Department of Mathematics 2007

Comparative study of algorithms and statistical tests: testing some properties of distributions

Candidate name: Hamaad Musharaf Shah

Submitted for the Master of Science, London School of Economics, University of London

Note: This MSc thesis was awarded the third highest mark among thirty students. It was also awarded an 'Outstanding Distinction'.

Contents

List of Tables

Summary

This paper makes a comparative study of three algorithms and some non-parametric statistical tests. The algorithms and statistical tests are programmed in Java. Large data sets with a large number of distinct elements are used in experiments to compare their performance in testing certain properties of probability distributions: closeness, identity and independence. The statistical tests used for comparison are the χ^2 one sample and two sample tests and the Kolmogorov-Smirnov one sample and two sample tests.

1 Introduction

The aim of this paper is to make a comparative study between three algorithms and some non-parametric statistical tests. The closeness algorithm tests whether two unknown probability distributions over a finite set of elements are similar (Batu et al., 2000). The identity algorithm tests whether an unknown probability distribution is similar to a known probability distribution, both of which are over a finite set of elements (Batu et al., 2003). The independence algorithm tests whether the joint probability distribution is similar to the product of its marginal distributions (if it is, then we have independence) (Batu et al., 2000; Batu et al., 2003).

These algorithms all rely on independent sampling from the unknown distributions, however they do not compute frequency distributions or cumulative probability distributions, like some non-parametric statistics. They make their decision by using the idea of collisions between elements from different sample sets and between elements of the same sample set. A detailed discussion of collisions, along with the explanation of each algorithm and how these collisions play a vital role in their mechanism is contained in section 2.

The non-parametric statistics we employ are the χ^2 one sample and two sample tests and the Kolmogorov-Smirnov one sample and two sample tests. Neave and Worthington (1988) contain a good explanation of the χ^2 one sample test and also of the Kolmogorov-Smirnov one sample and two sample tests, while Langley (1971) describes using the two sample version of the χ^2 test. However we employ the technique as suggested in the book by Durbin (1973) in order to compute our Kolmogorov-Smirnov two sample test statistic. These tests have been chosen for this comparative study as they attempt to answer the same questions (hypotheses) regarding probability distributions as our algorithms. Section 3 explains these statistical tests and the decision rules regarding rejecting (or accepting) the null hypothesis.

In section 4, I describe the Java implementation of the algorithms and statistical tests. All the relevant Java files and how they function will be explained in section 4.

In section 5 I conduct the experiments using large data sets on these algorithms and statistical tests. A discussion of the results of these experiments is also made.

2 The algorithms

There are three algorithms that are discussed in this paper. Each algorithm tests different properties of distributions, however they all rely on similar principles. The closeness algorithm by Batu et al. (2000) tests whether two unknown distributions are close in the L_1-norm. It uses the idea of collisions: we sample from the two distributions, and if they are close, one can expect to obtain a reasonably high proportion of elements in the sample sets that are the same. This idea of collisions is based on the paper by Goldreich and Ron (2000).

The identity algorithm by Batu et al. (2003) tests whether an unknown distribution is identical to a known distribution. One can see that the difference between the closeness algorithm and the identity algorithm is simply that in the latter, one of the distributions is completely known.

Lastly, the independence algorithm uses the aforementioned closeness algorithm to test whether the joint distribution of a pair of sets is close to the product of the marginal distributions of each set. However before I begin to explain the algorithms, I will discuss a few preliminaries: some notation and theorems that will be used throughout the paper.

2.1 Preliminaries

The notation and theorems in this section are taken from Batu (2001), and Batu et al. (2003).

For any $n \in \mathbb{N}$, let the set $\{1, ..., n\}$ be denoted by $[n]$. This can be thought of as the set of distinct elements in a data set: we might have a set of n different races, which we can code from 1 to n. The discrete probability distribution over $[n]$ is denoted by $\vec{p} = \{p_1, ..., p_n\}$. This vector gives us the probability of getting the i^{th} element in the set $[n]$: p_i. We denote uniform random sampling from $[n]$ as $x \in_R [n]$. Let $\vec{p}, \vec{q}, \vec{r}$ denote random variables over a set $[n]$, while A, B, C, D denote joint random variables over a pair of sets $[n] \times [m]$. Given any set R, the uniform distribution over it is denoted by U_R.

For any vector $\vec{v}$, its L_1-norm is denoted by $|\vec{v}|$, its L_2-norm is denoted by $\|\vec{v}\|$, and its L_∞-norm is denoted by $\|\vec{v}\|_\infty$. These norms are,

$$\begin{aligned} |\vec{v}| &= \textstyle\sum_{i=1}^{n} |v_i| \\ \|\vec{v}\| &= \sqrt{\textstyle\sum_{i=1}^{n} v_i^2} \\ \|\vec{v}\|_\infty &= \max_{i \in [n]} |v_i| \end{aligned}$$

If two probability distributions $\vec{p}$ and $\vec{q}$ are ϵ-close to each other in the L_1-norm, then $|\vec{p} - \vec{q}| \leq \epsilon$. This is the basic question that the algorithms will attempt to answer.

Another extremely important idea that underlies this paper is that of collision probabilities (Batu et al., 2000; Goldreich and Ron, 2000). There are two types of these probabilities. Self-collision probabilities are defined as follows.

Take a set of m samples, F_p, from a data set distributed according to the probability distribution $\vec{p}$. Then for all $i, j \in [m]$, where $i \neq j$, the following holds,

$$\begin{aligned}\Pr[F^i_{\vec{p}} &= F^j_{\vec{p}}] = \vec{p}.\vec{p} \\ \Pr[F^i_{\vec{p}} &= F^j_{\vec{p}}] = \|\vec{p}\|^2\end{aligned}$$

It is simply the probability of obtaining two of the same elements at different positions in the same sample set.

Similarly for collision probabilities, we take two sets of m samples each from two data sets distributed according to $\vec{p}$ and $\vec{q}$. Denote these sample sets by $Q_{\vec{p}}$ and $Q_{\vec{q}}$. Then for all $i, j \in [m]$, the following holds,

$$\begin{aligned}\Pr[Q^i_{\vec{p}} &= Q^j_{\vec{q}}] = \vec{p}.\vec{q} \\ \Pr[Q^i_{\vec{p}} &= Q^j_{\vec{q}}] = \sum_{i \in [m]} p_i q_i\end{aligned}$$

Simply put, it is the probability of obtaining the same element, regardless of position, in the two different sample sets.

If $f = O(gpoly(\log(g))$, then we denote this in short by $f = \tilde{O}(g)$. This is used simply to hide polynomial dependencies on a logarithm of any of the variables involved, and in this case it is the variable g. Furthermore we repeat all tests to get a high level of confidence in their results: a confidence level of $(1 - \delta)$ can be attained with $\log \frac{1}{\delta}$ number of trials.

We define two different kinds of probability distributions in the following manner.

Definition 2.1 (Batu, 2001, p.8) *A probability distribution $\vec{p}$ is called a black-box distribution if upon request it outputs a sample from the data distributed according to this $\vec{p}$.*

The black-box can simply give us a sample from the data distributed according to the distribution $\vec{p}$: it can not give us the probability of picking that element. This is the ethos of these algorithms, in particular that of the closeness algorithm: they would like to answer questions regarding probability distributions without have to explicitly calculate the probability values.

Definition 2.2 (Batu, 2001, p.8) *A probability distribution $\vec{p}$ is called an explicit distribution if it outputs the probability value of element i if an algorithm asks for it.*

So an explicit distribution, as the name implies, explicitly tells us the exact probability value of any element $i \in [n]$.

We will now present two important theorems from statistics that will be used in our algorithms. The first one is the Chebyshev inequality while the second one is Chernoff bounds.

Theorem 2.3 (Batu, 2001, p.9) *Let X be a random variable with expectation $E[X]$ and standard deviation σ_X. Then for any $k > 0$,*

$$\Pr[|X - E[X]| \geq k\sigma_X] \leq \frac{1}{k^2}$$

Assume we take $k = 2$. From the Chebyshev inequality we can expect that, with probability of at most 1/4, we will find values of the random variable X that are at least two standard deviations away from the mean. So this theorem tells us the probability of finding elements that are at a certain distance from the mean; this distance is in terms of the standard deviation.

Theorem 2.4 (Batu, 2001, p.9) *Let X_1,X_2,...,X_m be m independent random variables with $X_i \in [0,1]$. Define $\rho := \frac{1}{m} \sum_{i \in [m]} E[X_i]$. Then for every $\gamma \in [0,1]$, the following bounds hold,*

$$\Pr\left[\frac{1}{m}\sum_{i\in[m]} X_i > (1+\gamma)\rho\right] < \exp(-\gamma^2\rho m/3)$$
$$\Pr\left[\frac{1}{m}\sum_{i\in[m]} X_i < (1-\gamma)\rho\right] < \exp(-\gamma^2\rho m/2)$$

From above, we can see that the essence of the Chernoff bounds is that it bounds a sequence of random variables. ρ is the average of the sum of the averages of each random variable. The bounds tell us that the probability of the mean of each random variable X_i being greater than $(1+\gamma)\rho$ is bounded by a term involving γ, ρ and the number of random variables m. The probability of the mean of each random variable X_i being lesser than $(1-\gamma)\rho$ is also bounded by a similar, yet different, term.

Finally we will discuss two tools that are specifically used in our identity test. These are restriction and coarsening. Restriction is the conditional probability distribution when we restrict ourselves to a subset of the domain of the distribution $[n]$. Coarsening is the probability distribution over the partition set of the domain $[n]$: it contain probabilities of picking a particular restriction from the partition set (a partition set is simply a set of restrictions). Both of these tools are formally defined as follows.

Definition 2.5 (Batu, 2001, p.10) *Given a probability distribution $\vec{p}$ over R, and $R' \subseteq R$, we define the restriction as $(\vec{p}_{|R'})$, which is the distribution over R' such that for all $i \in R'$, $(\vec{p}_{|R'})_i = \frac{p_i}{\vec{p}(R')}$.*

So a restriction is a conditional probability distribution such that the probability of picking an element from it is equal to the probability of picking that element from the probability distribution $\vec{p}$ divided by the probability of picking that particular restriction.

Definition 2.6 (Batu, 2001, p.10) *Given a probability distribution $\vec{p}$ over R, and a partition set $\mathcal{R} = \{R_1, ..., R_k\}$ of R, the coarsening $(\vec{p}_{\langle\mathcal{R}\rangle})$ is the probability distribution over $[k]$ such that $(\vec{p}_{\langle\mathcal{R}\rangle})_i = \vec{p}(R_i)$.*

This is simply the probability distribution of picking a particular restriction from the partition set of R.

Moreover if $R' \subseteq R$, then the probability of picking the restriction is $\vec{p}(R') := \sum_{i \in R'} p_i$. That is to say the probability of picking a particular restriction is equal to the sum of the probabilities of picking the elements in that restriction from $\vec{p}$. This idea will be useful in our identity algorithm (step (9)).

The following are some important lemmas relating to restrictions and coarsening that will be used later on.

Lemma 2.7 (Batu et al., 2003, p.3) *Let $\vec{p}$ and $\vec{q}$ be probability distributions over $[n]$, and $\mathcal{R} = \{R_1, ..., R_k\}$ is a partition of $[n]$. If for all $i \in [k]$, $\left|\vec{p}_{|R_i} - \vec{q}_{|R_i}\right| \leq \epsilon_1$ and $\left|\vec{p}_{\langle\mathcal{R}\rangle} - \vec{q}_{\langle\mathcal{R}\rangle}\right| \leq \epsilon_2$, then $|\vec{p} - \vec{q}| \leq \epsilon_1 + \epsilon_2$.*

So lemma 2.7 tells us that if two distributions $\vec{p}$ and $\vec{q}$ are close with respect to each of their restrictions, and with respect to their coarsening, then they are close in the L_1-norm. One very important thing to note here is that if $(1-\epsilon)\vec{p}_{|R_i} \leq \vec{q}_{|R_i} \leq (1+\epsilon)\vec{p}_{|R_i}$ for all $i \in [k]$, then $\left|\vec{p}_{\langle\mathcal{R}\rangle} - \vec{q}_{\langle\mathcal{R}\rangle}\right| \leq \epsilon$. This is an important idea that will be used in our identity test. The following lemma shows a partial converse of this idea: if $\vec{p}$ and $\vec{q}$ are close, then they are also close when restricted to sufficiently heavy partitions (partitions with a high probability of being picked) of the domain.

Lemma 2.8 (Batu et al., 2003, p.3) *Let $\vec{p}$ and $\vec{q}$ be probability distributions over $[n]$ and let some $n' \subseteq [n]$. Then $\left|\vec{p}_{|n'} - \vec{q}_{|n'}\right| \leq 2\left|\vec{p} - \vec{q}\right| / \vec{p}(n')$.*

2.2 The closeness algorithm

Suppose there are two data sets, each with n distinct elements. The first data set has the probability distribution $\vec{p}$ over $[n]$ while the second one is $\vec{q}$ over $[n]$. Moreover, $\vec{p}$ and $\vec{q}$ are unknown. The closeness algorithm devised by Batu et al. (2000) outlines a procedure by which we can figure out whether $\vec{p}$ and $\vec{q}$ are ϵ-close in the L_1-norm, where ϵ is some distance parameter.

The main idea behind the closeness algorithm is that we take a few set of samples from the above data sets. Using the collision idea by Goldreich and Ron (2000), which is described in section 2.1, we check whether the sample set taken from $\vec{p}$ is similar in some sense to the sample set taken from $\vec{q}$. The appealing property of this algorithm is that it runs in sublinear time in the size of the domain of $\vec{p}$ and $\vec{q}$: its sample complexity (the number of samples needed) is sublinear in the size of the domain of $\vec{p}$ and $\vec{q}$.

The following theorem describes testing whether $\vec{p}$ and $\vec{q}$ are ϵ-close in the L_1-norm.

Theorem 2.9 (Batu et al. (2000, p.261) *Given two unknown distributions $\vec{p}$ and $\vec{q}$ over $[n]$, and a parameter δ, the closeness test runs in time $O(\epsilon^{-4} n^{2/3} \log n \log \frac{1}{\delta})$ such that if $|\vec{p} - \vec{q}| \leq \max(\frac{\epsilon^2}{32\sqrt[3]{n}}, \frac{\epsilon}{4\sqrt{n}})$, then the closeness test outputs `Pass` with at least $1-\delta$ probability and if $|\vec{p} - \vec{q}| > \epsilon$, then the test outputs `Fail` with at least $1-\delta$ probability.*

From theorem 2.1 we can clearly see that the test runs in sublinear time in the domain size of the distributions: I specifically refer to the $n^{2/3}$ component. It is also interesting to note

that the algorithm has an error probability of δ. For instance, if $|\vec{p} - \vec{q}| \leq \max(\frac{\epsilon^2}{32\sqrt[3]{n}}, \frac{\epsilon}{4\sqrt{n}})$ the algorithm outputs `Fail` with at most δ probability: it makes an error with at most δ probability. This is synonymous with the type I and type II errors in statistics. I will make a slight digression here in explaining these type I and type II errors and their relation with the parameter δ, which will prove useful later on.

Assume that we have the following statistical hypotheses (Anderson et al., 1994), where H_0 is called the null hypothesis and H_A is called the alternative hypothesis.

$$\begin{aligned} H_0 &: \vec{p} = \vec{q} \\ H_A &: \vec{p} \neq \vec{q} \end{aligned}$$

If the null hypothesis is true, however our statistical test rejects it, then we have made a type I error. The probability of rejecting a true null hypothesis is denoted by α and is called the level of significance. On the other hand, if the null hypothesis is false but our statistical test accepts it, then we have a type II error. The probability of making such an error is denoted by β. In practise we usually tend to control for the type I error explicitly by setting out a level of significance before conducting our statistical test.

Thus the δ parameter in the algorithm and the α parameter in statistics are related: they give us a notion of how error-prone our techniques are in rejecting a true null hypothesis. This relation will become useful when we run our algorithms and their statistical counterparts in section 5. For all our algorithms and statistical tests, we will run them such that δ and α have the same values: both the algorithms and statistical tests will have a similar probability of making the wrong decision.

Moving on, the closeness test for the L_1-norm relies upon a closeness test for the L_2-norm. The latter test is given by the following theorem.

Theorem 2.10 (Batu et al. (2000, p.261) *Given two unknown distributions $\vec{p}$ and $\vec{q}$ over $[n]$, and a parameter δ, the closeness test runs in time $O(\epsilon^{-4} \log \frac{1}{\delta})$ such that if $\|\vec{p} - \vec{q}\| \leq \epsilon/2$, then the closeness test outputs `Pass` with at least $1 - \delta$ probability and if $\|\vec{p} - \vec{q}\| > \epsilon$, then the test outputs `Pass` with less than δ probability.*

This L_2-norm closeness algorithm in described in the next section. The closeness algorithm for the L_1-norm is given in section 2.2.2.

2.2.1 The closeness algorithm for the L_2-norm

The closeness algorithm for the L_2-norm as described by theorem 2.10 is given as follows (Batu et al., 2000, p.261).

L_2`-closeness-test(`$\vec{p}$`,`$\vec{q}$`,`m`,`ϵ`,`δ`)`
`Run the test` $O(\log(\frac{1}{\delta}))$ `times`

1. `Let` F_p = `Set of` m `samples from` $\vec{p}$
2. `Let` F_q = `Set of` m `samples from` $\vec{q}$

3. Let r_p be the number of pairwise self-collisions in F_p
4. Let r_q be the number of pairwise self-collisions in F_q
5. Let Q_p = Set of m samples from $\vec{p}$
6. Let Q_q = Set of m samples from $\vec{q}$
7. Let s_{pq} be the number of collisions between Q_p and Q_p
8. Let $r = \frac{2m}{m-1}(r_p + r_q)$
9. Let $s = 2s_{pq}$
10. If $r - s > m^2\epsilon^2/2$ then reject
11. Reject if the majority of the above iterations reject
    Otherwise Accept

In the test above, we make estimations of the self-collision and collision probabilities as our probability distributions $\vec{p}$ and $\vec{q}$ are unknown, so we can not explicitly calculate the relevant probabilities. For the self-collision probabilities, we take a set of m samples from $\vec{p}$ and $\vec{q}$ each: F_p and F_q respectively. Then for all $i, j \in [m]$ where $i \neq j$, we count the number of samples that are the same as each other in each sample set. These are denoted by r_p and r_q. Similarly for the collision probabilities, we take a set of m samples from $\vec{p}$ and $\vec{q}$ each: Q_p and Q_q respectively. Then for all $i, j \in [m]$ we count the number of samples that occur in both sample sets. This is denoted by s_{pq}. In this manner we obtain estimations of self-collision and collision probabilities.

If $\vec{p}$ and $\vec{q}$ are close then we expect that there will be little difference between the self-collision and collision probabilities. This can be illustrated as follows.

$$\begin{aligned}
\|\vec{p} - \vec{q}\|^2 &= \sum_{i=1}^{n} (p_i - q_i)^2 \\
\|\vec{p} - \vec{q}\|^2 &= \sum_{i=1}^{n} ({p_i}^2 - 2p_i q_i + q_i^2) \\
\|\vec{p} - \vec{q}\|^2 &= \sum_{i=1}^{n} ({p_i}^2) - \sum_{i=1}^{n} (2p_i q_i) + \sum_{i=1}^{n} (q_i^2) \\
\|\vec{p} - \vec{q}\|^2 &= \|\vec{p}\|^2 - 2(\vec{p} \cdot \vec{q}) + \|\vec{q}\|^2 \\
\|\vec{p} - \vec{q}\|^2 &= (\|\vec{p}\|^2 + \|\vec{q}\|^2) - 2(\vec{p} \cdot \vec{q})
\end{aligned}$$

Thus from our estimations of the collision and self-collision probabilities, and by finding out the difference between the two we can check whether two distributions are close to each other. For instance, if the difference is zero, then $\vec{p}$ and $\vec{q}$ are the exact same distribution;

they would be 0-close. In the test above $(r - s)$ acts as an estimator of our requirement: $\|\vec{p} - \vec{q}\|^2$.

Since we need to take samples from two different distributions to calculate s, we must bound the variance of this variable. We also scale r_p and r_q appropriately as for the self-collision probabilities we consider pairs of samples $i, j \in [m]$ such that $i \neq j$, but we do not do this for the collision probabilities. Since there is a discrepency here in calculating the self-collision and collision probabilities, we rectify this by multiplying the sum of r_p and r_q by $\left(\frac{2m}{m-1}\right)$. So after these arguments, the following holds,

$$\begin{aligned} E[r] &= m^2(\|\vec{p}\|^2 + \|\vec{q}\|^2) \\ E[s] &= 2m^2(\vec{p} \cdot \vec{q}) \end{aligned}$$

and,

$$\begin{aligned} E[r - s] &= m^2(\|\vec{p}\|^2 + \|\vec{q}\|^2 - 2(\vec{p} \cdot \vec{q})) \\ E[r - s] &= m^2(\|\vec{p} - \vec{q}\|^2) \end{aligned}$$

However when calculating the collision probabilities the pairwise samples are not independent and for this reason we apply Chebyshev's inequality. To use this inequality we need a bound on the variance of the relevant random variable under consideration.

Moving on, we analyse r_p and r_q similarly as above by the following lemma.

Lemma 2.11 (Batu et al., 2000, p.262) *Let A be either r_p or r_q. Then $E[A] = \binom{m}{2} \cdot \|\vec{p}\|^2$ and $Var[A] \leq 2(E[A])^{\frac{3}{2}}$.*

One should note in the expectations value of the self-collision probabilities that we take into account the combination of samples that can be take from the relevant set of samples. Also, the variance is bounded by a function of the expectation: the second moment depends on the first moment.

The variance bound for $(r - s)$ is more involved. We use the supremum norms of the distributions $\vec{p}$ and $\vec{q}$ to define this variance. The supremum norm is simply the element with the largest weight (probability) in the distributions.

Lemma 2.12 (Batu et al., 2000, p.262) *There exists a constance c such that $Var[r-s] \leq c(m^3b^2 + m^2b)$, where $b = \max(\|p\|_\infty, \|q\|_\infty)$.*

Proof. Let F be the set $[m]$. For $(i, j) \in F \times F$, we define a binary (or dummy, indicator, etc) variable $C_{i,j}$ which takes on the value 1 if the i^{th} sample in Q_p is the same as the j^{th} sample in Q_q are the same. Thus we can connect this to our previous s_{pq} variable: $s_{pq} = \sum_{i,j} C_{i,j}$.

Furthermore, to assist in manipulating the variance later on, we define $\bar{C}_{i,j} = C_{i,j} - E[C_{i,j}]$. Now we define the variance of this binary variable.

$$\begin{aligned} Var[\textstyle\sum_{F\times F} C_{i,j}] &= E[(\textstyle\sum_{F\times F} \bar{C}_{i,j})^2] \\ Var[\textstyle\sum_{F\times F} C_{i,j}] &= E[\textstyle\sum_{i,j} (\bar{C}_{i,j})^2 + 2\sum_{(i,j)\neq(k,l)} \bar{C}_{i,j}\bar{C}_{k,l}] \\ Var[\textstyle\sum_{F\times F} C_{i,j}] &\leq m^2(p\cdot q) + 2E[\textstyle\sum_{(i,j)\neq(k,l)} \bar{C}_{i,j}\bar{C}_{k,l}] \end{aligned}$$

To analyse the last expectation we use two facts. Firstly, by the definition of covariance of two random variables,

$$E[\bar{C}_{i,j}\bar{C}_{k,l}] \leq E[C_{i,j}C_{k,l}]$$

Secondly, $C_{i,j}$ and $C_{k,l}$ are not independent when $i = k$ or $j = l$. Expanding the above sum leads to,

$$\begin{aligned} E\left[\sum_{\substack{(i,j),(k,l)\epsilon F\times F \\ (i,j)\neq(k,l)}} \bar{C}_{i,j}\bar{C}_{k,l}\right] &= E\left[\sum_{\substack{(i,j),(i,l)\epsilon F\times F \\ j\neq l}} \bar{C}_{i,j}\bar{C}_{k,l} + \sum_{\substack{(i,j),(k,j)\epsilon F\times F \\ i\neq k}} \bar{C}_{i,j}\bar{C}_{k,l}\right] \\ &\leq E\left[\sum_{\substack{(i,j),(i,l)\epsilon F\times F \\ j\neq l}} C_{i,j}C_{k,l} + \sum_{\substack{(i,j),(k,j)\epsilon F\times F \\ i\neq k}} C_{i,j}C_{k,l}\right] \\ &\leq cm^3 \sum_{f\epsilon[n]} p_f q_f^2 + p_f^2 q_f \\ &\leq cm^3 b^2 \sum_{f\epsilon[n]} q_f \leq cm^3 b^2 \end{aligned}$$

where c is some constant number. We use lemma 2.11 to bund the variance of $(r - s)$. As $Var[r] \leq cm^2 b$ and r and s are independent random variables, the following holds,

$$Var[r - s] \leq c(m^3 b^2 + m^2 b)$$

■

Thus in lemma 2.12, we have proven a bound for the variance of $(r - s)$. Now we can apply Chebyshev's inequality (theorem 2.3). If we let $m = O(\epsilon^{-4})$, we can achieve an error probability of less than 1/3. Furthermore, with $O(\log \frac{1}{\delta})$ iterations we can achieve an error probability of at most δ.

Lemma 2.13 (Batu et al., 2000, p.263) *Given two unknown probability distributions $\vec{p}$ and $\vec{q}$ such that $b = \max(\|\vec{p}\|_\infty, \|\vec{q}\|_\infty)$ and $m = O((b^2 + \epsilon^2\sqrt{b})/\epsilon^4)$, if $\|\vec{p} - \vec{q}\| \leq \epsilon/2$, then the L_2-closeness-test passes with a probability of at least $1 - \delta$. If $\|\vec{p} - \vec{q}\| > \epsilon$, then the L_2-closeness-test passes with a probability of less than δ. The running time of this test is $O(m \log(\frac{1}{\delta}))$.*

Proof. We use Chebyshev's inequality (theorem 2.3) here. Let $A = (r - s)$, then for some constant c, we have,

$$\Pr[|A - E[A]| > \rho] \leq \frac{c(m^3b^2 + m^2b)}{\rho^2}$$

We arrive at the above equation by letting $k\sigma_A$ (from theorem 2.3) be ρ, where $k > 0$ and σ_A is the standard deviation of the A. So, as the standard deviation is simply the square root of the variance and using lemma 2.12,

$$\begin{aligned}
\Pr[|A - E[A]| &> k\sigma_A] \leq \frac{1}{k^2} \\
\Pr[|A - E[A]| &> k\sigma_A] \leq \frac{Var[A]}{k^2 Var[A]} \\
\Pr[|A - E[A]| &> k\sigma_A] \leq \frac{c(m^3b^2 + m^2b)}{k^2\sigma^2} \\
\Pr[|A - E[A]| &> \rho] \leq \frac{c(m^3b^2 + m^2b)}{\rho^2}
\end{aligned}$$

The manipulation above simply shows how we have used the Chebyshev's inequality in this proof. If $\|\vec{p} - \vec{q}\| \leq \epsilon/2$, then for one iteration, we have the following probability of outputting pass,

$$\begin{aligned}
\Pr[\texttt{Pass}] &= \Pr[(r - s) < \frac{m^2\epsilon^2}{2}] \\
&\geq \Pr[|(r - s) - E(r - s)| < \frac{m^2\epsilon^2}{4}] \\
&\geq 1 - \frac{4c(m^3b^2 + m^2b)}{m^4\epsilon^4} \\
&\geq 1 - \frac{4Var[r - s]}{m^4\epsilon^4}
\end{aligned}$$

This probability will be at least $2/3$ whenever, for some constant c, the following holds,

$$m > \frac{c(b^2 + \epsilon^2\sqrt{b})}{\epsilon^4}$$

■

2.2.2 The closeness algorithm for the L_1-norm

The L_1-norm test is given as follows (Batu et al., 2000, p.262),

L_1-`closeness-test(`$\vec{p}$`,`$\vec{q}$`,`ϵ`,`δ`)`

1. `Sample` $\vec{p}$ `and` $\vec{q}$ `for` $M = O(\max(\epsilon^{-2}, 4)n^{2/3}\log n)$ `times`
2. `Let` S_p `and` S_q `be the sample sets obtained by discarding elements that occur less than` $(1 - \epsilon/63)Mn^{-2/3}$ `times`

3. `If` S_p `and` S_q `are empty`

4. L_2`-closeness-test(`$\vec{p}$`,`$\vec{q}$`,`$O(n^{2/3}/\epsilon^4)$`,`$\epsilon/2\sqrt{n}$`,`$\delta/2$`)`

5. `else`
   $L_i^p =$ `The number of times element` i `appears in` S_p
   $L_i^q =$ `The number of times element` i `appears in` S_q

6. `Fail if` $\sum_i |L_i^p - L_i^q| > \epsilon M/8$

7. `Define` $\vec{p}'$ `as: take a sample from` $\vec{p}$`. If this element is not in` S_p `output it; otherwise output an` $x \in_R [n]$`.`
   `Define` $\vec{q}'$ `similarly`

8. L_2`-closeness-test(`$\vec{p}'$`,`$\vec{q}'$`,`$O(n^{2/3}/\epsilon^4)$`,`$\epsilon/2\sqrt{n}$`,`$\delta/2$`)`

This test relies on the L_2-norm test described in section 2.2.1, and it proceeds in two stages (Batu et al., 2000, p.263). Before we proceed, the following definition will be needed.

Definition 2.14 (Batu et al., 2000, p.261) *An element is called big with respect to a distribution* $\vec{p}$ *if* $p_i > \frac{1}{n^{2/3}}$.

Using this definition, the first stage of the algorithm removes the big elements from the sample sets, and places them in sets called S_p and S_q, while estimating the contribution of these big elements to the distance between $\vec{p}$ and $\vec{q}$ in the L_1-norm.

The second stage of the algorithm (if S_p and S_q are not empty) calls the L_2-closeness-test upon our filtered distribution. The closeness parameter we apply for this call is $\frac{\epsilon}{2\sqrt{n}}$. This call is correct via lemma 2.13 using $b = n^{-2/3}$. Using these values of the closeness parameter and b, the number of samples is $m = O(n^{2/3}\epsilon^{-4})$. The heavy elements are defined as above by optimising the running-time trade-off between the two stages of the algorithm.

Moreover, to compute the weights of these big elements as defined above to within a multiplicative factor of $O(\epsilon)$, we need a sample set of size $O(\epsilon^{-2}n^{2/3}\log n)$. This is done by the following lemma.

Lemma 2.15 (Batu et al., 2000, p.263) *Let* $\epsilon \leq \frac{1}{2}$. *In* L_1*-closeness-test, after performing* $M = O(\epsilon^{-2}n^{2/3}\log n)$ *samples from the probability distribution* $\vec{p}$, *we define* $\bar{p}_i = \frac{l_i^p}{M}$. *Then with at least* $(1 - \frac{1}{n})$ *probability, the following hold for all* i, (1) *if* $p_i \geq \epsilon^2 n^{-2/3}$ *then* $|\bar{p}_i - p_i| < \frac{\epsilon}{63}\max(p_i, n^{-2/3})$, (2) *if* $p_i < \epsilon^2 n^{-2/3}$, *then* $\bar{p}_i < (1 - \epsilon/63)n^{-2/3}$.

Proof. To prove this lemma, we analyse three cases. Using Chernoff bounds (theorem 2.4) we show that for each i, with a probability of at least $(1 - \frac{1}{n^2})$, the following hold,

$$(1a) \text{ If } p_i > n^{-2/3} \text{ then } |\bar{p}_i - p_i| < \epsilon p_i/63$$

(1b) If $\epsilon^2 n^{-2/3} < \ p_i \leq n^{-2/3}$ then $|\bar{p}_i - p_i| < \epsilon n^{-2/3}/63$

(2) If $p_i < \epsilon^2 n^{-2/3}$ then $\bar{p}_i < 3\epsilon^2 n^{-2/3}$

However as we have assumed that $\epsilon \leq \frac{1}{2}$, then $3\epsilon^2 \leq (1 - \epsilon/63)$, and so the lemma holds. ■

The basic idea behind the previous lemma is that we can estimate the probabilities of heavy elements well while the same cannot be said for light elements. This is intuitive as we are more likely to pick a heavy element from a data set via sampling than we are of picking a light element: the heavy element occurs more times. So using lemma 2.15 we can identify the heavy elements.

Moving on, we make use of the following fact (Batu et al., 2000, p.263): for any vector $\vec{v}$, $\|\vec{v}\|^2 \leq |\vec{v}| \cdot \|\vec{v}\|_\infty$. We use this fact to prove the theorem relating to the L_1-closeness-test.

Theorem 2.16 (Batu et al., 2000, p.263) *The L_1-closeness-test passes for input distributions $\vec{p}$ and $\vec{q}$ such that $|\vec{p} - \vec{q}| \leq \max(\frac{\epsilon^2}{32\sqrt[3]{n}}, \frac{\epsilon}{4\sqrt{n}})$, and it fails when $|\vec{p} - \vec{q}| > \epsilon$. The probability of making the wrong decision is δ. The running time is $O(\epsilon^{-4} n^{2/3} \log n \log(\frac{1}{\delta}))$.*

Proof. Suppose that the properties (1a), (1b) and (2) from lemma 2.15 hold for all i, and for both distributions $\vec{p}$ and $\vec{q}$. This will occur with probability at least $(1 - \frac{2}{n})$. That is so as the properties defined in lemma 2.15 refer to sampling from only one distribution. In our case we are sampling from both distributions so the probability is multiplied by itself as follows,

$$\begin{aligned}(1 - \frac{1}{n})(1 - \frac{1}{n}) &= 1 - \frac{1}{n} - \frac{1}{n} + \frac{1}{n^2} \\ &= 1 - \frac{2}{n} + \frac{1}{n^2} \\ &\geq 1 - \frac{2}{n}\end{aligned}$$

So as we can see from the above equations, the probability of both properties from lemma 2.15 holding is at least $(1 - \frac{2}{n})$.

Furthermore, let $S = S_p \cup S_q$. This is our set of heavy elements. According to our intial assumptions, all the heavy elements from $\vec{p}$ and $\vec{q}$ are in S and no element with weight less than $\epsilon^2 n^{-2/3}$ (in either distribution) is in S. The next step is to estimate the contribution to the L_1-norm distance between $\vec{p}$ and $\vec{q}$ by all the elements in S: let us refer to this by Δ_1. This is done by brute force in steps (5)-(6). For the light elements, let us define a similar distance variable: $\Delta_2 = |\vec{p}' - \vec{q}'|$. If there are no heavy elements, that is to say $S = \emptyset$, then $\Delta_1 = 0$ and $\vec{p} = \vec{p}'$ and $\vec{q} = \vec{q}'$.

By the deifnition of Δ_1, it is at most $|\vec{p} - \vec{q}|$: it is simply a fraction of the total distance accounted for by the heavy elements. Furthermore we can prove the following,

$$\Delta_1 \leq |\vec{p} - \vec{q}|$$

$$|\vec{p} - \vec{q}| \leq 2\Delta_1 + \Delta_2$$

As mentioned before, we estimate Δ_1 by brute force to within an additive factor of $\epsilon/9$ (this is steps (5)-(6) in the algorithm's pseudocode). The error on the i^{th} term of the sum is bounded as follows,

$$\frac{\epsilon}{63}(\max(p_i, n^{-2/3}) + \max(q_i, n^{-2/3})) \leq \frac{\epsilon}{63}(p_i + q_i + 2n^{-2/3})$$

Now sum these error terms over i. This sum will be over at most $2n^{2/3}/(1-\epsilon/63)$ elements in the set S: if all elements in out initial sample sets in step (1) are heavy, then we will have to sum the error terms over $2n^{2/3}/(1-\epsilon/63)$ elements. Thus the total error is bounded as follows,

$$\begin{aligned}\sum_{i\in S}\frac{\epsilon}{63}(p_i + q_i + 2n^{-2/3}) &\leq \frac{\epsilon}{63}(2 + 4/(1-\epsilon/63)) \\ &\leq \frac{\epsilon}{9}\end{aligned}$$

Moreover, $\max(\|\vec{p}'\|_\infty, \|\vec{q}'\|_\infty) < n^{-2/3} + n^{-1}$. So we can use the L_2-closeness-test on $\vec{p}'$ and $\vec{q}'$ with the number of samples $m = O(\epsilon^{-4}n^{2/3})$ as shown in lemma 2.13.

If $|\vec{p} - \vec{q}| < \frac{\epsilon^2}{32\sqrt[3]{n}}$, then using the aforementioned argument, so are Δ_1 and Δ_2. So the first stage of the algorithm passes. Using aforementioned fact $\|\vec{p}' - \vec{q}'\| < \frac{\epsilon}{4\sqrt{n}}$. Thus the L_2-closeness-test passes as well.

Similarly, if $|\vec{p} - \vec{q}| > \epsilon$, then either $\Delta_1 > \epsilon/4$ or $\Delta_2 > \epsilon/2$. So either the first stage fails (step (6)) or the L_2-closeness-test fails.

Moreover, the running time for the first stage is $O(\epsilon^{-2}n^{2/3}\log n)$ and the running time for the L_2-closeness-test is $O(\epsilon^{-4}n^{2/3}\log\frac{1}{\delta})$. The probability of making an error (rejecting a true null hypothesis so to speak) is bounded by δ as the algorithm makes an error either when it makes a bad estimation of Δ_1 or when the L_2-closeness-test makes an error. ■

So the closeness algorithm works in this manner, as devised by Batu et al. (2000). Without actually having to calculate any probabilities or frequencies, the closeness test can inform us whether two unknown distributions $\vec{p}$ and $\vec{q}$ are close (or similar). The only technique it uses is to take sample sets, and count the similarities between sample sets from different distributions. The test passes if there are a lot of similarities, otherwise it simply fails.

2.3 The identity algorithm

In this section, I will explain the identity algorithm devised by Batu et al. (2003). This can be seen as a modification of the closeness algorithm, the modification being that we explicitly know one of the distributions under question. Given a black-box probability distribution $\vec{p}$ over $[n]$ and an explicit probability distribution $\vec{q}$ over $[n]$, the algorithm checks whether the two are identical to one another. The main use of this test is to find out whether a given data set is distributed according to some specific distribution in our mind: $\vec{q}$. This could be a uniform, gaussian, or a geometric distribution, or any distribution we have in mind. However, during our experiments with this algorithm, I will be employing only the uniform distribution as the explicit one.

One of the main tools used in this algorithm is called bucketing. It takes an explicit probability distribution $\vec{q}$ and outputs a set of distributions that are almost uniform. Formally, we define the bucketing method as follows (Batu et al., 2003, p.4).

We are given an explicit probability distribution X over R. The bucketing method ($Bucket(X, R, \epsilon)$) produces a partition $\mathcal{R} = \{R_0, ..., R_k\}$ with,

$$k = \frac{2}{\log(1+\epsilon)} \cdot \log n$$

$$R_0 = \left\{ i \mid q_i < \frac{1}{n \log n} \right\}$$

$$R_i = \left\{ j \mid \frac{(1+\epsilon)^{i-1}}{n \log n} \leq q_j < \frac{(1+\epsilon)^i}{n \log n}, \forall i \in [k] \right\}$$

If one considers the restriction of R to any of the aforementioned buckets, then the distribution is close to uniform. The following lemma proves this.

Lemma 2.17 (Batu et al., 2003, p.4) *Let $\vec{q}$ be an explicit probability distribution over $[n]$ and let $\{R_0, ..., R_k\} = Bucket(\vec{q}, [n], \epsilon)$. Then the following hold for all $i \in [k]$,*

$$\begin{aligned} \left| \vec{q}_{|R_i} - U_{R_i} \right| &\leq \epsilon \\ \left\| \vec{q}_{|R_i} - U_{R_i} \right\|^2 &\leq \frac{\epsilon^2}{|R_i|} \end{aligned}$$

and for $k = 0$,

$$\vec{q}_{R_0} \leq \frac{1}{\log n}$$

Proof. For the case when $i = 0$, $\vec{q}_{R_0} = \sum_{j \in R_0} q_j$ and each element in the restriction R_0 has a probability value of less than $1/(n \log n)$, so the sum over these probabilities has to be at most $1/\log n$.

For $i \geq 1$, assume an arbitray non-empty set $R_i = \{1, ..., l\}$ with $q_1 \leq ... \leq q_l$. Let $\vec{r} = \vec{q}_{|R_i}$. So, $r_l/r_1 = 1 + \epsilon$. By averaging, $r_1 \leq 1/l \leq r_l$. Hence $r_l \leq (1+\epsilon) r_1 \leq (1+\epsilon)/l$. Similarly, $r_1 \geq 1/(l(1+\epsilon)) > (1-\epsilon)/l$. Thus for all $j \in [l]$, $|r_j - 1/l| \leq \epsilon/l$. Therefore, the following hold,

$$\begin{aligned} \sum_{j \in R_i} |r_j - U_{R_i}| &\leq \epsilon \\ \sum_{j \in R_i} (r_j - U_{R_i})^2 &\leq \epsilon^2/l \end{aligned}$$

■

Having described the bucketing technique, we move on to describing the algorithm itself (Batu et al., 2003). Assume we have two probability distributions, just as in the closeness algorithm, $\vec{p}$ and $\vec{q}$ over a set $[n]$. However, $\vec{q}$ is now an explicit distribution. Our task is to determine whether $|\vec{p} - \vec{q}| < \epsilon$, using as few samples from the data distributed as $\vec{p}$. This can

be done using $\tilde{O}(\sqrt{n}poly(\epsilon^{-1}))$ samples. The essence of the test is to use the aforementioned bucketing technique to produce a partition set $\mathcal{R} = \{R_0, ..., R_k\}$ and then to check whether each probability distribution over each R_i is close to uniform: we accomplish this using lemma 2.17. For each partition, we sample from the data distributed as $\vec{p}$ and check if $\vec{p}_{|R_i}$ is close to U_{R_i}. This can be done using the following theorem.

Theorem 2.18 (Batu et al., 2003, p.6) *Given a black-box probability distribution $\vec{p}$ over $[n]$, there is a test which uses $O(\sqrt{n}\epsilon^{-2}\log(1/\delta))$ samples that estimates $\|\vec{p}\|^2$ (the self-collision probability from the closeness algorithm) to within a factor of $(1 \pm \epsilon)$, with probability at least $(1 - \delta)$.*

However this theorem uses the L_2-norm. The following are a few lemmas that help us interpret the result in our desired L_1-norm.

Lemma 2.19 (Batu et al., 2003, p.11) *For any probability distribution $\vec{p}$ over R, $\|\vec{p}\|^2 - \|U_R\|^2 = \|\vec{p} - U_R\|^2$.*

This lemma simply states that the difference between the self-collision probabilities of a distribution $\vec{p}$ over R and the uniform distribution over R, is equal to the self-collision probability of the difference of the two aforementioned probability distributions over R.

Lemma 2.20 (Batu et al., 2003, p.11) *Let $\vec{p}$ and $\vec{q}$ be probability distributions over $[n]$ and let $\{R_0, ..., R_k\} = Bucket(\vec{q}, [n], \epsilon)$. For each i in $[k]$, if $\left\|\vec{p}_{|R_i}\right\|^2 \leq (1 + \epsilon^2)/|R_i|$ then $\left|\vec{p}_{|R_i} - U_{R_i}\right| \leq \epsilon$ and $\left|\vec{p}_{|R_i} - \vec{q}_{|R_i}\right| \leq 2\epsilon$.*

Proof. By Cauchy-Schwartz inequality $\left|\vec{p}_{|R_i} - U_{R_i}\right| \leq \sqrt{R_i}\left\|\vec{p}_{|R_i} - U_{R_i}\right\|$. The right hand side of the inequality, by lemma 2.19, equals the following,

$$\begin{aligned}\sqrt{R_i}(\left\|\vec{p}_{|R_i}\right\|^2 - \|U_{R_i}\|^2)^{1/2} &= \sqrt{R_i}((1+\epsilon^2)/|R_i| - 1/|R_i|)^{1/2} \\ &= \epsilon\end{aligned}$$

For the second inequality, we use lemma 2.17 and the triangle inequality as follows,

$$\begin{aligned}\left|\vec{p}_{|R_i} - \vec{q}_{|R_i}\right| &< \left|\vec{p}_{|R_i} - U_{R_i}\right| + \left|U_{R_i} - \vec{q}_{|R_i}\right| \\ &\leq 2\epsilon\end{aligned}$$

■

The identity test is given as follows (Batu et al., 2003, p.11-12),

`Identity-test(`$\vec{p}$`,`$\vec{q}$`,`n`,`ϵ`,`δ`)`
`Run the test` $O(\log(\frac{1}{\delta}))$ `times`

1. `Let` $\mathcal{R} := \{R_0, ..., R_k\} = Bucket(\vec{q}, n, \epsilon/\sqrt{2})$
2. `Let` M `be a set of` $O(\sqrt{n}\epsilon^{-2}\log n)$ `samples from` $\vec{p}$

3. `For each partition` R_i do

4. `Let` $M_i = M \cap R_i$ `(we preserve the repetitions)`
   `Let` $L_i = |M_i|$ `(we count the repetitions)`

5. `If` $\vec{q}_{|R_i} \geq \epsilon/k$, `then`

6. `If` $L_i < O(\sqrt{n}\epsilon^{-2})$, `then Fail; else Pass`

7. `Estimate self-collision probabilities` $(\left\|\vec{p}_{|R_i}\right\|^2)$ `according to theorem 2.18, using` M_i

8. `If` $\left\|\vec{p}_{|R_i}\right\|^2 > (1+\epsilon^2)/|R_i|$ `then Fail; else Pass`

9. `If` $\left|\vec{p}_{\langle\mathcal{R}\rangle} - \vec{q}_{\langle\mathcal{R}\rangle}\right| > \epsilon$, `then Fail`

10. `Else Pass`

11. `If the majority of iterations of the test Pass, then Pass, else Fail`

We will now discuss and prove the following theorem regarding the identity algorithm (Batu et al., 2003, p.12).

Theorem 2.21 *The* `Identity-test`$(\vec{p},\vec{q},n,\epsilon,\delta)$ *is such that:* (1) *if* $|\vec{p}-\vec{q}| \leq \frac{\epsilon^3}{4\sqrt{n}\log n}$, *it outputs* `Pass` *with high probability and* (2) *if* $|\vec{p}-\vec{q}| > 6\epsilon$, *it outputs* `Fail` *with constant probability. This test uses* $\tilde{O}(\sqrt{n}poly(\epsilon^{-1}))$ *samples.*

Proof. In the identity algorithm, step (9) can be done by brute force. For the $\vec{q}_{\langle\mathcal{R}\rangle}$ we sum up the probabilities of seeing an element of a restriction in the unrestricted domain. For $\vec{p}_{\langle\mathcal{R}\rangle}$ we use the M_i sets. Each M_i can be thought to estimate the $\vec{p}_{|R_i}$ and so in this manner we can calculate $\vec{p}_{|R_i}$ for each i. Then we use the following to conduct this step: if $(1-\epsilon)\vec{p}_{|R_i} \leq \vec{q}_{|R_i} \leq (1+\epsilon)\vec{p}_{|R_i}$ for all i in $[k]$, then $\left|\vec{p}_{\langle\mathcal{R}\rangle} - \vec{q}_{\langle\mathcal{R}\rangle}\right| \leq \epsilon$.

Moving on, using Chernoff bounds (theorem 2.4) on the series of random variables L_i for $i = 0$ and for i in $[k]$, we can make the probability of failure very small. However this step will fail if there is a very large difference between $\vec{p}_{|R_i}$ and $\vec{q}_{|R_i}$ for some i: this will happen when we do not find too many elements that are both in M and also in $\vec{q}_{|R_i}$.

Suppose that the algorithm outputs `Pass`. This means that for each partition R_i for which step(6)-(8) were performed (the ones where $\vec{q}_{|R_i} \geq \epsilon/k$), the following holds,

$$\left\|\vec{p}_{|R_i}\right\|^2 \leq (1+\epsilon^2)/|R_i|$$

Then via lemma 2.20, we can translate this result to the L_1-norm,

$$\left|\vec{p}_{|R_i} - \vec{q}_{|R_i}\right| \leq 2\epsilon$$

We should note that the sum $\vec{q}_{|R_i}$ over all R_i for which steps (6)-(8) were skipped is at most ϵ. Also from step (9) $\left|\vec{p}_{\langle\mathcal{R}\rangle} - \vec{q}_{\langle\mathcal{R}\rangle}\right| \leq \epsilon$. So the total difference between $\vec{p}$ and $\vec{q}$ over

these partitions is no more than 3ϵ. Adding this to the 3ϵ difference over the partitions that were not skipped in steps (6)-(8) (given by applying lemma 2.7 with $\left|\vec{p}_{|R_i} - \vec{q}_{|R_i}\right| \leq 2\epsilon$ and $\left|\vec{p}_{\langle\mathcal{R}\rangle} - \vec{q}_{\langle\mathcal{R}\rangle}\right| \leq \epsilon$), we get that $|\vec{p} - \vec{q}| \leq 6\epsilon$.

If $|\vec{p} - \vec{q}| \leq \frac{\epsilon^3}{4\sqrt{n}\log n}$, then via the definition of bucketing, step (1) will return a partition with $k = (2/\log(1+\epsilon/\sqrt{2})) \cdot \log n < (2\sqrt{2}/\epsilon) \cdot \log n$ elements. Using lemma 2.8 for all the restrictions R_i with $\vec{q}_{|R_i} \geq \epsilon/k > \epsilon^2/2\sqrt{2}\log n$), we have $\left|\vec{p}_{|R_i} - \vec{q}_{|R_i}\right| < \epsilon/(\sqrt{2n})$. In terms of the L_2-norm, this implies that $\left\|\vec{p}_{|R_i} - \vec{q}_{|R_i}\right\|^2 < \epsilon^2/(2n) < \epsilon^2/(2\left|R_i\right|)$. Since via lemma 2.17, $\left\|\vec{q}_{|R_i} - U_{R_i}\right\|^2 \leq \epsilon^2/\left|R_i\right|$, then via the triangle inequality, $\left\|\vec{p}_{|R_i} - U_{R_i}\right\|^2 \leq \left\|\vec{p}_{|R_i} - \vec{q}_{|R_i}\right\|^2 + \left\|\vec{q}_{|R_i} - U_{R_i}\right\|^2 \leq \epsilon^2/\left|R_i\right|$. So by lemma 2.19, $\left\|\vec{p}_{|R_i}\right\|^2 = \left\|\vec{p}_{|R_i} - U_{R_i}\right\|^2 + \|U_{R_i}\|^2 \leq (1+\epsilon^2)/\left|R_i\right|$. Thus the algorithm will `Pass` with high probability on all such partitions. It will also `Pass` step (9) as well. Finally, the sample complexity for the test, from step (2), is $\tilde{O}(\sqrt{n}\epsilon^{-2})$. ■

So the test begins by making a partition set using the bucketing technique. We take samples from the distribution $\vec{p}$ and for all partitions R_i we make a set of all the elements in this sample set that also occur in the partition R_i. We also make a note of the size of each intersection set.

Moving on, for all partition sets that have a probability of being picked of more than or equal to ϵ/k, we initialise our first block of tests. If the intersection set found before had a small size (lesser than $O(\sqrt{n}\epsilon^{-2})$ to be exact), then $\vec{p}_{|R_i}$ and $\vec{q}_{|R_i}$ are far apart as the partition sets under the two different distributions do not have a lot of common elements, hence the small size of the intersection set. If this is true, then the test fails. If not, it moves to what is probably contains the ethos of this test. We are attempting to reduce our problem to one of testing whether partitions are close to uniform. If they are close to uniform, then we would expect them to have a small self-collision probability. For instance, take a set that contains a 100 distinct elements and each element occurs only once. We sample from this uniformly distributed data set and obtain 20 samples. As this is a uniformly distributed data set, we would not expect to obtain very many self-collisions as each element only occurs once in the data set. We can still get a few self-collisions if our sampling leads us to pick the same element a couple of times. This is possible, however only very slightly. So with a uniformly distributed data set, one would expect the self-collision probability to be low. This is the idea that underlies step (8). If $\left\|\vec{p}_{|R_i}\right\|^2$, the self-collision probability estimated using theorem 2.18, is larger than $(1+\epsilon^2)/\left|R_i\right|$, we fail at this particular partition as it is not uniform. If it was uniform, we would expect it to pass.

Step (9) can be thought of as the second block of tests. If both blocks pass, the test passes, according to theorem 2.7. We perform this step as follows: if $(1-\epsilon)\vec{p}_{|R_i} \leq \vec{q}_{|R_i} \leq (1+\epsilon)\vec{p}_{|R_i}$ for all $i \in [k]$, then $\left|\vec{p}_{\langle\mathcal{R}\rangle} - \vec{q}_{\langle\mathcal{R}\rangle}\right| \leq \epsilon$. For each i, we use the set M_i and the set M to obtain our $\vec{p}_{|R_i}$. As $\vec{q}$ is an explicit distribution, we can easily obtain the $\vec{q}_{|R_i}$ values by simply summing up over all $j \in R_i$. Thus in this manner we conduct the identity test.

2.4 The independence algorithm

The independence algorithm relies on the closeness algorithm described earlier (Batu et al., 2000; Batu et al., 2003). Given a joint probability distribution A over $[n] \times [m]$, we simulate samples from the product of its marginal distributions, denoted by $\pi_1 A \times \pi_2 A$. So by applying

the closeness algorithm on the distributions A and $\pi_1 A \times \pi_2 A$, we can check how close the two are. If they are sufficiently close, then we have independence. Before we proceed to describe the algorithm, I would like to describe the statistical idea behind it.

Given a universal set Ω, we are given two sets A and B (Pierce, 1970, p.48). The intersection of these sets, $A \cap B$, is not empty; this implies that A and B are not empty. Using the definition of conditional probability,

$$\Pr(B \mid A) = \frac{\Pr(A \cap B)}{\Pr(A)}$$

This means that the probability of picking the set B given that we pick the set A is the ratio of the probability of picking their intersection and the probability of picking set A. In other words, when we have picked set A, the probability of picking set B is the equation above. However for our purposes, it is useful to re-arrange it as follows,

$$\Pr(A \cap B) = \Pr(B \mid A) \cdot \Pr(A)$$

This simply states that the joint probability is equal to the conditional probability of picking B given A, multiplied by the probability of picking A. If we had independence between A and B, then picking set A could give us no useful information about the probability of picking set B. So, the conditional probability of picking B given A would simply be the probability of picking B. That is,

$$\Pr(A \cap B) = \Pr(B) \cdot \Pr(A)$$

So under independence, we have that the joint probability of picking two sets, or events, is simply the product of the probability of picking one and the probability of picking the other. This is exactly what our independence algorithm attempts to do using the closeness algorithm. The following two theorems from the closeness algorithm have been modified to reflect the independence algorithm.

Theorem 2.22 (Batu et al., 2000, p.261) *Given an unknown distribution A over $[n] \times [m]$, we transform A to a single random variable $\vec{p}$ over $[k]$, and simulate samples from $\pi_1 A \times \pi_2 A$ (these are also then transformed into single random variables). Given a parameter δ, the independence test runs in time $O(\epsilon^{-4} k^{2/3} \log k \log \frac{1}{\delta})$ such that if $|\vec{p} - (\pi_1 A \times \pi_2 A)| \leq \max(\frac{\epsilon^2}{32\sqrt[3]{k}}, \frac{\epsilon}{4\sqrt{k}})$, then the closeness test outputs* `Pass` *with at least $1 - \delta$ probability and if $|\vec{p} - (\pi_1 A \times \pi_2 A)| > \epsilon$, then the test outputs* `Fail` *with at least $1 - \delta$ probability.*

As before, this theorem relies on a similar one that works in the L_2-norm.

Theorem 2.23 (Batu et al., 2000, p.261) *Given an unknown distribution A over $[n] \times [m]$, we transform A to a single random variable $\vec{p}$ over $[k]$, and simulate samples from $\pi_1 A \times \pi_2 A$ (these are also then transformed into single random variables). Given a parameter δ, the closeness test runs in time $O(\epsilon^{-4} \log \frac{1}{\delta})$ such that if $\|\vec{p} - (\pi_1 A \times \pi_2 A)\| \leq \epsilon/2$, then the closeness test outputs* `Pass` *with at least $1 - \delta$ probability and if $\|\vec{p} - (\pi_1 A \times \pi_2 A)\| > \epsilon$, then the test outputs* `Pass` *with less than δ probability.*

Regarding the transformation of pairs of random variables (i, j) to a single random variable k, we will discuss it in detail in section 4. So essentially the independence algorithm is the same as the closeness algorithm, however in order to deal with joint probability distributions we use the aforementioned certain transformation.

Moreover, to simulate a sample from $(\pi_1 A \times \pi_2 A)$, we take two samples from A. Then we take the first coordinate from the first sample, and the second coordinate from the second sample. By glueing them together, we obtain our simulated sample from $(\pi_1 A \times \pi_2 A)$.

So in this manner, we simply adapt the closeness test to test for independence.

3 The statistical tests

I will describe the workings of the statistical tests that I have chosen to compare with the algorithms. The comparison involves obtaining the sample sets used by the algorithms in making their decisions, and then using the statistical tests on these sample sets to ascertain whether the algorithm and its statistical counterpart make similar decisions.

I have chosen the family of χ^2-tests and Kolmogorov-Smirnov tests, as these are the standard statistical tests used in testing properties of distributions. The main difference between the algorithms and their statistical counterparts, is that the algorithms work only by sampling from unknown distributions, with the exception of the identity algorithm where one of the probability distributions is explicitly given. The statistical tests, on the other hand, work by computing frequencies of observed data and expected data (χ^2 tests) or by computing the observed and expected cumulative probability distributions (Kolmogorov-Smirnov tests). They then go about checking whether the relevant frequencies or cumulative probability distributions from the observed and expected data are similar in some sense. This is just the gist of their working. They have different versions for testing different properties. I will begin by discussing the χ^2 one sample test.

3.1 χ^2 one sample test

This test is the statistical counterpart of the identity algorithm (Neave and Worthington, 1988, p.80-84). We are given a set of data whose probability distribution is unknown . We take a sample from the data set and compute the frequencies of each element in that sample. Moreover, suppose we have n distinct elements in this sample set. This is our observed frequency distribution vector, denoted by Obs. Then we construct a theoretical frequency distribution vector (denoted by Theo) and check whether the two are similar in some sense.

The hypotheses for this test are given as follows.

$$\begin{aligned} H_0 &: Obs = Theo \\ H_A &: Obs \neq Theo \end{aligned}$$

The relevant statistic we need to calculate to make a decision regarding the null hypothesis is as follows.

$$\chi^2 = \sum_{i \in [n]} \frac{(Obs_i - Theo_i)^2}{Theo_i}$$

For a given level of significance α, and degress of freedom ν, where ν is defined as the size of the domain of either the Obs or Theo frequency distribution minus one, we use the χ^2-distribution table to find the critical value. We reject the null hypothesis if,

$$\chi^2 \geq \text{critical value}$$

This rejection rule is intuitive as if the vectors Obs and Theo are significantly different, then over $i \in [n]$, we expect $(Obs_i - Theo_i)$ to be large, and so ultimately we can expect a large χ^2.

One important point to note here is that in this paper we will be dealing with large amounts of data with a large number of distinct elements, so the χ^2-distribution table will not be able to provide the relevant critical values. So we will have to use a normal approxiamtion of χ^2 (iFigure, 2007a). If $\nu > 30$, then we employ the following normal approximation of χ^2,

$$z = \frac{\sqrt[3]{\frac{\chi^2}{\nu}} - \left(1 - \frac{2}{9\nu}\right)}{\sqrt{\frac{2}{9\nu}}}$$

The rejection rule for using the z-value while conducting a 2-tailed test is that for a given α, we obtain the critical from the standard normal distribution. We reject the null hypothesis if,

$$|z| > \text{critical value}$$

3.1.1 The standard normal table

I will take a slight, but important, digression here to explain the use of the standard normal table. It is rather simple. For instance, we set an α of 0.05 and conduct a χ^2 one sample test. Assume that $\nu > 30$, so we have to get a normal approximation of the test statistic as shown in section 3.1. After obtaining our test statistic, we turn our attention to obtaining the critical value with which we will compare this test statistic. From the standard normal table of percentage points (Neave and Worthington, 1988, p.370), the relevant critical value is simply 1.96 at an α of 0.05 for a 2-tailed test. If our test statistic is larger than the critical value of 1.96, then we reject the null hypothesis. Otherwise, we accept it.

This will prove useful for the χ^2 one sample and two sample tests later on. We will see that if $\nu > 30$, then the decision rules are surprisingly easy to make using the standard normal table. A point to note here is that we do not use the p-values given in the table. We simply use the percentage points.

3.2 χ^2 two sample test

This test performs the same function as the χ^2 one sample test, however the difference being that we have two data sets and we must check whether the two are significantly different from each other, rather than having one data set and checking whether that is signfiicantly different from a theoretical frequency distribution (Langley, 1971, p.269). This is the statistical counterpart of the closeness algorithm and the independence algorithm.

This test is best illustrated with an example. Suppose we would like to see whether the distribution of degree classifications for the MSc in Statistics and the MSc in Applicable Mathematics is the same. We use data from LSE statistics regarding taught masters degree calssifications (LSE, 2007). The categories of each degree's classifications are distinctions, merit, and pass. The data can be tabulated as follows in table 1 on the next page.

The cells for each row i and column j, denoted by O_{ij}, represent the observed frequency of people having the two attributes (the attribute of row i and the attribute of column j) at the same time. To compute the expected frequencies, E_{ij}, (these would be the frequencies we

Table 1: Taught Masters Degree results 2005/06.

	Degree Class			
	Distinction	Merit	Pass	Row totals
MSc Applicable Mathematics	4	10	6	20
MSc Statistics	4	4	5	13
Column totals	8	14	11	Total = 33

would like to see if the two distributions are close or similar) we add up the row and column totals, obtaining R_i and C_j. We then perform the following calculation for each row i and each column j,

$$E_{ij} = \frac{R_i \cdot C_j}{N}$$

where N would be the total number of people in both the degrees. Having calculated these expected frequencies, the calculation of the test statistic is similar to that to the χ^2 one sample test. I will use a 5% level of significance. Before we calculate the test statistic, let us set out our hypotheses,

H_0 : The distributions of degree classifications in the MSc Applicable Mathematics and the MSc Statistics degree are the same.

H_A : The distributions of degree classifications in the MSc Applicable Mathematics and the MSc Statistics degree are different.

The test statistic is as follows, where we have two rows and three columns,

$$\chi^2 = \sum_{i \in [2]} \sum_{j \in [3]} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

One thing to note here is that in the general case, the number of rows will be the same as above, however the number of columns is equal to the size of the domain of the frequency distribution. So in the general case, where n is the size of the domain of the frequency distribution, the test statistic is,

$$\chi^2 = \sum_{i \in [2]} \sum_{j \in [n]} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

Coming back to our example, the calculation of the test statistic yields a χ^2 value of 1.24. The degrees of freedom, ν, for this test is $(2-1)(3-1)$, which equals 2. Again, for the general case, ν is $(2-1)(n-1)$. So with 2 degrees of freedom, and a 5% α, we consult our χ^2-distribution table for the critical value (Neave and Worthington, 1988, p.371): it is 5.991. So our test statistic is much lower than our critical value: we accept the null hypothesis in this case. Thus, the distribution of degree classificiations between the two degree programmes does not differ significantly at the α of 5%.

Since in this paper we are dealing with large datasets with a large number of distinct elements, we will more than likely use the previously mentioned normal approximation (section 3.1) to get a z value (iFigure, 2007b). We then compare this value to a critical value (percentage point) obtained from the standard normal distribution, which depends on the level of significance we have chosen. We reject the null hypothesis if the absolute value of our test statistic is larger than the critical value.

So in this manner we use the χ^2 two sample test to check if two unknown distributions are similar.

3.3 Kolmogorov-Smirnov one sample test

The χ^2 tests relied on observed and expected frequencies. If both were significantly different, then we rejected the null hypothesis. The Kolmogorov-Smirnov one sample test relies on computing the cumulative probability distribution of the sample set taken from the data, and comparing it with a theoretical cumulative probability distribution of our choice. If both are sufficiently close, we accept the null hypothesis. This test is the counterpart of the identity test.

The Kolmogorov-Smirnov one sample test (Neave and Worthington, 1988, p.89-93) relies on two statistics. $F_0(x)$ is the theoretical (or hypothesised) cumulative probability distribution. Similarly, $F_n(x)$ is the cumulative probability distribution of our sample. x is an element in the sample set, while n is the size of the sample set. Formally, they are defined as,

$$\begin{aligned} F_0(x) &= \text{Hypothesised proportion of sample observations less than or equal to x.} \\ F_n(x) &= \text{Actual proportion of sample observations less than or equal to x.} \end{aligned}$$

As before, we chose a level of significance α, and set out our hypotheses.

$$\begin{aligned} H_0 &: F_0(x) = F_n(x) \\ H_A &: F_0(x) \neq F_n(x) \end{aligned}$$

The test statistic is simply the maximum absolute difference between $F_0(x)$ and $F_n(x)$,

$$D_n = \max |F_n(x) - F_0(x)|$$

We obtain our critical values from the Kolmogorov-Smirnov one-sample test distribution table. This value depends on our n and α. As before, if our test statistic is larger or equal to the critical value, we reject the null hypothesis.

For dealing with large data sets, the Kolmogorov-Smirnov one sample distribution table provides critical values that depend on n and α in the following manner (ERI, 2007). Since we will be using the 5% level of significance α, the critical value associated with this is,

$$\text{Critical value} = \frac{1.36}{\sqrt{n}}$$

For different α, we will have different numbers instead of 1.36. Thus in this manner we calculate our relevant critical value. If the our test statistic is greater than the critical value, we reject the null hypothesis.

3.4 Kolmogorov-Smirnov two sample test

This test is a counterpart to both the closeness and independence algorithms. It relies on computing the cumulative probability distributions of sample sets obtained from our two datasets, and checking whether they are significantly similar or not (Neave and Worthington, 1988, p.149-152; Durbin, 1973, p.39-40).

As in the one sample version, we obtain the following: $F_n(x)$ and $G_m(x)$ where x is an element in the sample set; n is the size of the first sample set, while m is the size of the second sample set. These are the cumulative distributions of the two sample sets in question. Formally,

$$\begin{aligned} F_n(x) &= \text{Proportion of sample one observations less than or equal to x.} \\ G_m(x) &= \text{Proportion of sample two observations less than or equal to x.} \end{aligned}$$

As before, we chose a level of significance α, and set out our hypotheses.

$$\begin{aligned} H_0 &: F_n(x) = G_m(x) \\ H_A &: F_n(x) \neq G_m(x) \end{aligned}$$

The calculation of this test statistic is a bit more involved (Durbin, 1973, p.39-40). Combine the two sample sets in an ascending manner: $x_1 \leq ... \leq x_{n+m}$. Let members of the first sample set be denoted by a and members of the second sample set be denoted by b. As we run through this combined set, each time we meet an a the difference between the two cumulative distributions increases by $1/n$: that is to say $(F_n(x) - G_m(x))$ increases by $1/n$. Similarly, each time we encounter a b, $(F_n(x) - G_m(x))$ decreases by $1/m$. Denote n_i as the number of a's which are less than or equal to the i^{th} observation in this combined set. So according to this discussion,

$$\begin{aligned} F_n(x) - G_m(x) &= \frac{n_i}{n} - \frac{i - n_i}{m} \\ F_n(x) - G_m(x) &= \frac{n+m}{m}\left(\frac{n_i}{n} - \frac{i}{n+m}\right) \end{aligned}$$

We use this equation as follows,

$$\begin{aligned} D^+_{nm} &= \frac{n+m}{m} \max_{i \in [n+m]} \left(\frac{n_i}{n} - \frac{i}{n+m}\right) \\ D^-_{nm} &= \frac{n+m}{m} \max_{i \in [n+m]} \left(\frac{i}{n+m} - \frac{n_i}{n}\right) \\ D_{nm} &= \max(D^+_{nm}, D^-_{nm}) \end{aligned}$$

Our test statistic is D_{nm}. We then obtain our critical value from the Kolmogorov-Smirnov two sample test distribution table, according to n and m. If our test statistic is larger or equal to the critical value, we reject the null hypothesis.

As before, since we are dealing with large datasets from which large sample sets will be obtained, we use the following critical value at the 5% level of significance α (Wessel, 2003),

$$\text{Critical value} = 1.36 \cdot \sqrt{\frac{n+m}{n \cdot m}}$$

Once again, we reject the null hypothesis if our test statistic is larger than or equal to our critical value. One point to note here is that we will encounter problems when trying to compute our critical values for this test using Java. Our sample set will become increasingly large as the distance parameter ϵ increases, so there will come a point when Java will output `NaN`: not a number. At this point our test would have become useless/void. In some sense this hints at the inability of these non-parametric tests to deal with large data sets.

4 Java implementation

I have programmed all of these algorithms and statistical tests in the Java programming language. Moreover, I have also used Java to create artificial data sets for the experiments - also conducted by a simple piece of Java code - in section 5. In this section I will discuss how I went about doing this, and what the Java code accomplishes. The relevant files are Methods.java, Closeness.java, Identity.java, Independence.java, CS1.java, CS2.java, KS1.java, KS2.java, Data.java, CloseExp.java, IdExp.java and IndExp.java.

The Methods.java file contains important static methods that will be used throughout our algorithms, statistical tests, data generation and the experiments. The Closeness.java file runs the Closeness algorithm via data input from text files. It outputs the relevant sample sets used to come to a conclusion so that we can re-use these sample sets in our relevant statistical tests. Similarly, Identity.java and Independence.java run the Identity and Independence algorithms respectively. The CS1.java file is an implementation of the χ^2 one sample test, which uses the aforementioned sample sets outputed from the Identity algorithm. It gives the relevant test statistic to be compared with percentage points from the standard normal table. Similarly, the CS2.java file implements the χ^2 two sample test for comparison with the Closeness and Independence algorithm. The KS1.java file implements the Kolmogorov-Smirnov one sample test and the KS2.java file implements the Kolmogorov-Smirnov two sample test. The former is used to used for comparison with the identity algorithm, while the latter is used for comparison with the closeness and independence algorithms.

Data generation is done by a simple file called Data.java. It basically works on a set of 20,000 distinct elements (keys), generating large data sets with various distributions of our choice. For instance, I generate one data set where the first 25% of keys occur more frequently than the rest of the keys. So one can imagine that we can create quite a few interesting data sets of our choice.

Furthermore, we have the relevant experiment files: CloseExp.java runs the Closeness algorithm for the data set in question and outputs the result of the algorithm, along with the results of the χ^2 two sample test and Kolmogorov-Smirnov two sample test. Similarly, IdExp.java runs the Identity algorithm and the χ^2 one sample test and Kolmogorov-Smirnov one sample test, outputing the relevant answers from the algorithm and statistical tests. IndExp.java does a similar job.

In this section I will begin by briefly explaining the methods in the Methods.java file. I will then move onto the algorithms, statistical tests, data, and finally the experiments.

4.1 Methods

To run the statistical tests, algorithms, and to generate data and run the experiments on them, I employed a series of static methods that were used repeatedly. The following consist of brief descriptions of each method and its functions.

Read_Data_int(String Filename) This method takes in as input the filename of the data file, and outputs that data as an `int[]` array. It relies on a `LinkedList` to pick up the lines it reads from the file. For this purpose it creates a `FileReader` object with the

filename as the input. It then creates a `LineNumberReader` with the aforementioned `FileReader` object as an input. It uses the `readLine` method to add data to the `LinkList` as long as there are lines to be read (this is checked by the `ready()` method). Finally, the contents of the `LinkedList` are converted to a `String[]` array, and then to an `int[]` array via the `Integer.parseInt()` method. Similar methods exists for reading data and outputing it as a `long[]` array or as a `float[]` array or as a `String[]` array: `Read_Data_long(String Filename)`, `Read_Data_long(String Filename)` and `Read_Data_string(String Filename)`(Google groups forum, 2007).

Freq_index_int(int[] Data, int i) This method takes in as input a Data array and index i. It uses a `while` and a `for` loop to check how many times the element at that index i occurs in the Data array. It returns the frequency of the element at index i as an `int` value. A similar method exists for finding the frequency of the element at an index i in a `long[]` array: `Freq_index_long(long[] Data, int i)`.

Freq_element_int(int[] Data, int x) This method takes in as input a Data array and an element x. It uses a `for` loop to check how many times the element occurs in the Data array. It returns the frequency of the element as an `int` value. A similar method exist for finding the frequency of the element in a `long[]` array: `Freq_element_long(long[] Data, int x)`.

Frequency_vector_int(int[] Data) This method takes in as input an `int[]` type Data array. It converts it to an `Integer[]` array for use in a `TreeMap`. The `TreeMap` simply takes an element, runs along through the Data, and computes the f requency of that element. It does this for each distinct element in the Data array. Finally, we get our frequency values by using the `values().toArray()` method, which returns it as an `Integer[]` array. This is simply converted to an `int[]` array, which is returned as the output. So this method simply allows us to find the frequency of each distinct element in a Data array. Similar methods exist for finding the frequency vector of Data, which is in the form of a `float[]` and a `String[]` array: `Frequency_vector_float(float[] Data)` and `Frequency_vector_string(String[] Data)`(Sun, 2004).

Probability_vector_int(int[] Data) This method simply normalises the output of the previous method and returns a `float[]` array of probability values. A similar method exists for dealing with `float[]` Data: `Probability_vector_float(float[] Data)`.

Sample(int[] Data, int m) This method takes in as input a Data array of `int[]` type, and the number of samples needed, m, as an `int` value. It uses a `Random()` object called `generator` to uniformly chose a number from 0 to (`Data.length` - 1). The element at this index of the Data array is then assigned to our sample set array, which has a size of m. So this is done a total of m times until we have our sample set. The sample set is then output as an `int[]` array. A similar method is used for getting samples from a Data array ot type `long[]` (specifically used for the independence test): `Sample_for_ind(long[] Data, int m)`.

Sample_light(int[] Data,int[] S,int m) This method is a variation of the sampling method above, and is used in the closeness algorithm. In the `Sample()` method above, it uses a `Random()` object, called generator, to uniformly chose a number from 0 to (`Data.length` - 1). The Data element at this index is then compared with elements in the array S. If this Data element is not in S, we assign it to our sample set. If it is in S, then we uniformly chose a random number from the set $[n]$, where n would be the number of distinct elements in the Data array, and assign this to the sample set. After this is done for m times, we output the sample set as an `int[]` array. Similar methods are used in the independence test, however they work with a Data array and a S array of `long[]` type: `Sample_light_for_ind_1(long[] Data, long[] S, int[] m)` and `Sample_light_for_ind_2(long[] Data, long[] S, int[] m)`.

Self_col_prob(int[] x) This method is used to get the estimations of self-collision probabilities using input `int[]` x. It starts out by creating an `int` called Ans (our self-collision probability). Then it creates two other `int` variables called i and k. It initialises i as 0, while it initialises k as $(i+1)$. It starts a `for` loop from the initial value of i till `x.length`, and within this it nests another `for` loop, which it starts from the initial value of k till `x.length`. For every i, it runs through all the relevant values of k, checking whether the element at this index i is the same as the elements encountered at the various values of k. If a similar element is found, we have a self-collision, and so the Ans variable is increased by 1. Otherwise it stays the same. It simply outputs this Ans value once the loops have been completed. A similar method exist for dealing with `long[]` type arrays (specifically used in the independence algorithm): `Self_col_prob_for_ind(long[] x)`.

Col_prob(int[] x) This method is used to obtain estimations of collision probabilities. Our collision probability estimation is denoted by Ans. It takes in as input two `int[]` arrays x and y. It starts a for loop running from 0 till `x.length` (index i), and within this it nests another for loop running from 0 till `y.length` (index j). For each value of i it runs through all the values of j, and if the element of x at index i is the same as any of the elements of y at the index values of j, then we increase Ans by 1. Otherwise it remains the same. Finally we output this Ans as our collision probability once all the loops have concluded. A similar method exists for dealing with inputs of the `long[]` type (for the independence algorithm): `Col_prob_for_ind(long[] x, long[] y)`.

Bucket(float[] X, float e, int n, int i) This method, as its name implies, is used in the Identity algorithm. It takes in as input the (explicit) probability distribution X, the distance parameter e (ϵ), the size of the domain of X denoted by n, and the number (index) of the bucket we are trying to obtain. For the 0^{th} bucket (when $i = 0$), using the description of the bucketing tool in section 2.3, we obtain the index of those elements that satisfy this description. Similarly, for the other buckets (when we take an $i \in [k]$), we obtain the index of all those elements that satisfy the definition of bucketing (section 2.3). These index numbers are added to a `List<Integer>`, which is converted to an `Integer[]`, and then an `int[]` array and passed as output (Sun Java Tutorials, 2007).

Intersect(int[] M, int[] X) As its name implies, it is used to find the common elements

in the two input arrays `int[]` M and `int[]` X. It begins by converting both `int[]` arrays to `Integer[]` arrays, and then converting these `Integer[]` arrays to two different `List<Integer>` objects. If `M.length` is less than `X.length`, then we use the `c.retainAll(z)` method to obtain the common elements, which are then passed onto an `Integer[]` array via the `c.toArray(new Integer[0])` method. On the other hand, if `M.length` is more than `X.length`, then we use the `z.retainAll(c)` method, convert it to an `Integer[]` array via `z.toArray(new Integer[0])` method. Finally these `Integer[]` arrays of common elements are converted to `int[]` arrays, and passed as output (Sun Java Tutorials, 2007; Sun Developer Network, 2006; Java in a Nutshell, 2005, p.229-230).

Uncommon(int[] A, int[] B) This method is used to find all those elements of A that do not occur in B. It does this by creating a list of all the elements in A and B: Common. Then it creates a list of elements of only A: CA. Using the `Common.removeAll(CA)` method, we remove all the elements that occur in CA from Common. So we are left with only those elements of A that do not occur in B. This is then converted to an `Integer[]` array, and finally to an `int[]` array, which is then passed as output.

Keys_table_int(int[] Data) This method is used to obtain all the distinct elements in a data set. It uses the exact same method as the `Frequency_vector_int(int[] Data)` method, however instead of obtaining the frequencies of each distinct element, we obtain the distinct elements themselves, using the `keySet().toArray()` method. Similar methods are used for `long[]`, `float[]` and `String[]` arrays: `Keys_table_long(long[] Data)`, `Keys_table_float(float[] Data)` and `Keys_table_string(String[] Data)` (Sun, 2004).

Cumulative_freq(int[] Data) This method is used to compute the cumulative frequency distribution of a data set: the cumulative frequency of the i^{th} element is the number of elements less than or equal to i. It starts by computing the frequency distribution of the data set via the Frequency_vector_int(int[] Data) method: denoted by C_freq. Then it creates an int[] array called Ans, which will be our cumulative frequency distribution. The length of Ans is the same as C_freq. Obviously, we assign C_freq[0] to Ans[0]. Then we run a for loop from the index 1 to (Ans.length - 1), and at each index i we assign the sum of C_freq[i] and Ans[$i-1$] to Ans[i]. By doing this we obtain our cumulative frequency distribution.

CDF(int[] C_freq,int n) This method simply normalises the output of the `Cumulative_freq(int[] Data)` method above to obain the cumulative distribution function. Its main use will be in the Kolmogorov-Smirnov tests.

Max(float[] Data) This method simply outputs the element with maximum size in a `float[]` type Data array. It starts by setting the maximum as the first element in the Data set. It then runs a `for` loop through all the rest of the elements, and if an element is found that is bigger than our intial maximum then it assigns it as the new maximum. It returns this maximum as a `float` type data element (Swartz, 2006).

Joint_var_int(int[] A, int[] B) This method is simply used to take in as input two `int[]` arrays called A and B. It creates a 2-dimensional arrays, the length of whose first dimension is equal to `A.length` (or `B.length` as we assume that both A and B have the same number of elements), and whose second dimension's length is 2. It runs a `for` loop through this 2-dimesional array, and at each index i it assigns the element at this index i from A to the first position in the array's second dimension, while it assigns the element at index i from B to the second position in the array's second dimension. Similar methods exist for dealing with `float[]` and `String[]` types: `Joint_var_float(float[] A, float[] B)` and `Joint_var_string(String[] A, String[] B)`.

R_C_totals(int[][][] A, String Dimension) This method is used in the χ^2 two sample test. It is used to compute the row and column totals for obtaining expected frequencies. The `String` Dimension tells it which totals to output. If the Dimenion is "Rows", it outputs row totals, otherwise it outputs column totals. In calculating the totals, it simply runs `for` loops along the relevant dimensions of the `int[][][]` A array and sums up the elements in each position to obtain our totals.

conversion(int[][] A) This method is used for the independence algorithm for converting pairs of `int` type numbers to a single `long` type number. It runs a `for` loop through the first dimension of the array A. The first dimension of the array A can be thought of as the length of the data set, while at each index i in this first dimension, we obtain 2 coordinates that are kept in the second dimension. We multiply the second coordinate with 2^{32} and add this to the first coordinate, thereby obtaining a unique `long` type number that represents the pairs `int` type (Clarkson University CS658, 2006).

joint_sample(int[][] X, int m) This method outputs a sample set of length m from `int[][]` X. It does this by creating a `Random` object, called generator, and choosing a random number r from 0 till `(X.length - 1)`. The pair of `int` type at this index r in X is the assigned to our sample set. This is done for a total of m times until we have our sample set.

joint_int_sample(int[][] X, int m) This method simulates sampling from the product of the marginal distributions of a joint variable: `int[][]` X. To get a single sample from the aforementioned distribution, we take two samples from `int[][]` via the `joint_sample()` method above. Then we take the first coordinate from the first sample and the second coordinate from the second sample to and combine them to simulate a sample obtained from the product of the marginals. As before, this is done for a total of m times until we have our sample set.

Output_file_int(int[] Data, String File_name) As its name implies, this outputs the `int[]` array called Data as File_name.txt. It does this by creating a `File` object whose input is the computer's home directory and the File_name we want to give to the file. Then it creates a `PrintWriter` object whose input is a `FileWriter` object, whose input in turn is the previously created `File` object. This `PrintWriter` object, called out, is used to print the elements in the Data, each on new lines, to File_name.txt. Similar methods exist for outputing a `long[],float[]` and `String[]` type arrays as a file:

`Output_file_long(long[] Data, String File_name)`, `Output_file_float(float[] Data, String File_name)`,`Output_file_string(String[] Data, String File_name )`(Flanagan, 2005, p.255).

4.2 Algorithms

In this section I will give a brief overview of the implementation of the algorithms in this paper. We will begin with a discussion of the constructors used in these algorithms. This discussion is also relevant for the constructors of the statistical tests as their constructors are also based on similar principles. Then I discuss the implementation of the closeness algorithm, followed by the identity and the independence algorithm.

4.2.1 Constructors

I will briefly discuss the similarity in constructors of the three algorithms. This discussion is also relevant for the statistical tests. All three algorithms and four statistical tests have constructors designed to take in different type of data arrays and code them to `int` type data arrays.

All three algorithms have two similar instance fields: d (error-probability parameter) and e (closeness parameter). They are of `float` type. The closeness algorithm has two further instance fields, both of `int[]` type: A and B. The identity algorithm will have one `int[]` type instance field called A, while the other one will be of type `float[]` and be denoted by Y (this is our explicit distribution). Finally the independence algorithm will only have one `int[][]` type data array.

The closeness algorithm has three constructors: one taking in two `int[]` type arrays as data, the other taking two `float[]` type arrays as data, and the final one taking two `String[]` type arrays as data. All of them take in the aforementioned `float` variables d and e as well. These are passed on to the instance fields as themselves. However the two constructors, one with `float[]` type data and the other with `String[]` type data, manipulate the data arrays before passing them on to the relevant instance fields A and B.

The essence of this manipulation is that they combine the data elements in both arrays. Using the `Keys_table_float()` (or the `Keys_table_string()` method if we are dealing with the constructor that takes in `String[]` type arrays), it obtains an array of keys: all the distinct elements in the combined data set. These keys are then mapped on to an `int` type variable. For instance if we have the set of keys {a,b,c,d}, the mapping would lead to a set of keys of `int` type {0,1,2,3}. After we obtain our numerical key set, so to speak, we transform the `float[]` type (or `String[]` type as the case may be) by using this numerical key set. For instance if our data set was {a,a,a,b,b,c}, the resulting `int[]` type data set would be {0,0,0,1,1,2}. So in this manner we code `float` type and `String` type data to our `int` type data. This is necessary as our class and instance methods use `int` type data.

The identity and the independence algorithms have similar constructors. However, the difference in the independence algorithm is that it takes in joint variables: 2-dimensional array, where the first dimension acts as an index, and the second one holds the first and second coordinates of the variable. To convert `float[][]` type and `String[][]` type data,

we use a similar transformation shown before, however instead of combining data from two different 1-dimensional array (or only one 1-dimensional array in the case of the identity algorithm), we combine all the first and second coordinates in the second dimension of the array. Then we use a similar procedure as before to find the key set, map them onto a simple `int` variable, map these back on to the data, and the result will be a `int[][]` type variable. The same procedures apply for the statistical tests. So there is no need to repeat my explanations.

4.2.2 Closeness algorithm

I will give a brief overview of the implementation of the closeness algorithm in Batu et al. (2000). It begins by creating a `Closeness()` object, and applying the `Closeness()` instance method onto it. The `Closeness()` method then begins by obtaining the value of n: the number of distinct elements in our data sets. Using this n, we determine the number of samples to be taken in the `Closeness()` method, M, and the number of samples to be taken in the `L2_Test(...)` method, m. A point to note here is that the `L2_Test(...)` is a static method, as opposed to `Closeness()` being an instance method. I will denote static methods with three dots between their brackets, while non-static (instance) methods will not have these dots.

We take M samples, using the `Sample(...)` method, from A and B. Then employing the `Freq_index_int(...)` method we filter out the light elements from these sample sets, and add the the heavy elements to two different `List<Integer>` objects: `S_P_temp` and `S_Q_temp`. These are converted to `Integer[]` and then `int[]` type arrays: `S_P` and `S_Q`. If these list objects are empty, and we check for this via the `isEmpty()` method, then we do an `L2_Test(...)`, the "Normal" way. Doing the `L2_Test()` the "Normal" way would involve simply taking samples from the origional distributions. It will become clear later on why we make a distinction here between doing the `L2_Test(...)` in a normal way, as opposed to doing it in a "Light" way.

On the other hand, if `S_P_temp` and `S_Q_temp` are not empty, then we obtain a set of common elements in `S_P` and `S_Q`, compute the differences in frequencies between these common elements in the sets `S_P` and `S_Q` using the `Freq_element_int(...)` method and sum up the differences. If the sum is greater then $\frac{\epsilon M}{8}$, then the test has failed. Otherwise, we run the `L2_Test(...)` in the "Light" manner. Doing the `L2_Test(...)` in the "Light" manner involves a very different type of sampling, using the `Sample_light(...)` method.

So now we arrive at the `L2_Test(...)`. I will explain the excecution of this test in both the "Normal" and the "Light" manner. As mentioned before, we execute the test in the former manner, when the set of heavy elements (`S_P_temp` and `S_Q_temp`) are empty. It is run a $\log(1/\delta)$ number of times: the majority result is then output as the ultimate result.

For each iteration of the test, we take 4 sets of samples via the `Sample(...)` method. `F_p` and `Q_p` contain m elements, and are sampled from the data distributed as $\vec{p}$, which is A. Similarly, `F_q` and `Q_q` also contain m elements, however they are sampled from the data distributed as $\vec{q}$, which is B.

Moving on, we apply the `Self_col_prob(...)` method on the sets `F_p` and `F_q`, to obtain estimates of the self-collision probability. We use the sets `Q_p` and `Q_q` in the `Col_prob(...)`

method to obtain estimates of the collision probability. These probabilities are then scaled appropriately so as to be compared with each other. The results of this scaling are the `int` variables r and s. If the difference between r and s is greater than $\frac{m^2\epsilon^2}{2}$, then our test fails as the self-collision probabilities and the collision probabilities have been proven to be significantly different. If both data sets A and B were distributed similarly, we would expect the self-collision probabilities and collision probabilities to be quite similar. As mentioned earlier, this method is run for a number of times, and the majority result is output as the ultimate result.

Now let us discuss the case when the set of heavy elements are not empty, and we have to use the `L2_Test(...)` in the "Light" manner. The main difference (in fact the only difference) is how we sample from $\vec{p}$ and $\vec{q}$. Using a different sampling method, called `Sample_light(...)`, we obtain our required sample sets. However, this sampling works differently than our previous sampling method. In this method, we sample an element from the relevant distribution, lets say $\vec{p}$. This element is then checked against all elements in the set `S_P`, to see if it is contained there. If it is in `S_P`, it means that it is a heavy element, and we discard it. After discarding it, we simply chose an element uniformly from $[n]$. On the other hand, if it is not contained in `S_P`, we assign this element to our new sample set. We sample from $\vec{q}$ in a similar manner. As before, we apply the `Self_col_prob(...)` and `Col_prob(...)` methods on these sample sets to obtain our self-collision and collision probability estimates. We scale them, find their difference and if this is greater than $\frac{\epsilon M}{8}$ the test fails. The sample sets are also outputted using the `Output_file_int(...)` method, and they will be used by our statistical tests. Thus in this manner we run the Closeness algorithm.

4.2.3 Identity algorithm

The identity algorithm begins by creating an `Identity()` object. We then simply apply the instance method `Identity()` to run our test. I will now give an overview of the implementation of the identity algorithm (Batu et al., 2003).

As before we run the test for a $\log(1/\delta)$ number of times, outputing the majority result. This is done to get high confidence in our results, as after we are simply using a sampling algorithm to arrive at our answers. For instance we might pick a sample that does not represent the distribution we sampled from properly. We begin the `Identity()` method by obtaining a set of keys of the Data set (our black-box). The next step is to compute how many buckets we need given our distance parameter ϵ and number of distinct elements (this is simply the length of the array of keys obtained earlier).

The most important step is to apply the `Bucket(...)` method the required number of times: $k+1$. We then define the following arrays. An `int[][]` array called MM: intersection of our sample set M and a particular partition. An `int[]` array called L: size of the first dimension of the MM arrays (basically to check whether the intersection set is large or small). Three `float[]` arrays called x: self-collision probabilities of the sets contained at each position in the first dimension of MM; xx: probabilities of picking restrictions from the partition set over the black-box distribution; and y: probabilities of picking restrictions from the partition set over the explicit distribution.

We sample $\sqrt{n} \cdot \epsilon^{-2} \cdot \log(n)$ elements from our black-box, and denote it by a `int[]` array

called M. The next steps are done for each and every bucket. Since the buckets contain only the index number of the elements that satisfy the properties of being in that bucket, we run through the keys set to obtain the actual elements themselves and assign the relevant key to the relevant index. We denote this new set by a `int[]` array called temp. Using the `Intersect(...)` method, we obtain the elements that are common to both our sample set M, and our set of elements in the bucket, temp. This is denoted by MM[i], where i is the number of the bucket, and the second dimension of the MM[i] array contain the intersection set. We then obtain the length of MM[i]: L[i]. Furthemore, we apply the `Self_col_prob(...)` method on MM[i], and then find the probability of picking this restriction MM[i] from the partition set over the black-box distribution. The former is x[i], while the latter is xx[i]. In obtaining the probability of picking a particular restriction, since Y is an explicit distribution, it is easy to find this probability: this is y[i]. y[i] is simply found by summing up the probabilities of picking all the elements (from the explicit distribution) that occur in the restriction.

The next step can be seen as two blocks of tests that determine our result. The first block of tests run steps (5)-(8) of the algorithm. For each bucket, we check whether y[i] is greater or equal to $\frac{\epsilon}{k}$. If it is so, then we run a couple of more checks to get a result. At the first check, if L[i] is less than $\sqrt{n} \cdot \epsilon^{-2}$, the result of the first block of tests fails. The intersection set is simply too small so the two distributions can not be very identical. If it is greater or equal to $\sqrt{n} \cdot \epsilon^{-2}$ then we check whether x[i] is greater than $(1 + \epsilon^2)/|R_i|$. If it is so, then the test fails, otherwise it passes. If we pass for every bucket, or fail for every bucket, then the first block of tests passes, or fails, as the case maybe: this result is denoted by Ans_1. The second block of tests simply perform step(9) of the identity algorithm. It does this as follows. For each bucket, it check whether y[i] is less than (1 - e)(xx[i]) or whether y[i] is greater than (1 + e)(xx[i]). If either case is true then the iteration fails. If the iteration fails for all buckets, then the answer to the second block of tests also fails. The converse is true.

If both blocks of tests fail, then the algorithm fails altogether. Needless to say, that if both blocks pass, then the algorithm passes altogether. The sample set used in this test is also outputted using the `Output_file_int(...)` method. It will be used by our statistical tests. Thus in this manner we run our identity test.

4.2.4 Independence algorithm

The independence algorithm (Batu et al., 2003; Batu et al., 2000) is basically an application of the closeness algorithm on a different type of data, so I will not repeat my discussion in full. However I will point out the places in which independence algorithm is different from the closeness algorithm.

The main difference in the independence algorithm, compared with the closeness algrorithm, is that it takes in a joint variable (i, j), with $i \in [n]$ and $j \in [m]$. We have to be able to convert these pairs of values into single variables in order to apply the closeness test. As mentioned in section 4.1, we use a method called `conversion(...)` that takes these pairs of values, and outputs a single value that uniquely represents this pair; any other pair would have a different value. Our pairs of values are of `int` type (32-bit whole numbers), while the resulting single value is of `long` type (64-bit whole numbers). So the independence algorithm

works on `long[]` type data as opposed to the `int[]` type used by the closeness algorithm.

However, the structure of both is the the same. When the `Ind()` method is applied to an `Independence()` object, the sampling procedures are different. Firstly, the sample sets are of `long[]` type. Furthermore, our $\vec{p}$ represents the joint distribution, while our $\vec{q}$ would represent the product of the marginal distributions. We take samples from the joint distribution via the `joint_sample(...)` method, for a total of M samples. These are then converted via the `conversion(...)` method to single variables. For sampling from $\vec{q}$, we use the `joint_ind_sample(...)` method, which simulates joint samples from the product of the marginal distributions. As before, these are converted to single variables via the `conversion(...)` method.

Just as in the closeness test, if our sets of heavy elements `S_P_temp` and `S_Q_temp` are empty, then we do the `L2_Test_for_ind(...)` the "Normal" way, otherwise we do it the "Light" way. For the "Normal" way, sampling procedures to obtain `F_p`, `F_q`, `Q_p` and `Q_q` are the same as in the `Ind()` instance method as defined above. We also have defined separate methods to find estimates of the self-collision and the collision probability (section 4.1) that can deal with `long[]` type arrays: `Self_col_prob_for_ind(...)` and `Col_prob_for_ind(...)`.

For the "Light" way, there are two different sampling methods that help us obtain our sample sets: `Sample_heavy_for_Ind_1(...)` and `Sample_heavy_for_Ind_2(...)`. The two methods are similar, yet differ in where they take their initial sample from. The former method samples from the joint distribution using the `joint_sample(...)` method, while the latter method simulates samples from the product of the marginal distributions using the `joint_ind_sample(...)` method. In both cases, we check whether the samples we obtain are contained in the set of heavy elements. If they are we discard them and samply uniformly from $[n]$, where n is the number of distinct elements in our distributions. On the other hand, if the samples we obtain are not in the set of heavy elements, then we assign it to our sample set. After obtaining these sample sets, `F_p`, `F_q`, `Q_p`, and `Q_q` in this manner, we then simply follow the same procedures as in the closeness algorithm, using the `Self_col_prob_for_ind(...)` and `Col_prob_for_ind(...)` method to obtain our relevant estimates. These are then scaled properly and compared as before. The sample sets are also outputted using the `Output_file_long(...)` method, and they will be used by our statistical tests. Thus our independence algorithm runs in this manner.

4.3 Statistical tests

In this section I will briefly discuss the implementation of the statistical tests we have used in this paper. The constructors for these statistical tests are similar to the ones for the algorithms, in the sense that they have the ability to code different data inputs to int type data.

4.3.1 χ^2 one sample test

This test creates a `CS1()` (Chi-Square 1 sample test; Neave and Worthington, 1988, p.80-84) object, upon which the `CS1()` instance method is operated. This method simply computes the χ^2 test statistics as defined in section 3.1: it calculates the difference between Obs[i] and

Theo[i], squares it, divide by Theo[i], and finally sums it all up over all i using a `for` loop. It then calculates a normal approximation of the test statistic, which is output instead of the actual χ^2 test statistic as we will be using large data sets with a large number of distinct elements (iFigure, 2007a).

4.3.2 χ^2 two sample test

This test creates a `CS2()` (Chi-square 2 sample test; Langley, 1971, p.269) object, upon which the `CS2()` method is invoked. This method, as before, computes the relevant χ^2 statistic, then outputs its normal approximation.

Before it begins to compute the relevant statistic, we must take care of one hazard. The input to this test will be the sample sets used in the closeness and the independence algorithms. It is possible that these two sample sets might not contain the same number of distinct elements, or perhaps they might even contain completely different elements; one set might contain {1,2,3} while the other might be {4,5,6}. However the χ^2 must work on the premise that the two sample sets have the same number of categories (distinct elements) and each data set has the same set of categories. So in our previous example, both sample sets ideally ought to contain {1,2,3,4,5,6}. So before we begin to compute the test statistic we have to overcome this problem.

We start by obtaining the set of elements that are in one data set while not in the other. This is done using the `Uncommon(...)` method. We perform this two times. Firstly to obtain the elements that are in A but not in B (denote this set by an `int[]` array x) while secondly to obtain the elements that are in B but not in A (denote this set by an `int[]` array x). If both sets have length zero, then both A and B have the same categories. However if either (or both) set is not empty, then we perform a certain operation to obtain the correct frequency distributions of the two data sets.

The operation involves using the `Freq_element_int(...)` method. We firstly combine all the elements in A and B, and obtain a set of keys from this combined table. Now we apply the `Freq_element_int(...)` method on each data set A and B, to find the frequencies of all the elements in the set of combined keys. For instance, A is a data set {1,2,3} while B is simply {2,3}. So our combined keys set would be {1,2,3}. Now we apply the `Freq_element_int(...)` method on A and B and check how many times the elements in the combined keys set ({1,2,3}) occur in each data set. So we can see that the element 1 does not occur at all in the set B, so it will have a frequency of 0. The frequency vector of A will be {1,1,1} while that of B will be {0,1,1}. In this manner we ensure that each data set that we employ in our `CS2()` method has the same categories; some categories might have a frequency of zero, but that information is still important as well. Besides, if the frequency distributions of the two data sets are not of the same length, then we get a `ArrayIndexOutOfBoundsException`. If we are lucky and the data sets A and B have the same categories, then we can simply employ the `Frequency_vector_int(...)` method without having to run the previously described operation.

The next step is to create the array of observed frequencies, using the previouslly obtained frequency distributions (denoted by `Frq_A` and `Frq_B`). We create a three-dimensional `int[][][]` array called Joint. The first dimension is the number of rows. As this is a two

sample test, the number of rows will always be 2. The second dimension is the number of columns, which is simply the size of the domain of any one of our frequency distributions. Finally the third dimension contains our observed frequencies for each row i and each column j (refer to table 3.1). We fill up these using the `Frq_A` and `Frq_B` arrays.

Having obtained the observed frequencies, now we have to obtain the expected frequencies which is simply obtained for each row i and each column j by multipying the relevant row and column totals, and then dividing the result by the total number of elements in both data sets A and B. The row and column totals are obtained using the `R_C_totals(...)` method. We then create a 3-dimensional `float[][][]` array that will contain our relevant expected frequencies; the dimensions of this are the same as the 3-dimensional array we created for the observed frequencies. This will be called Prod_marg.

The final step is simply to obtain the differences between the observed and expected frequencies - (Joint[i][j][i] - Prod_marg[i][j][i]) -, square them up, divide them by Prod_marg[i][j][i], and add up the results for all the rows and columns in the data set. We also compute the normal approximation of this test statistic as well (iFigure, 2007). This normal approximation is outputted instead of the actual test statistic itself, as we will be using large data sets.

4.3.3 Kolmogorov-Smirnov one sample test

We create a `KS1()` (Kolmogorov-Smirnov 1 sample test; Neave and Worthington, 1988, p.89-93) object and apply the `KS1()` method to obtain our relevant test statistic. The `KS1()` method begins by obtaining the cumulative frequency distribution of the data called Obs, using the `Cumulative_freq(...)` method. This is then passed onto the CDF method which simply normalises it to obtain the cumulative probability distribution. We then obtain a `float[]` array of differences between our computed observed cumulative probability distribution and our given theoretical cumulative probability distribution. Using the `Max(...)` method, we obtain the maximum element in this array, which is our test statistic.

4.3.4 Kolmogorov-Smirnov two sample test

We create a `KS2()` (Kolmogorov-Smirnov 2 sample test; Neave and Worthington, 1988, p.149-152; Durbin, 1973, p.39-40; Trickett, 2004, 52-59) object and apply the `KS2()` method to obtain the relevant statistic. The method's first step is to combine the data using a `List<Integer>` Comb_data. We then convert this list to an `Integer[]` and then `int[]` array. The latter is called Comb_data_1. It then creates a `float[][]` array called x whose first dimension is the total number of elements in both data sets, while the second dimension is of size 1. The first dimension can be thought of as an index of the combined data set. At each index i, we add the element from the combined data set Comb_data_1 to x[i][0]. For the second position in the second dimension of this array x, we use a float variable to indicate where the element is from. Since the first `A.length` elements in the combined array come from A, we simply the float number 0.0F to the first `A.length` elements in x. Similarly for the next `B.length` elements, we simply assign the number 1.0F to indicate it is from B.

However we need to sort this data set. We accomplish this using `Arrays.sort(...)` (Java Sun Forum, 2006). So now our x is in ascending order. We have to find the cumulative

frequency of observing an element from A in this combined data set. As we have to do this for the combined data set, we can not employ the `Cumulative_freq(...)` method here. So we define a simple int variable k and initialise it from 1. We run a for loop through x, and at each index i, we do two things. If we encounter an element from A - x[i][1] == 0.0F -, we increase the count[i] by k, and we also increase k by 1. If we do not find an element, we increase the count[i] by (k-1), while we keep k the same.

So in this way we obtain our cumulative frequency distribution. Now we normalise it by dividing each count[i] by `A.length`: so at each index i we obtain the probability of observing an element (from A) that is lesser than or equal to the element at this index. We denote this by a `float[]` array F.

The next step is to calculate two similar `float[]` arrays, called Temp_2 and Temp_3. In the former, at each index i, we find the difference between F[i] and (i+1)/(A.length + B.length), while in the latter we simply reverse the terms and find the difference between (i+1)/(A.length + B.legnth) - F[i]. Then we obtain two floats called D_1 and D_2: the former being ((A.length + B.length) / (B.length)) * Max(Temp_2), and the latter being ((A.length + B.length) / (B.length)) * Max(Temp_3). We finally compute our test statistic: obtain the bigger of the two, D_1 and D_2.

4.4 Data generation

The Java file for data generation begins by creating 20,000 keys of float type. The keys are rather simple: $\{0.0, 1.0, 2.0, ..., 19,999.0\}$. These are stored in the aptly named `float[]` array Keys, whose length we denote by n. Now we can begin to generate aritificial data.

Firstly, I generate a data set of length $6 \cdot n$ whose probability distribution is uniform. This is simply done by adding all the elements of the Keys array a `List<Float>` object. We do this for 6 times. So this data set, called Data_U, will have length 120,000 and 20,000 distinct elements.

I generate all the other data sets similarly, however with a few artificial rules. Data_80 is a data array with length 120,000. 60,000 of those elements are simply the Keys array added 3 times to it via a process similar to that of the Data_U generation. This is done to maintain a large number of distinct elements, which is an important factor in our algorithms' sample complexity. The next is to fill in the remaining 60,000 places with only the first 80% is elements in the Keys array. This is done using a `Random()` object called generator, and applying the `nextInt(4.0/5.0 * n)` method on it. So in this manner Data_80 will contain more elements from the first 80% of keys than the last 20%. Data_60, Data_50 and Data_25 are created in a similar manner, where Data_60 is a data array where the first 60% of keys occur more frequently than the last 40%, Data_50 is a data array where the first 50% of keys occur more frequently than the last 50% and Data_25 is a data array where the first 25% of keys occur more frequently than the last 75%. We output all these data sets using the `Output_file_float(...)` method.

I also generate data that is then coded as a 2-dimensional array in the Independence algorithm. This is done by creating two different `float[]` arrays called Coordinate_1_C and Coordinate_2_C. Both have length $2 \cdot n$. In Coordinate_1_C, we add all the elements from the Keys array, however we do this only once. The remainder of the Coordinate_1_C array is filled

up by using a random number generator to pick elements at random from the keys array. Now to fill up the Coordinate_2_C array, we use an aritificial rule. At an index i in both arrays, if we see an element in Coordinate_1_C that is less than 8000.0, then we add 100.0 to the Coordinate_2_C array at that index. I am attempting to obtain a degree of association between what we observe in Coordinate_1_C and in Coordinate_2_C: if at any index we see an element in the former that is less than 8000.0, then the latter will have an element 100.0 at that index.

Lastly, I create two more coordinate arrays called Coordinate_1_D and Coordinate_2_D. Once again, in order to maintain a high level of distinct elements, I add the whole keys array to the Coordinate_1_D. This takes up half the space in the Coordinate_1_D array. To fill up the remaining half of the Coordinate_1_D array and also to fill up the Coordinate_2_D array, we simply randomly pick out elements from the keys array and assign it to the relevant coordinates, using `generator.nextInt(n)`. There are no rules so we would expect no association; we would expect independence.

All of these data sets are placed in our home directory which is usually: C:\Documents and Settings\Username.

4.5 Experiment design

There are three files that run our three experiments: CloseExp.java, IdExp.java and IndExp.java. These are the files that will have to be compiled. They work in the following manner.

CloseExp loads Data_80, Data_60, Data_50 and Data_25 using the `Read_Data_float(...)` method, creates the `Closeness()` object using the data arrays we wish to test upon, and applies the `Closeness()` method to it. The answer is then given. We then load the sample sets used by our algorithm; these were outputted to our home directory during the `Closeness()` test. These sets are known as Closeness_P and Closeness_Q and are loaded using the `Read_Data_int(...)` method.

We then create `CS2()` and `KS2()` objects, upon which we apply the `CS2()` and `KS2()` methods respectively, using Closeness_P and Closeness_Q. We compare the test statistic with the relevant critical value. As usual, if the test statistic is larger than our critical value, we reject the null hypothesis of similarity between the two distributions under question. We also output the test statistic and the critical value.

Moving on, IdExp loads Data_U and Data_25 using the same method as in CloseExp. We create the `Identity()` object, apply the `Identity()` method and output our answer. Then we load the sample set outputted by the `Identity()` method, which is called Identity. After creating the `CS1()` and `KS1()` objects, we apply the relevant `CS1()` and `KS1()` method respectively, using the Identity array, and obtain our answers. The same decision rules regarding rejection of the null hypothesis apply.

IndExp proceeds in the same manner as the other two experiments. It loads up Coordinate_1_C, Coordinate_2_C, Coordinate_1_D and Coordinate_2_D. Using the `Joint_var_float (...)` method on Coordinate_1_C and Coordinate_2_C, we obtain a joint variable called Data_A. Similarly Data_B is created using the Joint_var_float method on Coordinate_1_D and Coordinate_2_D. After creating an `Independence()` object, we apply the `Ind()` method

to it, and obtain our answer. We load up the sample sets outputted by the algorithm, however this time we use the `Read_Data_long(...)` method as we code the joint variable into a single random variable of long type. As before, we create the `CS2()` and `KS2()`, and apply the relevant `CS2()` and `KS2()` methods respectively on the sample sets. These sample sets are called Independence_P and Independence_Q. We output our results, using the same decision rules as before. So in this manner we conduct the experiments.

5 Experiments

In this section I will discuss the results of the experiments. Before we begin, I would like to say a few words about the critical value used in out tests. For the χ^2 one sample and two sample tests, the critical value is taken from the percentage points table of the standard normal distribution (Neave and Worthington, 1988, p.370). I will be using a 5% level of significance throughout, so our relevant critical value for a 2-tailed test is 1.96.

For the Kolmogorov-Smirnov one sample and two sample tests, the critical value, assuming a 5% level of significance, for the former is $\frac{1.36}{\sqrt{n}}$ where n is the sample size (ERI, 2007) and for the latter is $1.36 \cdot \sqrt{\frac{n+m}{n \cdot m}}$ where n and m are the sample sizes (Wessel, 2003). So our Kolmogorov-Smirnov critical values for both one sample and two sample tests depend on sample size(s). This will be a huge problem for the two sample critical value as we shall observe shortly.

Moreover, in all experiments we use a δ parameter of 5%, the same as our level of significance.

5.1 Closeness experiments

I will run the closeness algorithm, and its associated statistical counterparts, to check whether Data_80 is close to Data_80, Data_50 is close to Data_60 and Data_25 is close to Data_80. The first closeness experiment is simply used to see whether the algorithm and tests can pick up the fact that two distributions are close; we know they are close, as they are the same, however can the algorithm pick it up. The other two are used to check whether our aritificial data sets are close or not. The hypotheses for the statistical tests in each experiment is the same,

$$\begin{aligned} H_0 &: \text{The two probability distributions are similar.} \\ H_A &: \text{The two probability distributions are not similar.} \end{aligned}$$

I will use three different values of ϵ: 0.5, 0.4 and 0.3. On my attempts to use lesser values, I received an exception regarding Java Heap space. I believe that to test for smaller ϵ values, we will have to use faster computers with a larger memory. We will use the following abbreviations repeatedly in this section: TS stands for test statistic, CV stands for critical value, NaN stands for not a number, H_0 stands for null hypothesis, CS2 stands for Chi-square two sample test and KS2 stands for Kolmogorov-Smirnov two sample test. The results of the closeness experiment to check whether Data_80 is close to Data_80 is tabulated in table 2.

Table 2: Closeness experiment on Data_80 and Data_80.

ϵ	Closeness	CS2 TS	CS2 CV	CS2 Result	KS2 TS	KS2 CV	KS2 Result
0.5	Pass	-0.569	1.96	Accept H_0	0.013	0.018	Accept H_0
0.4	Pass	-1.509	1.96	Accept H_0	0.004	0.011	Accept H_0
0.3	Pass	1.309	1.96	Accept H_0	0.004	NaN	Test void

So our closeness algorithm performs very well. It picks up the fact that the two distributions are close for each value of ϵ. The statistical tests also accept the null hypothesis that the two distributions are similar, for each value of ϵ. However when $\epsilon = 0.3$, our KS2 test is void as the CV is NaN. This danger was mentioned previously: KS2 test's CV depends on the sizes of the samples it uses. As the sample sizes increase, and that occurs when we decrease our ϵ value, the CV is output as NaN. One can observe from table 2 that as the sample sizes increase (ϵ getting smaller), the KS2 test's CV became smaller. This shows that KS2 tests are not very useful for very large samples.

Our next experiment involves checking whether Data_50 and Data_60 are close. The results are in table 3.

Table 3: Closeness experiment on Data_50 and Data_60.

ϵ	Closeness	CS2 TS	CS2 CV	CS2 Result	KS2 TS	KS2 CV	KS2 Result
0.5	Fail	4.212	1.96	Reject H_0	0.079	0.018	Reject H_0
0.4	Fail	10.914	1.96	Reject H_0	0.077	0.011	Reject H_0
0.3	Fail	36.577	1.96	Reject H_0	0.082	NaN	Test void

From table 3 we can observe that the Data_50 and Data_60 are not close for each value of ϵ. This is confirmed by our CS2 and KS2 tests: all H_0 are rejected and as ϵ decreases we can see that the rejection becomes stronger as the TS moves further away from the CV. Once again, for $\epsilon = 0.3$, the KS2 test fails to perform properly.

Finally, our last closeness experiment checks whether Data_25 and Data_80 are close. The results are in table 4.

Table 4: Closeness experiment on Data_25 and Data_80.

ϵ	Closeness	CS2 TS	CS2 CV	CS2 Result	KS2 TS	KS2 CV	KS2 Result
0.5	Fail	10.376	1.96	Reject H_0	0.337	0.018	Reject H_0
0.4	Fail	30.596	1.96	Reject H_0	0.346	0.011	Reject H_0
0.3	Fail	92.948	1.96	Reject H_0	0.346	NaN	Test void

From table 4 we can observe that, as we expected, Data_25 and Data_80 are not close for each value of ϵ. Once again, our algorithm's results are backed up by the CS2 and KS2 tests: H_0 is rejected for each ϵ. The rejection becomes stronger as ϵ decreases as the TS moves further away from the CV. However, for $\epsilon = 0.3$, the KS2 test is void again.

So from these experiments, we can see that the algorithm works very well in comparison with CS2 and KS2. We can also observe that the KS2 test is not very useful for large sample sizes. This was expected, given the form of its CV, and also because non-parametric statistics usually work on small sample sizes. A cursory glance at any CV table of any non-parametric test will show us this.

5.2 Identity experiments

The identity experiments will check whether Data_U and Data_25 are uniformly distributed. In the former case we expect that the test will pass. That is why Data_U was chosen as it would give us an idea about whether the algorithm can pick up the similarity. The hypotheses for the statistical tests in each experiment is the same,

$$H_0 \quad : \quad \text{The probability distribution is uniform.}$$
$$H_A \quad : \quad \text{The probability distribution is not uniform.}$$

I have been able to use four different values of ϵ: 0.5, 0.4, 0.3 and 0.2. As before, on my attempts to go any lower, I received a Java Heap space exception. We will use the following abbreviations throughout: CS1 stands for Chi-square one sample test and KS1 stands for Kolmogorov-Smirnov one sample test. The results of the identity experiment on Data_U are tabulated in table 5.

Table 5: Identity experiment on Data_U.

ϵ	Identity	CS1 TS	CS1 CV	CS1 Result	KS1 TS	KS1 CV	KS1 Result
0.5	Pass	-63.916	1.96	Reject H_0	0.007	0.018	Accept H_0
0.4	Pass	-59.351	1.96	Reject H_0	0.004	0.015	Accept H_0
0.3	Pass	-25.708	1.96	Reject H_0	0.008	0.011	Accept H_0
0.2	Fail	-31.516	1.96	Reject H_0	0.003	0.007	Accept H_0

The test picks the right answer up until $\epsilon = 0.2$. One must remember though that at most 5% of the times the result will be incorrect. The KS1 test backs up the algorithm, and even picks the correct answer at $\epsilon = 0.2$. Although the KS1 test's CV also depends on the sample size, just as in the KS2 test, we see no problem with the ability of this test in dealing with large sample sizes.

The CS1 test exhibits signs of bias. This bias (unreliability) is most likely created by having quite a few expected values of less than 5. Often this is rectified by combining rarer categories, that is to say we can bin those distinct elements that are fairly similar and are fairly rare (Tricket, 2003). We can not do this with our sample set as each element is compleletly distinct from the other. So combining them would not be a solution in getting reliable CS1 results. CS1 is thus unreliable for large sample sizes.

Table 6: Identity experiment on Data_25.

ϵ	Identity	CS1 TS	CS1 CV	CS1 Result	KS1 TS	KS1 CV	KS1 Result
0.5	Pass	-58.287	1.96	Reject H_0	0.009	0.018	Accept H_0
0.4	Pass	-46.945	1.96	Reject H_0	0.015	0.014	Reject H_0
0.3	Pass	-11.246	1.96	Reject H_0	0.025	0.011	Reject H_0
0.2	Fail	-9.328	1.96	Reject H_0	0.048	0.007	Reject H_0

Table 6 shows the results of the identity experiment on Data_25. The results are similar to the previous experiment. The CS1 again shows similar signs of being unreliable. So we

should not take its results at face value. The algorithm gives similar answers as in the previous experiment, meaning that maybe Data_25 is not that much different from Data_U (maybe a closeness test is in order). The KS1 again provides results that go reasonably correct, especially for when ϵ is 0.5 and 0.2. When it is 0.4, the rejection of the null hypothesis is only marginal, as the KS1 TS and CV are very similar. So all in all, the KS1 test performs well in terms of picking the right answer. The identity algorithm performs well, however not as well as the closeness algorithm.

5.3 Independence experiments

The independence experiments will test whether the joint distribution of Data_A is close to the product of its marginals, that is Coordinate_1_C and Coordinate_2_C. If they are, then we have independence. The way in which Data_A was designed, with a reasonable level of assocation in mind, we would expect the algorithm to output fail. Moreover we also use the data set Data_B, where there is no association between the marginals, that is Coordinate_1_D and Coordinate_2_D. So we would expect the algorithm to pass. The hypotheses for the statistical tests in each experiment is the same,

$$\begin{aligned} H_0 &: \text{We have independence.} \\ H_A &: \text{We do not have independence.} \end{aligned}$$

Just as in the closeness experiments, I was only able to use ϵ values up until 0.3, as I kept getting Java Heap space exceptions. So the ϵ values we use are 0.5, 0.4 and 0.3. The results of the independence experiment on Data_A are as follows.

Table 7: Independence experiment on Data_A.

ϵ	Independence	CS2 TS	CS2 CV	CS2 Result	KS2 TS	KS2 CV	KS2 Result
0.5	Fail	13.658	1.96	Reject H_0	0.251	0.015	Reject H_0
0.4	Fail	42.659	1.96	Reject H_0	0.246	0.009	Reject H_0
0.3	Fail	144.475	1.96	Reject H_0	0.245	NaN	Test void

The results are good. The algorithm picks up on the fact that there is independence for each value of ϵ. It is backed up by the statistical tests CS2 and KS2. The rejection of the H_0 gets stronger as ϵ gets smaller: the TS get larger compared to the CV. As this independence test is basically a closeness test, we observe the same problem we had in the latter: KS2 test can not handle large sample sizes. As ϵ decreases, the sample size increases, the KS2 test becomes void as the CV become NaN.

Table 8: Independence experiment on Data_B.

ϵ	Independence	CS2 TS	CS2 CV	CS2 Result	KS2 TS	KS2 CV	KS2 Result
0.5	Fail	13.947	1.96	Reject H_0	0.009	0.014	Accept H_0
0.4	Fail	45.004	1.96	Reject H_0	0.005	0.009	Accept H_0
0.3	Fail	150.543	1.96	Reject H_0	0.003	NaN	Test void

The results of the independence experiment on Data_B (table 8) are not as good as the previous experiment. We expected the test to pass as we have independence, via the way in which we generated the data. However the test fails at each ϵ. Although the CS2 backs up the algorithm's answer, the KS2 test picks the right answer: it accepts the null hypothesis of independence. So in this experiment our algorithm fails. Overall, the independence algorithm performs does not perform too well.

6 Conclusion

In this paper, I have explained three sampling algorithms, their counterparts from statistics and have compared them using large artificial data sets. The closeness algorithm was the best performer amongst the algorithms, in terms of picking the right answer repeatedly. It was backed up by the χ^2 two sample test and the Kolmogorov-Smirnov two sample test. We also observed that the latter statistical test is not very useful for large sample sets, given the form of its critical value.

Moving on, the identity algorithm also performed reasonably well, however our χ^2 one sample test results were completely unreliable, and the reason is most likely small expected frequencies (Trickett, 2003). The unreliability of the χ^2 one sample test results is another advantage that the identity algorithm has: it does not put any restriction or condition on the sample set. The Kolmogorov-Smirnov one sample test performed well in picking the right answers.

Lastly, the independence algorithm did not fare so well. Further testing on different data sets will be useful in testing this algorithm, however in my experiments it did not perform all too well, especially in comparison with the Kolmogorov-Smirnov two sample test.

The obvious way to further this study is to use a larger range of ϵ values. My computer had limitations, however with a computer that is fast and has a large memory, further experiments with even smaller ϵ values will produce interesting results. Also, we can attempt to vary the error-probability parameter δ and the level of significance α. This would also lead to more insights regarding the performance of these algorithms.

We can also attempt to look at other statistical tests. However, the Kolmogorov-Smirnov tests are the best, in my opinion, as they can pick up any differences between two distributions: differences in location, spread and skewness (the first, second and third moments).

However at the end of the day, these are still simply sampling algorithms, and via sampling we have a chance (a small one but still a chance nevertheless) of obtaining an unrepresentative sample. Thus our results on such a sample set would not be correct.

References

[1] Anderson, D., Sweeney, D. and Williams, T. (1994). *Introduction to Statistics: Concepts and Applications.* Third edition, Minneapolis; St. Paul: West Pub.

[2] Batu, T. (2001). *Testing properties of distributions.* PhD dissertation, Cornell University. [Internet] London School of Economics, London. Available from: `http://www.maths.lse.ac.uk/Personal/batu/respub.html\#pub` [Accessed on 20th June 2007].

[3] Batu, T., Fischer, E., Fortnow, L., Kumar, R., Rubinfeld, R., and White, P. (2003). *Testing Random Variables for Independence and Identity.* [Internet] London School of Economics, London. Available from: `http://www.maths.lse.ac.uk/Personal/batu/papers/jtii.pdf` [Accessed on 20th June 2007].

[4] Batu, T., Fortnow, L., Rubinfeld, R., Smith, W. D., and White, P. (2000). *Testing that distributions are close.* [Internet] IEEE Computer Society. Available from: `http://csdl.computer.org/dl/proceedings/focs/2000/0850/00/08500259.pdf` [Accessed on 20th June 2007].

[5] Clarkson University CS643 (2007). [Internet]. Available from: `http://people.clarkson.edu/%7Ewhesse/cs643/assignments/assignment\_2.html` [Accessed on 17th August 2007].

[6] Durbin, J. (1973). *Distribution theory for tests based on the Sample Distribution Function.* Bristol, U.K.: Society for Industrial and Applied Mathematics.

[7] ERI. (2007). *Kolmogorov-Smirnov one sample distribution table.* [Internet] ERI. Available from: `http://www.eridlc.com/onlinetextbook/appendix/table7.htm` [Accessed on 4th July 2007].

[8] Flanagan, D. (2005). *Java in a Nutshell.* Fifth edition, Sebastopol, CA: O'Reilly Media Inc.

[9] Goldreich, O. and Ron, D. (2000). *On testing expansion in bounded-degree graphs.* [Internet] CiteSeer. Available from: `http://citeseer.ist.psu.edu/cache/papers/cs/17995/http:zSzzSzwww.wisdom.weizmann.ac.ilzSz\symbol{126}%odedzSzPSzSztestEXP.pdf/goldreich00testing.pdf` [Accessed on 15th August 2007].

[10] Google Groups Forum (2007). [Internet] Available from: `http://groups.google.com/group/comp.lang.java.help/browse\_thread/thread/72008b53ab794fb7/10f4e15b26559849?lnk=raot` [Accessed on 7th July 2007].

[11] Hubbard, J. (2000). *Schaum's outline of theory and problems of programming with Java.* McGraw-Hill.

[12] iFigure. (2007a). *Chi-square test for known distributions.* [Internet] IFA Services. Available from: `http://fonsg3.let.uva.nl/Service/Statistics/X2Test.html` [Accessed on 23rd July 2007].

[13] iFigure. (2007b). *Chi-square test for equality of distributions.* [Internet] IFA Services. Available from: http://fonsg3.let.uva.nl/Service/Statistics/EqualDistribX2.html [Accessed on 23rd July 2007].

[14] Java Sun Forum (2006). *Sorting a multidimensional array.* [Internet]. Available from: http://forum.java.sun.com/thread.jspa?threadID=716459\&messageID=4150630\% [Accessed on 26th July 2007].

[15] Langley, R. (1971). *Practical Statistics for Non-Mathematical people.* U.K.: David & Charles (Publishers) Limited.

[16] LSE. (2007). *Taught Masters Degree Results by programme-2005/06.* [Internet] London School of Economics, LSE. Available from: https://exchange.lse.ac.uk/public/LSE%%20Website/resources/statisticsOnLSE/TM%20degree\%20results\%20by\%20programme\%202005.6.pdf [Accessed on 15th August 2007].

[17] Neave, H. and Worthington, P. (1988). *Distribution-free Tests.* London: Unwin Hyman Ltd.

[18] Pierce, A. (1970). *Fundamentals of Nonparametric Statistics.* Belmont, CA: Dickenson Publishing Company, Inc.

[19] Sun. (2004). *Autoboxing.* [Internet] Sun Microsystems Inc. Available from: http://java.sun.com/j2se/1.5.0/docs/guide/language/autoboxing.html [Accessed on 28th June 2007].

[20] Sun Developer Network (2006). *Union and Intersection of two arrays.* [Internet] Sun Microsystems Inc. Available from: http://forum.java.sun.com/thread.jspa?threadID=450002\&messageID=2042326\% [Accessed on 17th July 2007].

[21] Sun Java Tutorials. (2007). *The Collection Interface.* [Internet] Sun Microsystems Inc. Available from: http://java.sun.com/docs/books/tutorial/collections/interfaces/collection.html [Accessed on 15th July 2007].

[22] Swartz, F. (2006). *Array Example - Maximum.* [Internet] Java Notes. Available from: http://www.leepoint.net/notes-java/data/arrays/arrays-ex-max.html [Accessed on 22nd July 2007].

[23] Trickett, A. (2004). *Further Applied Statistics booklet.* Course ES2211, 2004/05, University of Manchester, U.K.

[24] Trickett, A. (2003). *Applied Statistics booklet.* Course ES1142, 2003/04, University of Manchester, U.K.

[25] Wessel, P. (2003). *Critical Values for the Two-Sample Kolmogorov-Smirnov test (2-sided).* [Internet] University of Hawai'i at Manoa. Available from: http://www.soest.hawaii.edu/wessel/courses/gg313/Critical_KS.pdf [Accessed on 25th August 2007].

A Appendix

The CD submitted with this dissertation contains the following files: CloseExp.java, IdExp.java, IndExp.java, Closeness.java, Identity.java, Independence.java, CS1.java, CS2.java, KS1.java, KS2.java, Methods.java, Data.java, Data_U.txt, Data_25.txt, Data_50.txt, Data_60.txt, Data_80.txt, Coordinate_1_C.txt, Coordinate_2_C.txt, Coordinate_1_D.txt, Coordinate_2_D.txt.

There is also a copy of the dissertation: MA498_32187.pdf.