**Summary**

Predictive maintenance (**PdM**) is a popular application of predictive analytics that can help businesses in several industries achieve high asset utilization and savings in operational costs.

## Business case for predictive maintenance

Businesses require critical equipment to be running at peak efficiency and utilization to realize their return on capital investments. These assets could range from aircraft engines, turbines, elevators, or industrial chillers - that cost millions - down to everyday appliances like photocopiers, coffee machines, or water coolers.

* By default, most businesses rely on corrective maintenance, where parts are replaced as and when they fail. Corrective maintenance ensures parts are used completely (therefore not wasting component life), but costs the business in downtime, labor, and unscheduled maintenance requirements (off hours, or inconvenient locations).
* At the next level, businesses practice preventive maintenance, where they determine the useful lifespan for a part, and maintain or replace it before a failure. Preventive maintenance avoids unscheduled and catastrophic failures. But the high costs of scheduled downtime, under-utilization of the component before its full lifetime of use, and labor still remain.
* The goal of predictive maintenance is to optimize the balance between corrective and preventative maintenance, by enabling just in time replacement of components. This approach only replaces those components when they are close to a failure. By extending component lifespans (compared to preventive maintenance) and reducing unscheduled maintenance and labor costs (over corrective maintenance), businesses can gain cost savings and competitive advantages.

## Business problems in PdM

Businesses face high operational risk due to unexpected failures and have limited insight into the root cause of problems in complex systems. Some of the key business questions are:

* Detect anomalies in equipment or system performance or functionality.
* Predict whether an asset may fail in the near future.
* Estimate the remaining useful life of an asset.
* Identify the main causes of failure of an asset.
* Identify what maintenance actions need to be done, by when, on an asset.

Typical goal statements from PdM are:

* Reduce operational risk of mission critical equipment.
* Increase rate of return on assets by predicting failures before they occur.
* Control cost of maintenance by enabling just-in-time maintenance operations.
* Lower customer attrition, improve brand image, and lost sales.
* Lower inventory costs by reducing inventory levels by predicting the reorder point.
* Discover patterns connected to various maintenance problems.
* Provide KPIs (key performance indicators) such as health scores for asset conditions.
* Estimate remaining lifespan of assets.
* Recommend timely maintenance activities.
* Enable just in time inventory by estimating order dates for replacement of parts.

These goal statements are the starting points for:

* data scientists to analyze and solve specific predictive problems.
* cloud architects and developers to put together an end to end solution.

## Qualifying problems for predictive maintenance

It is important to emphasize that not all use cases or business problems can be effectively solved by PdM. There are three important qualifying criteria that need to be considered during problem selection:

* The problem has to be predictive in nature; that is, there should be a target or an outcome to predict. The problem should also have a clear path of action to prevent failures when they are detected.
* The problem should have a record of the operational history of the equipment that contains both good and bad outcomes. The set of actions taken to mitigate bad outcomes should also be available as part of these records. Error reports, maintenance logs of performance degradation, repair, and replace logs are also important. In addition, repairs undertaken to improve them, and replacement records are also useful.
* The recorded history should be reflected in relevant data that is of sufficient enough quality to support the use case. For more information about data relevance and sufficiency, see [Data requirements for predictive maintenance](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Data-requirements-for-predictive-maintenance).
* Finally, the business should have domain experts who have a clear understanding of the problem. They should be aware of the internal processes and practices to be able to help the analyst understand and interpret the data. They should also be able to make the necessary changes to existing business processes to help collect the right data for the problems, if needed.

## Sample PdM use cases

This section focuses on a collection of PdM use cases from several industries such as Aerospace, Utilities, and Transportation. Each section starts with a business problem, and discusses the benefits of PdM, the relevant data surrounding the business problem, and finally the benefits of a PdM solution.

| **Business Problem** | **Benefits from PdM** |
| --- | --- |
| **Aviation** |  |
| Flight delay and cancellations due to mechanical problems. Failures that cannot be repaired in time may cause flights to be canceled, and disrupt scheduling and operations. | PdM solutions can predict the probability of an aircraft being delayed or canceled due to mechanical failures. |
| Aircraft engine parts failure: Aircraft engine part replacements are among the most common maintenance tasks within the airline industry. Maintenance solutions require careful management of component stock availability, delivery, and planning | Being able to gather intelligence on component reliability leads to substantial reduction on investment costs. |
| **Finance** |  |
| ATM failure is a common problem within the banking industry. The problem here is to report the probability that an ATM cash withdrawal transaction gets interrupted due to a paper jam or part failure in the cash dispenser. Based on predictions of transaction failures, ATMs can be serviced proactively to prevent failures from occurring. | Rather than allow the machine to fail midway through a transaction, the desirable alternative is to program the machine to deny service based on the prediction. |
| **Energy** |  |
| Wind turbine failures: Wind turbines are the main energy source in environmentally responsible countries, and involve high capital costs. A key component in wind turbines is the generator motor. its failure renders the turbine ineffective. It is also highly expensive to fix. | Predicting KPIs such as MTTF (mean time to failure) can help the energy companies prevent turbine failures, and ensure minimal downtime. Failure probabilities will inform technicians to monitor turbines that are likely to fail soon, and schedule time-based maintenance regimes. Predictive models provide insights into different factors that contribute to the failure, which helps technicians better understand the root causes of problems. |
| Circuit breaker failures: Distribution of electricity to homes and businesses requires power lines to be operational at all times to guarantee energy delivery. Circuit breakers help limit or avoid damage to power lines during overloading or adverse weather conditions. The business problem here is to predict circuit breaker failures. | PdM solutions help reduce repair costs and increase the lifespan of equipment such as circuit breakers. They help improve the quality of the power network by reducing unexpected failures and service interruptions. |
| **Transportation and logistics** |  |
| Elevator door failures: Large elevator companies provide a full stack service for millions of functional elevators around the world. Elevator safety, reliability, and uptime are the main concerns for their customers. These companies track these and various other attributes via sensors, to help them with corrective and preventive maintenance. In an elevator, the most prominent customer problem is malfunctioning elevator doors. The business problem in this case is to provide a knowledge base predictive application that predicts the potential causes of door failures. | Elevators are capital investments for potentially a 20-30 year lifespan. So each potential sale can be highly competitive; hence expectations for service and support are high. Predictive maintenance can provide these companies with an advantage over their competitors in their product and service offerings. |
| Wheel failures: Wheel failures account for half of all train derailments and cost billions to the global rail industry. Wheel failures also cause rails to deteriorate, sometimes causing the rail to break prematurely. Rail breaks lead to catastrophic events such as derailments. To avoid such instances, railways monitor the performance of wheels and replace them in a preventive manner. The business problem here is the prediction of wheel failures. | Predictive maintenance of wheels will help with just-in-time replacement of wheels |
| Subway train door failures: A major reason for delays in subway operations is door failures of train cars. The business problem here is to predict train door failures. | Early awareness of a door failure, or the number of days until a door failure, will help the business optimize train door servicing schedules. |

The next section gets into the details of how to realize the PdM benefits discussed above.

## Data Science for predictive maintenance

This section provides general guidelines of data science principles and practice for PdM. It is intended to help a TDM, solution architect, or a developer understand the prerequisites and process for building end-to-end AI applications for PdM. You can read this section along with a review of the demos and proof-of-concept templates listed in [Solution Templates for predictive maintenance](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Solution-templates-for-predictive-maintenance). You can then use these principles and best practices to implement your PdM solution in Azure.

Note

This guide is NOT intended to teach the reader Data Science. Several helpful sources are provided for further reading in the section for [training resources for predictive maintenance](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Training-resources-for-predictive-maintenance). The [solution templates](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Solution-templates-for-predictive-maintenance) listed in the guide demonstrate some of these AI techniques for specific PdM problems.

## Data requirements for predictive maintenance

The success of any learning depends on (a) the quality of what is being taught, and (b) the ability of the learner. Predictive models learn patterns from historical data, and predict future outcomes with certain probability based on these observed patterns. A model's predictive accuracy depends on the relevancy, sufficiency, and quality of the training and test data. The new data that is 'scored' using this model should have the same features and schema as the training/test data. The feature characteristics (type, density, distribution, and so on) of new data should match that of the training and test data sets. The focus of this section is on such data requirements.

### Relevant data

First, the data has to be relevant to the problem. Consider the wheel failure use case discussed above - the training data should contain features related to the wheel operations. If the problem was to predict the failure of the traction system, the training data has to encompass all the different components for the traction system. The first case targets a specific component whereas the second case targets the failure of a larger subsystem. The general recommendation is to design prediction systems about specific components rather than larger subsystems, since the latter will have more dispersed data. The domain expert (see [Qualifying problems for predictive maintenance](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Qualifying-problems-for-predictive-maintenance)) should help in selecting the most relevant subsets of data for the analysis. The relevant data sources are discussed in greater detail in [Data preparation for predictive maintenance](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Data-preparation-for-predictive-maintenance).

### Sufficient data

Two questions are commonly asked with regard to failure history data: (1) "How many failure events are required to train a model?" (2) "How many records is considered as "enough"?" There are no definitive answers, but only rules of thumb. For (1), more the number of failure events, better the model. For (2), and the exact number of failure events depends on the data and the context of the problem being solved. But on the flip side, if a machine fails too often then the business will replace it, which will reduce failure instances. Here again, the guidance from the domain expert is important. However, there are methods to cope with the issue of rare events. They are discussed in the section [Handling imbalanced data](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Handling-imbalanced-data).

### Quality data

The quality of the data is critical - each predictor attribute value must be accurate in conjunction with the value of the target variable. Data quality is a well-studied area in statistics and data management, and hence out of scope for this guide.

Note

There are several resources and enterprise products to deliver quality data. A sample of references is provided below:

* Dasu, T, Johnson, T., Exploratory Data Mining and Data Cleaning, Wiley, 2003.
* [Exploratory Data Analysis, Wikipedia](https://en.wikipedia.org/wiki/Exploratory_data_analysis)
* [Hellerstein, J, Quantitative Data Cleaning for Large Databases](http://db.cs.berkeley.edu/jmh/papers/cleaning-unece.pdf)
* [de Jonge, E, van der loo, M, Introduction to Data Cleaning with R](https://cran.r-project.org/doc/contrib/de_Jonge+van_der_Loo-Introduction_to_data_cleaning_with_R.pdf)

## Data preparation for predictive maintenance

### Data sources

The relevant data sources for predictive maintenance include, but are not limited to:

* Failure history
* Maintenance/repair history
* Machine operating conditions
* Equipment metadata

#### Failure history

Failure events are rare in PdM applications. However, when building prediction models, the algorithm needs to learn about a component's normal operational pattern, as well as its failure patterns. So the training data should contain sufficient number of examples from both categories. Maintenance records and parts replacement history are good sources to find failure events. With the help of some domain knowledge, anomalies in the training data can also be defined as failures.

#### Maintenance/repair history

Maintenance history of an asset contains details about components replaced, repair activities performed etc. These events record degradation patterns. Absence of this crucial information in the training data can lead to misleading model results. Failure history can also be found within maintenance history as special error codes, or order dates for parts. Additional data sources that influence failure patterns should be investigated and provided by domain experts.

#### Machine operating conditions

Sensor based (or other) streaming data of the equipment in operation is an important data source. A key assumption in PdM is that a machine’s health status degrades over time during its routine operation. The data is expected to contain time-varying features that capture this aging pattern, and any anomalies that leads to degradation. The temporal aspect of the data is required for the algorithm to learn the failure and non-failure patterns over time. Based on these data points, the algorithm learns to predict how many more units of time a machine can continue to work before it fails.

#### Static feature data

Static features are metadata about the equipment. Examples are the equipment make, model, manufactured date, start date of service, location of the system, and other technical specifications.

Examples of relevant data for the [sample PdM use cases](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Sample-PdM-use-cases) are tabulated below:

| **Use Case** | **Examples of relevant data** |
| --- | --- |
| Flight delay and cancellations | Flight route information in the form of flight legs and page logs. Flight leg data includes routing details such as departure/arrival date, time, airport, layovers etc. Page log includes a series of error and maintenance codes recorded by the ground maintenance personnel. |
| Aircraft engine parts failure | Data collected from sensors in the aircraft that provide information on the condition of the various parts. Maintenance records help identify when component failures occurred and when they were replaced. |
| ATM Failure | Sensor readings for each transaction (depositing cash/check) and dispensing of cash. Information on gap measurement between notes, note thickness, note arrival distance, check attributes etc. Maintenance records that provide error codes, repair information, last time the cash dispenser was refilled. |
| Wind turbine failure | Sensors monitor turbine conditions such as temperature, wind direction, power generated, generator speed etc. Data is gathered from multiple wind turbines from wind farms located in various regions. Typically, each turbine will have multiple sensor readings relaying measurements at a fixed time interval. |
| Circuit breaker failures | Maintenance logs that include corrective, preventive, and systematic actions. Operational data that includes automatic and manual commands sent to circuit breakers such as for open and close actions. Device metadata such as date of manufacture, location, model, etc. Circuit breaker specifications such as voltage levels, geolocation, ambient conditions. |
| Elevator door failures | Elevator metadata such as type of elevator, manufactured date, maintenance frequency, building type, and so on. Operational information such as number of door cycles, average door close time. Failure history with causes. |
| Wheel failures | Sensor data that measures wheel acceleration, braking instances, driving distance, velocity etc. Static information on wheels like manufacturer, manufactured date. Failure data inferred from part order database that track order dates and quantities. |
| Subway train door failures | Door opening and closing times, other operational data such as current condition of train doors. Static data would include asset identifier, time, and condition value columns. |

### Data types

Given the above data sources, the two main data types observed in PdM domain are:

* Temporal data: Operational telemetry, machine conditions, work order types, priority codes that will have timestamps at the time of recording. Failure, maintenance/repair, and usage history will also have timestamps associated with each event.
* Static data: Machine features and operator features in general are static since they describe the technical specifications of machines or operator attributes. If these features could change over time, they should also have timestamps associated with them.

Predictor and target variables should be preprocessed/transformed into [numerical, categorical, and other data types](https://www.statsdirect.com/help/basics/measurement_scales.htm)depending on the algorithm being used.

### Data preprocessing

As a prerequisite to feature engineering, prepare the data from various streams to compose a schema from which it is easy to build features. Visualize the data first as a table of records. Each row in the table represents a training instance, and the columns represent predictor features (also called independent attributes or variables). Organize the data such that the last column(s) is the target (dependent variable). For each training instance, assign a label as the value of this column.

For temporal data, divide the duration of sensor data into time units. Each record should belong to a time unit for an asset, and should offer distinct information. Time units are defined based on business needs in multiples of seconds, minutes, hours, days, months, and so on. The time unit does not have to be the same as the frequency of data collection. If the frequency is high, the data may not show any significant difference from one unit to the other. For example, assume that ambient temperature was collected every 10 seconds. Using that same interval for training data only inflates the number of examples without providing any additional information. For this case, a better strategy would be to use average the data over 10 minutes, or an hour based on the business justification.

For static data,

* Maintenance records: Raw maintenance data has an asset identifier and timestamp with information on maintenance activities that have been performed at a given point in time. Transform maintenance activities into categorical columns, where each category descriptor uniquely maps to a specific maintenance action. The schema for maintenance records would include asset identifier, time, and maintenance action.
* Failure records: Failures or failure reasons can be recorded as specific error codes or failure events defined by specific business conditions. In cases where the equipment has multiple error codes, the domain expert should help identify the ones that are pertinent to the target variable. Use the remaining error codes or conditions to construct predictor features that correlate with these failures. The schema for failure records would include asset identifier, time, failure, or failure reason - if available.
* Machine and operator metadata: Merge the machine and operator data into one schema to associate an asset with its operator, along with their respective attributes. The schema for machine conditions would include asset identifier, asset features, operator identifier, and operator features.

Other data preprocessing steps include handling missing values and normalization of attribute values. A detailed discussion is beyond the scope of this guide - see the next section for some useful references.

With the above preprocessed data sources in place, the final transformation before feature engineering is to join the above tables based on the asset identifier and timestamp. The resulting table would have null values for the failure column when machine is in normal operation. These null values can be imputed by an indicator for normal operation. Use this failure column to create labels for the predictive model. For more information, see the section on [modeling techniques for predictive maintenance](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Modeling-techniques-for-predictive-maintenance).

## Feature engineering

Feature engineering is the first step prior to modeling the data. Its role in the data science process [is described here](https://docs.microsoft.com/azure/machine-learning/team-data-science-process/create-features). A featureis a predictive attribute for the model - such as temperature, pressure, vibration, and so on. For PdM, feature engineering involves abstracting a machine’s health over historical data collected over a sizable duration. In that sense, it is different from its peers such as remote monitoring, anomaly detection, and failure detection.

### Time windows

Remote monitoring entails reporting the events that happen as of points in time. Anomaly detection models evaluate (score) incoming streams of data to flag anomalies as of points in time. Failure detection classifies failures to be of specific types as they occur points in time. In contrast, PdM involves predicting failures over a future time period, based on features that represent machine behavior over historical time period. For PdM, feature data from individual points of time are too noisy to be predictive. So the data for each feature needs to be smoothened by aggregating data points over time windows.

### Lag features

The business requirements define how far the model has to predict into the future. In turn, this duration helps define 'how far back the model has to look' to make these predictions. This 'looking back' period is called the lag, and features engineered over this lag period are called lag features. This section discusses lag features that can be constructed from data sources with timestamps, and feature creation from static data sources. Lag features are typically numerical in nature.

Important

The window size is determined via experimentation, and should be finalized with the help of a domain expert. The same caveat holds for the selection and definition of lag features, their aggregations, and the type of windows.

#### Rolling aggregates

For each record of an asset, a rolling window of size "W" is chosen as the number of units of time to compute the aggregates. Lag features are then computed using the W periods before the date of that record. In Figure 1, the blue lines show sensor values recorded for an asset for each unit of time. They denote a rolling average of feature values over a window of size W=3. The rolling average is computed over all records with timestamps in the range t1 (in orange) to t2 (in green). The value for W is typically in minutes or hours depending on the nature of the data. But for certain problems, picking a large W (say 12 months) can provide the whole history of an asset until the time of the record.

 Figure 1. Rolling aggregate features

Examples of rolling aggregates over a time window are count, average, CUMESUM (cumulative sum) measures, min/max values. In addition, variance, standard deviation, and count of outliers beyond N standard deviations are often used. Examples of aggregates that may be applied for the [use cases](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Sample-PdM-use-cases) in this guide are listed below.

* Flight delay: count of error codes over the last day/week.
* Aircraft engine part failure: rolling means, standard deviation, and sum over the past day, week etc. This metric should be determined along with the business domain expert.
* ATM failures: rolling means, median, range, standard deviations, count of outliers beyond three standard deviations, upper and lower CUMESUM.
* Subway train door failures: Count of events over past day, week, two weeks etc.
* Circuit breaker failures: Failure counts over past week, year, three years etc.

Another useful technique in PdM is to capture trend changes, spikes, and level changes using algorithms that detect anomalies in data.

#### Tumbling aggregates

For each labeled record of an asset, a window of size W-k is defined, where k is the number of windows of size W. Aggregates are then created over k tumbling windows W-k, W-(k-1), …, W-2, W-1 for the periods before a record's timestamp. k can be a small number to capture short-term effects, or a large number to capture long-term degradation patterns. (see Figure 2).

 Figure 2. Tumbling aggregate features

For example, lag features for the wind turbines use case may be created with W=1 and k=3. They imply the lag for each of the past three months using top and bottom outliers.

### Static features

Technical specifications of the equipment such as date of manufacture, model number, location, are some examples of static features. They are treated as categorical variables for modeling. Some examples for the circuit breaker use case are voltage, current, power capacity, transformer type, and power source. For wheel failures, the type of tire wheels (alloy vs steel) is an example.

The data preparation efforts discussed so far should lead to the data being organized as shown below. Training, test, and validation data should have this logical schema (this example shows time in units of days).

| **Asset ID** | **Time** |  | **Label** |
| --- | --- | --- | --- |
| A123 | Day 1 | . . . | . |
| A123 | Day 2 | . . . | . |
| ... | ... | . . . | . |
| B234 | Day 1 | . . . | . |
| B234 | Day 2 | . . . | . |
| ... | ... | . . . | . |

The last step in feature engineering is the **labeling** of the target variable. This process is dependent on the modeling technique. In turn, the modeling technique depends on the business problem and nature of the available data. Labeling is discussed in the next section.

Important

Data preparation and feature engineering are as important as modeling techniques to arrive at successful PdM solutions. The domain expert and the practitioner should invest significant time in arriving at the right features and data for the model. A small sample from many books on feature engineering are listed below:

* Pyle, D. Data Preparation for Data Mining (The Morgan Kaufmann Series in Data Management Systems), 1999
* Zheng, A., Casari, A. Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists, O'Reilly, 2018.
* Dong, G. Liu, H. (Editors), Feature Engineering for Machine Learning and Data Analytics (Chapman & Hall/CRC Data Mining and Knowledge Discovery Series), CRC Press, 2018.

## Modeling techniques for predictive maintenance

This section discusses the main modeling techniques for PdM problems, along with their specific label construction methods. Notice that a single modeling technique can be used across different industries. The modeling technique is paired to the data science problem, rather than the context of the data at hand.

Important

The choice of labels for the failure cases and the labeling strategy  
should be determined in consultation with the domain expert.

### Binary classification

Binary classification is used to predict the probability that a piece of equipment fails within a future time period - called the future horizon period X. X is determined by the business problem and the data at hand, in consultation with the domain expert. Examples are:

* minimum lead time required to replace components, deploy maintenance resources, perform maintenance to avoid a problem that is likely to occur in that period.
* minimum count of events that can happen before a problem occurs.

In this technique, two types of training examples are identified. A positive example, which indicates a failure, with label = 1. A negative example, which indicates normal operations, with label = 0. The target variable, and hence the label values, are categorical. The model should identify each new example as likely to fail or work normally over the next X time units.

#### Label construction for binary classification

The question here is: "What is the probability that the asset will fail in the next X units of time?" To answer this question, label X records prior to the failure of an asset as "about to fail" (label = 1), and label all other records as being "normal" (label =0). (see Figure 3).

 Figure 3. Labeling for binary classification

Examples of labeling strategy for some of the use cases are listed below.

* Flight delays: X may be chosen as 1 day, to predict delays in the next 24 hours. Then all flights that are within 24 hours before failures are labeled as 1.
* ATM cash dispense failures: A goal may be to determine failure probability of a transaction in the next one hour. In that case, all transactions that happened within the past hour of the failure are labeled as 1. To predict failure probability over the next N currency notes dispensed, all notes dispensed within the last N notes of a failure are labeled as 1.
* Circuit breaker failures: The goal may be to predict the next circuit breaker command failure. In that case, X is chosen to be one future command.
* Train door failures: X may be chosen as two days.
* Wind turbine failures: X may be chosen as two months.

### Regression for predictive maintenance

Regression models are used to compute the remaining useful life (RUL) of an asset. RUL is defined as the amount of time that an asset is operational before the next failure occurs. Each training example is a record that belongs to a time unit nY for an asset, where n is the multiple. The model should calculate the RUL of each new example as a continuous number. This number denotes the period of time remaining before the failure.

#### Label construction for regression

The question here is: "What is the remaining useful life (RUL) of the equipment?" For each record prior to the failure, calculate the label to be the number of units of time remaining before the next failure. In this method, labels are continuous variables. (See Figure 4)

 Figure 4. Labeling for regression

For regression, labeling is done with reference to a failure point. Its calculation is not possible without knowing how long the asset has survived before a failure. So in contrast to binary classification, assets without any failures in the data cannot be used for modeling. This issue is best addressed by another statistical technique called [Survival Analysis](https://en.wikipedia.org/wiki/Survival_analysis). But potential complications may arise when applying this technique to PdM use cases that involve time-varying data with frequent intervals. For more information on Survival Analysis, see [this one-pager](https://www.cscu.cornell.edu/news/statnews/stnews78.pdf).

### Multi-class classification for predictive maintenance

Multi-class classification techniques can be used in PdM solutions for two scenarios:

* Predict two future outcomes: The first outcome is a range of time to failure for an asset. The asset is assigned to one of multiple possible periods of time. The second outcome is the likelihood of failure in a future period due to one of the multiple root causes. This prediction enables the maintenance crew to watch for symptoms and plan maintenance schedules.
* Predict the most likely root cause of a given failure. This outcome recommends the right set of maintenance actions to fix a failure. A ranked list of root causes and recommended repairs can help technicians prioritize their repair actions after a failure.

#### Label construction for multi-class classification

The question here is: "What is the probability that an asset will fail in the next nZ units of time where n is the number of periods?" To answer this question, label nZ records prior to the failure of an asset using buckets of time (3Z, 2Z, Z). Label all other records as "normal" (label = 0). In this method, the target variable holds categorical values. (See Figure 5).

 Figure 5. Labeling for multi-class classification for failure time prediction

The question here is: "What is the probability that the asset will fail in the next X units of time due to root cause/problem Pi?" where i is the number of possible root causes. To answer this question, label X records prior to the failure of an asset as "about to fail due to root cause Pi" (label = Pi). Label all other records as being "normal" (label = 0). In this method also, labels are categorical (See Figure 6).

 Figure 6. Labeling for multi-class classification for root cause prediction

The model assigns a failure probability due to each Pi as well as the probability of no failure. These probabilities can be ordered by magnitude to allow prediction of the problems that are most likely to occur in the future.

The question here is: "What maintenance actions do you recommend after a failure?" To answer this question, labeling does not require a future horizon to be picked, because the model is not predicting failure in the future. It is just predicting the most likely root cause once the failure has already happened.

## Training, validation, and testing methods for predictive maintenance

The [Team Data Science Process](https://docs.microsoft.com/azure/machine-learning/team-data-science-process/overview) provides a full coverage of the model train-test-validate cycle. This section discusses aspects unique to PdM.

### Cross validation

The goal of [cross validation](https://en.wikipedia.org/wiki/Cross-validation_(statistics)) is to define a data set to "test" the model in the training phase. This data set is called the validation set. This technique helps limit problems like overfitting and gives an insight on how the model will generalize to an independent data set. That is, an unknown data set, which could be from a real problem. The training and testing routine for PdM needs to take into account the time varying aspects to better generalize on unseen future data.

Many machine learning algorithms depend on a number of hyperparameters that can change the model performance significantly. The optimal values of these hyperparameters are not computed automatically when training the model. They should be specified by the data scientist. There are several ways of finding good values of hyperparameters.

The most common one is k-fold cross-validation that splits the examples randomly into k folds. For each set of hyperparameters values, run the learning algorithm k times. At each iteration, use the examples in the current fold as a validation set, and the rest of the examples as a training set. Train the algorithm over training examples and compute the performance metrics over validation examples. At the end of this loop, compute the average of k performance metrics. For each set of hyperparameter values, choose the ones that have the best average performance. The task of choosing hyperparameters is often experimental in nature.

In PdM problems, data is recorded as a time series of events that come from several data sources. These records may be ordered according to the time of labeling. Hence, if the dataset is split randomly into training and validation set, some of the training examples may be later in time than some of validation examples. Future performance of hyperparameter values will be estimated based on some data that arrived before model was trained. These estimations might be overly optimistic, especially if the time-series is not stationary and evolves over time. As a result, the chosen hyperparameter values might be suboptimal.

The recommended way is to split the examples into training and validation set in a time-dependent manner, where all validation examples are later in time than all training examples. For each set of hyperparameter values, train the algorithm over the training data set. Measure the model’s performance over the same validation set. Choose hyperparameter values that show the best performance. Hyperparameter values chosen by train/validation split result in better future model performance than with the values chosen randomly by cross-validation.

The final model can be generated by training a learning algorithm over entire training data using the best hyperparameter values.

### Testing for model performance

Once a model is built, an estimate of its future performance on new data is required. A good estimate is the performance metric of hyperparameter values computed over the validation set, or an average performance metric computed from cross-validation. These estimations are often overly optimistic. The business might often have some additional guidelines on how they would like to test the model.

The recommended way for PdM is to split the examples into training, validation, and test data sets in a time-dependentmanner. All test examples should be later in time than all the training and validation examples. After the split, generate the model and measure its performance as described earlier.

When time-series are stationary and easy to predict, both random and time-dependent approaches generate similar estimations of future performance. But when time-series are non-stationary, and/or hard to predict, the time-dependent approach will generate more realistic estimates of future performance.

### Time-dependent split

This section describes best practices to implement time-dependent split. A time-dependent two-way split between training and test sets is described below.

Assume a stream of timestamped events such as measurements from various sensors. Define features and labels of training and test examples over time frames that contain multiple events. For example, for binary classification, create features based on past events, and create labels based on future events within "X" units of time in the future (see the sections on [feature engineering](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Feature-engineering) and [modeling techniques](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Modeling-techniques-applied-to-PdM-use-cases)). Thus, the labeling time frame of an example comes later than the time frame of its features.

For time-dependent split, pick a training cutoff time Tc at which to train a model, with hyperparameters tuned using historical data up to Tc. To prevent leakage of future labels that are beyond Tc into the training data, choose the latest time to label training examples to be X units before Tc. In the example shown in Figure 7, each square represents a record in the data set where features and labels are computed as described above. The figure shows the records that should go into training and testing sets for X=2 and W=3:

 Figure 7. Time-dependent split for binary classification

The green squares represent records belonging to the time units that can be used for training. Each training example is generated by considering the past three periods for feature generation, and two future periods for labeling before Tc. When any part of the two future periods is beyond Tc, exclude that example from the training data set because no visibility is assumed beyond Tc.

The black squares represent the records of the final labeled data set that should not be used in the training data set, given the above constraint. These records will also not be used in testing data, since they are before Tc. In addition, their labeling time frames partially depend on the training time frame, which is not ideal. Training and test data should have separate labeling time frames to prevent label information leakage.

The technique discussed so far allows for overlap between training and testing examples that have timestamps near Tc. A solution to achieve greater separation is to exclude examples that are within W time units of Tc from the test set. But such an aggressive split depends on ample data availability.

Regression models used for predicting RUL are more severely affected by the leakage problem. Using the random split method leads to extreme over-fitting. For regression problems, the split should be such that the records belonging to assets with failures before Tc go into the training set. Records of assets that have failures after the cutoff go into the test set.

Another best practice for splitting data for training and testing is to use a split by asset ID. The split should be such that none of the assets used in the training set are used in testing the model performance. Using this approach, a model has a better chance of providing more realistic results with new assets.

### Handling imbalanced data

In classification problems, if there are more examples of one class than of the others, the data set is said to be imbalanced. Ideally, enough representatives of each class in the training data are preferred to enable differentiation between different classes. If one class is less than 10% of the data, the data is deemed to be imbalanced. The underrepresented class is called a minority class.

Many PdM problems face such imbalanced datasets, where one class is severely underrepresented compared to the other class, or classes. In some situations, the minority class may constitute only 0.001% of the total data points. Class imbalance is not unique to PdM. Other domains where failures and anomalies are rare occurrences face a similar problem, for examples, fraud detection and network intrusion. These failures make up the minority class examples.

With class imbalance in data, performance of most standard learning algorithms is compromised, since they aim to minimize the overall error rate. For a data set with 99% negative and 1% positive examples, a model can be shown to have 99% accuracy by labeling all instances as negative. But the model will mis-classify all positive examples; so even if its accuracy is high, the algorithm is not a useful one. Consequently, conventional evaluation metrics such as overall accuracy on error rateare insufficient for imbalanced learning. When faced with imbalanced datasets, other metrics are used for model evaluation:

* Precision
* Recall
* F1 scores
* Cost adjusted ROC (receiver operating characteristics)

For more information about these metrics, see [model evaluation](https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/cortana-analytics-playbook-predictive-maintenance#Model-evaluation).

However, there are some methods that help remedy class imbalance problem. The two major ones are sampling techniquesand cost sensitive learning.

#### Sampling methods

Imbalanced learning involves the use of sampling methods to modify the training data set to a balanced data set. Sampling methods are not to be applied to the test set. Although there are several sampling techniques, most straight forward ones are random oversampling and under sampling.

Random oversampling involves selecting a random sample from minority class, replicating these examples, and adding them to training data set. Consequently, the number of examples in minority class is increased, and eventually balance the number of examples of different classes. A drawback of oversampling is that multiple instances of certain examples can cause the classifier to become too specific, leading to over-fitting. The model may show high training accuracy, but its performance on unseen test data may be suboptimal.

Conversely, random under sampling is selecting a random sample from a majority class and removing those examples from training data set. However, removing examples from majority class may cause the classifier to miss important concepts pertaining to the majority class. Hybrid sampling where minority class is over-sampled and majority class is under-sampled at the same time is another viable approach.

There are many sophisticated sampling techniques. The technique chosen depends on the data properties and results of iterative experiments by the data scientist.

#### Cost sensitive learning

In PdM, failures that constitute the minority class are of more interest than normal examples. So the focus is mainly on the algorithm's performance on failures. Incorrectly predicting a positive class as a negative class can cost more than vice-versa. This situation is commonly referred as unequal loss or asymmetric cost of mis-classifying elements to different classes. The ideal classifier should deliver high prediction accuracy over the minority class, without compromising on the accuracy for the majority class.

There are multiple ways to achieve this balance. To mitigate the problem of unequal loss, assign a high cost to mis-classification of the minority class, and try to minimize the overall cost. Algorithms like SVMs (Support Vector Machines) adopt this method inherently, by allowing cost of positive and negative examples to be specified during training. Similarly, boosting methods such as boosted decision trees usually show good performance with imbalanced data.

## Model evaluation

Mis-classification is a significant problem for PdM scenarios where the cost of false alarms to the business is high. For instance, a decision to ground an aircraft based on an incorrect prediction of engine failure can disrupt schedules and travel plans. Taking a machine offline from an assembly line can lead to loss of revenue. So model evaluation with the right performance metrics against new test data is critical.

Typical performance metrics used to evaluate PdM models are discussed below:

* [Accuracy](https://en.wikipedia.org/wiki/Accuracy_and_precision) is the most popular metric used for describing a classifier’s performance. But accuracy is sensitive to data distributions, and is an ineffective measure for scenarios with imbalanced data sets. Other metrics are used instead. Tools like [confusion matrix](https://en.wikipedia.org/wiki/Confusion_matrix) are used to compute and reason about accuracy of the model.
* [Precision](https://en.wikipedia.org/wiki/Precision_and_recall) of PdM models relate to the rate of false alarms. Lower precision of the model generally corresponds to a higher rate of false alarms.
* [Recall](https://en.wikipedia.org/wiki/Precision_and_recall) rate denotes how many of the failures in the test set were correctly identified by the model. Higher recall rates mean the model is successful in identifying the true failures.
* [F1 score](https://en.wikipedia.org/wiki/F1_score) is the harmonic average of precision and recall, with its value ranging between 0 (worst) to 1 (best).

For binary classification,

* [Receiver operating curves (ROC)](https://en.wikipedia.org/wiki/Receiver_operating_characteristic) is also a popular metric. In ROC curves, model performance is interpreted based on one fixed operating point on the ROC.
* But for PdM problems, decile tables and lift charts are more informative. They focus only on the positive class (failures), and provide a more complex picture of the algorithm performance than ROC curves.
  + Decile tables are created using test examples in a descending order of failure probabilities. The ordered samples are then grouped into deciles (10% of the samples with highest probability, then 20%, 30%, and so on). The ratio (true positive rate)/(random baseline) for each decile helps estimate the algorithm performance at each decile. The random baseline takes on values 0.1, 0.2, and so on.
  + [Lift charts](http://www2.cs.uregina.ca/~dbd/cs831/notes/lift_chart/lift_chart.html) plot the decile true positive rate versus random true positive rate for all deciles. The first deciles are usually the focus of results, since they show the largest gains. First deciles can also be seen as representative for "at risk", when used for PdM.

## Model operationalization for predictive maintenance

The benefit the data science exercise is realized only when the trained model is made operational. That is, the model must be deployed into the business systems to make predictions based on new, previously unseen, data. The new data must exactly conform to the model signature of the trained model in two ways:

* all the features must be present in every logical instance (say a row in a table) of the new data.
* the new data must be pre-processed, and each of the features engineered, in exactly the same way as the training data.

The above process is stated in many ways in academic and industry literature. But all the following statements mean the same thing:

* Score new data using the model
* Apply the model to new data
* Operationalize the model
* Deploy the model
* Run the model against new data

As stated earlier, model operationalization for PdM is different from its peers. Scenarios involving anomaly detection and failure detection typically implement online scoring (also called real time scoring). Here, the model scores each incoming record, and returns a prediction. For anomaly detection, the prediction is an indication that an anomaly occurred (Example: One-class SVM). For failure detection, it would be the type or class of failure.

In contrast, PdM involves batch scoring. To conform to the model signature, the features in the new data must be engineered in the same manner as the training data. For the large datasets that is typical for new data, features are aggregated over time windows and scored in batch. Batch scoring is typically done in distributed systems like [Spark](http://spark.apache.org/) or [Azure Batch](https://docs.microsoft.com/azure/batch/batch-api-basics). There are a couple of alternatives - both suboptimal:

* Streaming data engines support aggregation over windows in memory. So it could be argued that they support online scoring. But these systems are suitable for dense data in narrow windows of time, or sparse elements over wider windows. They may not scale well for the dense data over wider time windows, as seen in PdM scenarios.
* If batch scoring is not available, the solution is to adapt online scoring to handle new data in small batches at a time.

## Problem Description

A major problem faced by businesses in asset-heavy industries such as manufacturing is the significant costs that are associated with delays in the production process due to mechanical problems. Most of these businesses are interested in predicting these problems in advance so that they can proactively prevent the problems before they occur which will reduce the costly impact caused by downtime. Please refer to the[playbook for predictive maintenance](https://azure.microsoft.com/en-us/documentation/articles/cortana-analytics-playbook-predictive-maintenance/) for a detailed explanation of common use cases in predictive maintenance and modelling approaches.

In this notebook, we follow the ideas from the playbook referenced above and aim to provide the steps of implementing a predictive model for a scenario which is based on a synthesis of multiple real-world business problems. This example brings together common data elements observed among many predictive maintenance use cases and the data itself is created by data simulation methods.

The business problem for this example is about predicting problems caused by component failures such that the question “What is the probability that a machine will fail in the near future due to a failure of a certain component” can be answered. The problem is formatted as a multi-class classification problem and a machine learning algorithm is used to create the predictive model that learns from historical data collected from machines. In the following sections, we go through the steps of implementing such a model which are feature engineering, label construction, training and evaluation. First, we start by explaining the data sources in the next section.

## Data Sources

Common data sources for predictive maintenance problems are

* Failure history: The failure history of a machine or component within the machine.
* Maintenance history: The repair history of a machine, e.g. error codes, previous maintenance activities or component replacements.
* Machine conditions and usage: The operating conditions of a machine e.g. data collected from sensors.
* Machine features: The features of a machine, e.g. engine size, make and model, location.
* Operator features: The features of the operator, e.g. gender, past experience

The data for this example comes from 4 different sources which are real-time telemetry data collected from machines, error messages, historical maintenance records that include failures and machine information such as type and age.

### Telemetry

The first data source is the telemetry time-series data which consists of voltage, rotation, pressure and vibration measurements collected from 100 machines in real time averaged over every hour collected during the year 2015. Below, we provide first 10 records of the first machine with machineID=1. A summary of the whole dataset is also provided.

In [1]:

*# Environment Setup*

library("AzureML") *# Connect to Azure Machine Learning*

library("dplyr") *# Data munging functions*

library("zoo") *# Feature engineering rolling aggregates*

install.packages("data.table")

library("data.table") *# Feature engineering*

library("ggplot2") *# Graphics*

library("scales") *# For time formatted axis*

*# connect to the workspace*

ws <- workspace()

Attaching package: &apos;dplyr&apos;

The following object is masked from &apos;package:stats&apos;:

filter

The following objects are masked from &apos;package:base&apos;:

intersect, setdiff, setequal, union

Attaching package: &apos;zoo&apos;

The following objects are masked from &apos;package:base&apos;:

as.Date, as.Date.numeric

Installing package into &apos;/home/nbcommon/R&apos;

(as &apos;lib&apos; is unspecified)

The downloaded source packages are in

&apos;/tmp/RtmpzbolXg/downloaded\_packages&apos;

Attaching package: &apos;data.table&apos;

The following objects are masked from &apos;package:dplyr&apos;:

between, last

In [2]:

*# download telemetry dataset*

telemetry <- download.datasets(ws, name = "telemetry")

*# format datetime field which comes in as.character*

telemetry$datetime <- as.POSIXct(telemetry$datetime,

format="%m/%d/%Y %I:%M:%S %p",

tz="UTC")

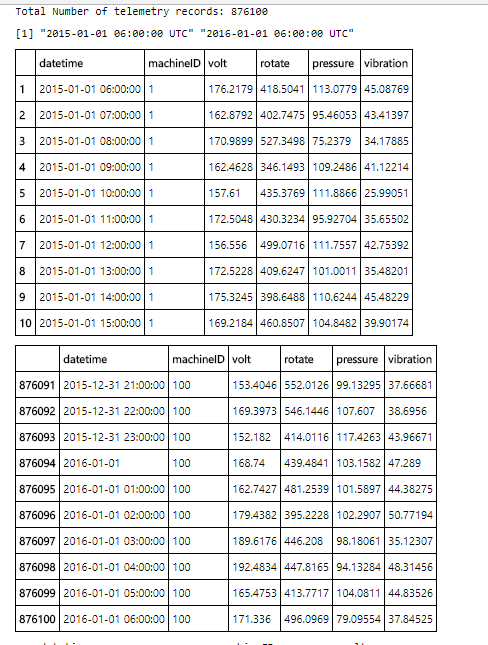
cat("Total Number of telemetry records:", nrow(telemetry))

range(telemetry$datetime)

head(telemetry,10)

tail(telemetry,10)

summary(telemetry)



datetime machineID volt

Min. :2015-01-01 06:00:00 Min. : 1.00 Min. : 97.33

1st Qu.:2015-04-02 12:00:00 1st Qu.: 25.75 1st Qu.:160.30

Median :2015-07-02 18:00:00 Median : 50.50 Median :170.61

Mean :2015-07-02 18:00:00 Mean : 50.50 Mean :170.78

3rd Qu.:2015-10-02 00:00:00 3rd Qu.: 75.25 3rd Qu.:181.00

Max. :2016-01-01 06:00:00 Max. :100.00 Max. :255.12

rotate pressure vibration

Min. :138.4 Min. : 51.24 Min. :14.88

1st Qu.:412.3 1st Qu.: 93.50 1st Qu.:36.78

Median :447.6 Median :100.43 Median :40.24

Mean :446.6 Mean :100.86 Mean :40.39

3rd Qu.:482.2 3rd Qu.:107.56 3rd Qu.:43.78

Max. :695.0 Max. :185.95 Max. :76.79

As an example, below is a plot of voltage values for two machineIDs for January 2015.

In [3]:

theme\_set(theme\_bw()) *# theme for figures*

options(repr.plot.width = 8, repr.plot.height = 6)

ggplot(data = telemetry %>% filter(machineID %in% 1:2,

datetime > as.POSIXct("2015-01-01"),

datetime < as.POSIXct("2015-02-01")),

aes(x = datetime, y = volt, col = factor(machineID))) +

geom\_line(alpha = 0.5) +

labs(y = "voltage", color = "machineID") +

facet\_wrap(~machineID, ncol=1)

### Errors

The second major data source is the error logs. These are non-breaking errors thrown while the machine is still operational and do not constitute as failures. The error date and times are rounded to the closest hour since the telemetry data is collected at an hourly rate.

In [4]:

*# download errors dataset*

errors <- download.datasets(ws, name = "errors")

*# format datetime and errorID fields*

errors$datetime <- as.POSIXct(errors$datetime,

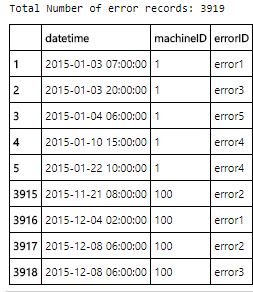
format="%m/%d/%Y %I:%M:%S %p",

tz="UTC")

errors$errorID <- as.factor(errors$errorID)

cat("Total Number of error records:",nrow(errors))

errors[c(1:5, nrow(errors)-4:1),]



In [5]:

options(repr.plot.width = 5, repr.plot.height = 3)

ggplot(errors, aes(x = errorID)) +

geom\_histogram(fill = "orange") +

labs(title = "Errors by type", x = "error types")

In [6]:

options(repr.plot.width = 6, repr.plot.height = 5)

ggplot(errors %>% filter(machineID < 4),

aes(x = errorID, fill = factor(machineID))) +

geom\_histogram(color = "black") +

labs(title = "MachineID errors by type", x = "error types", fill="MachineID")+

facet\_wrap(~machineID, ncol = 1)

In [7]:

options(repr.plot.width = 7, repr.plot.height = 5)

ggplot(errors %>% filter(machineID == 4),

aes(y = errorID, x = datetime)) +

geom\_point(color = "black", alpha = 0.5) +

labs(title = "MachineID 4 errors", x = "Date")

### Maintenance

These are the scheduled and unscheduled maintenance records which correspond to both regular inspection of components as well as failures. A record is generated if a component is replaced during the scheduled inspection or replaced due to a break down. The records that are created due to break downs will be called failures which is explained in the later sections. Maintenance data has both 2014 and 2015 records.

In [8]:

*# download maintenance dataset*

maint <- download.datasets(ws, name = "maint")

*# format datetime and comp fields*

maint$datetime <- as.POSIXct(maint$datetime,

format="%m/%d/%Y %I:%M:%S %p",

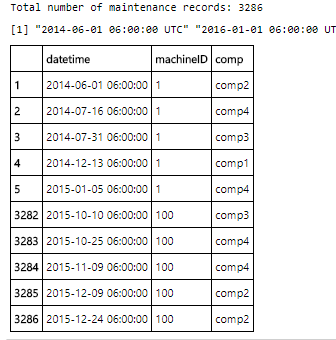
tz="UTC")

maint$comp <- as.factor(maint$comp)

cat("Total number of maintenance records:", nrow(maint))

range(maint$datetime)

maint[c(1:5, nrow(maint)-4:0),]



In [9]:

options(repr.plot.width = 5, repr.plot.height = 3)

ggplot(maint, aes(x = comp)) +

geom\_histogram(fill= "magenta") +

labs(title = "Component replacements", x = "component types")

In [10]:

options(repr.plot.width = 6, repr.plot.height = 8)

ggplot(maint %>% filter(machineID < 4),

aes(x = comp, fill = factor(machineID))) +

geom\_histogram(color = "black") +

labs(title = "Component replacements", x = "component types", fill = "Machine ID")+

facet\_wrap(~machineID, ncol = 1)

In [11]:

options(repr.plot.width = 7, repr.plot.height = 5)

ggplot(maint %>% filter(machineID == 4),

aes(y = comp, x = datetime)) +

geom\_point(color = "black", alpha = 0.5) +

labs(title = "MachineID 4 component replacements", x = "Date")

### Machines

This data set includes some information about the machines which are model type and age which is years in service.

In [12]:

*# download machines dataset*

machines <- download.datasets(ws, name = "machines")

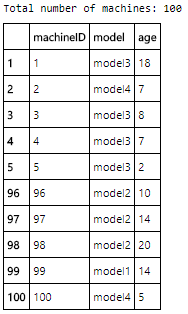
*# format model field*

machines$model <- as.factor(machines$model)

cat("Total number of machines:", nrow(machines))

machines[c(1:5, nrow(machines)-4:0),]

summary(machines)



machineID model age

Min. : 1.00 model1:16 Min. : 0.00

1st Qu.: 25.75 model2:17 1st Qu.: 6.75

Median : 50.50 model3:35 Median :12.00

Mean : 50.50 model4:32 Mean :11.33

3rd Qu.: 75.25 3rd Qu.:16.00

Max. :100.00 Max. :20.00

In [13]:

options(repr.plot.width = 8, repr.plot.height = 6)

ggplot(machines, aes(x = age, fill = model)) +

geom\_histogram(color = "black") +

labs(title = "Machines", x = "age (years)") +

facet\_wrap(~model)

### Failures

These are the records of component replacements due to failures. Each record has a date and time, machine ID and failed component type.

In [14]:

*# download failures dataset*

failures <- download.datasets(ws, name = "failures")

*# format datetime and failure fields*

failures$datetime <- as.POSIXct(failures$datetime,

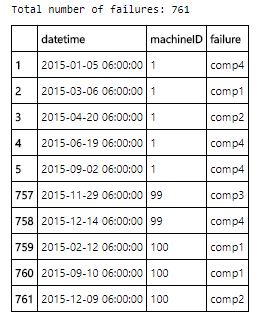
format="%m/%d/%Y %I:%M:%S %p",

tz="UTC")

failures$failure <- as.factor(failures$failure)

cat("Total number of failures:", nrow(failures))

failures[c(1:5, nrow(failures)-4:0),]



Below is the distribution of the failures due to each component. We see that the most failures happen due to component 2.

In [15]:

options(repr.plot.width = 5, repr.plot.height = 3)

ggplot(failures, aes(x = failure)) +

geom\_histogram(fill = "red") +

labs(title = "Failure distribution", x = "component type")

In [16]:

options(repr.plot.width = 6, repr.plot.height = 6)

ggplot(failures %>% filter(machineID < 4),

aes(x = failure, fill = factor(machineID))) +

geom\_histogram(color = "black") +

labs(title = "Failure distribution", x = "component type", fill = "MachineID") +

facet\_wrap(~machineID, ncol=1)

[Back to top](https://gallery.azure.ai/Notebook/Predictive-Maintenance-Modelling-Guide-R-Notebook-1#x_Modelling-Guide-for-Predictive-Maintenance)

## Feature Engineering

The first step in predictive maintenance applications is feature engineering which requires bringing the different data sources together to create features that best describe a machines's health condition at a given point in time. In the next sections, different type of feature engineering methods are used to create features based on the properties of each data source.

### Lag Features from Telemetry

Telemetry data almost always comes with time-stamps which makes it suitable for calculating lagging features. A common method is to pick a window size for the lag features to be created and compute rolling aggregate measures such as mean, standard deviation, minimum, maximum, etc. to represent the short term history of the telemetry over the lag window. In the following, rolling mean and standard deviation of the telemetry data over the last 3 hour lag window is calculated for every 3 hours.

In [17]:

*# calculate the rolling mean and rolling standard deviation*

*# on the last 3 hour lag window (width=3), for every 3 hours (by=3)*

*# for each machine ID.*

telemetrymean <- telemetry %>%

arrange(machineID, datetime) %>%

group\_by(machineID) %>%

mutate(voltmean = rollapply(volt, width = 3, FUN = mean, align = "right", fill = NA, by = 3),

rotatemean = rollapply(rotate, width = 3, FUN = mean, align = "right", fill = NA, by = 3),

pressuremean = rollapply(pressure, width = 3, FUN = mean, align = "right", fill = NA, by = 3),

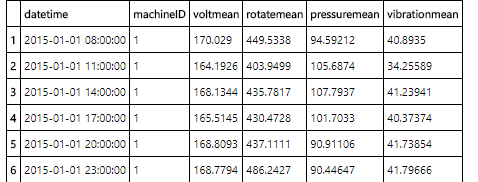
vibrationmean = rollapply(vibration, width = 3, FUN = mean, align = "right", fill = NA, by = 3)) %>%

select(datetime, machineID, voltmean, rotatemean, pressuremean, vibrationmean) %>%

filter(!is.na(voltmean))%>%

ungroup()

head(telemetrymean)



In [18]:

telemetrysd <- telemetry %>%

arrange(machineID, datetime) %>%

group\_by(machineID) %>%

mutate(voltsd = rollapply(volt, width = 3, FUN = sd, align = "right", fill = NA, by = 3),

rotatesd = rollapply(rotate, width = 3, FUN = sd, align = "right", fill = NA, by = 3),

pressuresd = rollapply(pressure, width = 3, FUN = sd, align = "right", fill = NA, by = 3),

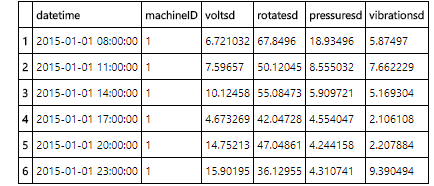
vibrationsd = rollapply(vibration, width = 3, FUN = sd, align = "right", fill = NA, by = 3)) %>%

select(datetime, machineID, voltsd, rotatesd, pressuresd, vibrationsd) %>%

filter(!is.na(voltsd)) %>%

ungroup()

head(telemetrysd)



For caputing a longer term effect, 24 hour lag features are also calculated as below.

In [19]:

*# calculate the rolling mean and rolling standard deviation*

*# on the last 24 hour lag window (width=24), for every 3 hours (by=3)*

*# for each machine ID.*

telemetrymean\_24hrs <- telemetry %>%

arrange(machineID, datetime) %>%

group\_by(machineID) %>%

mutate(voltmean\_24hrs = rollapply(volt, width = 24, FUN = mean, align = "right", fill = NA, by = 3),

rotatemean\_24hrs = rollapply(rotate, width = 24, FUN = mean, align = "right", fill = NA, by = 3),

pressuremean\_24hrs = rollapply(pressure, width = 24, FUN = mean, align = "right", fill = NA, by = 3),

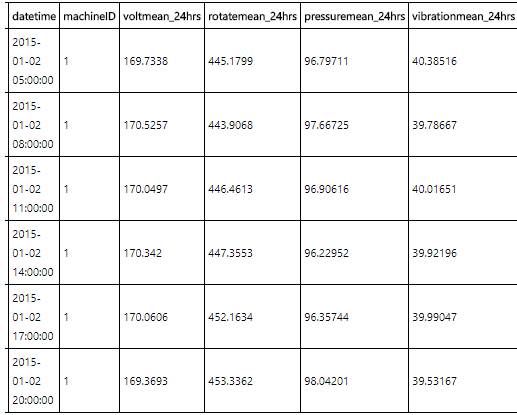
vibrationmean\_24hrs = rollapply(vibration, width = 24, FUN = mean, align = "right", fill = NA, by = 3)) %>%

select(datetime, machineID, voltmean\_24hrs, rotatemean\_24hrs, pressuremean\_24hrs, vibrationmean\_24hrs) %>%

filter(!is.na(voltmean\_24hrs)) %>%

ungroup()

head(telemetrymean\_24hrs)



In [20]:

telemetrysd\_24hrs <- telemetry %>%

arrange(machineID, datetime) %>%

group\_by(machineID) %>%

mutate(voltsd\_24hrs = rollapply(volt, width = 24, FUN = sd, align = "right", fill = NA, by = 3),

rotatesd\_24hrs = rollapply(rotate, width = 24, FUN = sd, align = "right", fill = NA, by = 3),

pressuresd\_24hrs = rollapply(pressure, width = 24, FUN = sd, align = "right", fill = NA, by = 3),

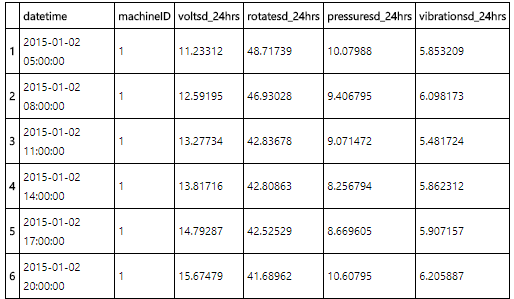
vibrationsd\_24hrs = rollapply(vibration, width = 24, FUN = sd, align = "right", fill = NA, by = 3)) %>%

select(datetime, machineID, voltsd\_24hrs, rotatesd\_24hrs, pressuresd\_24hrs, vibrationsd\_24hrs) %>%

filter(!is.na(voltsd\_24hrs)) %>%

ungroup()

head(telemetrysd\_24hrs)



Next, the columns of the feature datasets created earlier are merged to create the final feature set from telemetry.

In [21]:

*# merge columns of feature sets created earlier*

telemetryfeat <- data.frame(telemetrymean, telemetrysd[,-c(1:2)])

telemetryfeat\_24hrs <- data.frame(telemetrymean\_24hrs, telemetrysd\_24hrs[,-c(1:2)])

telemetryfeat <- telemetryfeat %>%

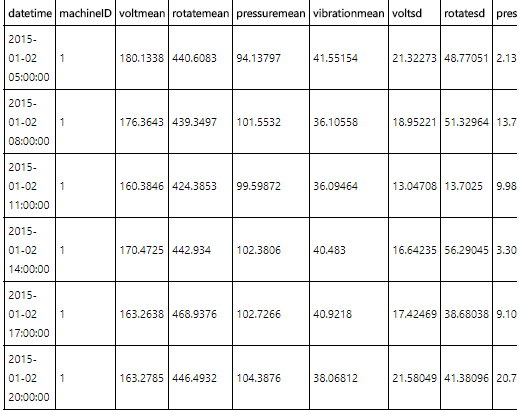
left\_join(telemetryfeat\_24hrs, by = c("datetime", "machineID")) %>%

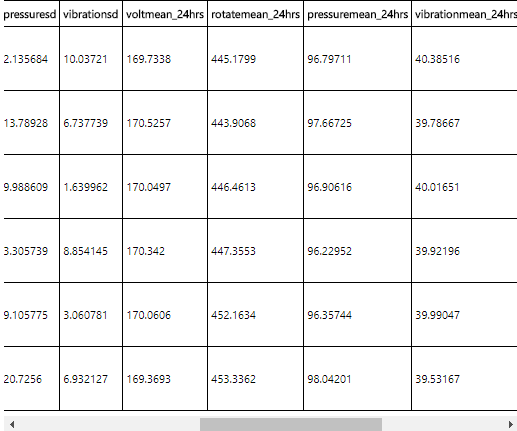
filter(!is.na(voltmean\_24hrs)) %>%

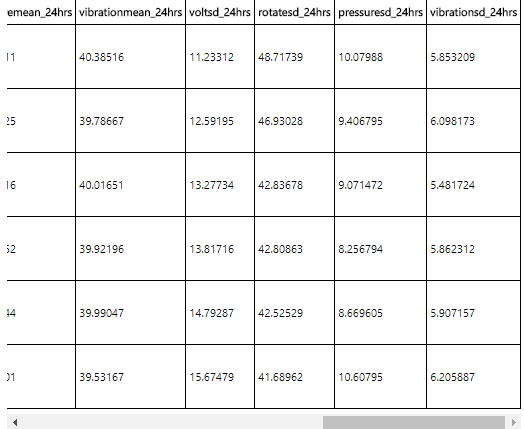
ungroup()

head(telemetryfeat)

summary(telemetryfeat)







datetime machineID voltmean rotatemean

Min. :2015-01-02 05:00:00 Min. : 1.00 Min. :125.5 Min. :211.8

1st Qu.:2015-04-03 05:00:00 1st Qu.: 25.75 1st Qu.:164.4 1st Qu.:427.6

Median :2015-07-03 05:00:00 Median : 50.50 Median :170.4 Median :448.4

Mean :2015-07-03 05:00:00 Mean : 50.50 Mean :170.8 Mean :446.6

3rd Qu.:2015-10-02 05:00:00 3rd Qu.: 75.25 3rd Qu.:176.6 3rd Qu.:468.4

Max. :2016-01-01 05:00:00 Max. :100.00 Max. :241.4 Max. :586.7

pressuremean vibrationmean voltsd rotatesd

Min. : 72.12 Min. :26.57 Min. : 0.02551 Min. : 0.07899

1st Qu.: 96.24 1st Qu.:38.15 1st Qu.: 8.02746 1st Qu.: 26.90357

Median :100.23 Median :40.15 Median :12.49560 Median : 41.79570

Mean :100.86 Mean :40.38 Mean :13.29986 Mean : 44.45634

3rd Qu.:104.40 3rd Qu.:42.23 3rd Qu.:17.68912 3rd Qu.: 59.10372

Max. :162.31 Max. :69.31 Max. :58.44433 Max. :179.90304

pressuresd vibrationsd voltmean\_24hrs rotatemean\_24hrs

Min. : 0.02742 Min. : 0.01528 Min. :156.3 Min. :267.0

1st Qu.: 5.37065 1st Qu.: 2.68431 1st Qu.:168.1 1st Qu.:441.6

Median : 8.34578 Median : 4.17385 Median :170.2 Median :449.2

Mean : 8.88582 Mean : 4.44066 Mean :170.8 Mean :446.6

3rd Qu.:11.78964 3rd Qu.: 5.89900 3rd Qu.:172.5 3rd Qu.:456.4

Max. :35.65937 Max. :18.30560 Max. :220.6 Max. :499.3

pressuremean\_24hrs vibrationmean\_24hrs voltsd\_24hrs rotatesd\_24hrs

Min. : 90.35 Min. :35.25 Min. : 6.503 Min. : 19.84

1st Qu.: 98.67 1st Qu.:39.36 1st Qu.:13.358 1st Qu.: 44.67

Median :100.10 Median :40.07 Median :14.856 Median : 49.61

Mean :100.85 Mean :40.38 Mean :14.919 Mean : 49.95

3rd Qu.:101.61 3rd Qu.:40.83 3rd Qu.:16.396 3rd Qu.: 54.80

Max. :152.66 Max. :61.85 Max. :27.914 Max. :105.33

pressuresd\_24hrs vibrationsd\_24hrs

Min. : 4.433 Min. : 2.108

1st Qu.: 8.925 1st Qu.: 4.461

Median : 9.922 Median : 4.958

Mean :10.047 Mean : 5.002

3rd Qu.:10.981 3rd Qu.: 5.484

Max. :28.868 Max. :12.609

### Lag Features from Errors

Similar to telemetry, errors also come with time-stamps. However, unlike telemetry that had numerical values, errors have categorical values denoting the type of error that occured at a time-stamp. In this case, aggregating methods such as averaging does not apply. Counting the different categories is a more viable approach where lagging counts of different types of errors that occured in the lag window are calculated. Below we create such lagging counts from the errors received.

In [22]:

*# create a column for each error type*

errorcount <- errors %>% select(datetime, machineID, errorID) %>%

mutate(error1 = as.integer(errorID == "error1"),

error2 = as.integer(errorID == "error2"),

error3 = as.integer(errorID == "error3"),

error4 = as.integer(errorID == "error4"),

error5 = as.integer(errorID == "error5"))

*# sum the duplicate errors in an hour*

errorcount <- errorcount %>%

group\_by(machineID,datetime)%>%

summarise(error1sum = sum(error1),

error2sum = sum(error2),

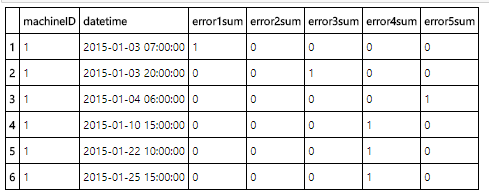
error3sum = sum(error3),

error4sum = sum(error4),

error5sum = sum(error5)) %>%

ungroup()

head(errorcount)



In [23]:

*# align errors with telemetry datetime field*

errorfeat <- telemetry %>%

select(datetime, machineID) %>%

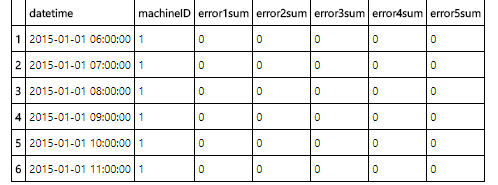
left\_join(errorcount, by = c("datetime", "machineID"))

*# replace missing values*

errorfeat[is.na(errorfeat)] <- 0

head(errorfeat)

summary(errorfeat)



datetime machineID error1sum

Min. :2015-01-01 06:00:00 Min. : 1.00 Min. :0.000000

1st Qu.:2015-04-02 12:00:00 1st Qu.: 25.75 1st Qu.:0.000000

Median :2015-07-02 18:00:00 Median : 50.50 Median :0.000000

Mean :2015-07-02 18:00:00 Mean : 50.50 Mean :0.001153

3rd Qu.:2015-10-02 00:00:00 3rd Qu.: 75.25 3rd Qu.:0.000000

Max. :2016-01-01 06:00:00 Max. :100.00 Max. :1.000000

error2sum error3sum error4sum error5sum

Min. :0.000000 Min. :0.0000000 Min. :0.0000000 Min. :0.0000000

1st Qu.:0.000000 1st Qu.:0.0000000 1st Qu.:0.0000000 1st Qu.:0.0000000

Median :0.000000 Median :0.0000000 Median :0.0000000 Median :0.0000000

Mean :0.001128 Mean :0.0009565 Mean :0.0008298 Mean :0.0004063

3rd Qu.:0.000000 3rd Qu.:0.0000000 3rd Qu.:0.0000000 3rd Qu.:0.0000000

Max. :1.000000 Max. :1.0000000 Max. :1.0000000 Max. :1.0000000

In [24]:

*# count the number of errors of different types in the last 24 hours, for every 3 hours*

errorfeat <- errorfeat %>%

arrange(machineID, datetime) %>%

group\_by(machineID) %>%

mutate(error1count = rollapply(error1sum, width = 24, FUN = sum, align = "right", fill = NA, by = 3),

error2count = rollapply(error2sum, width = 24, FUN = sum, align = "right", fill = NA, by = 3),

error3count = rollapply(error3sum, width = 24, FUN = sum, align = "right", fill = NA, by = 3),

error4count = rollapply(error4sum, width = 24, FUN = sum, align = "right", fill = NA, by = 3),

error5count = rollapply(error5sum, width = 24, FUN = sum, align = "right", fill = NA, by = 3)) %>%

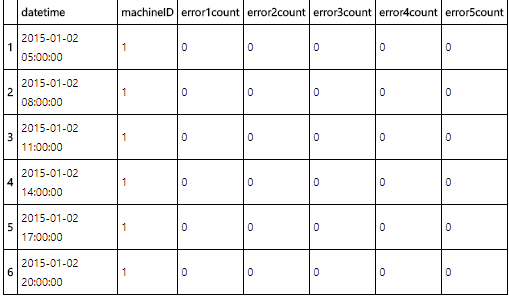
select(datetime, machineID, error1count, error2count, error3count, error4count, error5count) %>%

filter(!is.na(error1count)) %>%

ungroup()

head(errorfeat)

summary(errorfeat)



datetime machineID error1count

Min. :2015-01-02 05:00:00 Min. : 1.00 Min. :0.00000

1st Qu.:2015-04-03 05:00:00 1st Qu.: 25.75 1st Qu.:0.00000

Median :2015-07-03 05:00:00 Median : 50.50 Median :0.00000

Mean :2015-07-03 05:00:00 Mean : 50.50 Mean :0.02765

3rd Qu.:2015-10-02 05:00:00 3rd Qu.: 75.25 3rd Qu.:0.00000

Max. :2016-01-01 05:00:00 Max. :100.00 Max. :2.00000

error2count error3count error4count error5count

Min. :0.00000 Min. :0.00000 Min. :0.0000 Min. :0.000000

1st Qu.:0.00000 1st Qu.:0.00000 1st Qu.:0.0000 1st Qu.:0.000000

Median :0.00000 Median :0.00000 Median :0.0000 Median :0.000000

Mean :0.02707 Mean :0.02291 Mean :0.0199 Mean :0.009753

3rd Qu.:0.00000 3rd Qu.:0.00000 3rd Qu.:0.0000 3rd Qu.:0.000000

Max. :2.00000 Max. :2.00000 Max. :2.0000 Max. :2.000000

### Days Since Last Replacement from Maintenance

A crucial data set in this example is the maintenance records which contain the information of component replacement records. Possible features from this data set can be, for example, the number of replacements of each component in the last 3 months to incorporate the frequency of replacements. However, more relevent information would be to calculate how long it has been since a component is last replaced as that would be expected to correlate better with component failures since the longer a component is used, the more degradation should be expected.

As a side note, creating lagging features from maintenance data is not as sensible as it is for telemetry and errors so the features from this data are generated in a more custom way. This type of ad-hoc feature engineering is very common in predictive maintenance since domain knowledge plays a big role in understanding the predictors of a problem. In the following, the days since last component replacement are calculated for each component type as features from the maintenance data.

In [25]:

*# create a binary column for each component. 1 if replacement occured, 0 if not.*

comprep <- maint %>%

select(datetime, machineID, comp) %>%

mutate(comp1 = as.integer(comp == "comp1"),

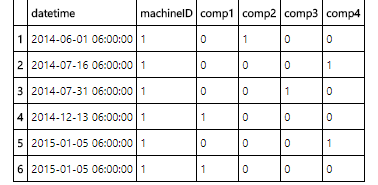
comp2 = as.integer(comp == "comp2"),

comp3 = as.integer(comp == "comp3"),

comp4 = as.integer(comp == "comp4")) %>%

select(-comp)

head(comprep)



In [26]:

comprep <- as.data.table(comprep)

setkey(comprep, machineID, datetime)

*# seperate different component type replacements into different tables*

comp1rep <- comprep[comp1 == 1, .(machineID, datetime, lastrepcomp1 = datetime)]*# component 1 replacements*

comp2rep <- comprep[comp2 == 1, .(machineID, datetime, lastrepcomp2 = datetime)]*# component 2 replacements*

comp3rep <- comprep[comp3 == 1, .(machineID, datetime, lastrepcomp3 = datetime)]*# component 3 replacements*

comp4rep <- comprep[comp4 == 1, .(machineID, datetime, lastrepcomp4 = datetime)]*# component 4 replacements*

*# use telemetry feature table datetime and machineID to be matched with replacements*

compdate <- as.data.table(telemetryfeat[,c(1:2)])

setkey(compdate, machineID, datetime)

*# data.table rolling match will attach the latest record from the component replacement tables*

*# to the telemetry date time and machineID*

comp1feat <- comp1rep[compdate[,.(machineID, datetime)],roll = TRUE]

comp1feat$sincelastcomp1 <- as.numeric(difftime(comp1feat$datetime, comp1feat$lastrepcomp1, units = "days"))

comp2feat <- comp2rep[compdate[,.(machineID, datetime)], roll = TRUE]

comp2feat$sincelastcomp2 <- as.numeric(difftime(comp2feat$datetime, comp2feat$lastrepcomp2, units = "days"))

comp3feat <- comp3rep[compdate[,.(machineID, datetime)], roll = TRUE]

comp3feat$sincelastcomp3 <- as.numeric(difftime(comp3feat$datetime, comp3feat$lastrepcomp3, units="days"))

comp4feat <- comp4rep[compdate[,.(machineID, datetime)], roll = TRUE]

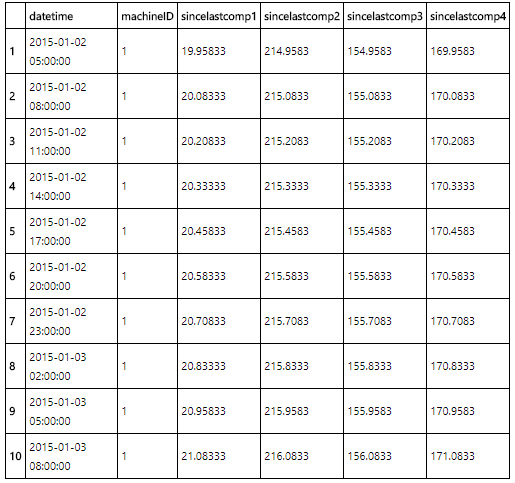
comp4feat$sincelastcomp4 <- as.numeric(difftime(comp4feat$datetime, comp4feat$lastrepcomp4, units = "days"))

*# merge all tables*

compfeat <-data.frame(compdate, comp1feat[,.(sincelastcomp1)], comp2feat[,.(sincelastcomp2)],

comp3feat[,.(sincelastcomp3)],comp4feat[,.(sincelastcomp4)])

head(compfeat,10)



### Machine Features

The machine features are used directly as they are since they hold descriptive information about the type of the machines and their ages which is the years in service. If the years in service information has been received in the form of dates denoting the date of first service then a transformation would have been necessary to turn those into a numeric values indicating the years in service.

Lastly, we merge all the feature data sets we created earlier to get the final feature matrix.

In [27]:

*# telemetry and error features have the same datetime*

finalfeat <- data.frame(telemetryfeat, errorfeat[,-c(1:2)])

*# merge with component features and machine features lastly*

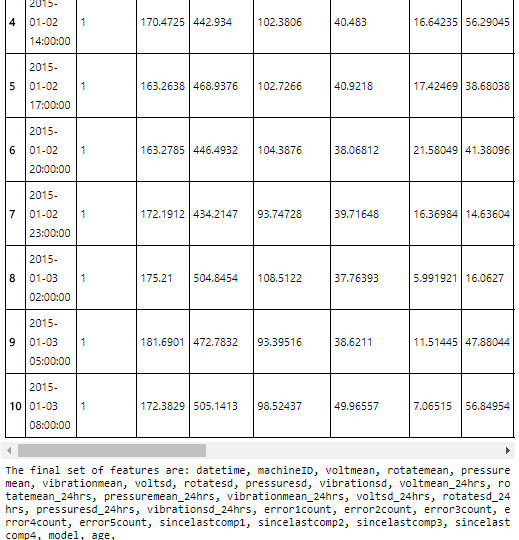
finalfeat <- finalfeat %>%

left\_join(compfeat, by = c("datetime","machineID")) %>%

left\_join(machines, by = c("machineID"))

head(finalfeat, 10)

cat("The final set of features are:",paste0(names(finalfeat), ","))



**Label Construction**

When using multi-class classification for predicting failure due to a problem, labeling is done by taking a time window prior to the failure of an asset and labeling the feature records that fall into that window as “about to fail due to a problem” while labeling all other records as “normal”. This time window should be picked according to the business case where in some situations it may be enough to predict failures hours in advance while in others days or weeks maybe needed to allow for the arrival of parts to be replaced as an example.

The prediction problem for this example scenerio is to estimate the probability that a machine will fail in the near future due to a failure of a certain component. More specifically, the goal is to compute the probability that a machine will fail in the next 24 hours due to a certain component failure (component 1,2,3 or 4). In the following, labelling is done by labeling all the feature records that fall into the 24 hours window before a failure due to component 1, component 2, component 3 and component 4 as comp1, comp2, comp3 and comp4 respectively.The rest of the records are labeled as "none" indicating, there is no failure within the next 24 hours.

In [28]:

*# left join final features with failures on machineID then mutate a column for datetime difference*

*# filter date difference for the prediction horizon which is 24 hours*

labeled <- left\_join(finalfeat, failures, by = c("machineID")) %>%

mutate(datediff = difftime(datetime.y, datetime.x, units = "hours")) %>%

filter(datediff <= 24, datediff >= 0)

*# left join labels to final features and fill NA's with "none" indicating no failure*

labeledfeatures <- left\_join(finalfeat,

labeled %>% select(datetime.x, machineID, failure),

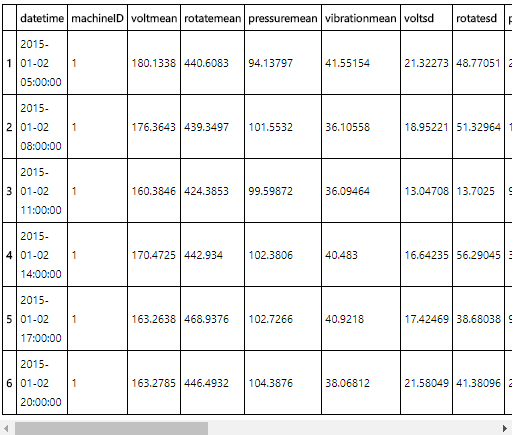
by = c("datetime" = "datetime.x", "machineID")) %>%

arrange(machineID,datetime)

levels(labeledfeatures$failure) <- c(levels(labeledfeatures$failure), "none")

labeledfeatures$failure[is.na(labeledfeatures$failure)]<-"none"

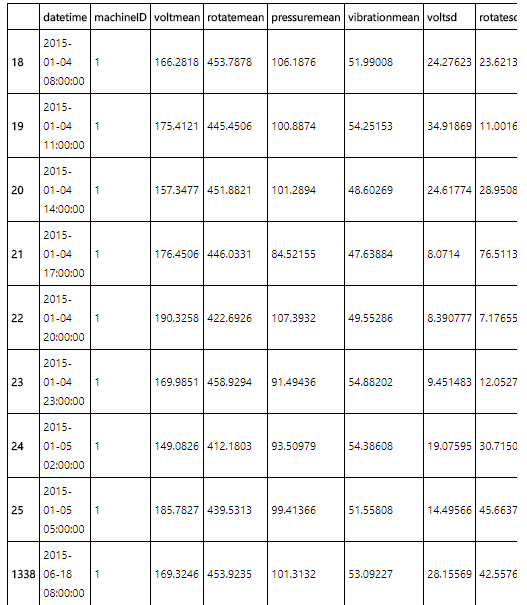
head(labeledfeatures)



Below is an example of records that are labeled as "comp4" in the failure column. First 8 records that fall into the same 24 hours are all labeled as a block. Next 8 records that are within 24 hours of another component 4 failure are also labeled as "comp4" as a block.

In [29]:

head(labeledfeatures[labeledfeatures$failure == "comp4",], 16)



## Modelling

In this section, we describe the modelling process and provide an example R model. With the same feature engineering, labelling and modelling steps, Azure Machine Learning Studio is also used to create a predictive model that uses multi-class logistic regression. You can check this experiment in the Cortana Intelligence Gallery: [Predictive Maintenance Modelling Guide Experiment](https://gallery.cortanaintelligence.com/Experiment/Predictive-Maintenance-Modelling-Guide-Experiment-1).

### Training, Validation and Testing

When working with data that comes with time-stamps such as telemetry and errors as in this example, splitting of data into training, validation and test sets should be performed carefully to prevent overestimating the performance of the models. In predictive maintenance, the features are usually generated using laging aggregates and consecutive examples that fall into the same time window may have similar feature values in that window. If a random splitting of training and testing is used, it is possible for some portion of these similar examples that are in the same window to be selected for training and the other portion to leak into the testing data. Also, it is possible for training examples to be ahead of time than validation and testing examples when data is randomly split. However, predictive models should be trained on historical data and valiadted and tested on future data. Due to these problems, validation and testing based on random sampling may provide overly optimistic results. Since random sampling is not a viable approach here, cross validation methods that rely on random samples such as k-fold cross validation is not useful either.

For predictive maintenance problems, a time-dependent spliting strategy is often a better approach to estimate performance which is done by validating and testing on examples that are later in time than the training examples. For a time-dependent split, a point in time is picked and model is trained on examples up to that point in time, and validated on the examples after that point assuming that the future data after the splitting point is not known. However, this effects the labelling of features falling into the labelling window right before the split as it is assumed that failure information is not known beyond the splitting cut-off. Due to that, those feature records can not be labeled and will not be used. This also prevents the leaking problem at the splitting point.

Validation can be performed by picking different split points and examining the performance of the models trained on different time splits. In the following, we use 3 different splitting points to train the model and look at the performances for different splits in the evaluation section.

In [23]:

*# split at 2015-08-01 01:00:00, to train on the first 8 months and test on last 4 months*

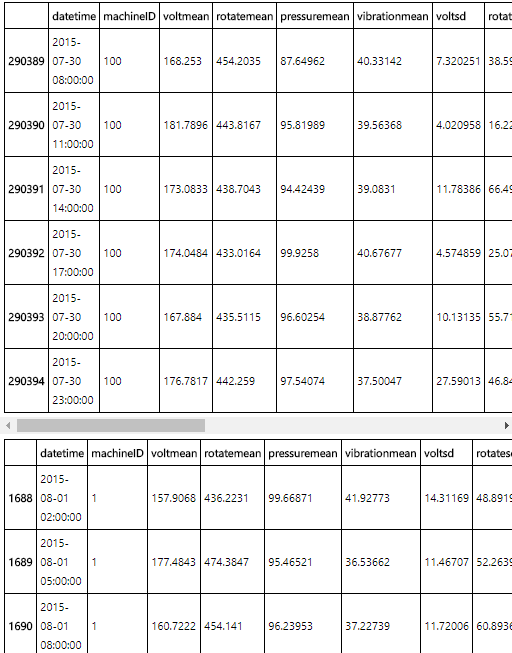
*# labelling window is 24 hours so records within 24 hours prior to split point are left out*

trainingdata1 <- labeledfeatures[labeledfeatures$datetime < "2015-07-31 01:00:00",]

testingdata1 <- labeledfeatures[labeledfeatures$datetime > "2015-08-01 01:00:00",]

tail(trainingdata1)

head(testingdata1)



In [24]:

*# split at 2015-09-01 01:00:00, to train on the first 9 months and test on last 3 months*

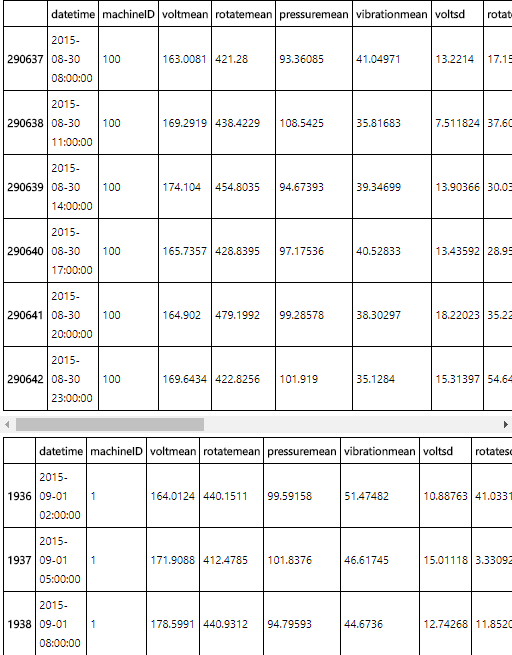
*# labelling window is 24 hours so records within 24 hours prior to split point are left out*

trainingdata2 <- labeledfeatures[labeledfeatures$datetime < "2015-08-31 01:00:00",]

testingdata2 <- labeledfeatures[labeledfeatures$datetime > "2015-09-01 01:00:00",]

tail(trainingdata2)

head(testingdata2)



In [25]:

*# split at 2015-10-01 01:00:00, to train on the first 10 months and test on last 2 months*

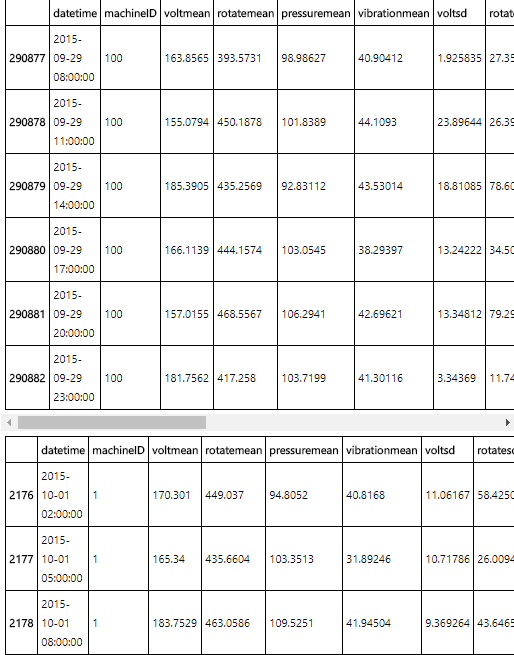
*# labelling window is 24 hours so records within 24 hours prior to split point are left out*

trainingdata3 <- labeledfeatures[labeledfeatures$datetime < "2015-09-30 01:00:00",]

testingdata3 <- labeledfeatures[labeledfeatures$datetime > "2015-10-01 01:00:00",]

tail(trainingdata3)

head(testingdata3)



In [26]:

install.packages("gbm")

library(gbm)

Installing package into &apos;/home/nbcommon/R&apos;

(as &apos;lib&apos; is unspecified)

The downloaded source packages are in

&apos;/tmp/RtmpaeVtR5/downloaded\_packages&apos;

Loading required package: survival

Loading required package: splines

Loading required package: lattice

Loading required package: parallel

Loaded gbm 2.1.1

In [27]:

*# create the training formula*

trainformula <- as.formula(paste('failure',

paste(names(labeledfeatures)[c(3:29)],collapse=' + '),

sep=' ~ '))

trainformula

failure ~ voltmean + rotatemean + pressuremean + vibrationmean +

voltsd + rotatesd + pressuresd + vibrationsd + voltmean\_24hrs +

rotatemean\_24hrs + pressuremean\_24hrs + vibrationmean\_24hrs +

voltsd\_24hrs + rotatesd\_24hrs + pressuresd\_24hrs + vibrationsd\_24hrs +

error1count + error2count + error3count + error4count + error5count +

sincelastcomp1 + sincelastcomp2 + sincelastcomp3 + sincelastcomp4 +

model + age

In [28]:

*# train model on 3 splits*

set.seed(1234)

gbm\_model1 <- gbm(formula = trainformula, data = trainingdata1,

distribution = "multinomial", n.trees = 50,

interaction.depth = 5, shrinkage = 0.1)

gbm\_model2 <- gbm(formula = trainformula, data = trainingdata2,

distribution = "multinomial", n.trees = 50,

interaction.depth = 5, shrinkage = 0.1)

gbm\_model3 <- gbm(formula = trainformula, data = trainingdata3,

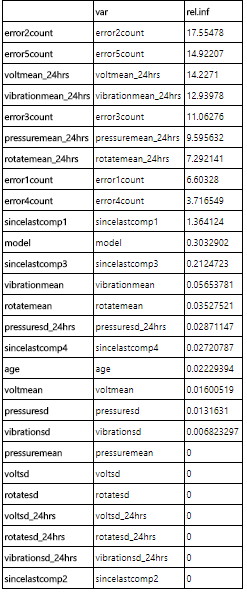
distribution = "multinomial", n.trees = 50,

interaction.depth = 5, shrinkage = 0.1)

In [29]:

*# print relative influence of variables for 1st model as an example*

summary(gbm\_model1)



**Evaluation**

In predictive maintenance, machine failures are usually rare occurrences in the lifetime of the assets compared to normal operation. This causes an imbalance in the label distribution which usually causes poor performance as algorithms tend to classify majority class examples better at the expense of minority class examples as the total misclassification error is much improved when majority class is labeled correctly. This causes low recall rates although accuracy can be high and becomes a larger problem when the cost of false alarms to the business is very high. To help with this problem, sampling techniques such as oversampling of the minority examples are usually used along with more sophisticated techniques which are not covered in this notebook.

Also, due to the class imbalance problem, it is important to look at evaluation metrics other than accuracy alone and compare those metrics to the baseline metrics which are computed when random chance is used to make predictions rather than a machine learning model. The comparison will bring out the value and benefits of using a machine learning model better.

In the following, we use an evaluation function that computes many important evaluation metrics along with baseline metrics for classification problems. For a detailed explanation of this function and the metrics please refer to the blog post [Computing Classification Evaluation Metrics in R](http://blog.revolutionanalytics.com/2016/03/com_class_eval_metrics_r.html) .

In [30]:

*# label distribution after features are labeled - the class imbalance problem*

ggplot(labeledfeatures, aes(x=failure)) +

geom\_bar(fill="red") +

labs(title = "label distribution", x = "labels")

In [31]:

*# define evaluate function*

Evaluate<-function(actual=NULL, predicted=NULL, cm=NULL){

if(is.null(cm)) {

actual = actual[!is.na(actual)]

predicted = predicted[!is.na(predicted)]

f = factor(union(unique(actual), unique(predicted)))

actual = factor(actual, levels = levels(f))

predicted = factor(predicted, levels = levels(f))

cm = as.matrix(table(Actual=actual, Predicted=predicted))

}

n = sum(cm) *# number of instances*

nc = nrow(cm) *# number of classes*

diag = diag(cm) *# number of correctly classified instances per class*

rowsums = apply(cm, 1, sum) *# number of instances per class*

colsums = apply(cm, 2, sum) *# number of predictions per class*

p = rowsums / n *# distribution of instances over the classes*

q = colsums / n *# distribution of instances over the predicted classes*

*#accuracy*

accuracy = sum(diag) / n

*#per class*

recall = diag / rowsums

precision = diag / colsums

f1 = 2 \* precision \* recall / (precision + recall)

*#macro*

macroPrecision = mean(precision)

macroRecall = mean(recall)

macroF1 = mean(f1)

*#1-vs-all matrix*

oneVsAll = lapply(1 : nc,

function(i){

v = c(cm[i,i],

rowsums[i] - cm[i,i],

colsums[i] - cm[i,i],

n-rowsums[i] - colsums[i] + cm[i,i]);

return(matrix(v, nrow = 2, byrow = T))})

s = matrix(0, nrow=2, ncol=2)

for(i in 1:nc){s=s+oneVsAll[[i]]}

*#avg accuracy*

avgAccuracy = sum(diag(s))/sum(s)

*#micro*

microPrf = (diag(s) / apply(s,1, sum))[1];

*#majority class*

mcIndex = which(rowsums==max(rowsums))[1] *# majority-class index*

mcAccuracy = as.numeric(p[mcIndex])

mcRecall = 0\*p; mcRecall[mcIndex] = 1

mcPrecision = 0\*p; mcPrecision[mcIndex] = p[mcIndex]

mcF1 = 0\*p; mcF1[mcIndex] = 2 \* mcPrecision[mcIndex] / (mcPrecision[mcIndex] + 1)

*#random accuracy*

expAccuracy = sum(p\*q)

*#kappa*

kappa = (accuracy - expAccuracy) / (1 - expAccuracy)

*#random guess*

rgAccuracy = 1 / nc

rgPrecision = p

rgRecall = 0\*p + 1 / nc

rgF1 = 2 \* p / (nc \* p + 1)

*#rnd weighted*

rwgAccurcy = sum(p^2)

rwgPrecision = p

rwgRecall = p

rwgF1 = p

classNames = names(diag)

if(is.null(classNames)) classNames = paste("C",(1:nc),sep="")

return(list(

ConfusionMatrix = cm,

Metrics = data.frame(

Class = classNames,

Accuracy = accuracy,

Precision = precision,

Recall = recall,

F1 = f1,

MacroAvgPrecision = macroPrecision,

MacroAvgRecall = macroRecall,

MacroAvgF1 = macroF1,

AvgAccuracy = avgAccuracy,

MicroAvgPrecision = microPrf,

MicroAvgRecall = microPrf,

MicroAvgF1 = microPrf,

MajorityClassAccuracy = mcAccuracy,

MajorityClassPrecision = mcPrecision,

MajorityClassRecall = mcRecall,

MajorityClassF1 = mcF1,

Kappa = kappa,

RandomGuessAccuracy = rgAccuracy,

RandomGuessPrecision = rgPrecision,

RandomGuessRecall = rgRecall,

RandomGuessF1 = rgF1,

RandomWeightedGuessAccurcy = rwgAccurcy,

RandomWeightedGuessPrecision = rwgPrecision,

RandomWeightedGuessRecall= rwgRecall,

RandomWeightedGuessWeightedF1 = rwgF1)))

}

In [32]:

*# evaluation metrics for first split*

pred\_gbm1 <- as.data.frame(predict(gbm\_model1, testingdata1,

n.trees = 50,type = "response"))

names(pred\_gbm1) <- gsub(".50", "", names(pred\_gbm1))

pred\_gbm1$failure <- as.factor(colnames(pred\_gbm1)[max.col(pred\_gbm1)])

eval1 <- Evaluate(actual=testingdata1$failure,predicted=pred\_gbm1$failure)

eval1$ConfusionMatrix

t(eval1$Metrics)

Predicted

Actual comp1 comp2 comp3 comp4 none

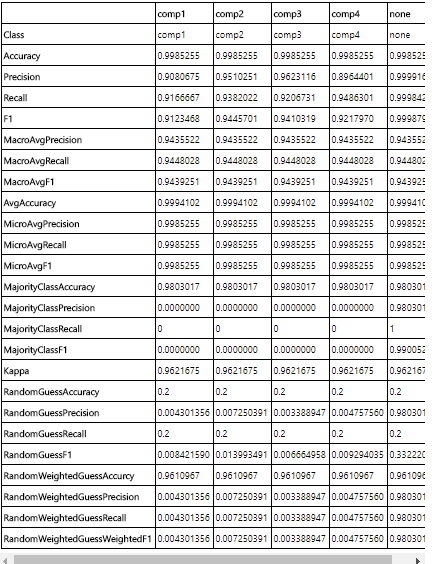
comp1 484 21 2 14 7

comp2 3 835 9 41 2

comp3 22 7 383 4 0

comp4 10 15 4 554 1

none 14 0 0 5 120315



In [33]:

*# evaluation metrics for second split*

pred\_gbm2 <- as.data.frame(predict(gbm\_model2, testingdata2,

n.trees = 50,type = "response"))

names(pred\_gbm2) <- gsub(".50", "", names(pred\_gbm2))

pred\_gbm2$failure <- as.factor(colnames(pred\_gbm2)[max.col(pred\_gbm2)])

eval2 <- Evaluate(actual=testingdata2$failure,predicted=pred\_gbm2$failure)

eval2$ConfusionMatrix

t(eval2$Metrics)

Predicted

Actual comp1 comp2 comp3 comp4 none

comp1 378 15 0 8 7

comp2 1 693 8 17 3

comp3 8 8 301 2 1

comp4 16 9 6 418 1

none 11 0 0 5 95982



In [34]:

*# evaluation metrics for third split*

pred\_gbm3 <- as.data.frame(predict(gbm\_model3, testingdata3,

n.trees = 50,type = "response"))

names(pred\_gbm3)<-gsub(".50", "", names(pred\_gbm3))

pred\_gbm3$failure <- as.factor(colnames(pred\_gbm3)[max.col(pred\_gbm3)])

eval3 <- Evaluate(actual=testingdata3$failure,predicted=pred\_gbm3$failure)

eval3$ConfusionMatrix

t(eval3$Metrics)

Predicted

Actual comp1 comp2 comp3 comp4 none

comp1 284 12 0 8 2

comp2 4 555 0 9 2

comp3 8 0 213 3 0

comp4 8 15 5 293 1

none 8 0 0 5 72437



In predictive maintenance, the interest is mostly around how well the model catces the actual failures which translates into the recall metric. Below, we compare the recall rates for each failure type for the three models. The recall rates for all components as well as no failure are all above 90% meaning the model was able to capture above 90% of the failures correctly.

In [35]:

*# report the recall rates for the models*

rownames <- c("comp1","comp2","comp3","comp4","none")

rownames

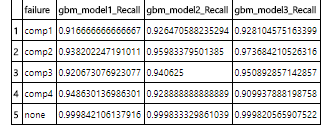
data.frame(cbind(failure = rownames,

gbm\_model1\_Recall = eval1$Metrics$Recall,

gbm\_model2\_Recall = eval2$Metrics$Recall,

gbm\_model3\_Recall = eval3$Metrics$Recall))

1. 'comp1'
2. 'comp2'
3. 'comp3'
4. 'comp4'
5. 'none'



## Summary

In this notebook, the steps of implementing a predictive maintenance model is provided using an example scenario where the goal is to predict failures due to certain components of a machine. Typical steps of predictive maintenance such as feature engineering, labelling, training and evaluation are explained using the example data sets. Predictive models are built both using R packages and Azure Machine Learning Studio.