Development of Social Applications – Entity Resolution

Seminaristischer Unterricht

Prof. Dr.-Ing. Hendrik Gärtner

Gliederung

- Review Spark
  - Akkumulatoren/Broadcast-Variablen
  - Caching
- EntityResolution
  - Motivation
  - Similarity Measures
  - Similarity Analysis
  - Evaluation of the results
  - Scaling the Similarity Analysis

Presentations

- Document Oriented Databases: Konstantin Kochetov, Mayk Akifovski
- Column Oriented Databases: Corinna Hillebrandt, Raimi Solorzano
- Graph Databases:
- Key-Value-Stores:
- Page Rank: Donat Brzoska
- Linear/Logistic Regression mit Spark
- Recommender Systems mit Spark: Colin Leu, Marcel Kleint
- Analysis of time series in finance: Ulrich Overdieck, Tobias Trame

Presentations

- Part Of Speech Tagging:
- Clustering: Lukas Gertsch
- Web-Crawling:
- Twitter Streaming: Tilman Möller
- Kafka: Mehdi Didri
- Word2Vec: Steven Mi, Oliver Kütemeier
- SBT: Claudio Vindimian, Andrej Loparev
- Apache Spark Streaming versus FLINK: Daniel Nagel, Florian Thom
- PredictionIO: Naomi Phan

Spark Architecture

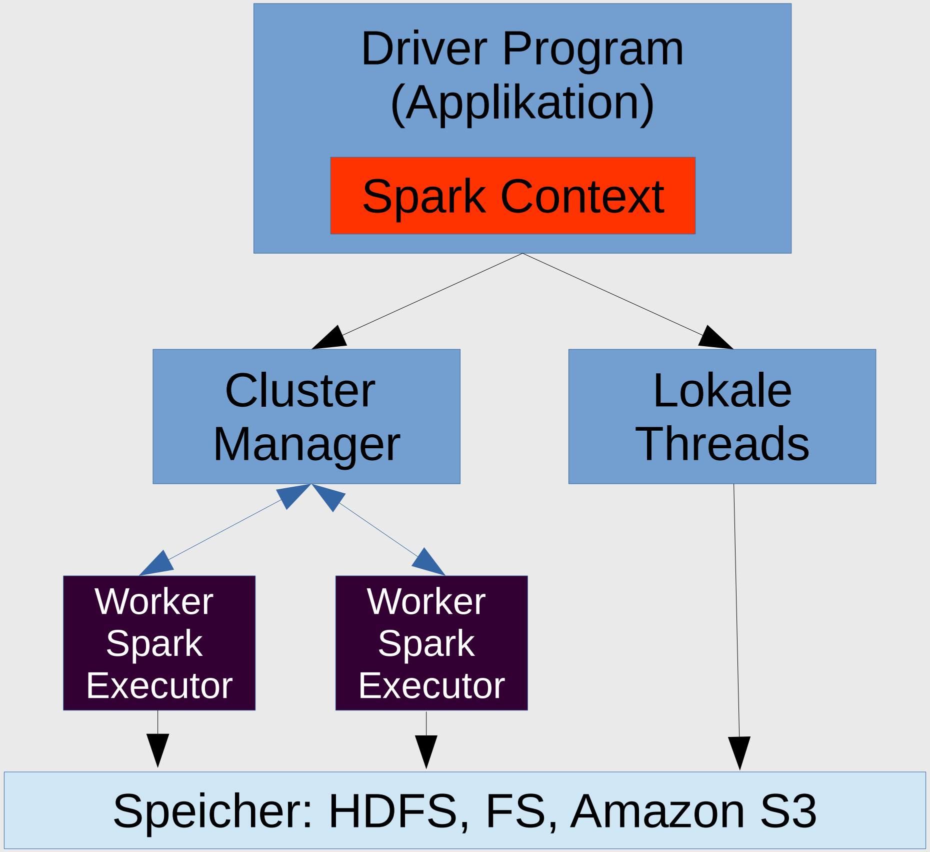
A Spark-Application consists of two programms:

- A Driver Program and
- Worker that actually do the work

A Worker runs either:

- within a cluster or
- in local threads

The RDDs distributed accross the workers



Example Scaling of the Sentiment Analysis

```
// Definition of a Dictionarie storing the Sentiment values of the words

val sentimentVals: Map[String, Double]= …

// Definition of a function to calculate the sentiment-Values

def calculateSentimentValues

        (line:String, sentimentDict:Map[String, Double]):Double


// Applying the function to the Distributed Resillient Dataset

dataRDD.map(x=> calculateSentimentValues(x,sentimentVals))
```

Spark need to transfer both to the workers: The function code of calculateSentimentValues and the variable sentimentVals

Spark's Approach

If Spark exectutes a transformation it automatically creates a **Closure** that

- becomes **serialised** on the driver node,
- will be transfered to the corresponding nodes of the cluster,
- becomes **deserialised** and
- finally gets executed on the node.

Closures

Closures are functions whose result depend on one or more variables that are declared outside of the function. Example:

```
def filterBelowFirst(xs:List[Int]):List[Int]  = {
    val firstEl = xs.head
    val isBelow =  (y:Int)=>(y < firstEl)
    xs filter isBelow
}
```

- The result of the function isBelow depends on the variable firstEl
- The variable firstEl is defined in the context of the function filterBelowFirst
- Scala defines automaticallly a Closure surrounding the function isBelow that contains the required context (the function filterBelowFist)

Serialisability of Functions

```
class SearchFunctions(val query:String){
   def isMatch(s:String):Boolean ={s.contains(query)}

    def getMatchesFunctionReference(rdd:RDD[String]):RDD[Boolean]={
       // problem: isMatch means this.isMatch), so we pass all of this
       rdd.map(isMatch)}

    def getMatchesFieldReference(rdd:RDD[String]):RDD[Array[String]]={
       // problem: isMatch means this.isMatch), so we pass all of this
       rdd.map(x=>x.split(query))}

  def getMatchesNoReference(rdd:RDD[String]):RDD[Array[String]]={
      // Safe: Extracts just the field we need into a local variable
      val query_ = this.query
      rdd.map(_.split(query_))}
}
```

Example from: Learning Spark – Lightning Fast Data Analysis, O'Reilley, 2015, page 32

Serializability of Functions

```
import org.apache.spark.rdd.RDD

class SearchFunctions(val query:String) {

     def getMatchesFunctionReference(rdd:RDD[String]):RDD[Boolean]={
     val f= SearchFunctions.isMatch(_:String)
     val r=true
     rdd.map(x=>f(x))
    }
}



object SearchFunctions{

  def isMatch(s:String):Boolean =true

}
```

Further Problems Spark-Closures

Example Sentiment Analysis:

dataRDD.map(x=> calculateSentimentValues(x,sentimentVals))

- How is it handled when big static data sets are transfered to the worker? How is it handled when they are used multiple times?
- How is it handled when special events should have a call back to the driver (for example when a text paragraph is below a specific treshold)

➢ **Broadcast-Variables**

➢ **Accumulators**

Broadcast-Variablen

- Transfers „Read-Only“-Data to all worker efficiently
- Are stored at all works for multiple usages
- Could be used for example for big lookup tables

Usage of Broadcast-Variables

Example Sentiment Analysis:

val sentimentVals:Map[String,Double]= loadDictionary(...)

// Definition of a Broadcast-Variable:

val sentimentValsBroadcast= **sc.broadcast(sentimentVals)**

// Übergabe der Broadcast-Variable

dataRDD.map(x=> calculateSentimentValues(x,**sentimentValsBroadcast**))

def calculateSentimentValues(line:String,
sentimentDict**:Broadcast[Map[String,Double]]**):Double={

...

val dictionary= **sentimentDict.value**

}

Direction of Closures

Example in Scala:

```
val l=List(1,2,3,4,5,6,7,8,9,10)
var counter=0
l.foreach(x=>counter=counter+x)
```

→ Counter is 55

Example with Spark

```
val rdd= sc.parallelize(l,8)
var counter=0
rdd.foreach(x=>counter=counter+x)
```

→ Counter is 0

The connection to the driver gets lost. Results will not be propagated.

Usage of Accumulators

```
val l=List(1,2,3,4,5,6,7,8,9,10)
val rdd= sc.parallelize(l,8)
val counteraccu= sc.accumulator(0)
def count(element:Int, counter:Accumulator[Int]):Unit={
   counter += element
}
rdd.foreach(count(_,counteraccu))
```

→ counteraccu ist 55!

- += is a special Operator that need to be defined
- Definition of particular accumulatoren types possible

Properties of Accumulators

- Akkumulatoren could only be used with associative aggregation operations
- Is often used for count and sum operations
- Only the driver can see the values of the accumlator – the tasks can only write to them
- Accumulators could be used in actions and transformations
  - Actions: each update of the Accu is only done once
  - Transformations: No guarantee (only for debugging purposes)
- Types: integers, double, long, float
- Custom type possible

Performance-Optimization

```
val set=Range(1,1000000)

val rdd= sc.parallelize(set,8)

val res= rdd.map(x=>Math.sqrt(x)).filter(x=>(x %2)==0).filter(x=>(x%3)==0)

res.count

res.collect
```

- Code-Fragment contains two Actions: count and collect
- For each action res is calculated
- In this case it would be better to store the result of the calculation

Caching

```
val set=Range(1,1000000)

val rdd= sc.parallelize(set,8)

val res= rdd.map(x=>Math.sqrt(x)).filter(x=>(x %2)==0).filter(x=>(x%3)==0).cache

res.count

res.collect
```

- Cache stores a calculated RDD on the JVM Heap
- It doesn't matter how often an action is performed, it is only calculated once
- If the memory is full the data is released using the LRU (Least Recently Used) principle
- If a node fails the data gets calculated again

Operationen: cache und persist

- The operation cache corresponds to the operation persist(StorageLevel.MEMORY_ONLY)
- Unpersist deletes the RDD from the memory

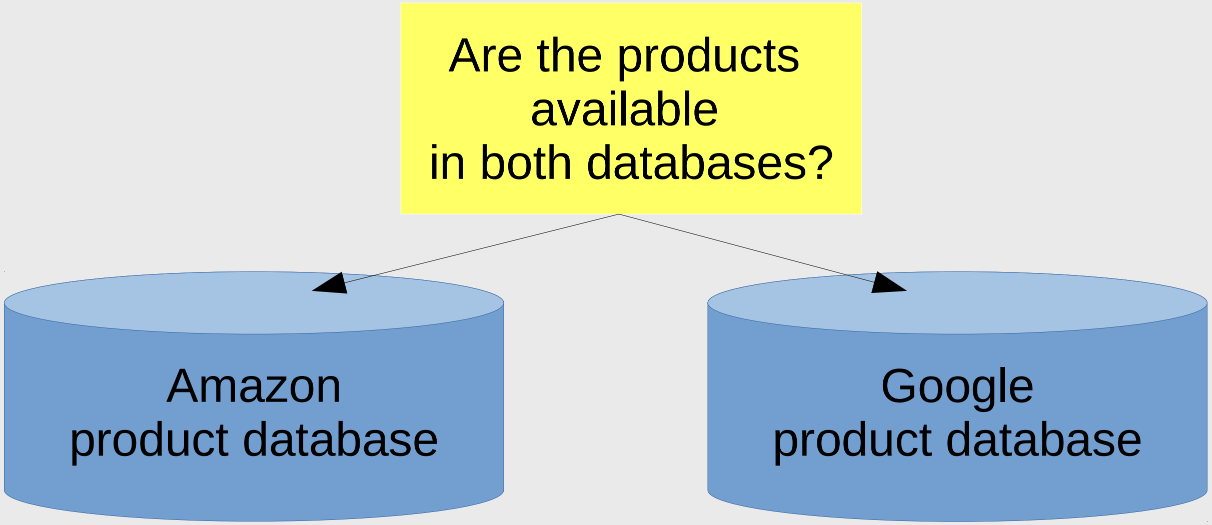
Level	Space Used	CPU time	In Memory	On Disk
MEMORY_ONLY	High	Low	Yes	No
MEMORY_ONLY_SER	Low	High	Yes	No
MEMORY_AND_DISK	High	Medium	Some	Some
MEMORY_AND_DISK_SER	Low	High	Some	Some
DISK_ONLY	Low	High	No	Yes

Entity Resolution

Entity Resolution – Record Linkage

- Record linkage (RL) deals with the task of finding duplicates from different sources
- Record Linkage is required for example if two databases are integrated and there is no mapping between the entitities of the two databases
- Record Linkage is often used for cleaning a big data set (generated for example by a web crawler)
- Plagiat analysis deals with finding copied pieces of a text

Example of Assignment



File format Amazon:

"id","title","description","manufacturer","price"

File format Google:

"id","name","description","manufacturer","price"

What steps are necessary product duplicates?

Entity Resolution – Approach

- Reading and preparing the data
- Implementation of a method to compare the records:
  - Deterministic result?
  - Determination on the basis of propabilities
- Scaling? Should every element need to be compared with all other elements?
- If the approach is based on propabities: How can I determine the quality of my result?

Reading and Preparing data Functions

- Methods to read and parse the data are given (Utils.scala)
  - getData to read
  - ParseLine for parsing of a line
  - TokenizeString to split the words
  - DeleteQuote for deleting double quotes
- Applying the given functions is part of the assignment

Result: For each product a tuple is created following the format:

(ProductID, Text composed of title, description, Manufacturer)

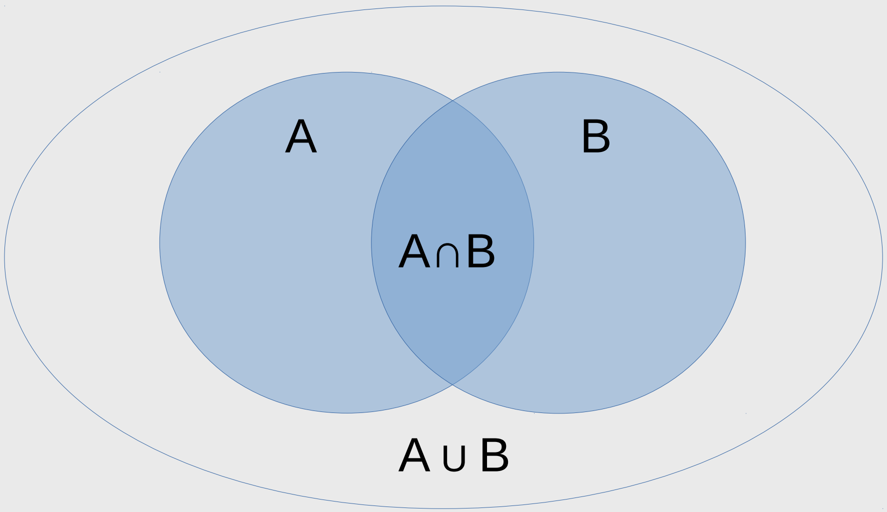
How can the „Similarity“ be assesed on the base of the product information which contains a natural language description of the product?

Distance Measures

- Levensteihn-Distance
- Number of Words? (Product description are about the same size?)
- Number of equal words?
  - Descriptions have different lengths
  - A lot of words without any information (stopwords - to, a, the, from...)
  - How often does a word occur in the description?
  - Which relevance has a word related to all product descriptions

Jaccard Distance

The Jaccard-Distance is a measure for the Similarity of two Sets



$J(A,B) = |A \cap B| / |A \cup B|$

Difference Bag of Words (with duplicates) and Set of Words (without duplicates)

Shingling

In natural language processing a w-shingling is a set of unique shingles (therefore n-grams) each of which is composed of contiguous subsequences of tokens within a document.

A) This is a rose and this is a car.

2-grams (bag): (this,is) (is, a) (a, rose) (rose, and) (this, is) (is, a) (a, car)

2-grams (set): (this,is) (is, a) (a, rose) (rose, and) (a, car)

B) This is a house.

2-grams (bag and set): (this,is) (is, a) (a, house)

Intersection $A \cap B$: (this,is) (is, a)

Union $A \cup B$ Bag: (this,is) (is, a) (a, rose) (rose, and) (this, is) (is, a) (a, car) (this,is) (is, a) (a, house)

Union $A \cup B$ Set: (this,is) (is, a) (a, rose) (rose, and) (a, car) (a, house)

Set: J(A,B)= 2/6   Bag: J(A,B)= 2/10

TF-IDF-Approach

TF-IDF-Idea

- Every document could be represented as a **vector**
- The dimension of the vector corresponds to the size of the **corpus**
- The corpus covers all words occuring in all documents
- Each component of the document vector is either zero if the word is not present in the document or a value greater than zero (the TF-IDF-value) if the word is at least once in the document
- The TF-IDF-value represents not only the occurence of the word but also the importance of a word
- The **term frequency** (TF) expresses the relative frequency of a word related to the text of a document
- The **inverse document frequency** asseses the relevance of a word related to the whole data set.

Vector-Poperties and Operations

- **Length of a vector** (L_2-Norm): |x| is the L_2-Norm of a vector that is calculated by sqrt($\Sigma\ a_i^2$)
- Dot-Product:

$$\mathbf{A}\cdot\mathbf{B}=\sum_{i=1}^{n}A_iB_i=A_1B_1+A_2B_2+\cdots+A_nB_n$$

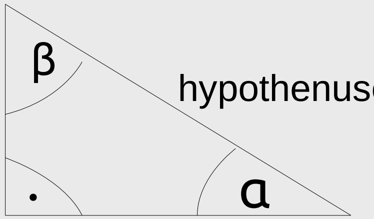
  - Commutative: (r.s) = (s.r)
  - Distributive: r.(s+t)=r.s+r.t
  - Associative over Scalar Multiplication: r.(a*s)=a*(r.s)
- Relation length of a vector to Dot-Product:
  (r.r)= $r_1*r_1+r_2*r_2+...r_n*r_n= r_1^2+r_2^2+...+r_n^2$
  $= ||r||^2$

How would you describe the information the value of the dot-product provides?

Cosine and Cosine-Rule

- Cosine in a right-angled triangle:

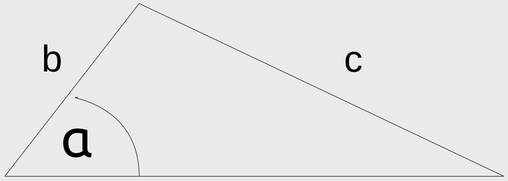
  cos ɑ= adjacent/hypothenuse= a/c



- β is calulacted by changing opposite and adjacent

- Cosine-Rule for abitrary triangles:

  $c^2 = a^2 + b^2 - 2ab \cos \alpha$



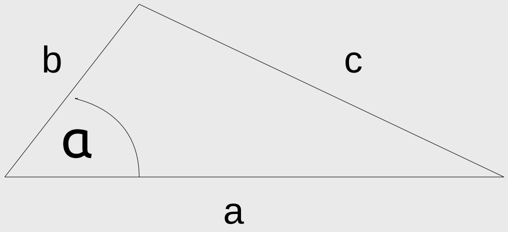
Special case right angled triangle:

$\alpha = 90°$ $\quad\quad$ $\cos 90° = 0$

Cosine-Rule with vectors

Cosine rule:

- Cosine-Rule for abitrary triangles:



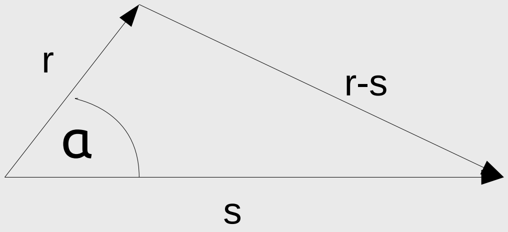
$c^2 = a^2 + b^2 - 2ab \cos \alpha$

$|r-s|^2 = |r|^2 + |s|^2 - 2|r||s| \cos \alpha$

$|r-s|^2 = (r-s).(r-s) = (r.r)-(r.s)-(s.r)+(s.s) =$

$|r|^2 - 2(r.s) + |s|^2$

Triangle described by vectors:



$\rightarrow |r|^2 - 2(r.s) + |s|^2 = |r|^2 + |s|^2 - 2|r||s| \cos \alpha$ (first equation)

$-2(r.s) = -2|r||s| \cos \alpha$

$(r.s) = |r||s| \cos \alpha$

$\mathbf{\cos \alpha = (r.s)/|r||s|}$

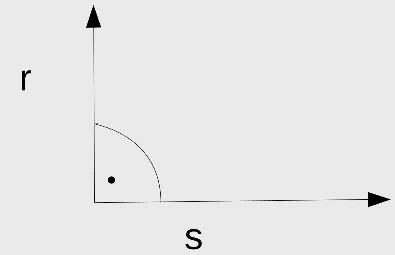
Cosine-Rule Intuitively (1/2)

- Case vectors are orthogonal:

  $\alpha=90^\circ \quad \cos\alpha=0$

  $(r.s)=|r||s|\ 0$

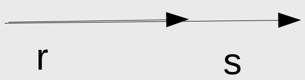
  The dot-Product is zero



- Case vectors pointing into the same direction:

  $\alpha=90^\circ \quad \cos\alpha=0$

  $(r.s)=|r||s|\ 1$

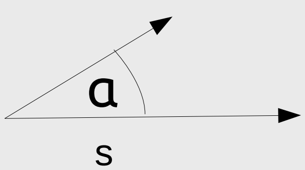


r

- Case vectors pointing into the same direction:

  $\alpha=45^\circ \quad \cos\alpha=0.707...$

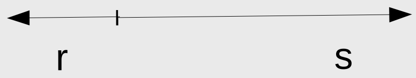
  $(r.s)=|r||s|\ 0.707...$



Cosine-Rule Intuitively (2/2)

- Case vectors pointing into the opposite direction:

  α=180°   cos α=-1

  (r.s)=|r||s| -1

- Applying the cosine rule to compare document vectors:

  The more words are in both documents the merrier the document vectors are pointing into same direction.

- All components of teh document vectors are positive (negative occurences of words are not possible, so the cosine similarity is a value between zero and one

TF-IDF-Approach (1/2)

Step 1: Deleting all stopwords (words without any information) from the product descriptions

Step 2: Calculating the term frequency for each single word in every description:

The term frequency expresses the relative frequency of a word related to the text.

Example:

„This is a text and that is also a Text“

The text contains 10 words – the term frequencies are: „this“: 0.1, „is“:0.2, „a“:0.2, „that“:0.1, „also“:0.1, „text:“ 0.2, „and“:0.1

TF-IDF Approach (2/2)

Step 3: Calculation of the Inverse Document Frequency.

The Inverse Document Frequency asseses the relevance of a word related to the whole data set.

Example:
Product 1: „Software Microsoft Word Text“

Produkt 2: „Software OpenOffice Text“

Produkt 3: „Software World of Warcraft – really super super super Game“

Produkt 4: „MS Word Textverarbeitung – also super“

- Determination of all occuring words (without duplicates): Set(Software, Microsoft, Text, Word, super,…)
- Determination of the number of occurences of a word in all products (frequency is thereby irrelevant – e.g. super:2, Text: 3, OpenOffice:1,…)
- IDF is the number of documents divided by the number of occurences
- TF-IDF is the term frequency multiplied by the idf value

Application of the Approach in the Assignment

1) Calculation of the term frequency for each product:
   For each product a Vector should be calculated that contains the term frequency

2) Calculation of an IDF-Dictionary:
   1) Merge the Amazon and Google product RDDs
   2) Determination of all occuring words in the whole corpus (eliminate all duplicates)
   3) Calculate for each word , in wie viel Produkten es vorkommt
   4) Berechnung des IDF-Wertes für jedes Wort (Anzahl Dokumente/Anzahl Vorkommen)

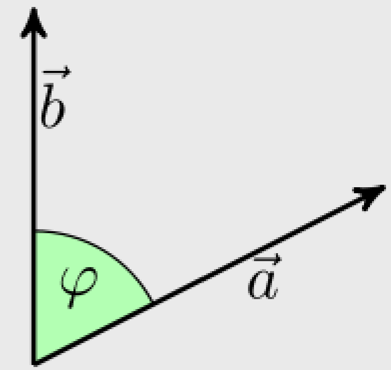
- Calculation of the TF-IDF-values for each product:
  TF-IDF= TF * IDF

Result: Each product consists of a document vector with TF-IDF-values

Cosinus Similarity (Kosinus Ähnlichkeit)

$$\text{Kosinus-Ähnlichkeit} = \cos(\theta) = \frac{a \cdot b}{\|a\|\|b\|} = \frac{\sum_{i=1}^{n} a_i \cdot b_i}{\sqrt{\sum_{i=1}^{n} (a_i)^2} \cdot \sqrt{\sum_{i=1}^{n} (b_i)^2}}$$

- a and b are document vectors
- a*b is the so called dot-Produkt (scalar produkt)



- $$\vec{a} \cdot \vec{b} = |\vec{a}|\,|\vec{b}|\,\cos\sphericalangle(\vec{a}, \vec{b}).$$
- ||x|| is the L_2-Norm of a vector that is calculated by sqrt(Σ a_i^2)
- The cosinus is the ankle between the vectors
- The algebraic definition of the scalar produkt is:

$$\mathbf{A} \cdot \mathbf{B} = \sum_{i=1}^{n} A_i B_i = A_1 B_1 + A_2 B_2 + \cdots + A_n B_n$$

Applying the cosinus-Similarity

- Calculation of the L_2-vector Norms
- Calculation of their scalar produkts
- Creation of all possible product combinations from the Google and the Amazon data set (cartesian product)
- Calculation of the cosinus similatirity for all product pairs

Evaluation:

What is the best threshold that marks the most duplicates?

How „good“ does this method work?

Evaluation with the Gold-Standard

- Gold Standard contains product links
- Evaluation using the gold standard:
  - Choosing a thresholds
  - Counting of True-Positives, False-Positive, True-Negative, False-Negative

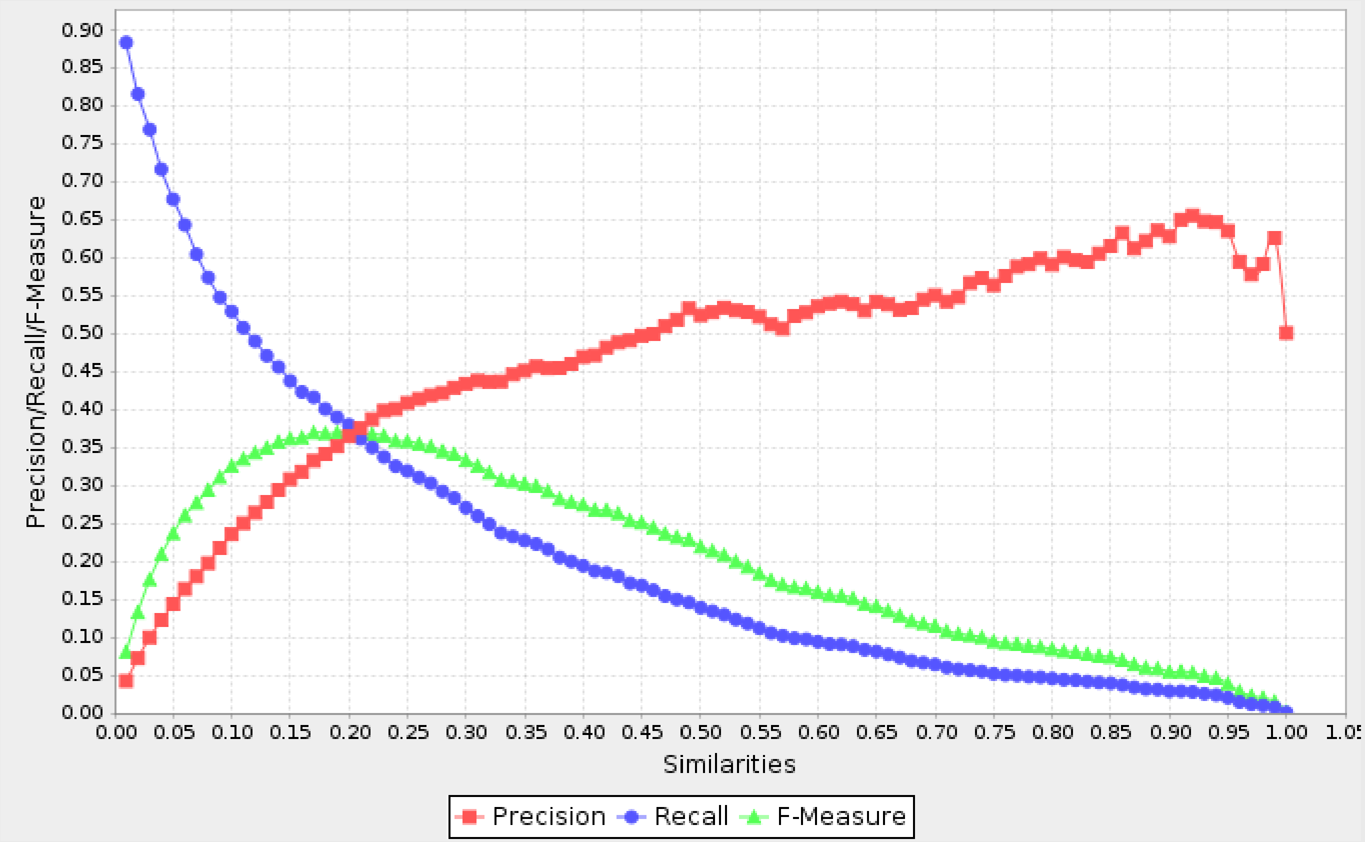
	Fact: Same Product	Fact: Different Product
Forecast: Same Product (Test Positive)	True-Positive	False-Positive
Forecast: Different Product (Test Negative)	False-Negative	True-Negative

Characteristic Numbers for Evaluting the Quality

- Simple Measure:
  Accuracy= Correct Result/total number of results
- Measures often problematic: All elements False-Postives and False-Negatives are weighted equally

  Better:
- **Precision** = true-positives / (true-positives + false-positives) (Genauigkeit)
- **Recall** = true-positives / (true-positives + false-negatives) (Trefferquote)
- **F-measure** = 2 x Recall x Precision / (Recall + Precision) (Harmonische Mittel)

Result of the Analysis – If functions work correctly ;-)



Functions for the visualisations are given.

Part 2 - Scaling of the Algorithm

- What happens if the data set grows?
- Number of product combinations depends on the number of elements of the data sets (|Amazon|*|Google|)
- Could grow quickly

Questions:

How could we reduce the number of combinations?

How could we increase the efficiency of the implementation?

Scaling of the Algorithm

- Using Broadcast Variablen
- Creation of an Inverse Indexe (Wort → DokID)
  - Idea: Consider only product combination that have at least one identical token
  - Calculation of the cosinus similarity for pairs that have at least one identical token
  - Use only common tokens for the calculation of the scalar product

Vielen Dank für

Ihre Aufmerksamkeit

Prof. Dr.-Ing. Hendrik Gärtner