Uniwersytet Mikołaja Kopernika
Wydział Fizyki, Astronomii i Informatyki Stosowanej
Katedra Informatyki Stosowanej

Olena Savelieva
nr albumu: 296024

Praca: inżynierska
na kierunku informatyka stosowana

Biblioteka .NET i pakiet NuGet do zaawansowanych obliczeń statystycznych

Opiekun pracy dyplomowej
dr hab. Jacek Matulewski
Katedra Informatyki Stosowanej
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Toruń 2021

Pracę przyjmuję i akceptuję

…….

data i podpis opiekuna pracy

Potwierdzam złożenie pracy dyplomowej

…….

data i podpis pracownika dziekanatu

Pragnę podziękować rodzicom oraz siostrom, bez których nie miałabym możliwości zdobycia cennej wiedzy. Składam również podziękowania mojemu promotorowi dr. hab. Jackowi Matulewskiemu za poświęcony czas, cierpliwość oraz cenne wskazówki.

Spis treści

1.Wstęp

Analiza statystyczna i przetwarzanie danych to jeden z kluczowych elementów rozwoju nauki oraz biznesu. Przewidywania, wyjaśnienie i opis zjawisk są kluczowym elementem rozwoju współczesnej nauki, zarówno dyscyplin przyrodniczych, jak i społecznych. Analiza statystyczna jest przeprowadzana w celu określenia podstawowych schematów i trendów, czyli tak naprawdę wydobycia z danych wiedzy o badanych obiekcie.

Celem pracy jest przygotowanie zbioru klas C# umieszczonych w bibliotece DLL dla platformy .NET 5 i dystrybuowanych za pomocą pakietu NuGet, które będą wspierać obliczanie podstawowych parametrów statystycznych, testów parametrycznych i nieparametrycznych oraz obliczanie korelacji. Praca obejmuje również przygotowani e testów jednostkowych weryfikujących poprawność zaimplementowanych algorytmów.

2. Podstawowe parametry statystyczne

W tym rozdziale zostaną opisane implementacje podstawowych miar położenia i rozproszenia.

Miary położenia, jak sama nazwa wskazuje, to wartości wokół których skupia się rozkład analizowanych zmiennych. Można je podzielić na klasyczne i pozycyjne ([4], str. 57). Do miar klasycznych należą średnie: arytmetyczna, harmoniczna i geometryczna. Do miar pozycyjnych należy dominanta (wartość modalna, wartość najczęstsza) oraz kwantyle.

Miary rozproszenia (zmienności, dyspersji) odnoszą się do określania różnic pomiędzy obserwacjami a wartością średnią. Do nich należą: rozstęp, wariancja, odchylenie standardowe oraz współczynnik zmienności.

2.1. Średnia arytmetyczna

Opis

Średnia arytmetyczna jest jedną z najprostszych i najbardziej intuicyjnych miar oceny populacji, zazwyczaj stosowaną w życiu codziennym. Obliczenia średniej arytmetycznej ($\bar{x}$) przeprowadza się według wzoru ([4], str. 57):

$$\bar{x} = \frac{\sum x_i}{n} = \frac{x_1+x_2+\ldots+x_n}{n} \quad (\mu = \frac{\sum x_i}{N}) \qquad (2.1)$$

gdzie: $\overline{x}$ – symbol średniej arytmetycznej dla próby, μ – symbol średniej arytmetycznej dla populacja.

Czym większa jest liczba prób, tym bardziej niezawodna jest średnia. Jednak trzeba pamiętać, że $\overline{x}$ prawidłowo charakteryzuje średni poziom w przypadku rozkładów symetrycznych lub o umiarkowanej sile asymetrii.

Implementacja

Listing.3.1. Metoda obliczającą średnią aretmytyczną dla liczb typu double

```
public static double CalculateMean(this IEnumerable<double> data)
{
      if (data.Count() == 0) throw new EmptyCollectionException();
      return data.Average();
}
```

Do obliczania średniej (listing 3.1) wykorzystujemy operator zapytań LINQ `Average`. W przypadku przesyłania pustej kolejki zgłoszony będzie wyjątek `EmptyListException` który jest zdefiniowany w pliku `ProgramExceptions`. W przypadku typów `int` i `long` korzystamy z powyżej metody rzutując otrzymaną kolekcje na typ `double` (listing 3.2).

Listing.3.2. Metoda obliczającą średnią arytmetyczną dla liczb typu int

```
public static double CalculateMean(this IEnumerable<int> data)
{
      if (data.Count() == 0) throw new EmptyCollectionException();
      return data.Select(i => (double)i).CalculateMean();
}
```

Testy

Metoda `CalculateMean` została obłożona testami jednostkowymi `TestCalculateMeanDouble` i `TestCalculateMeanInt` z klasy `TestStatystyka`, które sprawdzają ją dla prostych zbiorów liczb rzeczywistych i całkowitych. Przetestowane zostały także sytuacje, w których metoda otrzymuje zbiór pusty.

2.2. Mediana

Opis

Mediana, znana także jako wartość środkowa lub drugi kwartyl ([3], str.39), jest jedną z miar tendencji centralnej, dla której połowa obserwowanych próbek leży poniżej tej wartości, a druga połowa powyżej.

Mediana jest obliczana jako wartość leżąca pośrodku dla posortowanych liczb w próbie, jeżeli liczba obserwacji w próbie jest nieparzysta (2.3). Jeśli liczba obserwacji jest parzysta to mediana jest obliczana jako średnia z dwóch wartości leżących pośrodku (2.2).

$$m = \frac{X_{n/2} + X_{n/2+1}}{2} \qquad (2.2)$$

$$m = \frac{X_{n/2}}{2} \qquad (2.3)$$

Implementacja

Listing.3.3. Metoda obliczającą medianę dla liczb typu double

```
private static double _calculateMedian(List<double> data)
{
      if (data.Count == 0) throw new EmptyCollectionException();
```

```
        data.Sort();
        if (data.Count % 2 != 0)
                return data[data.Count / 2];
        else
                return (data[data.Count / 2 - 1] + data[data.Count / 2]) / 2.0;
}
public static double CalculateMedian(this IEnumerable<double> data)
{
        List<double> copy = new List<double>();
        foreach (double item in data) copy.Add(item);
        return _calculateMedian(copy);
}
```

Do obliczenia mediany wykorzystujemy metodę `CalculateMedian` która znajduje się w klasie `Statistics`. Ta metoda korzysta z metody `_calculateMedian` która jest prywatna, ponieważ w niej znajduje się wyłącznie szczegóły implementacji dla listy typu `double`. Metody dla zbiorów `int` i `long` również korzystają z tej metody, dla nich podobnie jak w `CalculateMedian` (listing 3.3), tworzymy kopie otrzymywanego zbioru z elementami zrzutowanymi na typ `double`. W `_calculateMedian` sprawdzamy czy długość listy jest nieparzysta, jeżeli tak jest, wykorzystujemy wzór (2.3), a wzór (2.2) jeżeli parzysta.

Testy

Listing.3.4. Test jednostkowy dla mediany

```
readonly double[] emptyTable = { };
readonly double[] tableDouble1 = { 1, 2, 3, 4, 5 };
readonly double[] tableDouble2 = { 1, 2, 3, 4 };
readonly double[] tableDouble3 = { 1, 1, 1, 2, 2 };
readonly double[] tableDouble4 = { 1, 1, 1, 2 };


[TestMethod]
public void TestCalculateMedianDouble()
{
        double median1 = Statystyki.CalculateMedian(tableDouble1);
        double median2 = Statystyki.CalculateMedian(tableDouble2);
        double median3 = Statystyki.CalculateMedian(tableDouble3);
        double median4 = Statystyki.CalculateMedian(tableDouble4);

        Assert.AreEqual(3, median1);
        Assert.AreEqual(2.5, median2);
        Assert.AreEqual(1, median3);
        Assert.AreEqual(1, median4);
        Assert.ThrowsException<EmptyListException>(() =>
        Statystyki.CalculateMedian(emptyTable));
```

```
}
```

Metoda `CalculateMedian` została obłożona testami jednostkowymi `TestCalculateMedianDouble` (listing 3.4) i `TestCalculateMedianInt` z klasy `TestStatystyka`. Podobnie jak dla średniej przetestowana została też sytuacja, w której otrzymujemy pustą listę.

2.3. Moda

Opis

Moda (dominanta, wartość najczęstsza) jest obliczana jako najczęściej występująca wartość w próbie. Moda może być stosowana w każdej skali pomiarowej tj. w skalach nominalnej, porządkowej, interwałowej, ilorazowej lub absolutnej [3].

Implementacja

Listing.3.5. Metoda obliczającą mody dla liczb typu double

```
public static double[] _calculateMode(List<double> data)
{
	if (data.Count == 0) throw new EmptyCollectionException();

	List<double> currentMax = new List<double>();

	Dictionary<double, int> dictNumberOfOccurrencesOfElements =
data.GroupBy(x =>  x).ToDictionary(x => x.Key, x => x.Count());

	if (!((dictNumberOfOccurrencesOfElements.Values.GroupBy(x =>
x).Where(x => x.Count() >= 1)).Count() > 1)) throw new Exception("There is no
dominant in a given set");

	int occurences = 0;

	foreach (KeyValuePair<double, int> x in
	dictNumberOfOccurrencesOfElements)
	{
		if (occurences < x.Value)
		{
			currentMax.Clear();
			currentMax.Add(x.Key);
			occurences = x.Value;
		}
		else if (occurences == x.Value)
		{
			currentMax.Add(x.Key);
		}
	}
	return currentMax.ToArray();
}
```

Podobnie jak i w przypadku obliczania mediany z prywatnej metody `_calculateMode` (listing 3.5) korzystają publiczne metody `CalculateMode` przeciążone dla różnych typów zbiorów. W celu obliczenia mody tworzymy słownik elementów, w którym kluczem jest element z listy, a wartością liczba jego wystąpień. W przypadku, gdy w zbiorze nie ma mody, co zachodzi, gdy wszystkie elementy są takie same lub każda wartość występuje tylko raz lub gdy wszystkie wartości występują taką samą liczbę razy (różną od 1), jest zgłaszany odpowiedni wyjątek.

Może zdarzyć się tak że w próbie występuje dwie mody, czyli tak zwana dominanta wielokrotna. W metodzie `_calculateMode(List<double> list)` uwzględniamy to, że może być kilka dominant w próbie, dlatego zwracamy tablicę elementów.

Testy

Metoda `CalculateMode` została obłożona testami jednostkowymi `TestCalculateModeDouble` oraz `TestCalculateModeInt`. Przetestowane zostały także sytuacje, gdy zbiór jest pusty oraz kiedy mediany nie ma w zbiorze. W tych sytuacjach sprawdzane jest zgłaszanie odpowiednich wyjątków.

2.4. Kwartyle

Opis

Kwartyl to rodzaj kwantyla([3] str.39), który dzieli próbę na cztery części lub ćwiartki o mniej więcej równej długości. Dane muszą być posortowane od najmniejszego do największego. Wydzielane są trzy główne kwartyle:

- Pierwszy kwartyl (Q1) znany również jako dolny lub 25-ty kwartyl empiryczny, ponieważ 25% danych znajduje się poniżej.
- Drugi kwartyl (Q2) jest medianą próby (zob. podrozdział 2.2).
- Trzeci kwartyl (Q3) jest wartością środkową pomiędzy medianą i maksymalną wartością. Także znany jako kwartyl górny lub 75-ty kwartyl empiryczny, ponieważ 75% danych znajduje się poniżej tego punktu.

Implementacja

Listing.3.6. Metoda obliczającą kwartyle dla liczb typu double

```
public struct Quartile
{
      public double q1;
```

```
        public double q2;
        public double q3;
}
public static Quartile CalculateQuartile(this IEnumerable<double> list)
{
        if (list.Count() == 0) throw new EmptyListException();
        List<double> _list = list.ToList();
        _list.Sort();

        List<double> lowerHalf = new List<double>();
        List<double> upperHalf = new List<double>();
        double q1, q2, q3;
        q2 = CalculateMedian(list);
        if (_list.Count % 2 != 0)
        {
                for (int i = 0; i < _list.Count / 2+1; ++i)
                lowerHalf.Add(_list[i]);
                for (int i = _list.Count / 2; i < _list.Count; ++i)
                upperHalf.Add(_list[i]);
        }
        else
        {
                for (int i = 0; i < _list.Count/2; ++i) lowerHalf.Add(_list[i]);
                for (int i = _list.Count / 2; i < _list.Count; ++i)
                upperHalf.Add(_list[i]);
        }

        q1 = _calculateMedian(lowerHalf);
        q3 = _calculateMedian(upperHalf);
        return new Quartile
        {
                q1 = q1,
                q2 = q2,
                q3 = q3
        };
}
```

Statyczna metoda obliczania kwartyli o nazwie `CalculateQuartile` jest zdefiniowana w klasie `Statistics` (listing 2.6) Jeżeli w próbie występuje nieparzysta liczba obserwacji dołączamy medianę do obu podzielonych równych danych. Jeżeli liczba obserwacji jest parzysta, to zbiór jest dzielony równo na pół. Wartość dolnego kwartylu jest mediana

dolnej połowy, a górnego - górnej. Wartości znalezione tą metodą znane są również jako "zawiasy Tukeya". Zwracamy trzy najważniejsze kwatryle.

Testy

Sprawdzania poprawności działania metody sprawdza się w teście jednostkowym `TestCalculateQuartile`. Wyniki uzyskane w testach są porównywane do wyników uzyskanych za pomocą funkcji `quantile(list,type=2)` w RStudio które traktujemy jako referencyjne.

2.5. Odchylenie standardowe

Opis

Odchylenie standardowe jest jedną z miar rozproszenia poszczególnych wartości wokół średniej arytmetycznej.
Odchylenie standardowe z próby oblicza się korzystając z następującego wzoru:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \qquad (2.4)$$

gdzie x_i to kolejne wartości zmiennej, a $\overline{x}$ to średnia arytmetyczna tych wartości, n - liczność próby.

Odchylenie standardowe z populacji:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}} \qquad (2.5)$$

gdzie x_i to kolejne wartości zmiennej a μ to średnia arytmetyczna tych wartości, N – liczność próby.

Czym wyższa wartość odchylenia standardowego tym bardziej zróżnicowana grupa. Odchylenie standardowe dla próby jest estymatorem (pewnym przybliżeniem) odchylenia standardowego dla populacji. Ponieważ próba to informacje tylko o części populacji, wynik odchylenia standardowego z próby nigdy nie jest dokładny. Wartość odchylenia

standardowego w populacji mieści się w pewnym przedziale zawierającym odchylenia standardowe z próby. Ten przedział nazywany jest przedziałem ufności dla odchylenia standardowego.

Implementacja

Listing.2.7. Metoda obliczająca odchylenie standardowe

```
public enum StandardDeviationType { Samples, Population };

public static double CalculateStandardDeviation(this IEnumerable<double>
list, StandardDeviationType type = StandardDeviationType.Samples)
{
	double mean = list.Average();
	double deviation = 0;
	foreach (double item in list)
	{
		deviation += (item - mean) * (item - mean);
	}
	switch(type)
	{
		case StandardDeviationType.Population:
			if (list.Count() == 0) throw new Exception("Standard
			deviation calculation error. The collection is empty");
			deviation /= list.Count();
			break;
		case StandardDeviationType.Samples:
			if (list.Count() <= 1) throw new Exception("Standard
			deviation calculation error. Population insufficient");
			deviation /= list.Count() - 1;
			break;
		default:
			throw new Exception("Unrecognized type of standard
		deviation");
	}
	deviation = Math.Sqrt(deviation);
	return deviation;
}
```

Metoda służąca do obliczania odchylenia standardowego (listing 2.7) jest zaimplementowana w taki sposób, żeby móc obliczać odchylenie standardowe z próby (2.4) i z populacji (2.5). Domyślnie będzie ono obliczana z próby.

Testy

Metoda `CalculateStandardDeviation` została obłożona testami jednostkowymi `TestCalculateStandardDeviationPopulationDouble`, `TestCalculateStandardDeviationSamplesDouble` i `TestCalculateStandardDeviationPopulationInt`. Watrości są porównywane do wartości uzyskanych przy pomocy Exela funkcją - `ODCH.STANDARD.POPUL.A(A1:A5)` – dla populacji i `ODCH.STANDARD.PRÓBKI(A1:A5)` – dla próbki.

2.6. Skośność

Opis

Skośność jest miarą asymetrii obserwowanych wyników, nazywaną także współczynnikiem asymetrii. Zdaje sprawę z tego, jak bardzo rozkład danych różni się od rozkładu symetrycznego. Jest różnie definiowany, ale zwykle bazuje na różnicy między średnią arytmetyczną a medianą Zwykle wartość współczynnika mieści się w przedziale [-1;1]. Im bliższy jest zeru, tym rozkład jest bardziej symetryczny. Wartości dodanie oznaczają prawoskośność rozkładu, a ujemne – lewoskośność.

Implementacja

Skośność została zaimplementowana w statycznej metodzie `_calculateSkewness` (listing 2.8) W tej metodzie zaimplementowano trzy najpopularniejsze estymatory parametryczne skośności [6]. „Zgodnie z pracami Joanesa i Gilla (1998) poniżej przedstawiono trzy najczęściej używane estymatory parametryczne skośności z tradycyjnych miar, które zostały opracowane przez SAS i MINITAB.” ([7], str. 728). Metoda oprócz kolekcji danych przyjmuje również argument `type` decydujący o typie obliczanej skośności.

Skośność typu pierwszego (argument type=1) jest klasyczną definicją skośności zaproponowaną przez Cramera (1946). Ma postać:

$$g_1 = \frac{m_3}{\sqrt{m_2^3}} \qquad (2.6)$$

gdzie momenty m_2 i m_3 dla zmiennej x są zdefiniowane jako:

$$m_r = \frac{1}{n}\Sigma(x_i - \bar{x})^r \qquad (2.7)$$

Listing.2.8. Metoda obliczającą skośność

```
public enum SkewnessType { type1, type2,type3 };

private static double _calculateSkewness(List<double> list, SkewnessType
type=SkewnessType.type3)
{
	if (list.Count == 0) throw new EmptyCollectionException();

	double m3 = 0;
	double m2 = 0;
	int n = list.Count();
	double mean = CalculateMean(list);
	double temp, _m2;
	foreach (double item in list)
	{
		temp = (item - mean);
		_m2 = temp * temp;
		m3 += _m2*temp;
		m2 += _m2;
	}
	double A;
	switch (type)
	{
		case SkewnessType.type1:
			A = Math.Sqrt(n / m2 * m2 * m2) * m3;
			break;
		case SkewnessType.type2:
			A = (n * Math.Sqrt((n - 1)/ m2 * m2 * m2) * m3) / (n - 2);
			break;
		case SkewnessType.type3:
			A = (Math.Sqrt(n * (1 - 1.0 / n) * (1 - 1.0 / n) * (1 -
1.0 / n) / m2 * m2 * m2) * m3);
			break;
		default:
			throw new InvalidArgument("type");
	}
```

```
        return Math.Round(A, 7);
}
```

Skośność typu drugiego (type = 2) jest używana w SAS i SPSS. Obliczana jest za pomocą wzoru:

$$G_1 = \frac{\sqrt{n(n-1)}}{n-2} g1 \qquad (2.8)$$

Skośność typu trzeciego (type = 3) używana jest w MINITAB i BMDP:

$$b1 = (\frac{n-1}{n})^{3/2} g1 \qquad (2.9)$$

Domyślnym typem obliczania skośności jest typ 3 (podobnie jest w RStudio).

Należy zwrócić uwagę, że wartości obliczane zgodnie z poszczególnymi wzorami nie różnią się zbytnio dla dużych próbek, ale mogą być znaczne dla małych.

Testy

Metoda `_calculateSkewness` została obłożona testami `TestCalculateSkewnessInt` oraz `TestCalculateSkewnessDouble`. Zostały przetestowane sytuacja dla danego zbioru liczb `int` i `double` ze znaną skośnością dla trzech rodzajów. Wyniki referencyjne zostały uzyskane za pomocą RStudio.

2.7. Kurtoza

Opis

Kurtoza jest to miara, która określa jak bardzo rozrzut danych wokół średniej arytmetycznej jest zbliżony do rozrzutu tych danych w rozkładzie normalnym.
Im bardziej wartość kurtozy różni się od zera, tym rozkład będzie albo bardziej płaski (wartości ujemne) albo bardziej spiczasty niż rozkład normalny (wartości dodatnie).
Wzór na kurtozę zwykle podaje się stosując moment czwartego rzędu:

$$K = \frac{m_4}{sd^4} = \frac{m_4 * n}{m_2 * m_2}, \qquad (2.10)$$

gdzie m_2 i m_4 to momenty centralne drugiego i czwartego rzędu obliczane przy pomocy wzoru (2.7).

Implementacja

Listing.2.9. Metoda obliczającą kurtozę

```
private static double _calculateKurtosis(List<double> data)
{
	if (data.Count() == 0) throw new EmptyCollectionException();
	int n = data.Count();
	double mean = CalculateMean(data);
	double m2 = 0;
	double m4 = 0;
	double help, _m2;
	foreach (double item in data)
	{
		help = (item - mean);
		_m2 = help*help;
		m2 += _m2;
		m4 += _m2*_m2;
	}
	double k = (n * m4) / (m2 * m2);
	return Math.Round(k, 6);
}

public static double CalculateKurtosis(this IEnumerable<double> data)
{
	List<double> copy = new List<double>();
	foreach (double item in data) copy.Add(item);
	return _calculateKurtosis(copy);
}
```

Dla obliczania kurtozy została przygotowana metoda statyczna `CalculateKurtosis` która korzysta z prywatnej metody `_calculateKurtosis` (listing 2.9) implementującej wzór (2.10).

Testy

Metoda `CalculateKurtosis` została obłożona testami jednostkowymi `TestCalculateKurtosisDouble` i `TestCalculateKurtosisInt`. Wyniki referencyjne zostały uzyskane za pomocą RStudio.

2.8. Korelacja Pearsona

Opis

Test istotności współczynnika korelacji liniowej Pearsona służy do sprawdzania braku zależności liniowej między badanymi cechami dwóch populacji. Opiera się na współczynniku korelacji Pearsona. Im wartość współczynnika korelacji liniowej r_p jest bliższa zera, tym słabsza zależność wiąże te cechy.

Współczynnik korelacji Pearsona jest obliczany ze wzoru:

$$r_p = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i-\bar{y})^2}} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}, \qquad (2.11)$$

gdzie: x_i, y_i to kolejne wartości X i Y, $\bar{x}, \bar{y}$ – średnie z wartości cechy X i Y, a n – liczebność próby.

Statystyka testowa ma postać:

$$t = \frac{r_p}{SE}, \qquad (2.12)$$

gdzie: $SE = \sqrt{\frac{1-r_p^2}{n-2}}$.

Statystyka testowa nie może być wyznaczona, gdy $r_p = 1$ lub $r_p = -1$, albo n<3. Statystyka testowa ma rozkład t Studenta z $n - 2$ stopniami swobody.

Wymagania

- Badane cechy w populacji muszą mieć rozkład normalny.

Implementacja

Listing.2.10. Metoda obliczającą istotność współczynnika korelacji Pearsona

```
public struct Correlation
```

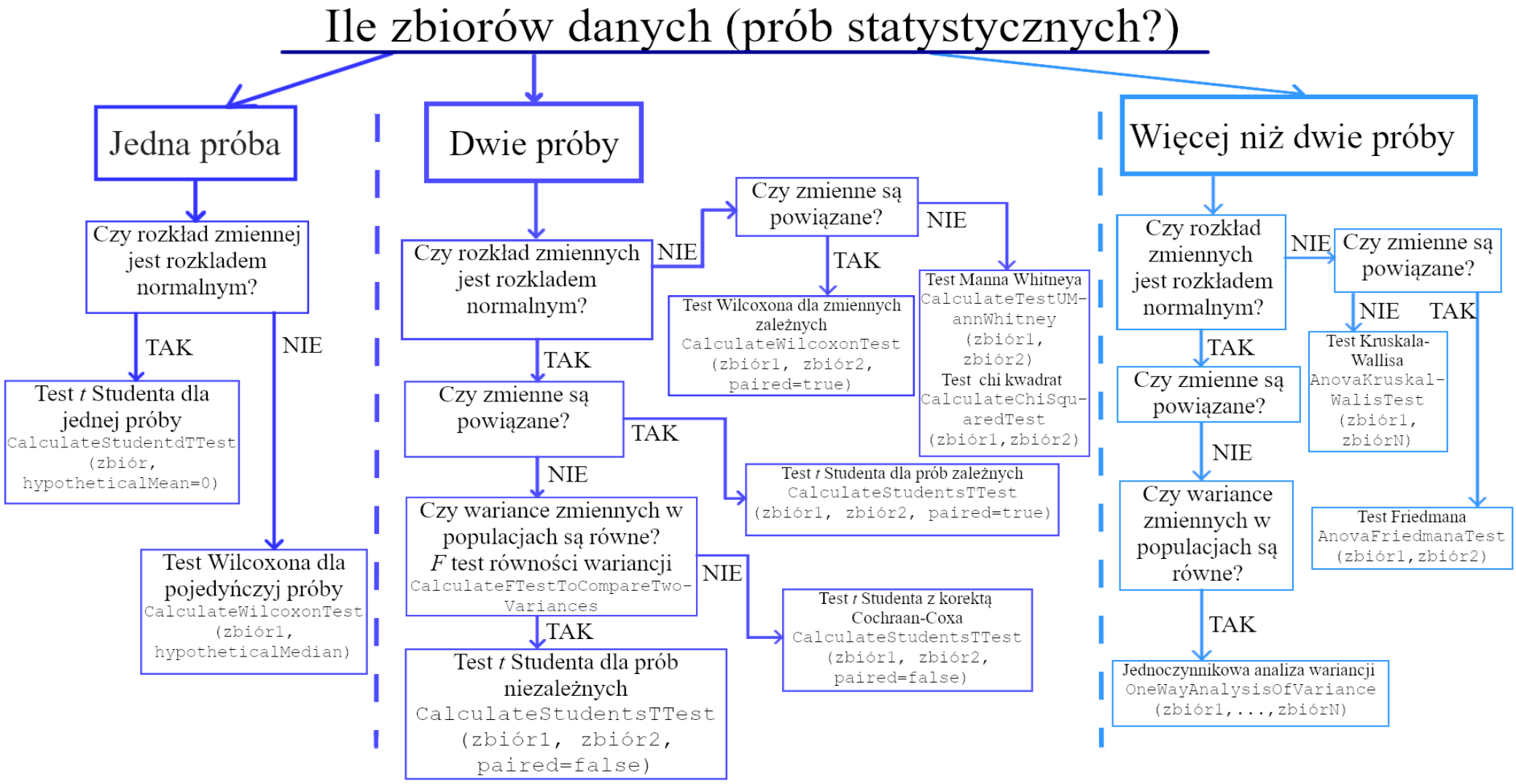
```
{
	public int SampleSize;
	public double CorrelationCoefficient;
	public double PValue;
	public double StudentsTValue;
}
private static Correlation _calculatePearsonsCorrelation(List<double> list1,
List<double> list2)
{
	if (list1.Count() != list2.Count()) throw new
SizeOutOfRangeException();
	int n = list1.Count();
	if (n == 0) throw new EmptyCollectionException();
	if (n < 3) throw new NotTheRightSizeException();
	double average1 = CalculateMean(list1);
	double average2 = CalculateMean(list2);

	double covariance = 0;
	double standardDeviation1 = 0;
	double standardDeviation2 = 0;
	for (int i = 0; i < n; ++i)
	{
		double s1 = (list1.ElementAt(i) - average1);
		double s2 = (list2.ElementAt(i) - average2);
		covariance += s1 * s2;
		standardDeviation1 += s1 * s1;
		standardDeviation2 += s2 * s2;
	}
	double degreeOfFreedom = n - 2.0;
	double r = covariance / Math.Sqrt(standardDeviation1 *
standardDeviation2);
	if (r >= 1 || r <= -1) throw new ArgumentException("Pearson
correlation coefficient must be in <-1;1>");
	double tDistribution = r * Math.Sqrt((n - 2) / (1 - r * r));
	double p = ContinuousDistribution.Student(tDistribution, n-2);
	double tValue = (average1-average2) / Math.Sqrt(standardDeviation1 *
standardDeviation1 + standardDeviation2 * standardDeviation2);
	double pForT = ContinuousDistribution.Student(tValue,
degreeOfFreedom);
	return new Correlation()
	{
		SampleSize = n,
		CorrelationCoefficient = Math.Round(r,7),
```

```
            PValue = Math.Round(p,5),
            StudentsTValue= tDistribution
        };
}
```

Korelacja Pearsona zaimplementowana w statycznej metodzie `CalculatePearsonsCorrelation` która korzysta z prywatnej metody `_calculatePearsonsCorrelation` (listing 2.10). Współczynnik korelacji jest obliczany ze wzoru (2.11), statystyka testowa (2.12). P-wartość jest obliczana z rozkładu t Studenta z $n - 2$ stopniami swobody.

Testy

Metoda `CalculatePearsonsCorrelation` została obłożona testami jednostkowymi `TestPearsonsCorrelationDouble` i `TestPearsonsCorrelationInt`. Sprawdzane jest także zgłaszanie wyjątków w przypadkach przesyłania pustego zbioru, gdy zbiory nie są równoliczne albo współczynnik korelacji nie mieści się w przedziale <-1;1>. Wyniki otrzymane za pomocą metody zostały porównane z wartościami uzyskanymi w RStudio przy pomocą funkcji `cor.test(x,y)`.



Rys.1. Schemat wszystkich testów statystycznych zaimplantowanych w pracy

3. Testy parametryczne

Testy parametryczne są używane do oceny wartości parametrów do danego rozkładu populacji, z którego jest losowana próba. Parametrami mogą być średnia wariancja czy odchylenie standardowe. Testy parametryczne cechują większa moc, dokładniejszy pomiar, lepsza interpretowalność uzyskanych wyników oraz większa liczba założeń, które muszą być spełnione.

3.1. F test równości wariancji

Opis

Test *F* jest używany do sprawdzenia tego, czy wariancje dwóch populacji są równe. Statystyka testowa ma postać ([3], str. 178):

$$F = \frac{S_X^2}{S_Y^2}, \qquad (3.1)$$

gdzie:

$$S_X^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2 \qquad (3.2)$$

Zakładamy, że X_1, ..., Xn i Y_1, ..., Y_m są niezależnymi i mający identyczny rozkład próbkami z dwóch populacji, z których każda ma rozkład normalny. Statystyka ma rozkład *F*, znany także jako rozkład F Snedecora, z $n - 1$ i $m - 1$ stopniami swobody. Parametr $n - 1$ jest często określany jako licznik stopni swobody, a parametr $m - 1$ jako mianownik. Należy pamiętać, że nie są one zamienne.

Wymagania

- Obydwie próby muszą mieć rozkład normalny.
- Obydwie próby muszą być losowe i reprezentatywne.

Implementacja

Listing.3.1. Metoda obliczającą test F dla zmiennych niezależnych

```
public struct FTest
```

```
{
        public double RatioOfVariances;
        public int NumDf;
        public int DenomDf;
        public double PValue;
}
public static FTest _calculateFTestToCompareTwoVariances(List<double> list1,
List<double> list2)
{
        int n = list1.Count();
        int m = list2.Count();
        if (n == 0 || m == 0) throw new EmptyCollectionException();
        double f = CalculateSampleMeans(list1) / CalculateSampleMeans(list2);

        double p =2* ContinuousDistribution.FCdf(f, n-1, m-1);
        return new FTest
        {
                RatioOfVariances=Math.Round(f,4),
                NumDf=n-1,
                DenomDf=m-1,
                PValue = Math.Round(p,5)
        };
}
```

Test równości wariancji zaimplementowany jest w statycznej metodzie `CalculateFTestToCompareTwoVariances` która korzysta ze statycznej prywatnej metody `_calculateFTestToCompareTwoVariances` (listing 3.1). Statystyka testowa jest wyliczana ze wzoru (3.1), za pomocą metody `CalculateSampleMeans`, która implementuje wzór (3.2). Do obliczenia *p*-wartości wykorzystujemy metodę `FCdf`(ang. *F Cumulative distribution function*)(zob. dodatek A.4), która znajduje się w klasie `ContinuousDistribution`. Metoda zwraca strukturę, która przechowuje statystykę testową (`RatioOfVariances`), stopnie swobody oraz *p*-wartość.

Testy

Metoda została obłożona testami jednostkowymi `TestFTestToCompareTwoVariancesInt` i `TestFTestToCompareTwoVariancesDouble`. Zostały przetestowane także przypadki przesyłania zbiorów takiej samej oraz różnej długości oraz przesyłania pustego zbioru. Wyniki

otrzymane za pomocą metody zostały porównane do wartości uzyskanych w RStudio przy pomocą funkcji `var.test(x,y)`.

3.2. Test t Studenta dla jednej zmiennej

Opis

Test *t* Studenta dla jednej próby wykorzystujemy, gdy chcemy porównać średnią „teoretyczną" ze średnią dla próby. Średnia teoretyczna to średnia pochodząca z innego badania lub wynikająca z rozważań teoretycznych.
Statystyka testowa ma postać:

$$t = \frac{\bar{x}-m}{s}\sqrt{n}, \qquad (3.3)$$

gdzie: $\bar{x}$ – średnia z próby, s – odchylenie standardowe, m – średnia teoretyczna, n – liczebność próby.
Statystyka testowa ma rozkład t Studenta z $n-1$ stopniami swobody. Kształt rozkładu t Studenta jest podobny do standaryzowanego rozkładu normalnego, ale ma dłuższe „ogony". Ze wzrostem liczby stopni swobody kształt rozkładu t Studenta przybliża się do kształtu rozkładu normalnego.

Wymagania

- Badana próba musi mieć rozkład normalny.
- Liczność próbki $n \geq 2$.
- Próba musi być losowa i reprezentatywna.

Implementacja

Listing.5.2. Metoda obliczającą test t Studenta dla jednej zmiennej

```
private static TestResult _calculateStudentsTTest(this IEnumerable<double>
list1, double hypotheticalMean)
{
      int n = list1.Count();
      if (n == 0) throw new EmptyCollectionException();
      double average = CalculateMean(list1);
      double standardDeviation = CalculateStandardDeviation(list1);
```

```
        double tforOneSample = ((average - hypotheticalMean) /
standardDeviation) *      Math.Sqrt(n);
        int df = 1;
        double p = ContinuousDistribution.Student(tforOneSample,n-1);
        return new TestResult()
        {
              TestValue = Math.Round(tforOneSample, 4),
              DegreesOfFreedom = df,
              PValue = Math.Round(p,5)
        };
}
public static TestResult CalculateStudentsTTest(this IEnumerable<double>
list, double hypotheticalMean = 0)
{
        List<double> copy = new List<double>();
        foreach (double item in list) copy.Add(item);
        return _calculateStudentsTTest(copy, hypotheticalMean);
}
```

Test t Studenta zaimplementowany jest w statycznej metodzie `CalculateStudentsTTest` która korzysta z statycznej prywatnej metody `_calculateStudentsTTest` (listing 3.2). Statystyka testowa (zmienna `tforOneSample`) jest obliczana ze wzoru (5.3), p-wartość obliczana jest z rozkładu t Studenta za pomocą metody `Student` (zob. dodatek A.2), która znajduje się w klasie `ContinuousDistribution` [7]. Do obliczania rozkładu t Studenta wykorzystano algorytm AMC numer 395. Ten algorytm ma trzy wersje. Pierwsza jest używana, gdy mamy niecałkowitą liczbę stopni swobody (tzn. typ `double`) lub *df* jest dużą liczbą całkowitą (*df* > *19*). Druga wersja dotyczy sytuacji, gdy liczba stopni swobody (*df*) jest małą liczbą całkowitą i *t* jest mniejsza od 4. Trzecia wersja dotyczy sytuacji, gdy *df* jest małą liczbą całkowitą a *t* jest duże. Wartość średniej teoretycznej domyślnie jest ustawiona na 0.

Testy

Metoda `CalculateStudentsTTest` została obłożona testami jednostkowymi `TestCalculateSingleSampleStudentsTTestDouble` i `TestCalculateSingleSampleStudentsTTestInt`. Zostały przetestowane także przypadki średniej „teoretycznej", dla średniej domyślnej(zero) oraz czy w przypadku przesyłania pustego zbioru zgłaszany jest wyjątek. Wyniki otrzymane za pomocą stworzonej

metody w zostały porównane do wartości uzyskanych w RStudio przy pomocą funkcji `t.test(x, mu=3)`.

3.3. Test t Studenta dla niezależnych zmiennych

Opis

Test t Studenta jest jednym z powszechniej stosowanych metod oceny różnic między średnimi w dwóch grupach. Niezależnymi nazywamy te zmienne wartość których zmieniamy, podczas gdy zmienne zależne są jedynie mierzone. Jedynymi warunkami stosowania tego testu jest normalność rozkładu zmiennych oraz brak istotnych różnic między wariancjami. Założenia normalności można sprawdzić za pomocą testu normalności.

Statystyka testowa ma postać:

$$t = \frac{\overline{x_1} - \overline{x_2}}{\sqrt{\frac{(n_1-1)sd_1^2+(n_2-1)sd_2^2}{n_1+n_2-2}\left(\frac{1}{n_1}+\frac{1}{n_2}\right)}} \qquad (3.4),$$

gdzie: $\overline{x_1}$ i $\overline{x_2}$ to średnie w pierwszej i drugiej próbie, n_1 i n_2 to liczebności pierwszej i drugiej próby, a sd_1^2 i sd_2^2 – wariancje w pierwszej i drugiej próbie.

Statystyka testowa ma rozkład t Studenta z $df = n_1 + n_2 - 2$ stopniami swobody.

Gdy wariancje badanych zmiennych w obu populacjach są różne to test *t* Studenta jest obliczany z poprawką Cochrana-Coxa. Statystyka testowa ma wtedy postać:

$$t = \frac{\overline{x_1} - \overline{x_2}}{\sqrt{\frac{sd_1^2}{n_1}+\frac{sd_2^2}{n_2}}} \qquad (3.5)$$

Statystyka testowa ma rozkład *t* Studenta z liczbą stopni swobody zaproponowaną przez Satterthwaite i jest wyliczana ze wzoru:

$$df = \frac{\left(\frac{sd_1^2}{n_1}+\frac{sd_2^2}{n_2}\right)^2}{\left(\frac{sd_1^2}{n_1}\right)^2\frac{1}{(n_1-1)}+\left(\frac{sd_2^2}{n_2}\right)^2\frac{1}{(n_2-1)}} \qquad (3.6)$$

Wymagania

- Uzyskane próby muszą być losowe i reprezentatywne,
- Obydwie próby muszą mieć rozkład normalny.
- Równość wariancji w obu zmiennych.

Implementacja

Listing.3.3. Metoda obliczającą test t Studenta dla zmiennych niezależnych

```
private static TestResult _calculateStudentsTTest(this IEnumerable<double>
list1,IEnumerable<double> list2,bool pairs=false)
{
	int n1 = list1.Count();
	int n2 = list2.Count();
	double average1 = CalculateMean(list1);
	double average2 = CalculateMean(list2);

	double standardDeviation1 = CalculateStandardDeviation(list1);
	double standardDeviation2 = CalculateStandardDeviation(list2);
	double t, df;

	if (!pairs)
	{
		double sd1 = (standardDeviation1 * standardDeviation1) /
(double)n1;
		double sd2 = (standardDeviation2 * standardDeviation2) /
(double)n2;
		double standardDeviationForTwoList = ((((double)n1 - 1.0) *
standardDeviation1 * standardDeviation1 + ((double)n2 - 1.0) *
standardDeviation2 * standardDeviation2) / ((double)n1 + (double)n2 - 2.0));
		if (sd1 == sd2)
		{
			t = (average1 - average2) /
(Math.Sqrt(standardDeviationForTwoList * ((1.0 / (double)n1) + (1.0 /
(double)n2))));
			df = n1 + n2-2;
		}
		else
		{
			t = (average1 - average2) / Math.Sqrt(sd1 + sd2);
			df = ((sd1 + sd2) * (sd1 + sd2)) / ((sd1 * sd1 * (1 /
((double)n1 - 1))) + (sd2 * sd2 * (1 / ((double)n2 - 1))));
		}
	}
```

```
        ...
        double p = ContinuousDistribution.Student(t, df);
        return new TestResult()
        {
                TestValue = Math.Round(t,4),
                DegreesOfFreedom= Math.Round(df,4),
                PValue= Math.Round(p, 4)
        };
}
```

Test t Studenta dla zmiennych niezależnych zaimplementowany jest w statycznej metodzie `CalculateStudentsTTest`, która korzysta ze statycznej prywatnej metody `_calculateStudentsTTest` (listing 3.3). Statystyka testowa jest obliczana ze wzoru (3.4), gdy wariancje badanych zmiennych są równe, i ze wzoru (3.5), gdy te wariancje nie mają podobnych wartości. *P*-wartość jest obliczana z rozkładu *t* Studenta, w metodzie `Student` (zob. dodatek A.2.), z klasy `ContinuousDistribution`. Stopnie swobody wyznaczane są w sytuacji, gdy wariancje są równe ze wzoru $df = n_1 + n_2 - 2$, a gdy nie są równe – ze wzoru (5.6).

Testy

Metoda `CalculateStudentsTTest` została obłożona testami jednostkowymi `TestCalculateStudentsTtestForIndependentGroups` i `TestCalculateStudentsTtestForIndependentGroupsInt`. Zostały przetestowane dla przypadku równolicznych kolekcji oraz dla kolekcji o różnej długości. Sprawdzane jest także to, czy w przypadku przesyłania pustego zbioru zgłoszony zostaje wyjątek. Wyniki otrzymane za pomocą metody w celu testowania zostały porównane do wartości uzyskanych w RStudio przy pomocą funkcji `t.test(x,y)`.

3.4. Test t Studenta dla zmiennych zależnych

Opis

Test *t* Studenta dla grup zależnych stosuje się w sytuacji, gdy dwie grupy obserwacji zostały oparte na tej samej grupie cech obiektów, ale zmierzonej dwukrotnie (np. przed i po ingerencji). Wówczas znacząca część zmienności wewnątrzgrupowej w obydwu grupach wyników może zostać wyjaśniona początkową indywidualną różnicą między obiektami (przy

czym zakładamy, że wariancje zmiennej w obu pomiarach są sobie bliskie). Test t Studenta wymaga, aby było spełnione założenie o normalności rozkładu w obu pomiarach.

Interesuje nas różnica pomiędzy parami pomiarów $(d_i = x_{1i} - x_{2i})$, którą wykorzystujemy do weryfikacji hipotezy, że średnia dla tej różnicy wynosi zero.

Statystyka testowa ma postać:

$$t = \frac{\bar{d}}{s_d}\sqrt{n}\,, \qquad (3.7)$$

gdzie: $\bar{d}$ – średnia różnic d_i w próbie, s_d – odchylenie standardowe różnic d_i w próbie, n – liczność różnic d_i w próbie.

Statystyka testowa ma rozkład t Studenta z $n - 1$ stopniami swobody.

Wymagania

- Badana próba musi być losowa i reprezentatywna,
- zbiór różnic d_i musi mieć rozkład normalny.

Implementacja

Listing.5.4. Metoda obliczającą test t Studenta dla zmiennych zależnych

```
private static TestResult _calculateStudentsTTest(this IEnumerable<double>
list1,IEnumerable<double> list2,bool pairs=false)
{
      int n1 = list1.Count();
      int n2 = list2.Count();
      double average1 = CalculateMean(list1);
      double average2 = CalculateMean(list2);

      double standardDeviation1 = CalculateStandardDeviation(list1);
      double standardDeviation2 = CalculateStandardDeviation(list2);
      double t, df;
      ...
      else
      {
            if (n1 != n2) throw new SizeOutOfRangeException();
            List<double> listOfPairs =
Util.DifferenceBetweenPairsOfMeasurements(list1, list2);
            double averageForPairs = CalculateMean(listOfPairs);
```

```
            double standardDeviationForPairs =
CalculateStandardDeviation(listOfPairs);
            t = averageForPairs / standardDeviationForPairs *
Math.Sqrt(listOfPairs.Count());
            df = n1 - 1;
      }
      double p = ContinuousDistribution.Student(t, df);
      return new TestResult()
      {
            TestValue = Math.Round(t,4),
            DegreesOfFreedom= Math.Round(df,4),
            PValue= Math.Round(p, 4)
      };
}
```

Test t Studenta dla zmiennych zależnych zaimplementowany jest w statycznej metodzie `CalculateStudentsTTest`, która korzysta ze statycznej prywatnej metody `_calculateStudentsTTest` (listing 3.4). Podczas wywoływania tej ostatniej metody trzeba ustawić parametr `pairs` na `true`. Jeżeli tego nie zrobimy, otrzymamy obliczania testu *t* Studenta dla wartości niezależnych. Kolekcje przesyłane do metody muszą być równoliczne, w przeciwnym przypadku zgłaszany jest wyjątek `SizeOutOfRangeException`. Statystyka testowa jest obliczana ze wzoru (3.7), a różnica pomiędzy parami pomiarów za pomcoą statycznej metody `DifferenceBetweenPairsOfMeasurements`, która znajduje się w klasie `Util`. Metoda ta zwraca listę elementów $x_{1i} - x_{2i}$.

Testy

Metoda `CalculateStudentsTTest` została obłożona testami jednostkowymi `TestCalculateSingleSampleStudentsTTestDouble` i `TestCalculateSingleSampleStudentsTTestInt`. Zostały przetestowane także przypadki dla średniej zadanej oraz domyślnej oraz to czy w przypadku przesyłania pustego zbioru lub zbiorów o różnym rozmiarze zgłaszane są wyjątki. Wyniki otrzymane za pomocą metody zostały porównane do wartości uzyskanych w RStudio przy pomocą funkcji `t.test(x,y,paired=TRUE)`.

3.5. Test chi-kwadrat dla jednej zmiennej

Opis

Test χ^2 zgodności (dobroci dopasowania) także nazywany testem χ^2 dla pojedynczej próby. Służy do testowania zgodności wartości obserwowanych dla n > 1 jednej cechy χ z hipotecznymi wartościami testowymi dla danej cechy. Statystyka testowa jest obliczana ze wzoru (4.1). Wartość oczekiwana dana jest przez $E_i = \frac{\sum_{i=1}^{n} X_i}{n}$. Statystyka testowa ma asymptotyczny rozkład chi-kwadrat z $n - 1$ stopniami swobody.

Wymagania

- Suma liczności obserwowanych powinna być taka sama jak i suma liczności oczekiwanych.
- Badana próba musi mieć rozkład normalny.
- Próba musi być losowa i reprezentatywna.

Implementacja

Listing.3.5. Metoda obliczającą test chi-kwadrat dla jednej zmienne

```
private static TestResult _calculateChiSquaredTest(List<double> list)
{
	int n = list.Count();
	if (n == 0) throw new EmptyCollectionException();
	double statistic = 0;
	double expectedA = list.Average();

	for (int i = 0; i < n; i++)
	{
		statistic += ((list.ElementAt(i) - expectedA) *
	(list.ElementAt(i) - expectedA)) / expectedA;
	}
	int df = n - 1;
	double pval = 1.0 - ContinuousDistribution.ChiSquareCdf(statistic,
df);
	return new TestResult
	{
		TestValue = Math.Round(statistic, 5),
		DegreesOfFreedom = df,
		PValue = Math.Round(pval, 4)
	};
}
public static TestResult CalculateChiSquaredTest(params IEnumerable<double>[]
args)
```

```
{
	if (args.Length == 1)
	{
		List<double> copy = new List<double>();
		foreach(IEnumerable<double> item in args)
		{
			foreach (double element in item)
				copy.Add(element);
		}
		return _calculateChiSquaredTest(copy);
	}
	else
	{
		List<double>[] copy = new List<double>[args.Length];
		for(int i=0; i<args.Length;i++)
		{
			copy[i] = new List<double>();
		}
		int j = 0;
		foreach (IEnumerable<double> list in args)
		{
			foreach(double element in list)
			{
				copy[j].Add(element);
			}
			j++;
		}
		return _calculateChiSquaredTest(copy);
	}
}
```

Test chi-kwadrat zaimplementowany w statycznej metodzie `CalculateChiSquaredTest` która korzysta z prywatnej metody `_calculateChiSquaredTest` (listing 3.5). Statystyka obliczana jest ze wzoru (4.1), a wartość oczekiwana faktycznie jest średnią. Metoda zwraca strukturę Test, która przechowuje wartości statystyki testowej, liczbę stopni swobody oraz *p*-wartość, wyliczaną z rozkładu chi-kwadrat. Ta ostatnia obliczana jest statyczną metodą `ChiSquareCdf` (zob. dodatek A.3), w klasie `ContinuousDistribution`.

Testy

Metoda `CalculateChiSquaredTest` została obłożona testami jednostkowymi `ChiSquaredTestDouble` oraz `ChiSquaredTestInt`. Sprawdzane jest także, czy w przypadku przesyłania pustej kolekcji zgłaszany jest wyjątek. Wartości uzyskane za pomocą metody `CalculateChiSquaredTest` są porównywane z wartościami uzyskanymi w RStudio przy pomocy funkcji `chisq.test`.

3.6. Jednoczynnikowa analiza wariancji

Opis

Jednoczynnikowa analiza wariancji (ang. *one-way analysis of variance*, ANOVA) służy do porównania średnich badanych zmiennych w kilku ($k \geq 2$) populacjach. Zazwyczaj jednoczynnikowa analiza wariancji jest używana do testowania różnic między co najmniej trzema grupami, ponieważ w przypadku dwóch grup można wykorzystać test *t* Studenta. W przypadku dwóch populacji, związek pomiędzy jednoczynnikową analizą wariancji i testem *t* Studenta można określić jako $F = t^2$.

Statystyka testowa *F* jest wyliczana ze wzoru:

$$F = \frac{MS_{BG}}{MS_{WG}}, \qquad (3.8)$$

gdzie $MS_{BG} = \frac{1}{df_{BG}} \sum_{j=1}^{k} n_j (\overline{x} - \hat{x})^2$ – średnia kwadratów między grupami,

$MS_{WG} = \frac{1}{df_{WG}} \sum_{j=1}^{k} \sum_{i=1}^{n_j} (x_{ij} - \overline{x_i})^2$ – średnia kwadratów wewnątrz grup,

$df_{BG} = k - 1$ – liczba stopni swobody między grupami,

$df_{WG} = (N - 1) - df_{BG}$ – liczba stopni swobody wewnątrz grup,

$N = \sum_{j=1}^{k} n_j$ – całkowita liczebność prób dla wszystkich populacji,

n_j – liczebność prób dla poszczególnych populacji (j=1, 2, ... k),

x_{ij} – wartości w próbach (i=1, 2, ... n_j) dla (j=1, 2, ... k),

$\overline{x_i}$ – średnia arytmetyczna z i-tej próby,

$\hat{x}$ – średnia arytmetyczna ze wszystkich populacji dla wszystkich prób k.

Statystyka ma rozkład *F*, znany także jako rozkład F Snedecora, z df_{BG} i df_{WG} stopniami swobody.

Wymagania

- Wszystkie badany populacje muszą mieć rozkład normalny.
- Próby pobrane z każdej populacji muszą być losowe i reprezentatywne.
- Wariancje w populacjach muszą być równe.

W przypadku, gdy nie są spełnione powyższe wymagania można posłużyć się testem Kruskala-Wallisa (zob. podrozdział 4.2).

Implementacja

Listing.3.6. Metoda obliczającą test chi-kwadrat dla jednej zmienne

```
public struct AnovaResult
{
	public double TestValue;
	public double Dfwg;
	public double Dfbg;

	public double MsWG;
	public double MsBG;

	public double SsWG;
	public double SsBG;

	public double PValue;
}
public static AnovaResult OneWayAnalysisOfVariance(params
IEnumerable<double>[] args)
{
	if (args.Length == 1) throw new ArgumentException("Need at least two
collection");
	foreach (IEnumerable<double> item in args)
	{
		if (item.Count()==0) throw new EmptyCollectionException();
	}
	int r = args.Length;
	int N=0;
	List<double> meanInEachGroup = new List<double>();
	double ssWG = 0;
	int nmean = 0;
	foreach (IEnumerable<double> list in args)
	{
		meanInEachGroup.Add(list.Average());
		for (int i = 0; i < list.Count(); i++)
		{
			ssWG += (list.ElementAt(i) -
meanInEachGroup.ElementAt(nmean))* (list.ElementAt(i) -
meanInEachGroup.ElementAt(nmean));
		}
```

```
            nmean++;
            N += list.Count();
      }
      double overallMean = meanInEachGroup.Sum() / meanInEachGroup.Count();
      double ssBG = 0;
      for (int k = 0; k < meanInEachGroup.Count(); k++)
            ssBG += args[k].Count() * (meanInEachGroup.ElementAt(k) -
overallMean) * (meanInEachGroup.ElementAt(k) - overallMean);
      double dfBG = r - 1;
      double msBG = ssBG / dfBG;
      double dfWG = N-1-dfBG;
      double msWG = ssWG / dfWG;
      double statistic = msBG / msWG;
      double p = ContinuousDistribution.FCdf(statistic, (int)dfBG,
(int)dfWG);

      return new AnovaResult
      {
            TestValue = Math.Round(statistic,3),
            Dfbg=dfBG,
            Dfwg=dfWG,
            MsBG=msBG,
            MsWG=msWG,
            SsBG=ssBG,
            SsWG=ssWG,
            PValue=p
      };
}
```

Jednoczynnikowa analiza wariancji jest zaimplementowany w statycznej metodzie `OneWayAnalysisOfVariance` (listing 3.6). Statystyka obliczana jest ze wzoru (3.8). Metoda zwraca strukturę `AnovaResult`, która przechowuje wartości statystyki testowej, sumę kwadratów, średnią kwadratów oraz liczbę stopni swobody między grupami i wewnątrz grup i *p*-wartość, wyliczaną z rozkładu chi-kwadrat. Ta ostatnia obliczana jest statyczną metodą `ChiSquareCdf` (zob. dodatek A.3.), w klasie `ContinuousDistribution`.

Testy

Metoda `OneWayAnalysisOfVariance` została obłożona testem jednostkowym `TestOneWayAnalysisOfVariance`. Wartości uzyskane za pomocą metody `OneWayAnalysisOfVariance` są porównywane do wartości uzyskanych w RStudio przy

pomocy funkcji `summary(aov(weight ~ group, data = dataFrame))`. Przetestowane zostały także przypadki dla dwóch i czterech parametrów jednakowej oraz różnej długości.

4. Testy nieparametryczne

Stosowanie testów nieparametrycznych nie polega na estymacji parametrów, takich jak odchylenie standardowe lub średnia arytmetyczna, które opisują rozkład danej zmiennej w populacji. Często na ich określenie stosowane są także nazwy „metody niezależne od parametrów" lub „niezależne od rozkładów". W związku z tym, że dane dla testów nieparametrycznych powinny spełniać mniej wymogów niż w przypadku testów parametrycznych, to ich moc oraz dokładność jest mniejsza. Te testy powinny być stosowane dopiero, gdy nie udaje się spełnić założeń testów parametrycznych.

Metody nieparametryczne najlepiej wykorzystywać dla zbiorów o małych rozmiarach. Dla zbiorów o dużych rozmiarach ($n > 100$) stosowanie takich testów najczęściej nie ma uzasadnienia. „Gdy liczność próby bardzo wzrasta, wówczas średnie prób podlegają rozkładowi normalnemu nawet w sytuacji, gdy odpowiednia zmienna w populacji nie posiada rozkładu normalnego lub nie jest wystarczająco dobrze zmierzona." [1]. Twierdzenie to nosi nazwę centralnego twierdzenia granicznego (termin ten został użyty po raz pierwszy przez Pólya, 1920).

4.1. Test chi-kwadrat

Opis

Test zgodności chi-kwadrat (χ^2) inaczej nazywany także testem Pearson, służy do zbadania związku pomiędzy dwoma zmiennymi nominalnymi. Bazuje on na porównaniu ze sobą wartości obserwowanych z wartościami oczekiwanymi. Jeżeli różnica pomiędzy tymi wartościami jest duża to możemy powiedzieć, że występuje relacja pomiędzy dwoma zmiennymi. Warunkiem stosowania testu jest odpowiednia liczba próbek. Przyjmuje się, że można go stosować, gdy wielkość populacji jest większa niż 30. To wynika z konieczności podziału populacji na klasy i warunku obecności w każdej klasie minimum 5 przypadków.

Ten test jest rozszerzeniem na 2 cechy testu zgodności chi-kwadrat (dobroci dopasowania), który jest przeznaczony dla jednej próby, bardziej dokładnie opisany w rozdziale 3.5.

Statystyka testowa ma postać:

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}, \qquad (4.1)$$

gdzie:

O - wartość obserwowana

E - Wartość oczekiwana

Wartość oczekiwana jest obliczana:

$$E_{oczekiwana} = \frac{r.sum * c.sum}{grand.total}, \qquad (4.2)$$

Po tym jak obliczyliśmy wartość statystyki musimy sprawdzić jej istotność. Dlatego obliczamy stopnie swobody, posługując się wzorem:

$$df = (r-1)(c-1), \qquad (4.2)$$

gdzie r i c to liczba poziomów zmiennych, czyli w naszym przypadku liczba wierszy(r) oraz kolumn(c).

Wymagania

- Liczebność próbek $n > 30$, w każdej klasie minimum 5.

Implementacja

Listing.4.1. Metoda obliczającą test chi-kwadrat dla parametrów niezależnych

```
private static TestResult _calculateChiSquaredTest(params List<double>[]
args)
{
      int nCols = args.Length;
      int nRows = args.FirstOrDefault().Count();
      foreach (List<double> list in args)
      {
            if (nRows != list.Count()) throw new SizeOutOfRangeException();
      }
      List<double> n = Enumerable.Repeat<double>(0, nRows).ToList();
      List<double> sumOfCols = new List<double>();

      foreach (IEnumerable<double> list in args)
      {
```

```
            for (int i = 0; i < nRows; i++)
            {
                  n[i] += list.ElementAt(i);
            }
            sumOfCols.Add(list.Sum());
      }
      double totalSum = sumOfCols.Sum();
      double statistic = 0;
      double expected;
      int rowNum = 0;
      foreach (IEnumerable<double> list in args)
      {
            for (int i = 0; i < nRows; i++)
            {
                  expected = (n.ElementAt(i) / totalSum) *
sumOfCols.ElementAt(rowNum);
                  statistic += ((list.ElementAt(i) - expected) *
(list.ElementAt(i) - expected)) / expected;
            }
            rowNum++;
      }
      int df = (nCols - 1) * (nRows - 1);
      double pval = 1.0 - ContinuousDistribution.ChiSquareCdf(statistic,
df);
      return new TestResult
      {
            TestValue = Math.Round(statistic, 5),
            DegreesOfFreedom = df,
            PValue = Math.Round(pval, 4)
      };
}
```

Test chi-kwadrat zaimplementowany został w statycznej metodzie `CalculateChiSquaredTest` która korzysta z prywatnej metody `_calculateChiSquaredTest`(listning 4.1). Metoda jest zaimplementowana w taki sposób, że może przyjmować nieokreśloną liczbę parametrów typu `List<double>`, które potem rozpatrywany jako macierz $r \times c$, w której r (*row*) to liczba elementów w liście (w jakiejkolwiek dlatego, że wszystkie listy muszą być takiego samego rozmiaru), a c (*column*) to liczba przesyłanych argumentów. Kolejność przesłanych argumentów nie ma znaczenia.

Statystyka testowa ma rozkład chi kwadrat z liczbą stopni swobody obliczanych ze wzoru (4.2).

Zakładamy, że poziom istotności wynosi 0.05, żeby mieć z czym je porównać obliczamy *p*-wartość (prawdopodobieństwo testowe) za pomocą funkcji ChisquareCdf (zob. dodatek A.3.) która oblicza rozkład chi-kwadrat, kształt, którego zależy od liczby stopni swobody (*df*). Ze wzrostem *df* kształt rozkładu χ^2 zbliża się do kształty rozkładu normalnego.

Metoda ta zwraca strukturę `TestResult`, której używamy dla wygody, a która pozwala na przekazanie trzech podstawowych parametrów: statystyki testowej, liczby stopni swobody oraz *p*-wartość, świadczącej o istotności.

Testy

Metoda `CalculateChiSquaredTest` została obłożona testami jednostkowymi `ChiSquaredTestDouble` oraz `ChiSquaredTestInt`. Wartości uzyskane za pomocą metody `CalculateChiSquaredTest` są porównywane do wartości uzyskanych w RStudio przy pomocy funkcji `chisq.test`. Przetestowane zostały także przypadki dla dwóch i trzech parametrów.

4.2. Test Kruskala-Wallisa

Opis

Ten test jest nieparametryczną alternatywą jednoczynnikowej analizy wariancji. Opisany został przez Kruskala i Wallisa [12] jako rozszerzenie testu U Manna-Whitneya na więcej niż dwie populacji. Ten test służy do porównywania dwóch lub więcej niezależnych próbek o jednakowej lub różnej długości, jest w szczególności użyteczny, kiedy wielkość próbki jest mała a dane nie są symetryczne.

Statystyka testowa ma postać [15]:

$$H = \frac{1}{C}\left(\frac{12}{N(N+1)}\sum_{j=1}^{k}\left(\frac{\left(\sum_{j=1}^{n_j} R_{ij}\right)^2}{n_j}\right) - 3(N+1)\right), \qquad (4.4)$$

gdzie:

$N = \sum_{j=1}^{k} n_i$,

n_j – liczność próby dla (j=1, 2, ...k),

R_{ij} – rangi przypisane do wartości zmiennej, dla (i=1, 2, ... n_j), (j=1, 2, ...k),

$C = 1 - \frac{\sum(t^3 - t)}{N^3 - N}$ – poprawka na rangi wiązane (powtarzającym się wartościom w zbiorze przypisuje się rangę wiązaną), gdzie t – liczba przypadków rangi wiązanej.

Ta poprawka jest stosowana wtedy, gdy w zbiorze występują rangi wiązane. W przeciwnym przypadku wartość C = 1.

Statystyka ma asymptotyczny rozkład chi-kwadrat z liczbą stopni swobody wyliczanej według wzoru: $df = (k - 1)$.

Wymagania

- Każda próba musi mieć co najmniej 5 elementów.
- Badane próby muszą być losowe i niezależne.

Implementacja

Listing.4.2. Metoda obliczającą test Kruskala-Wallisa

```
public static TestResult AnovaKruskalWalisTest(params IEnumerable<double>[]
args)
{
      if (args.Length == 1) throw new ArgumentException("Need at least two
collection");
      int n = 0;
      List<double> list = new List<double>();
      foreach(IEnumerable<double> el in args)
      {
            n += el.Count();
            list=list.Concat(el).ToList();
      }
      list = list.OrderBy(x => Math.Abs(x)).ToList();
      Dictionary<double, double> dictOfPairs = Ranks.CalculateRanks(list);
      double sumRij = 0;
      double totalSum = 0;
      foreach (IEnumerable<double> el in args)
      {
            for(int i=0;i<el.Count();i++)
            {
                  sumRij+= dictOfPairs[Math.Abs(el.ElementAt(i))];
            }
            totalSum += (sumRij * sumRij) / el.Count();
            sumRij = 0;
```

```
        }
        double correctionForTied = 1.0 - (Ranks.SumOfTiedPairs(list) / (n * n
* n - n));

        double kwScore = ((12.0 * totalSum) / (n * (n + 1.0)) - 3.0 * (n +
1.0));
        kwScore = kwScore / correctionForTied;
        int df = args.Length - 1;
        double pVal = 1.0 - ContinuousDistribution.ChiSquareCdf(kwScore, df);
        return new TestResult
        {
                TestValue = Math.Round(kwScore,4),
                PValue = Math.Round(pVal, 6),
                DegreesOfFreedom = df
        };
}
```

Test Kruskala-Wallisa jest zaimplementowany w statycznej metodzie `AnovaKruskalWalisTest` (listing 4.2), która znajduje się w klasie `ANOVA`. W przypadku przynajmniej jednej kolekcji lub gdy kolekcje nie mają takiego samego rozmiaru, zgłaszane są odpowiednie wyjątki. Metoda jest zaimplementowana zgodnie ze wzorem (4.4).

Testy

Metoda `calculateKruskalaWalisaTest` została obłożona testami jednostkowymi `TestKruskalaWalisaTestDouble`, `TestKruskalaWalisaTestInt`. Zostały przetestowane sytuacje, w których dane zamieniano miejscami, przesłana została jedna lub dwie kolekcje puste lub kolekcje nie są równoliczne.

4.3. Test Manna Whitney

Opis

Test U Manna-Whitneya, znany również jako test Wilcoxona Manna-Whitneya, służy do porównania różnic pomiędzy dwoma niezależnymi grupami danych (zakładamy, że rozkłady zmiennej są sobie bliskie – rozproszenie danych w porównanych populacjach jest takie same), gdy zależną zmienna jest porządkowa lub ciągła, ale nie ma rozkładu normalnego.

Statystyka testowa ma różne postaci, w zależności od wielkości próby. Dla malej liczności próby:

$$U = n_1 n_2 + \frac{n_1(n_1+1)}{2} - R_1 \qquad (4.5)$$

lub

$$U = n_1 n_2 + \frac{n_2(n_2+1)}{2} - R_2, \qquad (4.6)$$

gdzie n_1, n_2 to liczności prób, R_1, R_2 to sumy rang dla prób.
Statystyka testowa dla próby o dużej liczności to:

$$Z = \frac{U - \frac{n_1 n_2}{2}}{\sqrt{\frac{n_1 n_2 (n_1+n_2+1)}{12} - \frac{n_1 n_2 \sum(t^3-t)}{12(n_1+n_2)(n_1+n_2+1)}}} \qquad (4.7)$$

Statystyka testowa Z zawiera korektę na rangi wiązane $C = \frac{n_1 n_2 \sum(t^3-t)}{12(n_1+n_2)(n_1+n_2+1)}$. Korekta ta jest stosowana, gdy obecne są rangi wiązane. Jeżeli ich nie ma to C = 0.
Statystyka testowa Z ma asymptotyczny rozkład normalny.

Wymagania

- Badane próby muszą być losowe i niezależne

Implementacja

Listing.4.3. Metoda obliczającą test `U Manna-Whitneya`

```
private static WilcoxonResults _calculateTestUManaWhitney(List<double> list1,
List<double> list2)
{
      if (list1.Count() == 0 || list2.Count() == 0) throw new
EmptyCollectionException();
      int n1 = list1.Count();
      int n2 = list2.Count();
      List<double> list3 = list1.Concat(list2).OrderBy(x =>
Math.Abs(x)).ToList();
```

```
        Dictionary<double, double> dictOfPairs = Ranks.CalculateRanks(list3);
        double r1 = 0;
        double r2 = 0;

        double u1 = 0, u2 = 0;
        foreach(double item in list1)
        {
                r1+= dictOfPairs[Math.Abs(item)];
        }
        foreach (double item in list2)
        {
                r2 += dictOfPairs[Math.Abs(item)];
        }
        u1 = n1 * n2 + ((n1 * (n1 + 1.0)) / 2.0) - r1;

        u2 = n1 * n2 + ((n2 * (n2 + 1.0)) / 2.0) - r2;

        double correctionForTied = (n1 * n2 * Ranks.SumOfTiedPairs(list3)) /
(12.0 * (n1 + n2) * (n1 + n2 - 1.0));
        double zValue = (Math.Abs(u2 - (n1 * n2) / 2.0)-
0.5)/Math.Sqrt((n1*n2*(n1+n2+1.0))/12.0-correctionForTied);
        double p = 2 * ContinuousDistribution.Gauss(-Math.Abs(zValue));


        return new WilcoxonResults
        {
                T = u2,
                ZStatistic = zValue,
                PValue = Math.Round(p, 4)
        };
}
public static WilcoxonResults CalculateTestUMannWhitney(this
IEnumerable<double> list1, IEnumerable<double> list2)
{
        List<double> copy1 = new List<double>();
        List<double> copy2 = new List<double>();
        foreach (double item in list1) copy1.Add(item);
        foreach (double item in list2) copy2.Add(item);
        return _calculateTestUManaWhitney(copy1, copy2);
}
```

Test U Manna-Whitneya zaimplementowany jest w statycznej metodzie `CalculateTestUMannWhitney` która korzysta z prywatnej statycznej

metody `_calculateTestUManaWhitney` (listing 4.3). Statystyki testowe u_1 i u_2 obliczane są ze wzorów (4.5) i (4.6). Rangi są wyliczane dla `list3` która jest połączeniem dwóch otrzymanych próbek, dla wyliczania rang korzystamy z funkcji `CalculateRanks` (zob. dodatek B.2.) która znajduje się w klasie `Ranks`. Do obliczania *Z* stosowana jest poprawka na ciągłość. Stosuje się ją, aby zapewnić możliwość przejmowania przez statystykę testową wszystkich wartości liczb rzeczywistych zgodnie z założeniem rozkładu normalnego. Wzór na statystykę testową z poprawką na ciągłość ma postać:

$$Z = \frac{\left|U - \frac{n_1 n_2}{2}\right| - 0.5}{\sqrt{\frac{n_1 n_2 (n_1 + n_2 + 1)}{12} - \frac{n_1 n_2 \sum (t^3 - t)}{12(n_1 + n_2)(n_1 + n_2 + 1)}}} \qquad (4.8)$$

P-wartość wyliczamy przy pomocy funkcji `Gauss` (zob. dodatek A.1.) która znajduje się w klasie `ContinuousDistribution` i jest mnożona przez 2 dlatego, że wartość p wynosi $2 \times (T > t)$, gdzie T podąża naśladuję rozkład t z n-2 stopniami swobody.

Testy

Metoda `CalculateTestUMannWhitney` została obłożona testami jednostkowymi `TestUMannaWhitneyaDouble` i `TestUMannaWhitneyaInt`. Przetestowano sytuacje, gdy długości kolekcji są takie same i gdy są różne. Wartości zwracane przez metody zostały porównane z wartościami uzyskanymi w RSudio przy pomocy funkcji `wilcoxon.test(x,y,paired=FALSE)`. Została także przetestowana sytuacja, gdy jedna lub obie kolekcje są puste.

4.4. Test Wilcoxona dla jednej zmiennej

Opis

Test Wilcoxona dla rangowych znaków, znany jest też jako test Wilcoxona dla pojedynczej próby. Służy do sprawdzenia czy badana próba pochodzi z populacji, dla której mediana θ jest równa wskazanej wartości. W zależności od wielkości próby statystyka testowa ma różne postaci.

Dla prób o małym rozmiarze wykorzystywany jest wzór:

$$T = Max(\sum R_-, \sum R_+), \qquad (4.9)$$

gdzie R_- to suma rang ujemnych, a R_+ - suma rang dodatnich.

Dla prób o dużej liczebności używany jest wzór:

$$Z = \frac{\left|T - \frac{n(n+1)}{4}\right| - 0.5}{\sqrt{\frac{n(n+1)(2n+1)}{24} - \frac{\sum t^3 - \sum t)}{48}}} \qquad (4.10)$$

Statystyka testowa Z zawiera poprawkę na rangi wiązane. Stosujemy ją, żeby zapewnić możliwość przejmowania przez statystykę wszystkich wartość liczb rzeczywistych zgodnie z założeniem rozkładu normalnego. Wzór (4.10) zawiera korektę na rangi wiązane, gdy rangi wiązane nie występują w próbie, to wyrażenie w mianowniku $\frac{\sum t^3 - \sum t)}{48} = 0$, więc nie wpływa na statystykę.

Implementacja

Listing.4.4. Metoda obliczającą test Wilcoxona dla jednej próby

```
public struct WilcoxonResults
{
	public double T;
	public double ZStatistic;
	public double PValue;
}
private static WilcoxonResults _calculateWilcoxonTest(List<double> list,double
hypotheticalMedian=0)
{
	if (list.Count() == 0) throw new EmptyListException();
	int n = list.Count();
	Rank r;
	if (hypotheticalMedian!= 0)
		r = Ranks.CalculateRankForWilcoxonTest
(Util.DifferenceBetweenPairsOfMeasurements(list, hypotheticalMedian));
	else
		r = Ranks.CalculateRankForWilcoxonTest(list);

	double t = Math.Max(r.SumOfNegativeRanks, r.SumOfPositiveRanks);
	double nRanks = r.NumberOfRanks;
	double zValue = (Math.Abs(t - ((nRanks * (nRanks + 1.0)) / 4.0)) -
0.5) / Math.Sqrt(((nRanks * (nRanks + 1.0) * (2.0 * nRanks + 1.0)) / 24.0) -
r.CorrectionForTiedRanks);
```

```
        double p = 2*ContinuousDistribution.Gauss(-Math.Abs(zValue));
        return new WilcoxonResults
        {
                T = t,
                ZStatistic = zValue,
                PValue = Math.Round(p, 4)
        };
}
public static WilcoxonResults CalculateWilcoxonTest(this IEnumerable<double>
list, double hypotheticalMedian = 0)
{
        List<double> copy = new List<double>();
        foreach (double item in list) copy.Add(item);
        return _calculateWilcoxonTest(copy, hypotheticalMedian);
}
```

Listing.4.5. Metoda obliczającą rangi dla testu Wilcoxona

```
public static Rank CalculateRankForWilcoxonTest(this IEnumerable<double>
list)
{
        list = list.Where(x => x != 0);
        int n = list.Count();
        list = list.OrderBy(x => Math.Abs(x)).ToList();
        Dictionary<double, double> dictOfPairs = CalculateRanks(list);

        double sumPositive = 0;
        double sumNegative = 0;

        if (n == 1)
        {
                if (dictOfPairs[1] > 0)
                {
                        sumPositive++;
                }
                else
                {
                        sumNegative++;
                }
        }
        foreach (double item in list)
        {
```

```
            if (item > 0)
            {
                sumPositive += dictOfPairs[Math.Abs(item)];
            }
            else
            {
                sumNegative += dictOfPairs[Math.Abs(item)];
            }
        }

        double correctionForTied = (SumOfTiedPairs(list) / 48.0);

        return new Rank
        {
            NumberOfRanks = n,
            SumOfPositiveRanks = sumPositive,
            SumOfNegativeRanks = sumNegative,
            CorrectionForTiedRanks = correctionForTied
        };

    }
```

Test Wilcoxona dla jednej próby jest zaimplementowany w metodzie `CalculateWilcoxonTest`, która korzysta z prywatnej metody `_calculateWilcoxonTest` (listing 4.4). Metoda `_calculateWilcoxonTest` przyjmuje listę elementów typu `double` oraz weryfikowaną wartość mediany. Domyślna wartość mediany równa jest 0. Jeżeli wartość mediany jest zadana, to w celu wyliczenia rang musimy od każdego elementu listy odjąć medianę. Korzystamy do tego ze statycznej metody `DifferenceBetweenPairsOfMeasurements` (zob. dodatek B.1), która znajduję się w klasie `Util`. Żeby obliczyć rangi korzystamy ze statycznej metody `CalculateRankForWilcoxonTest` (listing 4.5), która znajduje się w klasie `Ranks`. Statystyka testowa jest obliczana ze wzoru (4.9), a statystyka testowa *Z* ze wzoru (4.10).

Testy

Metoda `CalculateWilcoxonTest` została obłożona testami jednostkowymi `TestCalculateWilcoxonTestDouble`, `TestCalculateWilcoxonTestInt`. Zostały przetestowane sytuacji dla kolekcji typu `int` i `double` oraz sytuacji, kiedy jest wartość mediany jest zadana i reprezentowana przez populacje. Wartości uzyskiwane w

metodzie `CalculateWilcoxonTest` porównywane do wartości uzyskanych w RStudio przy pomocy funkcji `wilcox.test(x,mu=2)`, mu – wartość mediany jeżeli ona nie będzie ustalona to domyślnie równa się zero.

4.5. Test Wilcoxona dla zmiennych zależnych

Opis

Test kolejności par Wilcoxona, znany również jako test Wilcoxona dla grup zależnych. Jest nieparametrycznym odpowiednikiem testu t Studenta dla grup zależnych. Stosujemy ten test, gdy pomiarów badanej zmiennej dokonujemy dwukrotnie w różnych warunkach. Ten test jest rozszerzeniem testu rangowanych znaków Wilcoxona opisanego w rozdziale 4.4. Nas interesuje różnica pomiędzy parami pomiarów $(d_i = x_{1i} - x_{2i})$ dla każdej wartości i indeksującego badane obiekty.

Wymagania

- Zależność grup, gdy pomiary wykonujemy kilkukrotnie dla tych samych obiektów.

Implementacja

Listing.4.6. Metoda obliczającą test Wilcoxona dla dwóch zależnych prób

```
private static WilcoxonTest _calculateWilcoxonTest(List<double>
list1,List<double> list2, bool pairs)
{
      if (list1.Count() == 0 || list2.Count() == 0) throw new
EmptyListException();
      if (pairs) {
            if(list1.Count()!=list2.Count())throw new
NotTheSameLengthException();
            Rank r =Ranks.CalculateRankForWilcoxonTest(
            Util.DifferenceBetweenPairsOfMeasurements(list1, list2));
            double t = Math.Max(r.SumOfNegativeRanks, r.SumOfPositiveRanks);
            double nRanks = (double)r.NumberOfRanks;

            double zValue = ((Math.Abs(t - ((nRanks * (nRanks +1))/4))) -
0.5) /            Math.Sqrt(((nRanks * (nRanks + 1) * (2 * nRanks +1))/24) -
                        r.CorrectionForTiedRanks);

            double p = 2 * ContinuousDistribution.Gauss(-Math.Abs(zValue));
```

```
            return new WilcoxonTest
            {
                  T = t,
                  ZStatistic = zValue,
                  PValue = Math.Round(p, 4)
            };
      }
      else
      {
            return CalculateTestUMannaWhitneya(list1, list2);
      }

}
public static WilcoxonTest CalculateWilcoxonTest(this IEnumerable<double>
list1, IEnumerable<double> list2, bool pairs = false)
{
      List<double> copy1 = new List<double>();
      List<double> copy2 = new List<double>();
      foreach (double item in list1) copy1.Add(item);
      foreach (double item in list2) copy2.Add(item);
      return _calculateWilcoxonTest(copy1, copy2,pairs);
}
```

Test kolejności par Wilcoxona jest zaimplementowany w statycznej metodzie `CalculateWilcoxonTest`, która korzysta z prywatnej metody `_calculateWilcoxonTest` (Lisiting.4.6.). Metoda jest prawie taka sama, jak metoda `_calculateWilcoxonTest` (Listining.4.4.) służąca do obliczania testu Wilcoxona dla jednej próby. Różni się tylko tym, że przyjmuje dwie kolekcje i oblicza różnicę pomiędzy parami obiektów. Różnice te są obliczane za pomocą statycznej metody `DifferenceBetweenPairsOfMeasurements` (zob. dodatek B.1.), która znajduje się w klasie `Util`. Statystykę testową i statystykę *Z* obliczamy odpowiednio przy pomocy wzorów (4.9) i (4.10). Żeby poprawnie skorzystać z danej metody trzeba obowiązkowo zaznaczyć argument funkcji `pairs` jako `true`. Domyślnie ustawiony jest na `false`, co oznacza, że obliczany będzie test Wilcoxona dla zmiennych niezależnych, czyli test U Manna Whitneya opisany w rozdziale 4.3.

Testy

Metoda `CalculateWilcoxonTest` została obłożona testami jednostkowymi `TestWilcoxonMatchedPairsTestDouble` i `TestWilcoxonMatchedPairsTestInt`. Zostały przetestowane także przypadki, gdy kolekcje nie są równoliczne oraz gdy jedna z kolekcji jest pusta. Wyniki otrzymane w metodzie `CalculateWilcoxonTest` porównywane zostały do wyników otrzymane w RStudio za pomocą funkcji `wilcox.test(x,y, paired=TRUE)`.

4.6. Korelacja Spearmana

Opis

Korelacja rang wprowadzona przez Spearmana jest jedną z najstarszych i najbardziej znanych procedur nieparametrycznych [14]. Współczynnik korelacji rangowej Spearmana r_s służy do badania siły związku zależności liniowej pomiędzy cechami X i Y. Wyznacza się go dla skali interwałowej(przedziałowej) lub porządkowej.
Wartość współczynnika jest wyliczana ze wzoru:

$$r_s = \frac{6\sum_{i=1}^{n} d_i^2}{n(n^2-1)}, \qquad (4.11)$$

gdzie: $d_i = R_{xi} - R_{yi}$ to różnica rang dla cechy X i Y, a n – liczebność próby d_i.
Jeżeli w jednym ze zbiorów występują rangi wiązane, to wzór (4.11) zastępowany jest przez:

$$r_s = \frac{\sum X + \sum Y - \sum_{i=1}^{n} d_i^2}{2\sqrt{\sum X \sum Y}} \qquad (4.12)$$

gdzie: $\sum X = \frac{n^3-n-T_X}{12}$, $\sum Y = \frac{n^3-n-T_Y}{12}$, $T_X = \sum_{i=1}^{s}(t_{i_{(X)}}^3 - t_{i_{(X)}})$, $T_Y = \sum_{i=1}^{s}(t_{i_{(Y)}}^3 - t_{i_{(Y)}})$, a t to liczba przypadków wchodzących w rangę wiązaną.

Wartość współczynnika korelacji rangowej mieści się w przedziale <-1;1> i interpretowany jest w następujący sposób:

- $r_s \approx 1$ oznacza silną dodatnią zależność monotoniczną, czyli wzrostowi zmiennej zależnej odpowiada wzrost zmiennej niezależnej.
- $r_s \approx -1$ oznacza silną ujemną zależność monotoniczną, czyli spadku zmiennej zależnej odpowiada wzrost zmiennej niezależnej.

- $r_s \approx 0$ między badanymi parametrami nie istnieje zależność monotoniczna.

Test *t* służący do sprawdzenia istotności współczynnika korelacji rangowej Spearmana pozwala na weryfikację hipotezy o braku zależności monotonicznej pomiędzy badanymi cechami populacji. Ten test opiera się na współczynniku korelacji rangowej Spearmana. Statystyka testowa ma postać:

$$t = \frac{r_s\sqrt{n-2}}{\sqrt{1-r_s^2}} \qquad (4.13)$$

Statystyka testowa nie może być wyznaczona, gdy $r_p = 1$ lub $r_p = -1$, albo gdy $n < 3$. Statystyka testowa ma rozkład t Studenta z $n - 2$ stopniami swobody.

Wymagania

- Równa liczność populacji dla badanych kolekcji.

Implementacja

Listing.4.7. Metoda obliczającą istotność współczynnika korelacji Spearmana

```
private static Correlation _calculateSpearmanCorrelation(List<double> list1,
List<double> list2)
{
      if (list1.Count() != list2.Count()) throw new
SizeOutOfRangeException();
      int n = list1.Count();
      if (n == 0) throw new EmptyCollectionException();
      if (n < 3) throw new NotTheRightSizeException();
      List<double> d = new List<double>();
      Dictionary<double, double> xRanks = Ranks.CalculateRanks(list1);
      Dictionary<double, double> yRanks = Ranks.CalculateRanks(list2);
      double rSum = 0;

      for (int i = 0; i < n; i++)
      {
            d.Add(xRanks[Math.Abs(list1.ElementAt(i))] -
      yRanks[Math.Abs(list2.ElementAt(i))]);
            rSum += d.ElementAt(i) * d.ElementAt(i);
      }

      double r;
```

```
        int nd = d.Count();
        if(Ranks.SumOfTiedPairs(list1)==0 && Ranks.SumOfTiedPairs(list2) == 0)
        {
                r= 1.0 - (6 * rSum) / ((double)nd * ((double)nd * (double)nd -
1.0));
        }
        else
        {
                double sumX = (nd * nd * nd - nd - Ranks.SumOfTiedPairs(list1))
/ 12.0;
                double sumY = (nd * nd * nd - nd - Ranks.SumOfTiedPairs(list2))
/ 12.0;
                r = (sumX + sumY - rSum) / (2 * Math.Sqrt(sumX*sumY));
        }
        double t = r * Math.Sqrt(((double)n - 2.0) / (1.0 - r * r));
        double pval =ContinuousDistribution.Student(t,n-2);

        return new Correlation
        {
                CorrelationCoefficient = Math.Round(r, 6),
                StudentsTValue = Math.Round(t, 6),
                PValue = Math.Round(pval, 5)
        };
}
public static Correlation CalculateSpearmanCorrelation(this
IEnumerable<double> list1, IEnumerable<double> list2)
{
        List<double> copy1 = new List<double>();
        List<double> copy2 = new List<double>();
        foreach (double item in list1) copy1.Add(item);
        foreach (double item in list2) copy2.Add(item);
        return _calculateSpearmanCorrelation(copy1, copy2);
}
```

Korelacja Spearmana zaimplementowana została w statycznej metodzie `CalculateSpearmanCorrelation`, która korzysta z prywatnej metody `_calculateSpearmanCorrelation` (Listining.4.7.). Współczynnik korelacji jest obliczany ze wzoru (4.11) w przypadku, gdy nie występują rangi wiązane i ze wzoru (4.12) gdy rangi występują w jednej lub w obu kolekcjach. Statystyka testowa dana jest wzorem (4.13). *P*-wartość jest obliczana z rozkładu *t* Studenta (zob. dodatek A.2.) z $n - 2$ stopniami swobody.

Testy

Metoda `CalculateSpearmanCorrelation` została obłożona testami jednostkowymi `TestSpearmanCorrelationDouble` i `TestSpearmanCorrelationInt`. W przypadku przesyłania pustego zbioru lub gdy zbiory nie są równoliczne zgłaszane są wyjątki. Wyniki otrzymane za pomocą przygotowanej metody zostały porównane do wartości uzyskanych w Rstudio przy pomocą funkcji `cor.test(x,y,method="spearman")`.

4.7. Test Friedmana

Opis

Test Friedmana jest nieparametrycznym testem statystycznym opracowanym przez Miltona Friedmana [15]. Służy do jednokierunkowej analizy powtarzanych pomiarów według rang. Stosuje się go, gdy pomiary wykonujemy kilkukrotnie ($p \geq 2$) w różnych warunkach. Statystyka testowa jest obliczana ze wzoru:

$$Q = \frac{1}{C}\left(\frac{12}{np(p+1)}\sum_{j=1}^{n}\left(\sum_{i=1}^{n} r_{ij}\right)^2 - 3n(p+1)\right), \qquad (4.14)$$

gdzie r_{ij} to rangi przypisane kolejnym pomiarom (j = 1, 2, …, p), oddzielnie dla każdego z badanych obiektów (i= 1, 2, …, n), n – liczność próby, p – liczba wykonanych pomiarów. $C = 1 - \frac{\sum_i (t_i^3 - t_i)}{n(p^3 - p)}$ to korekta na rangi wiązane (powtarzającym się wartościom w zbiorze przypisuje się rangę wiązaną), gdzie t_i – liczba przypadków w i-tym zestawie danych.
Ta poprawka jest stosowana wtedy, gdy w zbiorze występują rangi wiązane. W przeciwnym przypadku wartość C = 1.
Statystyka ma asymptotyczny rozkład chi-kwadrat z liczbą stopni swobody wyliczanej według wzoru: $df = (p - 1)$.

Wymagania

- Liczebność próbek n > 15 lub każdej klasie minimum 5.
- Minimum dwa zestawy danych

Implementacja

Listing.4.8. Metoda obliczającą test Friedmana.

```
public static TestResult AnovaFriedmanaTest(params IEnumerable<double>[]
args)
{
	int n = args.FirstOrDefault().Count();
	int p= args.Length;
	if (args.Length == 1) throw new ArgumentException("Need at least two
collection");
	foreach (IEnumerable<double> item in args)
	{
		if (n != item.Count()) throw new SizeOutOfRangeException();
	}
	List<double> list = new List<double>();
	List<double> sumRj = Enumerable.Repeat<double>(0, p).ToList();
	Dictionary<double, double> dictOfPairs= new Dictionary<double,
double>();

	double sumCorrectionForTiedRanks = 0;
	for (int i = 0; i < n; i++)
	{
		foreach(IEnumerable<double> el in args)
		{
			list.Add(el.ElementAt(i));
		}
		sumCorrectionForTiedRanks += Ranks.SumOfTiedPairs(list);
		dictOfPairs =Ranks.CalculateRanks(list);
		for(int j = 0; j < p; j++)
		{
			sumRj[j]+= dictOfPairs[Math.Abs(list.ElementAt(j))];
		}
		list.Clear();
	}
	double totalSum = 0;
	foreach(double element in sumRj)
	{
		totalSum += element * element;
	}
	double kwScore = ((12.0 * totalSum) / (n*p * (p + 1.0)) - 3.0 * n*(p +
1.0));
	int df = (int)p- 1;
```

```
        double correctionForTied = 1.0 - sumCorrectionForTiedRanks / (n *( p *
p*p - p));
        double statistics = kwScore / correctionForTied;
        double pVal = 1.0 - ContinuousDistribution.ChiSquareCdf(statistics,
df);
        return new TestResult
        {
                TestValue = Math.Round(statistics, 4),
                PValue = Math.Round(pVal, 5),
                DegreesOfFreedom = df
        };
}
```

Test Friedmana jest zaimplementowany w statycznej metodzie `AnovaFriedmanaTest` (listing 4.8), która znajduje się w klasie `ANOVA`. W przypadku przesyłania jednej kolekcji lub gdy kolekcje nie mają takiego samego rozmiaru, zgłaszane są odpowiednie wyjątki. Metoda jest zaimplementowana zgodnie ze wzorem (4.14).

Testy

Metoda `AnovaFriedmanaTest` została obłożona testem jednostkowym `TestFriedmanTest`. Wartości uzyskane za pomocą metody `AnovaFriedmanaTest` są porównywane do wartości uzyskanych w RStudio przy pomocy funkcji `friedman.test`. Przetestowane zostały także przypadki dla trzech i czterech parametrów.

5. Testy normalności rozkładu

5.1. Test Kołmogorowa-Smirnowa

Opis

Test normalności Kołmogorowa-Smirnowa należy do podklasy statystyk dobroci dopasowania, tak zwanych statystyk EDF (ang. *Empirical Distribution Function*). Statystyki, które bazują na porównaniu funkcji dystrybucji skumulowanej populacji [9]. Ten test jest stosowany dla większych prób ($N > 100$). Dla mniejszych stosuje się test Shapiro-Wilka przedstawiony w kolejnym podrozdziale.

Statystyka Kolmogorowa-Smirnowa obliczana jest:

$$D = \max_x \left\{ \left| \frac{i+1}{n} - F(X_{(i)}) \right|, \left| F(X_{(i)}) - \frac{i}{n} \right| \right\} \qquad (5.1)$$

gdzie funkcja $F(X_{(i)})$ – empiryczna dystrybuanta rozkładu normalnego, wyliczana w poszczególnych punktach rozkładu:

$$F(X_{(i)}) = \Phi\left(\frac{X_i - \mu}{\sigma}\right) \qquad (5.2)$$

Funkcja Φ(x) jest skumulowaną funkcją gęstości (ang. *cumulative distribution function*, CDF) standardowej normalnej zmiennej losowej.

Wymagania

- Próbki muszą być losowe i reprezentatywne.

Implementacja

Listing.5.1. Metoda obliczającą test Kołmogorowa-Smirnowa

```
private static TestResult
_calculateKolmogorovSmirnovTestForNormality(List<double> list)
{
      if (list.Count() == 0) throw new EmptyCollectionException();
      list.OrderBy(x => x);
      int n = list.Count();
      List<double> Di = new List<double>();
```

```
        List<double> D_i = new List<double>();

        double average = list.Average();
        double standartDeviation = CalculateStandardDeviation(list);
        for (int i = 0; i < n; i++)
        {
                Di.Add(Math.Abs((((double)i+1.0)/(double)n)-
Util.Phi((list.ElementAt(i)-average)/standartDeviation)));
                D_i.Add(Math.Abs(Util.Phi((list.ElementAt(i) - average) /
standartDeviation) - ((double)i / (double)n)));
        }
        double d = (Math.Max(Di.Max(), D_i.Max()));

        double pValue = 0;
        double zz = d*d*(double)n;
        for (int i =1; i < 1000; i++)
        {
                pValue += (Math.Pow((-1.0), ((double)i - 1.0))) * (Math.Exp((-
2.0) * ((double)i * (double)i) * zz));
        }
        double p =2.0 * pValue;
        return new TestResult
        {
                TestValue = Math.Round(d, 5),
                PValue= Math.Round(p, 4)
        };
}
public static TestResult CalculateKolmogorovSmirnovTestForNormality(this
IEnumerable<double> list)
{
        List<double> copy = new List<double>();
        foreach (double item in list) copy.Add(item);
        return _calculateKolmogorovSmirnovTestForNormality(copy);
}
```

Listing.5.2. Metoda obliczającą funkcje Φ(*x*)

```
public static double Phi(double x)
{
        double a1 = 0.254829592;
        double a2 = -0.284496736;
        double a3 = 1.421413741;
        double a4 = -1.453152027;
```

```
        double a5 = 1.061405429;
        double p = 0.3275911;

        int sign = 1;
        if (x < 0)
              sign = -1;
        x = Math.Abs(x) / Math.Sqrt(2.0);

        double t = 1.0 / (1.0 + p * x);
        double y = 1.0 - (((((a5 * t + a4) * t) + a3) * t + a2) * t + a1) * t
* Math.Exp(-x * x);

        return 0.5 * (1.0 + sign * y);
}
```

Test Kołmogorowa-Smirnowa jest zaimplementowany w metodzie `CalculateKolmogorovSmirnovTest` która korzysta z prywatnej metody `_calculateKolmogorovSmirnovTest` (listing 5.1). W przypadku przesyłania pustej kolekcji danych, zgłoszony będzie wyjątek `EmptyListException`.

Zmienne `Di` oraz `D_i` obliczane są za pomocy wzoru (5.1), funkcja $\Phi(x)$ jest zaimplementowana w metodzie `Phi` (listing 5.2) w klasie `Util`. *P*-wartość dla tego testu to prawdopodobieństwo wyrażone wzorem:

$$P(D > d) = 2\sum_{i=1}^{\infty}(-1)^{(i-1)e^{-2i^2z^2}} \qquad (5.3)$$

Testy

Metoda `CalculateKolmogorovSmirnovTestForNormality` została obłożona testami jednostkowymi `TestKolmogorovSmirnovTestDouble` i `TestKolmogorovSmirnovTestInt`. Wyniki otrzymane dzięki tej metodzie są porównywane do wyników otrzymanych w RStudio za pomocą funkcji `ks.test(x, "pnorm", mean=mean(d1), sd=sd(d1))`.

5.2. Test Shapiro-Wilka

Opis

Test normalności Shapiro-Wilka, służy do oceny tego, czy zebrane przez nas dane mają rozkład normalny. Ten test, podobnie jak test Kołmogorowa-Smirnowa, należy wykonywać przed analizą statystyczną, aby rozstrzygnąć, czy należy użyć testów parametrycznych, czy nieparametrycznych.

Statystyka testowa Shapiro-Wilka opisana jest wzorem:

$$W = \frac{\left(\sum_{i=1}^{n} a_i x_i\right)^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \qquad (5.4)$$

Gdzie: a_i – współczynniki wyznaczane w oparciu o wartości oczekiwane dla statystyk uporządkowanych (próba niezależnych zmiennych losowych posortowanych w porządku rosnącym), przypisanych wag oraz macierzy kowariancji, $\bar{x}$ – średnia danych z próby.

Statystyka ta przekształca się do statystyki o rozkładzie normalnym:

$$Z = \frac{w - \mu}{\sigma} \qquad (5.5)$$

Dla statystyki (4.13) parametr w(przekształcone W według wzorów (5.6) i (5.10)), μ i σ zależą od wielkości próby [10]. Dla prób o małych rozmiarach ($n \in < 4; 12$)), stwierdzono, że trójparametrowy rozkład logarytmiczno-normalny pasuje do empirycznego rozkładu $\ln(1 - W)$, rozkładu $\ln(1 - W)$, a dwuparametrowy logarytm normalny dopasowany do górnej połowy rozkładu $(1 - W)$ dla n > 11. Dla $n \in < 4; 12$):

$$w = -\ln[\gamma - \ln(1 - W)] \qquad (5.6)$$

$$\gamma = -2.273 + 0.459n \qquad (5.7)$$

$$\mu = 0.5440 - 0.39978n + 0.025054n^2 - 0.0006714n^3 \qquad (5.8)$$

$$\sigma = \exp(1.3822 - 0.77857n + 0.062767n^2 - 0.0020322n^3) \qquad (5.9)$$

Dla $n \in < 4; 5000 >$:

$$u = \ln n$$

$$w = \ln(1 - W) \qquad (5.10)$$

$$\mu = -1.5861 - 0.31082u - 0.083751u^2 + 0.0038915u^3 \qquad (5.11)$$

$$\sigma = \exp(-0.4803 - 0.082676u + 0.0030302u^2) \qquad (5.12)$$

Ten test charakteryzuje wysoka moc, najsilniejsza jest przeciwko rozkładom o krótkich ogonach (ang. *short-tailed*) oraz skośnych, a najsłabsze w porównaniu z rozkładamy symetrycznymi.

Wymagania

- Liczność próbek $n > 3$ i $n < 5000$.
- Próbki muszą być losowe i reprezentatywne.

Implementacja

Listing.5.3. Metoda obliczającą test Shapiro-Wilka

```
private static TestResult _calculateShapiroWilkTestForNormality(List<double>
list)
{
	if (n == 3)
	{
		a[1] = Math.Sqrt(0.5);
	}
	else
	{
		an25 = an + 0.25;
		summ2 = 0.0;
		for (i = 1; i <= nn2; i++)
		{
			a[i] = ContinuousDistribution.NormalQuantile((i - 0.375) /
an25, 0, 1);
			summ2 += a[i] * a[i];
		}
		summ2 *= 2.0;
		ssumm2 = Math.Sqrt(summ2);
		rsn = 1.0 / Math.Sqrt(an);
		a1 = poly(c1, 6, rsn) - a[1] / ssumm2;

		if (n > 5)
		{
			i1 = 3;
			a2 = -a[2] / ssumm2 + poly(c2, 6, rsn);
```

```
                    fac = Math.Sqrt((summ2 - 2.0 * (a[1] * a[1]) - 2.0 * (a[2]
* a[2])) / (1.0 - 2.0 * (a1 * a1) - 2.0 * (a2 * a2)));
                    a[2] = a2;
             }
             else
             {
                    i1 = 2;
                    fac = Math.Sqrt((summ2 - 2.0 * (a[1] * a[1])) / (1.0 - 2.0
* (a1 * a1)));
             }
             a[1] = a1;
             for (i = i1; i <= nn2; i++)
             {
                    a[i] /= -fac;
             }
      }
. . .
      sa /= n;
      sx /= n;
      ssa = ssx = sax = 0;
      for (i = 0, j = n - 1; i < n; i++, j--)
      {
             if (i != j)
             {
                    asa = sign(i - j) * a[1 + Math.Min(i, j)] - sa;
             }
             else
             {
                    asa = -sa;
             }
             xsx = list.ElementAt(i) / range - sx;
             ssa += asa * asa;
             ssx += xsx * xsx;
             sax += asa * xsx;
      }
      ssassx = Math.Sqrt(ssa * ssx);
      w1 = (ssassx - sax) * (ssassx + sax) / (ssa * ssx);
      double w = 1.0 - w1;
. . .
}
public static TestResult CalculateShapiroWilkTest(this IEnumerable<double>
list)
{
```

```
        List<double> copy = new List<double>();
        foreach (float item in list) copy.Add(item);
        return _calculateShapiroWilkTest(copy);
}
```

Listing.5.4. Metoda obliczającą funkcje kwantylową dla rozkładu normalnego.

```
public static double NormalQuantile(double p, double mu, double sigma)
{
        if (sigma < 0)
        {
                throw new ArgumentException("The sigma parameter must be
positive.");
        }
        else if (sigma == 0)
        {
                return mu;
        }

        double r;
        double val;

        double q = p - 0.5;

        if (0.075 <= p && p <= 0.925)
        {
                r = 0.180625 - q * q;
                val = q * (((((((r * 2509.0809287301226727 +
33430.575583588128105) * r + 67265.770927008700853) * r
+ 45921.953931549871457) * r + 13731.693765509461125) * r +
1971.5909503065514427) * r + 133.14166789178437745) * r
+ 3.387132872796366608) / (((((((r * 5226.495278852854561 +
28729.085735721942674) * r + 39307.89580009271061) * r
+ 21213.794301586595867) * r + 5394.1960214247511077) * r +
687.1870074920579083) * r + 42.313330701600911252) * r + 1);
        }
        else
        {
                if (q > 0)
                {
                        r = 1 - p;
                }
                else
```

```
            {
                  r = p;
            }

            r = Math.Sqrt(-Math.Log(r));

            if (r <= 5.0)
            {
                  r += -1.6;
                        val = (((((((r * 7.745450142783414076e-4 +
0.022723844989269184583) * r + 0.24178072517745061177) * r
+ 1.2704582524523683826) * r + 3.6478483247632046050) * r +
5.7694972214606914055) * r
+ 4.6303378461565452959) * r + 1.4234371107496835773) / (((((((r *
1.0507500716444168432e-9 + 5.4759380849953449460e-4) * r
+ 0.015198666563616457197) * r + 0.14810397642748007459) * r +
0.68976733498510000455) * r + 1.6763848301838038494) * r
+ 2.0531916266377588218) * r + 1.0);
            }
            else
            {
                  r += -5.0;
                  val = (((((((r * 2.0103343992922881326e-7 +
2.7115555687434875781e-5) * r + 0.00124266094738807843860) * r
+ 0.026532189526576123093) * r + 0.29656057182850489123) * r +
1.7848265399172913358) * r + 5.4637849111641143699) * r
+ 6.6579046435011037772) / (((((((r * 2.0442631033899397856e-15 +
1.4215117583164458887e-7) * r
+ 1.8463183175100546818e-5) * r + 7.8686931114561325910e-4) * r +
0.014875361290850614852) * r
+ 0.13692988092273580531) * r + 0.59983220655588793769) * r + 1.0);
            }

            if (q < 0.0)
            {
                  val = -val;
            }
      }
      return mu + sigma * val;
}
```

Test Shapiro-Wilka jest zaimplementowany w metodzie `CalculateShapiroWilkTest` która korzysta z prywatnej metody `_calculateShapiroWilkTest`. W przypadku przesyłania kolekcji danych mniejszej niż 3 i większej niż 5000, zgłoszony będzie wyjątek `SizeOutOfRangeException`. Statystyka testowa jest obliczana ze wzoru (5.4) i jest zwracana przez test w postaci $1 - w$, w celu uniknięcia nadmiernego zaokrąglenia dla W bardzo bliskiego 1 (potencjalny problem dla bardzo dużych próbek) (listing 5.3). Współczynnik a_i jest wyliczany przy pomocy statycznej metody `NormalQuantile`(listing 5.4), która oblicza funkcję kwantylową dla rozkładu normalnego, znajdującej się w klasie `ContinuousDistribution`.

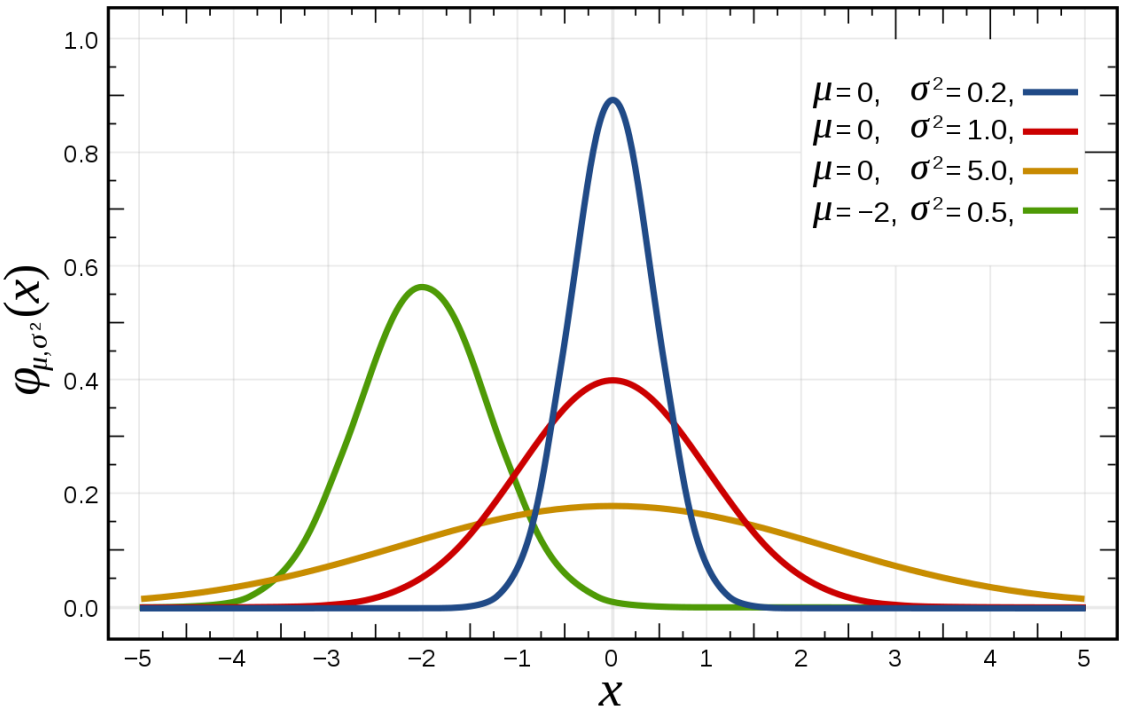
Testy

Metoda `CalculateShapiroWilkTestForNormality` została obłożona testami jednostkowymi `TestShapiroWilkTestDouble`, `TestShapiroWilkTestInt`. Wartości uzyskiwane w metodzie `CalculateShapiroWilkTestForNormality` porównywane do wartości uzyskanych w RStudio przy pomocy funkcji `shapiro.test(x)`.

Dodatek A. Ciągłe rozkłady prawdopodobieństwa

A.1. Rozkład normalny

Rozkład normalny (ang. *normal distribution*), inaczej zwany rozkładem Gaussa, jeden z najważniejszych rozkładów prawdopodobieństwa. Bardzo często występuje w naturze, dlatego jest tak popularny. Istnieje wiele sposobów numerycznego obliczania pola powierzchni pod standardową krzywą rozkładu normalnego. Wykres funkcji prawdopodobieństwa tego rozkładu jest w kształcie dzwona (rys. A.1.1). Rozkład normalny o średniej $\mu = 0$ i $\sigma^2 = 0.1$ jest to rozkład standaryzowany na rysunku (rys. A.1.1) odpowiada mu czerwona linia. Do jego obliczenia wykorzystany jest np. algorytm ACM (ang. *Association for Computing Machinery*) numer 209 [16], algorytm jest zaimplementowany w statycznej funkcji `Gauss` (listing A.1.1).



Rys.A.1.1. Rozkład normalny [20]

Listing.A.1.1. Statyczna metoda implementująca rozkład normalny (rozkład Gaussa)

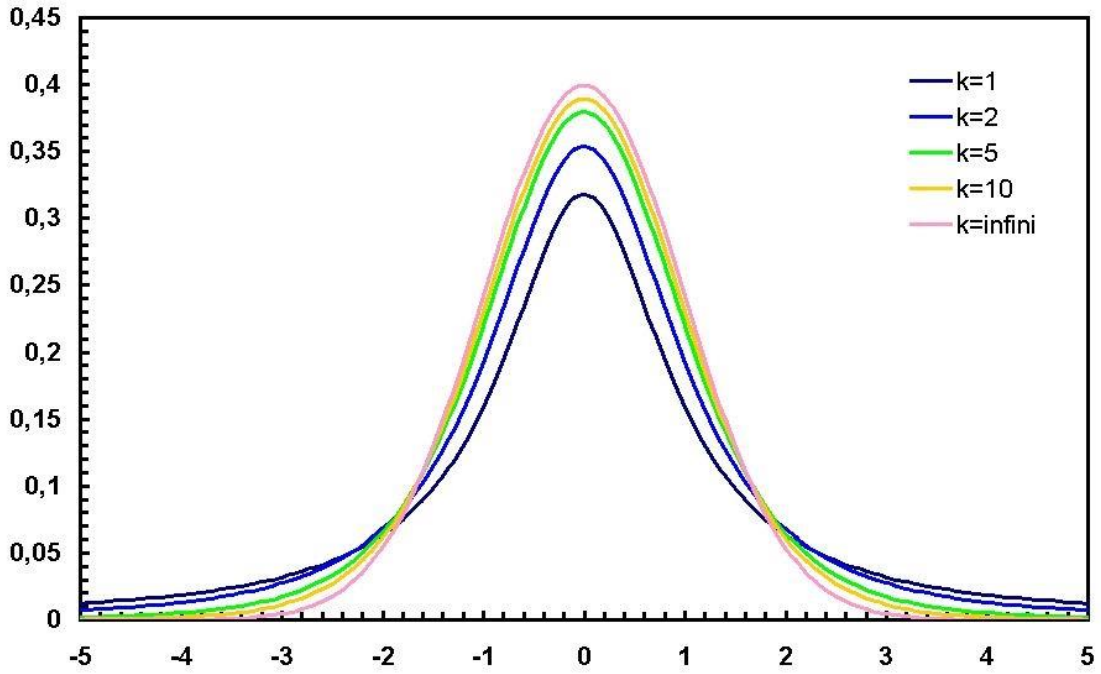
```
public static double Gauss(double z)
{
        double y;
        double p;
        double w;
        if (z == 0.0)
```

```
                p = 0.0;
        else
        {
                y = Math.Abs(z) / 2;
                if (y >= 3.0)
                {
                        p = 1.0;
                }
                else if (y < 1.0)
                {
                        w = y * y;
                        p = ((((((((0.000124818987 * w
                        - 0.001075204047) * w + 0.005198775019) * w
                        - 0.019198292004) * w + 0.059054035642) * w
                        - 0.151968751364) * w + 0.319152932694) * w
                        - 0.531923007300) * w + 0.797884560593) * y * 2.0;
                }
                else
                {
                        y = y - 2.0;
                        p = ((((((((((((-0.000045255659 * y
                        + 0.000152529290) * y - 0.000019538132) * y
                        - 0.000676904986) * y + 0.001390604284) * y
                        - 0.000794620820) * y - 0.002034254874) * y
                        + 0.006549791214) * y - 0.010557625006) * y
                        + 0.011630447319) * y - 0.009279453341) * y
                        + 0.005353579108) * y - 0.002141268741) * y
                        + 0.000535310849) * y + 0.999936657524;
                }
        }
        if (z > 0.0)
                return (p + 1.0) / 2;
        else
                return (1.0 - p) / 2;
}
```

A.2. Rozkład *t* Studenta

Rozkład *t* Studenta stosowany jest w procedurach testowania hipotez statystycznych i ocenie niepewności pomiaru. Kształt rozkładu jest podobny do rozkładu normalnego, lecz jest bardziej rozciągnięty (rys. A.2.1). Pole powierzchni pod krzywą rozkładu *t* Studenta jest obliczane w statycznej metodzie `Student` (listing A.2.1). Dla zmiennoprzecinkowej liczby

pole jest obliczane z algorytmu ACMnumer 209 [16], a w przypadku liczby całkowitej z algorytmu numer ACM numer 395 [11].



Rys.A.2.1. Rozkład *t* Studenta [21]

Listing.A.2.1. Metody implementujące rozkład *t* Studenta dla całkowitej oraz zmiennoprzecinkowej liczby stopni swobody

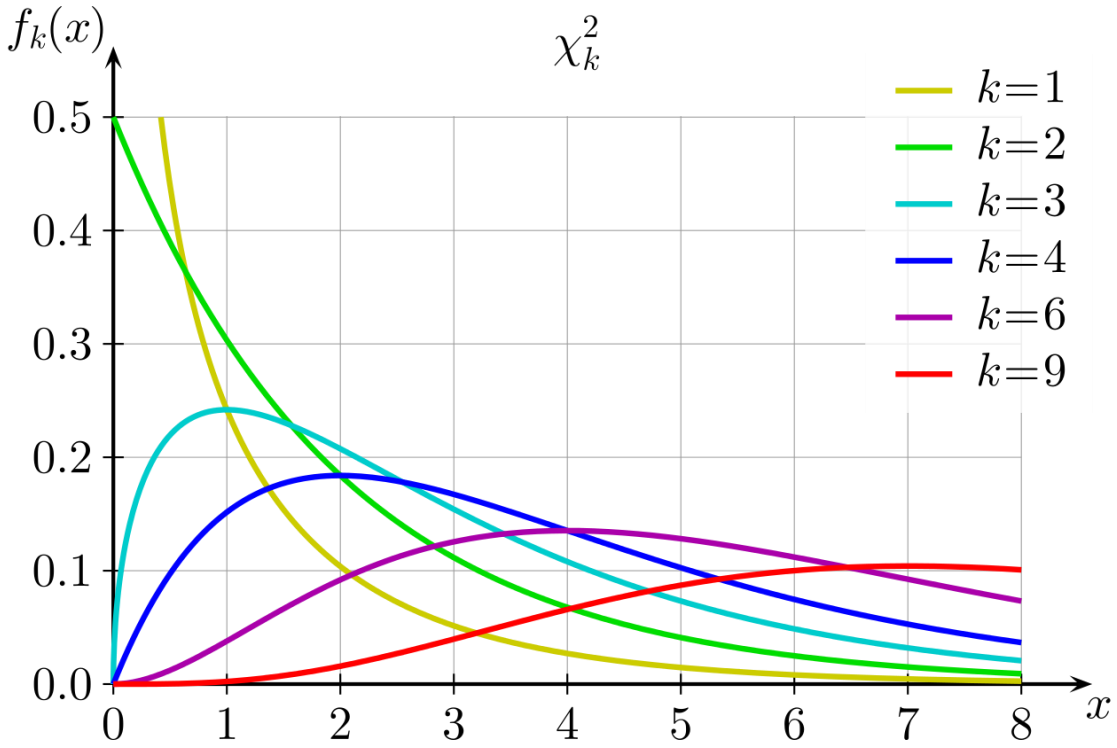
```
public static double Student(double t, double df)
{
	if (df <= 0) throw new ArgumentException("The degrees of freedom need
to be positive.");

	double n = df;
	double a, b, y;
	t = t * t;
	y = t / n;
	b = y + 1.0;
	if (y > 1.0E-6) y = Math.Log(b);
	a = n - 0.5;
	b = 48.0 * a * a;
	y = a * y;
	y = (((((-0.4 * y - 3.3) * y - 24.0) * y - 85.5) /
	(0.8 * y * y + 100.0 + b) + y + 3.0) / b + 1.0) *
	Math.Sqrt(y);
	return 2.0 * Gauss(-Math.Abs(y));
}
```

```
public static double Student(double t, int df)
{
	int n = df;
	double a, b, y, z;

	z = 1.0;
	t = t * t;
	y = t / n;
	b = 1.0 + y;

	if (n >= 20 && t < n || n > 200)
	{
		double x = (double)n;
		return Student(t, x);
	}

	if (n < 20 && t < 4.0)
	{
		a = Math.Sqrt(y);
		y = Math.Sqrt(y);
		if (n == 1)
			a = 0.0;
	}
	else
	{
		a = Math.Sqrt(b);
		y = a * n;
		for (int j = 2; a != z; j += 2)
		{
			z = a;
			y = y * (j - 1) / (b * j);
			a = a + y / (n + j);
		}
		n = n + 2;
		z = y = 0.0;
		a = -a;
	}

	int nCt = 0;
	while (true && nCt < 10000)
	{
```

```
            ++nCt;
            n = n - 2;
            if (n > 1)
            {
                  a = (n - 1) / (b * n) * a + y;
                  continue;
            }
            if (n == 0)
                  a = a / Math.Sqrt(b);
            else
                  a = (Math.Atan(y) + a / b) * 0.63661977236;

            return z - a;
      }

      return -1.0;
}
```

A.3. Rozkład chi kwadrat (χ^2)

Rozkład χ^2 jest szczególnym przypadkiem rozkładu gamma i jest z jednym z najczęściej stosowanych rozkładów prawdopodobieństwa w statystyce wnioskowania (ang. *Statistical inference*). Dany rozkład jest rozkładem prawostronnym, którego kształt jest zależny od liczby stopni swobody *df* (rys. A.3.1). Obliczanie rozkładu jest zaimplementowane w statycznej metodzie `ChiSquareCdf` (listing A.3.1), która do obliczania pola pod krzywą korzysta z metody `Gamma` (listing A.3.2).



Rys.A.3.1. Rozkład chi kwadrat [22]

Listing.A.3.1. Metoda implementująca rozkład chi kwadrat

```
public static double ChiSquareCdf(double x, int df)
{
	if (df <= 0)
	{
		throw new ArgumentException("The degrees of freedom need to be positive.");
	}

	return Gamma(x / 2.0, df / 2.0);
}
```

Listing.A.3.2. Metoda implementująca rozkład Gamma

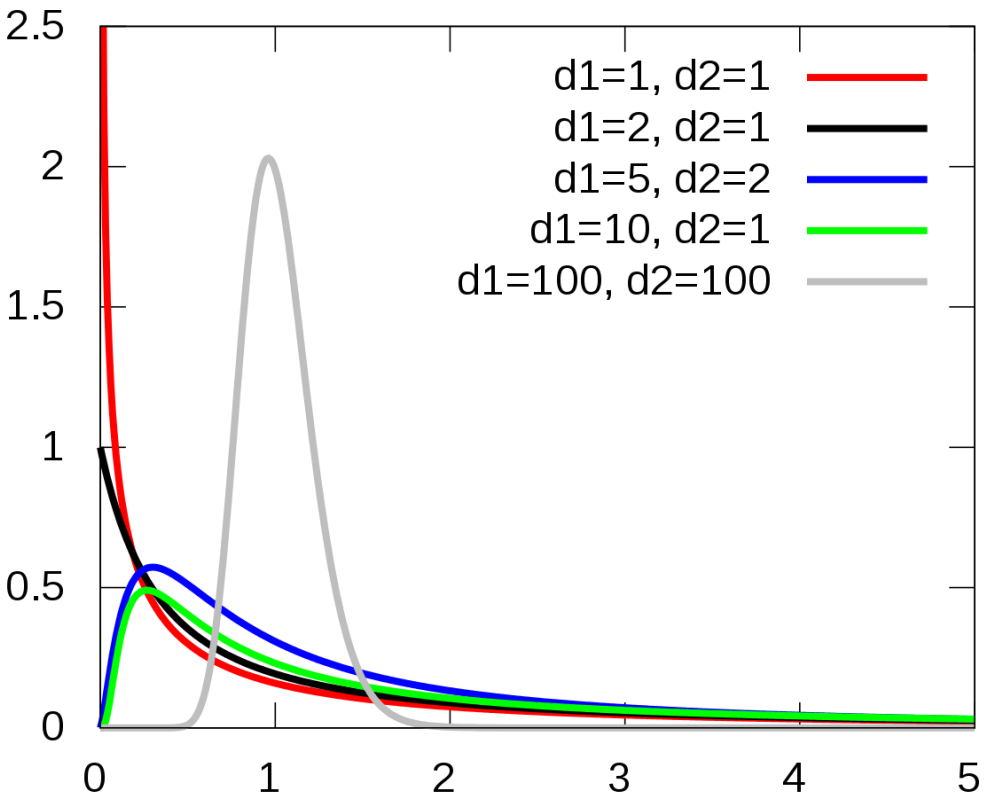
```
private static double GCdf(double x, double A)
{
	double a = 0;
	double b = 1;
	double a1 = 1;
	double b1 = x;
	double aOld = 0;
	double N = 0;
	while (Math.Abs((a1 - aOld) / a1) > .00001)
```

```
        {
                aOld = a1;
                N = N + 1;
                a = a1 + (N - A) * a;
                b = b1 + (N - A) * b;
                a1 = x * a + N * a1;
                b1 = x * b + N * b1;
                a = a / b1;
                b = b / b1;
                a1 = a1 / b1;
                b1 = 1;
        }
        double Prob = Math.Exp(A * Math.Log(x) - x - LogGamma(A)) * a1;

        return 1.0 - Prob;
}
private static double gSer(double x, double A)
{
        double temp = 1 / A;
        double g = temp;
        double i = 1;
        while (temp > g * 0.00001)
        {
                temp = temp * x / (A + i);
                g = g + temp;
                ++i;
        }
        g = g * Math.Exp(A * Math.Log(x) - x - LogGamma(A));

        return g;
}
public static double Gamma(double x, double a)
{
        if (x < 0)
        {
                throw new ArgumentException("The x parameter must be
positive.");
        }

        double gamma;
        if (a > 200)
        {
                double z = (x - a) / Math.Sqrt(a);
```

```
            double y = Gauss(z);
            double b1 = 2 / Math.Sqrt(a);
            double phiz = 0.39894228 * Math.Exp(-z * z / 2);
            double w = y - b1 * (z * z - 1) * phiz / 6;
            double b2 = 6 / a;
            int zXor4 = ((int)z) ^ 4;
            double u = 3 * b2 * (z * z - 3) + b1 * b1 * (zXor4 - 10 * z * z
+ 15);
            gamma= w - phiz * z * u / 72;
      }
      else if (x < a + 1)
      {
            gamma= gSer(x, a);
      }
      else
      {
            gamma= GCdf(x, a);
      }

      return gamma;
}
```

A.4. Rozkład F Snedecora

Rozkład F znany również jako rozkład F Snedecora lub rozkład Fishera–Snedecora jest ciągłym rozkładem prawdopodobieństwo, wykorzystywany przede wszystkim w analizie wariancji (ANOVA). Ten rozkład charakteryzuje dłuższy prawy ogon i kształt zależny od liczby stopni swobody df_1 i df_2. Obliczanie rozkładu zostało zaimplementowane w statycznej metodzie `FCdf` (listing A.4.1), która do obliczania pola pod krzywą korzysta z metody `Betinc` (listing A.4.2).



Rys.A.4.1. Rozkład F Snedecora [23]

Listing.A.4.1. Metoda implementująca rozkład F Snedecora

```
public static double FCdf(double x, int d1, int d2)
{
	if (x < 0 || d1 <= 0 || d2 <= 0)
		throw new ArgumentException("All the parameters must be
positive.");
	double Z = (d1*x)/(d1*x+d2);
	double FCdf = Beta(Z, d1 / 2.0, d2 / 2.0);
	return 1.0-FCdf;
}
```

Listing.A.4.2. Metoda implementujące rozkład Beta

```
public static double Betinc(double x, double A, double B)
{
	double a = 0.0;
	double b = 1.0;
	double a1 = 1.0;
	double b1 = 1.0;
	double temp = 0.0;
	double a2 = 0.0;
	while (Math.Abs((a1 - a2) / a1) > 0.00001)
```

```
        {
            a2 = a1;
            double c = -(A + temp) * (A + B + temp) * x / (A + 2.0 * temp) /
(A + 2.0 * temp + 1.0);
            a = a1 + c * a;
            b = b1 + c * b;
            temp = temp + 1;
            c = temp * (B - temp) * x / (A + 2.0 * temp - 1.0) / (A + 2.0 *
temp);
            a1 = a + c * a1;
            b1 = b + c * b1;
            a = a / b1;
            b = b / b1;
            a1 = a1 / b1;
            b1 = 1.0;
        }
        return a1 / A;
}
public static double Beta(double x, double a, double b)
{
        if (x < 0 || a <= 0 || b <= 0)
        {
            throw new ArgumentException("All the parameters must be
positive.");
        }

        double beta = 0.0;

        if (x == 0)
        {
            return beta;
        }
        else if (x >= 1)
        {
            return 1.0;
        }

        double s = a + b;

        double btemp = Math.Exp(LogGamma(s) - LogGamma(b) - LogGamma(a) + a *
Math.Log(x) + b * Math.Log(1 - x));
        if (x < (a + 1.0) / (s + 2.0))
        {
```

```
                beta = btemp * Betinc(x, a, b);
        }
        else
        {
                beta = 1.0 - btemp * Betinc(1.0 - x, b, a);
        }
        return beta;
}
```

Dodatek B. Metody pomocnicze

B.1. Obliczanie różnicy pomiędzy parami pomiarów

Statyczna metoda `DifferenceBetweenPairsOfMeasurements` (listing B.1.1) jest zaimplementowana w klasie `Util` w dwóch wersjach: dla dwóch kolekcji o takiej samej lub różnych długościach oraz 2) dla kolekcji i liczby zmiennoprzecinkowej. Różnice obliczane dla każdej pary elementów w przypadku, gdy jedna kolekcja jest dłuższa to elementy są przepisywane do zwracanej listy z dłuższej kolekcji danych.

Listing.B.1.1. Metoda obliczająca różnice między parami

```
public static List<double> DifferenceBetweenPairsOfMeasurements(this
IEnumerable<double> list1, IEnumerable<double> list2)
{
	List<double> listOfDifference = new List<double>();
	int minN = Math.Min(list1.Count(), list2.Count());
	int maxN = Math.Max(list1.Count(), list2.Count());
	for (int i = 0; i < minN; i++)
	{
		listOfDifference.Add(list1.ElementAt(i) - list2.ElementAt(i));
	}
	if(list1.Count() != list2.Count())
	{
		if (list1.Count() == maxN)
		{
			for(int i = minN; i < maxN; i++)
			{
				listOfDifference.Add(list1.ElementAt(i));
			}
		}
		if (list2.Count() == maxN)
		{
			for (int i = minN; i < maxN; i++)
			{
				listOfDifference.Add(list2.ElementAt(i));
			}
		}
	}
	return listOfDifference;
}
public static List<double> DifferenceBetweenPairsOfMeasurements(this
IEnumerable<double> list1, double number)
```

```
{
	List<double> listOfDifference = new List<double>();
	int n = list1.Count();

	for (int i = 0; i < n; ++i)
	{
		listOfDifference.Add(list1.ElementAt(i) - number);
	}
	return listOfDifference;
}
```

B.2. Obliczanie rang

Statyczna metoda `CalculateRanks` (listing B.1.1) jest zaimplementowana w klasie `Ranks` i służy do obliczania rang dla zadanej kolekcji. Zwraca słownik, w którym kluczem jest element kolekcji, a wartością ranga przypisana do elementu kolekcji.

Listing.B.2.1. Metoda obliczająca rangi w kolekcji

```
public struct Rank
{
	public int NumberOfRanks;
	public double SumOfPositiveRanks;
	public double SumOfNegativeRanks;
	public double CorrectionForTiedRanks;
}
public static Dictionary<double, double> CalculateRanks(this
IEnumerable<double> list)
{
	list = list.Where(x => x != 0);
	int n = list.Count();
	list = list.OrderBy(x => Math.Abs(x)).ToList();
	Dictionary<double, double> dictOfPairs = list.GroupBy(x =>
Math.Abs(x)).ToDictionary(x => Math.Abs(x.Key), x => (double)0);
	List<double> listOfRanks = list.ToList();

	double m = 0;
	double nSum = 0;
	for (int i = 0; i < n - 1; i++)
	{
		m += 1;
		nSum += (i + 1);
```

```
            if (Math.Abs(listOfRanks.ElementAt(i)) !=
Math.Abs(listOfRanks.ElementAt(i + 1)))
            {
            if (m == 0)
            {
                  dictOfPairs[Math.Abs(listOfRanks.ElementAt(i))] = nSum;
            }
            else
            {
                  dictOfPairs[Math.Abs(listOfRanks.ElementAt(i))] = (nSum /
m);
            }
            m = 0;
            nSum = 0;
            }
            if (Math.Abs(listOfRanks.ElementAt(n - 2)) !=
      Math.Abs(listOfRanks.ElementAt(n - 1)))
            {
                  dictOfPairs[Math.Abs(listOfRanks.ElementAt(n - 1))] = n;
            }
            if (Math.Abs(listOfRanks.ElementAt(n - 2)) ==
      Math.Abs(listOfRanks.ElementAt(n - 1)))
            {
                  dictOfPairs[Math.Abs(listOfRanks.ElementAt(n - 1))] =
((nSum + n) / (m + 1));
            }
      }
      return dictOfPairs;
}
```

6. Literatura

[1] https://www.statsoft.pl/textbook/stathome.html - Internetowy Podręcznik Statystyki.

[2] http://manuals.pqstat.pl/ - Baza wiedzy PQStat.

[3] J. Jóźwiak, J. Podgórski, *Statystyka od podstaw*, PWE, Warszawa 2012.

[4] Jerzy Wierzbiński, *Statystyka Opisowa*, WWZ, Warszawa 2008.

[5] A. Komosa, J. Musiałkiewicz, *Statystyka*, Ekonomik, Warszawa 1996

[6] D. N. Joanes and C. A. Gill (1998), Comparing measures of sample skewness and kurtosis. The Statistician, 47, 183-189.

[7] Yawen Guo, B. M. Golam Kibria, *On Some Statistics for Testing the Skewness in a Population: An Empirical Study*, AAM, Vol. 12, Issue 2 (2017), 726 -752

[8] https://www.rdocumentation.org/ - dokumentacja dla języka R.

[9] Zofia Hanusz, Joanna Tarasińska, *Normalization of the Kolmogorov–Smirnov and Shapiro–Wilk tests of normality*, Biometrical Letters, 52, (2015), 85 – 93.

[10] Patrick Royston, *Approximating the Shapiro–Wilk W-test for non-normality*, Statistics and Computing, 2, (1992),117-119.

[11] G. W. Hill, *ACM Algorithm 395: Student's t-Distribution*, Communications of the ACM, Vol.13, No.10, (1970), 617-619.

[12] W.H Kruskal, W.A Wallis, *Use of ranks in one-criterion variance analysis*, Journal of the American Statistical Association, 47, (1952), 583-621.

[13] W.H Kruskal, W.A Wallis, *Use of ranks in one-criterion variance analysis*, Journal of the American Statistical Association, 47, (1952), 583-621.

[14] Jerrold H. Zar, *Significance Testing of the Spearman Rank Correlation Coefficient*, Journal of the American Statistical Association, Vol. 67, No. 339, (1972), 578-580.

[15] Milton Friedman, *The use of ranks to avoid the assumption of normality implicit in the analysis of variance*, Journal of the American Statistical Association, 32, (1937), 675-701.

[16] D. Ibbetson, *ACM Algorithm 209: GAUSS*, Communications of the ACM, Vol.6, No.10, (1963), 616.

[17] J. Sharp, *Microsoft Visual C# 2012*, APN Promise, Warszawa 2013.

[18] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/ - dokumentacja na temat LINQ.

[19] https://docs.microsoft.com/en-us/dotnet/ - dokumentacja na temat .NET od Microsoft.

[20] https://pl.wikipedia.org/wiki/Rozkład_normalny - artykuł na temat rozkładu normalnego.

[21] https://pl.wikipedia.org/wiki/Rozkład_Studenta - artykuł na temat rozkładu *t* Studenta.

[22] https://en.wikipedia.org/wiki/Chi-square_distribution - artykuł na temat rozkładu chi kwadrat.

[23] https://pl.wikipedia.org/wiki/Rozkład_F_Snedecora - artykuł na temat rozkładu F Snedecora.