MSc Statistical Programming 2024

P468

January 21, 2025

1 U.K. House Prices

1. In Figure 1, we plot the average house price for the four nations: England, Northern Ireland, Scotland and Wales. We observe that the average house price is currently the highest in England, followed by Wales, Scotland and Northern Ireland. Besides that, when looking further back in time, we observe that this ordering has been relatively stable over time. One exception is maybe between 2000 and 2008. During this period, the average house prices in all four countries rose rapidly, but especially those in Northern Ireland. This led to a climax in the years before 2008, when the house prices in Norther Ireland were on average even larger than those in England. Interestingly, the apparent bubble in the Irish housing market collapsed in 2008, since the average house price declined drastically. This is also known as the Irish property bubble and marks the end of the so-called *Celtic Tiger*.

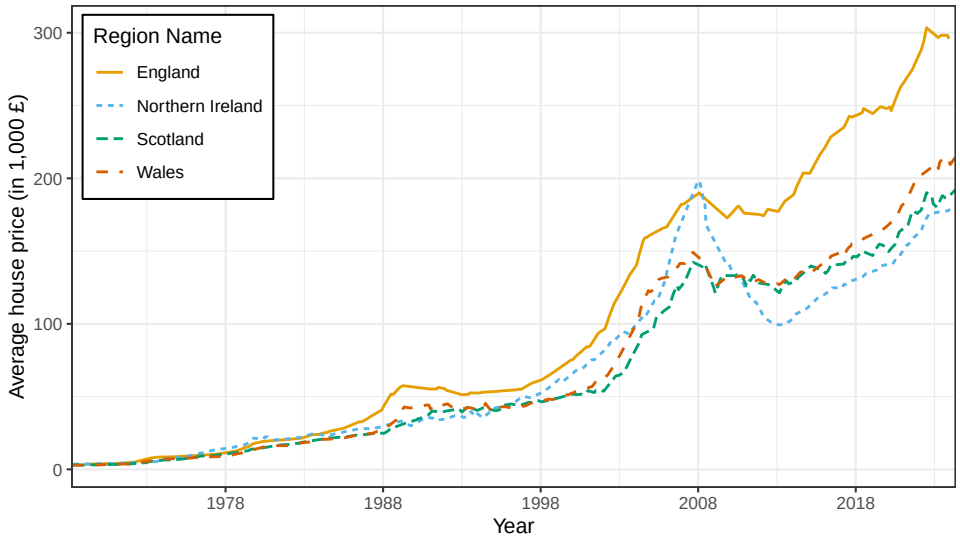


Figure 1: Average house price for England, Scotland, Wales, and Northern Ireland

2. Moreover, we want to plot the average house prices for England and the four regions that have "Cambridge" in the region name. We want to restrict the date range in the plot to only those dates for which England and Cambridge regional information are available. Table 1 presents the four regions of Cambridge together with the first and last date for which

the average house price is available in our sample. We observe that for all Cambridge regions the price series is available from 1995-01-01. For England, however, the first observation is already in 1968-04-01. The last observation for all regions is made at 2024-07-01. Furthermore, it can be shown that we do not have regions with missing price-observations between 1995-01-01 and 2024-07-01. Therefore, we restrict the plotting period to these dates.

Table 1
The four regions of Cambridge together with the first and last date observed in our sample

Region Name	First date	Last date
Cambridge	1995-01-01	2024-07-01
Cambridgeshire	1995-01-01	2024-07-01
East Cambridgeshire	1995-01-01	2024-07-01
England	1968-04-01	2024-07-01
South Cambridgeshire	1995-01-01	2024-07-01

Figure 2 plots the average house price for England and the four Cambridge regions. In general, we observe that the average house price in the Cambridge regions is higher than those in England since 1995. Especially the prices in Cambridge and Cambridgeshire are high relative to England, and this absolute difference seems to have grown over time.

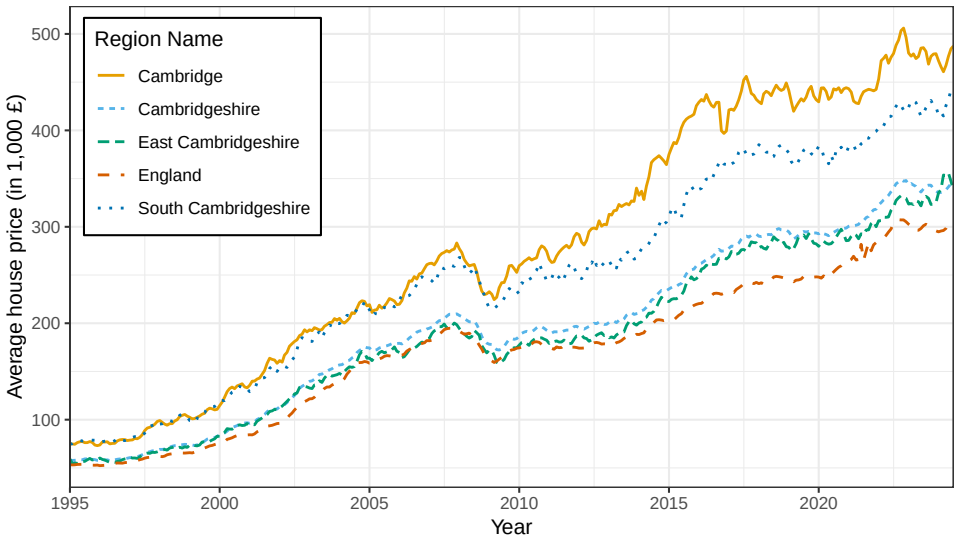


Figure 2: Average house price for England and the four regions of Cambridge

3. In this section, we determine the regions in England for which the average house price has been the highest over time. We define the regions in England as those for which the first character of the `Area_Code` variable is "E". Then, for each of those regions, we calculate the monthly ratio of the average house price for that region relative to the average price in England, where the observations are matched by date. By comparing the median of those

ratios between regions, we get some idea of where in England the average house prices have been highest over time.

Table 2
This table presents the 10 regions in England with the highest median house-price ratio relative to England (as described in the text above). Besides that, this table presents the initial and final average house price, and the percentage increase between these two values. The average prices are presented in 1,000£.

Region Name	Median Ratio	Initial Price*	Final Price*	Increase (%)
Kensington and Chelsea	4.59	182.69	1164.14	537.20
City of Westminster	3.30	133.03	904.36	579.84
Camden	2.90	120.93	858.30	609.73
Hammersmith and Fulham	2.78	124.90	797.03	538.12
City of London	2.60	91.45	766.88	738.59
Richmond upon Thames	2.50	109.33	744.28	580.79
Elmbridge	2.32	106.52	679.30	537.70
Islington	2.31	92.52	684.84	640.24
Wandsworth	2.14	88.56	631.94	613.59
Inner London	1.98	78.25	592.73	657.47

* Expressed in 1,000 GBP.

Table 2 presents the 10 regions with the highest median ratio. Perhaps not surprisingly, these are all regions in or near London. Thus, while the prices of houses in Cambridge are elevated, they are not in the top 10 most elevated regions. We observe that Kensington and Chelsea has the highest median ratio, with a value of 4.59. Table 2 also shows the initial and final average house prices for the period over which the medians are calculated, and the percentage increase between these two values. We observe that the percentage increase is highest for City of London, with an increase of 738.59%.

4. Now, we are going to use the inflation index to investigate whether the house prices in England increased, on average, faster than inflation. We use the "CPIH INDEX 00" as consumer price index for the UK, which also includes owner-occupied housing costs. Since the baseline value of this inflation index is 2015 (with value 100), we are particularly interested in the increase of inflation compared to house prices from 2015 onwards. Therefore, we appropriately scale the average house price series of England such that it intersects with the inflation series in 2015. In this way, we can easily compare the relative increase since then. The resulting plot is shown in Figure 3, where the left-axis corresponds to the inflation series (CPIH) and the right-axis to the average house price series of England. The baseline year, 2015, is highlighted with a red-dashed vertical line.

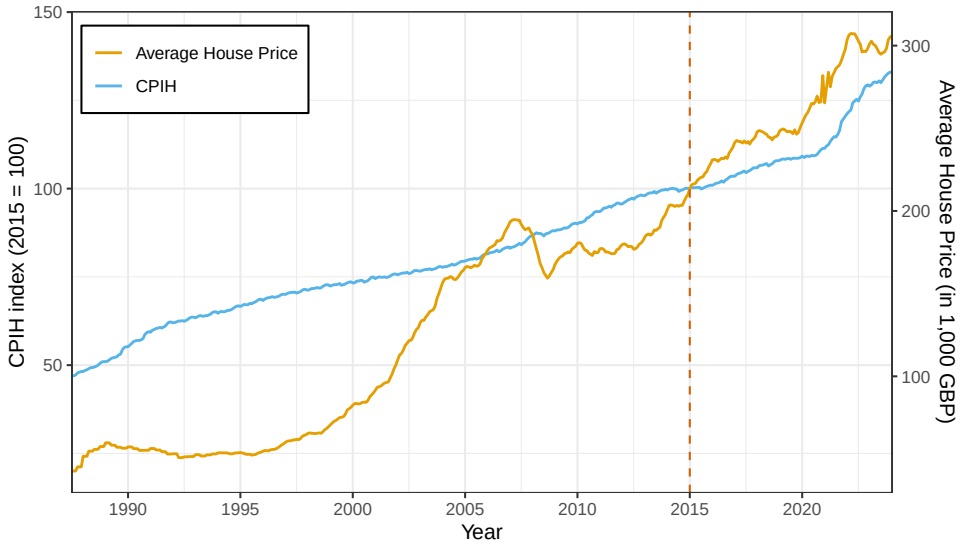


Figure 3: Increase in average house price in England versus increase in inflation (CPIH).

In Figure 3, we observe that the average house price series clearly increased faster than inflation since 2015. More specifically, the house price series increased by 43.3% since 2015, while the CPIH index increased by 32.9%. Also since 1988-01-01 (i.e., the first date for which both inflation and price data are available) the house price in England increased faster than inflation, with a value of 616% and 183.4%, respectively.

5. Finally, we investigate whether seasonal patterns are present in the average house price changes of the United Kingdom. We do this by plotting the month-over-month percentage changes in average house prices, where the observations are grouped by month and coloured by decade in a beeswarm plot. The resulting plot is given in Figure 4. The large black dots in the figure, which correspond to the mean value, indicate that the average monthly change in UK house price is larger in the spring/summer period than in the autumn/winter period. This observation could potentially be explained by the weather. For instance, in the winter period it gets dark early, which might reduce the number of visits by potential buyers. Moreover, Figure 4 annotates the observation associated with the largest monthly price drop in the UK. We observe that this drop happened in July 2021, when the average house price declined by 4.7% in a single month. Interestingly, July 2021 is also associated with record-breaking temperatures in the UK, and there were COVID-19 restrictions in place. These events might explain the large drop in average house price.

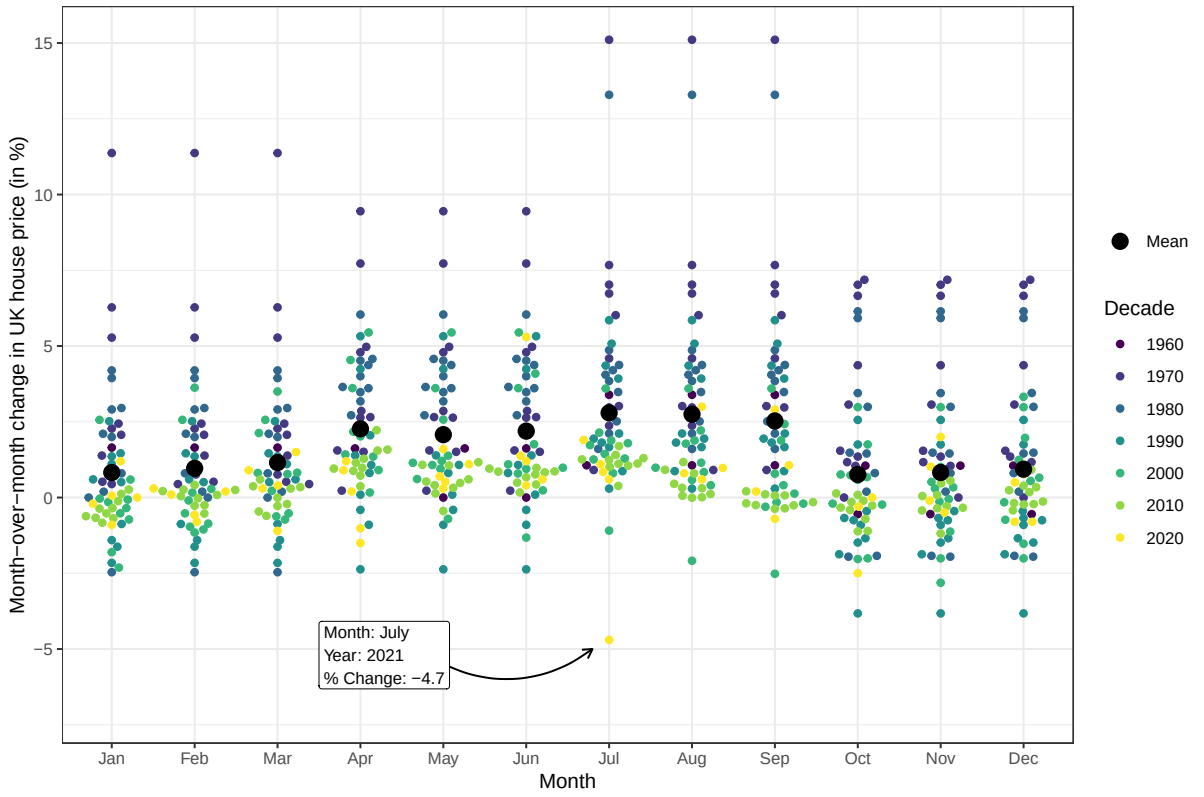


Figure 4: Beeswarm plot for the month-over-month percentage change in average house prices in the UK

2 Chemical similarity

1. In this section, we analyse the amino-acid data in the CSV file `AA-NNAA-FP.csv`. First, we load the dataset into R.

```
amino_acids <- read_csv("AA-NNAA-FP.csv", progress = FALSE)
```

The resulting dataframe has 2994 rows (i.e., amino acids) and 7 columns. We obtain the column names with by running the following expression in R:

```
amino_acids %>%
  colnames() %>%
  kable(col.names = c("Column Names"))
```

Column Names
Name
IsNatural
SMILES
TotalCharge
clogP
xlogP
FP

Next, we would like to know the number of natural and non-natural amino acids in our sample. For this purpose, we use the `isNatural` column, which equals 1 when the amino acid is natural and 0 otherwise.

```
amino_acids %>%
  group_by(IsNatural) %>%
  summarize(Count = n()) %>%
  kable()
```

IsNatural	Count
0	2950
1	44

In the table above, we observe that our dataset contains 44 natural amino acids and 2950 non-natural amino acids. Note that our dataset contains more than 20 natural amino acids, which might seem like a mistake. However, our dataset also contains the natural amino acids of the *zwitterionic* form. When we take this into account, we observe that our dataset contains 19 *non-zwitterionic* natural amino acids, as can be seen in the table below.

```
amino_acids %>%
  mutate(
    IsZwitterionic = as.integer(str_detect(Name, pattern = "Zwitterionic"))
  ) %>%
  group_by(IsNatural, IsZwitterionic) %>%
  summarize(Count = n()) %>%
  kable()
```

IsNatural	IsZwitterionic	Count
0	0	2950
1	0	19
1	1	25

2. Next, we separate the dataframe `amino_acids` into two new dataframes, `dfAA` and `dfNNAA`. `dfAA` contains the natural amino acids (AAs). Similarly, `dfNNAA` contains the non-natural amino acids (NNAAs).

```r
dfAA <- amino_acids %>%
  filter(IsNatural == 1)
dfNNAA <- amino_acids %>%
  filter(IsNatural == 0)
```

The last column in these dataframes is `FP`, which contains the ECFP4 2048-bit fingerprints. We would like to test whether all these fingerprints are indeed 2048 bits long. We check this with the expression given below. Note that the fingerprints are stored as strings in `amino_acids`, where the bits are separated by spaces.

```r
test_that("All fingerprints in dataframe are 2048 bits long", {
  expect_true(
    all(
      amino_acids %>%
        pull(FP) %>%
        str_split(pattern = " ") %>%
        sapply(FUN = length)
      == 2048
    )
  )
})

## Test passed
```

Since the test is passed, we conclude that all the fingerprints in the original dataframe are indeed 2048 bits long.

3. Now that we know all fingerprints are of the correct length, it is time to develop our first function to calculate the Tanimoto similarity measure. This function, which is named `get_tanimoto_similarity`, calculates the Tanimoto similarity measure for two fingerprints by comparing each pair of bits using a for loop. The function is defined as follows.

```r
get_tanimoto_similarity <- function(fp1, fp2) {
  n_intersection <- 0
  n_union <- 0

  for (i in seq_along(fp1)) {
    if (fp1[i] || fp2[i]) {
      n_union <- n_union + 1
      if (fp1[i] && fp2[i]) {
        n_intersection <- n_intersection + 1
      }
    }
```

```
  }

  if (n_union == 0) {
    return(1.0)
  }
  n_intersection / n_union
}
```

In order to prevent a division by zero, we include a if-condition before we calculate the Tanimoto similarity measure and return the output. The only situation that can result in a division by zero is when both fingerprints contain only zeroes. This means that no single chemical substructure is present in any of the two fingerprints. Even though this situation is very unlikely to occur, and might rather be an indicator of a typo in the data, we do take it explicitly into account here. Formally, two fingerprints with only zeroes are identical and, thus, the Tanimoto similarity measure should be one.

Furthermore, in the function above we assume that the fingerprints provided as arguments (`fp1` and `fp2`) are integer vectors of binary numbers. However, as we mentioned before, the fingerprints in the original dataframe are strings. We decide to convert the strings to integer vectors outside the function above (and any function below), since this is more efficient (but it is maybe less beneficial when someone cares about storage). For example, if we would like to compare all the fingerprints to all other fingerprints, the number of conversions from string to integer vector would be way less when we do this outside the function call (only once) than when we do this inside the function.

The tests below check that the numeric value returned by `get_tanimoto_similarity` is never negative or greater than 1.

```
test_that("Identical fingerprints give tanimoto similarity of 1", {
  fp1 <- c(1, 0, 1, 0, 1)
  fp2 <- fp1
  expect_equal(
    get_tanimoto_similarity(fp1, fp2),
    1
  )
})

## Test passed

test_that("Completely different fingerprints have similarity of 0", {
  fp1 <- c(1, 0, 1, 0, 1)
  fp2 <- c(0, 1, 0, 1, 0)
  expect_equal(
    get_tanimoto_similarity(fp1, fp2),
```

```
    0
  )
})

## Test passed

test_that("Two fingerprints have similarity between 0 and 1", {
  fp1 <- c(1, 0, 1, 0, 1)
  fp2 <- c(0, 0, 1, 1, 1)
  value <- get_tanimoto_similarity(fp1, fp2)
  expect_true(value <= 1 & value >= 0)
})

## Test passed

test_that("Two fingerprints with only zeroes have similarity of 1", {
  fp1 <- rep(0, times = 5)
  fp2 <- rep(0, times = 5)
  expect_equal(
    get_tanimoto_similarity(fp1, fp2),
    1
  )
})

## Test passed
```

4. Instead of using a for-loop it is probably more efficient to implement the Tanimoto similarity calculation using vectorized R. This function, named `get_tanimoto_vectorized`, is defined as follows:

```
get_tanimoto_vectorized <- function(fp1, fp2) {
  n_union <- sum(fp1 | fp2)
  n_intersection <- sum(fp1 & fp2)

  if (n_union == 0) {
    return(1.0)
  }
  n_intersection / n_union
}
```

5. For our third implementation of the Tanimoto simularity calculation, we use the `bitops` package. It provides functions to perform bitwise comparisons of integer vectors. Using these function, we define the following `get_tanimoto_bitops` function.

```r
get_tanimoto_bitops <- function(fp1, fp2) {
  n_union <- sum(bitops::bitOr(fp1, fp2))
  n_intersection <- sum(bitops::bitAnd(fp1, fp2))

  if (n_union == 0) {
    return(1.0)
  }
  n_intersection / n_union
}
```

6. Finally, we define the C++ function `get_tanimoto_Rcpp` in R using the `Rcpp` package.

```r
Rcpp::cppFunction("
double get_tanimoto_Rcpp(IntegerVector fp1, IntegerVector fp2) {
    double n_intersection = 0;
    double n_union = 0;

    for (int i = 0; i < fp1.size(); i++) {
        if (fp1[i] == 1 || fp2[i] == 1) {
            n_union++;
            if (fp1[i] == 1 && fp2[i] == 1) {
                n_intersection++;
            }
        }
    }

    if (n_union == 0) {
        return 1.0;
    }
    return n_intersection / n_union;
}
")
```

7. It is important to test that all the functions that we defined above give the same Tanimoto similarity measure. Therefore, we use the `testthat` package to define an unit test for the equality of output. This test is given below, and uses the first natural amino acid in `dfAA` and non-natural amino acid in `dfNNAA` as input arguments.

```r
test_that("All functions generate the same Tanimoto similarity result", {
  fp1 <- as.integer(str_split_1(dfAA$FP[1], pattern = " "))
  fp2 <- as.integer(str_split_1(dfNNAA$FP[1], pattern = " "))
  expect_equal(
```

```
    get_tanimoto_similarity(fp1, fp2),
    get_tanimoto_vectorized(fp1, fp2)
  )
  expect_equal(
    get_tanimoto_vectorized(fp1, fp2),
    get_tanimoto_bitops(fp1, fp2)
  )
  expect_equal(
    get_tanimoto_bitops(fp1, fp2),
    get_tanimoto_Rcpp(fp1, fp2)
  )
})

## Test passed
```

8. Ultimately, we want to know which implementation calculates the Tanimoto similarity the fastest. For this reason, we will benchmark the four implementations below. We define the function **get_running_times**, which gives the running times in seconds (by default, **unit** = "s") for each Tanimoto function that is runned **runs** times.

```
get_running_times <- function(runs, unit = "s") {
  fp1 <- as.integer(str_split_1(dfAA$FP[1], pattern = " "))
  fp2 <- as.integer(str_split_1(dfNNAA$FP[1], pattern = " "))
  times <- microbenchmark(
    "R loop" = get_tanimoto_similarity(fp1, fp2),
    "R vectorized" = get_tanimoto_vectorized(fp1, fp2),
    bitops = get_tanimoto_bitops(fp1, fp2),
    Rcpp = get_tanimoto_Rcpp(fp1, fp2),
    times = runs,
    unit = unit
  )
  summary(times)[, c("expr", "mean", "neval")]
}
```

   (a) First, we use **get_running_times** to obtain the running times (in seconds) of the four Tanimoto implementations, and for different number of **runs**. More specifically, we consider the running times when each implementation is called 1000, 5000, 10000, 25000, 50000, 75000, or 100000 times. The running times are given in Table 3.

```
seq_runs <- c(1, 5, 10, 25, 50, 75, 100) * 1000

total_run_times <- bind_rows(
```

```
  lapply(seq_runs, FUN = function(x) get_running_times(runs = x))
) %>%
  mutate(total = mean * neval) %>%
  select(-mean)

total_run_times %>%
  pivot_wider(names_from = expr, values_from = total) %>%
  rename(Runs = neval) %>%
  kbl(
    digits = 2,
    caption = paste0("Running times (in seconds) for the four ",
                     "Tanimoto similarity functions defined in ",
                     "this paper."),
    booktabs = TRUE,
    linesep = "",
    table.envir = "table} \\captionsetup{margin = 20pt"
  ) %>%
  column_spec(1:5, width = "2.5 cm") %>%
  kable_paper(latex_options = c("HOLD_position"))
```

Table 3
Running times (in seconds) for the four Tanimoto similarity functions defined in this paper.

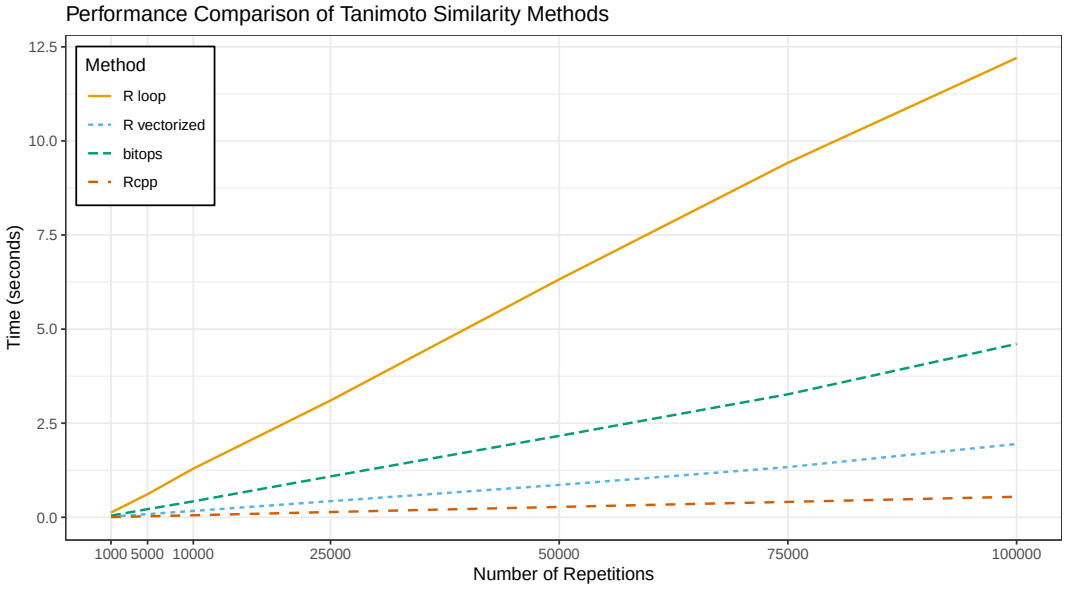
Runs	R loop	R vectorized	bitops	Rcpp
1000	0.13	0.02	0.04	0.01
5000	0.61	0.08	0.22	0.03
10000	1.29	0.17	0.42	0.05
25000	3.10	0.43	1.09	0.14
50000	6.32	0.86	2.16	0.28
75000	9.42	1.33	3.27	0.41
100000	12.21	1.95	4.61	0.55

We observe that the C++ implementation, with the function `get_tanimoto_Rcpp`, has the lowest running times. Furthermore, the remaining implementations can be ordered from low to high running times by: "R vectorized", "bitops" and "R loop".

(b) Next, we plot the running times for each method against the number of repetitions. We use the following R code to generate the figure.

```
ggplot(
  data = total_run_times,
  mapping = aes(
    x = neval,
```

```
    y = total,
    linetype = expr,
    colour = expr
  )
) +
  geom_line(linewidth = 0.7) +
  scale_colour_manual(values = cbPalette[-1]) +
  scale_x_continuous(breaks = seq_runs, minor_breaks = NULL) +
  labs(
    x = "Number of Repetitions",
    y = "Time (seconds)",
    linetype = "Method",
    colour = "Method",
    title = "Performance Comparison of Tanimoto Similarity Methods"
  ) +
  theme(
    legend.position = c(0.08, 0.82),
    legend.background = element_rect(colour = "black")
  )
```



In the figure above, we observe that, for each method, the running time increases linearly with the number of repetitions. However, the slope of this linear relationship differs between the methods. Especially, the running time for the "R loop" method (`get_tanimoto_similarity`) increases faster than the running time for other methods. Furthermore, it is not surprising that we observe a linear relationship because the size of the input arguments stays the same (i.e., each fingerprints has 2048 bits). Additionally, since no randomness is involved in the calculation of the Tanimoto similarity, the average running time is likely constant for these large number or repetitions. Thus, the total running time will increase by a constant factor for each

additional repetition.

(c) Now, we will analyse the time-complexity of the four Tanimoto algorithms using the Big-O notation. The figure above might suggest that the time-complexity is linear for each method. However, note that this figure does not show how the running time increases as a function of the size of the input arguments (i.e., the number of bits in each fingerprint). Instead, the complexity of an algorithm is related to this relationship between input size and number of operations (which can be related to running time if we assume that each operation takes a fixed amount of time). Let N be the length of the fingerprint-vectors (i.e., the number of bits). Then, the Big-O notation for each implementation can be derived as follows:

R loop

i. We loop over the length of the fingerprints (N). Thus, the number of operations we perform inside this loop should be multiplied by a factor N.

ii. In each iteration, we compare the two bits at the corresponding positions in `fp1` and `fp2` (i.e., the two fingerprints). Then, given this comparison, we perform a fixed number of operations (i.e., does not increase with the size of any input arguments). Thus, the complexity for each iteration is $\mathcal{O}(1)$.

iii. After running the loop, we check the value of one variable, which can be seen as one operation. Then, we immediately return the desired result.

iv. **Conclusion:** the complexity is roughly given by $\mathcal{O}(N) * \mathcal{O}(1) + \mathcal{O}(1)$, which is equivalent to $\mathcal{O}(N)$.

R vectorized

i. First, we use the element-wise OR and AND operators on the two fingerprints of length N. So, these operations have a time-complexity of $\mathcal{O}(2N) = \mathcal{O}(N)$. We are left with two boolean vectors of length N.

ii. Then, we sum over the two boolean vectors to obtain two numeric values (`n_union` and `n_intersection`). These operations again have a complexity of roughly $\mathcal{O}(2N) = \mathcal{O}(N)$.

iii. Finally, we again check one variable to avoid a division by zero, which gives one additional operation. After that, we return the final result.

iv. **Conclusion:** the complexity is roughly $\mathcal{O}(N) + \mathcal{O}(N) + \mathcal{O}(1)$, which can be summarized as $\mathcal{O}(N)$. Given the first 2 steps above, you might believe that the complexity is given by $\mathcal{O}(N * N)$. However, note that the sum is performed after we compared all the N pairs of bits. We do not sum over N elements for each pair of bits we compare, which would give $\mathcal{O}(N^2)$.

Bitops

The same analysis as for "R vectorized". However, we use the `bitAnd` and `bitOr` functions instead of the element-wise "&" and "|" equivalents. We assume that the complexity of both functions is $\mathcal{O}(N)$. Then, the complexity of `get_tanimoto_bitops` is also $\mathcal{O}(N)$.

Rcpp

The C++ function is exactly the same as the `get_tanimoto_similarity` function. The only difference is that it is implemented in C++ instead of R. Therefore, the same analysis as for "R loop" holds here. Thus, the complexity of `get_tanimoto_Rcpp` is $\mathcal{O}(N)$.

(d) All in all, we observe that the C++ implementation has the lowest running times. Given the analysis above, this cannot be explained by the time-complexity of the algorithms, since all implementations have the same complexity. Instead C++ is just faster. For instance, in the lecture notes it is written that C++ runs for-loops faster. This could potentially explain why `get_tanimoto_Rcpp` is faster than the other functions.

9. In this section, we will make three tables that show the most, second most, and third most similar non-natural amino acids (NNAAs) for each natural amino acid (AA). The final tables report the natural AAs in alphabetical order. We use our `get_tanimoto_Rcpp` function (i.e., our fastest function) to calculate the Tanimoto similarities between all NNAAs in `dfNNAA` and AAs in `dfAA`. In order to calculate these similarity measures, we need the fingerprints as integer vectors. As we argued earlier, we prefer to convert all fingerprint strings to integer vectors only once. We store all the bit-vectors in lists by running the following R expression:

```
AA_FP_list <- lapply(
  X = str_split(dfAA$FP, pattern = " "),
  FUN = as.integer
)
names(AA_FP_list) <- dfAA$Name
NNAA_FP_list <- lapply(
  X = str_split(dfNNAA$FP, pattern = " "),
  FUN = as.integer
)
names(NNAA_FP_list) <- dfNNAA$Name
```

Then, we code a function that calculates the n most similar NNAAs for each natural amino acid (AA). We use this function to make a variable `three_most_similar_NNAAs`, which is a list and stores the `tibble` objects with the n most similar NNAAs for each AA in `dfAA`.

```
find_n_most_similar_NNAAs <- function(AA_FP, NNAA_FP_list, n = 3) {
  similarity <- lapply(
    X = NNAA_FP_list,
    FUN = function(x) get_tanimoto_Rcpp(fp1 = x, fp2 = AA_FP)
  )

  tibble(
```

```
    NNAAName = names(similarity),
    Similarity = as.numeric(similarity)
  ) %>%
    arrange(desc(Similarity)) %>%
    head(n = n)
}

three_most_similar_NNAAs <- lapply(
  X = AA_FP_list,
  FUN = function(x) find_n_most_similar_NNAAs(AA_FP = x, NNAA_FP_list)
)
```

Finally, we make the similarity tables with the `kableExtra` package. The table for a specific `rank` (between 1 and n), and with a specific `caption`, is obtained by calling the function `make_similarity_table`.

```
make_similarity_table <- function(
    most_similar_NNAAs,
    rank,
    caption,
    label
) {
  tibble(
    AAName = names(most_similar_NNAAs),
    bind_rows(lapply(X = most_similar_NNAAs, FUN = function(x) x[rank,]))
  ) %>%
    arrange(AAName) %>%
    kbl(
      digits = 3,
      format = "latex",
      booktabs = TRUE,
      caption = caption,
      linesep = "",
      table.envir = "table} \\captionsetup{margin = 55pt",
      label = label
    ) %>%
    kable_styling(latex_options = c("HOLD_position"), font_size = 10)
}

table_input_args <- list(
  c(rank = 1, cap = "Most Similar Non-natural AA", lab = "most1"),
  c(rank = 2, cap = "Second Most Similar Non-natural AA", lab = "most2"),
  c(rank = 3, cap = "Third Most Similar Non-natural AA", lab = "most3")
```

```
)
for (args in table_input_args) {
  print(make_similarity_table(
    three_most_similar_NNAAs,
    rank = args["rank"],
    caption = args["cap"],
    label = args["lab"]
  ))
}
```

Table 4
Most Similar Non-natural AA

AAName	NNAAName	Similarity
Alanine	Molport-001-791-314	1.000
Arginine	Molport-003-933-594	1.000
Asparagine	Molport-003-932-124	1.000
Aspartic Acid	Molport-003-986-289	1.000
Glutamic Acid	Molport-003-986-303	1.000
Glutamine	Molport-003-932-123	1.000
Glycine	Molport-002-899-759	0.333
Histidine (neutral)	Molport-051-737-967	0.706
Isoleucine	Molport-000-145-944	0.652
Leucine	Molport-000-145-428	1.000
Lysine	Molport-003-927-113	1.000
Methionine	Molport-003-934-222	1.000
Phenylalanine	Molport-003-930-436	1.000
Proline	Molport-003-934-152	1.000
Serine	Molport-003-934-205	1.000
Threonine	Molport-023-331-007	1.000
Tryptophan	Molport-003-932-125	1.000
Tyrosine	Molport-003-934-162	1.000
Valine	Molport-000-145-945	1.000
Zwitterionic Alanine	Molport-001-757-318	0.316
Zwitterionic Asparagine	Molport-003-932-124	0.385
Zwitterionic Aspartic Acid	Molport-000-162-471	0.417
Zwitterionic Cysteine	Molport-008-267-648	0.320
Zwitterionic Glutamic Acid	Molport-003-725-296	0.407
Zwitterionic Glutamine	Molport-000-156-122	0.429
Zwitterionic Glycine	Molport-008-267-708	0.182
Zwitterionic Histidine (N-delta)	Molport-011-127-945	0.541
Zwitterionic Histidine (N-eta)	Molport-051-737-965	0.439
Zwitterionic Histidine (neutral)	Molport-051-737-965	0.439
Zwitterionic Isoleucine	Molport-003-698-931	0.393
Zwitterionic Leucine	Molport-000-145-428	0.370
Zwitterionic Methionine	Molport-000-150-578	0.433
Zwitterionic Negative Aspartic Acid	Molport-000-162-471	0.269
Zwitterionic Negative Glutamic Acid	Molport-003-725-296	0.233
Zwitterionic Phenylalanine	Molport-001-834-415	0.484
Zwitterionic Positive Arginine	Molport-008-267-643	0.238
Zwitterionic Positive Histidine (positive)	Molport-051-737-965	0.229
Zwitterionic Positive Lysine	Molport-020-014-656	0.243
Zwitterionic Proline	Molport-002-916-126	0.321
Zwitterionic Serine	Molport-003-934-153	0.292
Zwitterionic Threonine	Molport-001-835-643	0.320
Zwitterionic Tryptophan	Molport-001-813-192	0.585
Zwitterionic Tyrosine	Molport-003-934-162	0.485
Zwitterionic Valine	Molport-000-145-945	0.292

Table 5
Second Most Similar Non-natural AA

AAName	NNAAName	Similarity
Alanine	Molport-003-934-647	1.000
Arginine	Molport-006-394-208	0.893
Asparagine	Molport-003-986-289	0.650
Aspartic Acid	Molport-003-932-124	0.650
Glutamic Acid	Molport-002-344-02	0.727
Glutamine	Molport-003-986-303	0.682
Glycine	Molport-003-934-564	0.333
Histidine (neutral)	Molport-051-737-965	0.611
Isoleucine	Molport-003-698-931	0.652
Leucine	Molport-003-934-581	1.000
Lysine	Molport-020-003-905	0.857
Methionine	Molport-051-692-586	1.000
Phenylalanine	Molport-051-693-323	1.000
Proline	Molport-002-500-233	0.857
Serine	Molport-051-602-461	1.000
Threonine	Molport-003-934-206	0.684
Tryptophan	Molport-001-813-192	0.778
Tyrosine	Molport-001-758-55	0.714
Valine	Molport-003-934-587	1.000
Zwitterionic Alanine	Molport-001-791-314	0.316
Zwitterionic Asparagine	Molport-003-987-178	0.385
Zwitterionic Aspartic Acid	Molport-046-834-716	0.391
Zwitterionic Cysteine	Molport-008-267-667	0.320
Zwitterionic Glutamic Acid	Molport-003-986-303	0.407
Zwitterionic Glutamine	Molport-003-932-123	0.429
Zwitterionic Glycine	Molport-020-004-612	0.182
Zwitterionic Histidine (N-delta)	Molport-011-127-975	0.541
Zwitterionic Histidine (N-eta)	Molport-051-737-966	0.439
Zwitterionic Histidine (neutral)	Molport-051-737-966	0.439
Zwitterionic Isoleucine	Molport-021-212-06	0.310
Zwitterionic Leucine	Molport-001-757-006	0.370
Zwitterionic Methionine	Molport-003-894-343	0.433
Zwitterionic Negative Aspartic Acid	Molport-046-834-716	0.240
Zwitterionic Negative Glutamic Acid	Molport-003-986-303	0.233
Zwitterionic Phenylalanine	Molport-003-930-436	0.484
Zwitterionic Positive Arginine	Molport-008-267-651	0.238
Zwitterionic Positive Histidine (positive)	Molport-051-737-966	0.229
Zwitterionic Positive Lysine	Molport-020-014-657	0.243
Zwitterionic Proline	Molport-003-725-272	0.321
Zwitterionic Serine	Molport-003-934-205	0.292
Zwitterionic Threonine	Molport-023-331-007	0.320
Zwitterionic Tryptophan	Molport-003-932-125	0.585
Zwitterionic Tyrosine	Molport-006-666-30	0.485
Zwitterionic Valine	Molport-001-756-78	0.292

Table 6
Third Most Similar Non-natural AA

AAName	NNAAName	Similarity
Alanine	Molport-006-705-555	1.000
Arginine	Molport-003-987-423	0.733
Asparagine	Molport-003-987-178	0.619
Aspartic Acid	Molport-003-934-063	0.650
Glutamic Acid	Molport-051-787-435	0.696
Glutamine	Molport-000-156-122	0.652
Glycine	Molport-003-934-801	0.333
Histidine (neutral)	Molport-051-737-966	0.611
Isoleucine	Molport-005-934-106	0.652
Leucine	Molport-003-934-583	1.000
Lysine	Molport-006-705-651	0.682
Methionine	Molport-051-694-501	1.000
Phenylalanine	Molport-051-693-572	1.000
Proline	Molport-047-646-814	0.818
Serine	Molport-003-725-379	0.579
Threonine	Molport-003-983-076	0.684
Tryptophan	Molport-003-934-131	0.778
Tyrosine	Molport-006-666-30	0.714
Valine	Molport-003-934-588	1.000
Zwitterionic Alanine	Molport-001-794-136	0.316
Zwitterionic Asparagine	Molport-006-666-304	0.385
Zwitterionic Aspartic Acid	Molport-000-003-735	0.360
Zwitterionic Cysteine	Molport-044-832-334	0.200
Zwitterionic Glutamic Acid	Molport-005-942-153	0.407
Zwitterionic Glutamine	Molport-000-164-874	0.400
Zwitterionic Glycine	Molport-006-705-557	0.174
Zwitterionic Histidine (N-delta)	Molport-020-004-756	0.541
Zwitterionic Histidine (N-eta)	Molport-051-737-967	0.439
Zwitterionic Histidine (neutral)	Molport-051-737-967	0.439
Zwitterionic Isoleucine	Molport-000-145-944	0.300
Zwitterionic Leucine	Molport-003-700-319	0.370
Zwitterionic Methionine	Molport-003-934-222	0.433
Zwitterionic Negative Aspartic Acid	Molport-051-787-373	0.226
Zwitterionic Negative Glutamic Acid	Molport-005-942-153	0.233
Zwitterionic Phenylalanine	Molport-003-934-09	0.484
Zwitterionic Positive Arginine	Molport-003-927-113	0.237
Zwitterionic Positive Histidine (positive)	Molport-051-737-967	0.229
Zwitterionic Positive Lysine	Molport-023-332-708	0.243
Zwitterionic Proline	Molport-003-725-316	0.321
Zwitterionic Serine	Molport-011-285-017	0.292
Zwitterionic Threonine	Molport-035-395-092	0.296
Zwitterionic Tryptophan	Molport-003-934-131	0.585
Zwitterionic Tyrosine	Molport-011-285-058	0.485
Zwitterionic Valine	Molport-001-757-007	0.292

The most, second most, and third most similar NNAAs are shown in Table 4, Table 5, and Table 6, respectively. Note that for some natural amino acids the Tanimoto similarity for the top 3 most similar NNAAs is the same (e.g., Alanine). Besides that, it is also possible that there is a fourth NNAA which has a similarity measure equal to the the third most similar NNAA, but which is of course not shown in any of the tables. In Table 4, we observe that a lot of AA fingerprints are exactly the same as the fingerprint of the corresponding most similar NNAA (i.e., have Tanimoto similarity of 1). This holds especially for the *non-zwitterionic* AAs. In Table 5 and Table 6, we observe that the Tanimoto similarities decline as we move from the most similar NNAA to the second most and third most similar NNAA (logically), but the similarity for most of the *non-zwitterionic* AAs are still high.

10. In the three tables above, we observe that all non-natural amino acids (NNAAs) have the chemical supplier *MolPort* in their name. These NNAAs can be purchased from MolPort. Furthermore, we observe in Table 4 that most *non-zwitterionic* natural amino acids have a Tanimoto similarity of 1 with their corresponding most similar NNAA (which is supplied by MolPort). Therefore, it is very likely that MolPort sells *non-zwitterionic* natural amino acids, or at least NNAAs which have a very similar function as them (*similar compounds have similar properties*). For the *zwitterionic* natural amino acids, however, the most similar NNAAs have a much smaller Tanimoto similarity (Table 4). Thus, it is very unlikely we can buy a *zwitterionic* natural amino acid from MolPort.

11. In this section, we compare the running times of the serial implementations of the Tanimoto similarity function against two parallel implementations using forks and sockets. Therefore, we first define the two functions which use parallel computing to compare all fingerprints in (a subset of) `dfAA` with all fingerprints in (a subset of) `dfNNAA`. These functions are named `get_tanimoto_sockets` and `get_tanimoto_forks` for the sockets and forks implementation, respectively. Inside these functions, the Tanimoto similarity between two fingerprints is calculated with `get_tanimoto_vectorized`. Besides that, we also define a function `make_fingerprint_list` which, given a dataframe of amino acids, returns a named list with integer-vector fingerprints as elements. Note that we could just make this fingerprint-list once, and then pass (a subset of it) every time to the parallel functions. However, we decide to make this fingerprint-list inside the parallel functions, such that we obtain running times in a setting that is as realistic as possible.

```
make_fingerprint_list <- function(amino_acids) {
  FP_list <- lapply(
    X = str_split(amino_acids$FP, pattern = " "),
    FUN = as.integer
  )
  names(FP_list) <- amino_acids$Name
  FP_list
}

get_tanimoto_sockets <- function(dfAA, dfNNAA, n_cores) {
```

```
  AA_FP_list <- make_fingerprint_list(dfAA)
  NNAA_FP_list <- make_fingerprint_list(dfNNAA)
  cl <- makeCluster(n_cores)
  clusterExport(cl, c("NNAA_FP_list", "get_tanimoto_vectorized"))
  result <- parLapply(
    cl = cl,
    X = AA_FP_list,
    fun = function(x) {
      sapply(NNAA_FP_list, function(y) get_tanimoto_vectorized(x, y))
    }
  )
  stopCluster(cl)
  bind_rows(result, .id = "AAName")
}

get_tanimoto_forks <- function(dfAA, dfNNAA, n_cores) {
  AA_FP_list <- make_fingerprint_list(dfAA)
  NNAA_FP_list <- make_fingerprint_list(dfNNAA)
  result <- mclapply(
    X = AA_FP_list,
    mc.cores = n_cores,
    FUN = function(x) {
      sapply(NNAA_FP_list, function(y) get_tanimoto_vectorized(x, y))
    }
  )
  bind_rows(result, .id = "AAName")
}
```

Furthermore, note that the (non-parallel) Tanimoto functions we defined earlier only calculate the similarity measure for two fingerprints. In order to calculate the Tanimoto similarities between all fingerprints in (a subset of) `dfAA` and `dfNNAA`, we define the following `get_tanimoto_serial` function:

```
get_tanimoto_serial <- function(dfAA, dfNNAA, func) {
  AA_FP_list <- make_fingerprint_list(dfAA)
  NNAA_FP_list <- make_fingerprint_list(dfNNAA)
  result <- lapply(
    X = AA_FP_list,
    FUN = function(x) {
      sapply(NNAA_FP_list, function(y) func(x, y))
    }
  )
```

```
  bind_rows(result, .id = "AAName")
}
```

Now, we compute the running times for all the Tanimoto similarity functions we defined in this paper. The function `get_grid_of_times`, defined below, has as input arguments the original dataframes `dfAA` and `dfNNAA`, and obtains for various grid sizes of AAs and NNAAs the running time of the functions with `microbenchmark`. Specifically, starting with 2 AAs in `dfAA` and 5 NNAAs in `dfNNAA`, we derive the running time of comparing all AAs in `dfAA` with all NNAAs in `dfNNAA`. After obtaining this running time, we increase the number of amino acids in `dfAA` and `dfNNAA` by 2 and 5, respectively, and do the same. This procedure stops when we have compared a `dfAA` with 20 rows with a `dfNNAA` of 50 rows, to give a total of 1000 fingerprint-comparisons. Furthermore, we would like to highlight that we collect the average running time of `times = 20L` function calls in `microbenchmark`. Of course, if we set `times` to a larger integer value, we would get more robust results, but this will also increase the running time of the function `get_grid_of_times` itself.

```
get_grid_of_times <- function(dfAA, dfNNAA) {
  AA_seq <- seq(2, 20, 2)
  NNAA_seq <- seq(5, 50, 5)
  N <- min(length(AA_seq), length(NNAA_seq))
  result <- NULL

  func_vectorized <- get_tanimoto_vectorized
  for (i in 1:N) {
    fp1s <- dfAA[1:AA_seq[i], ]
    fp2s <- dfNNAA[1:NNAA_seq[i], ]
    times <- microbenchmark(
      "R loop" = get_tanimoto_serial(fp1s, fp2s, get_tanimoto_similarity),
      "R vectorized" = get_tanimoto_serial(fp1s, fp2s, func_vectorized),
      "bitops" = get_tanimoto_serial(fp1s, fp2s, get_tanimoto_bitops),
      "Rcpp" = get_tanimoto_serial(fp1s, fp2s, get_tanimoto_Rcpp),
      "forks" = get_tanimoto_forks(fp1s, fp2s, n_cores = 2),
      "sockets" = get_tanimoto_sockets(fp1s, fp2s, n_cores = 2),
      times = 20L,
      unit = "s"
    )

    summary <- cbind(
      nAA = rep(AA_seq[i], times = 6),
      nNNAA = rep(NNAA_seq[i], times = 6),
      summary(times)[, c("expr", "mean")]
    )
```

```
    result <- bind_rows(result, summary)
  }
  result %>%
    rename(time = mean)
}

df_times <- get_grid_of_times(dfAA, dfNNAA) %>%
  mutate(comparisons = nAA * nNNAA)
```

Now that we have collected the running times and stored them in `df_times`, we will plot these times against the total number of fingerprint comparisons. Figure 5 presents the resulting plot. We observe that the running times for 1000 fingerprint comparisons are similar to the running times in Table 3, as expected. Besides that, we observe that the running time for `get_tanimoto_sockets` is substantially larger than the running time for the other methods. This can be explained by the large *overhead* of calling this parallel function, since already for a handful of comparisons the running time is beyond the maximum running time of the others and this stays relatively constant when increasing the number of comparisons. Note that for the sockets implementation we needed to export a function and data to the clusters, this is probably why the overhead is so large, especially in comparison with forks. Even though the `get_tanimoto_forks` also has some *overhead*, we observe that this is considerably smaller than for the sockets implementation. More specifically, we observe that the forks implementation eventually performs faster than *"R loop"* and *"bitops"*. If we would increase the number of fingerprint-comparisons to a bit more than 1000, the forks implementation would probably also be faster than the *"R vectorized"* method. Lastly, Figure 5 shows that the C++ function (*"Rcpp"*) is extremely efficient. If the forks implementation will eventually outperform the C++ implementation, this will probably happen for fingerprint-comparisons well exceeding 1000.

```
ggplot(df_times, mapping = aes(x = comparisons, y = time, colour = expr)) +
  geom_point() +
  geom_line() +
  labs(
    x = "Number of comparisons",
    y = "Time (in seconds)",
    colour = "Method"
  ) +
  scale_colour_manual(values = cbPalette[-1]) +
  theme(legend.background = element_rect(colour = "black"))
```

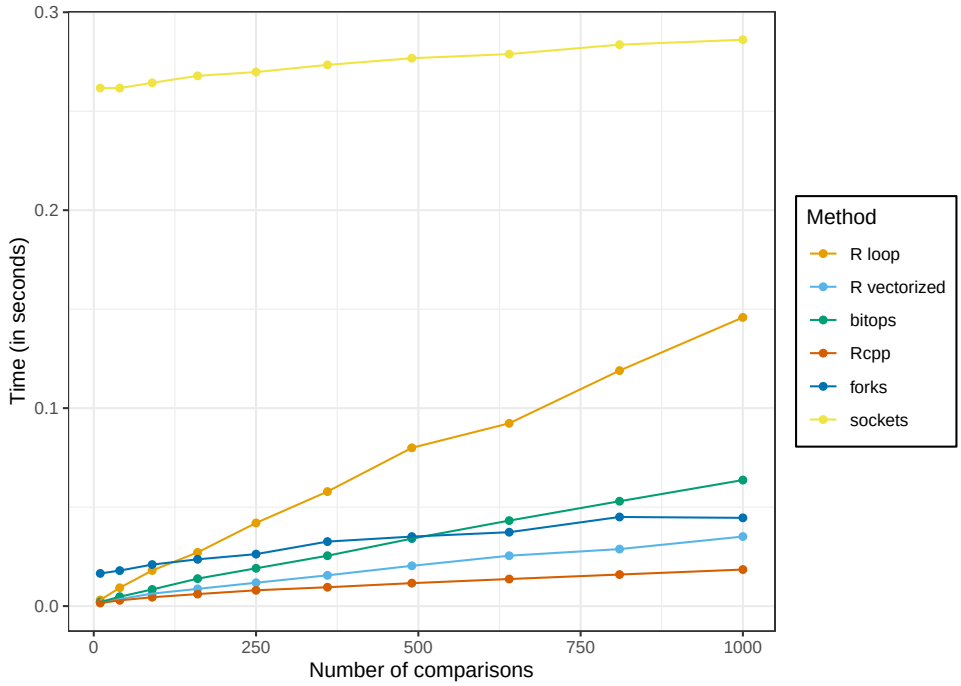


Figure 5: This figure plots the running time in seconds against the number of fingerprint-comparisons for each Tanimoto similarity function defined in this paper.

To conclude this paper, we will fit linear and quadratic models to the running times in Figure 5. Table 7 presents the R^2 values obtained. We observe that the R^2 values for all models are large, and the differences in R^2 between the linear and quadratic models are very small. Therefore, the results in Table 7, together with Figure 5, indicate that the complexity of all functions is linear. Of course, the rate at which the running times increase with the number of comparisons is different for each method (see Figure 5).

```
linear_models <- tapply(
  X = df_times,
  INDEX = df_times$expr,
  FUN = function(x) lm(time ~ comparisons, data = x)
)
R2_linear <- sapply(linear_models, FUN = function(x) summary(x)$r.squared)

quadratic_models <- tapply(
  X = df_times,
  INDEX = df_times$expr,
  FUN = function(x) lm(time ~ comparisons + I(comparisons^2), data = x)
)
R2_quad <- sapply(quadratic_models, FUN = function(x) summary(x)$r.squared)

rbind(
  "Linear" = R2_linear,
```

```
  "Quadratic" = R2_quad
) %>%
  kbl(
    digits = 3,
    format = "latex",
    booktabs = TRUE,
    linesep = "",
    table.envir = "table} \\captionsetup{margin = 60pt",
    caption = paste0("$R^2$ values for the linear and quadratic model ",
                     "regressing running time (in seconds) on the ",
                     "number of comparisons made for each of the ",
                     "methods as used in the figure above.")
  ) %>%
  kable_styling(latex_options = c("HOLD_position"))
```

Table 7
R^2 values for the linear and quadratic model regressing running time (in seconds) on the number of comparisons made for each of the methods as used in the figure above.

	R loop	R vectorized	bitops	Rcpp	forks	sockets
Linear	0.997	0.993	0.998	0.980	0.956	0.976
Quadratic	0.998	0.998	1.000	0.995	0.987	0.995