Hybrid Methods: Definition

The definition here is quite simple—rather than using a single approach to select feature subsets as the previous methods do, hybrid methods combine the different approaches to get the best possible feature subset.

The way to combine these approaches is up to the engineer, given that you have a lot of methods in your toolbox. You can, for example, start by performing filter methods by eliminating constant, quasi-constant and duplicated features. Then, in the second step, you could use wrapper methods to select the best feature subset from the previous step. This is just one simple, high-level approach.

We’ll explore a few different hybrid methods in this article, but the idea is to combine weaker methods to end up with a more powerful one.

Hybrid Methods: Advantages

The big advantage that hybrid methods offer is that they take the best advantages from other feature selection methods, and as such, can reduce their disadvantages. This can (and hopefully will) result in:

* High performance and accuracy
* Better computational complexity than wrapper methods
* Models that are more flexible and robust against high dimensional data

Machine learning models don’t have to live on servers or in the cloud — they can also live on your smartphone. [And Fritz AI has the tools to easily teach mobile apps to see, hear, sense, and think.](https://www.fritz.ai/product/platform.html?utm_campaign=buildmodels2&utm_source=heartbeat)

Hybrid Methods: Process

The process of creating hybrid feature selection methods depends on what you choose to combine. The main priority is to select the methods you’re going to use, then to follow their processes.

In the following sections, you’ll see how we can combine methods, starting with wrapper approaches since they provide the best feature subset most of the time.

Using Filter & Wrapper methods

In part two of our series, which covered filter methods, we saw ranking methods like mutual information and Chi score, which order features independently without involving any learning algorithm. From there, the best features are selected from the ranking list.

The idea here is to use these ranking methods to generate a feature ranking list in the first step, then use the top k features from this list to perform wrapper methods (like SFS or SBS).

With that, we can reduce the feature space of our dataset using these filter-based rangers in order to improve the time complexity of the wrapper methods.

*You can use the code from the previous articles (filter methods and wrapper methods and combine them in ways that work for your use case.*

Using Embedded & Wrapper Methods

The thought here is to also acquire a ranking of features and then establish wrapper methods to search for the best possible features subset.

If you’ll recall, embedded methods offer a way to establish feature importance. This can be used to select top features and then perform a wrapper methods search.

*You can use the code from the previous articles (embedded methods and wrapper methods) and combine them in ways that work for your use case.*

Another way to use embedded methods is by using what’s called **recursive feature elimination** and **recursive feature addition**, which are illustrated in more detail below.

Machine learning is rapidly moving closer to where data is collected — edge devices. [Subscribe to the Fritz AI Newsletter to learn more about this transition and how it can help scale your business.](https://www.fritz.ai/newsletter?utm_campaign=fritzai-newsletter-scale6&utm_source=heartbeat)

Recursive Feature Elimination

This is just a fancy name for a simple method that works as follows:

1. Train a model on all the data features. This model can be a tree-based model, lasso, logistic regression, or others that can offer feature importance. Evaluate its performance on a suitable metric of your choice.
2. Derive the feature importance to rank features accordingly.
3. Delete the least important feature and re-train the model on the remaining ones.
4. Use the previous evaluation metric to calculate the performance of the resulting model.
5. Now test whether the evaluation metric decreases by an arbitrary threshold (you should define this as well). If it does, that means this feature is important. Otherwise, you can remove it.
6. Repeat steps 3–5 until all features are removed (i.e. evaluated).

You might be thinking to say that this is just like the step backward features selection that we did in our post on wrapper methods, **but it isn’t**. The difference is that SBS eliminates all the features first in order to determine which one is the least important. But here, we’re getting this information from the machine learning model’s derived importance, so it removes the feature only once rather than removing all the features at each step.

That's why this approach is faster than pure wrapper methods and better than pure embedded methods. But as a drawback, the main problem with that is we have to use an arbitrary threshold value to decide whether to keep a feature or not.

As a consequence, the smaller this threshold value, the more features will be included in the subset, and vice versa.

Here’s some sample code that works with a RandomForestClassifier to select the best features:

We’re also using [RFECV](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html)to do recursive feature elimination with cross-validation. Here’s a description of the different parameters:

* min\_feature\_to\_select: As its name suggests, this parameter sets the minimum number of features to select.
* step: How many features we remove at each step.
* cv: An integer, generator, or an iterable that describes the cross-validation splitting strategy.
* scoring: The evaluation metric we use.

You also can use the different code snippets from the previous articles to create your own implementation of RFE. Here’s an example for your reference:

Recursive Feature Addition

With the previous method, we started from all the features and removed one at the time. Now it’s the opposite case—we start with no features and add one feature at the time. Here are the steps:

1. Train a model on all the data and derive the feature importance to rank it accordingly. This model can be a tree-based model, lasso, logistic regression, or others that can offer feature importance.
2. From that initial model, create another with the most important feature and evaluate it with an evaluation metric of your choice.
3. Add another important feature and use it to re-train the model, along with any feature from the previous step.
4. Use the previous evaluation metric to calculate the performance of the resulting model.
5. Now test whether the evaluation metric increases by an arbitrarily-set threshold (you should define this as well). If that’s the case, it means that this feature is important; otherwise, we can remove it.
6. Repeat steps 3–5 until all features are added (i.e. evaluated).

The difference between this method and step forward feature selection is similar to what we discussed in RFE—it doesn’t look for all features first in order to determine which ones to add, so it’s faster than wrapper methods.

sklearn doesn’t provide a RecursiveFeatureAddition algorithm, so you’ll need to implement it on your own:

There’s also [this repository](https://github.com/heberleh/recursive-feature-addition) that provides this functionality, which you may want to check out.