Graduation Project
Estats: A Non-Parametric Python Statistical Library

Omar Ghanem 20221446808
Faidullah Moftah 20221445636
Eiad Samih 20221372889
Mohamed Samir Elsayed 20221447318
Supervised by Dr. Amr Amin

Contents

Chapter 1

Introduction

In the fields of data science and statistical computing, there exists two main programming languages: Python and R. (One might add Julia to the mix, but as the language is still in its childhood phase of life, we may safely ignore it as superfluous to our analysis.)

The statistical packages ecosystem of these two languages differ in several respects:

1. The package's maintainers.
2. The package's users.
3. The package's value.

It will prove useful to discuss these three point first, as justification for our creation of the *estats library*, before we discuss the scientific value of *estats*.

1.1 Maintainers

In Python, the package maintainers are mostly software engineers, working programmers and computer scientists. From these kinds of people we can expect robust, relatively bug-free code; what we cannot usually expect are statistical packages with the interpretability and elasticity of use that the working statistician need. We can expect powerful *algorithms* but not subtle *inference*.

An example may make things clearer.

Everybody knows of *sklearn*. Think of any machine learning algorithm you like and, unless it's imaginary and you have dreamed it up in your sleep, you will most likely find it implemented in *sklearn*. Nonetheless, just a few versions ago, its implementation of linear regression used regularization by default, with no boolean parameter to prevent it.

From this, one can glean a few facts about the objective function which most statistical python packages try to maximize.

In contrast, R packages are usually maintained primarily by statisticians who have a deep understanding of the underlying mathematics; some of the implemented algorithms may be obscure (to some) but they will also be amply documented, and designed in such a way so as to give the user maximum freedom[1].

1.2 Users

From the last section we may easily guess what kind of users are attracted to what kind of ecosystem. The ease-of-use of most python packages[2] has attracted those who lack the necessary background to conduct complex, robust statistical analysis, but for whom the allure of the popular baggage-free packages proves irresistible. More statistically literate users in turn are more attracted towards the R ecosystem.

Such a process is by no means the only one way. For every force there exists a corresponding force, equal in magnitude but not necessarily opposite in direction[3].

As some users become entangled in a certain ecosystem, that ecosystem in turn tries to become a more perfect habitat for those it hosts; to bend to their whim, let's say. It's therefore no surprise that most packages in the python ecosystem consists only of the popular algorithms and what has been deemed by the hive-mind as *useful*; to find an obscure or newly created statistical technique one has to turn to R.

[1] A not insignificant fact which has led to this divide is that, originally, Python was designed as a general-purpose programming language while R was designed primarily for statistical computing.

[2] As writing "statistical packages" has proven slightly cumbersome, we will now use the shorter "packages" while assuming its meaning to be the formerly quoted.

[3] Newton's third law, slightly altered.

1.3 Value

For every commodity there must be a demand, else the commodity will be no commodity at all, as it will therefore have no use-value. For a package, which is but a commodity in different form, its use-value will determine how many will install and use it.

But is that the only available metric?

One might say that the inherent value of a thing, as determined by its spirit, its theoretical elegance, and the labor which was the cause of its creation, is also a viable alternative. We will call that abstract value.

Under such classification then, we would classify Python packages as high use-value low abstract-value, while R packages are low use-value high abstract-value.

We hope to create a package in the vein of the R ecosystem.

1.4 Estats

The package we are developing then, *estats*, is a python library that, unlike most others, has a high abstract value. We do not expect its users to be those newly introduced to the field of statistical analysis; rather its users should be experienced statisticians who desire a change from the standard, widely available methods in the python ecosystem. And the usual course of statistical analysis, as everybody knows, goes a bit like this:

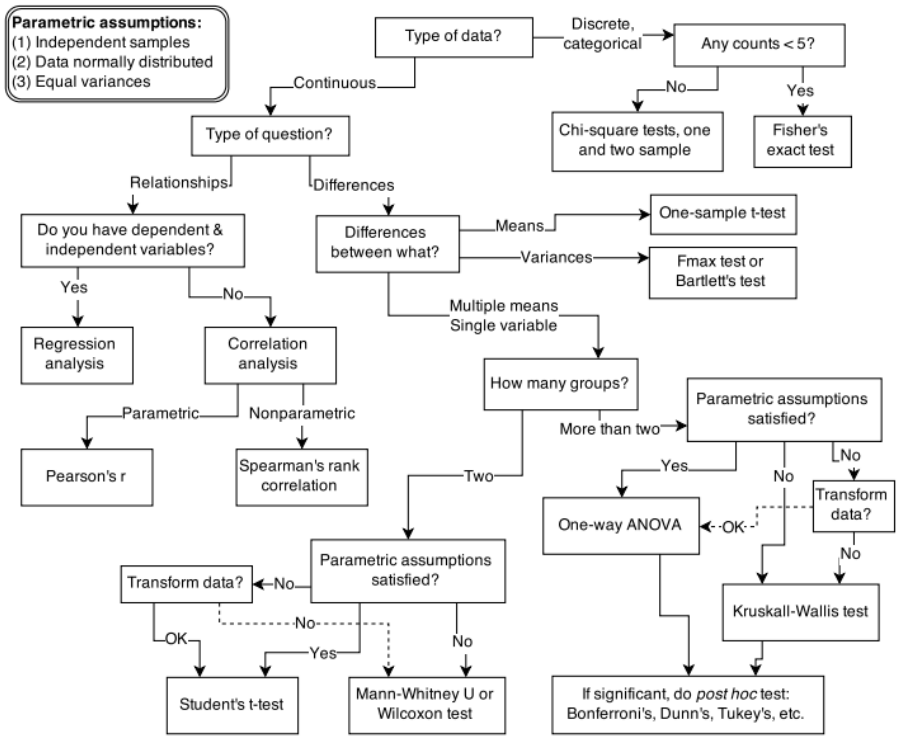

Figure 1.1: The standard path of statistical analysis

A long, contemplating look and it's obvious how limited most of these tools are (all of them are implemented in *statsmodels*[4]). Consider a few:

First, we have Pearson's correlation coefficient. A useful device, certainly, for when the number of variables is too large to be amenable to bivariate plotting (if we have n variables then the number of plots would be $\frac{n(n+1)}{2}$; substitute a realistic number for n, say 20, and we have 210 plots). Pearson's correlation coefficient is a way of automating this search for bivariate dependence by generating a statistic r, such that $0 \leq r \leq 1$, with 1 implying that the variables have a dependence of the form $y = mx + c$, while 0 implies that there is no

[4]The preeminent python statistics library.

linear relationship. The defects of this are obvious; the variables may possess a *non-linear* dependence, which will therefore not be discovered by Pearson's correlation coefficient.

Second, we have the Student's t-test. This test of means requires two thing: First, that

$$\bar{x} \sim \mathcal{N}(\mu, \alpha/n)$$

and, second, that

$$\frac{(n-1)S^2}{\alpha^2} \sim \mathcal{X}_{n-1}$$

where S^2 is the sample variance. People like to speak airily of the first condition. The central limit theorem will take care of all that, they say. As long as $n \geq 30$ everything is fine and well, they will cheerfully exclaim. One cocks an eye at them, and thinks: How blessed are the delusional! They forget that the central limit theorem says nothing concerning the time of convergence; all that it say is, that as $n \to \infty$, the distribution of the sample mean will approach the normal distribution. How long that will take depends, naturally, on the underlying distribution of the data. A lot of the time the convergence happens with small n, that's true, but that's by no means a *guarantee*.

Example 1.1. As a simple example, consider a bernoulli distribution with $p = 0.001$, and let's simulate many datasets (say 20000) of size $n = 1000$. 1000 is certainly greater than 30. We will then calculate the mean of each dataset. if the folktale that $n \geq 30$ is all that's required is true, then we would expect the distribution of the sample means to be normal. But consider the actual distribution:

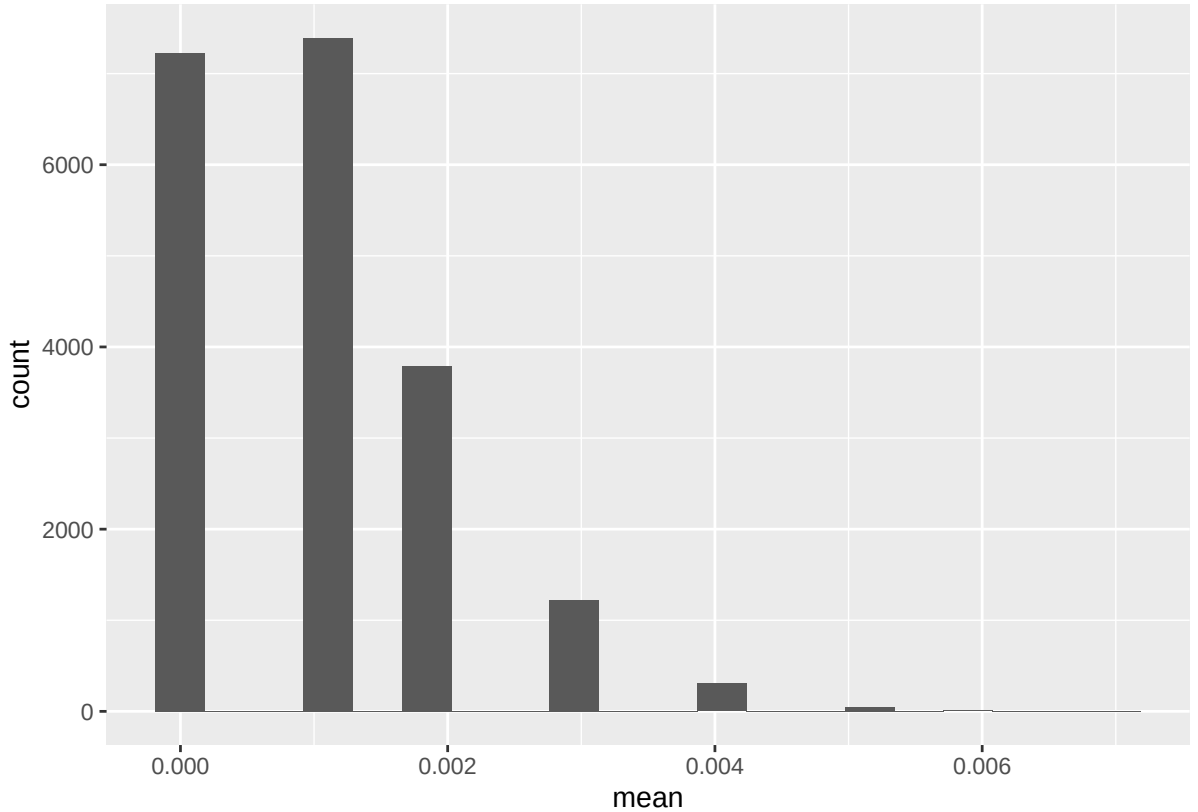

Figure 1.2: Distribution of sample means

That certainly isn't normal. So much then, for needing only that $n \geq 30$.

With regards to the second condition concerning the sample variance, matters are far simpler, it's satisfied only if the distribution of the data is explicitly normal.

We can say more, naturally, far more, concerning these little foibles and delusion that many applied statisticians fall into (the ANOVA, for example, assumes plainly that the data is normally distributed), but what would be the point? The general theme is by now quite clear: *Parametric methods possess many defects which make their use in real-world data analysis dubious.*

But because such methods are the most popular and well-known, they are the ones usually

implemented in statistical packages, and thus the ones most people use; we hope for *estats* to be different. In it we will implement many methods useful in statistical analysis: two-sample test, analysis of variance, clustering, independence testing and so on and so forth, but with the added caveat that we require our implementations to be *non-parametric* and *distribution-free*[5].

To implement these methods non-parametically, we use *energy statistics*, from which our package get its name, a field of statistics introduced by a mathematician named Gábor J. Székely. In the following sections we will explain in detail how *energy statistics*[6] work and what methods we have implemented and plan to implement in *estats*.

[5]But what do we mean by non-parametric? Only that we make *minimal* assumptions concerning the data-generating distribution. In most of the methods we implemented in *estats*, we assume only that one of the three first moments exists, which is certainly a very minimal assumption; or at least better than assuming the data came from the normal distribution.

[6]And that name, by the way, has nothing to do with oil, crude oil, natural gas, electricity, or renewable energy sources.

Chapter 2

Energy Statistics

2.1 Preliminaries

Energy statistics are based upon reducing all the information the data contains to just information about the distances between observations. More plainly, let's assume we have two data matrices $X \subset R^{n \times k}$, and $Y \subset R^{m \times l}$, so that the first data matrix contains n observations with k features while the second contains m observations with l features. We then forget about the points themselves and consider only their distances, or rather their distance matrices, the first one being D_x such that

$$D_x^{ij} = ||X_i - X_j||$$

while the second matrix D_y is

$$D_y^{ij} = ||Y_i - Y_j||$$

If $k = l$, we can form a third distance matrix D_{xy} with

$$D_{xy}^{ij} = ||X_i - Y_j||$$

Naturally, all these matrices will be symmetric. What we have done here is that we have *transformed* our data. Instead of dealing with n points $\in R^k$ and m points $\in R^l$, we are now dealing with, if $k = l$, a set of $\frac{n}{2} + \frac{m}{2} + \frac{nm}{2}$ data points $\subset R^1$. We then define our statistics as functions of these distances.

Whether this is a useful thing to do is at this stage not quite clear. But later on we will describes statistics defined on these distances such that a statistic equals 0 if and only if an underlying null hypothesis is true. Statistics can be defines for many null hypothesizes, such as:

1. $F_x = F_y$
2. X and Y are independent
3. $X \sim \mathcal{N}(0, 1)$
4. Diagonal symmetry, i.e, $X \sim -X$
5. Given multiple random variable $X_1, X_2, X_3, ..., X_1 \sim X_2 \sim X_3 \sim ...$

And many other hypothesizes. Gábor J. Székely, the aforementioned father of energy statistics, likes to compare energy statistics with distance functions in physics that measure the potential energy. One of the most famous of these functions is, of course, the famous function measuring the gravitational potential energy between two bodies, which is derived from the law of universal gravitation usually attributed to Newton[2]:

$$P_e = -G\frac{M_1 M_2}{r}$$

where G is the gravitational constant, and r is the distance between the two objects. This function is 0 if and only if the two masses are infinitely far apart, a theme which will be part and parcel of the methods we shall discuss later on, beginning with the energy distance in the next section.

[1]If instead $k \neq l$, then we would have $\frac{n}{2} + \frac{m}{2}$ points

[2]But wrongly. Robert Hooke was the discoverer of the law, and Netwon can claim only that he put it on firmer grounds, and used it to prove Kepler's third law. Many things are attributed to Newton which are rightfully the achievements of others.

2.2 Energy Distance

The potential gravitational energy equals 0 when the masses are infinitely far apart, but suppose we wanted the opposite[3], i.e, a measure that is 0 when two random variable are infinitely *close* together, how would we go around defining such a measure?

Assuming that we have two random variables X and Y, and that we know their cumulative distribution functions F and G, then a commonsense way to measure the differences between the two would be

$$\int_{-\infty}^{\infty} (F(x) - G(x))^2 dx$$

assuming, of course, that the random variables are one-dimensional; if instead X and Y are random vectors of size n then the measure would be

$$\int_{R^n} ||F(\mathbf{x}) - G(\mathbf{x})||^2$$

Such a measure is obviously 0 if and only if $F = G$, which is exactly what we want. But from the above measure, we get no sense what a *statistic* should look like. Assuming we have random samples in the wild, how are we supposed to get an estimate for this integral? One hopes that the integral can turn out to be extremely simple and elegant; perhaps even– and now we are suffering from the third of the seven deadly sins– being a function of expectation. Surprisingly enough, that turns out to be the case.

Proposition 2.1. *If f and g are the pdfs of the n-dimensional random variables X and Y, and $\bar{f}$ and $\bar{g}$ are the fourier transforms of f and g, then*

$$\int_{R^n} ||F(\boldsymbol{x}) - G(\boldsymbol{x})||^2 = \frac{\Gamma(\frac{n+1}{2})}{2\pi^{(n+1)/2}} \int_{R^n} \frac{\bar{f}(t) - \bar{g}(t)}{||t||}$$

where Γ is the gamma function. It can also be proved that

[3]Sometimes, to get what one wants, one has to develop an intimate relationship with what one *does not* want

$$\frac{\Gamma(\frac{n+1}{2})}{2\pi^{(n+1)/2}}\int_{R^n}\frac{\bar{f}(t)-\bar{g}(t)}{||t||} = E(||X-Y||) - \frac{1}{2}E(||X-X^{\backslash}||) - \frac{1}{2}E(||Y-Y^{\backslash}||)$$

where $X^{\backslash}$ is an independent identically-distributed copy of X, and the same relation holds between $Y^{\backslash}$ and Y. From the previous two equalities we have

$$\int_{R^n}||F(\boldsymbol{x})-G(\boldsymbol{x})||^2 = E(||X-Y||) - \frac{1}{2}E(||X-X^{\backslash}||) - \frac{1}{2}E(||Y-Y^{\backslash}||)$$

We can now define the energy distance very simply as follows:

Definition 2.1. The energy distance function between two random variables X and Y $\mathcal{E}(X,Y)$ is defined as

$$\mathcal{E}(X,Y) = 2\int_{R^n}||F(\mathbf{x})-G(\mathbf{x})||^2 = 2E(||X-Y||) - E(||X-X^{\backslash}||) - E(||Y-Y^{\backslash}||)$$

And we multiplied by 2 for those people who are a little averse to fractions. The terrible integral, then, has transformed into a simple sum of expectations. Assuming then that we have two data matrices $X \subset R^{n\times k}$, and $Y \subset R^{m\times k}$ we can easily estimate $\mathcal{E}(X,Y)$.

Definition 2.2. The estimator for $\mathcal{E}(X,Y)$ is

$$\overline{\mathcal{E}}(X,Y) = \frac{2}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m}||X_i-Y_j|| - \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}||X_i-X_j|| - \frac{1}{m^2}\sum_{i=1}^{m}\sum_{j=1}^{m}||Y_i-Y_j||$$

A cursory look and one notices that this estimator equals 0 if and only if X and Y are permutations of each other, i.e, that each point in X exists in Y but with a different index and vice-versa.

There exists many applications for the energy distance. A few examples:

1. $dCor(X,Y)$, a measure of independence that equals 0 if and only if X and Y are independent(statistically far apart). This is the statistical equivalent of the gravitational potential energy P_e.

2. Consistent multi-sample tests of equality of distributions
3. Characterization and test for multivariate independence
4. Change point analysis
5. Distance components (DISCO), a nonparametric extension of analysis of variance for structured data

We have implemented several of these methods in *estats*– and plan to add several more– and shall discuss them in the next section.

Chapter 3

Estats

3.1 Goodness-of-Fit-Tests

The classical application. Given data points in R^k, x_1, x_2, x_3, ..., x_n generated from a certain distribution F, we would like to test whether F equals another hypothesized distribution F_0. Naturally, we do not actually know what F is, but we *do* know what the empirical cdf, let's call it F_n, is. Is knowing that
in anyway helpful? W don't mean generally, because, as everybody knows, the Kolmogrov Smirnov test depends on comparing the empirical cdf with the theoretical cdf of the hypothesized distribution; rather I mean in the paradigm of energy statistics. If we recall the definition of energy distance, which is,

$$\mathcal{E}(X,Y) = 2\int_{R^n} ||F(\mathbf{x}) - G(\mathbf{x})||^2 = 2E(||X-Y||) - E(||X-X^{\backslash}||) - E(||Y-Y^{\backslash}||)$$

we might start getting the correct idea. Because after all the empirical cumulative distribution function is nonetheless *a* cumulative distribution function. So we might as well try plugging it into the formula to see what happens. Let X be a random variable distributed as

$$X \sim F_n$$

and let Y be a random variable distributed as

$$Y \sim F_0$$

then $\mathcal{E}(X, Y)$ will be

$$\mathcal{E}(X,Y) = \frac{2}{n}\sum_{i=1}^{n} E(||x_i - Y||) - E(||Y - Y^{\backslash}||) - \sum_{i,j=1}^{n} ||x_i - x_j||$$

Gábor J. Székely proved that under the null hypothesis $F = F_0$, $\mathcal{E}$ multiplied by n, i.e, $n\mathcal{E}$, has a non-degenerate asymptotic distribution as $n \to \infty$. From this we can apply the test without actually *knowing* what that asymptotic distribution is. We can simulate many values from F_0 and construct an empirical distribution for $n\mathcal{E}$, which we can thus compare with the statistic calculated from the data $n\overline{\mathcal{E}}$, getting the dreaded and much sought after p-value.

To test equality for a certain distribution then, we have to calculate $E(|x_i - Y|)$ and $E(||Y - Y^{\backslash}||)$, which will differ depending on which distribution F_0 we want to test. As an example of these kinds of test, we will discuss the problem of testing for the uniform and normal distribution.

Example 3.1. Given data points in R, x_1, x_2, x_3, ..., x_n, we would like to test whether they come from a uniform distribution with parameters b and a and whose pdf f will therefore be $f = \frac{1}{b-a}$. Let Y be a random variable with pdf f, then we can compute $E(x_i - Y)$ and $E(||Y - Y^{\backslash}||)$ for any $x_i \in [a, b]$ as follows:

$$E(|x_i - Y|) = \int_a^b |x_i - y| \frac{1}{b-a} dy$$

If $y \leq x_i$ then $|x_i - y| = x_i - y$, and if $y > x_i$ then $|x_i - y| = y - x_i$, therefore we have

$$E(|x_i - Y|) = \int_a^{x_i} (x_i - y) \frac{1}{b-a} dy + \int_{x_i}^b (y - x_i) \frac{1}{b-a} dy$$

Computing the first term we have

$$\int_a^{x_i}(x_i-y)\frac{1}{b-a}dy=\frac{1}{b-a}\int_a^{x_i}=\frac{1}{b-a}(\int_a^{x_i}x_i-\int_a^{x_i}y)=\frac{1}{b-a}(x_i^2-ax_i-\frac{x_i^2}{2}+\frac{a^2}{2})=$$

$$\frac{x_i^2-ax_i}{b-a}+\frac{a^2-x_i^2}{2(b-a)}=\frac{x_i^2-2ax_i+a^2}{2(b-a)}=\frac{(x_i-a)^2}{2(b-a)}$$

The second term is computed similarly and will be

$$\frac{(x_i-b)^2}{2(b-a)}$$

So $E(|x_i-Y|)$ is

$$E(|x_i-Y|)=\frac{(x_i-a)^2+(x_i-b)^2}{2(b-a)}$$

Now we need to calculate the second quantity, $E(|Y-Y^{\backslash}|)$. This can be derived easily without recourse to complicated integrals by using our previous work in the following way: Using the law of total expectation we have

$$E(|Y-Y^{\backslash}|)=E(E(|Y-Y^{\backslash}|;Y))$$

The interior expectation is the exact previous case we have already solved, since we can treat Y as constant, so we will now have

$$E(E(|Y-Y^{\backslash}|;Y))=E(\frac{(Y-a)^2}{2(b-a)}+\frac{(Y-b)^2}{2(b-a)})=\frac{1}{2(b-a)}E((Y-a)^2+(Y-b)^2)=$$

$$\frac{1}{2(b-a)}E(2Y^2-(2a+2b)Y+a^2+b^2)=\frac{1}{2(b-a)}(2E(Y^2)-2(a+b)E(Y)+a^2+b^2)=$$

$$\frac{1}{2(b-a)}(2E(Y^2)-2(a+b)E(Y)+a^2+b^2$$

At once– hopefully– we remember the formula for the variance, which states that

$$Var(Y) = E(Y^2) - E(Y)^2$$

And so we have

$$\frac{1}{2(b-a)}(2E(Y^2)-2(a+b)E(Y)+a^2+b^2) = \frac{1}{2(b-a)}(2Var(Y)+E(Y)^2-2(a+b)E(Y)+a^2+b^2) =$$

$$\frac{1}{b-a}(\frac{a^2+b^2-2ab}{12} + \frac{a^2+b^2+2ab}{4} + \frac{-a^2-b^2-2ab}{2} + \frac{a^2+b^2}{2})$$

which will equal, by subtracting all the terms,

$$\frac{1}{b-a}(\frac{16a^2+16b^2+16ab}{48} - ab) = \frac{1}{b-a}(\frac{a^2+b^2-2ab}{3}) = \frac{(b-a)^2}{3(b-a)} = \frac{b-a}{3}$$

And we are done! We have calculated both $E(|x_i - Y|)$ and $E(||Y - Y^{\setminus}||)$, and so we can easily simulate from $\mathcal{U}(a,b)$ to create the empirical distribution for $n\mathcal{E}$. Naturally, our calculations were so simple and elegant only because the uniform distribution *is* a very simple distribution. One can easily imagine distributions that would make one shiver in fear of calculating $E(|x_i - Y|)$ and $E(||Y - Y^{\setminus}||)$. But there is no actual need to find a closed form for these two terms. It would be helpful, of course, for computational and algorithmic purposes, but we could just as well use numerical integration. Let's see another example, just to realize how complex the calculations can get.

Example 3.2. Assuming we want to test the hypohesis that F_0 is the n-dimensional standard normal distribution, such that $Y \sim \mathcal{Z}(0,1)$, we will have

$$E(||Y - Y^{\setminus}||) = 2\frac{\Gamma((n+1)/2)}{\Gamma(n/2)}$$

and

$$E(||x_i - Y||) = \frac{\sqrt{2}\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n}{2})} + \frac{2}{\pi}\sum_{i=0}^{\infty}\frac{(-1)^i}{i!2^i}\frac{||x_i||^{2i+2}}{(2i+1)(2i+2)}\frac{\Gamma(\frac{n+1}{2})\Gamma(i+\frac{3}{2})}{\Gamma(i+\frac{n}{2}+1)}$$

One had better not think about how these results were arrived at.

Goodness-of-fit-tests based on energy statistic naturally have something to offer that differs from the standard and wide-spread tests, else why should we have implemented them? In the interest of conciseness, we shall mention only two:

1. Energy goodness-of-fit tests are nearly always more powerful than their standard counterparts. In other words, we are more likely to reject the null hypothesis given that it's false when we use an energy goodness-of-fit test than a standard test.

2. Energy goodness-of-fit tests can easily be modified to work in the multi-dimensional case; a negligible benefit one might say, but then almost certainly he hadn't gone through the nightmare of trying to make the Kolmogrov-Smirnov test work with multidimensional data. Just ask Scipy[1] maintainers; they have implemented the Kolmogrov-Smirnov test only for the one-dimensional case.

To demonstrate how big an improvement energy goodness-of-fit tests are, let's see an example.

Example 3.3. We will compare the power of the standard Kolmogrov-Smirnov test and the energy test for the normal distribution. First, we will simulate several thousands datasets of 300 data points from the student t distribution. On each dataset we will use the standard test and the energy test. Naturally, since the data doesn't come from the normal distribution, we would expect the better test to reject the null hypotheis more frequently, i.e, to have more power.

[1]The preeminent python library for *scientific* computation

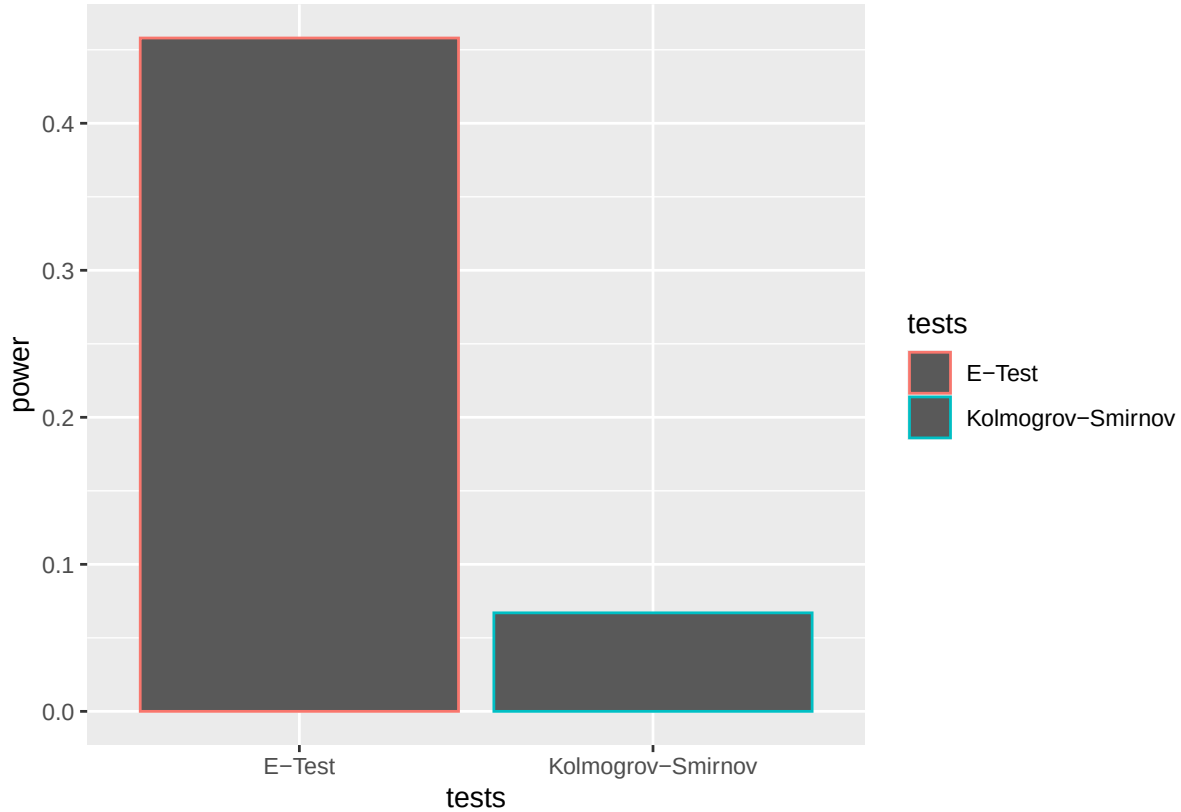

Figure 3.1: Comparison between the Kolmogrov-Smirnov test and the e-test

The energy rejects the null hypothesis nearly 50 percent of the time, while the Kolmogrov-Smirnov test rejects it only 5 percent of the time.

3.2 Distance Correlation

Of the many data analysis tools available to the non-expert, none is perhaps more well known than Pearson's correlation coefficient, defined as, between two random variables X and Y

$$Cor(X,Y) = \frac{Cov(X,Y)}{\sqrt{var(X)var(Y)}}$$

A measure between 0 and 1 which equals 1 if and only if the two random variables being tested are linearly related and 0 if not; a good enough thing, one might say, except that there are many forms of dependency other than the linear at which the bias of the correlation coefficient appears clear as day.

Example 3.4. Let X be a random variable which has the following distribution

$$X \sim U(-10, 10)$$

and we will then define the random variable Y as

$$Y = X^2$$

If we simulate several values from X and Y we will get the following plot:

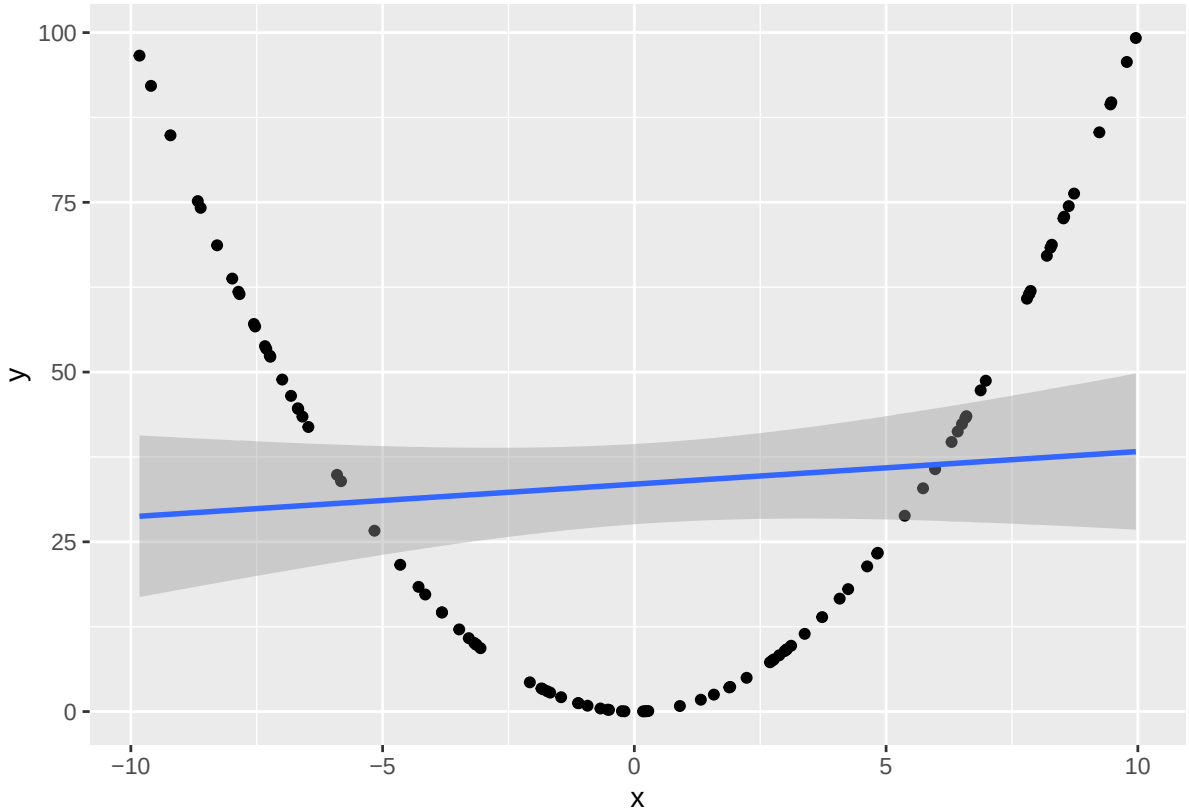

Figure 3.2: An example at which the correlation coefficent fails utterly

As one can see from the blue line (which we fitted by least squares) there exists no linear relationship between the two variable. And if we compute the correlation coefficient it will be 0. But X and Y are certainly dependent.

What we decided to add to *estats* then is a different measure of dependency, one that is capable of discovering any pattern of relation that exists between two random variables. Such a measure, let's call it R, should satisfy two properties:

- $0 \leq R \leq 1$
- $R = 0$ if and only if the two random variables being tested are independent

We will now discuss how to define R so that it satisfy these two properties. First of all, let's recall the definition of independence. Two random variables X and Y with pdfs f_X and f_Y and which has a joint distribution f_{XY} are independent if and only if

$$f_{XY} = f_X f_Y$$

A metric of general dependence would then have to measure the distance between the joint distribution pdf and the individual pdfs of each random variables. More clearly, we need to find, assuming X is n-dimensional and Y is k-dimensional

$$\int_{R^n \times R^k} ||F_{XY} - f_X f_Y||^2$$

Without getting into too much details, the estimator for this will be

$$V(X,Y) = \sum_{i,j} \frac{1}{n^2} \bar{D}^{ij}_{xx} \bar{D}^{ij}_{yy}$$

And we will call the quantity V the *distance covariance*. The distance covariance has the useful properties of being invariant to scaling, shifting, and orthogonal transformations:

$$V(a_1 + b_1 C_1 X, a_2 + b_2 C_2 Y) = b_1 b_2 V^2(X,Y)$$

much like the covariance $E((X - \bar{X})(Y - \bar{Y}))$ of basic statistics. We can define the distance correlation R using V^2 as follows,

$$R = \frac{V(X,Y)}{\sqrt{V(X,X), V(Y,Y)}}$$

And we see the beautiful symmetry that exists between the distance correlation and the correlation coefficient. The distance correlation is far more powerful than the correlation coefficient primarily for two reasons:

1. $R(X,Y)$ is defined even when X and Y are random vector, unlike Pearson's correlation coefficient.

2. R can detect any degree of dependency.

Consider the following table that compares between the two, there the random variable X is uniformly distributed between $[-10, 10]$:

Formula	Correlation Coefficent	Distance Correlation
y = x^2	-0.0155670	0.4935853
y = 7x + 6	1.0000000	1.0000000
y = e^x - 6x^3	-0.2635365	0.6152393
y = sin(x) + cos(x) - 6	0.1169696	0.2032442
y = gamma(abs(x)) - x^2	0.0072091	0.3011515

For some further examples to compare between the correlation coefficient and the distance correlation, it's possible to generate several famous graphs on which the correlation coefficient does well (i.e, show that there is a linear relationship when the variables are dependents) and some on which it does not do very well (does not discover the relationship between the variables). Surely it interests us very much to consider how the distance correlation does in these circumstances.

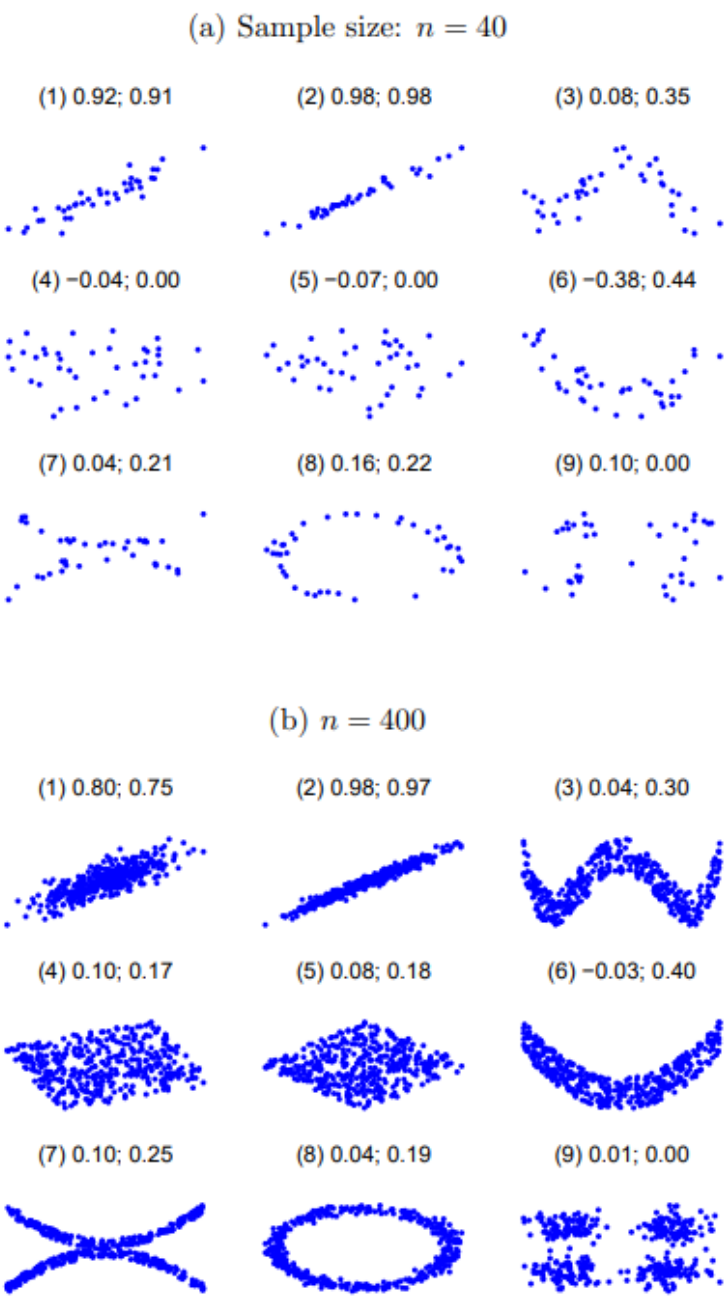

Figure 3.3: Comparison of the Pearson's correlation and the distance correlation in nine cases. In each sub-figure , the two coordinates correspond to the randov variables (X, Y). Each dot is a sample point. In the title, the first value is the Pearson's correlation, and

3.3 On The Computational Efficiency of Dcor

It may have appeared from our previous discussion that the distance correlation is a magical technique with no deficiencies, to be used in place of the distance correlation in all cases, and if it cannot be used, then, in fact, *nothing* should be used. Sadly, a pessimist may say, one can't have one's cake and eat it. One must give away a slice; or two; or three. And the slice that we must give away in the case of *dcor* is the sweet, moist slice of computational efficiency. Note: computational efficiency as in both time and space efficiency. Why this is so is obvious. To compute *dcor* on two variables X and Y of size n, we have to compute D_x and ${D_y}^2$. We do not only have to *compute* them, we also have to *store* them, and so both time and space complexity will be $O(n^2)$. Is this bad? For the naive questioner, a simple illustration: Assuming $n = 10^6$, and that we store each element of the distance matrix as a 32 bit floating point number, to store the whole distance matrix we will need approximately 7TB of memory; if each element is instead stored as a 64 bit floating point number, we will need about 14 TB. So bad, yes.

But this one rare time the pessimist is mistaken. For in [9] Xiaoming Huo and Gabor J. Szekely show there is an algorithm with $O(n \log n)$ running time that does not require the storage of the distance matrix. An attempt at a derivation will useless if the goal is clarity, and moreover fruitless. We shall therefore discuss the improved algorithm in only general terms, making sure to empathize its indispensable components.

First of all we remember that the distance covariance is:

$$V(X,Y) = \sum_{i,j} \frac{1}{n^2} \bar{D}^{ij}_{xx} \bar{D}^{ij}_{yy}$$

And this gives us our first piece of evidence that an improvement over the naive algorithm is possible. If, somehow, we are able to find a way to compute the mean of elementwise product of two distance matrices in time $O(n \log n)$, then we are set. Denoting our distance matrices as A and B, and their ith elements as a_i, b_i, our problem then is the computation of

[2]One may object that we do not actually have to compute the whole matrix. A distance matrix is symmetric, therefore we have only to compute half the number of elements; but such multiplication by a constant does not affect the space and time complexity

$$\sum_{i \neq j} a_{ij} b_{ij}$$

in $O(n \log n)$ time. Is that possible? It is, Xiaoming Huo and Gabor J. Szekely answer, and their answer is as follows.

First we define a sign function S_{ij}, such that $\forall 1 \leq i, j \leq n$

$$S_{ij} = \begin{cases} +1, & \text{if } (x_i - x_j)(y_i - y_j) > 0, \\ -1, & \text{otherwise.} \end{cases}$$

For any sequence $\{c_j, j = 1, \ldots, n\}$, for $1 \leq i \leq n$, we define

$$\gamma_i(\{c_j\}) = \sum_{j \neq i} c_j S_{ij}.$$

We will then have that

$$\sum_{i \neq j} a_{ij} b_{ij} = \sum_{i \neq j} |x_i - x_j| \cdot |y_i - y_j|$$

$$= \sum_{i=1}^{n} \sum_{j \neq i} (x_i y_i + x_j y_j - x_i y_j - x_j y_i) S_{ij}$$

$$= \sum_{i=1}^{n} \left[x_i y_i \sum_{j \neq i} S_{ij} + \sum_{j \neq i} x_j y_j S_{ij} - x_i \sum_{j \neq i} y_j S_{ij} - y_i \sum_{j \neq i} x_j S_{ij} \right]$$

And using the definition of $gamma_i$ we get that

$$\sum_{i \neq j} a_{ij} b_{ij} = \sum_{i=1}^{n} \left[x_i y_i \gamma_i(\{1\}) + \gamma_i(\{x_j y_j\}) - x_i \gamma_i(\{y_j\}) - y_i \gamma_i(\{x_j\}) \right]$$

Our problem, then, is reduced to finding a way to compute $gamma_i(\{c_j\})$ in $O(n \log n)$ time. This $O(n \log n)$ is as follows.

Algorithm: Fast Computing for Distance Covariance (FaDCor)

Inputs: Observations $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$.

Outputs: The distance covariance of X and Y.

1. Sort $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$. Let F^x and F^y denote the order indices; i.e., if for i, $1 \le i \le n$, $F^x(i) = k$, then x_i is the kth smallest observation among $x_1, \ldots, x_n$. Similarly, if $F^y(i) = k$, then y_i is the kth smallest among $y_1, \ldots, y_n$.

2. Let $x_{(1)} < \cdots < x_{(n)}$ and $y_{(1)} < \cdots < y_{(n)}$ denote the order statistics. Denote the partial sums:

$$s^x(i) = \sum_{j=1}^{i} x_{(j)}, \quad s^y(i) = \sum_{j=1}^{i} y_{(j)}, \quad i = 1, \ldots, n.$$

   They can be computed using the recursive relation: $s^x(1) = x_{(1)}, s^y(1) = y_{(1)}$,

$$s^x(i+1) = s^x(i) + x_{(i+1)}, \quad s^y(i+1) = s^y(i) + y_{(i+1)}, \quad \text{for } i = 1, \ldots, n-1.$$

3. Compute α_i^x, α_i^y, β_i^x, and β_i^y using:

$$\alpha_i^x = F^x(i) - 1, \quad \alpha_i^y = F^y(i) - 1,$$

$$\beta_i^x = s^x(F^x(i) - 1), \quad \beta_i^y = s^y(F^y(i) - 1).$$

4. Compute a_i and b_i per their definitions.

5. Compute $\sum_{i=1}^{n} a_i b_i$.

6. Compute $a_{\cdot}$ and $b_{\cdot}$.

7. Use Algorithm `PartialSum2D` to compute for $\gamma(\{1\})$, $\gamma(\{x_j y_j\})$, $\gamma(\{y_j\})$, and $\gamma(\{x_j\})$.

8. Compute $\sum_{i \neq j} a_{ij} b_{ij}$.

Algorithm 1: The $\mathcal{O}(n \log n)$ algorithm to compute the distance covariance.

Algorithm: Fast Algorithm for a 2-D Partial Sum Sequence (PartialSum2D)

Inputs: Observations $x_1, \ldots, x_n$, $y_1, \ldots, y_n$, and $c_1, \ldots, c_n$.

Outputs: Quantity $\gamma_i(\{c_j\}) = \sum_{j \neq i} c_j S_{ij}$ that is defined in Lemma 4.4.

1. Compute the order statistics $x_{(1)} < \cdots < x_{(n)}$ for $x_1, \ldots, x_n$. Then rearrange triplets (x_i, y_i, c_i) such that we have $x_1 < \cdots < x_n$. Each triplet (x_i, y_i, c_i) $(1 \leq i \leq n)$ stays unchanged.

2. Let $y_{(1)} < \cdots < y_{(n)}$ denote the order statistics for $y_1, \ldots, y_n$, and assume that $I^y(i), i = 1, \ldots, n$, are the order indices; i.e., if $I^y(i) = k$, then y_i is the kth smallest among $y_1, \ldots, y_n$. Without loss of generality, we may assume that $y_i = y_{(i)}$.

3. Evidently, the aforementioned function $I^y(i)$ is invertible. Let $(I^y)^{-1}(j)$ denote its inverse. Define the partial sum sequence: for $1 \leq i \leq n$,

$$s^y(i) = \sum_{j=1}^{i} c_{(I^y)^{-1}(j)}.$$

   The following recursive relation enables an $O(n)$ algorithm to compute all $s^y(i)$s,

$$s^y(1) = c_{(I^y)^{-1}(1)}, \quad s^y(i+1) = s^y(i) + c_{(I^y)^{-1}(i+1)}, \quad \text{for } i \geq 1.$$

4. For $1 \leq i \leq n$, define

$$s^c(i) = \sum_{j=1}^{i} c_j.$$

   Again, the above partial sums can be computed in $O(n)$ steps.

5. Compute $c_\cdot = \sum_{j=1}^{n} c_j$.

6. Call Subroutine `DyadUpdate` to compute $\sum_{j:x_j<x_i,y_j<y_i} c_j$ for all $1 \le i \le n$.

7. By (A.6), we have that

$$\gamma_i(\{c_j\}) = c_\cdot - c_i - 2s^y(i) - 2s^c(i) + 4 \sum_{j:x_j<x_i,y_j<y_i} c_j.$$

Algorithm 2: A subroutine that will be needed in the fast algorithm for the distance covariance.

Subroutine: A Dyadic Updating Scheme (DyadUpdate)

Inputs: Sequence $y_1, \ldots, y_n$ and $c_1, \ldots, c_n$, where $y_1, \ldots, y_n$ is a permutation of $\{1, \ldots, n\}$.

Outputs: Quantities $\gamma_i = \sum_{j:j<i,y_j<y_i} c_j$, $i = 1, 2, \ldots, n$.

1. Recall that we have assumed $n = 2^L$. If n is not dyadic, we simply choose the smallest L such that $n < 2^L$. Recall that for $\ell = 0, 1, \ldots, L-1$, $k = 1, 2, \ldots, 2^{L-\ell}$, we define a closed interval

$$I(\ell, k) := [(k-1) \cdot 2^\ell + 1, \ldots, k \cdot 2^\ell].$$

2. Assign $s(\ell, k) = 0$, $\forall \ell, k$, and $\gamma_i = 0$.

3. For $i = 2, \ldots, n$, we do the following.

   (a) For all (ℓ, k)'s, such that $y_i - 1 \in I(\ell, k)$. Then for these (ℓ, k)'s, do update

$$s(\ell, k) \leftarrow s(\ell, k) + c_{y_i - 1}.$$

   (b) Find nonnegative integers $\ell_1 > \cdots > \ell_\tau \ge 0$ such that

$$y_i - 1 = 2^{\ell_1} + \cdots + 2^{\ell_\tau}.$$

Let $k_1 = 1$. For $j = 2, \ldots, \tau$, compute

$$k_j = (2^{\ell_j+1} + \cdots + 2^{\ell_j-1}) \cdot 2^{\ell_j} + 1.$$

(c) Compute $\gamma_i = \sum_{j=1}^{\tau} s(\ell_j, k_j)$.

Algorithm 3: A subroutine that will be called in Algorithm 2.

Now, *theoretically*, this algorithm is vast improvement on our old, naive way of computing the distance covariance. Yet practically we are faced with two problems:

- This dependency on sorting is not good, not good at all. We can sort a data set X when $X \subset R^1$ easily enough, but what if $X \subset R^2$, or R^3, or any R^n such that $n > 1$? We cannot sort vectors. Therefore our new algorithm will be useless and we will have to resort to the $O(n^2)$ algorithm.
- Several steps in the new algorithm cannot be vectorized, i.e, we are unable to write it in pure Numpy, and so will have to use for loops, leading to a great increase in the running time.

We will discuss each of these two problems, starting with the second one. In the next section, we shall discuss the first.

To deal with the second problem, we implemented the improved algorithm with Numba instead of Numpy. Numba is an open-source just-in-time (JIT) compiler that translates a subset of Python and NumPy code into fast machine code at runtime. In other words, Numba enabled us to write code with as much for loops as we liked, trusting in Numba to compile that part of the code during runtime. Ignoring the first problem of multivariate datasets for a moment, let us see how much of a speedup we have extracted (or better say exploited) using the $O(n \log n)$ algorithm and Numba.

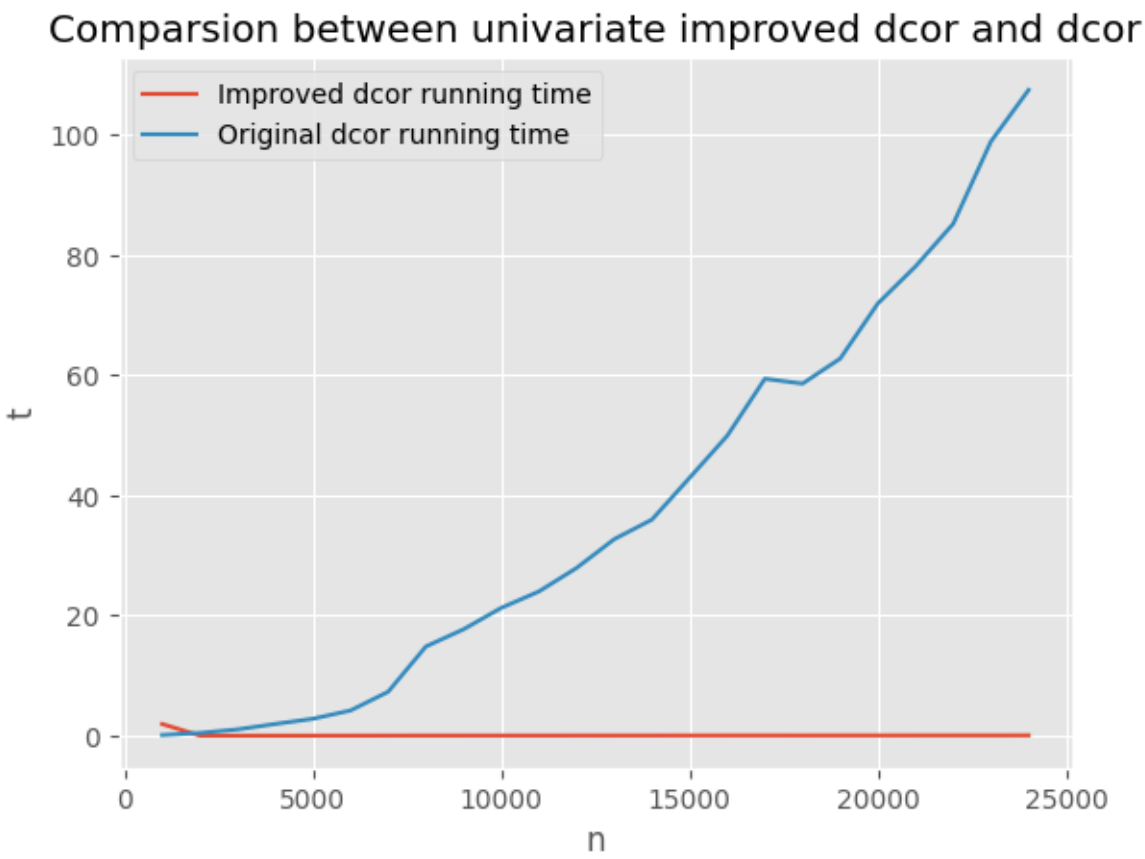

Figure 3.4: Speed Comparsion between dcor and fast dcor.

A comparison with the y-axis being $\log t$ instead of t might make things clearer (for it is obviously not the case that our function computes the distance correlation in 0 time; we are not magicians):

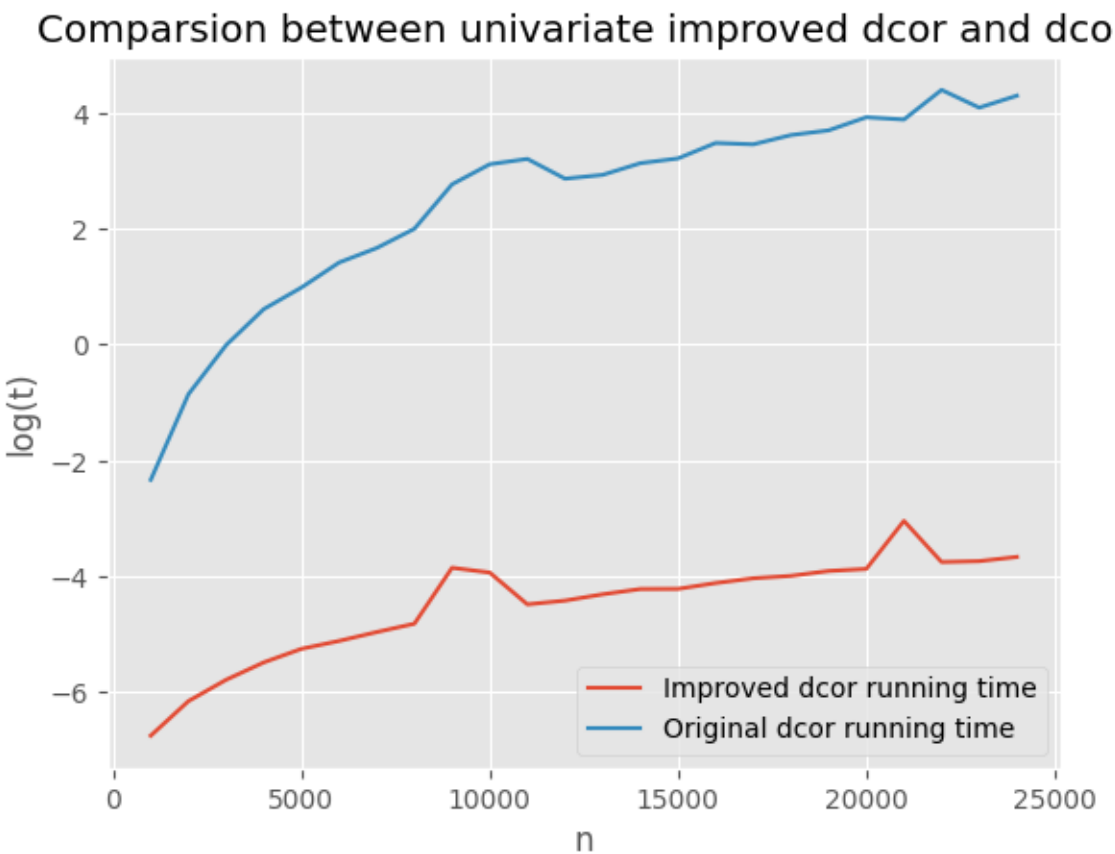

Figure 3.5: Speed Comparsion between dcor and fast dcor with the y axis being log t.

3.4 Multivariate Dcor

So much then for an improved single variable dcor.

In R^1 we are untroubled; we have an *exact* algorithm whose speed is incomparable to the previous one. But in any other R^n we have no choice other than to recourse to the slow algorithm.

Naturally, it is to be hoped that there exists an algorithm than can work in the multivariate case. Any such algorithm can be classified into two types, depending on its relation to the

other single variable algorithm:

1. The multivariate algorithm is completely new, using no previous work.
2. Under the hood the multivariate algorithm uses the single variable algorithm.

We are not Einsteins. To develop an algorithm of the first kind is beyond our capabilities. And yet the second way seems obscure; in what way are we meant to use the single variable algorithm? Given a dataset in R^n such that $n > 1$ we might think of *transforming* the dataset into R^1, but that necessarily involves information loss. So what is to be done? Well, instead of transforming the random vector into only one single random variable in R^1, we might transform it into several such datasets, and take the distance correlation of the multivariate datasets to the be mean of the distance correlation of the transformed datasets.

Will that work? Intuitively, we would say yes. Any relation between the target variable y and the explanatory variables X, ought to be, in some sense, *found* in the relation between y and a random variable x in R^1.

To make this more precise we should speak of a random *projection* instead of a random variable. A review of what it means to project onto a random subspace could prove helpful, though certainly the reader is a master of far more esoteric mathematical topics, and I realize I do him a great injustice and injury in even thinking of discussing this; but needs must, and so on and so forth.

Given a vector $v \in R^n$ it's sometimes useful (as is the case here) to encode it using a single number $x \in R^3$. If we have another vector $c \in R^n$ we might use it to encode v in a single real number k as follows:

$$k^* = \arg\min_{k \in \mathbb{R}} ||kc - v||$$

And k^* turns out to be

$$k^* = \frac{v \cdot c}{||c||^2}$$

[3]Or, in general, a single vector $x \in R^k$ such that $k < n$

Now obviously by such an encoding we lose so great an amount information that the difference $(dcor(v, y) - dcor(k, y))$ will be quite large. But what if we project onto so many random vectors $c_1, c_2, \ldots, c_n$ and in the end take their mean? Intuitively it appears that as $n \to \infty$ we will have that

$$\frac{dcor(k_1, y) + dcor(k_2, y) + \cdots + dcor(k_n, y)}{n} \to dcor(v, y)$$

That is exactly what is proved in Huang and Huo [2017].

And moreover, the algorithm is not only consistent but also unbiased. To check its speed, we have compared it with the old algorithm for the multivariate case:

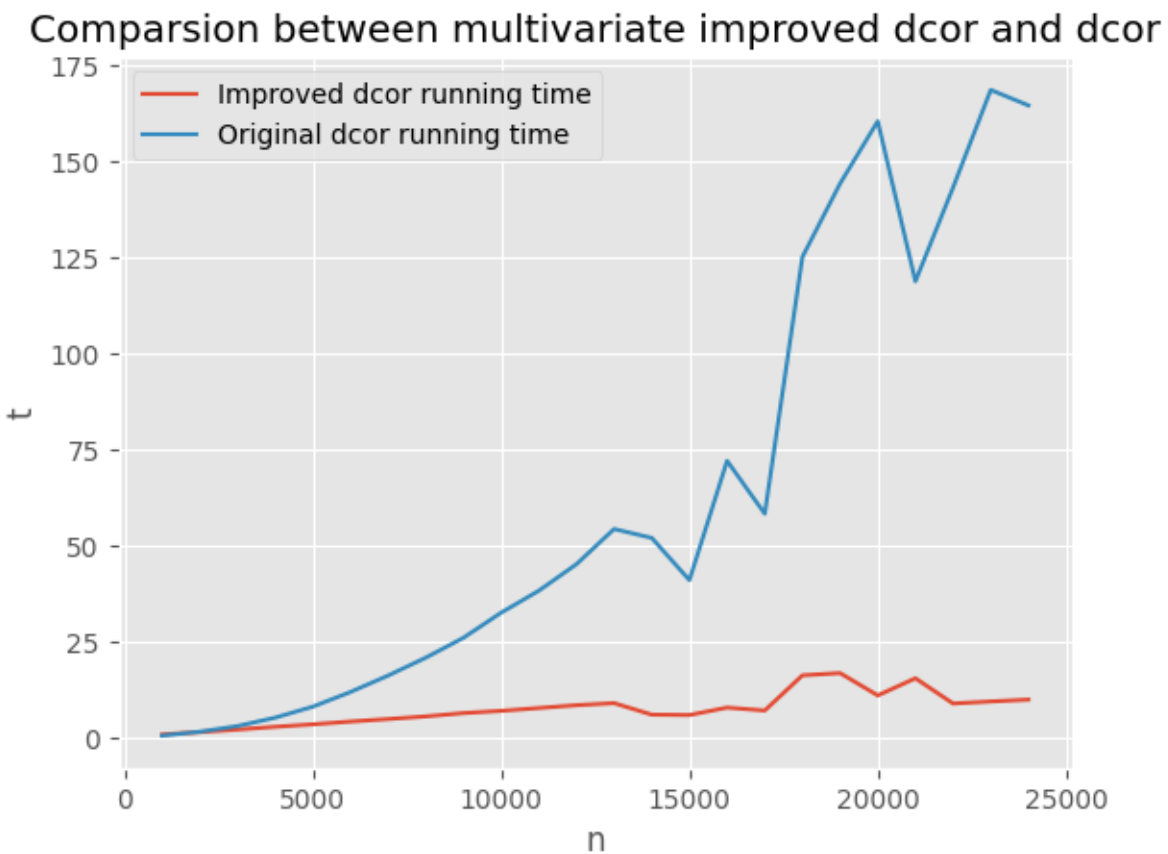

Figure 3.6: Speed Comparsion between dcor and fast dcor in the multivariate case.

3.5 Two Sample Test

Let's assume that we have samples from a random vectors x_1, x_2, x_3, ..., and samples from *another* random vector y_1, y_2, y_3, ... and we want to test whether their distributions F and G are equal. Naturally we straight away think about the energy distance

$$\mathcal{E}(X,Y) = 2\int_{R^n} ||F(\mathbf{x}) - G(\mathbf{x})||^2 = 2E(||X-Y||) - E(||X-X^{\backslash}||) - E(||Y-Y^{\backslash}||)$$

How are we to use it for our current problem?[4]? Well, we can plug in the empirical cumulative distribution functions, so that $X \sim F_N$ and $Y \sim G_n$, and our energy distance statistic then becomes

$$\overline{\mathcal{E}}(X,Y) = \frac{2}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m}||x_i - y_i|| - \frac{1}{n^2}\sum_{i=1,j=1}^{n}||x_i - x_j|| - \frac{1}{m^2}\sum_{i=1,j=1}^{m}||y_i - y_j||$$

The question then remains of what our distribution under the null hypothesis looks like. Under the null hypothesis, the distribution of X and Y are the same, and so we can in a certain sense *pool* the data into a set P

$$\{x_1, x_2, \dots x_n, y_1, y_2, ..., y_n\}$$

And to simulate from it, we can form a new data set Z of size n by choosing points at random from P, so that

$$Z = \{z_1, z_2, \dots, z_n\}$$

where each $z_i \in P$. Let's now see how this test works in the wild.

Example 3.5. A popular dataset is the *birthwt* dataset, which records the birth weigh of several newborns, alongside an indicator variable recording whether their mother smoked or not. Naturally, it proves of interest to compare the weight distributions of newborn whose mother didn't smoke with the weight distribution of newborns whose mother did smoke. If we plot their histograms, we will see:

[4]A little thought does wonders for the brain; it causes confusion, and confusion leads to even more confusion, until one is sucked into a whirlpool of tautologies, contradictions and absurdities, and it appears an utter certainty both that he knows everything and that he knows nothing.

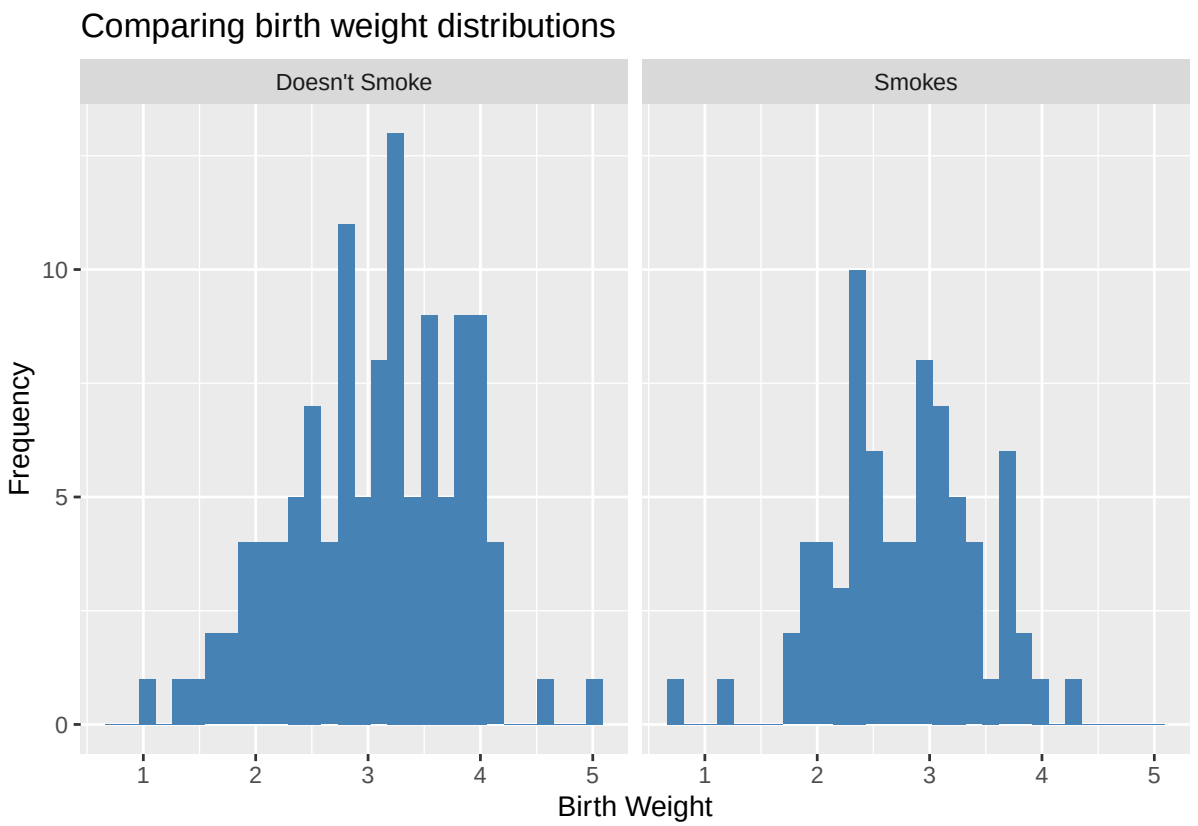

Figure 3.7: Comparing birth weight distributions

we see that the distributions are somewhat similar, with mild differences.

Variable	**Doesn't Smoke**, N = 115	**Smokes**, N = 74	**p-value**[1]
bwt			0.007
Mean (Variance)	3.06 (0.57)	2.77 (0.44)	
Median (IQR)	3.10 (2.51, 3.62)	2.78 (2.37, 3.25)	
Range	1.02, 4.99	0.71, 4.24	

Table 3.2: Two Sample Test

	Test	Statistic	Pvalue
E-statistic	Energy Test	3.748466	0.0049751

Variable	**Doesn't Smoke**, N = 115	**Smokes**, N = 74	**p-value**[1]

[1]Wilcoxon rank sum test

We see that the mean and the median of for the first category is higher, and that the Wilconx test rejects the null hypothesis. Our test does much the same:

3.6 K-Groups Clustering and Disco

3.6.1 K-Groups

To explain k-groups clustering, a new clustering method based on energy statistics, let's first recall the k-means clustering method. It assumes that data can be clustered into k groups such that the difference in the distribution of the two groups consist only in the difference of means. Basically, given two clusters π_k and π_l generated by the k-means algorithm, we expect their means to be different– but that is all! K-means looks no further than the first moment. It pay no heed to $E(X^2)$, $E(X^3), ..., E((X - \bar{X})^2)$ and so on and so forth. What mprovement there would be, if k-means measured the differences between *distributions* instead of only the difference between means.

That is exactly what the k-groups clustering algorithm is about. It uses the two sample to test to measure the difference between any two clusters. The algorithm goes like this:

First, we define the dispersion between two sets A and B as

$$g(A, B) = \sum_{i,j} ||a_i - b_j||$$

First, we randomly assign the data points to k clusters. We define the total dispersion given a data set X as

Table 3.3: Comparsion of K-means and K-groups

	k_means	k_groups
Rand	0.9052627	0.9503936
cRand	0.7055921	0.8561299

$$T(\pi_1, \pi_2, \ldots, \pi_k) = g(X, X)$$

and the within groups dispersion as

$$W(\pi_1, \pi_2, \ldots, \pi_k) = \sum_{j=1}^{k} \frac{n_j}{2} g(\pi_j, \pi_j)$$

while the between sample dispersion is

$$B(\pi_1, \pi_2, \ldots, \pi_k) = \sum_{1 \leq i,j, \leq k} \bar{\mathcal{E}}(\pi_i, \pi_j)$$

The updating step is the same as k-means. We try to assign the points in such a way that we minimize W and maximize B.

Example 3.6. The Dermatology dataset measures several health characterstics of patients suffering from the Eryhemato-Squamous Disease. This disease comes in six types: Psoriasis, seboreic dermatitis, lichen planus, pityriasis rosea, cronic dermatitis, and pityriasis rubra pilaris. Patients were first evaluated clinically with 12 features. Afterwards, skin samples were taken for the evaluation of 22 histopathological features. The values of the histopathological features were determined by an analysis of the samples under a microscope. The type of the disease of each patient was measured as well.

We will compare the k-means and k-groups algoritms on this dataset with $k = 6$ using the Rand index and the corrected Rand index. The results were as follows:

We see that k-groups does better on both metrics, and that it vastly improves on the k-means algorithm's corrected Rand score.

Table 3.4: DISCO Table

Test	Statistic	Pvalue
Distance Component Analysis	1849.29	0.001

3.6.2 Disco

Disco, an alternative to Anova, tests equality of distributions instead of equality of means. There is no need to explain the theory in detail, because all we have said about k-groups clustering applies. Both techniques are identical. One can think of Disco as applying k-groups with pre-determined clusters, or one can think of k-groups as trying to find partitions over which we can reject the null hypothesis of the equality of distributions which the Distance components algorithm tests.

The within-sample dispersion statistic W and the total dispersion T are defined as in the k-groups algorithm, though in this case we have pre-determined clusters representing the different groups whose equality of distributions we wish to test. Their sum then, the between sample statistic B, is an approximation to the average pairwise two sample energy statistics between all groups[5]

For the test our statistic, or our "F-ratio", will be

$$F = \frac{S/(k-1)}{W/(n-k)}$$

a statistic perfectly analogous to that of the ANOVA test.

Example 3.7. The penguins dataset contains measurments (for flipper length, body mass, bill dimensions and sex) done on penguins living on three different islands (Biscoe, Dream and Torgersen).

Using DISCO to check for differences between the penguins living on different islands, we will have:

[5]If all groups are of the same size, then it *will* be the average of the average pairwise two sample energy statistics between all groups

3.7 Outlier Detection

Outlier detection is a thorny problem. We wish not to make light of things, therefore let us start with the question: What *is* an outlier? It is a data point that appears not to conform to our notion of normal behavior. The concept of an *outlier* is then a dependent concept; it is not an objective fact or a thing in itself. Therefore it is, like all subjective concepts, dangerous to use mechanically.

We might define outliers as:

1. A data point is an outlier if its method of collection differs from that of other points in the dataset.
2. A data point is an outlier if it possesses low probability given a certain data generating distributions that we have assumed and whose parameters we have estimated.
3. A data point is an outlier if a certain measure is sensitive to it. For example, the measure might be the mean squared error, and a certain data point is considered an outlier if on removing it the mean squared error either increases or decreases dramatically. The measure might or might not depend on a particular model. In our example, if we compute the mean squared error with respect to a linear regression model, then our data point is an outlier with reference to not only the mean squared error measure but also to the linear regression model.

The method we have implemented is of the third type. Models of the third type can be further classified into two categories: Those that are model dependent, and those that are model independent. Naturally, given all that we have said so far about Estats, it would be a great disappointment if our method turned out to be model dependent. Spoiler: It is not.

We simply use the distance correlation as measure, in order to discover which observations in the data are *influential*. Note our use of the word *influential*; we use that word because, really, finding influential observations is all that outlier detection methods do. If a data set is small, then we would expect *all* observations to be influential. If it is large, then almost certainly there will be a few influential observation; and there is no getting out of this dilemma: Given the influential points the outlier detection methods have produced, which of them are, in some sense, *natural*, and which are, in some sense, *unnatural*? Such a judgment must be taken and can by no means be automated. But it is not as if this

is a great failure which ought be lamented. For, supposedly, the one behind the screen using these methods is not a monkey, and, given that he is not monkey, he should be prove capable of deciding which points ought be discared[6] and which points ought to be explained away by a causal mechanism for their being influential.

Our method is quite simple and obvious. It is a jack knife procedure. We First of all calculate the distance correlation between the target variable and the explanatory variables on the whole data set X getting a value R, and then we remove each observation i for $1 < i < n$ so that our dataset become X^{-i} and we calculate the distance correlation on that dataset, getting a value R_i. Once we finish this procedure, we compare each R_i with R, and if $|R - R_i|$ is too large, we consider observation number i to be *influential*. Naturally we expect $|R - R_i|$ to be normally distributed, so we give the user the option to decide whether sensitivity of the algorithm by deciding the parameter c such that an observation i is considered influential iff $p(|R - R_i|) < c$.

A simple example might prove enlightening.

Example 3.8. We will implement the outlier detection method we have described on a dataset of explanatory variables on each american state alongside the crime rate on that state.

The variables are:

- **Population**: Total number of citizens in 1968, measured in thousands.
- **Nonwhite** : The percentage of non-white population in 1960.
- **Density**: The population per square mile in 1968.
- **Crime**: The crime rate per 100,000 citizen in 1969.

The distance correlation as computed on the whole dataset is 0.390433. Let us now see how it changes on removing each observation:

[6]Because they are artifacts; because they are of mistakes made during the data collection process; because they will cause our model to be confused in way we do not want it to be confused in; or because they are proving to be hindrances in our dastardly plan of getting p-values <= 0.05

Table 3.5: Replicates by City

replicates	point_labels	differences
0.390156	Akron	0.000277
0.391328	Albany	0.000895
0.390006	Allentown	0.000427
0.388642	Anaheim	0.001791
0.390664	Atlanta	0.000231
0.397042	Bakersfield	0.006609
0.381676	Baltimore	0.008757
0.387503	Beaumont	0.002930
0.384980	Binghamton	0.005453
0.390359	Birmingham	0.000074
0.397195	Boston	0.006762
0.390520	Bridgeport	0.000087
0.395497	Buffalo	0.005064
0.389072	Canton	0.001361
0.397836	Charlotte	0.007403
0.390569	Chattanooga	0.000136
0.415047	Chicago	0.024614
0.398946	Cincinnati	0.008513
0.388850	Cleveland	0.001583
0.390412	Columbus	0.000021
0.394829	Corpus.Christi	0.004396
0.387336	Dallas	0.003097
0.387062	Davenport	0.003371
0.390902	Dayton	0.000469
0.391081	Denver	0.000648
0.374539	Detroit	0.015894
0.386647	Duluth	0.003786
0.389019	El.Paso	0.001414
0.394064	Flint	0.003631

Table 3.5: Replicates by City *(continued)*

replicates	point_labels	differences
0.398097	Fort.Lauderdale	0.007664
0.391227	Fort.Worth	0.000794
0.401635	Fresno	0.011202
0.392953	Gary	0.002520
0.388899	Grand.Rapids	0.001534
0.389157	Greensboro	0.001276
0.386326	Harrisburg	0.004107
0.391529	Hartford	0.001096
0.391224	Honolulu	0.000791
0.386350	Houston	0.004083
0.390774	Indianapolis	0.000341
0.399412	Jacksonville	0.008979
0.414696	Jersey.City	0.024263
0.384636	Johnstown	0.005797
0.388917	Kansas.City	0.001516
0.386833	Knoxville	0.003600
0.385303	Lancaster	0.005130
0.393678	Lansing	0.003245
0.398434	Little.Rock	0.008001
0.363498	Los.Angeles	0.026935
0.392997	Louisville	0.002564
0.390342	Memphis	0.000091
0.391211	Miami	0.000778
0.397850	Milwaukee	0.007417
0.388047	Minneapolis	0.002386
0.389298	Mobile	0.001135
0.394084	Nashville	0.003651
0.390096	New.Haven	0.000337
0.390028	New.Orleans	0.000405

Table 3.5: Replicates by City *(continued)*

replicates	point_labels	differences
0.366749	New.York	0.023684
0.390052	Newark	0.000381
0.390749	Norfolk	0.000316
0.389536	Oklahoma.City	0.000897
0.390495	Omaha	0.000062
0.389859	Orlando	0.000574
0.402565	Paterson	0.012132
0.387620	peoria	0.002813
0.422909	philadelphia	0.032476
0.395079	phoenix	0.004646
0.402749	pittsburgh	0.012316
0.392039	Portland	0.001606
0.389889	Providence	0.000544
0.385448	Reading	0.004985
0.393773	Richmond	0.003340
0.392690	Rochester	0.002257
0.395803	Sacramento	0.005370
0.387413	St.Louis	0.003020
0.391818	Salt.Lake	0.001385
0.392384	San.Antonio	0.001951
0.391311	San.Bernardino	0.000878
0.391990	San.Diego	0.001557
0.370359	San.Francisco	0.020074
0.389812	San.Jose	0.000621
0.388586	Seattle	0.001847
0.386689	Shreveport	0.003744
0.386995	South.Bend	0.003438
0.387492	Spokane	0.002941
0.389643	Springfield	0.000790

Table 3.5: Replicates by City *(continued)*

replicates	point_labels	differences
0.389551	Syracuse	0.000882
0.392189	Tacoma	0.001756
0.390257	Tampa	0.000176
0.390366	Toledo	0.000067
0.392121	Tulsa	0.001688
0.385438	Utica	0.004995
0.379585	Washington	0.010848
0.389880	Wichita	0.000553
0.386342	Wilkes	0.004091
0.389240	Wilmington	0.001193
0.391259	Worcester	0.000826
0.385078	York	0.005355
0.389671	Youngstown	0.000762

From which we see how natural it is to make an observation's influence a function of the absolute difference between the distance correlation of the dataset with and without that observation. And it does not take much insight to realize that, as $n \to \infty$, and regardless of the dataset, the distribution of $R - R_i\hat{}\{$And if we take the absolute value its distribution will be the folded normal distribution$\}$ will be normally distributed, which allows us easily enough to find out which observations are *improbable*, i.e, *influential*. We may check the distribution:

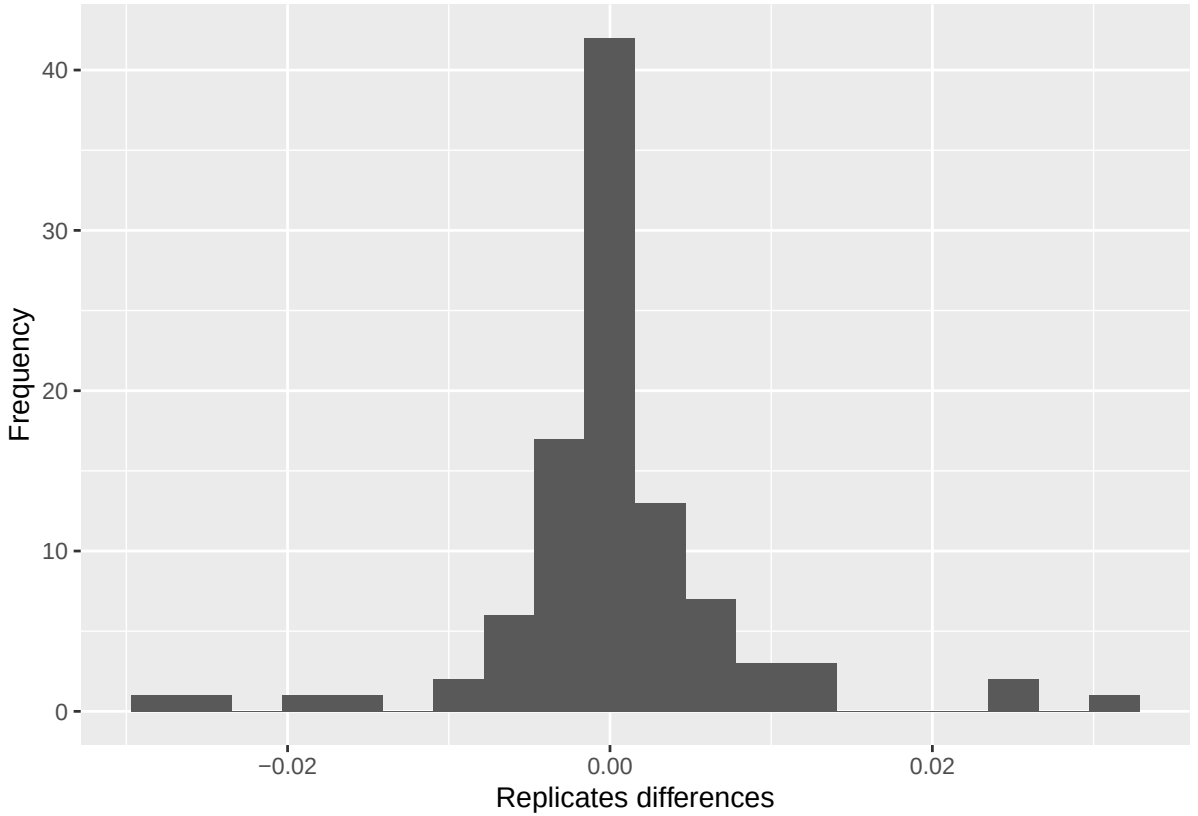

Figure 3.8: Replicate Differences of the Distance Correlation

Moreover, we can draw another very useful plot in such way: Let the x-axis represent the indices of the observations, and let the y-axis represent $R - R_i$ of observation number i. In the case of the crime dataset, the data points will be gather around the straight line $y = 0.390433x$; so the farther a point is from this straight line the more influential it is.

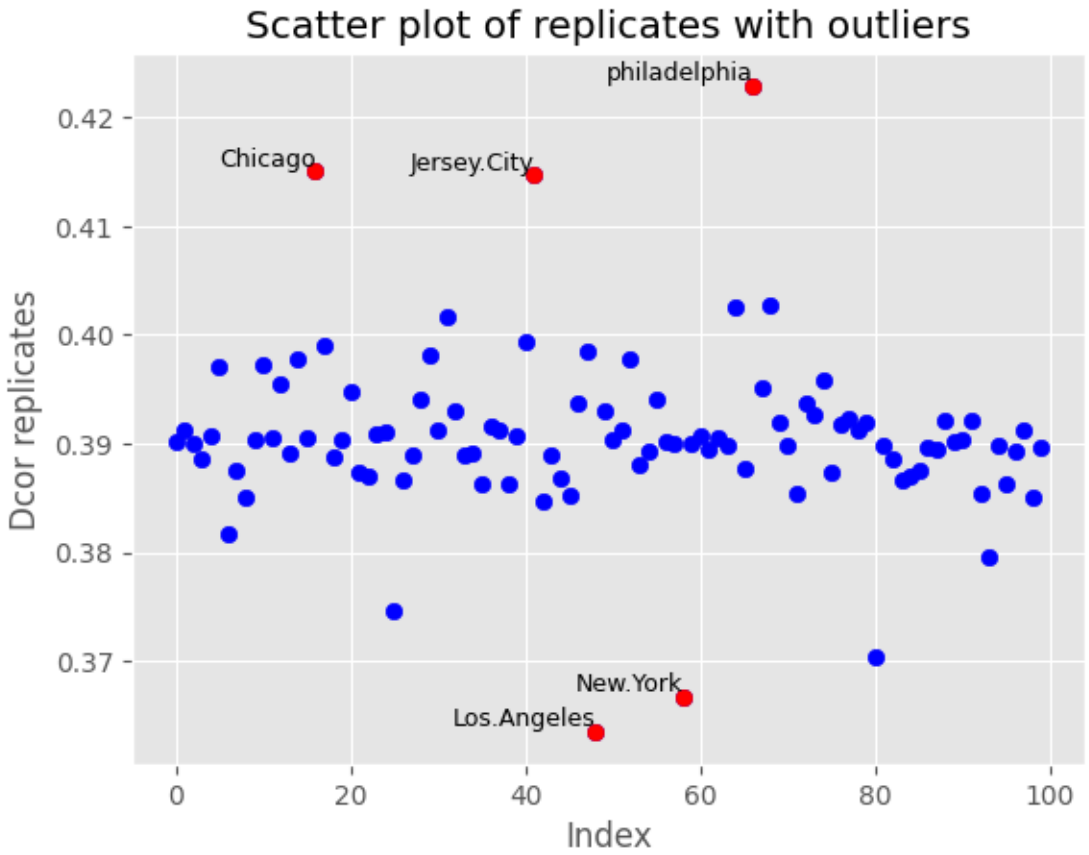

Figure 3.9: A plot of dcor replicates with the outliers in red

And, indeed, we see some quite clearly influential observations, such as Philadelphia and Los Angeles. What we ought to do with these observation is not for the method to tell us, but for us to decide after a through data analysis.

3.8 Distance Component Analysis

PCA (Principal Component Analysis) is a very popular method of dimensionality reduction. Given a set of variables $X \subset R^n$ it assumes that X is well approximated by lower

dimensional subspace of dimension k such that $k << n$. It has multiple uses:

- **Exploratory Data Analysis.** Too many dimensions and one's eyes stray and blur. Finding patterns in data with a small number of features is far easier than finding patterns in data with a high number.
- **Visualizations.** When $n > 3$ it's impossible to plot visualizations of the data in its totality. One must take subsets, slice and cleave and stare at many plots instead of just one.
- **Curse of Dimensionality.** For many machine learning methods, too many features are a plague. A method faced with a large number of features will be like a consumer faced with too many products; his saliva will overflow, but he will be unable to decide which product to choose.

So the usefulness of PCA is well understood. Yet the same predicate is satisfied for its limitations. For usually in the dataset we have alongside X another random variable Y which we call the target variable, and in most cases all that concerns is the relation between X and Y not the explanatory variables X themselves.

Therefore we might use PCA on X to get a smaller number of variables $\tilde{X}$ that for a certain sense approximate X– But what use is that? Consider the case when X offers no explanation for Y, then what we want wouldn't be $\tilde{X}$, but rather *nothing*.

True, we want a smaller set of variables and PCA gives us that, but also we would like a smaller set of variable that is related to Y. This is an additional constraint which PCA pays no heed. DCA (Distance Covariance Analysis), in contrast, is built on that constraint.

What DCA does is that instead of finding a smaller dimensional subspace that best approximates X, it tries to find a smaller dimensional subspace such that the projection of X onto it, suppose we call them X_P maximize $dcor(X_P, Y)$. This is a straightforward optimization problem. All the benefits of PCA are retained– with the additional gain of being assured that there is a strong relation of dependency between the projected data and the target variable.

Example 3.9. Let's compare between DCA and PCA with an example. Consider the following data generating process:

$$
\begin{aligned}
x_1 &\sim \mathcal{N}(0, 1) \\
x_2 &\sim \mathcal{U}(-10, 10) \\
x_3 &\sim \text{Pareto}(10) \\
x_4 &= x_1^2 \\
z &\sim \text{Bernoulli}(0.5) \\
\varepsilon &\sim \mathcal{N}(0, \sigma^2) \\
y &= x_1 + z \cdot x_2 + x_3 - x_4^2 + \varepsilon
\end{aligned}
$$

Using PCA we can decompose the data set to a single variable as follows:

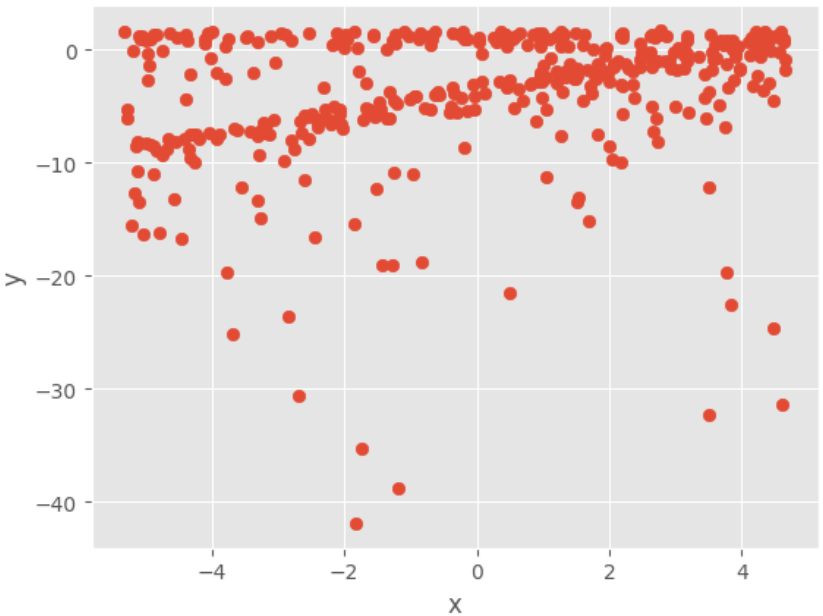

Figure 3.10: The PCA derived random variable

while with DCA we get

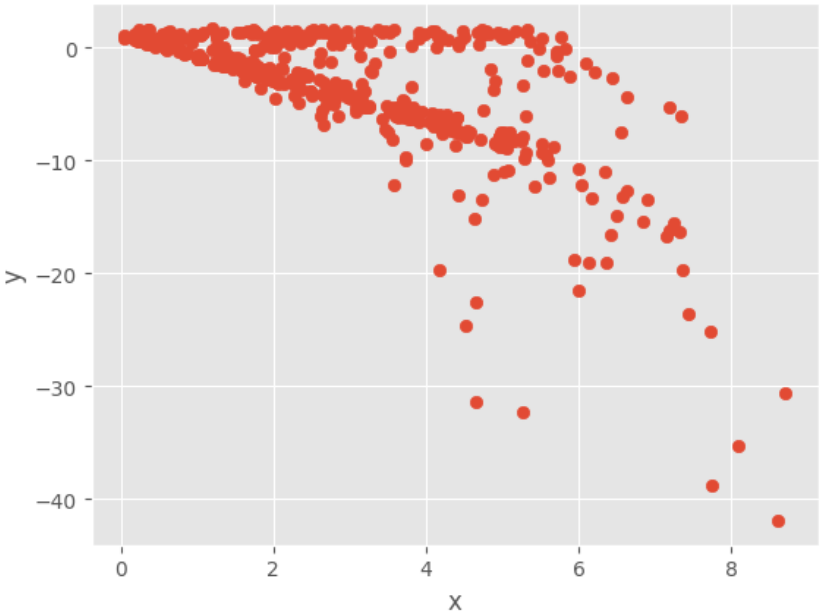

Figure 3.11: The DCA derived random variable

3.9 Independent Component Analysis

Independent Component Analysis(ICA) attempts to decompose a set of random variable into a set of independent random variables, such that the first set is a linear function of the second, in contrast to PCA, which merely attempts to find random variables that are *uncorrelated*.

Given a data matrix matrix X, we wish to find a weights matrix W, such that the columns of

$S = WX$

are independent.

This is a very simple optimization problem. We need to find a W that that minimizes

$$|S - X| = |WX - X|$$

While also minimizing

$$dcor(S_1, S_2, \cdots, S_k)$$

Having computed such a weight matrix, we can either use the whole of W, or a mere subset of its columns $\hat{W}$. Usually we want the whole of W; because the most-widespread use of the ICA algorithm is to simulate the *Cocktail party effect*. Consider being in a crowded area where multiple people are talking, yelling, and poking holes in each other's ears. *A priori*, we would expect a single person in the midst of this mess to be unable to differentiate one voice from another. But that is not the case; a person, by concentrating, can find the lone needle in the haystack.

How does the mind do that? By finding independent components.

Chapter 4

Implementations

This section contains our implementations as coded in *estats*.

4.1 Distance Correlation

```
import numpy as np
from scipy.spatial import distance_matrix

def double_center_matrix(m: np.array) -> np.array:
    ###
    Given a matrix m, returns the doubly centered matrix k, such that,
    k_ij = m - mean(the ith row of m) - mean(the jth column of m) +
       total mean.

    ###
    row_means = np.mean(m, 1, keepdims=True)
    column_means = np.mean(m, 0, keepdims=True)
    total_mean = np.mean(row_means)
    return m - row_means - column_means + total_mean


def dcov(x: np.array, y : np.array) -> np.float64:
    ###
    Returns the distance covariance, which is an always positive number
```

```
    representing the degree of dependency of two random variables.
    ###
    if np.ndim(x) == 1:
        x = np.expand_dims(x, 1)
    if np.ndim(y) == 1:
        y = np.expand_dims(y, 1)

    a = distance_matrix(x, x)
    b = distance_matrix(y, y)
    a_centered, b_centered = double_center_matrix(a),
       double_center_matrix(b)

    return np.sqrt(np.mean(a_centered * b_centered))

def dcor(x: np.array, y: np.array) -> np.float64:
    ###
    Returns an estimation of the distance correlation, R, which has
       the following two properties:
    1- 0 <= R <= 1.
    2- R = 0 iff x and y are independent.
    ###
    var_x = dcov(x, x)
    var_y = dcov(y, y)

    if np.isclose(var_x * var_y, 0):
        return 0

    return dcov(x, y) / np.sqrt(var_x * var_y)
```

4.2 BTree

```
from numba import njit
import numpy as np
import math


@njit
def powers2(L):
    pwr2 = np.zeros(L, dtype=np.int64)
    pwr2[0] = 2
```

```
    for k in range(1, L):
        pwr2[k] = pwr2[k-1] * 2
    return pwr2

@njit
def p2sum(pwr2):
    L = len(pwr2)
    psum = np.zeros(L, dtype=np.int64)
    psum.fill(pwr2[L-1])
    for i in range(1, L):
        psum[i] = psum[i-1] + pwr2[L-i-1]
    return psum

@njit
def container_nodes(y, pwr2, psum):
    L = len(pwr2)
    nodes = np.zeros(L, dtype=np.int64)

    nodes[0] = y
    for i in range(L-1):
        nodes[i+1] = math.ceil(y / pwr2[i]) + psum[i]

    return nodes

@njit
def sub_nodes(y, pwr2, psum):
    L = len(psum)
    nodes = np.full(L, -1, dtype=np.int64)

    k = y
    for level in range(L-1, 0, -1):
        p2 = pwr2[level-1]
        if k >= p2:
            idx = psum[level-1] + (y // p2)
            nodes[L-level-1] = idx
            k -= p2

    if k > 0:
        nodes[L-1] = y

    return nodes
```

```
@njit
def btree_sum(y, z):
    n = len(y)
    L = math.ceil(math.log2(n))

    pwr2 = powers2(L)
    psum = p2sum(pwr2)

    sums = np.zeros(2 * pwr2[L-1], dtype=np.float64)
    gamma1 = np.zeros(n, dtype=np.float64)

    for i in range(1, n):
        nodes = container_nodes(y[i-1], pwr2, psum)
        for p in range(L):
            sums[nodes[p]] += z[i-1]

        nodes = sub_nodes(y[i]-1, pwr2, psum)
        for p in range(L):
            node = nodes[p]
            if node > 0:
                gamma1[i] += sums[node]

    return gamma1
```

4.3 Computationally Efficient Distance Correlation

```
def fast_dcor(x, y):

    if x.ndim > 1 or y.ndim > 1:
        if np.ndim(x) == 1:
            x = np.expand_dims(x, 1)
        if np.ndim(y) == 1:
            y = np.expand_dims(y, 1)

        variance_x = multi_fast_dcov(x, x)
        variance_y = multi_fast_dcov(y, y)
        dcov_x_y = multi_fast_dcov(x, y)

        return dcov_x_y / np.sqrt(variance_x * variance_y)
```

```
    else:
        variances = fast_dcov(x, y)
        dcor = variances['dcov'] / np.sqrt(variances['dvar_x'] *
           variances['dvar_y'])
        return dcor

def multi_fast_dcov(x, y):

    if isinstance(x, pd.DataFrame):
        x = x.to_numpy()

    if isinstance(y, pd.DataFrame):
        y = y.to_numpy()

    p = x.shape[1]
    q = y.shape[1]
    k = 6*p*q

    R = np.random.randn(k, p)
    R /= np.linalg.norm(R, axis=1, keepdims=True)
    Q = np.random.randn(k, q)
    Q /= np.linalg.norm(Q, axis=1, keepdims=True)

    P_x = x @ R.T
    P_y = y @ Q.T

    u = np.zeros(k)

    for i in range(0, k):
        u[i] = fast_dcov(P_x[:, i], P_y[:, i])['dcov']

    C_p = np.sqrt(np.pi) * np.exp(np.log(gamma((p+1) / 2)) -
       np.log(gamma(p / 2)))
    C_q = np.sqrt(np.pi) * np.exp(np.log(gamma((q+1) / 2)) -
       np.log(gamma(q / 2)))

    return C_p * C_q * np.mean(u)



def fast_dcov(x, y):
```

```
    n = len(x)
    d1 = n**2
    d2 = n**3
    d3 = n**4

    sums = dcov_sums(x, y)

    dcov = np.sqrt(sums['s_1'] / d1 - 2 * sums['s_2'] / d2 +
       sums['s_3'] / d3)
    dvar_x = np.sqrt(sums['s_1a'] / d1 - 2 * sums['s_2a'] / d2 +
       sums['s_3a'] / d3)
    dvar_y = np.sqrt(sums['s_1b'] / d1 - 2 * sums['s_2b'] / d2 +
       sums['s_3b'] / d3)

    return {'dcov': dcov, 'dvar_x': dvar_x, 'dvar_y': dvar_y}

def dcov_sums(x: np.array, y: np.array) -> np.float64:

    n = len(x)
    rs_x = rank_sort(x)
    rs_y = rank_sort(y)

    a = row_sums(x, rs_x)
    b = row_sums(y, rs_y)

    a_b_sum = np.sum(a* b)
    a_sum = np.sum(a)
    b_sum = np.sum(b)
    ab_term = a_sum * b_sum

    x_sorted = rs_x['sorted']
    y_x = y[rs_x['sorted_indices']]
    rs_y_x = rank_sort(y_x)
    ones = np.ones(n)

    g_1 = partial_sum(x_sorted, y_x, ones, rs_x, rs_y_x)
    g_x = partial_sum(x_sorted, y_x, x_sorted, rs_x, rs_y_x)
    g_y = partial_sum(x_sorted, y_x, y_x, rs_x, rs_y_x)
    g_xy = partial_sum(x_sorted, y_x, x_sorted * y_x, rs_x, rs_y_x)
    s = np.sum(x * y * g_1 + g_xy - x * g_y - y * g_x)

    l = {
```

```
        's_1': s,
        's_2': a_b_sum,
        's_3': ab_term,
        's_1a': 2 * n * (n-1) * np.var(x, ddof=1),
        's_1b': 2 * n * (n-1) * np.var(y, ddof=1),
        's_2a': np.sum(a**2),
        's_2b': np.sum(b**2),
        's_3a': a_sum**2,
        's_3b': b_sum**2,
        'row_sums_a': a,
        'row_sums_b': b,
        'sum_a': a_sum,
        'sum_b': b_sum

    }

    return l

def partial_sum(x: np.array, y: np.array, z: np.array, sr_x, sr_y):
    rank_x = sr_x['ranks']
    order_y = sr_y['sorted_indices']
    rank_y = sr_y['ranks']

    cum_sum_y = (np.cumsum(z[order_y]) - z[order_y])[rank_y]
    cum_sum_x = np.cumsum(z) - z


    gamma = btree_sum(rank_y + 1, z)
    g = sum(z) - z - 2 * cum_sum_y - 2 * cum_sum_x + 4 * gamma

    return g[rank_x]

def row_sums(x, rs_x):

    x_sorted = rs_x['sorted']
    ranks = rs_x['ranks']
    n = len(x)


    x_sum = np.sum(x)
    cum_sum = (np.cumsum(x_sorted) - x_sorted)[ranks]
```

```
    row_sum = x_sum + (2 * ranks - n) * x - 2 * cum_sum

    return row_sum

def rank_sort(x):
    sorted_indices = np.argsort(x)
    x_sorted = x[sorted_indices]

    r = np.empty_like(sorted_indices)
    r[sorted_indices] = np.arange(len(x))

    return {'sorted': x_sorted, 'sorted_indices': sorted_indices,
       'ranks': r}
```

4.4 Normality Test

```
import numpy as np
from scipy.stats import zscore, norm
from scipy.special import hyp1f1, gamma
from scipy.spatial import distance_matrix

def normality_statistic(x: np.array) -> np.float64:
    ###
    Returns the statistic for the univariate normality test
    ###
    x = np.array(x)
    n = len(x)

    y = np.sort(zscore(x, ddof=1))
    k = np.arange(1 - n, n, 2)

    statistic = 2 * (np.sum(2 * y * norm.cdf(y) + 2 * norm.pdf(y))
    - n/np.sqrt(np.pi) - np.mean(k * y))

    return statistic

def m_normality_statistic(x: np.array) -> np.float64:
    ###
    Returns the statistic for the multivariate normality test
    ###
    n = len(x)
```

```
    y = zscore(x)
    d = x.shape[1]
    neg_norms = - (np.linalg.norm(y, axis=1)**2 / 2)
    k = gamma((d+1) / 2) / gamma(d/2)

    first_term = np.sqrt(2) * k * (2/n) * np.sum(hyp1f1(-1/2, d/2,
       neg_norms))
    second_term = 2 * k +  np.sum(distance_matrix(y, y)) / n**2

    return n * (first_term - second_term)



def normality_etest(x: np.array, iterations: int = 1000, statistic =
   normality_statistic) -> dict:
    ###
    Given a data matrix x which is n * m, tests whether the  m random
       variable are normal
    and returns the test statistic and p-value in a dictionary
    ###
    statistic = normality_statistic
    if np.ndim(x) > 1:
        statistic = m_normality_statistic
    observed_statistic = statistic(x)
    simulations = []

    for i in range(iterations):
        sim_x = norm.rvs(size = x.shape)
        simulations.append(statistic(sim_x))

    p_value = np.mean(simulations > observed_statistic)
    return {'observed_statistic': observed_statistic, 'p_value':
       p_value}
```

4.5 Bernoulli Distribution Test

```
import numpy as np

def compute_statistic_T(n, h, p_bar):
    ###
    Computes the statistic T for the Bernoulli goodness-of-fit test.
```

```
    Parameters:
        n (int): Total number of samples.
        h (int): Number of successes (1s) in the sample.
        p_bar (float): Estimate for the success probability p.
    Returns:
        float: The test statistic T.
    ###
    term_1 = (2 / n) * (h * (1 - p_bar) + (n - h) * p_bar)
    term_2 = -2 * p_bar * (1 - p_bar)
    term_3 = -2 * h * (n - h) / n**2
    T = term_1 + term_2 + term_3
    return T

def energy_goodness_of_fit_bernoulli(sample=None, n=None,
h=None, p_bar=None, R=1000):
    ###
    Conducts the energy goodness-of-fit test for a Bernoulli
       distribution.
    Parameters:
        sample (list or np.ndarray, optional): A sample of 0s and 1s.
        n (int, optional): Total number of samples.
        h (int, optional): Number of successes (1s) in the sample.
        p_bar (float, optional): Estimate for the success probability p.
        R (int): Number of resampling permutations.
    Returns:
        dict: Test statistic and p-value.
    ###

    # If sample is provided, calculate n, h, and p_bar from it
    if sample is not None:
        n = len(sample)
        if n == 0:
            raise ValueError("Sample␣is␣empty.␣Cannot␣compute␣
               goodness-of-fit␣test.")

        h = np.sum(sample)
        p_bar = h / n
    elif n is None or h is None or p_bar is None:
        raise ValueError("Provide␣either␣'sample'␣or␣'n',␣'h',␣and␣
           'p_bar'.")

    # Compute the observed statistic T
```

```
    observed_statistic = compute_statistic_T(n, h, p_bar)

    # Generate R simulated samples from Bernoulli(p_bar) and compute T
       for each
    simulated_statistics = []
    for _ in range(R):
        simulated_sample = np.random.binomial(1, p_bar, n)
        h_simulated = np.sum(simulated_sample)
        simulated_statistics.append(compute_statistic_T(n, h_simulated,
           p_bar))

    # Compute p-value as proportion of simulated T's >= observed T
    p_value = np.mean([sim_stat >= observed_statistic for sim_stat in
       simulated_statistics])

    return {"T": observed_statistic, "p-value": p_value}
```

4.6 Geometric Distribution Test

```
import numpy as np
from scipy.special import gammaincc

def geometric_expected_distance(p, k):
    ###
    Calculate the expected distance E|k - X| for a geometric
       distribution with parameter p.
    Parameters:
        p (float): Probability of success in the geometric distribution.
        k (int): Value for which to compute the expected distance.
    Returns:
        float: Expected distance E|k - X|.
    ###
    q = 1 - p
    theta = (1 - p) / p
    F_k_minus_1 = 1 - q**k  # CDF of Geometric at (k - 1) for
       Geometric(p)

    return k + theta - 2 * theta * F_k_minus_1

def expected_pairwise_distance(p):
```

```
    ###
    Calculate E|X - X| for two independent geometric random variables
       with parameter p.
    Parameters:
        p (float): Probability of success in the geometric distribution.
    Returns:
        float: Expected distance E|X - X|.
    ###
    q = 1 - p
    return 2 * q / (1 - q**2)

def geometric_energy_test(sample, p, R=1000):
    ###
    Perform the energy goodness-of-fit test for a geometric
       distribution.
    Parameters:
        sample (np.ndarray): Observed sample from the distribution.
        p (float): Hypothesized parameter of the geometric distribution.
        R (int): Number of permutations for resampling.
    Returns:
        float: Test statistic for the geometric distribution.
        float: p-value from permutation test.
    ###
    n = len(sample)
    observed_distance_sum = np.mean([np.abs(k - np.mean(sample)) for k
       in sample])
    expected_distance = geometric_expected_distance(p, np.mean(sample))
    intra_sample_distance = np.mean([np.abs(xi - xj) for xi in sample
       for xj in sample]) / n

    test_stat = n * (2 * observed_distance_sum - expected_distance -
       intra_sample_distance)

    perm_stats = []
    for _ in range(R):
        perm_sample = np.random.geometric(p, size=n)
        perm_stats.append(n * (2 * np.mean(np.abs(perm_sample -
           np.mean(perm_sample))) -
                               expected_distance -
                               np.mean([np.abs(xi - xj) for xi in
                                  perm_sample for xj in perm_sample]) /
                                  n))
```

```
    p_value = np.mean([stat >= test_stat for stat in perm_stats])

    return test_stat, p_value

def longest_run(sequence):
    ###
    Calculate the longest run of consecutive identical values (0s or
       1s) in a 0-1 sequence.
    Parameters:
        sequence (list or np.ndarray): Binary sequence of 0s and 1s.
    Returns:
        int: Length of the longest run.
    ###
    if len(sequence) == 0:
        return 0  # Return 0 for an empty sequence

    max_run = current_run = 1
    for i in range(1, len(sequence)):
        if sequence[i] == sequence[i - 1]:
            current_run += 1
            max_run = max(max_run, current_run)
        else:
            current_run = 1
    return max_run

def test_iid_sequence(sequence, p, R=1000):
    ###
    Test if a 0-1 sequence resembles an IID sequence with Bernoulli(p)
       trials.
    Parameters:
        sequence (list or np.ndarray): Binary sequence of 0s and 1s.
        p (float): Hypothesized success probability for Bernoulli
           trials.
        R (int): Number of simulations for permutation testing.
    Returns:
        float: Longest run in observed sequence.
        float: Expected longest run in a random sequence of similar
           length.
        float: p-value for independence test.
    ###
    n = len(sequence)
```

```
    if n == 0:
        raise ValueError("The␣input␣sequence␣is␣empty.␣Cannot␣calculate␣
            longest␣run.")

    observed_longest_run = longest_run(sequence)
    expected_longest_run = np.log2(n)

    perm_longest_runs = []
    for _ in range(R):
        perm_sequence = np.random.binomial(1, p, size=n)
        perm_longest_runs.append(longest_run(perm_sequence))

    p_value = np.mean([run >= observed_longest_run for run in
        perm_longest_runs])

    return observed_longest_run, expected_longest_run, p_value
```

4.7 Poisson Distribution Test

```
import numpy as np
from scipy.special import gammaincc, i0, i1

def poisson_expected_distance(lambda_val):
    ###
    Calculate the expected distance E|X - X| for a Poisson distributed
        X with mean lambda.
    Uses equation E|X - X| = 2lambdaexp(-2lambda) ((2lambda) +
        (2lambda)),
    where and are modified Bessel functions of the first kind.
    Parameters:
        lambda_val (float): Mean of the Poisson distribution.
    Returns:
        float: The expected distance E|X - X|.
    ###
    return 2 * lambda_val * np.exp(-2 * lambda_val) * (i0(2 *
        lambda_val) + i1(2 * lambda_val))

def energy_test_statistic(sample, lambda_val):
    ###
    Compute the energy test statistic for a sample from a Poisson
        distribution.
```

```
    Parameters:
        sample (np.ndarray): Array of observed counts.
        lambda_val (float): Hypothesized mean of the Poisson
           distribution.
    Returns:
        float: The energy test statistic value.
    ###
    n = len(sample)
    observed_distance_sum = np.mean([np.abs(x - lambda_val) for x in
       sample])
    expected_distance = poisson_expected_distance(lambda_val)
    intra_sample_distance = np.mean([np.abs(xi - xj) for xi in sample
       for xj in sample]) / n

    return n * (2 * observed_distance_sum - expected_distance -
       intra_sample_distance)

def m_test_statistic(sample, lambda_val):
    ###
    Compute the M-test statistic based on mean distances for a sample
       from a Poisson distribution.
    Parameters:
        sample (np.ndarray): Array of observed counts.
        lambda_val (float): Hypothesized mean of the Poisson
           distribution.
    Returns:
        float: The M-test statistic value.
    ###
    # Handle edge case for lambda_val = 0
    if lambda_val == 0:
        return 0.0  # Return 0 since all values should theoretically
           match lambda = 0 in this case

    k_vals = np.arange(max(sample) + 1)
    mean_distances = np.array([np.mean([np.abs(k - xi) for xi in
       sample]) for k in k_vals])
    expected_mean_distances = 2 * k_vals * gammaincc(k_vals,
       lambda_val) + lambda_val - k_vals - 2 * k_vals *
       np.exp(-lambda_val)

    return np.sum((mean_distances - expected_mean_distances) ** 2)
```

```
def poisson_e_test(sample, lambda_val, R=1000):
    ###
    Perform the Poisson E-test (energy test) with permutation
       resampling.
    Parameters:
        sample (np.ndarray): Array of observed counts.
        lambda_val (float): Hypothesized mean of the Poisson
           distribution.
        R (int): Number of resampling permutations.
    Returns:
        float: E-test statistic.
        float: p-value from the permutation test.
    ###
    observed_stat = energy_test_statistic(sample, lambda_val)
    permuted_stats = []

    for _ in range(R):
        perm_sample = np.random.poisson(lambda_val, size=len(sample))
        permuted_stats.append(energy_test_statistic(perm_sample,
           lambda_val))

    p_value = np.mean([stat >= observed_stat for stat in
       permuted_stats])
    return observed_stat, p_value

def poisson_m_test(sample, lambda_val):
    ###
    Perform the Poisson M-test based on mean distances.
    Parameters:
        sample (np.ndarray): Array of observed counts.
        lambda_val (float): Hypothesized mean of the Poisson
           distribution.
    Returns:
        float: M-test statistic.
    ###
    return m_test_statistic(sample, lambda_val)
```

4.8 Two Sample Test

```
import numpy as np
```

```
def energy_distance(x, y):
    ###
    Calculate the sample energy distance between two samples x and y.
    Parameters:
        x (np.ndarray): Sample from distribution X.
        y (np.ndarray): Sample from distribution Y.
    Returns:
        float: The calculated energy distance.
    ###
    n, m = len(x), len(y)
    A = np.sum([np.abs(xi - yj) for xi in x for yj in y]) / (n * m)
    B = np.sum([np.abs(xi - xj) for xi in x for xj in x]) / (n ** 2)
    C = np.sum([np.abs(yi - yj) for yi in y for yj in y]) / (m ** 2)

    # Calculate and return the energy distance statistic
    return 2 * A - B - C

def two_sample_energy_test(x, y, R = 1000):
    ###
    Perform a two-sample energy test to determine if two samples come
       from the same distribution.
    Parameters:
        x (np.ndarray): Sample from distribution X.
        y (np.ndarray): Sample from distribution Y.
        R (int): Number of permutations to generate the null
           distribution.
    Returns:
        float: Test statistic value.
        float: p-value from the permutation test.
    ###
    n, m = len(x), len(y)
    combined = np.concatenate([x, y])
    observed_stat = (n * m / (n + m)) * energy_distance(x, y)

    # Permutation test
    permuted_stats = []
    for _ in range(R):
        np.random.shuffle(combined)
        perm_x = combined[:n]
        perm_y = combined[n:]
        permuted_stat = (n * m / (n + m)) * energy_distance(perm_x,
           perm_y)
```

```
        permuted_stats.append(permuted_stat)

    p_value = np.mean([stat >= observed_stat for stat in
       permuted_stats])

    return observed_stat, p_value
```

4.9 K-Groups Clustering

```
import numpy as np
from scipy.spatial.distance import cdist

class KGroupsClustering:
    def __init__(self, n_clusters=3, alpha=1, max_iter=100,
       tolerance=1e-4):
        ###
        Initializes the K-groups clustering model.
        Parameters:
            n_clusters (int): Number of clusters.
            alpha (float): Exponent for distance calculation.
            max_iter (int): Maximum number of iterations.
            tolerance (float): Threshold for stopping criterion.
        ###
        self.n_clusters = n_clusters
        self.alpha = alpha
        self.max_iter = max_iter
        self.tolerance = tolerance
        self.labels_ = None
        self.cluster_centers_ = None

    def _calculate_within_dispersion(self, X, labels):
        ###
        Calculates the within-cluster dispersion for given data and
           labels.
        Parameters:
            X (np.ndarray): Data points.
            labels (np.ndarray): Cluster labels for each point.
        Returns:
            float: Within-cluster dispersion.
        ###
```

```
        dispersion = 0
        for k in range(self.n_clusters):
            cluster_points = X[labels == k]
            if len(cluster_points) > 1:
                distances = cdist(cluster_points, cluster_points,
                    metric='minkowski', p=self.alpha)
                dispersion += np.sum(distances) / (2 *
                    len(cluster_points))
        return dispersion

    def fit(self, X):
        ###
        Fits the K-groups clustering model to the data.
        Parameters:
            X (np.ndarray): Data points.
        ###
        n_samples = X.shape[0]
        # Randomly assign points to clusters
        self.labels_ = np.random.randint(0, self.n_clusters, n_samples)
        dispersion_prev = float('inf')

        for iteration in range(self.max_iter):
            dispersion = self._calculate_within_dispersion(X,
                self.labels_)

            # Check for convergence
            if abs(dispersion_prev - dispersion) < self.tolerance:
                print(f"Converged at iteration {iteration}")
                break

            dispersion_prev = dispersion

            # Reassign points to minimize within-cluster dispersion
            for i in range(n_samples):
                best_label = self.labels_[i]
                best_dispersion = dispersion

                # Test moving point i to each cluster
                for k in range(self.n_clusters):
                    if self.labels_[i] == k:
                        continue
```

```
                    old_label = self.labels_[i]
                    self.labels_[i] = k
                    new_dispersion =
                        self._calculate_within_dispersion(X,
                        self.labels_)

                    # Revert change if dispersion does not decrease
                    if new_dispersion < best_dispersion:
                        best_dispersion = new_dispersion
                        best_label = k
                    self.labels_[i] = old_label

                # Update label for point i
                self.labels_[i] = best_label

        # Calculate final cluster centers
        self.cluster_centers_ = np.array([X[self.labels_ ==
            k].mean(axis=0) for k in range(self.n_clusters)])

    def predict(self, X):
        ###
        Predicts cluster labels for new data points.
        Parameters:
            X (np.ndarray): Data points.
        Returns:
            np.ndarray: Cluster labels.
        ###
        distances = np.array([
            np.sum(cdist(X, self.cluster_centers_[k].reshape(1, -1),
                metric='minkowski', p=self.alpha), axis=1)
            for k in range(self.n_clusters)
        ])
        return np.argmin(distances, axis=0)
```

4.10 Disco

```
import numpy as np
from scipy.spatial.distance import pdist, squareform

def within_sample_dispersion(X, alpha=1):
```

```
    ###
    Compute within-sample dispersion (W_alpha) for a single sample.
    Parameters:
        X (np.ndarray): Array of sample data.
        alpha (float): Exponent for distance weighting, default is 1.
    Returns:
        float: The within-sample dispersion W_alpha.
    ###
    n = X.shape[0]
    if n < 2:
        return 0
    distances = pdist(X, 'euclidean')
    weighted_distances = distances**alpha
    return np.sum(weighted_distances) / (n * (n - 1))

def total_dispersion(X, alpha=1):
    ###
    Compute the total dispersion (T_alpha) for all samples pooled
       together.
    Parameters:
        X (np.ndarray): Pooled array of data from all samples.
        alpha (float): Exponent for distance weighting, default is 1.
    Returns:
        float: The total dispersion T_alpha.
    ###
    n = X.shape[0]
    distances = pdist(X, 'euclidean')
    weighted_distances = distances**alpha
    return np.sum(weighted_distances) / (n * (n - 1))

def disco_decomposition(groups, alpha=1, R=1000):
    ###
    Perform the DISCO decomposition to partition the total dispersion
       into
    within-sample and between-sample components.
    Parameters:
        groups (list of np.ndarray): List where each element is a
           sample array.
        alpha (float): Exponent for distance weighting, default is 1.
        R (int): Number of permutations for the between-sample
           component test.
    Returns:
```

```
    dict: Contains T_alpha, W_alpha, S_alpha, and p-value for the
       between-sample test.
###
# Compute within-sample dispersion
W_alpha = sum(within_sample_dispersion(X, alpha) * len(X) for X in
   groups) / sum(len(X) for X in groups)

# Combine all samples for total dispersion
pooled_data = np.vstack(groups)
T_alpha = total_dispersion(pooled_data, alpha)

# Compute between-sample component
n = sum(len(X) for X in groups)
S_alpha = T_alpha - W_alpha

# Permutation test for the between-sample component
observed_statistic = S_alpha
perm_stats = []
for _ in range(R):
    # Permute labels
    permuted_data = np.random.permutation(pooled_data)
    perm_groups = np.split(permuted_data, np.cumsum([len(X) for X
       in groups])[:-1])
    perm_W_alpha = sum(within_sample_dispersion(X, alpha) * len(X)
       for X in perm_groups) / n
    perm_T_alpha = total_dispersion(permuted_data, alpha)
    perm_stats.append(perm_T_alpha - perm_W_alpha)

# Calculate p-value
p_value = np.mean([stat >= observed_statistic for stat in
   perm_stats])

return {
    "T_alpha": T_alpha,
    "W_alpha": W_alpha,
    "S_alpha": S_alpha,
    "p-value": p_value
}
```

4.11 Outlier Detection

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
from estats.tests.cor import dcor


def find_outliers(data, target, threshold = 3, row_names = None,):
    numeric_data = data.to_numpy()
    n = len(numeric_data)
    dcor_replicates = []
    original_dcor = dcor(data, target)

    for i in range(0, n):
        sample_data = data.drop([i], axis=0)
        sample_target = target.drop([i], axis=0)
        dcor_replicates.append(dcor(sample_data, sample_target))

    dcor_replicates = pd.DataFrame({'replicates' : dcor_replicates,
       'point_labels': row_names})
    z = np.abs(stats.zscore(dcor_replicates['replicates']))
    outliers = dcor_replicates[z > threshold]


    return {'original_dcor': original_dcor, 'dcor_replicates':
       dcor_replicates, 'outliers' : outliers }
```

4.12 Distance Component Analysis

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from estats.tests.cor import fast_dcov
from scipy.optimize import minimize

def constraint(u):
    return 1 - np.linalg.norm(u)
```

```
def objective(u, x, y):
    u /= np.linalg.norm(u)
    return -fast_dcov(x @ u, y)['dcov']

def dca(x, y, k=2):
    m = x.shape[1]
    U = np.random.randn(m, k)
    U /= np.linalg.norm(U, axis=0)
    X_c = x

    for i in range(k):
        u_k = U[:, i]

        cons = {'type': 'ineq', 'fun': constraint}
        res = minimize(objective, x0=u_k, args=(X_c, y),
           constraints=cons)

        u = res.x / np.linalg.norm(res.x)
        U[:, i] = u
        U_k = U[:, :i+1]
        p = U_k @ U_k.T
        X_c = x - x @ p

    return {'projections': x @ U, 'axes': U,}
```

4.13 Independent Component Analysis

```
import numpy as np
import scipy.linalg as la
from energystats.tests.cor import multi_fast_dcov, dcov,
   _dcov_grad_x_vec as _dcov_grad

def dcov_single(X, fast = 1):
    fun = multi_fast_dcov if fast else dcov
    return fun(X[:,0,None], X[:,1:])

def grad_single(X):
    gx,gY = _dcov_grad(X[:,0], X[:,1:])
    G = np.zeros_like(X)
    G[:,0] = gx
```

```
    G[:,1:]= gY
    return G

def givens(t,d,p,q):
    G = np.eye(d); c,s=np.cos(t),np.sin(t)
    G[p,p] = G[q,q] =c
    G[p,q] = -s
    G[q,p] = s
    return G

def rotation(T):
    d = T.shape[0]; W = np.eye(d)
    for i in range(d-1):
        for j in range(i+1,d):
            W=W@givens(T[i,j],d,i,j)
    return W
def unpack_theta(v,d):
    T=np.zeros((d,d)); T[np.triu_indices(d,1)]=v; return T

def gradient_descent(grad, theta0, Z, tol=1e-5, max_iter=100):
    alpha = 0.7
    for i in range(max_iter):
        grad_val = grad(theta0, Z)
        theta_new = theta0 - alpha * grad_val
        if np.linalg.norm(theta_new - theta0) < tol:
            break
        theta0 = theta_new
        alpha *= 0.95
    return theta0

def obj(theta, Z):
    d=Z.shape[1]; S=Z@rotation(unpack_theta(theta,d)).T
    return sum(dcov_single(S[:,i:]) for i in range(d-1))

def grad_theta(theta, Z):
    d=Z.shape[1]; T=unpack_theta(theta,d)
    pairs=[(i,j) for i in range(d-1) for j in range(i+1,d)]
    prefix=[np.eye(d)]
    for (i,j) in pairs: prefix.append(prefix[-1]@givens(T[i,j],d,i,j))
    suffix=[np.eye(d)]
    for (i,j) in reversed(pairs):
       suffix.append(givens(T[i,j],d,i,j)@suffix[-1])
```

```
    suffix=list(reversed(suffix))
    W=prefix[-1]; S=Z@W.T
    GS=np.zeros_like(S)
    for i in range(d-1): GS[:,i:]+=grad_single(S[:,i:])
    GW=GS.T@Z
    g=[]
    for k,(i,j) in enumerate(pairs):
        A=prefix[k]; B=suffix[k+1]
        c,s=np.cos(T[i,j]),np.sin(T[i,j])
        dW=np.zeros((d,d));
           dW[np.ix_([i,j],[i,j])]=np.array([[-s,-c],[c,-s]])
        g.append(np.sum(GW*(A@dW@B)))
    return np.array(g)

def dCovICA(Y, starts=1000, **kwargs):
    mean=Y.mean(0);
       Z=(Y-mean)@la.sqrtm(np.linalg.inv(np.cov(Y-mean,rowvar=False))).real.T
    d=Z.shape[1]; p=d*(d-1)//2
    best=np.inf; best_theta=None
    print(Y.shape)
    for _ in range(starts):
        th0=np.random.uniform(-np.pi,np.pi,p)
        res=gradient_descent(grad_theta, th0, Z, tol =
           kwargs.get('tol', 1e-5), max_iter=kwargs.get('max_iter', 50))
        val_res = obj(res, Z)
        if val_res<best:
            best,res_theta=val_res,res
            best_theta=res_theta
    S = Z@rotation(unpack_theta(best_theta,d)).T
    return rotation(unpack_theta(best_theta,d)).T, S-S.mean(0)
```

Chapter 5

Appendix A: Datasets

5.1 Running Time of Dcor and Improved Dcor dataset

Table 5.1: Comparison of running times (in seconds) between standard and improved dcor implementations across different sample sizes.

num_samples	dcor_running_time	improved_dcor_running_time
1000	0.101952	0.001147
2000	0.423244	0.002076
3000	0.967069	0.003388
4000	1.840559	0.004363
5000	2.759299	0.005313
6000	4.145951	0.006325
7000	5.392993	0.006878
8000	7.422989	0.008128
9000	8.936315	0.008791
10000	11.634925	0.009995
11000	13.532491	0.011429
12000	17.095752	0.012249

Table 5.1: Comparison of running times (in seconds) between standard and improved dcor implementations across different sample sizes. *(continued)*

num_samples	dcor_running_time	improved_dcor_running_time
13000	19.004895	0.013713
14000	23.247329	0.014341
15000	24.810966	0.015204
16000	32.272147	0.016559
17000	31.366162	0.018634
18000	38.697574	0.018272
19000	40.300381	0.019829
20000	49.242613	0.020086
21000	49.708993	0.020721
22000	57.337229	0.023018
23000	56.277050	0.027726
24000	76.278003	0.026615

5.2 Crime Rate by American State Dataset

Table 5.2: A data set of American states with the crime rate and other explanatory variables

population	nonwhite	density	crime	city
675	7.3	746	2602	Akron
713	2.6	322	1388	Albany
534	0.8	491	1182	Allentown
1261	1.4	1612	3341	Anaheim
1330	22.8	770	2805	Atlanta
331	7.0	41	3306	Bakersfield
1981	21.6	877	4256	Baltimore

Table 5.2: A data set of American states with the crime rate and other explanatory variables *(continued)*

population	nonwhite	density	crime	city
315	20.7	240	2117	Beaumont
305	0.6	147	1063	Binghamton
739	32.1	272	2285	Birmingham
3239	3.4	1831	2835	Boston
785	5.3	1252	2630	Bridgeport
1324	6.8	832	2171	Buffalo
363	5.4	630	1764	Canton
388	24.2	328	3447	Charlotte
306	17.6	308	2643	Chattanooga
6815	14.8	1832	2680	Chicago
1376	10.4	640	1780	Cincinnati
2068	13.7	1361	3162	Cleveland
870	11.0	583	3037	Columbus
297	4.2	194	3077	Corpus.Christi
1459	15.0	320	3701	Dallas
366	2.1	215	1877	Davenport
836	9.7	489	2392	Dayton
1129	4.2	308	4095	Denver
4127	13.1	2114	4283	Detroit
271	0.8	37	1736	Duluth
351	3.3	332	2430	El.Paso
487	9.0	374	3164	Flint
526	16.6	431	3688	Fort.Lauderdale
680	10.7	423	2927	Fort.Worth
415	7.5	70	3863	Fresno
608	15.3	649	3310	Gary
514	3.4	362	2188	Grand.Rapids
592	19.9	268	1987	Greensboro

Table 5.2: A data set of American states with the crime rate and other explanatory variables *(continued)*

population	nonwhite	density	crime	city
398	6.3	245	1339	Harrisburg
802	5.4	1085	2390	Hartford
628	64.3	1053	3283	Honolulu
1867	19.8	297	3511	Houston
1062	10.8	345	2798	Indianapolis
513	23.4	670	4035	Jacksonville
615	6.9	13087	2301	Jersey.City
270	1.3	151	458	Johnstown
1231	10.9	445	3748	Kansas.City
399	7.5	281	1597	Knoxville
302	1.3	319	627	Lancaster
361	2.8	212	2991	Lansing
323	19.9	217	3420	Little.Rock
6860	9.7	1686	4852	Los.Angeles
802	11.6	883	3758	Louisville
770	38.0	565	2691	Memphis
1150	14.9	563	4460	Miami
1344	5.3	923	1929	Milwaukee
1677	1.8	796	3231	Minneapolis
382	30.8	135	2433	Mobile
536	18.4	329	3185	Nashville
726	7.7	1200	2726	New.Haven
1064	30.9	539	3467	New.Orleans
11551	12.0	5408	4732	New.York
1881	13.4	2683	3261	Newark
653	26.4	963	3160	Norfolk
605	9.4	283	2343	Oklahoma.City
525	6.0	342	2663	Omaha

Table 5.2: A data set of American states with the crime rate and other explanatory variables *(continued)*

population	nonwhite	density	crime	city
390	16.6	321	2560	Orlando
1353	3.8	3170	1922	Paterson
336	3.3	186	2125	peoria
4829	15.7	1359	1753	philadelphia
872	5.5	94	3962	phoenix
2387	6.8	783	2127	pittsburgh
957	3.0	262	3627	Portland
749	1.9	1221	2903	Providence
292	1.8	338	1052	Reading
515	26.3	430	3183	Richmond
853	4.0	368	1820	Rochester
765	6.8	222	3793	Sacramento
2327	14.2	565	3302	St.Louis
534	1.4	504	2977	Salt.Lake
850	7.1	434	3459	San.Antonio
1086	4.7	40	3743	San.Bernardino
1221	5.5	286	2587	San.Diego
2999	12.5	1210	5441	San.Francisco
985	3.2	758	3000	San.Jose
1340	4.8	317	4329	Seattle
299	34.1	171	1864	Shreveport
275	5.3	302	1973	South.Bend
273	2.0	155	2130	Spokane
554	3.1	481	2467	Springfield
625	2.6	259	1577	Syracuse
401	5.1	239	2843	Tacoma
924	11.5	709	3247	Tampa
678	7.3	446	2050	Toledo

Table 5.2: A data set of American states with the crime rate and other explanatory variables *(continued)*

population	nonwhite	density	crime	city
466	9.2	123	2840	Tulsa
350	1.6	132	869	Utica
2751	24.5	1170	4019	Washington
406	5.8	166	2532	Wichita
340	0.3	383	613	Wilkes
488	11.7	419	2393	Wilmington
612	0.9	405	2874	Worcester
316	2.0	220	1062	York
528	9.2	513	1698	Youngstown

Chapter 6

Appendix B: Proof of the O(nlogn) Algorithm for Dcor

One can verify the following equalities:

$$\sum_{i\neq j} a_{ij}b_i = \sum_{i=1}^{n} a_{i.}b_i, \sum_{i\neq j} a_{ij}b_{.j} = \sum_{j=1}^{n} a_{.j}b_{.j}, \sum_{i\neq j} b_{ij}a_i = \sum_{i=1}^{n} a_{i.}b_i, \sum_{i\neq j} b_{ij}a_{.j} = \sum_{j=1}^{n} a_{.j}b_{.j} \quad (A.1)$$

$$\sum_{i\neq j} a_i = (n-1)a_{..}, \quad \sum_{i\neq j} b_i = (n-1)b_{..}; \quad (A.2)$$

$$a_i = a_{.i}, \text{ and } b_i = b_{.i}. \quad (A.3)$$

The following will be used in our simplification too. We have

$$\begin{aligned}\sum_{i\neq j} a_{i}.b_{.j} &= \sum_{i=1}^{n} a_i \sum_{j:j\neq i} b_{.j} \\ &= \sum_{i=1}^{n} a_i \left(b_{..} - b_{.i}\right) \\ &= a_{..}b_{..} - \sum_{i=1}^{n} a_i\, b_{.i} \\ &\overset{(A.3)}{=} a_{..}b_{..} - \sum_{i=1}^{n} a_i\, b_i; \quad (A.4)\end{aligned}$$

Similarly, we have

$$\sum_{i\neq j} b_{i}.a_{.j} = a_{..}b_{..} - \left(\sum_{i=1}^{n} a_i.b_i\right). \quad (A.5)$$

In the following, we simplify the statistic. We have

$$\begin{aligned}\Omega_n &\overset{(3.2)}{=} \frac{1}{n(n-3)} \sum_{i\neq j} \widetilde{A}_{i,j}\widetilde{B}_{i,j} \\ &\overset{(3.1)}{=} \frac{1}{n(n-3)} \sum_{i\neq j} \left(a_{ij} - \frac{a_i}{n-2} - \frac{a_{.j}}{n-2} + \frac{a_{..}}{(n-1)(n-2)}\right) \\ &\times \left(b_{ij} - \frac{b_i}{n-2} - \frac{b_{.j}}{n-2} + \frac{b_{..}}{(n-1)(n-2)}\right) \\ &= \frac{1}{n(n-3)} \sum_{i\neq j} \left[a_{ij}b_{ij} - \frac{a_{ij}b_i + b_{ij}a_i}{n-2} - \frac{a_{ij}b_{.j} + b_{ij}a_{.j}}{n-2} + \frac{a_{ij}b_{..} + b_{ij}a_{..}}{(n-1)(n-2)}\right. \\ &\left. + \frac{a_ib_i + a_{.j}b_{.j}}{(n-2)^2} + \frac{(a_i + a_{.j})b_{..} + (b_i + b_{.j})a_{..}}{(n-1)(n-2)^2} - \frac{a_{..}b_{..}}{(n-1)^2(n-2)^2}\right].\end{aligned}$$

$$
\begin{aligned}
\Omega_n &\overset{(A.2)}{=} \frac{1}{n(n-3)}\sum_{i\neq j} a_{ij}b_{ij} \\
&\quad - \frac{1}{n(n-2)(n-3)}\sum_{i\neq j}[a_{ij}(b_{i.}+b_{.j})+b_{ij}(a_{i.}+a_{.j})] \\
&\quad + \frac{1}{n(n-2)^2(n-3)}\sum_{i\neq j}(a_{i.}+a_{.j})(b_{i.}+b_{.j}) \\
&\quad - \frac{a_{..}b_{..}}{(n-1)(n-2)^2(n-3)} \\
&\overset{(A.1),(A.3)}{=} \frac{1}{n(n-3)}\sum_{i\neq j} a_{ij}b_{ij} - \frac{4}{n(n-2)(n-3)}\sum_{i=1}^{n} a_{i.}b_{i.} \\
&\quad + \frac{1}{n(n-2)^2(n-3)}\sum_{i\neq j}(a_{i.}+a_{.j})(b_{i.}+b_{.j}) \\
&\quad - \frac{a_{..}b_{..}}{(n-1)(n-2)^2(n-3)}.
\end{aligned}
$$

Now bringing in (A.4) and (A.5), we have

$$
\begin{aligned}
\Omega_n &= \frac{1}{n(n-3)}\sum_{i\neq j} a_{ij}b_{ij} - \frac{4}{n(n-2)(n-3)}\sum_{i=1}^{n} a_{i.}b_{i.} \\
&\quad - \frac{a_{..}b_{..}}{(n-1)(n-2)^2(n-3)} \\
&\quad + \frac{1}{n(n-2)^2(n-3)}\left[2(n-1)\sum_{i=1}^{n} a_{i.}b_{i.} + 2\left(a_{..}b_{..} - \sum_{i=1}^{n} a_{i.}b_{i.}\right)\right] \\
&= \frac{1}{n(n-3)}\sum_{i\neq j} a_{ij}b_{ij} - \frac{2}{n(n-2)(n-3)}\sum_{i=1}^{n} a_{i.}b_{i.} \\
&\quad + \frac{a_{..}b_{..}}{n(n-1)(n-2)(n-3)},
\end{aligned}
$$

We use arithmetic induction. Suppose $n = r + 1$,

$$(r+1)U_{r+1,r}(x_1,\ldots,x_{r+1}) = \sum_{i=1}^{r+1} U^{-i}_{r+1,r}(x_1,\ldots,x_{r+1}).$$

By defining $h(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_{r+1}) = U^{-i}_{r+1,r}(x_1,\ldots,x_{r+1})$, we can verify that $h(\cdot)$ is a kernel function with r variables. Consequently, $U_{r+1,r}(x_1,\ldots,x_{r+1})$ can be written as.

Now suppose for any $n \geq n'$, $U_{n'}(x_1,\ldots,x_n)$ has the form as in with the function $h(\cdot)$ that was defined above. Applying (3.6) with $n = n'+1$, we can show that $U_{n'+1,r}(x_1,\ldots,x_{n'+1})$ still has the form with the same function $h(\cdot)$ that was defined above. We omit further details.

It is evident to verify that the followings are true: for $i \neq k$,

$$\begin{aligned}
a_i^{-k} &= a_i - a_{ik},\\
b_i^{-k} &= b_i - b_{ik},\\
a_{\cdot\cdot}^{-k} &= a_{\cdot\cdot} - a_{k\cdot} - a_{\cdot k} = a_{\cdot\cdot} - 2a_{\cdot k},\\
b_{\cdot\cdot}^{-k} &= b_{\cdot\cdot} - 2b_{\cdot k}.
\end{aligned}$$

We then have

$$\begin{aligned}
\Omega_{n-1}^{-k} = \frac{\sum_{i\neq j, i\neq k, j\neq k} a_{ij}b_{ij}}{(n-1)(n-4)} - \frac{2\sum_{i=1,i\neq k}^{n}(a_i - a_{ik})(b_i - b_{ik})}{(n-1)(n-3)(n-4)}\\
+ \frac{(a_{\cdot\cdot} - 2a_{\cdot k})(b_{\cdot\cdot} - 2b_{\cdot k})}{(n-1)(n-2)(n-3)(n-4)}.
\end{aligned}$$

We have the following:

$$
\begin{aligned}
\sum_{k=1}^{n} \Omega_{n-1}^{-k} &= \sum_{k=1}^{n} \frac{\sum_{i\neq j, i\neq k, j\neq k} a_{ij} b_{ij}}{(n-1)(n-4)} \\
&\quad - \sum_{k=1}^{n} \frac{2\sum_{i=1, i\neq k}^{n} (a_i - a_{ik})(b_i - b_{ik})}{(n-1)(n-3)(n-4)} \\
&\quad + \sum_{k=1}^{n} \frac{(a_{..} - 2a_{.k})(b_{..} - 2b_{.k})}{(n-1)(n-2)(n-3)(n-4)} \\
&= \frac{(n-2)\sum_{i\neq j} a_{ij} b_{ij}}{(n-1)(n-4)} \\
&\quad - \frac{2\left[(n-3)\sum_{i=1}^{n} a_{i.} b_{i.} + \sum_{i\neq k} a_{ik} b_{ik}\right]}{(n-1)(n-3)(n-4)} \\
&\quad + \frac{(n-4) a_{..} b_{..} + 4\sum_{k=1}^{n} a_{k.} b_{k.}}{(n-1)(n-2)(n-3)(n-4)} \\
&= \frac{\sum_{i\neq j} a_{ij} b_{ij}}{n-3} - \frac{2}{(n-2)(n-3)} \sum_{i=1}^{n} a_{i.} b_{i.} \\
&\quad + \frac{a_{..} b_{..}}{(n-1)(n-2)(n-3)}.
\end{aligned}
$$

We can verify that the above equates to $n \cdot \Omega_n$, which indicates that Ω_n is a U-statistic. We have

$$
\begin{aligned}
a_i &= \sum_{\ell=1}^{n} a_{i\ell} = \sum_{\ell=1}^{n} |x_i - x_\ell| \\
&= \sum_{x_\ell < x_i} (x_i - x_\ell) + \sum_{x_\ell > x_i} (x_\ell - x_i) \\
&= x_i \left(\sum_{x_\ell < x_i} 1 - \sum_{x_\ell > x_i} 1 \right) - \sum_{x_\ell < x_i} x_\ell + \sum_{x_\ell > x_i} x_\ell.
\end{aligned}
$$

It is easy to verify that

$$
\sum_{x_\ell > x_i} 1 = n - 1 - \alpha_i^x,
$$

and

$$\sum_{x_\ell > x_i} x_\ell = x_{\cdot\cdot} - x_i - \beta_i^x.$$

Taking into account the above two equations, we have

$$\begin{aligned} a_i &= (2\alpha_i^x - n + 1)x_i - \beta_i^x + x_{\cdot\cdot} - x_i - \beta_i^x \\ &= x_i + (2\alpha_i^x - n)x_i - 2\beta_i^x + x_{\cdot\cdot}, \end{aligned}$$

Without loss of generality (WLOG), we assume that $x_1 < x_2 < \cdots < x_n$. We have

$$\begin{aligned} \gamma_i(\{c_j\}) &= \sum_{j:j\neq i} c_j S_{ij} \\ &= \sum_{\substack{j:j>i \\ y_j > y_i}} c_j + \sum_{\substack{j:j<i \\ y_j < y_i}} c_j - \sum_{\substack{j:j>i \\ y_j < y_i}} c_j - \sum_{\substack{j:j<i \\ y_j > y_i}} c_j. \end{aligned}$$

Note that we can verify the following equations:

$$\begin{aligned} &\sum_{\substack{j:j<i \\ y_j < y_i}} c_j + \sum_{\substack{j:j>i \\ y_j < y_i}} c_j = \sum_{j:y_j < y_i} c_j, \\ &\sum_{\substack{j:j<i \\ y_j > y_i}} c_j + \sum_{\substack{j:j<i \\ y_j > y_i}} c_j = \sum_{j:j<i} c_j, \\ &\sum_{\substack{j:j>i \\ y_j > y_i}} c_j + \sum_{\substack{j:j<i \\ y_j < y_i}} c_j + \sum_{\substack{j:j>i \\ y_j < y_i}} c_j + \sum_{\substack{j:j<i \\ y_j > y_i}} c_j = \sum_{j:j\neq i} c_j = c_{\cdot\cdot} - c_i, \end{aligned}$$

where $c_{\cdot\cdot} = \sum_{j=1}^n c_j$. We can rewrite $\gamma_i(\{c_j\})$ as follows:

$$\gamma_i(\{c_j\}) = c_{\cdot\cdot} - c_i - 2\sum_{j:y_j < y_i} c_j - 2\sum_{j:j<i} c_j + 4\sum_{\substack{j:j<i \\ y_j < y_i}} c_j. \quad (A.6)$$

We will argue that the three summations on the right hand side can be implemented by $O(n \log n)$ algorithms. First, term $\sum_{j:j<i} c_j$ is a formula for partial sums. It is known that an $O(n)$ algorithm exists, by utilizing the relation:

$$\sum_{j:j<i+1} c_j = c_i + \sum_{j:j<i} c_j.$$

Second, after sorting y_j's in increasing order, sums $\sum_{j:y_j<y_i} c_j$ become a partial sums sequence. Hence it can be implemented via an $O(n)$ algorithm. If QuickSort (Hoare 1961) (Knuth 1997, Section 5.2.2: Sorting by Exchanging (pages 113-122)) is adopted, the sorting of y_j's can be done via an $O(n \log n)$ algorithm.

We will argue that sums $\sum_{j:j<i,y_j<y_i} c_j$ for $i = 1, \ldots, n$ can be computed in an $O(n \log n)$ algorithm. WLOG, we assume that: 1. y_i for $i = 1, 2, \ldots, n$ is a permutation of $\{1, 2, \ldots, n\}$ 2. n is dyadic (i.e., $n = 2^L$ where $L \in \mathbb{N}$)

For $\ell = 0, 1, \ldots, L-1$ and $k = 1, 2, \ldots, 2^{L-\ell}$, we define a closed interval:

$$I(\ell, k) := [(k-1) \cdot 2^\ell + 1, k \cdot 2^\ell]$$

We then define the following function:

$$s(i, \ell, k) := \sum_{\substack{j:j<i \\ y_j \in I(\ell,k)}} c_j$$

where $i = 1, \ldots, n$, $\ell = 0, 1, \ldots, L-1$, and $k = 1, 2, \ldots, 2^{L-\ell}$.

Complexity Analysis:

1. **Base Case:** For all ℓ, k we have $s(1, \ell, k) \equiv 0$
2. **Inductive Step:** Suppose for all $i' \leq i$, $s(i', \ell, k)$ values are computed. For each $0 \leq \ell \leq L - 1 < \log_2 n$, there exists exactly one k^* such that $y_i \in I(\ell, k^*)$. The update rule is:

$$s(i+1,\ell,k) = \begin{cases} s(i,\ell,k) + c_i & \text{if } k = k^* \\ s(i,\ell,k) & \text{otherwise} \end{cases}$$

3. **Computational Cost:**
   - The dynamic programming update runs n times (for all $1 \le i \le n$)
   - Each update requires $\le \log_2 n$ operations
   - Total complexity: $O(n \log n)$

Computing the Target Sum: For fixed i ($1 \le i \le n$), to compute $\sum_{j:j<i, y_j<y_i} c_j$:

1. If $y_i = 1$, then $\sum_{j:j<i, y_j<y_i} c_j = 0$
2. For $y_i > 1$, there exists a unique sequence $\ell_1 > \ell_2 > \cdots > \ell_r > 0$ such that:

$$y_i - 1 = 2^{\ell_1} + 2^{\ell_2} + \cdots + 2^{\ell_r}$$

Since $y_i \le n$, we must have $\tau \le \log_2 n$. We then define $k_\alpha, \alpha = 1, \ldots, \tau$ as follows

$$\begin{aligned} k_1 &= 1, \\ k_2 &= 2^{l_1 - l_2} + 1, \\ &\vdots \\ k_n &= (2^{l_1} + \cdots + 2^{l_{n-1}})/2^{l_n} + 1, \\ &\vdots \\ k_\tau &= (2^{l_1} + \cdots + 2^{l_{\tau-1}})/2^{l_\tau} + 1. \end{aligned}$$

One can then verify the following: for $2 \le i \le n$,

$$\sum_{j:j<i, j_1<y_i} c_j = \sum_{\alpha=1}^{\tau} s(i, t_\alpha, k_\alpha).$$

Since $\tau \le \log_2 n$, the above takes no more than $\mathrm{O}(\log n)$ numerical operations. Consequently, computing $\sum_{j:j<i, y_i<y_i} c_j$ for all $i, 1 \le i \le n$, can be done in $\mathrm{O}(n \log n)$. From all the above, we established the result.

We have the following sequence of equations:

$$\sum_{j=1}^{n}\left[\sum_{i=1}^{n} x_1 \mathbf{1}(y_i < y_j)\right]^2 = \sum_{j=1}^{n}\left[\sum_{i=1}^{n} x_1 \mathbf{1}(y_i < y_j)\right] \cdot \left[\sum_{k=1}^{n} x_k \mathbf{1}(y_k < y_j)\right]$$

$$= \sum_{j=1}^{n}\sum_{i=1}^{n}\sum_{k=1}^{n} x_i \cdot x_k \cdot \mathbf{1}(y_i < y_j \text{ and } y_k < y_j)$$

$$= \sum_{i=1}^{n} x_i \left[\sum_{k:y\le y_k} x_k \sum_{j=1}^{n} \mathbf{1}(y_k < y_j) + \sum_{k:y>y_k} x_k \sum_{j=1}^{n} \mathbf{1}(y_k < y_j)\right].$$

The last expression implies the following steps to compute for SIRS(X,Y).

1. 1.For $k = 1, \ldots, n$, compute $\alpha_k := \sum_{j=1}^{n} \mathbf{1}(y_k < y_j)$;
2. 2.For $i = 1, \ldots, n$, compute $\beta_i := \sum_{k:y_k \ge y_i} x_k \alpha_k$;
3. 3.For $i = 1, \ldots, n$, compute $\gamma_i := \sum_{k:y_k < y_i} x_k$;
4. 4.Compute
$$\mathrm{SIRS}(X, Y) = \frac{\sum_{i=1}^{n} x_i(\beta_i + \gamma_i \alpha_i)}{n(n-1)(n-2)}.$$

Since α's, β's, and γ's are partial sums, it is easy to verify that each of the above steps can be done within $\mathrm{O}(n \log n)$ operations on average, hence the entire algorithm takes $\mathrm{O}(n \log n)$ operations on average.

References

[1] Gabor J. Szekely and Maria L. Rizzo, *Energy statistics: A class of statistics based on distances*. Journal of Statistical Planning; Inference, 2013. Available: https://www.sciencedirect.com/science/article/abs/pii/S0378375813000633

[2] Gabor J. Szekely and Maria L. Rizzo, *The energy of data and distance correlation*. Chapman & Hall/CRC Monographs on Statistics; Applied Probability, 2023. Available: https://www.routledge.com/The-Energy-of-Data-and-Distance-Correlation/Szekely-Rizzo/p/book/9781482242744

[3] G. J. Székely, M. L. Rizzo, and N. K. Bakirov, "Measuring and testing dependence by correlation of distances," *The Annals of Statistics*, vol. 35, no. 6, pp. 2769–2794, 2007, doi: 10.1214/009053607000000505.

[4] Songzi Li and Maria L. Rizzo, *K-groups: A generalization of k-means clustering*. 2017. Available: https://arxiv.org/pdf/1711.04359

[5] M. L. Rizzo and G. J. Székely, "DISCO analysis: A nonparametric extension of analysis of variance," *The Annals of Applied Statistics*, vol. 4, no. 2, pp. 1034–1055, 2010, doi: 10.1214/09-AOAS245.

[6] George Casella and Roger L. Berger, *Statistical inference*. Duxbury Press, Pacific Grove, 2002. Available: https://www.routledge.com/Statistical-Inference/Casella-Berger/p/book/9781032593036

[7] J. Raymaekers and P. J. Rousseeuw, "Distance covariance, independence, and pairwise differences," *The American Statistician*, vol. 0, no. 0, pp. 1–7, 2024, doi: 10.1080/00031305.2024.2374966.

[8] N. Ilter and H. Guvenir, "Dermatology." UCI Machine Learning Repository, 1998.

[9] X. Huo and G. J. Szekely, "Fast computing for distance covariance." 2014. Available: https://arxiv.org/abs/1410.1503

[10] R. Lyons, "Distance covariance in metric spaces," *The Annals of Probability*, vol. 41, Jun. 2011, doi: 10.1214/12-AOP803.

[11] G. De Pierris and M. Friedman, "Kant and Hume on Causality," in *The Stanford encyclopedia of philosophy*, Fall 2024., E. N. Zalta and U. Nodelman, Eds., https://plato.stanford.edu/archives/fall2024/entries/kant-hume-causality/; Metaphysics Research Lab, Stanford University, 2024.

[12] V. S. Koroljuk and Y. V. Borovskich, *Theory of u-statistics*, vol. 273. in Mathematics and its applications, vol. 273. Dordrecht: Kluwer Academic Publishers Group, 1994.

[13] D. Christensen, "Fast algorithms for the calculation of kendall's t," *Computational Statistics*, vol. 20, pp. 51–62, Mar. 2005, doi: 10.1007/BF02736122.

[14] S. Leyder, J. Raymaekers, P. J. Rousseeuw, T. V. Deuren, and T. Verdonck, "Independent component analysis by robust distance correlation." 2025. Available: https://arxiv.org/abs/2505.09425

[15] C. Potvin and D. A. Roff, "Distribution-free and robust statistical methods: Viable alternatives to parametric statistics?" *Ecology*, vol. 74, pp. 1617–1628, 1993.

[16] M. L. Puri and P. K. Sen, *Nonparametric methods in multivariate analysis*. New York: Wiley, 1971.

[17] G. J. Székely and N. K. Bakirov, "Extremal probabilities for gaussian quadratic forms," *Probab. Theory Related Fields*, vol. 126, pp. 184–202, 2003.

[18] G. J. Székely and M. L. Rizzo, "A new test for multivariate normality," *J. Multivariate Anal.*, vol. 93, pp. 58–80, 2005.

[19] G. J. Székely and M. L. Rizzo, "Hierarchical clustering via joint between-within distances: Extending ward's minimum variance method," *J. Classification*, vol. 22, pp. 151–183, 2005.

[20] S. M. Tracz, P. B. Elmore, and J. T. Pohlmann, "Correlational meta-analysis: Independent and nonindependent cases," *Educational and Psychological Measurement*, vol. 52, pp. 879–888, 1992.

[21] R. von Mises, "On the asymptotic distribution of differentiable statistical functionals," *Ann. Math. Statist.*, vol. 18, pp. 309–348, 1947.

[22] S. S. Wilks, "On the independence of k sets of normally distributed statistical variables," *Econometrica*, vol. 3, pp. 309–326, 1935.

[23] C. Huang and X. Huo, "A statistically and numerically efficient independence test based on random projections and distance covariance." 2017. Available: https://arxiv.org/abs/1701.06054

[24] K.-Y. Lee, B. Li, and H. Zhao, "Variable selection via additive conditional independence," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 78, no. Part 1, pp. 1037–1055, 2016.

[25] L. Runze, Z. Wei, and L. Zhu, "Feature screening via distance correlation learning," *Journal of the American Statistical Association*, vol. 107, no. 499, pp. 1129–1139, 2012.

[26] L. Miles, L. Jacob, and M. J. Wainwright, "A more powerful two-sample test in high dimensions using random projection," in *Advances in neural information processing systems*, 2011, pp. 1206–1214.

[27] D. Lopez-Paz, P. Hennig, and B. Schölkopf, "The randomized dependence coefficient," in *Advances in neural information processing systems*, 2013, pp. 1–9.