Revision Notes

Artem Vasenin

Last updated April 11, 2017

Contents

Chapter 1

Information Retrieval

1.1 Boolean Retrieval

1.1.1 Index

Any query is posed as a boolean expression of terms. First we create an index of words that occur in each text, this is done offline. E.g.

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello
Antony	1	1	0	0	0
Brutus	1	1	0	1	0
Caesar	1	1	0	1	1
Calpurnia	0	1	0	0	0
Cleopatra	1	0	0	0	0
mercy	1	0	1	1	1
worser	1	0	1	1	1

When we compute results of a query we first invert corresponding to NOT terms and then compute a bitwise AND and OR between vectors for each term. A few observations:

- The matrix is very sparse.
- Doesn't support more complex operations, such as proximity search.

1.1.2 Inverted Index

To make storing information more efficient we can use inverted matrix which only documents things that occur. The inverted index is a dictionary, with terms for keys and *posting lists* for values. A posting list is a sorted list which documents which documents the term occurs in.

Now to compute an AND query we compute and intersection of posting lists. This can be done in $O(n)$ where n is the number of documents in collection, in practice much faster.

When computing a conjunctive query with more than two terms we process them based on length of posting lists, in ascending order. For disjunctive terms we can estimate the size of result using the sum of sizes of disjuncts.

1.2 Indexing

The major steps in inverted index constructions are:

1. Collect the documents to be indexed.
2. Tokenize the text.
3. Perform linguistic preprocessing of tokens.
4. Index the documents that each term occurs in.

1.2.1 Skip lists

When we construct an inverted index we can use either a linked list or a variable length array. In the case we use a linked list there is a natural optimisation we can perform. To create a skip list we can link together every n elements. This allows us to compare to lists quicker.

Size of n, number of elements skipped, presents a trade-off: number of items skipped vs. frequency that skip can be taken. Usually $n = \sqrt{L}$ where L is the length of the list.

N.B. Skip lists used to help a lot, but with today's fast CPUs they don't help that much anymore.

1.2.2 SPIMI

During indexing large collections we can't keep and sort all postings in-memory. We cannot sort very large records on disk either (too many disk seeks, expensive). We can use **Block-Based** sorting algorithm. Two key ideas:

- Generate separate dictionaries for each block.
- Accumulate postings in posting lists as they occur.

Using those two ideas we can generate complete invert index for each block and then merge blocks at the end.

1.2.3 Document and term normalisation

A lot of words used in documents have similar meaning and should be tokenised the same, e.g. multiples, capitalisation, misspelling, etc. Most of the time we also need to prepare a document before it can be tokenised, we need to consider:

- Compression and binary representation.
- Format (excel, pdf, latex, etc.)
- Character set
- Language the document is written in.

Each of these is a statistical classification problem. Also deciding what is a "document" is not trivial.

Tokenisation It is frequently difficult to decide how to tokenise the text. Splitting text into words is non trivial, so is parsing numbers. Text can use different scripts (Japanese) and can be written in different directions (Arabic).

There are several useful techniques:

Case folding Reduce the text to lowercase.

Stop words Frequent words that don't matter can be excluded (a, the, an, etc).

Equivalence classing Words with the same meaning should result in the same token (car = automobile).

Lemmatisation Reduce inflections, derivations to base form (am, are, is → be).

Stemming Chopping the end of word off (cheaper lemmatisation).

1.2.4 Phrase Queries

Need to answer some queries as a phrase, e.g. "Cambridge University", but sentences line "The Duke of Cambridge recently went for a term-long course to a famous university" should not match. Therefore it is no longer sufficient to store docIDs in posting lists. Two ways to extend the inverted index:

- Biword index
- Positional index

Biword index Index every consecutive pair of terms in the documents as a phrase. Two-word phrases can now easily be answered. For longer phrases use a conjunction, e.g. "cambridge university west campus" becomes "cambridge university AND university west AND west campus". Need to do post filtering to check whether the whole phrase is actually used. **Issues:** a lot of false positives and index blowup.

Positional index In addition to storing the docID, also store position in the document. We can now check phrases using position. It can also be used for proximity search.

N.B. We want to return the actual matching position, not just the document.

Combination Many biwords are extremely frequent (Michael Jackson, Los Angeles). For such biwords using a simple positional index is slow. Therefore to improve performance we can include frequent biwords as vocabulary terms.

1.2.5 Term Vocabulary

Can we estimate how many distinct words are used in a collection? Yes we can, *Heap's Law*: $M = kT^b$ where M is the size of vocabulary, T is the number of tokens in the collection. Typical values for k and b are : $30 \leq k \leq 100$ and $b \approx 0.5$. Heaps' Law is an empirical law.

Chapter 2

Bioinformatics

2.1 Biology background

Structure of DNA DNA is structured as a double helix. Each helix is a sequence of nucleotides, there are four types of nucleotides: A,C,T and G. The sequences are aligned with each other such that at each position the nucleotide on one helix is paired with nucleotide on the other helix in one of the four possibilities: A-T, T-A, C-G or G-C.

Structure of proteins DNA uses a triplet code, triplet of nucleotides is called a "codon". Each codon is a code for a specific amino acid. When multiple amino acids join together they form a protein. There are 20 types of amino acids which can be used to form a protein. Some codons are used for punctuation. A gene is a sequence of codons between start and stop codons. Each gene is responsible for one protein.

DNA - Protein translation To copy DNA, it is first untwisted and unzipped inside the nucleus. Then free RNA nucleotides form complementary base pairs with one of the DNA strands. RNA uses a U nucleotide instead of DNA's T, so they can be paired as follows: A-U, T-A, C-G and G-C; DNA on the left, RNA on the right. When multiple RNA nucleotides bond next to each other, weak hydrogen bons form between those nucleotides. mRNA transcribes amino acid code using codons. mRNA strand is then synthesised from the sequence of RNA nucleotides. mRNA strand peels off the DNA and moves out of the nucleus into the cytoplasm.

Translation takes place on the ribosome in the cytoplasm. The ribosomes are sites of protein synthesis. mRNA strand attaches to a ribosome and tRNA molecules transport specific amino acids to the ribosome. The anti-codons and codons match up to form complementary base pairs. Peptide bonds form between the adjacent amino acids to form the polypeptide, which is a protein.

2.2 Sequence Alignment

The goal of the alignment algorithms is to find the longest common subsequence of two sequences. Alignment of n sequences is an n-row matrix, where nth row represents the symbols of nth sequence (in order) interspersed by "-". Corresponding symbols in different sequences can have four different relations to each other:

Match The symbols are the same.

Mismatch The symbols are different.

Insertion "Primary" symbol is "-".

Deletion "Secondary" symbol is "-".

Matches in alignment of two sequences form "common subsequence".

Global Alignment We can represent the alignment search space as an edge-weighted directed acyclic graph. We can construct such DAC by creating a rectangular grid of nodes, with source in the top-left and destination in the bottom-right. Then we add edges, of which there are three types:

Match/Mismatch Connects neighbouring nodes diagonally from top-left to bottom-right.

Insertion Connects neighbouring nodes horizontally from left to right.

Deletion Connects neighbouring nodes vertically from top to bottom.

Global alignment can be found by finding the longest path from the source to the destination. To find the longest path we start at the source and calculate the length of path to all its children. To do that we

consider all incoming edges and their source nodes. The longest path will go through the edge which has the greatest sum of its cost and path cost to its source node. Once we calculate which edge the longest path goes through that's the only information we need to store. Therefore we can iteratively calculate path length to all nodes. Finally to calculate the longest path from source to destination we start at the destination and iteratively move backwards along the stored edges until we arrive at the source.

Parametrisation We can parametrise the longest path by assigning different weights to different types of edges. Match/mismatch edges can be weighted by considering how likely it is for the codon in primary sequence to mutate into codon in the secondary sequence. These weights are usually calculated empirically. Insertions and deletions on the other hand depend only on their length. Combined insertions/deletion should usually have lower penalty that separate ones. There are two costs: cost to begin insertion/deletion and cost to extend it. Our DAC can be extended to represent that by adding vertical and horizontal edges that go across multiple nodes.

Local alignment Local alignment refers to a long sequence of uninterrupted matches. This corresponds to the idea that two sequences should not align perfectly, but rather only their ends should align. We can compute local alignment by computing global alignment in sub-rectangle of the graph. To do this we add a zero-weight edge between top-left node in the sub-rectangle and source and another between bottom-right node and destination.

Improving space complexity Since DNA sequences are very long, we frequently run of working space. Finding the longest path has $O(nm)$ space complexity where n and m are lengths of sequences to align. We can improve space complexity to $O(n)$ by sacrificing time. This is possible since to we can compute the middle node using only $O(n)$ space by considering each column in turn. Since the results of column i only rely on the results of column $i-1$, we only need to store at most two columns in memory. If we compute scores for middle column coming from both sides, we can find the middle node. After we found the middle node, we divide the graph into four rectangles and then recur on top-left and bottom-right rectangles. By doing that we improve space

complexity to $O(n)$, since we only ever finding the middle node, while keeping time complexity the same, but increasing it by a constant.

BLAST BLAST stands for Basic Local Alignment Search Tool. It is a program designed for comparing protein sequences against a large database to find local similarity. It uses a hash table to store all possible sub-sequences of a given length in the database. BLAST relies on assumption that we only care for near perfect alignments >95%, therefore we expect to see long runs of perfectly matching sub-sequences. It uses a heuristic to find short matches between sequences and then uses those to start the alignment.

2.3 Clustering

Clustering combines elements that are close to each other into the same cluster. Since we expect similar sequences to have the same ancestor, we can use clustering to build the evolutionary tree.

2.3.1 Types of clustering algorithms

There are four main types of clustering algorithms:

- Hierarchical
- k-means
- Distribution based
- Density based

Hierarchical clustering This approach clusters points based on the distance between them. It works as follows:

1. Consider each node as a cluster
2. Calculate distances between all clusters
3. Find the smallest distance and combine corresponding clusters into one, using weighted average for new position.
4. Repeat previous two steps until there is only one cluster left.

In parallel to merging clusters we create a leaf for every cluster at the start. When two clusters are merged we create a new vertex and connect it to the vertices which represent the clusters which were combined. After clustering is complete we can separate our set into n clusters by undoing n merges in reverse order.

k-means clustering This approach combines nodes into k clusters, it works as follows:

1. Assign k arbitrary centres.
2. Assign each node to a cluster based on the closest centre.
3. Calculate the centre of gravity of each cluster and move its centre to the node which is closest to the centre of gravity.
4. Repeat previous two steps until centres stop moving.

Optimal number of clusters (k) has high cluster stability, e.g. it doesn't matter which nodes you choose as initial centres, final clusters will always be the same. k-means is very fast as well.

Distribution based clustering

Density based clustering

2.3.2 Clustering vs. Classification

Clustering is unsupervised and should be used when we want to find if there are any relationships between the data points. Classification on the other hand is supervised and should be used when we want to separate the data into predetermined classes.

2.3.3 Distance and Similarity

Distance and similarity refer to what percentage of symbols mismatch between two data-sets. Distance metrics must obey triangle inequality. E.g. if we have two biased coins and we flip them 100 times we can count how many times each coin landed heads up. Based on how close the two counts are, we can infer that coins are either similarly weighted or not.

This is relevant because we can infer from how objects behave how similar they are. Similar things usually are genetically closer to each other than distant ones. Therefore we can construct evolution tree using similarities.

2.4 Markov models

Markov property The future only depends on the present and not on the past.

Markov chain is a stochastic process that satisfies markov property. Hidden markov model is a model which is assumed to be a markov chain with unobserved states.

2.4.1 Markov-clustering (MCL)

MCL assumes that dense cluster correspond to regions with a large number of paths. It takes an undirected graph as an input and alternatively expands the matrix (multiplying it by itself) and inflating it (element wise multiplication). It's a good algorithm to use if we don't know how many clusters exist in the data.