The Artificial Language Toolkit, II: Reproduction Using ALTK

Shane Steinert-Threlkeld
shanest@uw.edu — www.shane.st

MIT Linguistics
16 March 2023

Today's Plan

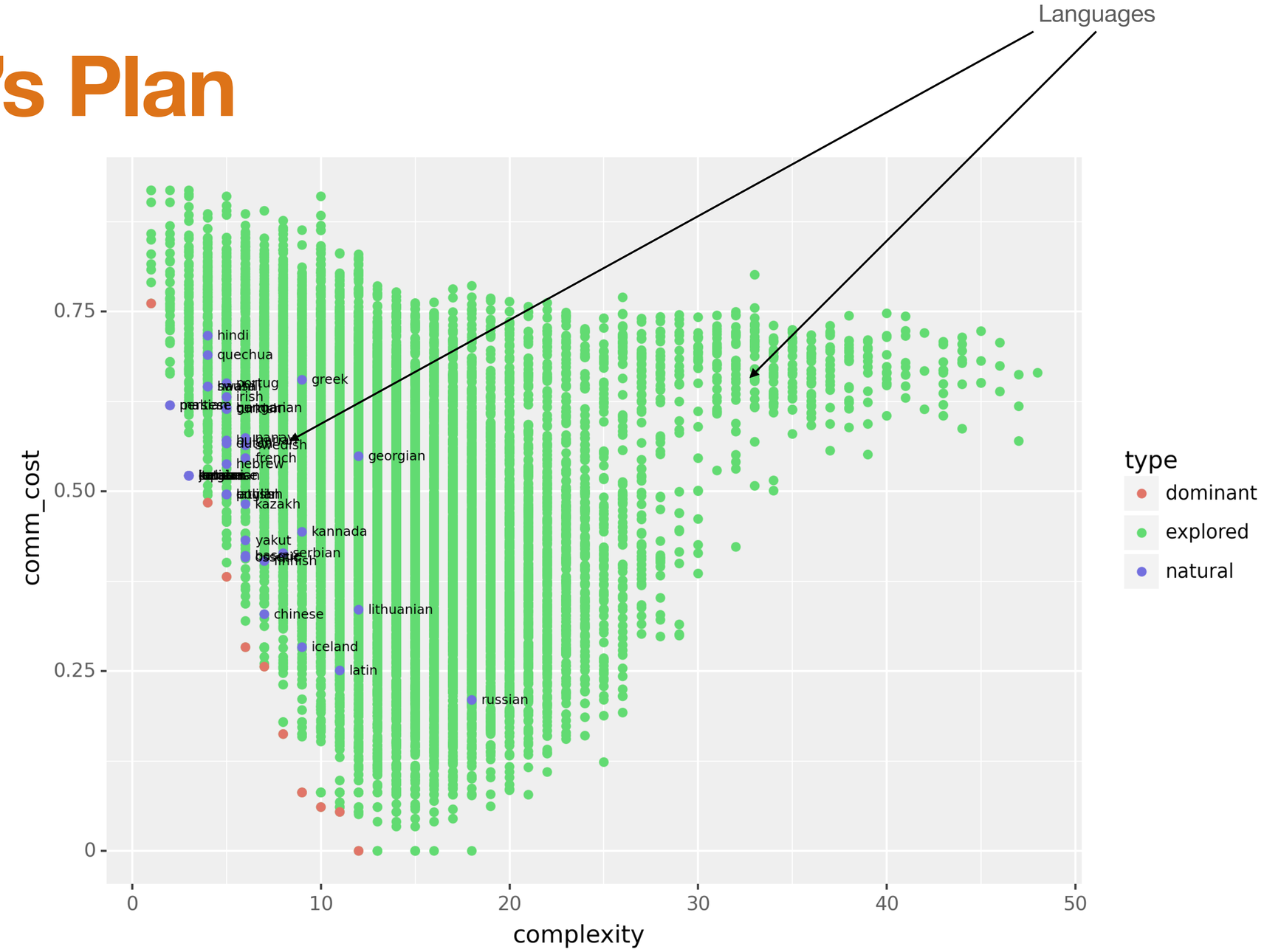
- Walk through reproducing yesterday's results in ALTK
- What abstractions ALTK provides
- What pieces are case-study specific

- Will show examples in Python; ask for clarification at any point!
  - Also happy to hop over to the docs or code, play around with the demo, et cetera at any point

Today's Plan


comm_cost
0.75
0.50
0.25
0
0
10
20
30
40
50
complexity
hindi
quechua
greek
irish
georgian
swedish
french
hebrew
kazakh
kannada
yakut
serbian
lithuanian
chinese
iceland
latin
russian
type
dominant
explored
natural

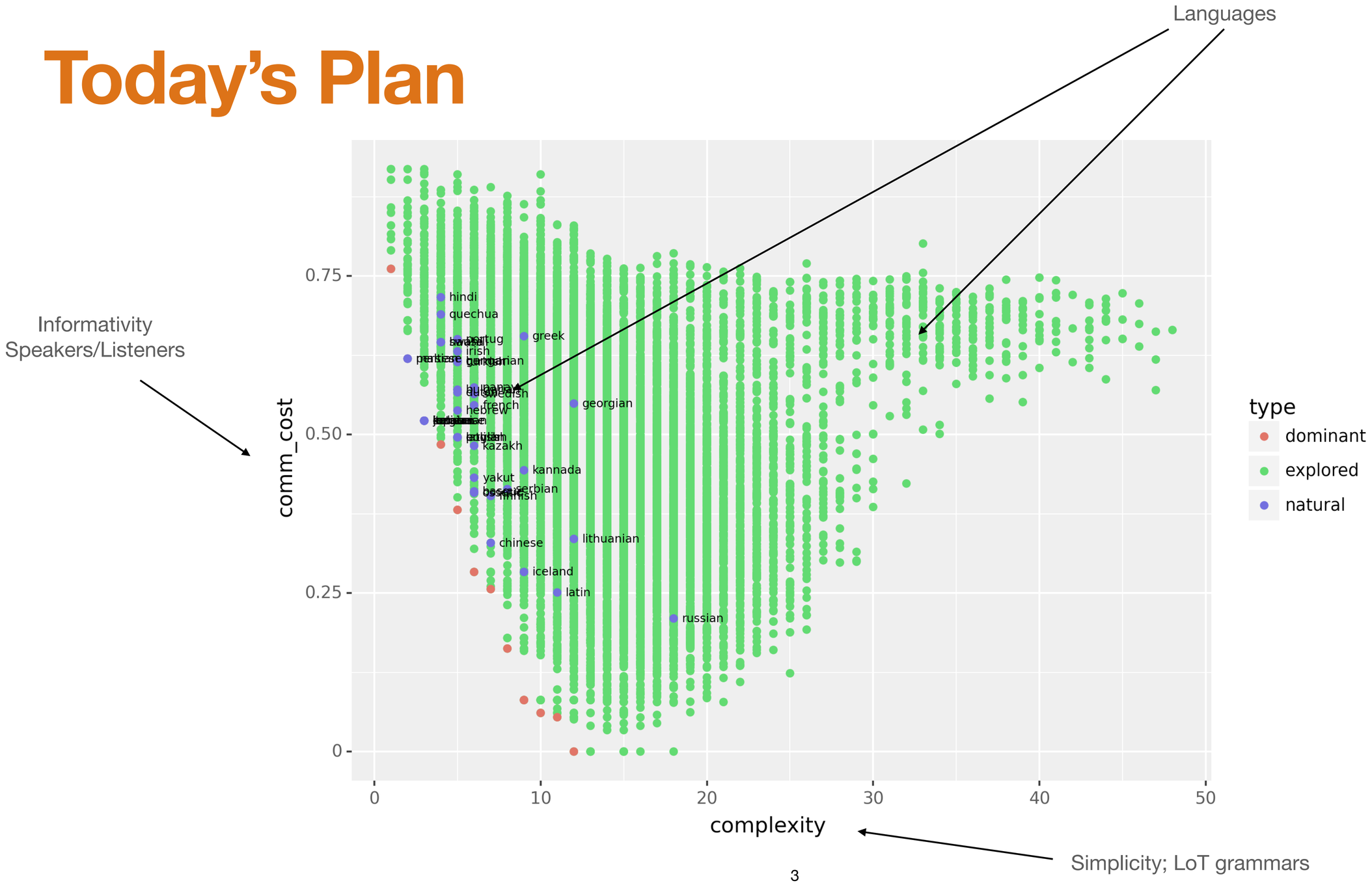
Today's Plan


Languages
comm_cost
0.75
0.50
0.25
0
0
10
20
30
40
50
complexity
hindi
quechua
greek
irish
georgian
swedish
french
hebrew
kazakh
kannada
yakut
serbian
chinese
lithuanian
iceland
latin
russian
type
dominant
explored
natural

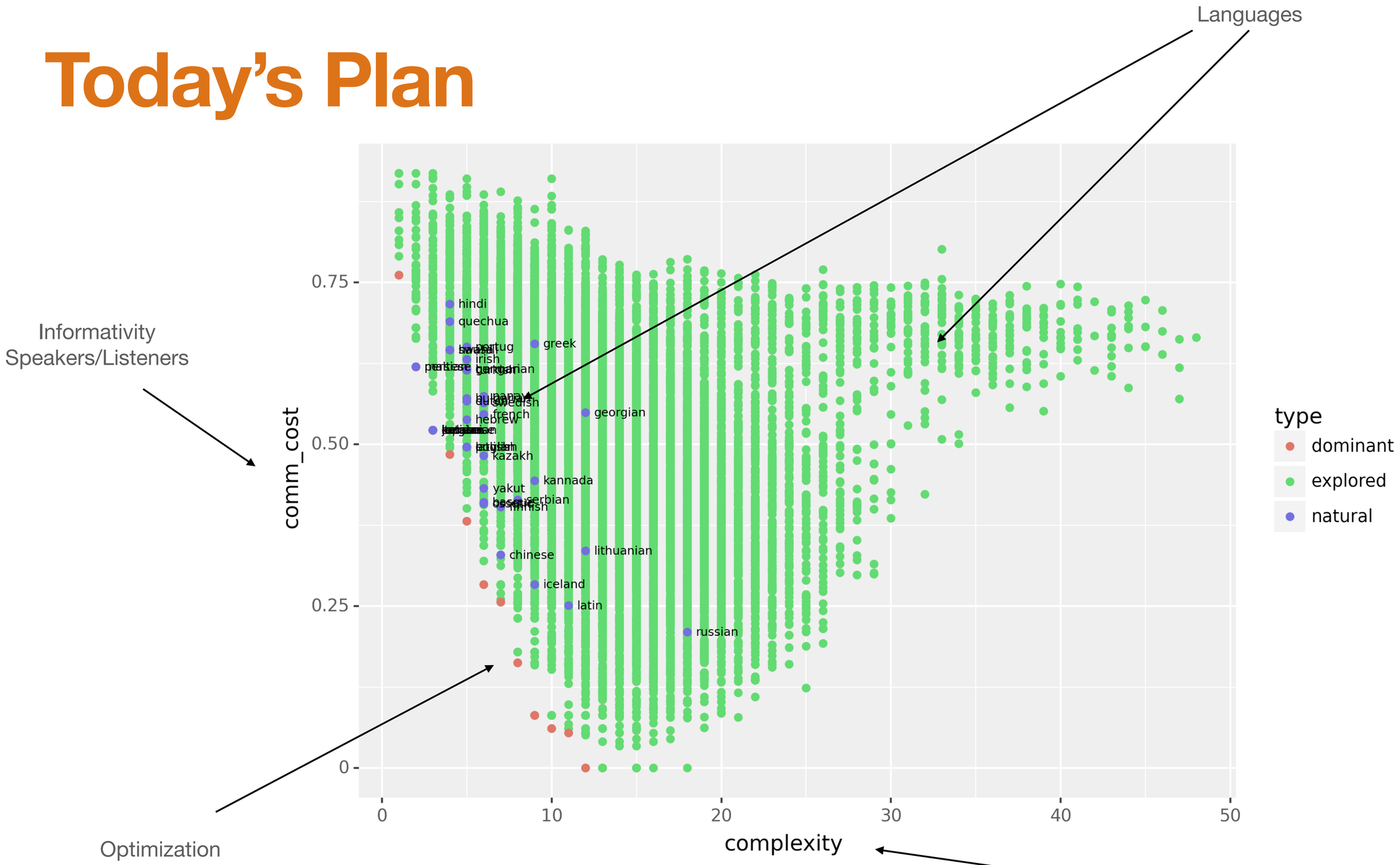
Today's Plan


Languages
Informativity
Speakers/Listeners
comm_cost
complexity
0.75
0.50
0.25
0
0
10
20
30
40
50
hindi
quechua
greek
irish
georgian
hebrew
french
kazakh
kannada
yakut
chinese
lithuanian
iceland
latin
russian
type
dominant
explored
natural

Today's Plan


Languages
Informativity
Speakers/Listeners
comm_cost
0.75
0.50
0.25
0
0
10
20
30
40
50
complexity
hindi
quechua
greek
irish
georgian
french
hebrew
kazakh
kannada
yakut
serbian
chinese
lithuanian
iceland
latin
russian
type
dominant
explored
natural
Simplicity; LoT grammars

Today's Plan


Languages
Informativity
Speakers/Listeners
Optimization
Simplicity; LoT grammars
comm_cost
complexity
0.75
0.50
0.25
0
0
10
20
30
40
50
hindi
quechua
greek
irish
georgian
swedish
hebrew
french
kazakh
kannada
yakut
serbian
finnish
chinese
lithuanian
iceland
latin
russian
type
dominant
explored
natural

Languages

Languages

- Recall from yesterday: language = lexicon = set of meanings
- In ALTK:
  - A `Language` is basically a set of `Expression`s, plus some helpers
  - An `Expression` contains a form (string) and a `Meaning`

Languages

- Recall from yesterday: language = lexicon = set of meanings
- In ALTK:
  - A `Language` is basically a set of `Expression`s, plus some helpers
  - An `Expression` contains a form (string) and a `Meaning`

```
>>> from altk.language.language import Expression, Language
>>> # assuming the meaning `a_few_meaning` has already been constructed
>>> # define the expression
>>> a_few = NumeralExpression(form="a few", meaning=a_few_meaning)
>>> # define a very small language
>>> lang_1 = Language([a_few])
>>> # or a slightly larger one with synonymy
>>> lang_2 = Language([a_few] * 3)
```

Languages

- Recall from yesterday: language = lexicon = set of meanings
- In ALTK:
  - A `Language` is basically a set of `Expression`s, plus some helpers
  - An `Expression` contains a form (string) and a `Meaning`

```
>>> from altk.language.language import Expression, Language
>>> # assuming the meaning `a_few_meaning` has already been constructed
>>> # define the expression
>>> a_few = NumeralExpression(form="a few", meaning=a_few_meaning)
>>> # define a very small language
>>> lang_1 = Language([a_few])
>>> # or a slightly larger one with synonymy
>>> lang_2 = Language([a_few] * 3)
```

Can sub-class any of these types for
Domain-specific uses (will also see this later)

Meanings

- What is a `Meaning`?
  - A (distribution over a) set of `Referent`s
  - A `Referent`: effectively any Python object
  - Set of referents: a `Universe`
    - With a prior; uniform if not specified

Meanings for Indefinites

- `referents.csv`:
  - Assumed to have a “name” column
  - With unique entries

```
name
specific-known
specific-unknown
nonspecific
freechoice
negative-indefinite
npi
```

- `data/Beekhuizen_prior.csv`:
  - “name” assumed to align

name	probability
specific-known	0.08125
specific-unknown	0.08125
nonspecific	0.2625
freechoice	0.0979166666666667
negative-indefinite	0.15
npi	0.327083333333333

Meanings for Indefinites

- `meaning.py`: defines the `Universe`

```python
import pandas as pd
from altk.language.semantics import Universe

referents = pd.read_csv("indefinites/referents.csv")
prior = pd.read_csv("indefinites/data/Beekhuizen_priors.csv")
assert (referents["name"] == prior["name"]).all()
referents["probability"] = prior["probability"]
universe = Universe.from_dataframe(referents)
```

Each row will be one `Referent`, can have arbitrary columns (features) and values in the CSV

Meanings for Indefinites

- `meaning.py`: defines the `Universe`

Could also put “probability” column in referents.csv

```python
import pandas as pd
from altk.language.semantics import Universe

referents = pd.read_csv("indefinites/referents.csv")
prior = pd.read_csv("indefinites/data/Beekhuizen_priors.csv")
assert (referents["name"] == prior["name"]).all()
referents["probability"] = prior["probability"]
universe = Universe.from_dataframe(referents)
```

Each row will be one `Referent`, can have arbitrary columns (features) and values in the CSV

Languages

Artificial

- `altk.language.sampling` provides some convenient methods for sampling random languages (we'll see more later)

Languages

Natural

- `data/natural_language_indefinites.csv`:
  - (see `scripts/convert_haspelmath.py` for cleaning the raw data we inherited)

language	expression	specific-known	specific-unknown	nonspecific	freechoice	negative-indefinite	npi
german	01n	False	False	False	False	True	False
german	01e	True	True	True	False	False	False
german	01i	False	True	True	True	False	True
dutch	02d	False	False	True	True	False	True
dutch	02i	True	True	True	False	False	False
dutch	02n	False	False	False	False	True	False
english	03a	False	False	False	True	False	True
english	03n	False	False	False	False	True	False
english	03s	True	True	True	False	False	False

Languages

Natural

- `util.read_natural_languages`
  - reads the aforementioned CSV file and produces a list of `Language`s

Informativity

Informativity

Recall from yesterday

$$
\begin{aligned}
C(S,R) &:= 1 - I(S,R) \\
I(S,R) &:= \mathbb{E}_P[u(p,p')] \\
&= \sum_p P(p) \sum_e P_S(e \mid p) \sum_{p'} P_R(p' \mid e) \cdot u(p,p')
\end{aligned}
$$

Informativity

Recall from yesterday

$$
\begin{aligned}
C(S,R) &:= 1 - I(S,R) \\
I(S,R) &:= \mathbb{E}_P[u(p,p')] \\
&= \sum_p P(p) \sum_e P_S(e \mid p) \sum_{p'} P_R(p' \mid e) \cdot u(p,p')
\end{aligned}
$$

Prior (comm. need)

Informativity

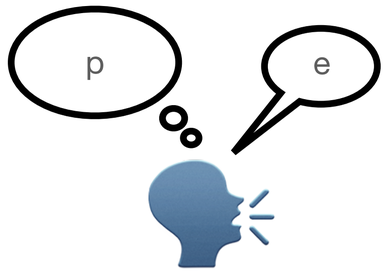
Recall from yesterday

$$C(S,R) := 1 - I(S,R)$$

$$I(S,R) := \mathbb{E}_P[u(p,p')]$$

$$= \sum_p P(p) \sum_e P_S(e \mid p) \sum_{p'} P_R(p' \mid e) \cdot u(p,p')$$

Prior (comm. need)



Informativity

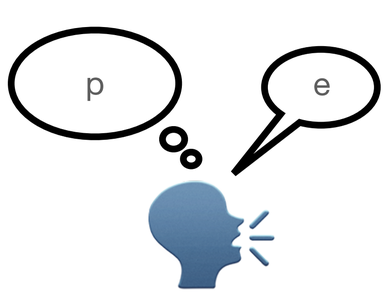
Recall from yesterday

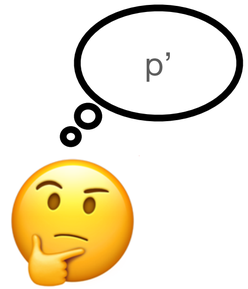
$$C(S,R) := 1 - I(S,R)$$

$$I(S,R) := \mathbb{E}_P[u(p,p')]$$

$$= \sum_p P(p) \sum_e P_S(e \mid p) \sum_{p'} P_R(p' \mid e) \cdot u(p,p')$$

Prior (comm. need)





Informativity

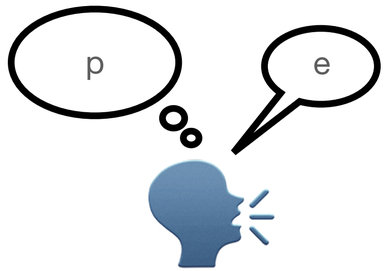
Recall from yesterday

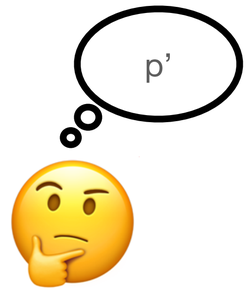
$$C(S,R) := 1 - I(S,R)$$

$$I(S,R) := \mathbb{E}_P[u(p,p')]$$

$$= \sum_p P(p) \sum_e P_S(e \mid p) \sum_{p'} P_R(p' \mid e) \cdot u(p,p')$$

Prior (comm. need)





Utility

Informativity in ALTK

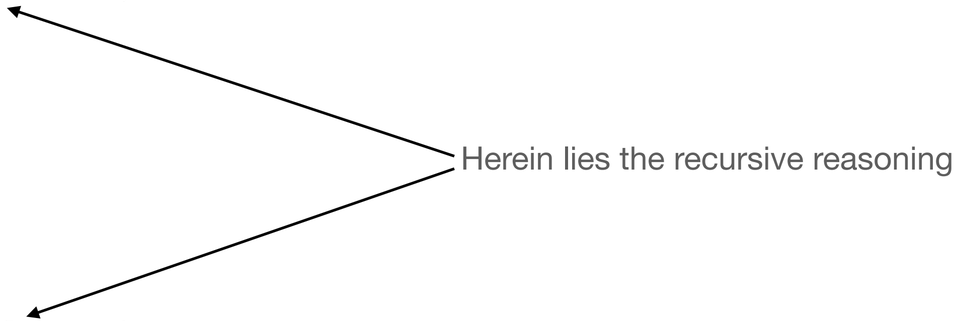
Speakers, Listeners

- `altk.effcomm.agent.CommunicativeAgent`: conditional probability matrix
  - `Speaker`
    - `LiteralSpeaker(language)`
    - `PragmaticSpeaker(language, listener)`
  - `Listener`
    - `LiteralListener(language)`
    - `PragmaticListener(language, speaker)`

Informativity in ALTK

Speakers, Listeners

- `altk.effcomm.agent.CommunicativeAgent`: conditional probability matrix
  - `Speaker`
    - `LiteralSpeaker(language)`
    - `PragmaticSpeaker(language, listener)`
  - `Listener`
    - `LiteralListener(language)`
    - `PragmaticListener(language, speaker)`



Informativity in ALTK

The core computation

- `altk.effcomm.informativity.informativity`

```python
def informativity(
    language: Language,
    prior: np.ndarray,
    utility: Callable[[Referent, Referent], float] = indicator_utility,
    agent_type: str = "literal",
) -> float:
```

- Calls `altk.effcomm.informativity.communicative_success`

$$I(S,R) = \sum \text{diag}(P)SR \odot U$$

Informativity for Indefinites

- Very easy to just use these
- `measures.py`

From meaning.py

```python
prior = indefinites_universe.prior_numpy()


def comm_cost(language: Language) -> float:
    """Get C(L) := 1 - informativity(L).
    Passes in the prior from `indefinites_universe` to altk's informativity calculator.
    """
    return 1 - informativity(language, prior)
```

`from altk.effcomm.informativity import informativity`

Simplicity

Grammars in ALTK

- `altk.language.grammar.Grammar`
  - Basically a PCFG, which can generate/parse expressions, which are themselves callable functions (more in a second)
- `altk.language.grammar.Rule`
  - One rule: lhs, rhs, name, function
- `altk.language.grammar.GrammaticalExpression`
  - Subclass of `Expression`
  - `evaluate`: calls on `Universe`, generates `Meaning`

Grammars in ALTK

Useful methods

- `altk.language.grammar.Grammar`
  - `parse`: string to `GrammaticalExpression`
  - `enumerate`: generate all (possibly unique) expressions up to a given depth
  - `get_unique_expressions`:

```python
def get_unique_expressions(
    self,
    depth: int,
    lhs: Any = None,
    max_size: float = float("inf"),
    unique_key: Callable[[GrammaticalExpression], Any] = None,
    compare_func: Callable[
        [GrammaticalExpression, GrammaticalExpression], bool
    ] = None,
) -> dict[GrammaticalExpression, Any]:
```

Which Expression to store in cases of “ties”?
Key use case: the shorter (`len`) `Expression`!

Grammars in ALTK

Useful methods

- `altk.language.grammar.Grammar`
  - `parse`: string to `GrammaticalExpression`
  - `enumerate`: generate all (possibly unique) expressions up to a given depth
  - `get_unique_expressions`:

```python
def get_unique_expressions(
    self,
    depth: int,
    lhs: Any = None,
    max_size: float = float("inf"),
    unique_key: Callable[[GrammaticalExpression], Any] = None,
    compare_func: Callable[
        [GrammaticalExpression, GrammaticalExpression], bool
    ] = None,
) -> dict[GrammaticalExpression, Any]:
```

What property determines uniqueness?
Key use case: the `Meaning` of an `Expression`!

Which Expression to store in cases of “ties”?
Key use case: the shorter (`len`) `Expression`!

LoT Grammar for Indefinites

- `grammar.py`:

```python
from altk.language.grammar import Grammar, Rule


indefinites_grammar = Grammar(bool)
# basic propositional logic
indefinites_grammar.add_rule(Rule("and", bool, (bool, bool), lambda p1, p2: p1 and p2))
indefinites_grammar.add_rule(Rule("or", bool, (bool, bool), lambda p1, p2: p1 or p2))
indefinites_grammar.add_rule(Rule("not", bool, (bool,), lambda p1: not p1))
```

LoT Grammar for Indefinites

- `grammar.py`:

```
indefinites_grammar.add_rule(
    Rule("K+", bool, None, lambda point: point.name == "specific-known")
)
indefinites_grammar.add_rule(
    Rule("K-", bool, None, lambda point: point.name != "specific-known")
)
indefinites_grammar.add_rule(
    Rule(
        "S+",
        bool,
        None,
        lambda point: point.name in ("specific-known", "specific-unknown"),
    )
)
indefinites_grammar.add_rule(
    Rule(
        "S-",
        bool,
        None,
        lambda point: point.name not in ("specific-known", "specific-unknown"),
    )
)
```

Optimization

Evolutionary Algorithm

For estimating Pareto frontier

- `altk.effcomm.optimization.EvolutionaryOptimizer`:
  - Objectives to minimize (complexity, communicative cost)
  - Expressions: possible expressions
  - `Mutation`s: how to generate offspring from parent `Language`s
    - `AddExpression`
    - `RemoveExpression`

```python
class Mutation:
    @abstractmethod
    def precondition(language: Language, **kwargs) -> bool:
        """Whether a mutation is allowed to apply to a language."""
        raise NotImplementedError

    @abstractmethod
    def mutate(language: Language, expressions: list[Expression], **kwargs) -> Language:
        """Mutate the language, possibly using a list of expressions."""
        raise NotImplementedError()
```

Evolutionary Algorithm

For estimating Pareto frontier

- `EvolutionaryOptimizer.fit(seed_population)`:

```python
for _ in tqdm(range(self.generations)):

    # Keep track of visited
    explored_languages.extend(copy.copy(languages))

    # Calculate dominating individuals
    dominating_languages = pareto_optimal_languages(
        languages, self.objectives, unique=True
    )
    # Possibly explore
    parent_languages = sample_parents(
        dominating_languages, explored_languages, explore
    )

    # Mutate dominating individuals
    mutated_result = self.sample_mutated(parent_languages)
    languages = mutated_result
```

```
from altk.effcomm.tradeoff import pareto_optimal_languages
```

All Together Now

Executing the Indefinites Analysis

- `scripts/`: scripts for each stage, using what we've just defined
- `README.md` describes in some detail what each one does
  - Which files/data it consumes
  - While files/data it produces

Step 1: Generate Expressions

- `generate_expressions`:

```
expressions_by_meaning = indefinites_grammar.get_unique_expressions(
    3,
    max_size=2 ** len(indefinites_universe),
    unique_key=lambda expr: expr.evaluate(indefinites_universe),
    compare_func=lambda e1, e2: len(e1) < len(e2),
)
```

- `outputs/generated_expressions.yml`:

```
- form: ''
  grammatical_expression: K+
  length: 1
  meaning:
    referents:
    - &id001
      name: specific-known
      probability: 0.08125
- form: ''
  grammatical_expression: K-
  length: 1
```

Step 2: Estimating Pareto Frontier

Step 2: Estimating Pareto Frontier

- `estimate_pareto`:

```
expressions, expressions_by_meaning = read_expressions(
    "indefinites/outputs/generated_expressions.yml",
    universe=indefinites_universe,
    return_by_meaning=True,
)

seed_languages = random_languages(expressions, 1000, max_size=10)

def lang_complexity(language):
    return complexity(language, expressions_by_meaning)

optimizer = EvolutionaryOptimizer(
    [lang_complexity, comm_cost], expressions, 1000, 3, 50, 10
)
result = optimizer.fit(seed_languages)
```

Step 2: Estimating Pareto Frontier

- `estimate_pareto`:

```
expressions, expressions_by_meaning = read_expressions(
    "indefinites/outputs/generated_expressions.yml",
    universe=indefinites_universe,
    return_by_meaning=True,
)

seed_languages = random_languages(expressions, 1000, max_size=10)

def lang_complexity(language):
    return complexity(language, expressions_by_meaning)

optimizer = EvolutionaryOptimizer(
    [lang_complexity, comm_cost], expressions, 1000, 3, 50, 10
)
result = optimizer.fit(seed_languages)
```

from `altk.language.sampling`

Step 2: Estimating Pareto Frontier

- `estimate_pareto`:

```
expressions, expressions_by_meaning = read_expressions(
    "indefinites/outputs/generated_expressions.yml",
    universe=indefinites_universe,
    return_by_meaning=True,
)

seed_languages = random_languages(expressions, 1000, max_size=10)

def lang_complexity(language):
    return complexity(language, expressions

optimizer = EvolutionaryOptimizer(
    [lang_complexity, comm_cost], expressio
)
result = optimizer.fit(seed_languages)
```

from `altk.language.sampling`

- `outputs/dominating_languages.yml`:

```
- comm_cost: 0.5218750000000001
  complexity: 3
  expressions:
  - R+
  - R-
  - SE-
  name: dominating-0
  type: dominant
- comm_cost: 0.1625000000000003
  complexity: 8
  expressions:
```

Step 3: Measuring Natural Languages

- Read in languages, read in expressions by meaning, measure, write
- `measure_natural_languages`:

```
write_languages(
    natural_languages,
    "indefinites/outputs/natural_languages.yml",
    {
        "name": lambda _, lang: lang.name,
        "type": lambda _1, _2: "natural",
        "lot_expressions": lambda _, lang: [
            str(expressions_by_meaning[expr.meaning]) for expr in lang.expressions
        ],
        "complexity": lambda _, lang: complexity(lang, expressions_by_meaning),
        "comm_cost": lambda _, lang: comm_cost(lang),
    },
)
```

Steps 4-5: Combine and Visualize

- `combine_data`: writes natural, dominant, explored languages in one table
  - `outputs/combined_data.csv`

```
name,comm_cost,complexity,type
explored-0,0.6313946759259261,13,explored
explored-1,0.45329861111111125,13,explored
explored-2,0.6292708333333334,4,explored
explored-3,0.3983506944444448,17,explored
explored-4,0.6804976851851853,6,explored
explored-5,0.6337962962962964,10,explored
```

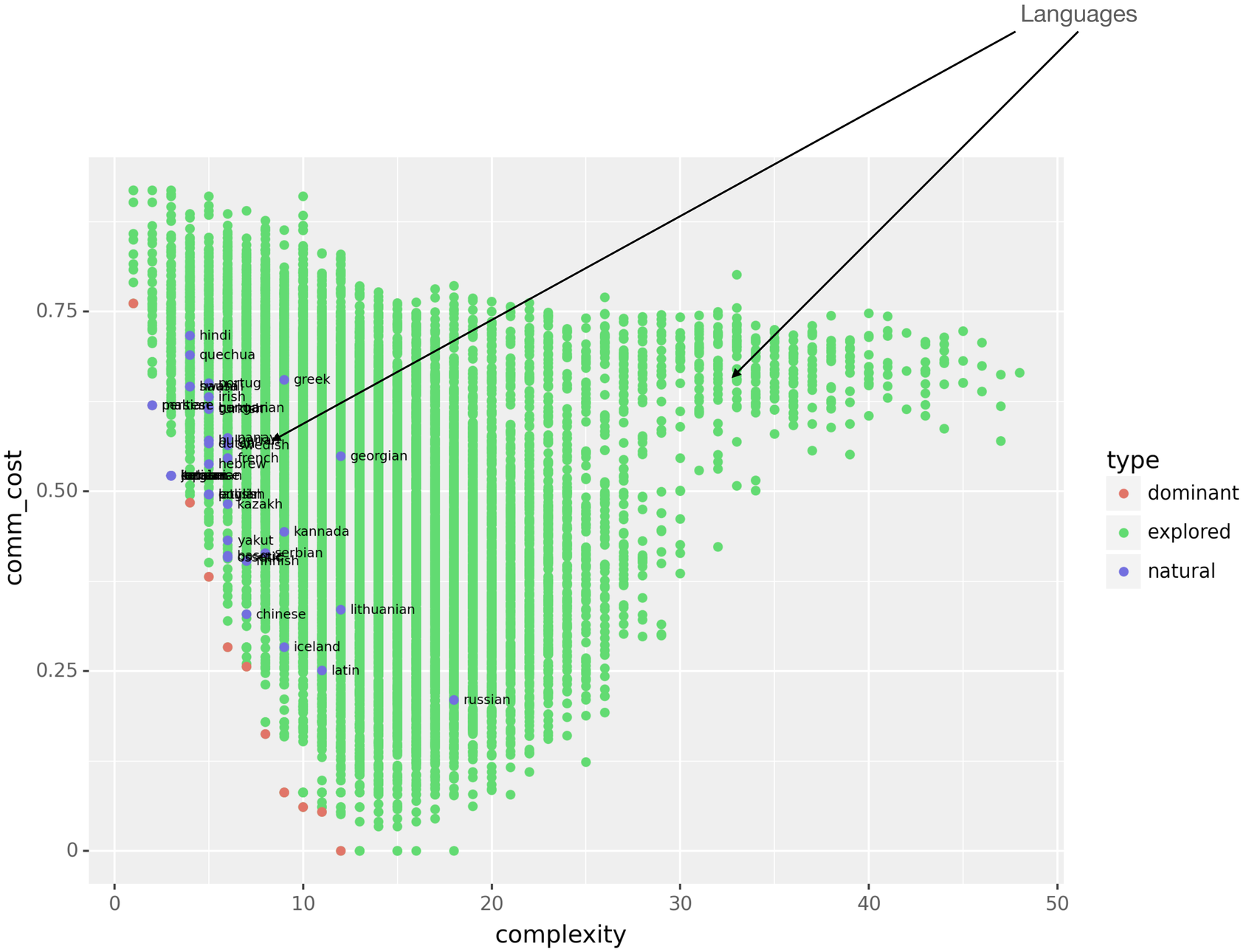
- `analyze`: just produces the plot (for now)
  - Using `plotnine`, a quite faithful port of `ggplot`

Wrapping up

Recap


comm_cost
0.75
0.50
0.25
0
0
10
20
30
40
50
complexity
hindi
quechua
greek
irish
georgian
swedish
french
hebrew
kazakh
kannada
yakut
serbian
chinese
lithuanian
iceland
latin
russian
type
dominant
explored
natural

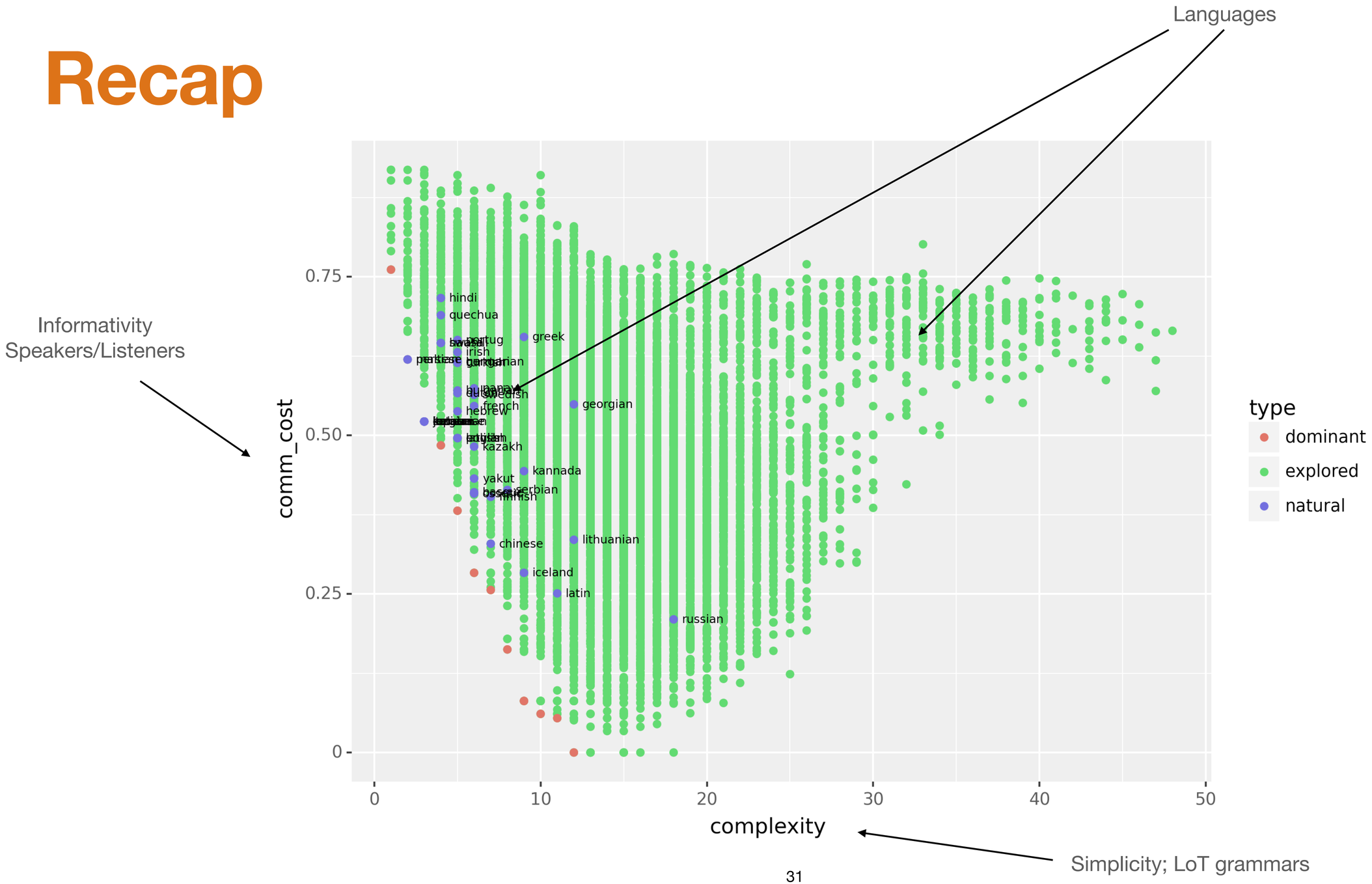
Recap


Languages
comm_cost
0.75
0.50
0.25
0
0
10
20
30
40
50
complexity
hindi
quechua
greek
irish
georgian
swedish
french
hebrew
kazakh
kannada
yakut
serbian
lithuanian
chinese
iceland
latin
russian
type
dominant
explored
natural

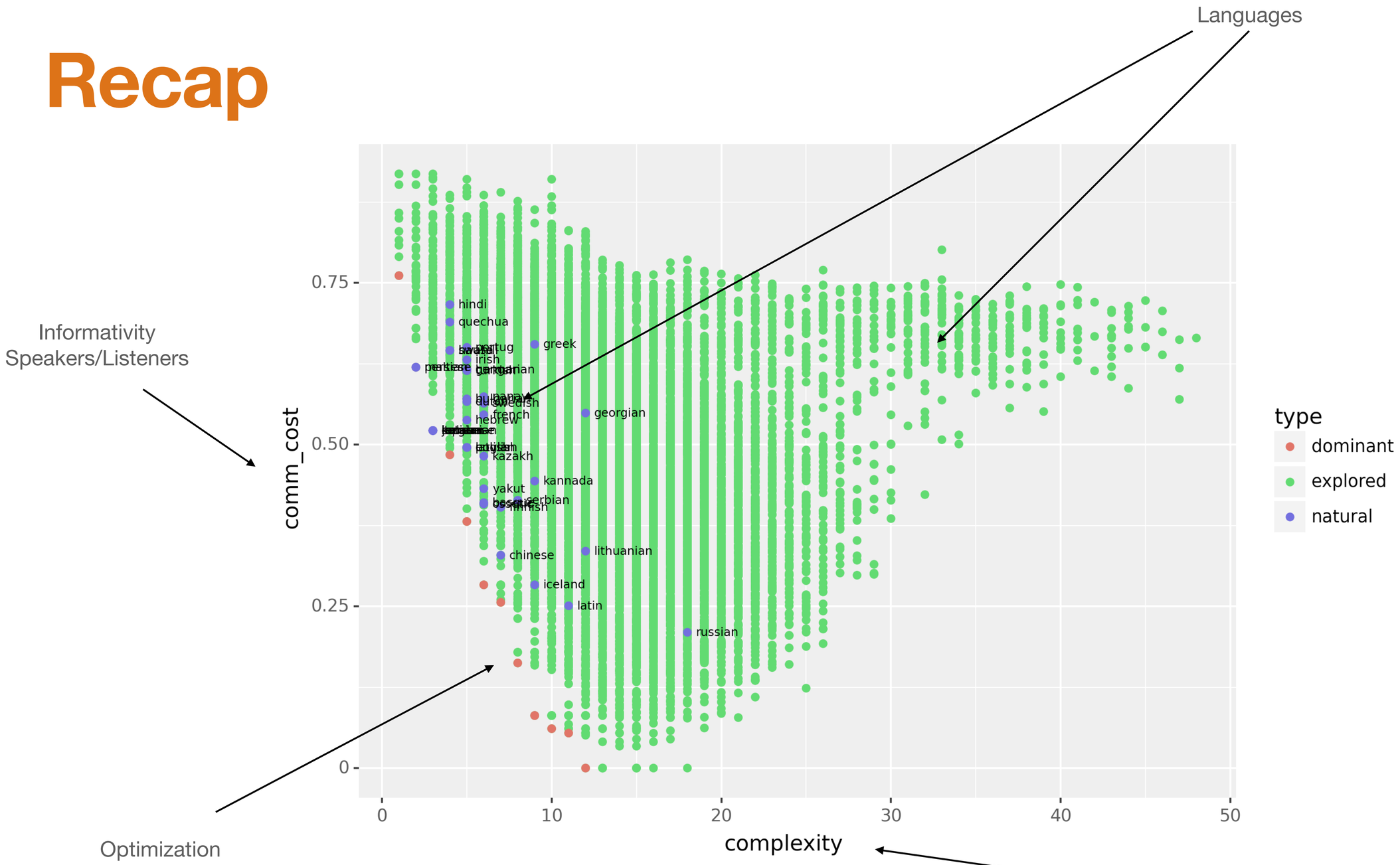
Recap


Languages
Informativity
Speakers/Listeners
comm_cost
complexity
0
0.25
0.50
0.75
0
10
20
30
40
50
hindi
quechua
greek
irish
georgian
swedish
french
hebrew
kazakh
kannada
yakut
serbian
chinese
lithuanian
iceland
latin
russian
type
dominant
explored
natural

Recap


Languages
Informativity
Speakers/Listeners
comm_cost
0.75
0.50
0.25
0
0
10
20
30
40
50
complexity
hindi
quechua
greek
irish
georgian
hebrew
french
kazakh
kannada
yakut
chinese
lithuanian
iceland
latin
russian
type
dominant
explored
natural
Simplicity; LoT grammars

Recap


Languages
Informativity
Speakers/Listeners
Optimization
Simplicity; LoT grammars
comm_cost
complexity
0.75
0.50
0.25
0
0
10
20
30
40
50
hindi
quechua
greek
irish
georgian
hebrew
french
kazakh
kannada
yakut
chinese
lithuanian
iceland
latin
russian
type
dominant
explored
natural

The Artificial Language ToolKit (ALTK)

- The Artificial Language ToolKit:
  - Provides useful abstractions for efficient communication analyses
  - Intended to lower barrier to entry
  - Pieces that were specific to indefinites:
    - Data (referents, nat lang expressions, prior)
    - LoT grammar
    - Some reading/writing utilities
- Still in very active development: input, suggestions, contributions are very welcome!

Looking Ahead

- Not covered:
  - Information Bottleneck (integrated with `embo`)
  - Configuration files for better experiment management (not really ALTK specific)
- Immediate future work:
  - Read grammars from text files (e.g. yaml)
  - Add plotting to / clean up analysis in the core library
  - … (let us know)
- Longer term: learning, other complexities, …

Special Thanks

- Thanks to:
  - Wouter Posdijk
  - Charlie Guo
- Especially!
  - Nathaniel Imel

Thank you! Questions?