

Layers and Model Architectures

Power spaCy components with custom neural networks


A **model architecture** is a function that wires up a [Thinc Model](#) instance. It describes the neural network that is run internally as part of a component in a spaCy pipeline. To define the actual architecture, you can implement your logic in Thinc directly, or you can use Thinc as a thin wrapper around frameworks such as PyTorch, TensorFlow and MXNet. Each `Model` can also be used as a sublayer of a larger network, allowing you to freely combine implementations from different frameworks into a single model.


spaCy's built-in components require a `Model` instance to be passed to them via the config system. To change the model architecture of an existing component, you just need to [update the config](#) so that it refers to a different registered function. Once the component has been created from this config, you won't be able to change it anymore. The architecture is like a recipe for the network, and you can't change the recipe once the dish has already been prepared. You have to make a new one.




```
[components.tagger]
factory = "tagger"

[components.tagger.model]
@architectures = "model.v1"
width = 512
classes = 16
```

Type signatures

The Thinc `Model` class is a **generic type** that can specify its input and output types. Python uses a square-bracket notation for this, so the type `Model[List, Dict]` says that each batch of inputs to the model will be a list, and the outputs will be a dictionary. You can be even more specific and write for instance `Model[List[Doc], Dict[str, float]]` to specify that the model expects a list of [Doc](#)  objects as input, and returns a dictionary mapping of strings to floats. Some of the most common types you'll see are:

TYPE	DESCRIPTION
<code>List[Doc]</code>	A batch of <code>Doc</code>  objects. Most components expect their models to take this as input.
<code>Floats2d</code>	A two-dimensional <code>numpy</code> or <code>cupy</code> array of floats. Usually 32-bit.
<code>Ints2d</code>	A two-dimensional <code>numpy</code> or <code>cupy</code> array of integers. Common dtypes include <code>uint64</code> , <code>int32</code> and <code>int8</code> .
<code>List[Floats2d]</code>	A list of two-dimensional arrays, generally with one array per <code>Doc</code> and one row per token.
<code>Ragged</code>	A container to handle variable-length sequence data in an unpadded contiguous array.
<code>Padded</code>	A container to handle variable-length sequence data in a padded contiguous array.



See the [Thinc type reference](#) for details. The model type signatures help you figure out which model architectures and components can **fit together**. For instance, the `TextCategorizer`  class expects a model typed `Model[List[Doc], Floats2d]`, because the model will predict one row of category probabilities per `Doc` . In contrast, the `Tagger`  class expects a model typed `Model[List[Doc], List[Floats2d]]`, because it needs to predict one row of probabilities per token.

There's no guarantee that two models with the same type signature can be used interchangeably. There are many other ways they could be incompatible. However, if the types don't match, they almost surely *won't* be compatible. This little bit of validation goes a long way, especially if you [configure your editor](#) or other tools to highlight these errors early. The config file is also validated at the beginning of training, to verify that all the types match correctly.

Tip: Static type checking in your editor



Swapping model architectures

If no model is specified for the `TextCategorizer` , the `TextCatEnsemble`  architecture is used by default. This architecture combines a simple bag-of-words model with a neural network, usually resulting in the most accurate results, but at the cost of speed. The config file for this model would look something like this:

```

[components.textcat]
factory = "textcat"
labels = []

[components.textcat.model]
@architectures = "spacy.TextCatEnsemble.v2"
n0 = null


[components.textcat.model.tok2vec]
@architectures = "spacy.Tok2Vec.v2"

[components.textcat.model.tok2vec.embed]
@architectures = "spacy.MultiHashEmbed.v2"
width = 64
rows = [2000, 2000, 1000, 1000, 1000, 1000]
attrs = ["ORTH", "LOWER", "PREFIX", "SUFFIX", "SHAPE", "ID"]
include_static_vectors = false

[components.textcat.model.tok2vec.encode]
@architectures = "spacy.MaxoutWindowEncoder.v2"
width = ${components.textcat.model.tok2vec.embed.width}
window_size = 1
maxout_pieces = 3
depth = 2

[components.textcat.model.linear_model]
@architectures = "spacy.TextCatBOW.v3"
exclusive_classes = true
length = 262144
ngram_size = 1
no_output_layer = false

```

spaCy has two additional built-in `textcat` architectures, and you can easily use those by swapping out the definition of the textcat's model. For instance, to use the simple and fast bag-of-words model [TextCatBOW](#) , you can change the config to:


```

[components.textcat]
factory = "textcat"
labels = []

[components.textcat.model]
@architectures = "spacy.TextCatBOW.v3"
exclusive_classes = true
length = 262144


```

```
ngram_size = 1
no_output_layer = false
n0 = null
```

For details on all pre-defined architectures shipped with spaCy and how to configure them, check out the [model architectures](#)  documentation.

Defining sublayers

Model architecture functions often accept **sublayers as arguments**, so that you can try **substituting a different layer** into the network. Depending on how the architecture function is structured, you might be able to define your network structure entirely through the [config system](#), using layers that have already been defined.


In most neural network models for NLP, the most important parts of the network are what we refer to as the [embed and encode](#) steps. These steps together compute dense, context-sensitive representations of the tokens, and their combination forms a typical `Tok2Vec`  layer:

```
[components.tok2vec]
factory = "tok2vec"

[components.tok2vec.model]
@architectures = "spacy.Tok2Vec.v2"

[components.tok2vec.model.embed]
@architectures = "spacy.MultiHashEmbed.v2"
# ...

[components.tok2vec.model.encode]
@architectures = "spacy.MaxoutWindowEncoder.v2"
# ...
```

By defining these sublayers specifically, it becomes straightforward to swap out a sublayer for another one, for instance changing the first sublayer to a character embedding with the [CharacterEmbed](#)  architecture:

```
[components.tok2vec.model.embed]
@architectures = "spacy.CharacterEmbed.v2"
# ...

[components.tok2vec.model.encode]
```

```
@architectures = "spacy.MaxoutWindowEncoder.v2"  
# ...
```

Most of spaCy's default architectures accept a `tok2vec` layer as a sublayer within the larger task-specific neural network. This makes it easy to **switch between** transformer, CNN, BiLSTM or other feature extraction approaches. The [transformers documentation](#) section shows an example of swapping out a model's standard `tok2vec` layer with a transformer. And if you want to define your own solution, all you need to do is register a `Model[List[Doc], List[Floats2d]]` architecture function, and you'll be able to try it out in any of the spaCy components.

Wrapping PyTorch, TensorFlow and other frameworks

Thinc allows you to [wrap models](#) written in other machine learning frameworks like PyTorch, TensorFlow and MXNet using a unified `Model` API. This makes it easy to use a model implemented in a different framework to power a component in your spaCy pipeline. For example, to wrap a PyTorch model as a Thinc `Model`, you can use Thinc's [PyTorchWrapper](#):

```
from thinc.api import PyTorchWrapper  
  
wrapped_pt_model = PyTorchWrapper(torch_model)
```

Let's use PyTorch to define a very simple neural network consisting of two hidden `Linear` layers with `ReLU` activation and dropout, and a softmax-activated output layer:

```
from torch import nn  
  
torch_model = nn.Sequential(  
    nn.Linear(width, hidden_width),  
    nn.ReLU(),  
    nn.Dropout2d(dropout),  
    nn.Linear(hidden_width, n0),  
    nn.ReLU(),  
    nn.Dropout2d(dropout),
```


```
nn.Softmax(dim=1)
```

```
)
```

The resulting wrapped `Model` can be used as a **custom architecture** as such, or can be a **subcomponent of a larger model**. For instance, we can use Thinc's [chain](#) combinator, which works like `Sequential` in PyTorch, to combine the wrapped model with other components in a larger network. This effectively means that you can easily wrap different components from different frameworks, and “glue” them together with Thinc:


```
from thinc.api import chain, with_array, PyTorchWrapper
from spacy.ml import CharacterEmbed

wrapped_pt_model = PyTorchWrapper(torch_model)
char_embed = CharacterEmbed(width, embed_size, nM, nC)
model = chain(char_embed, with_array(wrapped_pt_model))
```

In the above example, we have combined our custom PyTorch model with a character embedding layer defined by spaCy. [CharacterEmbed](#)  returns a `Model` that takes a `List[Doc]` as input, and outputs a `List[Floats2d]`. To make sure that the wrapped PyTorch model receives valid inputs, we use Thinc's [with_array](#) helper.

You could also implement a model that only uses PyTorch for the transformer layers, and “native” Thinc layers to do fiddly input and output transformations and add on task-specific “heads”, as efficiency is less of a consideration for those parts of the network.

Using wrapped models

To use our custom model including the PyTorch subnetwork, all we need to do is register the architecture using the [architectures registry](#) . This assigns the architecture a name so spaCy knows how to find it, and allows passing in arguments like hyperparameters via the [config](#). The full example then becomes:

```
from typing import List
from thinc.types import Floats2d
from thinc.api import Model, PyTorchWrapper, chain, with_array
import spacy
from spacy.tokens.doc import Doc
from spacy.ml import CharacterEmbed
from torch import nn
```

```

@spacy.registry.architectures("CustomTorchModel.v1")
def create_torch_model(
    n0: int,
    width: int,
    hidden_width: int,
    embed_size: int,
    nM: int,
    nC: int,
    dropout: float,
) -> Model[List[Doc], List[Floats2d]]:
    char_embed = CharacterEmbed(width, embed_size, nM, nC)
    torch_model = nn.Sequential(
        nn.Linear(width, hidden_width),
        nn.ReLU(),
        nn.Dropout2d(dropout),
        nn.Linear(hidden_width, n0),
        nn.ReLU(),
        nn.Dropout2d(dropout),
        nn.Softmax(dim=1)
    )
    wrapped_pt_model = PyTorchWrapper(torch_model)
    model = chain(char_embed, with_array(wrapped_pt_model))
    return model

```

The model definition can now be used in any existing trainable spaCy component, by specifying it in the config file. In this configuration, all required parameters for the various subcomponents of the custom architecture are passed in as settings via the config.

```

[components.tagger]
factory = "tagger"

[components.tagger.model]
@architectures = "CustomTorchModel.v1"
n0 = 50
width = 96
hidden_width = 48
embed_size = 2000
nM = 64
nC = 8
dropout = 0.2

```

Note that when using a PyTorch or Tensorflow model, it is recommended to set the GPU memory allocator accordingly. When `gpu_allocator` is set to "pytorch" or "tensorflow" in the training config, cupy will

allocate memory via those respective libraries, preventing OOM errors when there's available memory sitting in the other library's pool.

```
[training]
gpu_allocator = "pytorch"
```

Custom models with Thinc

Of course it's also possible to define the `Model` from the previous section entirely in Thinc. The Thinc documentation provides details on the [various layers](#) and helper functions available. Combinators can be used to [overload operators](#) and a common usage pattern is to bind `chain` to `>>`. The "native" Thinc version of our simple neural network would then become:

```
from thinc.api import chain, with_array, Model, Relu, Dropout, Softmax
from spacy.ml import CharacterEmbed

char_embed = CharacterEmbed(width, embed_size, nM, nC)
with Model.define_operators({">>": chain}):
    layers = (
        Relu(hidden_width, width)
        >> Dropout(dropout)
        >> Relu(hidden_width, hidden_width)
        >> Dropout(dropout)
        >> Softmax(n0, hidden_width)
    )
    model = char_embed >> with_array(layers)
```


Shape inference in Thinc

It is **not** strictly necessary to define all the input and output dimensions for each layer, as Thinc can perform [shape inference](#) between sequential layers by matching up the output dimensionality of one layer to the input dimensionality of the next. This means that we can simplify the `layers` definition:


```
with Model.define_operators({">>": chain}):
    layers = (
        ReLU(hidden_width, width)
        >> Dropout(dropout)
        >> ReLU(hidden_width)
        >> Dropout(dropout)
        >> Softmax(n0)
    )
```

Thinc can even go one step further and **deduce the correct input dimension** of the first layer, and output dimension of the last. To enable this functionality, you have to call `Model.initialize` with an **input sample** `X` and an **output sample** `Y` with the correct dimensions:

```
with Model.define_operators({">>": chain}):
    layers = (
        ReLU(hidden_width)
        >> Dropout(dropout)
        >> ReLU(hidden_width)
        >> Dropout(dropout)
        >> Softmax()
    )
    model = char_embed >> with_array(layers)
    model.initialize(X=input_sample, Y=output_sample)
```

The built-in [pipeline components](#) in spaCy ensure that their internal models are **always initialized** with appropriate sample data. In this case, `X` is typically a `List[Doc]`, while `Y` is typically a `List[Array1d]` or `List[Array2d]`, depending on the specific task. This functionality is triggered when `nlp.initialize`  is called.

Dropout and normalization in Thinc

Many of the available Thinc [layers](#) allow you to define a `dropout` argument that will result in “chaining” an additional [Dropout](#) layer. Optionally, you can often specify whether or not you want to add layer normalization, which would result in an additional [LayerNorm](#) layer. That means that the following `layers` definition is equivalent to the previous:


```
with Model.define_operators({">>": chain}):
    layers = (
```

```

    Relu(hidden_width, dropout=dropout, normalize=False)
    >> Relu(hidden_width, dropout=dropout, normalize=False)
    >> Softmax()
)
model = char_embed >> with_array(layers)
model.initialize(X=input_sample, Y=output_sample)



```

Create new trainable components

In addition to [swapping out](#) layers in existing components, you can also implement an entirely new, [trainable](#) pipeline component from scratch. This can be done by creating a new class inheriting from `TrainablePipe` , and linking it up to your custom model implementation.

Example: Entity relation extraction component

This section outlines an example use-case of implementing a **novel relation extraction component** from scratch. We'll implement a binary relation extraction method that determines whether or not **two entities** in a document are related, and if so, what type of relation connects them. We allow multiple types of relations between two such entities (a multi-label setting). There are two major steps required:

1. Implement a [machine learning model](#) specific to this task. It will have to extract candidate relation instances from a `Doc`  and predict the corresponding scores for each relation label.
2. Implement a custom [pipeline component](#) - powered by the machine learning model from step 1 - that translates the predicted scores into annotations that are stored on the `Doc`  objects as they pass through the `nlp` pipeline.

Step 1: Implementing the Model

We need to implement a [Model](#) that takes a **list of documents** (`List[Doc]`) as input, and outputs a **two-dimensional matrix** (`Floats2d`) of predictions:

```

@spacy.registry.architectures("rel_model.v1")
def create_relation_model(...) -> Model[List[Doc], Floats2d]:

```

```
model = ... # 🐱 model will go here
return model
```

We adapt a **modular approach** to the definition of this relation model, and define it as chaining two layers together: the first layer that generates an instance tensor from a given set of documents, and the second layer that transforms the instance tensor into a final tensor holding the predictions:

```
@spacy.registry.architectures("rel_model.v1")
def create_relation_model(
    create_instance_tensor: Model[List[Doc], Floats2d],
    classification_layer: Model[Floats2d, Floats2d],
) -> Model[List[Doc], Floats2d]:
    model = chain(create_instance_tensor, classification_layer)
    return model
```

The `classification_layer` could be something like a [Linear](#) layer followed by a [logistic](#) activation function:

```
@spacy.registry.architectures("rel_classification_layer.v1")
def create_classification_layer(
    n0: int = None, nI: int = None
) -> Model[Floats2d, Floats2d]:
    return chain(Linear(n0=n0, nI=nI), Logistic())
```

The first layer that **creates the instance tensor** can be defined by implementing a [custom forward function](#) with an appropriate backpropagation callback. We also define an [initialization method](#) that ensures that the layer is properly set up for training.

We omit some of the implementation details here, and refer to the [spaCy project](#) </> that has the full implementation.

```
@spacy.registry.architectures("rel_instance_tensor.v1")
def create_tensors(
    tok2vec: Model[List[Doc], List[Floats2d]],
    pooling: Model[Ragged, Floats2d],
    get_instances: Callable[[Doc], List[Tuple[Span, Span]]],
) -> Model[List[Doc], Floats2d]:

    return Model(
        "instance_tensors",
        instance_forward,
        init=instance_init,
```

```

        layers=[tok2vec, pooling],
        refs={"tok2vec": tok2vec, "pooling": pooling},
        attrs={"get_instances": get_instances},
    )



# The custom forward function
def instance_forward(
    model: Model[List[Doc], Floats2d],
    docs: List[Doc],
    is_train: bool,
) -> Tuple[Floats2d, Callable]:
    tok2vec = model.get_ref("tok2vec")
    tokvecs, bp_tokvecs = tok2vec(docs, is_train)
    get_instances = model.attrs["get_instances"]
    all_instances = [get_instances(doc) for doc in docs]
    pooling = model.get_ref("pooling")
    relations = ...

    def backprop(d_relations: Floats2d) -> List[Doc]:
        d_tokvecs = ...
        return bp_tokvecs(d_tokvecs)


    return relations, backprop

# The custom initialization method
def instance_init(
    model: Model,
    X: List[Doc] = None,
    Y: Floats2d = None,
) -> Model:
    tok2vec = model.get_ref("tok2vec")
    tok2vec.initialize(X)
    return model


```

This custom layer uses an [embedding layer](#) such as a `Tok2Vec`  component or a `Transformer` . This layer is assumed to be of type `Model[List[Doc], List[Floats2d]]` as it transforms each **document into a list of tokens**, with each token being represented by its embedding in the vector space.

The `pooling` layer will be applied to summarize the token vectors into **entity vectors**, as named entities (represented by `Span` objects) can consist of one or multiple tokens. For instance, the pooling layer could resort to calculating the average of all token vectors in an entity. Thinc provides several [built-in pooling operators](#) for this purpose.

Finally, we need a `get_instances` method that **generates pairs of entities** that we want to classify as being related or not. As these candidate pairs are typically formed within one document, this function takes a `Doc`  as input and outputs a `List` of `Span` tuples. For instance, the following implementation takes any two entities from the same document, as long as they are within a **maximum distance** (in number of tokens) of each other:

```
@spacy.registry.misc("rel_instance_generator.v1")
def create_instances(max_length: int) -> Callable[[Doc], List[Tuple[Span, Span]]]:
    def get_candidates(doc: "Doc") -> List[Tuple[Span, Span]]:
        candidates = []
        for ent1 in doc.ents:
            for ent2 in doc.ents:
                if ent1 != ent2:
                    if max_length and abs(ent2.start - ent1.start) <= max_length:
                        candidates.append((ent1, ent2))
        return candidates
    return get_candidates
```


This function is added to the `@misc registry`  so we can refer to it from the config, and easily swap it out for any other candidate generation function.

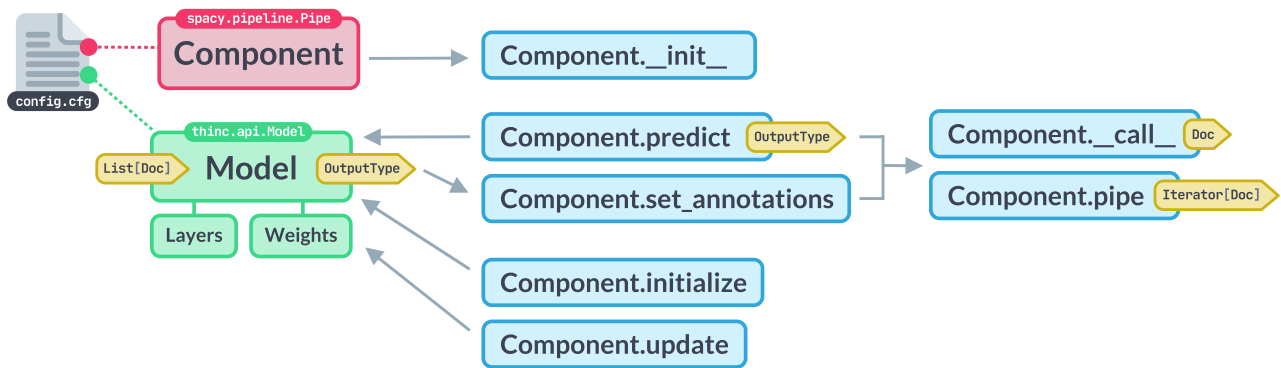
Intermezzo: define how to store the relations data

For our new relation extraction component, we will use a custom [extension attribute](#) `doc._.rel` in which we store relation data. The attribute refers to a dictionary, keyed by the **start offsets of each entity** involved in the candidate relation. The values in the dictionary refer to another dictionary where relation labels are mapped to values between 0 and 1. We assume anything above 0.5 to be a `True` relation. The [Example](#) instances that we'll use as training data, will include their gold-standard relation annotations in `example.reference._.rel`.

```
from spacy.tokens import Doc
Doc.set_extension("rel", default={})
```

Step 2: Implementing the pipeline component

To use our new relation extraction model as part of a custom [trainable component](#), we create a subclass of `TrainablePipe`  that holds the model.



```

from spacy.pipeline import TrainablePipe

class RelationExtractor(TrainablePipe):
    def __init__(self, vocab, model, name="rel"):
        """Create a component instance."""
        self.model = model
        self.vocab = vocab
        self.name = name

    def update(self, examples, drop=0.0, sgd=None, losses=None):
        """Learn from a batch of Example objects."""
        ...

    def predict(self, docs):
        """Apply the model to a batch of Doc objects."""
        ...

    def set_annotations(self, docs, predictions):
        """Modify a batch of Doc objects using the predictions."""
        ...

    def initialize(self, get_examples, nlp=None, labels=None):
        """Initialize the model before training."""
        ...

    def add_label(self, label):
        """Add a label to the component."""
        ...

```

Typically, the **constructor** defines the vocab, the Machine Learning model, and the name of this component. Additionally, this component, just like the `textcat` and the `tagger`, stores an **internal list of labels**. The ML model will predict scores for each label. We add convenience methods to easily retrieve and add to them.


```

def __init__(self, vocab, model, name="rel"):
    """Create a component instance."""
    # ...
    self.cfg = {"labels": []}

@property
def labels(self) -> Tuple[str, ...]:
    """Returns the labels currently added to the component."""
    return tuple(self.cfg["labels"])

def add_label(self, label: str):
    """Add a new label to the pipe."""
    self.cfg["labels"] = list(self.labels) + [label]

```

After creation, the component needs to be [initialized](#). This method can define the relevant labels in two ways: explicitly by setting the `labels` argument in the `initialize` block  of the config, or implicitly by deducing them from the `get_examples` callback that generates the full **training data set**, or a representative sample.



The final number of labels defines the output dimensionality of the network, and will be used to do [shape inference](#) throughout the layers of the neural network. This is triggered by calling [Model.initialize](#).



```

from itertools import islice


def initialize(
    self,
    get_examples: Callable[[], Iterable[Example]],
    *,
    nlp: Language = None,
    labels: Optional[List[str]] = None,
):
    if labels is not None:
        for label in labels:
            self.add_label(label)
    else:
        for example in get_examples():
            relations = example.reference._rel
            for indices, label_dict in relations.items():
                for label in label_dict.keys():
                    self.add_label(label)
        subbatch = list(islice(get_examples(), 10))
        doc_sample = [eg.reference for eg in subbatch]
        label_sample = self._examples_to_truth(subbatch)
        self.model.initialize(X=doc_sample, Y=label_sample)

```


The `initialize` method is triggered whenever this component is part of an `nlp` pipeline, and `nlp.initialize`  is invoked. Typically, this happens when the pipeline is set up before training in `spacy train` . After initialization, the pipeline component and its internal model can be trained and used to make predictions.

During training, the method `update`  is invoked which delegates to `Model.begin_update` and a `get_loss`  function that **calculates the loss** for a batch of examples, as well as the **gradient** of loss that will be used to update the weights of the model layers. Thinc provides several [loss functions](#) that can be used for the implementation of the `get_loss` function.

```
def update(
    self,
    examples: Iterable[Example],
    *,
    drop: float = 0.0,
    sgd: Optional[Optimizer] = None,
    losses: Optional[Dict[str, float]] = None,
) -> Dict[str, float]:
    # ...
    docs = [eg.predicted for eg in examples]
    predictions, backprop = self.model.begin_update(docs)
    loss, gradient = self.get_loss(examples, predictions)
    backprop(gradient)
    losses[self.name] += loss
    # ...
    return losses
```

After training the model, the component can be used to make novel **predictions**. The `predict`  method needs to be implemented for each subclass of `TrainablePipe`. In our case, we can simply delegate to the internal model's `predict` function that takes a batch of `Doc` objects and returns a `Floats2d` array:

```
def predict(self, docs: Iterable[Doc]) -> Floats2d:
    predictions = self.model.predict(docs)
    return self.model.ops.asarray(predictions)
```


The final method that needs to be implemented, is `set_annotations` . This function takes the predictions, and modifies the given `Doc` object in place to store them. For our relation extraction component, we store the data in the [custom attribute](#) `doc._.rel`.

To interpret the scores predicted by the relation extraction model correctly, we need to refer to the model's `get_instances` function that defined which pairs of entities were relevant candidates, so that the predictions can be linked to those exact entities:


```
def set_annotations(self, docs: Iterable[Doc], predictions: Floats2d):
    c = 0
    get_instances = self.model.attrs["get_instances"]
    for doc in docs:
        for (e1, e2) in get_instances(doc):
            offset = (e1.start, e2.start)
            if offset not in doc._.rel:
                doc._.rel[offset] = {}
            for j, label in enumerate(self.labels):
                doc._.rel[offset][label] = predictions[c, j]
            c += 1
```


Under the hood, when the pipe is applied to a document, it delegates to the `predict` and `set_annotations` methods:



```
def __call__(self, doc: Doc):
    predictions = self.predict([doc])
    self.set_annotations([doc], predictions)
    return doc
```

There is one more optional method to implement: `score`  calculates the performance of your component on a set of examples, and returns the results as a dictionary:

```
def score(self, examples: Iterable[Example]) -> Dict[str, Any]:
    prf = PRFScore()
    for example in examples:
        ...

    return {
        "rel_micro_p": prf.precision,
        "rel_micro_r": prf.recall,
        "rel_micro_f": prf.fscore,
    }
```

This is particularly useful for calculating relevant scores on the development corpus when training the component with `spacy train` .

Once our `TrainablePipe` subclass is fully implemented, we can [register](#) the component with the `@Language.factory`  decorator. This assigns it a name and lets you create the component with `nlp.add_pipe`  and via the [config](#).

```
from spacy.language import Language

@Language.factory("relation_extractor")
def make_relation_extractor(nlp, name, model):
    return RelationExtractor(nlp.vocab, model, name)
```

You can extend the decorator to include information such as the type of annotations that are required for this component to run, the type of annotations it produces, and the scores that can be calculated:

```
from spacy.language import Language

@Language.factory(
    "relation_extractor",
    requires=["doc.ents", "token.ent_iob", "token.ent_type"],
    assigns=["doc._.rel"],
    default_score_weights={
        "rel_micro_p": None,
        "rel_micro_r": None,
        "rel_micro_f": None,
    },
)
def make_relation_extractor(nlp, name, model):
    return RelationExtractor(nlp.vocab, model, name)
```

[SUGGEST EDITS](#)

READ NEXT
spaCy Projects

