In my previous posts (Part 1 and Part 2), we discussed why feature selection is a great technique for improving your models. By having the model analyze the important signals, we can focus on the right set of attributes for optimization. As a side effect, less attributes also mean that you can train your models faster, making them less complex and easier to understand. Finally, less complex models tend to have a lower risk of overfitting. This means that they are more robust when it comes to creating the predictions for new data points.

We also discussed why a brute force approach to feature selection is not feasible for most data sets and tried multiple heuristics for overcoming the computation problem. We looked at evolutionary algorithms which turned out to be fast enough for most data sets. And they have a higher likelihood to find the optimal attribute subset.

So, we've found the solution then, right? Well, not quite yet.

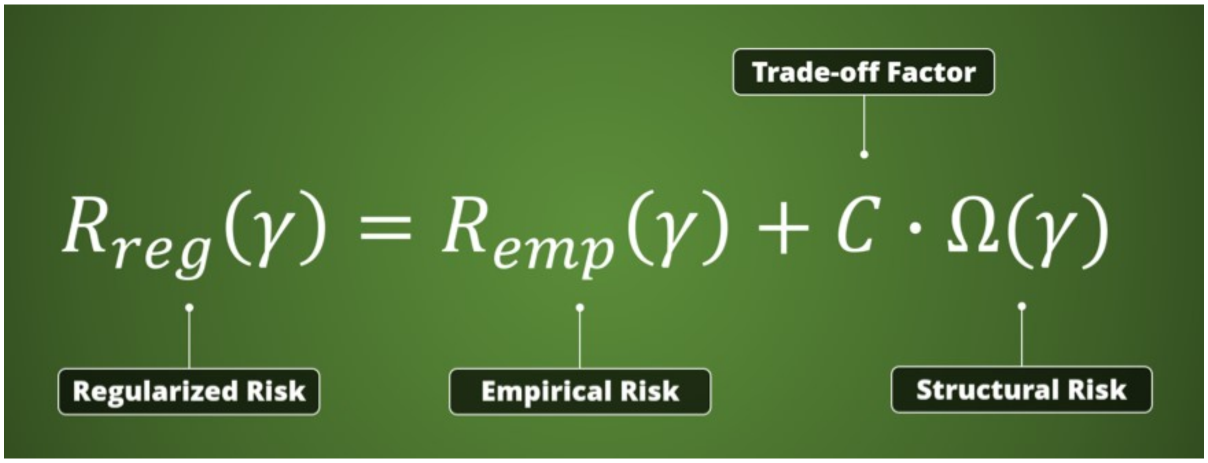
Regularization for Feature Selection

So far, we have been optimizing for model accuracy alone. We know from regular machine learning methods that this is a bad idea. If we only look for the most accurate model on the training data it will lead to overfitting. Our models will perform worse on new data points as a result. Most people only think about overfitting when it comes to the model itself. We should be equally concerned about overfitting when we make any decision in data analysis.

If we decide to merge two data sets, take a sample, or filter down the data, we are making a modeling decision. In fact, a more sophisticated machine learning model could have made this decision on its own. We need to be careful about overfitting and validating all decisions, not just the model itself. This is the reason it does not make sense to separate data preparation from modeling. We need to do both in an integrated fashion, and validate them together.

It doesn’t matter if we do feature selection automatically or manually. Any selection becomes a part of the model. And as such it needs to be validated and controlled for overfitting. Learn more about this in our recent blog series on correct validation.

We need to perform regularization to control overfitting for feature selection, just like we do for any other machine learning method. The idea behind regularization is to penalize complexity when you build models. The concept of regularization is truly at the core of statistical learning. The image below defines *regularized risk* based on the empirical risk and the structural risk. The *empirical risk* is the error we make on the training data. Which is simply the data we use for doing our feature selection. And the *structural risk* is a measurement of complexity. In case of an SVM the structural risk would be a low width of the margin. In case of feature selection it is simply the number of features. The more features, the higher the structural risk. We of course want to minimize both risks, error and complexity, at the same time.



Minimizing the number of features and maximizing the prediction accuracy are conflicting goals. Less features means reduced model complexity. More features mean more accurate models. If we can’t minimize both risks at the same time, we need to define which one is more important. This is necessary so we can decide in cases where we need to sacrifice one for the other.

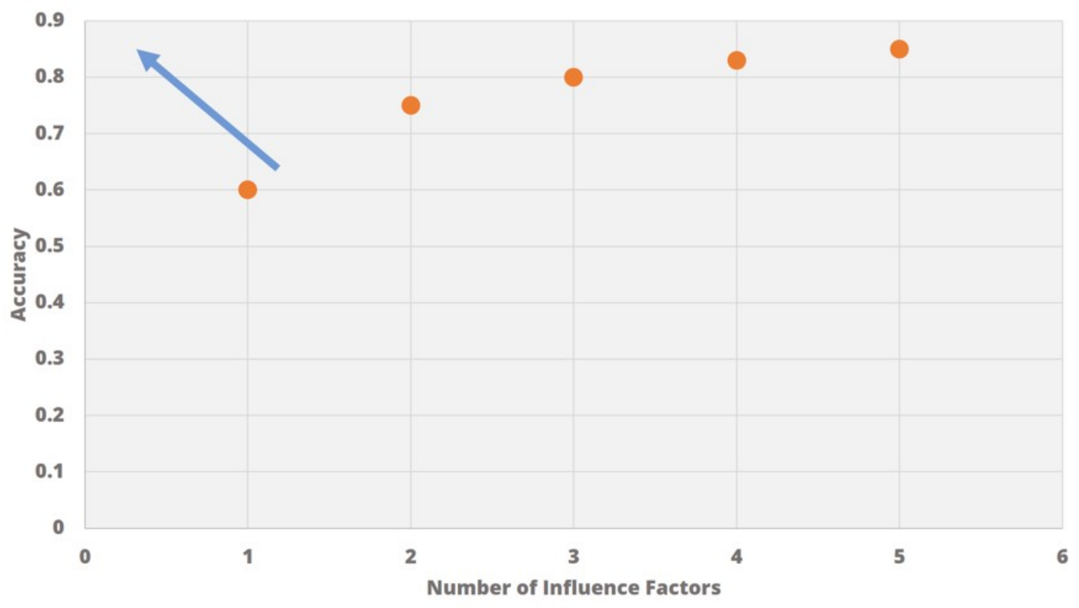
When we make decisions dealing with conflicting objectives, we can introduce a trade-off factor. This is the factor "C" in the formula above. The problem is that we cannot determine C without running multiple experiments. What are the possible accuracy ranges we can reach with our models on the given data set? Do we need 10 or 100 features for this? We can define C without knowing those answers. We could use hold-out data set for testing and then try to optimize for a good value of C, but that takes time.

Wouldn't it be great if we didn't have to deal with this additional parameter "C"? Finding a complete range of potential solutions. Some models would be great for lots of accuracy while others would use as little attributes as possible. And then of course some solutions for the trade-off in between. At this point, we also want to use an automated algorithm in a fast and feasible way.

The good news: this is exactly the key idea behind multi-objective optimization. We will see that we can adapt the evolutionary feature selection from our previous post. Once we do that, our model will deliver all good results rather than a single solution.

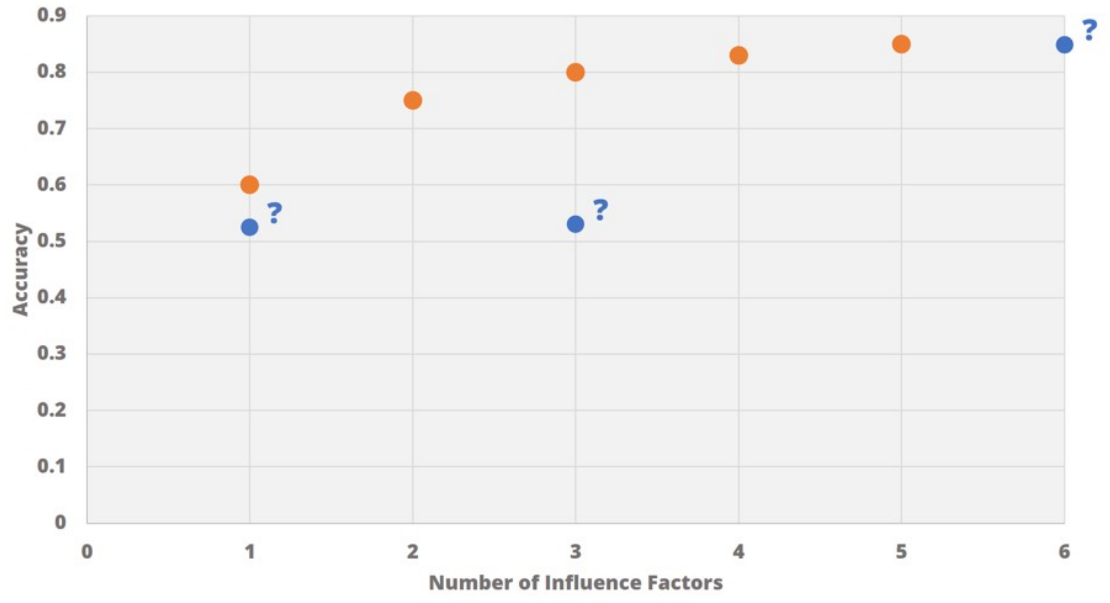
Or… You just optimize for both simultaneously

We want to maximize the accuracy and minimize the number of features at the same time. Let's begin by drawing this solution space so that we can compare different solutions. The result will be an image like the one below. We use the number of features on the x-axis and the achieved accuracy on the y-axis. Each point in this space is now representing a model using a specific feature set. The orange point on the left, for example, represents a model which uses only one feature. And the accuracy of this model is 60% (or "0.6" like in the image).

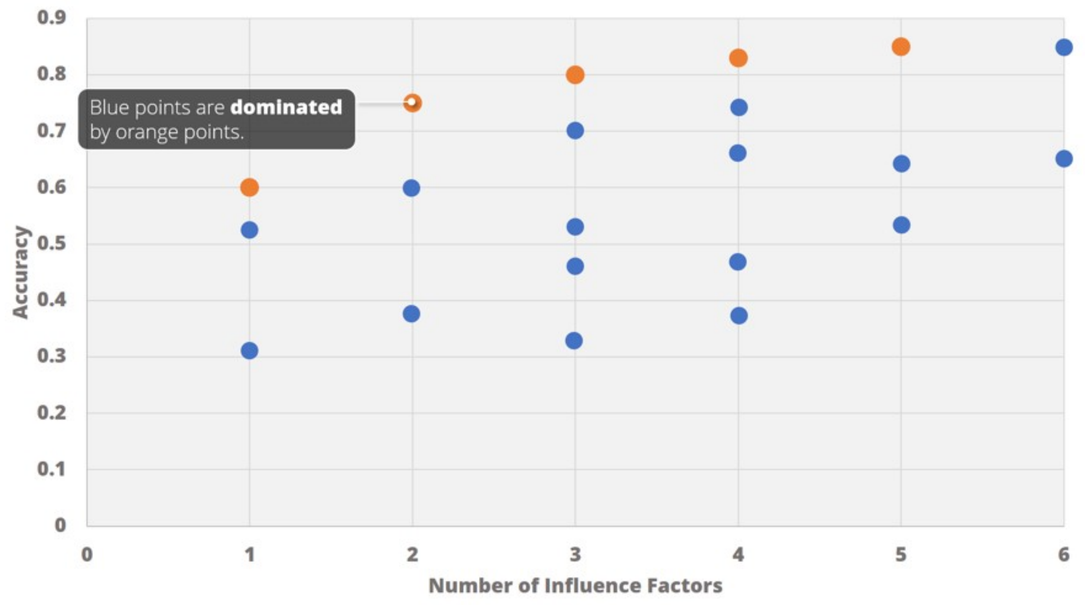


Is the point on the left now better or worse than the other ones? We do not know. Sometimes we prefer less features which makes this a good solution. Sometimes we prefer more accurate models where more features work better. One thing is for sure, we want to find solutions towards the top left corner in this chart. Those are the models run with as little features as possible, and are also the most accurate. This means that we should prefer solutions in this corner over those more towards the bottom right.

Let’s have a look at some examples to make this clearer. In the image below we added three blue points to the orange ones we already had. Are any of those points better than the orange ones? The blue point on the left has only one attribute, which is good. But we have a better model with one attribute: the orange point on top of it. Hence, we prefer the left orange solution over the left blue one. Something similar is true for the blue point on the right. We achieve 85% accuracy, but this is already possible with the solution using only 5 instead of 6 features. We would prefer the less complex model over the right blue point then. The blue point in the middle is even worse: it has less accuracy and more features than necessary. We certainly would prefer any of the orange points over this blue one.

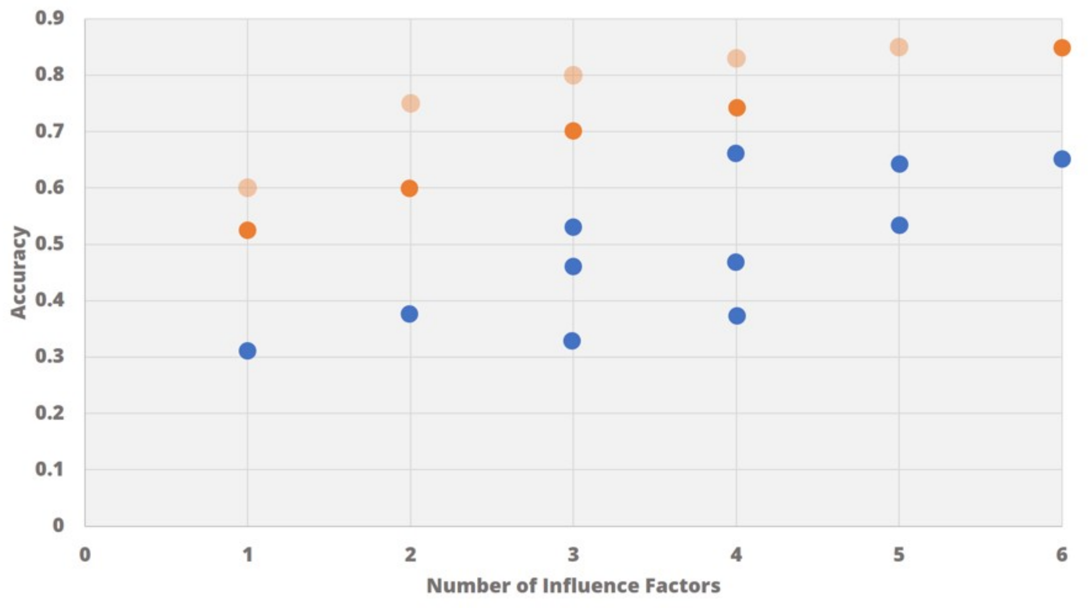


In short, the blue points are clearly inferior to the orange ones. We can say that the orange points *dominate* the blue ones. The image below shows a bigger set of points (blue) which are all dominated by the orange ones:

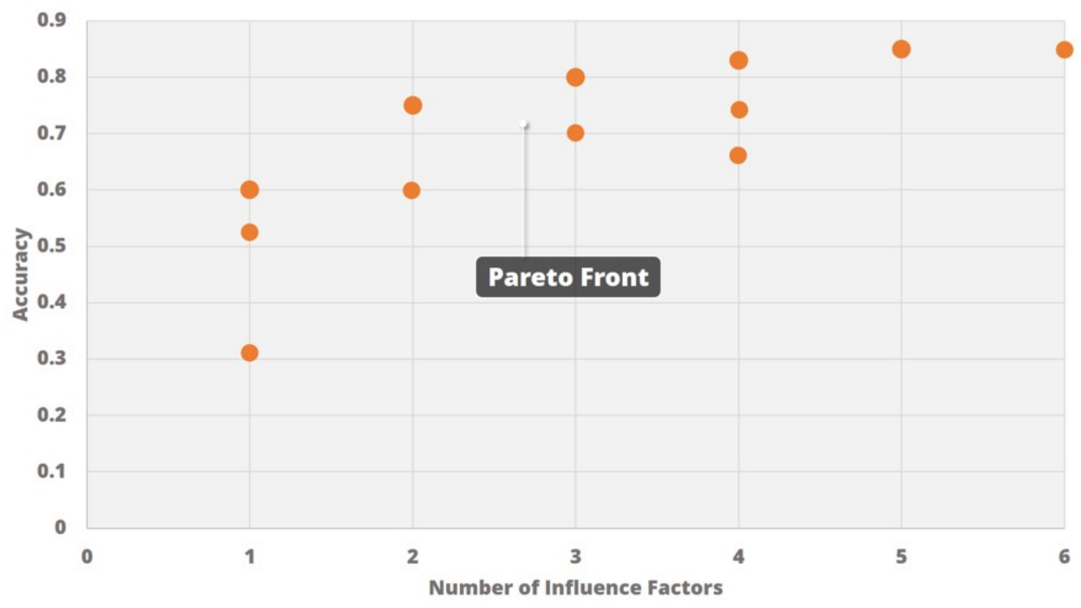


We will now transform those concepts into a new feature selection algorithm. Evolutionary algorithms are among the best when you optimize for multiple conflicting criteria. All we need is a new selection approach for our evolutionary feature selection. This new selection technique will simultaneously optimize for more accuracy and less features.

We call this approach *non-dominated sorting selection*. We can simply replace the single-objective tournament selection with the new one. The image below shows the idea of non-dominated sorting. We start with the first rank of points which are dominating the other points. Those solutions (feature sets in our case) make it into the population of the next generation. They are shown as transparent below. After removing those feature sets, we look for the next rank of dominating points (in solid orange below). We again add those points to the next population as well. We continue this until we reach the desired population size.



The result of such a non-dominated sorting selection is what we call a *Pareto front*. See the image below for an example. Those are the feature sets which dominate all others. Any solution taken from this Pareto front is equally good. Some are more accurate, some are less complex. They describe the trade-off between both conflicting objectives. We will find those solutions without the need of defining a trade-off factor beforehand.



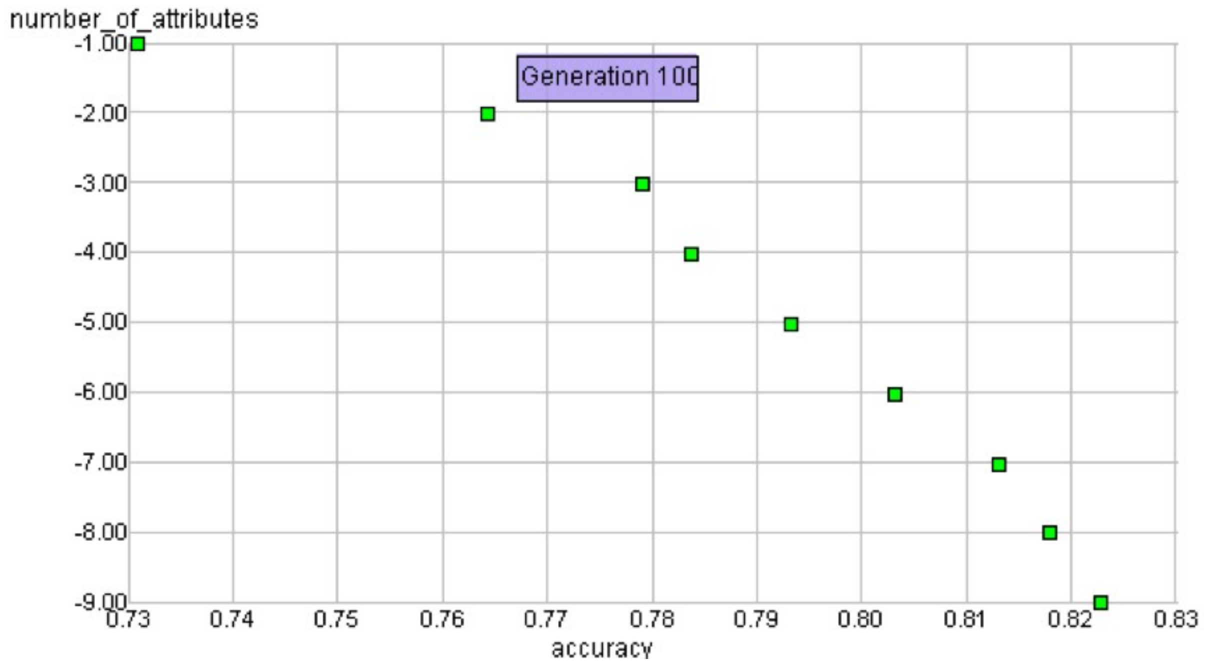
Multi-Objective Feature Selection in Practice

This is one of things which makes multi-objective optimization so great for feature selection. We can find all potentially good solutions without defining a trade-off factor. Even better, we can find all those solutions with a single optimization run. So, it is also a very fast approach. We can inspect the solutions on such a Pareto front. By doing so we can learn from the interactions of features. Some features might be very important in smaller feature sets, but become less important in larger ones. It can happen that interactions of other features become stronger predictors instead. Those are additional insights we can get from the Pareto front. We can see at a glance what the achievable accuracy range is, and what the good range of features is for which we should focus on. Do we need to consider 10 to 20 features or between 100 and 200? This is valuable information in building models.

Let's now run such a multi-objective optimization for feature selection. Luckily we do not need to code all those algorithms. In RapidMiner, we just need to make two little adaptions in the visual workflow. First, we have to change the selection scheme from tournament selection to non-dominated sorting. This is a parameter of the regular evolutionary feature selection operator. Second, we need to add a second performance criterion besides the accuracy. This would be the number of features. Although not necessary, I have also defined some parameters to add some visual output. This will show the movement of the Pareto front during the optimization. And it will also display the details of all feature sets on the Pareto front at the end.

Results

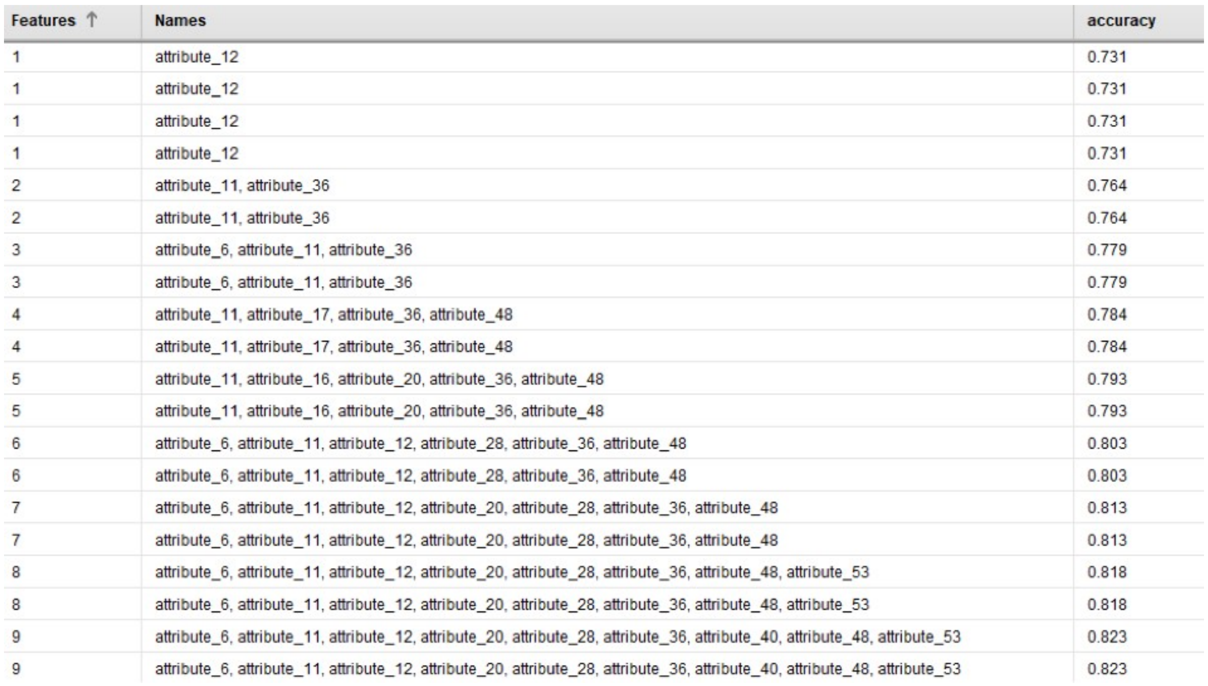
We are going to use the Sonar data set. As a reminder, the attributes represent bands in a frequency spectrum. And the goal is to classify if an object is a rock or a mine. The basic setup is the same. We have now applied the discussed changes to turn this into a multi-objective feature selection. The image below shows the resulting Pareto front of the feature selection:



The resulting Pareto front has 9 different feature sets. They span a range between 1 and 9 attributes. And the accuracy range we can achieve is between 73% and 82%. We show the attribute count as a negative number, simply because RapidMiner always tries to maximize all objectives. Minimizing the number of attributes or maximizing the negative count are the same. This also means that this Pareto front will move to the top right corner, not to the top left like we discussed before.

Here is the full trade-off between complexity and accuracy. We can see that it does not make sense to use more than 9 features. And we should not accept less than 73% accuracy, since that result can already be achieved with one single feature.

It is also interesting to look further into the details of the resulting attribute sets. If we configure the workflow, we end up with a table showing the details of the solutions:



Features ↑	Names	accuracy
1	attribute_12	0.731
1	attribute_12	0.731
1	attribute_12	0.731
1	attribute_12	0.731
2	attribute_11, attribute_36	0.764
2	attribute_11, attribute_36	0.764
3	attribute_6, attribute_11, attribute_36	0.779
3	attribute_6, attribute_11, attribute_36	0.779
4	attribute_11, attribute_17, attribute_36, attribute_48	0.784
4	attribute_11, attribute_17, attribute_36, attribute_48	0.784
5	attribute_11, attribute_16, attribute_20, attribute_36, attribute_48	0.793
5	attribute_11, attribute_16, attribute_20, attribute_36, attribute_48	0.793
6	attribute_6, attribute_11, attribute_12, attribute_28, attribute_36, attribute_48	0.803
6	attribute_6, attribute_11, attribute_12, attribute_28, attribute_36, attribute_48	0.803
7	attribute_6, attribute_11, attribute_12, attribute_20, attribute_28, attribute_36, attribute_48	0.813
7	attribute_6, attribute_11, attribute_12, attribute_20, attribute_28, attribute_36, attribute_48	0.813
8	attribute_6, attribute_11, attribute_12, attribute_20, attribute_28, attribute_36, attribute_48, attribute_53	0.818
8	attribute_6, attribute_11, attribute_12, attribute_20, attribute_28, attribute_36, attribute_48, attribute_53	0.818
9	attribute_6, attribute_11, attribute_12, attribute_20, attribute_28, attribute_36, attribute_40, attribute_48, attribute_53	0.823
9	attribute_6, attribute_11, attribute_12, attribute_20, attribute_28, attribute_36, attribute_40, attribute_48, attribute_53	0.823

We again see the range of attributes (between 1 and 9) and accuracies (between 73% and 82%). We can gain some additional insights if we look into the actual attribute sets. If we only use one single feature, let it be attribute_12. But if we use two features, then attribute_12 is inferior to the combination of attribute_11 and attribute_36. This is another indicator why hill-climbing heuristics like forward selection have such a hard time.

The next attribute we should add is attribute_6. But we should drop it again for the attribute sets with 4 and 5 features in favor of other combinations. This attribute becomes interesting only for the larger sets again.

Finally, we can see that the largest attribute sets consisting of 8 or 9 attributes cover all relevant areas of the frequency spectrum. See the previous post for more details on this.

It is these insights which make multi-objective feature selection the go-to-method for this problem. Seeing good ranges for attribute set sizes or the interactions between features allow us to build better models. But there is more! We can use multi-objective feature selection for unsupervised learning methods like clustering. Stay tuned, we discuss this in the next blog post.

RapidMiner Processes

You can download RapidMiner here. Download the processes below to build this machine learning model yourself in RapidMiner.

- Process: Multi-objective Feature Selection on Sonar

Download the zip-file and extract its contents. The result will be an .rmp file which can be loaded into RapidMiner via “File” -> “Import Process”.