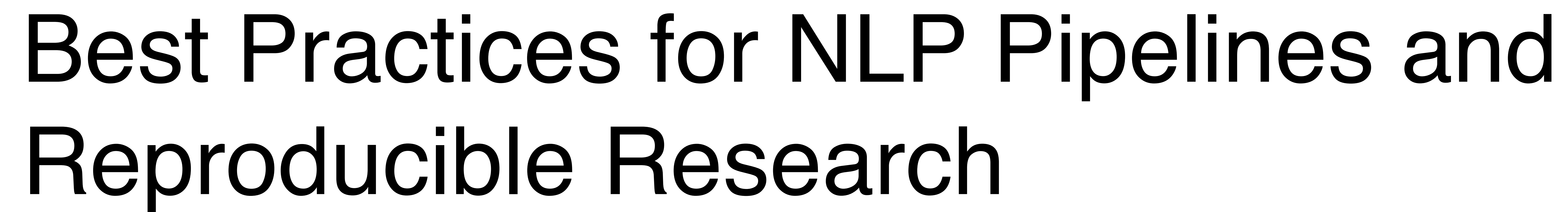

Best Practices for NLP Pipelines and
Reproducible Research

Vitalii Radchenko @ YouScan

You’ll find out

- What is a good pipeline
- How to process effectively input data
- How to build a train pipeline
- Why is a declarative syntax useful
- Reproducible research
- How to alter training to predictive pipeline with small efforts

Pipeline

- sequences of processing and analysis steps applied to data for a specific purpose
- save time on design time and coding, if you expect to encounter similar tasks
- simplifies deployment to production

What is a good pipeline?

- Reusable pipeline

  (applied to different tasks with minor changes)

- Structured

  (logical chain, easy understandable)

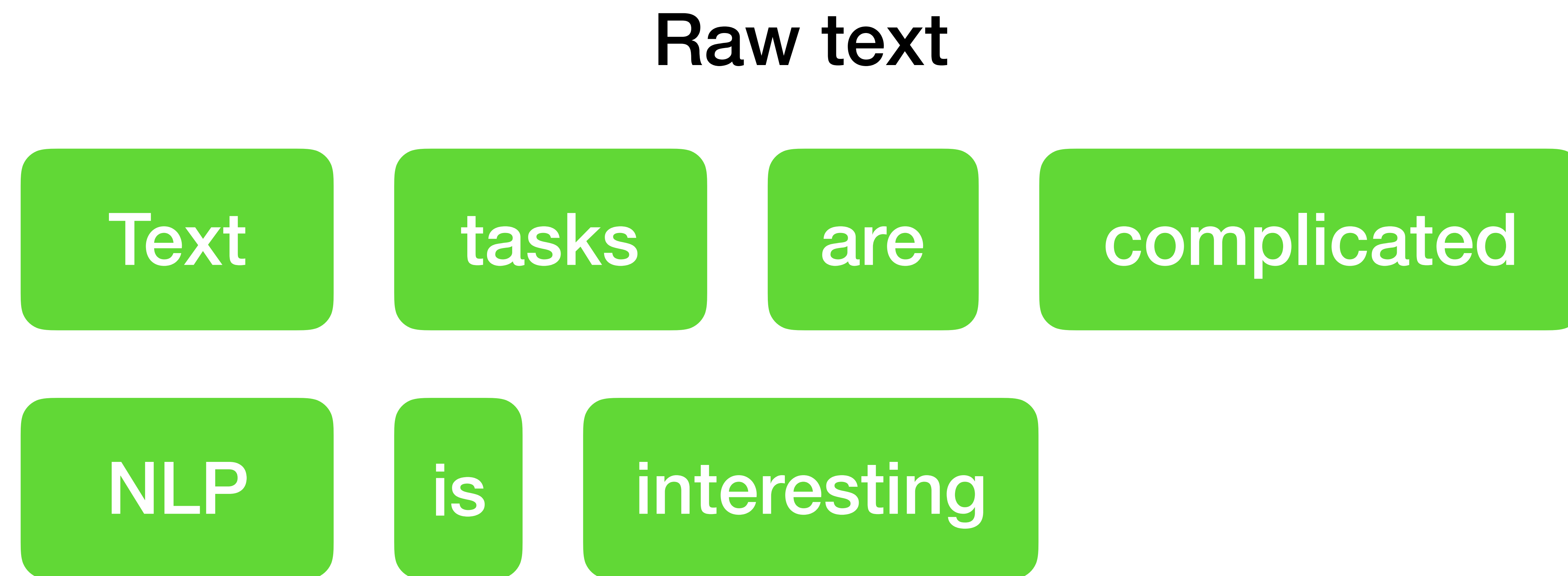
- Documented

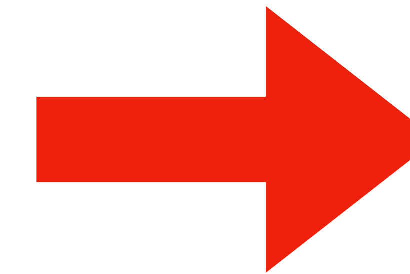
  (fully commented, defined arguments types, readme)

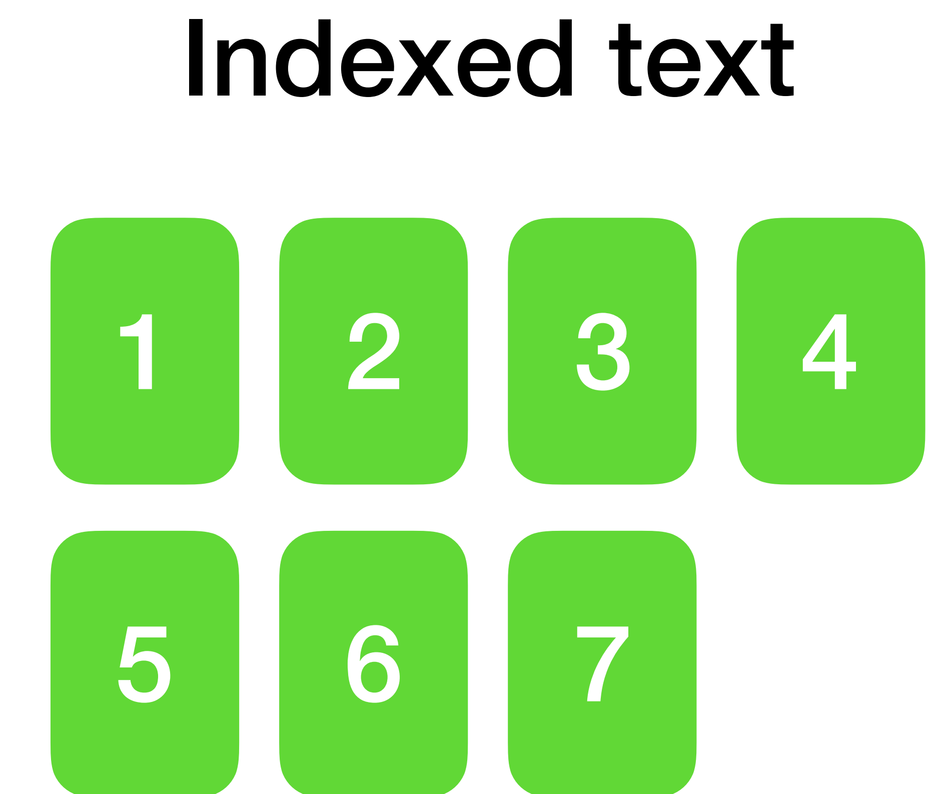
- Covered by tests

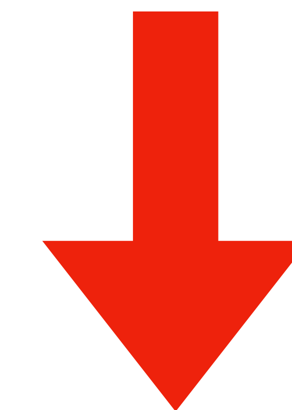
  (simple checks for input data/batch shape/model output, and unit tests for each pipeline step)

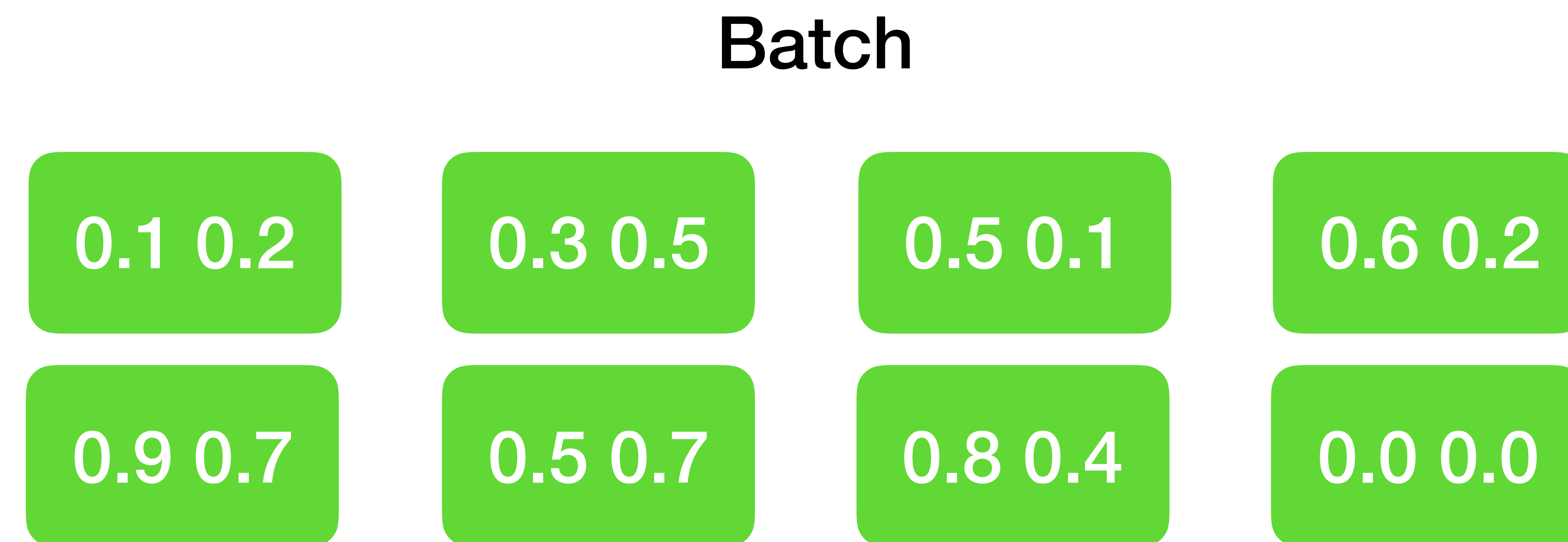
Problem statement

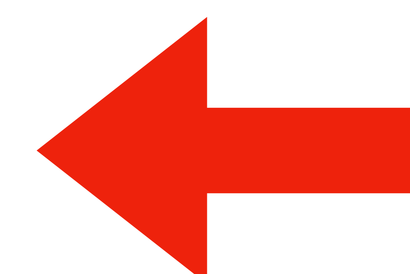












Input data in NLP

- Text is always a part of an input

  (token, sentence, article, dialogue, html page etc)

- Tags/Labels

- Spans (start and end indexes)

Field

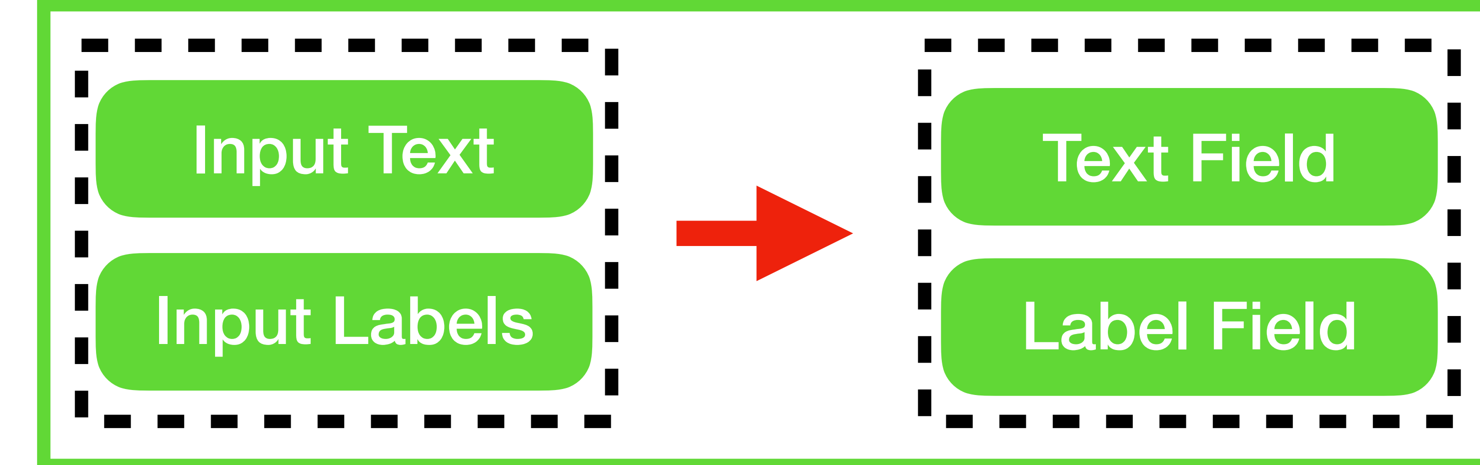
- Field is a main class which will be inherited by others
- Main methods:
  - **count_vocab_items** (count items for specific field)
  - **index** (field to vector of indexes, argument – a vocabulary)
  - **get_padding_lengths** (get field lengths)
  - **as_tensor** (padded tensor of indexes, argument – padding_lengths)
  - **batch_tensor** (create batch of fields, argument – list of tensors)

Field

Code

```
DataArray = TypeVar("DataArray", torch.Tensor, Dict[str, torch.Tensor])


class Field(Generic[DataArray]):
    def count_vocab_items(self, counter: Dict[str, Dict[str, int]]):
        pass

    def index(self, vocab):
        pass

    def get_padding_lengths(self) -> Dict[str, int]:
        raise NotImplementedError

    def as_tensor(self, padding_lengths: Dict[str, int]) -> DataArray:
        raise NotImplementedError

    def batch_tensors(self, tensor_list: List[DataArray], batch_first: bool) -> DataArray:
        return torch.stack(tensor_list)
```

Text Field



Text Field should have next methods:

- **preprocess** (preprocess text, argument – a list of preprocessors)
- **tokenize** (tokenize text, argument – a tokenizer)
- **index** (index text, argument – a vocabulary)
- **get padding length** (pad text, argument – maximum number of tokens)
- **as_tensor** (returns a tensor, basic method for all fields)

Text Field

Code

```
def _preprocess(self, text: str) -> str:
    return self.preprocessor.preprocess(text)

def _tokenize(self, text: str) -> List[str]:
    return self.tokenizer.tokenize(text=text)

@overrides
def index(self, vocab: Vocabulary):
    self._indexed_tokens = [vocab.get_token_index(token, self._text_namespace) for token in self._tokenized_text]

@overrides
def as_tensor(self, padding_length: Dict[str, int]) -> torch.Tensor:
    if self.sequence_length() >= padding_length["num_tokens"]:
        return torch.LongTensor(self._indexed_tokens[:padding_length["num_tokens"]])
    n_padded_elements = padding_length["num_tokens"] - self.sequence_length()
    return torch.cat([torch.LongTensor(self._indexed_tokens),
                      torch.zeros([n_padded_elements], dtype=torch.long)])

@overrides
def get_padding_lengths(self) -> Dict[str, int]:
    if self._indexed_tokens is None:
        raise ConfigurationError("You must call .index(vocabulary) on a field before determining padding lengths.")
    if self._max_padding_length is not None:
        return {"num_tokens": min(len(self._indexed_tokens), self._max_padding_length)}
    return {"num_tokens": len(self._indexed_tokens)}
```

Text Field

Code

```
def _preprocess(self, text: str) -> str:
    return self.preprocessor.preprocess(text)

def _tokenize(self, text: str) -> List[str]:
    return self.tokenizer.tokenize(text=text)

@overrides
def index(self, vocab: Vocabulary):
    self._indexed_tokens = [vocab.get_token_index(token, self._text_namespace) for token in self._tokenized_text]

@overrides
def as_tensor(self, padding_length: Dict[str, int]) -> torch.Tensor:
    if self.sequence_length() >= padding_length["num_tokens"]:
        return torch.LongTensor(self._indexed_tokens[:padding_length["num_tokens"]])
    n_padded_elements = padding_length["num_tokens"] - self.sequence_length()
    return torch.cat([torch.LongTensor(self._indexed_tokens),
                      torch.zeros([n_padded_elements], dtype=torch.long)])

@overrides
def get_padding_lengths(self) -> Dict[str, int]:
    if self._indexed_tokens is None:
        raise ConfigurationError("You must call .index(vocabulary) on a field before determining padding lengths.")
    if self._max_padding_length is not None:
        return {"num_tokens": min(len(self._indexed_tokens), self._max_padding_length)}
    return {"num_tokens": len(self._indexed_tokens)}
```

Preprocessing and tokenization

Text Field

Code

```
def _preprocess(self, text: str) -> str:
    return self.preprocessor.preprocess(text)

def _tokenize(self, text: str) -> List[str]:
    return self.tokenizer.tokenize(text=text)

@overrides
def index(self, vocab: Vocabulary):
    self._indexed_tokens = [vocab.get_token_index(token, self._text_namespace) for token in self._tokenized_text]

@overrides
def as_tensor(self, padding_length: Dict[str, int]) -> torch.Tensor:
    if self.sequence_length() >= padding_length["num_tokens"]:
        return torch.LongTensor(self._indexed_tokens[:padding_length["num_tokens"]])
    n_padded_elements = padding_length["num_tokens"] - self.sequence_length()
    return torch.cat([torch.LongTensor(self._indexed_tokens),
                      torch.zeros([n_padded_elements], dtype=torch.long)])

@overrides
def get_padding_lengths(self) -> Dict[str, int]:
    if self._indexed_tokens is None:
        raise ConfigurationError("You must call .index(vocabulary) on a field before determining padding lengths.")
    if self._max_padding_length is not None:
        return {"num_tokens": min(len(self._indexed_tokens), self._max_padding_length)}
    return {"num_tokens": len(self._indexed_tokens)}
```

Index fields using vocabulary

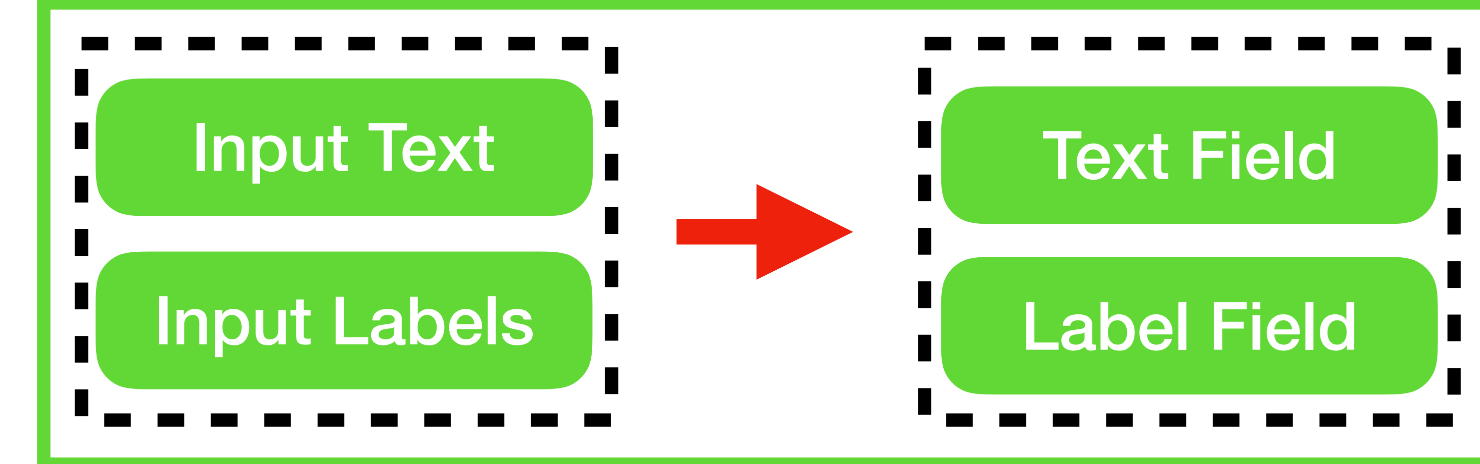
Text Field

Code

```
def _preprocess(self, text: str) -> str:
    return self.preprocessor.preprocess(text)

def _tokenize(self, text: str) -> List[str]:
    return self.tokenizer.tokenize(text=text)

@overrides
def index(self, vocab: Vocabulary):
    self._indexed_tokens = [vocab.get_token_index(token, self._text_namespace) for token in self._tokenized_text]

@overrides
def as_tensor(self, padding_length: Dict[str, int]) -> torch.Tensor:
    if self.sequence_length() >= padding_length["num_tokens"]:
        return torch.LongTensor(self._indexed_tokens[:padding_length["num_tokens"]])
    n_padded_elements = padding_length["num_tokens"] - self.sequence_length()
    return torch.cat([torch.LongTensor(self._indexed_tokens),
                      torch.zeros([n_padded_elements], dtype=torch.long)])

@overrides
def get_padding_lengths(self) -> Dict[str, int]:
    if self._indexed_tokens is None:
        raise ConfigurationError("You must call .index(vocabulary) on a field before determining padding lengths.")
    if self._max_padding_length is not None:
        return {"num_tokens": min(len(self._indexed_tokens), self._max_padding_length)}
    return {"num_tokens": len(self._indexed_tokens)}
```

Get padding length and return field as tensor

Label Field



Label field should be convertible to appropriate model format based on vocabulary:

- **index** (label names to index, argument – vocabulary)
- **as_tensor** (returns a tensor with a proper shape)
- **count_vocab_items** (count labels for vocabulary creating)

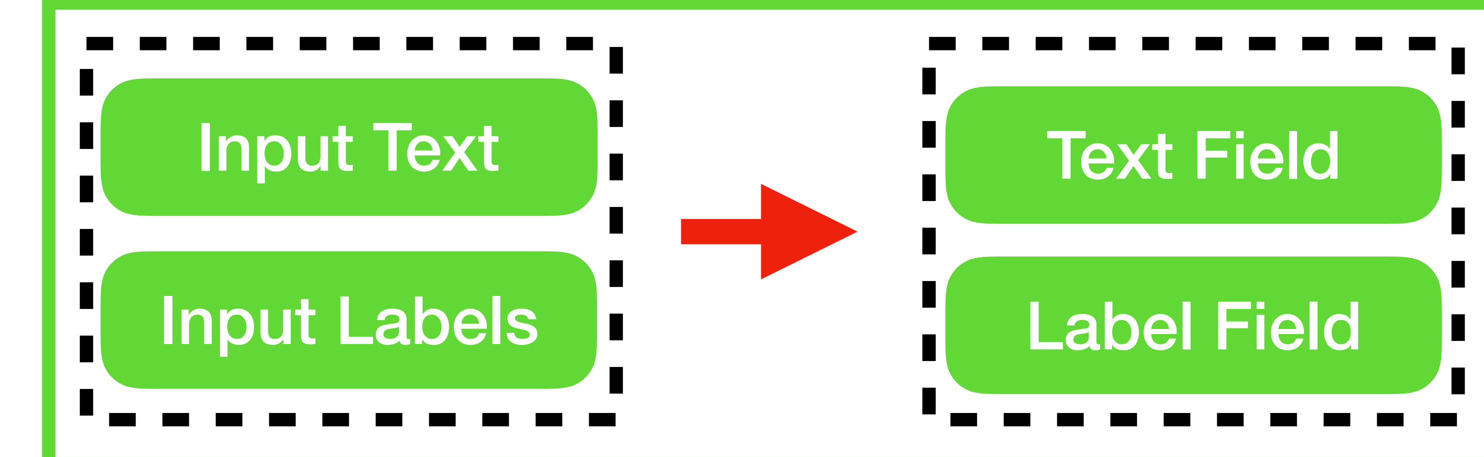
Label Field

Code

Count labels to create a vocabulary

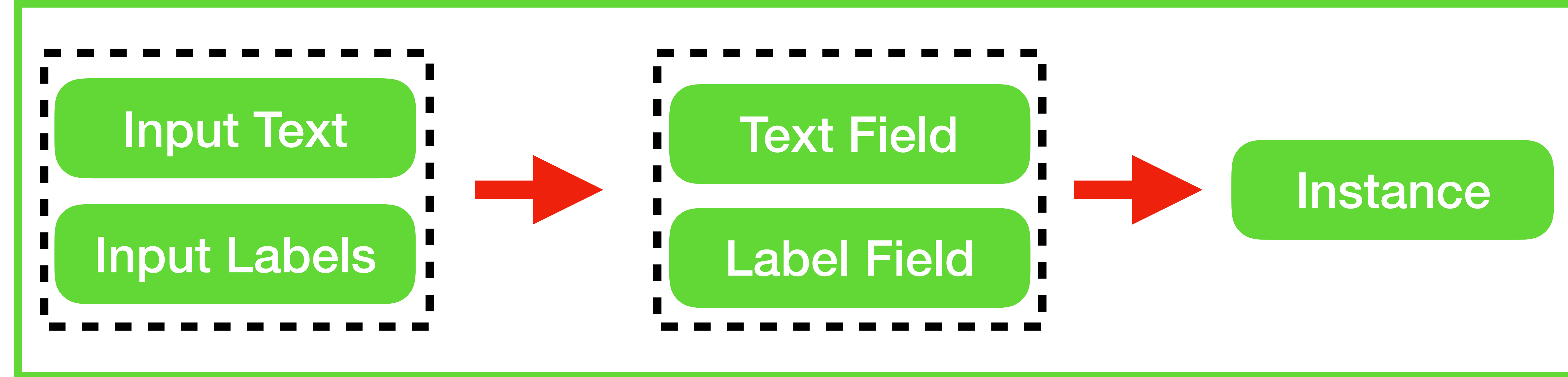
```
@overrides
def count_vocab_items(self, counter: Dict[str, Dict[str, int]]):
    if self._label_id is None:
        counter[self._label_namespace][self.label] += 1
    return counter

@overrides
def index(self, vocab: Vocabulary):
    if self._label_id is None:
        self._label_id = vocab.get_token_index(self.label, self._label_namespace)

@overrides
def as_tensor(self, padding_lengths: Dict[str, int]) -> torch.Tensor:
    tensor = torch.tensor(self._label_id, dtype=torch.long)
    return tensor
```

Other fields



- Metadata Field – other information which will not be used for training (raw text, author, topic, etc)index (label names to index, argument – vocabulary)
- Categorical Field – for any categorical data which could be encoded as embeddings or OHE (post type, source etc)
- Span Field – start, end or inside index
- Index field – index for the right answer over the sequence
- Create your own fields which contain “as tensor” method, for comfortable use inside your model

Instance



- Instance is a collection of fields
- One sample – one instance
- Has Mapping[str, Field] type, all fields are keyed
- Get field as tensor by key (“text”, “labels" – in our example)
- Should contain “**index_fields**” method to index all fields by the given vocabulary

Instance

Code

```
class Instance(Mapping[str, Field]):
    def __init__(self, fields: Dict[str, Field]) -> None:
        self.fields = fields
        self.indexed = False

    def __getitem__(self, key: str) -> Field:
        return self.fields[key]

    def __iter__(self):
        return iter(self.fields)

    def __len__(self) -> int:
        return len(self.fields)

    def add_field(self, field_name: str, vocab=None) -> None:
        self.fields[field_name] = field

    def index_fields(self, vocab, serial_index: int = None) -> None:
        if not self.indexed:
            self.indexed = True
            for field_name, field in self.fields.items():
                field.index(vocab)
            if "serial_index" not in self.fields and serial_index is not None:
                self.fields["serial_index"] = MetadataField(serial_index)

    def get_padding_lengths(self) -> Dict[str, Dict[str, int]]:
        lengths = {}
        for field_name, field in self.fields.items():
            lengths[field_name] = field.get_padding_lengths()
        return lengths

    def as_tensor_dict(self, padding_lengths: Dict[str, Dict[str, int]] = None) -> Dict[str, torch.tensor]:
        padding_lengths = padding_lengths or self.get_padding_lengths()
        tensors = {}
        for field_name, field in self.fields.items():
            tensors[field_name] = field.as_tensor(padding_lengths[field_name])
        return tensors
```

Instance

Code

```
class Instance(Mapping[str, Field]):
    def __init__(self, fields: Dict[str, Field]) -> None:
        self.fields = fields
        self.indexed = False

    def __getitem__(self, key: str) -> Field:
        return self.fields[key]

    def __iter__(self):
        return iter(self.fields)

    def __len__(self) -> int:
        return len(self.fields)

    def add_field(self, field_name: str, vocab=None) -> None:
        self.fields[field_name] = field

    def index_fields(self, vocab, serial_index: int = None) -> None:
        if not self.indexed:
            self.indexed = True
            for field_name, field in self.fields.items():
                field.index(vocab)
            if "serial_index" not in self.fields and serial_index is not None:
                self.fields["serial_index"] = MetadataField(serial_index)

    def get_padding_lengths(self) -> Dict[str, Dict[str, int]]:
        lengths = {}
        for field_name, field in self.fields.items():
            lengths[field_name] = field.get_padding_lengths()
        return lengths

    def as_tensor_dict(self, padding_lengths: Dict[str, Dict[str, int]] = None) -> Dict[str, torch.tensor]:
        padding_lengths = padding_lengths or self.get_padding_lengths()
        tensors = {}
        for field_name, field in self.fields.items():
            tensors[field_name] = field.as_tensor(padding_lengths[field_name])
        return tensors
```

Default settings

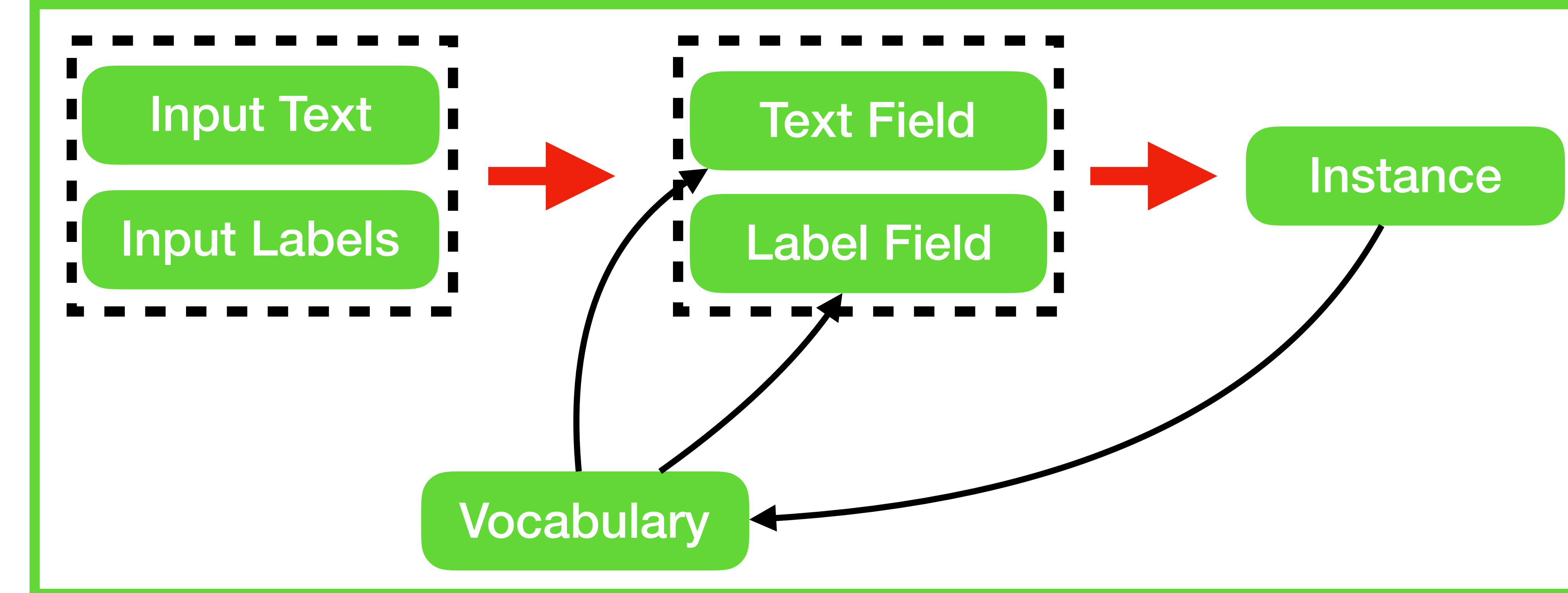
Instance

Code

```
class Instance(Mapping[str, Field]):
    def __init__(self, fields: Dict[str, Field]) -> None:
        self.fields = fields
        self.indexed = False

    def __getitem__(self, key: str) -> Field:
        return self.fields[key]

    def __iter__(self):
        return iter(self.fields)

    def __len__(self) -> int:
        return len(self.fields)

    def add_field(self, field_name: str, vocab=None) -> None:
        self.fields[field_name] = field

    def index_fields(self, vocab, serial_index: int = None) -> None:
        if not self.indexed:
            self.indexed = True
            for field_name, field in self.fields.items():
                field.index(vocab)
            if "serial_index" not in self.fields and serial_index is not None:
                self.fields["serial_index"] = MetadataField(serial_index)

    def get_padding_lengths(self) -> Dict[str, Dict[str, int]]:
        lengths = {}
        for field_name, field in self.fields.items():
            lengths[field_name] = field.get_padding_lengths()
        return lengths

    def as_tensor_dict(self, padding_lengths: Dict[str, Dict[str, int]] = None) -> Dict[str, torch.tensor]:
        padding_lengths = padding_lengths or self.get_padding_lengths()
        tensors = {}
        for field_name, field in self.fields.items():
            tensors[field_name] = field.as_tensor(padding_lengths[field_name])
        return tensors
```

Methods are applied to all fields

Vocabulary



- Vocabulary is a class where all vocabularies (text, labels, categories) are available by namespace
- We should be able to create a vocabulary from list of instances, counters or predefined files
- Add special tokens: OOV (Out-of-Vocabulary), padded
- Should have options to get statistics, “string to index” and “index to string”

Vocabulary

Code

```
@classmethod
def from_instances(cls,
                   instances: Iterable['adi.Instance'],
                   min_count: Dict[str, int] = None,
                   max_vocab_size: Union[int, Dict[str, int]] = None,
                   non_padded_namespaces: Iterable[str] = DEFAULT_NON_PADDED_NAMESPACES,
                   pretrained_files: Optional[Dict[str, str]] = None,
                   only_include_pretrained_words: bool = False,
                   tokens_to_add: Dict[str, List[str]] = None,
                   limit_pretrained_embeddings: Dict[str, int] = None,
                   exclude_fields: List[str] = ()) -> 'Vocabulary':
    namespace_token_counts: Dict[str, Dict[str, int]] = defaultdict(lambda: defaultdict(int))
    for instance in Tqdm.tqdm(instances):
        instance.count_vocab_items(namespace_token_counts, exclude_fields)

    return cls(counter=namespace_token_counts,
               min_count=min_count,
               max_vocab_size=max_vocab_size,
               non_padded_namespaces=non_padded_namespaces,
               pretrained_files=pretrained_files,
               only_include_pretrained_words=only_include_pretrained_words,
               tokens_to_add=tokens_to_add,
               limit_pretrained_embeddings=limit_pretrained_embeddings)
```

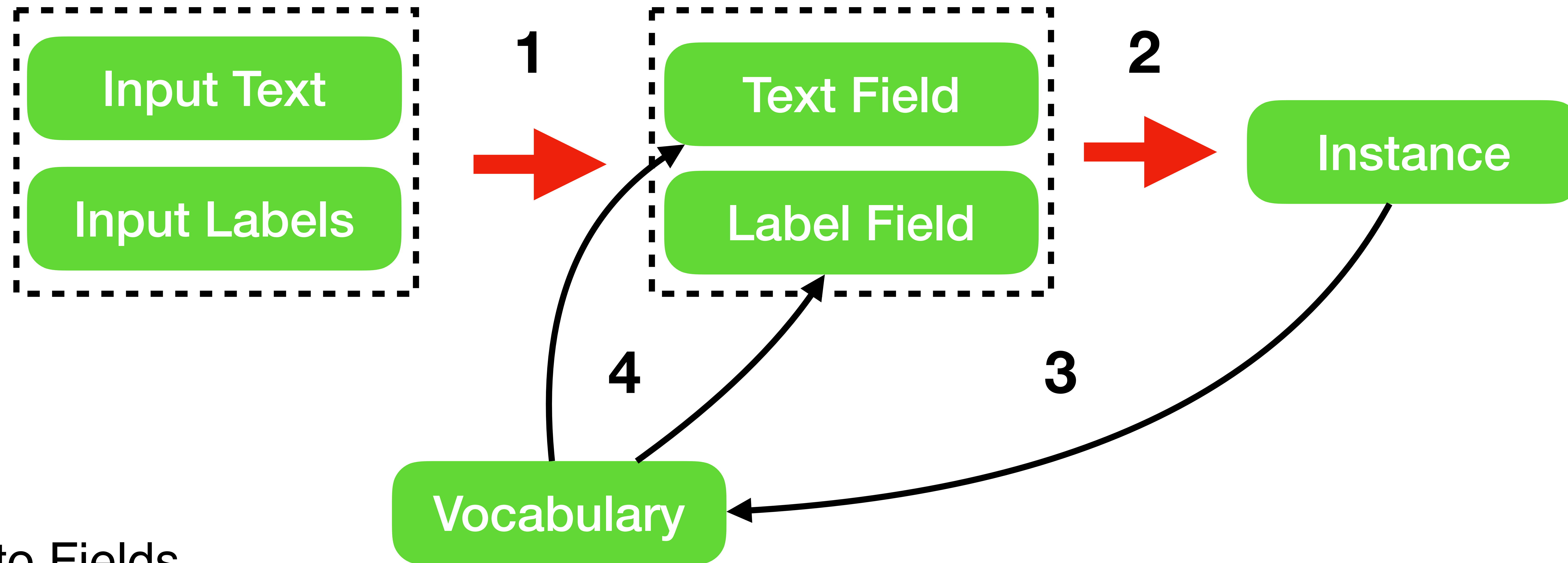
Vocabulary

Code

```
@classmethod
def from_instances(cls,
                   instances: Iterable['adi.Instance'],
                   min_count: Dict[str, int] = None,
                   max_vocab_size: Union[int, Dict[str, int]] = None,
                   non_padded_namespaces: Iterable[str] = DEFAULT_NON_PADDED_NAMESPACES,
                   pretrained_files: Optional[Dict[str, str]] = None,
                   only_include_pretrained_words: bool = False,
                   tokens_to_add: Dict[str, List[str]] = None,
                   limit_pretrained_embeddings: Dict[str, int] = None,
                   exclude_fields: List[str] = ()) -> 'Vocabulary':
    namespace_token_counts: Dict[str, Dict[str, int]] = defaultdict(lambda: defaultdict(int))
    for instance in Tqdm.tqdm(instances):
        instance.count_vocab_items(namespace_token_counts, exclude_fields)
    return cls(counter=namespace_token_counts,
               min_count=min_count,
               max_vocab_size=max_vocab_size,
               non_padded_namespaces=non_padded_namespaces,
               pretrained_files=pretrained_files,
               only_include_pretrained_words=only_include_pretrained_words,
               tokens_to_add=tokens_to_add,
               limit_pretrained_embeddings=limit_pretrained_embeddings)
```

Count items for all fields

Starting a pipeline schema



1. Input data to Fields
2. Fields to Instance
3. List of Instances to Vocabulary
4. Index Fields with Vocabulary

DataSet Reader

- Our pipeline schema should be used in dataset reader
- Each task should have own dataset reader in which you will construct instances and vocabulary
- Could be lazy

Iterator

- Gets output from Dataset Reader
- Types:
  - “Simple Iterator” just shuffles data
  - “Bucket Iterator” creates batches with the same length
- Index fields before training

Iterator

Code

```
def __call__(self,
             instances: Iterable[Instance],
             num_epochs: int = None,
             shuffle: bool = True) -> Iterator[TensorDict]:

    for epoch in range(num_epochs):
        batches = self._create_batches(instances, shuffle)

        for batch in batches:

            if self.vocab is not None:
                batch.index_instances(self.vocab)

            padding_lengths = batch.get_padding_lengths()
            tensor_dict = batch.as_tensor_dict(padding_lengths=padding_lengths,
                                               batch_first=self.batch_first)

            yield tensor_dict

def _create_batches(self, instances: Iterable[Instance], shuffle: bool) -> Iterable[Batch]:
    raise NotImplementedError
```

Iterator

Code

```
def __call__(self,
             instances: Iterable[Instance],
             num_epochs: int = None,
             shuffle: bool = True) -> Iterator[TensorDict]:

    for epoch in range(num_epochs):
        batches = self._create_batches(instances, shuffle)
        for batch in batches:

            if self.vocab is not None:
                batch.index_instances(self.vocab)

            padding_lengths = batch.get_padding_lengths()
            tensor_dict = batch.as_tensor_dict(padding_lengths=padding_lengths,
                                               batch_first=self.batch_first)

            yield tensor_dict

def _create_batches(self, instances: Iterable[Instance], shuffle: bool) -> Iterable[Batch]:
    raise NotImplementedError
```

Each iterator has its own create_batches method

Iterator

Code

```
def __call__(self,
             instances: Iterable[Instance],
             num_epochs: int = None,
             shuffle: bool = True) -> Iterator[TensorDict]:

    for epoch in range(num_epochs):
        batches = self._create_batches(instances, shuffle)

        for batch in batches:

            if self.vocab is not None:
                batch.index_instances(self.vocab)

            padding_lengths = batch.get_padding_lengths()
            tensor_dict = batch.as_tensor_dict(padding_lengths=padding_lengths,
                                               batch_first=self.batch_first)

            yield tensor_dict

def _create_batches(self, instances: Iterable[Instance], shuffle: bool) -> Iterable[Batch]:
    raise NotImplementedError
```

Index batch instances

Iterator

Code

```
def __call__(self,
             instances: Iterable[Instance],
             num_epochs: int = None,
             shuffle: bool = True) -> Iterator[TensorDict]:

    for epoch in range(num_epochs):
        batches = self._create_batches(instances, shuffle)

        for batch in batches:

            if self.vocab is not None:
                batch.index_instances(self.vocab)

            padding_lengths = batch.get_padding_lengths()
            tensor_dict = batch.as_tensor_dict(padding_lengths=padding_lengths,
                                               batch_first=self.batch_first)

            yield tensor_dict

def _create_batches(self, instances: Iterable[Instance], shuffle: bool) -> Iterable[Batch]:
    raise NotImplementedError
```

Batch to TensorDict

Batch

- Batch is an intermediate class, where we do a “dirty job”:
  - **index**
  - **get_padding_lengths**
  - **as_tensor**
- We prepare `TensorDict` as an input to our model

Batch (TensorDict type)

```
{
    “input”: torch.Size([70, 152]),
    “sequence_length”: torch.Size([70]),
    “labels”: torch.Size([70, 29]),
    “serial_index”: torch.Size([70])
}
```

Batch

Code

```
def index_instances(self, vocab: Vocabulary) -> None:
    for instance in self.instances:
        instance.index_fields(vocab)

def get_padding_lengths(self) -> Dict[str, Dict[str, int]]:
    padding_lengths: Dict[str, Dict[str, int]] = defaultdict(dict)
    all_instance_lengths: List[Dict[str, Dict[str, int]]] = [instance.get_padding_lengths()
                                                             for instance in self.instances]
    all_field_lengths: Dict[str, List[Dict[str, int]]] = defaultdict(list)
    for instance_lengths in all_instance_lengths:
        for field_name, instance_field_lengths in instance_lengths.items():
            all_field_lengths[field_name].append(instance_field_lengths)
    for field_name, field_lengths in all_field_lengths.items():
        for padding_key in field_lengths[0].keys():
            max_value = max(x[padding_key] if padding_key in x else 0 for x in field_lengths)
            padding_lengths[field_name][padding_key] = max_value
    return {**padding_lengths}

def as_tensor_dict(self,
                   padding_lengths: Dict[str, Dict[str, int]] = None,
                   batch_first: bool = False) -> Dict[str, Union[torch.Tensor, Dict[str, torch.Tensor]]]:

    instance_padding_lengths = self.get_padding_lengths()
    lengths_to_use: Dict[str, Dict[str, int]] = defaultdict(dict)
    for field_name, instance_field_lengths in instance_padding_lengths.items():
        for padding_key in instance_field_lengths.keys():
            if padding_lengths[field_name].get(padding_key) is not None:
                lengths_to_use[field_name][padding_key] = padding_lengths[field_name][padding_key]
            else:
                lengths_to_use[field_name][padding_key] = instance_field_lengths[padding_key]

    field_tensors: Dict[str, list] = defaultdict(list)
    for instance in self.instances:
        for field, tensors in instance.as_tensor_dict(lengths_to_use).items():
            field_tensors[field].append(tensors)

    field_classes = self.instances[0].fields
    final_fields = {}
    for field_name, field_tensor_list in field_tensors.items():
        final_fields[field_name] = field_classes[field_name].batch_tensors(field_tensor_list, batch_first)
    return final_fields
```

Batch

Code

```
def index_instances(self, vocab: Vocabulary) -> None:
    for instance in self.instances:
        instance.index_fields(vocab)
```

Index instances

```
def get_padding_lengths(self) -> Dict[str, Dict[str, int]]:
    padding_lengths: Dict[str, Dict[str, int]] = defaultdict(dict)
    all_instance_lengths: List[Dict[str, Dict[str, int]]] = [instance.get_padding_lengths()
                                                             for instance in self.instances]
    all_field_lengths: Dict[str, List[Dict[str, int]]] = defaultdict(list)
    for instance_lengths in all_instance_lengths:
        for field_name, instance_field_lengths in instance_lengths.items():
            all_field_lengths[field_name].append(instance_field_lengths)
    for field_name, field_lengths in all_field_lengths.items():
        for padding_key in field_lengths[0].keys():
            max_value = max(x[padding_key] if padding_key in x else 0 for x in field_lengths)
            padding_lengths[field_name][padding_key] = max_value
    return {**padding_lengths}

def as_tensor_dict(self,
                   padding_lengths: Dict[str, Dict[str, int]] = None,
                   batch_first: bool = False) -> Dict[str, Union[torch.Tensor, Dict[str, torch.Tensor]]]:

    instance_padding_lengths = self.get_padding_lengths()
    lengths_to_use: Dict[str, Dict[str, int]] = defaultdict(dict)
    for field_name, instance_field_lengths in instance_padding_lengths.items():
        for padding_key in instance_field_lengths.keys():
            if padding_lengths[field_name].get(padding_key) is not None:
                lengths_to_use[field_name][padding_key] = padding_lengths[field_name][padding_key]
            else:
                lengths_to_use[field_name][padding_key] = instance_field_lengths[padding_key]

    field_tensors: Dict[str, list] = defaultdict(list)
    for instance in self.instances:
        for field, tensors in instance.as_tensor_dict(lengths_to_use).items():
            field_tensors[field].append(tensors)

    field_classes = self.instances[0].fields
    final_fields = {}
    for field_name, field_tensor_list in field_tensors.items():
        final_fields[field_name] = field_classes[field_name].batch_tensors(field_tensor_list, batch_first)
    return final_fields
```

Batch

Code

```
def index_instances(self, vocab: Vocabulary) -> None:
    for instance in self.instances:
        instance.index_fields(vocab)

def get_padding_lengths(self) -> Dict[str, Dict[str, int]]:
    padding_lengths: Dict[str, Dict[str, int]] = defaultdict(dict)
    all_instance_lengths: List[Dict[str, Dict[str, int]]] = [instance.get_padding_lengths()
                                                             for instance in self.instances]
    all_field_lengths: Dict[str, List[Dict[str, int]]] = defaultdict(list)
    for instance_lengths in all_instance_lengths:
        for field_name, instance_field_lengths in instance_lengths.items():
            all_field_lengths[field_name].append(instance_field_lengths)
    for field_name, field_lengths in all_field_lengths.items():
        for padding_key in field_lengths[0].keys():
            max_value = max(x[padding_key] if padding_key in x else 0 for x in field_lengths)
            padding_lengths[field_name][padding_key] = max_value
    return {**padding_lengths}
```

Get padding length for each field

```
def as_tensor_dict(self,
                   padding_lengths: Dict[str, Dict[str, int]] = None,
                   batch_first: bool = False) -> Dict[str, Union[torch.Tensor, Dict[str, torch.Tensor]]]:

    instance_padding_lengths = self.get_padding_lengths()
    lengths_to_use: Dict[str, Dict[str, int]] = defaultdict(dict)
    for field_name, instance_field_lengths in instance_padding_lengths.items():
        for padding_key in instance_field_lengths.keys():
            if padding_lengths[field_name].get(padding_key) is not None:
                lengths_to_use[field_name][padding_key] = padding_lengths[field_name][padding_key]
            else:
                lengths_to_use[field_name][padding_key] = instance_field_lengths[padding_key]

    field_tensors: Dict[str, list] = defaultdict(list)
    for instance in self.instances:
        for field, tensors in instance.as_tensor_dict(lengths_to_use).items():
            field_tensors[field].append(tensors)

    field_classes = self.instances[0].fields
    final_fields = {}
    for field_name, field_tensor_list in field_tensors.items():
        final_fields[field_name] = field_classes[field_name].batch_tensors(field_tensor_list, batch_first)
    return final_fields
```

Batch

Code

```
def index_instances(self, vocab: Vocabulary) -> None:
    for instance in self.instances:
        instance.index_fields(vocab)

def get_padding_lengths(self) -> Dict[str, Dict[str, int]]:
    padding_lengths: Dict[str, Dict[str, int]] = defaultdict(dict)
    all_instance_lengths: List[Dict[str, Dict[str, int]]] = [instance.get_padding_lengths()
                                                            for instance in self.instances]
    all_field_lengths: Dict[str, List[Dict[str, int]]] = defaultdict(list)
    for instance_lengths in all_instance_lengths:
        for field_name, instance_field_lengths in instance_lengths.items():
            all_field_lengths[field_name].append(instance_field_lengths)
    for field_name, field_lengths in all_field_lengths.items():
        for padding_key in field_lengths[0].keys():
            max_value = max(x[padding_key] if padding_key in x else 0 for x in field_lengths)
            padding_lengths[field_name][padding_key] = max_value
    return {**padding_lengths}

def as_tensor_dict(self,
                   padding_lengths: Dict[str, Dict[str, int]] = None,
                   batch_first: bool = False) -> Dict[str, Union[torch.Tensor, Dict[str, torch.Tensor]]]:

    instance_padding_lengths = self.get_padding_lengths()
    lengths_to_use: Dict[str, Dict[str, int]] = defaultdict(dict)
    for field_name, instance_field_lengths in instance_padding_lengths.items():
        for padding_key in instance_field_lengths.keys():
            if padding_lengths[field_name].get(padding_key) is not None:
                lengths_to_use[field_name][padding_key] = padding_lengths[field_name][padding_key]
            else:
                lengths_to_use[field_name][padding_key] = instance_field_lengths[padding_key]

    field_tensors: Dict[str, list] = defaultdict(list)
    for instance in self.instances:
        for field, tensors in instance.as_tensor_dict(lengths_to_use).items():
            field_tensors[field].append(tensors)

    field_classes = self.instances[0].fields
    final_fields = {}
    for field_name, field_tensor_list in field_tensors.items():
        final_fields[field_name] = field_classes[field_name].batch_tensors(field_tensor_list, batch_first)
    return final_fields
```

Get tensors

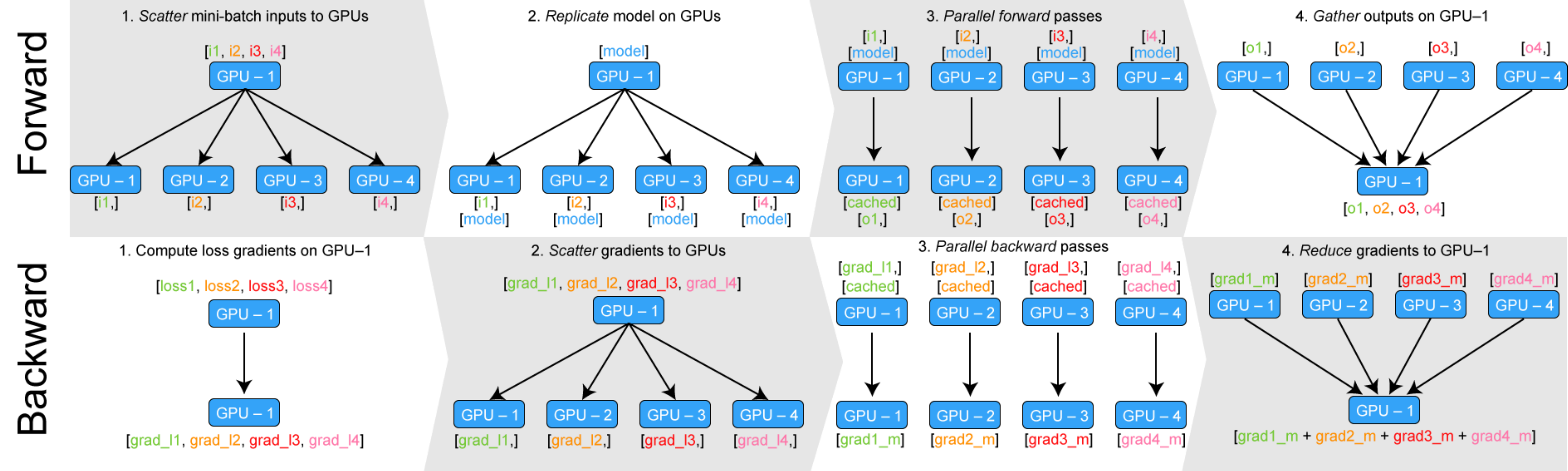
Batch

Code

```
def index_instances(self, vocab: Vocabulary) -> None:
    for instance in self.instances:
        instance.index_fields(vocab)

def get_padding_lengths(self) -> Dict[str, Dict[str, int]]:
    padding_lengths: Dict[str, Dict[str, int]] = defaultdict(dict)
    all_instance_lengths: List[Dict[str, Dict[str, int]]] = [instance.get_padding_lengths()
                                                             for instance in self.instances]
    all_field_lengths: Dict[str, List[Dict[str, int]]] = defaultdict(list)
    for instance_lengths in all_instance_lengths:
        for field_name, instance_field_lengths in instance_lengths.items():
            all_field_lengths[field_name].append(instance_field_lengths)
    for field_name, field_lengths in all_field_lengths.items():
        for padding_key in field_lengths[0].keys():
            max_value = max(x[padding_key] if padding_key in x else 0 for x in field_lengths)
            padding_lengths[field_name][padding_key] = max_value
    return {**padding_lengths}

def as_tensor_dict(self,
                   padding_lengths: Dict[str, Dict[str, int]] = None,
                   batch_first: bool = False) -> Dict[str, Union[torch.Tensor, Dict[str, torch.Tensor]]]:

    instance_padding_lengths = self.get_padding_lengths()
    lengths_to_use: Dict[str, Dict[str, int]] = defaultdict(dict)
    for field_name, instance_field_lengths in instance_padding_lengths.items():
        for padding_key in instance_field_lengths.keys():
            if padding_lengths[field_name].get(padding_key) is not None:
                lengths_to_use[field_name][padding_key] = padding_lengths[field_name][padding_key]
            else:
                lengths_to_use[field_name][padding_key] = instance_field_lengths[padding_key]

    field_tensors: Dict[str, list] = defaultdict(list)
    for instance in self.instances:
        for field, tensors in instance.as_tensor_dict(lengths_to_use).items():
            field_tensors[field].append(tensors)

    field_classes = self.instances[0].fields
    final_fields = {}
    for field_name, field_tensor_list in field_tensors.items():
        final_fields[field_name] = field_classes[field_name].batch_tensors(field_tensor_list, batch_first)
    return final_fields
```

Aggregate in batch

Model



Training Neural Nets on Larger Batches:
Practical Tips for 1-GPU, Multi-GPU & Distributed setups,
Thomas Wolf

- Model should have next methods
  - from_config (build model from config)
  - _load (load model weights)
- Calculate loss and metrics inside forward pass:
  - loss and metrics depend on your task and could be very specific —> easier implementation
  - for correct parallelization

Model

Code

```
def forward(self, text: torch.Tensor,
            sequence_length: torch.Tensor = None,
            labels: torch.Tensor = None,
            serial_index: torch.Tensor = None) -> Dict[str, torch.Tensor]:
    ...
    if labels is not None:
        loss_fn = Loss.by_name(self._loss["type"])(**self._loss["params"])
        output["loss"] = loss_fn(prediction_scores, labels)
        for metric in self.metrics.values():
            metric(class_probabilities.float(), labels.float())
    label_names = self._predictions_to_labels(class_probabilities)
    output["label_names"] = label_names
    ...
    return output

def get_metrics(self, reset: bool = False) -> Dict[str, float]:
    metrics_to_return = {}
    for model_metric_name, metric in self.metrics.items():
        for metric_name, metric_value in metric.get_metric(reset).items():
            metrics_to_return[metric_name] = metric_value
    return metrics_to_return
```

Model

Code

```
def forward(self, text: torch.Tensor,
            sequence_length: torch.Tensor = None,
            labels: torch.Tensor = None,
            serial_index: torch.Tensor = None) -> Dict[str, torch.Tensor]:
    ...
    if labels is not None:
        loss_fn = Loss.by_name(self._loss["type"])(**self._loss["params"])
        output["loss"] = loss_fn(prediction_scores, labels)
        for metric in self.metrics.values():
            metric(class_probabilities.float(), labels.float())
    label_names = self._predictions_to_labels(class_probabilities)
    output["label_names"] = label_names
    ...
    return output

def get_metrics(self, reset: bool = False) -> Dict[str, float]:
    metrics_to_return = {}
    for model_metric_name, metric in self.metrics.items():
        for metric_name, metric_value in metric.get_metric(reset).items():
            metrics_to_return[metric_name] = metric_value
    return metrics_to_return
```

Loss

Model

Code

```
def forward(self, text: torch.Tensor,
            sequence_length: torch.Tensor = None,
            labels: torch.Tensor = None,
            serial_index: torch.Tensor = None) -> Dict[str, torch.Tensor]:
    ...
    if labels is not None:
        loss_fn = Loss.by_name(self._loss["type"])(**self._loss["params"])
        output["loss"] = loss_fn(prediction_scores, labels)
        for metric in self.metrics.values():
            metric(class_probabilities.float(), labels.float())
    label_names = self._predictions_to_labels(class_probabilities)
    output["label_names"] = label_names
    ...
    return output
```

Metrics

```
def get_metrics(self, reset: bool = False) -> Dict[str, float]:
    metrics_to_return = {}
    for model_metric_name, metric in self.metrics.items():
        for metric_name, metric_value in metric.get_metric(reset).items():
            metrics_to_return[metric_name] = metric_value
    return metrics_to_return
```

Model

Code

```
def forward(self, text: torch.Tensor,
            sequence_length: torch.Tensor = None,
            labels: torch.Tensor = None,
            serial_index: torch.Tensor = None) -> Dict[str, torch.Tensor]:
    ...
    if labels is not None:
        loss_fn = Loss.by_name(self._loss["type"])(**self._loss["params"])
        output["loss"] = loss_fn(prediction_scores, labels)
        for metric in self.metrics.values():
            metric(class_probabilities.float(), labels.float())
    label_names = self._predictions_to_labels(class_probabilities)
    output["label_names"] = label_names
    ...
    return output

def get_metrics(self, reset: bool = False) -> Dict[str, float]:
    metrics_to_return = {}
    for model_metric_name, metric in self.metrics.items():
        for metric_name, metric_value in metric.get_metric(reset).items():
            metrics_to_return[metric_name] = metric_value
    return metrics_to_return
```

Other useful outputs

Trainer

Training process is fully configurable in Trainer:

- Train and Validation dataset (List of Instances)
- Iterator
- Model
- Optimizer
- Train configurations (number of epochs, shuffling, metrics)
- Callbacks (early stopping, learning rate schedule, logging etc)

Trainer. Initialization

Code

```
def __init__(self,
             model: Model,
             optimizer: torch.optim.Optimizer,
             iterator: DataIterator,
             train_dataset: Iterable[Instance],
             validation_dataset: Optional[Iterable[Instance]] = None,
             patience: Optional[int] = None,
             early_stopping_metric: str = "-loss",
             validation_iterator: DataIterator = None,
             shuffle: bool = True,
             num_epochs: int = 20,
             tb_logging_dir: str = "tb_logs",
             serialization_dir: Optional[str] = None,
             accumulation_steps: int = 0,
             experiment_name: Optional[str] = None,
             cuda_device: Union[int, List] = -1,
             grad_norm: Optional[float] = 1.0,
             lr_scheduler: Optional[Dict] = None,
             fp16: bool = False,
             fp16_opt_level: str = "O1",
             gradual_unfreezing_steps: Optional[List[List[str]]] = ()) -> None:
```

Trainer. Initialization

Code

```
def __init__(self,
             model: Model,
             optimizer: torch.optim.Optimizer,
             iterator: DataIterator,
             train_dataset: Iterable[Instance],
             validation_dataset: Optional[Iterable[Instance]] = None,
             patience: Optional[int] = None,
             early_stopping_metric: str = "-loss",
             validation_iterator: DataIterator = None,
             shuffle: bool = True,
             num_epochs: int = 20,
             tb_logging_dir: str = "tb_logs",
             serialization_dir: Optional[str] = None,
             accumulation_steps: int = 0,
             experiment_name: Optional[str] = None,
             cuda_device: Union[int, List] = -1,
             grad_norm: Optional[float] = 1.0,
             lr_scheduler: Optional[Dict] = None,
             fp16: bool = False,
             fp16_opt_level: str = "O1",
             gradual_unfreezing_steps: Optional[List[List[str]]] = ()) -> None:
```

Already predefined

Trainer. Initialization

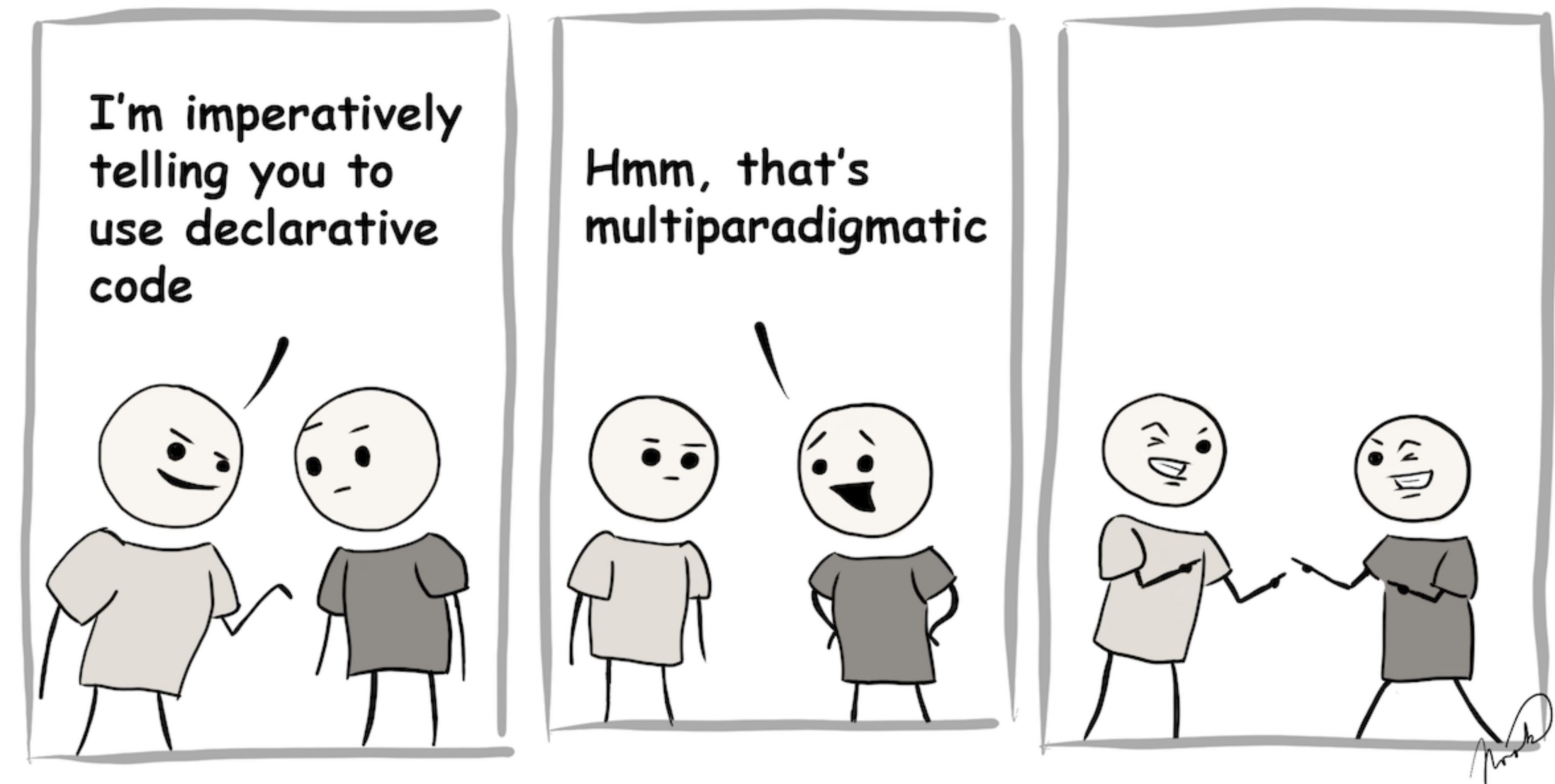
Code

```
def __init__(self,
             model: Model,
             optimizer: torch.optim.Optimizer,
             iterator: DataIterator,
             train_dataset: Iterable[Instance],
             validation_dataset: Optional[Iterable[Instance]] = None,
             patience: Optional[int] = None,
             early_stopping_metric: str = "-loss",
             validation_iterator: DataIterator = None,
             shuffle: bool = True,
             num_epochs: int = 20,
             tb_logging_dir: str = "tb_logs",
             serialization_dir: Optional[str] = None,
             accumulation_steps: int = 0,
             experiment_name: Optional[str] = None,
             cuda_device: Union[int, List] = -1,
             grad_norm: Optional[float] = 1.0,
             lr_scheduler: Optional[Dict] = None,
             fp16: bool = False,
             fp16_opt_level: str = "O1",
             gradual_unfreezing_steps: Optional[List[List[str]]] = ()) -> None:
```

Training configurations

Declarative syntax

- focus on building logic of software without actually describing its flow
- allows us to specify an entire experiment using JSON
- allows us to change architectures without changing code



Config

Data, Iterator

Data

```
"data_folder": "data/training_data",
"dataset_reader": {
   "type": "multilabel_classification"
}
"preprocessing": {
    "lower": true,
    "max_seq_len": 150,
    "include_length": true,
    "label_one_hot": true,
    "preprocessors": [ ["HtmlEntitiesUnescaper"], ["BoldTagReplacer"],
                       ["HtmlTagReplacer", [" "]], ["URLReplacer", [" urlTag "]]]
 }
```

Iterator

```
"type": "bucket_iterator",
"params": {
   "batch_size": 1000,
   "shuffle": true,
   "sort_key": {
      "field": "text",
      "type": "length"
   },
  "biggest_batch_first": true,
  "batch_first": true
}
```

Config

Model, Optimizer, Trainer

Model

```
"type": "mixed_rnn",
"params": {
    "embedding_dropout": 0.3,
    "rnn_1": {
        "rnn_type": "lstm",
        "hidden_cells": 100,
        "hidden_layers": 1
    },
    "rnn_2": {
        "rnn_type": "gru",
        "hidden_cells": 100,
        "hidden_layers": 1
    },
    "aggregation_layers": {
        "types": ["max_pool", "mean_pool"]
    },
    "activation": “sigmoid”,
    "loss": {
        "type": “bce_with_logits",
        "params": {}
    }
    "metrics": {
        "fscore": {
            "average": “macro"
        }
    }
}
```

Optimizer

```
"optimizer": {
    "type": "adam",
    "params": {"lr": 0.001 }
}
```

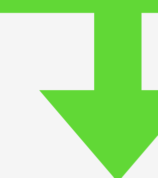
Trainer

```
"serialization_dir": "models",
"accumulation_steps": 2,     "grad_norm": 1,
"num_epochs": 4,
"cuda_device": 0,
"patience": 2,
"early_stopping_metric": "-loss",     "lr_scheduler":
{
    “type”: “w_linear",
    "params": {
        "warmup_steps": 300
    },
"fp16": true,
“fp16_opt_level": "O2"
```

How does declarative syntax work?

- get model class “**by name**”
- initialize model with “**from_config**” method
- the same approach for other objects
  (Dataset Readers, Iterators, Optimizers, Loss, etc.)

```
class Model(torch.nn.Module, Registrable):
    ...

@Model.register("mixed_rnn")
class MixedRnn(Model):
    def __init__(self,
                 word_embeddings: TextFieldEmbedder,
                 vocab: Vocabulary,
                 model_params: Dict[str, Any],
                 label_namespace: str = "labels"):
    ...

@Model.register("transformer_bert_model")
class TransformerBert(Model):
    def __init__(self,
                 transformer: PreTrainedModel,
                 vocab: Vocabulary,
                 model_params: Dict[str, Any],
                 label_namespace: str = "labels"):
    ...
```



```
Model.by_name("transformer_bert_model").from_config(config, vocab, cuda_device)
```

Model training pipeline

Code

```
training_config = TrainingConfig.read(args.training_config_file)
logger = training_config.logger()

dataset_loader = DatasetReader.by_name(
    training_config.data_preprocessing["dataset_reader"])(
        **training_config.data_preprocessing["params"])
train_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                              "train.csv"))
val_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                            "validation.csv"))

vocab = Vocabulary.from_instances(train_data + val_data,
                                  **training_config.vocabulary)

iterator = DataIterator.by_name(training_config.iterator["type"])(
    **training_config.iterator["params"])
iterator.index_with(vocab)

model = Model.from_config(training_config.model, vocab)

optimizer = Optimizer.by_name(training_config.optimizer["type"])(
    model.parameters(), **training_config.optimizer["params"])

trainer = Trainer(model=model,
                  optimizer=optimizer,
                  iterator=iterator,
                  train_dataset=train_data,
                  validation_dataset=val_data,
                  experiment_name=training_config.experiment_name,
                  **training_config.trainer)
trainer.train()
vocab.save_to_files(os.path.join(training_config.trainer["serialization_dir"],
                                 "vocabulary"))
training_config.save()
```

Model training pipeline

Code

```
training_config = TrainingConfig.read(args.training_config_file)
logger = training_config.logger()
```

Load config file

```
dataset_loader = DatasetReader.by_name(
    training_config.data_preprocessing["dataset_reader"])(
        **training_config.data_preprocessing["params"])
train_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                              "train.csv"))
val_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                            "validation.csv"))

vocab = Vocabulary.from_instances(train_data + val_data,
                                  **training_config.vocabulary)

iterator = DataIterator.by_name(training_config.iterator["type"])(
    **training_config.iterator["params"])
iterator.index_with(vocab)

model = Model.from_config(training_config.model, vocab)

optimizer = Optimizer.by_name(training_config.optimizer["type"])(
    model.parameters(), **training_config.optimizer["params"])

trainer = Trainer(model=model,
                  optimizer=optimizer,
                  iterator=iterator,
                  train_dataset=train_data,
                  validation_dataset=val_data,
                  experiment_name=training_config.experiment_name,
                  **training_config.trainer)
trainer.train()
vocab.save_to_files(os.path.join(training_config.trainer["serialization_dir"],
                                 "vocabulary"))
training_config.save()
```

Model training pipeline

Code

```
training_config = TrainingConfig.read(args.training_config_file)
logger = training_config.logger()

dataset_loader = DatasetReader.by_name(
    training_config.data_preprocessing["dataset_reader"])(
        **training_config.data_preprocessing["params"])
train_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                              "train.csv"))
val_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                            "validation.csv"))

vocab = Vocabulary.from_instances(train_data + val_data,
                                  **training_config.vocabulary)

iterator = DataIterator.by_name(training_config.iterator["type"])(
    **training_config.iterator["params"])
iterator.index_with(vocab)

model = Model.from_config(training_config.model, vocab)

optimizer = Optimizer.by_name(training_config.optimizer["type"])(
    model.parameters(), **training_config.optimizer["params"])

trainer = Trainer(model=model,
                  optimizer=optimizer,
                  iterator=iterator,
                  train_dataset=train_data,
                  validation_dataset=val_data,
                  experiment_name=training_config.experiment_name,
                  **training_config.trainer)
trainer.train()
vocab.save_to_files(os.path.join(training_config.trainer["serialization_dir"],
                                 "vocabulary"))
training_config.save()
```

Load data

Model training pipeline

Code

```
training_config = TrainingConfig.read(args.training_config_file)
logger = training_config.logger()

dataset_loader = DatasetReader.by_name(
    training_config.data_preprocessing["dataset_reader"])(
        **training_config.data_preprocessing["params"])
train_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                              "train.csv"))
val_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                            "validation.csv"))

vocab = Vocabulary.from_instances(train_data + val_data,
                                  **training_config.vocabulary)

iterator = DataIterator.by_name(training_config.iterator["type"])(
    **training_config.iterator["params"])
iterator.index_with(vocab)

model = Model.from_config(training_config.model, vocab)

optimizer = Optimizer.by_name(training_config.optimizer["type"])(
    model.parameters(), **training_config.optimizer["params"])

trainer = Trainer(model=model,
                  optimizer=optimizer,
                  iterator=iterator,
                  train_dataset=train_data,
                  validation_dataset=val_data,
                  experiment_name=training_config.experiment_name,
                  **training_config.trainer)
trainer.train()
vocab.save_to_files(os.path.join(training_config.trainer["serialization_dir"],
                                 "vocabulary"))
training_config.save()
```

Create vocab

Model training pipeline

Code

```
training_config = TrainingConfig.read(args.training_config_file)
logger = training_config.logger()

dataset_loader = DatasetReader.by_name(
    training_config.data_preprocessing["dataset_reader"])(
        **training_config.data_preprocessing["params"])
train_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                              "train.csv"))
val_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                            "validation.csv"))

vocab = Vocabulary.from_instances(train_data + val_data,
                                  **training_config.vocabulary)

iterator = DataIterator.by_name(training_config.iterator["type"])(
    **training_config.iterator["params"])
iterator.index_with(vocab)

model = Model.from_config(training_config.model, vocab)

optimizer = Optimizer.by_name(training_config.optimizer["type"])(
    model.parameters(), **training_config.optimizer["params"])

trainer = Trainer(model=model,
                  optimizer=optimizer,
                  iterator=iterator,
                  train_dataset=train_data,
                  validation_dataset=val_data,
                  experiment_name=training_config.experiment_name,
                  **training_config.trainer)
trainer.train()
vocab.save_to_files(os.path.join(training_config.trainer["serialization_dir"],
                                 "vocabulary"))
training_config.save()
```

Initialize iterator

Model training pipeline

Code

```
training_config = TrainingConfig.read(args.training_config_file)
logger = training_config.logger()

dataset_loader = DatasetReader.by_name(
    training_config.data_preprocessing["dataset_reader"])(
        **training_config.data_preprocessing["params"])
train_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                              "train.csv"))
val_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                            "validation.csv"))

vocab = Vocabulary.from_instances(train_data + val_data,
                                  **training_config.vocabulary)

iterator = DataIterator.by_name(training_config.iterator["type"])(
    **training_config.iterator["params"])
iterator.index_with(vocab)

model = Model.from_config(training_config.model, vocab)

optimizer = Optimizer.by_name(training_config.optimizer["type"])(
    model.parameters(), **training_config.optimizer["params"])

trainer = Trainer(model=model,
                  optimizer=optimizer,
                  iterator=iterator,
                  train_dataset=train_data,
                  validation_dataset=val_data,
                  experiment_name=training_config.experiment_name,
                  **training_config.trainer)
trainer.train()
vocab.save_to_files(os.path.join(training_config.trainer["serialization_dir"],
                                 "vocabulary"))
training_config.save()
```

Define model and set optimizer

Model training pipeline

Code

```
training_config = TrainingConfig.read(args.training_config_file)
logger = training_config.logger()

dataset_loader = DatasetReader.by_name(
    training_config.data_preprocessing["dataset_reader"])(
        **training_config.data_preprocessing["params"])
train_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                              "train.csv"))
val_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                            "validation.csv"))

vocab = Vocabulary.from_instances(train_data + val_data,
                                  **training_config.vocabulary)

iterator = DataIterator.by_name(training_config.iterator["type"])(
    **training_config.iterator["params"])
iterator.index_with(vocab)

model = Model.from_config(training_config.model, vocab)

optimizer = Optimizer.by_name(training_config.optimizer["type"])(
    model.parameters(), **training_config.optimizer["params"])

trainer = Trainer(model=model,
                  optimizer=optimizer,
                  iterator=iterator,
                  train_dataset=train_data,
                  validation_dataset=val_data,
                  experiment_name=training_config.experiment_name,
                  **training_config.trainer)
trainer.train()
vocab.save_to_files(os.path.join(training_config.trainer["serialization_dir"],
                                 "vocabulary"))
training_config.save()
```

Training loop

Model training pipeline

Code

```
training_config = TrainingConfig.read(args.training_config_file)
logger = training_config.logger()

dataset_loader = DatasetReader.by_name(
    training_config.data_preprocessing["dataset_reader"])(
        **training_config.data_preprocessing["params"])
train_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                              "train.csv"))
val_data = dataset_loader.read(os.path.join(training_config.data_folder,
                                            "validation.csv"))

vocab = Vocabulary.from_instances(train_data + val_data,
                                  **training_config.vocabulary)

iterator = DataIterator.by_name(training_config.iterator["type"])(
    **training_config.iterator["params"])
iterator.index_with(vocab)

model = Model.from_config(training_config.model, vocab)

optimizer = Optimizer.by_name(training_config.optimizer["type"])(
    model.parameters(), **training_config.optimizer["params"])

trainer = Trainer(model=model,
                  optimizer=optimizer,
                  iterator=iterator,
                  train_dataset=train_data,
                  validation_dataset=val_data,
                  experiment_name=training_config.experiment_name,
                  **training_config.trainer)
trainer.train()
vocab.save_to_files(os.path.join(training_config.trainer["serialization_dir"],
                                 "vocabulary"))
training_config.save()
```

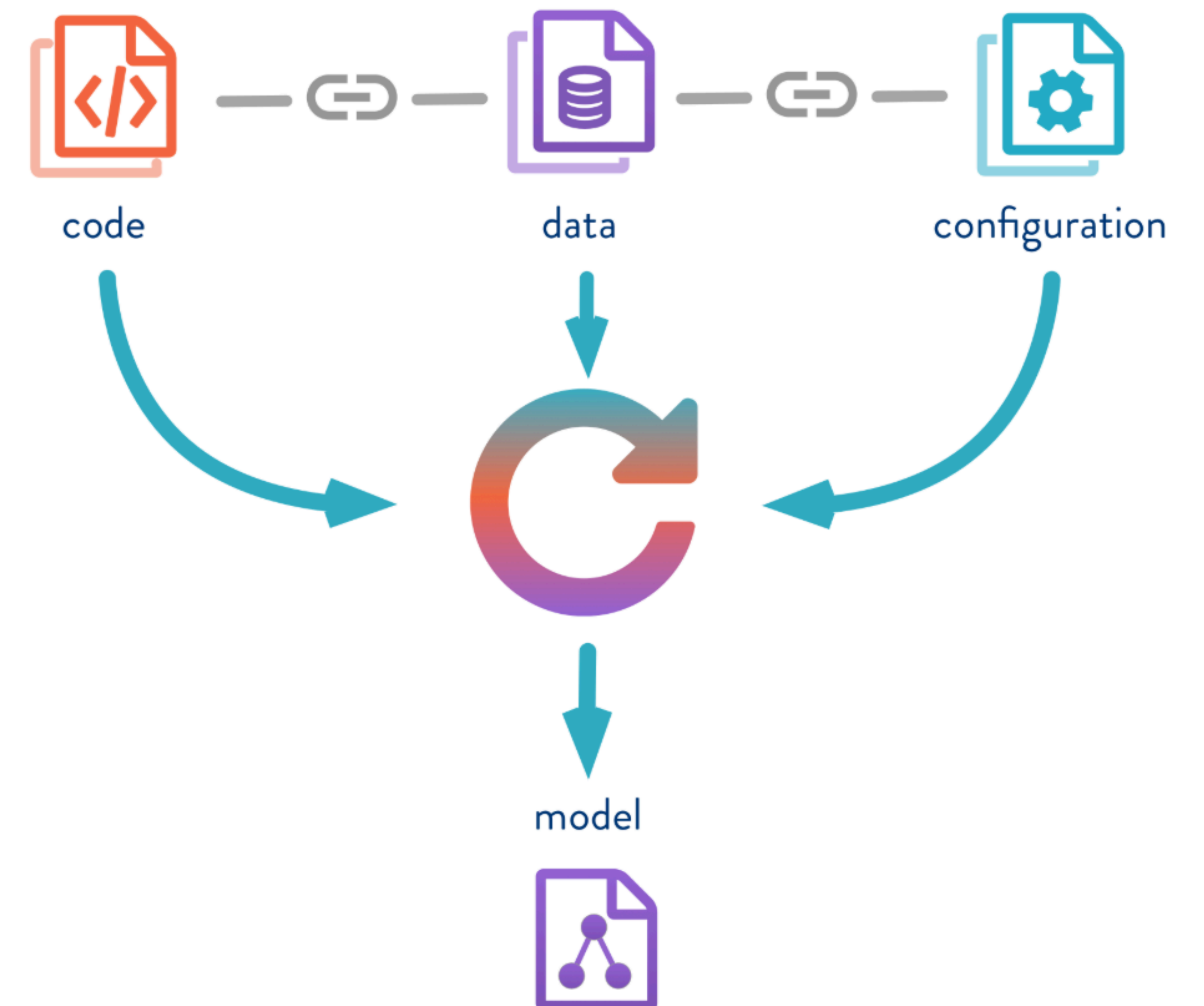
Save vocab and config file

Save model

- **Config file** (you can reproduce an experiment if you have a config file)
- **Vocabulary** (match embeddings indexes or labels)
- **Model weights** (load model)

Reproduce experiments

- To reproduce experiments you should have the same:
  - code
  - data
  - configuration files
- DVC is the best choice for this purpose



DVC pipeline

- Add data to DVC
- **dvc run** pipeline code (data preprocessing —> model training —> model evaluation)
- change something in your pipeline code
- make new **git checkout**
- **dvc repro** will reproduce your experiment with new changed code (dvc detects changes and run your pipeline one more time)
- commit code and experiment result

DVC

DVC

Easy to version data

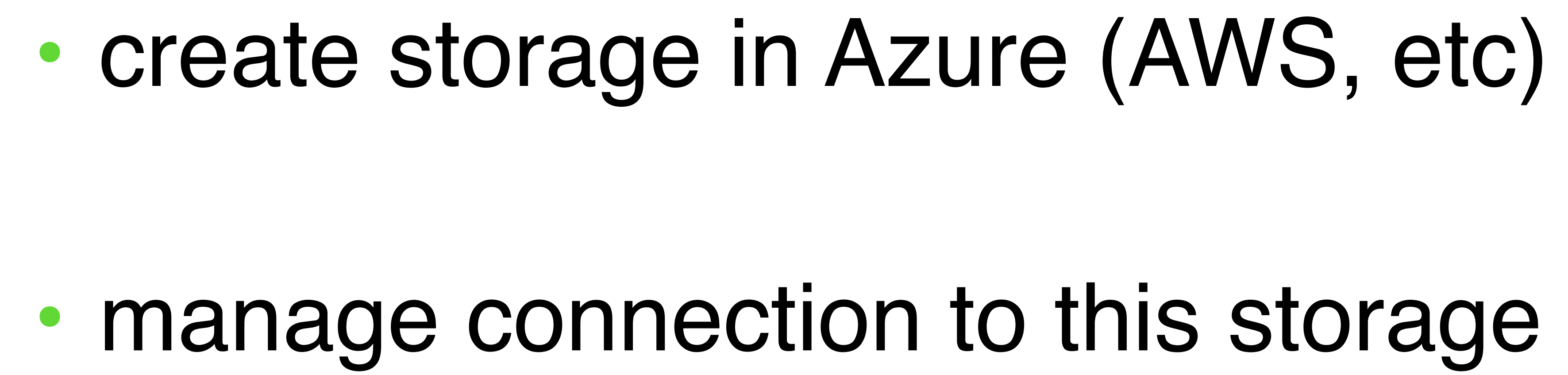
DVC

Easy to version data

- create storage in Azure (AWS, etc)

DVC

Easy to version data



- create storage in Azure (AWS, etc)
- manage connection to this storage

DVC

Easy to version data

- create storage in Azure (AWS, etc)
- manage connection to this storage
- create local dvc cache

DVC

Easy to version data

- create storage in Azure (AWS, etc)
- manage connection to this storage
- create local dvc cache
- use "dvc push" and "dvc pull" to upload/download data

DVC

Easy to version data

- create storage in Azure (AWS, etc)
- manage connection to this storage
- create local dvc cache
- use "dvc push" and "dvc pull" to upload/download data
- commit dvc config

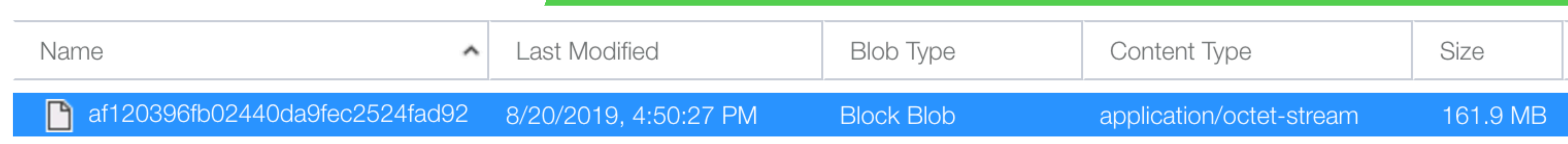

```
md5: c0070f0c9d6fd660c185070e21a59046
wdir: .
outs:
- md5: 24af120396fb02440da9fec2524fad92
  path: raw_data.zip
  cache: true
  metric: false
  persist: false
```

DVC

Easy to version data

- create storage in Azure (AWS, etc)
- manage connection to this storage
- create local dvc cache
- use "dvc push" and "dvc pull" to upload/download data
- commit dvc config
- download required data by hash

```
md5: c0070f0c9d6fd660c185070e21a59046
wdir: .
outs:
- md5: 24af120396fb02440da9fec2524fad92
  path: raw_data.zip
  cache: true
  metric: false
  persist: false
```



Name	Last Modified	Blob Type	Content Type	Size
af120396fb02440da9fec2524fad92	8/20/2019, 4:50:27 PM	Block Blob	application/octet-stream	161.9 MB

DVC

I don't recommend to use DVC for a pipeline

- we declare all training settings in configuration file and can store just config file
- if we want to compare training curves, we should create a dummy config, where a name of the experiment will be used (not convenient)
- each experiment will have its own GitHub branch (don’t need this)

Reproducible workflow

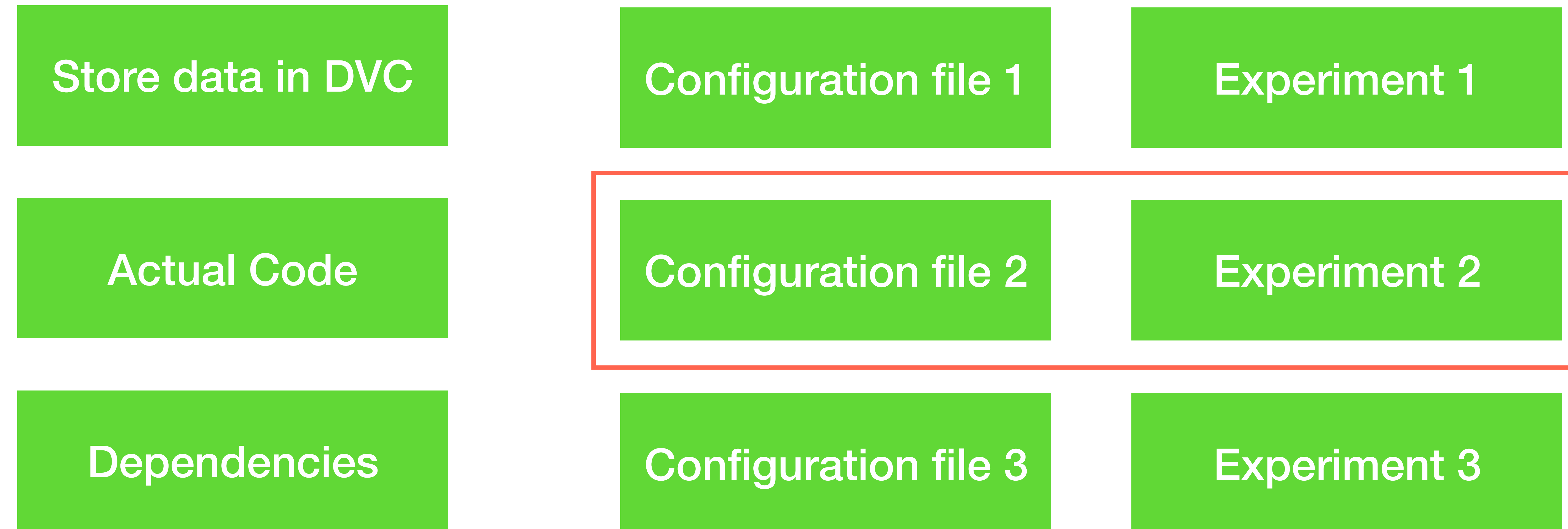
Store data in DVC

Actual Code

Dependencies

Each version has its unique data
and not changeable your code
with dependencies versions

Reproducible workflow



Each version has its unique data
and not changeable your code
with dependencies versions

Reproducible workflow

Run experiments, look at metrics and choose the best configuration

Each version has its unique data and not changeable your code with dependencies versions

You can always pull this commit, download data by hash, reproduce environment (from requirements.txt) and experiments (from config)

Predictor

- Input data – JSON-format (API format)
- Dataset Reader adopted to JSON-format
- Iterator for batch processing
- Options to get predictions with probabilities, return top-5 most probable labels, change thresholds and other options for validating results

Json DataSet Reader

Code

```
@DatasetReader.register("json_dataset_reader")
class JsonDatasetReader(DatasetReader):
    ...
    @overrides
    def _read(self, file_path: str) -> Iterable[Instance]:
        data: List[Dict] = load_pickle(file_path)
        for sample in data:
            text = sample["text"]
            labels = sample["labels"]
            instance = self.text_to_instance(text=text, labels=labels)
            if instance is not None:
                yield instance

    @overrides
    def text_to_instance(self, text: str, labels: Union[str, int] = None) -> Instance:
        fields: Dict[str, Field] = {}
        fields["text"] = TextField(text=text,
                                   preprocessor=self.preprocessor,
                                   tokenizer=self.tokenizer,
                                   max_padding_length=self.max_padding_length)
        if labels is not None:
            fields["labels"] = MultiLabelField(labels=labels)
        return Instance(fields)
```

is used for data preprocessing in Predictor

Predictor

Code

```
class Predictor(Registrable):
    ...
    def predict_batch_json(self, inputs: List[JsonDict]) -> List[JsonDict]:
        instances = self._batch_json_to_instances(inputs)
        return self.predict_batch_instance(instances)

    def _batch_json_to_instances(self, json_dicts: List[JsonDict]) -> List[Instance]:
        instances = []
        for json_dict in json_dicts:
            instances.append(self._json_to_instance(json_dict))
        return instances

    def _json_to_instance(self, json_dict: JsonDict) -> Instance:
        raise NotImplementedError

    def predict_batch_instance(self, instances: List[Instance]) -> List[JsonDict]:
        data_iterator = self._iterator(instances, num_epochs=1, shuffle=False)
        predictions = []
        with torch.no_grad():
            for batch in data_iterator:
                batch = nn_util.move_to_device(batch, self._cuda_device)
                predictions.append(self._model(**batch))
        sorted_predictions = sorted(predictions, key=lambda x: x["serial_index"])
        return sorted_predictions
```

Predictor

Code

```
class Predictor(Registrable):
    ...
    def predict_batch_json(self, inputs: List[JsonDict]) -> List[JsonDict]:
        instances = self._batch_json_to_instances(inputs)
        return self.predict_batch_instance(instances)

    def _batch_json_to_instances(self, json_dicts: List[JsonDict]) -> List[Instance]:
        instances = []
        for json_dict in json_dicts:
            instances.append(self._json_to_instance(json_dict))
        return instances

    def _json_to_instance(self, json_dict: JsonDict) -> Instance:
        raise NotImplementedError

    def predict_batch_instance(self, instances: List[Instance]) -> List[JsonDict]:
        data_iterator = self._iterator(instances, num_epochs=1, shuffle=False)
        predictions = []
        with torch.no_grad():
            for batch in data_iterator:
                batch = nn_util.move_to_device(batch, self._cuda_device)
                predictions.append(self._model(**batch))
        sorted_predictions = sorted(predictions, key=lambda x: x["serial_index"])
        return sorted_predictions
```

main method

Predictor

Code

```
class Predictor(Registrable):
    ...
    def predict_batch_json(self, inputs: List[JsonDict]) -> List[JsonDict]:
     1  instances = self._batch_json_to_instances(inputs)
        return self.predict_batch_instance(instances)

    def _batch_json_to_instances(self, json_dicts: List[JsonDict]) -> List[Instance]:
        instances = []
        for json_dict in json_dicts:
  1         instances.append(self._json_to_instance(json_dict))
        return instances

    def _json_to_instance(self, json_dict: JsonDict) -> Instance:
        raise NotImplementedError

    def predict_batch_instance(self, instances: List[Instance]) -> List[JsonDict]:
        data_iterator = self._iterator(instances, num_epochs=1, shuffle=False)
        predictions = []
        with torch.no_grad():
            for batch in data_iterator:
                batch = nn_util.move_to_device(batch, self._cuda_device)
                predictions.append(self._model(**batch))
        sorted_predictions = sorted(predictions, key=lambda x: x["serial_index"])
        return sorted_predictions
```

Json to List[Instance]

Predictor

Code

```
class Predictor(Registrable):
    ...
    def predict_batch_json(self, inputs: List[JsonDict]) -> List[JsonDict]:
        instances = self._batch_json_to_instances(inputs)
      2 return self.predict_batch_instance(instances)

    def _batch_json_to_instances(self, json_dicts: List[JsonDict]) -> List[Instance]:
        instances = []
        for json_dict in json_dicts:
            instances.append(self._json_to_instance(json_dict))
        return instances

    def _json_to_instance(self, json_dict: JsonDict) -> Instance:
        raise NotImplementedError

    def predict_batch_instance(self, instances: List[Instance]) -> List[JsonDict]:
        data_iterator = self._iterator(instances, num_epochs=1, shuffle=False)
        predictions = []
        with torch.no_grad():
  2         for batch in data_iterator:
                batch = nn_util.move_to_device(batch, self._cuda_device)
                predictions.append(self._model(**batch))
        sorted_predictions = sorted(predictions, key=lambda x: x["serial_index"])
        return sorted_predictions
```

Making predictions

Production pipeline

Code

```
deploy_config = DeployConfig.read(DEPLOY_CONFIG_PATH)

deploy_dataset_reader = DatasetReader.by_name(
    deploy_config.data_preprocessing["dataset_reader"])(
    **deploy_config.data_preprocessing["params"])

iterator = DataIterator.by_name(deploy_config.iterator["type"])(
    **deploy_config.iterator["params"])
model = Model.load(deploy_config.model, deploy_config.serialization_dir,
                   cuda_device=deploy_config.predictor["cuda_device"])
predictor = Predictor.by_name(deploy_config.predictor["type"])(
    model=model,
    dataset_reader=deploy_dataset_reader,
    iterator=iterator,
    cuda_device=deploy_config.predictor[
    "cuda_device"])
...
predictor.predict_batch_json(mentions)
```

Production pipeline

Code

```
deploy_config = DeployConfig.read(DEPLOY_CONFIG_PATH)
```

Load config file

```
deploy_dataset_reader = DatasetReader.by_name(
    deploy_config.data_preprocessing["dataset_reader"])(
    **deploy_config.data_preprocessing["params"])

iterator = DataIterator.by_name(deploy_config.iterator["type"])(
    **deploy_config.iterator["params"])
model = Model.load(deploy_config.model, deploy_config.serialization_dir,
                   cuda_device=deploy_config.predictor["cuda_device"])
predictor = Predictor.by_name(deploy_config.predictor["type"])(
    model=model,
    dataset_reader=deploy_dataset_reader,
    iterator=iterator,
    cuda_device=deploy_config.predictor[
    "cuda_device"])
...
predictor.predict_batch_json(mentions)
```

Production pipeline

Code

```
deploy_config = DeployConfig.read(DEPLOY_CONFIG_PATH)

deploy_dataset_reader = DatasetReader.by_name(
    deploy_config.data_preprocessing["dataset_reader"])(
    **deploy_config.data_preprocessing["params"])

iterator = DataIterator.by_name(deploy_config.iterator["type"])(
    **deploy_config.iterator["params"])
model = Model.load(deploy_config.model, deploy_config.serialization_dir,
                   cuda_device=deploy_config.predictor["cuda_device"])
predictor = Predictor.by_name(deploy_config.predictor["type"])(
    model=model,
    dataset_reader=deploy_dataset_reader,
    iterator=iterator,
    cuda_device=deploy_config.predictor[
    "cuda_device"])
...
predictor.predict_batch_json(mentions)
```

Initialize dataset reader

Production pipeline

Code

```
deploy_config = DeployConfig.read(DEPLOY_CONFIG_PATH)

deploy_dataset_reader = DatasetReader.by_name(
    deploy_config.data_preprocessing["dataset_reader"])(
    **deploy_config.data_preprocessing["params"])

iterator = DataIterator.by_name(deploy_config.iterator["type"])(
    **deploy_config.iterator["params"])
model = Model.load(deploy_config.model, deploy_config.serialization_dir,
                   cuda_device=deploy_config.predictor["cuda_device"])
predictor = Predictor.by_name(deploy_config.predictor["type"])(
    model=model,
    dataset_reader=deploy_dataset_reader,
    iterator=iterator,
    cuda_device=deploy_config.predictor[
    "cuda_device"])
...
predictor.predict_batch_json(mentions)
```

Define iterator

Production pipeline

Code

```
deploy_config = DeployConfig.read(DEPLOY_CONFIG_PATH)

deploy_dataset_reader = DatasetReader.by_name(
    deploy_config.data_preprocessing["dataset_reader"])(
    **deploy_config.data_preprocessing["params"])

iterator = DataIterator.by_name(deploy_config.iterator["type"])(
    **deploy_config.iterator["params"])
model = Model.load(deploy_config.model, deploy_config.serialization_dir,
                   cuda_device=deploy_config.predictor["cuda_device"])
predictor = Predictor.by_name(deploy_config.predictor["type"])(
    model=model,
    dataset_reader=deploy_dataset_reader,
    iterator=iterator,
    cuda_device=deploy_config.predictor[
    "cuda_device"])
...
predictor.predict_batch_json(mentions)
```

Load model

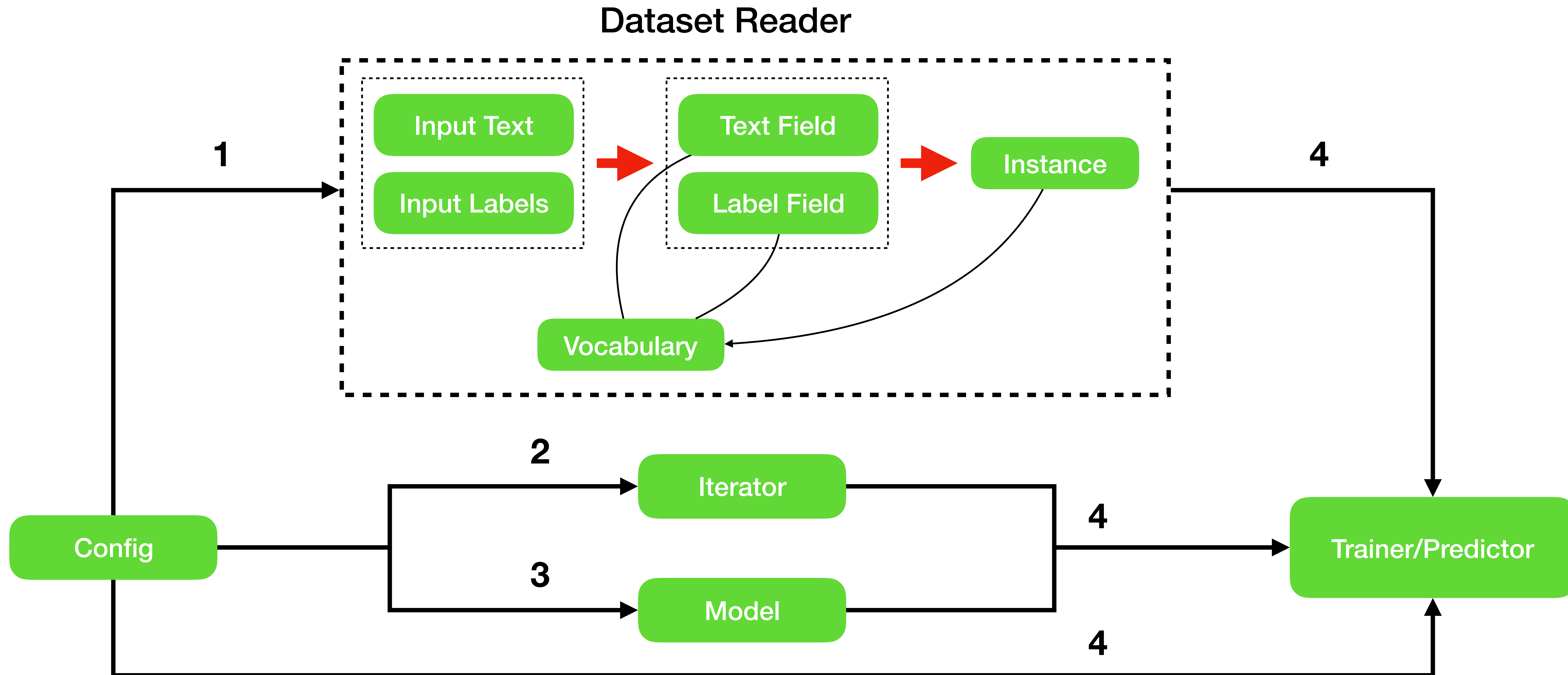
Production pipeline

Code

```
deploy_config = DeployConfig.read(DEPLOY_CONFIG_PATH)

deploy_dataset_reader = DatasetReader.by_name(
    deploy_config.data_preprocessing["dataset_reader"])(
    **deploy_config.data_preprocessing["params"])

iterator = DataIterator.by_name(deploy_config.iterator["type"])(
    **deploy_config.iterator["params"])
model = Model.load(deploy_config.model, deploy_config.serialization_dir,
                   cuda_device=deploy_config.predictor["cuda_device"])
predictor = Predictor.by_name(deploy_config.predictor["type"])(
    model=model,
    dataset_reader=deploy_dataset_reader,
    iterator=iterator,
    cuda_device=deploy_config.predictor[
    "cuda_device"])
...
predictor.predict_batch_json(mentions)
```

Define predictor and integrate it in your production pipeline

Pipeline schema



Conclusions

- Abstractions make a pipeline more structured, logical and understandable
- But be careful with that in case that every new abstraction makes your code more complicated
- Declarative syntax helps to keep a pipeline simple and set the whole experiment in config without code changing
- With good base, prototyping and deployment become very fast

Resources

- AllenNLP GitHub: https://github.com/allenai/allennlp
- Writing Code for NLP Research: https://bit.ly/2Desi4b
- TorchText: https://torchtext.readthedocs.io/en/latest/
- DVC: https://dvc.org/

Vitalii Radchenko

Data Scientist @ YouScan

ODS-slack: @vradchenko

Email: radchenko.vitaliy.o@gmail.com

FB: vradchenko