The Basics of Feature Selection

Feature selection can greatly improve your machine learning models. In this blog series, I'll outline all you need to know about feature selection. In Part 1 below I discuss why feature selection is important, and why it's in fact a very hard problem to solve. I'll detail some of the different approaches which are used to solve feature selection today.

Why should we care about Feature Selection?

There is a consensus that feature engineering often has a bigger impact on the quality of a model than the model type or its parameters. Feature selection is a key part of feature engineering, not to mention Kernel functions and hidden layers are performing implicit feature space transformations. Therefore, is feature selection then still relevant in the age of support vector machines (SVMs) and Deep Learning? Yes, absolutely.

First, we can fool even the most complex model types. If we provide enough noise to overshadow the true patterns, it will be hard to find them. The model starts to use the noise patterns of the unnecessary features in those cases. And that means, that it does not perform well. It might even perform worse if it starts to overfit to those patterns and fail on new data points. This is made even easier for a model with many data dimensions. No model type is better than others in this regard. Decision trees can fall into this trap as well as multi-layer neural networks. Removing noisy features can help the model focus on relevant patterns.

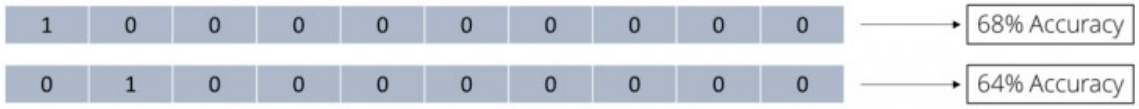
But there are other advantages of feature selection. If we reduce the number of features, models are generally trained much faster. And often the resulting model is simpler and easier to understand. We should always try to make the work easier for we model. Focus on the features which carry the signal over those that are noise and we will have a more robust model.

Why is this a hard problem?

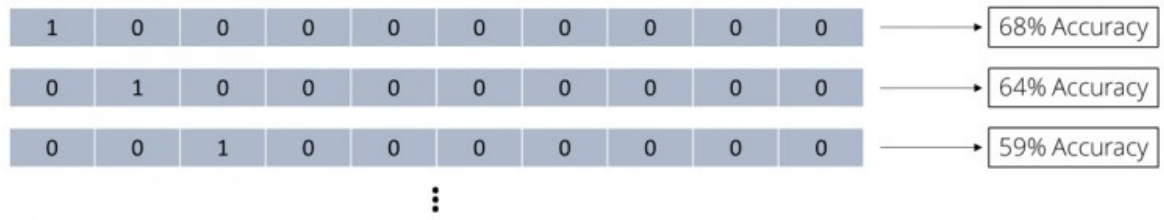
Let's begin with an example. Let's say we have a data set with 10 attributes (features, variables, columns) and one label (target, class). The label column is the one we want to predict. We've trained a model on this data and determined the accuracy of the model built on data is 62%. Can we identify a subset of those 10 attributes where a trained model would be more accurate?

We can depict any subset of 10 attributes as bit vectors, i.e. as a vector of 10 binary numbers 0 or 1. Zero means that the specific attribute is not used, and 1 depicts an attribute which is used for this subset. If we want to indicate that we use all 10 attributes, we would use the vector (1 1 1 1 1 1 1 1 1 1). Feature selection is the search for such a bit vector that produces the optimal accuracy. One possible approach for this would be to try out all the possible combinations. Let's start with using only a single attribute. The first bit vector looks like this:

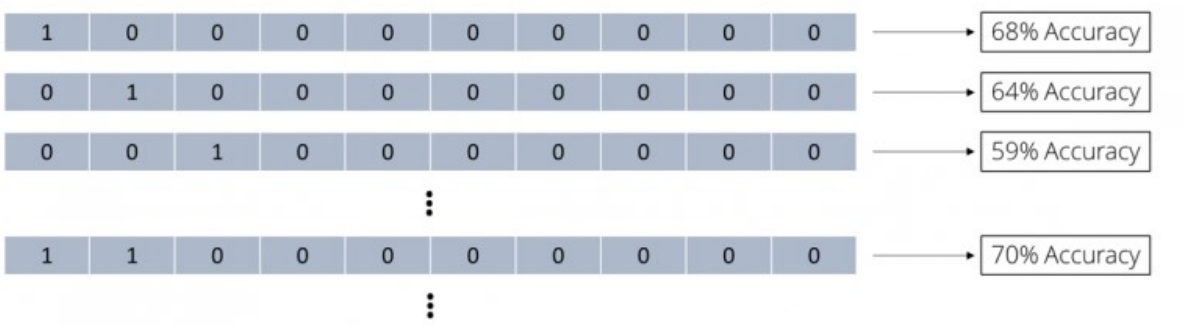
As we can see, when we use the first attribute we come up with an accuracy of 68%. That's already better than our accuracy with all attributes, 62%. But can we improve this even more? Let's try using only the second attribute:



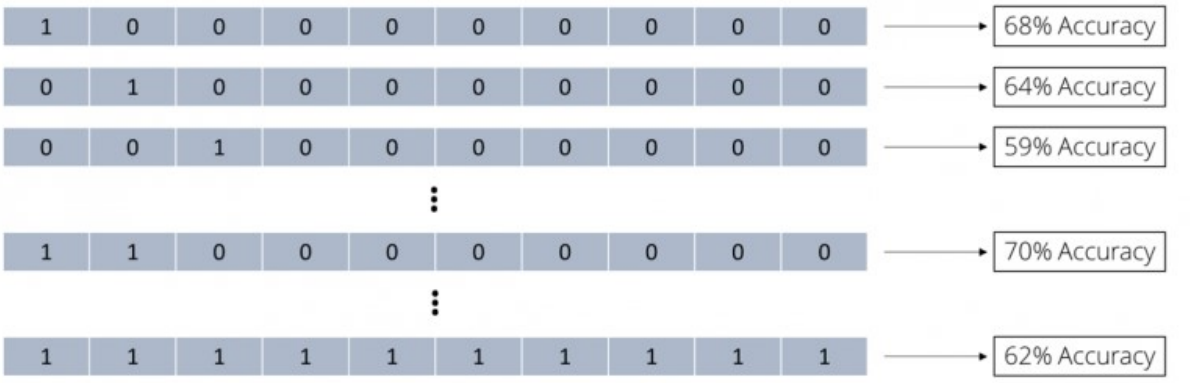
Still better than using all 10 attributes, but not as good as only using the first.



We could continue to go through all possible subsets of size 1. But why we should stop there? We can also try out subsets of 2 attributes now:



Using the first two attributes immediately looks promising with 70% accuracy. We can collect all accuracies of these subsets until we have tried all of the possible combinations:



We call this a *brute force* approach.

How many combinations did we try for 10 attributes? We have two options for each attribute: we can decide to either use it or not. And we can make this decision for all 10 attributes which results in 2 x 2 x 2 x ... = 2^{10} or 1,024 different outcomes. One of those combinations does not make any sense though, namely the one which does not use any features at all. So, this means that we only need to try $2^{10} - 1 = 1{,}023$ subsets. Even for a small data set, we can see there are a lot of attribute subsets. It is also helpful to keep in mind that we need to perform a model validation for every single one of those combinations. If we use a 10-fold cross-validation, we need to train 10,230 models. It is still doable for fast model types on fast machines.

But what about more realistic data sets? If we have 100 instead of only 10 attributes in our data set, we already have $2^{100} - 1$ combinations bringing the number combination to 1,267,650,600,228,229,401,496,703,205,375. Even the largest computers can no longer perform this.

Heuristics to the Rescue!

Going through all possible attribute subsets is not a feasible approach then. We should however try to focus only the combinations which are more likely to lead to more accurate models. We could try to prune the search space and ignore feature sets which are not likely to produce good models. However, there is of course no guarantee that we will find the optimal solution any longer. If we ignore complete areas of our solution space, it might be that we also skip the optimal solution, but these heuristics are much faster than our brute force approach. And often we end up with a good, and sometimes even with the optimal solution in a much faster time. There are two widely used approaches for feature selection heuristics in machine learning. We call them *forward selection* and *backward elimination*.

Forward Selection

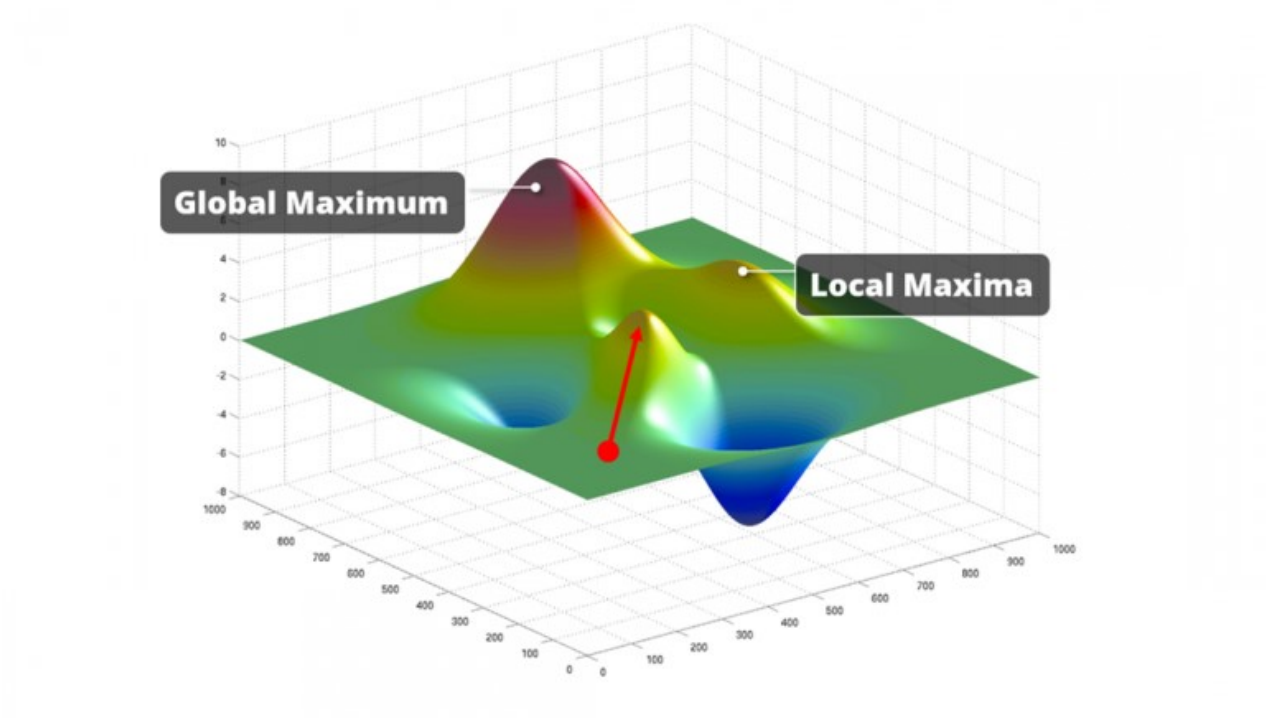
The heuristic behind forward selection is very simple. We first try out all subsets with only one attribute and keep the best solution. But instead of trying all possible subsets with two features next, we only try specific 2-subsets. We try the 2-subsets which contain the best attribute from the previous round. If we do not improve, we stop and deliver the best result from before, i.e. the single attribute. But if we have improved the accuracy, we continue trying by keeping the best attributes so far and try to add one more. We continue this until we no longer have to improve.

What does this mean for the runtime for our example with 10 attributes from above? We start with the 10 subsets of only one attribute which is 10 model evaluations. We then keep the best performing attribute and try the 9 possible combinations with the other attributes. This is another 9 model evaluations then. We stop if there is no improvement or keep the best 2-subset if we get a better accuracy. We now try the 8 possible 3-subsets and so on. So, instead of going brute force through all 1,023 possible subsets, we only go through 10 + 9 + … + 1 = 55 subsets. And we often will stop much earlier as soon as there is no further improvement. We see below that this is often the case. This is an impressive reduction in runtime. And the difference becomes even more obvious for a case with 100 attributes. Here we will only try at most 5,050 combinations instead of the 1,267,650,600,228,229,401,496,703,205,375 possible ones.

Backward Elimination

Things are similar with backward elimination, we just turn the direction around. We begin with the subset consisting of all attributes first. Then, we try to leave out one single attribute at a time. If we improve, we keep going. But we still leave out the attribute which led to the biggest improvement in accuracy. We then go through all possible combinations by leaving out one more attribute. This is in addition to the best ones we already left out. We continue doing this until we no longer improve. Again, for 10 attributes this means that we will have at most 1 + 10 + 9 + 8 + … + 2 = 55 combinations we need to evaluate.

Are we done? It looks like we found some heuristics which work much faster than the brute force approach. And in certain cases, these approaches will deliver a very good attribute subset. The problem is that in most cases, they unfortunately will not. For most data sets, the model accuracies form a so-called multi-modal fitness landscape. This means that besides one global optimum there are several local optima. Both methods will start somewhere on this fitness landscape and will move from there. In the image below, we have marked such a starting point with a red dot. From there, we continue to add (or remove) attributes if the fitness improves. They will always climb up the nearest hill in the multi-modal fitness landscape. And if this hill is a local optimum they will get stuck in there since there is no further climbing possible. Hence, those algorithms do not even bother with looking out for higher hills. They take whatever they can easily get. Which is exactly why we call those "greedy" algorithms. And when they stop improving, there is only a very small likelihood that they made it on top of the highest hill. It is much more likely that they missed the global optimum we are looking for. Which means that the delivered feature subset is often a sub-optimal result.



Slow vs. Bad. Anything better out there?

This is not good then, is it? We have one technique which would deliver the optimal result, but is computationally not feasible. This is the brute force approach. But as we have seen, we cannot use it at all on realistic data sets. And we have two heuristics, forward selection and backward elimination, which deliver results much quicker. But unfortunately, they will run into the first local optimum they find. And that means that they most likely will not deliver the optimal result.

Don't give up though – in our next post we will discuss another heuristic which is still feasible even for larger data sets. And it often delivers much better results than forward selection and backward elimination. This heuristic is making use of evolutionary algorithms.