Aalto University
School of Science
Master's Programme in ICT Innovation

Ádám Kapos

# Methods for Identifying Songs

Master's Thesis

Espoo, June 15, 2015

Supervisors: Professor Heikki Saikkonen, Aalto University School of Science
Associate Professor Zoltán Illés, Eötvös Loránd University

Instructor: András Velvárt, M.Sc. (Tech)

Aalto University
School of Science

| | |
|---|---|
| Aalto University<br>School of Science<br>Master's Programme in ICT Innovation | ABSTRACT OF THE MASTER'S THESIS |

Author: Ádám Kapos

Title: Methods for Identifying Songs

| | | |
|---|---|---|
| Number of pages: 77 | Date: 15.06.2015 | Language: English |

| | |
|---|---|
| Professorship: Software Systems | Code: T-106 |

Supervisor: Professor Heikki Saikkonen, Aalto University School of Science
Associate Professor Zoltán Illés, Eötvös Loránd University

Instructor: András Velvárt, M.Sc. (Tech)

Abstract:

The goal of the thesis was to identify algorithms that can be used to tell if two songs are the same based on the artist, tile and duration metadata. I based my research on metadata for more than 95000 songs from the game SongArc. From this large data set I have selected three smaller data sets to focus on. The first sample set contained the three popular songs, their variances and songs that were similar in either the title or the artist to those songs. The second sample set contained songs with similar titles. The third sample set included songs that had metadata containing non-Latin characters.

These three sample set were used to measure how well algorithms perform at detecting similar songs. I have selected various approximate string matching algorithms to examine. These algorithms were based on two distinct approaches: edit-distance and tokens. My final goal was to improve the accuracy of the algorithms by transforming their input. I have found that removing text starting with the first bracket character improved the results significantly. The edit-distance based algorithms benefited the most, especially when the goal was to find a low number of false positives. Ignoring the artist metadata field if it contained the word "unknown" improved the accuracy of all algorithms. I have found that it had the greatest effect on token based algorithms.

In conclusion, I have found that using my improvements the cosine similarity metric is best to be used to detect if two songs are similar based on the artist and title metadata, being able to detect 87.27% of the similar songs when high accuracy is required at a sensitivity of 0.55. When high accuracy is not required, it is able to detect 96.78% of the similar songs using a sensitivity of 0.42.

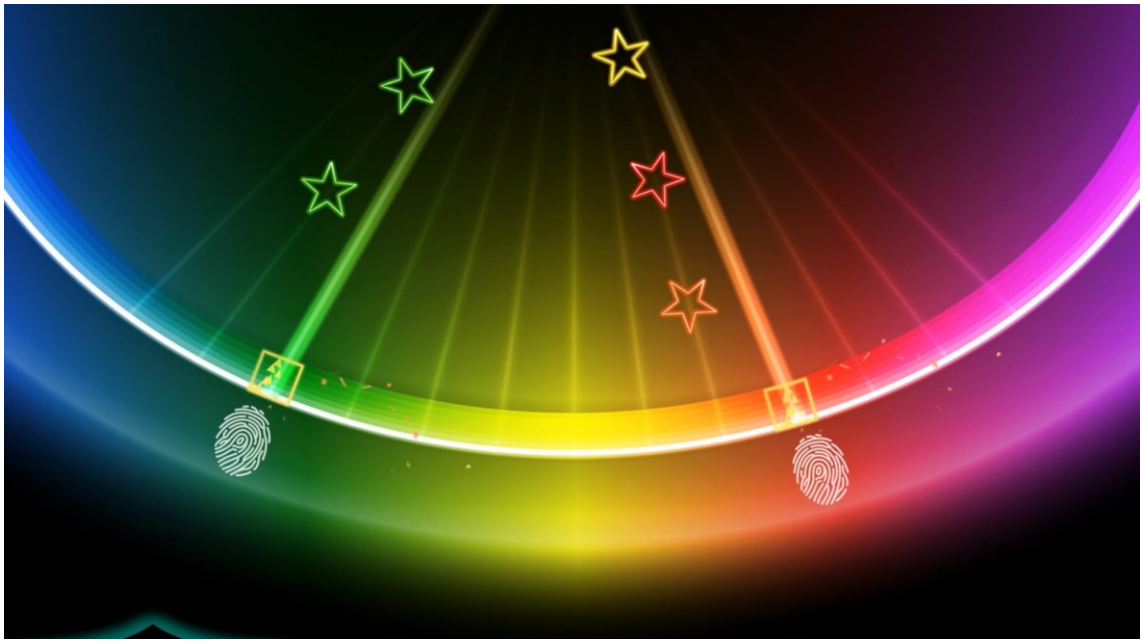
Keywords: Approximate string matching, Fuzzy string matching, Fuzzy string searching, String metric, Song similarity detection

# I Introduction

## 1 The context

The topic of this thesis was born because I faced a problem while working on the game *SongArc*. The game is developed together with András Velvárt and Dávid Kiss. The goal of the game is to make players feel like they are playing the music - without requiring any previous experience in playing instruments.

In SongArc, the players don't play an actual instrument. The game features a fictional instrument that is similar to a piano. There are thirteen lanes on which notes travel from the top to the bottom as time passes. The players have to hit the notes as they reach the white arc by tapping the right position of the screen or shaking their devices. For successful hits they get points. If they miss too many notes the game is over.



For music games like SongArc, it is essential to have popular songs. High budget games like *Guitar Hero* and *Rock Band* include dozens of well-known songs. I contacted Universal Music Hungary regarding the licensing of popular song and I was told that licensing fees start at $10 000 per song. This was clearly not something we could afford for a mobile game, so we had to look for other ways to have music people frequently listen to and want to play with.

We decided to utilize the songs people already have on their devices. The game parses the music library of the device and makes it possible to play with the songs in the library. This way the players can play with their favorite songs.

However, this raised a new problem: How are we going to translate the music into actual gameplay? The pattern of notes the players have to hit has to be defined somehow for each and every song. This is what we call a *sheet*. The solution was to crowdsource the creation of sheets – meaning that the players are the ones who actually create playable content. This is done using the in-game *Sheet Creator*.


It works in a similar way to playing. In *record mode*, as players listen to the song, they can create notes in the same fashion as they would hit a note. Players can fine-tune their creations in *edit mode* and then test them in *play mode*. Once they are happy with the result, they can upload the sheet they created to our servers – but not the song itself.

However, the uploaded sheets lead to a problem too: A player wants to play a sheet that has been created by another user. How can we tell if the player has the song for the sheet?

# 2 The problem

The topic of this thesis is to solve the generalization of the problem described earlier: How can we tell if two songs are actually the same?

One way to approach this problem is to compare the audio of the songs. For example, such algorithms are used by software which can tell what song is on the radio from a short recording. When it comes to analyzing the player's media library, this approach requires access to the raw audio data of the song. However, the songs may be encrypted. The encryption is part of Digital Rights Management (DRM), which aims to control the use of digital content. While DRM is rarely utilized for traditional song purchases, it is widely used by music streaming or subscription services that allow offline listening.

For example songs downloaded using the Xbox Music Pass are protected by DRM on Windows Phone. The song can be played by asking the operating system to do so, but the contents of the file itself could not be analyzed because of the encryption. Currently only the first-party music players created by Microsoft can decrypt these songs.

Since I could not rely on having access to the song file itself, I had to look for other approaches to solve the problem. The information that is available on all major mobile platforms (iOS, Android and Windows Phone) is metadata for songs. This includes artist, title, album name and duration among others.

I decided to rely only on the artist, the title and the duration because using other metadata did not contain significant information from the perspective of the problem. For example I have found that the same song can be found on many albums. While a song is usually included on a single album, it can be featured on compilation albums many times.

The goal of the thesis is to identify algorithms that can be used to decide if two songs are the same based on the artist, tile and duration metadata. Various approximate string matching algorithms that can be used to compare the title and artist information will be investigated. Duration will serve as a heuristic, assuming that two songs which have a big difference in duration cannot be the same.

# II Understanding the Problem

## 1 Looking at Popular Songs

To understand what kind of differences can be in the title and artist metadata of songs, I gathered data from SongArc. When a player uploads a new sheet, the metadata of the song it was created for is also sent to the server. This includes the title, the artist and the duration of the song. I gathered more than 95000 songs this way, which are not equal to each other in all metadata. I will use these as the basis of my research.

To investigate what kind of variances the algorithms will have to deal with, I looked up metadata that belongs to three popular songs. I have chosen these because their popularity resulted in many different versions of metadata.

The first one is “Titanium” by David Guetta with vocals by Sia from 2011. I have found 85 versions for the song.

The second song is “Wake Me Up” by Avicii with vocals by Aloe Blacc from 2013. I have found 95 versions for the song.

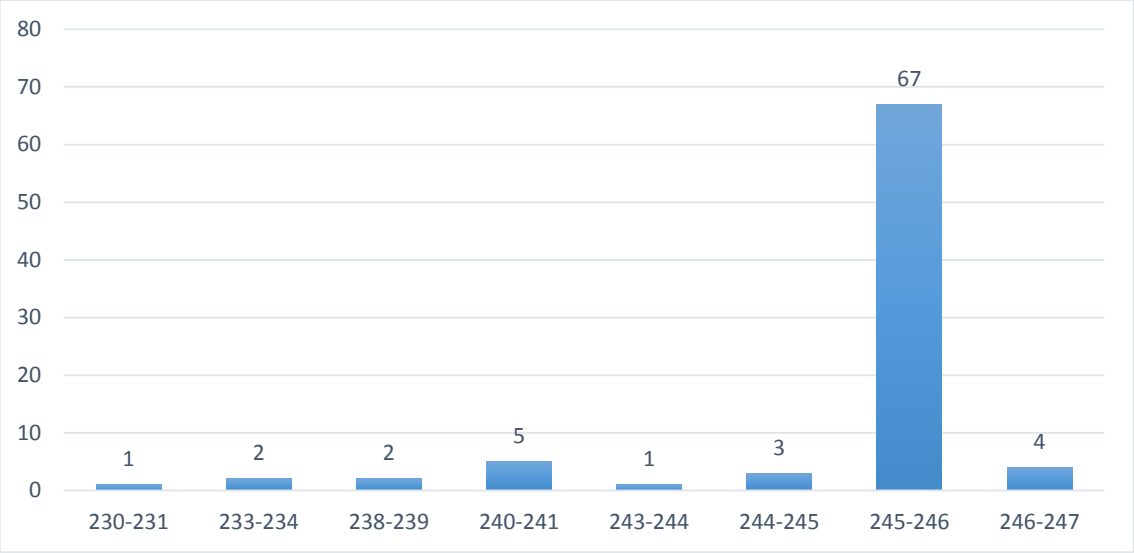
The third one is “Animals” by Martin Garrix from 2014. I have found 91 versions for this song.

Some songs that had a duration of zero. They were most likely incorrectly reported durations. Songs with a duration of zero were not included in the research.

# 2 Variance in Duration

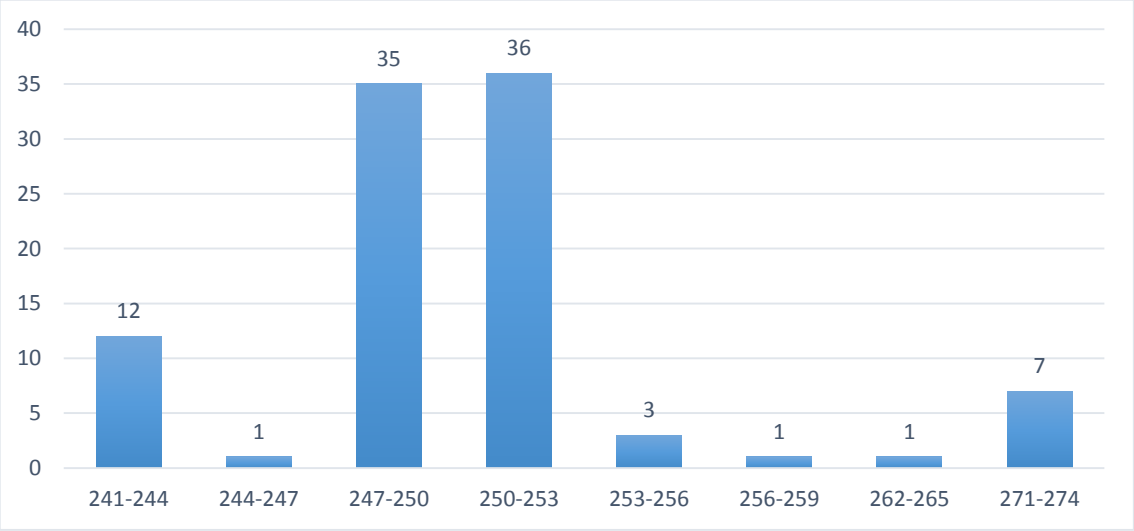
## 2.1 David Guetta featuring Sia - Titanium

In case of this song, a single duration is dominating. All songs are within a range of 17 seconds. 78.8% of the songs are between 245 and 246 seconds in duration. In this case a version is clearly dominating when it comes to duration.



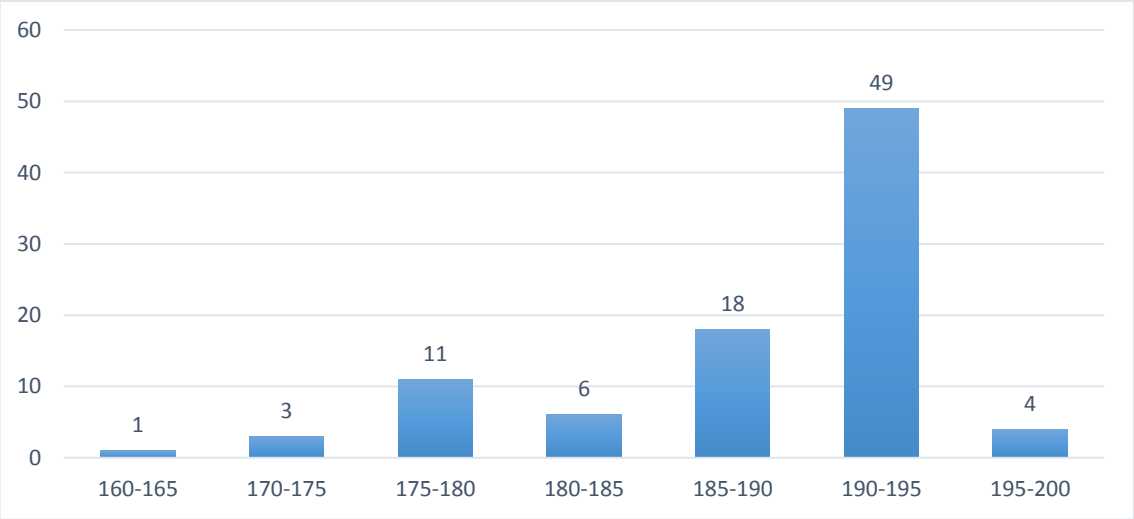

## 2.2 Avicii featuring Aloe Blacc – Wake Me Up

The duration of the song shows that this song has multiple popular versions. The duration of the most popular one is between 247 and 253 seconds. There are shorter and longer versions too. The shortest version is around 10 seconds shorter, while the longest one is roughly 25 seconds longer than the most popular one.



## 2.3 Martin Garrix – Animals

Similarly to the previous song by Avicii, this song has a wide scale of versions. The different versions are within a range of 40 seconds. 73.6% of songs are between 185 and 195 seconds in length. Most other versions are shorter than 185 seconds.




## 2.4 Partitioning and duration

From my sample it seems that when it comes to the duration of the song, there can be many different versions. The range of the length can be as narrow as 20 seconds or as wide as 40 seconds. There is no clear separation between versions in duration.

The lack of clear separation makes the duration of the song hard to use as a tool to find similar or different versions. Assuming that two songs can only be the same if they differ less than 10 seconds in duration leads to a problem: If we take a version of “Animals” that’s 176.66 seconds long, it can be declared similar to the 184.97 second version. The 184.97 second version can be declared similar to the 191.71 second version. But the 176.66 second version cannot be declared similar to the 191.71 second version because the difference in duration is too large.

If duration is taken into account, the similarity is lacks the property of transitivity. It is not an equivalence relation, hence it will not create partitions (equivalence classes) of songs that are the same. Therefore I will not take the duration of the songs into account in this thesis.

# 3 Variance in Title and Artist

Among the songs there are many that were only different in length. I selected 15 title and artist pairs for each song to show what kind of variances we can expect.

## 3.1 David Guetta featuring Sia - Titanium

| **Artist** | **Title** |
|---|---|
| David Gueta & Sia | Titanium |
| David Guetta | Titanium |
| David Guetta | Titanium (ft. Sia) |
| David Guetta | Titanium (feat. Sia) |
| David Guetta | Titanum (featuring Sia) |
| David Guetta - Sia | Titanium (Feat. Sia) |
| David Guetta feat. Sia | Titanium (www.primemusic.ru) |
| David Guetta&Sia | Titanium |
| David Guetta,Sia | Titanium (feat. Sia) |
| Sia | Titanim |
| Unknown | David Guetta - Titanium ft. Sia |
| Unknown | David Guetta  Titanium ft Sia.mp3 |
| Unknown | david_guetta_titanium_ft._sia_mp3_66896 |
| Unknown | David Guetta feat. Sia - Titanium Official Music |
| Unknown | David Guetta Feat.Sia- Titanium.mp3 |

The main source of variation is that there were two artists who produced this song. David Guetta is the main artist and Sia is the featured artist. In most cases David Guetta is the artist. The name of Sia is added either after the name of David Guetta or after the title of the song. The name of David Guetta is missing in 3 cases. Knowing that song titles are not unique, pairing those songs to song that only have David Guetta as the artist is not possible without additional knowledge.

The secondary source of variation is missing or not properly populated metadata. 12 songs out of 85 have “Unknown” as the artist and have both the artist and the title in the title field. In 2 cases, the artist is not present in either fields. This is most likely because the song did not have stored metadata and the name of the file was used to populate the artist field, hence the mp3 extension that is present at the end of the title field in some cases.

Misspellings are an infrequent source of variation. Out of 85 songs, only 3 are misspelled, either as “Titanum” or “Titanim”. In a single song, David Guetta was spelled as “David Gueta”.

Finally, illegal music download sites may add their URL to the metadata. In two songs, “www.primemusic.ru” was added to the title of the song in brackets. In one song, “FULETEO.CO” was added to the title.

## 3.2 Avicii featuring Aloe Blacc – Wake Me Up

| **Artist** | **Title** |
|---|---|
| Avicii | Wake Me Up |
| avicii | so wake me up |
| Avicii | Wake Me Up (Lyric Video) |
| Avicii | 01. Avicii Ft. Aloe Blacc - Wake Me Up [128].mp3 |
| Avicii | Avicii feat Aloe Blacc - Wake Me Up (Radio Edit) |
| Avicii | Wake Me Up (Official Video) |
| Avicii (3nfermushDJ) | Wake Me Up (feat. Aloe Blacc) |
| avicii feat aloe black | wake me up |
| Avicii feat. Aloe Blacc | Wake Me Up (Radio Edit) - http://soundvor.ru/ |
| Unknown | Avicii - Wake Me Up.m4a |
| Unknown | aviicii-wake me up (1).mp3 |
| Unknown | vitordouglas - Avicii - Wake Me Up.mp3 |
| Unknown | Avicii - Avicii - Wake Me Up (Lyric Video)-[www_flvto_com] (2) |
| Unknown | Wake Me Up  Avicii ft. Aloe Blacc  Lyrics |
| Unknown | Avicii  Wake Me Up ft Aloe Blacc (mashup music) |

This song contains similar sources of variation as the previous song. While there is an artist who is featured in the song, his name appears much less frequently as in the previous case. There is no song that only has his name as the artist.

It seems to me that for this song the main source of variation is the lack of clear sources of music. Many titles have a “Lyric Video” or “Official Video” part. One of those contains an URL of a website where people can turn YouTube videos into MP3 songs.

42 songs out of the 95 have improperly populated metadata, with “Unknown” as the artist.

There is only a single misspelling that has “Aviicii” as the artist in the title field.

## 3.3 Margin Garrix – Animals

| Artist | Title |
|---|---|
| Martin Garrix | Animals |
| Martin Garrix | Animals (Radio Edit).mp3 |
| Martin Garrix | Animals (UK Radio Edit) |
| Martin Garrix | Animals (Original Mix) |
| Martin Garrix | Animals (Official Video).mp3 |
| Unknown | MArtin Garrix - Animals |
| Unknown | Martin_Garrix_-_Animals.mp3 |
| Unknown | Martin Garrix - Animals (Official Video) |
| Unknown | Martin Garrix - Animals (Official Video HD).mp3 |
| Unknown | Martin Garrix - Animals (Official Video) - YouTube |
| Unknown | Dubstep-Martin Garrix - Animals (Official Audio) |
| Unknown | Martin Garrix - Animals OFFICIAL TV CENSORED VIDEO HD |
| Unknown | Martin Garrix - Animals OFFICIAL VIDEO HD-www_flvto_com |
| Unknown | martin_garrix_animals_official_video_mp3_79257 |
| Unknown | Martin%20Garrix%20-%20Animals%20(Official%20Video).mp3 |

Albeit there is no featured artist in this song, there are still plenty of variations in the title and artist of this song, mainly because of different sources people used to obtain the song. In this case there were also three different versions: radio edit, UK radio edit and original mix. Looking at the duration of these versions, it seems that the radio edit is 10 seconds shorter than the original mix. The UK radio edit is even shorter than the regular radio edit. In this case there was not a single misspelling.

55 songs out of 91 have “Unknown” as the artist. This is more than half of the songs.

## 3.4 Similarities in the Title and Artist Fields

The best approach to find similar songs is to use approximate string matching algorithms. Spelling mistakes are infrequent. Therefore I expect algorithms that focus on correcting misspellings will not have edge over others. This includes algorithms that are based on similarly sounding words.

My sample shows that if there is a featured artist, the name of the featured artist can be both in the artist and the title field. I expect those algorithms to perform best that are not sensitive to interchanging words (e.g. “Apple Tree” is similar to “Tree Apple”).

I will look at reducing the variation of metadata in Chapter 6, which can improve results.

# III Method Used

## 1 Background Study

To figure out what kind of algorithms can be used to solve the problem, I have used literature study. I have found that there is a wide array of methods to calculate how similar two strings are. My key source for such methods was Sam Chapman's work at the University of Sheffield [1]. It describes 30 methods for approximate string matching. These methods are generally called metrics and output a value, which denotes how similar or dissimilar two string are. For some metrics high values mean more differences, while for others low values mean more differences. In addition to Chapman's work, I relied on the introduction of the paper by Wang et al [2]. The paper mainly focuses on execution speed, something which I decided not to investigate in this thesis.

In 2004, Chapman developed a library called *SimMetrics*. I have utilized its C# implementation in my research [3]. I have also used implementations from the *FuzzyString* library [4]. For the implementation of *Double Metaphone*, I have used Adam Nelson's work [5]. I have utilized the Jaro-Winkler distance implementation from the *Lucene.Net* library [6]. I have verified that all algorithms work correctly, incorporating my own fixes. In order to compare the output of the metrics, I have normalized their output to a range of 0 to 1 if no normalization was present, where 1 means an excellent match.

## 2 Sample Sets

To be able to measure how well different algorithms perform I have created three sets of songs. My goal with these is to measure the performance of algorithms in three areas.

The first set measures how well algorithms perform at detecting similar songs. The second set measures how well they can detect songs that are not similar. The third set measures how well the algorithms perform when non-Latin characters are present.

## 2.1 Popular Songs

The first sample set wishes to represent the general music taste of people. Its goal is to show how well an algorithm performs at detecting songs that are the same. This sample set includes the three songs from Chapter 2:

- Avicii featuring Aloe Blacc – Wake Me Up
- David Guetta featuring Sia – Titanium
- Martin Garrix – Animals

To be able to detect false positives, I added a number of songs that could potentially be detected as matches. They are either from the same artist or have similar title. Variations of the following songs were added:

- Avicii – You Make Me
- Avicii – Silhouettes
- Avicii – The Nights
- Chris Brown – Don't Wake Me Up
- Green Day – Wake Me Up When September Ends
- Martin Garrix & Jay Hardway – Wizard
- Dimitri Vegas, Martin Garrix, Like Mike – Tremor
- Afrojack & Martin Garrix – Turn Up The Speakers
- Martin Garrix & MOTi – Virus
- Maroon 5 – Animals
- Muse – Animals
- David Guetta featuring Ne-Yo and Akon – Play Hard
- David Guetta featuring Akon – Sexy Chick
- David Guetta featuring Sam Martin – Lovers On The Sun
- Sia – Elastic Heart
- Sia – Cellophane
- Sia – Chandelier

## 2.2 Similar Songs

The second sample set includes songs that are similar in title. The goal of this sample set is to show how well an algorithm performs at detecting songs that are not the same.

It contains variations of the following songs:

- Simple Plan – Welcome To My Life
- The Strokes – Welcome To Japan
- Taylor Swift – Welcome To New York
- Avenged Sevenfold – Welcome To The Family
- My Chemical Romance – Welcome To The Black Parade
- Guns N' Roses – Welcome To The Jungle
- Kanye West featuring Kid Cudi – Welcome To Heartbreak
- Manian – Welcome To The Club
- DJ Tiesto – Welcome To Ibiza
- Joan Jett & the Blackhearts – I Love Rock N' Roll
- Britney Spears – I Love Rock 'N' Roll
- Apocalyptica – One
- Metallica – One
- Depapepe – One
- Demi Lovato – Let It Go
- The Neighbourhood – Let It Go
- Christophe Beck – Let It Go
- Idina Menzel – Let It Go

## 2.3 Foreign Songs

The other two sample sets only include songs with English titles. The third sample set contains songs that have non-Latin characters. Since SongArc, the source of the songs, does not support any language other than English, there is only a small number of foreign songs and those have small variance. Because of this, the sample set does not contain the same level of challenges the other sets included. The sole goal is to find out whether the algorithm has the basic ability to work with non-Latin alphabets.

# 3 Pipeline

I have created a pipeline that can be used to evaluate algorithms. The main design goal was to have a flexible system that is easy to modify. It makes it possible to execute multiple algorithms at the same time, verify the results and generate summaries.

The pipeline can be built using five types of components. Each of them have different roles. The five types of components are executed as steps, sequentially for a provided list of songs.

## 3.1 Preprocessing

The first step is the preprocessing. Multiple preprocessing components can be added to the pipeline. The components will operate sequentially on all songs. The input of the first component will be the list of provided songs. The output of the first component will be the input of the second and so on.

Preprocessing components may add additional information to processed songs. For example fields of the song can be converted to lower-case to ignore case differences later on, or n-gram tokens can be generated for the title and artist of the song in this step. In practice, adding preprocessing steps is done the following way:

```
public class SamplePipeline : Pipeline<Song>
{
    public SamplePipeline()
    {
        this.Preprocessings.Add(new IgnoreCasePreprocessing());
        this.Preprocessings.Add(new NGramTokenizer(3));
    }
}
```

## 3.2 Heuristics

The goal of this step is to improve the execution speed of the matching. All songs are matched against all songs, resulting in $n^2$ of possible matches for n songs. This can take a long time to calculate. Heuristics can be applied to improve execution speed.

Multiple heuristic components can be added. If any of the heuristics return false for a given pair of songs, the matching will not be executed for that pair.

One such heuristic is the duration of the song. In practice, adding heuristics is done the following way:

```
public class SamplePipeline : Pipeline<Song>
{
    public SamplePipeline()
    {
        // Preprocessings

        this.Heuristics.Add(new DurationHeuristic());
    }
}
```

## 3.3 Metrics

In this step metrics can be executed to measure the similarity of two songs. Multiple metrics can be added to save time on preprocessing.

Metrics should evaluate how well two songs match, producing an output between 0 and 1, where 0 means no match and 1 means perfect match.

In practice, adding metrics is done the following way:

```
public class SamplePipeline : Pipeline<Song>
{
    public SamplePipeline()
    {
        // Preprocessings
        // Heuristics

        this.Metrics.Add(new CosineSimilarityMetric());
        this.Metrics.Add(new OverlapCoefficientMetric());
    }
}
```

## 3.4 Verifier

The verifier tells if a pair of songs is considered the same or not. There can be only one verifier.

It is possible to utilize other metrics as verifiers to directly compare the results of two metrics. However, all metrics are prone to make mistakes and without knowing which results are wrong, it is hard to interpret the results. Therefore I decided to use a manually assembled database of matches as the base of my verifier. This database is based on my personal judgement of similarity. It is called *ClusterVerifier* and it creates a lookup table that is utilized to quickly decide if two songs are considered similar or not.

In practice, setting the verifier is done the following way:

```
public class SamplePipeline : Pipeline<Song>
{
    public SamplePipeline()
    {
        // Preprocessings
        // Heuristics
        // Metrics

        this.Verifier = new ClusterVerifier();
    }
}
```

## 3.5 Summarizers

The summarizer creates an output that depends on the needs of the user. It receives the match scores given by the metrics and whether it is an actual match, as determined by the verifier. In my case I have written a summarizer that created CSV files showing how accurate the metrics were at different sensitivity levels.

In practice, adding summarizers is done the following way:

```
public class SamplePipeline : Pipeline<Song>
{
    public SamplePipeline()
    {
        // Preprocessings
        // Heuristics
        // Metrics
        // Verifier

        this.Summarizers.Add(new CsvSummarizer());
    }
}
```

## 3.6 Running the Pipeline

The pipeline can be started by calling the *Run* function the following way:

```
var results = new SamplePipeline().Run(System.Console.WriteLine, l1, l2, l3);
```

It takes a callback as the first parameter. The callback will be called by the pipeline multiple times with a string that contains a small status report. This status report can be written on the console for example using the *System.Console.WriteLine* function.

The callback parameter is followed by one or more lists of songs. Every list of song will be matched only against itself. The results from all lists of songs will be summarized.

# IV Edit-distance Based Algorithms

Edit-distance based approximate string matching algorithms quantify the similarity of two strings based on the minimum number of single character edit operations (i.e. insertion, deletion and substitution) required to transform one string into the other.

In this chapter I analyze how different edit-distance based approximate string matching algorithms perform using only a single, basic preprocessing that converts all song artist and title fields to lower case to ignore case differences.

The pipeline is constructed in the following way:

```
    public class EditDistancePipeline : Pipeline<Song>
    {
        public EditDistancePipeline()
        {
            this.Preprocessings.Add(new IgnoreCasePreprocessing());

            this.Metrics.Add(new JaroDistanceConcatMetric());
            this.Metrics.Add(new JaroWinklerDistanceConcatMetric());
            this.Metrics.Add(new LevenshteinDistanceConcatMetric());
            this.Metrics.Add(new LongestCommonSubsequenceOverlapConcatMetric());
            this.Metrics.Add(new LongestCommonSubsequenceDiceConcatMetric());
            this.Metrics.Add(new LongestCommonSubsequenceSquareConcatMetric());
            this.Metrics.Add(new LongestCommonSubsequenceJaccardConcatMetric());
            this.Metrics.Add(new LongestCommonSubstringOverlapConcatMetric());
            this.Metrics.Add(new LongestCommonSubstringDiceConcatMetric());
            this.Metrics.Add(new LongestCommonSubstringSquareConcatMetric());
            this.Metrics.Add(new LongestCommonSubstringJaccardConcatMetric());
            this.Metrics.Add(new NeedlemanWunchDistanceMetric());
            this.Metrics.Add(new SmithWatermanDistanceMetric());

            this.Verifier = new ClusterVerifier();

            this.Summarizers.Add(new CsvSummarizer());
        }
    }
```

Heuristics are not used because the three sample sets are small and does not take long time to run the pipeline.

The pipeline can be ran the following way:

```
    var popularSongs = DataSource.GetSongs("Songs.Popular");
    var similarSongs = DataSource.GetSongs("Songs.Similar");
    var foreignSongs = DataSource.GetSongs("Songs.Foreign");

    var results = new EditDistancePipeline().Run(System.Console.WriteLine,
popularSongs, similarSongs, foreignSongs);
```

In the pipeline, the summarizer will generate strings which contain how well the metrics perform at different sensitivity values in a CSV file format. These results are analyzed in this chapter, highlighting three particular areas: First, how many false positives (mismatches) there are when 90% of the similar songs are found? Second, how many similar songs are found of only 5% false positives are allowed? Finally, how many similar songs are found of only 0.5% false positives are allowed?

# 1 Levenshtein Distance

## 1.1 Definition

Levenshtein distance is the basic edit distance function. It uses the following edit operations with their respective costs:

- Delete character with a cost of 1
- Insert character with a cost of 1
- Substitute character with a cost of 1

The distance is given as the minimum edit distance which transforms one string into the other. Formally:

$$lev(i,j) = min \begin{cases} lev(i-1,j-1) + d(a_i, b_j) \\ lev(i-1,j) + 1 \\ lev(i,j-1) + 1 \end{cases}$$

Where the *d* function is defined as:

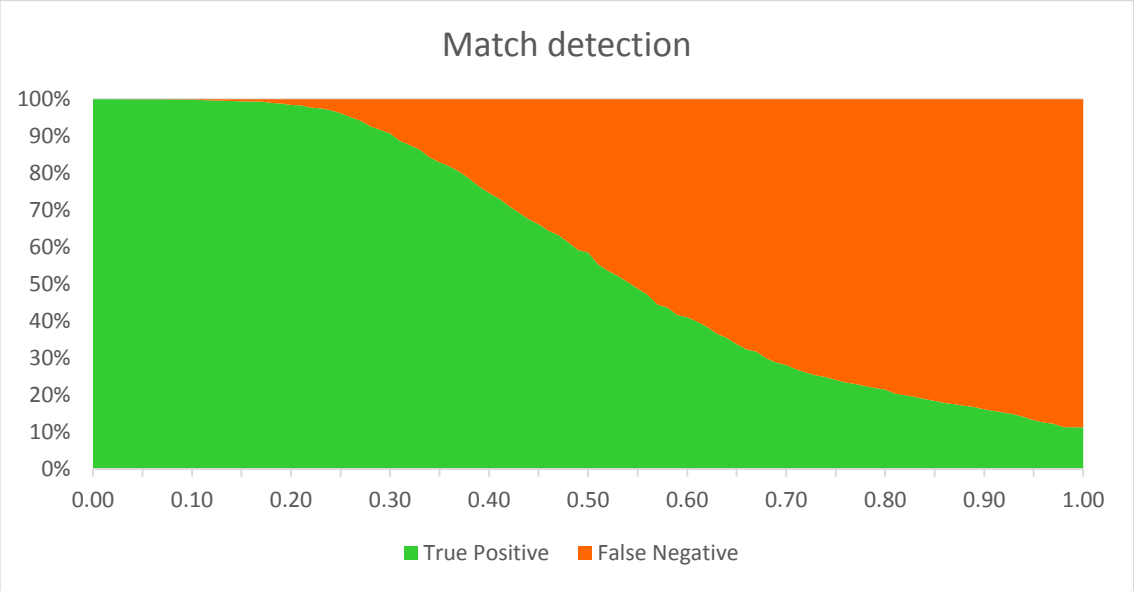
$$d(c,d) = \begin{cases} 0 \; if \; c = d \\ 1 \; if \; c \neq d \end{cases}$$

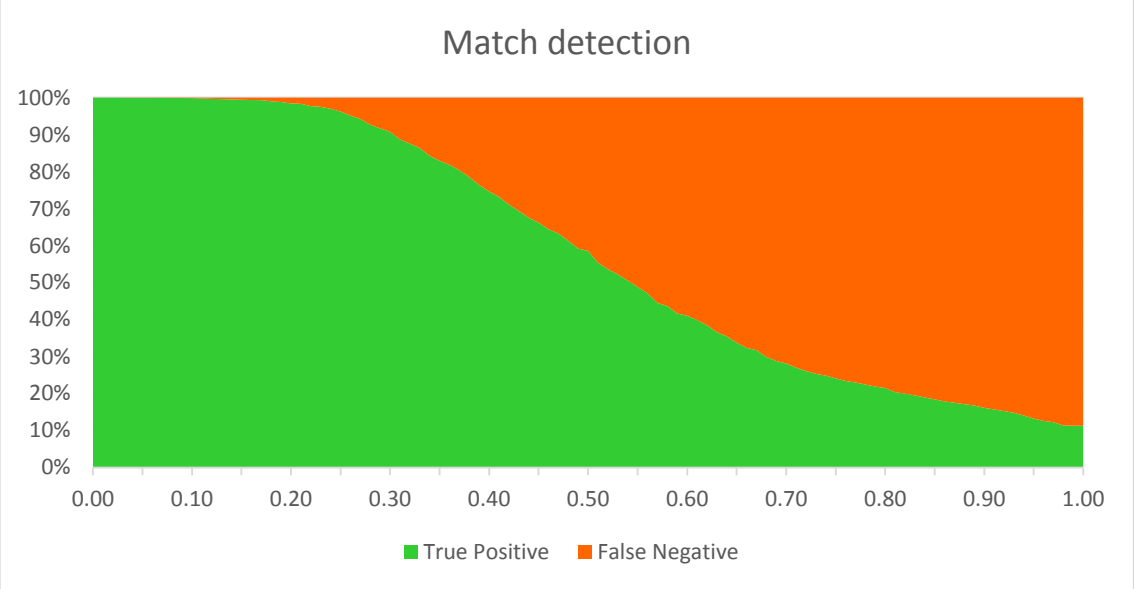
## 1.2 Normalization

The minimum of the Levenshtein distance is 0. The maximum is the length of the longer string. To normalize the output of the Levenshtein distance, the following function is used:

$$1 - \frac{lev(|a|,|b|)}{\max(|a|,|b|)}$$

## 1.3 Results




Using the Levenshtein distance it is possible to detect 90.75% of the matches at a sensitivity of 0.3. However, at this sensitivity it only detects 80.93% of the mismatches, resulting in frequent false positive matches.

For good mismatch detection the sensitivity has to be at least 0.44, at which the algorithm detects 95.19% of the mismatches. However, at this sensitivity it only detects 67.56% of the matches, which means that approximately one in three matches are not detected.

With 99.72% mismatch detection at a sensitivity of 0.62 it detects 38.46% of the matches. In practice this means that 4.5% of the matches identified by the algorithm are false positive and almost two thirds of the similar songs are not identified as matches.

# 2 Needleman-Wunch Distance

## 2.1 Definition

The Needleman-Wunch distance is similar to the Levenshtein distance. It makes is possible to adjust the gap cost of the Levenshtein distance. Formally:

$$nw(i,j) = min\begin{cases} nw(i-1,j-1) + d(a_i, b_j) \\ nw(i-1,j) + G \\ nw(i,j-1) + G \end{cases}$$

where G is the gap cost. The function *d* is an arbitrary distance function. In my test case the distance function will be the same as the one used in Levenshtein distance, except the substitution will have a cost equal to the gap cost:

$$d(c,d) = \begin{cases} 0 \; if \; c = d \\ G \; if \; c \neq d \end{cases}$$

With this *d* function, if the gap cost is 1, the output of the Needleman-Wunch distance is equal to the output of the Levenshtein distance.

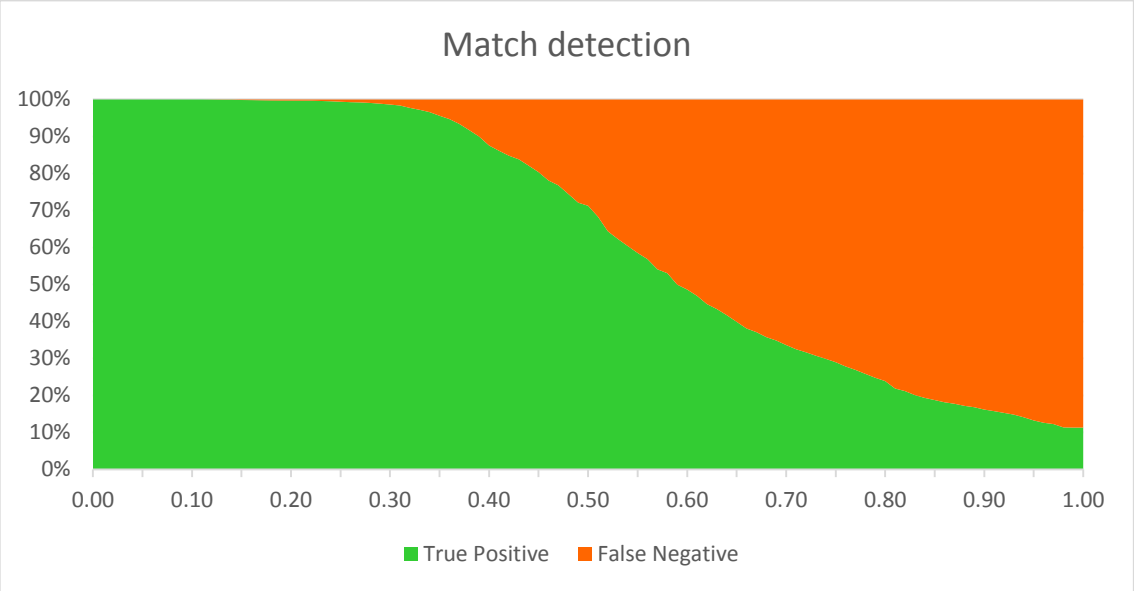
## 2.2 Normalization

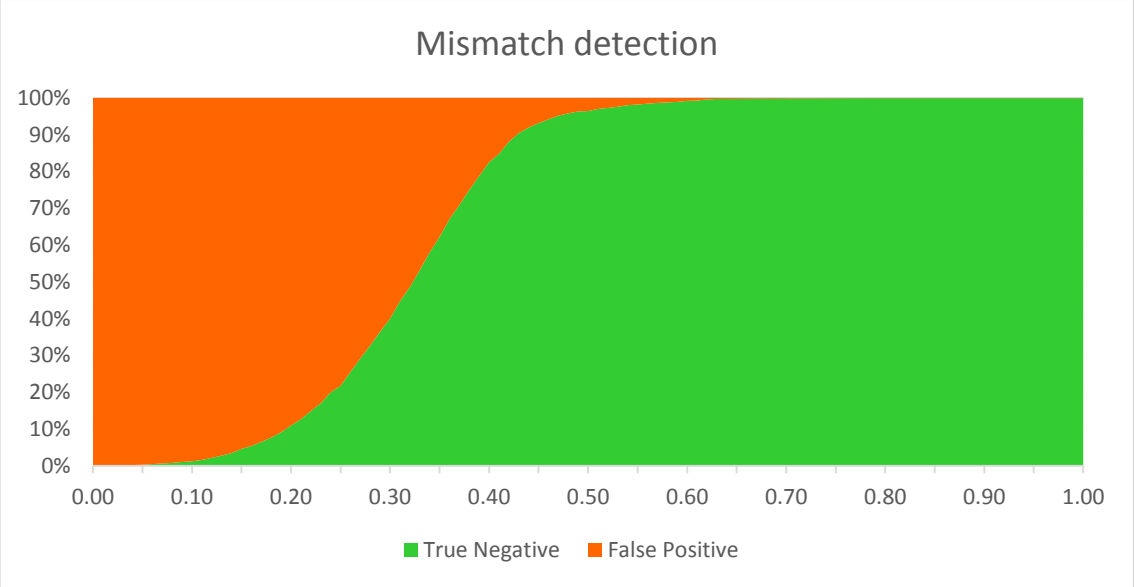
The minimum of the Needleman-Wunch distance is 0. The maximum is the length of the longer string multiplied by the gap cost. To normalize the output of the Needleman-Wunch distance, a function similar to the one used to normalize the output of the Levenshtein distance is used:

$$1 - \frac{nw(|a|,|b|)}{G \cdot \max(|a|,|b|)}$$

## 2.3 Results

The following results were obtained by having a gap cost (G) of 2:




The results are similar to the one of Levenshtein distance, since the two methods are similar, except for the gap cost. By having a gap cost of 2 instead of 1, using the Needleman-Wunch distance it is possible to detect 89.96% of the matches at a sensitivity of 0.39. At this sensitivity it only detects 78.8% of the mismatches, which is slightly lower than the mismatch detection of the Levenshtein distance at a similar level of match detection.

For good mismatch detection the sensitivity has to be at least 0.47, at which the algorithm detects 95.18% of the mismatches. At this sensitivity it detects 76.8% of the matches, which is 9.24% higher than the match detection of the Levenshtein distance at a similar level of mismatch detection.

With 99.60% mismatch detection at a sensitivity of 0.62 it detects 44.69% of the matches, which is significantly better than the accuracy of the Levenshtein distance in this area.

# 3 Smith-Waterman Distance

## 3.1 Definition

The Smith-Waterman distance is also similar to the Levenshtein distance. The similarity distance table is calculated as:

$$sw(i,j) = max \begin{cases} 0 \\ sw(i-1,j-1) + d(a_i, b_j) \\ sw(i-1,j) - G \\ sw(i,j-1) - G \end{cases}$$

where G is the gap cost and in my case the function *d* is defined as:

$$d(c,d) = \begin{cases} 1 \; if \; c = d \\ -2 \; if \; c \neq d \end{cases}$$

The final distance will be maximum over all *i* and *j* values in the table of *sw*.

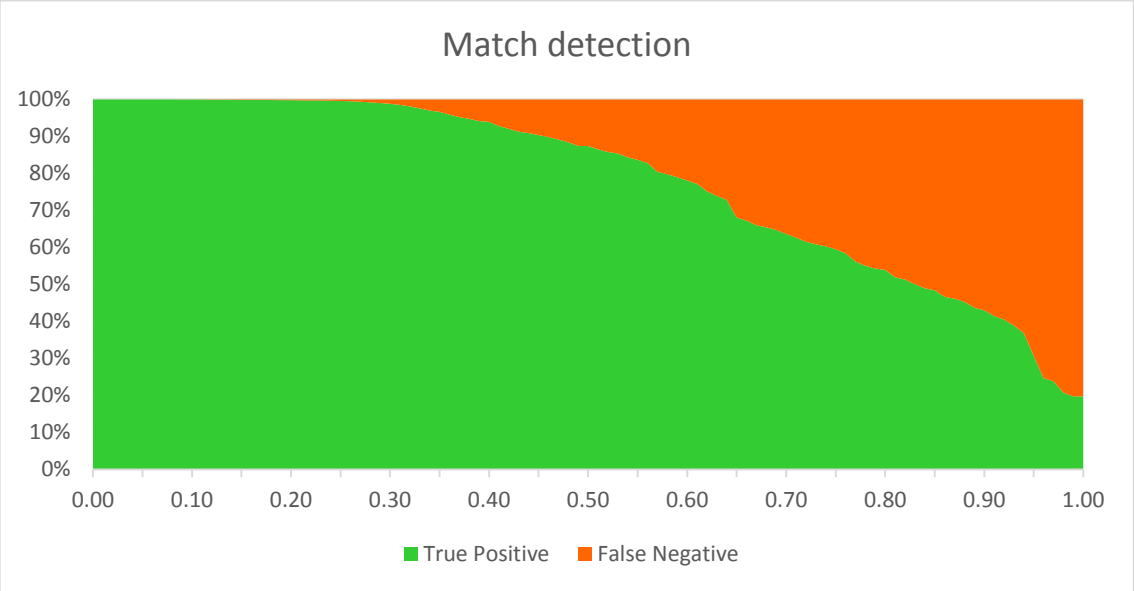
## 3.2 Normalization

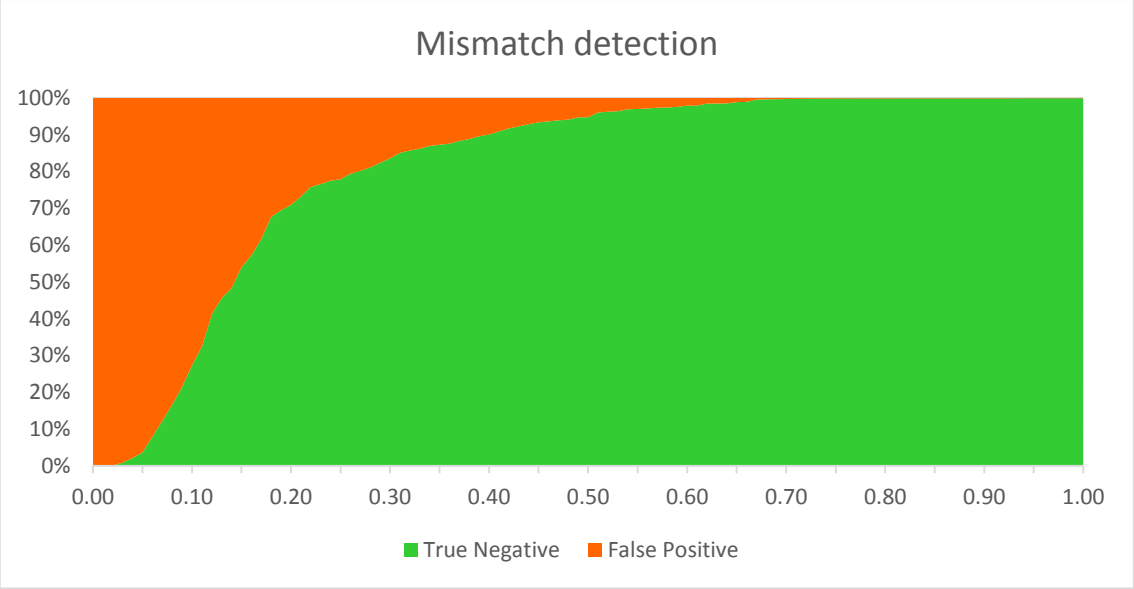
The Smith-Waterman distance indicates the length of the longest approximately matching string, hence the higher the score the better. The minimum of the distance is 0, while the maximum is the length of the shorter string with the *d* function defined earlier. The output is normalized as:

$$\frac{sw(|a|,|b|)}{\min(|a|,|b|)}$$

## 3.3 Results

The following results were obtained by having a gap cost (G) of -0.5:




Using the Smith-Waterman distance it is possible to detect 90.38% of the matches at a sensitivity of 0.45. At this sensitivity it detects 93.37% of the mismatches, which is a great result.

For good mismatch detection the sensitivity has to be at least 0.5, at which the algorithm detects 94.76% of the mismatches. At this sensitivity it detects 87.43% of the matches. The overall result is far better than the results from the Levenshtein distance and the Needleman-Wunch distance.

With 99.56% mismatch detection at a sensitivity of 0.67 it detects 66.00% of the matches, which is significantly better than both the Levenshtein and the Needleman-Wunch distance's accuracy in this area.

# 4 Jaro Distance

## 4.1 Definition

The Jaro distance is one of the more advanced edit distance based string similarity metrics. It was developed by Matthew A. Jaro. He used this method to match the results of an independent test census focusing on hard-to-count population to the census of Tampa, Florida [7]. The goal was to find out how undercounted these people are in the census. Hence, this method is best suited for names of people.

It is calculated as:

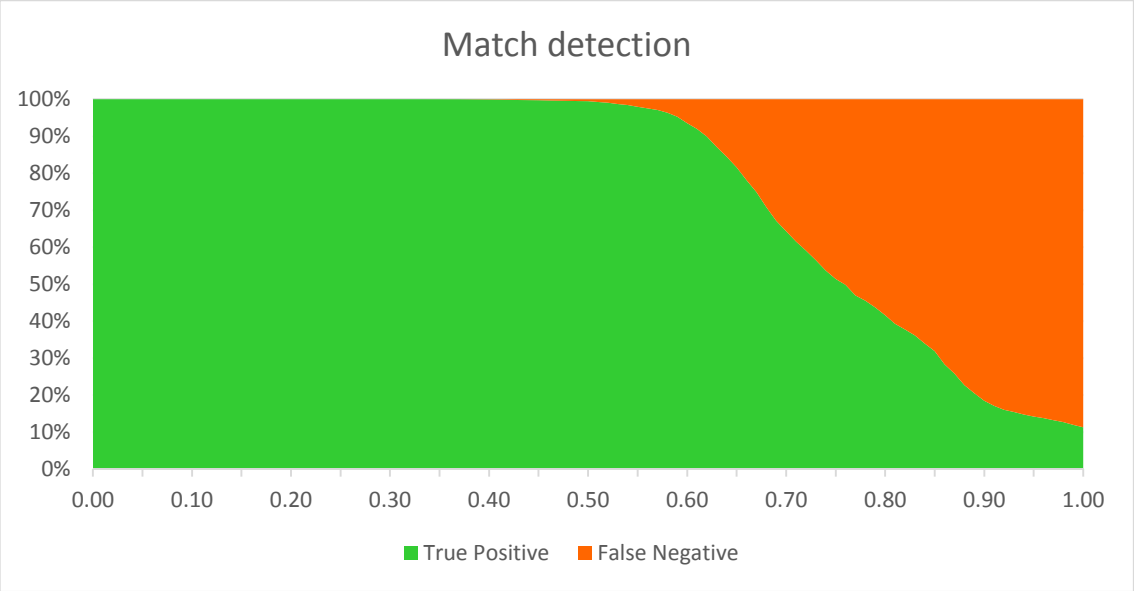
$$Jaro(s,t) = \frac{1}{3} \cdot \left( \frac{|s'|}{|s|} + \frac{|t'|}{|t|} + \frac{|s'| - T_{s',t'}}{2|s'|} \right)$$

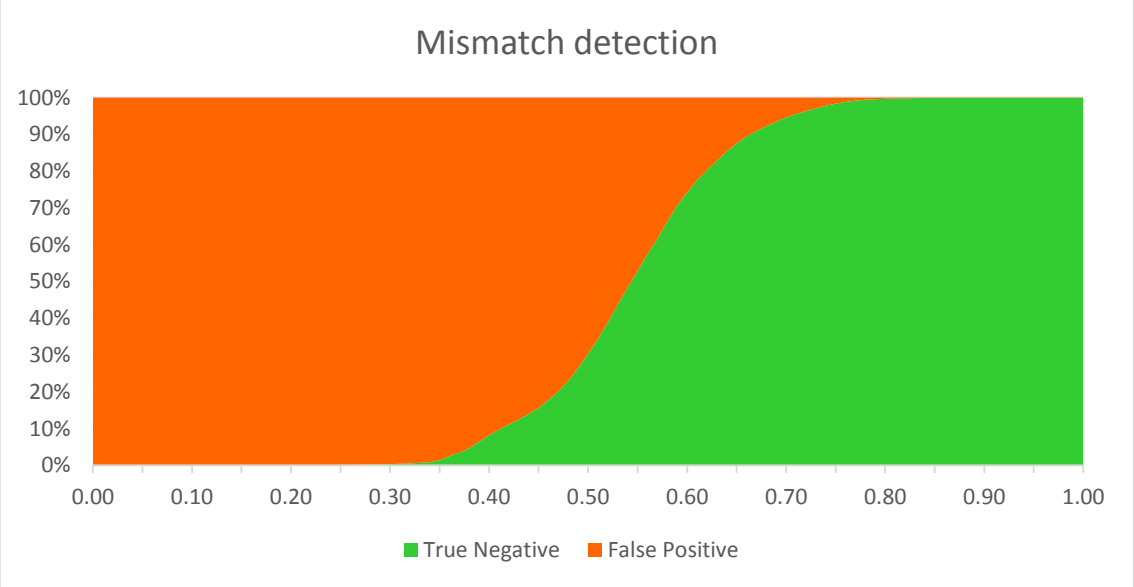
where *s'* is the characters in *s* that are in common with *t*, and *t'* is the characters in *t* that are in common with *s*. $T_{s',t'}$ is the number of transpositions of characters in *s'* relative to *t'*.

## 4.2 Normalization

The metric outputs values between 0 and 1, where 1 is a good match, thus no additional normalization is required.

## 4.3 Results




Using the Jaro distance it is possible to detect 89.89% of the matches at a sensitivity of 0.62. At this sensitivity it detects 80.26% of the mismatches, which is a mediocre result.

For good mismatch detection the sensitivity has to be at least 0.71, at which the algorithm detects 95.56% of the mismatches. At this sensitivity it only detects 61.52% of the matches. In this area it performs worse than the Levenshtein distance.

With 99.58% mismatch detection at a sensitivity of 0.79 it detects 43.70% of the matches, which is slightly worse than the result of the Needleman-Wunch distance in this area.

# 5 Jaro-Winkler Distance

## 5.1 Definition

The Jaro-Winkler distance is an extension of the Jaro distance. It modifies the weights of poorly matching pairs that share a common prefix. The output score is calculated as follows:

$$JaroWinkler(s,t) = Jaro(s,t) + prefixLength \cdot p \cdot (1 - Jaro(s,t))$$

where *Jaro(s,t)* is the Jaro distance of the two strings, *prefixLength* is the length of the common prefix and *p* is weight.

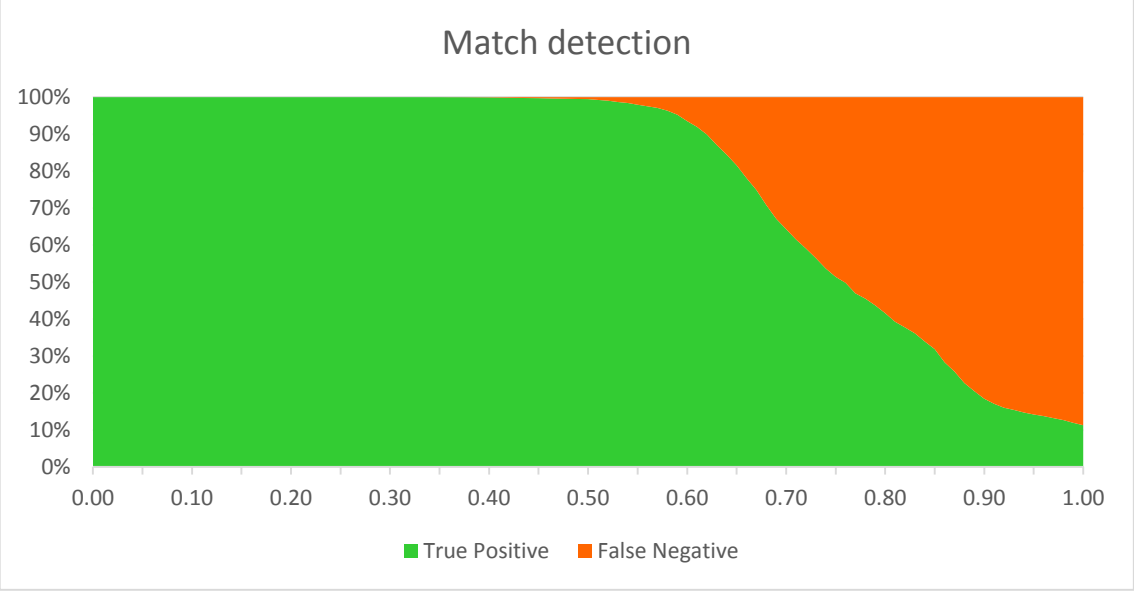
A prefix boost is generally only added if the Jaro distance of the two string is above a certain threshold.

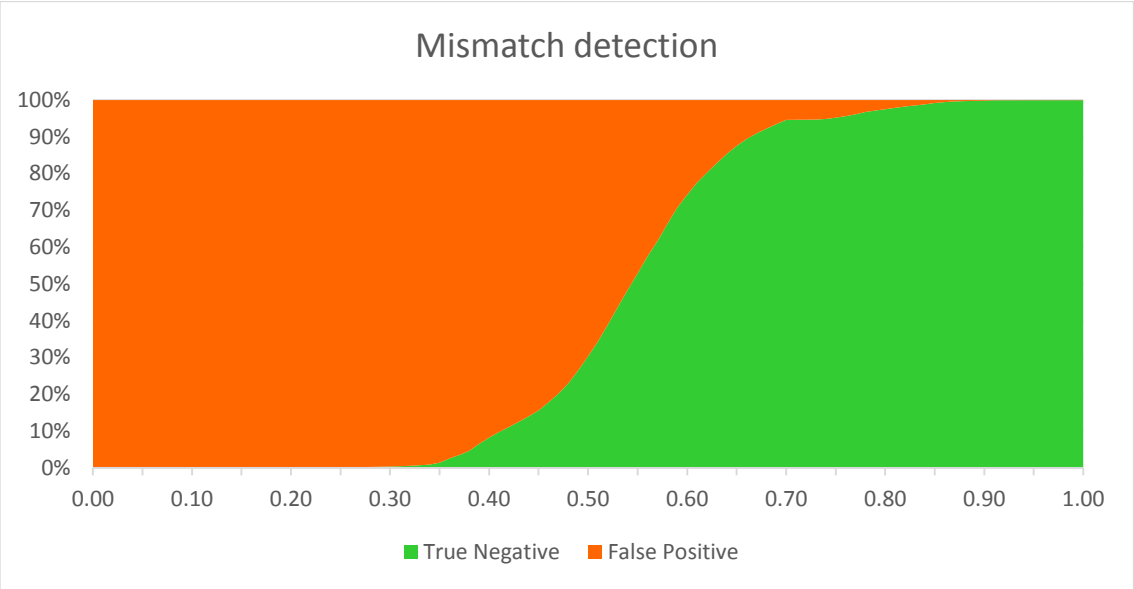
## 5.2 Normalization

Just like the output of the Jaro distance, the output of the Jaro-Winkler distance does not need additional normalization.

## 5.3 Results

Using a weight of 0.1 and a minimum prefix bonus threshold of 0.7 the following results were obtained:






Using the Jaro-Winkler distance it is possible to detect 89.89% of the matches at a sensitivity of 0.62. At this sensitivity it detects 80.26% of the mismatches, which is a mediocre result.

For good mismatch detection the sensitivity has to be at least 0.75, at which the algorithm detects 95.21% of the mismatches. At this sensitivity it only detects 56.5% of the matches. This result is worse than the result of the Jaro distance. This is because the boost reduced the algorithm's ability to detect mismatches above the threshold of 0.7.

With 99.63% mismatch detection at a sensitivity of 0.87 it detects 39.14% of the matches, which is worse than the result of the Jaro distance in this area.

# 6 Longest Common Substring and Subsequence

## 6.1 Definition

The longest common substring and the longest common subsequence of two strings can be used to calculate various metrics that can be used to tell how similar two strings are.

In the longest common substring, the characters are required to be strictly next to each other. This is not required for the longest common subsequence. For example for the string "abc" and "aabbcc" the longest common substring has a length of two ("ab" and "bc" are both longest common substring). The longest common subsequence for the two strings is "abc" and has a length of three.

## 6.2 Metrics

### 6.2.1 Square coefficient

The square coefficient can be calculated as:

$$sq(a,b) = \frac{|lcs(a,b)|^2}{|a| + |b|}$$

where the *lcs* function is either the longest common substring or the longest common subsequence.

### 6.2.2 Overlap coefficient

The overlap coefficient can be calculated as:

$$ol(a,b) = \frac{|lcs(a,b)|}{\min(|a|,|b|)}$$

Where the *lcs* function is either the longest common substring or the longest common subsequence.

### 6.2.3 Dice coefficient

The Dice coefficient can be calculated as:

$$Dice(a,b) = \frac{2 \cdot |lcs(a,b)|}{|a| + |b|}$$

Where the *lcs* function is either the longest common substring or the longest common subsequence.

### 6.2.4 Jaccard coefficient

The Dice coefficient can be calculated as:

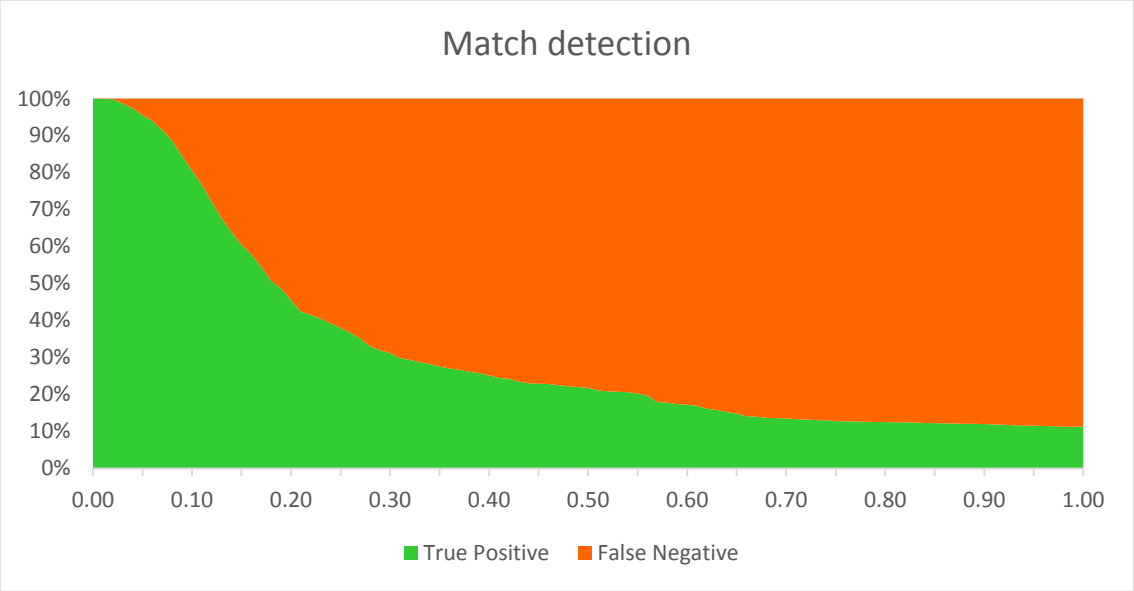
$$Jaccard(a,b) = \frac{|lcs(a,b)|}{|a| + |b| - |lcs(a,b)|}$$

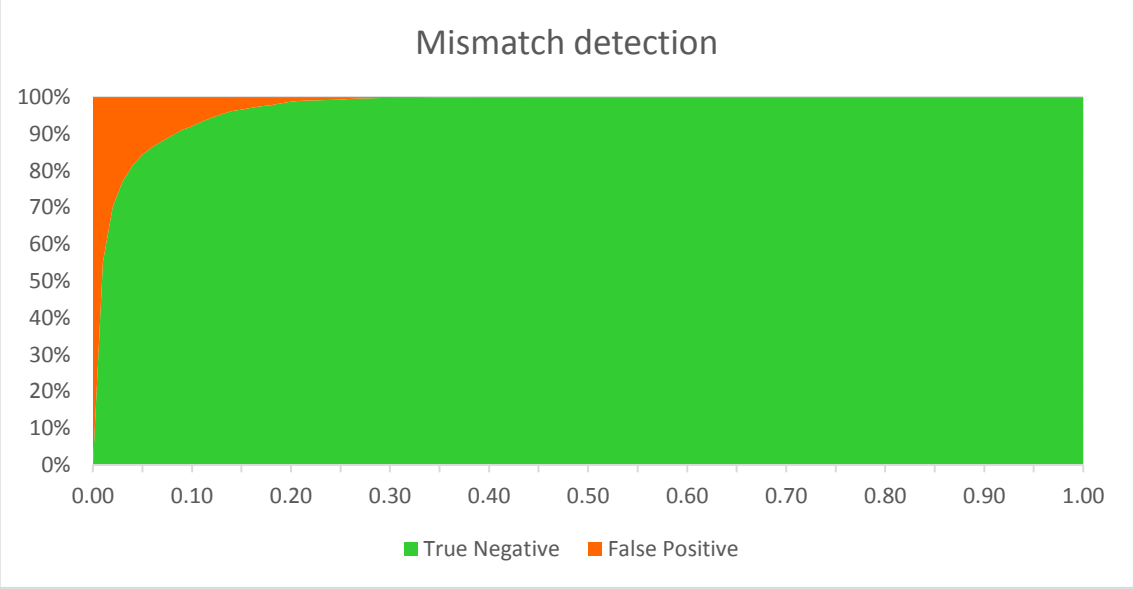
where the *lcs* function is either the longest common substring or the longest common subsequence.

## 6.3 Results

### 6.3.1 Square coefficient

#### 6.3.1.1 Longest Common Substring



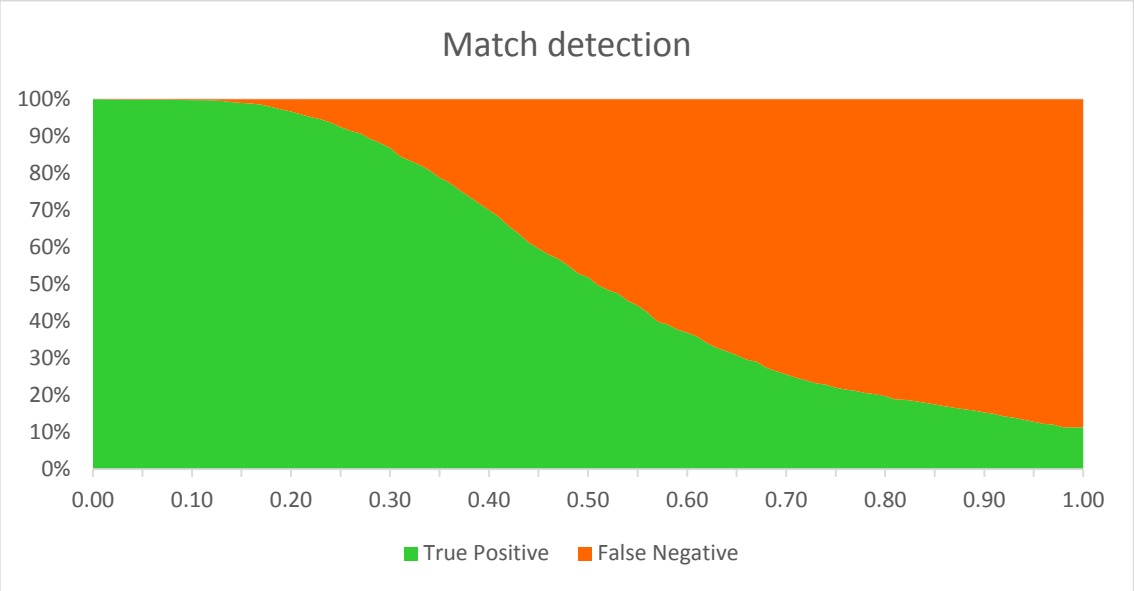

Using the square coefficient with longest common substring it is possible to detect 91.47% of the matches at a sensitivity of 0.07. At this sensitivity it detects 88.06% of the mismatches, resulting in a fair amount of false positive matches.

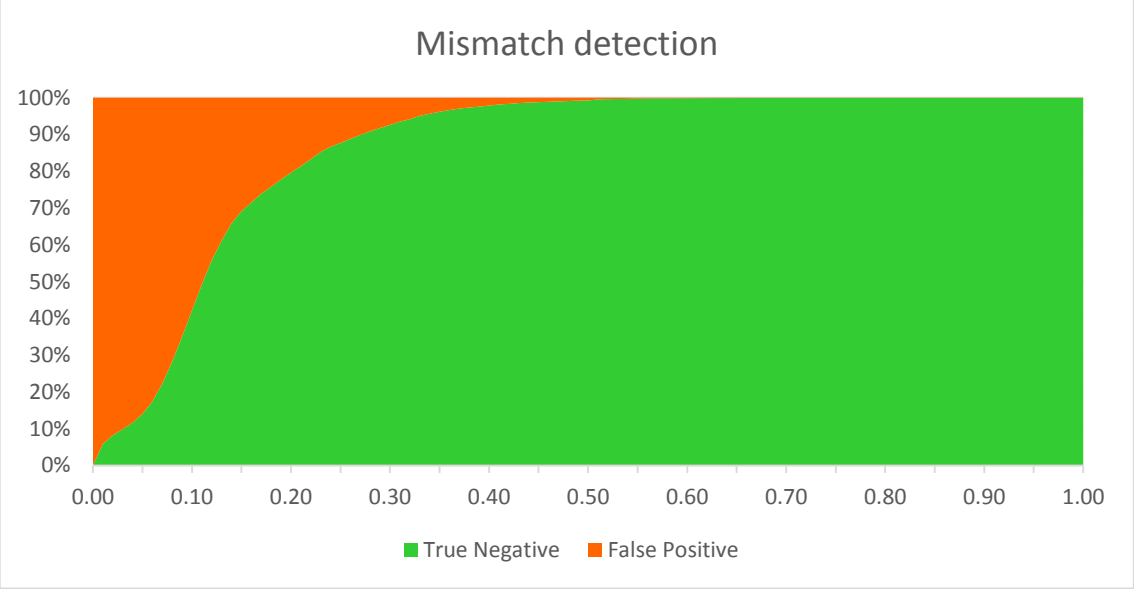
For good mismatch detection the sensitivity has to be at least 0.13, at which the algorithm detects 95.33% of the mismatches. However, at this sensitivity it only detects

67.59% of the matches, which means that approximately one in three matches are not detected.

With 99.52% mismatch detection at a sensitivity of 0.26 it detects 36.70% of the matches, which is slightly worse than the result of the Levenshtein distance in this area.

#### 6.3.1.2 Longest Common Subsequence



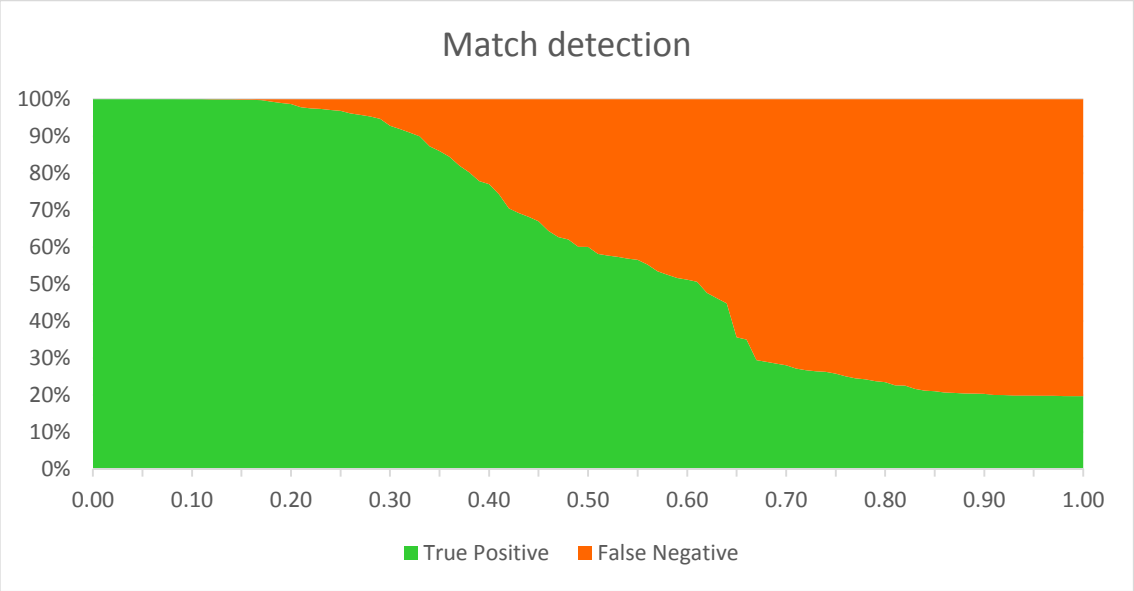

Using the square coefficient with longest common subsequence it is possible to detect 90.76% of the matches at a sensitivity of 0.27. At this sensitivity it detects 89.89% of the mismatches, which on par with the results of the longest common substring version.

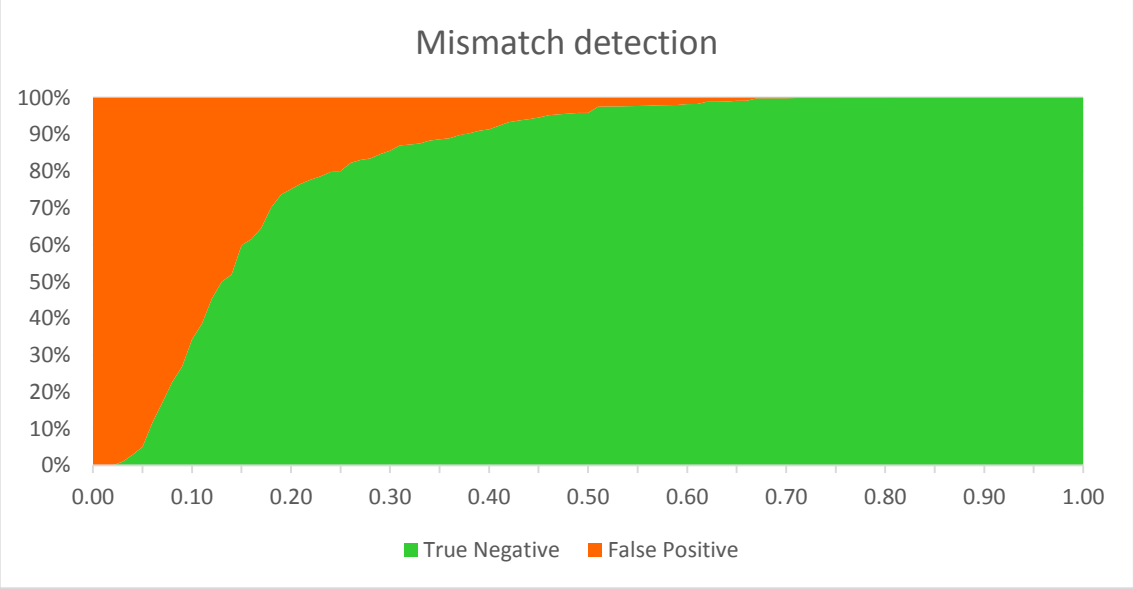
For good mismatch detection the sensitivity has to be at least 0.33, at which the algorithm detects 95.09% of the mismatches. At this sensitivity it only detects 82.33% of the matches, which is much better than the when we used longest common substring.

With 99.55% mismatch detection at a sensitivity of 0.51 it detects 49.73% of the matches, which is better than the Needleman-Wunch distance, but worse than the Smith-Waterman distance.

### 6.3.2 Overlap coefficient

#### 6.3.2.1 Longest Common Substring



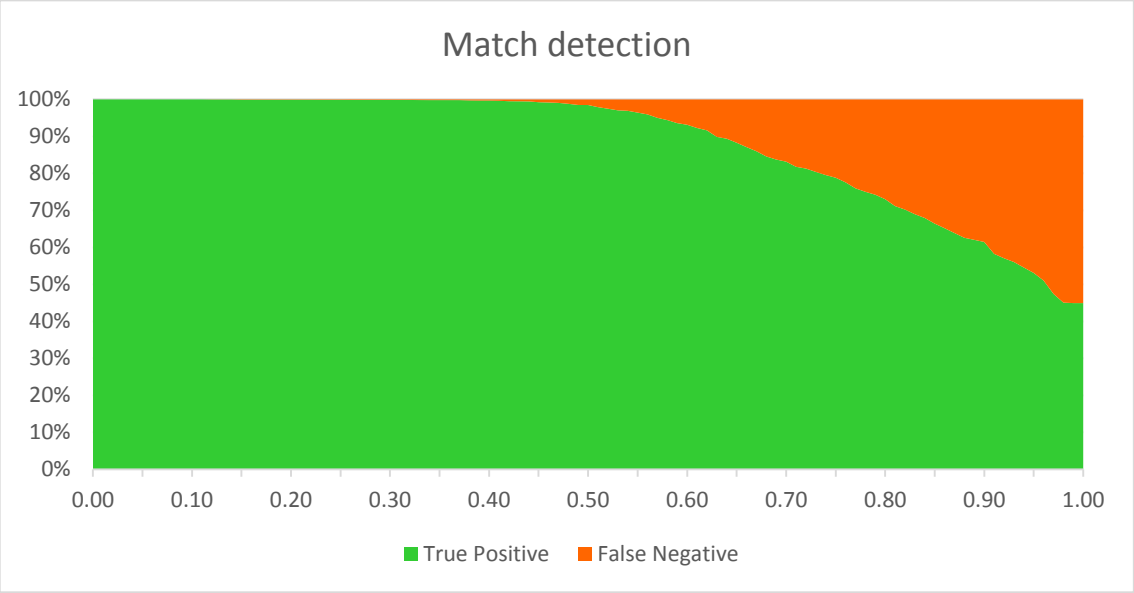



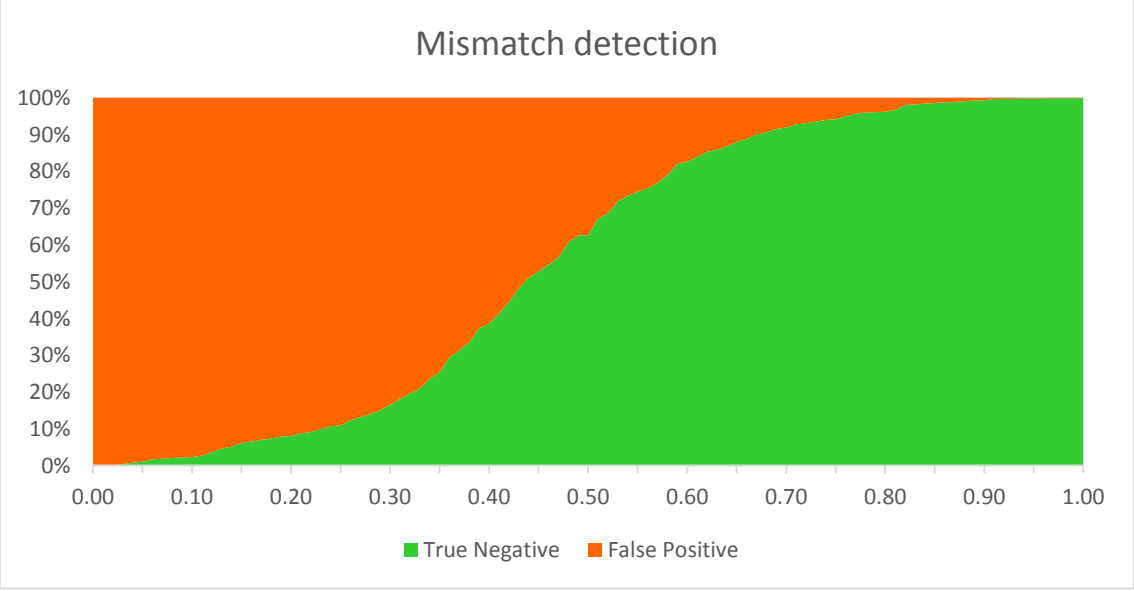
Using the overlap coefficient with longest common substring it is possible to detect 89.95% of the matches at a sensitivity of 0.33. At this sensitivity it detects 87.55% of the mismatches, which is slightly worse than the square coefficient version.

For good mismatch detection the sensitivity has to be at least 0.46, at which the algorithm detects 95.23% of the mismatches. However, at this sensitivity it only detects 64.39% of the matches. This is 3.2% worse than the result for the square coefficient.

With 99.81% mismatch detection at a sensitivity of 0.67 it detects 29.45% of the matches, which is the worst result so far.

#### 6.3.2.2 Longest Common Subsequence



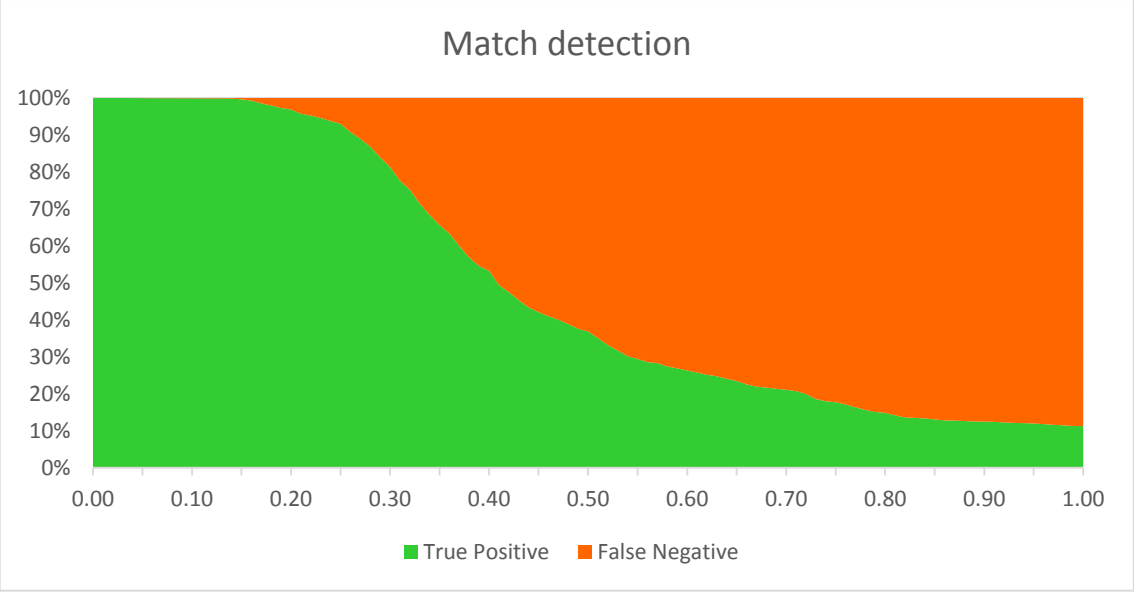
Using the overlap coefficient with longest common subsequence it is possible to detect 89.87% of the matches at a sensitivity of 0.63. At this sensitivity it detects 85.87% of the mismatches, resulting in more false positive matches than the longest common substring version.

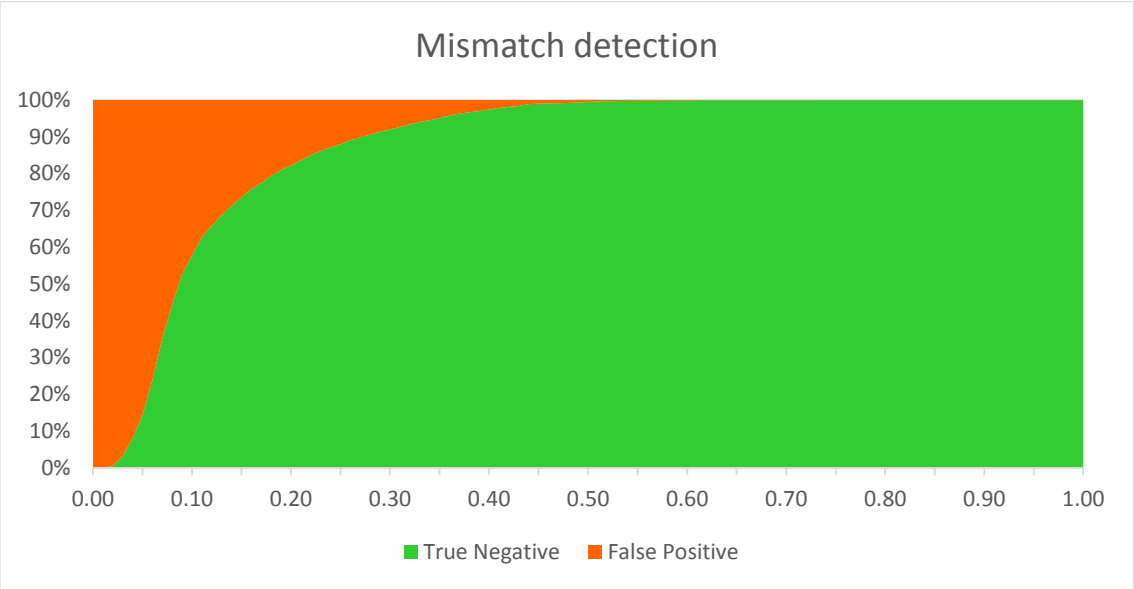
For good mismatch detection the sensitivity has to be at least 0.76, at which the algorithm detects 95% of the mismatches. At this sensitivity it only detects 77.58% of the matches, which is better than the when we used longest common substring, but worse than the square coefficient with longest common subsequence.

With 99.70% mismatch detection at a sensitivity of 0.91 it detects 58.25% of the matches, which is better than the overlap coefficient when used with longest common subsequence, but worse than the Smith-Waterman distance.

### 6.3.3 Dice coefficient

#### 6.3.3.1 Longest Common Substring



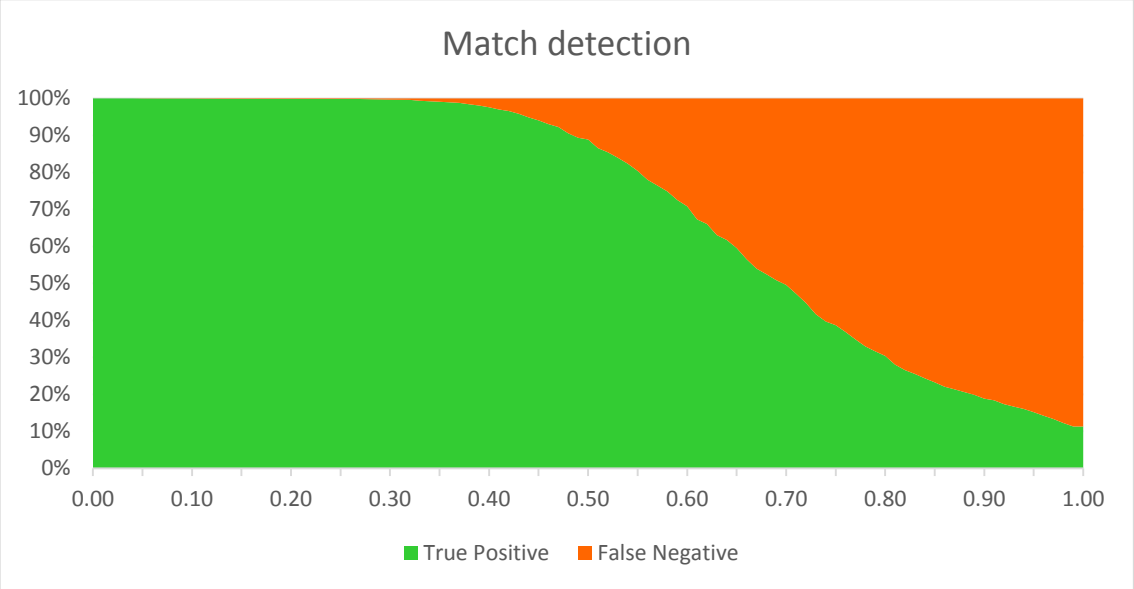


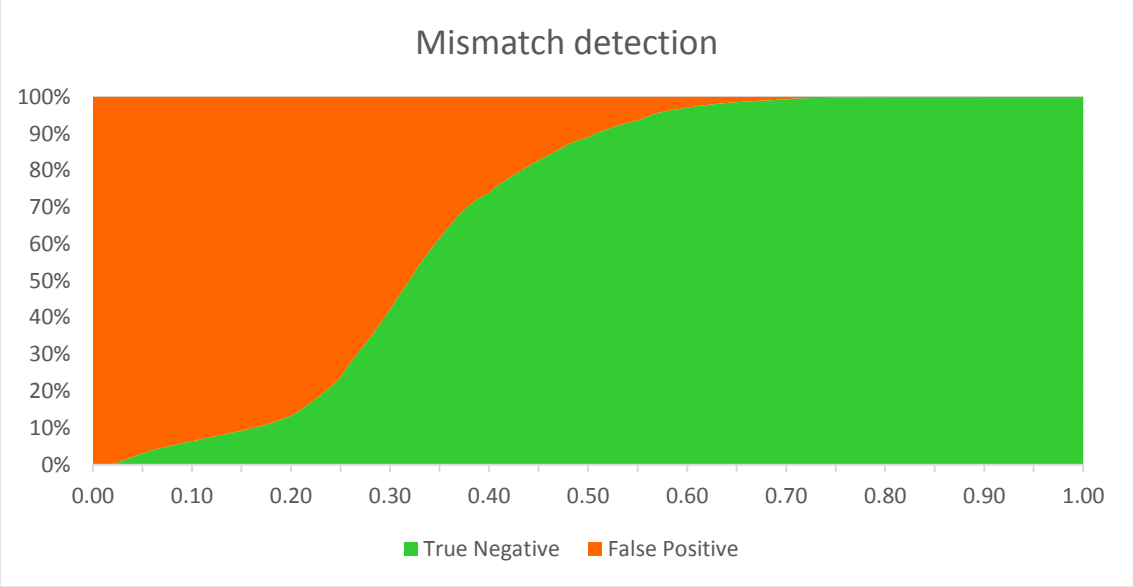

Using the Dice coefficient with longest common substring it is possible to detect 90.79% of the matches at a sensitivity of 0.26. At this sensitivity it detects 89.08% of the mismatches, which is on par with the square coefficient when longest common substring is used.

For good mismatch detection the sensitivity has to be at least 0.35, at which the algorithm detects 95.17% of the mismatches. However, at this sensitivity it only detects 65.64% of the matches. This is between the results of the square and the overlap coefficient when longest common substring was used.

With 99.66% mismatch detection at a sensitivity of 0.51 it detects 35.06% of the matches, which is better than the overlap coefficient, but worse than the square coefficient when used with longest common substring.

#### 6.3.3.2 Longest Common Subsequence



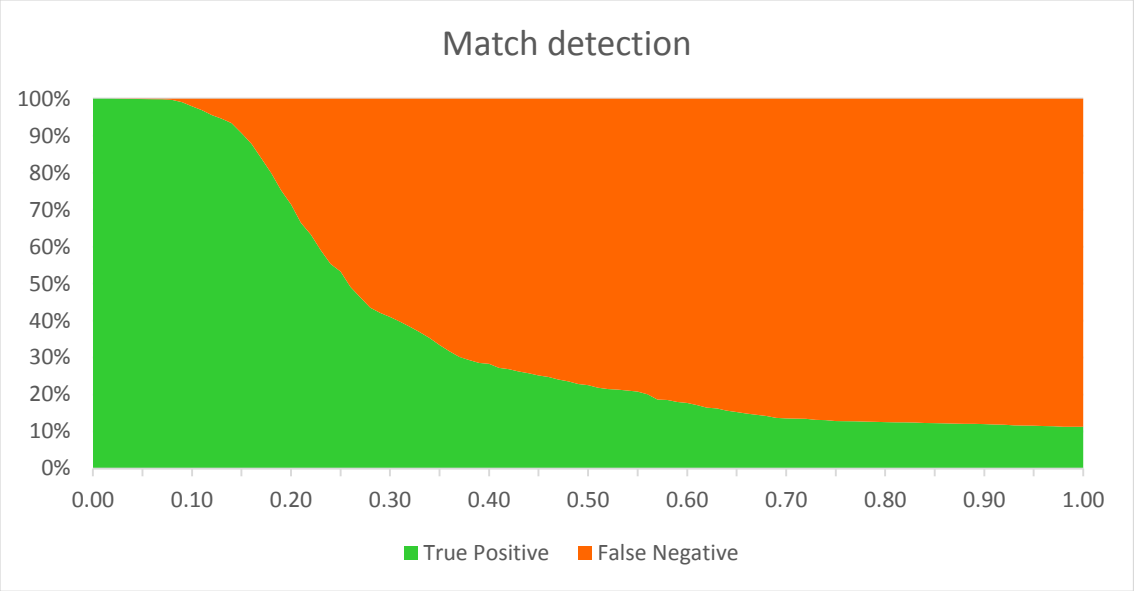

Using the Dice coefficient with longest common subsequence it is possible to detect 90.56% of the matches at a sensitivity of 0.48. At this sensitivity it detects 87.13% of the mismatches, resulting in slightly more false positive matches than the longest common substring version.

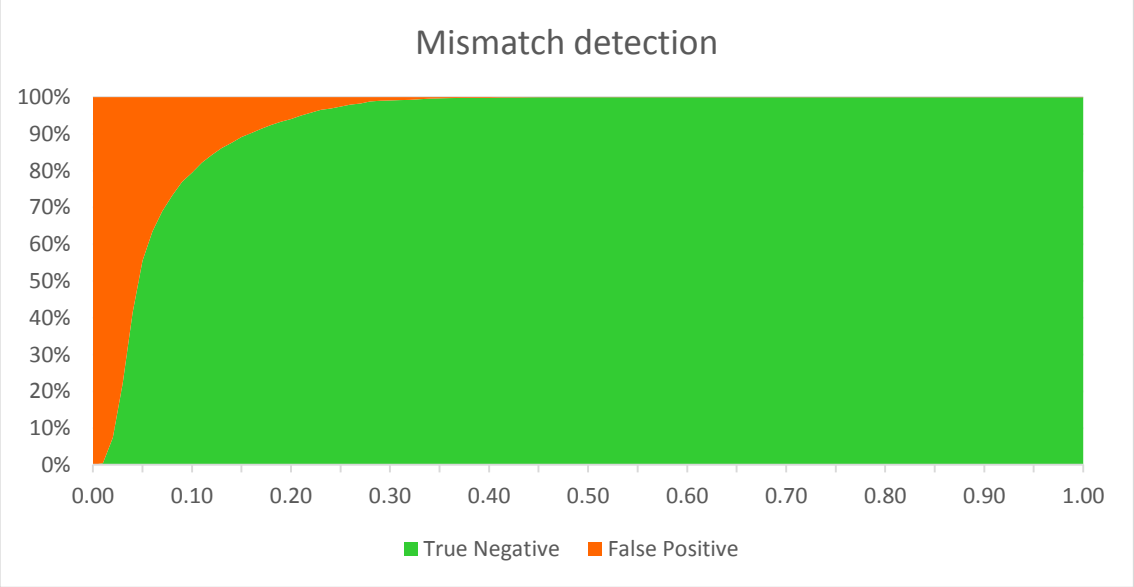
For good mismatch detection the sensitivity has to be at least 0.57, at which the algorithm detects 95.65% of the mismatches. At this sensitivity it only detects 76.46% of the matches, which is worse than the both the square coefficient and the overlap coefficient with longest common subsequence.

With 99.60% mismatch detection at a sensitivity of 0.71 it detects 47.21% of the matches, which is better than the Needleman-Wunch distance, but worse than the square coefficient when used with longest common subsequence.

### 6.3.4 Jaccard coefficient

#### 6.3.4.1 Longest Common Substring



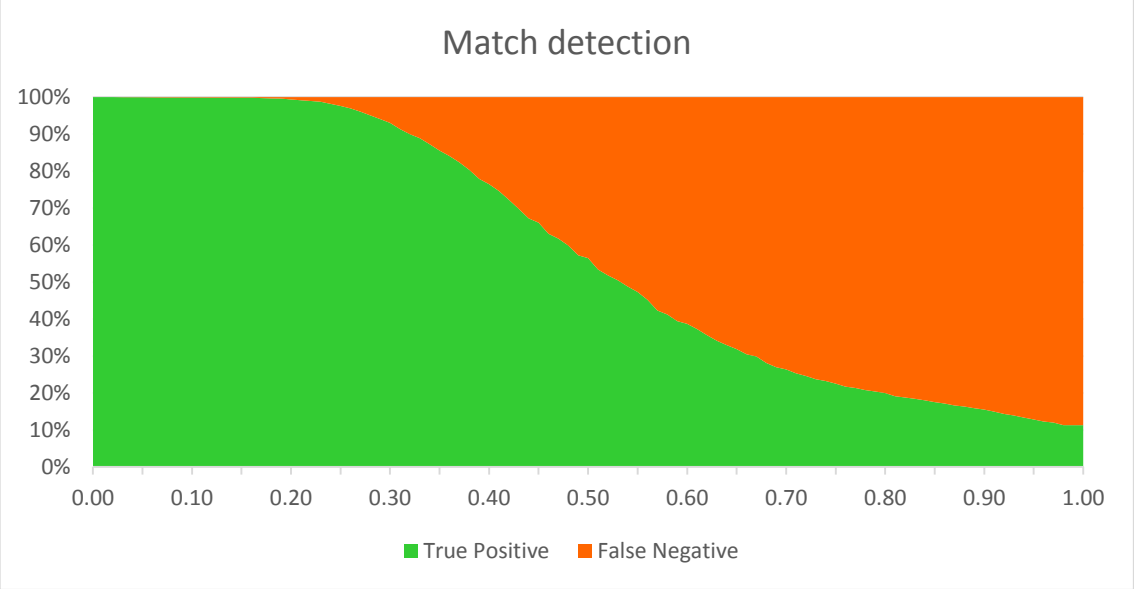



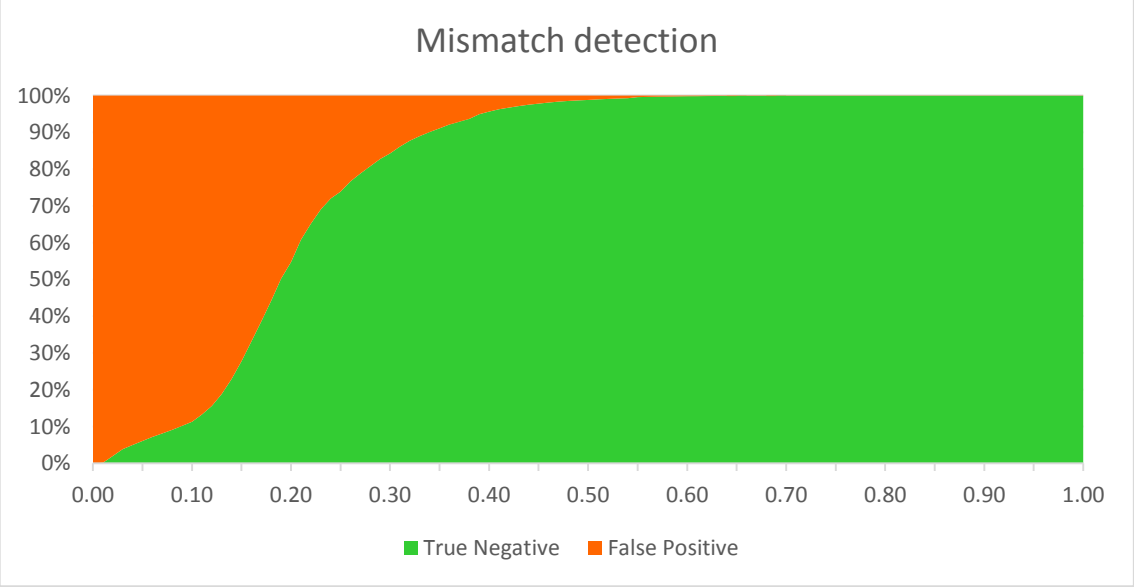

Using the Jaccard coefficient with longest common substring it is possible to detect 90.69% of the matches at a sensitivity of 0.15. At this sensitivity it detects 89.17% of the mismatches, which is on par with the Dice coefficient when longest common substring is used.

For good mismatch detection the sensitivity has to be at least 0.21, at which the algorithm detects 95 % of the mismatches. However, at this sensitivity it only detects 66.42% of the matches. This is worse than the result of square coefficient when longest common substring was used.

With 99.59% mismatch detection at a sensitivity of 0.34 it detects 35.29% of the matches, which is slightly better than the Dice coefficient, but worse than the square coefficient when used with longest common substring.

#### 6.3.4.2 Longest Common Subsequence



Using the Jaccard coefficient with longest common subsequence it is possible to detect 89.98% of the matches at a sensitivity of 0.32. At this sensitivity it detects 87.73% of the mismatches, resulting in slightly more false positive matches than the longest common substring version.

For good mismatch detection the sensitivity has to be at least 0.4, at which the algorithm detects 95.65% of the mismatches. At this sensitivity it only detects 76.46% of the matches. Both of these numbers happen to be equal than the results of the Dice coefficient when used with longest common subsequence.

With 99.59% mismatch detection at a sensitivity of 0.55 it detects 47.31% of the matches, which is slightly better than the Dice coefficient, but worse than the square coefficient when used with longest common subsequence.

# 7 Summary

I analyzed how well edit-distance based approximate string matching algorithms performed, highlighting three particular areas.

The first one was: How many false positives (mismatches) there are when 90% of the similar songs are found?

This is how the metrics I have tried performed in this area, ordered by their accuracy of mismatch detection:

| **Algorithm** | **Sensitivity** | **Match** | **Mismatch** |
|---|---|---|---|
| Smith-Waterman distance | 0.45 | 90.38% | **93.37%** |
| Longest Common Subsequence Square coefficient | 0.27 | 90.76% | **89.89%** |
| Longest Common Substring Jaccard coefficient | 0.15 | 90.69% | **89.17%** |
| Longest Common Substring Dice coefficient | 0.26 | 90.79% | **89.08%** |
| Longest Common Substring Square coefficient | 0.07 | 91.47% | **88.06%** |
| Longest Common Subsequence Jaccard coefficient | 0.32 | 89.98% | **87.73%** |
| Longest Common Substring Overlap coefficient | 0.33 | 89.95% | **87.55%** |
| Longest Common Subsequence Dice coefficient | 0.48 | 90.56% | **87.13%** |
| Longest Common Subsequence Overlap coefficient | 0.63 | 89.87% | **85.87%** |
| Levenshtein distance | 0.30 | 90.75% | **80.93%** |
| Jaro distance | 0.62 | 89.89% | **80.26%** |
| Jaro-Winkler distance | 0.62 | 89.89% | **80.26%** |
| Needleman-Wunch distance | 0.39 | 89.96% | **78.80%** |

The Smith-Waterman distance performed best. It recognized 93.37% of different songs as different. It is followed by the metrics that are based on longest common subsequence and substring. The Needleman-Wunch distance was the worse, which only recognized 78.80% of the different songs correctly.

The second area focuses on how many similar songs are found when only 5% false positives are allowed. In real-world usage this meant that there were at least two false positive matches for every positive match detected by the algorithms.

This is how the metrics performed in this area, ordered by their accuracy of match detection:

| **Algorithm** | **Sensitivity** | **Mismatch** | **Match** |
|---|---|---|---|
| Smith-Waterman distance | 0.51 | 96.04% | **86.53%** |
| Longest Common Subsequence Square coefficient | 0.33 | 95.09% | **82.22%** |
| Longest Common Subsequence Overlap coefficient | 0.76 | 95.00% | **77.58%** |
| Needleman-Wunch distance | 0.47 | 95.18% | **76.80%** |
| Longest Common Subsequence Dice coefficient | 0.57 | 95.65% | **76.46%** |
| Longest Common Subsequence Jaccard coefficient | 0.40 | 95.65% | **76.46%** |
| Longest Common Substring Square coefficient | 0.13 | 95.33% | **67.59%** |
| Levenshtein distance | 0.44 | 95.19% | **67.56%** |
| Longest Common Substring Jaccard coefficient | 0.21 | 95.00% | **66.42%** |
| Longest Common Substring Dice coefficient | 0.35 | 95.17% | **65.64%** |
| Longest Common Substring Overlap coefficient | 0.46 | 95.23% | **64.39%** |
| Jaro distance | 0.71 | 95.56% | **61.52%** |
| Jaro-Winkler distance | 0.75 | 95.21% | **56.60%** |

The Smith-Waterman distance did best in this area too by finding 86.53% of the similar songs. Just like in the previous area, it is followed by longest common subsequence with the square coefficient. The Needleman-Wunch distance did much better in this area, being the 4$^{th}$ best. At the bottom is the Jaro distance followed by the Jaro-Winkler distance. These two metrics performed among the worst in both areas.

In the last area only very few false positives are allowed, less than 0.5%. In real-world usage this meant that there was at least one false positive match for every ten to twenty positive match detected by the algorithms.

This is how the metrics performed in the last area, ordered by their accuracy in mismatch detection:

| Algorithm | Sensitivity | Mismatch | Match |
|---|---|---|---|
| Smith-Waterman distance | 0.67 | 99.56% | **66.00%** |
| Longest Common Subsequence Overlap coefficient | 0.91 | 99.70% | **58.25%** |
| Longest Common Subsequence Square coefficient | 0.51 | 99.55% | **49.73%** |
| Longest Common Subsequence Jaccard coefficient | 0.55 | 99.59% | **47.31%** |
| Longest Common Subsequence Dice coefficient | 0.71 | 99.60% | **47.21%** |
| Needleman-Wunch distance | 0.62 | 99.60% | **44.69%** |
| Jaro distance | 0.79 | 99.58% | **43.70%** |
| Jaro-Winkler distance | 0.87 | 99.63% | **39.14%** |
| Levenshtein dinstance | 0.62 | 99.72% | **38.46%** |
| Longest Common Substring Square coefficient | 0.26 | 99.52% | **36.70%** |
| Longest Common Substring Jaccard coefficient | 0.34 | 99.59% | **35.29%** |
| Longest Common Substring Dice coefficient | 0.51 | 99.66% | **35.06%** |
| Longest Common Substring Overlap coefficient | 0.67 | 99.81% | **29.45%** |

Smith-Waterman distance is a clear winner here too. The overlap coefficient took over the square coefficient with longest common subsequence. The Needleman-Wunch distance did not perform better at this accuracy than the longest common subsequence based algorithms. The longest common subsequence based metrics became the worst at this level of accuracy.

Based on these results, I will investigate how the performance of the Smith-Waterman distance can be improved, along with the overlap coefficient metric used with longest common subsequence in Chapter 6.

# V Token Based Algorithms

Token based approximate string matching algorithms first tokenize strings, resulting in bags of tokens. One such tokenization is to take contiguous sequences of n characters in length. These are called n-grams. Once the strings are tokenized, a metric is used to quantify the similarity of the two bags.

In this chapter I analyze how different token based approximate string matching algorithms perform using only a single, basic preprocessing that converts all song artist and title fields to lower case to ignore case differences. Strings will be tokenized into trigrams (n=3).

The pipeline is constructed in the following way:

```
public class NGramPipeline : Pipeline<Song>
{
    public NGramPipeline(int length = 3)
    {
        this.Preprocessings.Add(new IgnoreCasePreprocessing());
        this.Preprocessings.Add(new NGramTokenizer(length));

        this.Metrics.Add(new BlockDistanceMetric());
        this.Metrics.Add(new CosineSimilarityMetric());
        this.Metrics.Add(new DiceSimilarityMetric());
        this.Metrics.Add(new EuclideanDistanceMetric());
        this.Metrics.Add(new JaccardSimilarityMetric());
        this.Metrics.Add(new MatchingCoefficientMetric());
        this.Metrics.Add(new OverlapCoefficientMetric());

        this.Verifier = new ClusterVerifier();

        this.Summarizers.Add(new CsvSummarizer());
    }
}
```

Heuristics are not used because the three sample sets are small and does not take long time to run the pipeline.

The pipeline can be ran the following way:

```
var popularSongs = DataSource.GetSongs("Songs.Popular");
var similarSongs = DataSource.GetSongs("Songs.Similar");
var foreignSongs = DataSource.GetSongs("Songs.Foreign");

var results = new NGramPipeline().Run(System.Console.WriteLine, popularSongs,
similarSongs, foreignSongs);
```

Just like in the previous chapter, the summarizer will generate strings which contain how well the metrics perform at different sensitivity values in a CSV file format. These results are analyzed, highlighting the same three areas used in the previous chapter: First, how many false positives (mismatches) there are when 90% of the similar songs are found? Second, how many similar songs are found of only 5% false positives are allowed? Finally, how many similar songs are found of only 0.5% false positives are allowed?

# 1 Block Distance

## 1.1 Definition

Block distance – also known as city block distance, Manhattan distance and $L_1$ distance – is a metric that measures the distance of two vectors by summing the edge distances. Formally:

$$x, y \in \mathbb{R}^n : L_1(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

The block distance can be utilized to calculate the distance between two tokenized strings by thinking of every possible token as a dimension. In case of trigrams in the English alphabet it's a $26^3$ dimensional space ("aaa", "aab", etc). This way every bag of tokens can be represented by a vector.

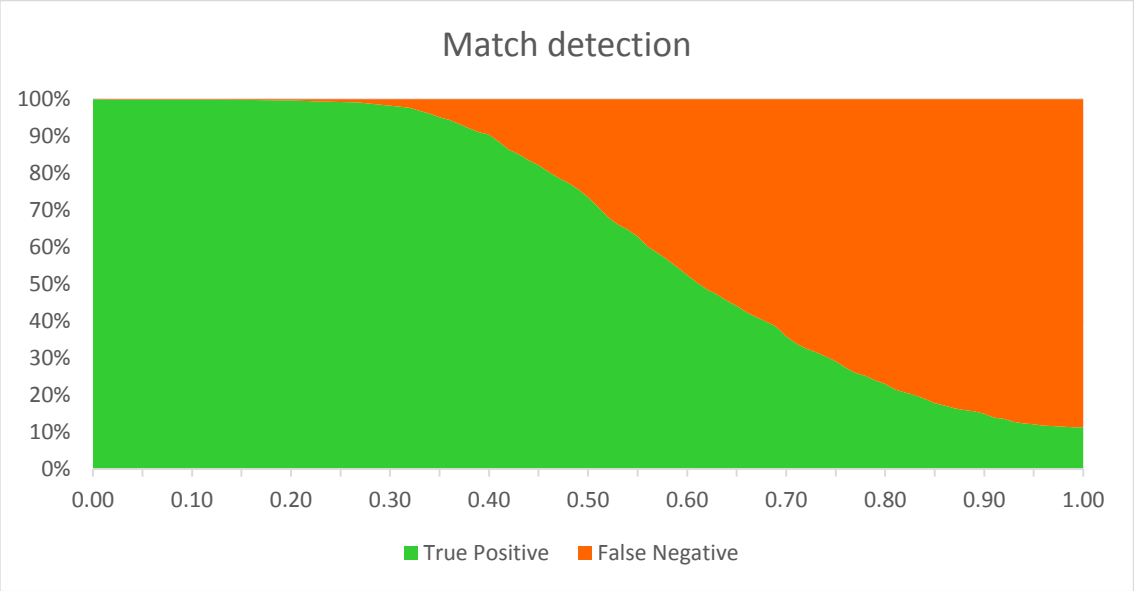
## 1.2 Normalization

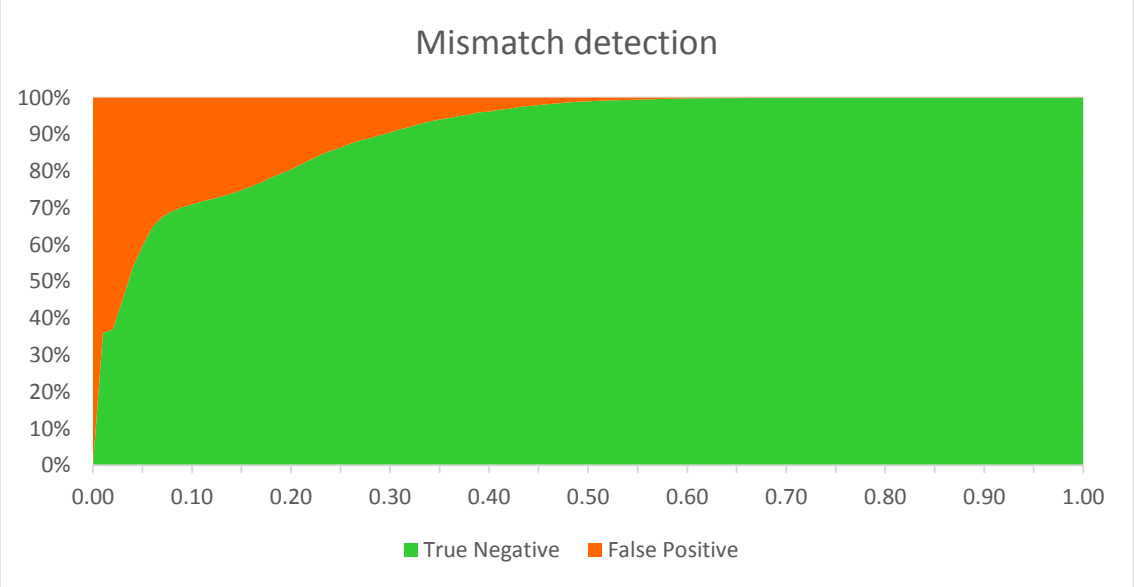
The minimum of the block distance is 0. The maximum is the sum of the number of tokens in both words. To normalize the output of the block distance, the following function is used:

$$\frac{|T_x| + |T_y| - L_1(x, y)}{|T_x| + |T_y|}$$

where $T_x$ denotes the bag of tokens of the string $x$ and $T_y$ denotes the bag of tokens of the string $y$.

## 1.3 Results




Using the block distance it is possible to detect 90.42% of the matches at a sensitivity of 0.40. At this sensitivity the mismatch detection is better than 95%: It detects 96.20% of the mismatches, which is better than any of the edit-distance based algorithms I have tried in the previous chapter.

When at least 95% mismatch detection is required, it detects 93.27% of the matches at a sensitivity of 0.37. With 99.53% mismatch detection at a sensitivity of 0.56 it detects 60.17% of the matches, which is better than all of the edit-based distance algorithms, except for the Smith-Waterman distance.

# 2 Euclidean Distance

## 2.1 Definition

Euclidean distance – also known as $L_2$ distance – is a metric similar to the block distance. It measures the distance of two vectors as follows:

$$x, y \in \mathbb{R}^n : L_2(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

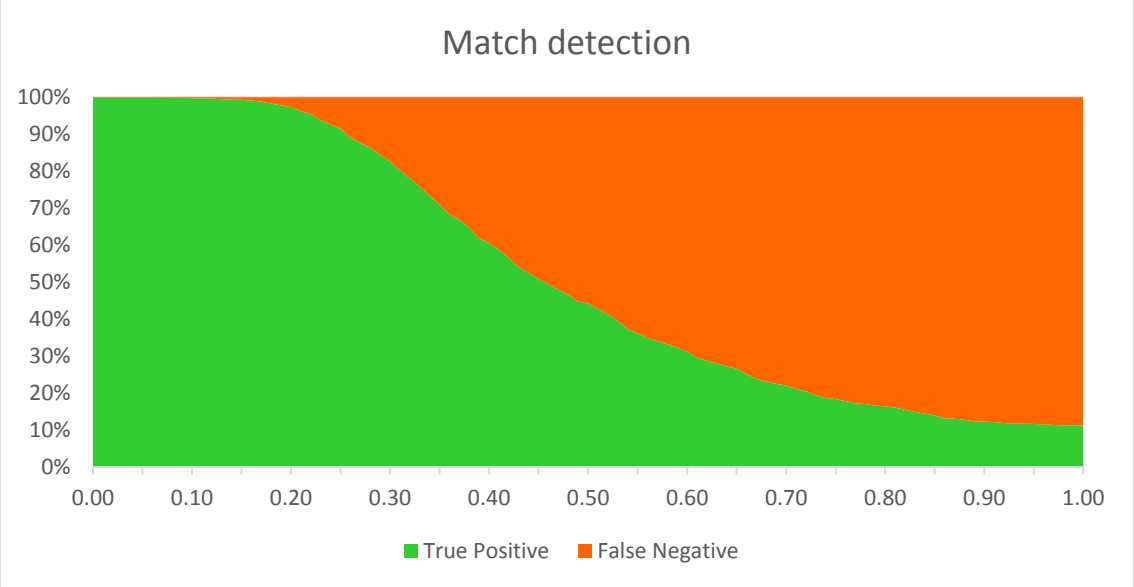
## 2.2 Normalization

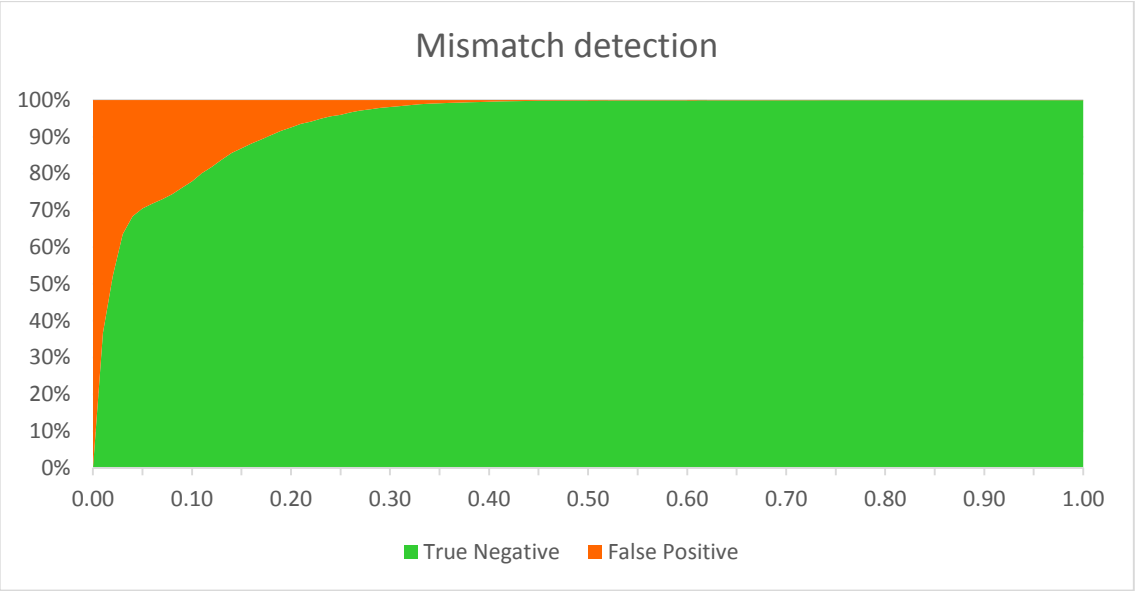
The minimum of the Euclidean distance is 0. The maximum is the square root of sum of the number of tokens in both words. To normalize the output, the following function is used:

$$\frac{\sqrt{|T_x| + |T_y|} - L_2(x, y)}{\sqrt{|T_x| + |T_y|}}$$

where $T_x$ denotes the bag of tokens of the string $x$ and $T_y$ denotes the bag of tokens of the string $y$.

## 2.3 Results






Using the Euclidean distance it is possible to detect 90.11% of the matches at a sensitivity of 0.20. At this sensitivity the mismatch detection is slightly better than 95%. It detects 95.07% of the mismatches. In this area it is worse than the block distance, but still better than any of the edit-distance based algorithms I have tried in the previous chapter.

With 99.51% mismatch detection at a sensitivity of 0.32 it detects 63.17% of the matches, which is better than the result of the block distance, but worse than the Smith-Waterman distance's result.

# 3 Cosine Similarity

## 3.1 Definition

Cosine similarity is also a vector based similarity measure – just like block distance. The idea is that if two vectors are similar if the angle between them is small. The tokens are represented as sets instead of bags. Cosine similarity is calculated as:

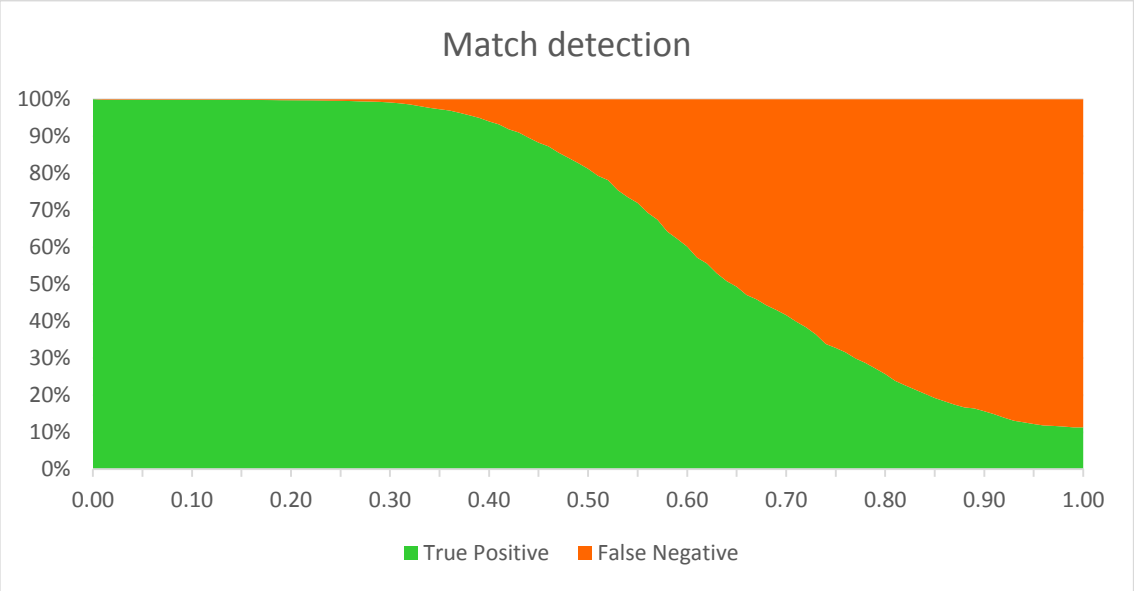
$$Cosine(x, y) = \frac{|T_x \cap T_y|}{\sqrt{|T_x| \cdot |T_y|}}$$

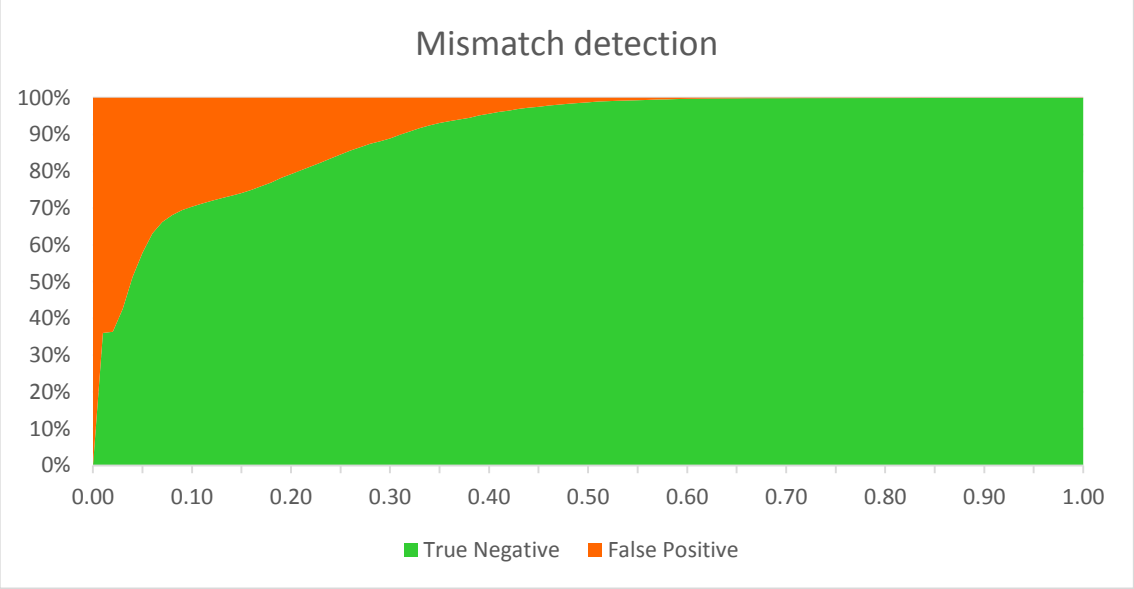
where $T_x$ denotes the set of tokens of the string $x$ and $T_y$ denotes the set of tokens of the string $y$.

## 3.2 Normalization

The metric outputs values between 0 and 1, where 1 is a good match, thus no additional normalization is necessary.

## 3.3 Results




Using the cosine similarity it is possible to detect 91.03% of the matches at a sensitivity of 0.43. At this sensitivity it detects 96.98% of the mismatches, which is better than any of the edit-distance based algorithms and the block distance.

When at least 95% mismatch detection is required, it detects 94.98% of the matches at a sensitivity of 0.39, performing better than the edit-distance based algorithms and the block distance. With 99.53% mismatch detection at a sensitivity of 0.57 it detects 67.46% of the matches, which is better than all of the edit-based distance algorithms, including the Smith-Waterman distance.

# 4 Dice Similarity

## 4.1 Definition

Dice similarity is similar to cosine similarity in the way that tokens are represented here as sets too. It is calculated as:

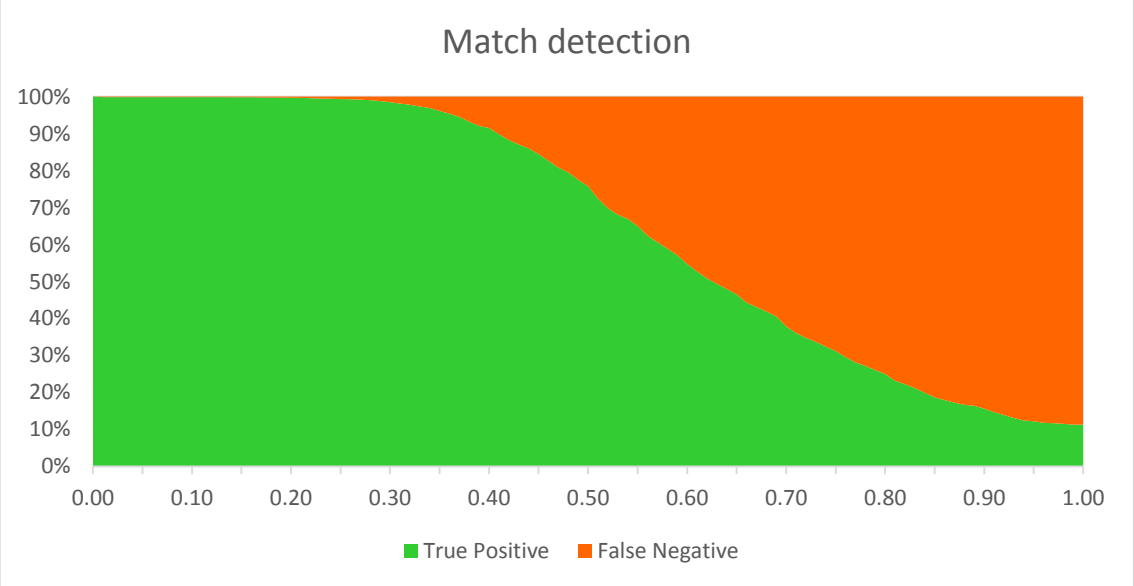
$$Dice(x, y) = \frac{2 \cdot |T_x \cap T_y|}{|T_x| + |T_y|}$$

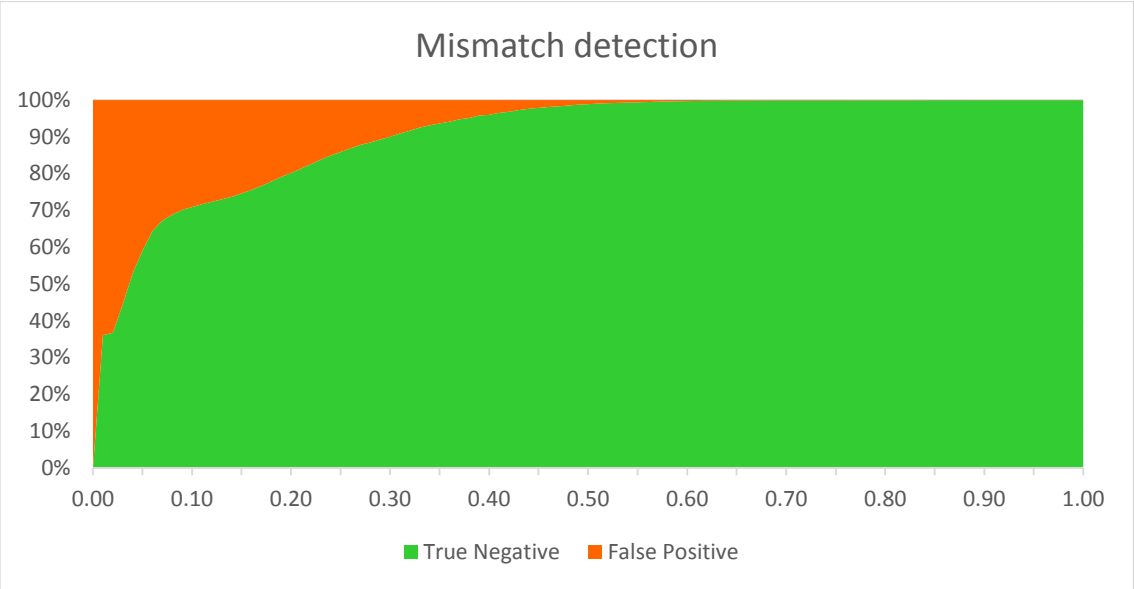
where $T_x$ denotes the set of tokens of the string $x$ and $T_y$ denotes the set of tokens of the string $y$.

## 4.2 Normalization

The metric outputs values between 0 and 1, where 1 is a good match, thus no additional normalization is necessary.

## 4.3 Results






Using the Dice similarity it is possible to detect 89.82% of the matches at a sensitivity of 0.41. At this sensitivity the mismatch detection 96.52%, a result worse than cosine similarity, but better than block distance.

When at least 95% mismatch detection is required, it detects 93.18% of the matches at a sensitivity of 0.38, which is very close to the result of the block distance. With 99.55% mismatch detection at a sensitivity of 0.57 it detects 60.63% of the matches, which is better than the result of the block distance, but worse than the cosine similarity's 67.46% detection.

# 5 Jaccard Similarity

## 5.1 Definition

Jaccard similarity is similar to Dice similarity. It is calculated as:

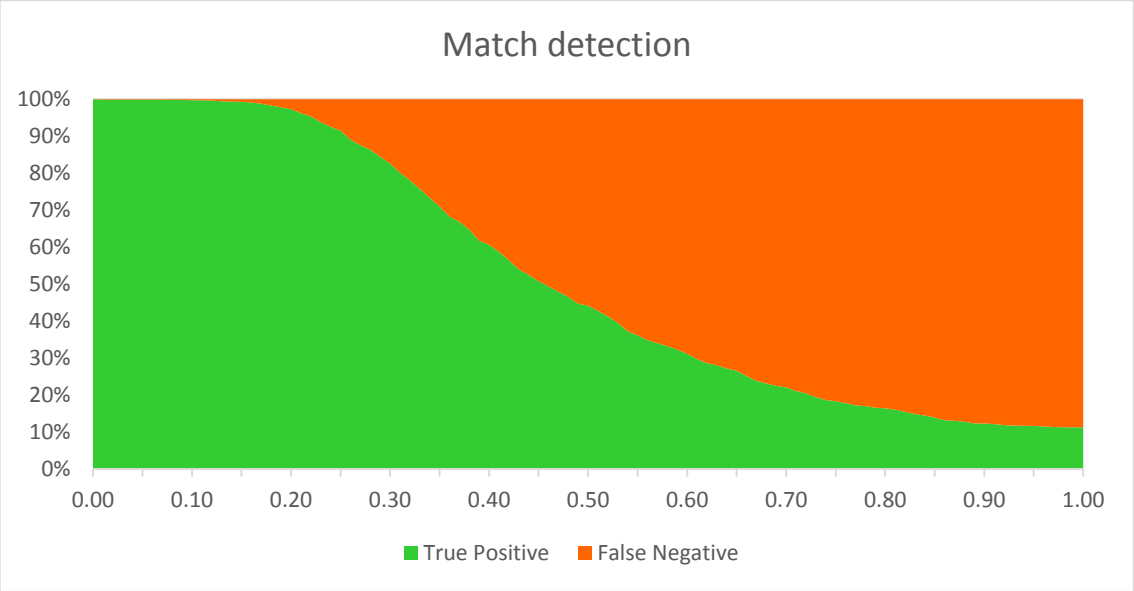
$$Jaccard(x, y) = \frac{|T_x \cap T_y|}{|T_x \cup T_y|}$$

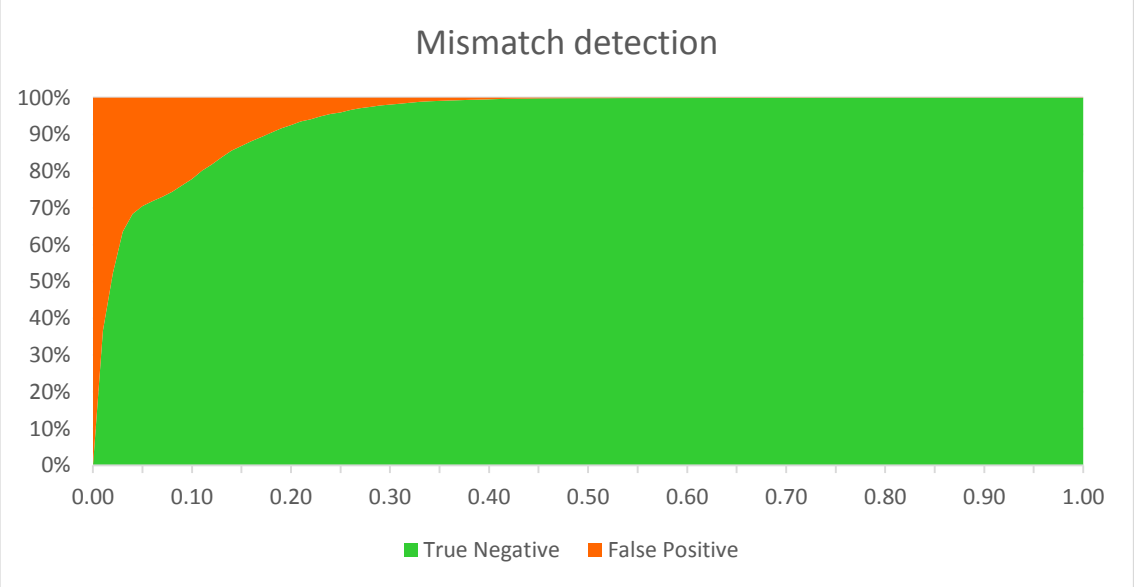
where $T_x$ denotes the set of tokens of the string $x$ and $T_y$ denotes the set of tokens of the string $y$.

## 5.2 Normalization

The metric outputs values between 0 and 1, where 1 is a good match, thus no additional normalization is required.

## 5.3 Results




Using the Jaccard similarity it is possible to detect 91.51% of the matches at a sensitivity of 0.25. At this sensitivity it detects 95.98% of the mismatches, which is worse than the block distance, but better than the Euclidean distance.

When at least 95% mismatch detection is required, it detects 92.52% of the matches at a sensitivity of 0.24. Again, this is worse than the block distance, but better than the Euclidean distance. With 99.55% mismatch detection at a sensitivity of 0.40 it detects 60.63% of the matches, which is on par with the Dice similarity.

# 6 Matching Coefficient

## 6.1 Definition

Matching coefficient is a simple metric. It can be viewed as the vector version of Hamming distance. It is calculated as:

$$Matching(x, y) = |T_x \cap T_y|$$

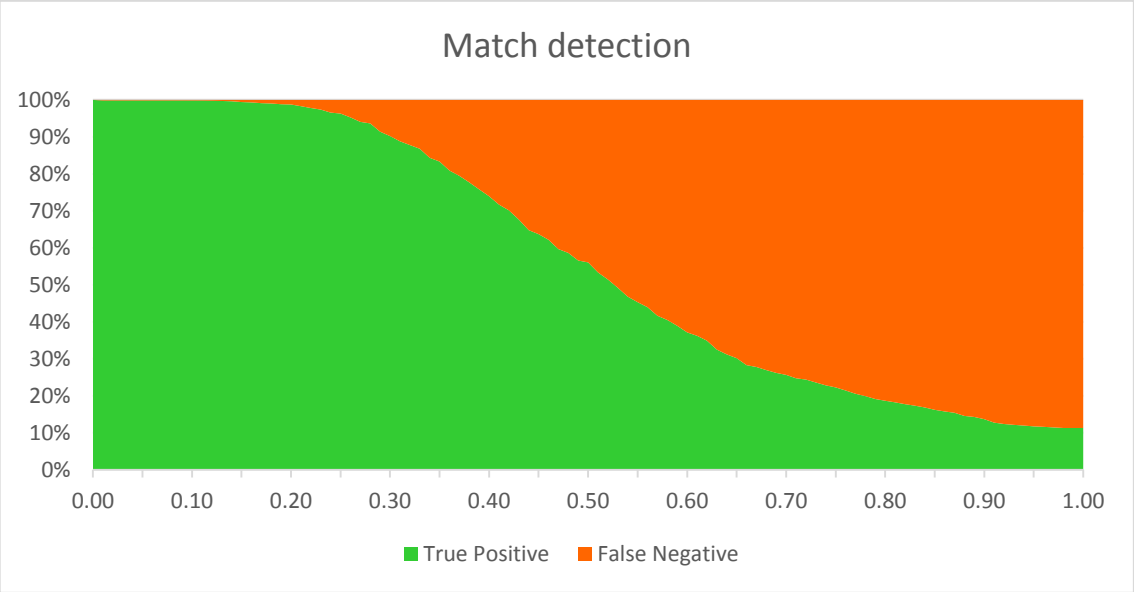
where $T_x$ denotes the set of tokens of the string $x$ and $T_y$ denotes the set of tokens of the string $y$.

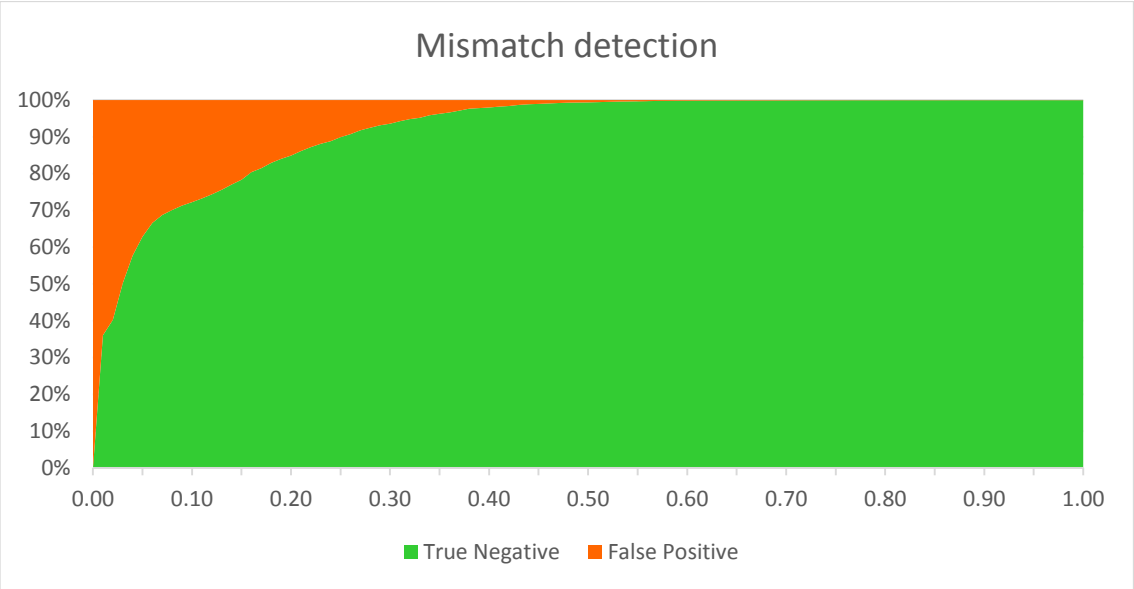
## 6.2 Normalization

The minimum of the matching coefficient is 0. The maximum is size of the larger token set. To normalize the output, the following function is used:

$$\frac{Matching(x, y)}{\max(|T_x|, |T_y|)}$$

## 6.3 Results






The matching coefficient is the worst performer out of all token based algorithms. It detects 90.21% of the matches at a sensitivity of 0.30, resulting in a mismatch detection of 93.58% using the sample set.

When 95% mismatch detection is needed, it only detects 86.79% of the matches at a sensitivity of 0.33. At a sensitivity of 0.52 it detects 99.52% of the mismatches, but only 51.45% of the matches, which is 8.72% worse than when block distance was used.

# 7 Overlap Coefficient

## 7.1 Definition

Overlap coefficient is similar to dice similarity. It is calculated as:

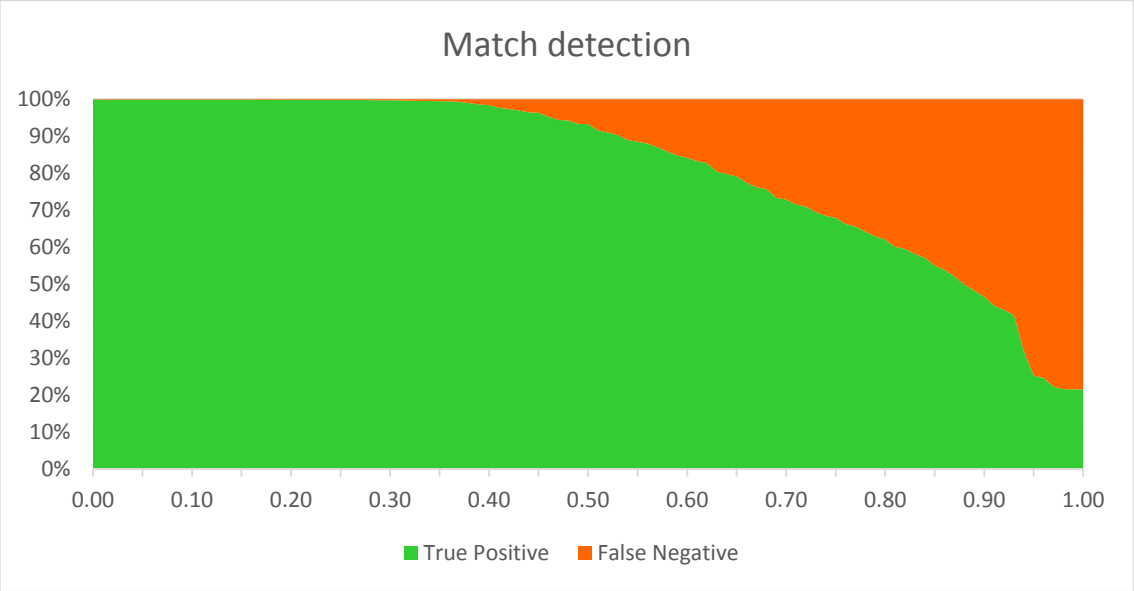
$$Overlap(x, y) = \frac{|T_x \cap T_y|}{\min(|T_x|, |T_y|)}$$

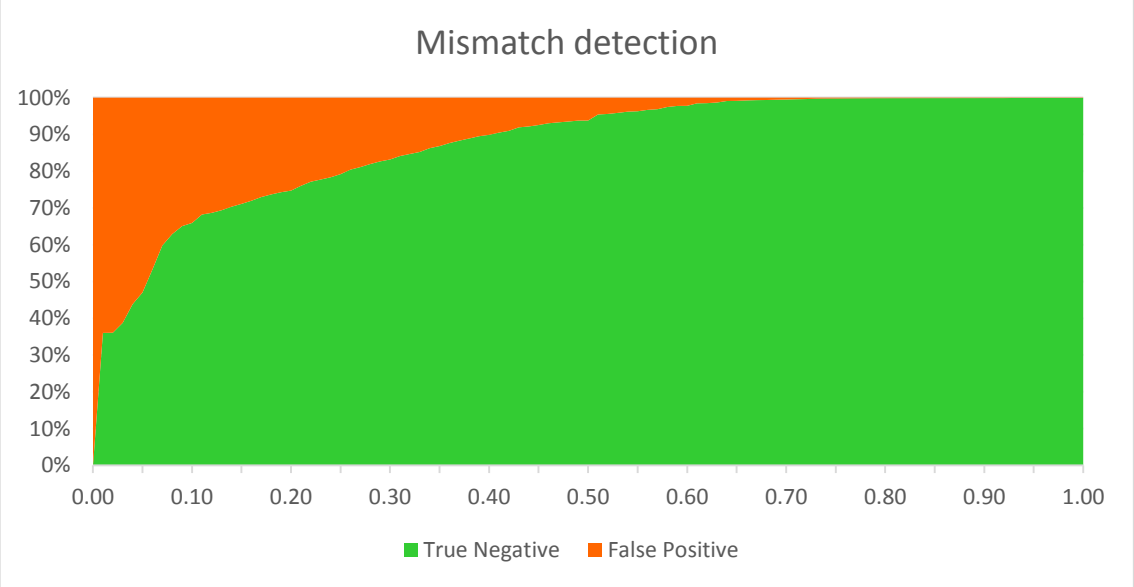
where $T_x$ denotes the set of tokens of the string $x$ and $T_y$ denotes the set of tokens of the string $y$.

## 7.2 Normalization

The metric outputs values between 0 and 1, where 1 is a good match, thus no additional normalization is required.

## 7.3 Results




Using the overlap coefficient it is possible to detect 90.25% of the matches at a sensitivity of 0.53. At this sensitivity it detects 95.87% of the mismatches, which is slightly worse than Jaccard similarity.

When at least 95% mismatch detection is required, it detects 91.56% of the matches at a sensitivity of 0.51, performing almost one percent worse than Jaccard similarity. With 99.50% mismatch detection at a sensitivity of 0.70 it detects 72.84% of the matches, making this the best performing algorithm in this area.

# 8 Summary

I analyzed how well token based approximate string matching algorithms performed using trigrams, highlighting two particular areas.

The first one was: How many false positives (mismatches) there are when 90% of the similar songs are found?

This is how the metrics I have tried performed in this area:

| Algorithm | Sensitivity | Match | Mismatch |
|---|---|---|---|
| Cosine similarity | 0.43 | 91.03% | **96.98%** |
| Dice similarity | 0.41 | 89.82% | **96.52%** |
| Block distance | 0.40 | 90.42% | **96.20%** |
| Jaccard similarity | 0.25 | 91.51% | **95.98%** |
| Overlap coefficient | 0.53 | 90.25% | **95.87%** |
| Euclidean distance | 0.20 | 90.11% | **95.08%** |
| Matching coefficient | 0.30 | 90.21% | **93.58%** |

The cosine similarity performed best. It recognized 96.98% of different songs as different. It is closely followed by dice similarity and block distance. Matching coefficient is the performing the worst out of the seven metrics, having more than twice as many false positives as cosine similarity.

The second area focuses on how many similar songs are found when only 5% false positives are allowed.

This is how the metrics I have tried performed in this area:

| Algorithm | Sensitivity | Mismatch | Match |
|---|---|---|---|
| Cosine similarity | 0.39 | 95.20% | **94.98%** |
| Block distance | 0.37 | 95.03% | **93.27%** |
| Dice similarity | 0.38 | 95.14% | **93.18%** |
| Jaccard similarity | 0.24 | 95.59% | **92.53%** |
| Overlap coefficient | 0.51 | 95.43% | **91.56%** |
| Euclidean distance | 0.20 | 95.08% | **90.11%** |
| Matching coefficient | 0.33 | 95.18% | **86.79%** |

Again, cosine similarity did best by finding 94.98% of the similar songs. It is followed by block distance. The order is almost the same as in the previous area. The only difference is that block distance had slightly better results than dice similarity.

To match the summary of the edit-distance based algorithms, I have looked at how the metrics performed when at least 99.50% of mismatch detection is needed:

| **Algorithm** | **Sensitivity** | **Mismatch** | **Match** |
|---|---|---|---|
| Overlap coefficient | 0.70 | 99.50% | **72.84%** |
| Cosine similarity | 0.57 | 99.52% | **67.46%** |
| Euclidean distance | 0.32 | 99.51% | **63.17%** |
| Dice similarity | 0.57 | 99.55% | **60.63%** |
| Jaccard similarity | 0.40 | 99.55% | **60.63%** |
| Block distance | 0.56 | 99.53% | **60.17%** |
| Matching coefficient | 0.52 | 99.52% | **51.45%** |

Overlap coefficient performed very well in this area, pushing cosine similarity to second place. Euclidean distance is the third best, followed by dice similarity. Block distance is second to last. Matching coefficient is still the worst out of the seven metrics.

Based on these results, I will investigate how the performance of the Cosine similarity can be improved, along with the square coefficient in Chapter 6.

# VI Improving Accuracy

In Chapter 4, I selected two edit-distance based algorithms for further investigation: the Smith-Waterman distance and the overlap coefficient metric used with longest common subsequence. In Chapter 5, I selected two token based algorithms: the cosine similarity and the overlap coefficient metrics.

In this chapter I will show how the accuracy of these four algorithms can be improved by transforming the data they operate on.

## 1 Removing Brackets

### 1.1 Motivation

In Chapter 2 I have observed that additional information is often added to the song metadata in brackets. The idea is that removing contents of brackets will make the metadata of similar songs more similar. Removing all text after the first opening bracket appears to be a great solution, as it removes differences like:

| **Artist** | **Title** |
|---|---|
| Martin Garrix | Animals (Radio Edit).mp3 |
| Martin Garrix | Animals (UK Radio Edit) |
| Martin Garrix | Animals (Original Mix) |
| Martin Garrix | Animals (Official Video).mp3 |
| Avicii | Wake Me Up (Lyric Video) |
| Avicii | Wake Me Up (Official Video) |
| Unknown | Martin Garrix - Animals (Official Video) |
| Unknown | Martin Garrix - Animals (Official Video HD).mp3 |
| Unknown | Martin Garrix - Animals (Official Video) - YouTube |

However, this might also have downsides. For example the UK Radio edit might be different than the original mix version, but if they really are different, the length will be different and that can be used to further refine the results.

## 1.2 Method

To test what effect removing brackets has, I have created a preprocessing that removes any text starting with the first bracket opening character. It also includes square brackets and braces in addition to the regular bracket character.

```
public class RemoveBracketsPreprocessing : Preprocessing<Song>
{
    private static readonly char[] Bracketchars = { '(', '[', '{' };

    public override Song Process(Song input)
    {
        input.Artist = RemoveBrackets(input.Artist);
        input.Title = RemoveBrackets(input.Title);

        return input;
    }

    public static string RemoveBrackets(string input)
    {
        int index = input.IndexOfAny(Bracketchars);
        return index > 0 ? input.Substring(0, index) : input;
    }
}
```

I created a pipeline that included this preprocessing. It also contains the previously used ignore case preprocessing and the n-gram tokenizer with a length of 3. It contains the four metrics I have selected for further investigation in Chapters 3 and 4.

```
public class BracketPipeline : Pipeline<Song>
{
    public BracketPipeline()
    {
        this.Preprocessings.Add(new IgnoreCasePreprocessing());
        this.Preprocessings.Add(new RemoveBracketsPreprocessing());
        this.Preprocessings.Add(new NGramTokenizer(3));

        this.Metrics.Add(new LongestCommonSubsequenceOverlapConcatMetric());
        this.Metrics.Add(new SmithWatermanDistanceMetric());
        this.Metrics.Add(new CosineSimilarityMetric());
        this.Metrics.Add(new OverlapCoefficientMetric());

        this.Verifier = new ClusterVerifier();

        this.Summarizers.Add(new CsvSummarizer());
    }
}
```

## 1.3 Results

Using the pipeline I have obtained the following results:

| | Before | | After | | Change | |
|---|---|---|---|---|---|---|
| **At least 90% match** | **Sens.** | **Mism.** | **Sens.** | **Mism.** | **Sens.** | **Mism.** |
| Cosine similarity | 0.43 | 96.98% | 0.51 | 98.51% | +0.08 | +1.54% |
| Overlap coefficient (LCS) | 0.63 | 85.87% | 0.71 | 93.12% | +0.08 | +7.25% |
| Smith-Waterman dist. | 0.45 | 93.37% | 0.56 | 95.36% | +0.11 | +1.99% |
| Overlap coefficient (token) | 0.53 | 95.87% | 0.65 | 99.17% | +0.12 | +3.30% |
| | | | | | | |
| **At least 95% mismatch** | **Sens.** | **Match** | **Sens.** | **Match** | **Sens.** | **Match** |
| Cosine similarity | 0.39 | 94.98% | 0.42 | 95.92% | +0.03 | +0.94% |
| Overlap coefficient (LCS) | 0.76 | 77.58% | 0.76 | 87.65% | 0.00 | +10.07% |
| Smith-Waterman dist. | 0.51 | 86.53% | 0.54 | 91.21% | +0.03 | +4.68% |
| Overlap coefficient (token) | 0.51 | 91.56% | 0.53 | 95.28% | +0.02 | +3.72% |
| | | | | | | |
| **At least 99.5% mismatch** | **Sens.** | **Match** | **Sens.** | **Match** | **Sens.** | **Match** |
| Cosine similarity | 0.57 | 67.46% | 0.60 | 76.61% | +0.03 | +9.14% |
| Overlap coefficient (LCS) | 0.91 | 58.25% | 0.87 | 79.77% | -0.04 | +21.52% |
| Smith-Waterman dist. | 0.67 | 66.00% | 0.69 | 77.70% | +0.02 | +11.70% |
| Overlap coefficient (token) | 0.70 | 72.84% | 0.69 | 86.95% | -0.01 | +14.12% |

To achieve the same accuracy goals as earlier, sensitivity had to be increased in all but three cases. The removal of brackets had positive effect on the results of all algorithms. It had the largest effect combining all three areas on the overlap coefficient when used with longest common subsequence. The new preprocessing had the smallest effect on the cosine similarity metric, but it was still significant in the third area.

## 1.4 Conclusion

The removal of text starting from the first opening bracket had positive effects on all four algorithms in all three areas using the three sample sets of songs from Chapter 3. I recommend using this preprocessing to improve results for all algorithms.

# 2 Phonetic Indexing

## 2.1 Motivation

I was looking for a way to accurately deal with spelling mistakes. Phonetic algorithms were originally developed to match names despite minor misspellings, based on their pronunciation. This makes these algorithms a good candidate to solve the problem of spelling mistakes in song metadata. Since such algorithms depend on the pronunciation, they are most of the times designed for a single language.

The original phonetic indexing is *Soundex*. It was designed to index English surnames. *Metaphone* is a phonetic algorithm created as an improvement over Soundex. It produces more accurate results and is suitable for not just first names, but English words in general.

*Double Metaphone* is an improved version of the Metaphone algorithm. It supports a number of other languages other than English. It is called “double” because it produces two encodings to account for different of pronunciations.

*Metaphone 3* is the current version of the algorithm, which achieves even better accuracy. It receives frequent updates, the latest being version 2.5.4 at the time of writing. However, unlike previous versions, it is a commercial software.

## 2.2 Method

Due to Metaphone 3’s commercial nature, I tested how the usage of Double Metaphone can improve the accuracy of song matching. I have created a preprocessing that splits the author and title fields into words based on 9 characters I have chosen because hey can be used to denote the end of the word. These are: whitespace, underscore, dot, comma, minus sign, single quote, backslash, dash and the ampersand. For each word, I calculate the Double Metaphone and append the resulting primary keys into a single string, separated by whitespace characters. It is important to note that the secondary encoding provided by the algorithm is not utilized in this case because they would have to be independently compared, something which is not supported by the pipeline.

```
    public class MetaphoneGenerationPerWordPreprocessing : Preprocessing<Song>
    {
        private static readonly char[] Splitchars = { ' ', '_', '.', ',', '-',
'`', '\'', '–', '&' };

        private DoubleMetaphone doubleMetaphone = new DoubleMetaphone();
        private StringBuilder stringBuilder = new StringBuilder();

        public override Song Process(Song input)
        {
            input.Artist = GenerateMetaphone(input.Artist);
            input.Title = GenerateMetaphone(input.Title);

            return input;
        }

        private string GenerateMetaphone(string input)
        {
            string[] parts = input.Split(Splitchars,
StringSplitOptions.RemoveEmptyEntries);
            foreach (string part in parts)
            {
                this.doubleMetaphone.computeKeys(part);
                this.stringBuilder.Append(this.doubleMetaphone.PrimaryKey);
                this.stringBuilder.Append(" ");
            }

            var result = stringBuilder.ToString();
            stringBuilder.Clear();

            return result;
        }
    }
```

I have created a pipeline that included this preprocessing, along with the previously used preprocessing methods and the metrics:

```
    public class MetaphonePipeline : Pipeline<Song>
    {
        public MetaphonePipeline()
        {
            this.Preprocessings.Add(new IgnoreCasePreprocessing());
            this.Preprocessings.Add(new
MetaphoneGenerationPerWordPreprocessing());
            this.Preprocessings.Add(new NGramTokenizer(3));

            this.Metrics.Add(new LongestCommonSubsequenceOverlapConcatMetric());
            this.Metrics.Add(new SmithWatermanDistanceMetric());
            this.Metrics.Add(new CosineSimilarityMetric());
            this.Metrics.Add(new OverlapCoefficientMetric());

            this.Verifier = new ClusterVerifier();

            this.Summarizers.Add(new CsvSummarizer());
        }
    }
```

The Double Metaphone algorithm is designed to work with the Latin alphabet. Thus it will not be able to match songs that contain non-Latin characters in their title or artist field. To account for use where the primary focus is on English songs, I separated the results of the foreign sample set from the other two sample sets.

## 2.3 Results

### 2.3.1 Popular and Similar Data Set

Using the pipeline I have obtained the following results for the poplar and similar data sets:

| | **Before** | | **After** | | **Change** | |
|---|---|---|---|---|---|---|
| **At least 90% match** | **Sens.** | **Mism.** | **Sens.** | **Mism.** | **Sens.** | **Mism.** |
| Cosine similarity | 0.44 | 96.97% | 0.50 | 97.77% | +0.06 | +0.79% |
| Overlap coefficient (LCS) | 0.62 | 83.01% | 0.71 | 76.67% | +0.09 | -6.34% |
| Smith-Waterman dist. | 0.45 | 92.43% | 0.52 | 92.54% | +0.07 | +0.11% |
| Overlap coefficient (token) | 0.53 | 95.24% | 0.57 | 95.86% | +0.04 | +0.61% |
| | | | | | | |
| **At least 95% mismatch** | **Sens.** | **Match** | **Sens.** | **Match** | **Sens.** | **Match** |
| Cosine similarity | 0.40 | 94.58% | 0.44 | 96.22% | +0.04 | +1.64% |
| Overlap coefficient (LCS) | 0.77 | 75.92% | 0.88 | 68.51% | +0.11 | -7.41% |
| Smith-Waterman dist. | 0.51 | 86.62% | 0.61 | 82.21% | +0.10 | -4.41% |
| Overlap coefficient (token) | 0.53 | 90.67% | 0.56 | 92.48% | +0.03 | +1.82% |
| | | | | | | |
| **At least 99.5% mismatch** | **Sens.** | **Match** | **Sens.** | **Match** | **Sens.** | **Match** |
| Cosine similarity | 0.57 | 67.60% | 0.61 | 72.60% | +0.04 | +5.00% |
| Overlap coefficient (LCS) | 0.91 | 58.18% | - | - | N/A | N/A |
| Smith-Waterman dist. | 0.67 | 66.11% | 0.78 | 64.47% | +0.11 | -1.64% |
| Overlap coefficient (token) | 0.71 | 71.88% | 0.76 | 77.95% | +0.05 | +6.08% |

To achieve the same three accuracy goals as earlier, sensitivity had to be increased in all cases. The overlap coefficient metric based on the longest common subsequence suffered the most from the Double Metaphone preprocessing. It was not capable of achieving 99.5% mismatch detection accuracy. The other edit-distance based algorithm, the Smith-Waterman distance saw a slight improvement in the first area, but in the other two areas its accuracy suffered from the new preprocessing.

The phonetic indexing had positive effect on the results of both token based algorithms. The overlap coefficient has improved slightly better than cosine similarity in the second and the third area.

### 2.3.2 Foreign Data Set

Using the pipeline I have obtained the following results for the foreign data set:

| | **Before** | | **After** | | **Change** | |
|---|---|---|---|---|---|---|
| **At least 90% match** | **Sens.** | **Mism.** | **Sens.** | **Mism.** | **Sens.** | **Mism.** |
| Cosine similarity | 0.29 | 98.09% | 0.35 | 77.20% | +0.06 | -20.89% |
| Overlap coefficient (LCS) | 0.67 | 98.60% | 0.94 | 53.31% | +0.27 | -45.29% |
| Smith-Waterman dist. | 0.43 | 98.15% | 0.80 | 64.46% | +0.37 | -33.68% |
| Overlap coefficient (token) | 0.40 | 98.42% | 0.78 | 68.49% | +0.38 | -29.93% |
| | | | | | | |
| **At least 95% mismatch** | **Sens.** | **Match** | **Sens.** | **Match** | **Sens.** | **Match** |
| Cosine similarity | 0.08 | 96.18% | - | - | N/A | N/A |
| Overlap coefficient (LCS) | 0.51 | 95.42% | - | - | N/A | N/A |
| Smith-Waterman dist. | 0.23 | 96.44% | - | - | N/A | N/A |
| Overlap coefficient (token) | 0.11 | 96.18% | - | - | N/A | N/A |
| | | | | | | |
| **At least 99.5% mismatch** | **Sens.** | **Match** | **Sens.** | **Match** | **Sens.** | **Match** |
| Cosine similarity | 0.54 | 67.68% | - | - | N/A | N/A |
| Overlap coefficient (LCS) | 0.81 | 71.50% | - | - | N/A | N/A |
| Smith-Waterman dist. | 0.67 | 63.36% | - | - | N/A | N/A |
| Overlap coefficient (token) | 0.63 | 68.96% | - | - | N/A | N/A |

As expected, all algorithms accuracy has heavily suffered from the Double Metaphone preprocessing. All of them became incapable of detecting 95% of the mismatches.

## 2.4 Conclusion

Preprocessing words in the title and artist fields using the Double Metaphone algorithm resulted in improvements when used with token based algorithms on English song metadata. If edit-distance based algorithms were used or non-English songs were matched, phonetic indexing had negative effect on the accuracy. I recommend using this preprocessing only if songs are guaranteed to have English metadata. It that cannot be guaranteed, it is best to avoid the Double Metaphone phonetic indexing.

# 3 Unknown Artist

## 3.1 Motivation

In Chapter 2 I have observed that song metadata is often malformed. The artist field is often empty and information on both the artist and the title is stored in the title field. For example:

| Artist | Title |
|---|---|
| Unknown | David Guetta - Titanium ft. Sia |
| Unknown | David Guetta  Titanium ft Sia.mp3 |
| Unknown | david_guetta_titanium_ft._sia_mp3_66896 |
| Unknown | David Guetta feat. Sia - Titanium Official Music |
| Unknown | David Guetta Feat.Sia- Titanium.mp3 |
| Unknown | Avicii - Wake Me Up.m4a |
| Unknown | aviicii-wake me up (1).mp3 |
| Unknown | Avicii - Avicii - Wake Me Up (Lyric Video)-[www_flvto_com] (2) |
| Unknown | Wake Me Up  Avicii ft. Aloe Blacc  Lyrics |
| Unknown | Avicii  Wake Me Up ft Aloe Blacc (mashup music) |
| Unknown | MArtin Garrix - Animals |
| Unknown | Martin_Garrix_-_Animals.mp3 |
| Unknown | Martin Garrix - Animals (Official Video) |
| Unknown | Martin Garrix - Animals OFFICIAL TV CENSORED VIDEO HD |

The idea is to ignore the content of the artist field if it start with the word “Unknown”. I expect this to improve the matching accuracy for songs with incomplete metadata.

## 3.2 Method

To test what effect removing the artist has if it is unknown, I have created a preprocessing that clears the artist field if it starts with the word “Unknown”.

```
    public class UnknownArtistPreprocessing : Preprocessing<Song>
    {
        public override Song Process(Song input)
        {
            if (input.Artist.StartsWith("unknown",
StringComparison.OrdinalIgnoreCase))
                input.Artist = null;

            return input;
        }
    }
```

I created a pipeline that included this preprocessing, along with the previously used preprocessing methods and the metrics:

```
public class UnknownArtistPipeline : Pipeline<Song>
{
    public UnknownArtistPipeline()
    {
        this.Preprocessings.Add(new IgnoreCasePreprocessing());
        this.Preprocessings.Add(new UnknownArtistPreprocessing());
        this.Preprocessings.Add(new NGramTokenizer(3));

        this.Metrics.Add(new LongestCommonSubsequenceOverlapConcatMetric());
        this.Metrics.Add(new SmithWatermanDistanceMetric());
        this.Metrics.Add(new CosineSimilarityMetric());
        this.Metrics.Add(new OverlapCoefficientMetric());

        this.Verifier = new ClusterVerifier();

        this.Summarizers.Add(new CsvSummarizer());
    }
}
```

## 3.3 Results

Using the pipeline I have obtained the following results:

| | Before | | After | | Change | |
|---|---|---|---|---|---|---|
| **At least 90% match** | **Sens.** | **Mism.** | **Sens.** | **Mism.** | **Sens.** | **Mism.** |
| Cosine similarity | 0.43 | 96.98% | 0.44 | 97.95% | +0.01 | +0.98% |
| Overlap coefficient (LCS) | 0.63 | 85.87% | 0.65 | 89.97% | +0.02 | +4.10% |
| Smith-Waterman dist. | 0.45 | 93.37% | 0.46 | 94.25% | +0.01 | +0.88% |
| Overlap coefficient (token) | 0.53 | 95.87% | 0.54 | 96.82% | +0.01 | +0.95% |
| | | | | | | |
| **At least 95% mismatch** | **Sens.** | **Match** | **Sens.** | **Match** | **Sens.** | **Match** |
| Cosine similarity | 0.39 | 94.98% | 0.37 | 97.40% | -0.02 | +2.43% |
| Overlap coefficient (LCS) | 0.76 | 77.58% | 0.76 | 80.63% | 0.00 | +3.05% |
| Smith-Waterman dist. | 0.51 | 86.53% | 0.49 | 88.59% | -0.02 | +2.06% |
| Overlap coefficient (token) | 0.51 | 91.56% | 0.51 | 92.80% | 0.00 | +1.24% |
| | | | | | | |
| **At least 99.5% mismatch** | **Sens.** | **Match** | **Sens.** | **Match** | **Sens.** | **Match** |
| Cosine similarity | 0.57 | 67.46% | 0.53 | 77.50% | -0.04 | +10.04% |
| Overlap coefficient (LCS) | 0.91 | 58.25% | 0.91 | 60.61% | 0.00 | +2.36% |
| Smith-Waterman dist. | 0.67 | 66.00% | 0.67 | 67.57% | 0.00 | +1.57% |
| Overlap coefficient (token) | 0.70 | 72.84% | 0.64 | 80.73% | -0.06 | +7.89% |

To achieve the same three goals as earlier, sensitivity had to be adjusted in most cases. The preprocessing had positive effect on the results of all algorithms. There is no algorithm that can be highlighted as the one that benefited the most from the

preprocessing. While the two token based algorithms benefited the most in the third area, the longest subsequence based overlap coefficient benefited the most in the first two areas.

## 3.4 Conclusion

Ignoring the content of the artist field if it starts with unknown had positive effects on all four algorithms in all three areas using the three sample sets of songs from Chapter 3. I recommend using this preprocessing to improve results for all algorithms.

# 4 Different Tokenization Methods

## 4.1 S-Grams

### 4.1.1 Definition

I have measured the accuracy of the algorithms used in Chapter 5 using N-grams with a length of 3, also known as trigrams. However, there are others. S-grams may increase accuracy by not just including contiguous sequences of characters, but also non-adjacent character sequences by skipping a given number of characters. For example for the word “abc” the N-gram with a length of 2 are “ab” and “bc”. S-grams with a length of 2 also include “ac” in addition to “ab” and “bc” if a single character is skipped. To test their accuracy I have generated S-grams where a single character is skipped in the non-adjacent sequences of characters.

### 4.1.2 Results

This is how N-grams and S-grams perform with a length of 2 to 4 in the three areas I am focusing on:

Mismatch accuracy with at least 90% match accuracy:

| Tokenization | BD | CS | DS | ED | JS | MC | OC |
|---|---|---|---|---|---|---|---|
| N-Gram 2 | 95.52% | 96.33% | 95.43% | 94.34% | 94.99% | 90.86% | 93.07% |
| N-Gram 3 | **96.20%** | **96.98%** | **95.98%** | 95.08% | **95.98%** | **93.47%** | **95.87%** |
| N-Gram 4 | 95.27% | 96.44% | 95.56% | 94.42% | 95.15% | 92.85% | 95.78% |
| S-Gram 2 | 94.85% | 95.48% | 94.00% | 94.87% | 94.05% | 87.60% | 92.43% |
| S-Gram 3 | 95.81% | 96.72% | 95.94% | **95.51%** | 95.77% | 93.34% | 95.52% |
| S-Gram 4 | 94.86% | 95.80% | 94.77% | 94.25% | 94.51% | 92.57% | 95.86% |

Match accuracy with at least 95% mismatch accuracy:

| Tokenization | BD | CS | DS | ED | JS | MC | OC |
|---|---|---|---|---|---|---|---|
| N-Gram 2 | 90.70% | 93.30% | 90.71% | 89.82% | 89.71% | 79.18% | 87.14% |
| N-Gram 3 | **93.27%** | **94.98%** | **93.18%** | 90.11% | 92.53% | **86.41%** | 91.56% |
| N-Gram 4 | 90.55% | 92.07% | 91.21% | 89.08% | 91.21% | 85.23% | **93.51%** |
| S-Gram 2 | 88.41% | 92.09% | 88.13% | 88.11% | 86.93% | 76.86% | 85.85% |
| S-Gram 3 | 92.00% | 93.47% | 92.78% | **90.32%** | **92.78%** | 85.44% | 92.58% |
| S-Gram 4 | 89.33% | 91.75% | 89.56% | 89.50% | 89.59% | 84.21% | 92.09% |

Match accuracy with at least 99.5% mismatch accuracy:

| Tokenization | BD | CS | DS | ED | JS | MC | OC |
|---|---|---|---|---|---|---|---|
| N-Gram 2 | 58.33% | 58.13% | 53.79% | 59.74% | 55.63% | 46.13% | 71.74% |
| N-Gram 3 | 60.17% | **67.46%** | 60.63% | **63.17%** | 60.63% | 54.09% | **72.84%** |
| N-Gram 4 | 58.85% | 66.59% | 60.32% | 61.61% | 61.71% | **54.34%** | 70.49% |
| S-Gram 2 | 56.84% | 57.72% | 53.18% | 59.13% | 54.07% | 44.01% | 69.12% |
| S-Gram 3 | **61.80%** | 66.45% | **62.87%** | 60.72% | **62.87%** | 53.88% | 71.86% |
| S-Gram 4 | 58.81% | 62.89% | 59.69% | 56.35% | 58.06% | 53.24% | 69.23% |

Legend:

BD: Block Distance, CS: Cosine Similarity, DS: Dice Similarity, ED: Euclidean Distance, JS: Jaccard Similarity, MC: Matching Coefficient, OC: Overlap Coefficient

### 4.1.3 Conclusion

N-grams with a length of three had the best accuracy overall, followed by S-grams with a length of 3. N-gram with a length of 3 had the best accuracy in all three areas for cosine similarity. For overlap coefficient it was the best in two areas.

## 4.2 Extended N-Grams and S-Grams

### 4.2.1 Definition

Extended N-grams and S-grams append special characters to the beginning and the end of the string. This way matching tokens at the beginning and the end of the word will count as an extra match, increasing their weight. Usually the “?” character is used to denote the beginning, and the “#” character is used to denote the end of the string.

For example the extended N-grams with a length of 2 for the word “abc” are: “?a”, “ab”, “bc”, “c#”.

### 4.2.2 Results

This is how extended N-grams and S-grams perform with a length of 2 to 4 in the three areas I am focusing on (N-gram with length of 3 is included for reference):

Mismatch accuracy with at least 90% match accuracy:

| Tokenization | BD | CS | DS | ED | JS | MC | OC |
|---|---|---|---|---|---|---|---|
| N-Gram 3 | **96.20%** | **96.98%** | **95.98%** | **95.08%** | **95.98%** | **93.47%** | **95.87%** |
| N-Gram 2 Extended | 95.08% | 95.51% | 94.46% | 93.94% | 94.48% | 89.46% | 93.22% |
| N-Gram 3 Extended | 94.19% | 95.44% | 94.49% | 94.02% | 94.80% | 91.90% | 94.92% |
| N-Gram 4 Extended | 92.64% | 93.32% | 92.48% | 91.57% | 92.97% | 91.30% | 93.19% |
| S-Gram 2 Extended | 93.52% | 94.21% | 92.45% | 93.72% | 92.21% | 86.03% | 91.91% |
| S-Gram 3 Extended | 94.29% | 95.16% | 94.13% | 93.37% | 93.96% | 91.59% | 94.46% |
| S-Gram 4 Extended | 92.18% | 93.55% | 92.64% | 92.29% | 92.68% | 89.95% | 93.53% |

Match accuracy with at least 95% mismatch accuracy:

| Tokenization | BD | CS | DS | ED | JS | MC | OC |
|---|---|---|---|---|---|---|---|
| N-Gram 3 | **93.27%** | **94.98%** | **93.18%** | **90.11%** | **92.53%** | **86.41%** | **91.56%** |
| N-Gram 2 Extended | 90.89% | 90.20% | 87.52% | 86.74% | 88.77% | 76.61% | 87.83% |
| N-Gram 3 Extended | 88.60% | 91.45% | 89.56% | 85.39% | 89.02% | 80.96% | 88.33% |
| N-Gram 4 Extended | 83.49% | 85.24% | 84.60% | 83.28% | 85.21% | 79.40% | 86.60% |
| S-Gram 2 Extended | 85.13% | 87.56% | 84.79% | 86.31% | 83.65% | 73.26% | 84.19% |
| S-Gram 3 Extended | 88.19% | 90.44% | 88.82% | 87.16% | 88.35% | 80.61% | 88.37% |
| S-Gram 4 Extended | 84.01% | 85.67% | 84.70% | 83.79% | 84.46% | 77.49% | 85.55% |

Match accuracy with at least 99.5% mismatch accuracy:

| Tokenization | BD | CS | DS | ED | JS | MC | OC |
|---|---|---|---|---|---|---|---|
| N-Gram 3 | **60.17%** | **67.46%** | **60.63%** | **63.17%** | **60.63%** | **54.09%** | **72.84%** |
| N-Gram 2 Extended | 55.04% | 53.29% | 50.69% | 53.29% | 52.29% | 44.87% | 67.46% |
| N-Gram 3 Extended | 53.91% | 56.49% | 55.79% | 55.43% | 55.79% | 49.08% | 67.01% |
| N-Gram 4 Extended | 51.78% | 53.74% | 51.58% | 50.21% | 51.65% | 46.59% | 59.32% |
| S-Gram 2 Extended | 52.21% | 53.43% | 50.38% | 53.99% | 50.68% | 43.42% | 66.48% |
| S-Gram 3 Extended | 58.10% | 57.72% | 57.03% | 56.94% | 56.81% | 51.30% | 67.14% |
| S-Gram 4 Extended | 52.83% | 54.00% | 53.62% | 52.23% | 54.13% | 50.13% | 58.88% |

Legend:

BD: Block Distance, CS: Cosine Similarity, DS: Dice Similarity, ED: Euclidean Distance, JS: Jaccard Similarity, MC: Matching Coefficient, OC: Overlap Coefficient

### 4.2.3 Conclusion

All tested extended tokenization methods are outperformed by N-gram tokenization with a length of 3.

## 4.3 Summary

The overall best accuracy is achieved by N-grams with a length of three, followed by S-grams with a length of 3. All extended N-grams and S-grams had worse accuracy than trigrams. I recommend using N-grams with a length of three for tokenizing song artist and title fields.

# 5 Combining Improvements

## 5.1 Improved Pipeline

I have found that removing text starting from the first opening bracket of both the artist and title fields have improved matching accuracy significantly with all algorithms used. It is included in the improved pipeline.

Phonetic indexing using Double Metaphone has only increased accuracy if all song metadata is in English, this is however not always the case. It made it impossible to detect similar songs which contain only non-Latin characters. Therefore I will not include it in the improved pipeline.

Ignoring the content of the artist field if it starts with “unknown” improved matching accuracy of all algorithms. Albeit it resulted in smaller gains than the removal of brackets, it had significant effect on the token based algorithms. As a result of that, I will include it in the improved pipeline.

N-grams with a length of three have performed the best out of the tokenization methods I have investigated. While S-grams with a length of three performed better in some cases, the best results were obtained using N-grams. Therefore the improved pipeline will use trigrams for tokenization.

The improved pipeline is the following:

```
public class ImprovedPipeline : Pipeline<Song>
{
    public ImprovedPipeline()
    {
        this.Preprocessings.Add(new IgnoreCasePreprocessing());
        this.Preprocessings.Add(new UnknownArtistPreprocessing());
        this.Preprocessings.Add(new RemoveBracketsPreprocessing());
        this.Preprocessings.Add(new NGramTokenizer(3));

        this.Metrics.Add(new LongestCommonSubsequenceOverlapConcatMetric());
        this.Metrics.Add(new SmithWatermanDistanceMetric());
        this.Metrics.Add(new CosineSimilarityMetric());
        this.Metrics.Add(new OverlapCoefficientMetric());

        this.Verifier = new ClusterVerifier();

        this.Summarizers.Add(new CsvSummarizer());
    }
}
```

## 5.2 Results

Using the improved pipeline I have obtained the following results:

| | **Before** | | **After** | | **Change** | |
|---|---|---|---|---|---|---|
| **At least 90% match** | **Sens.** | **Mism.** | **Sens.** | **Mism.** | **Sens.** | **Mism.** |
| Cosine similarity | 0.43 | 96.98% | 0.51 | 98.95% | +0.08 | +1.97% |
| Overlap coefficient (LCS) | 0.62 | 91.64% | 0.76 | 96.24% | +0.14 | +4.60% |
| Smith-Waterman dist. | 0.45 | 93.37% | 0.58 | 96.11% | +0.13 | +2.74% |
| Overlap coefficient (token) | 0.53 | 95.87% | 0.65 | 99.76% | +0.12 | +3.89% |
| | | | | | | |
| **At least 95% mismatch** | **Sens.** | **Match** | **Sens.** | **Match** | **Sens.** | **Match** |
| Cosine similarity | 0.39 | 94.98% | 0.42 | 96.78% | +0.03 | +1.80% |
| Overlap coefficient (LCS) | 0.77 | 75.97% | 0.74 | 91.37% | -0.03 | +15.40% |
| Smith-Waterman dist. | 0.51 | 86.53% | 0.55 | 93.10% | +0.04 | +6.57% |
| Overlap coefficient (token) | 0.51 | 91.56% | 0.52 | 96.48% | +0.01 | +4.92% |
| | | | | | | |
| **At least 99.5% mismatch** | **Sens.** | **Match** | **Sens.** | **Match** | **Sens.** | **Match** |
| Cosine similarity | 0.57 | 67.46% | 0.55 | 87.27% | -0.02 | +19.80% |
| Overlap coefficient (LCS) | 0.91 | 58.25% | 0.82 | 87.30% | -0.09 | +29.05% |
| Smith-Waterman dist. | 0.67 | 66.00% | 0.67 | 78.78% | 0.00 | +12.78% |
| Overlap coefficient (token) | 0.70 | 72.84% | 0.64 | 91.91% | -0.06 | +19.07% |

The improvements had positive effect on the matching accuracy of all algorithms in all three areas. The longest common subsequence based overlap coefficient metric saw the

biggest improvements, between 4% and 29%. The overall biggest improvement was in the third area, where the match accuracy has improved by 12% to 29% for all algorithms.

# 6 Real-World Usage

I have implemented the four algorithms with the suggested improvements in the game SongArc, in the context described in Chapter 1. Real-world usage revealed that token based algorithms are not just more accurate, but are also faster on mobile devices.

Both overlap coefficient based methods turned out to be a source of large number of mismatches in case of very short metadata. For example if a song had a missing artist field and a title field with the letter “A”, it was a perfect match for all songs that contained the letter “A” in either the artist or the title, leading to tens of thousands of false positives. Cosine similarity and Smith-Waterman distance had no such problems. Cosine similarity seemed to perform better than Smith-Waterman distance. This makes it the algorithm I recommend for actual usage.

# VII Summary

The goal of the thesis was to identify algorithms that can be used to tell if two songs are the same based on the artist, tile and duration metadata. To understand the problem I gathered metadata for more than 95000 songs. I had a look at what kind of variances there are for three popular songs. I have found 85 to 95 different versions of metadata for the three songs in the whole collection. The difference of duration between the shortest and the longest version of the same song was between 20 and 40 seconds.

The artist and title fields of the metadata had large variances. In the case of the three songs I selected I have found that spelling mistakes were infrequent. The main source of variance was the presence of additional artists, often referred to as featured artists. If there was more than one artist, the name of the additional artist was present after the name of the main artist, after the title, or was omitted completely. In a few cases the name of the main artist was omitted and only the name of the additional artist was present. I have also found that metadata is frequently incomplete. In those cases, the artist field is empty or contains the word “Unknown”. Metadata is only present in the title field, which can contain the name of the artist.

To identify how well different algorithms perform at finding similar songs I have created three sample sets. The first sample set contained the three popular songs I have analyzed earlier, and their variances. This sample set was used to measure how well the algorithms perform with songs that are found in the collection of most people. To be able to detect false positives I have selected additional songs that were similar in either the artist or the title. The second sample set contained songs with similar titles. Some of the songs were only different in their artist. This sample set was used to measure how well algorithms perform at detecting different songs when the metadata is similar. The third sample set included songs that had metadata containing non-Latin characters. This sample set was used to measure how well algorithms perform at detecting similar songs that are not part of the main stream, English music culture. To test the algorithms, I have created a pipeline that could be customized with different kinds of components. The pipeline ran the algorithms on three sample sets.

I have selected various approximate string matching algorithms to examine. These algorithms were based on two distinct approaches. The first approach was to tell how similar two strings are based on edit-distance. I have selected five algorithms plus four metrics that were based on either the longest common substring or the longest common subsequence of two strings. The other approach was based on tokens. In that case, strings were turned into a bag of tokens. I have used trigrams – continuous sequences of characters with a length of three – to examine seven metrics. For all algorithms I have inspected how many similar and different songs they can detect at various sensitivity levels. Based on the overall accuracy I have identified two algorithms from each of the two approaches that had the best results. These were the Smith-Waterman distance and the longest common subsequence based overlap coefficient metric from the edit-distance based algorithms. From the token based algorithms I have selected the cosine similarity and the overlap coefficient.

In the final investigation my goal was to improve the accuracy of the four algorithms I have selected to be the best, by transforming their input. I have found that removing text starting with the first bracket character improved the results significantly. The edit-distance based algorithms benefited the most, especially when the goal was to find a low number of false positives.

Phonetic indexing using the Double Metaphone algorithm had positive effect on the token based algorithms, but only with the first two datasets. Since Double Metaphone does not work with non-Latin characters, it had a devastating effect on the algorithms ability to work with the foreign data set.

Ignoring the artist metadata field if it contained the word “unknown” improved the accuracy of all algorithms. I have found that it had the greatest effect on token based algorithms.

Finally, I have investigated if the token based algorithms can be improved by using different tokenization methods. I have measured the matching accuracy of all seven token based metrics using various tokenization methods: n-grams, s-grams, extended n-grams and extended s-grams with a length of 2, 3 and 4. I have found that the best

overall accuracy is achieved by n-grams with a length of three – the same tokenization method that was used in the previous chapters, known as trigrams.

To conclude, I have combined the two removal of text starting with the first opening bracket and the ignoring of the artist field if contained to word “unknown”. Using these two improvements in conjunction with trigrams improved the accuracy of all four algorithms, both in finding similar and different songs.

When at least 90% of the similar songs were required to be detected, thanks to the improvements the algorithms detected 1.7 to 16.91 times less false positives. When at least 95% of the different songs were required to be detected, the algorithms detected 1.9% to 20.27% more similar songs. When at least 99.5% of the different songs were required to be detected, the algorithms detected 19.37% to 49.87% more of the similar songs.

In conclusion, I have found that using my improvements the cosine similarity metric is best to be used to detect if two songs are similar based on the artist and title metadata, being able to detect 87.27% of the similar songs when high accuracy is required at a sensitivity of 0.55. When high accuracy is not required, it is able to detect 96.78% of the similar songs using a sensitivity of 0.42.

# VIII Attachments

## 1 DVD

The DVD contains two directories. The “Pipeline” directory contains the C# solution that was used to investigate the various algorithms. The solution contains five projects. The first project is “SongMatching.Algorithms”, which contains the reference algorithm implementations [2] [3] [5] [6]. The “SongMatching.Data” project contains the three song samples and the complete song collection with over 95000 songs. It also contains a cluster that is used to verify if two songs are similar or not. The “SongMatching.Pipeline” project contains the generic implementation of the pipeline. The “SongMatching.Matchmaker” project contains the pipeline modules and the pipelines that were used in the thesis. Finally, the “SongMatching.Console” project is a runnable, console application that runs the pipelines using the sample data and saves the output into CSV files.

The “Table” directory contains Excel spreadsheets that were used to create the diagrams in the thesis, including the raw data.

# IX References


[1] S. Chapman, “String Similarity Metrics for Information Integration,” 1996. [Online]. Available: http://web.archive.org/web/20050606011649/http://www.dcs.shef.ac.uk/~sam/stringmetrics.html. [Accessed 16 04 2015].

[2] Wang, Jiannan, Guoliang Li and Jianhua Fe, “Fast-join: An efficient method for fuzzy token matching based string similarity join.,” *Data Engineering (ICDE), 2011 IEEE 27th International Conference on,* pp. 458-469, 2011.

[3] S. Chapman and C. Parkinson, “SimMetrics,” 24 09 2006. [Online]. Available: http://sourceforge.net/projects/simmetrics/. [Accessed 05 05 2015].

[4] K. Jones, “FuzzyString - Approximate String Comparison in C#,” 01 05 2013. [Online]. Available: https://fuzzystring.codeplex.com/. [Accessed 05 05 2015].

[5] A. Nelson, “Implement Phonetic ("Sounds-like") Name Searches with Double Metaphone Part V: .NET Implementation,” 19 03 2007. [Online]. Available: http://www.codeproject.com/Articles/4624/Implement-Phonetic-Sounds-like-Name-Searches-wit. [Accessed 05 05 2015].

[6] Lucene.Net, “Jaro-Winkler Distance,” 03 01 2013. [Online]. Available: http://lucenenet.apache.org/docs/3.0.3/db/d12/_jaro_winkler_distance_8cs_source.html. [Accessed 05 05 2015].

[7] M. A. Jaro, “Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida,” *Journal of the American Statistical Association,* vol. 84, no. 406, pp. 414-420, 1989.