POS tagging with the perceptron

Marco Kuhlmann
Department of Computer and Information Science

The perceptron algorithm

Eisenstein § 2.3.1

- Before the neural era, the state of the art in tagging was defined by linear classifiers with manually engineered features.
- The **perceptron algorithm** is a simple but surprisingly effective algorithm for training a linear classifier.

  fast even in pure Python, strong baseline for many tasks

- In contrast to current deep learning libraries, the perceptron algorithm can efficiently deal with sparse feature vectors.

The general linear model

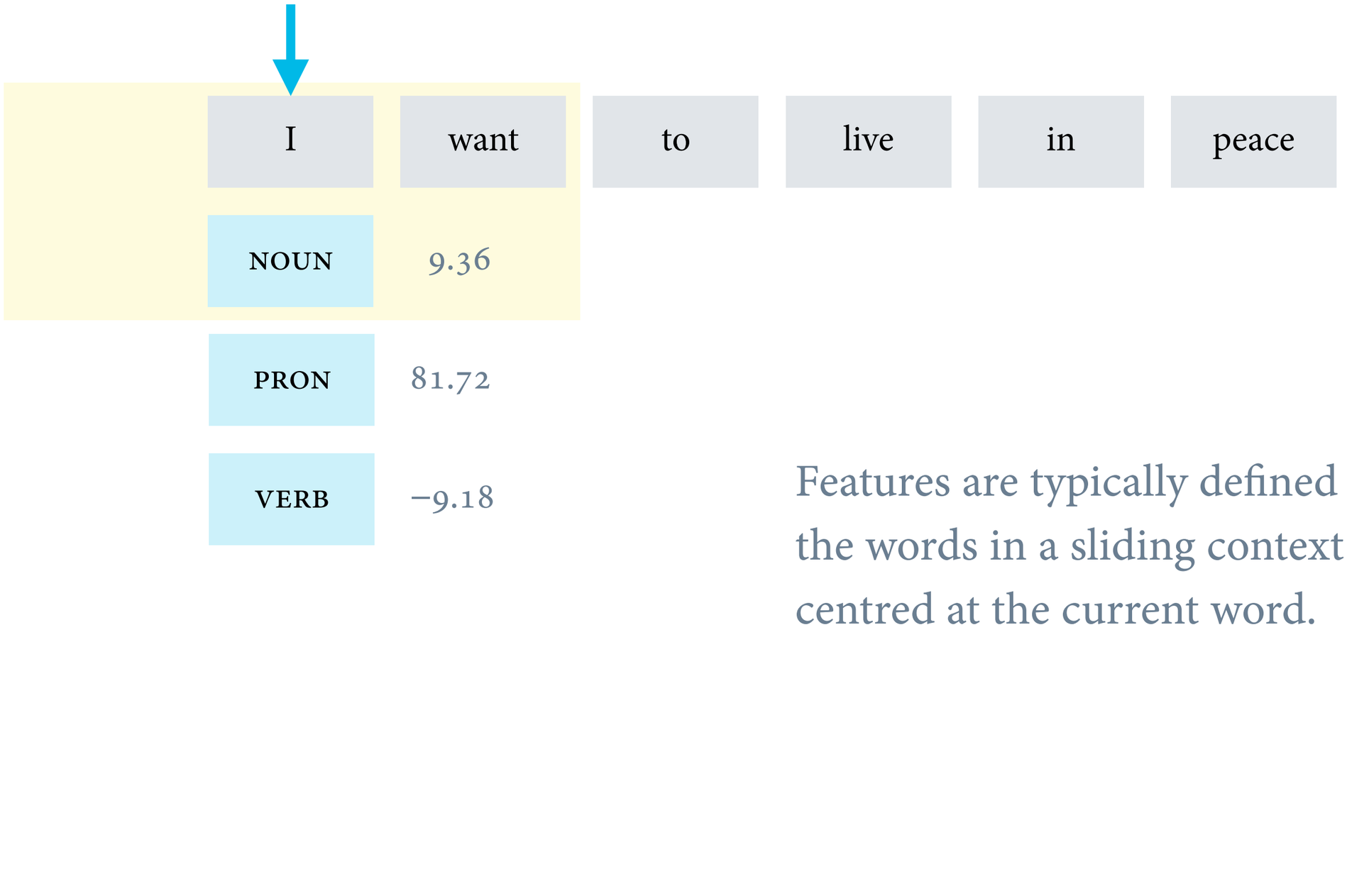
feature vector (1-by-F)

output vector (1-by-K) — $\hat{y} = xW + b$ — bias vector (1-by-K)

weight matrix (F-by-K)

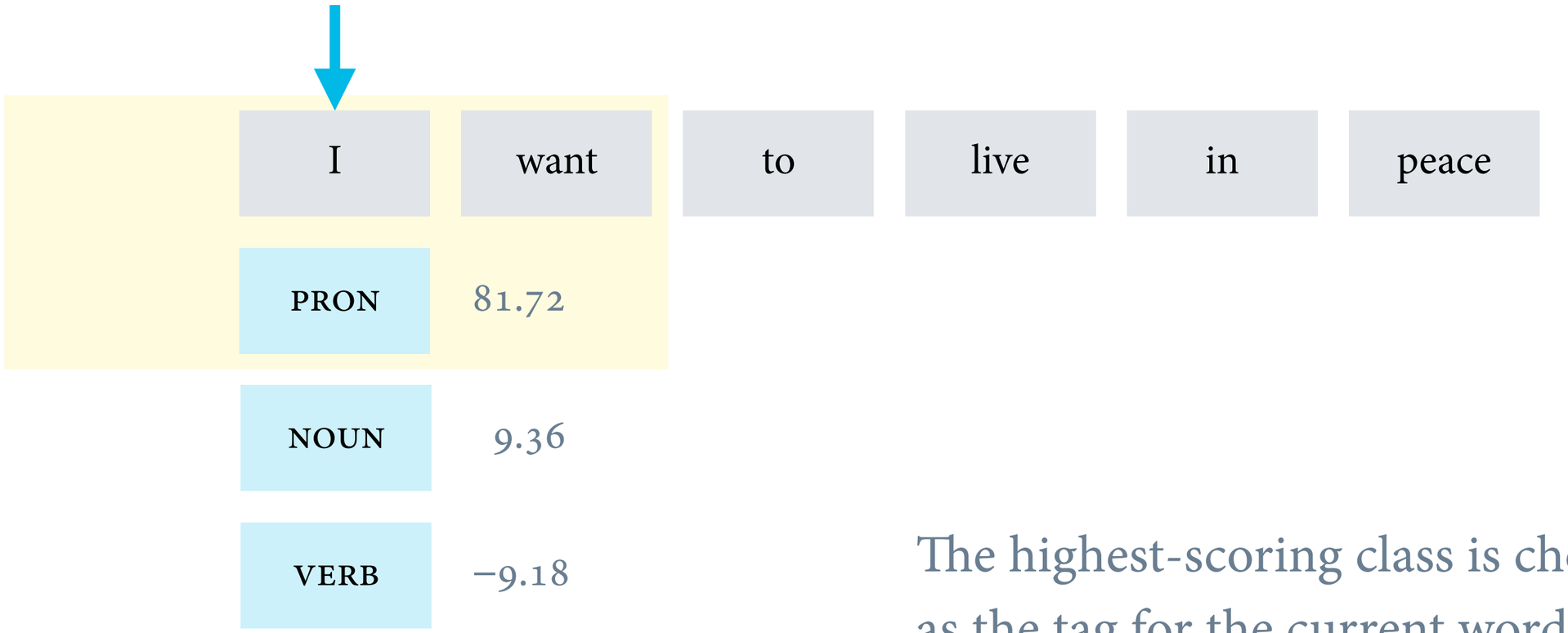
F = number of features, K = number of classes

Part-of-speech tagging with a linear classifier



Features are typically defined over the words in a sliding context window centred at the current word.

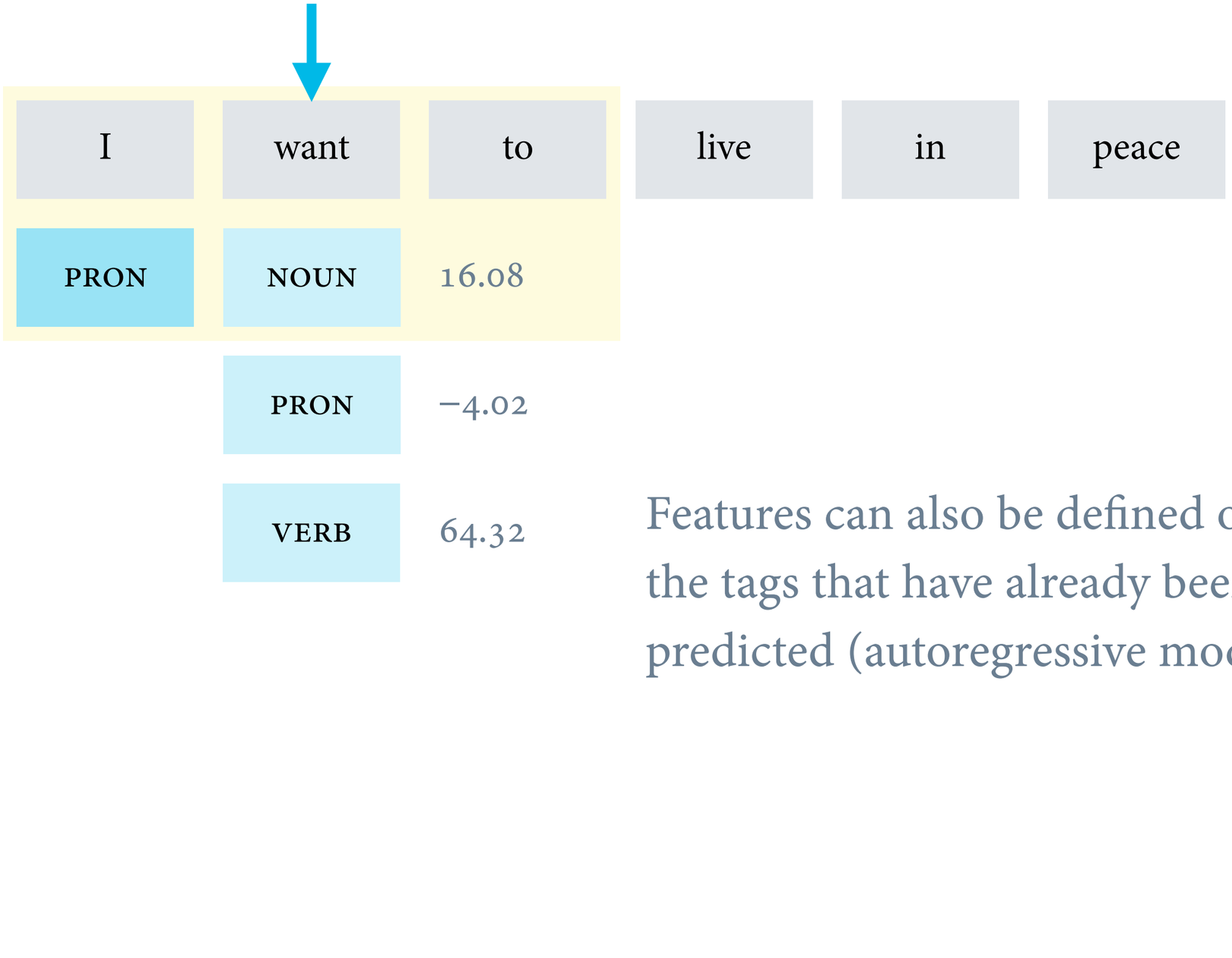
Part-of-speech tagging with a linear classifier



The highest-scoring class is chosen as the tag for the current word:

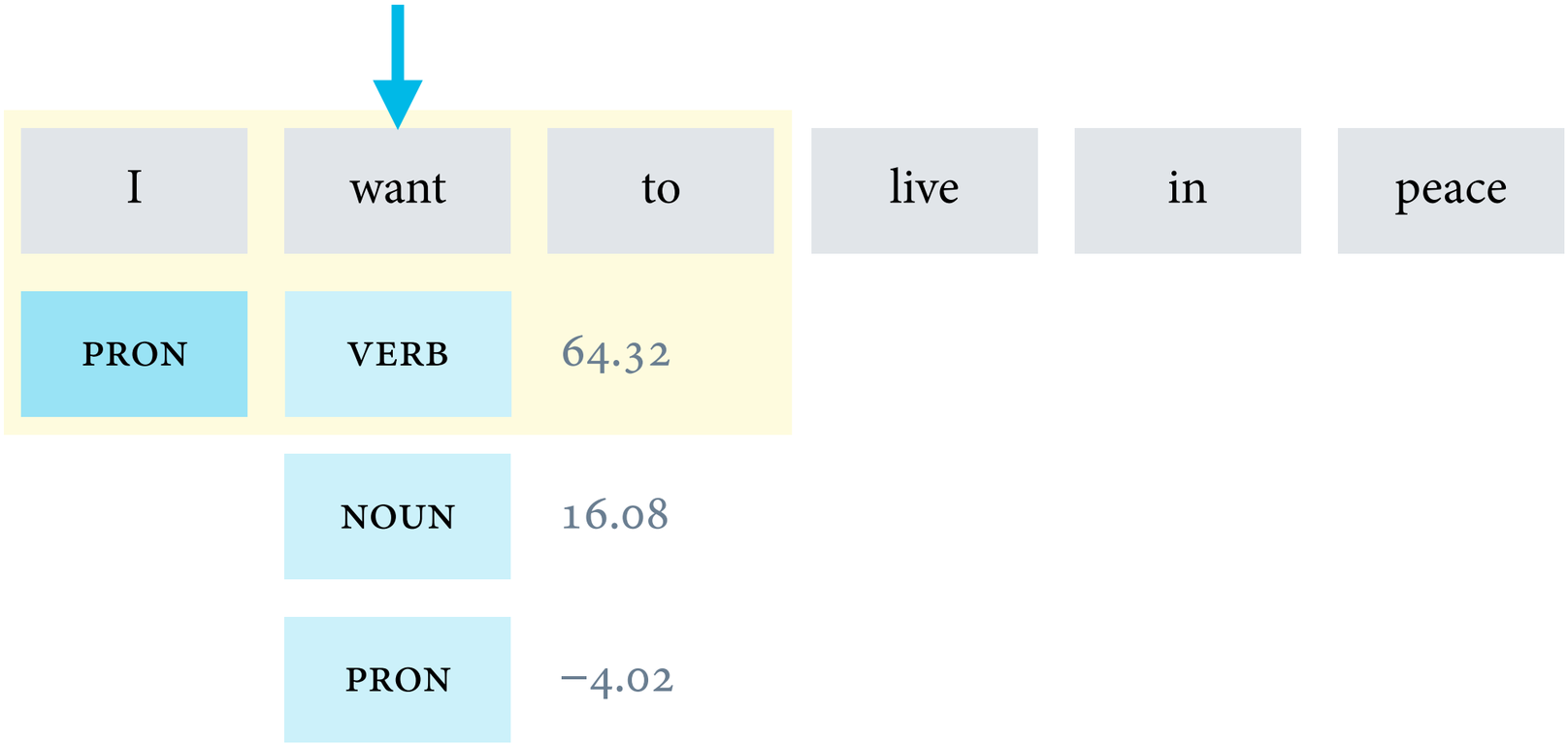
$$\hat{y} = \arg\max_{k} \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}$$

Part-of-speech tagging with a linear classifier

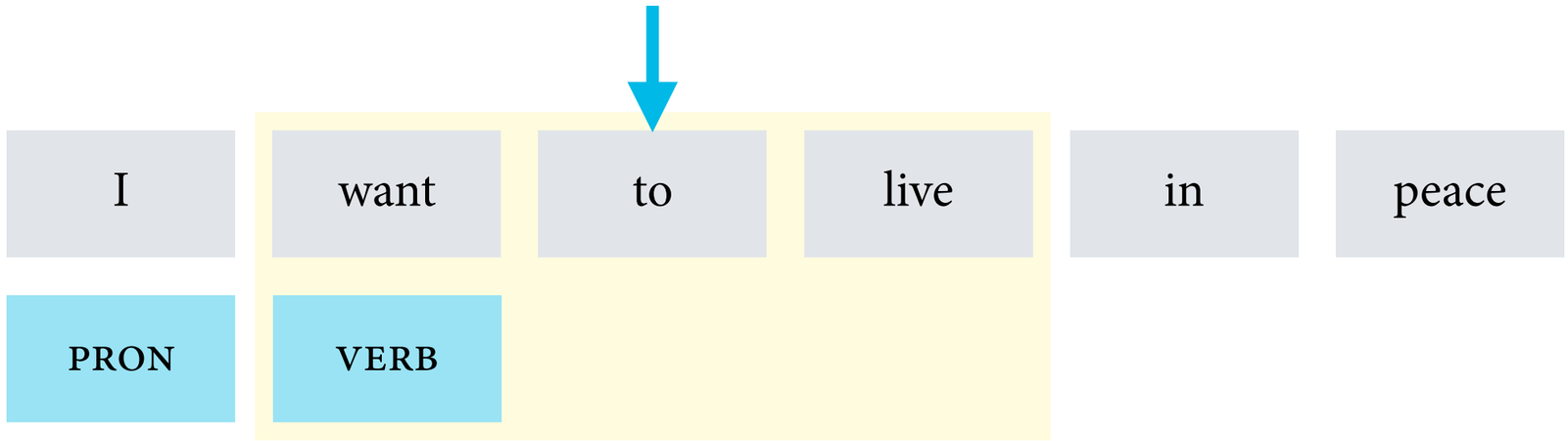


Features can also be defined over the tags that have already been predicted (autoregressive model).

Part-of-speech tagging with a linear classifier



Part-of-speech tagging with a linear classifier



Examples of features in part-of-speech tagging

- (lowercase) word form of the current token
- word forms of the preceding tokens, next tokens
- capitalisation of the current token (upper, lower, N/A)
- type of the current token (digits, letters, symbols)
- various prefixes and suffixes of the current token
- whether the current token is hyphenated
- whether the token is first or last in the sentence
- various combinations of the features above

Östling (2013)

Python implementation

```
# Features can be identified by feature ids (= vector dimensions)
x = {11: 1, 23: 1, 42: 2}

# There is one weight vector for each class
w = {11: 3.14, 23: -2.72, 42: 0}

# The following function computes the dot product x.w:
def dot(x, w):
    return sum(v * w[f] for f, v in x.items())
```

Python implementation

```
# Features can be identified by any (hashable) value!
x = {'want': 1, 'to': 1, 'live': 1}

# There is one weight vector for each class
w = {'want': 3.14, 'to': -2.72, 'live': 0}

# The computation of the dot product remains unchanged
def dot(x, w):
    return sum(v * w[f] for f, v in x.items())
```

Python implementation

```
class Linear(object):

    def __init__(self, classes):
        self.classes = classes
        self.w = {k: defaultdict(float) for k in classes}
        self.b = {k: 0.0 for k in classes}

    def forward(self, x):
        scores = {}
        for k in classes:
            scores[k] = self.bias[k]
            for f, v in x.items():
                scores[k] += v * self.w[k][f]
        return scores
```

Feature templates

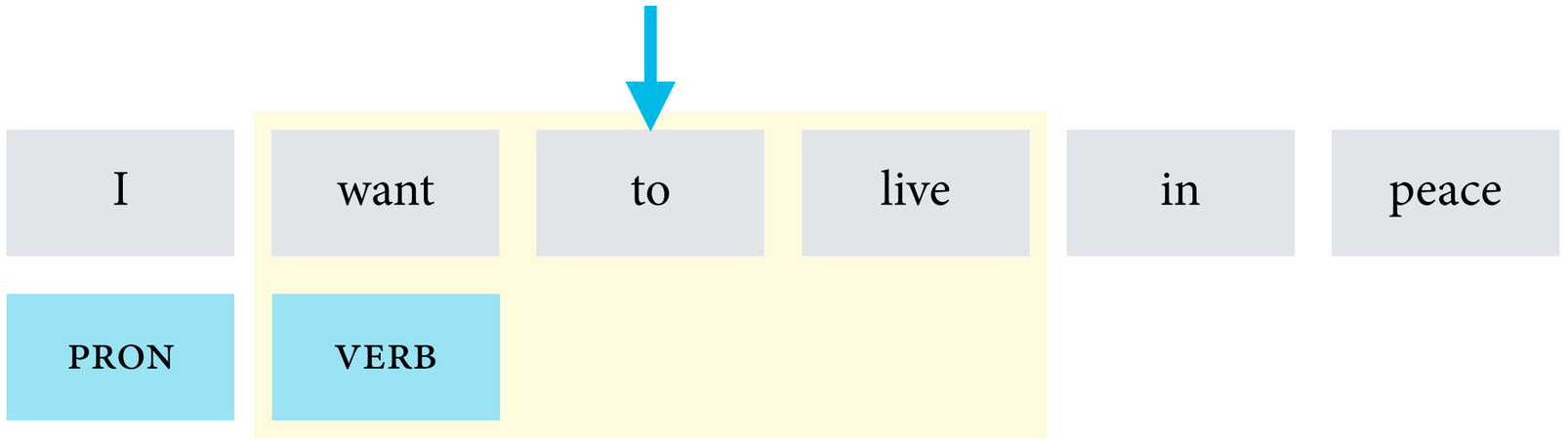
- Features can be structured with the help of **feature templates**.
- A feature template is a Boolean predicate with variables that can take concrete values from a finite set of possible values.

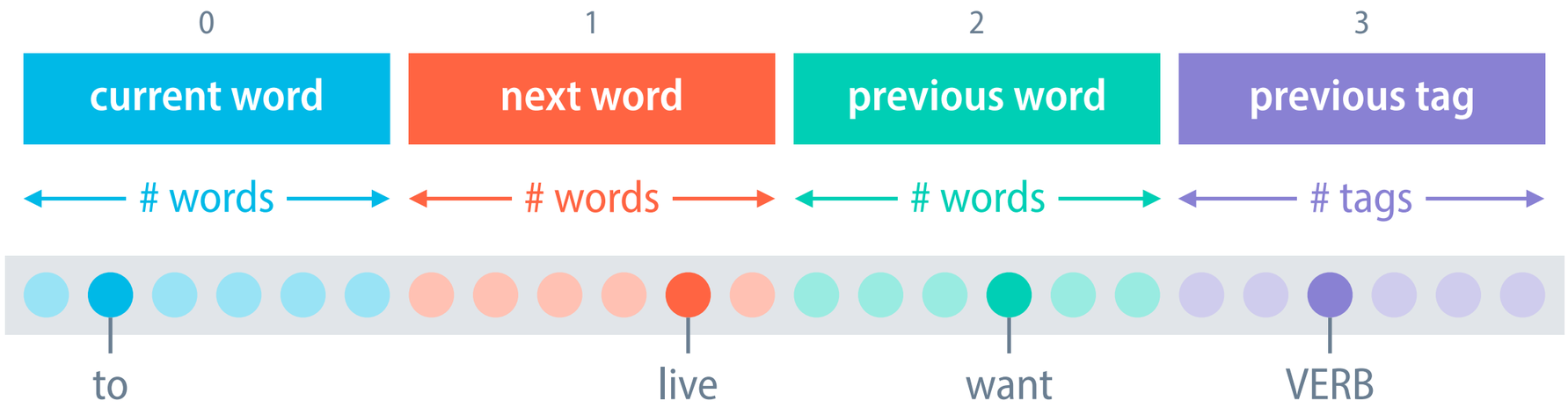
  Example: 'The current word is *w*', where *w* is a word from the vocabulary.

- Each feature template describes a finite set of binary features (1/0), one for each assignment of values to the variables.

  'The current word is *blue*.' – 'The current word is *red*.'

Feature vector from four templates





Python implementation

```
# Features can be identified by (template id, feature name)
x = {(0, 'to'): 1, (1, 'live'): 1, (2, 'want'): 1}

# There is one weight vector for each class
w = {(0, 'to'): -2.72, (1, 'live'): 0, (2, 'want'): 3.14}

# The computation of the dot product remains unchanged
def dot(x, w):
    return sum(v * w[f] for f, v in x.items())
```