Classification
Regularization
Tuning

LING83800: METHODS IN COMPUTATIONAL LINGUISTICS II

May 13, 2024

Spencer Caplan

Today

1. Features in classification
    - Bag-of-words
    - TF-iDF
2. Stemming / Lemmatization
3. Human learning for morphology
4. Tuning
    - Regularization
    - Hyperparameters

Text classification

- An enormous number of tasks in NLP can be framed as a simple machine learning problem in which one assigns exactly one label (from a finite set) to each document
- For instance…
  - Genre classification
  - Abuse detection
  - Sentiment analysis
  - Spam filtering
  - Language ID

Discriminative classifiers (like logistic regression) are highly effective at this task, and only recently have been surpassed by neural networks.

Words as features

- One of the fundamental issues in text classification is turning documents into feature vectors

- Up to now we've sometimes assumed that features are binary-values (thus a feature is either present or absent for a given document)

- For text classification problems, we could simply treat every unique token as a indicator feature.

E.g., if the document contains the token *Noriega*, we have the feature tuple

```
("Noriega", True)
```

Bag of words

- We've also seen a feature representation where we keep track of how often a word occurs in a document, which gives us integer-values features

  E.g., if the document contains the token *Noriega* three times, we have the feature tuple `("Noriega", 3)`

Since this ignores word order, it is sometimes called a bag of words model

Bags of words in scikit-learn (1/2)

- Scikit-learn does not force us to extract and encode these word features ourselves: rather, the process is largely automated by the `CountVectorizer` class.

- This class, once instantiated, can be fed (via `fit` and `transform` methods) sentences or documents or even lists of filenames, and can perform various forms of preprocessing, including…
  - Case-folding
  - “accent stripping”
  - Removal of “stop words”, and
  - A crude form of tokenization

Bags of words in scikit-learn (2/2)

CountVectorizer can also generate higher-order n-gram features and filter the feature vocabulary by

- `min_df=:` *document frequency* (i.e., require that features occur in at least k documents),
- `max_features=:` an overall feature cutoff (i.e., limit itself to the k most frequent features), or
- `vocabulary=:` simply use a pre-computed vocabulary of features.

Visualizing count vectors

Term-Document Matrix

	D1	D2	D3	D4	D5
abandon					
abdicate					
abhor					
academic					
…					
zygodactyl					
zymurgy					

Each column vector represents a Document

Term-Document Matrix

	D1	D2	D3	D4	D5
abandon					
abdicate					
abhor					
academic					
…					
zygodactyl					
zymurgy					

Each row vector represents a Term

Term-Document Matrix

	D1	D2	D3	D4	D5
abandon					
abdicate					
abhor					
academic					
…					
zygodactyl					
zymurgy					

The value in a cell is based on how often that term occurred in that document

Term-Document Matrix

	D1	D2	D3	D4	D5
abandon					
abdicate					
abhor					
academic					
…					
zygodactyl					
zymurgy					

The length of the document vectors is the size of the vocabulary

Term-Document Matrix

	D1	D2	D3	D4	D5
abandon					
abdicate					
abhor					
academic					
…					
zygodactyl					
zymurgy					

Document vectors
can be sparse
(most values are 0)

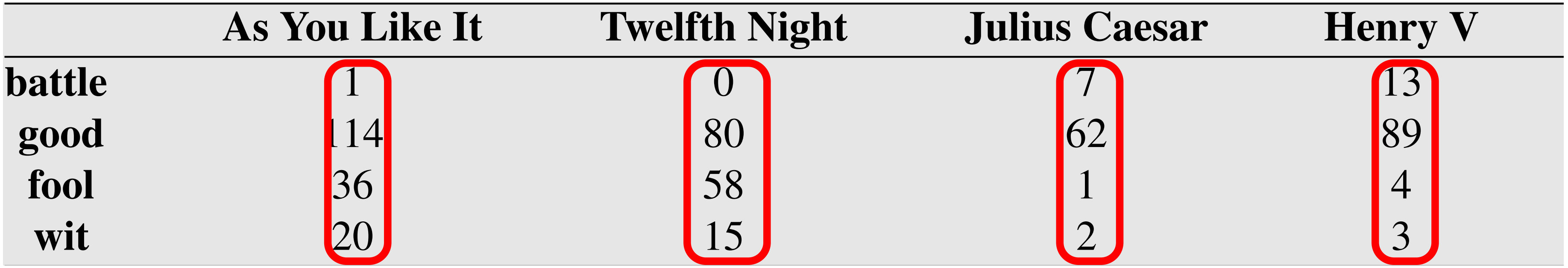
Term-Document Matrix

	D1	D2	D3	D4	D5
abandon					
abdicate					
abhor					
academic					
…					
zygodactyl					
zymurgy					

We can measure
how similar two
documents are
by comparing their
column vectors

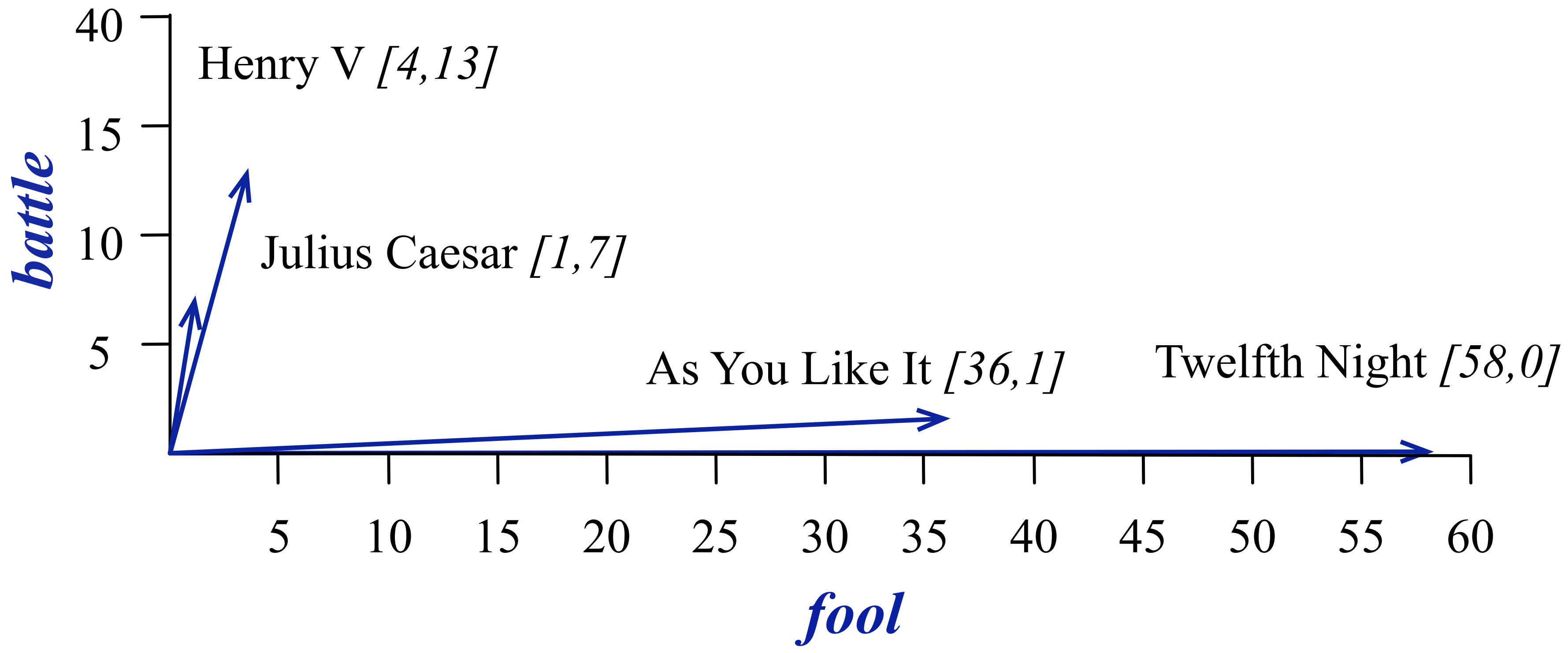
Term-Document Matrix

Each document is represented by a vector of words

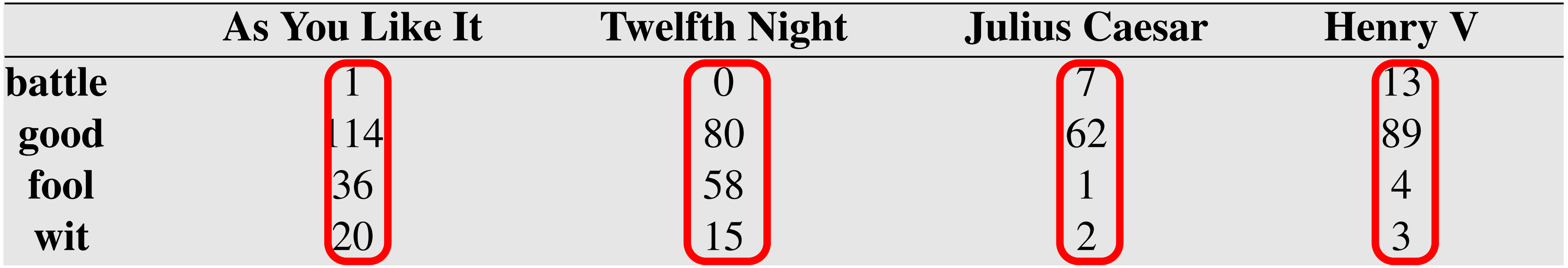


	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Visualizing document vectors



Vectors for information retrieval



	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

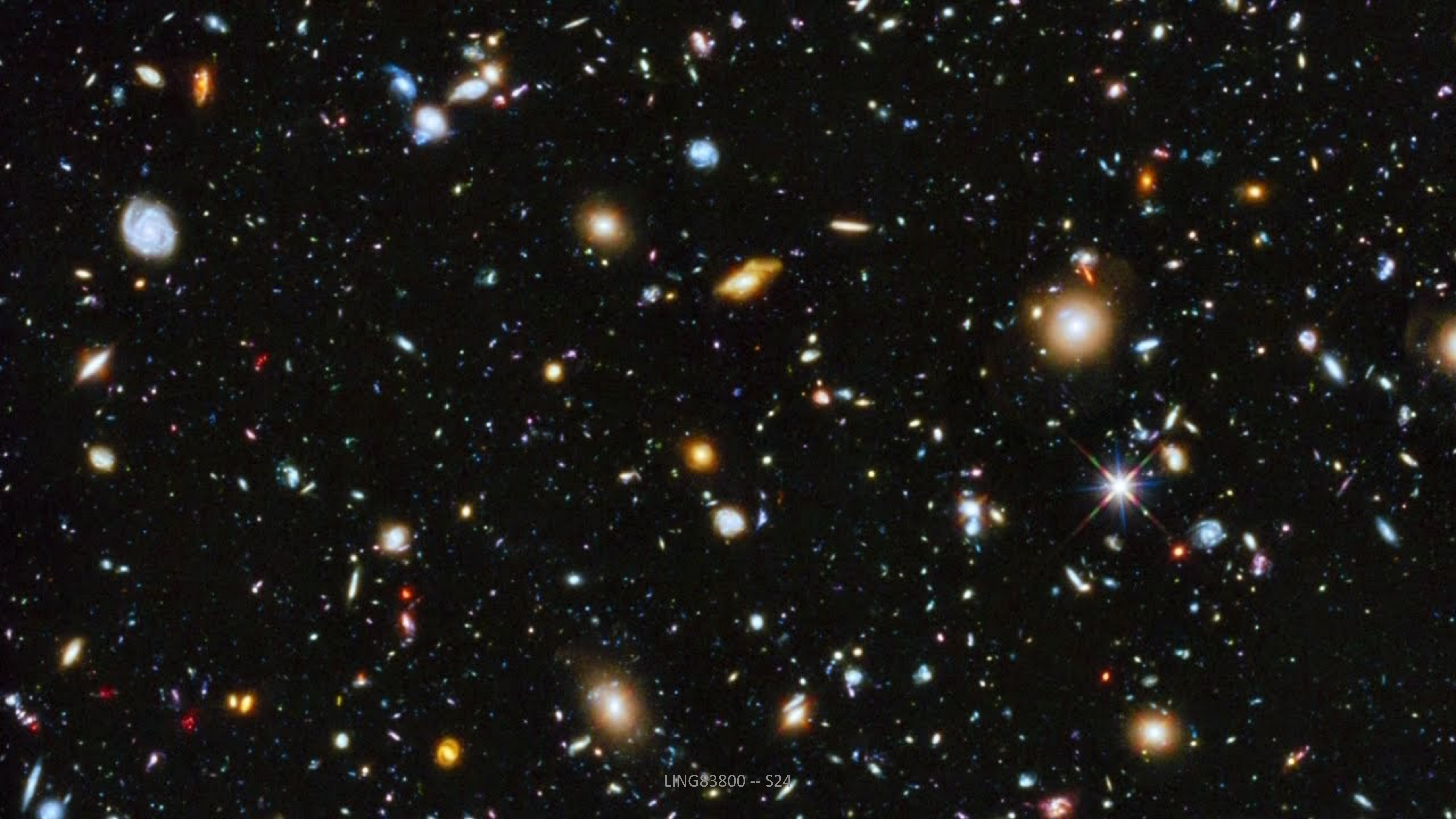
Vect

Diffe

Com

fewe

Right but how many dimensions did we have again?



Problems with using raw frequency

Frequency is clearly useful!

But nearly all dimensions will have a value of 0 for any given document

And the highly frequent words like *the*, *it,* or *they* actually aren't very informative about the content (they'll appear in nearly every document)

Need a function that resolves this frequency "paradox"

Intuitions we’d like to capture

- Raw token refrequency is less informative than we’d like

1. Ubiquitous words tend to carry little information...
   - ...e.g., they may be syntactically required functional elements rather than semantically-rich lexical items themselves.

2. Words that occur in many documents/contexts tend to bear less information than words which occur in few documents/contexts
   - e.g., if a document mentions *Noriega*, it's probably about *Noriega* too.

Tf-idf: combing two factors

- **tf: term frequency**. frequency count (usually log-transformed):

$$\mathrm{tf}_{t,d} = \begin{cases} 1 + \log_{10}\mathrm{count}(t,d) & \text{if } \mathrm{count}(t,d) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Note from the book:
You could also add pseudo-count first instead

Tf-idf: combing two factors

- **tf: term frequency**. frequency count (usually log-transformed):

$$\mathrm{tf}_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{count}(t,d) & \text{if } \mathrm{count}(t,d) > 0 \\ 0 & \text{otherwise} \end{cases}$$

- **Idf: inverse document frequency: tf-**

Total # of docs in collection

$$\mathrm{idf}_i = \log\left(\frac{N}{\mathrm{df}_i}\right)$$

# of docs that have word i

tf-idf value for word t in document d:

$$w_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t$$

Tf-idf: combing two factors

- **tf: term frequency**. frequency count (usually log-transformed):

$$\mathrm{tf}_{t,d} = \begin{cases} 1 + \log_{10} \mathrm{count}(t,d) & \text{if } \mathrm{count}(t,d) > 0 \\ 0 & \text{otherwise} \end{cases}$$

- **Idf: inverse document frequency: tf-**

$$\mathrm{idf}_i = \log\left(\frac{N}{\mathrm{df}_i}\right)$$

Total # of docs in collection

# of docs that have word i

What happens if a word appears in every document?

tf-idf value for word t in document d:

$$w_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t$$

Example (J&M, §6.3)

	As You Like It	**Twelfth Night**	**Julius Caesar**	**Henry V**
battle (id 1)	1	0	**7**	**13**
good (id 2)	114	80	62	89
fool (id 3)	**36**	**58**	1	4
wit (id 4)	**20**	**15**	2	3

Sparse representation: `{"As You Like It": [(1, 1), (2, 114), (3, 36), (4, 20)], ...}`

Example (J&M, §6.3)

$\text{tfidf}_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle (id 1)	.07	.00	**.22**	**.28**
good (id 2)	.00	.00	.00	.00
fool (id 3)	**.02**	**.02**	.00	.01
wit (id 4)	.05	.04	.02	.02

Sparse representation: `{"As You Like It": [(1, 1), (2, 114), (3, 36), (4, 20)], ...}`

TF-IDF weighting in scikit-learn

The `TfidfVectorizer` class is similar to the `CountVectorizer`, but during the `fit` step, it computes DF statistics, and scales the TFs during the `transform` step.

By default, it uses add-one smoothing for the DF counts but you can disable this with `smooth_idf=False`.

Stemming and lemmatization

Morphology and sparsity

- Much early work in computational linguistics treated every unique word as an atomic element, ignoring the systematic, rule-governed relationship between words like fish, fisher, and fishing.

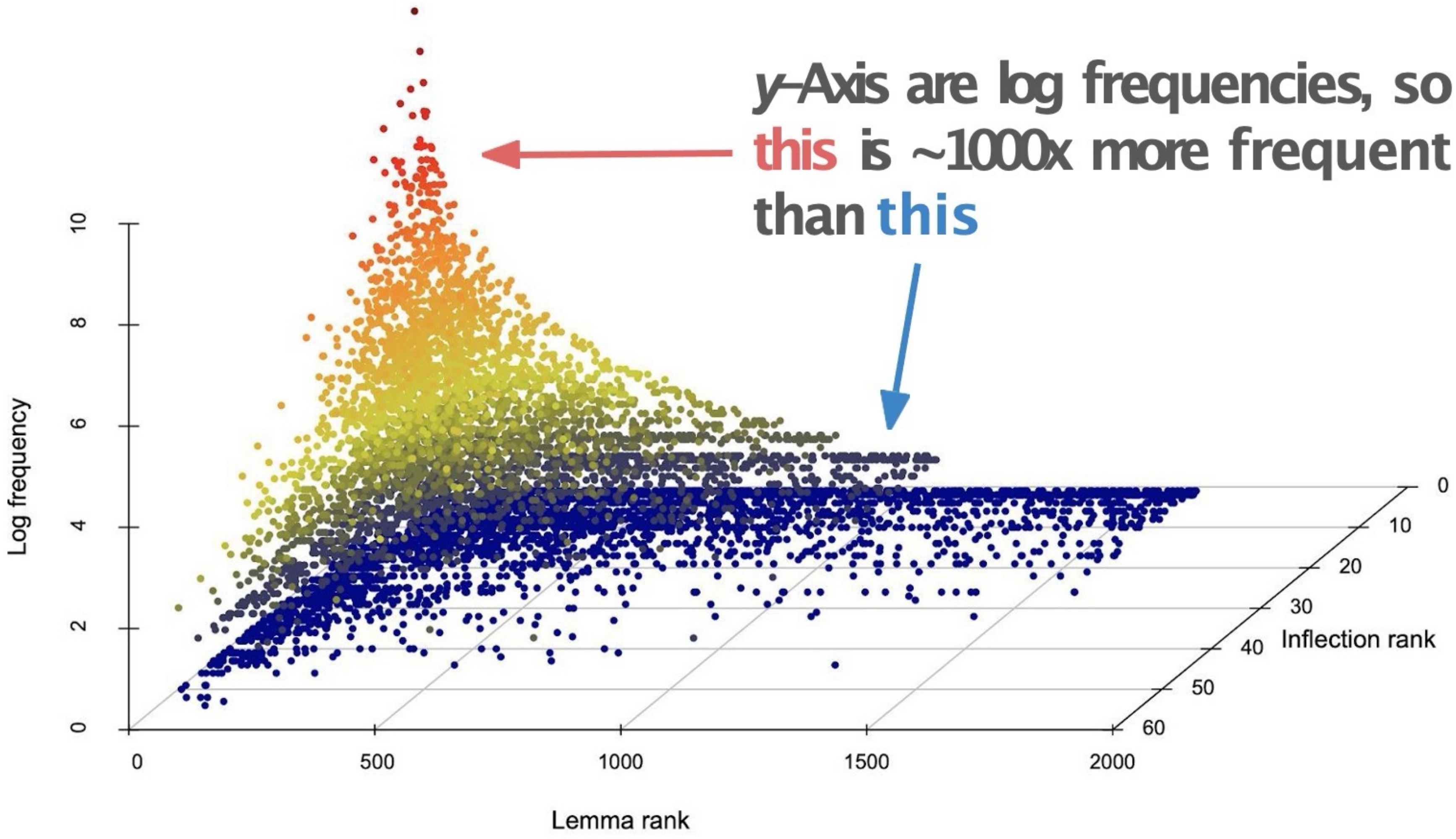
- If we've never seen the word *fisher* (or only seen it a few times) in the training set, we may have a poor ("high variance") representation of features involving that word.

- **Stemming** and **lemmatization** are two technologies we use to generalize across morphologically related words.

Lexicon compression

- Personal computers of the 1980s did not have enough memory to store a reasonably comprehensive lexicon of English (or Japanese) and engineers were forced to develop compression heuristics.

- This gets a lot worse quickly: English verbs may have as many as six forms, but verbs in Archi have more than 1.5m unique forms (Kibrik 1998).



y-Axis are log frequencies, so this is ~1000x more frequent than this

Figure 1: Frequencies of CHILDES Spanish lemmas across inflection categories.

Lexicon compression

- Personal computers of the 1980s did not have enough memory to store a reasonably comprehensive lexicon of English (or Japanese) and engineers were forced to develop compression heuristics.

- This gets a lot worse quickly: English verbs may have as many as six forms, but verbs in Archi have more than 1.5m unique forms (Kibrik 1998).

- Perhaps unsurprisingly, one of the most effective heuristics for compressing a lexicon is to store a list of "stems" and rules to generate inflectional variants.
    - E.g., the MITalk TTS system (the voice behind Stephen Hawking) generates the word scarcity from stem scarce by appending -ity and deleting a final e.

Stemmers

Generalizing this, Porter (1980) proposes a list of English affix-stripping rules.

- E.g.: Do you really think it is weakness that yields to temptation? I tell you that there are terrible temptations which it requires strength, strength and courage, to yield to?

Stemmers

Generalizing this, Porter (1980) proposes a list of English affix-stripping rules.

- E.g.: Do you really think it is weak~~ness~~ that yield~~s~~ to temptat~~ion~~? I tell you that there are terrible temptat~~ions~~ which it requir~~es~~ strength, strength and courage, to yield to?

Crucially, the "stems" generated by the Porter stemmer (as this is known) need not correspond to English words, or to the linguistic notion of the stem; they just need to form useful, semantically coherent equivalence classes (e.g., temptat- as a stem shared by temptation and temptations).

Lemmatizers

- A lemma is, *roughly*, the citation form or head word (the form you look up in a dictionary) of a word.

- Lemmatizers attempt to map words onto their lemmas, which, in contrast to the "stems" produced by stemmers, are guaranteed to be pronounceable wordforms.

Software

- Snowball stemmers are available for roughly 20 languages via `nltk.stem.snowball`.

- The WordNet lemmatizer, a knowledge-driven English lemmatizer, is available via `nltk.stem.wordnet`.

- Most of the 100 or so languages supported by Universal Dependencies could be used to train a data-driven lemmatizer using tools like Morfette or UDPipe.

Ambiguity

- Neither stemmers nor knowledge-driven lemmatizers can fully resolve ambiguity

For instance in Latvian, the wordform *ceļu* is ambiguous when considered in isolation:

it could be an inflected variant for the verb *celt* ("to lift")

or the nouns *celis* ("knee") or *ceļš* ("road")

Without context the lemmatizer can only guess

Data-driven lemmatizers frame the problem as a tagging task in which the tags are edit scripts, a set of string rewrite instructions generating a lemma from the inflected form (e.g., Chrupała 2008, 2014).

Practical / Engineering Tips

- To apply a stemmer or lemmatizer for text classification problems, one may
  - replace all words with their stem/lemma before feature extraction, or
  - extract word features, then augment them with additional features based on stems/lemmas.

- You usually don't want to display stemmed text to non-specialist end-users, since they're not guaranteed to be words or even “word-like” units. One simple solution is to keep track of the most frequent full words associated with each stem and use those in place of the stems.

- UDPipe 1, available from the command-line or Python, has great sentence boundary detection, tokenization, and lemmatization in about 60 languages:

```
udpipe --tokenize --tag en.model inputdata > outputdata
```

Survey of (mostly) English Morphology: Inflection

Morphological forms of irregular verbs

stem	eat	catch	cut
-*s* form	eats	catches	cuts
-*ing* principle	eating	catching	cutting
Past form	ate	caught	cut
–*ed* participle	eaten	caught	cut

stem	walk	merge	try	map
-*s* form	walks	merges	tries	maps
-*ing* principle	walking	merging	trying	mapping
Past form or –*ed* participle	walked	merged	tried	mapped

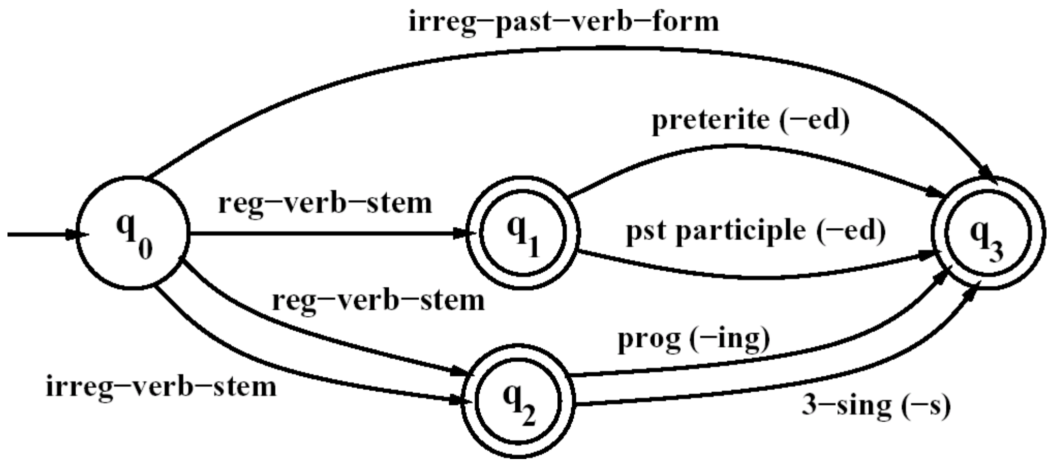
- These regular verbs and forms are significant in the morphology of English because of their *majority* and being *productive*.

Finite-State Morphological Parsing

We need at least the following to build a morphological parser:

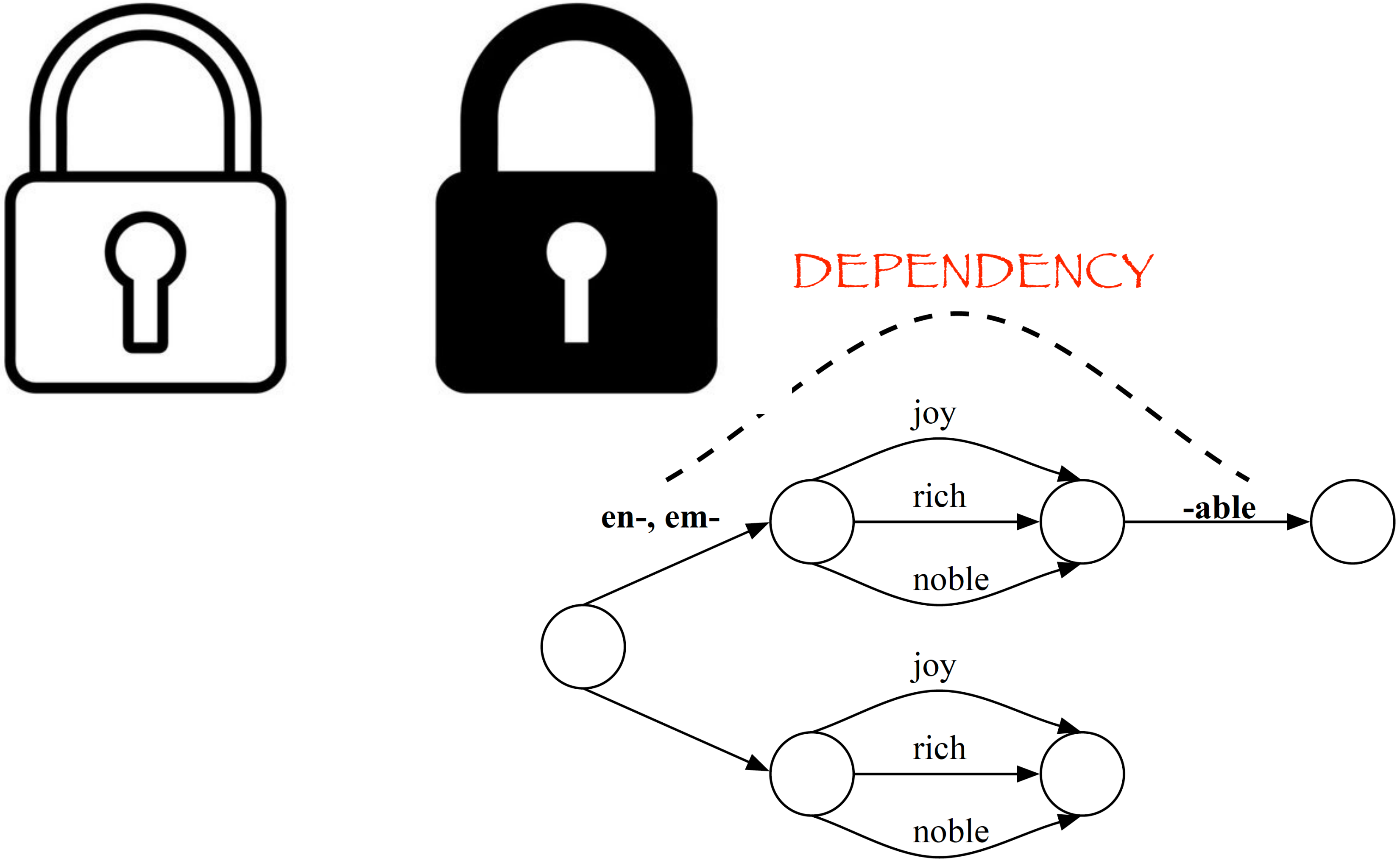
1. **Lexicon**: the list of stems and affixes, together with basic information about them (Noun stem or Verb stem, etc.)
2. **Morphotactics**: the model of morpheme ordering that explains which classes of morphemes can follow other classes of morphemes inside a word. E.g., the rule that English plural morpheme follows the noun rather than preceding it.
3. **Orthographic rules**: these **spelling rules** are used to model the changes that occur in a word, usually when two morphemes combine (e.g., the *y*→*ie* spelling rule changes *city* + *-s* to *cities*).

Finite-State Morphological Parsing: Lexicon and Morphotactics



An FSA for English verbal inflection

Reg-verb-stem	Irreg-verb-stem	Irreg-past-verb	past	Past-part	Pres-part	3sg
walk fry talk impeach	cut speak sing sang spoken	caught ate eaten	-ed	-ed	-ing	-s



- Morphology is not associative: (a + b) + c != a + (b + c)

- **non+im**+partial, **non+il**+legible, **non+in**+frequent
- **not** im+non+partial, il+non+legible, in+non+frequent
- boor+**ish-ness**, slav+**ish-ness**, baboon+**ish-ness**
- **not** boor+ish-ity, sla+ish-ity, baboon+ish-ity
- Generally, **latinate** (in-, -ity, -ic, -al, ...) suffixes must be used before **native** ones (un-, non-, -ness, -ish, -hood)

Representation requires learning

- FSAs, FSTs, CFGs….

….They provide a way to ***represent*** and ***parse*** morphological structure, but where does the capacity for those representations come from?

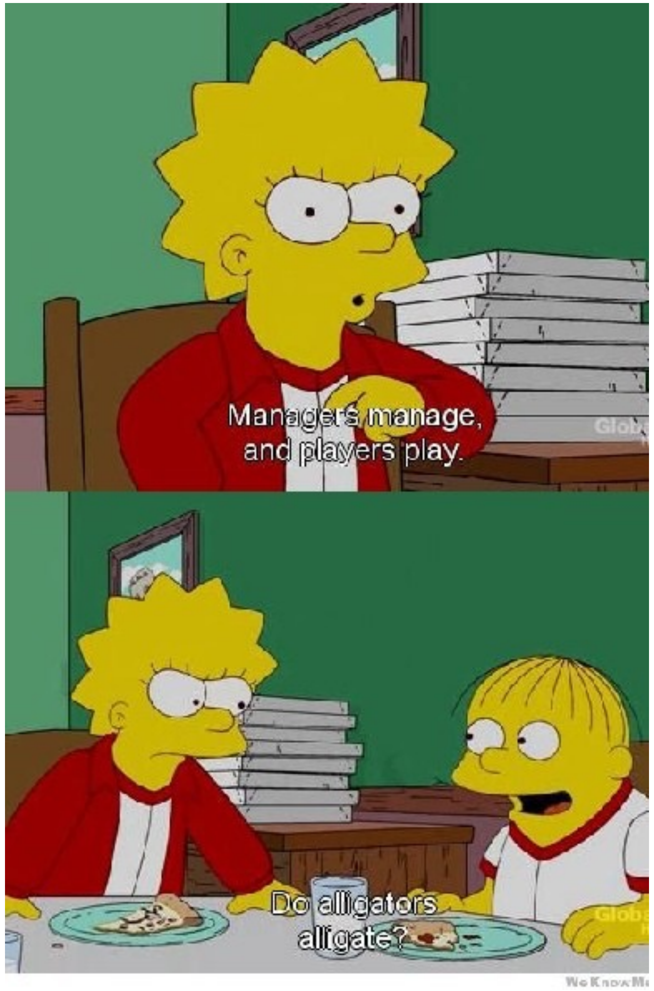
We need to actually acquire morphological rules from the input!

Big data not enough for a big problem

Corpus	Tokens (millions)	Infl. categories	Max. infl. categories per lemma	Max. saturation
Brown Corpus	1.2	6	6	100.0
Wall Street Journal Corpus	1.3	6	6	100.0
Basque	0.6	22	16	72.7
Czech	2.0	72	41	56.9
Finnish	2.1	365	147	40.3
Greek	2.8	83	45	54.2
Hungarian	1.2	76	48	63.2
Hebrew	2.5	33	23	69.7
Slovene	2.4	32	24	75.0
Spanish	2.6	51	34	66.7
Swedish	1.0	21	14	66.7
Catalan	1.7	45	33	73.3
Italian	1.4	55	47	85.5
CHILDES Spanish	1.4	55	46	83.6
CHILDES Catalan	0.3	39	27	69.2
CHILDES Italian	0.3	49	31	63.3

Fighting your inner Ralph Wiggum

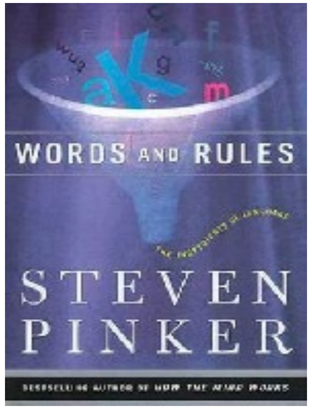
- Kids eventually learn productivity categorically
- What can we learn about the path that brings them there?
- In particular, let's look at a scheme for evaluating rule productivity
  - Rather than inducing where the hypotheses come from

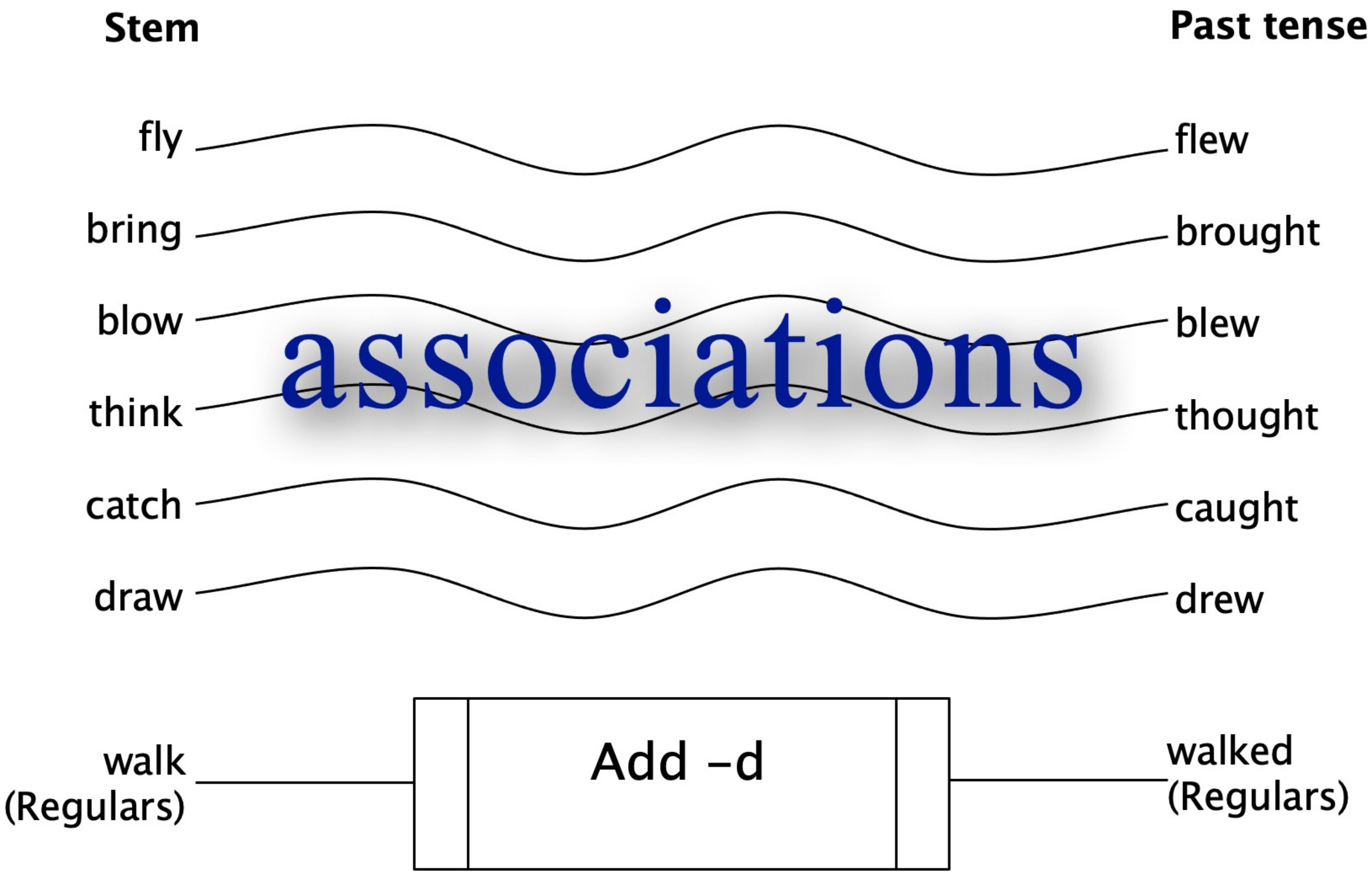


Some morphological errors in the wild

- The flatter (referring to the rolling pin; 2;7).
- She and Jenny took the sounder off with the needle (referring to an LP record; 4;6).
- But you really call it the Darth Vader collection caser (referring to a container; 4;2)
- It always sweats me. That sweater is a hot sweaty sweater (referring to the causer of sweating; 4;3)

Words and Rules





Stem **Past tense**

fly flew

bring brought

blow blew

think thought

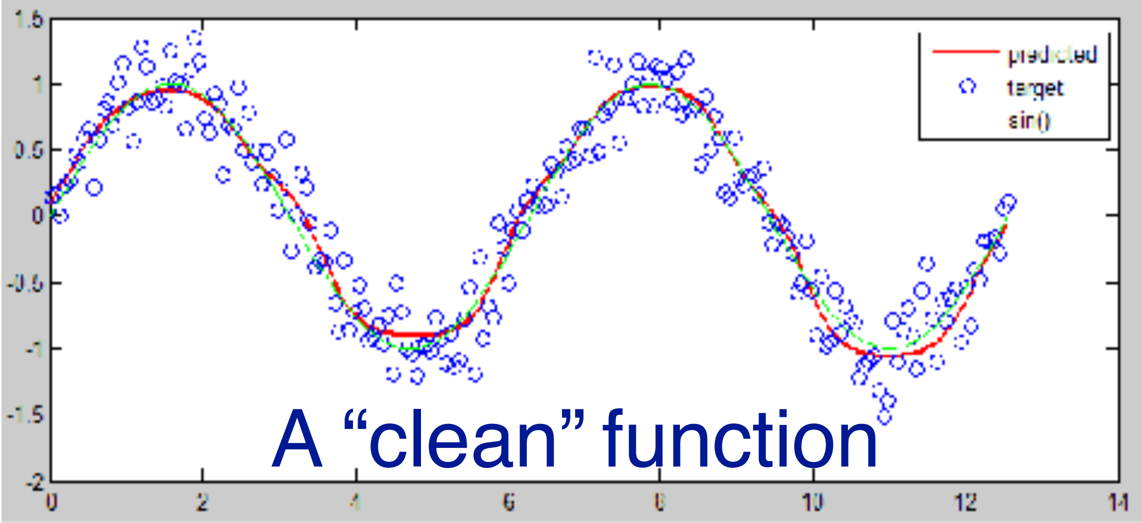
catch caught

draw drew

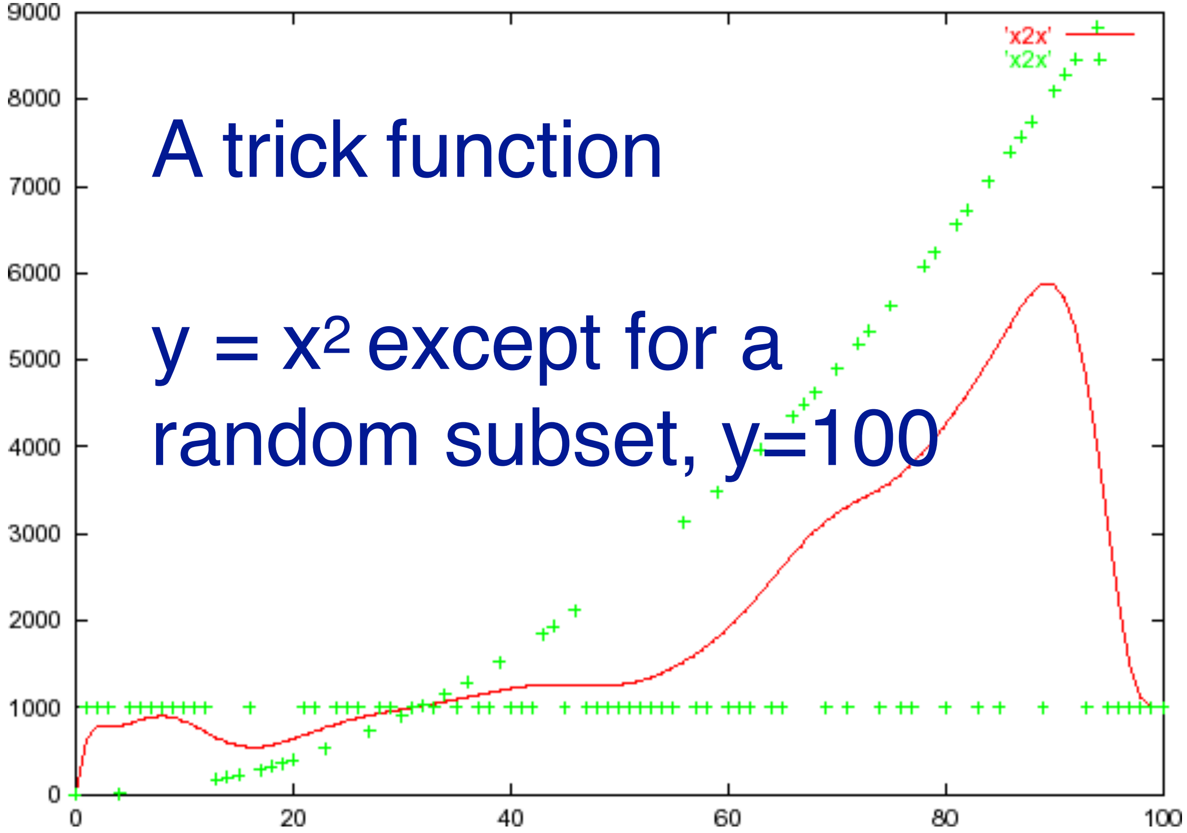
associations

walk (Regulars) Add –d walked (Regulars)

Optimizing an objective function (e.g. in NNs, for MLE, etc) can only reduce the training error



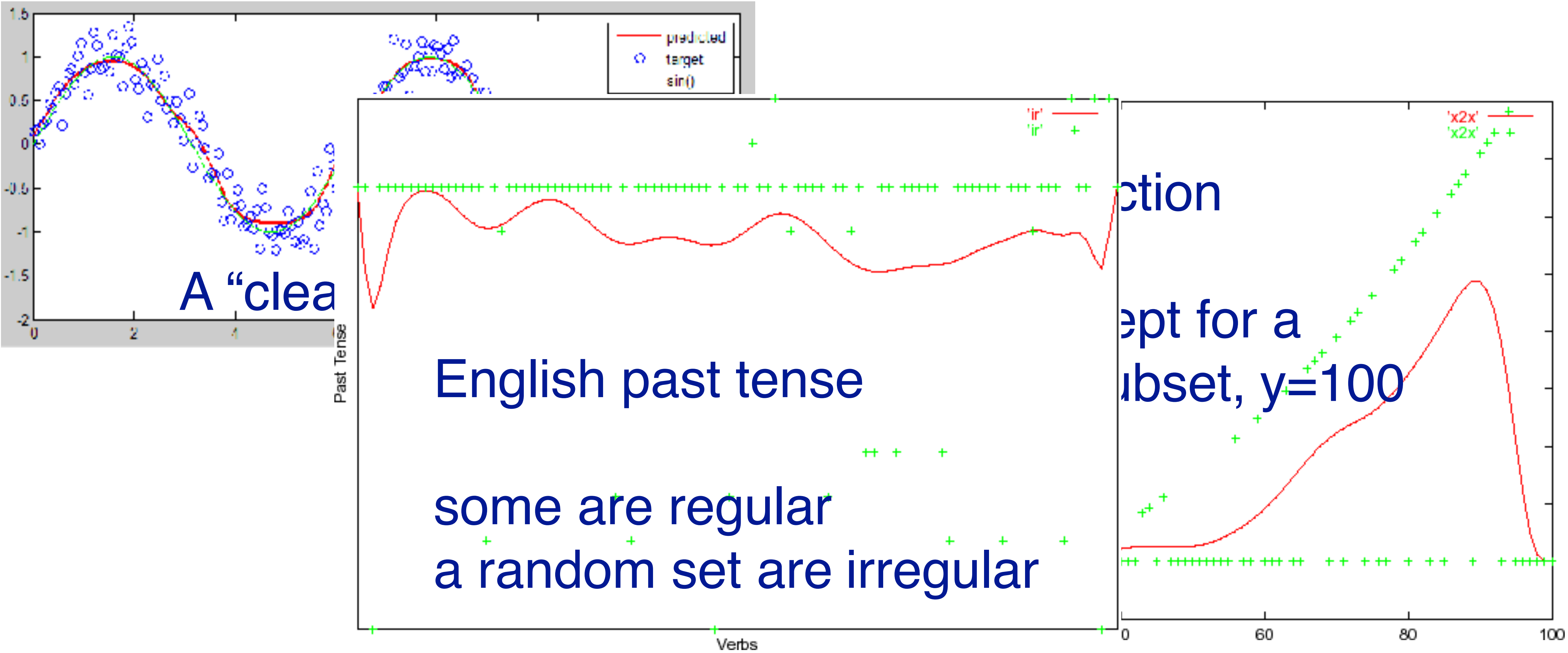
A “clean” function



A trick function

y = x^2 except for a random subset, y=100

Optimizing an objective function (e.g. in NNs, for MLE, etc) can only reduce the training error



A “clea

ction

ept for a

ubset, y=100

English past tense

some are regular

a random set are irregular

Rules and Exceptions

How to represent the following mapping:

(2,4), (3,4), (4,8), (5,10), (6,7), (7,8), (8,16)

- Could do it rote (memorize all the pairs)
- Could do it by rules:
  - y=x+1: 3, 6, 7
  - y=2x: 2, 4, 5, 8
- Either solution involves some kind of **memorization** of an arbitrary list, but they differ in **how** that memorization occurs

Do we need to memorize anything?

- What if we had a bunch of rules in competition?

- catch, buy, think, bring, seek, teach
- hit, slit, split, quit, bid vs. sit, spit

Feels like this should there'd be a competition between these "analogy" classes and the "add -ed" rule – then we won't need to memorize any words

- Just look at the pronunciation each time and decide

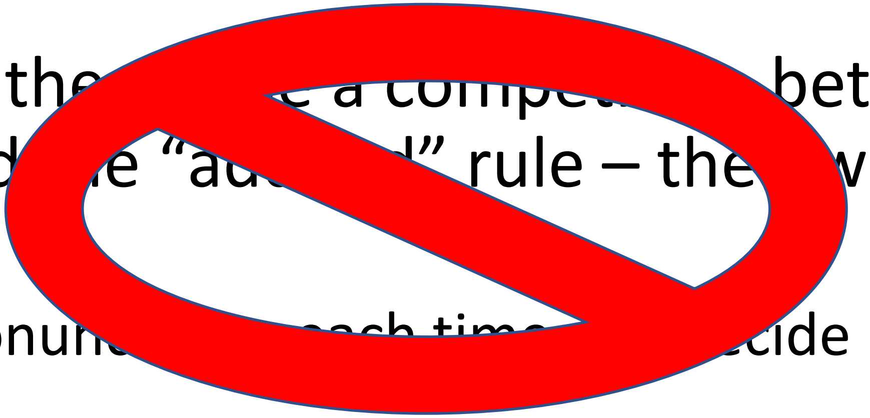
Do we need to memorize anything?

Several reasons why that’s not the case:

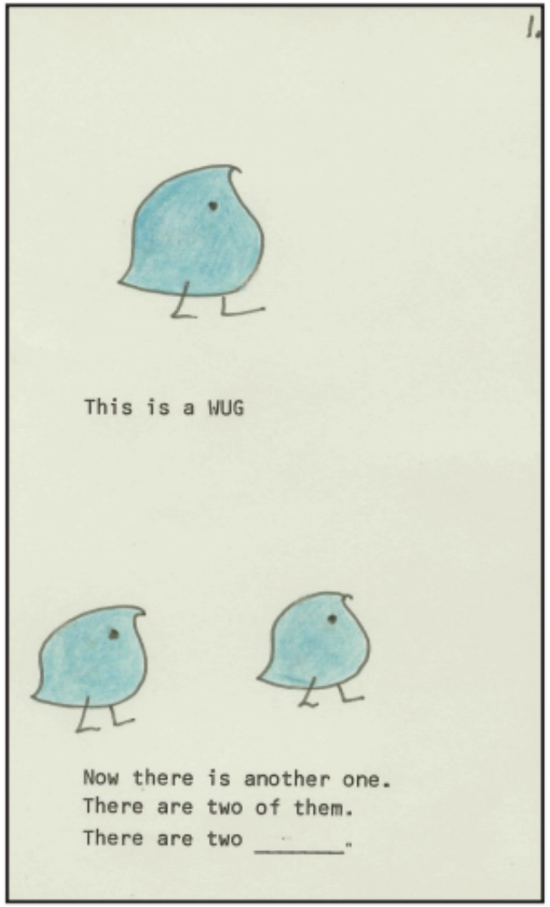
- Children do not over-irregularize (either naturally or in experiments)
- English has not gained a **new** irregular verb in the last 200 years

Feels like this should the e a compe between these “analogy” classes and e “ad ” rule – the we won’t need to memorize any words

- Just look at the pronun each tim ecide

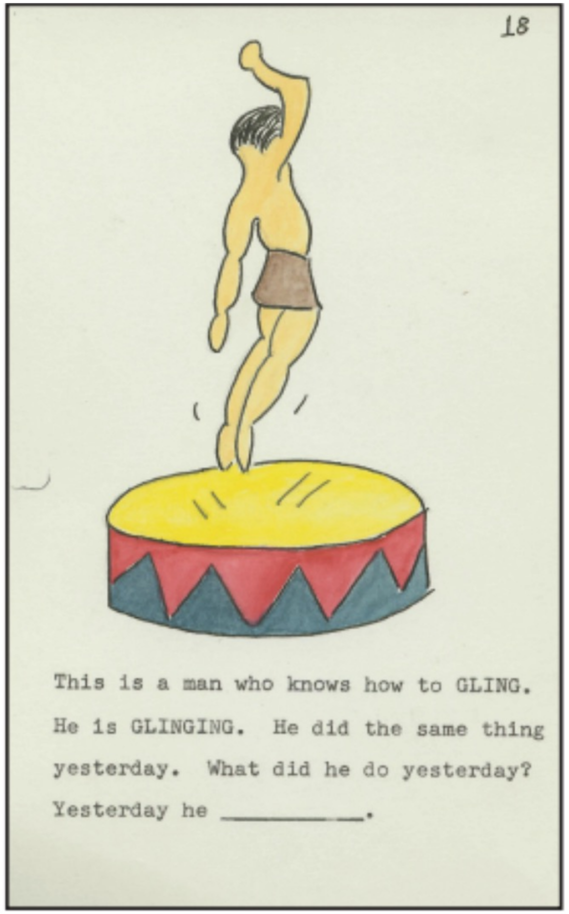


Sub-parts of the wug test

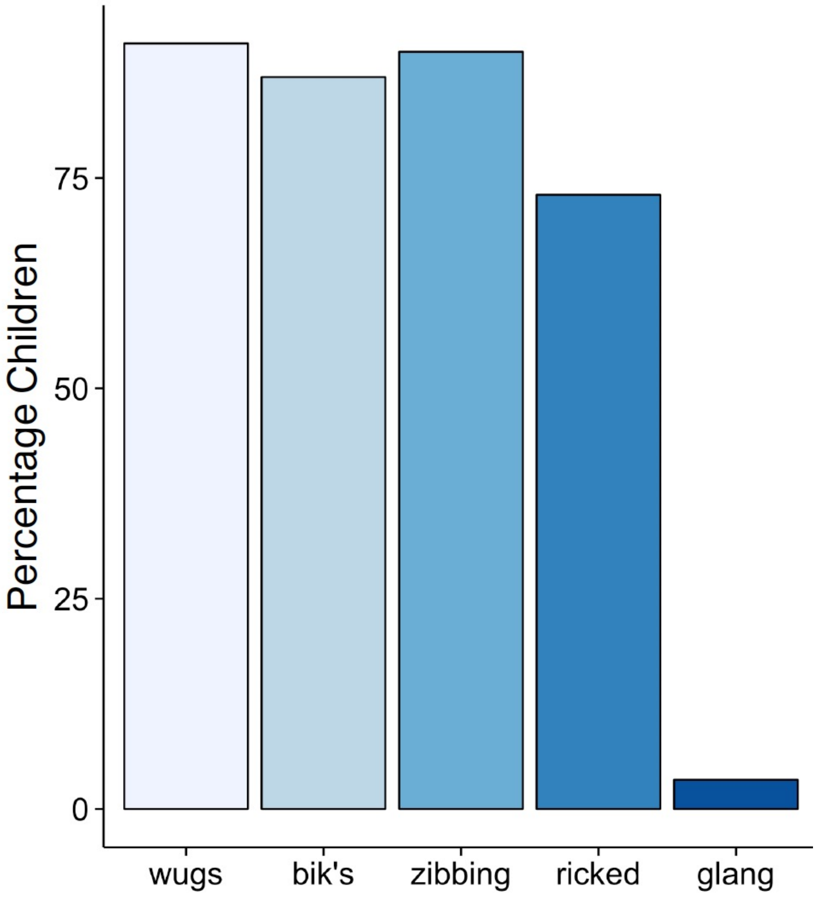


This is a WUG

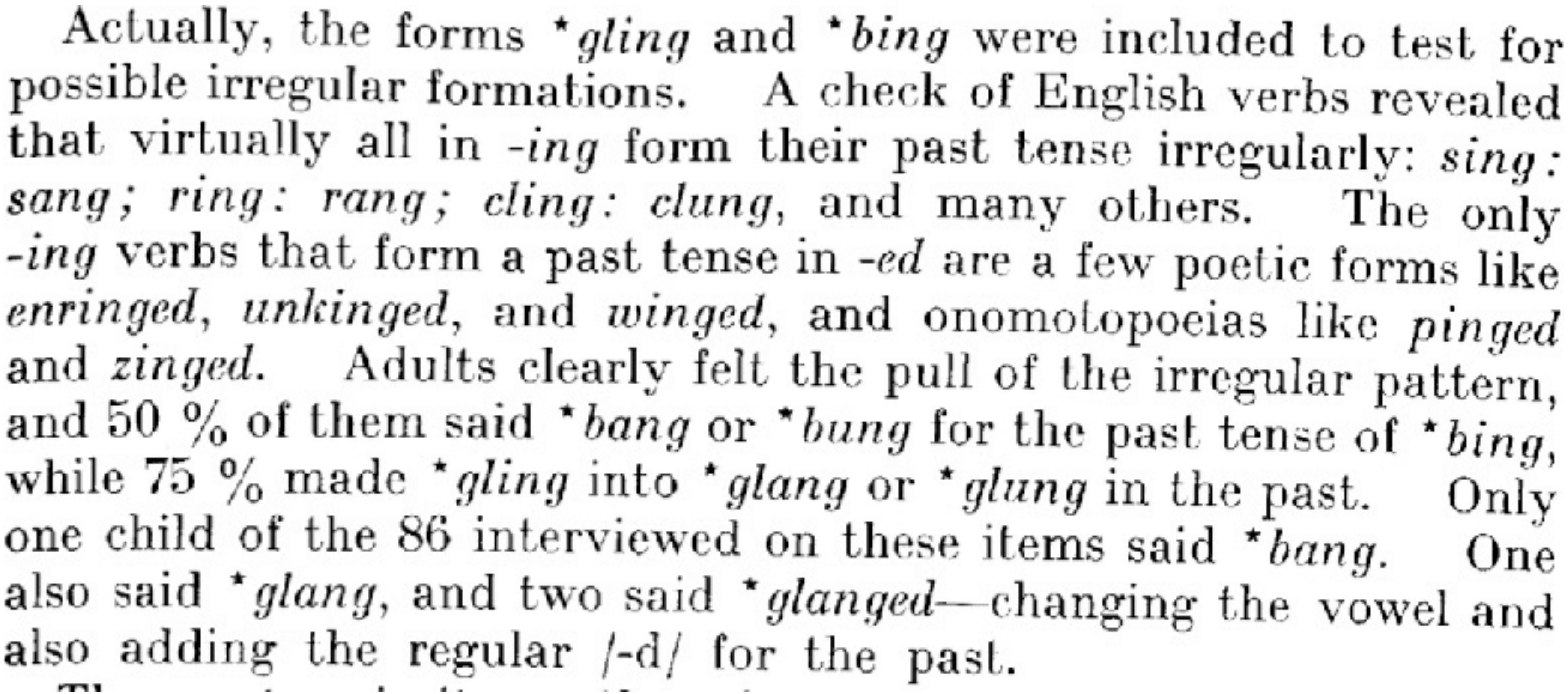
Now there is another one.
There are two of them.
There are two ______.



This is a man who knows how to GLING.
He is GLINGING. He did the same thing
yesterday. What did he do yesterday?
Yesterday he ________.

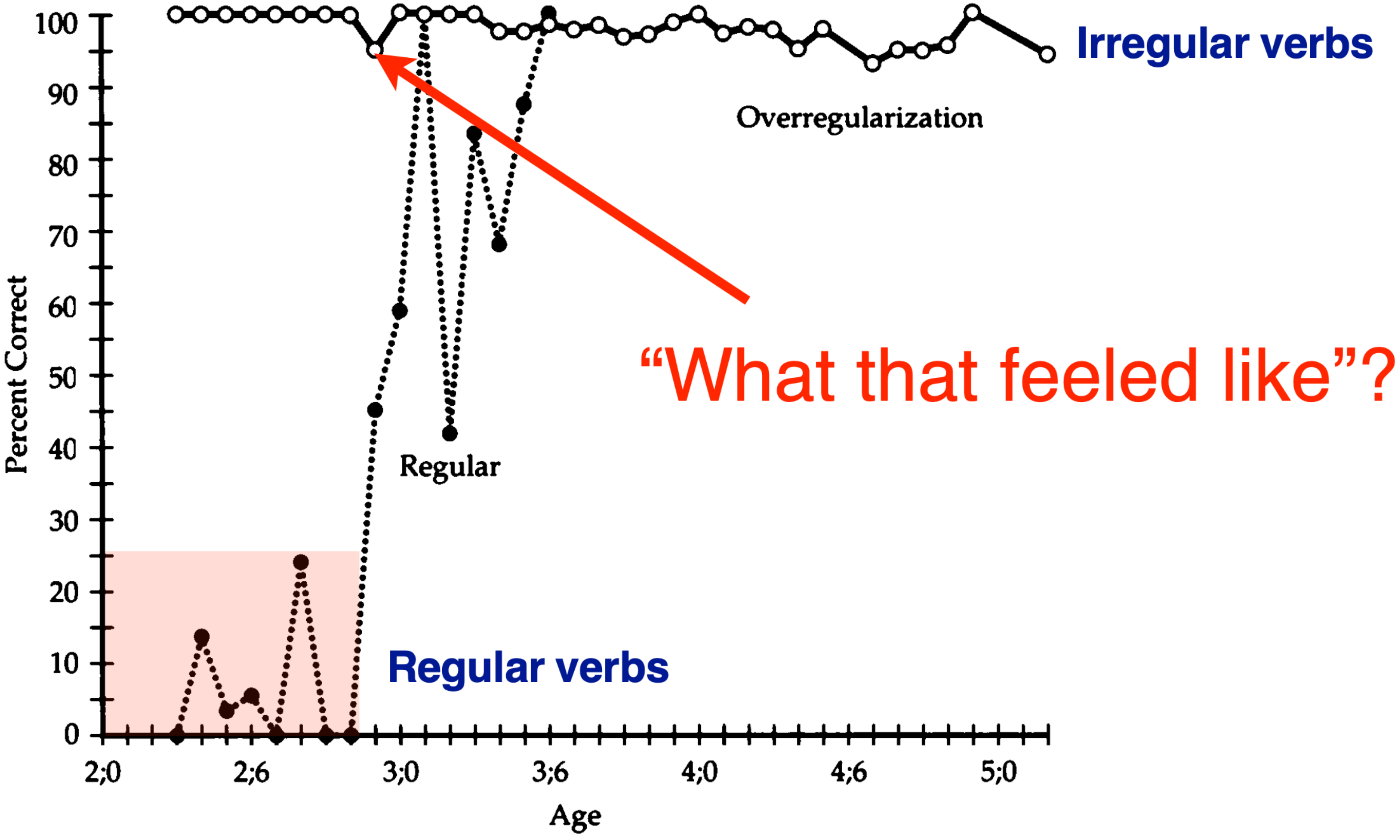


What about gling and bing?



Actually, the forms *_gling_ and *_bing_ were included to test for possible irregular formations. A check of English verbs revealed that virtually all in _-ing_ form their past tense irregularly: _sing: sang; ring: rang; cling: clung_, and many others. The only _-ing_ verbs that form a past tense in _-ed_ are a few poetic forms like _enringed_, _unkinged_, and _winged_, and onomotopoeias like _pinged_ and _zinged_. Adults clearly felt the pull of the irregular pattern, and 50 % of them said *_bang_ or *_bung_ for the past tense of *_bing_, while 75 % made *_gling_ into *_glang_ or *_glung_ in the past. Only one child of the 86 interviewed on these items said *_bang_. One also said *_glang_, and two said *_glanged_—changing the vowel and also adding the regular /-d/ for the past.

Young children make errors of omission, not co-mission



Pinker (1995, *An invitation to cognitive science*)

What’s “good enough”

(I was originally going to walk through the derivation and application of a particular model for morphological learning in language acquisition (the Tolerance Principle) but we skipped this due to time

Please see here if you’re interested, or come talk to me (Spencer)

https://www.ling.upenn.edu/~ycharles/pop.html

Regularization

Motivations

- All machine learning algorithms (specifically the training algorithms, the algorithms for learning from data) can be said to be *optimizing* some numerical quantity.

E.g., the passive-aggressive learning algorithm (Crammer et al. 2006), a variant of the perceptron learning algorithm, is an online method for maximizing the margin $\gamma = y \cdot \sigma(F; \theta)$.

Assuming this objective is well-defined, optimizing this objective on the basis of the training data makes no guarantees about performance on held-out (e.g., development or test) data.

Intuition behind regularization

Regularization is the set of techniques by which we increase *empirical error* (the fit to the training data) to reduce *generalization error* (performance on held-out data) and avoid *overfitting*.

How do we diagnose overfitting? One simple method is as follows:

- Compute *resubstitution* performance: i.e., apply the model to the training data and compute the appropriate evaluation metrics
- Compute performance to a random sample of held-out data.
- If resubstitution performance is substantially better than held-out performance, overfitting may be occurring.

What does this remind you of?

Smoothing!

The language model smoothing techniques we saw earlier this semester are regularization strategies specific to the estimation of Markovian probability distributions used in classical language models. Laplace smoothing, which we reused for Naïve Bayes classifiers, is another example.

We'll need to use something new for logistic regression, however, since we don't observe any of the probabilities directly.

Intuition behind regularization

Learning algorithms for logistic regression attempt to minimize the training data *loss* ℓ, a quantity closely related to *cross-entropy*.

To regularize logistic regression, we simultaneously minimize loss and some other quantity R that is smaller the more "general" the model is.

The trade-off between loss and regularization can be written

$$\min(\ell + R/C)$$

where C is a hyperparameter, and C is inversely proportional to regularization strength.

Three types of regularization (1/3)

Intuitively, a parameter of lower magnitude is less "committal" and therefore will generalize better to unseen data.

In *L2 regularization*, *R* is the square root of the sum of squared parameter values

$$R = \sqrt{(\beta_1^2 + \beta_2^2 + \ldots)}$$

L2-regularized logistic regression is traditionally known as *ridge regression*.

Three types of regularization (2/3)

Just as smaller parameters are "less committal", parameters whose values are 0 are the least committal of all because they can be ignored altogether.

In *L1 regularization*, *R* is the sum of the absolute value of all parameters

$$R = |\beta_1| + |\beta_2| + \ldots$$

L1-regularized logistic regression is traditionally known as *LASSO regression*.

L1 regularization has the effect of causing parameters to go exactly to zeros, in which case they can be discarded. In other words, it *induces sparsity* in parameters.

Three types of regularization (3/3)

Finally, we can use a mixture of L1 and L2 regularization, a formulation is traditionally known as *ElasticNet regression*.

See the scikit-learn documentation on ElasticNet regression for a full breakdown of the ElasticNet hyperparameters.

Alternative regularization strategies

While L1 and L2 regularization techniques are applicable to many ML models, other models make use of specific regularization tricks.

E.g., the *weight averaging* (Freund & Schapire 1999) used by the averaged perceptron has a regularizing effect so long as all weights are initialized at 0 (as is standard practice with perceptrons).

E.g., in *dropout* (Hinton et al. 2012), the parameters of neural networks are regularized by randomly replacing some parameters with zero during training.

Tuning hyperparameters

Parameters vs. hyperparameters

The *parameters* of a model refer to the weights, probabilities, etc. learned during the training phase.

> E.g., in a binary logistic regression, the parameters are given by Θ = (*b*, β) where *b* is the bias term and β is a vector of weights.

The *hyperparameters* of a model are properties of the model that have to be set by the experimenter rather than learned directly from data.

Most hyperparameters relate directly to the learning algorithm's behavior (e.g., learning rate, regularization coefficients) and cannot be changed without repeating the training phase.

Hyperparameter tuning

How do we set hyperparameters?

- 😱 use default values in our software package of choice
- 🧐 use our prior experience to guess appropriate values
- 🤓 use automatic hyperparameter search

You will achieve better results if your classification task is well-specified and you use automatic hyperparameter search.

Grid search

In *grid search*, we specify possible values for each hyperparameter, and then tune the cross-product of these values.

E.g., if we wanted to tune regularization strength *C* and the L1 ratio for an
ElasticNet logistic regression, we might consider

$C \in [.001, .01, .1, 1, 10, 100]$
L1 ratio $\in [0, .1, .3, .5, .7, .9, 1]$

This defines a "grid" containing 6 x 7 = 42 possible pairs of hyperparameter values.

Random search

In *random search* (Bergstra & Bengio 2012), we create a random grid of size *k* by sampling *k* possible values from a probability distribution associated with each hyperparameter.

E.g., instead of specifying seven possible values for the L1 ratio, we sample possible values for this hyperparameter by drawing *k* random samples from the uniform distribution [0, 1].

Black box search

Black box search methods (e.g., Golovin et al. 2017) also work from a random grid but use clever (often Bayesian) heuristics to speed up search. This is not yet built into scikit-learn. Some examples of black box search include:

- Google Vizier and its open-source version
- Weights & Biases `"bayes"` sweeping

Automatic tuning with a static split

1. For each element in the grid (whether fixed or random):
    1. We train on the training set using those hyperparameters.
    2. Using the model from (a), we predict labels for the development set.
    3. Using the predictions from (b), we compute our preferred metric (e.g., accuracy) for the development set.
2. Using the best model from (1), we predict labels for the test set.
3. Using the predictions from (2), we compute our preferred metric (e.g., accuracy) for the test set.

The results obtained in (3) are then reported.

Automatic tuning in scikit-learn (1/4)

scikit-learn does not easily support tuning based on a fixed development set. Rather, it is designed for tuning via *cross-validation* (CV).

However, a little hacking is all it takes to tune with a fixed development set...

Let us suppose *X* refers to encoded features and *Y* to labels. We begin by concatenating the train and dev sets' *X* and *Y* vectors:

```
x = numpy.concatenate([x_train, x_dev])
y = numpy.concatenate([y_train, y_dev])
```

Automatic tuning in scikit-learn (2/4)

We then inform scikit-learn about the train/development split.

```
    test_fold = numpy.concatenate(
        [
            numpy.full(x_train.shape[1], -1),
            numpy.full(x_dev.shape[1], 0)
        ]
    )
    cv =
sklearn.model_selection.PredefinedSplit(test_fold)
```

This `cv` object forces scikit-learn to only use the training set for training and the development set for tuning.

Automatic tuning in scikit-learn (3/4)

Finally, we construct the hyperparameter grid and pass it to scikit-learn.

```
    grid = {
         "C": [.001, .01, .1, 1, 10, 100],
         "penalty": ["l1", "l2"],
    }
     model = sklearn.model_select.GridSearchCV(

sklearn.linear_model.LogisticRegression(),
        grid,
        cv=cv
     )
    model.fit(x, y)
```

Automatic tuning in scikit-learn (4/4)

The approach for scikit-learn random hyperparameter search looks quite similar except that the values of your `grid` dictionary are probability distributions and you use `RandomizedSearchCV` instead of `GridSearchCV`.

For more information, see:

- the scikit-learn documentation on hyperparameter tuning
- Kyle has a blog post about using fixed training/dev/test set in scikit-learn

Final notes

- It is *essential* to use regularization and to tune *C* to obtain optimal logistic regression performance
  - Though this is a different usage of logistic regressions as common for doing statistical analysis of experimental or observational data
- Not all logistic regression solvers support all regularization techniques. The SAGA solver (`solver="saga"`) is the most general one and works well for large, sparse data sets.
- If you're intending to perform inference using the probability distribution computed by the model (rather than merely predicting the best label), set `multi_class="multinomial"`.
- See [the detailed user guide]() for more information.