])
```

In [154]:

```
X_pca = pca_pipeline.fit_transform(X)
```

In [155]:

```
#Applying PCA
from sklearn.decomposition import PCA
var_ratio={}
for n in range(2,150):
```

```
    pc=PCA(n_components=n)
    df_pca=pc.fit(X_pca)
    var_ratio[n]=sum(df_pca.explained_variance_ratio_)
```

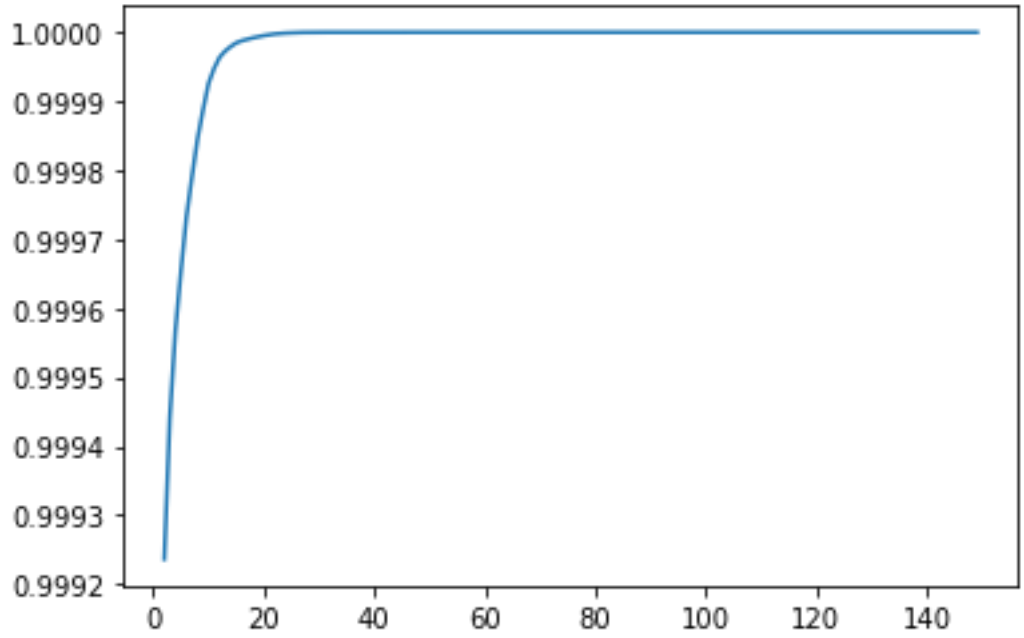
Variance Plot

In [156]:

```
# plotting variance ratio
pd.Series(var_ratio).plot()
```

Out[156]:

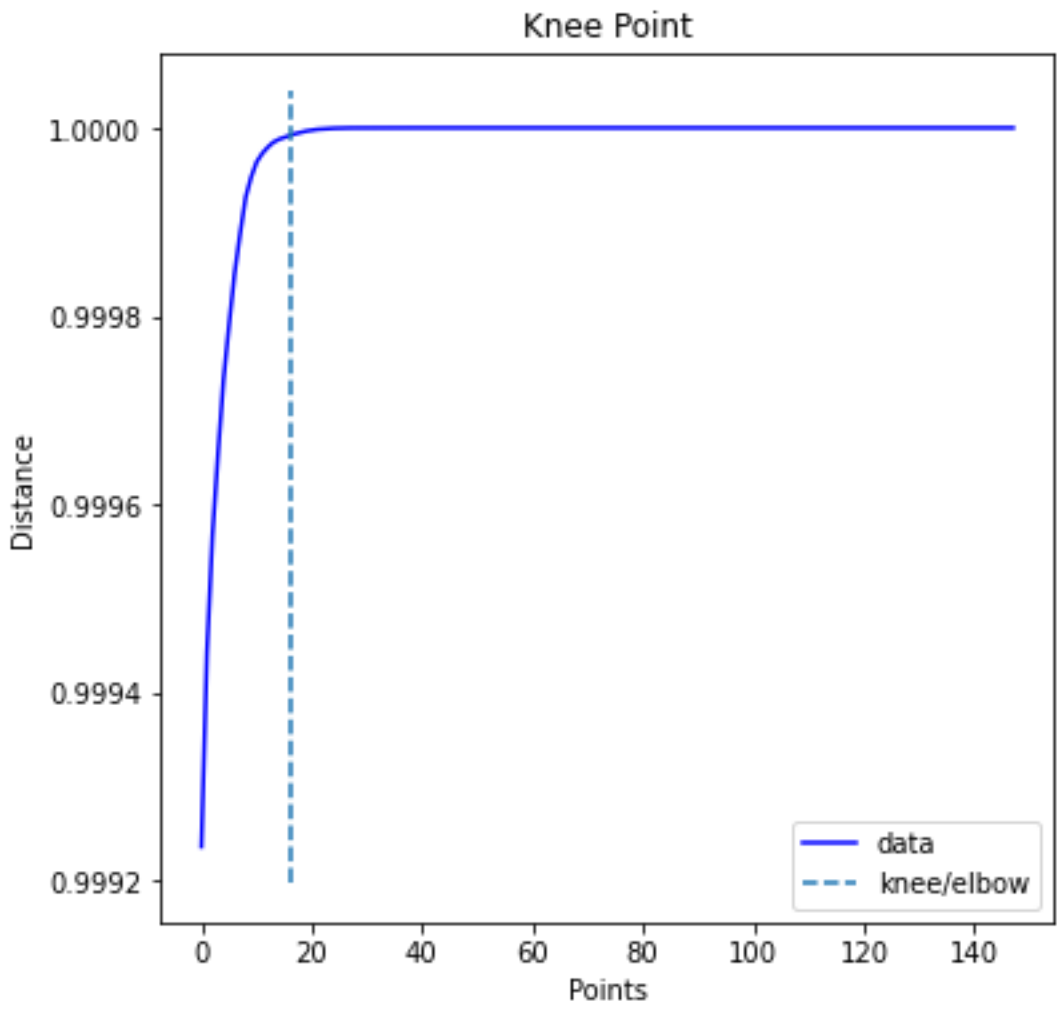
```
<AxesSubplot:>
```



Kneed algorithm to find the elbow point

In [157]:

```
from kneed import KneeLocator

i = np.arange(len(var_ratio))
variance_ratio= list(var_ratio.values())
components=  list(var_ratio.keys())
knee = KneeLocator(i, variance_ratio, S=1, curve='concave',
interp_method='polynomial')

fig = plt.figure(figsize=(5, 5))
knee.plot_knee()
plt.xlabel("Points")
plt.ylabel("Distance")
plt.show()
k= components[knee.knee]
print('Knee Locator k =', k)
<Figure size 360x360 with 0 Axes>
```




```
Knee Locator k = 18
```

In [159]:

```
# Reducing the dimensions of the data
pca_final=PCA(n_components=18,random_state=42).fit(X_res)

reduced=pca_final.fit_transform(X_pca)
```

In [160]:

```
# Resampling the minority class. The strategy can be changed as required.
smt = SMOTETomek(random_state=42,sampling_strategy='minority', n_jobs=-1)
# Fit the model to generate the data.
X_res, y_res = smt.fit_resample(reduced, y)
```

In [161]:

```
# Training all models
report_pca = evaluate_models(X_res,y_res, models)
Random Forest
Model performance for Training set
- Accuracy: 0.9986
```

- F1 score: 0.9986
- Precision: 1.0000
- Recall: 0.9972
- Roc Auc Score: 0.9986
- COST: 39500.

Model performance for Test set
- Accuracy: 0.9830
- F1 score: 0.9831
- Precision: 0.9787
- Recall: 0.9876
- Roc Auc Score: 0.9830
- COST: 45010.
==========

Decision Tree
Model performance for Training set
- Accuracy: 0.9986
- F1 score: 0.9986
- Precision: 1.0000
- Recall: 0.9972
- Roc Auc Score: 0.9986
- COST: 39500.

Model performance for Test set
- Accuracy: 0.9741
- F1 score: 0.9743
- Precision: 0.9708
- Recall: 0.9778
- Roc Auc Score: 0.9741
- COST: 80060.
==========

Gradient Boosting
Model performance for Training set
- Accuracy: 0.9416
- F1 score: 0.9414
- Precision: 0.9442
- Recall: 0.9385
- Roc Auc Score: 0.9416
- COST: 874480.

Model performance for Test set
- Accuracy: 0.9381
- F1 score: 0.9380
- Precision: 0.9440
- Recall: 0.9320
- Roc Auc Score: 0.9382
- COST: 242380.
==========

Logistic Regression
Model performance for Training set
- Accuracy: 0.8516
- F1 score: 0.8313
- Precision: 0.9618
- Recall: 0.7319
- Roc Auc Score: 0.8515
- COST: 3753110.

Model performance for Test set
- Accuracy: 0.8490
- F1 score: 0.8291
- Precision: 0.9597
- Recall: 0.7298
- Roc Auc Score: 0.8494
- COST: 950150.
==================================

K-Neighbors Classifier
Model performance for Training set
- Accuracy: 0.9722
- F1 score: 0.9724
- Precision: 0.9640
- Recall: 0.9809
- Roc Auc Score: 0.9722
- COST: 276720.

Model performance for Test set
- Accuracy: 0.9652
- F1 score: 0.9657
- Precision: 0.9565
- Recall: 0.9751
- Roc Auc Score: 0.9652
- COST: 90610.
==================================

XGBClassifier
Model performance for Training set
- Accuracy: 0.9905
- F1 score: 0.9905
- Precision: 0.9904
- Recall: 0.9906
- Roc Auc Score: 0.9905
- COST: 133670.

Model performance for Test set
- Accuracy: 0.9788
- F1 score: 0.9790
- Precision: 0.9746
- Recall: 0.9835

```
- Roc Auc Score: 0.9788
- COST: 59800.
==================================


CatBoosting Classifier
Model performance for Training set
- Accuracy: 0.9824
- F1 score: 0.9824
- Precision: 0.9808
- Recall: 0.9840
- Roc Auc Score: 0.9824
- COST: 228370.
----------------------------------
Model performance for Test set
- Accuracy: 0.9743
- F1 score: 0.9745
- Precision: 0.9702
- Recall: 0.9789
- Roc Auc Score: 0.9743
- COST: 76110.
==================================


AdaBoost Classifier
Model performance for Training set
- Accuracy: 0.9215
- F1 score: 0.9207
- Precision: 0.9291
- Recall: 0.9126
- Roc Auc Score: 0.9215
- COST: 1240960.
----------------------------------
Model performance for Test set
- Accuracy: 0.9195
- F1 score: 0.9189
- Precision: 0.9294
- Recall: 0.9086
- Roc Auc Score: 0.9196
- COST: 325340.
==================================
```

Report for PCA and Mean imputed data

In [162]:

```
report_pca
```

Out[162]:

	Model Name	Cost
0	Random Forest	45010
5	XGBClassifier	59800
6	CatBoosting Classifier	76110
1	Decision Tree	80060
4	K-Neighbors Classifier	90610
2	Gradient Boosting	242380
7	AdaBoost Classifier	325340
3	Logistic Regression	950150

Final Model

In [168]:

```
from prettytable import PrettyTable

pt=PrettyTable()
pt.field_names=["Model","Imputation_method","Total_cost"]
pt.add_row(["XGBClassifier","Simple Imputer-Constant","2950"])
pt.add_row(["XGBClassifier","Mice","3510"])
pt.add_row(["XGBClassifier","Knn-Imputer","4460"])
pt.add_row(["XGBClassifier","Simple Imputer-Mean","4950"])
pt.add_row(["CatBoostClassifier","Median","5760"])
pt.add_row(["Random Forest","PCA","34150"])
print(pt)
```

```
+--------------------+-------------------------+------------+
|       Model        |    Imputation_method    | Total_cost |
+--------------------+-------------------------+------------+
|   XGBClassifier    | Simple Imputer-Constant |    2950    |
|   XGBClassifier    |           Mice          |    3510    |
|   XGBClassifier    |       Knn-Imputer       |    4460    |
|   XGBClassifier    |   Simple Imputer-Mean   |    4950    |
| CatBoostClassifier |          Median         |    5760    |
|   Random Forest    |           PCA           |   34150    |
```

```
+--------------------+------------------------+------------+
```

Report

- From the final report we can see than XGBClassifier with Simple imputer with strategy constant has performed the best with cost of 2950

Fitting the Final model and get reports

In [169]:

```
final_model = XGBClassifier()

# Resampling the minority class. The strategy can be changed as required.
smt = SMOTETomek(random_state=42,sampling_strategy='minority', n_jobs=-1)
# Fit the model to generate the data.
X_res, y_res = smt.fit_resample(X_const, y)
```

In [173]:

```
X_train, X_test, y_train, y_test =
train_test_split(X_res,y_res,test_size=0.2,random_state=42)

final_model = final_model.fit(X_train, y_train)
y_pred = final_model.predict(X_test)
```

In [176]:

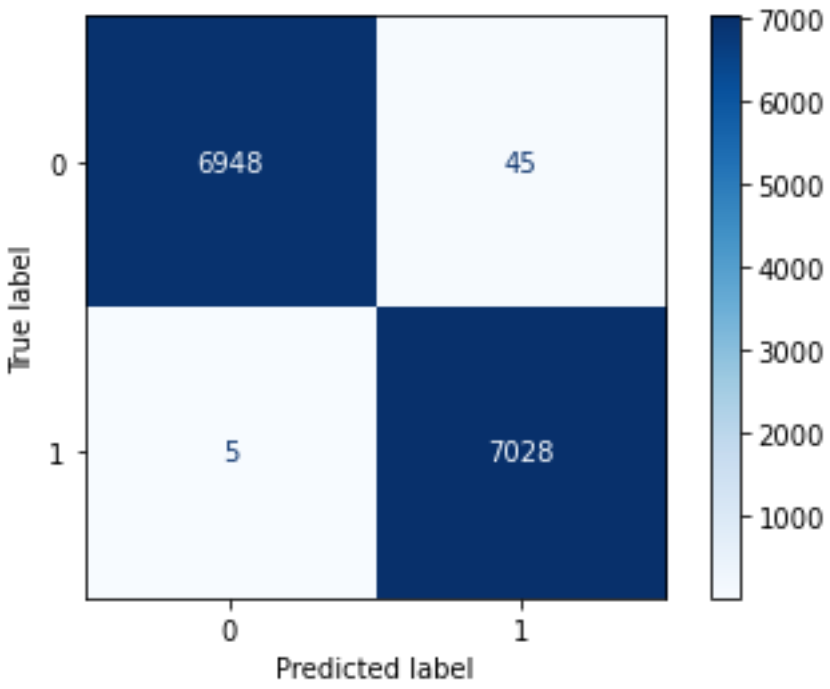
```
print("Final XGBoost Classifier Accuracy Score (Train) :",
final_model.score(X_train,y_train))
print("Final XGBoost Classifier Accuracy Score (Test) :",
accuracy_score(y_pred,y_test))
Final XGBoost Classifier Accuracy Score (Train) : 0.9999821759589335
Final XGBoost Classifier Accuracy Score (Test) : 0.9964351917866818
```

In [182]:

```
print("Final XGBoost Classifier Cost Metric(Test) :",total_cost(y_test,
y_pred))
Final XGBoost Classifier Cost Metric(Test) : 2950
```

In [181]:

```
from sklearn.metrics import plot_confusion_matrix

#plots Confusion matrix
plot_confusion_matrix(final_model, X_test, y_test, cmap='Blues',
values_format='d')
```



The best Model is XGBoost Classifier with 99.6% accuracy and cost of 2950

Product details:

1) **Data sources:**

   https://www.kaggle.com/datasets/arashnic/sensor-fault-detection-data

2) **Team required to develop:**
   - **Machine learning engineering**
   - **Data researcher**
   - **ETL Team**
   - **Web developer**

Conclusion:

In conclusion, a sensor-fault-detection system using machine learning can help improve the reliability and safety of critical systems by detecting faults and failures in real-time. By

utilizing machine learning algorithms, the system can analyze sensor data and identify patterns and anomalies that indicate a fault. The system can be developed using a range of concepts, such as anomaly detection, pattern recognition, fault signature analysis, multi-sensor fusion, and predictive maintenance, depending on the specific application and operational constraints. The success of the system depends on the availability and quality of the data used to train and validate the machine learning model, the selection of the appropriate machine learning algorithm, and the development of an effective fault detection system that integrates the machine learning model. Regular maintenance and updates are necessary to ensure that the system continues to perform effectively over time. Overall, a sensor-fault-detection system using machine learning can provide significant benefits in terms of safety, reliability, and cost savings.