IBM Resilient

resilient

Incident Response Platform

MACHINE LEARNING FUNCTION REFERENCE GUIDE v1.1.1

Resilient Incident Response Platform Machine Learning Function Reference Guide

Platform Version	Publication	Notes
1.1.1	July 2020	Include pandas .whl file in source code.
1.1.0	March 2020	Information on new ml.config file that is used to build, train, and test machine learning models.
1.0.0	December 2018	Initial publication.

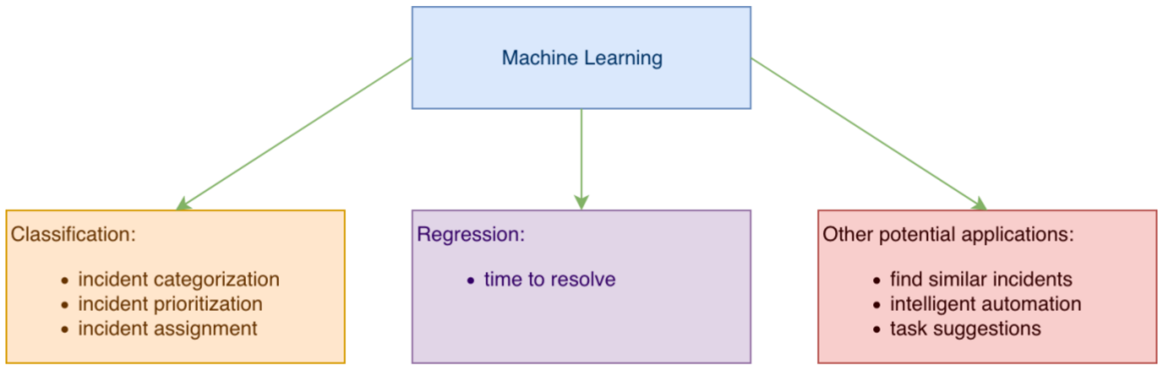
Table of Contents

1. Overview

This is a reference guide for the Resilient Machine Learning Function. It is a companion guide to the *Resilient Incident Response Platform Machine Learning Function User Guide*.

Machine learning can extract knowledge from historical incidents stored in a Resilient platform. This knowledge normally includes useful pattern information of the customer environment. More importantly, it can include the experience of security analysts who have worked on those historical incidents. In some sense, a machine learning model learns from the previous decisions the security analysts made. Thus, machine learning can be very useful in assisting security analysts to make quick and proper response.

The following graph shows possible applications of machine learning in an incident response system like the Resilient platform. Please note that this release supports classification only. Regression and other potential applications are not supported.



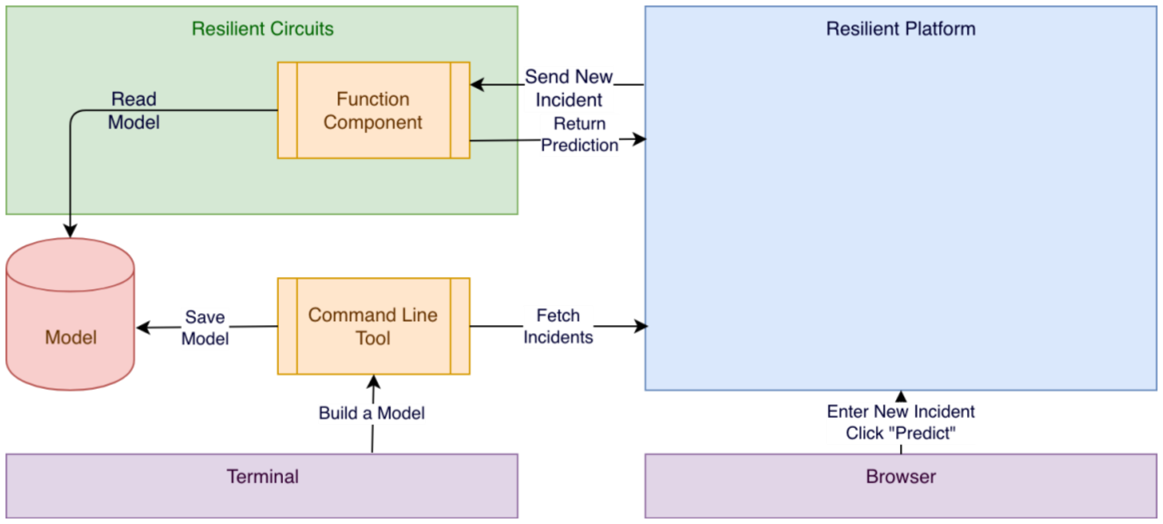
The Resilient Machine Learning Function is highly customized for the incident response system. It is also fully integrated to the Resilient platform to provide the best user experience, and can be incorporated into a custom workflows in flexible ways. In addition, user data is processed locally, thus customers do not need to be concerned about transmitting sensitive data to the cloud when a machine learning model is built.

This guide includes background information about features supported by the Resilient Machine Learning Function. In addition, it includes recommendations to resolve common issues Resilient platform users might face.

1.1. Architecture

The Resilient Machine Learning Function contains two components.

- A command line tool, called res-ml, to build a machine model.
- A function component to predict using a machine model.



This guide focuses on the res-ml command line tool to build machine learning models.

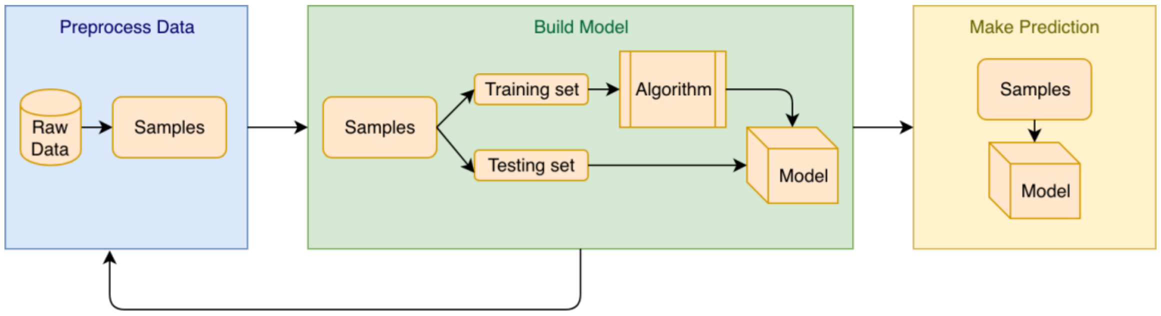
1.2. Dependencies

The res-ml tool is a Python based application. It depends on the following third party Python packages for machine learning support:

- Scikit-learn 0.20.4: An open source Python package for machine learning with BSD license. It is built on Numpy and SciPy. The res-ml tool uses the machine learning algorithms supported by scikit-learn.
- Pandas 0.24.2: An open source Python package for easy-to-use data structures and related analysis tools. It also has a BSD license. The res-ml tool uses Pandas to manipulate datasets, which are collection of incidents. This includes extracting features from incidents, filtering samples, and up-sampling training dataset.
- Numpy 1.16.6: A fundamental package for scientific computing with Python with BSD license.
- Scipy 1.2.3: An open source Python package for mathematics, science, and engineering with BSD license.

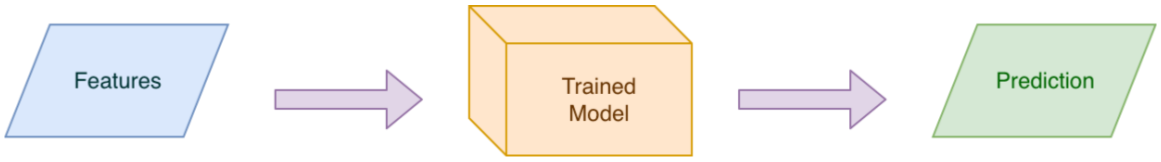
2. Workflow of Machine Learning

To use the res-ml tool to build a machine learning model, follow the general workflow of machine learning as shown below.



There are three steps. The first step is to preprocess the raw data and get it into shape. Filters can be used in this to remove dirty data. In addition, some raw data needs to be scaled and transformed for optimal performance of machine learning algorithms. The raw data here is the collection of historical incidents. The output of this step is the samples to be used in building a machine model. These samples are sometimes referred to as dataset.

The second step is to build a machine learning model. In this step, users need to pick which incident field they want a machine model to predict. This is referred to as the prediction field. They also need to pick incident fields that are relevant to the prediction fields. Those fields will be the features for the machine learning model. Simply put, the features are the inputs for the machine learning model, and the prediction field is the output from the machine learning model. Therefore, the quality of the input data is critical for building a successful machine learning model. One can say that the quality of the input data determines the upper limit of the performance that a machine learning model can possibly reach.



Also in this step, users need to pick an algorithm for the machine learning model. The samples generated from the first step is split into two subsets. One is used for training the model using the selected algorithm. The other is used to test the trained model. The performance of the trained model is measured during the test. Note that if the performance is not satisfactory, users can go back to the beginning and make adjustments in the first two steps, so as to fine-tune the model. They can repeat this until a good model is built.

The third step is to make a prediction using the model built in the second step. New data is fed into the model to generate a prediction. Note that after the machine learning model has been trained and tested, this can be done repeatedly and very rapidly.

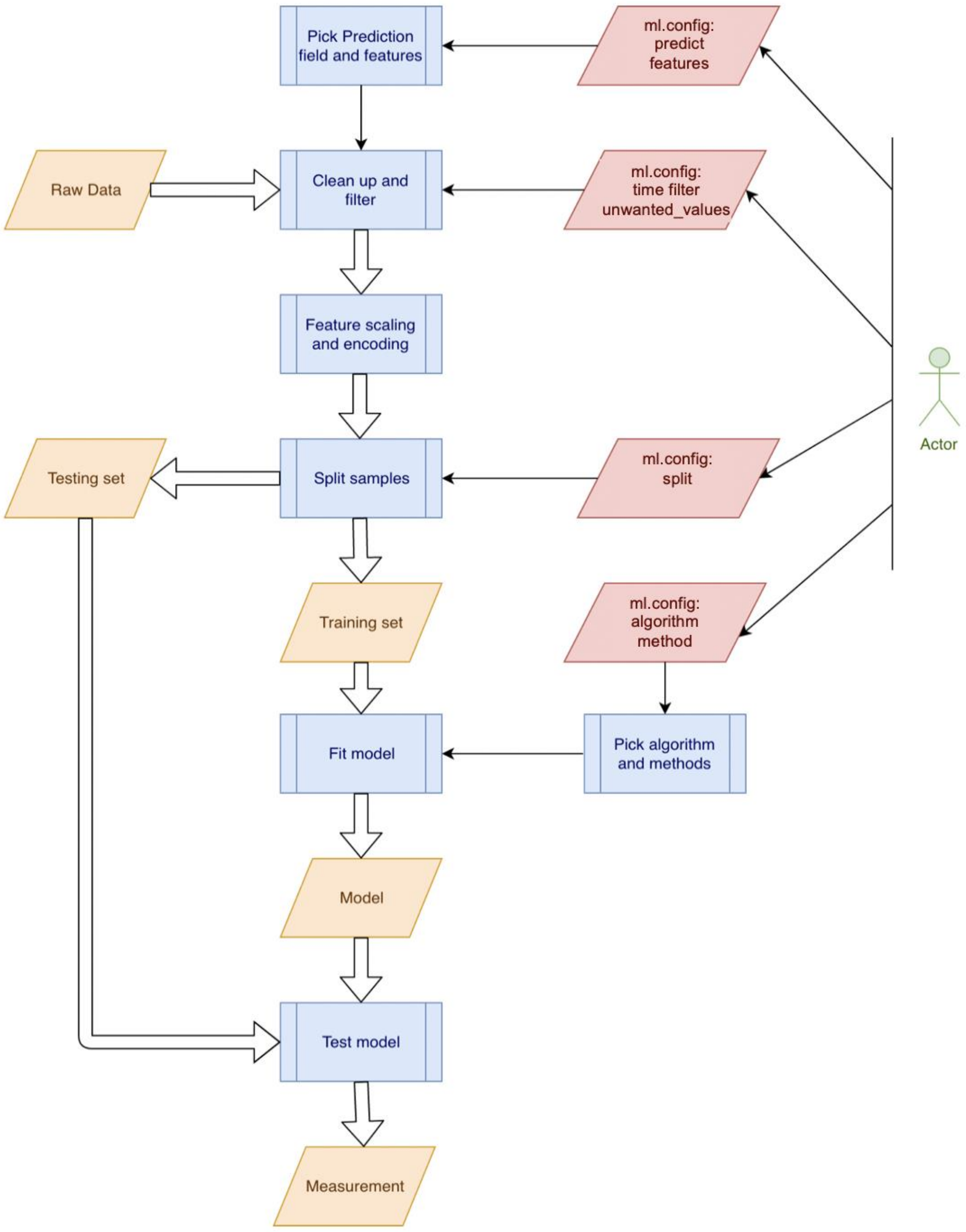
For the Resilient Machine Learning Function, the res-ml tool handles the first two steps, and the function component handles the last step. A machine learning model built by the res-ml tool is saved into a file. The function component reads the model from a file, and then makes a prediction using it.

Note that the Python package, pickle, is used to serialize and de-serialize a machine learning model. If users want to build a model from one machine and then use it from another machine, they need to make sure that the same version of Python and pickle are installed into both machines.

3. Overview of the res-ml Tool

The res-ml tool follows the general workflow described in the previous section.

The following data/control flow diagram provides an overview about how the raw data is built into a machine model, and how users can control the building process by modifying settings in the ml.config file. The following sections provide the details of each process shown in the flow diagram.


Pick Prediction field and features
ml.config: predict features
Raw Data
Clean up and filter
ml.config: time filter unwanted_values
Feature scaling and encoding
Actor
Testing set
Split samples
ml.config: split
Training set
ml.config: algorithm method
Fit model
Pick algorithm and methods
Model
Test model
Measurement

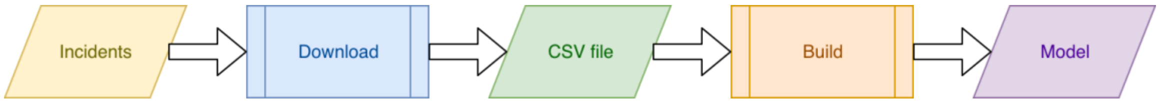
4. Preprocess Raw Data

This process is about getting raw data in shape, and it is a critical step in machine learning. The res-ml tool applies the following techniques automatically because they are commonly recognized as the best practices.

4.1. Remove Samples with Blank Values

When an incident is created, it is quite possible that one or more fields are left blank. A field with a missing value can cause difficulties for a machine learning model if this field is selected as a feature or the prediction field. For example, users can select “confirmed” as a feature to predict “severity”. If “confirmed” is left blank when an incident is created, then this incident can confuse a machine learning model. It needs to be fixed before the dataset is used.

There are two commonly used solutions for this problem. First, if users know how to fill in the missing values, they can do it in this step. The advantage of this solution is that it preserves the total count of samples. But the disadvantage is that it is not an easy approach. As shown in the dataflow of the Resilient Machine Learning Function below, the incidents are first downloaded and saved into a CSV file before they are used to build a machine model. This gives users the flexibility to preprocess the data before it is used.



Therefore, users run the following command:

```
res-ml download -o resilient_incidents.csv
```

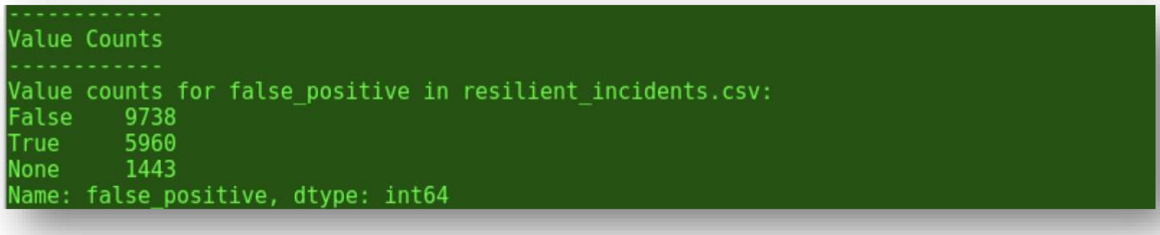
Then they can edit the CSV file and fill in the missing values if they want.

The other approach is to remove the samples with missing values if the corresponding fields are selected as features. The res-ml tool automatically removes those samples in the build process before data is fed to a machine learning algorithm. In other words, if advanced users want to do custom data preprocessing, they need to do it using the CSV file *before* the build command is used.

For the advanced user, there is a useful command from the res-ml tool called count_value. It can show the sample count for each value of a given field. The user can use it to check a field before it is used as one of the features or the prediction field.

Example:

```
res-ml count_value -i resilient_incidents.csv -f false_positive
```



```
------------
Value Counts
------------
Value counts for false_positive in resilient_incidents.csv:
False    9738
True     5960
None     1443
Name: false_positive, dtype: int64
```

In this example, the value count of a custom field called “false_positive” is shown. There are 9738 samples for value “False” here. Note that value “None” represents a missing value. The Pandas package used in the res-ml tool automatically converts a blank value into “None”. Here we can see that there are 1443 samples with blank values.

4.2. Feature Scaling

For most machine learning algorithms, to get the optimal performance, the numerical features need to be brought to the same scale. Otherwise, the algorithms would pay the most attention to the feature with the bigger scale and ignore the others.

The res-ml tool performs feature scaling automatically. The following two approaches are used depending on the algorithm selected. The following sections provide details about the algorithms and the approaches.

For advanced users, note that the approach cannot be customized. They are hard coded in the res-ml tool.

Algorithm	LabelEncoder	Min-max scaling
Logistic Regression	Yes	
Decision Tree	Yes	
Random Forest	Yes	
SVM		Yes
Gaussian Naïve Bayes	Yes	
Bernoulli Naïve Bayes	Yes	
K-Nearest Neighbor	Yes	

4.2.1. LabelEncoder

The scikit-learn LabelEncoder is used to scale numerical features for some algorithms, as shown in the above table. The value of a feature is encoded between 0 and number_of_values-1. For example, assume that a feature has the following values:

```
[1, 2, 10, 100, 1000]
```

It is encoded into:

```
[0, 1, 2, 3, 4]
```

Note that the maximum value after being encoded is 4, which is the count of distinct values minus 1.

4.2.2. Min-Max Scaling

The other approach to normalize a numerical feature is to do a special case of min-max scaling. It uses the following equation to encode value a value x:

$$x_{encoded} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

4.3. Encoding Categorical Features

For many algorithms implemented in scikit-learn, categorical features (for example, SELECT fields in the Resilient platform) need to be encoded as integer values. Here, the scikit-learn LabelEncoder is used to perform the encoding.

4.4. Filters

Filters can be used to exclude the dirty data. The res-ml tool supports two kinds of filters.

4.4.1. Time Filter

Incident samples can be filtered by setting start time and end time. As mentioned above, the quality of the data determines the best possible performance any machine learning model can reach. To get high quality data, the users of the Resilient platform need to follow formal procedures and fill in values for necessary fields accurately. If the users know that the quality of data is good in a certain time frame, they can use this time filter to include incidents only created within that time frame.

To set a time filter, edit the ml.config and enter values for time_start and/or time_end. The format is “YYYY-mm-dd”.

Example:

```
time_start=2018-10-01
time_end=2018-10-08
```

This time filter includes incidents created between 2018-10-01 and 2018-10-08 only in the training and testing a machine learning model.

4.4.2. Samples of Unwanted Value of Prediction Field

This filter is specific for prediction field.

Some values of the prediction field can cause confusion to a machine learning model. For example, a Resilient platform user creates an “Unknown” selection for the incident field “severity”. This could be true when an incident is created, but eventually before the incident is closed, a more accurate value shall be selected and stored. If a user forgot to update this field, then the corresponding incident can cause confusion when a machine model is trained using this data to predict “severity”.

Also, just like features with missing values, a prediction field with a missing value shall be excluded as well.

To exclude samples with an unwanted value, the users can edit ml.config and enter this value into the list of unwanted_values.

Example:

```
unwanted_value=None, 1234
```

Note that the res-ml tool uses a Python package called Pandas to manipulate samples. Pandas treats an empty value as None. As a result, the following line in ml.config excludes samples with a blank value for the prediction field.

```
unwanted_value=None
```

If users do not want to use this setting, they can remove it or comment it out.

4.5. Max Sample Counts

The res-ml tool builds a machine learning model locally. Depending on the spec of the local machine, the build process could be lengthy. The users can control the maximum number of incidents to be used as samples to train an algorithm. However, please keep in mind that, in general, more incidents give a more accurate result. In addition, for most algorithms (except K-Nearest Neighbor), this affects only the performance of the process of building a model. Once a model is built, the performance of the prediction process is not affected by the number of incidents used to build the model.

This is done in the ml.config by setting:

```
max_count=10000
```

4.6. Splitting Samples for Training and Testing

Once the samples are cleaned up and properly encoded, they are split into two subsets, one for training and one for testing. It is important to test a trained model with unseen samples.

The main reason for validating a machine model is that we need to know how well a model can predict real-world data in the future. This is called generalization performance of a model, because a model essentially generalizes what it learns from the historical data and applies the knowledge to predict real-world data in the future.

The validation using the test data subset generates a measurement that we can use to estimate the generalization performance.

Please refer to the “Test a machine learning model” section for more details.

Users can change the setting of “split” in ml.config:

```
split=0.5
```

5. Train a Machine Learning Model

5.1. Select a Prediction Field

The Prediction field can be selected from built-in or custom fields of an incident. For example, “severity” is one possible choice to predict.

This is done by editing the ml.config file. For example:

```
prediction=severity_code
```

Another possible choice for a built-in field is the owner of an incident. A machine learning model can then look for possible pattern in the relation between certain types of incidents and the owner of them.

5.2. Select Features

Features can be selected from custom fields as well. Features can be set in the ml.config file as a list of features separated by comma. For example:

```
features=incident_type_ids, confirmed, negative_pr_likely
```

5.3. Select an Algorithm

Generally speaking, each machine learning algorithm has its advantages and disadvantages. *For advanced users*, it is recommended that they build several models using different algorithms, and then compare the performance.

The following algorithms are supported:

- Logistic Regression
- Decision Tree
- Random Forest
- SVM
  - Linear kernel
  - Gaussian kernel
- Gaussian Naïve Bayes
- Bernoulli Naïve Bayes
- K-Nearest Neighbor

Please refer to the “Supported Algorithms” section for more information about these algorithms.

Users can select an algorithm by setting the algorithm field in the ml.config file. For example:

```
algorithm=Logistic Regression
```

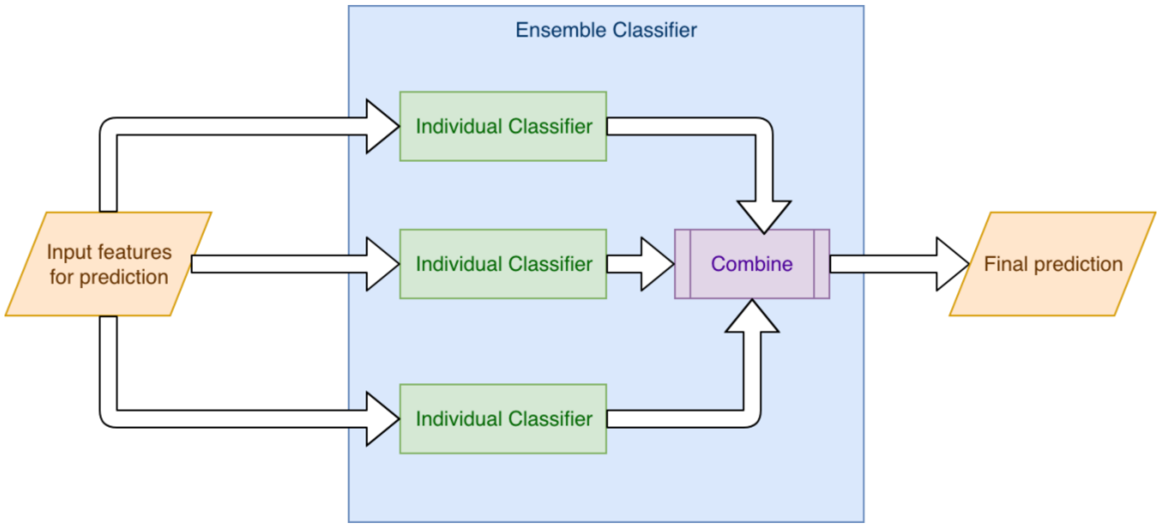
Use “SVM” for SVM with linear kernel, and “SVM with Gaussian kernel” for the other non-linear SVM.

5.4. Select Ensemble Methods (Optional)

This is for advanced users. Ensemble methods are optional in building a machine learning model. Also, ensemble methods discussed here only apply to classification models. Therefore a model is also referred as a classifier.

An ensemble method combines multiple classifiers into an ensemble classifier. These classifiers can use the same or different algorithms in general. But at this point, the res-ml tool only supports individual classifiers using the same algorithm. The resulted ensemble classifier theoretically can provide better generalization performance than each individual classifier.

In general, multiple classifiers are trained using the whole or different portions of the same dataset. When these classifiers are used to predict, the same input is fitted to each individual classifier. Then the predictions from them are combined to generate the final prediction. One popular way to combine prediction is majority voting.



The res-ml tool supports two ensemble methods:

- Bagging
- Adaptive Boosting

The users can select an ensemble method by setting the method field in the ml.config file. For example:

```
method=Adaptive Boosting
```

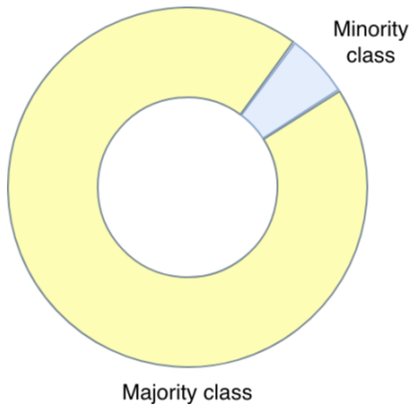
The res-ml tool does not apply an ensemble method if the method field is commented out or if it is set to None. Please refer to the “Supported Ensemble Methods” for more information about these methods.

5.5. Compensate Imbalanced Dataset

An imbalanced dataset is quite common for user data found in incident response systems. To get better performance, there are techniques to compensate an imbalanced dataset.

5.5.1. Imbalanced Dataset

An imbalanced dataset is one that, for a prediction field, one or more classes dominate. The under-presented class is normally called minority class, while the over-presented one is called majority class.



Imbalanced dataset may occur in data collected from spam filtering, fraud detection, or malware attack. This depends on the prediction field being selected. Thus for the same dataset, it can be both balanced and imbalanced, depending on which field is selected as the prediction field. For example, using the same set of incidents, it can be imbalanced when it is used to predict severity, while it is balanced when it is used to predict the owner of an incident.

When imbalanced dataset is used to train a machine model, high overall accuracy can be easily achieved without the need to learn any useful knowledge from the dataset. For example, assume that data is collected from monitoring emails in order to detect spam. If the spam emails only represent 10% of the total data being collected, then this is a typical imbalanced dataset. The majority class represents normal emails, while the minority class represents spam emails here. To predict the future using machine learning, it is assumed that the future data has the same distribution as the historical data. Thus an overall accuracy as high as 90% can be easily achieved if a dummy model just predicts everything to be the majority class, meaning a normal email. This dummy model does not need to learn anything from the features. This dummy model is a special case of the Dummy Classifier discussed in the following subsection.

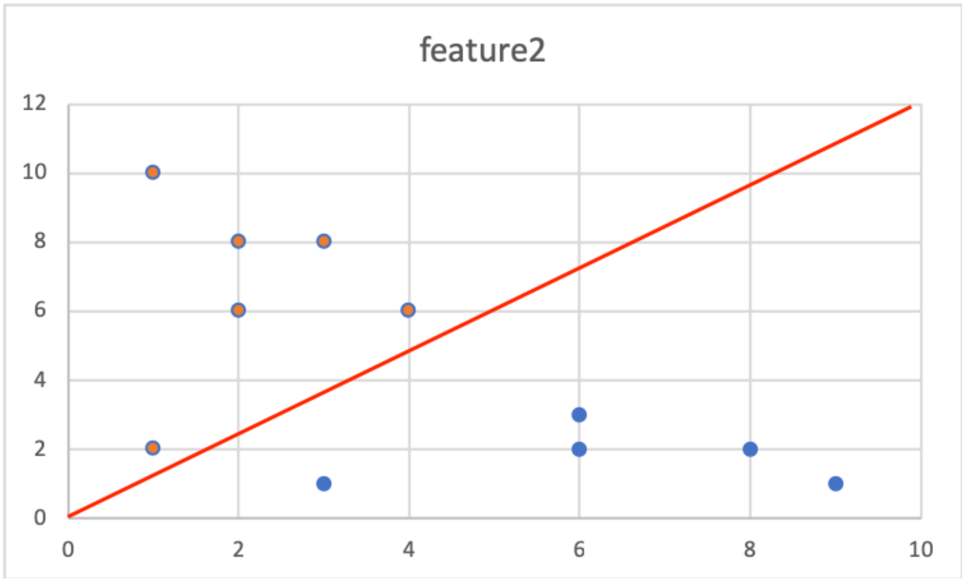
The overall accuracy of 90% seems high, but if we look at the accuracy for each value of the prediction field, the results are less satisfactory. The accuracy for the majority class is good, but this dummy model completely ignores the minority class.

	Accuracy
Overall	90%
For normal	100%
For spam	0%

This example shows that the overall accuracy is not the perfect measurement when handling an imbalanced dataset, since a model with a high overall accuracy like this dummy model actually does not know how to predict a spam email.

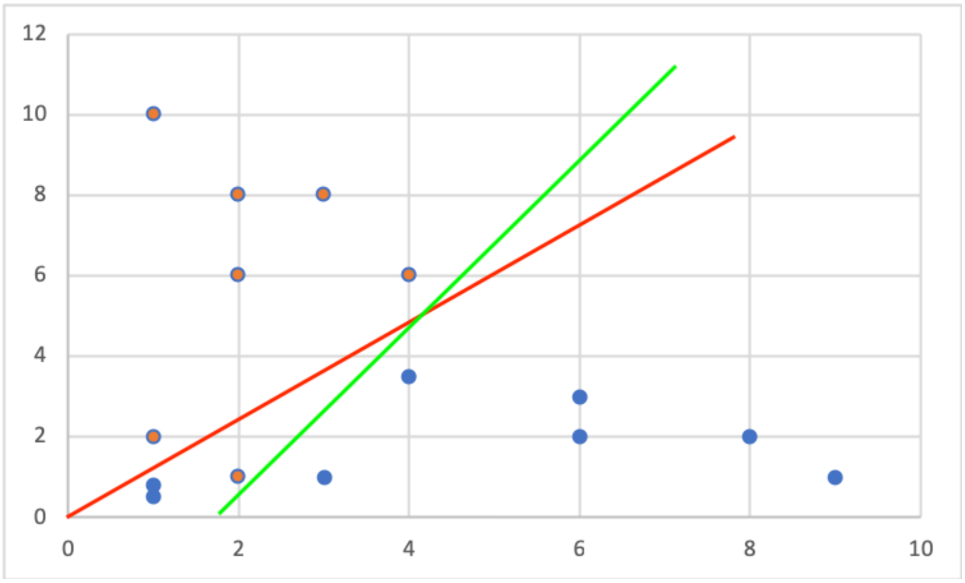
A real machine learning model (without compensation) in general can improve the overall accuracy. The accuracy for the majority is most likely lower than 100%. The accuracy for minority class is most likely higher than 0%. However, most of the time, the accuracy for a minority class is still low if no compensation technique is used, as explained below using a binary classification.

Assume that a machine learning model is built for a binary classification. For simplicity, we consider only two features so that we can draw it in a 2D space. Each sample is represented by a dot, with the x-axis for feature1 and y-axis for feature2.



This graph shows an ideal case in which a perfect solution exists. The red line can separate the red dots and blue dots completely. Note that the red line is what the machine model learns from the data. Later on, when this model is used to make prediction, it just needs to figure out whether the new dot is above or below the red line. An ideal case shown here means 100% overall accuracy and 100% accuracies for both the red class and the blue class.

In reality, this does not happen so often. Most of the time, the overall accuracy cannot reach 100%. This means the machine learning model misclassifies some of the samples. When this happens, the model needs to make decisions to choose different solutions. For example, in the following graph, the red line missed one red sample, while the green line missed two blue samples. The samples misclassified results in accuracy smaller than 100%. In general, an algorithm tries its best to maximize the overall accuracy, and thus the red line here is a preferable solution. This is the correct choice if the dataset is balanced.



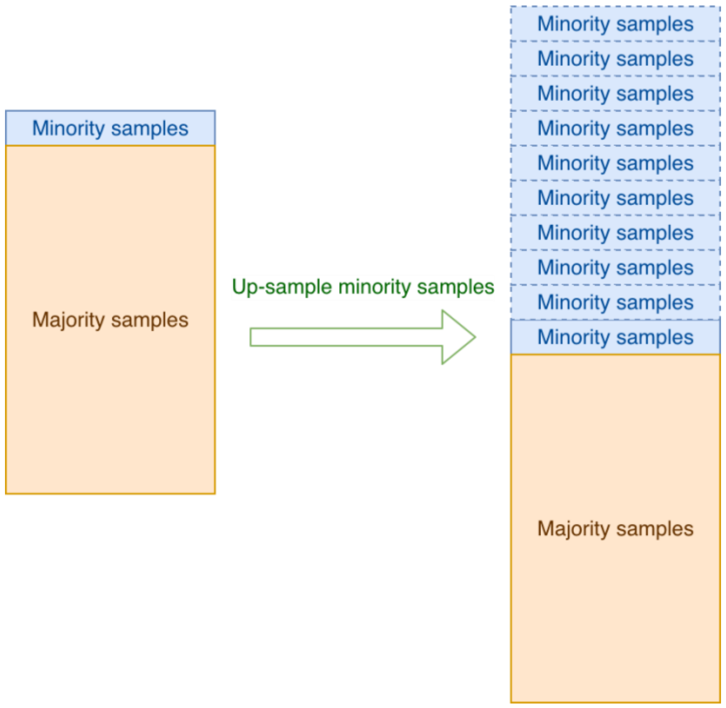
If the dataset is imbalanced, then there is another factor to consider. Assume now that the blue class above is the minority class with only 50 samples, while the count of red samples is 1000. Missing two red dots only has a small negative impact on the accuracy of the red class (around 0.2%). But missing one blue dot lowers the accuracy for blue class by 2%.

Now which line to choose depends on what the users care about. If the users care more about the overall accuracy and the accuracy for the majority class, then the red line is the preferable one. Or say there is no need to compensate the model. If the users care more about the accuracy for the minority class, then the green line is better. This solution sacrifices the overall accuracy and the accuracy for the majority class a little, but it has a big improvement for the accuracy of the minority class. For our example here of monitoring emails, the accuracy for predicting spam email is more important than the accuracy of predicting a normal email. Thus it makes sense to compensate the model.

A machine learning algorithm without compensation for the imbalance dataset, in general, picks the red line to maximize the overall accuracy. There are techniques to compensate so that an algorithm will pick the green line instead.

5.5.2. Up-Sampling

The first technique is called up-sampling. This technique makes multiple copies of each minority sample, and inserts these copies to the original dataset, so that the count of the minority class is the same as the majority class.



Now each minority sample dot multiplies. As a result, a machine learning algorithm shall choose to misclassify a single majority sample, rather than a group of minority samples.

One important note here. Up-sampling must be applied after the original dataset is split into the training set and the testing set. If we apply up-sampling first, then copies from the same original minority samples might be split into the training set and the testing set. This means a machine learning model is then trained and tested with the same sample. Also, up-sampling shall be applied to the training set only so that it does not affect the performance measurement obtained from the testing dataset.

To use up-sampling to compensate an imbalanced dataset, set the following in ml.config:

```
imbalance_upsampling=true
```

5.5.3. Balanced class_weight

Another useful technique is to set the class_weight of an algorithm. Some algorithms support setting weight to each class of the prediction field. The res-ml tool supports setting class_weight as “balanced”. It means a weight inversely proportional to the class frequencies in the input dataset. For example, if a minority class represents only 1/3 of the total samples, then its weight is 3 when class_weight is set to “balanced”. This setting essentially tells a machine learning algorithm that a single minority sample shall be considered as three. Thus it can achieve similar impact as the up-sampling technique.

Not all the algorithms support class_weight. In general, one of these two techniques is needed to compensate an imbalanced dataset.

Algorithm	Up-sampling	Class_weight
Logistic Regression	Yes	Yes
Decision Tree	Yes	Yes
Random Forest	Yes	Yes
SVM	Yes	Yes
SVM with Gaussian kernel	Yes	Yes
Gaussian Naïve Bayes	Yes	No
Bernoulli Naïve Bayes	Yes	No
K-Nearest Neighbors	Yes	No

To use this technique, set class_weight in ml.config as follows:

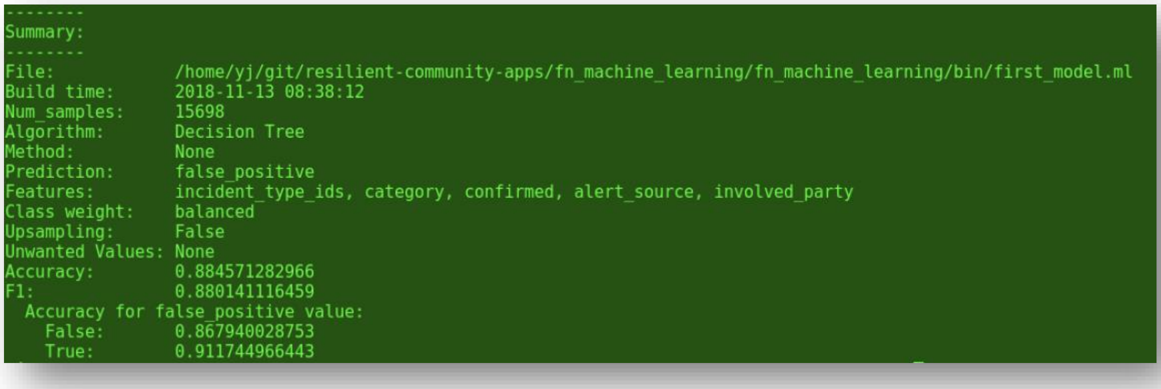
```
class_weight=balanced
```

6. Test a Machine Learning Model

The res-ml tool prints out a summary of the performance once a model is built. The measurements shown in this summary are obtained by validating the trained model using the test dataset. The following measurements are shown:

- Overall accuracy
- Individual accuracy for each value of the predicted field
- F1-score

The following is an example of the summary of building a machine learning model, which shows information about how the model is configured then the measurements. The overall accuracy is 88.46%. F1-score is 0.8801. Accuracy for False class is 86.79% and the accuracy for True class is 91.17%.



```
--------
Summary:
--------
File:            /home/yj/git/resilient-community-apps/fn_machine_learning/fn_machine_learning/bin/first_model.ml
Build time:      2018-11-13 08:38:12
Num_samples:     15698
Algorithm:       Decision Tree
Method:          None
Prediction:      false_positive
Features:        incident_type_ids, category, confirmed, alert_source, involved_party
Class weight:    balanced
Upsampling:      False
Unwanted Values: None
Accuracy:        0.884571282966
F1:              0.880141116459
  Accuracy for false_positive value:
    False:       0.867940028753
    True:        0.911744966443
```

6.1. Overall Accuracy

The overall accuracy is computed using the following equation:

$$Overall\ Accuracy = \frac{Count\ of\ samples\ predicted\ correctly}{Total\ count\ of\ testing\ samples}$$

The total accuracy given in the above screenshot is 88.46%.

6.2. Individual Accuracy

The accuracy for each value is defined as the following:

$$Individual\ Accuracy = \frac{Count\ of\ correct\ prediction\ of\ \ individual\ value}{Count\ of\ samples\ with\ individual\ value}$$

For example, the above screenshot shows two values: False and True.

The individual accuracy for False is 86.79%. The individual accuracy for True is 91.17%.

The individual accuracies are important when handling an imbalanced dataset.

6.3. F1-Score

F1-score can be used as a measure of a machine learning model. If the prediction field has only two possible values, it is then a binary classification. For this case, F1-score is defined as below:

$$F1 = \frac{2 * precision * recall}{(presion + recall)}$$

$$precision = \frac{true\_positive}{true\_positive + false\_positive}$$

$$recall = \frac{true\_positive}{true\_positive + false\_negative}$$

As shown in its definition, F1-score is a weighted average of the precision and recall. Its minimum value is 0, and maximum is 1. In general, a higher F1-score means a better performance of a machine learning model.

As for the multi-class case, F1-score is the average of the F1-score of each class.

The scikit-learn.metrics.f1_score function is used to compute the F1-score.

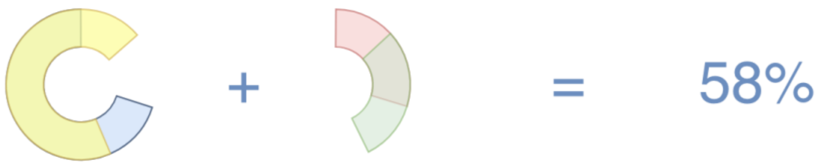
6.4. Dummy Classifier

This for advanced users. The scikit-learn package provides a test classifier called Dummy Classifier. It uses simple rules to make a prediction. The purpose of this classifier is to generate a simple baseline that can be used as comparison in evaluating other real classifiers.

The dummy classifier included in the res-ml tool uses the “stratified” strategy. This means it makes a prediction by respecting the training set’s class probability. For example, assume that the training set has two classes only: “True” and “False” for a customer field called “is_significant”, and the distribution is 70% vs 30%. The dummy classifier randomly makes 70% “True” prediction and 30% “False” predictions.

Now assuming the testing data set has a similar class distribution, the overall accuracy of this dummy classifier can theoretically computed as:

$$Overall\ Accuracy = 70\% * 70\% + 30\% * 30\% = 58\%$$



The reason is the following. As shown about, 70% of the testing set shall be the “True” class, as shown above as the blue arc. Note that portion of the blue arc is actually under the yellow arc. For these samples, the dummy classifier randomly predicts 70% of them to be “True”, represented by the portion of the yellow arc that overlaps on top of the blue arc, and 30% to be “False”, represented by the rest of the yellow arc. So the true-positive rate is 70%*70% here, meaning 70% of all the samples are “True” and out of them 70% predicted correctly by the Dummy Classifier. Similarly, the true-negative is 30%*30% here.

The accuracy of each class is the same as its probability:

$$Accuracy\ for\ Class\ True = 70\%$$
$$Accuracy\ for\ Class\ False = 30\%$$

For “is_significant” as “True”, the F1-score can be computed as:

$$True\_positive = 70\% * 70\% = 0.49$$

$$False\_positive = 30\% * 70\% = 0.21$$

$$True\_negative = 30\% * 30\% = 0.09$$

$$False\_negative = 70\% * 30\% = 0.21$$

$$precision = \frac{true\_positive}{true\_positive + false\_positive} = \frac{0.49}{0.49 + 0.21} = 0.7$$

$$recall = \frac{true\_positive}{true\_positive + false\_negative} = \frac{0.49}{0.49 + 0.21} = 0.7$$

$$F1\ for\ True = \frac{2 * precision * recall}{(presion + recall)} = \frac{2 * 0.7 * 0.7}{0.7 + 0.7} = 0.7$$

For “significant” as “False”, the F1-score can be computed as:

$$True\_positive = 70\% * 70\% = 0.09$$

$$False\_positive = 30\% * 70\% = 0.21$$

$$True\_negative = 30\% * 30\% = 0.49$$

$$False\_negative = 70\% * 30\% = 0.21$$

$$precision = \frac{true\_positive}{true\_positive + false\_positive} = \frac{0.09}{0.09 + 0.21} = 0.3$$

$$recall = \frac{true\_positive}{true\_positive + false\_negative} = \frac{0.09}{0.09 + 0.21} = 0.3$$

$$F1\ for\ False = \frac{2 * precision * recall}{(presion + recall)} = \frac{2 * 0.3 * 0.3}{0.3 + 0.3} = 0.3$$

So the F1 score is:

$$F1\ score = \frac{F1\ for\ True + F1\ for\ False}{2} = \frac{0.7 + 0.3}{2} = 0.5$$

As shown in the above discussion, the dummy classifier does not use any of the features in its prediction. Or say it completely ignores any possible relation between the features and the prediction field. Therefore, it is only a test classifier, and shall not be used for any real prediction. It shall only be used for comparison.

Similarly, if a real machine model fails to gain any relation from the features and the prediction field, it shall have a performance similar to the dummy classifier. This could happen if a user fails to pick any meaningful features. For example, assume that a user picks incident ID as the only feature to predict severity of an incident. Since there might not be any meaningful relation between the incident ID and the severity of any incident, a machine model built like this would not be able to learn any useful knowledge from the data. As a result, the best the machine model can do in this case is to predict like the dummy classifier. Therefore, the dummy classifier sets the baseline for other real (useful) classifiers.

To get the base line using Dummy Classifier, set the algorithm field in the ml.config file.

Example:

```
algorithm=Dummy Classifier
```

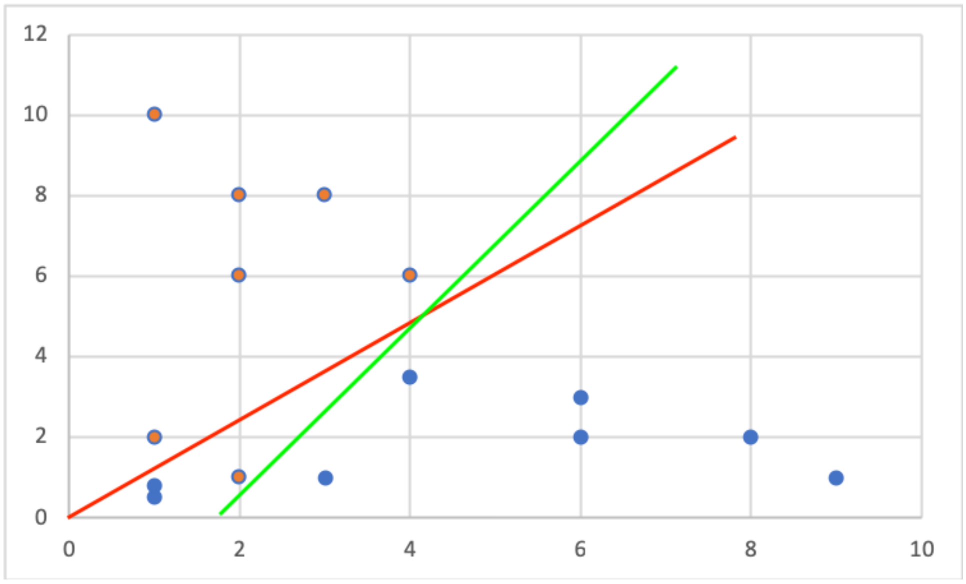
In practice, the users can first set the algorithm to “Dummy Classifier”, and build a model. Then they can change the algorithm to another real algorithm to build another real model. Comparing the performances from the two models can show whether the real model is able to learn any useful knowledge from the data.

7. Supported Algorithms

The res-ml tool supports the following machine learning algorithms. *For advanced users*, more information is provided for each of the algorithms below.

7.1. Logistic Regression

Logistic regression is a powerful algorithm for classifying good performance on linearly separable classes. It is based on the Perceptron rule first published by Frank Rosenblatt in 1957. The original Perceptron however has a disadvantage. The basic idea of Perceptron is that it implements a loop to seek a linear boundary that can perfectly separate the samples into proper classes.



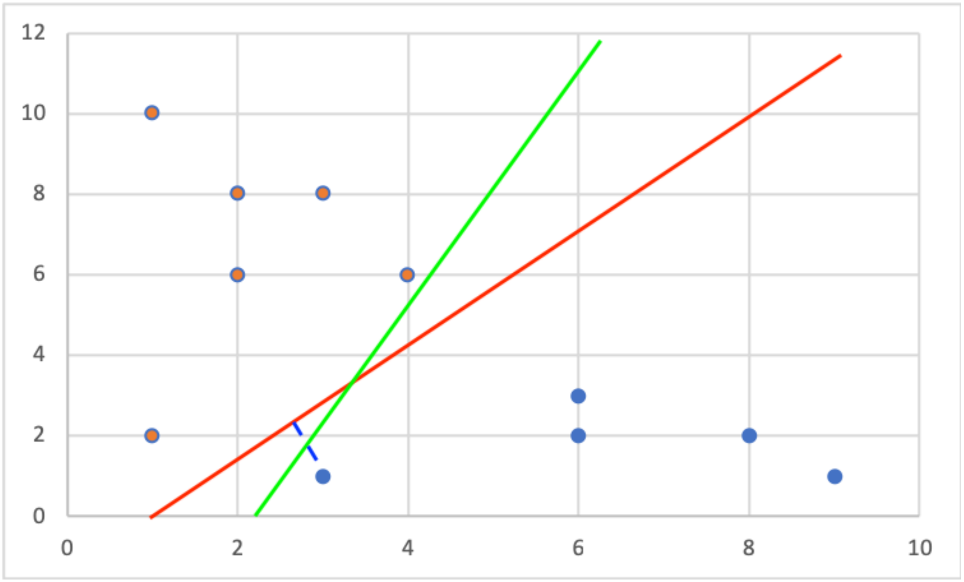
When the samples are not linear separable as shown in the above graph, a perceptron never stops.

For Logistic Regression, instead of looking for the perfect solution that can classify all the samples perfectly, it seeks a solution that can minimize a cost function. In other words, it stops when it finds the best solution, which is not necessarily the perfect one.

Please refer to this link for more details about the cost function of Logistic Regression.

7.2. Support Vector Machine (SVM)

SVM is an extension of the classic perceptron. Instead of minimizing a cost function like Logistic Regression, SVM seeks to maximize the margin.



For a training dataset like the one shown above, both the red line and green line can classify the samples perfectly. SVM looks at the samples that are closest to the boundary, and seeks to maximize the distance. This distance is defined as the margin and shown as the blue dotted line in the graph above. Thus the red line is better than the green line according to the logic of SVM.

An SVM model, in general, has lower generalization error.

7.3. SVM with Gaussian Kernel

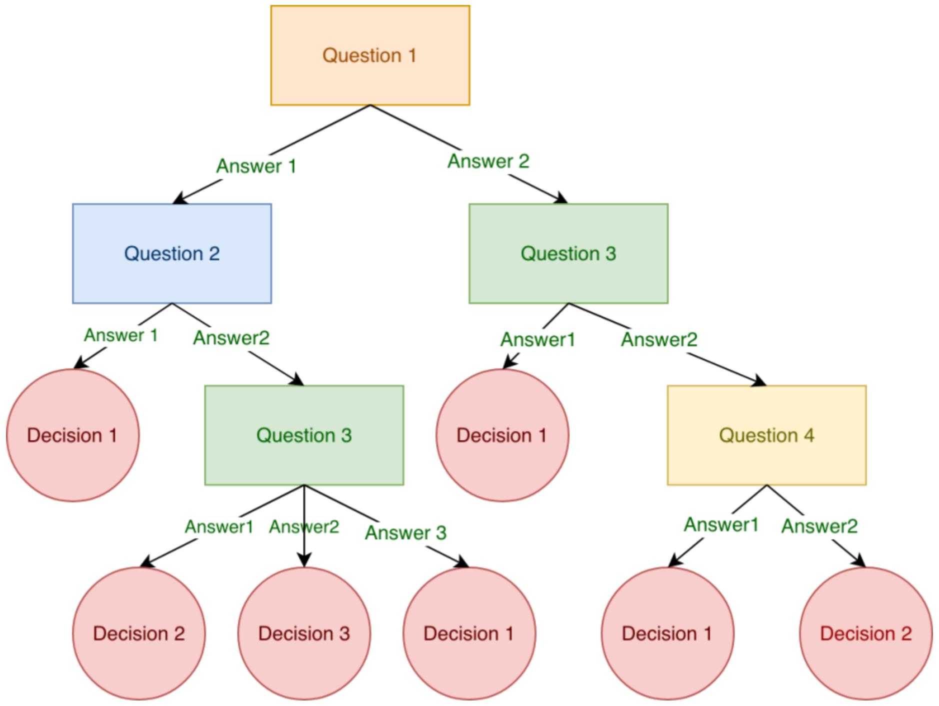
SVM with Gaussian kernel is an extension of SVM for solving non-linear problem. Both Logistics Regression and SVM mentioned above are for linear problems. Their solution is a straight line in a two dimensional plane, or a hyperplane in a multi-dimensional space. But in reality, samples might not be linearly separable.

One trick to handle non-linear separable samples is to map them into higher dimensional feature space, hoping that they are linearly separable in new space. A kernel function is an easy way to do this mapping efficiently.

One popular kernel function is called Radial Basis Function (RBF). It is also referred to as the Gaussian kernel.

7.4. Decision Tree

A Decision Tree algorithm builds a tree like the one shown below. When it is used to predict, it seeks answers to questions below by looking at the input features. The decision at the end determine the class.



Thus the key point of building a decision tree is to find the right questions to ask. This is done by maximizing the Information Gain at each node of the tree.

For more information of Decision Tree, please refer to this link.

7.5. Random Forest

Random Forest is built by combining multiple decision trees. The key concept of random forest is to improve the generalization performance by averaging multiple decision trees.

The basic idea of Random Forest includes the following steps. As mentioned above, multiple decision trees are built. For each decision tree, first randomly pick *n* samples from the training set with replacement. The "with replacement" means that the same samples can be used in more than one training tree. Then a number of features, depending on how Random Forest is configured, are selected without replacement. After that, this decision tree is built like a normal decision tree. At the end, the prediction results from multiple decision trees are aggregated by using majority vote.

Please refer to this link for more information.

7.6. Naïve Byes

Naïve Byes algorithms are based on the Bayes' theorem. It provides a way to compute conditional probability.

Conditional probability is defined as:

$$P(e|c)$$

It means the probability of event *e* when condition *c* is true. Using this definition, Bayes' theorem can be written as:

$$P(e|c) = \frac{P(e|h) * P(h)}{P(c)}$$

Basically a Naïve Bayes algorithm computes P(e|c) based on P(e|h), P(h) and P(c). Here P(e|c) determines how the algorithm predicts. The condition *h* here represents different features. The algorithm seeks the best combination of features to maximize P(e|c).

7.6.1. Gaussian Naïve Bayes

Gaussian Naïve Bayes is an extension of the original Naïve Bayes. Here the real-valued features are assumed to fit Gaussian distribution. For example, if one of the features, *x*, picked by the user is a continuous float number, then this algorithm segments the range of this feature by the class. When it predicts, it uses the Gaussian distribution to compute the conditional probability for a given value, x=v, given a class C_k:

$$p(x = v\ |C_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-\frac{(v-\mu_k)^2}{2\sigma_k^2}}$$

Note here, σ is the standard deviation and μ is the mean of the value.

7.6.2. Bernoulli Naïve Bayes

Bernoulli Naïve Bayes is another extension of the original Naïve Bayes. It is good for features that are independent Booleans. Thus in practice, this algorithm is a popular one for classifying documents.

Here the conditional probability is given by:

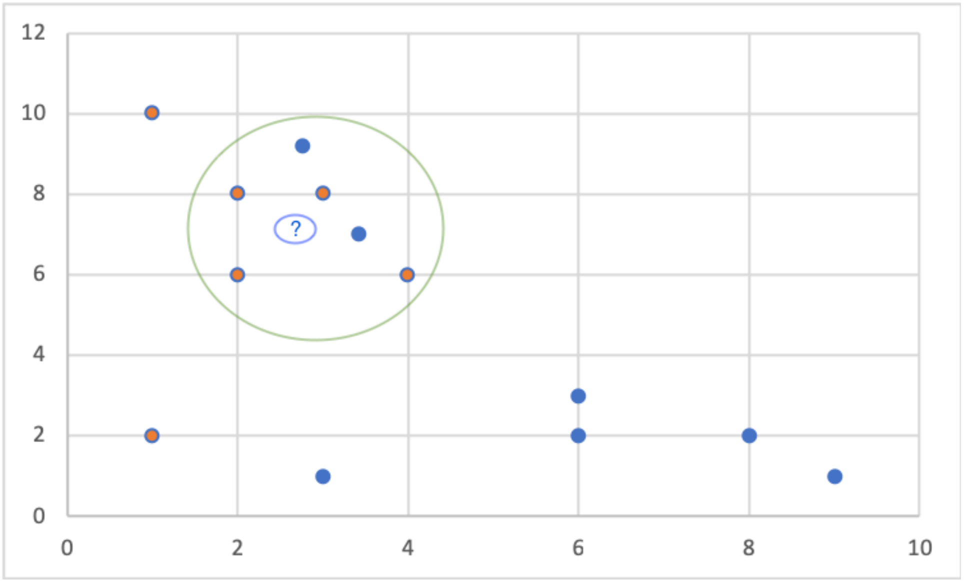
$$p(x|C_k) = \prod_{i=1}^{n} p_{ki}^{x_i}(1 - p_{ki})^{(1-x_i)}$$

To learn more about Naïve Bayes, please refer to this link.

7.7. K-Nearest Neighbor

K-Nearest Neighbor (KNN) algorithm takes a fundamentally different approach. It is also an example of lazy learner, in the sense that it just memorizes the whole training dataset when the model is built.

When a KNN model is used to predict an input, it looks for k samples nearest to the input data, and then uses majority vote to make a prediction. This can be shown in the following graph. The input data is represented by a blue circle with a question mark. The KNN looks for the 6 samples closest to this blue circle. Here we assume that k=6. Those 6 samples are shown within the green circle. There are 4 red dots and 2 blue dots. Based on this, the KNN model predicts the new input data to be a red dot.



The advantage of KNN is that it can easily take new data into its collection and make decision with an updated collection. But the disadvantages are the required computation power and storage grows quickly when the size of the training dataset grows bigger.

8. Supported Ensemble Methods

The reason that an ensemble method can in general boost the performance can be explained in the following simple example.

Assume that we ensemble three classifiers to predict True and False. Each of the classifier has an overall accuracy of 60%. When this is validated by using the testing dataset, an individual classifier has 60% chance of predicting a sample correctly. What happens to the ensemble model? Assume further that each individual classifier is making independent decision. Then we have the following cases:

1. All three classifiers are able to make correct prediction. The ensemble model will be able to make a correct prediction for this case. The probability of this case is:

$$P_3 = C_3^3 * 60\% * 60\% * 60\% = 21.6\%$$

2. Any two of the three classifiers are able to make a correct prediction. The ensemble model will also able to predict correctly since majority vote is used to aggregate the results. The probability of this case is:

$$P_2 = C_3^2 * 60\% * 60\% * 40\% = 43.2\%$$

Therefore, the overall accuracy of this ensemble mode is:

$$P = P_3 + P_2 = 21.6\% + 43.2\% = 64.8\%$$

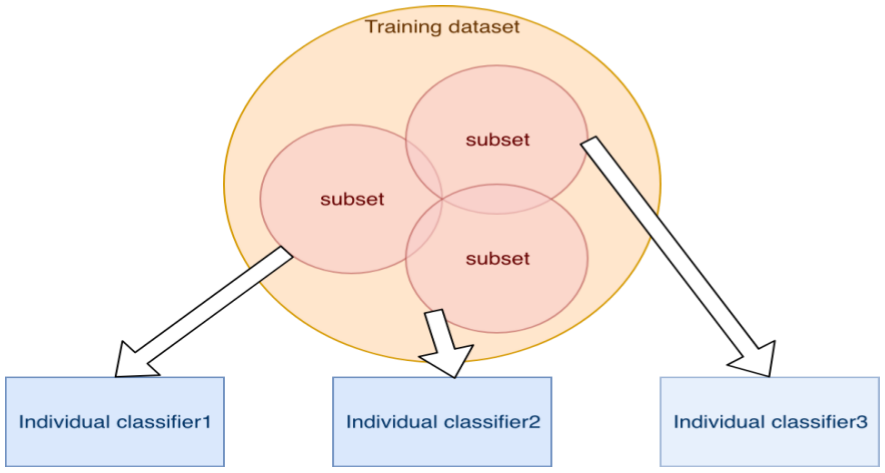
This overall accuracy is higher than the accuracy of 60% for each individual classifier.

It can be proved that, theoretically, as long as each individual classifier has a better than 50% accuracy, an ensemble model that aggregates multiple classifiers can give a better performance. Of course, if each individual classifier has a perfect 100% accuracy, then the ensemble model has a perfect accuracy as well.

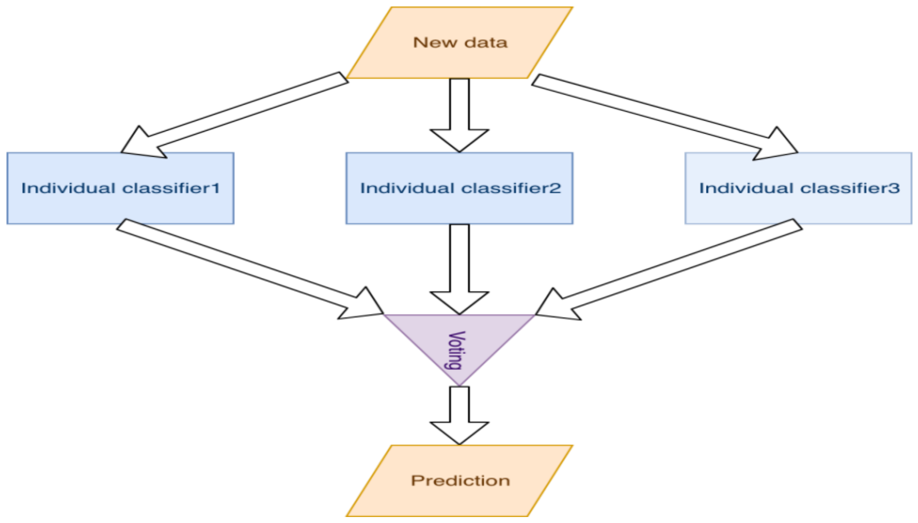
There are different ensemble methodsThe following sections describes those supported by the res-ml tool.

8.1. Bagging

Bagging was proposed by Leo Breiman in 1994. To build an ensemble model using Bagging, each individual classifier is fitted with a random subset of the training dataset. Since each classifier picks its subset by replacement, a certain portion of data duplicates of the training dataset duplicate in the dataset used by different classifiers.



The prediction from individual classifiers are aggregated at the end by majority vote.



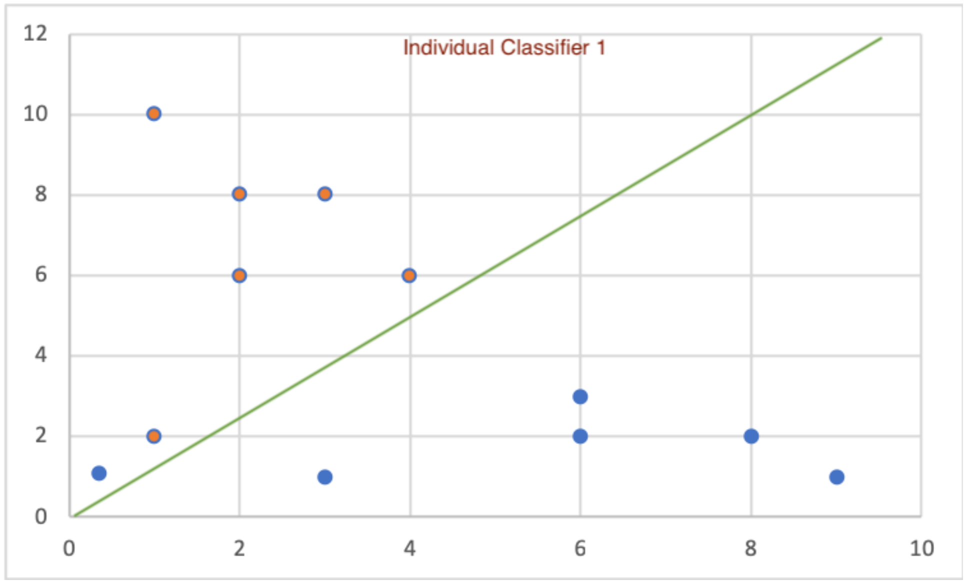
Note that random forest is a special case of bagging.

8.2. Adaptive Boost

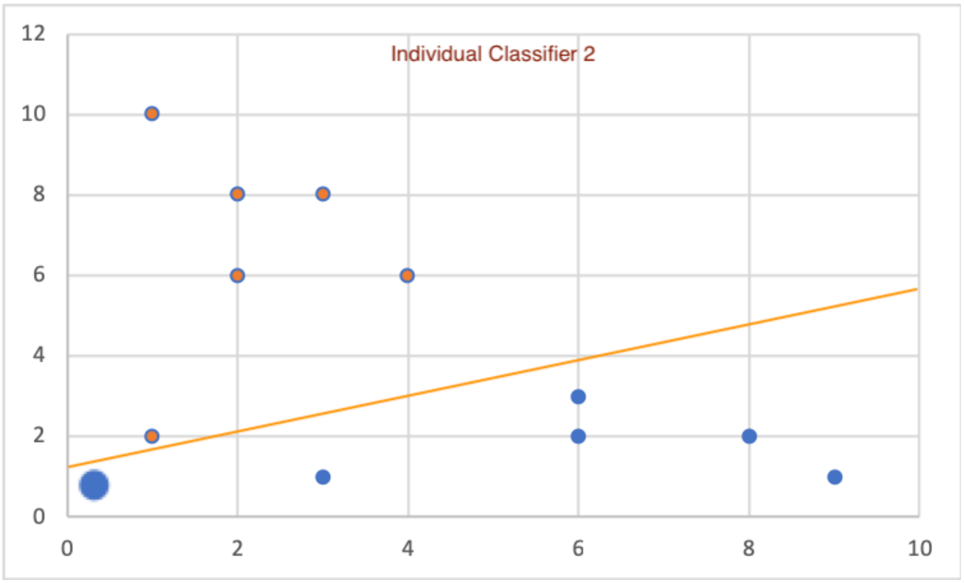
The other ensemble method supported by the res-ml tool is Adaptive Boost, which is also called AdaBoost. AdaBoost is designed to boost performance of weak learners. A weak learner refers to a very simple base classifier that can barely over-perform the Dummy Classifier, which uses random guessing essentially.

Being adaptive means that the individual classifiers are trained and tested one by one, instead of all at the same time. The training of a latter one can then learn from the former ones. Here is how it works.

The first individual classifier is trained by using the whole training dataset as shown in the following graph.



Then all the misclassified samples in the training dataset are identified. In the above example,, there is one blue dot being misclassified. The weight for these samples are increased, and then the whole training dataset with updated weight is fed to the next classifier.



Here we use a bigger dot to represent the increased weight for the blue dot missed by the first classifier. As shown in the discussion of class_weight section, this can force the algorithm to pay more attention to fitting these samples with higher weight properly. In other words, these samples that were considered as errors before will less likely show up as errors this time. For the second individual classifier, the decision boundary represented by the orange is thus different from the one of the first individual classifier. The second individual classifier misclassifies some samples. Then the weight of those samples are increased before being fed to the third individual classifier. This process is repeated for all the remining classifiers. Note that even though the latter ones are improvements of the former ones, all the classifiers are kept. The prediction of the ensemble model uses all classifiers.

To predict, the input data is fed into each classifier trained above, and the predictions from them are aggregated by majority vote.

9. Limitations

The following lists the limitations of the Machine Learning Function:

- Additional features

  Only incident fields (built-in or custom) can be selected as prediction field or features. Fields from artifacts or tasks are not supported.

- Regression support

  This integration supports classification only. Regression prediction is not supported.

- NLP

  Free-text inputs like description are not supported. Natural Language Processing is needed in order to handle free-text inputs.

10. Troubleshooting

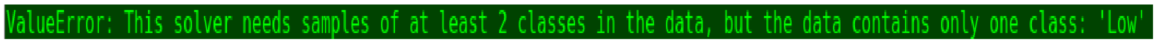
These are some possible errors users might encounter.

10.1. Single Class Error

This error happens when all the samples carry the same value for the prediction field. For example, if the prediction field is a custom field “urgency”, and all incidents have the same value, such as “Low” as shown in the following table

Incident_id	Customer	Urgency	Severity
1001	ABC	Low	High
1002	EFG	Low	High
1003	HIJ	Low	Medium
1004	KLM	Low	Medium
1005	XYZ	Low	Medium
1006	DEF	Low	Low

If the users select Urgency as the prediction field, and set algorithm to Logistic Regression, then the output of the build command looks like this:

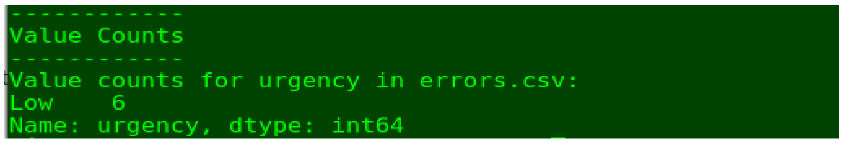


```
ValueError: This solver needs samples of at least 2 classes in the data, but the data contains only one class: 'Low'
```

This means all the samples given in the above table has “Urgency” as “Low”. Basically a machine model does not need to do anything but just predict “Low’ all the time. Some algorithms like Decision Tree does just that, others like Logistic Regression shows an error message like the one above.

Users can use the count_value subcommand of the res-ml tool to detect this.

Sample output:



```
------------
Value Counts
------------
Value counts for urgency in errors.csv:
Low    6
Name: urgency, dtype: int64
```

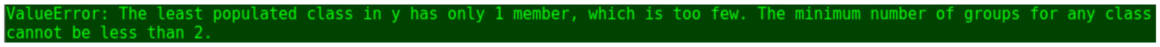
The output shows that all six samples carry the same value “Low”.

The resolution for this error is to wait for more data, because at this point there is nothing for a machine learning model to learn.

10.2. Too Few Samples for a Particular Value

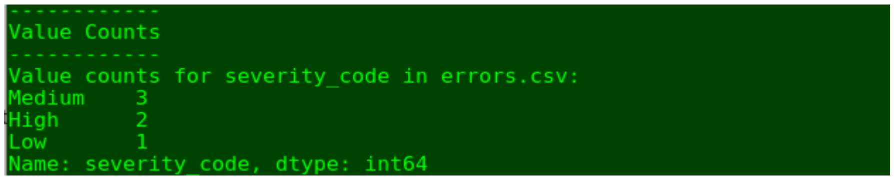
When a machine model is built, it needs to be trained and tested. The input samples are split into two subsets. For each value of the predicted field, the samples are split equally into those two subsets. For example, if the prediction field is “severity”, and there are 100 samples carrying severity value as “High”, then 50 samples of these are put into the training subset and the rest into the testing subset. Therefore, there needs to be at least two samples for each value of the prediction field. This error is caused by prediction value(s) with only one sample. For the samples shown in the table above, if the prediction field is “severity”, then there is only one sample (id = 1006) has “severity” as “Low”.

If a machine model is built using the data above by selecting "severity" as the predicted field, an error message like this is shown:



```
ValueError: The least populated class in y has only 1 member, which is too few. The minimum number of groups for any class
cannot be less than 2.
```

Also, users can use the count_value subcommand of the res-ml tool to detect this:



```
------------
Value Counts
------------
Value counts for severity_code in errors.csv:
Medium    3
High      2
Low       1
Name: severity_code, dtype: int64
```

There are two possible approaches.

One is to remove this value, using the "unwanted_value" feature of ml.config. Note for this approach, the resulted machine learning model will not predict the corresponding value since it never "sees" this value in the training samples.

If the above approach is not desired, then users need to wait for more data.