# Assignment 3 — DIRT: Extraction of Lexico-syntactic Similarities

## Design Report

### 1. Goal and Outputs

This system implements the main DIRT pipeline (Lin & Pantel) over the Google Syntactic N-grams "Biarcs" corpus and evaluates similarity between predicate templates from a provided test set.

**System outputs**

1. **MI feature table**: `mi.tsv` containing `p \t slot \t w \t mi(p,slot,w)` for every discovered `(predicate, slot, filler)` triple.
2. **Similarity scores for the test set**: `out_scores.tsv` containing `label \t p1 \t p2 \t score`.

### 2. Input

**Input corpus**: Biarcs records (Google Syntactic N-grams). Each record corresponds to a dependency fragment with token information and an aggregated count.

**Test set**

- `positive-preds.txt` — tab-separated predicate pairs labeled positive (entails).
- `negative-preds.txt` — tab-separated predicate pairs labeled negative (not-entails).

### 3. Predicate/Path Extraction (Local per record)

Component: `com.dsp.dirt.extract.PathExtractor`

Given one parsed Biarcs record, we:

1. Reconstruct token array `tok[index]` and a `parent[]` pointer array (head indices).
2. Collect candidate **noun tokens** (`NN*`) as potential **X/Y slot fillers**.
3. For every noun pair `(a,b)`:
    - Compute `l = LCA(a,b)` in the dependency tree.
    - **Constraint**: the path head must be a **verb** (`VB*`).
    - **Auxiliary filtering**: skip auxiliary verb heads (e.g., *is/are/was/have/do*).
    - Build dependency path in both directions:
        - `a → b` produces predicate with `X=a, Y=b`
        - `b → a` produces predicate with `X=b, Y=a`
4. **Stanford-parser preposition handling constraint**:
    - Unlike MiniPar, Stanford dependencies do not omit prepositions from tree nodes.
    - Therefore the predicate path explicitly **includes prepositions** (`IN`, `TO`) in addition to content POS:
        - verbs (`VB*`), adjectives (`JJ*`), adverbs (`RB*`), particles (`RP`), and prepositions (`IN`, `TO`).

**Stemming requirement**

- Tokens are not stemmed in the corpus; test predicates are in base form.

- We apply a Porter stemmer wrapper (`com.dsp.dirt.extract.Stemmer`) to normalize fillers and predicate words.

**Per extracted instance** We emit `PathInstance(pathKey, slot, filler, count)` where:

- `pathKey` is a space-separated template like: `X accompani by Y`
- `slot ∈ {X,Y}`
- `filler` is the stemmed noun at that slot
- `count` is the record count

## 4. MapReduce Pipeline

### Job A: Triple Counts

Component: `com.dsp.dirt.job.TripleCountsJob`

**Mapper input**: raw Biarcs lines
**Mapper output**: four key types, each emitted with the instance count.

Key encodings (`Text` keys; all tab-separated):

1. `T\tp\tslot\tw` meaning `|p,slot,w|`
2. `PS\tp\tslot` meaning `|p,slot,*|`
3. `SW\tslot\tw` meaning `|*,slot,w|`
4. `S\tslot\t*` meaning `|*,slot,*|`

**Reducer/Combiner**: sums counts per key.

**Global total for MI** We also maintain a counter:

- `DIRT:TOTAL_T` = sum of counts of all emitted `T` events (i.e., `|*,*,*|` over triples)

This counter is later injected into Job B via configuration under:

- `DIRT_TOTAL_T`

**Scale and memory notes (qualitative)**

- Mapper emits up to `4 * (#PathInstances)` key-value pairs per input record.
- Reducer is a streaming sum; memory usage is O(1) per key (plus Hadoop overhead).

### Job B: MI Computation

Component: `com.dsp.dirt.job.ComputeMIJob`

**Job A output input format**: lines of `KEY \t COUNT`

**Mapper**

- Partitions by `slot` so each reducer sees all counts needed for that slot.
- Emits `slot` as the reducer key and a tagged value:
  - `S\tcount`
  - `SW\tw\tcount`

    - `PS\tp\tcount`
    - `T\tp\tw\tcount`

**Reducer logic** For each `slot`:

- Load:
    - `c_ps[p] = |p,slot,*|`
    - `c_sw[w] = |*,slot,w|`
    - and stream over `T` triples `|p,slot,w|`
- Use global `totalT = |*,*,*|` from `DIRT_TOTAL_T`.

Compute: [ MI(p,slot,w) = \log \frac{|p,slot,w|\cdot |,,|}{|p,slot,|\cdot |*,slot,w|} ]

Emit:

- key: `p \t slot \t w`
- value: `mi` (Double)

**Reducer memory considerations**

- For each slot we hold two hash maps:
    - `c_ps`: number of predicates in that slot
    - `c_sw`: number of fillers in that slot
- This is the dominant reducer memory cost. In practice slot ∈ {X,Y}, so data is split into two partitions.

## 5. Local Similarity Scoring

Component: `com.dsp.dirt.local.EvalPairs` + `SimilarityScorer`

Inputs:

- `mi.tsv` (merged from Job B output)
- `positive-preds.txt`, `negative-preds.txt`

Steps:

1. Normalize predicates using the same stemming logic used in extraction.
2. Load MI features for predicates that appear in the test pairs.
3. Compute DIRT path similarity as: [ sim(p,q) = \sqrt{ sim_X(p,q)\cdot sim_Y(p,q) } ] Slot similarity uses MI-weighted overlap (implemented in `SimilarityScorer.slotSimilarity`).

Output:

- `out_scores.tsv`: `label \t p1 \t p2 \t score`

## 6. Large (100 files) Placeholder

The same pipeline is executed with a larger input set. Differences expected:

- More predicate coverage (fewer “missing predicate” cases)
- More non-zero overlaps → higher dynamic range in similarity scores
- Potentially more false positives (spurious overlap) due to increased corpus breadth