# Relation extraction

Bill MacCartney
CS224u
Stanford University

## Evaluation

# Overview

- ~~The task of relation extraction~~
- ~~Data resources~~
- ~~Problem formulation~~
- Evaluation
- Simple baselines
- Directions to explore

# Evaluation

- Test-driven development
- Splitting the data
- Precision and recall
- F-measure
- Micro-averaging and macro-averaging
- Figure of merit

# Test-driven development

Good software engineering uses *test-driven development*:

First, write unit tests that check whether the code works.

Then, start writing the code, iterating until it passes the tests.

Good model engineering can use a similar paradigm:

First, build a test harness that performs a quantitative evaluation.

Then, start building models, hill-climbing on your evaluation.

# Splitting the data

As usual, we'll want to partition our data into multiple splits:

| | |
|---|---|
| Tiny | 1% |
| Train | 74% |
| Dev | 25% |
| Test | ? |

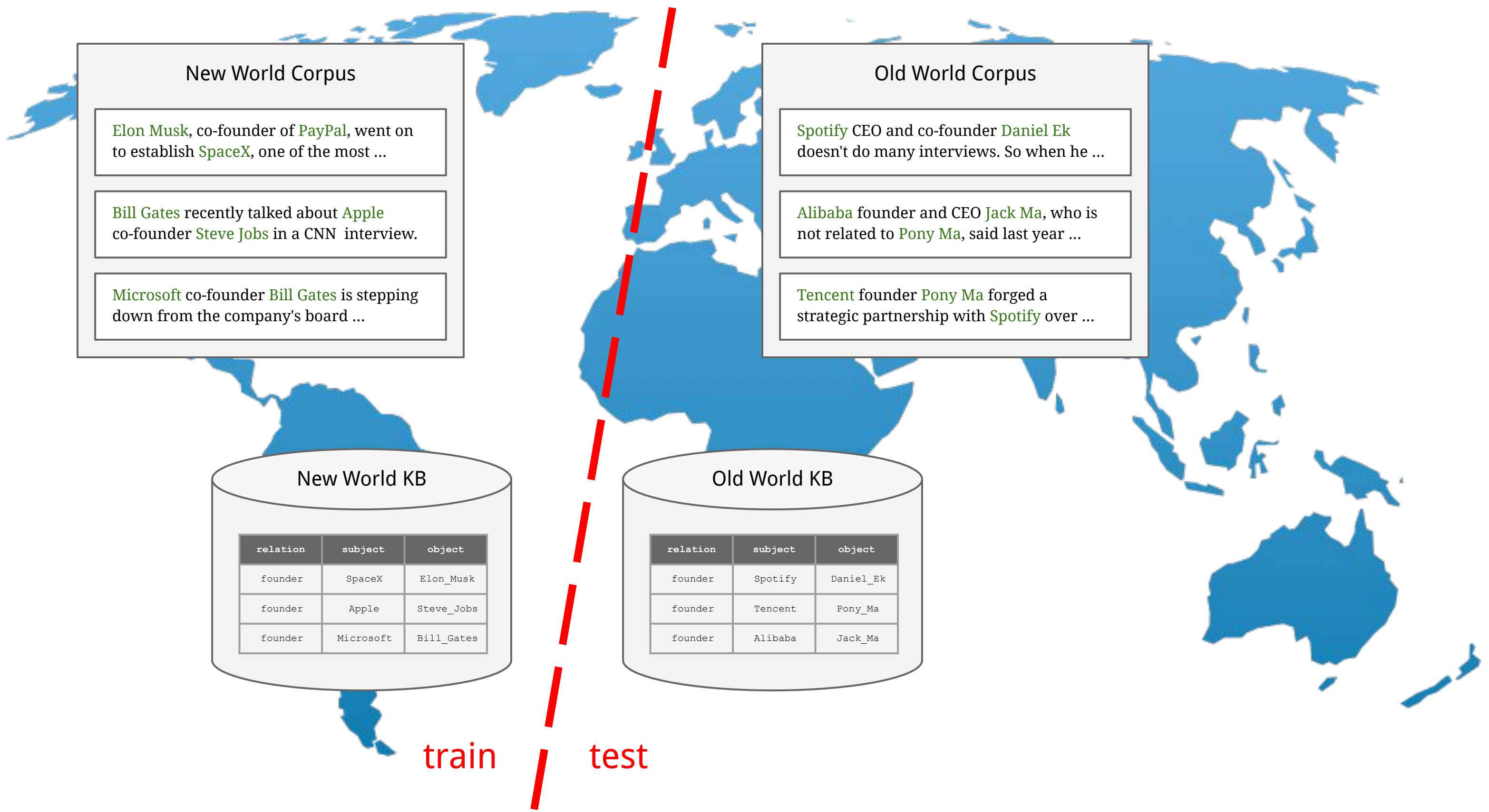
Complication: we need to split both corpus and KB.

We want relations to span splits, so that we can assess our success in learning how a given relation is expressed in natural language.

But ideally, we'd like the splits to *partition* the entities, to avoid leaks.

Evaluation

# Splitting the data: the ideal




New World KB

| relation | subject | object |
|---|---|---|
| founder | SpaceX | Elon_Musk |
| founder | Apple | Steve_Jobs |
| founder | Microsoft | Bill_Gates |

Old World KB

| relation | subject | object |
|---|---|---|
| founder | Spotify | Daniel_Ek |
| founder | Tencent | Pony_Ma |
| founder | Alibaba | Jack_Ma |

# Splitting the data: the achievable

But the world is strongly entangled, and the ideal is hard to achieve.

Instead, we’ll approximate the ideal:

- First, split KB triples by subject entity.
- Then, split corpus examples:
    - If entity_1 is in a split, assign example to that split.
    - Or, if entity_2 is in a split, assign example to that split.
    - Otherwise, assign example to split randomly.

# Splitting the data: `build_splits()`

```
splits = dataset.build_splits(
    split_names=['tiny', 'train', 'dev'],
    split_fracs=[0.01, 0.74, 0.25],
    seed=1)

splits
```

```
{'tiny': Corpus with 3,474 examples; KB with 445 triples,
 'train': Corpus with 249,003 examples; KB with 34,229 triples,
 'dev': Corpus with 79,219 examples; KB with 11,210 triples,
 'all': Corpus with 331,696 examples; KB with 45,884 triples}
```

# Precision and recall

Precision and recall are the standard metrics for binary classification.

| | | predicted | | |
|---|---|---|---|---|
| | | F | T | |
| actual | F | 86 | 2 | 88 |
| | T | 9 | 3 | 12 |
| | | 95 | 5 | 100 |

$$\text{recall} = \frac{3}{12} = 25\%$$

$$\text{accuracy} = \frac{89}{100} = 89\%$$

$$\text{precision} = \frac{3}{5} = 60\%$$

# $F_1$

The $F_1$ score combines precision and recall using the harmonic mean.

| | | predicted | | |
|---|---|---|---|---|
| | | F | T | |
| actual | F | 86 | 2 | 88 |
| | T | 9 | 3 | 12 |
| | | 95 | 5 | 100 |

$$\text{recall} = \frac{3}{12} = 25\%$$

$$\text{precision} = \frac{3}{5} = 60\%$$

$$F_1 = \frac{2}{1/P + 1/R} = 35.3\%$$

# F-measure

F-measure is a weighted combination of precision and recall.

$$F_\beta = \frac{1 + \beta^2}{1/P + \beta^2/R}$$

| | | |
|---|---|---|
| $P$ | 0.800 | high precision |
| $R$ | 0.200 | low recall |
| $F_1$ | 0.320 | equal weight to precision and recall |
| $F_{0.5}$ | 0.500 | more weight to precision |
| $F_2$ | 0.235 | more weight to recall |

For relation extraction, precision probably matters more than recall.
So, let's use $F_{0.5}$ as our evaluation metric.

Evaluation

# Micro-averaging and macro-averaging

Micro-averaging gives equal weight to each problem instance.

Macro-averaging gives equal weight to each relation.

| **relation** | **instances** | **F-score** |
|---|---:|---:|
| adjoins | 100 | 0.700 |
| author | 100 | 0.800 |
| contains | 1000 | 0.900 |
| micro-average | | 0.875 |
| macro-average | | 0.800 |

We'll use macro-averaging, so that we don't overweight large relations.

Evaluation

# Figure of merit

Your “figure of merit” is the one metric — a *single* number — you’re seeking to optimize in your iterative development process.

We’re choosing macro-averaged $F_{0.5}$ as our figure of merit.