Graphical Inference with Convolutional Neural Networks

Abstract

Understanding and recognizing trends in scatter plots is a keep step in many statistical analyses, but these trends are not always obviously apparent. Unclear trends can be particularly problematic during exploratory data analysis. In this paper I present a way to use convolutional neural networks to detect trends in scatter plots, taking some inspiration from previous work done in quantifying scatter plots using scatter plot diagnostics or scagnostics.

1 Introduction

Over thirty years ago Paul Tukey introduced the idea of scatter-plot diagnostics or scagnostics. After their introduction Tukeys ideas on scagnostics laid mostly dormant for nearly twenty years until Wilkinson et al. revisited the idea, and introduced a number of new graph-theoretic measures that introduced high speed computing the scagnostics. [8]

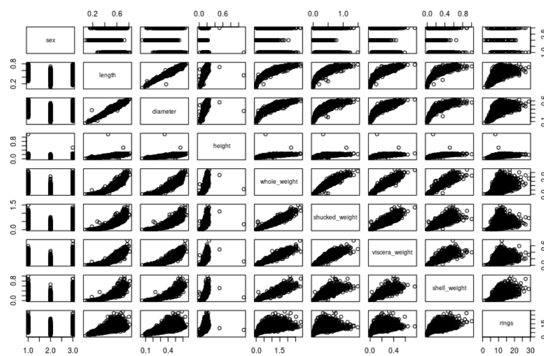

Figure 1: Scatter Plot Matrix of the Abalone Data [7]

Humans are generally quite good at noticing and categorizing trends present in scatter-plots, but hand classification can be a tedious and time intensive process. In scatter plot matrices like the one seen in figure 1 individual trends can become indistinguishable requiring individual inspection of individual plots. Even in a relatively small data set like the one used to produce figure 1 we have over 50 unique plots. While 50 plots can be inspected within a reasonable amount of time it is not uncommon for data sets to have more than 15 features, and a corresponding set of unique scatter plots that numbers more than 100. Wilkinson tried to cut down on the amount of time required to analyze these plots by extracting high level statistics from the plots, and then categorizing them. I similarly reduce the total time required to interpret scatter plots by using a convolutional neural network to classify the plots. These methods can also be used to provide a second opinion on whether a trend is present in a plot when a relationship appears uncertain.

Wilkinson used polygons fit to the point distributions found in scatter plots to develop several measures that can be used to evaluate said scatter plots. These measures are intended to reflect the shape of the fit polygon by considering factors like their area, perimeter, and width.

In this paper I have take the ideas originally presented in the in Wilkinson paper in a slightly different direction by applying the convolutional neural networks to the the problem of scatter plot diagnostics. Rather than attempting to develop hand crafted measures I allow the network to directly optimize a set of weights used to categorize the plots. I do this by using some of the graph-theoretic ideas presented by Wilkinson to reduce help reduce noisy scatter plots to recognizable classes, and then applying techniques from modern image recognition systems to train a con-

volutional neural network to detect those classes.

In section 2 I begin by quickly summarizing some of the measures presented in the Wilkinson paper. I then introduce the theory behind both standard feedforward neural networks, and convolutional neural networks in sections 3 and 4 respectively. Following the introduction of the convolutional neural networks (or CNNs for short) in section 5 I introduce the data used to train the model, and lay out the specific CNN that I have used in section 6 along with the training process behind it. Finally, I will review the performance of the CNN I use in this paper, its limitations, potential improvements, and how it can applied to real data in sections 7 and 8.

2 Graphs and Hulls

The Wilkinson paper focuses on how geometric graphs G* = [f(V), g(E), S] (a graph that maps vertices to points and edges to lines segments connecting pairs of those points) can be used to derive measures of scatter-plots using three different geometric graphs: the Convex Hull, Alpha Hull, and MST (Minimum Spanning Tree). [9]

Convex hulls, and minimum spanning trees can be very effective in some contexts, but in this paper I will focus on the Alpha Hull. The benefit of using an Alpha Hull over graphs described in the Wilkinson paper is that their shape can tightly encompass a mass of points. This allows for the fitting of a shape that is much more representative of a trend than say a convex hull. If fit properly (a valid α value is chosen) an Alpha Hull can also avoid including points that are extreme outliers, which some shapes like minimum spanning trees will always include. [8]

An Alpha Hull is a non-convex shape where an edge exists between any pair of points that can be touched by an open disk $D(\alpha)$ containing no points. Roughly speaking we can think of an Alpha Hull as a shape that could be created by rolling a disk of radius α around a mass of points. This differs from Convex Hulls that must include all points in the distribution, and appear as if a rubber band were fit around the distribution. [8] The exclusion of points that occurs during fitting of an Alpha Hull provides a more granular detailed image than the all inclusive Convex Hulls, and more obviously identifiable trends.

The fit hulls are then used to categorize the plots via a set of measures. A few examples of these measures are the convex measure $c_{convex} = area(A)/area(H)$, and the skinny measure $c_{skinny} = 1 - \sqrt{4\pi area(A)}/perimeter(A)$ (where A is an alpha hull and H is a convex hull). [8] The other measures introduced in the original paper are similarly dependent upon the fitting of polygons to point distributions.

3 Feedforward Neural Networks

To develop a model that can learn from these generated images, and eventually classify new images I turned to neural networks due to their known efficacy in image recognition tasks. A basic neural network consists of three layers: the input layer, the hidden layer, and the output layer.

- The input layer of a basic neural network consists of a number of nodes that simply pass data to the hidden layers without preforming any computation. [3]
- The hidden layer (or layers) are where the bulk of the computation in a neural network takes place. As data passes through the hidden layers computation is executed in order to find weights that are transferred to the following layer. Each of these weights is a weight that exists between two nodes n and m resulting in a weight W_{nm}. [3]
- The output layer maps the final weights produced by the hidden layer to an useful output. These outputs are found using an activation function. [3]

In a standard feed-forward neural network all data move from layer to layer without any backward, or cyclical motion occurring. In this feed forward system all nodes in a given layer are directly connected to all nodes in the following layer. [3]

More specifically, feedforward neural networks approximate a function $f(x) = y$ that maps an input x to a category y. A feedforward network maps an approximate function $f^*(x;\theta)$ to y where θ are a set of parameters learned by the function. [3] This approximate function is called a network,

because each layer in the network is just a function f^n within a larger composition of functions. In an n layer feedforward network for example f^* can be rewritten as $f^*(x) = f^n(f^{n-1}(...f^1(x)))$. [3] These functions are vector to vector functions, but are commonly thought of as layers of vector to scalar nodes acting in parallel. The network learns the parameters that best approximate $f^* \approx f$ by taking a set of training examples paired with their true categories, so the network can see what it must due with a given input at each layer to produce the correct output.

Although, these networks are inherently obtuse they can be loosely analogized to familiar linear models. Simple linear models are incredibly useful due to their efficiency, reliability, and general efficacy, but these models can sometimes ineffective when the relationship between between the input and output is highly non-linear in nature. One common solution to this problem when working with linear models is to apply some sort of transformation to the data that better matches it to a linear model. This is particularly common in the use of support vector machines when a kernel transformation ϕ is employed to avoid the problems presented by non-linearity. In feedforward networks rather than applying some form of transformation to the data we can think of the network as learning the transformation ϕ needed to linearly model the data. [3, 4]

To learn these non-linear relationships we need to choose a valid set of functions to use in the hidden layers of the network. Given that we want to learn potentially very complex non-linear relationships we cannot use linear functions, because if $f(x;\theta_1)$ and $g(x;\theta_2)$ are both linear than any composition of the two $g(f(x;\theta_1);\theta_2) = h$ will be linear as well. [3] A common way to get around this problem is to have each hidden layer be a combination of an affine transformation and an activation function. The hidden layer can be defined as $h = g(W^T x + b)$. The first portion (the affine transformation) is $W^T x + b$, W^T is a set of weights, and b is a bias vector. $g(z)$ is the activation function, and is normally applied to each element of the vector individually such that $h_i = g(W_{:,i} x^T + b_i)$, and $g(z) = max\{0, z\}$. [3]

For classification tasks the activation function of choice is most commonly a softmax function. The softmax function acts much like an ordinary logit function by mapping real valued inputs to outputs on the range (0, 1). In binary cases the simple logit function works well, because we only need to output a single value. In multi-class classification problems we need to output n numbers in the form of a vector that sums to 1 so the output acts as a probability distribution. To do this we turn to the softmax function $softmax(z)_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$ where $z_i = log\ P(y = i|x)$. [3, 4] The softmax thus normalizes the exponentiated outputs of each element by summing over all exponentiations.

4 Convolutional Neural Networks

The task of image classification is a particularly difficult one for standard feed-forward neural networks rendering them fairly effective. So, for image classifications we need to turn to a variation of the classic neural network architecture known as a convolutional neural network.

The standard convolution operation can be written as $s(t) = \int x(a)w(t-a)da$ and is sometimes represented using an asterisk as $s(t) = (x * w)(t)$. In many statistical learning settings we are provided with a distinct set of data points, so we use the discrete version of the convolution $s(t) = (x * w)(t) = \sum x(a)w(t-a)$. [3] Just as in the functions used in the feedforward networks x represents an input, and w a set of weights. In convolutional networks, however, the weights w are sometimes referred to as a kernel. [3] I will refers to these weights as a kernel moving forward into the 2d setting of this problem.

In cases where our input is not just a vector of values (as is the cases when we are working with images), but rather a matrix we move into the 2d setting of the problem, and apply the two dimensional convolution operation: $S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$ where I is an input image, and K is the kernel. [3]

The nodes of convolutional neural networks operate over portions of matrices known as perceptive fields. [3] This is particularly useful when working with images, because images can easily be reduced to matrices of pixels, and by working over regions of matrices CNNs can better reflect the spatial dependencies present in images. Al-

though the images we are working with in this problem are black and white CNN regions can easily be expanded to include channels (by essentially stacking several regions atop one another), so each color of the image has its own channel. [3]

Each layer of a CNN is to applies a kernel to the input matrix to generate a kernel feature. These kernel features are found by iterating over the entire input matrix with an nxn region and adding a single weight to an nxn kernel for each region checked. The goal of this kernel is to reduce the size of the feature space present in the original image, and extract high level features of greater importance from each region.

After using a kernel in the convolutional layer we can further reduce the number of features used (computational power needed) by applying a pooling layer. Like the convolution operation pooling functions over a certain region of the an nxn kernel with each step. But rather applying a somewhat complex function to reduce the region it simply applies either a max, or mean function. Mean pooling reduces each region by taking its mean value, but generally it is more effective to use max pooling, which simply takes the max value of each region. [3] Pooling also offers one other advantage over a an architecture that consists of purely convolutional layers: noise reduction. Pooling layers effectively filter out noise by discarding less important, and potentially noisy weights through the application of the max function.

A combination of a convolutional layer, and a pooling layer compose a single layer in a standard CNN architecture used in image classification. [3]

Another additional layer used in this image classification model is a dropout layer. A dropout layer is used to ignore some nodes in the network. A each phase of training every node is either keep or removed based on a probability p. The point of dropout is to reduce the chance of over-fitting to the training set. By directly ignoring certain nodes at random through the addition of dropout the network is less likely to over-fit, because the dropout layer is effectively reducing codependency between the nodes.

The application of dropout layers is quite comparable to how bagging is used in ensemble methods. For most ensemble methods it is computationally possible to train and evaluate a large number number of relatively simple models (like decision trees) using a form of bagging. [3] With neural networks, however, bagging is not a practical method due to the computational cost of training each individual network. To avoid this we use a dropout layer to train an ensemble on sub-networks found within a neural network. There are of course a number of important distinctions between a dropout layer and an ensemble method. Namely that a dropout layer works on sub-networks of a single parent network, and as a result all sub-networks share a single set of parameters are not independent. The non-independence of these sub-networks lends itself to the reduction of a single cost function that dictates which sub-networks (nodes) to include following the dropout layer. [3]

5 Data Overview

In order to train our CNN quickly and effectively I turned to the 2D point distributions used in the later Scagnostic Distributions to evaluate the consistency and accuracy of the metric laid out in the initial Wilkinson paper. Given that the accuracy of a CNN is largely depended on its training data, and a properly trained CNN requires a fairly large number of examples from each class we wish to identify I implemented a 2D point distribution generator that builds upon the 10 distributions used in the Scagnostic Distributions instead of scraping the web for the 100s of data-sets that would be required to generate the number of scatter plots needed to train the CNN.

5.1 Data Generation

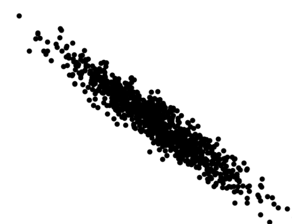

Figure 2: A Point Distribution Generated with the Negative Binomial Process

In order to generate data to be used in the classification system I generated data according to the 14

classes laid out below:

- Uniform: two sets of uniform random points combined together
- Spherical: multivariate normal with a $[[1, 0], [0, 1]]$ covariance matrix
- Binormal: multivariate normal with one of the following covariance matrices
  - $[[2, 2], [2, 3]]$
  - $[[5, 5], [5, 6]]$
  - $[[8, 8], [8, 9]]$
- Funnel: multivariate log normal with $\rho = 0.6$
- Negative Exponential: exponential decay with added noise
- Quadratic: $-x^2$ with added noise
- Clustered: a number of clusters generated using the *scikit-learn* `make_blobs` function [6]
- Doughnut: two circles of varying diameters created using the *scikit-learn* `make_circles` function [6]
- Stripe: uniform random points multiplied by random normal integers
- Sparse: random integers multiplied by random integers
- Exponential: exponential growth
- Logarithmic: logarithmic growth
- Negative binormal: equivalent process to binormal, but with negative ρ values
- Negative funnel: multivariate log normal with $\rho = 0.6$ and negative x values

All of the images generated by these processes are injected with a randomly chosen amount of added noise, and number of points. Noise is added directly to generation functions to ensure that generated points do not perfectly match a known function. This additional noise allows us to get a more representative training set that better represents both the messiness of real world data, and the murkiness that sometimes occurs between classes. The primary way noise is added to each and every one of the generations is by choosing the number of points that are going to be drawn from a given process. At the beginning of each generation the number of points drawn is chosen at random from the set $\{50, 150, 500\}$. Each of these different points levels are intended to represent a different level of scatter plot sparsity ranging from quite sparse to quite dense. For the uniform and spherical distributions the only noise needed comes from the number of points, but all other distributions include at least one other form of added noise.

These other forms of noise are then added to the generation process based on which process is being used. In the case of the binormal distributions that noise simply comes from the set of ρ values chosen. For the generation processes that follow a function of some sort like the exponential or quadratic distributions noise is directly injected into the y values of each pair of points by choosing a random number from a randomly chosen range of values. For the generations that involve integer values like the stripe process the range, and number of distinct integer values is chosen at random. The clustered generation process follows a similar pattern by choosing the number of clusters from a set of integers at random.

5.2 Image Generation

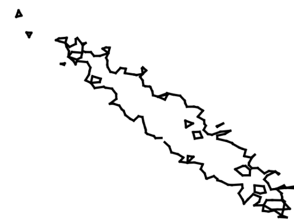

Figure 3: An Alpha Hull Fit to the Point Distribution in Figure 2

generating each data set using the methods outlined above the distributions are converted to classifiable 20 dpi images by fitting an Alpha Hull, and outputting the outer edges of the fit alpha hull as an image. The final image output into either a training, or validation set is essentially a black and white outline of the point distribution fit according to the alpha hull specifications.

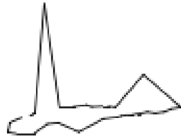

Figure 4: An Alpha Hull With a Poor Fit Due to an Improper Choice of α

Each hull is fit using a randomly chosen alpha value. These values are chosen randomly to both add extra noise to the image set, and simulate the inconsistencies in hull fitting that may occur during either a batch based hull fitting process, or an untrained individual. The random choice of alpha values also results in a few incredibly unhelpful outlines due to a rare inconsistencies in the way alpha hulls are fit. In addition to these (arguably overly noisy) outcomes, the random choice of alpha also provides different levels of sharpness to each image. This allows two images generated using equivalent processes will not necessarily look exactly the same adding an extra layer of protection against potentially very similar images.

The one exception to this process has to do with the generation of images related to sparse point distributions. Sparse distributions are effectively straight vertical lines separated by white space. Due to this highly uneven distribution of points it is impossible to fit a representative alpha hull, because any hull fit would essentially be a rectangle. To avoid this sparse images are output directly as a their point distributions.

6 Model Outline

The model I use to in this paper consists of the following blocks:

- A convolutional input layer
- 4 convolutional layers combined with pooling layers
- 3 dropout layers interspersed between the main convolutional and pooling layers
- A softmax activation function

Each image is input into an initial convolutional layer to begin. The output of that first convolutional layer is then sent into a series of four combined covolutional/pooling layers. After the first, second, and fourth of those layers there is a dropout layer. After the final dropout layer the output is flattened into a dense layer, and feed through a softmax layer that produces final class labels.

The model is compiled to use categorical cross entropy as its loss function, and the commonly used *adam* optimizer as its optimizer. The model was implemented using the *Keras* library in python. [1]

7 Model Evaluation

Validation Set Scores			
Class	Precision	Recall	F1-Score
Uniform	0.97	0.99	0.98
Spherical	0.70	0.61	0.65
Binormal	0.85	0.81	0.83
Funnel	0.96	0.90	0.93
Neg-Exponential	0.97	0.99	0.98
Quadratic	0.96	0.97	0.96
Clustered	0.73	0.83	0.78
Doughnut	0.73	0.83	0.77
Stripe	0.89	0.98	0.93
Sparse	1.00	1.00	1.00
Exponential	0.97	0.97	0.97
Logarithmic	0.99	0.92	0.95
Neg-Binormal	0.96	0.87	0.91
Neg-Funnel	0.94	0.93	0.93
Average	0.90	0.90	0.90

Table 1: Validation Set Scores by Class

I decided upon the final architecture for the model based on its performance on a validation set that consisted of 150 images per class for a total size of 2100 images. In Table 2 we can see the performance of the final model. For each class precision, recall, and F1-Score were calculated using the *scikit-learn* `metrics.confusion_matrix` function. Precision, recall, and F1-Score are calculated in the following manner:

- $Precision = \frac{true\ positives}{true\ positives + false\ positives}$ [5]
- $Recall = \frac{true\ positives}{true\ positives + false\ negatives}$ [5]
- $F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R}$ where $\beta = 1$ [5]

Nearly all classes scored over 0.90 in terms of F1-Score showing us that the CNN is effectively learning to predict the classes of images produced

by the process laid out in section 5. Despite this high average F1-Score, however, there are a number of classes on which the model performs well above, or below the mark. Spherical, doughnut, and clustered distributions seem to be particularly difficult for the model to interpret. This is likely caused by the high potential for similar hulls to be fit to these distributions due to their shared production of circular shapes. The sparse class lies on the opposite side of things being the most unique of the classes, and the only class for which an Alpha Hull is not fit. The lack of a hull in images from the sparse class is likely what leads to its perfect F1-Score of 1.00.

Evaluation of Model Output for Abalone Data	
Accuracy	0.72

Table 2: Model Accuracy on the Abalone Data [1]

After training the CNN I turned to the Abalone data-set shown in the original Wilkinson paper to test the how effectively the CNN would predict the class of each of the scatter plots yielded by the Abalone data set. [9, 2, 7] The CNN was able to correctly identify the majority of the plots without issue, and only incorrectly identified a small minority of the plots. It should be noted that among the incorrectly identified plots there was always a clear error in the way the alpha hull had been fit to the scatter plot, such that there was too much noise present in the image to identify it correctly based on the alpha hull alone.

Figure 5: An Ambiguous Alpha Hull from the Abalone Data Set

There are, of course, no predetermined class labels in for the scatter plots taken from the Abalone data set, so I cannot report accuracy scores directly, but I can report a rough accuracy score based on my own hand created labels. According to my own hand labeling of the plots the model had an accuracy score of 0.72 on the Abalone data. As was the case in the in class by class breakdown of the validation set accuracy the misclassification of trends in the Abalone scatter plots was not uniform. There were, for example, no plots that were incorrectly classified as being random noise (uniform or spherical). If there was some sort of trend in the data the model always detected that a trend existed even if it ultimately applied the wrong class label. Aside from the $< 10\%$ of plots that the model identified in a flatly incorrect manner the primary source of inaccuracy came from plots that were somewhat ambiguous in nature. As an example, *Stripe* and *exponential* plots can look very similar. Due to this grayness, however, I would not say that the model is wrong outright. Many of these plots (like the one in Figure 5) are not obviously members of one class, but rather on the boarder of two classes, so I would evaluate the model as being correct so long as it outputs one of two classes they boarder.

[1]This accuracy value is based on my own hand labeling of the Abalone Data

8 Future Improvements

As is always the case there are a number of possible ways to improve this model.

The first major source of potential improvement comes from the classes I chose to use. During the early stages of the model building process I tried using a number of different sets of classes, and corresponding generative methods. The number of classes used produced some of the most significant changes in the model's output. Having too few classes leads to high class overlap, while too many classes can lead to overly niche classes that only crowd things out. I chose the set of classes that I found most useful and produced the best results, but it is likely that the class list can be improved. There may also be important trends that I have failed to include in my class list, and should moving forward.

A major source of error in the model came from difficult to classify images due to poorly fit alpha hulls. Although, the alpha hulls are fit well the vast majority of the time the model (or more specifically the final class labels it produces) would be improved by better hulls.

The model is another area of probable potential improvement. Although, I do not expect to see major improvements over the model I use in this paper in terms of loss, or accuracy I believe there

is room for optimization. I think it is conceivable that a slightly stripped back model would yield gains in efficiency without major losses in accuracy. In particular, it is possible that the hyperparameters could be tuned further.

Image generation could also be changed. There is room for experimentation with different DPIs, edge widths, whether the shapes are filled, and a number of other factors. Any of these changes in how the images are developed could very well disentangle some of the ambiguity currently present in the images.

Finally, it may be helpful to return to some of the other graphs mentioned in the Wilkinson paper. Here I focused on Alpha Hulls, but I may be that MSTs or Convex hulls are also effective individually or in combination with the other hulls.

9 Conclusion

Overall, convolutional neural networks show great promise in the identification of trends present in scatter plots. Given their consistent results across real data, and clear performance on the validation set it seems as if this method has potential to succeed in a number of different real world tasks. There is room for improvement in the model, but as a prototype the model has exceeded my expectations. Going forward if this model were to built into a more easy to use package it could speed up processes ranging from general trend identification to lineup tasks.

Overall, I am confident that with slightly better tuning the convolutional neural network used in this paper can wide successfully applied to a wide variety of problems. I also believe that by slightly changing the way in which the CNN is used that set of possible applications can be further expanded. By simply reducing the classes used to noise/not noise the same method could be used to detect trends in a more general sense instead to trying to return a specfic trend label. The CNN could also be re-appropriated to other similar graphs (like time series) to attempt to label trends within a longer process.

This capacity is (at least personally) a source of optimism for an unknown number of new methods that may soon be applied to visual inference problems.

10 Acknowledgements

I would like to thank Professor Adam Loy for advising me throughout the course of this project, and keeping things on track. Additional thanks go to Jack Hessel for helping me through a couple of tricky implementation errors that popped up along the way. I also want to thank the folks who maintain the UCI Machine Learning repository for helping to fuel this project. Finally I just want to thank the authors of *Keras*, *scikit-learn*, and all other packages I used during this project for making your tools publicly available.

References

[1] François Chollet et al. Keras. https://keras.io, 2015.

[2] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[4] Hastie, Trevor, Tibshirani, Robert, Friedman, J. H. *The Elements of Statistical Learning : Data Mining, Inference, and Prediction*. Springer, New York, 2009.

[5] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition (Prentice Hall Series in Artificial Intelligence)*. Prentice Hall, 1 edition, 2009.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[7] Simon R Talbot Andrew J Cawthorn Warwick J Nash, Tracy L Sellers and Wes B Ford. The population biology of abalone in tasmania. i. blacklip abalone from the north coast and islands of bass strait. *Sea Fisheries Division, Technical Report*, 48, 1994.

[8] Leland Wilkinson, A Anand, and Robert Grossman. Graph-theoretic scagnostics. volume 5, pages 157–164, 11 2005.

[9] Leland Wilkinson and Graham Wills. Scagnostics distributions. *Journal of Computational and Graphical Statistics*, 17(2):473–491, 2008.