Dimensionality Reduction

Dimensionality reduction isn’t quite the same as feature selection, even though both try to reduce the number of features. While feature selection selects and excludes some features without making any transformation, dimensionality reduction transforms features into a lower dimension.

The methods explained in this article use dimensionality reduction as a way to perform feature selection.

Principal Component Analysis

[Principal component analysis](https://heartbeat.fritz.ai/demystifying-principal-component-analysis-handling-the-curse-of-dimensionality-6b3101e5ae56) (or PCA for short) is a dimensionality reduction technique that uses linear algebra to transform a dataset into a compressed form.

To transform data from one dimension to another, PCA starts by calculating something called the Eigen decomposition (or singular value decomposition — SVD in other implementation) of the covariance matrix of the features.

This means that PCA first searches the correlation between features and then builds new features that **preserve**the same explained variance of the original ones. This process results in a lower-dimensional projection of the data that presents the **maximal data variance**.

Using this information from features’ explained variance, we can measure the importance of a given variable and see how much it’s contributing to the reduced feature space that PCA obtains.

So feature selection using PCA involves calculating the explained variance of each feature, then using it as feature importance to rank variables accordingly.

Here is a code snippet to start with:

There are many other algorithms to do dimensionality reduction to obtain feature importance, one of which is called linear discriminant analysis (LDA). It’s worth checking out, and here’s a good [blog post](https://machinelearningmastery.com/linear-discriminant-analysis-for-machine-learning/) to get started.

Did you know: Machine learning isn’t just happening on servers and in the cloud. [It’s also being deployed to the edge. Fritz AI has the developer tools to make this transition possible.](https://www.fritz.ai/product/platform.html?utm_campaign=buildmodels1&utm_source=heartbeat)

Heuristic Search Algorithms

Heuristic search algorithms attempt to perform optimization tasks in an iterative fashion to find an approximate solution when classic methods fail to find an exact solution. The method doesn’t always find the best or even the optimal solution, but it may instead find a good or acceptable solution within a reasonable amount of time and memory space.

Since we can end up with a lot of feature subsets, we can perform a heuristic search across them in order to find the best one. We already did this while working with wrapper methods, but this time we’re exploring more advanced search algorithms.

Next we’ll thoroughly explore one of these search methods: the genetic algorithm.

Genetic Algorithms

GAs or [genetic algorithms](https://heartbeat.fritz.ai/using-genetic-algorithms-to-automate-the-chrome-dinosaur-game-part-2-1c0007334297) are global optimization techniques for searching very large spaces. You can think of them as a sort of randomized search. They were inspired by the biological mechanisms of natural selection and reproduction.

They work throughout populations of possible solutions (called generations), wherein each solution in the search space is represented as a finite length string (chromosome) over some finite set of symbols, which then uses an objective (or fitness) function to evaluate the suitability of each solution.

In terms of feature selection, each chromosome will represent a feature subset, and it will be represented with binary encoding: 1 means **“choose”** a given feature and 0 means **“do not choose”** a feature.

*So for instance, the string 1001 means choose the first and the last feature as a feature subset.*

After that, we conduct many iterations to create a new generation (new feature subset) of possible solutions from the current generation using a few operators:

* **Selection:** Probabilistically filters out solutions that perform poorly while choosing high-performing solutions to exploit.
* **Cross Over:** This is the GA way to explore new solutions and exchange information between strings. This operator will be applied to select pairs of chromosomes randomly, with a probability equal to a given *crossover rate.* That way, we generate new chromosomes that hopefully will retain good features from the previous generations.
* **Mutation**: This operator protects GAs against the irrecoverable loss of good solution features. It changes a symbol of some chromosomes with a probability equal to a very low given *mutation rate*in order to restore lost genetic material.

The advantage of GAs in contrast to traditional search optimization methods (like gradient descent) is that GAs work with a population of solutions, which makes them more effective to escape local minima.

Here are the steps to feature selection with genetic algorithms:

1. Initialize a population with randomly-generated individuals—in our case, that means different feature subsets—and create a machine learning algorithm.
2. Evaluate the fitness of each feature subset with an evaluation metric of your choice, depending on the chosen algorithm.
3. Reproduce high-fitness chromosomes (feature subset) in the new population.
4. Remove poor-fitness chromosomes (selection).
5. Construct new chromosomes (crossover).
6. Recover lost features (mutation).
7. Repeat steps 2-6 until a stopping criterion is met (can be the number of iterations, for example).

To start using GA algorithms in your features selection process, you can either implement it yourself or use an open-source library called [sklearn-genetic](https://github.com/manuel-calzolari/sklearn-genetic). Here’s how you can install it using pip:

pip install sklearn-genetic

And here’s some sample code that uses this library to accomplish GA-based feature selection:

This object uses different parameters to configure the genetic algorithm search. Here are quick descriptions of the parameters:

* estimator: Model used to evaluate the suitability of the feature subset, alongside an evaluate metric.
* cv: int, generator, or an iterable used to determine the cross-validation splitting strategy.
* scoring: The evaluation metric, or what we can call the**fitness function** that evaluates the performance of a chromosome, i.e. the machine learning model’s performance against a subset of features.
* max\_features: Determines the maximum number of features selected.
* n\_population: Number of populations for the genetic algorithm, which means the different feature subsets.
* crossover\_proba: Probability value of a crossover operation for the genetic algorithm.
* mutation\_proba: Probability value of a mutation operation for the genetic algorithm.
* n\_generations: An integer describing the number of generations for the genetic algorithm — the stopping criterion.
* crossover\_independent\_proba: The independent probability for each attribute to be exchanged—that way we offer much more flexibility for the genetic algorithm to search.
* mutation\_independent\_proba: The independent probability for each attribute to be mutated by the genetic algorithm.
* n\_gen\_no\_change: The number of generations needed to terminate the search if the best individuals aren’t changing.

There’s also another technique we can use to search for best feature subsets with heuristic methods. This one is called **simulated annealing**, which is a global search algorithm that allows a suboptimal solution to be accepted in the hope that a better solution will show up eventually. I definitely encourage you to check out this concept—here’s a [good resource](https://medium.com/ai-techsystems/simulated-annealing-580f73bd807a) for that.

Feature importance

We used feature importance in a previous article while exploring tree-based algorithms. Here, we’ll take a look at another simple method to obtain feature importance: **permutation importance**.

Permutation importance

What permutation importance does is basically randomly shuffle the values of a feature (without touching the other variables or the target) to see how this permutation affects the performance metric of the machine learning model. Note that the choice of the metric is up to you.

It’s actually surprisingly straightforward. We measure the importance of a feature by measuring the increase in the model’s prediction error after permuting the feature values. This way, we’re breaking the relationshipbetween the feature and the true outcome.

* A feature is “important” if shuffling its values increases the model error. In this case, the model relies on the feature for the prediction.
* A feature is “non-important” if shuffling its values leaves the model error unchanged. In this case, the model ignores the feature for the prediction.

Here’s a code snippet that implements permutation importance:

There is a library called [skater](https://oracle.github.io/Skater/)that also implements a more advanced technique in permutation importance called **model-agnostic permutation feature importance**, so check it out.

The future of machine learning is on the edge. [Subscribe to the Fritz AI Newsletter to discover the possibilities and benefits of embedding ML models inside mobile apps.](https://www.fritz.ai/newsletter?utm_campaign=fritzai-newsletter-fomo4&utm_source=heartbeat)

Deep learning

Deep learning involves the use of neural networks to build high-performing machine learning models. What’s particularly interesting about neural networks is their ability to learn **nonlinear**relationships among features. Most of the traditional embedded-based methods that we explored do not—generally speaking, the methods we’ve covered can only explore linear relationships across features.

Here, we’re talking about the ability to derive feature importance from neural networks to help us select good feature subsets. These solutions use a particular neural network architecture called autoencoders.

Autoencoders

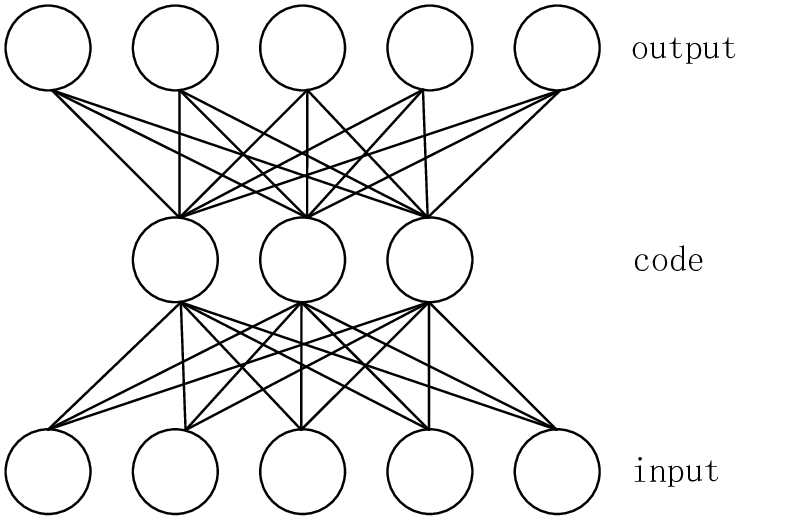
Autoencoders as a solution for feature selection was proposed in a paper called [“Autodencoder Inspired Unsupervised Feature Selection”](https://arxiv.org/abs/1710.08310),where the authors used what’s called **AEFS,**or **A**uto**E**ncoder for **F**eature **S**election to uncover existing nonlinear relationships between features.

To understand how the solution works, we first have to look at how autoencoders work. Autoencoders are unsupervised learners that can learn to compress and encode data, then learn how to reconstruct the data back from the reduced encoded representation. The resulting representation is as close as possible to the original input.

In other words, we’re taking the feature space and reducing its dimensionality, then reconstructing it from its reduced format—sounds strange, right?

You might be wondering why we’re trying to reconstruct something we have already. The answer is contained in the principle of autoencoders—by reducing the data dimensions, we learn how to **ignore the noise** in them, and moreover which features can best help us reconstruct the data.

AEFS uses a single-layer autoencoder to reconstruct data.



Single Layer Autoencoder

So after reconstructing the data, we use the first weight matrix of the autoencoder that connects the input feature layer to the reduced layer:

* If a feature contributes little to the representation of others, it means that the corresponding weight squared (w²) **is close to zero.**
* On the other hand, if the feature plays an important role in the representation of other features, the corresponding weight **must be significant**.

Here’s a sample code snippet in Keras:

One of the main drawbacks of this method arises from the simplicity of the autoencoder, where a simple single-layer autoencoder cannot model complex non-linear feature dependencies.

There are other deep learning-based solutions for feature selection, one of which is called [“Deep Feature Selection using a Teacher-Student Network](https://arxiv.org/abs/1903.07045)”, which tries to improve upon the disadvantages of the AEFS method**.**

Conclusion

Feature selection is a vast, complicated area, and a lot of research has been done to figure out the best methods that improve both the computational cost and the performance of the machine learning algorithms.

You can apply basic ideas that generally exist in computer science like searching algorithms, and more advanced concepts like genetic algorithms and deep learning to obtain feature subsets that will hopefully give better results.

In this field, as you have likely already realized, you have to test, combine, and innovate approaches to see what works better for your problem. There is no magic code snippet that works and gives the best result all the time — but this is an art that requires creativity.