NL2Type: Inferring Types from Natural Language Information

Rabee Sohail Malik, Jibesh Patra, Michael Pradel

TU Darmstadt / Facebook

Why Infer Types?

- **Dynamically typed languages:**
  Extremely popular
- **Lack of type annotations:**
  - Type errors
  - Hard-to-understand APIs
  - Poor IDE support
- **Gradual types to the rescue**

Why Infer Types?

- **Dynamically typed languages**: **Extremely popular**
- **Lack of type annotations**:
  - Type errors
  - Hard-to-understand APIs
  - Poor IDE support
- **Gradual types to the rescue**

But: Annotating types is painful

Running Example

Type signature of this function?

```
/** Calculates the area of a rectangle.
  * @param length The length of the rectangle.
  * @param breadth The breadth of the rectangle.
  * @returns The area of the rectangle in meters.
*/
getArea: function(length, breadth) {
  ...
}
```

Running Example

Type signature of this function?

```
/** Calculates the area of a rectangle.
  * @param length The length of the rectangle.
  * @param breadth The breadth of the rectangle.
  * @returns The area of the rectangle in meters.
*/
getArea: function(length, breadth) {
  ...
}
```

Identifiers and comments:
Implicit type hints

NL2Type

Idea: **Predict types from natural language information**

Identifiers & comments → **NL2Type** → **Type annotations**

Natural language in code

- Usually ignored by program analyses
- But: Extremely valuable

Usage Scenario 1

Predict missing type annotations

```
/** Calculates the area of a rectangle.
  * @param length The length of the rectangle.
  * @param breadth The breadth of the rectangle.
  * @returns The area of the rectangle in meters.
  *          May also be used for squares.
*/
getArea: function(length, breadth) {
  ...
}
```

Usage Scenario 1

Predict missing type annotations

```
/** Calculates the area of a rectangle.
  * @param {number} length The length of the rectangle.
  * @param {number} breadth The breadth of the rectangle.
  * @returns {number} The area of the rectangle in meters.
  *                   May also be used for squares.
*/
getArea: function(length, breadth) {
  ...
}
```

Usage Scenario 2

Find inconsistent annotations

```
/** Calculates the area of a rectangle.
  * @param {number} length The length of the rectangle.
  * @param {string} breadth The breadth of the rectangle.
  * @returns {number} The area of the rectangle in meters.
  *                   May also be used for squares.
*/
getArea: function(length, breadth) {
  ...
}
```

Usage Scenario 2

Find inconsistent annotations

```
/** Calculates the area of a rectangle.
  * @param {number} length The length of the rectangle.
  * @param {string} breadth The breadth of the rectangle.
  * @returns {number} The area of the rectangle in meters.
  *                   May also be used for squares.
*/
getArea: function(length, breadth) {
  ...
}
```

Usage Scenario 3

Improve auto-completion and code navigation

```
area = getArea(23, 42);
name = firstName();
...
writer.appendNumber( ??? )
```

Usage Scenario 3

Improve auto-completion and code navigation

```
area = getArea(23, 42);
name = firstName();
...
writer.appendNumber( ??? )
```

Rank suggestions based on inferred types

Overview of NL2Type

Corpus of annotated functions →

- Data extraction
- ↓
- NL preprocessing
- ↓
- Data representation
- ↓
- Neural network training
- ↓

New function → **NL2Type model** → Likely type signature

Data Extraction

- **Lightweight AST-based static analysis**
- **From each function, extract:**
  - Names of function and parameters
  - Comments associated with function, parameters, and return type
  - Types of parameters and return type

Data Extraction: Example

```
/** Calculates the area of a rectangle.
  * @param {number} length The length of the rectangle.
  * @param {number} breadth The breadth of the rectangl
  * @returns {number} The area of the rectangle in mete
*/
getArea: function(length, breadth) {
  ...
}
```

Data Extraction: Example

```
/** Calculates the area of a rectangle.
  * @param {number} length The length of the rectangle.
  * @param {number} breadth The breadth of the rectangl
  * @returns {number} The area of the rectangle in mete
*/
getArea: function(length, breadth) {
  ...
}
```

Data Extraction: Example

```
/** Calculates the area of a rectangle.
  * @param {number} length The length of the rectangle.
  * @param {number} breadth The breadth of the rectangl
  * @returns {number} The area of the rectangle in mete
*/
getArea: function(length, breadth) {
  ...
}
```

Data Extraction: Example

```
/** Calculates the area of a rectangle.
  * @param {number} length The length of the rectangle.
  * @param {number} breadth The breadth of the rectangl
  * @returns {number} The area of the rectangle in mete
*/
getArea: function(length, breadth) {
  ...
}
```

NL Preprocessing

Challenges

- Huge vocabulary
- Different variants of same word
- Uninformative words

NL Preprocessing

Challenges

- Huge vocabulary
- Different variants of same word
- Uninformative words

Addressed by

- Tokenizing identifiers: getArea $\rightarrow$ get, area
- Lemmatizing words: Calculates $\rightarrow$ calculate
- Removing stop words: the, of, etc.

Data Representation: NL Words

How to feed the NL data into a neural network?

”area” → Learned embedding → [0.32, 2.17, ..., 1.89]

- Map each word into a compact vector (length=100)
- Embeddings encode semantic similarity
- Different embeddings for identifiers and comments

Data Representation: Types

How to represent types as vectors?

number → **One-hot encoding** → **[0, 0, ..., 1, 0]**

- Encode most frequent types as one-hot vectors (default: 1000 types)
- Infrequent types encoded as ”other”
  → Think: ”don’t know”

Training the Neural Network

Type as one-hot vector

↑

Recurrent neural network

↑

Sequence of embeddings of NL info

Training the Neural Network

Type as one-hot vector

↑

Recurrent neural network

↑

Sequence of embeddings of NL info

Flag: Return type + Words in fct. comment + Words in param. names + Words in fct. name + Words in return comment

Shape: 43x100

Training the Neural Network

Type as one-hot vector

↑

Recurrent neural network

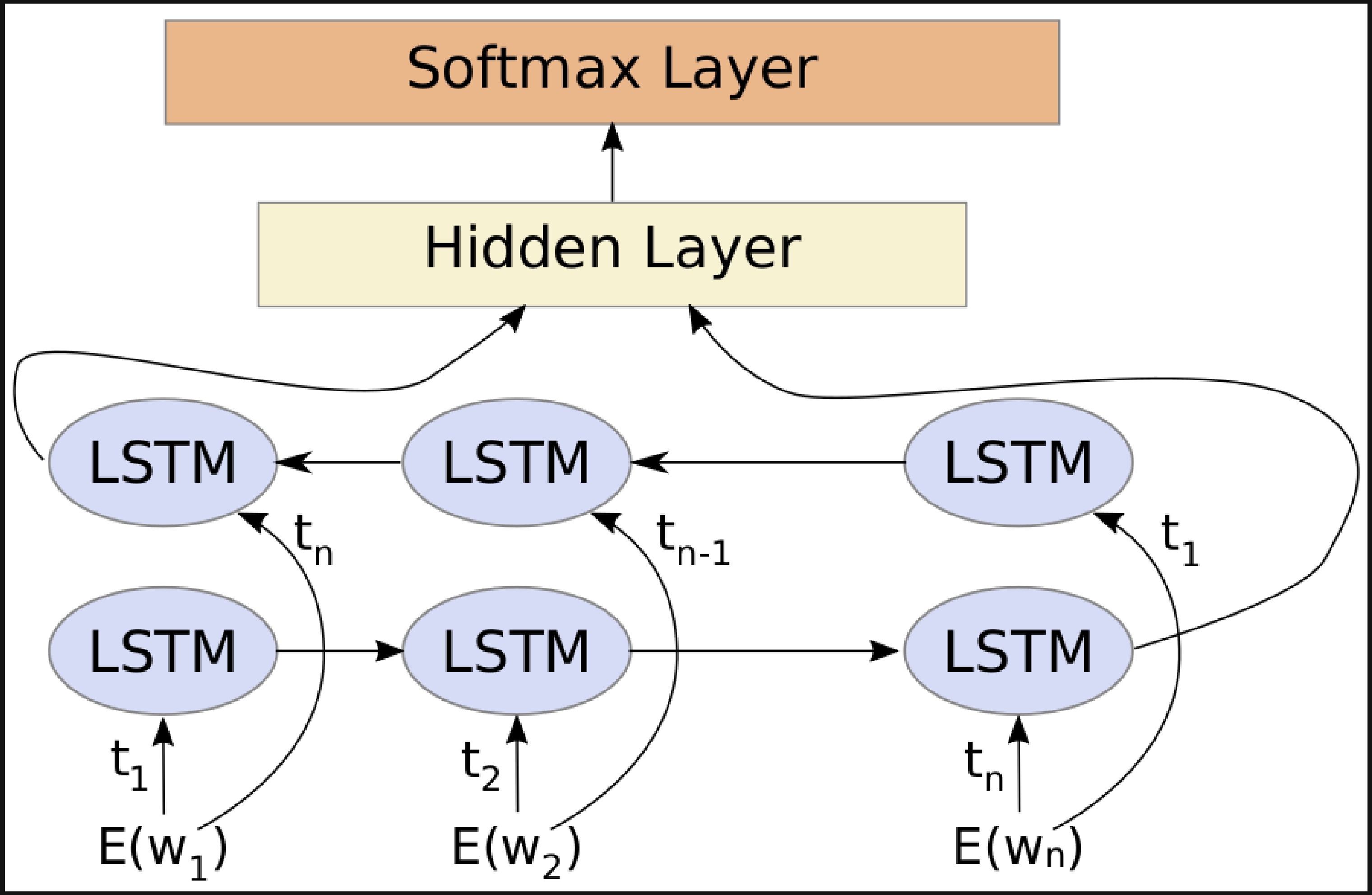
↑

Sequence of embeddings of NL info

Flag: Param. type + Words in param. comment + Padding + Words in param. name + Padding

Shape: 43x100

Bi-directional RNN



hidden layer size: 256, batch size: 256, epochs: 12,
dropout: 20%, loss: categorical cross entropy, optimizer: Adam

Prediction

Softmax: Probabilities of different types

↑

Recurrent neural network

↑

Sequence of embeddings of NL info

Prediction

Softmax: Probabilities of different types

↑

Recurrent neural network

↑

Sequence of embeddings of NL info

How confident is the model in a prediction?

- Naive approach: Rank by probability
- Problem: Model is overconfident

Prediction

Softmax: Probabilities of different types

↑

Recurrent neural network

↑

Sequence of embeddings of NL info

How confident is the model in a prediction?

- Better approach:
  Dropout during prediction [Gal, ICML'16]

Implementation

- Data extraction: Parser of JSDoc tool
- Preprocessing: NLTK library
- Embedding learning:
  Word2Vec by gensim
- Neural network:
  Keras and TensorFlow

Evaluation: Data

- **162k JavaScript files**
  - JS150 corpus [Raychev, POPL'16]
  - Popular libraries from cdnjs.com
- **618k data points**
  - 31% return types, 69% parameter types
  - 80% have a comment

Evaluation: Metrics

- **Precision and recall in top-k predictions**

$$precision = \frac{pred_{corr}}{pred_{all}} \qquad recall = \frac{pred_{corr}}{data\ points}$$

- **Usefulness of inconsistencies**
  - Manually inspect predictions that differ from actual annotation

Effectiveness of Prediction

Approach	Top-1		
	Prec	Rec	F1
NL2Type	84.1	78.9	81.4

Effectiveness of Prediction

Approach	Top-1			Top-3			Top-5		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
NL2Type	84.1	78.9	81.4	93.0	87.3	90.1	95.5	89.6	92.5

Effectiveness of Prediction

Approach	Top-1			Top-3			Top-5		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
NL2Type	84.1	78.9	81.4	93.0	87.3	90.1	95.5	89.6	92.5

Scenario:
Fully automated annotation

Effectiveness of Prediction

Approach	Top-1			Top-3			Top-5		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
NL2Type	84.1	78.9	81.4	93.0	87.3	90.1	95.5	89.6	92.5

Scenario:
Semi-automated annotation
or improved IDE support

Effectiveness of Prediction

Approach	Top-1			Top-3			Top-5		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
NL2Type	84.1	78.9	81.4	93.0	87.3	90.1	95.5	89.6	92.5
No comm.	72.3	68.3	70.3	86.6	81.8	84.1	91.4	86.3	88.8
Baseline	18.5	17.3	17.9	49.0	46.0	47.4	66.3	62.3	64.2

Effectiveness of Prediction

Approach	Top-1			Top-3			Top-5		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
NL2Type	84.1	78.9	81.4	93.0	87.3	90.1	95.5	89.6	92.5
No comm.	72.3	68.3	70.3	86.6	81.8	84.1	91.4	86.3	88.8
Baseline	18.5	17.3	17.9	49.0	46.0	47.4	66.3	62.3	64.2

Use function names and parameter names only

Effectiveness of Prediction

Approach	Top-1			Top-3			Top-5		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
NL2Type	84.1	78.9	81.4	93.0	87.3	90.1	95.5	89.6	92.5
No comm.	72.3	68.3	70.3	86.6	81.8	84.1	91.4	86.3	88.8
Baseline	18.5	17.3	17.9	49.0	46.0	47.4	66.3	62.3	64.2

Predict k most frequent types

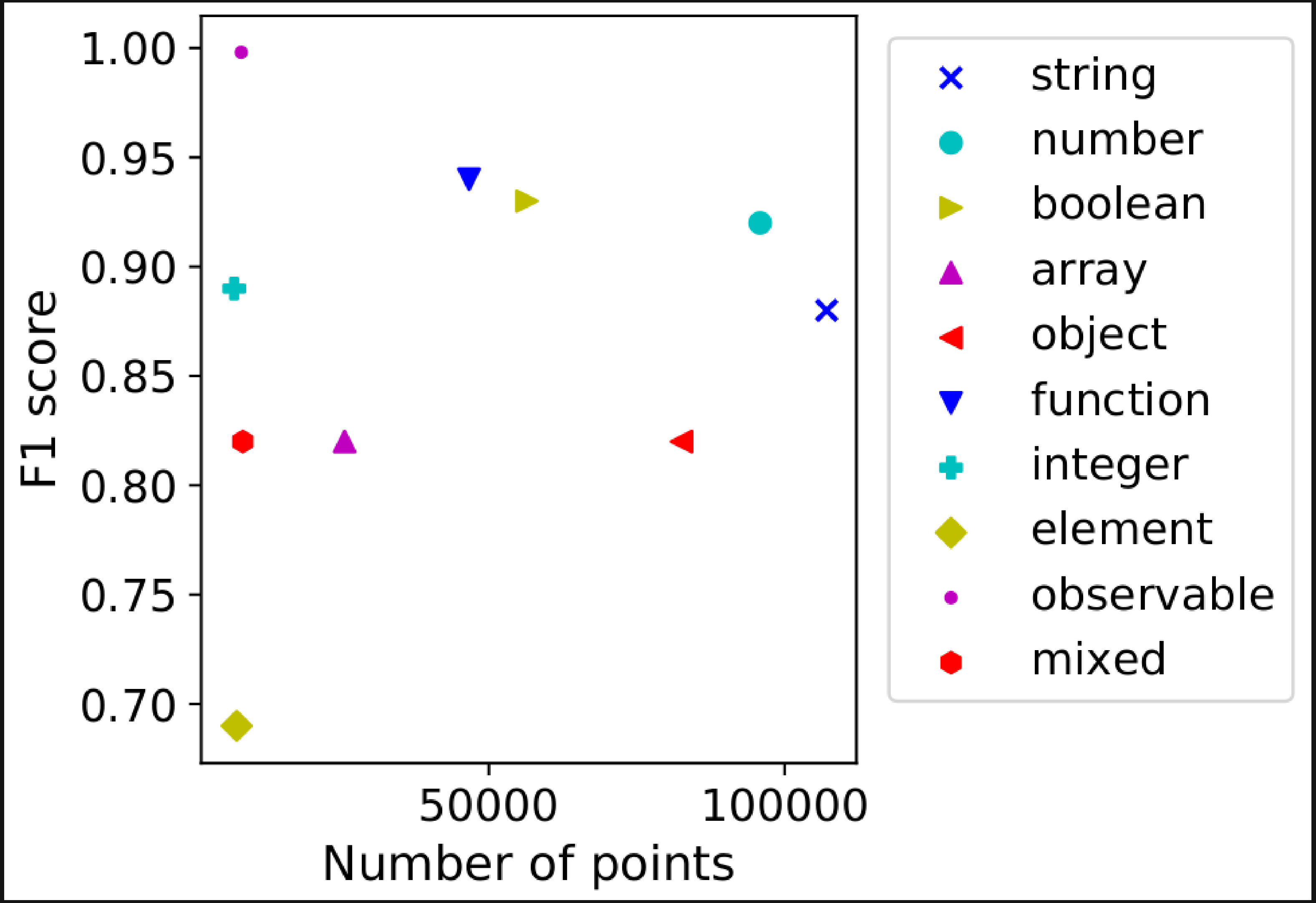
Example: Correct Prediction

```
/** Get the appropriate anchor and focus node/offset
 * pairs for IE.
 * @param {???} node
 * @return {???}
 */
function getIEOffsets(node) {
...
}
```

Example: Correct Prediction

```
/** Get the appropriate anchor and focus node/offset
 * pairs for IE.
 * @param {DOMElement} node
 * @return {object}
 */
function getIEOffsets(node) {
...
}
```

Effectiveness by Type



Comparison with Prior Work

- **JSNice: Structural relations between program elements** [Raychev, POPL'15]
  - Precision: 62.5% $\rightarrow$ 84.1%
  - Recall: 45% $\rightarrow$ 78.9%
- **DeepTyper: Seq-to-seq based on parallel corpus** [Hellendoorn, FSE'18]
  - Precision: 68.6% $\rightarrow$ 77.5% *
  - Recall: 44.0% $\rightarrow$ 44.6% *

* on a TypeScript-based corpus

Usefulness of Inconsistencies

Manual classification of top-50 warnings:

Category	Total	Percentage
Inconsistencies	25	50%
Non-standard type annotations	14	28%
Misclassifications	11	22%

Usefulness of Inconsistencies

Manual classification of top-50 warnings:

Category	Total	Percentage
Inconsistencies	25	50%
Non-standard type annotations	14	28%
Misclassifications	11	22%

78% true positives

Examples

Inconsistency: Incorrect annotation

```
/** Utility function to ensure that object properties
  * are copied by value, and not by reference
  * @param {Object} target Target object to copy
  * properties into
  * @param {Object} source Source object for the
  * proporties to copy
  * @param {string} propertyObj Object containing
  * properties names we want to loop over
*/
function deepCopyProperties(target, source, propertyObj
...
}
```

Examples

Inconsistency: Incorrect annotation

```
/** Utility function to ensure that object properties
  * are copied by value, and not by reference
  * @param {Object} target Target object to copy
  * properties into
  * @param {Object} source Source object for the
  * properties to copy
  * @param {string} propertyObj Object containing
  * properties names we want to loop over
*/
function deepCopyProperties(target, source, propertyObj
...
}
```

Examples

Non-standard type annotation

```
/** Tests to see if a point (x, y) is within a range
  * of current Point
  * @param {Numeric} x - the x coordinate of tested poi
  * @param {Numeric} y - the x coordinate of tested poi
  * @param {Numeric} radius - the radius of the vicinit
**/
near: function(x, y, radius) {
  ...
}
```

Examples

Non-standard type annotation

```
/** Tests to see if a point (x, y) is within a range
  * of current Point
  * @param {Numeric} x - the x coordinate of tested poi
  * @param {Numeric} y - the x coordinate of tested poi
  * @param {Numeric} radius - the radius of the vicinit
**/
near: function(x, y, radius) {
  ...
}
```

Examples

Incorrect prediction

```
/** Calculate the average of two 3d points
  * @param {Point3d} a
  * @param {Point3d} b
  * @return {Point3d} The average, (a+b)/2
  */
Point3d.avg = function(a, b) {
  ...
}
```

Examples

Incorrect prediction

```
/** Calculate the average of two 3d points
  * @param {Point3d} a
  * @param {Point3d} b
  * @return {Point3d} The average, (a+b)/2
  */
Point3d.avg = function(a, b) {
  ...
}
```

Efficiency

- **Data extraction**: 44ms per function
- **Preprocessing**: 23ms per function
- **Training**: 93 minutes (one-time effort)
- **Prediction**: 5ms per function

Intel Xeon E5-2650 processor with 48 cores, 64GB of memory, NVIDIA Tesla P100 GPU with 16GB of memory

Why Does It Work?

Developers use **meaningful names**

Source code is **repetitive**

Annotated code available as **training data**

Probabilistic models + NL = ♡

Conclusions

- **NL2Type: Predict types from NL info**
  - F1-score of 81.4% (top-1) to 92.5% (top-5)
  - 39/50 detected inconsistencies motivate a code improvement
- **Open challenges**
  - Integrate into development workflow
  - Long tail of types