Table of Contents

Introduction to the ML-SPL API........................................................................1
   About the ML-SPL API................................................................................1

Develop and package a custom algorithm........................................................3
   Add a custom algorithm to the Machine Learning Toolkit overview..........3
   Register an algorithm in the Machine Learning Toolkit..........................3
   Write a Python algorithm class...............................................................4
   Custom algorithm template......................................................................6
   Running process and method calling conventions....................................9
   Using codecs............................................................................................12
   Package an algorithm for Splunkbase....................................................21

Custom algorithm examples.........................................................................26
   Correlation Matrix example......................................................................26
   Agglomerative Clustering example..........................................................31
   Support Vector Regressor example..........................................................40
   Savitzky-Golay Filter example..................................................................43

Additional resources....................................................................................47
   Custom algorithms and PSC libraries version dependencies..................47
   Learn more about the Machine Learning Toolkit.................................48

Troubleshooting............................................................................................49
   Create user facing messages....................................................................49
   Use custom logging..................................................................................50
   Support for the ML-SPL API.....................................................................50

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 the Algorithms section in the Splunk Machine Learning Toolkit User Guide.

You can also extend the Splunk 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 Splunk 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 that other Splunk Machine Learning Toolkit users can use it.

Coding is required to add a custom algorithm to the Splunk Machine Learning Toolkit. Being an advanced Python user or having development experience is 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 uses 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.

Cloud 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. Cloud 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 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 the Algorithms section in the Splunk Machine Learning Toolkit 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.
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.

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

**Arguments**

Specify data and options as arguments.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>options</td>
<td>Options include:</td>
</tr>
<tr>
<td></td>
<td>- <code>args</code> (list): A list of the fields used.</td>
</tr>
<tr>
<td></td>
<td>- <code>params</code> (dict): Any parameters (key-value) pairs in the search.</td>
</tr>
<tr>
<td></td>
<td>- <code>feature_variables</code> (list): The fields to be used as features.</td>
</tr>
<tr>
<td></td>
<td>- <code>target_variable</code> (list): The target field for prediction.</td>
</tr>
<tr>
<td></td>
<td>- <code>algo_name</code> (str): The name of algorithm.</td>
</tr>
<tr>
<td></td>
<td>- <code>mlspl_limits</code> (dict): mlspl.conf stanza properties that may be used in utility methods.</td>
</tr>
</tbody>
</table>

Other options 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).

df

A pandas DataFrame of the input data from the search results. See

Example:

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

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

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

Custom algorithm template

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

**BaseAlgo class**

From base import BaseAlgo.

```python
class CustomAlgoTemplate(BaseAlgo):
    def __init__(self, options):
```

6
# 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

@register

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 BaseAlgorithm(object):
    """The BaseAlgorithm 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 BaseAlgorithm class is not
    required - however, doing so will ensure that the algorithm implements the
    required methods or that this 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
        ML-SPL search query. For a simple query such as:
        | fit LinearRegression sepall_width from petal* fit_intercept=
        The 'options' returned will be:
        |
        | \{'args\': \[u'sepall_width', u'petal'\],
        | \'params\': \{u'fit_intercept': u't\},
        | \'feature_variables\': ['petal'],
        | \'target_variable\': ['sepall_width']
        |
        | algo_name: u'LinearRegression',
        | nMspl_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
        - nMspl_limits (dict): nMspl.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 file ("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 that these feature variables in the 'options'
        have not been wildcard searched 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.'.format(self.class_name)
        raise MLSPLOnImplementedError(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.
    Any data _ith_ 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

        If the algorithm necessarily makes predictions while fitting, return
        the output DataFrame here. Additionally, if the algorithm cannot be
        saved, make predictions and return them. Otherwise, make predictions in
        the apply method and do not return anything here.

        The 'df' argument is a pandas DataFrame from the search results. Note
        that modification to 'df' within this method will also modify the
        DataFrame to be used in the subsequent apply method.

        The 'options' argument is the same as those described in the _init_
        method.
    """

        msg = 'The {} algorithm does not support fit.'.format(self.class_name)
        raise MLSPLOnImplementedError(msg)

def partial_fit(self, df, options):
    """The partial_fit method updates a model incrementally.
    partial_fit is used in the fit command when partial_fit is added to the
    search. It is for incrementally updating an algorithm. If the algorithm
does not require a full dataset in order to update, partial_fit can
be used to update the estimator with each "chunk" of data, rather than
waiting for the full dataset to arrive.

        On the initial partial_fit, the 'options' are the same as described in
        the fit method, however, on the subsequent calls - the 'options' from
        the initial fit are used.

        The 'df' argument is a pandas DataFrame from the search results.
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`).

![Flowchart of fit command process](image)
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.

```
[Diagram of apply command process]
```

**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.

```
[Diagram of summary command process]
```
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()`.

```python
__builtin__.object
__builtin__.slice
__builtin__.set
__builtin__.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:

```python
@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:

```python
>>> 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:

```python
{'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),}
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**

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

from sklearn.neighbors import KNeighborsClassifier
from codec import codecs_manager
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'],
        aliases={'k': 'n_neighbors'}
    )
    self.estimator = 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',
                              SimpleObjectCodec)
```

15
In this case, the SimpleObjectCodec is not sufficient. When you run command ...
| fit KNClassifier into my_model, it will return an error like this:

```
Investigate an object for a custom codec
```

The error message indicated that part of the model sklearn.neighbors.kd_tree.KDTree is not serializable. You can investigate the object in Python terminal:

```python
>>> from sklearn.datasets import load_iris
>>> from sklearn.neighbors import KNeighborsClassifier

>>> iris = load_iris()
>>> X = iris.data
>>> y = iris.target
>>> classifier = KNeighborsClassifier()

>>> classifier.fit(X, y)
>>> classifier._dict__
```

which gives us back:

```python
{'_fit_X': array([[ 5.1,  3.5,  1.4,  0.2],
  ...,
  [ 5.9,  3. ,  5.1,  1.8]],
'_fit_method': 'kd_tree',
'_tree': <sklearn.neighbors.kd_tree.KDTree at 0x7ffe07902500>,
'_y': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  ...,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]),
'algebra': 'auto',
'classes_': array([0, 1, 2]),
'effective_metric_': 'euclidean',
'effective_metric_params_': {},
'leaf_size': 30,
'metric': 'minkowski',
'metric_params': None,'n_jobs': 1,
'n_neighbors': 5,
'outputs_2d_': False,
'p': 2,
'radius': None,
'weights': 'uniform'}
```
In this case, '_tree': <sklearn.neighbors.kd_tree.KDTree at 0x7ffe07902500> is not an object SimpleObjectCodec can encode or decode.

You have two options to deal with this.

**Option 1: Avoid writing the codec by limiting the algorithm choice**

A simple and quick solution, and a way to avoid writing a custom codec, is to add a parameter to the estimator to avoid using a KDTree:

```python
self.estimator = KNeighborsClassifier(algorithm='brute', **out_params)
```

**Option 2: Write a Custom Codec**

If you must use a codec, you can save the KDTree state and reconstruct it using a custom codec. In Python terminal, run:

```python
>>> kdtree_in_memory = classifier.__dict__['_tree']
>>> kdtree_in_memory.__getstate__()
```

which prints the state of "_tree" in classifier:

```python
(array([[ 5.1,  3.5,  1.4,  0.2],
           ...[
           [ 5.9,  3. ,  5.1,  1.8]],
          array([[ 2,  13,  14,  16,  22,  35,  36,  38,  40,  41,  42,  49,  12,  
           ...[
           143, 144, 145, 107, 120, 102, 122]]),
         array([((0, 150, 0, 10.296358579614444), (0, 75, 0, 3.5263295365010903),
                (75, 150, 0, 4.506106967216822), (0, 37, 1, 0.8774964387392121),
                (37, 75, 1, 3.0364452901377956), (75, 112, 1, 3.0401480227120525),
                (112, 150, 1, 2.874456470360963)],
             dtype=[('idx_start', '<i8'), ('idx_end', '<i8'), ('is_leaf', '<i8'), ('radius', '<f8')]),
         array([[ 4.3,  2. ,  1. ,  0.1],
                ...[
                [ 7.9,  3.8,  6.9,  2.5]]),
         30,
         3,
         7,
         0,
         0,
         0,
         <sklearn.neighbors.dist_metrics.EuclideanDistance at 0x10d94d320>)
```
Most of the objects are numbers and arrays, which are covered by Python built-in and pre-registered codecs. At the end of the printed state, there is a second embedded object that is not supported by Python build-in or pre-registered codecs:

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

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

```python
>>> dist_metric = kd_tree_in_memory.__getstate__()[0][-1]
>>> 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()`

```python
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']
```

18
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 {
            '__mlsptl_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)
codes_manager.add_codec('sklearn.neighbors.kd_tree', 'KDTree',
KDTreeCodec)
codes_manager.add_codec('sklearn.neighbors.dist_metrics',
'EuclideanDistance', EuclideanDistanceCodec)

**Complete example**

**KNClassifier.py**

```python
#!/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.estimator = 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)
```

20
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 Working with Splunkbase.
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.

<table>
<thead>
<tr>
<th>Required Field</th>
<th>Example User Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>application name</td>
</tr>
<tr>
<td>Folder name</td>
<td>application name</td>
</tr>
<tr>
<td>Template</td>
<td>barebones</td>
</tr>
</tbody>
</table>

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**:
   ```python
   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**

**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:

   ```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:

   ```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 algs.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 algs.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 ' + ', '.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

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_samples
   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.

```python
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:
```

33
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.

```python
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)
```

34
# 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**

```python
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: ' + ', '.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: ' + ', '.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:

      ```bash
      $ 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`.

      ```python
      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.

```python
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 {
            '__mlspl_type': [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:

```python
@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)
```

42
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:

      ```bash
      $ 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`.

   ```python
   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.

   ```python
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.

   ```python
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:
          ```
5. Define the `fit` method.

```python
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**

```python
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 4.4.1 of the MLTK requires Splunk Enterprise 7.0 or later or Splunk Cloud. The Splunk Machine Learning Toolkit requires the Python for Scientific Computing (PSC) add-on. Upgrading to version 3.4.0 or above (4.0.0, 4.1.0, 4.2.0, 4.3.0, and 4.4.0 currently) of the toolkit requires upgrading to version 1.3 of the Python for Scientific Computing add-on.

If you have written any custom algorithms that rely on the PSC libraries, upgrading to version 1.3 or 1.4 the PSC library 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.

Choose to upgrade to version 1.4 of the Python for Scientific Computing add-on to access all the features in version 4.3 and above of the toolkit.

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

You cannot access new features in the MLTK without upgrading to the latest version of the toolkit. Versions 3.4.0 and above of the toolkit require upgrading to versions 1.3 or 1.4 of the PSC add-on. See the version dependencies table for the specific requirements between toolkit and PSC add-on versions.

**Specific version dependencies**

<table>
<thead>
<tr>
<th>MLTK Version</th>
<th>PSC Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.1</td>
<td>1.3 or 1.4</td>
</tr>
<tr>
<td>4.4.0</td>
<td>1.3 or 1.4</td>
</tr>
<tr>
<td>4.3.0</td>
<td>1.3 or 1.4</td>
</tr>
<tr>
<td>4.2.0</td>
<td>1.3 or 1.4</td>
</tr>
<tr>
<td>4.1.0</td>
<td>1.3</td>
</tr>
<tr>
<td>4.0.0</td>
<td>1.3</td>
</tr>
<tr>
<td>3.4.0</td>
<td>1.3</td>
</tr>
</tbody>
</table>
Any algorithms that have been imported from the Python for Scientific Computing 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.

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 About vetting for Splunk Cloud.
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:

```python
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:

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

This code produces the search warning in the following example:

![Search warning example](image)

You can similarly produce error messages:

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

This code produces the error message in the following example:

![Error example](image)
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:

```python
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:

```
```

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.

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

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

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

<table>
<thead>
<tr>
<th>MLTK Version</th>
<th>PSC Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>1.3 or 1.4</td>
</tr>
<tr>
<td>4.2</td>
<td>1.3 or 1.4</td>
</tr>
<tr>
<td>4.1</td>
<td>1.3</td>
</tr>
<tr>
<td>4.0</td>
<td>1.3</td>
</tr>
<tr>
<td>3.4</td>
<td>1.3</td>
</tr>
<tr>
<td>3.3</td>
<td>1.2 or 1.3</td>
</tr>
<tr>
<td>3.2</td>
<td>1.2 or 1.3</td>
</tr>
<tr>
<td>3.1</td>
<td>1.2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>MLTK Version</th>
<th>PSC Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>1.3 or 1.4</td>
</tr>
<tr>
<td>4.1</td>
<td>1.3</td>
</tr>
<tr>
<td>4.0</td>
<td>1.3</td>
</tr>
<tr>
<td>3.4</td>
<td>1.3</td>
</tr>
<tr>
<td>3.3</td>
<td>1.2 or 1.3</td>
</tr>
<tr>
<td>3.2</td>
<td>1.2 or 1.3</td>
</tr>
<tr>
<td>3.1</td>
<td>1.2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>MLTK Version</th>
<th>PSC Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>1.3</td>
</tr>
<tr>
<td>4.0</td>
<td>1.3</td>
</tr>
<tr>
<td>3.4</td>
<td>1.3</td>
</tr>
<tr>
<td>3.3</td>
<td>1.2 or 1.3</td>
</tr>
<tr>
<td>3.2</td>
<td>1.2 or 1.3</td>
</tr>
<tr>
<td>3.1</td>
<td>1.2</td>
</tr>
</tbody>
</table>
**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**

<table>
<thead>
<tr>
<th>MLTK Version</th>
<th>PSC Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>1.3</td>
</tr>
<tr>
<td>3.4</td>
<td>1.3</td>
</tr>
<tr>
<td>3.3</td>
<td>1.2 or 1.3</td>
</tr>
<tr>
<td>3.2</td>
<td>1.2 or 1.3</td>
</tr>
<tr>
<td>3.1</td>
<td>1.2</td>
</tr>
</tbody>
</table>

**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 \texttt{fit} command) using those algorithms after you upgrade PSC.

**Specific version dependencies**

<table>
<thead>
<tr>
<th>MLTK Version</th>
<th>PSC Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>1.3</td>
</tr>
<tr>
<td>3.3</td>
<td>1.2 or 1.3</td>
</tr>
<tr>
<td>3.2</td>
<td>1.2 or 1.3</td>
</tr>
<tr>
<td>3.1</td>
<td>1.2</td>
</tr>
</tbody>
</table>