157
104
184
139
161
89
163
181
259
73
224
235
200
119
262
193
9
107
227
62
3
42
56
125

IVPY: ICONOGRAPHIC VISUALIZATION INSIDE COMPUTATIONAL NOTEBOOKS

DAMON CROCKETT

ABSTRACT | Iconographic Visualization in Python, or ivpy, is a software module, written in the Python programming language, that provides a set of functions for organizing iconographic representations of data, including images and glyphs. The module also provides methods for extracting visual features from images; generating and hand-tuning clusters of data points; and embedding high-dimensional data in 2D coordinate spaces. It is designed for use inside computational notebooks, so that users working with data needn't leave the notebook environment in order to generate visualizations. The software is designed primarily for those researchers working with large image datasets in fields where human visual expertise cannot be replaced with or superseded by machine vision, such as art history and media studies.

KEYWORDS | iconographic visualization; direct visualization; big image data; distant viewing; feature extraction

Introduction[1]

There persists, in the computational sciences, a common misconception about the nature of visual evidence. This misconception is particularly common in analyses of image data, which data often appear first, to the researcher, as massive arrays of numbers and only later (or perhaps hardly at all) as seen images. This way of thinking treats images as mere numerical signals, to be used as inputs to predictive models or as containers for basic object and scene contents. To be sure, images can and do serve these purposes. But in the background of this general attitude is the implication that, from an information-theoretic perspective, the image is nothing more than the numbers used to reconstruct it on a computer screen. And this is simply false: the information conveyed by an image will depend in part on facts about the viewer, facts not carried in the image's digital encoding.

Importantly, human experts remain our best and perhaps only judges of the significance of particular findings—visual or otherwise—and so to restrict search to machine perceivers is to search without understanding. It is not enough for the human expert to certify this or that result of machine search; it is rather required that she search, at least some of the time, with her own eyes, because no amount of priming the machine to see can confer the wisdom and flexibility of her own expertise. The human expert may use known, explicit categories to understand her observations, but she also works under a vast network of implicit biases that develop over time in response to new evidence or changing contexts in ways she may fail even to be aware of. It is this drifting collection of working principles that the human expert brings to bear on her domain and on the specific deployment of her public concepts, and no machine can replicate it in full. This is the motivation for the ivpy software module.

Iconographic Visualization in Python—ivpy—is a software module, written in the Python programming language, for visualizing large image datasets, or, alternatively, large datasets represented as collections of glyphs. The module offers a variety of methods for assigning images to (abstract) positions in visual similarity spaces, as well as a set of flexible plotting functions for organizing images and glyphs according to these and other (nonvisual) properties. These tools enable the researcher to extend the reach of her visual expertise into image collections too large to examine serially. Although the module can be used in Python scripts to save visualizations to disk, it is designed primarily for use inside computational notebooks, which are quickly becoming the default working environments for academic computational scientists of all kinds. For such researchers, ivpy has, accordingly, a distinct advantage over web-based, point-and-click visualization tools, because in order to use these tools, the researcher must either shuttle between incompatible working environments or settle for the relatively limited analytical expressiveness characteristic of graphical user interfaces.[2] Additionally, the integration of ivpy into a Python-based analytic workflow means that it sits atop Python's massive scientific toolkit, including modules for data retrieval, management, storage, modeling, and visualization.[3]

Ivpy is designed primarily for researchers whose data are images, whose goals are description and explanation (as opposed to, say, prediction or intervention), and whose domains of study are sufficiently complex that the giving of descriptions and explanations requires human expertise. This includes (inter alia) those working in art history, visual culture, media studies, social science (using social media images) and architecture and urban studies (using street level and satellite imagery).

Iconographic Visualization

Ivpy is a software module for iconographic visualization, which is a special case of unit visualization. Unit visualizations are those in which each data record is represented by its own visual mark.[4] This is in contrast with aggregative visualizations—e.g., bar, line, and pie charts—whose visual marks can represent any number of data records or statistical transformations on those records (e.g., averages). The conceptual simplicity of a one-to-one mapping between data and graphics is intuitively appealing, of course, although it's difficult to say precisely what advantage this confers on the user, apart from its (perhaps) being easier to learn. Importantly for us, unit visualizations make it possible to identify individuals in the data (assuming their numbers do not exceed the display resolution). This means that in any analytical context where individuals figure prominently, as we might expect to be the case in analyses of cultural collections, unit visualizations are common. And in general, unit visualizations have become more common in the age of computer graphics, because they are no more difficult to make than their aggregative counterparts.[5]

Iconographic visualizations are unit visualizations whose visual marks can be distinguished by their nonrelational (or 'local') visual characteristics—the qualities they carry with them regardless of their spatial positions, things like shape, size, and color.[6] Distinguishing nonrelational characteristics allow icons to carry information even outside the plotting context. In this way, each icon is itself a data visualization, and iconographic visualizations are therefore (in principle) viewable at multiple scales, from the global to the maximally local.

Strictly speaking, the visualizations ivpy produces needn't be iconographic. As we shall see, the module's core plotting functions are indifferent to the units they receive, and simple, identical marks can be used, in the manner of a traditional scatterplot. However, the density of information carried in the visualizations, and thus the analytical power of the tool, is increased by the use of icons, and one special class of icons in particular—image thumbnails—will be the focus of this paper.

Direct Visualization and Machine Intelligence

In analytical contexts where images are the objects of study, visualizations using those images as plotting units can be said to be 'direct', because perceptual access to them is unmediated.[7] Or at least, the mediations are both information- and format-preserving, limited typically to digital encoding and subsequent reconstruction on a screen at near lossless fidelity. Images therefore tend to be very information-rich relative to mere icons, which are typically lossy by design. In cases where lossless transmission is possible, mere icons are appropriate only when certain other conditions are met—if, for example, the analysis is finished and communication is the goal, or if the domain of study is so well-understood that what is lost is known to be noise. These conditions will likely be unmet in any case where images are the proper objects of study. In such cases, direct visualization is essential.

But how, it might be asked, can a lossless viewing of the data possibly yield anything of analytical value? After all, it is widely acknowledged that, past a certain density threshold, a data visualization ceases to be effective, because it overwhelms the analyst with information.[8] In a lossless data display, everything is preserved, but little is discovered, because signal and noise exist side-by-side, competing for attention—or so it might be in most cases. But images are special. For one, images, unlike mere icons, do not require a preliminary decoding step.[9] Moreover, the viewing of images by human experts is itself an implicit process of feature extraction, legitimized by the viewer's expertise. When, for example, the art historian casts her eyes over a collection, she separates signal from noise on her own; the visual interface needn't do it for her.

And indeed, in fields like art history and media studies, direct visualization is the norm. Researchers working in these disciplines have always looked at their images. The distinction between this traditional activity and direct visualization of the sort ivpy makes possible is a matter of scale, and at scale, the process of making observations cannot be (or at least, it oughtn't be) wholly unstructured, as it might be if the data are very small. Too much structure, or too much of the wrong kind of structure, however, threatens the analytical independence of the human expert. A process of direct visualization is successful if it allows the researcher to search and understand an otherwise unmanageable amount of data with her own eyes.

In this way, direct visualization serves as a bulwark against the encroachment of machine intelligence into data analysis. In certain cases, such encroachment is appropriate or at least unobjectionable, as it might be if your goal is an accurate predictive model and nothing more. In such cases, the identification of stable numerical regularities will suffice, and, assuming the conditions are favorable, a suitably designed machine can find them. If, however, your goal is description or explanation, and the domain of study is sufficiently complex that the identification of stable numerical regularities is either impossible or inadequate to the task, you cannot rely exclusively on machine search.

The difference in goals is crucial. If your goal is prediction, and your model performs well, you are indifferent to its features. But if your goal is description or explanation, you don't have the luxury of indifference: your model needs a very particular set of features—the descriptive or explanatory ones—and the machine simply cannot know which ones they are. In the best cases, you can review its findings and identify the features of interest; in the worst cases, the features of interest are not found at all. The machine can only find what it has been taught to find, and you can't anticipate everything it might encounter. For this reason, it is absolutely essential that you search for yourself.

This is a quite general problem plaguing the application of statistical learning to data projects with descriptive or explanatory aspirations. In most such cases, direct visualization is not available as an alternative. We should count ourselves lucky, then, if our data are images.

Direct Visualization and the Graphical User Interface

Despite its considerable power as an analytical method, there are few publicly available software tools designed specifically for direct visualization.[10] Of course, direct visualizations can be made 'from scratch' using software libraries like Processing[11] or imaging applications like Adobe Photoshop and ImageJ.[12] Ideally, however, the user should not have to generate the low-level plotting logic on her own or construct a plot manually from image files. The specification of direct visualizations should happen at a higher level, so that the user can focus on the analytical properties of the plots rather than on the details of their construction.

There is a small but growing number of software tools for the high-level specification of direct visualizations, and all are embedded in graphical user interfaces. Graphical user interfaces are designed to increase both the efficiency and accessibility of the computational tools they make available to the user. The efficiency of a tool is a measure of the effort required to use it; the accessibility of a tool is a measure of the effort required to learn it.[13] User inputs to a graphical user interface are typically limited to things like clicking buttons, toggles, and other control elements; navigating menus; scrolling; and typing small bits of text into search bars. These forms of user input are now so ubiquitous they hardly need to be learned, and, if they are well-designed, they offer significant gains in efficiency relative to standard programming environments.

The Software Studies Initiative's ImagePlot is perhaps the only graphical tool in existence for high-level specification of static direct visualizations, and ivpy's functionality is based in large part on its techniques.[14] All other software tools for direct visualization are graphical user interfaces for specifying direct visualizations interactively, usually in a web browser. Microsoft Research's PivotViewer is perhaps the earliest such example but now appears to be defunct. It was designed for plotting iconographic representations of all kinds and offered controls for sorting and filtering, with the resulting visualizations appearing as unit histograms or bar charts.[15] Timeline Tools, by Florian Kräutli, is designed specifically for cultural image collections, and plots collection items together as a unit histogram along a horizontal temporal axis.[16] Users can select individual collection items to reveal catalog metadata. PixPlot,[17] by Doug Duhaime, is designed for use with images, and extracts image features using a pre-trained convolutional neural network;[18] it then uses those features to plot the images in a visual similarity space, compressed to two dimensions from over two thousand.[19] Additionally, the tool identifies centers of dense clusters of images, which clusters can then be selected to re-center the plot. VIKUS Viewer,[20] by Christopher Pietsch, combines several functional elements we've seen already: like Timeline Tools, it plots images together as a unit histogram along a temporal axis; like PivotViewer, it allows the user to filter the data using metadata variables; and like PixPlot, it can extract visual features using a neural network and allows users to view the images in the resulting similarity space, compressed by a dimension reduction algorithm.[21]

All of these tools offer significant gains in both accessibility and efficiency relative to low-level specification in programming environments. But what they gain in accessibility and efficiency, they lose in expressiveness. The expressiveness of a visualization tool is a measure of the diversity of visualizations it can specify. A graphical user interface will always be less expressive than the programming language it is built on, but some graphical tools—like Adobe Photoshop and Illustrator—are designed with expressiveness in mind. In most cases, this will mean adding functionality in menus, thereby incurring losses to efficiency and accessibility. The visualization tools described above have all of them chosen to optimize for efficiency and accessibility at the cost of expressiveness, and because of this, I argue, none are ideal for data science.[22]

Each of these tools presents the same dilemma to the data scientist: either switch frequently between working environments or settle for the relatively limited expressiveness of the graphical user interface. Because none of these tools is a software module, all require the data scientist to generate visualizations outside her usual working environment, a place she can manipulate her data using code. If she chooses to stay in her programming environment, she cannot use these tools; if she chooses to stay in the graphical user interface, she cannot manipulate her data in the usual ways.

Ivpy resolves the dilemma by providing a set of plotting functions written in the Python programming language, which means that the functions are usable in all the same environments as any other Python libraries, including those belonging to Python's data science stack, libraries like NumPy,[23] pandas,[24] and scikit-learn.[25] Ivpy is therefore backed by a powerful set of tools for data manipulation—the very tools, indeed, that many data scientists are already using to manipulate their data. Together with ivpy's flexible plotting functions, this backing provides the data scientist with a highly expressive tool for generating direct visualizations inside her native working environment.

Computational Notebooks

Because it is a Python module, ivpy can be used anywhere Python is used, including interactive terminals, scripts, and integrated development environments. It is designed, however, for use inside computational notebooks. The computational notebook is a particular form of integrated development environment that presents as an electronic version of the scientist's laboratory notebook, itself a place to record the various tests and procedures that make up the daily business of laboratory research. This record is essential to the practice of science, because it serves as a kind of scientific memory—what was tried, what were the precise parameters of the attempts, and what happened as a result. Computational notebooks improve in some ways on their computationally inert predecessors, because they include executable code cells alongside basic text formatting and figure display. Blocks of code can be run and rerun one at a time, in any order, making individual computational steps very easy to debug, and making it possible not only to reproduce the original research but also iteratively to test re-orderings and other modifications to the process.[26]

Computational notebooks have become exceedingly popular amongst data scientists in recent years, a trend that is due in part to the Jupyter project, which brings the notebook format to some of the most important programming languages in data science: Julia, Python, R and others.[27] In a testament to the exploding popularity of the format, the code-sharing site GitHub reported that the number of Jupyter notebooks uploaded to its servers increased from 200,000 in 2015 to 2.5 million in 2018.[28] With its rapidly expanding user base of computational scientists, the Jupyter project has become an important site for data science tooling, and in particular for visualization tools, which derive special benefit from having crucial user interface elements already in place: functions that return images (or any graphics) see them displayed immediately below;[29] images can be saved to disk simply by dragging them to the desktop or into a folder; and Jupyter's support for markdown text formatting means that visualizations can be captioned in place and integrated into passages of text. These and other features allow the developer to focus on the analytical power of the tool rather than spending time building the user interface from scratch. It is because of these distinct advantages—a reproducible research record, a large user base, a native data science environment, and a pre-existing user interface—that I chose to develop ivpy with Jupyter notebooks in mind.

The IVPY Plotting Functions

From the perspective of the user, ivpy is five plotting functions, together with submodules for feature engineering and icon generation. These represent the essential operations of an iconographic visualization system: the generation of icons, the assignment of icons to positions in abstract feature spaces, and the subsequent placement of icons in plots. Icon placement is handled by the plotting functions, which form the core of the module; indeed, the submodules for feature engineering and icon generation are not strictly necessary, in case the user already has in hand a dataset of icons and features.

Ivpy instantiates a declarative unit visualization grammar built on the notions of a data axis and a coordinate system.[30] Users specify a zero-, one-, or two-axis plot by function choice: zero-axis plots are chosen by calling montage(), one-axis plots by calling histogram(), and two-axis plots by calling scatter(). For each axis, the user specifies a corresponding column from the data table using the 'xcol' and 'ycol' keyword arguments. It's important to keep in mind that a data axis is not the same as a spatial dimension: although the plot types differ in their numbers of data axes, all occupy two spatial dimensions. Additionally, although the histogram is treated as a one-axis plot, its axis is always binned and therefore distinct from the axes of the scatterplot, which in ivpy, though binnable, are continuous by default. The addition of an axis to a plot will mean in any case that data items are assigned to fixed positions along its extent. The affixing of items to axis positions is distinct, then, from mere sorting, because sorting positions are relative and not fixed. And so although montages do not have genuine axes, their items can be sorted, if desired, using 'xcol'. Similarly, although items in a single histogram bin share their horizontal axis positions, they can also be sorted vertically—using 'ycol'—despite the fact that the histogram lacks a vertical plotting axis.

All three plot types have both rectilinear and circular forms, which for montages are simply square and circular shapes, while for histograms and scatterplots, they are Cartesian and polar coordinate systems. There is also a special form of montage, show(), which is optimized for the notebook format by having its width matched to a notebook cell. Faceting is made possible by compose(), which takes plots as arguments and returns a montage of plots. Users can either call compose() directly or specify a faceting column in any plotting function call, which will then return a faceted plot using compose().[31] The latter approach is more efficient but can only deliver a single outcome: a plot grouped into facets by values for a single (often categorical) data variable. If the facets are to be distinguished in some other way, the user must call compose() directly.

The inclusion of compose() in the module makes it easy to generate plots of plots—'second-order' plots, perhaps, or 'metaplots'—but in fact, because ivpy reads its plotting units from image files, any plotting function can use plots as units, provided they are saved to disk. It's worth dwelling on this fact. Ivpy has a 'pure' unit visualization grammar, because its drawing engine (if used at all) is wholly separate from its compositional operations. Users can create and modify icons using the module, but these are saved to disk before being used in plots, and the plotting functions work the same regardless of the units they receive. In fact, ivpy's plotting functions are indifferent even to icon size, because all icons are thumbnailed to the same (adjustable) size before plotting.[32] Ivpy's grammar, we might say, is wholly separate from its 'lexicon', which as a result is effectively infinite, comprising any graphical objects that can be saved as image files.

Often, though not always, the choice of lexical items—i.e., the plotting units—will be made in advance of using ivpy, because each data record will have an image to represent it, and the user need only tell the plotting functions where to find these images. But even in such cases, the functions are capable of modifying the units prior to plotting, either by printing the index of the associated data record in the upper left corner or by printing an annotation along the bottom. Annotations are fixed by the user's choice of 'notecol', which can be any column in the data table. Because indices and annotations are simply printed as text, they allow the user to add information locally to each unit without having to learn a new graphical encoding.

All other unit-making operations—e.g., icon generation—happen outside the plotting context. But the plotting functions additionally play a role in deciding which data records make it to the screen at all. Minimally, the user must assign a sequence of filepaths to 'pathcol', but this sequence can be randomly sampled by assigning an integer value to 'sample'. This is useful if the dataset is very large or if the user needs to fix the number of items across multiple plots. If axes are added to a plot, axis ranges can be controlled using 'xdomain' and 'ydomain'. It's possible, using these arguments, to plot to a range larger than the data range, which allows for the fixing of axis ranges in faceted plots.

Feature Engineering in IVPY

The scope of the plotting functions thus extends beyond mere unit placement. But most of the unit-making and data operations happen prior to plotting, and were it not for specialized submodules for icon generation and feature engineering, most would happen outside ivpy altogether, either in pure Python or in one of Python's existing data science libraries. Because ivpy is designed primarily for image analysis, icon generation is the least developed of ivpy's operations, and apart from a single example in the next section, I'll have little else to say about it here.

I use the term 'feature engineering' in a quite broad sense to mean any sort of data transformation whatever, since all yield the same thing: an abstract space in which data records can be compared. All such spaces provide the analyst with distinct ways of modeling her data, which for ivpy users will mean, ultimately, selecting a particular set of images and positioning them in a plot. The particular forms of feature engineering that ivpy packages into functions are fixtures of computer vision research: image feature extraction, dimension reduction, and clustering.

Ivpy provides wrappers for a variety of Python-based image feature extractors, including basic visual properties like hue, saturation, and brightness, as well as both entropy and standard deviation of brightness; textural features derived from the gray-level co-occurrence matrix (GLCM) including contrast, dissimilarity, homogeneity, angular second moment (ASM), energy, and correlation;[33] and what we might call 'neural similarity': the 2048-dimensional output vector of the penultimate layer of ResNet50,[34] an artificial neural network trained on ImageNet.[35] Ivpy also provides wrappers for three familiar dimension reduction algorithms, implemented in Python's scikit-learn library:[36] principal component analysis (PCA); t-distributed stochastic neighbor embedding (t-SNE);[37] and uniform manifold approximation and projection (UMAP).[38] Finally, ivpy wraps under a single function call nine different clustering algorithms, again using scikit-learn implementations, including k-means, agglomerative hierarchical clustering, DBSCAN,[39] mean shift,[40] and others.

There is nothing strictly necessary about ivpy's providing these wrappers, since the user, working in a Python environment, already has access to the original functions. But ivpy makes the feature engineering process more efficient and accessible by unifying these operations under single, concise APIs. If the user wishes to extract neural similarity from her images, for example, she can do so simply by calling extract('neural'), and similarly for all other features. If she wishes to find clusters of images in a similarity space defined by those features, she can call cluster(X), where X is the matrix containing all and only the extracted features, and the function will return, by default, four clusters of images, found by the k-means algorithm. Alternative parameter settings can be chosen by keyword arguments.

More substantively, ivpy provides a set of functions for hand-tuning machine clusters, a capability that is of particular importance in ivpy's target domains. In such domains, machine classifications are mere suggestions, to be evaluated and possibly adjusted by human expertise. Ivpy users can move items or groups of items from one cluster to another; cut items or groups of items from clusters without reassigning them or even remove entire clusters altogether; merge multiple clusters into a single cluster; initialize a new cluster; and swap the cluster assignments of two items or groups of items. To help this process along, ivpy also makes it easy to measure, for each item in a cluster, what we might call its 'centrality', or how far it is from the center of its cluster, making it easy to identify the likeliest candidates for reassignment.

Expressiveness and Data Manipulation

Feature engineering increases ivpy's 'lexical' expressiveness—the diversity of atomic meanings composable by its grammar. We've said that ivpy's lexicon is effectively infinite, comprising all possible images, and we might say, similarly, that feature engineering yields infinities of higher cardinality, because it decomposes single images into many constituent 'words'. And although it's true that the user can and will do this on her own, by looking, she can't do it without looking, and in most cases, she can't look at everything. Feature engineering helps to guide the process of looking by identifying lexical properties of the data that can be used both to narrow the search space and to make patterns more legible.

None of the graphical tools discussed earlier make it possible to increase lexical expressiveness in this way. While it's true that both PixPlot and VIKUS Viewer rely on feature engineering, they offer far fewer options, and the engineering takes place outside the graphical environment, as a preprocessing step. Ivpy thus has greater lexical expressiveness than any of these tools. It also has greater syntactical expressiveness than any of these tools, both because it can produce a wider range of plot types and because it makes those types usable with any data you choose. But I don't want to hang the argument for ivpy's utility entirely on the gains in expressiveness that result from these advantages, because any of the existing graphical tools could be augmented to offer additional plot types or full service, in-browser feature engineering. What they can't do—or what it would take a monumental and probably ill-advised effort to do—is allow for the kind of data manipulation that data scientists do regularly in their native working environments. If I want, for example, to join new data to my existing data, use string operations to correct typos in filenames, or use complex conditional filtering to select some subset of my data, I will need to migrate from the graphical environment back to my native working environment. The graphical user interface expects a single, fully cleaned and processed dataset—something you'll never have in the early stages of research, if at all. This is why ivpy is necessary.

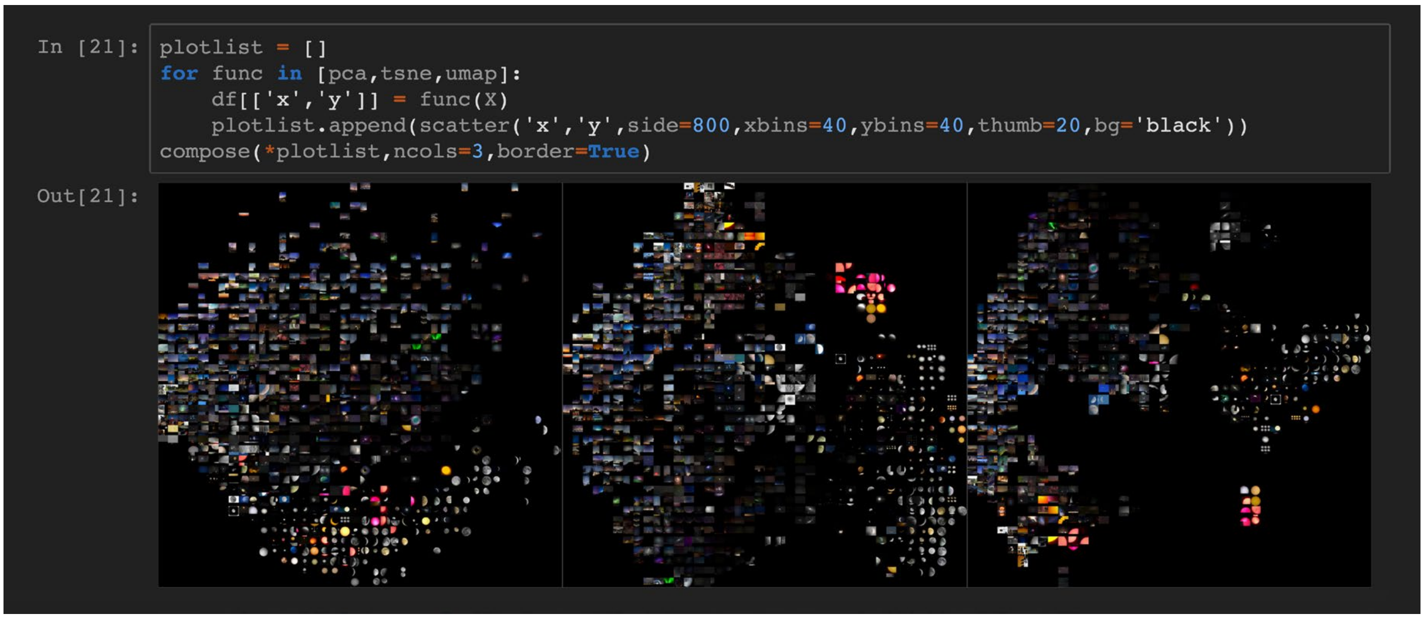
I want to be clear that I don't see ivpy as a direct competitor with graphical tools. I'm certain that as time goes on, they will narrow the expressiveness gap, and their greater efficiency and accessibility make them attractive to anyone who doesn't want to write code. But there will remain a need for direct visualization in native working environments, and ivpy is currently the only high-level tool in existence that makes it possible.

Example of Use

In this section, I'll walk through some examples designed to help illustrate both what is possible when basic module elements are combined and what sorts of analyses these combinations support. There are, of course, countless ways the module can be used, and I will here cover only a tiny fraction. The examples are chosen both for their familiarity to me and out of a desire to cover as much ground as possible in a handful of examples.

The goal of the first example is to place images in a visual similarity space. There are as many such spaces as there are kinds of visual similarity, and as we've seen, ivpy makes a variety of these spaces available to the user. But many are structured by very specific kinds of visual similarity—e.g., average hue—and for this example, we'd like as general a measure as possible. What we above called 'neural similarity' tends to work well as a general measure, likely because it makes possible the machine recognition of a vast range of familiar objects—viz., the label set of ImageNet, which includes over 20,000 object categories. For this reason, neural similarity is commonest amongst the features used to sort images in contemporary image visualization applications, including, as we've seen, both PixPlot and VIKUS Viewer.[41]

In this example, we've extracted a neural feature vector for each of a collection of astrophotography images found in the Yahoo! Flickr Creative Commons 100M dataset.[42] The vectors populate a matrix, assigned to 'X'. Figure 1 is a screenshot from a Jupyter notebook and contains a single code cell along with its output, a triptych of image scatterplots. The plots are generated in a loop, each pass of which uses a different algorithm to compress the 2,048 dimensions of X to just two. The 'side', 'thumb', 'xbins', and 'ybins' keyword arguments are used to force the resulting continuous plotting coordinates to a grid, so that all visible images are fully visible. Images stack



```
In [21]: plotlist = []
         for func in [pca,tsne,umap]:
             df[['x','y']] = func(X)
             plotlist.append(scatter('x','y',side=800,xbins=40,ybins=40,thumb=20,bg='black'))
         compose(*plotlist,ncols=3,border=True)
Out[21]:
```

Figure 1. Three image scatterplots, each using a different dimension reduction algorithm, are generated in a loop and added to a list, which is then expanded and passed to compose()

atop each other, with the topmost image fully occluding all beneath it.[43] As they are generated, the plots are appended to a list, and the expanded list is then passed to compose().

The resulting metaplot allows us to see the effects of applying three different forms of dimension reduction to the same vector space. The leftmost plot, which uses PCA, has the evenest spread of the three, but its groupings do not appear as sharp as the other two. The rightmost plot, which uses UMAP, has an interesting shape and successful groupings, but its spread is so uneven that many images are hidden. The middle plot, which uses t-SNE, balances the competing virtues of grouping quality and spread. Figure 2 shows the result of recreating the middle plot at a much larger size and saving it to disk. The pattern of analysis used here—testing a number of analytical possibilities by generating plots in a loop—is perfectly general and can be used to visualize the effects of differential parameter settings of all kinds.

The second example, like the first, begins by extracting a neural feature vector for each of its images—here, digitized watercolors from the Yale Center for British Art's collection—and compressing the resulting vector space to two dimensions, here using UMAP. We saw in Figure 1 that UMAP produces an interesting global shape but encourages occlusion. For the purposes of drawing a 'map' of visual similarity, as in the previous example, the tendency to occlude is a problem, but we can recover the occluded images by applying a clustering algorithm to UMAP's similarity space and subsequently plotting each cluster as a simple montage.

Figure 3 demonstrates one way of doing this. We use ivpy's cluster() function to add a column of cluster assignments to our data table, 'df';[44] we then measure, for each cluster, the centrality of each of its members, adding the resulting sequence as another column; finally, we specify a circular montage, faceted by cluster assignment and sorted radially by centrality. In each facet, the cluster's most prototypical images are plotted in the center; its outliers are plotted in the periphery. Figure 4 zooms in on several of these clusters, revealing how neatly our method has separated the images into visual style groups. It's important to note that no metadata is being used to group the images, and the clusters are not modified in any way. The groupings result from the plain application of neural similarity, dimension reduction, and clustering. Remarkably, although the leftmost clusters in Figure 4 very clearly share a visual style, we've managed to split images in this style into plant and animal clusters. In the rightmost clusters of Figure 4, we've similarly separated the natural from the built environment, again using only visual similarity as a guide.

The first two examples illustrate ways that a researcher might begin looking at her data in order to get a sense of its global structure. But these techniques can figure in more targeted analyses as well. The image collection used in the second example is the result of a massive digitization effort by the Yale Center for British Art, and attached to the collection is a trove of human-generated metadata about the works. This metadata can be used together with measures of visual similarity to reveal aesthetic patterns across time periods,

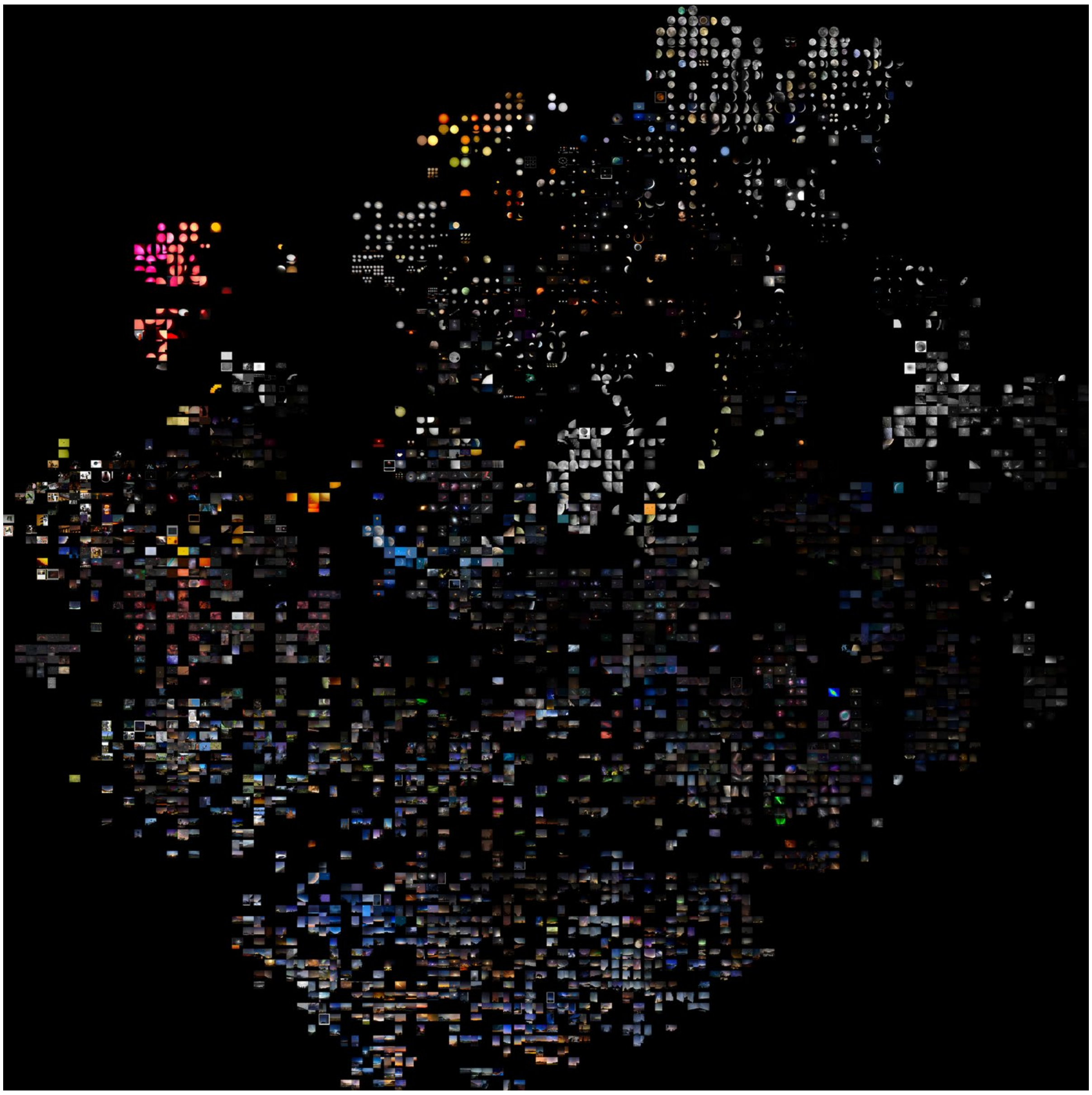
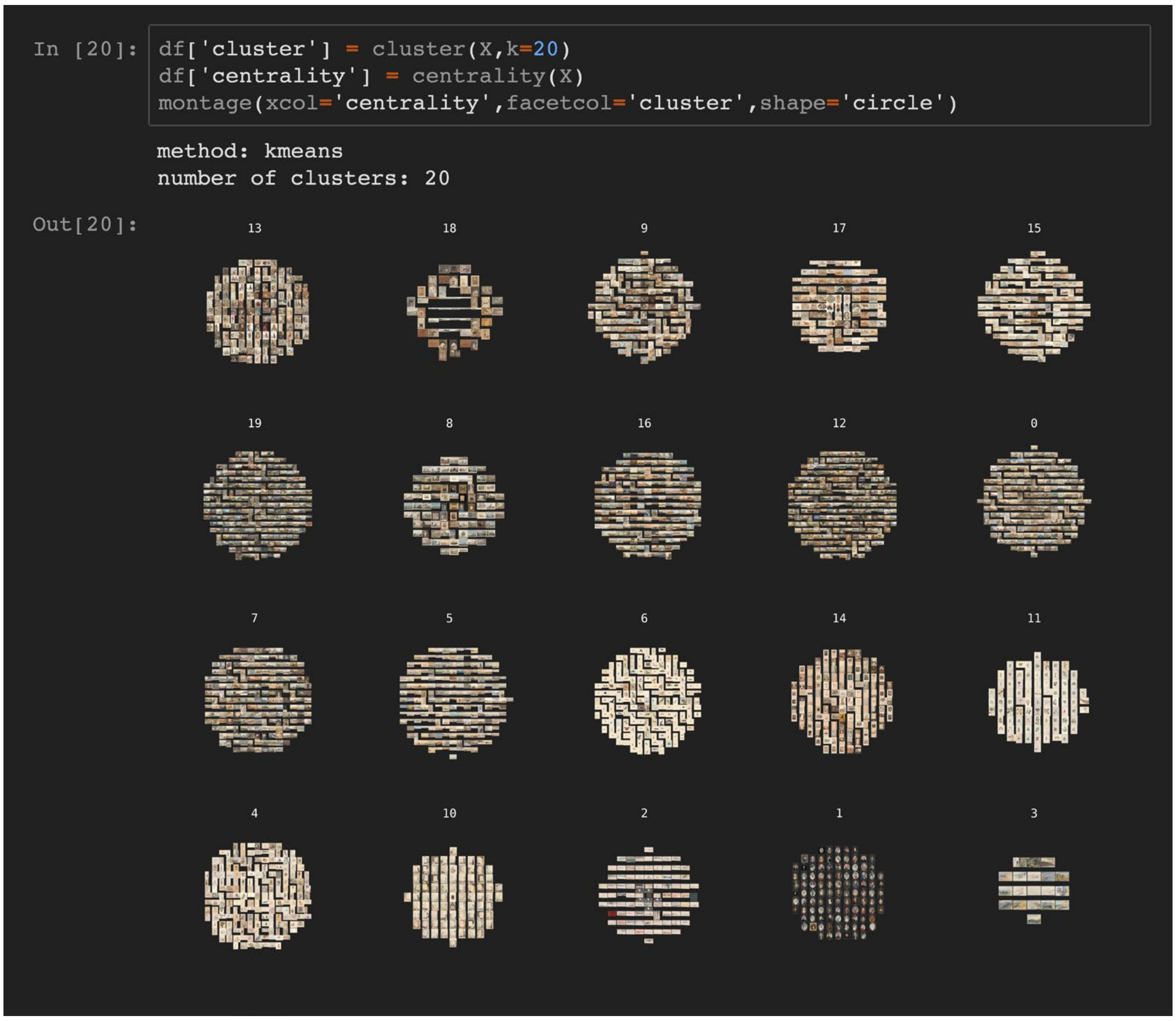


Figure 2. Astrophotography images posted To Flickr, arranged by visual similarity using an artificial neural network and the t-SNE dimension reduction algorithm.

artists' bodies of work, artistic media and materials, and even depicted subject matter. Say, for example, that we are interested in which subject concepts are the most visually diverse. We could start with a visual similarity 'map', as in Figure 2, and then, using the subject concept tags contained in our metadata, create a map for each concept showing only its instances. It would be trivially easy to see, for each concept, the degree of its spread across the global visual landscape, thereby discovering which concepts have the greatest visual diversity (according to the chosen measure). There are countless such ways we might use these techniques to better understand our data, and targeted analyses like these derive special benefit from ivpy's embedding in Python, because the often complex data manipulations they require can be carried out alongside the plotting operations.

Like the previous two, the third example, shown in Figure 5, uses neural similarity as a basic relation between images, but unlike the previous two, there is no attempt to represent the totality of these relations in a single plot. Instead, we begin with an image of interest—here, a particular photogram by the artist László Moholy-Nagy—and proceed to find its thirty nearest neighbors, which are then plotted as a simple montage using show(). These steps are packaged into a single function call, nearest(), because the nearest neighbor search itself requires several lines of code, and I wanted to make the technique as efficient and accessible as possible, given its considerable utility.[45]

Our distance metric is the one used to measure the centrality of cluster members: Euclidean distance in the high-dimensional space defined by the image collection's neural similarity vectors. The target image is plotted first, in the upper left, and the rest follow in reading order. The resulting plot is a rather satisfying validation of the metric, as the first six or so neighbors appear to feature the very same object (whatever it is). Neighbors further down the list all contain something that explains their close relationship to the target,



```
In [20]: df['cluster'] = cluster(X,k=20)
         df['centrality'] = centrality(X)
         montage(xcol='centrality',facetcol='cluster',shape='circle')

         method: kmeans
         number of clusters: 20

Out[20]:
```

Figure 3. Watercolors from the Yale Center for British Art's collection are grouped into clusters by visual similarity.

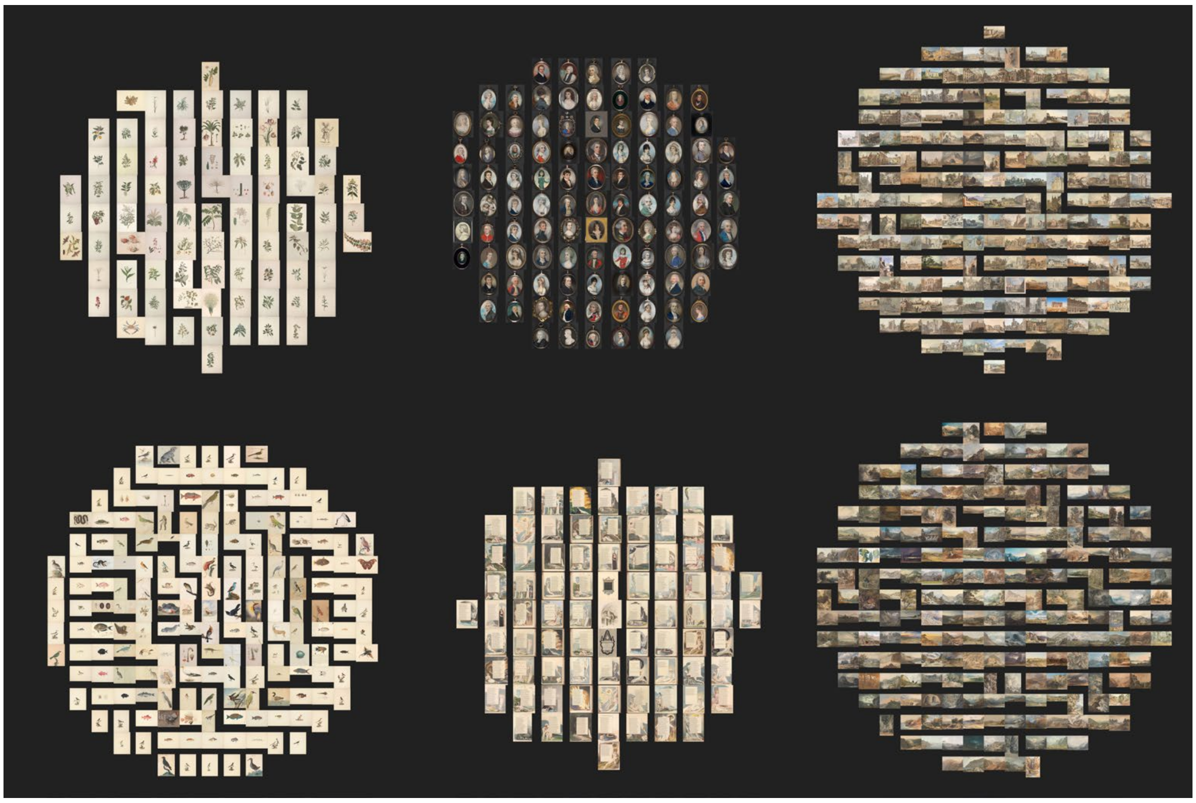
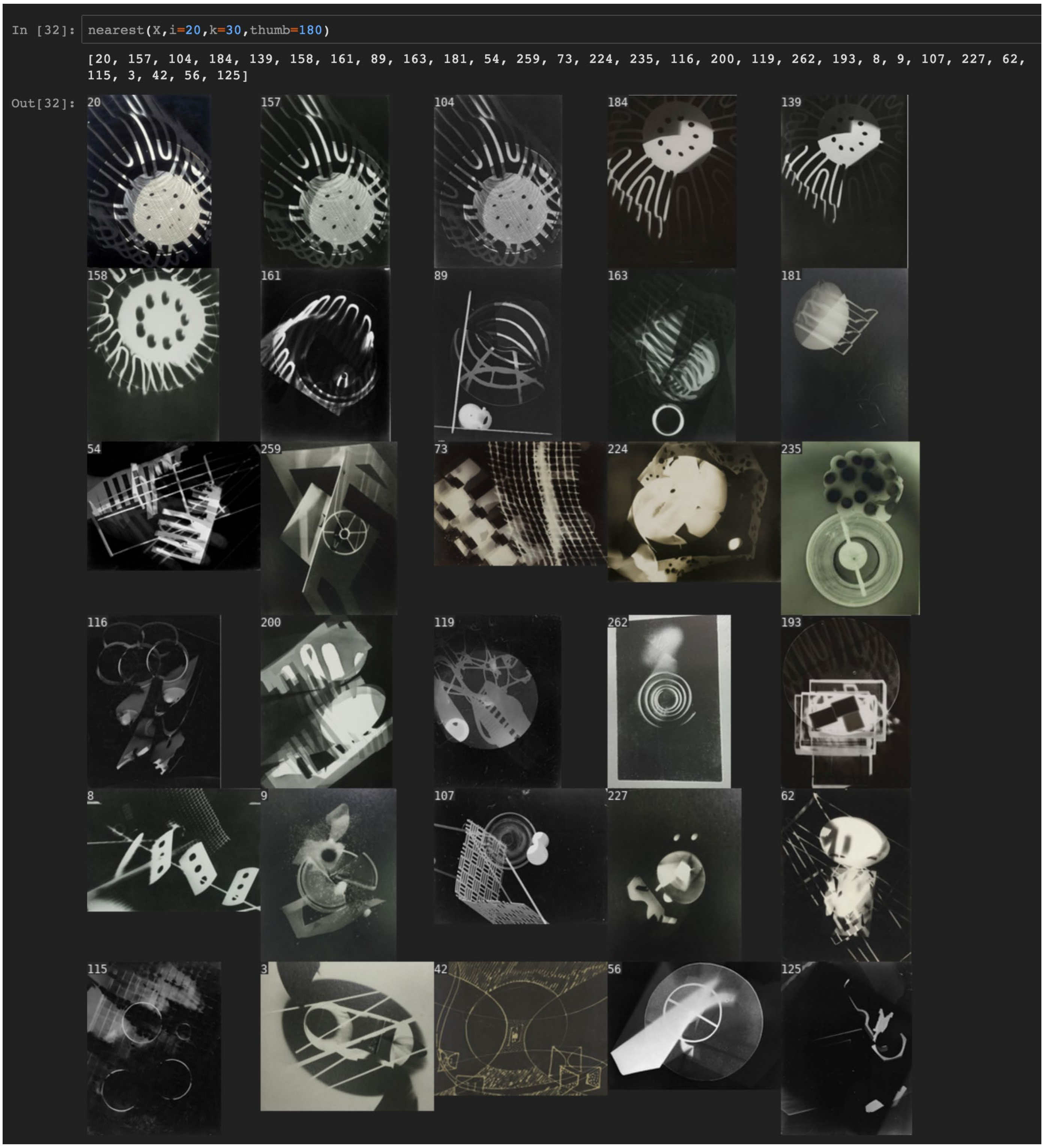


Figure 4. Zoom detail of six clusters from Fig. 3.

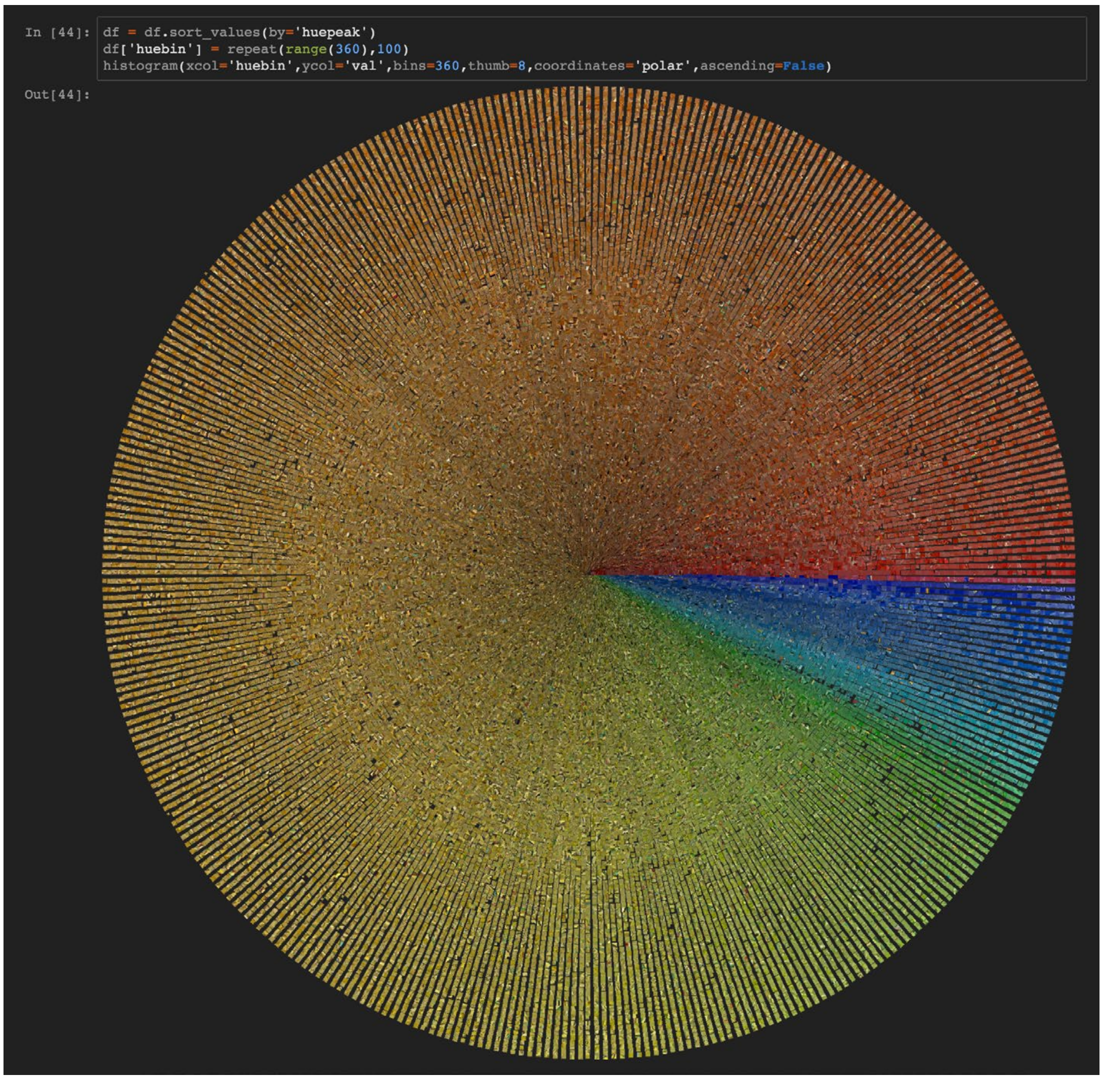


```
In [32]: nearest(X,i=20,k=30,thumb=180)

         [20, 157, 104, 184, 139, 158, 161, 89, 163, 181, 54, 259, 73, 224, 235, 116, 200, 119, 262, 193, 8, 9, 107, 227, 62,
         115, 3, 42, 56, 125]

Out[32]:
```

Figure 5. The 30 nearest neighbors of a photogram by artist László Moholy-Nagy, arranged in reading order by their similarity with the target image, which appears first, in the upper left. In addition to the plot, the function returns a list containing each item's data table index, which is also printed in the upper left corner of each thumbnail.



```
In [44]: df = df.sort_values(by='huepeak')
         df['huebin'] = repeat(range(360),100)
         histogram(xcol='huebin',ycol='val',bins=360,thumb=8,coordinates='polar',ascending=False)
Out[44]:
```

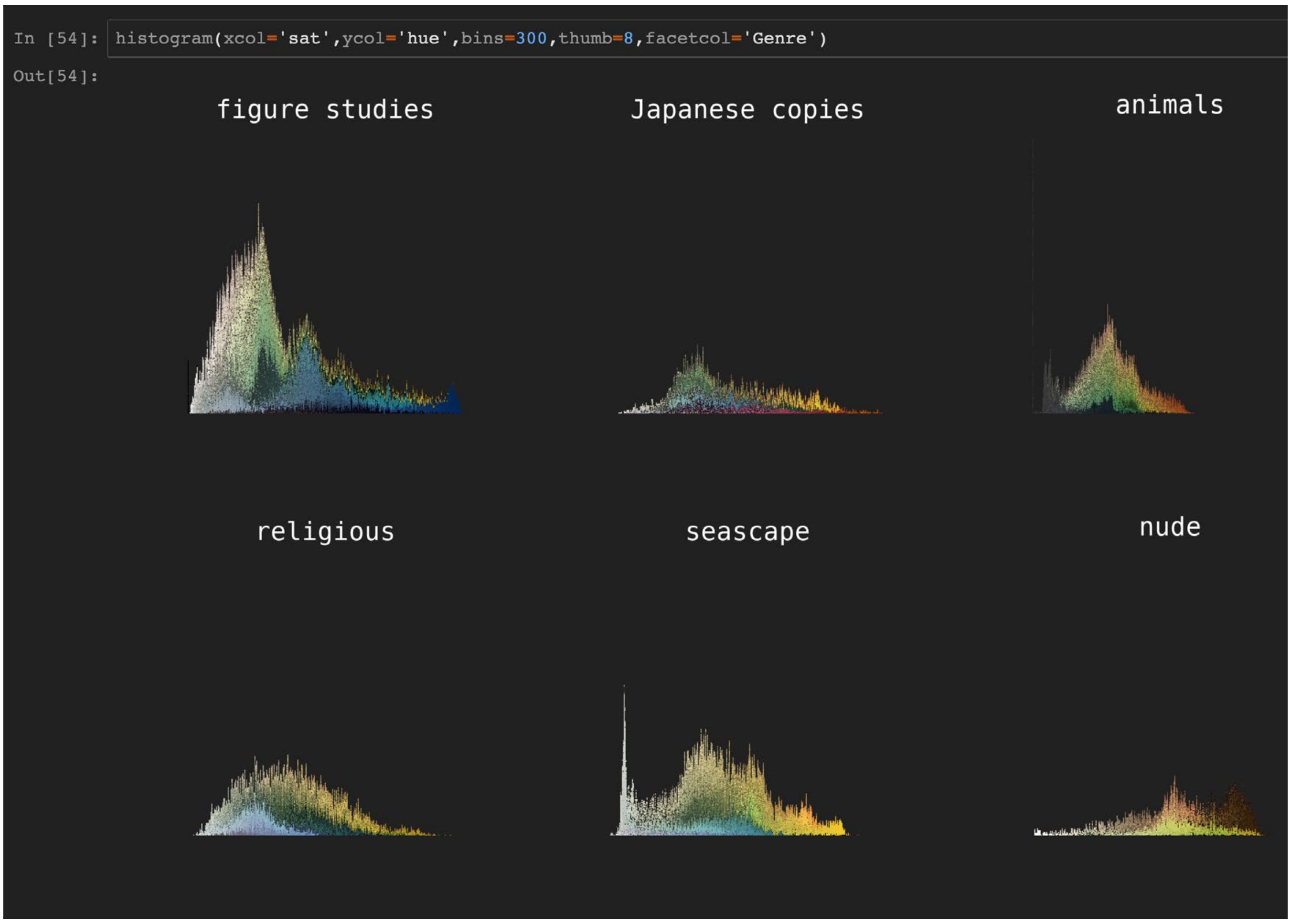
Figure 6. Small fragments of digitized van Gogh paintings are arranged as a polar histogram with bins of equal height, sorted angularly by hue and radially by brightness.

generally something circular, something trellis-like, or both. The table index of each image is printed in its upper left corner, and the entire neighbor set is returned as text, appearing just above the plot. Users can copy this text and assign it to a variable, so that the set is easily selectable for further analysis. One might, for example, use this technique to build clusters of images from a set of chosen prototypes, rather than allowing an algorithm to choose the cluster centers.

The neural feature vectors used in the previous three examples allow us to compare the visual contents of images in as comprehensive a way as possible, because, as we've said, the unanalyzed measure of similarity they encode is what grounds the successful machine recognition of over 20,000 object categories. But even if we were to select a set of images defined by a single object category, visualizations using only those images would nonetheless contain objects in other categories as well. In direct visualizations, sorting and filtering may bring certain contents to the foreground, but they do not close off our perceptual access to the background. Direct visualizations are thus semantically richer than their sorting and filtering parameters might suggest, a circumstance which might be undesirable in certain cases.

If a more targeted semantics is what we want, we need to remove foregrounded contents from their background contexts. There are sophisticated ways of doing this for particular object classes, although ivpy does not use them.[46] Ivpy does, however, offer a utility—shatter()—for slicing images into fragments of roughly equal size, which, depending on the chosen fragment size, may succeed in isolating the contents of interest. Technically, the slicing operation is a form of feature engineering, since it transforms our data and expands our lexicon, but because the resulting fragments are plain image files, they cannot be used to narrow the search space or make patterns more legible unless we first extract their features.

In the fourth example, we do just that. A collection of digitized van Gogh paintings is sliced into fragments small enough that their basic visual properties—hue, saturation, and brightness—are roughly uniform. At this size, object categories are no longer legible, although certain fragments might contain hints about their origins. What's left is a clear look at the collection's color palette, which can be difficult to discern when the fragments are embedded in their original contexts. In Figure 6, a bright, saturated subset of the fragments is plotted as a 'flat' polar histogram, sorted angularly by hue and radially



```
In [54]: histogram(xcol='sat',ycol='hue',bins=300,thumb=8,facetcol='Genre')
Out[54]:
```

Figure 7. Small fragments of digitized van Gogh paintings are grouped into genres and plotted as Cartesian histograms, sorted horizontally by saturation and vertically by hue.

by brightness.[47] Although flat histograms, with bins of equal height, are not natively specifiable in ivpy, the code cell in Figure 6 shows how two lines of Python can be used to coerce histogram() to produce them.

The plot in Figure 6 is composed of fragments from every image in the collection and thus reveals van Gogh's overall hue palette, dominated by oranges and yellows. In Figure 7, we've used collection metadata to split the fragments into six genre groups, chosen here for the distinctiveness of their palettes. Where previously we were interested only in hue and thus used only those fragments with legible hues, we now add back all of the blacks, whites, and greys in order to render a complete color palette for each genre. Each palette is a Cartesian histogram with 300 saturation bins of equal width, sorted vertically by hue.[48] Genres with greater representation in the collection will have correspondingly larger histograms, as they are composed of more fragments.

The analytical possibilities of a fragment plot will depend a great deal on the choice of fragment size: the larger the fragments, the easier it is to recover their origins and the less likely they are to isolate particular contents of interest. If we use larger fragments here, we might be better able to explain the dominance of oranges and yellows by identifying what they are used to depict, but we might fail to notice that dominance in the first place. We can circumvent this tradeoff by tracing fragments back to their source images and even back to their pixel positions, both of which are recoverable if shatter() is used to create the fragments.

In each of the previous examples, our visual lexicon is delimited by a source collection of images or image fragments and whatever numerical features can be derived from them. So although the diversity of visualizations expressible by the plotting grammar will depend in part on our ingenuity—in feature engineering, data manipulation, etc.—the plotting units themselves are given to us essentially fully formed. That this is the default mode for ivpy means that its codebase can remain relatively lean without affecting its expressiveness. Nearly all visualization systems include (or borrow) a core drawing module out of necessity, and most of the time, ivpy can get by without one. But if our data are not images, and we thus cannot visualize them directly, we'll have to do so indirectly, and we'll therefore need to draw our own plotting units.

In the final example, we visualize a small set of photographic prints using glyphs as our plotting units.[49] The plot itself, shown in Figure 8, is a simple montage, specified using show(). Each glyph represents a particular print made from the same photographic negative, and accordingly, we are interested not in the images themselves (which are identical) but in the photographic papers, whose properties are mapped to the axes of a modified radar chart. There are four axes, each encoding what is called an 'expressive dimension' of the paper: thickness on the left, color on top, gloss on the right,

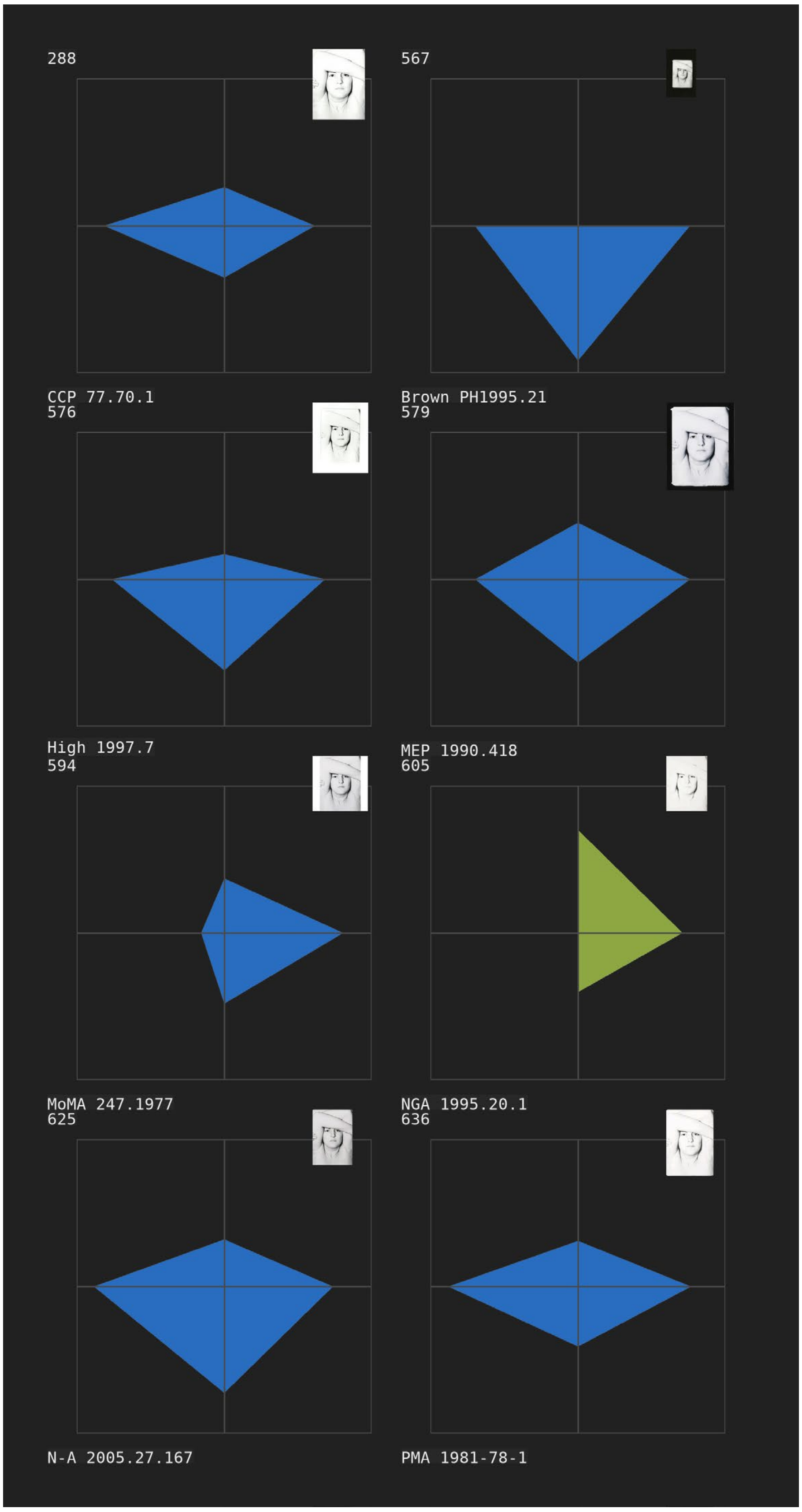


Fig. 8. Eight photographic prints by Harry Callahan, each from the same negative, are represented as compound glyphs. A thumbnail of each print is affixed to the upper right, while the expressive dimensions of their photographic papers are mapped to the axes of a modified radar chart. Along the bottom is an annotation that contains both the print's holding institution, as well as an institutional index called an 'accession number'. Indices for the local data table are printed in the upper left.

and texture on the bottom. Each property value is mapped to an axis location, and these points form the vertices of a polygon, whose fill color encodes another data variable—here, a Boolean, although it could likewise be categorical or continuous. Additionally, the table index is printed in the upper left (as in the third example); an annotation is printed along the bottom; and a small thumbnail of the print is affixed to the upper right.

These are compound glyphs, each containing three distinct forms of representation: direct, glyphic, and textual. This considerable density of information and diversity of presentational formats is appropriate in this context, because our aim is to make detailed comparisons among a small set of individually important items. Each component serves a distinct purpose: the thumbnail image tells us which negative was used to make the print; the annotation tells us which collecting institution holds the print and gives us its accession number, allowing us to trace the print back to its holding institution's database; the radar chart tells us about the expressive dimensions of the paper it is printed on and thus allows us quickly to identify differences in the ways the prints will appear to the viewer; the color of the polygon tells us whether the print is 'vintage', or contemporary with its negative; and the table index gives us a path back to our own data table, should we need it. When the number of plotting units is so few, glyphs like these can be read without difficulty, but in case that number is drastically increased, we might opt to remove the annotations, indices, thumbnails, and even the bounding box, leaving only the radar chart. In ivpy's glyph module, all of these elements can be toggled on or off.

At present, glyphs of this type are the only ones ivpy can generate. They are designed for relatively high-dimensional data; indeed, the design of the radar chart generalizes to arbitrarily many dimensions, and ivpy allows the user specify any number of radar axes. The choice to focus ivpy's glyph generation on high-dimensional data was made in part because low-dimensional glyph plotting is handled adequately by existing tools, but also because low-dimensional glyphs—e.g., circles of different sizes and colors—are easy enough to generate and save to disk using the Python Imaging Library directly. But development of ivpy is ongoing, and future versions will likely include support for a wider range of glyph types.

Limitations

These examples serve to illustrate both ivpy's considerable expressive range and its relative efficiency and accessibility when compared with low-level, imperative specification of iconographic visualizations. But no tool can optimize for every use case, and ivpy is no exception. I'll finish with a discussion of ivpy's limitations, including those that reflect deliberate design choices, as well as those that point to areas for future development.

Ivpy is the sole occupant of the space between low-level drawing libraries like Processing and the Python Imaging Library on the one hand and high-level graphical tools like PixPlot and VIKUS Viewer on the other. And we might ask why there are no others in this space. Of course, the total space of available tools is quite small, and thus we shouldn't expect any particular tool to have a great many neighbors, but all available tools have collected at the poles, choosing to optimize either for expressiveness or for efficiency and accessibility. This is not insignificant. In choosing to sit between the poles, ivpy makes a pair of sacrifices: it is less expressive than the drawing library that powers it, and, relative to graphical tools, a greater effort is required to use it. Throughout, I've made a case for ivpy's utility in spite of these sacrifices—in short, that its middle position also yields a pair of advantages—but I'd like, for a moment, to focus on the sacrifices themselves. What, precisely, do we give up?

On one side, we give up polish. The space of possible graphic customizations is simply too large for ivpy to cover in full. And while it's true that, for example, ivpy's plotting functions could be modified to allow for greater customization, there is an associated risk of making the module seem difficult to learn. And because ivpy courts a user base that includes researchers in the humanistic sciences, many of whom are new to programming, its seeming difficult to learn would be a liability. Ivpy's relatively simple conceptual scheme is a deliberate choice, and I'm not open to changing it. But this means that for highly specialized forms of analysis or highly stylized graphic outputs, ivpy will, at the very least, need to be supplemented with additional programming in a low-level grammar or additional manipulation with tools like Adobe Photoshop.

On the other side, we give up speed. While it's true that graphical tools are marginally easier to learn than ivpy's function calls, their chief advantage lies in allowing users to move quickly through a series of analytical steps, keeping pace with their thoughts. I discussed above a set of functions that allow the user to modify cluster memberships in various ways. But let these functions be as learnable and powerful as you like, they can't match the user experience of clicking and dragging images between clusters and seeing them update in real time, like sorting Polaroids on a tabletop. To be fair, no such tool currently exists. But if it did, it could only exist in a graphical user interface.

The graphical interface is more efficient—or at least, it can be—because clicking is faster than coding, no matter how concise the grammar. The choice to embed ivpy in a programming language, then, places a speed limit on plot specification. The choice to use image files as ivpy's primary inputs and outputs places similar limitations on performance. While ivpy itself does not limit the number of allowable plotting units, its performance will suffer as the user approaches the hardware limits of her machine. An ivpy plot is a single raster image file that must fit into working memory if it is to be displayed. This is in contrast with, for example, image viewers that display tiled canvases at multiple scales to create the experience of panning and zooming massive images, ones far too large to fit in working memory. An ivpy plot is also made up of raster images, read into memory one at a time. This means that ivpy's plotting speed cannot exceed the read speed of the source hard drive. So, while ivpy can be used to explore and analyze image collections of any size, very large collections will have to be examined in parts small enough to be manageable by the host hardware.

But how should this examination proceed? I've emphasized the importance of the expert's searching 'with her own eyes', but what does that look like for collections too large to examine in full? We here confront, once again, the core difficulty of descriptive and explanatory data science at scale: striking a balance between aimless, random search on the one hand and excessive machine guidance on the other. If we surrender our search entirely to the machine, we cannot escape its biases; if we search aimlessly, we risk finding nothing at all. We know that, at least for image collections of manageable size, direct visualization offers an easy solution: let the human expert look at everything. What to do, then, if she cannot? In the paragraphs above, we discussed those limitations arising from ivpy's basic design characteristics, all of them independently motivated and therefore unlikely to change, despite their negative impacts. But the problem of guided search is one we might mitigate with additional development. And to some extent, we have, by adding to the module various utilities for feature engineering. Feature engineering, as we've said, helps us to narrow the search space, but how can we know whether this narrowing has the effect of closing off promising ways of analysis?

In truth, we can never know for sure, but we can minimize the chances of our missing something important by searching our collections under as many forms of narrowing as possible. And although ivpy already makes available a considerable variety of image feature extractors, it could always offer more. In fact, I believe this to be the most promising direction for ivpy's future development. The problem, then, is that the features with the greatest descriptive and explanatory utility depend essentially on human visual expertise, which is both costly and time-consuming to obtain. Moreover, the very best features in a given domain will come from its experts, who typically have better things to do than to sit at computer terminals labeling images. To date, the most significant effort to collect human labels for artistic images is the Behance Artistic Media (BAM) dataset, which is conceived as a non-photographic alternative to ImageNet.[50] Models trained on its data will therefore expose alternative forms of visual similarity, ones with particular relevance for ivpy's target domains. Relatedly, the most significant effort to train such models comes from Saleh and Elgammal, who develop specialized measures of visual similarity between paintings using the Wikiart dataset.[51] Activity in this space is likely to increase over time, and although ivpy is neither a tool for collecting labels nor a platform for training computer vision models, it can participate in this movement forward by offering ways of visualizing the results of these experiments and comparing them with existing models. Ivpy's embedding in Python is particularly useful in this case, as the vast majority of computer vision is now done in Python.

Ivpy offers an expressive, declarative grammar for iconographic visualization that is easy to learn, easy to use, and, perhaps most importantly, usable in the native working environments of the modern data scientist. I hope that its particular advantages will lead to widespread adoption in its target domains, and that, accordingly, it will help keep human visual judgment at the forefront in these domains, where it belongs.

NOTES

1 The coding patterns used in this module were developed initially in the Cultural Analytics Lab at the University of California, San Diego. During that time, my work was funded by a Frontiers of Innovation Scholarship. The module itself was developed during my postdoctoral associateship in the Digital Humanities Lab at Yale University and can be found at https://github.com/damoncrockett/ivpy.

2 The term 'expressiveness' is being used here in a technical sense—as applied to visualization grammars in the information visualization literature—and is discussed below in the section titled 'Direct Visualization and the Graphical User Interface'.

3 Much, though not all, of this toolkit exists as part of the SciPy project; see Oliphant, "Python for Scientific Computing."

4 Drucker and Fernandez, "A Unifying Framework for Animated and Interactive Unit Visualizations." See also Park et al., "Atom: A Grammar for Unit Visualizations." It might be pointed out that all visualizations are unit visualizations on a suitably general notion of a 'data record'; the distinction between unit and aggregative visualization can therefore be made only in particular contexts of analysis. If the 'atomic' units of analysis, whatever they be, have their own visual marks, we have unit visualization.

5 A list of prominent unit visualizations in the wild appears in Park et al., "Atom: A Grammar for Unit Visualizations."

6 There is a strict reading of 'icon' according to which it must bear some pictorial resemblance to its target—e.g., Borgo et al., "Glyph-Based Visualization."—but I am using the term in the

more permissive sense of Pickett and Grinstein, "Iconographic Displays For Visualizing Multidimensional Data."

[7] Manovich, "What Is Visualization?"

[8] Whether a particular data visualization has exceeded this threshold will depend on a host of contextual factors. For a detailed discussion of this problem as it pertains to iconographic visualization, see Borgo et al., "Glyph-Based Visualization."

[9] Images are of course multiply interpretable, and we might view interpretation as a kind of decoding. But in most cases, even the basic meaning of an icon will depend on arbitrary associations the user has to learn before the visualization can carry any information for her. Put succinctly, icons need a 'legend'; images don't.

[10] I am here exempting image browsers, fixtures of nearly every computer operating system since the 1990s. There are too many to list, and their analytical expressiveness is typically limited to filtering and sorting by metadata properties of the image files (e.g., filename, date created, date modified). Moreover, most are proprietary software and thus their use is restricted in various ways.

[11] Reas and Fry, "Processing."

[12] Abramoff, Magalhães, and Ram, "Image Processing with ImageJ."

[13] The terms 'efficient, 'accessible', and 'expressive', as they are used in this paper, are defined in Bostock, Ogievetsky, and Heer, "D3 Data-Driven Documents."

[14] For discussion of these techniques, see Manovich, "Media Visualization." The ImagePlot software itself, implemented as a macro for ImageJ, can be found at http://lab.softwarestudies.com/p/imageplot.html.

[15] Google's People + AI Research Initiative (PAIR) has developed a tool called Facets Dive that is similar in some respects to PivotViewer, although its functionality is more limited, and it is designed specifically for exploring machine learning datasets. It offers interactive specification of iconographic plots that can be faceted by data values. See https://pair-code.github.io/facets/.

[16] Kräutli and Boyd Davis, "Revealing Cultural Collections Over Time."

[17] For the PixPlot source code, visit https://github.com/YaleDHLab/pix-plot.

[18] PixPlot's feature extraction uses the Inception network; see Szegedy et al., "Going Deeper With Convolutions."

[19] PixPlot's dimensional compression uses Uniform Manifold Approximation and Projection, or UMAP; see McInnes, Healy, and Melville, "UMAP."

[20] For the VIKUS Viewer source code, visit https://github.com/cpietsch/vikus-viewer. For the project website, visit https://vikusviewer.fh-potsdam.de/.

[21] VIKUS's dimension reduction uses t-distributed Stochastic Neighbor Embedding, or t-SNE; see Maaten and Hinton, "Visualizing Data Using T-SNE."

[22] Arguments here and in the next section draw on remarks made by Hadley Wickham in his 2017 IEEE VIS keynote, "You Can't Do Data Science in a GUI". To be clear, the argument is not that PixPlot and VIKUS Viewer ought to be made more expressive and thus better suited to data science; rather, it is that graphical user interfaces are simply the wrong place to do data science, however they are designed. I discuss this further in the section below, titled 'Expressiveness and Data Manipulation'.

[23] https://www.numpy.org/

[24] https://pandas.pydata.org/

[25] Pedregosa et al., "Scikit-Learn."

[26] It should be pointed out that certain exploratory behaviors like cell editing and nonsequential cell execution can undermine the record-keeping function of the notebook. This tension is discussed at length in Rule, Tabard, and Hollan, "Exploration and Explanation in Computational Notebooks."

[27] Ragan-Kelley et al., "The Jupyter/IPython Architecture."

[28] Perkel, "Why Jupyter Is Data Scientists' Computational Notebook of Choice."

[29] Ivpy functions return instances of the Python Imaging Library's 'Image' class. Ivpy's source code uses Pillow, a popular fork of the Python Imaging Library; see https://pillow.readthedocs.io/.

[30] Declarative grammars are popular in information visualization, because they allow users to specify visualizations by declaring their properties, rather than by giving explicit instructions for building them. Declarative grammars tend to be both more efficient and more accessible than imperative grammars. For a discussion of visualization grammars, see Park et al., "Atom: A Grammar for Unit Visualizations." The authors also contribute a declarative unit visualization grammar very different in design from the one presented here. Though I make no explicit rejection of the Atom grammar in ivpy's design, it wouldn't work as an ivpy grammar. For example, because my plotting units are usually images, I can't control their colors—and thus cannot signal group memberships using color—a constraint that would make unavailable a considerable portion of Atom's expressive space.

[31] Faceted plots are sometimes called 'small multiples' or 'trellis' plots; see Becker, Cleveland, and Shyu, "The Visual Design and Control of Trellis Display." I borrow the language of 'faceting' from ggplot2; see Wickham, "A Layered Grammar of Graphics."

[32] Ivpy's grammar is in this way distinct from Atom (Park et al., "Atom: A Grammar for Unit Visualizations."). Differential unit sizes are possible in Atom, and Atom's filling and packing logic therefore needs to be unit-aware in a way ivpy's compositional operations do not.

[33] All of the preceding extractors—some of which are used by ivpy in novel ways—are found in scikit-image; see Walt et al., "Scikit-Image."

[34] He et al., "Deep Residual Learning for Image Recognition." Ivpy uses ResNet50, but any successful architecture trained on ImageNet will perform similarly.

[35] Deng et al., "ImageNet."

[36] Pedregosa et al., "Scikit-Learn."

[37] Maaten and Hinton, "Visualizing Data Using T-SNE."

[38] McInnes, Healy, and Melville, "UMAP."

[39] Ester et al., "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise."

[40] Fukunaga and Hostetler, "The Estimation of the Gradient of a Density Function, with Applications in Pattern Recognition."

[41] Other notable examples of neural similarity being used (along with dimension reduction) to sort images include Free Fall, Curator Table, and t-SNE Map, all of them Google Arts & Culture Experiments by Cyril Diagne, Nicolas Barradeau, and Simon Doury. See https://experiments.withgoogle.com/collection/arts-culture.

[42] Thomee et al., "YFCC100M."

[43] The problem of occlusion is sometimes resolved by special gridding algorithms that move occluded images to nearby open grid locations, but because such algorithms necessarily introduce error, ivpy does not use them. See, for example, Mario Klingemann's RasterFairy: https://github.com/Quasimondo/RasterFairy.

[44] The name 'df' is the one conventionally given to instances of the DataFrame object in pandas, a Python library for managing

tabular data. See https://pandas.pydata.org/. Ivpy makes extensive use of the pandas library; in fact, when ivpy users call plotting functions, they are required to pass pandas objects to any 'column' arguments—'pathcol', 'xcol', 'ycol', 'facetcol', and 'clustercol'. Many ivpy functions return pandas objects in order to satisfy this requirement.

45 Ivpy uses annoy, a C++ library with Python bindings for approximate nearest neighbor search, written by Erik Bernhardsson. See https://github.com/spotify/annoy.

46 Mask R-CNN, for example, can identify the precise boundaries of objects of particular types—e.g., cars, trees, traffic lights, pedestrians. See He et al., "Mask R-CNN."

47 Flat histograms are analytically useful only if composed of distinguishable units—only if, that is, they are iconographic histograms.

48 Plots like these are called 'slice histograms' in Crockett, "Direct Visualization Techniques for the Analysis of Image Data."

49 The prints are made by photographer Harry Callahan and each belongs to a different collecting institution.

50 Wilber et al., "BAM! The Behance Artistic Media Dataset for Recognition Beyond Photography."

51 Saleh and Elgammal, "Large-Scale Classification of Fine-Art Paintings."

BIBLIOGRAPHY

Abramoff, M. D., Paulo J. Magalhães, and Sunanda J. Ram. "Image Processing with ImageJ." Article. Biophotonics international, 2004. http://dspace.library.uu.nl/handle/1874/204900.

Becker, Richard A., William S. Cleveland, and Ming-Jen Shyu. "The Visual Design and Control of Trellis Display." *Journal of Computational and Graphical Statistics* 5, no. 2 (June 1, 1996): 123–55. https://doi.org/10.1080/10618600.1996.10474701.

Borgo, Rita, Johannes Kehrer, David H. S. Chung, Eamonn Maguire, Robert S. Laramee, Helwig Hauser, Matthew Ward, and Min Chen. "Glyph-Based Visualization: Foundations, Design Guidelines, Techniques and Applications," 2013. http://dx.doi.org/10.2312/conf/EG2013/stars/039-063.

Bostock, M., V. Ogievetsky, and J. Heer. "D^3 Data-Driven Documents." *IEEE Transactions on Visualization and Computer Graphics* 17, no. 12 (December 2011): 2301–9. https://doi.org/10.1109/TVCG.2011.185.

Crockett, Damon. "Direct Visualization Techniques for the Analysis of Image Data: The Slice Histogram and the Growing Entourage Plot." *International Journal for Digital Art History* 0, no. 2 (October 18, 2016). https://doi.org/10.11588/dah.2016.2.33529.

Deng, J., W. Dong, R. Socher, L. Li, and and. "ImageNet: A Large-Scale Hierarchical Image Database." In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–55, 2009. https://doi.org/10.1109/CVPR.2009.5206848.

Drucker, Steven M, and Roland Fernandez. "A Unifying Framework for Animated and Interactive Unit Visualizations." *Microsoft Research*, 2015.

Ester, Martin, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise." *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 1996.

Fukunaga, K., and L. Hostetler. "The Estimation of the Gradient of a Density Function, with Applications in Pattern Recognition." *IEEE Transactions on Information Theory* 21, no. 1 (January 1975): 32–40. https://doi.org/10.1109/TIT.1975.1055330.

He, Kaiming, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. "Mask R-CNN," March 20, 2017. https://arxiv.org/abs/1703.06870v3.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition," 770–78, 2016. http://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html.

Kräutli, Florian, and Stephen Boyd Davis. "Revealing Cultural Collections Over Time." London, 2016. http://researchonline.rca.ac.uk/1725/.

Maaten, Laurens van der, and Geoffrey Hinton. "Visualizing Data Using T-SNE." *Journal of Machine Learning Research* 9, no. Nov (2008): 2579–2605.

Manovich, Lev. "Media Visualization." In *The International Encyclopedia of Media Studies*. 2012. https://doi.org/10.1002/9781444361506.wbiems144.

Manovich, Lev. "What Is Visualization?" Accessed December 13, 2018. https://publishup.uni-potsdam.de/frontdoor/index/index/docId/5047.

McInnes, Leland, John Healy, and James Melville. "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," February 9, 2018. https://arxiv.org/abs/1802.03426v2.

Oliphant, T. E. "Python for Scientific Computing." *Computing in Science Engineering* 9, no. 3 (May 2007): 10–20. https://doi.org/10.1109/MCSE.2007.58.

Park, D., S. M. Drucker, R. Fernandez, and N. Elmqvist. "Atom: A Grammar for Unit Visualizations." *IEEE Transactions on Visualization and Computer Graphics* 24, no. 12 (December 2018): 3032–43. https://doi.org/10.1109/TVCG.2017.2785807.

Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, et al. “Scikit-Learn: Machine Learning in Python.” *Journal of Machine Learning Research* 12, no. Oct (2011): 2825–30.

Perkel, Jeffrey M. “Why Jupyter Is Data Scientists’ Computational Notebook of Choice.” *Nature 563* (October 30, 2018): 145. https://doi.org/10.1038/d41586-018-07196-1.

Pickett, R. M., and G. G. Grinstein. “Iconographic Displays For Visualizing Multidimensional Data.” In *Proceedings of the 1988 IEEE International Conference on Systems, Man, and Cybernetics*, 1:514–19, 1988. https://doi.org/10.1109/ICSMC.1988.754351.

Ragan-Kelley, M., F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, and M. Bussonnier. “The Jupyter/IPython Architecture: A Unified View of Computational Research, from Interactive Exploration to Communication and Publication.” *AGU Fall Meeting Abstracts* 44 (December 1, 2014): H44D-07.

Reas, Casey, and Ben Fry. “Processing: Programming for the Media Arts.” *AI & SOCIETY* 20, no. 4 (September 1, 2006): 526–38. https://doi.org/10.1007/s00146-006-0050-9.

Rule, Adam, Aurélien Tabard, and James D. Hollan. “Exploration and Explanation in Computational Notebooks.” In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 32:1–32:12. CHI ’18. New York, NY, USA: ACM, 2018. https://doi.org/10.1145/3173574.3173606.

Saleh, Babak, and Ahmed Elgammal. “Large-Scale Classification of Fine-Art Paintings: Learning The Right Metric on The Right Feature.” *ArXiv:1505.00855 [Cs]*, May 4, 2015. http://arxiv.org/abs/1505.00855.

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. “Going Deeper With Convolutions,” 1–9, 2015. https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Szegedy_Going_Deeper_With_2015_CVPR_paper.html.

Thomee, Bart, David A. Shamma, Gerald Friedland, Benjamin Elizalde, Karl Ni, Douglas Poland, Damian Borth, and Li-Jia Li. “YFCC100M: The New Data in Multimedia Research.” *Communications of the ACM* 59, no. 2 (January 25, 2016): 64–73. https://doi.org/10.1145/2812802.

Walt, Stéfan van der, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. “Scikit-Image: Image Processing in Python.” *PeerJ* 2 (June 19, 2014): e453. https://doi.org/10.7717/peerj.453.

Wickham, Hadley. “A Layered Grammar of Graphics.” *Journal of Computational and Graphical Statistics* 19, no. 1 (January 1, 2010): 3–28. https://doi.org/10.1198/jcgs.2009.07098.

Wilber, Michael J., Chen Fang, Hailin Jin, Aaron Hertzmann, John Collomosse, and Serge Belongie. “BAM! The Behance Artistic Media Dataset for Recognition Beyond Photography,” 1202–11, 2017. http://openaccess.thecvf.com/content_iccv_2017/html/Wilber_BAM_The_Behance_

DAMON CROCKETT is the Principal Data Scientist at the Institute for the Preservation of Cultural Heritage at Yale University. He manages the institute’s data science projects, focusing on data integration, visualization, interpretability, and communication. Prior to joining the institute, he was a Postdoctoral Associate in Computer Science at Yale, working in the Digital Humanities Lab. He has also worked as a researcher at the Center for Data Science and Public Policy at the University of Chicago; the Institute for Pure and Applied Mathematics at UCLA; and the Cultural Analytics Lab at UC San Diego, where he graduated in 2015 with a Ph.D. in Philosophy and Cognitive Science.

Correspondence email: damoncrockett@gmail.com