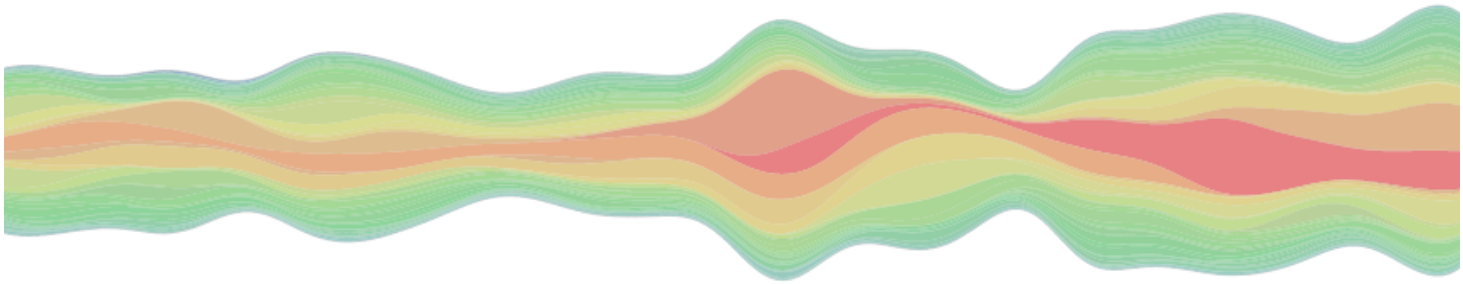
CPSC 490, Senior Project

An Extensible Package for the Creation of Generalized "Streamgraph" Visualizations In Ruby

Author Bay Gross
Advisor Brad Rosen

December 9, 2018



1 Abstract

The past few years have witnessed a steady rise in the realm of *personal analytics*: the quantitative study and visualization of personal data. New technologies supporting both the capture and presentation of personal data have helped spur this growth, with some of the most popular visualization products taking advantage of social data via platforms like Facebook, Twitter, and Foursquare. Here we focus on one rich but relatively unexamined data source: SMS message history. Specifically, we detail the design and implementation of a generalizable graphing utility called "Streamgraph", with SMS data as a case study. While there are many potential data presentations that can be layered on top of texting data, we focus on the Streamgraph because of its computational complexity, aesthetic novelty, and potential for popular appeal. We will briefly discuss a utility for mining the (otherwise unattainable) text-message data off of iPhones, and then cover in detail the implementation and customization of our Streamgraph visualization software.

2 Background and Inspiration

2.1 The Rise of Personal Data

Personal data is growing as a trend in consumer products, in part driven by new data API's such as the Facebook Open Graph. Several companies have sprung up around this niche, with devices like *FitBit* being marketed as a way to collect and analyze your life and movements. Nicholas Feltron, a designer at Facebook, has become a brand name in this realm for the creation of his "Annual Reports" that offer highly granular and highly designed summaries of his day-to-day life. Google now generates monthly infographics representing your search and mail activity upon demand, and millions of otherwise privacy-conscious consumers submit their banking information to the third-party site Mint.com to visualize their spending habits in real-time. And yet, for all of its wealth of information, SMS texting data has remained relatively unexplored, in part due to its limited exposure. The iPhone suite, for instance, does not provide any native interface for extracting your texting data - either for backup or analysis.

2.2 New Data Visualizations

In 2008, the New York Times published a novel infographic based on a relatively new graphing style, called *Stacked Graph*. The chart showed an elongated horizontal "stream", with many individual layers that grew and shrunk to represent tha aggregated box office revenues of over seven thousand movies from the past two decades. In a paper later that year, the authors Byron and Wattenberg discuss the design considerations and precedent for their new chart. The StackedGraph approach was especially interesting in its commitment to the aesthetic, a far jump from the standard histograms and line plots of statistical analysis. Many of the conclusions they reached when optimizing for this movie data drive the design of our own text-based visualization.

While the majority of the inspiration and research for this project comes from the StackedGraph paper, some broader conclusions and heuristics are drawn from its predecessor: *Theme River*, published in 2000. ThemeRiver is a similar, but less-optimized, graphing approach that uses "colored currents" to "facilitate the identification of trends, patterns, and unexpected occurrence or non-occurrence of themes or topics". Both graphing styles aim to clearly and efficiently represent individual data series as well as their aggregate.

3 SMS Extraction and Mining

An individual's SMS history can hold a wealth of information. While the message content itself is an especially granular and comprehensive source of data, the aggregation of texts in the abstract can also inform on the relationships, lifestyle, and interactions of an individual over time. We will specifically focus on how the frequency and magnitude of texting can be viewed as a proxy for following relationships longitudinally. Unfortunately Apple does not natively allow an iPhone user to extract his or her own text history. There are some commercial products that will do just this for a small fee, and if you are reasonably comfortable with scripting languages there are a few outdated projects around the web that can (to some degree) mine the texts out of older iOS backups. One side deliverable of this project, then, is an extensible ORM *Object Relational Model* (built on top of the ruby Active Record gem) for directly interfacing with your iOS 6+ message backups (stored across multiple SQLite files in a user's OSX preferences folder). The ORM allows for simple exports to CSV or JSON formats - for backup or analysis. It is, after all, *your* data! A readme and image guide is provided alongside the codebase to step through this process, although the task itself is relatively painless and user-friendly.

4 Algorithms and Design Considerations Behind Streamgraph

Here we discuss the design considerations behind our main deliverable, the Streamgraph generator.

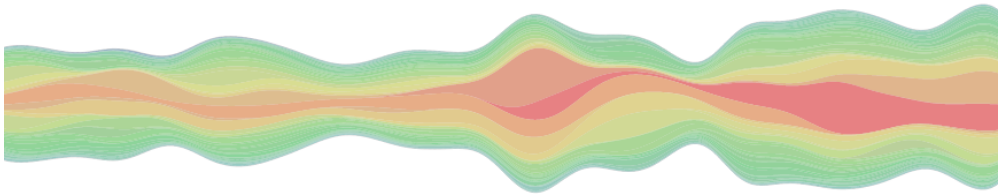


Figure 1: Streamgraph, with axis unlabeled. The Streamgraph is an aggregation of many individual continuous data streams, in this case text message frequency over time. The individual streams are smoothed, stacked, and colored according to a variety of heuristics to best communicate data about the individual trends and their relationship with the aggregate stream.

4.1 Interpolation and Smoothing

The first task in generating a Streamgraph is to smooth the discrete data for each contact history into a continuous stream. With small-range datasets it may sometimes be necessary to interpolate across the series, but for most text histories of at least a few months this is unnecessary. Instead, all that is required is a blurring operation. Our Streamgraph module accepts a blurring function and coefficient, and accordingly modulates that function across the data. By default the module provides a linear "triangle" modulation function or a basic Gaussian curve. See Figure 2 for a comparative view of this process.

4.2 Ordering Layers

After the layers are blurred, they must be stacked. While the order of stacking can simply be random, it comes as no surprise that intelligently sorting the layers can lead to significant increases in legibility and aesthetic. To start, the Streamgraph module provides several basic metrics for sorting: popularity (or total text count), mean texting date for contact, median texting date for contact, or start date for the contact. While start date can be valuable in some scenarios (such as the movie release chart studied by the StackedGraph paper), it can be misleading or confusing when applied to texting data. Rather, because a texting 'relationship' does not necessarily start at first text, our metric of a 'weighted start date' comes in to play and provides for a cleaner, more intuitive ordering by time. Even more useful are the metrics for 'volatility' and 'burstiness'. Here we use volatility to mean the variance of the layer, when calculated across the discrete data series. Burstiness is similar, though slightly more subtle in nature. Rather than weigh for variance alone, our Burstiness metric attempts to cluster and identify 'bursts' within an individual layer, and then apply weighting to these bursts themselves. The result is that relationships that would otherwise appear as "volatile" because they start late in the timeline or are just prone to some fluctuation are separated out from the more 'bursty' layers that spike in and out in just a few finite spurts. A visual analysis of these styles is provided in Figure 3. As the graphic will illustrate, each 'sorting' metric can then further be ordered top to bottom, bottom to top, inside out, or outside in, with the latter two being strictly more useful in generating clear Streamgraphs.

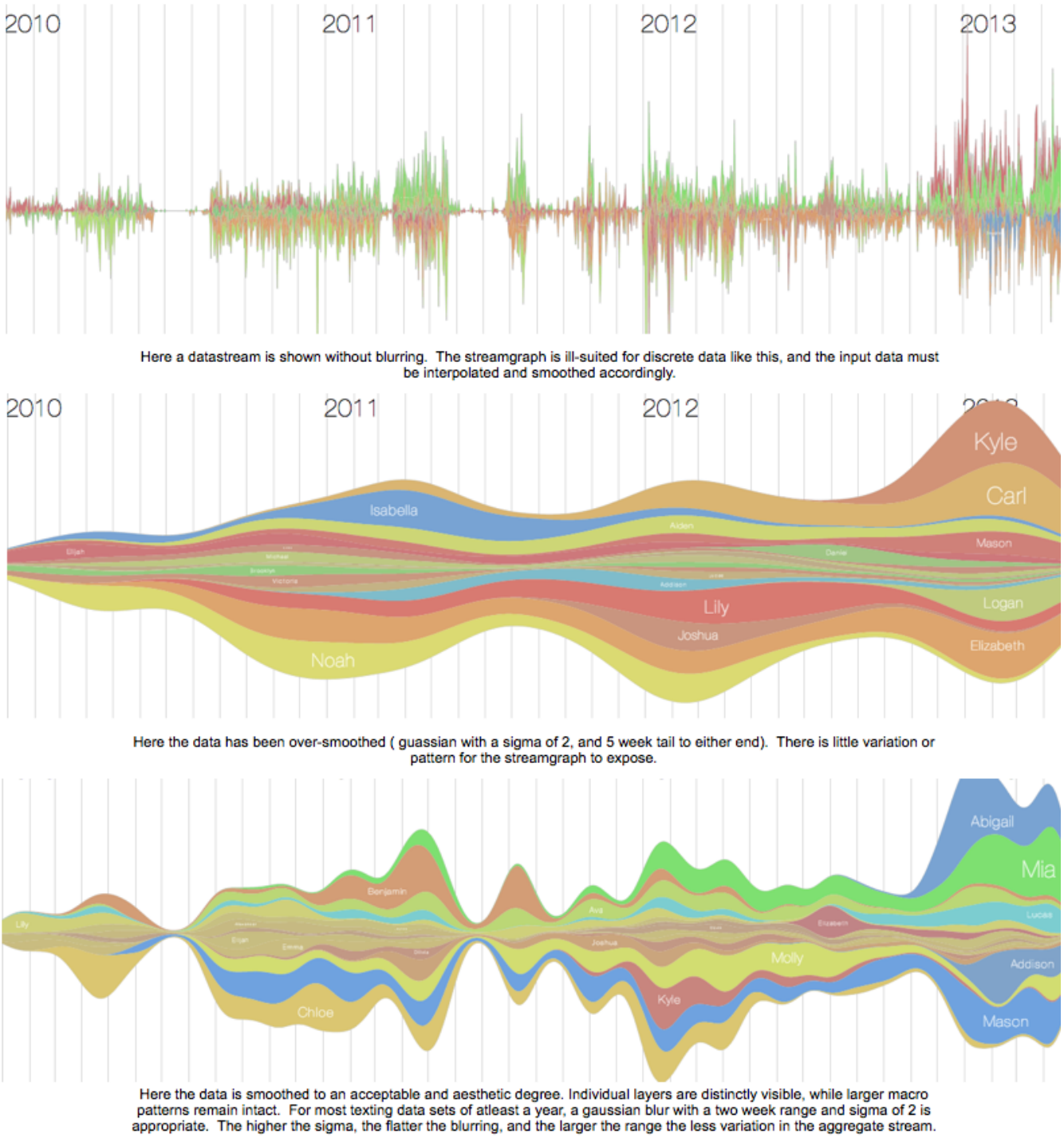
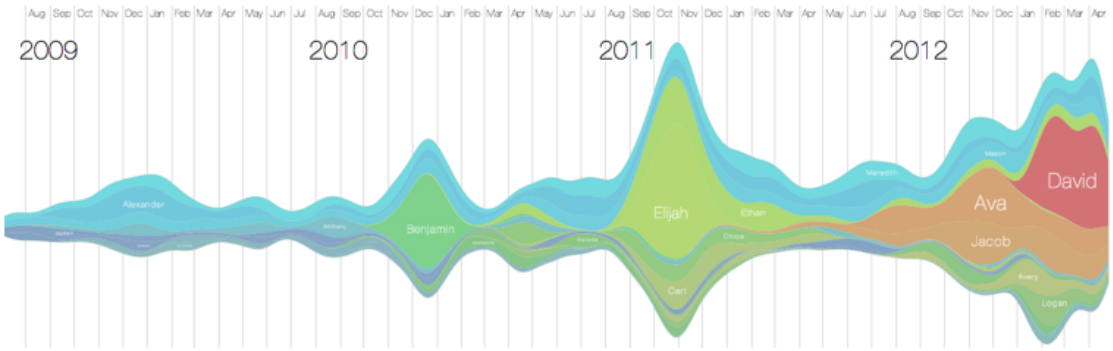
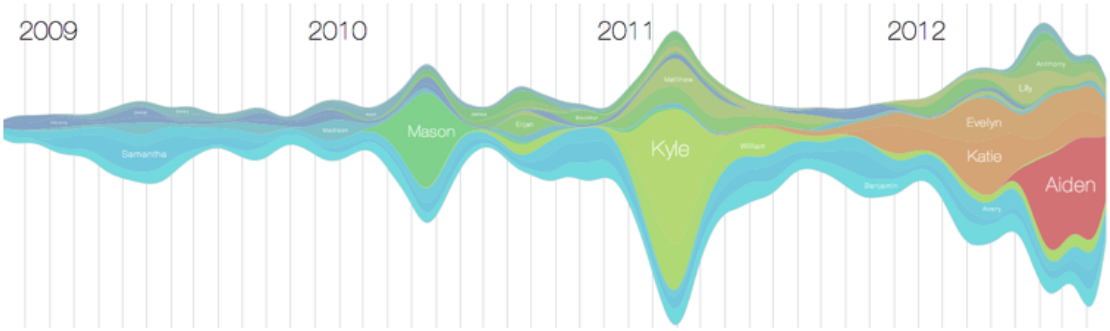


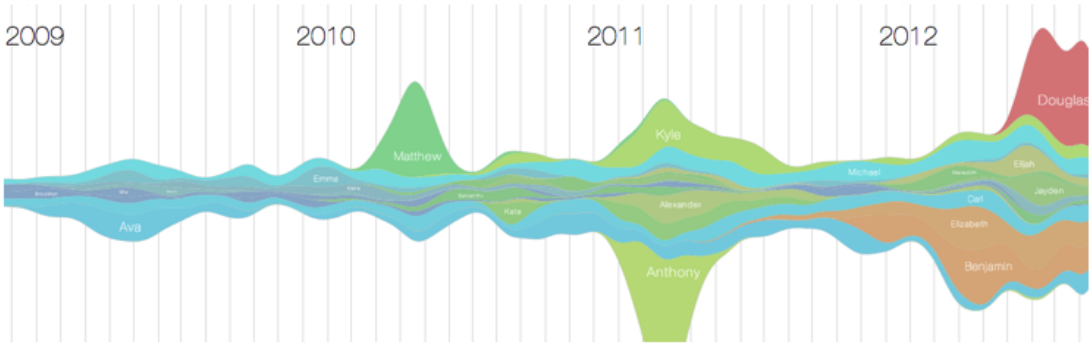
Figure 2: Blurring the discrete text-history data for use in the Streamgraph.



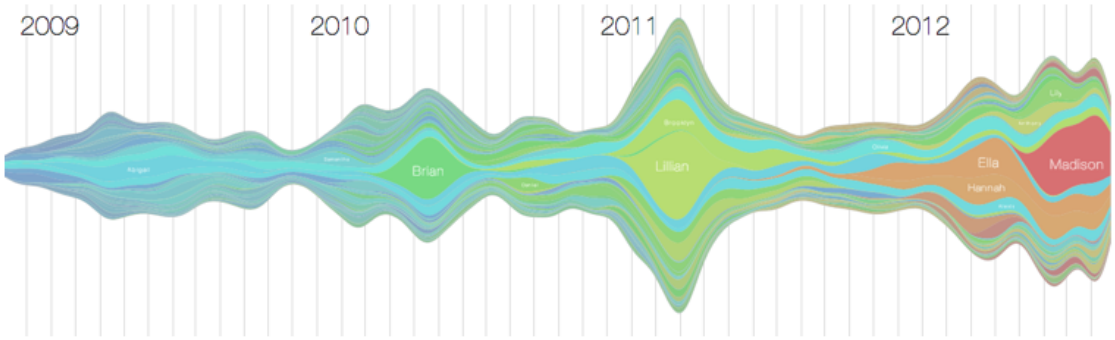
Sorted by total number of texts, naively ordered top to bottom. The outer edge layers (even those with high aggregate counts) are negatively effected by the volatility inside, making individual streams difficult to read.



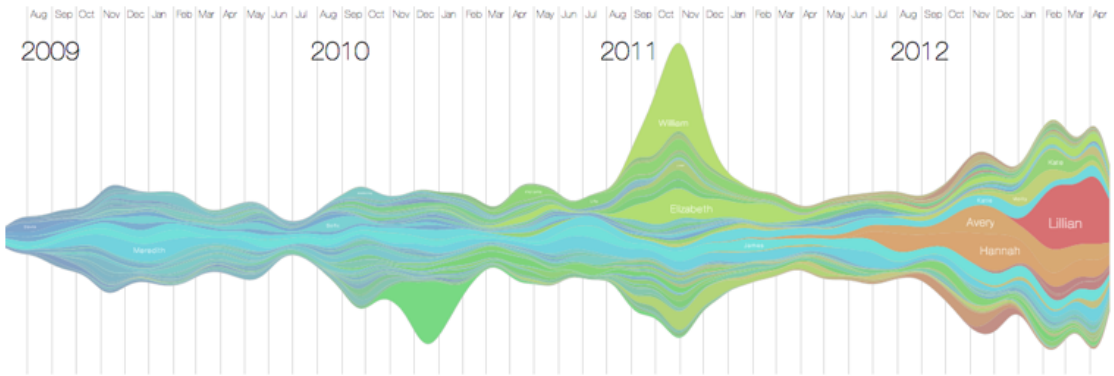
Again sorted by popularity, this time inside out. While in some cases this will help highlight the most "important" or popular layers, it can still be subject to the same volatility and distotions as above.



Here, we are sorting by volatility (a simple calculation of variance agross each layer's data), and pushing those most volatile layers to the outside, where they are less able to effect neighboring streams.



As n increases, it can make aesthetic sense to move the most volatile layers back into the center, as the smallest layers along the tail blend together and can appear too crunched or crowded when forced together. This ordering also emphasizes the effect a strong relationship can have on the periphery: visually edging other layers to the side.



Here we have sorted by our custom metric of "burstiness". Individual 'bursts' of activity within each layer are identified, clustered, and weighed. These burst-scores then contribute to a sorting that acheives many of the same desireable effects as sorting by volatility, without forcing new (but otherwise non-volatile) relationships to the periphery.

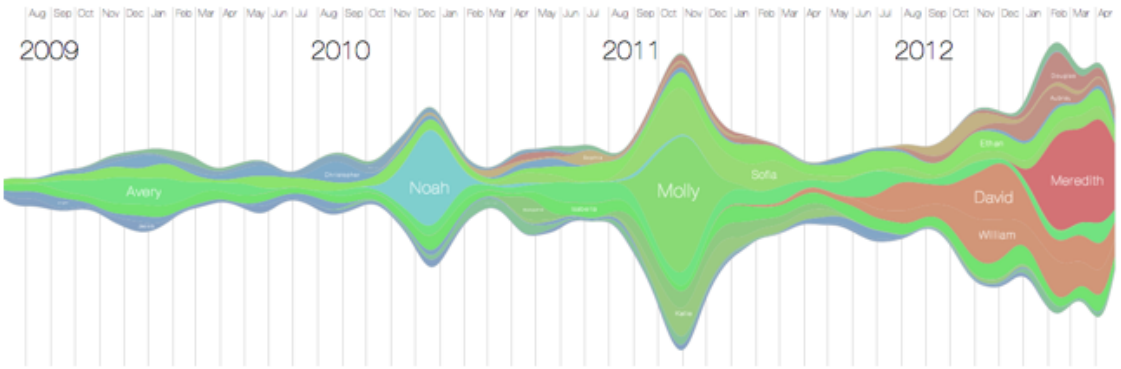
4.3 Coloring

Intelligently coloring each layer was a key focus and insight of the Byron paper. We follow their lead by exposing hue and saturation (within certain bounds) as variable ranges that fluctuate to reflect insights in the data. In general, all the same metrics available for sorting are available for coloring, and can be applied to either saturation or hue. The overall ranges for hue and saturation can also be customized for any given design. In general, the color values are linearly distributed according to the log transformation of each layer on the chosen metric. Examples are seen and discussed in detail within Figure 4.

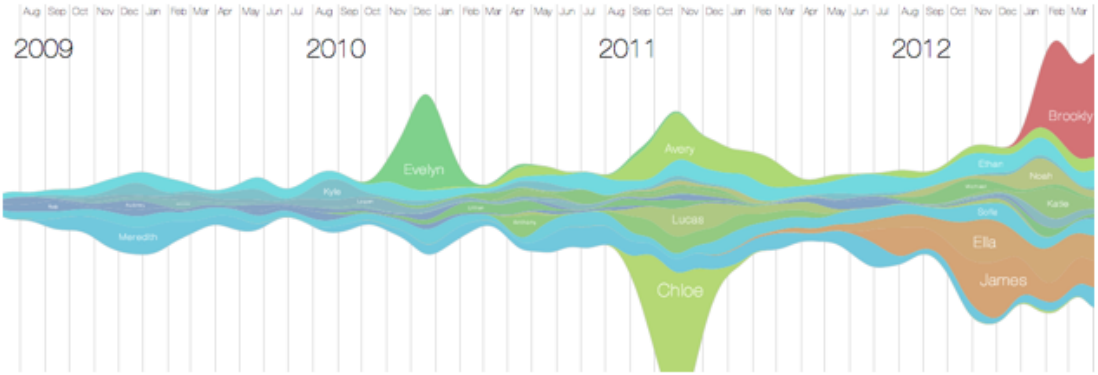
In general, coloring by volatility or burstiness can help highlight breaks in pattern and unique relationships. Coloring by time can result in aesthetically pleasing flows that document the range of relationships over time. And linking saturation to layer-popularity is an easy way to visually accentuate the most important relationships in your stream.

4.3.1 Color by Gender

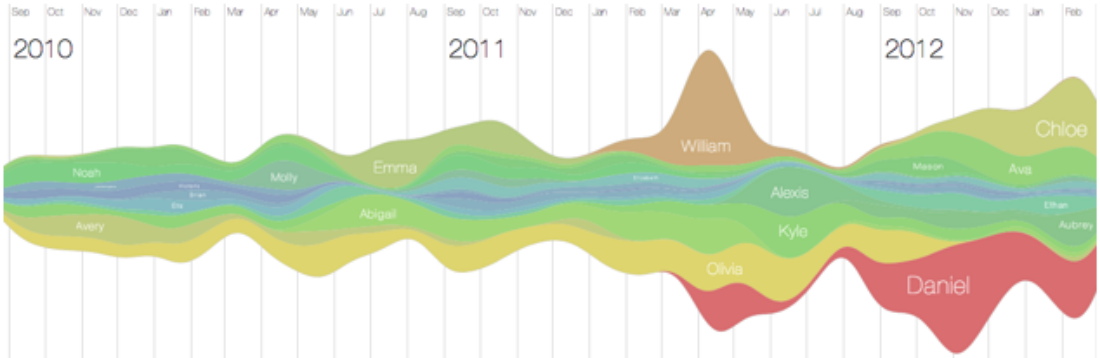
Specifically for use with texting datasets, we incorporate the feature of 'coloring by gender'. To this end, we scraped census data on the top 2,000 boy and girl names worldwide (note: a US data set may have proven more valuable). Each name is associated with a population frequency which allows for a simple Bayesian comparison in the case of gender conflicts: ie, Charlie can be *either* male or female. Enabling this coloring leads to several especially interesting social visualizations, as shown in Figure 5. Lastly, for datasets where color metrics are unavailable or inappropriate, we provide two preloaded 'random' themes in the visualization front-end.



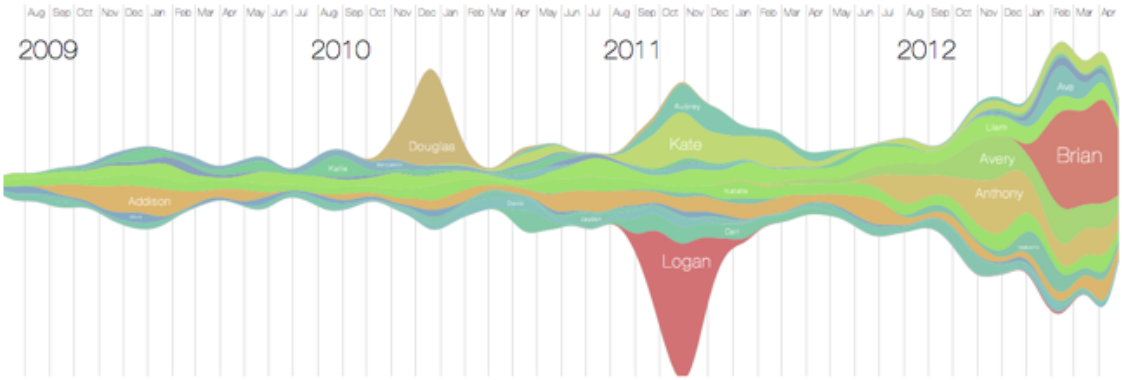
A streamgraph colored by median. Results in extremes on either end, and a relatively indistinguishable center.



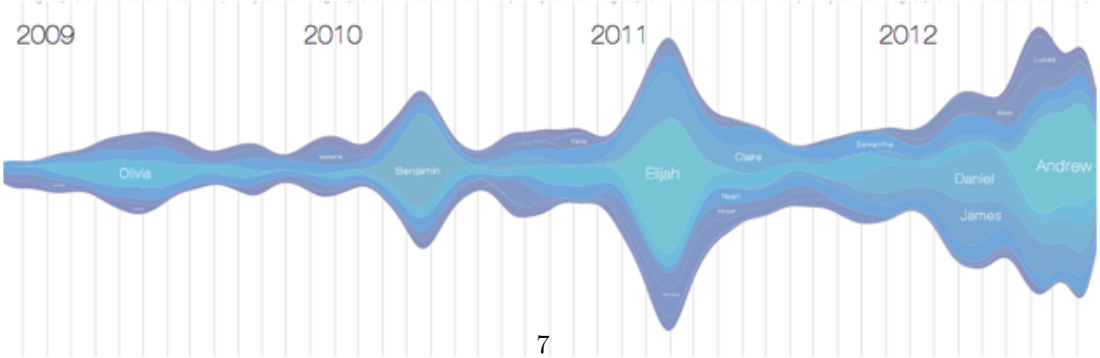
Coloring by 'weighted start date' is more valuable, allows for easy identification of how/what relationships last.



Coloring by volatility produces some of the most aesthitically engaging charts. Here warm hues and high saturation indicate high volatility, while cooler hues and lower saturations are assigned to less volatile layers.



Colored by volatility, saturated by popularity.



Again, colored volatility and saturated by popularity.. This time with the hue range restricted to blue through violet.

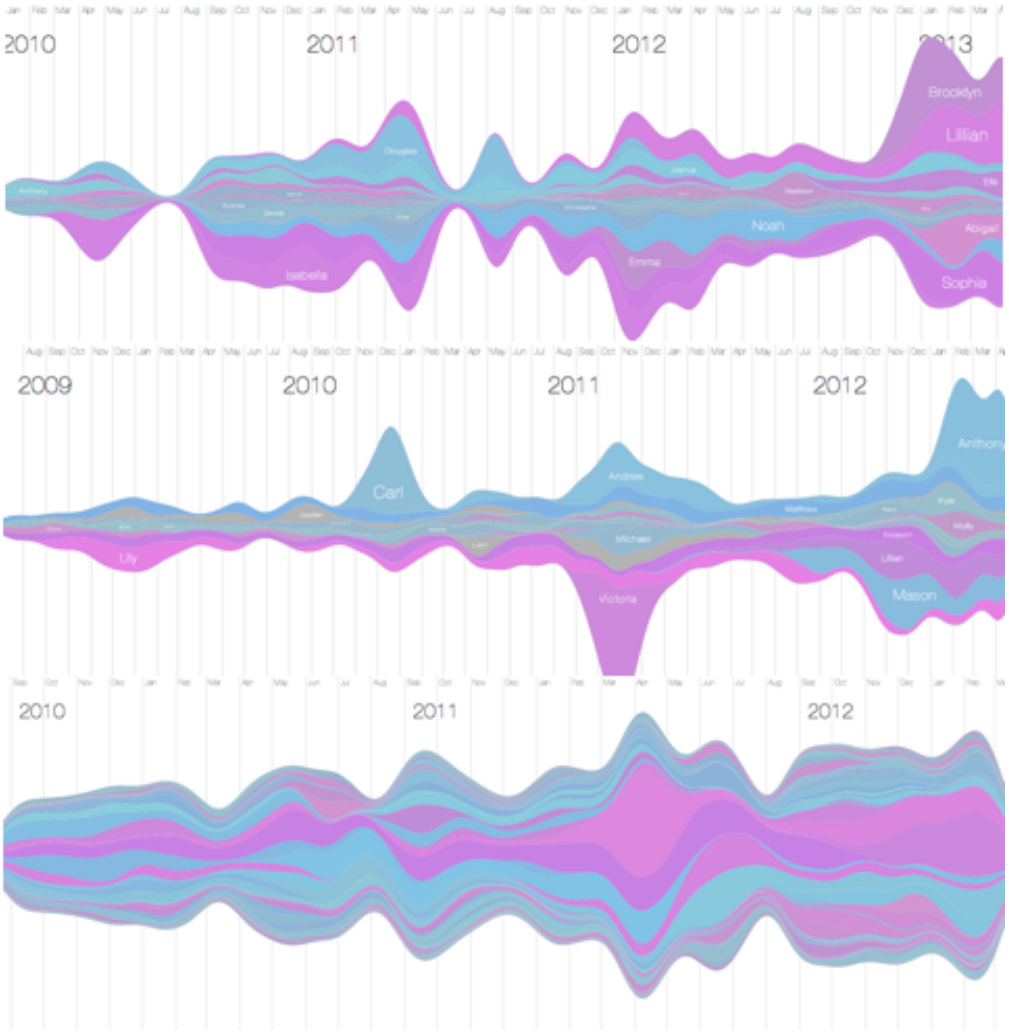


Figure 5: Three seperate Streamgraphs, colored by gender. Males are shades of blue (saturated by popularity). Females are shades of pink, (saturated by popularity).

4.4 Baseline

The key insight behind ThemeRiver and StackedGraph, of course, was their vertically balanced *flowing* design. Although we have smoothed our data and layered it, without a proper calculation for the 'base-line' of the chart, we are left with little more than a blurry histogram. What exactly is a baseline? When constructing our Streamgraph, we can think of each continuous layer as being defined by some function f_i, where i ranges from 1 to n (being the number of layers). To stack the layers, each function f_i is transformed as g_i where:

$$g_i = \sum_{j=0}^{i} g_j$$

g_0, then, represents our baseline, however we choose to define it.

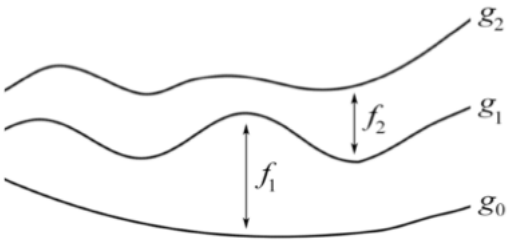


Figure 6: Via Byron and Wattenberg: *a visual description of stacked graph functions.*

ThemeRiver implements the most basic 'balancing' baseline: that of perfect vertical symmetry. To compute this baseline, one needs simply aggregate the n-layer functions at each point i, and divide by 2. This balancing baseline results in a pleasing aggregate stream that is optimized for minimal slope (along the outer edges) at each point in time. Alternatively, the StackedGraph paper puts forth a function for calculating the *weighted wiggle* baseline, which optimally reduces slope variation or 'wiggle' across ALL internal layers with special preference/weighting given to the largest (and presumably most important) layers. While this optimization serves as an excellent baseline in their box office revenue chart, it is perhaps less appropriate for text data. A key tradeoff with the weighted wiggle is a loss in consistency to the aggregate stream's shape and flow. The aggregate stream for text data (representing the sum of one's texting relationships at any given time) is an integral part of our visualization and we do not want to sacrifice its legibility too much for the sake of the individual layers. A good compromise is the *wiggle* baseline, shown in Figure 7. The *wiggle* baseline also optimizes for minimal slope across each layer, but without sacrificing the legibility of the overall structure. What is more, as Byron and Wattenberg demonstrate, it is extremely efficient to calculate:

$$g_0 = \frac{-1}{n+1} \sum_{i=0}^{n} \sum_{j=1}^{i} f_i$$

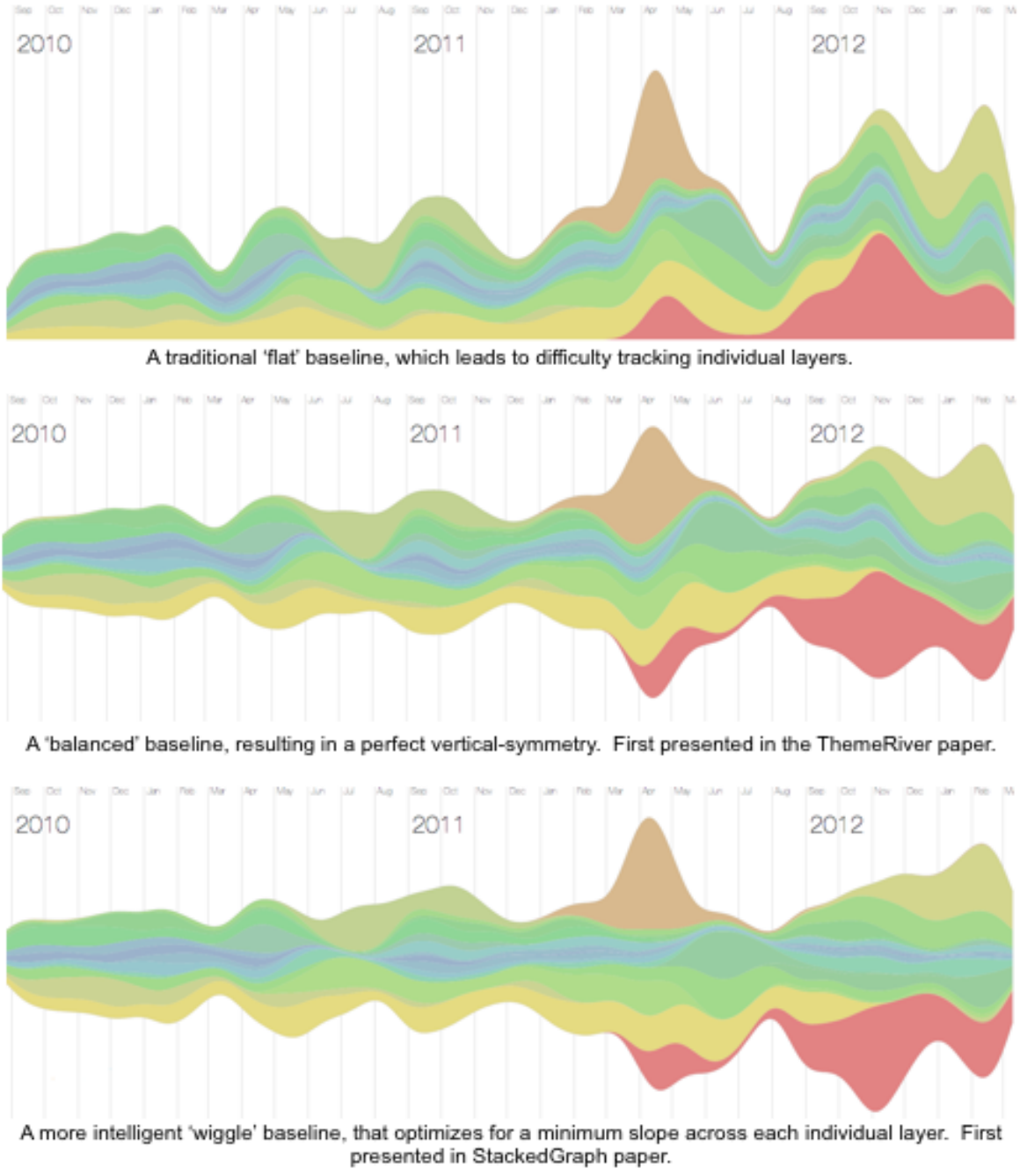
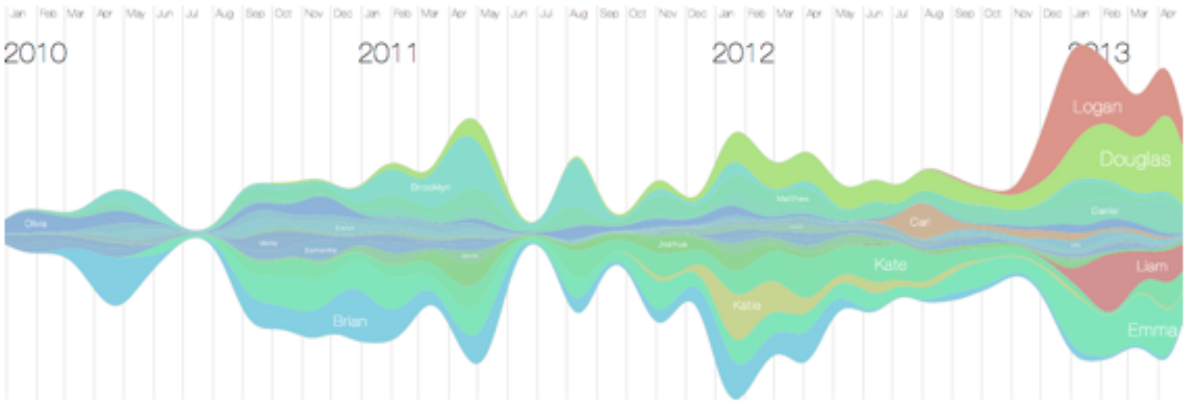


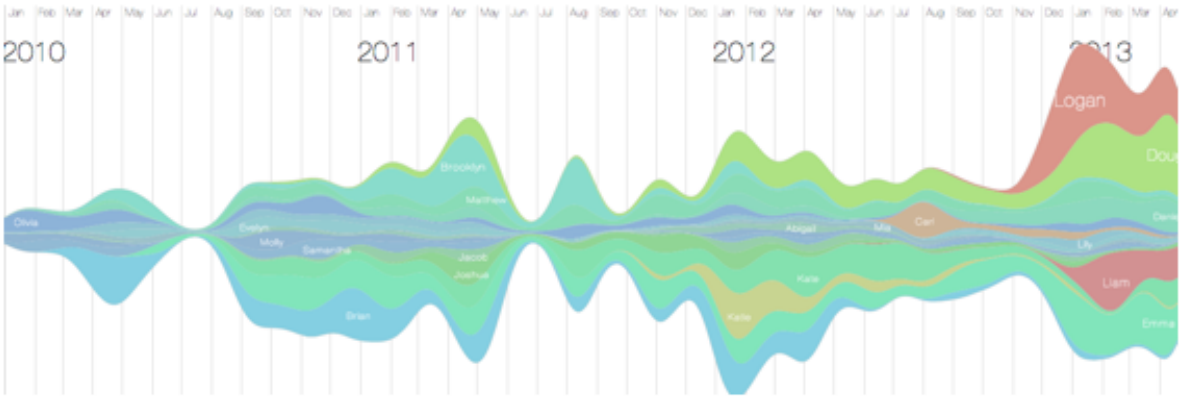
Figure 7: Three seperate calculations for baseline, and the resulting chart formation. The wiggle function, shown last, minimizes the effect that volatile 'bursty' layers along the edges can have on their neighboring layers, while still optimizing for the legibility of the aggregate stream.

4.5 Labeling

One of the more computationally interesting components of the graph generation comes from layer labeling. The challenge here is given n-many layers, label each layer optimally in terms of location and font size. Streamgraph implements two labeling algorithms to choose from to this end. The first is a fast 'greedy' algorithm that scans through each layer, finds the region with the highest height-differential, and then locally expands around that point, estimating an appropriate width and font size. It runs in just over linear time, but the results can be somewhat sporadic. In general, it is appropriate for shallow, smooth graphs, but can break down around volatility and bursts. Our 'brute-force' algorithm is more optimal, and surprisingly fast as well. Here we estimate the aspect ratio for the ultimate text label, and then dynamically scan down the entire layer (worst case m^2 time for each layer, where m is the number of samples) to locate the region that can contain the largest possible bounding box. Due to the flat/collapsing nature of most labels, the 'brute-force' algorithm has acceptable real-time performance once a few basic heuristics are implemented to skip poor regions and memoize constraints along the way. The results of the two labeling techniques are shown below in Figure 8.



Brute-force labelling. Iteratively search each layer for largest bounding box that meets aspect ratio.



Greedy Algorithm. Faster, but less aestheticly successful. Identifies region of maximum height differential and estimates box width + placement at that local.

Figure 8: The two labeling techniques for Streamgraph: greedy and brute-force. Both with acceptable realtime performance.

5 Discussion

5.1 Tech Summary and Final Product

The end result is around 1500 lines of code, split largely between a Ruby module (for the heavy data sourcing, blending, and manipulation) and a Javascript front-end (handling the output logic and labeling algorithms). The final product, which can be scaled to any desired dimensions, is written out as an SVG vector graphic, and an interactive visualization of this graphic is launched on a local web browser for easy exploration and sharing.

5.2 Customization and Out-of-the-Box

Streamgraph was built to be highly generalizable, and to this end we have exposed over 30 of the key visualization variables as parameters, described in detail in the Readme. The parameters allow you to choose everything from the number of contacts to display and specifically who they should be; how much you want to blur, and over what date range; the coloring and ordering of layers; what baseline and labeling algorithms to apply; and even whether or not you want to swap in 'Fake' names as we have done in this paper for privacy reasons.

5.3 Analysis of Streamgraph as a Visualization Tool

As discussed in the Byron + Wattenberg paper, the goal of Streamgraph is not to be the most technically rigorous graphing style, but to be a highly approachable and resonant graphing style. It is true that a non-blended histogram yields more raw information about a given data series than does the Streamgraph, but the beauty of Streamgraph is in its approachability and wide appeal. The focus on the aesthetic yields a graph that can be both emotional, as well as informative. Anecdotally, many friends have asked for blown up prints of their own charts for display.

5.4 Timing

The final Streamgraph generation runs in around 6-8 seconds on a standard Macbook Pro. A third of that time is spent just parsing the message data into memory, and another 1-2 seconds is generally allocated for calculating the various ordering/coloring metrics. We find this to be an acceptable constraint for real-time implementation of the graph, although offline processing is obviously more often appropriate. The SMS mining operation, in contrast, runs over the course of 5-20 seconds depending on the size of the data in question. This is due largely to speed constraints inherent to the SQLite framework, as well as the iOS architecture which requires joins across 4 tables to link contacts to messages. Because we save locally to CSV, this mining operation need only run for the first chart's generation.

5.5 Future work

Looking forward, there are three clear next steps to improve the Streamgraph platform as is. First, we aim to expand the mining scripts to be backwards compatible with older iOS implementations (specifically iOS 5 and iOS 4, which still make up around 9 and 3 percent of the iPhone market respectively). Next, the main executables should be packaged as a end-solution for OSX systems. The appeal of Streamgraph, after all, is its approachable, aesthetic nature. The generation of these graphs should be similarly intuitive and so any final product would need to be GUI / Point and Click. Our third goal is to implement a cloud platform for calculating and sharing these graphs. Even more valuable than a local executable, a cloud implementation could run in real-time, allowing for dynamic adjustments to the final graph via a web interface, as well as final

export or sharing across a number of social platforms and formats. Given the personal nature of texting data, however, serious privacy considerations must drive the design of any online component.

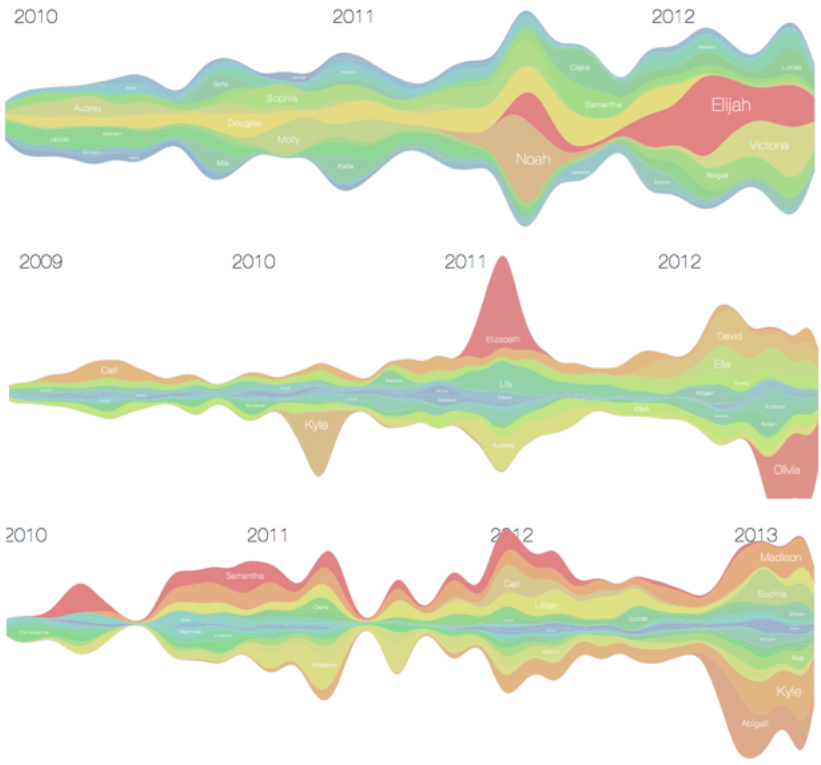


Figure 9: Here are three different SMS data sets, run out-of-the-box through the Streamgraph generator without specific tweaking beyond the default heuristics.

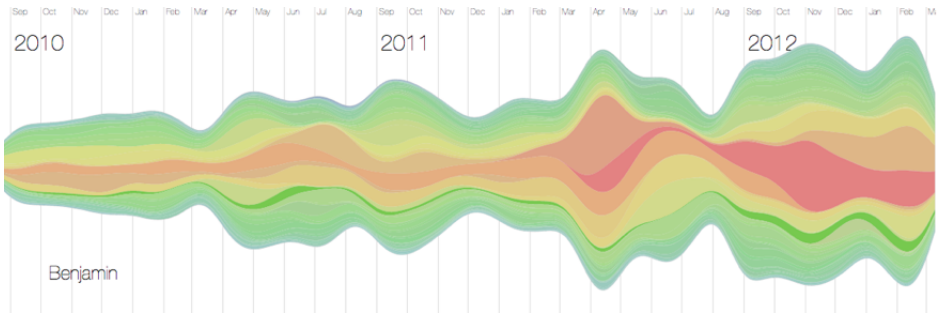


Figure 10: A screenshot of the interactive visualization that launches by default when running the demo package. Hovering over layers highlights them in green, and displays the contact's name in title font.

6 References

1. Byron, Lee; Wattenberg, Martin; (2008) *Stacked Graphs, Geometry and Aesthetics.*

2. Havre, S., Hetzler, B., Nowell, L. (2000) ThemeRiver: Visualizing Theme Changes over Time. *Proceedings of the IEEE Symposium on Information Visualization.*

3. Chitika, *iOS 6 Users Generate 81.3% of All Traffic in North America.* 13 February 2013. http://chitika.com/ios-version-distribution

7 Appendix

1. Codebase attached, seperated into three distinct modules covering *text extraction*, *text mining*, and *Streamgraph generation.*