Master Thesis - Explainable Machine Learning - Visualization of Random Forests

Fabio Rougier

October 16, 2022

1 Summary

DO SUMMARY AT THE END

Contents

2 Introduction

WRITE THIS AT THE END Random Forests (RF) (Breiman, 2001) are a powerful ensemble method with a low barrier of entry. Because of their ease of use and performance they are used in many applications. However, they fall short when it comes to transparency. On the one hand RFs can offer a multitude of valuable insights into their decision making and the data they process. On the other hand visualizing this is usually a challenge because of the inherent scale of a RF and the aggregation of their decision making process. The goal of this work was to provide relative beginners with a tool to explore a RF on a detailed level.

2.1 Visualizing Decision Trees

An obvious approach to visualizing a RF is inspecting the decision trees that constitute the RF. One example for the visualization of a decision tree is *BaobabView* (Van Den Elzen & Van Wijk, 2011). As many other approaches visualizing decision trees, it utilizes Node-Link Diagrams (NLDs) as its' main visualization. A confusion matrix yields additional insights into the relations of the underlying features. However this visualization does fall short when trees grow too large, as it becomes hard to inspect every individual node and branch of the tree. This is one of the problems that *TaxonTree* tries to overcome (Parr, Lee, Campbell, & Bederson, 2003). It uses a tree visualization approach that is scalable for large trees by adding the possibility to zoom, browse and search the tree. While this does help if a tree grows too large, it does not provide any insight on the general structure of the tree as a whole. Generalizing either of the approaches towards RFs is also non trivial. Both simply lack the scalability in the desired dimension. It also leads into a dangerous territory of focusing too much on the structure of individual trees. RFs - as all ensemble methods - arrive at their decisions by combining the decisions of the individual trees. Inspecting and even understanding individual trees, will only yield limited insights over the RF.

2.2 Visualizing Random Forests

To fully understand the structure and decision making of a RF, many aspects of the RF have to be conveyed by the visualization. First of all typical indicators like a *F1 score* give an indication of the overall performance. Additionally there are some metrics specific to RFs that should be included to get a deeper understanding of a particular instance of the RF, like the *mean impurity* of the individual trees or the *out of bag error*. With its' unique way of providing a distance measure for features, a confusion matrix can also yield valuable insights to the relations of the features and how the RF interprets them. As stated before, the visualization of RFs heavily relies on how it overcomes the scalability issues of RFs.

3 Related Work

3.1 ReFine

This approach is rather old, but still worth mentioning, because it highlights the fact, that visualizing RFs has been a challenge for some time now (Kuznetsova, Westenberg, Buchin, Dinkla, & van den Elzen, 2014). In this particular case, the technical implementation is of course out-dated, but the ideas are still relevant. *ReFine* uses a small multiples view of the trees to give the user insights from different angles. The main visualization uses icicle plots, as the author deems them most suitable due to their efficient use of space. While this does allow to show a considerable number of trees in one view, it does not scale well enough for large amounts of trees in a RF. An important distinction raised by the author is that there are different users for RFs, with very different requirements towards a RF visualization. While a machine learning expert might be more focused on improving model performance, an analyst or domain expert would be more concerned about the insights provided by the models.

3.2 iForest

The *iForest* visualization is one of the most promising visualization approaches for RFs. (Zhao, Wu, Lee, & Cui, 2018) It focuses on the interpretability of the RF and raises the concern that many domains would not even consider using RFs because of their lack thereof. The authors also approach the understanding of RFs by coming from two different angles: *Feature Analysis* and *Case Based Reasoning.* Both require different charts and give the user valuable insight into the dataset and the RF. The elaborate, dashboard-like web application utilizes the benefits of small multiples and has interconnected and interactive charts, each devoted to offer a specific perspective. One especially unique part of the dashboard is the use of *Partial Dependence Plots* to display the RF's classification behavior in regard to each feature. This is supported by a bar chart of the feature's distribution in order to support this view with more context. While the dashboard is a powerful tool, it can be overwhelming for users. It also requires some in-depth knowledge about RFs in order to come to conclusions or even an actionable instruction. *iForest* is clearly aimed at supporting data scientists to understand their own creations and not suited for a domain expert.

3.3 ExMatrix

The *ExMatrix* follows an out-of-the-box-thinking approach by breaking up machine learning models into a set of rules (Neto & Paulovich, 2020) (Ming, Qu, & Bertini, 2018). It is therefore even more flexible because it is not limited to be used with RFs only, but could be applied to almost any kind of machine learning model. It is best suited to abstract ensemble models and simplify them. This is however not to be mistaken with creating surrogates from the given models, as it reflects the underlying models exactly. By breaking down a RF in said rules, it is possible to display the entire RF in a matrix structure. It does this by representing columns as features and rows as rules, resulting in cells as so-called *rules predicates*. This entire rethinking of the RF as a whole allows for unique graphs and insights in the RFs' decision making. Thinking of the RF as a set of comparable rules and evaluating those, yields a very deep understanding, both on a case-based sample level, but also on a global level. While not supported in the original paper, this would also allow for intricate comparisons of models on the same data set. The biggest issue with this approach is, that while it does yield powerful insights, it can be very difficult to convey these insights to a domain expert. While a data scientist can be expected to wrap their head around the idea of disassembling a RF into a set of rules, this idea is not intuitive for a domain expert who might already have trouble understanding how the RF works in the first place.

3.4 Summary

While the main challenge of gaining access to the insights and details of the inner workings of a RF might have been solved already they seem to be hidden behind a kind of complexity layer. Powerful approaches for RF visualization exist, but there is a need to make them more accessible for domain experts. Considering not only the audience, but also the intended use of the visualization is paramount for the visualization to be useful. There is a lot of value to be derived from the existing work and some of the ideas have influenced this work.

4 Methodology

4.1 Important Packages

4.1.1 Python

This work was implemented in *Python 3.10.4* (Van Rossum & Drake, 2009), as it has become one of the standard programming languages for data science and machine learning. This allowed the usage of many commonly used libraries, like *NumPy* (Harris et al., 2020), *pandas* (Wes McKinney, 2010), *scikit-Learn* (Pedregosa et al., 2011) and more. Using Python also opens up the possibility to further extend this work in the future, while the libraries and possibilities of Python keep improving and expanding. Some of the libraries used in this work make use of *Cython* (Behnel et al., 2011) to speed up the execution of certain functions. This was particularly relevant for this work, as will be elaborated in a later section.

The built-in *pickle* module contains functionalities to serialize and deserialize *Python* objects. Since some of the computations made by the backend of this work are quite intensive, the repository contains *pickle* files of the precomputed results of a matrix computation. This allows for a quick display of the two example use cases.

Another built-in module used was the *multiprocessing* module. It allows for parallel execution of methods on multiple CPU cores. As the necessary matrix computations scale with the number of trees in the forest, it was crucial to speed them up significantly, which was achieved by using the *Pool* method of the module.

4.1.2 pandas

The *pandas* library offers a powerful data structure called *DataFrame*, which in essence is just a table. Its' implementation is made so convenient and efficient though, that it allows for both intuitive and performant data manipulation.

In this work, *pandas 1.5.0* is used as a backbone throughout the entire lifecycle of the visualization.

4.1.3 scikit-Learn

With the *scikit-Learn 1.1.2* library, machine learning is made easily accessible, even to beginners, while still providing some more advanced configurations if necessary.

For this work, both the *RandomForestClassifier* and the *DecisionTreeClassifier* are a main part of the processes in the background. The two example use cases, discussed in this work are based on the data provided by the *scikit-Learn* library, specifically the *Iris* and the *Digits* dataset. Additionally, the *DBSCAN* (Ester, Kriegel, Sander, Xu, et al., 1996) and *t-SNE* (van der Maaten & Hinton, 2008) algorithms are used as provided by the package.

4.1.4 Streamlit

Streamlit 1.13.0 provides users with an incredibly easy entry point to transferring Python visualizations into the web browser. With a simple *"st.write()"* command, any text, visual or even DataFrame can be displayed in the browser. While the package certainly has its' limits, it is a great tool for a fast development of dashboards and visualizations.

Streamlit was used in this work to handle the web application and the user interface that enables interactions with the visualizations and algorithms. The *Form* class was particularly useful for creating the sidebar that the user can interact with. Many sliders, paired with explanations and optional help-texts offer the user a lot of control and clarity when using the dashboard. While *Streamlit* has recently rolled out support for multi-page-apps, this work uses a well known workaround to achieve the same result, because it allows for more flexibility and did not require a fixed folder structure of the app. Some limitations of the package were overcome with the use of *HTML* and *CSS* in the *Streamlit* code, which is not ideal, but common practice in the *Streamlit* community.

4.1.5 Vega-Altair

Speaking of visualizations, *Vega-Altair 4.2.0* (VanderPlas et al., 2018) is a great library for creating interactive charts, with a great deal of flexibility for customization. It is built on top of *Vega-Lite* (Satyanarayan, Moritz, Wongsuphasawat, & Heer, 2017), which is a declarative grammar using the JSON standard for building graphs, that are then rendered using the *Vega-Lite* compiler. *Vega-Altair*, or *Altair* for short, is especially unique in its' way of constructing graph, by using *encodings* for each visual element. This syntax is very intuitive to use makes the library both powerful and easy to use.

Every chart in this work is created using *Altair*, which in turn allows the use of *Altair's* interactive capabilities like brushing and linking. Some of the charts in the dashboard offer the user the ability to highlight sections and see the corresponding data changes in a different chart right next to it.

4.1.6 NetworkX

NetworkX 2.8.7 (Hagberg, Schult, & Swart, 2008) is one of the most popular libraries for manipulating graphs in Python. It contains many useful functions for traversing, building and analyzing graphs.

Decision Trees are essentially binary trees, which can be represented as graphs. As such, *NetworkX* was used for some particular graph operations like the graph edit distance. In order to transform the *DecisionTreeClassifier* into a graph, *PyGraphviz* was used as an intermediate step.

4.2 Algorithms

4.2.1 Decision Trees and Random Forests

A RF, as described earlier, is an ensemble method combining multiple decision trees into a single classifier. By combining multiple decorrelated trees, the tree's performance can be much better than a single tree, while avoiding overfitting.

Decision trees are one of the most intuitive machine learning approaches, because they provide an interpretable path for their decisions. They are built recursively by splitting the training data into subsets based on a threshold value, computed by a predefined measure like *impurity* or *entropy*. This process is deterministic and can lead to very high performing models. Unfortunately, they are prone to overfitting the training data, which makes them less useful in the real world. Pruning and the use of larger training sets can usually negate this only to a degree.

The idea of RFs is to make use of this overfitting problem and turn it into an advantage. Through a process called *bagging*, each tree is trained on a different subset of the training data. Additionally, by choosing the splitting feature at each recursive step randomly, the trees are decorrelated and result in heterogenous set of trees. The most common way of combining the trees is a majority vote, in the case of classification, or a mean of the predictions, in the case of regression.

Their good performance is not their only strength though. Through libraries like *scikit-Learn*, RFs can be set up with just a few lines of code. Depending on the size of the training data, even the training process can be done in a matter of seconds. As displayed in the dashboard developed in this work, this can deliver extremely high performing models with very little background knowledge or further tuning from the user. This makes them a very attractive tool for domain experts.

The main downside is the lack of interpretability of RFs. While it would be possible to inspect individual trees and every single decision they make, it is hard to interpret their impact on the decisions of the global model.

4.2.2 Graph Edit Distance

A crucial algorithm used in the backend of the dashboard is the *graph edit distance* (Sanfeliu & Fu, 1983). It is a basic measure of similarity between two graphs, working in a similar way as the *Levenshtein Distance* (Levenshtein et al., 1966), but for graph structures. It considers two graphs and computes the operations needed to transform one into the other. Each operation can have a cost attributed to it, which is then summed up to get the final distance.

In this dashboard the metric serves as the central measure of similarity between two trees. The *NetworkX* implementation of the algorithm allows to specify cost functions for each operation individually, but uses the same cost for all operations if nothing else is specified. The default configuration of the *graph edit distance* method does not take into account the node labels, which is why a custom cost function was used, to ensure that not only the morphology of the trees is considered, but also the labels. It was also vital to pass the root nodes of the two trees to the method in order to improve calculation performance.

Calculating the *graph edit distance* between two trees is a very expensive operation, which is why the method was configured to utilize the *timeout* parameter with 0.5 seconds. The method will then return the best result it could find within the given time. This was tested and deemed to resemble the best compromise between performance and accuracy.

While other metrics exist to compute the similarity between two trees, the *graph edit distance* was chosen, because it is intuitive to understand and easy to implement. Since the basic graph structure in this use case will always be a binary tree, more elaborate metrics are not necessary.

4.2.3 Pairwise Distance Matrix Calculation

The assembly of the pairwise distance matrix was a crucial part of the back-end code to get its' performance up to par. Since the matrix is symmetric, it is only necessary to calculate half of it. The metric used in calculating each pairwise distance is the aforementioned *graph edit distance*, each individual calculation is expensive on its' own. It was therefore critical to have a fast implementation of the matrix computation overall and to avoid unnecessary calculations as much as possible.

As described in the *Python* subsection, the *multiprocessing Pool* class was essential in speeding up the computation. It is written in such a way, that the calculating method expects only a single tree which is compared against all other trees, minus the ones that have already been compared. This is done through the *map* method, which receives a list of all the trees and the method to be applied to each of them. This resulted in a significant speedup and allowed to set the *timeout* parameter of the *graph edit distance* to a higher value and in turn have more accurate results. The iterative calculation could only handle timeouts in the milisecond range, which still resulted in calculation times of over 10 minutes, while the current implementation, running on an *AMD Ryzen 5 5600X* 6-Core Processor with 12 threads, takes about 5 minutes.

Obviously, this is still not a fast enough implementation for a user to be run on the fly, which is why the pairwise distance matrix is only calculated once and then stored in a *pickle* file. For the two example use cases these initial calculations have already been computed for the RFs with 100 trees and the *pickle* files are included in the repository, so that the user will not have to wait for the calculations on startup.

After the calculation, the results are normalized, using a *MinMaxScaler*, also from the *scikit-Learn* library. This is necessary, because the *graph edit distance* can result in larger values for larger trees, as each operation has a cost associated with it. To retain the distribution of the distance values as best as possible, the *MinMaxScaler* was considered appropriate, considering that the resulting distance values have to be positive and the distribution is not necessarily gaussian.

4.2.4 DBSCAN

The main algorithm used in this work to cluster the trees is the *Density-based spatial clustering of applications with noise* or *DBSCAN* algorithm. It is a density-based cluster algorithm, which is capable of finding clusters in high dimensional data. A clear advantage of the DBSCAN algorithm is that the amount of clusters is not defined as a hyperparameter, but is determined by the algorithm itself. It is also capable of defining data points as noise, which makes sense for the given use case, as it is quite possible that a considerable amount of trees is not associated to any cluster at all. The idea to use the *DBSCAN* in this context was considered intuitive, as trees can share similar features starting from their root node and will differ more and more as they grow deeper. Since the algorithm assumes *core points*, trees with these shared features would be expected to be in a cluster together and form core-points, while trees with different labels on their first few levels would be assigned to a different cluster.

As described in the previous section, the distance calculation between two trees is computationally expensive and recalculations should be avoided.

The *DBSCAN* algorithm is therefore implemented by making use of the *precomputed* parameter of the *sklearn* implementation. The *pairwise distance matrix* is passed to the algorithm as computed earlier. Which results in a significant speedup, as the distance calculation is the most time consuming part of the algorithm. The two example use cases have both been configured with default parameters in order to show a certain behaviour. The results of which will be discussed in the *Analysis* section.

4.2.5 t-SNE

The *t-distributed stochastic neighbor embedding* or *t-SNE* algorithm is based on the *stochastic neighbor embedding* or *SNE* (Hinton & Roweis, 2002) algorithm and is used to visualize high dimensional data in lower dimensional space. By going through two phases, the algorithm first computes pairwise probability distributions between points, usually using the euclidian distance. In this case, the *pairwise distance matrix* provides the results for the *graph edit distance* for this phase. In the second phase, the *Kullback-Leibler divergence* (Csiszár, 1975) is minimized between every pair of probability distributions in the low dimensional space. The number of components in the low dimensional space can also be specified and was set to two for this work. *t-SNE* is a non-deterministic algorithm, which means that the results will vary slightly each time it is run. We avoid this in the dashboard by using the *random state* parameter, which is set to a fixed value.

For our use case, the *t-SNE* algorithm is used with the *precomputed* parameter, just as the *DBSCAN* algorithm before to avoid unnecessary calculations. The *t-SNE* results are a double edged sword however, because the algorithm will always output some kind of embedding. This can make it hard to evaluate the output, as the problem is high dimensional and there is no way to know the *true* cluster assignment. By presenting the user with both, the *t-SNE* and the *DBSCAN* results, it puts the resulting cluster assignments into context.

4.3 Target Audience

The goal of this work was to provide a tool that is easy to use and has a low barrier of entry. As the target audience, domain experts with little experience with data science were considered. Since a RF is not something used by anyone in their daily life, it was expected, that the user would have at least a basic understanding of statistics and machine learning, but has not yet been in contact with the concept of a RF.

Given this premise, the dashboard has an educational aspiration and is in its' current state not intended to be used as a tool for data scientists. As such, all graphs are accompanied by explanations and the user is introduced to concepts one step at a time. Some of the graphs and and explanations simplify concepts or just briefly touch certain aspects. This is intentional and has the purpose of not overwhelming the user with too much information at once. The contents discussed in the dashboard are already quite complex, so every introduction of new concepts or more elaborate explanations were carefully weighed between the benefit of the explanation and the risk of overloaded information.

4.4 The Dashboard

4.4.1 Backend

The backend of the dashboard is constructed in a three-layer architecture, separating data, logic and visualization. While the logic and data layers are more tightly coupled, there is a clear separation between the creation of the visualizations and the creation of the dashboard pages. By separating the layers in this way, further development of the dashboard is made easier, as a change in one layer will not affect the other layers. As stated in the previous section, the two datasets *Iris* and *Digits* from the *sci-kit Learn* library are used for the example use cases. It was therefore not necessary to implement any elaborate data loading process.

The data layer as it exists now, mostly loads the respective dataset into a *pandas Dataframe* and performs some basic preprocessing.

The RF training and all other computations are happening in the logic layer. It contains the code for training the Random Forest, computing feature importances and the calculation of a distance matrix required for the *DBSCAN* clustering and a *t-SNE* embedding of the trees. If the standard use case is selected, the distance matrix is pre-loaded from the *pickle* file. The results are then assembled in the central *tree Dataframe*, containing detailed information about each tree in the forest. This assembly is handled by a dedicated class. After constructing the *tree Dataframe*, two classes construct the *Streamlit* page. While one of them contains logic and information about the layout of the respective page, the other creates the individual visualizations. The text segments are loaded from the *markdown* files located under the *"text"* folder of the app. By separating the creation of individual graphs from the actual creation of the page in any layout, further development of the dashboard is made easier.

4.4.2 Frontend

The frontend of the dashboard is based on two main components of the code base: The layout, defined in the respective class and a **.toml* file, where the theme of the dashboard is determined. On the left side is a sidebar, containing navigation options for the user. Radio buttons allow to switch between the different pages, while a dropdown menu offers the option to select between the two different use cases. The initial use case presented is the *Iris* dataset. The first page shown to the user is the *Tutorial* page, containing a short welcome message, explaining what the dashboard is about. What follows is an explanation of the currently selected use case, decision trees and random forests, supported by some explanatory pictograms. The goal of this page was to make sure, that anyone accessing the dashboard could get an immediate idea of what the dashboard is about and have a basic understanding of the topic. Each use case has a short description of the dataset and also the explanation of the decision tree is altered slightly to fit the respective dataset.

The second page is the *Dashboard*, where the user is introduced to the use case application of the RF. When changing to this page, the *Algorithm Parameters* appears in the sidebar. This section starts with a short warning about the fact, that the dashboard will be reloaded upon pressing *"Run"* and that this action can take some time. After this prompt, some parameters can be selected for the dashboard. Each of the parameters has a help text next to it, that can be expanded by hovering over the question mark. In the help text, the user is given some basic idea for what a change of the parameter would do. The only available parameter for the RF is the number of trees, which ranges between 20 and 200. These boundaries were chosen, because a random forest with less than 20 trees is less likely to be used in practice and values over 200 would take the matrix calculation excessively long to complete. Below this slider inside two *streamlit metric* widgets, the user can see the current *Silhouette Score* (Rousseeuw, 1987) of the clustering and the amount of trees that are currently assigned to a cluster. The *Silhouette Score* gives an indiciation of how well each point is assigned to its cluster. The score ranges from -1 to 1, with 1 being the best possible score. It can be computed for each point, as shown in later plots, but also as an average over all points, or trees in this case. It is important to note at this point, that the Silhouette Score is not optimally suited to be used for a DBSCAN clustering, as it also classifies points as *noise*. Factoring these points into the score would artificially decrease the score depending on the amount of noise detected. This is solved here by not including the noise points in the calculation of the score. Since this in turn artificially increases the score, especially for clusterings with a large amount of noise, the user sees the amount of trees currently assigned to a cluster in a metric widget right next to the *Silhouette Score* metric widget. This puts the score into perspective and the user is made aware of the fact that the score should not be taken at face value, which is additionally supported by the explanatory text boxes souröunding the metric widgets and *Silhouette Score* text elements.

Following these metrics, the *DBSCAN* parameters *'min samples'* and *'eps'* and below that the parameters *learning rate*, *perplexity* and *early exaggeration* for the *t-SNE* embedding can be adjusted. Each parameter has a help text next to it, to explain its' basic behaviour to the user. Unfortunately, the size of the sidebar is only adjustable by the user and there is currently no official way to set the standard size of the sidebar to a fixed value. After introducing the user to the RF with some basic information, the first graph and its' explanation are shown below. Starting with a simple bar chart, displaying the feature importances of the RF, the user is introduced to the concept of feature importances. This is the most common graph in the context of RFs and gives the user a first impression of the inner workings of the RF. The next graph covers the F1-score of the RF for each class. A blue line indicates the average *F1-score* of the RF to put this measure into context. Showing the discrepancies between classes can give users a hint of certain classes, that might be harder to predict than others. An explanation accompanies the graph and gives a brief explanation of the *F1 score*. From an analytical perspective it is interesting to see that the RF performs especially well on the *Iris* dataset, and achieves an *F1 score* of almost 1 for the *setosa* class.

The following chart is a heatmap showing the *pairwise distance matrix* of all trees. This was constructed by calculating the graph edit distance between all trees with the goal to show how similar the trees are to each other. The idea behind this, was to exploratively be able to discover groups of similar trees in the forest and further inspect them. The darker spots in the heat map are where trees are especially similar to each other. Since this is sorted by the *DBSCAN* clustering, the user can already see some of these cluster patterns in the heatmap. What follows is a more elaborate explanation of this idea. It is crucial for the user to understand this concept, as the subsequent charts are based on this idea.

The text also briefly explains the *DBSCAN* algorithm as well as the concept behind the *Silhouette Score* so that the user is able to interpret the following charts. The successive chart combines the bar plot shown before, comparing the *F1-score* over the different classes, but this time discerning between the different clusters. With this, the user can see which clusters are better or worse at predicting certain classes. Again, a blue line indicates the average *F1-score* of the RF for the respective class. Additionally, the bars are color coded according to the *Silhouette Score* of their respective cluster. It is important to note here, that the *Silhouette Score* is not originally meant to be used with clusters that identify outliers. The data points classified as *noise* by the algorithm, are therefore automatically assigned a value of -1. This visually separates the *noise* bars from the others, while also giving the user a hint, that the *noise* cluster should not be viewed as a uniform cluster, like the others. As noted before, especially the *setosa* class performances are very good, but across the classes and clusters, it is clear that some trees perform better than others.

The *t-SNE* embedding is the last major chart for the user to explore. It shows a two-dimensional representation of the trees as a scatter plot. The trees are colored according to their respective *Silhouette Score*. Next to the plot is the *Silhouette Plot* showing how well each cluster was identified by the *DBSCAN* algorithm. With this two dimensional embedding the user can have a second look at the clustering. If both algorithms align on the clusters, chances are that the parameters were chosen correctly. The *t-SNE* algorithm is not explained in detail, as this would be too much information at this point. This is the first plot, where brushing comes into play. The user can select a group of points on the left to see them highlighted in the plot on the right. This gives the user the opportunity to more deeply explore the trees.

The last chart repeats two of the charts that the user already knows from before, but put into a new and interactive context. With the *t-SNE* plot on the left, the user can again select a group of trees, but this time the bar chart on the right will show the feature importances of the selected trees. From here, the user is free to explore the trees of the forest in the regards to their feature importances and it is possible to see, that certain classes prefer certain features over others. This is especially interesting for further analysis and combining it with the knowledge of the preceding charts, that already showed that some clusters are better or worse at predicting certain classes.

The user is then invited to explore the parameters and the other use case through the sidebar to see how the different parameters affect the clustering results.

5 Analysis

5.1 Random Forest

RFs performen sehr gut auf Iris, auf Digit immer noch decent ohne viel tuning
Noch was zu feature importances sagen

5.2 Class Performance Comparison

5.3 Pairwise Distance Matrix

5.4 DBSCAN

5.5 t-SNE

6 Conclusions

6.1 Future Work

7 Statement of Independent Work

References

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., & Smith, K. (2011). Cython: The best of both worlds. *Computing in Science & Engineering*, *13*(2), 31–39.

Breiman, L. (2001). Random forests. *Machine learning*, *45*(1), 5–32.

Csiszár, I. (1975). I-divergence geometry of probability distributions and minimization problems. *The annals of probability*, 146–158.

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd* (Vol. 96, pp. 226–231).

Hagberg, A. A., Schult, D. A., & Swart, P. J. (2008). Exploring network structure, dynamics, and function using networkx. In G. Varoquaux, T. Vaught, & J. Millman (Eds.), *Proceedings of the 7th python in science conference* (p. 11 - 15). Pasadena, CA USA.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... Oliphant, T. E. (2020, September). Array programming with NumPy. *Nature*, *585*(7825), 357–362. Retrieved from `https://doi.org/10.1038/s41586-020-2649-2` doi: 10.1038/s41586-020-2649-2

Hinton, G. E., & Roweis, S. (2002). Stochastic neighbor embedding. *Advances in neural information processing systems*, *15*.

Kuznetsova, N., Westenberg, M., Buchin, K., Dinkla, K., & van den Elzen, S. (2014). Random forest visualization. *Eindhoven University of Technology*.

Levenshtein, V. I., et al. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady* (Vol. 10, pp. 707–710).

Ming, Y., Qu, H., & Bertini, E. (2018). Rulematrix: Visualizing and understanding classifiers with rules. *IEEE transactions on visualization and computer graphics*, *25*(1), 342–352.

Neto, M. P., & Paulovich, F. V. (2020). Explainable matrix-visualization for global and local interpretability of random forest classification ensembles. *IEEE Transactions on Visualization and Computer Graphics*, *27*(2), 1427–1437.

Parr, C. S., Lee, B., Campbell, D., & Bederson, B. B. (2003, January). *Taxontree: Visualizing biodiversity information* (Tech. Rep. Nos. HCIL-2003-40, CS-TR-4678). Retrieved from `https://www.microsoft.com/en-us/research/publication/taxontree-visualizing-b`

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Rousseeuw, P. J. (1987). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, *20*, 53–65.

Sanfeliu, A., & Fu, K.-S. (1983). A distance measure between attributed relational graphs for pattern recognition. *IEEE transactions on systems, man, and cybernetics*(3), 353–362.

Satyanarayan, A., Moritz, D., Wongsuphasawat, K., & Heer, J. (2017). Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, *23*(1), 341-350.

Van Den Elzen, S., & Van Wijk, J. J. (2011). Baobabview: Interactive construction and analysis of decision trees. In *2011 ieee conference on visual analytics science and technology (vast)* (pp. 151–160).

van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, *9*(86), 2579–2605. Retrieved from `http://jmlr.org/papers/v9/vandermaaten08a.html`

VanderPlas, J., Granger, B., Heer, J., Moritz, D., Wongsuphasawat, K., Satyanarayan, A., ... Sievert, S. (2018). Altair: Interactive statistical visualizations for python. *Journal of Open Source Software*, *3*(32), 1057. Retrieved from `https://doi.org/10.21105/joss.01057` doi: 10.21105/joss.01057

Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual.* Scotts Valley, CA: CreateSpace.

Wes McKinney. (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt & Jarrod Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (p. 56 - 61). doi: 10.25080/Majora-92bf1922-00a

Zhao, X., Wu, Y., Lee, D. L., & Cui, W. (2018). iforest: Interpreting random forests via visual analytics. *IEEE transactions on visualization and computer graphics*, *25*(1), 407–416.