Textual Data Analysis

Named entity recognition and sequence labeling

TURKUNLP
.ORG

UNIVERSITY
OF TURKU

Sequence labeling

- **Input:** text, represented as sequence of tokens
- **Output:** sequence of labels (one per token) from predefined categories
  - contrast text classification: label(s) for text as a whole
- Terminology: sequence labeling can be also called *token classification* or *sequence tagging*
  - Do not confuse *sequence classification* with *token classification* or *sequence labeling*!
  - *Label*, *class* and *tag* are largely synonyms

Sequence labeling

positive

↑

This is a good movie.

The	DET
dog	NOUN
runs	VERB
in	ADP
the	DET
park	NOUN
.	PUNCT

Don't copy, modify, resell, distribute or reverse engineer this app.

↑

Niantic grants you a limited nonexclusive nontransferable non sublicensable license to download and install a copy of the app on a mobile device and to run such copy of the app solely for your own personal noncommercial purposes. Except as expressly permitted in these terms you may not a) copy, modify or create derivative works based on the app, b) distribute transfer sublicense lease lend or rent the app to any third party, c) reverse engineer decompile or disassemble the app…

(Example from Manor and Li, 2019.)

Text classification
Document classification
Sequence classification

Sequence labeling
Sequence tagging
Token classification

Sequence to sequence
Text generation

Sequence labeling

- Token labels frequently represent either
  - Independent token categories (e.g. parts of speech)
  - Starts and ends of tokens spans (NER, chunking, etc.)

The	DET
dog	NOUN
runs	VERB
in	PREP
the	DET
park	NOUN
.	PUNCT

New	B-GPE	} New York / GPE
York	I-GPE	
is	O	
in	O	
the	O	
United	B-GPE	} United States / GPE
States	I-GPE	

TURKUNLP .ORG

UNIVERSITY OF TURKU

Sequence labeling

- **Examples:**
  - Part-of-speech tagging (labels: NOUN, VERB, …)
  - Named entity recognition (labels: O, B-PER, I-PER, …)
  - Chunking (e.g. shallow parsing; labels: B-NP, I-VP, …)
  - Span marking e.g. for question answering (labels: I, O)

Tasks: part of speech tagging

Assign each word a part of speech (POS) tag (noun, verb, etc.)

- POS tags used in older corpora varied considerably e.g. by language
- Coarse-grained "universal" tagsets common in recent work

https://universaldependencies.org/u/pos/index.html

Example: Universal Dependencies POS tags

```
ADJ: adjective
ADP: adposition
ADV: adverb
AUX: auxiliary
CCONJ: coordinating conjunction
DET: determiner
INTJ: interjection
NOUN: noun
NUM: numeral
PART: particle
PRON: pronoun
PROPN: proper noun
PUNCT: punctuation
SCONJ: subordinating conjunction
SYM: symbol
VERB: verb
X: other
```

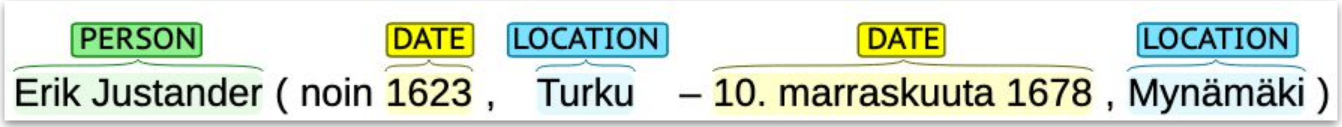
Tasks: Named Entity Recognition

Identify **token spans** constituting **mentions of names** and assign them types

- Often extended to include also mentions of e.g. as times and dates
- Note: names frequently span **multiple tokens** (contrast POS)

Span start and extent typically marked using IOB (aka BIO) tags or a variation such as IOBES (adds [E]nd, [S]ingle)

```
Barack B-Person
Obama  I-Person
was    O
born   O
in     O
Hawaii B-Location
```



UNIVERSITY OF TURKU

Named Entity Recognition: BIO tags

Begin-In-Out (BIO, aka IOB) tagging is frequently used to represent annotation that marks (non-overlapping) sequences of tokens

- Begin: start of annotated span
- In: token inside annotated span
- Out: not part of annotated span

Also reduced form IO (In-Out) and extended form IOBES (+End-Single)

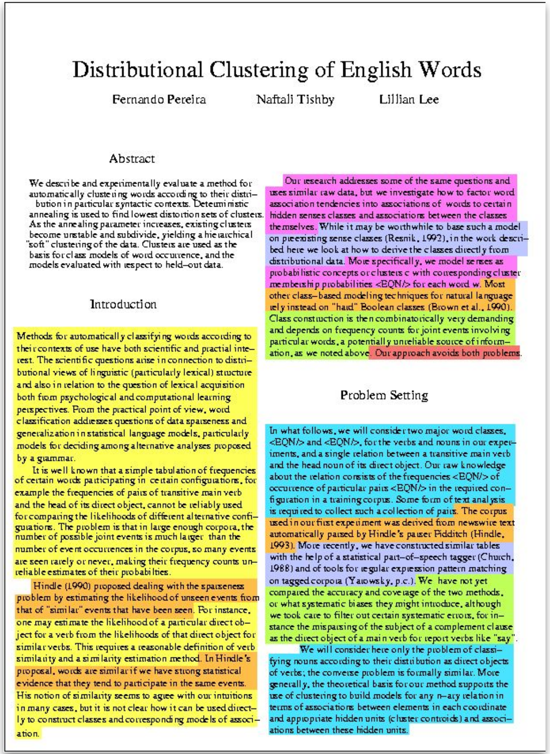
Example: named entity recognition

```
Barack B-Person
Obama  I-Person
was    O
born   O
in     O
Hawaii B-Location
```

Tasks: Span marking / classification

IOB tagging can be applied to mark any **continuous**, **non-overlapping** spans of tokens and assign them to categories

- Phrases (chunks)
- Argumentative zones →
- Semantic roles
- Hedged claims (e.g. “may …”)
- …



Distributional Clustering of English Words

Fernando Pereira Naftali Tishby Lillian Lee

Abstract

We describe and experimentally evaluate a method for automatically clustering words according to their distribution in particular syntactic contexts. Deterministic annealing is used to find lowest distortion sets of clusters. As the annealing parameter increases, existing clusters become unstable and subdivide, yielding a hierarchical "soft" clustering of the data. Clusters are used as the basis for class models of word occurrence, and the models evaluated with respect to held–out data.

Introduction

Methods for automatically classifying words according to their contexts of use have both scientific and practial interest. The scientific questions arise in connection to distributional views of linguistic (particularly lexical) structure and also in relation to the question of lexical acquisition both from psychological and computational learning perspectives. From the practical point of view, word classification addresses questions of data sparseness and generalization in statistical language models, particularly models for deciding among alternative analyses proposed by a grammar.

It is well known that a simple tabulation of frequencies of certain words participating in certain configurations, for example the frequencies of pairs of transitive main verb and the head of its direct object, cannot be reliably used for comparing the likelihoods of different alternative configurations. The problem is that in large enough corpora, the number of possible joint events is much larger than the number of event occurrences in the corpus, so many events are seen rarely or never, making their frequency counts unreliable estimates of their probabilties.

Hindle (1990) proposed dealing with the sparseness problem by estimating the likelihood of unseen events from that of "similar" events that have been seen. For instance, one may estimate the likelihood of a particular direct object for a verb from the likelihoods of that direct object for similar verbs. This requires a reasonable definition of verb similarity and a similarity estimation method. In Hindle's proposal, words are similar if we have strong statistical evidence that they tend to participate in the same events. His notion of similarity seems to agree with our intuitions in many cases, but it is not clear how it can be used directly to construct classes and corresponding models of association.

Our research addresses some of the same questions and uses similar raw data, but we investigate how to factor word association tendencies into associations of words to certain hidden senses classes and associations between the classes themselves. While it may be worthwhile to base such a model on preexisting sense classes (Resnik, 1992), in the work described here we look at how to derive the classes directly from distributional data. More specifically, we model senses as probabilistic concepts or clusters c with corresponding cluster membership probabilities <EQN/> for each word w. Most other class–based modeling techniques for natural language rely instead on "hard" Boolean classes (Brown et al., 1990). Class construction is then combinatorically very demanding and depends on frequency counts for joint events involving particular words, a potentially unreliable source of information, as we noted above. Our approach avoids both problems.

Problem Setting

In what follows, we will consider two major word classes, <EQN/> and <EQN/>, for the verbs and nouns in our experiments, and a single relation between a transitive main verb and the head noun of its direct object. Our raw knowledge about the relation consists of the frequencies <EQN/> of occurrence of particular pairs <EQN/> in the required configuration in a training corpus. Some form of text analysis is required to collect such a collection of pairs. The corpus used in our first experiment was derived from newswire text automatically parsed by Hindle's parser Fidditch (Hindle, 1993). More recently, we have constructed similar tables with the help of a statistical part–of–speech tagger (Church, 1988) and of tools for regular expression pattern matching on tagged corpora (Yarowsky, p.c.). We have not yet compared the accuracy and coverage of the two methods, or what systematic biases they might introduce, although we took care to filter out certain systematic errors, for instance the misparsing of the subject of a complement clause as the direct object of a main verb for report verbs like "say".

We will consider here only the problem of classifying nouns according to their distribution as direct objects of verbs; the converse problem is formally similar. More generally, the theoretical basis for our method supports the use of clustering to build models for any n–ary relation in terms of associations between elements in each coordinate and appropriate hidden units (cluster centroids) and associations between these hidden units.

Figure from https://www.cl.cam.ac.uk/~sht25/az.html

Tasks: Character sequences

Sequence labeling in NLP not limited to *token* sequences

Example: joint tokenization and sentence segmentation with labels

- `token-break`: token ends after character
- `sentence-break`: sentence ends after character
- `inside`: no break after character

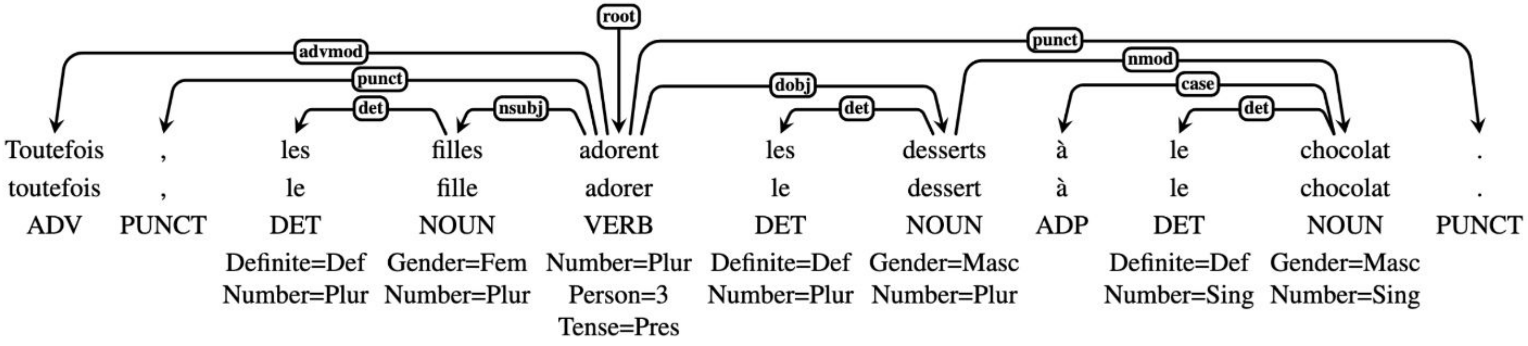
Note that pre-trained models cannot label smaller items than their subwords!

```
INPUT: Is it you?

I      inside
s      token-break
       token-break
i      inside
t      token-break
       token-break
y      inside
o      inside
u      token-break
?      sentence-break
```

UNIVERSITY OF TURKU

Part of Speech (POS) datasets

- Universal Dependencies: syntactic annotation for 100+ languages (https://universaldependencies.org/)



Nivre et al. (2016) Universal Dependencies v1: A Multilingual Treebank Collection

NER datasets

- CoNLL’02/03: Spanish, Dutch, English, German
  - PER(SON), LOC(ATION), ORG(ANIZATION), MISC
- OntoNotes: English, Chinese, Arabic:
  - PER, LOC, ORG, FAC(ILITY), GPE (geopolitical entity), PRODUCT, LANGUAGE, LAW, DATE, MONEY, etc.
- Turku NER corpus / TurkuONE: Finnish
  - Turku NER includes 6 entity types (PER, ORG, LOC, PRO, EVENT, DATE), TurkuONE increases data and extends to 18 types

Tjong Kim Sang and De Meulder (2003): Introduction to the CoNLL-2003 Shared Task
Hovy et al. (2006) OntoNotes: The 90% Solution
Luoma et al. (2020) A Broad-coverage Corpus for Finnish Named Entity Recognition
Luoma et al. (2021) Fine-grained Named Entity Annotation for Finnish

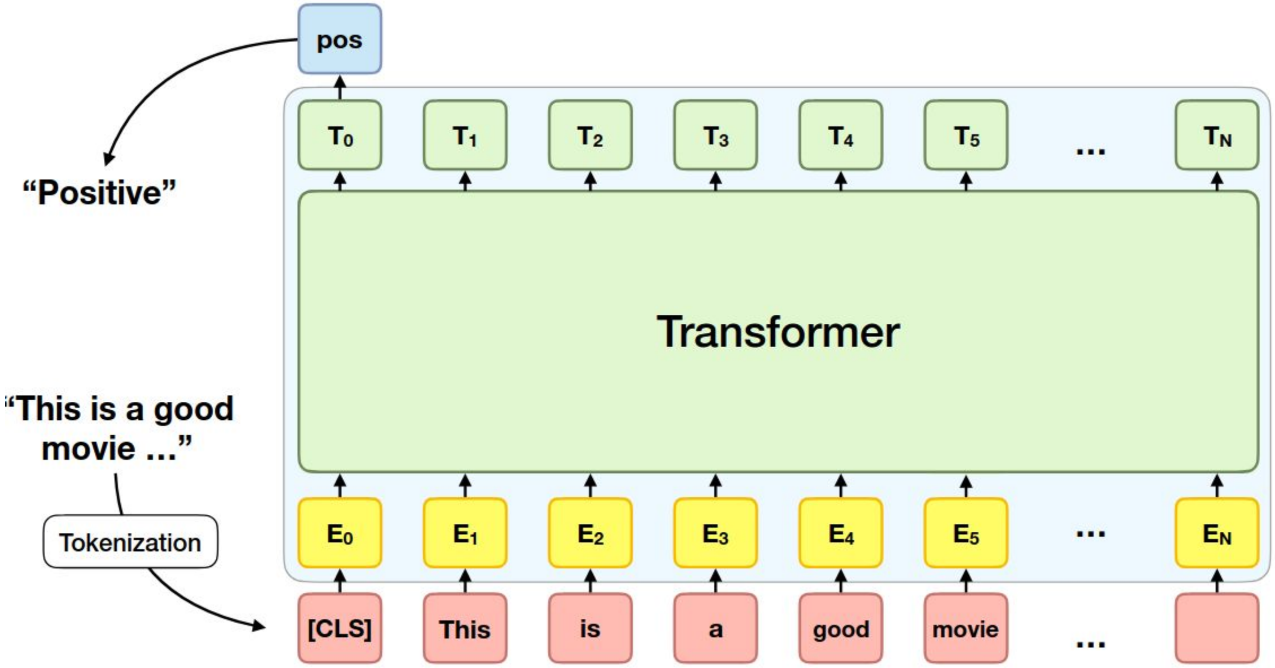
Data notebook

- https://github.com/TurkuNLP/textual-data-analysis-course/blob/main/sequence_labeling_dataset_examples.ipynb

Sequence labeling: methods

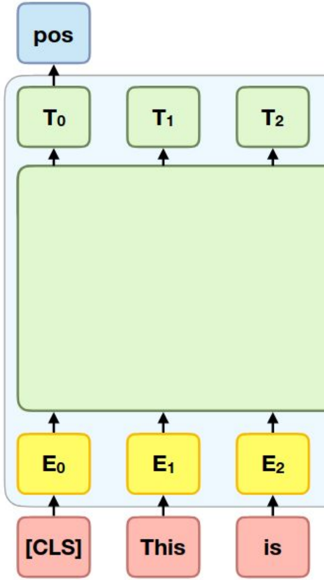
- Rule-based systems
- (Some) Supervised machine learning approaches:
  - Hidden Markov Models
  - Conditional Random Fields
  - Convolutional and Recurrent neural nets
  - Transformers
- Zero/few-shot approaches

Recap: Classification with transformers



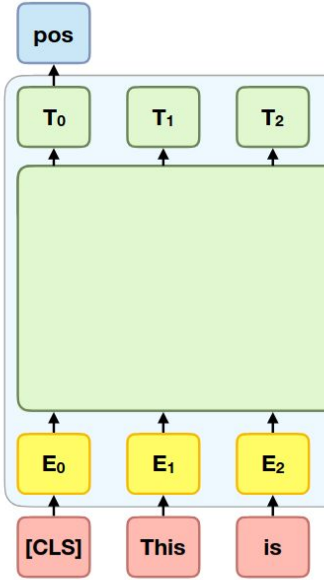
Recap: Classification with transformers

- For BERT and many related models, classification head attached to special [CLS] token added to start of token sequence
  - … other options also viable, e.g. pooling (max/avg/etc.) token representations
- Additional, randomly initialized output layer added to pre-trained model
- Fine-tuning can either only train this layer (faster) or continue training also other parts of the model (higher capacity)



Recap: Classification with transformers

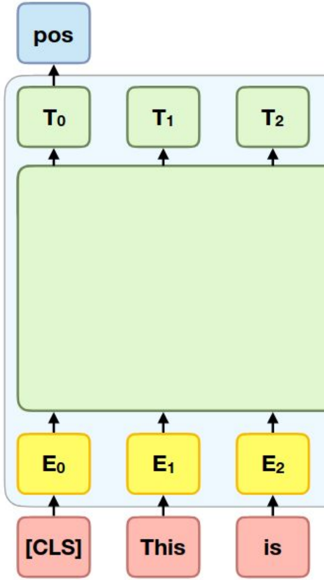
```
import torch
import transformers

model_name = "bert-base-cased"
model = transformers.AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=2)
```



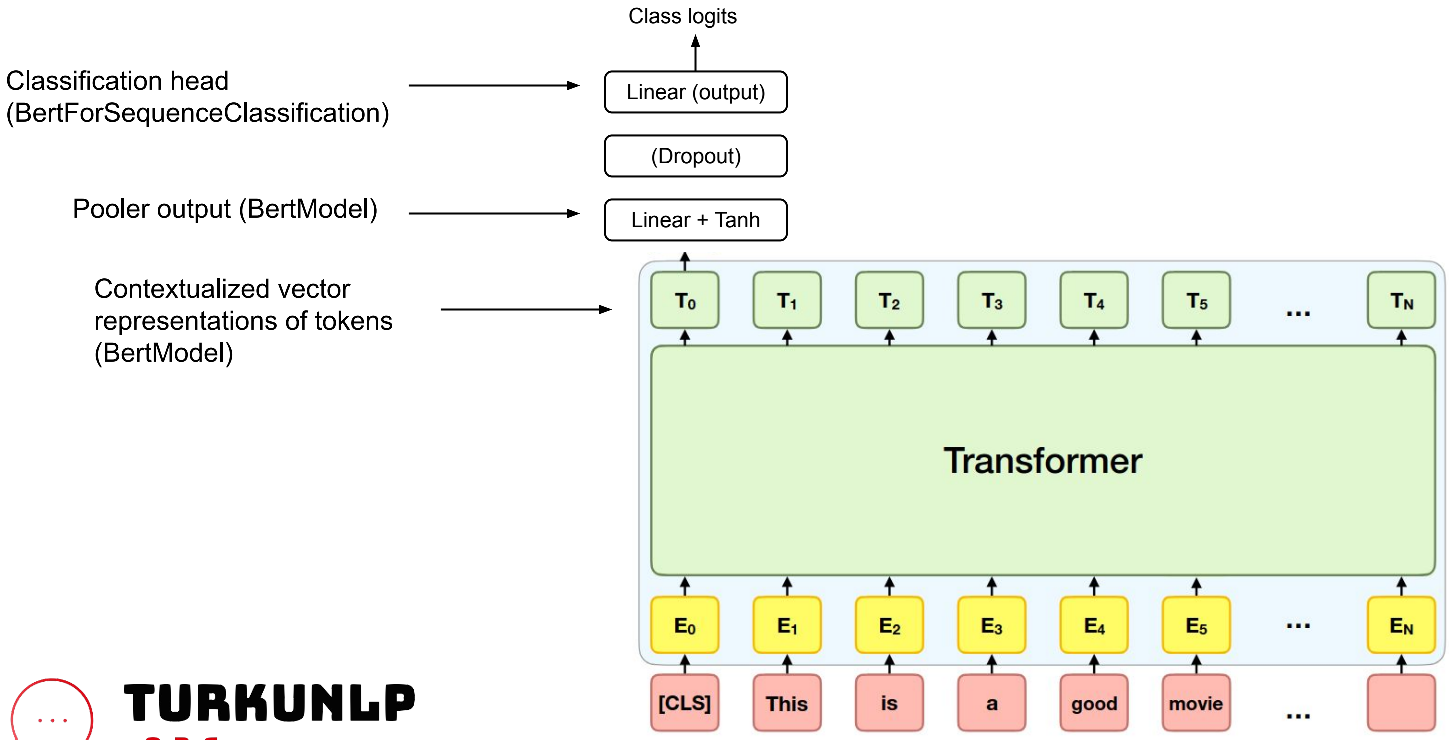
Recap: Classification with transformers

```
import torch
import transformers

model_name = "bert-base-cased"
model = transformers.AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=2)
```

- **AutoModel** loads the correct model architecture (Bert in our case)
- **ForXXX** loads the correct output configuration (“head”)
  - BertModel (no head, used e.g. for embedding)
  - ForPreTraining (loads pretraining head)
  - ForSequenceClassification



Recap: Classification with transformers



Recap: Classification with transformers

```
class BertForSequenceClassification(BertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels
        self.config = config

        self.bert = BertModel(config)
        classifier_dropout = (
            config.classifier_dropout if config.classifier_dropout is not None else config.hidden_dropout_prob
        )
        self.dropout = nn.Dropout(classifier_dropout)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

        # Initialize weights and apply final processing
        self.post_init()
```

Bert model without any output layer

Optional dropout

New linear output layer

```
def forward(
    self,
    input_ids: Optional[torch.Tensor] = None,
    attention_mask: Optional[torch.Tensor] = None,
    token_type_ids: Optional[torch.Tensor] = None,
    position_ids: Optional[torch.Tensor] = None,
    head_mask: Optional[torch.Tensor] = None,
    inputs_embeds: Optional[torch.Tensor] = None,
    labels: Optional[torch.Tensor] = None,
    output_attentions: Optional[bool] = None,
    output_hidden_states: Optional[bool] = None,
    return_dict: Optional[bool] = None,
) -> Union[Tuple[torch.Tensor], SequenceClassifierOutput]:
    r"""
    labels (`torch.LongTensor` of shape `(batch_size,)`, *optional*):
        Labels for computing the sequence classification/regression loss. Indices should be in `[0, ...,
        config.num_labels - 1]`. If `config.num_labels == 1` a regression loss is computed (Mean-Square
        `config.num_labels > 1` a classification loss is computed (Cross-Entropy).
    """
    return_dict = return_dict if return_dict is not None else self.config.use_return_dict

    outputs = self.bert(
        input_ids,
        attention_mask=attention_mask,
        token_type_ids=token_type_ids,
        position_ids=position_ids,
        head_mask=head_mask,
        inputs_embeds=inputs_embeds,
        output_attentions=output_attentions,
        output_hidden_states=output_hidden_states,
        return_dict=return_dict,
    )

    pooled_output = outputs[1]

    pooled_output = self.dropout(pooled_output)
    logits = self.classifier(pooled_output)
```

Bert outputs (contextualized vectors)

Extract “pooled output” from outputs

Run through dropout, and output layer to get classification logits

TURKUNLP .ORG

```python
def forward(
    self,
    input_ids: Optional[torch.Tensor] = None,
    attention_mask: Optional[torch.Tensor] = None,
    token_type_ids: Optional[torch.Tensor] = None,
    position_ids: Optional[torch.Tensor] = None,
    head_mask: Optional[torch.Tensor] = None,
    inputs_embeds: Optional[torch.Tensor] = None,
    labels: Optional[torch.Tensor] = None,
    output_attentions: Optional[bool] = None,
    output_hidden_states: Optional[bool] = None,
    return_dict: Optional[bool] = None,
) -> Union[Tuple[torch.Tensor], SequenceClassifierOutput]:
    r"""
    labels (`torch.LongTensor` of shape `(batch_size,)`, *optional*):
        Labels for computing the sequence classification/regression loss. Indices should be in `[0, ...,
        config.num_labels - 1]`. If `config.num_labels == 1` a regression loss is computed (Mean-Square
        `config.num_labels > 1` a classification loss is computed (Cross-Entropy).
    """
    return_dict = return_dict if return_dict is not None else self.config.use_return_dict

    outputs = self.bert(
        input_ids,
        attention_mask=attention_mask,
        token_type_ids=token_type_ids,
        position_ids=position_ids,
        head_mask=head_mask,
        inputs_embeds=inputs_embeds,
        output_attentions=output_attentions,
        output_hidden_states=output_hidden_states,
        return_dict=return_dict,
    )

    pooled_output = outputs[1]

    pooled_output = self.dropout(pooled_output)
    logits = self.classifier(pooled_output)
```

Bert outputs (contextualized vectors)

What is “pooled output” (outputs

Extract “pooled output” from outputs

Run through dropout, and output
layer to get classification logits

TURKUNLP
.ORG

Recap: Classification with transformers

class transformers.BertModel

(config, add_pooling_layer = Tr

Parameters

- **config** (BertConfig) — Model configu
config file does not load the weights
from_pretrained() method to load t

The bare Bert Model transformer outp

Returns transformers.modeling_outputs.BaseModelOutputWithPoolingAndCrossAttentions **or** `tuple(torch.FloatTensor)`

A transformers.modeling_outputs.BaseModelOutputWithPoolingAndCrossAttentions or a tuple of `torch.FloatTensor` (if `return_dict=False` is passed or when `config.return_dict=False`) comprising various elements depending on the configuration (BertConfig) and inputs.

- **last_hidden_state** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`) — Sequence of hidden-states at the output of the last layer of the model.
- **pooler_output** (`torch.FloatTensor` of shape `(batch_size, hidden_size)`) — Last layer hidden-state of the first token of the sequence (classification token) after further processing through the layers used for the auxiliary pretraining task. E.g. for BERT-family of models, this returns the classification token after processing through a linear layer and a tanh activation function. The linear layer weights are trained from the next sentence prediction (classification) objective during pretraining.
- **hidden_states** (`tuple(torch.FloatTensor)`, *optional*, returned when `output_hidden_states=True` is passed or when `config.output_hidden_states=True`) — Tuple of `torch.FloatTensor` (one for the output of the embeddings, if the model has an embedding layer, + one for the output of each layer) of shape `(batch_size, sequence_length, hidden_size)`.

  Hidden-states of the model at the output of each layer plus the optional initial embedding outputs.

Recap: Classification with transformers

class transformers.BertModel

(config, add_pooling_layer = Tr

Parameters

- **config** (BertConfig) — Model configu
config file does not load the weights
from_pretrained() method to load t

The bare Bert Model transformer outp

Returns transformers.modeling_outputs.BaseModelOutputWithPoolingAndCrossAttentions **or** `tuple(torch.FloatTensor)`

A transformers.modeling_outputs.BaseM
`torch.FloatTensor` (if `return_dict=F`
various elements depending on the confi

- **last_hidden_state** (`torch.FloatTe`
— Sequence of hidden-states at the c
- **pooler_output** (`torch.FloatTenso`
state of the first token of the sequen
used for the auxiliary pretraining tas
after processing through a linear lay
from the next sentence prediction (c
- **hidden_states** (`tuple(torch.FloatTensor)`, *optional*, returned when `output_hidden_states=True` is passed or when `config.output_hidden_states=True`) — Tuple of `torch.FloatTensor` (one for the output of the embeddings, if the model has an embedding layer, + one for the output of each layer) of shape `(batch_size, sequence_length, hidden_size)`.

Hidden-states of the model at the output of each layer plus the optional initial embedding outputs.

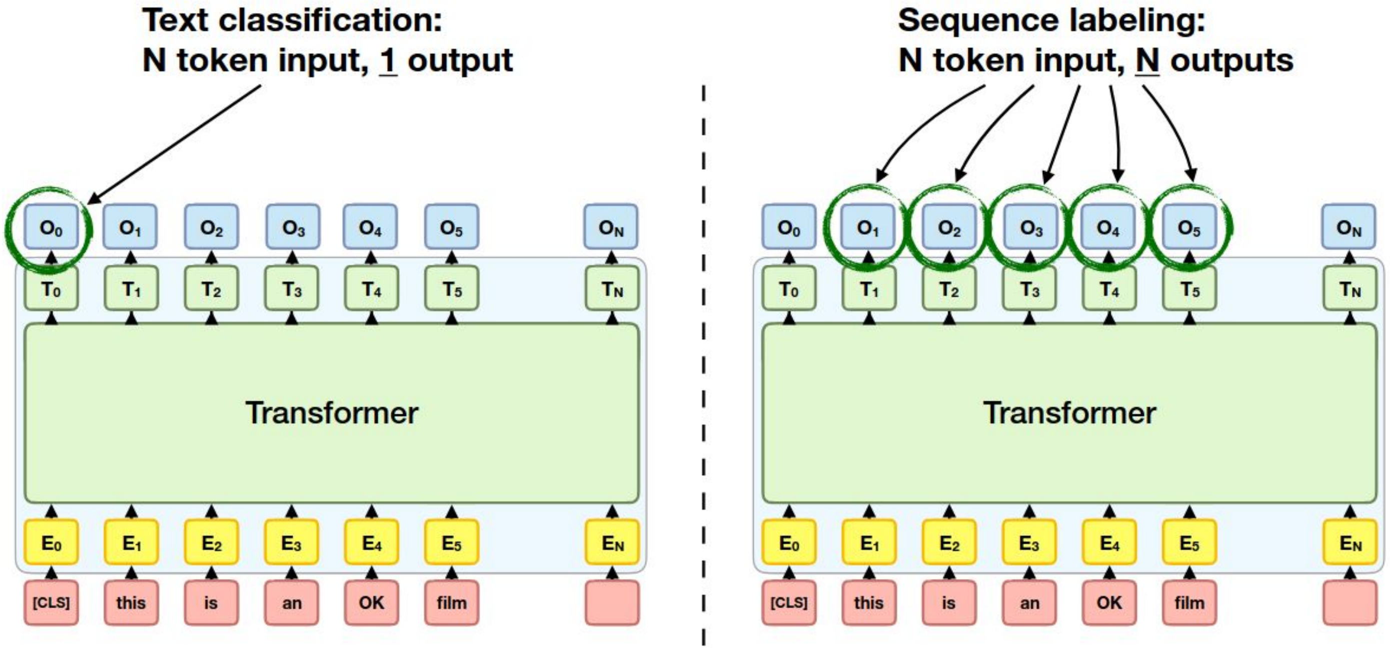
```python
class BertPooler(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        self.activation = nn.Tanh()

    def forward(self, hidden_states: torch.Tensor) -> torch.Tensor:
        # We "pool" the model by simply taking the hidden state corresponding
        # to the first token.
        first_token_tensor = hidden_states[:, 0]
        pooled_output = self.dense(first_token_tensor)
        pooled_output = self.activation(pooled_output)
        return pooled_output
```

Sequence labeling with transformers

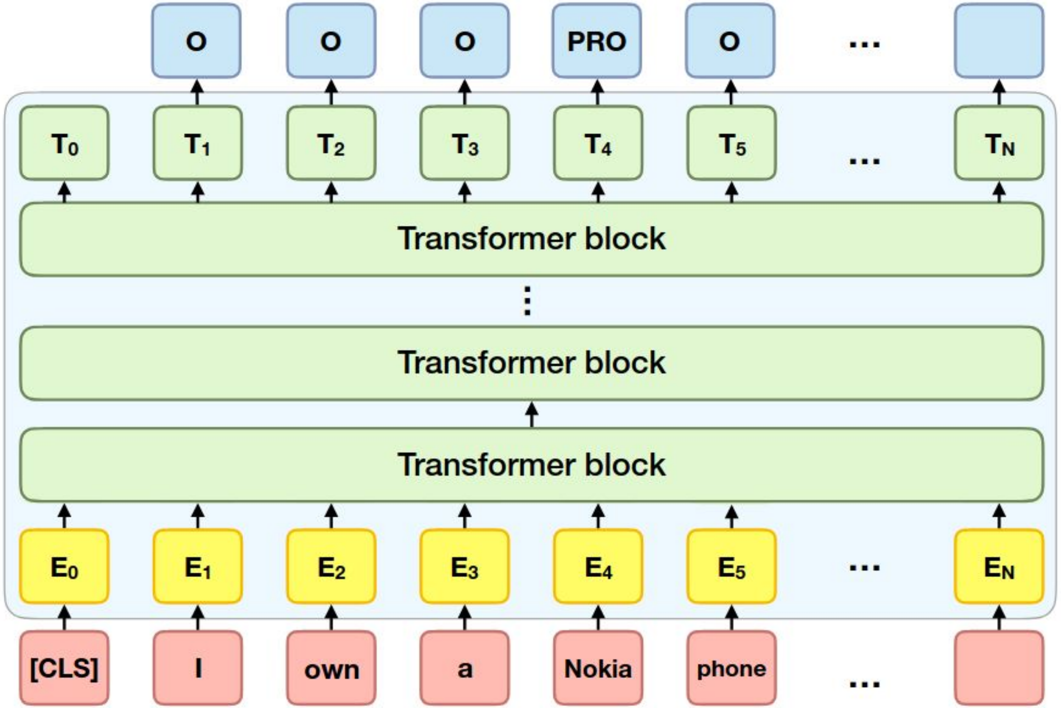
The	DET
dog	NOUN
runs	VERB
in	PREP
the	DET
park	NOUN
.	PUNCT

New	B-GPE	New York / GPE
York	I-GPE	
is	O	
in	O	
the	O	
United	B-GPE	United States / GPE
States	I-GPE	

Sequence labeling with transformers

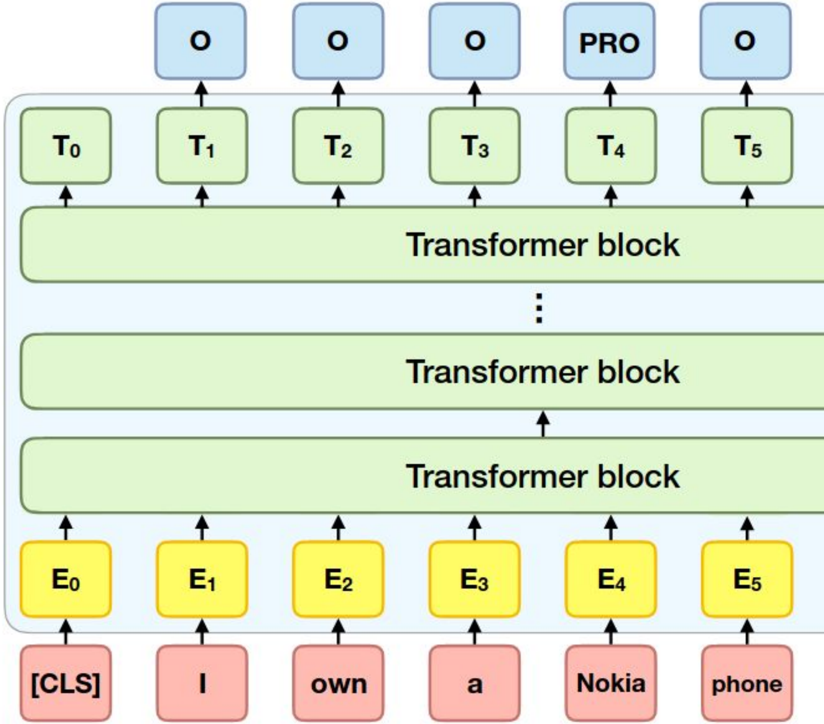


Sequence labeling with transformers



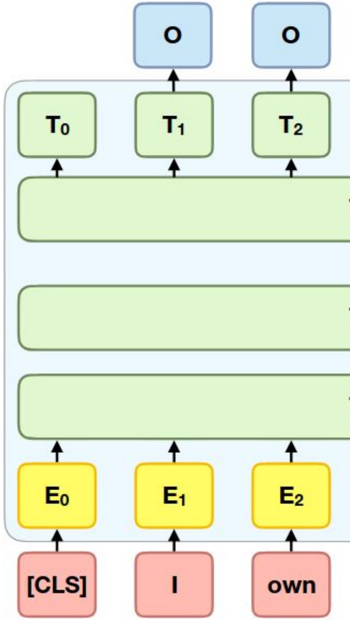
Sequence labeling with transformers

- Randomly initialized output layer attached to each token (shared weights).
- (Possible special tokens such as [CLS] generally not used)
- As in classification, fine-tuning can be performed either only for output layer or for whole model



Sequence labeling with transformers

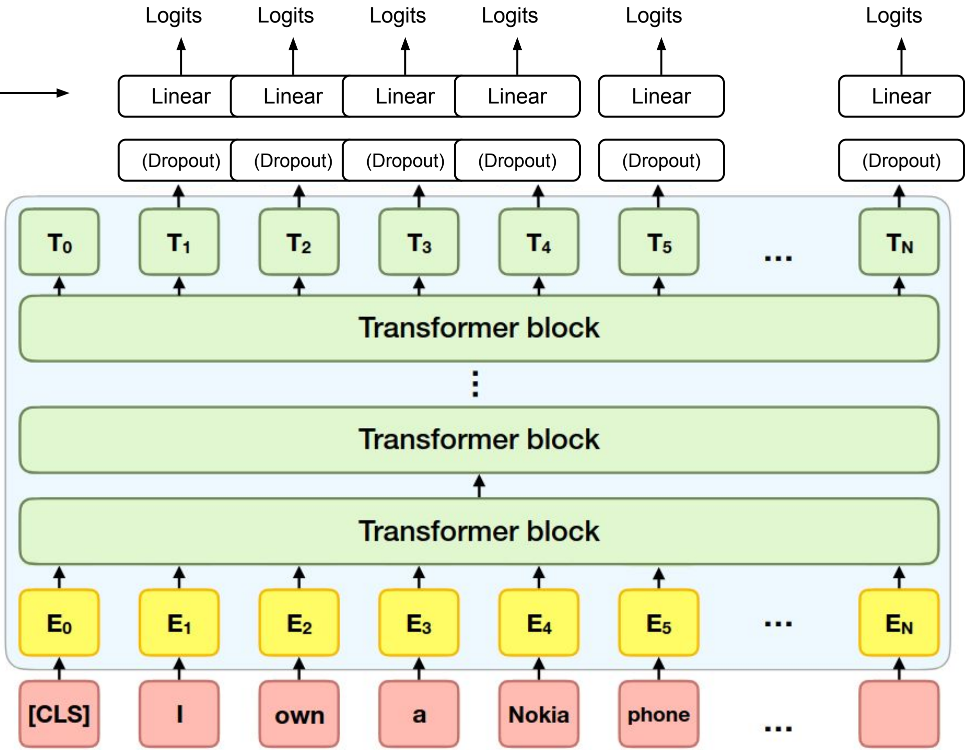
```
import torch
import transformers

model = 'bert-base-cased'
model = transformers.AutoModelForTokenClassification.from_pretrained(model, num_labels=6)
```



Sequence labeling with transformers

Classification head
(BertForTokenClassification)

Contextualized vector representations of tokens
(BertModel)



Same layer repeated / shared weights!

What happens in the code?

```
class BertForTokenClassification(BertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.bert = BertModel(config, add_pooling_layer=False)
        classifier_dropout = (
            config.classifier_dropout if config.classifier_dropout is not None else config.hidden_dropout
        )
        self.dropout = nn.Dropout(classifier_dropout)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

        # Initialize weights and apply final processing
        self.post_init()
```

Bert model without any output layer

Optional dropout

New linear output layer

Exactly same init() as in sequence classification, except pooling layer not needed!

```python
def forward(
    self,
    input_ids: Optional[torch.Tensor] = None,
    attention_mask: Optional[torch.Tensor] = None,
    token_type_ids: Optional[torch.Tensor] = None,
    position_ids: Optional[torch.Tensor] = None,
    head_mask: Optional[torch.Tensor] = None,
    inputs_embeds: Optional[torch.Tensor] = None,
    labels: Optional[torch.Tensor] = None,
    output_attentions: Optional[bool] = None,
    output_hidden_states: Optional[bool] = None,
    return_dict: Optional[bool] = None,
) -> Union[Tuple[torch.Tensor], TokenClassifierOutput]:
    r"""
    labels (`torch.LongTensor` of shape `(batch_size, sequence_length)`, *optional*):
        Labels for computing the token classification loss. Indices should be in `[0, ..., config.num_labels - 1]`.
    """
    return_dict = return_dict if return_dict is not None else self.config.use_return_dict

    outputs = self.bert(
        input_ids,
        attention_mask=attention_mask,
        token_type_ids=token_type_ids,
        position_ids=position_ids,
        head_mask=head_mask,
        inputs_embeds=inputs_embeds,
        output_attentions=output_attentions,
        output_hidden_states=output_hidden_states,
        return_dict=return_dict,
    )

    sequence_output = outputs[0]

    sequence_output = self.dropout(sequence_output)
    logits = self.classifier(sequence_output)
```

Bert outputs (contextualized vectors)

Extract “sequence outputs”

Run through dropout, and output layer to get classification logits

TURKUNLP
.ORG

What happens in code?

class transformers.BertModel

(config, add_pooling_layer = Tr

Parameters

- **config** (BertConfig) — Model configu
config file does not load the weights
from_pretrained() method to load t

The bare Bert Model transformer outp

Returns transformers.modeling_outputs.BaseModelOutputWithPoolingAndCrossAttentions **or** `tuple(torch.FloatTensor)`

A transformers.modeling_outputs.BaseModelOutputWithPoolingAndCrossAttentions or a tuple of `torch.FloatTensor` (if `return_dict=False` is passed or when `config.return_dict=False`) comprising various elements depending on the configuration (BertConfig) and inputs.

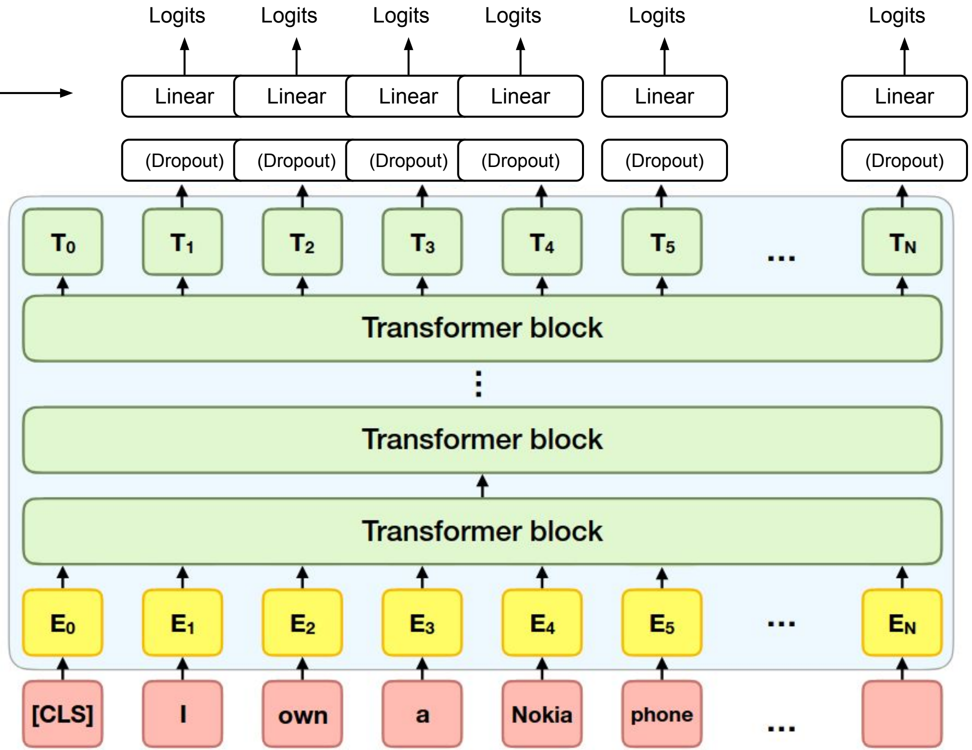
- **last_hidden_state** (`torch.FloatTensor` of shape `(batch_size, sequence_length, hidden_size)`) — Sequence of hidden-states at the output of the last layer of the model.
- **pooler_output** (`torch.FloatTensor` of shape `(batch_size, hidden_size)`) — Last layer hidden-state of the first token of the sequence (classification token) after further processing through the layers used for the auxiliary pretraining task. E.g. for BERT-family of models, this returns the classification token after processing through a linear layer and a tanh activation function. The linear layer weights are trained from the next sentence prediction (classification) objective during pretraining.
- **hidden_states** (`tuple(torch.FloatTensor)`, *optional*, returned when `output_hidden_states=True` is passed or when `config.output_hidden_states=True`) — Tuple of `torch.FloatTensor` (one for the output of the embeddings, if the model has an embedding layer, + one for the output of each layer) of shape `(batch_size, sequence_length, hidden_size)`.

Hidden-states of the model at the output of each layer plus the optional initial embedding outputs.

Sequence labeling with transformers

Classification head
(BertForTokenClassification)

Contextualized vector
representations of tokens
(BertModel)



Same layer
repeated / shared
weights!

TURKUNLP
.ORG

UNIVERSITY
OF TURKU

Sequence labeling with transformers

- Note that label sequences are not directly modeled in AutoModelForTokenClassification
    - Transition probabilities (e.g. *how likely is label I-PER, given the previous predicted label is B-PER*)
    - In theory, can produce invalid sequences (e.g. I-PER after O), but in practice does not really happen
- Adding a CRF (Conditional Random Field) on top of transformer adds direct knowledge of previous labels (dependencies between the predictions)

Sequence labeling with transformers

- Practical note: unlike most text classification datasets, sequence labeling datasets will normally come with a definition of what constitutes a “token” so a label (tags) can be assigned to each, e.g.
  - tokens = [“Se”, “on”, “Turun”, “lähellä”]
  - labels = [“O”, “O”, “B-LOC”, “O”]
- Transformer models have their own definitions of tokens, which cannot be readily changed. This adds a requirement to map from corpus tokens to transformer tokens and back.
- (The challenge here is mostly technical: the different definitions of “token” are mostly not an issue for machine learning model performance.)

Evaluation metrics

- Token-level classification accuracy (correct predictions out of all predictions) generally used to evaluate e.g. POS tagging
- For tasks involving marking spans (e.g. NER), performance typically measured on span level in terms of exact-match precision, recall and F1-score:
  - Compare predicted and gold standard spans in terms of (start-token, end-token, type)
  - Only triples where all values match between predicted and gold are correct
- Precision: fraction of predicted spans that are correct
- Recall: fraction of gold standard spans that are correctly predicted
- F1-score: balanced harmonic mean of precision and recall

Model notebook

- https://github.com/TurkuNLP/textual-data-analysis-course/blob/main/sequence_labeling_example.ipynb

Alternative approaches

- Fine-tuned decoder LMs for sequence labeling (token classification head)?
  - Causal attention mask not optimal for sequence labeling
  - *Nokia is a …*
- Like text classification, sequence labeling tasks can be approached also using generative models with in-context learning
  - Especially useful in cases where interested entities are not standard NER entities → No supervised training data
  - E.g. Hobbies, professions etc.
  - Generative models (language modelling head) read the whole text before generating an answer

Alternative approaches

- How to formulate a prompt for NER?
  - IOB-tagging prompt may not be the optimal
  - Ask the model to return / tag all entities of certain type from the text
    - “*List the person names appearing in this sentence …*”
    - “*Mark all person names appearing in this sentence with <PERSON> and </PERSON> tags.*”
  - Computationally heavier compared to multi-class approach, as this needs to be run separately for each entity type
    - Of course one can ask several types at the same time, but in general one-by-one seem to work better accuracy wise

Summary

- Text classification maps a token sequence into a label (or set of labels), sequence labeling into one label per token
- Substantial number of pre-trained models and datasets for these tasks openly available
- Fine-tuned encoder LM still the standard method for sequence labeling
- In-context learning with generative models especially useful for entity types not annotated in standard NER datasets