ml4ir

Release 0.2.0

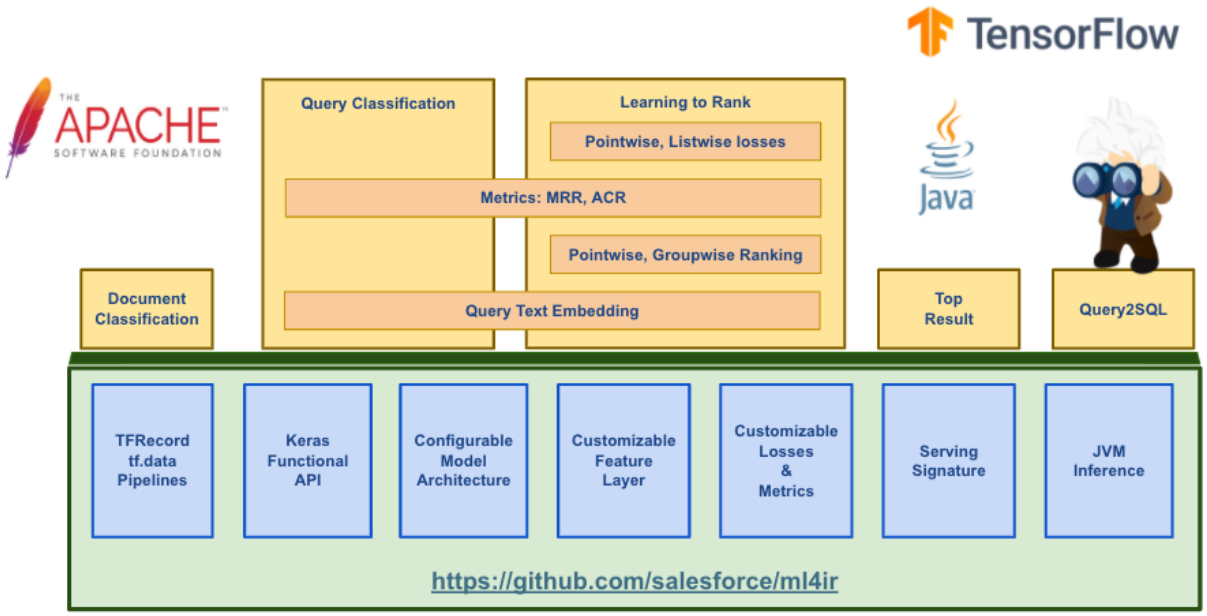
Feb 24, 2023

Contents

ml4ir is an open source library for training and deploying deep learning models for search applications. ml4ir is built on top of **python3** and **tensorflow 2.x** for training and evaluation. It also comes packaged with scala utilities for **JVM inference**.

ml4ir is designed as modular subcomponents which can easily be combined and customized to build a variety of search ML models such as:

- Learning to Rank
- Query Auto Completion
- Document Classification
- Query Classification
- Named Entity Recognition
- Top Results
- Query2SQL
- *add your application here*



CHAPTER 1

Motivation

Search is a complex data space with lots of different types of ML tasks working on a combination of structured and unstructured data sources. There existed no single library that

- provides an end-to-end training and serving solution for a variety of search applications
- allows training of models with limited coding expertise
- allows easy customization to build complex models to tackle a variety of problems in the search domain
- focuses on performance, robustness and offline-online feature parity
- enables fast prototyping

So, we built ml4ir.

CHAPTER 2

Guiding Principles

Customizable Library

Firstly, we want ml4ir to be an easy-to-use and highly customizable library so that you can build the search application of your need. ml4ir allows each of its subcomponents to be overriden, mixed and match with other custom modules to create and deploy models.

Configurable Toolkit

While ml4ir can be used as library, it also comes prepackaged with all the popular search based losses, metrics, embeddings, layers, etc. to enable someone with limited tensorflow expertise to quickly load their training data and train models for the task of interest. ml4ir achieves this by following a hybrid approach which allow for each subcomponent to be completely controlled through configurations alone. Most search based ML applications can be built this way.

Performance First

ml4ir is built using the TFRecord data pipeline, which is the recommended data format for tensorflow data loading. We combine ml4ir's high configurability with out of the box tensorflow data optimization utilities to define model features and build a data pipeline that easily allows training on huge amounts of data. ml4ir also comes packaged with utilities to convert data from CSV and libsvm format to TFRecord.

Training-Serving Handshake

As ml4ir is a common library for training and serving deep learning models, this allows us to build tight integration and fault tolerance into the models that are trained. ml4ir also uses the same configuration files for both training and inference keeping the end-to-end handshake clean. This allows user's to easily plug in any feature store(or solr) into ml4ir's serving utilities to deploy models in one's production environments.

Search Model Hub

The goal of ml4ir is to form a common hub for the most popular deep learning layers, losses, metrics, embeddings used in the search domain. We've built ml4ir with a focus on quick prototyping with wide variety of network architectures and optimizations. We encourage contributors to add to ml4ir's arsenal of search deep learning utilities as we continue to do so ourselves.

CHAPTER 3

Contents

3.1 Installation

3.1.1 Using ml4ir as a library

3.1.1.1 Requirements

- python3.{6,7} (tf2.0.3 is not available for python3.8)
- pip3

ml4ir can be installed as a pip package by using the following command

```
pip install ml4ir
```

This will install **ml4ir-0.1.3** (the current version) from PyPI.

To install optional dependencies like pygraphviz, use the following command:

```
pip3 install ml4ir[visualization]
```

To use pre-built pipelines that come with ml4ir, make sure to install it as follows (this installs pyspark and pygraphviz as well)

```
pip install ml4ir[all]
```

3.1.2 Using ml4ir as a toolkit or contributing to ml4ir

Firstly, clone ml4ir

```
git clone https://github.com/salesforce/ml4ir
```

You can use and develop on ml4ir either using docker or virtualenv

3.1.2.1 Docker (Recommended)

Requirements

- docker (18.09+ tested)
- docker-compose

We have set up a `docker-compose.yml` file for building and using docker containers to train models.

Change the working directory to the python package

```
cd path/to/ml4ir/python/
```

To build the docker image and run unit tests

```
docker-compose up --build
```

To only build the ml4ir docker image without running tests

```
docker-compose build
```

3.1.2.2 Virtual Environment

Requirements

- python3.{6,7} (tf2.0.3 is not available for python3.8)
- pip3

Change the working directory to the python package

```
cd path/to/ml4ir/python/
```

Install virtualenv

```
pip3 install virtualenv
```

Create new python3 virtual environment inside your git repo (it's .gitignored, don't worry)

```
python3 -m venv env/.ml4ir_venv3
```

Activate virtualenv

```
source env/.ml4ir_venv3/bin/activate
```

Install all dependencies

```
pip3 install --upgrade setuptools
pip3 install --upgrade pip
pip3 install -r requirements.txt
```

Set the PYTHONPATH environment variable to point to the python package

```
export PYTHONPATH=$PYTHONPATH:`pwd`
```

For more information in pygraphviz and its prerequisites, refer to pygraphviz documentation

3.1.2.3 Contributing to ml4ir

- Install python dependencies from the `build-requirements.txt` to setup the dependencies required for pre-commit hooks.
- `pre-commit-hooks` are required, and installed as a requirement for contributing to ml4ir. If an error results that they didn't install, execute `pre-commit install` to install git hooks in your .git/ directory.

3.1.3 Running Tests

To run all the python based tests under `ml4ir`

Using docker

```
docker-compose up
```

Using virtualenv

```
python3 -m pytest
```

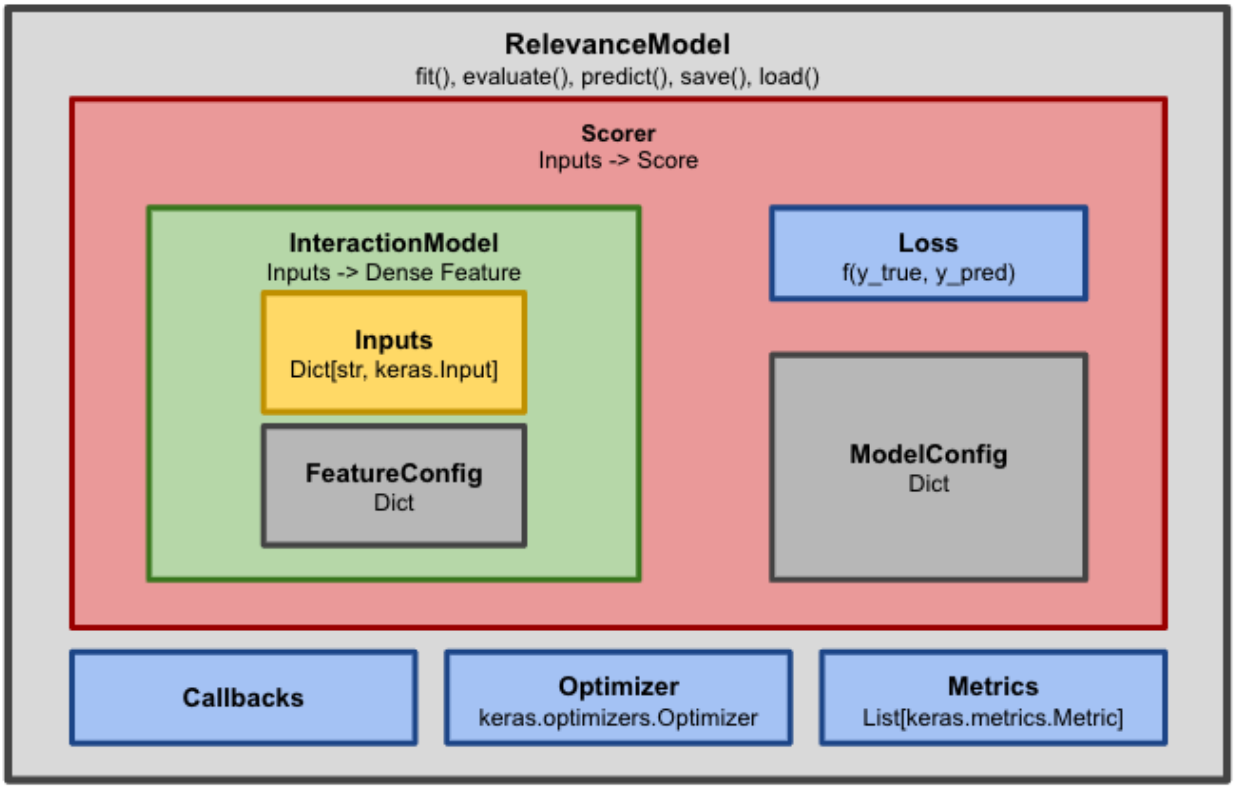
To run specific tests,

```
python3 -m pytest /path/to/test/module
```

3.2 Quickstart

3.2.1 ml4ir's Architecture

ml4ir is designed as a network of tightly coupled modular subcomponents. This lends itself to high customizability. In this section, we will briefly describe each of the subcomponents and how it all fits together.



FeatureConfig

The `FeatureConfig` is the main driver of ml4ir bridging the gap between the training and serving side. It is loaded from a YAML file and can be used to configure the list of features used by the model for training and serving. Additionally, it can be used to define preprocessing and feature transformation functions and their respective arguments. It can be extended to configure additional metadata for the features as needed.

More details about defining a FeatureConfig for your ml4ir model ***here***

Inputs

Keras Input placeholders constructed from the `FeatureConfig` that forms the first layer of the `RelevanceModel` and will be used by the model for learning a scoring function. Additionally, metadata features can also be made available in the Input layer that can be used to compute custom losses and metrics.

InteractionModel

The `InteractionModel` defines the feature transformation layer that converts the Input layer into numeric tensors that can be used to learn a scoring function. This layer can be used for a variety of transformations. Few examples are:

- converting categorical text labels into embedding vectors
- converting text into character embeddings and sequence encoding via a variety of layers like LSTM, GRU, transformers, etc.
- contextual embedding layers such as BERT, ELMO, GPT, etc.

Currently, ml4ir supports a univariate interaction model where a transformation function can be applied to a single feature. This can be extended to define custom interaction models that allow for cross feature interaction based transformations.

Loss

`Loss` is an implementation of the `RelevanceLossBase` that can be used to define the loss function and the corresponding final activation layer to be used to train a `RelevanceModel`. The loss function is defined on `y_true` and `y_pred`, the labels and predicted scores from the model, respectively. Metadata features can be used to define complex and custom loss functions to be used with `RelevanceModel`.

ModelConfig

`ModelConfig` is a YAML configuration file that defines the scoring function of the `RelevanceModel`. Specifically, it defines the logic to convert the transformed feature vectors into the model score. Currently, the ModelConfig only supports a DNN(multi layer perceptron like) architecture, but can be extended to handle sequential and convolution based scoring functions.

Scorer

`Scorer` defines the tensorflow layers of the model to convert the Input layer to the scores by combining and wrapping together the `ModelConfig`, `InteractionModel` and the `Loss`. Custom scorer objects can be defined and used with ml4ir as needed.

Callbacks

A list of keras Callbacks that can be used with the `RelevanceModel` for training and evaluation. ml4ir already comes packaged with commonly used callbacks for model checkpointing, early stopping and tensorboard. Additionally, ml4ir also defines debugging callbacks to log training and evaluation progress. Users have the flexibililty to use custom callback functions with ml4ir models as well.

Optimizer

Tensorflow's keras based optimizer object that is used for gradient optimization and learning the model weights. ml4ir also plays well with a wide variety of optimizers with custom learning rate schedules such as exponential decay, cyclic learning rate, etc.

Metrics

List of keras `Metric` classes that can be used to compute validation and test metrics for evaluating the model. Metadata features can be used to define custom and complex metrics to be used with `RelevanceModel`.

RelevanceModel

The `Scorer` is wrapped with the keras callbacks, optimizer and metrics to define a `RelevanceModel`. The `RelevanceModel` can be used like a Keras model with `fit()`, `predict()`, `evaluate()`, `save()` and `load()` which allow training, evaluation of models for search applications. Pretrained models and weights can be loaded for fine tuning or computing test metrics.

To learn more about constructing a `RelevanceModel` from the ground up check ***this guide***

3.2.2 Using ml4ir as a toolkit

3.2.2.1 Pipelines

ml4ir comes packaged with pre-defined configurable pipelines for popular search ML tasks. Currently, ml4ir supports the following tasks.

Learning to Rank

Learning to Rank(LTR) is the task of learning an ranking function that finds the most optimal ordering of a list of documents for a given query to improve relevance. Each document is represented in the dataset as a feature set computed for the query-document pair. The labels for this task can either be graded relevance values defined for the list of records in a query or a binary click/no-click label.

query_id	query_text	record_text	rank	popularity_score	page_views_score	domain_name	clicked
query_2	food processor	Hamilton Beach 12-Cup Stack & Snap Food Processor	1	0.898135	0.205381	Kitchen	1
query_2	food processor	Ninja BN601 Professional Plus Food Processor	2	1.147889	0.000000	Kitchen	0
query_5	Arthur C Clarke	Childhood's End	1	1.990899	0.042572	Science Fiction Books	1
query_5	Arthur C Clarke	Rendezvous with Rama	2	0.881701	0.042572	Science Fiction Books	0
query_5	Arthur C Clarke	2001: A Space Odyssey	3	-1.565636	0.041261	Science Fiction Books	0
query_5	Arthur C Clarke	2010: Odyssey Two	4	-0.921634	0.082535	Science Fiction Books	0
query_5	Arthur C Clarke	The Fountains of Paradise	5	-1.658981	0.046478	Science Fiction Books	0
query_5	Arthur C Clarke	Rama II	6	-0.702505	0.030636	Science Fiction Books	0
query_6	insect spray	Raid Flying Insect Killer	1	-0.384123	0.040924	Pest Control	1
query_6	insect spray	Ortho Home Dense Max Indoor Insect Barrier	2	0.531182	0.000000	Pest Control	0

In the sample ranking data above, each row represents a query-document pair of features. Features like `query_text`, `domain_name` are common across documents. Whereas features like `record_text`, `popularity_score`, `quality_score` are unique to each document. In this example, we learn a ranking function using binary clicks as the label. The state of the art LTR models of today rely on listwise losses and complex groupwise scoring functions.

To train and evaluate a learning to rank model, use the predefined `RankingPipeline`.

Query Classification

Query Classification is the task of classifying a given user query into a set of predefined categories. Additional features such as user context, domain of query can be used to personalize the predictions.

query_id	query_text	domain_name	previous_products	product_group
query_id_0	couch cushion	Furniture	[Couch, Bookmarks]	Cushion
query_id_1	wok	Kitchen	[Cooking Oil]	Frying pan
query_id_2	cuisinart knife	Kitchen	[Cutting Board]	Knives
query_id_3	duracell	Electronics	[Flashlight, Insect Spray]	Batteries
query_id_4	iphone x	Electronics	[Graphics Card, Mouse]	Mobile Phones

In the sample query classification data above, each row represents a user query. We try to predict the `product_group` category using the `query_text`, `domain_name` and `previous_products`. These fea-

tures define the user's context at the time of querying and also the actual query text made by the user. This type of query classification can be used to further narrow down search results and enhance the user search experience.

To train and evaluate a learning to rank model, use the predefined `ClassificationPipeline`.

Custom Pipeline

To define your own custom ml4ir pipeline, you can override the `RelevancePipeline**` to plug in the `RelevanceModel` you want to train and evaluate.

3.2.2.2 Command Line Arguments

Name	Type	Default	Description
–data_dir	<class 'str'>	None	Path to the data directory to be us
–data_format	<class 'str'>	tfrecord	Format of the data to be used. Sho
–tfrecord_type	<class 'str'>	example	TFRecord type of the data to be u
–feature_config	<class 'str'>	None	Path to YAML file or YAML strin
–model_file	<class 'str'>		Path to a pretrained model to load
–model_config	<class 'str'>	ml4ir/base/conf ig/default_mode l_config.yaml	Path to the Model config YAML u
–optimizer_key	<class 'str'>	adam	Optimizer to use. Has to be one o
–loss_key	<class 'str'>	None	Loss to optimize. Has to be one o
–metrics_keys	<class 'str'>	None	Metric to compute. Can be a list.
–monitor_metric	<class 'str'>	None	Metric name to use for monitoring
–monitor_mode	<class 'str'>	None	Metric mode to use for monitoring
–num_epochs	<class 'int'>	5	Max number of training epochs(o
–batch_size	<class 'int'>	128	Number of data samples to use pe
–learning_rate	<class 'float'>	0.01	Step size (e.g.: 0.01)
–learning_rate_decay	<class 'float'>	1.0	Decay rate for the learning rate.Cl
–learning_rate_decay_steps	<class 'int'>	10000000	Decay rate for the learning rate.Cl
–compute_intermediate_stats	<class 'bool'>	True	Whether to compute intermediate
–execution_mode	<class 'str'>	train_inference _evaluate	Execution mode for the pipeline.
–random_state	<class 'int'>	123	Initialize the seed to control rando
–run_id	<class 'str'>		Unique string identifier for the cu
–run_group	<class 'str'>	general	Unique string identifier to group r
–run_notes	<class 'str'>		Notes for the current training run.
–models_dir	<class 'str'>	models/	Path to save the model. Will be ex
–logs_dir	<class 'str'>	logs/	Path to save the training/inference
–checkpoint_model	<class 'bool'>	True	Whether to save model checkpoin
–train_pcent_split	<class 'float'>	0.8	Percentage of all data to be used f
–val_pcent_split	<class 'float'>	-1	Percentage of all data to be used f
–test_pcent_split	<class 'float'>	-1	Percentage of all data to be used f
–max_sequence_size	<class 'int'>	0	Maximum number of elements pe
–inference_signature	<class 'str'>	serving_default	SavedModel signature to be used
–use_part_files	<class 'bool'>	False	Whether to look for part files whil
–logging_frequency	<class 'int'>	25	How often to log results to log file
–group_metrics_min_queries	<class 'int'>	None	Minimum number of queries per g
–gradient_clip_value	<class 'float'>	5.0	Gradient clipping value/threshold
–compile_keras_model	<class 'bool'>	False	Whether to compile a loaded Save
–use_all_fields_at_inference	<class 'bool'>	False	Whether to require all fields in the
–pad_sequence_at_inference	<class 'bool'>	False	Whether to pad sequence at infere

Name	Type	Default	Description
–output_name	<class 'str'>	relevance_score	Name of the output node of the m
–early_stopping_patience	<class 'int'>	2	How many epochs to wait before
–file_handler	<class 'str'>	local	String specifying the file handler t
–initialize_layers_dict	<class 'str'>	{}	Dictionary of pretrained layers to
–freeze_layers_list	<class 'str'>	[]	List of layer names that are to be f

3.2.2.3 Usage Examples

Learning to Rank

Using TFRecord input data

```
python ml4ir/applications/ranking/pipeline.py \
--data_dir ml4ir/applications/ranking/tests/data/tfrecord \
--feature_config ml4ir/applications/ranking/tests/data/configs/feature_config.yaml \
--run_id test \
--data_format tfrecord \
--execution_mode train_inference_evaluate
```

Using CSV input data

```
python ml4ir/applications/ranking/pipeline.py \
--data_dir ml4ir/applications/ranking/tests/data/csv \
--feature_config ml4ir/applications/ranking/tests/data/configs/feature_config.yaml \
--run_id test \
--data_format csv \
--execution_mode train_inference_evaluate
```

Running in inference mode using the default serving signature

```
python ml4ir/applications/ranking/pipeline.py \
--data_dir ml4ir/applications/ranking/tests/data/tfrecord \
--feature_config ml4ir/applications/ranking/tests/data/configs/feature_config.yaml \
--run_id test \
--data_format tfrecord \
--model_file `pwd`/models/test/final/default \
--execution_mode inference_only
```

Training a simple 1 layer linear ranking model

```
python ml4ir/applications/ranking/pipeline.py \
--data_dir ml4ir/applications/ranking/tests/data/tfrecord \
--feature_config ml4ir/applications/ranking/tests/data/configs/linear_model/feature_
↪config.yaml \
--model_config ml4ir/applications/ranking/tests/data/configs/linear_model/model_
↪config.yaml \
--run_id test \
--data_format tfrecord \
--execution_mode train_inference_evaluate
```

Query Classification

Using TFRecord

```
python ml4ir/applications/classification/pipeline.py \
--data_dir ml4ir/applications/classification/tests/data/tfrecord \
--feature_config ml4ir/applications/classification/tests/data/configs/feature_config.
↪yaml \
--model_config ml4ir/applications/classification/tests/data/configs/model_config.yaml␣
↪\
--batch_size 32 \
--run_id test \
--data_format tfrecord \
--execution_mode train_inference_evaluate
```

Using CSV

```
python ml4ir/applications/classification/pipeline.py \
--data_dir ml4ir/applications/classification/tests/data/csv \
--feature_config ml4ir/applications/classification/tests/data/configs/feature_config.
↪yaml \
--model_config ml4ir/applications/classification/tests/data/configs/model_config.yaml␣
↪\
--batch_size 32 \
--run_id test \
--data_format csv \
--execution_mode train_inference_evaluate
```

Running in inference mode using the default serving signature

```
python ml4ir/applications/classification/pipeline.py \
--data_dir ml4ir/applications/classification/tests/data/tfrecord \
--feature_config ml4ir/applications/classification/tests/data/configs/feature_config.
↪yaml \
--model_config ml4ir/applications/classification/tests/data/configs/model_config.yaml␣
↪\
--batch_size 32 \
--run_id test \
--data_format tfrecord \
--model_file `pwd`/models/test/final/default \
--execution_mode inference_only
```

3.2.3 Using ml4ir as a library

Let's try to train a simple Learning-to-Rank model with some sample data...

Setup FileIO handler(and logger)

```
from ml4ir.base.io.local_io import LocalIO
from ml4ir.base.io.file_io import FileIO
import logging

file_io : FileIO = LocalIO()

# Set up logger
logger = logging.getLogger()
```

Load the FeatureConfig from a predefined YAML file

More information about defining a feature configuration YAML file *here*

```
from ml4ir.base.features.feature_config import FeatureConfig,␣
↪SequenceExampleFeatureConfig
from ml4ir.base.config.keys import *

feature_config: SequenceExampleFeatureConfig = FeatureConfig.get_instance(
    tfrecord_type=TFRecordTypeKey.SEQUENCE_EXAMPLE,
    feature_config_dict=file_io.read_yaml(FEATURE_CONFIG_PATH),
    logger=logger)
```

Create a RelevanceDataset

More information about the data loading pipeline ***here***

```
from ml4ir.base.data.relevance_dataset import RelevanceDataset

relevance_dataset = RelevanceDataset(data_dir=DATA_DIR,
                             data_format=DataFormatKey.CSV,
                             feature_config=feature_config,
                             tfrecord_type=TFRecordTypeKey.SEQUENCE_EXAMPLE,
                             max_sequence_size=MAX_SEQUENCE_SIZE,
                             batch_size=128,
                             preprocessing_keys_to_fns={},
                             file_io=file_io,
                             logger=logger)
```

Define an InteractionModel

```
from ml4ir.base.model.scoring.interaction_model import InteractionModel,␣
↪UnivariateInteractionModel

interaction_model: InteractionModel = UnivariateInteractionModel(
                                            feature_config=feature_config,
                                            tfrecord_type=TFRecordTypeKey.SEQUENCE_
↪EXAMPLE,
                                            max_sequence_size=MAX_SEQUENCE_SIZE,
                                            feature_layer_keys_to_fns={},
                                            file_io=file_io,
                                        )
```

Define losses, metrics and optimizer

Here, we are using predefined losses, metrics and optimizers. But each of these can be customized as needed.

```
from ml4ir.base.model.losses.loss_base import RelevanceLossBase
from ml4ir.applications.ranking.model.losses import loss_factory
from ml4ir.applications.ranking.model.metrics import metric_factory
from ml4ir.applications.ranking.config.keys import LossKey, MetricKey, ScoringTypeKey

# Define loss object from loss key
loss: RelevanceLossBase = loss_factory.get_loss(
                                loss_key=LossKey.RANK_ONE_LISTNET,
                                scoring_type=ScoringTypeKey.POINTWISE)

# Define metrics objects from metrics keys
metric_keys = [MetricKey.MRR, MetricKey.ACR]
metrics: List[Union[Type[Metric], str]] = [metric_factory.get_metric(metric_key=m)␣
↪for m in metric_keys]
```

(continues on next page)

(continued from previous page)

```
# Define optimizer
optimizer: Optimizer = get_optimizer(
                            optimizer_key=OptimizerKey.ADAM,
                            learning_rate=0.001
                        )
```

Define the Scorer object by wrapping the InteractionModel and the loss function

```
scorer: RelevanceScorer = RelevanceScorer.from_model_config_file(
    model_config_file=MODEL_CONFIG_PATH,
    interaction_model=interaction_model,
    loss=loss,
    logger=logger,
    file_io=file_io,
)
```

Combine it all to create a RankingModel

```
ranking_model: RelevanceModel = RankingModel(
                                    feature_config=feature_config,
                                    tfrecord_type=TFRecordTypeKey.SEQUENCE_EXAMPLE,
                                    scorer=scorer,
                                    metrics=metrics,
                                    optimizer=optimizer,
                                    file_io=file_io,
                                    logger=logger,
                                )
```

Training the RankingModel and monitor for MRR metric

```
ranking_model.fit(dataset=relevance_dataset,
                  num_epochs=3,
                  models_dir=MODELS_DIR,
                  logs_dir=LOGS_DIR,
                  monitor_metric="new_MRR",
                  monitor_mode="max")
```

Run inference on the RankingModel

```
ranking_model.predict(test_dataset=relevance_dataset.test).sample(10)
```

Finally, save the model

One can additionally pass preprocessing functions to be persisted as part of the SavedModel and into the tensorflow graph. For more information on how to do this, check ***here***

```
ranking_model.save(models_dir=MODELS_DIR,
                   preprocessing_keys_to_fns={},
                   required_fields_only=True)
```

For details on serving this model on the JVM check ***this guide***

3.2.4 Data Loading Pipeline

The ml4ir data loading pipeline is built on top of the tensorflow recommended data format called TFRecords. TFRecords is built using protocol buffers, which is a cross-language cross-platform serialization format for structured

data. This makes it the best data format for search based applications like ranking, classification, etc.

There are two types of TFRecord messages provided - `Example` and `SequenceExample`.

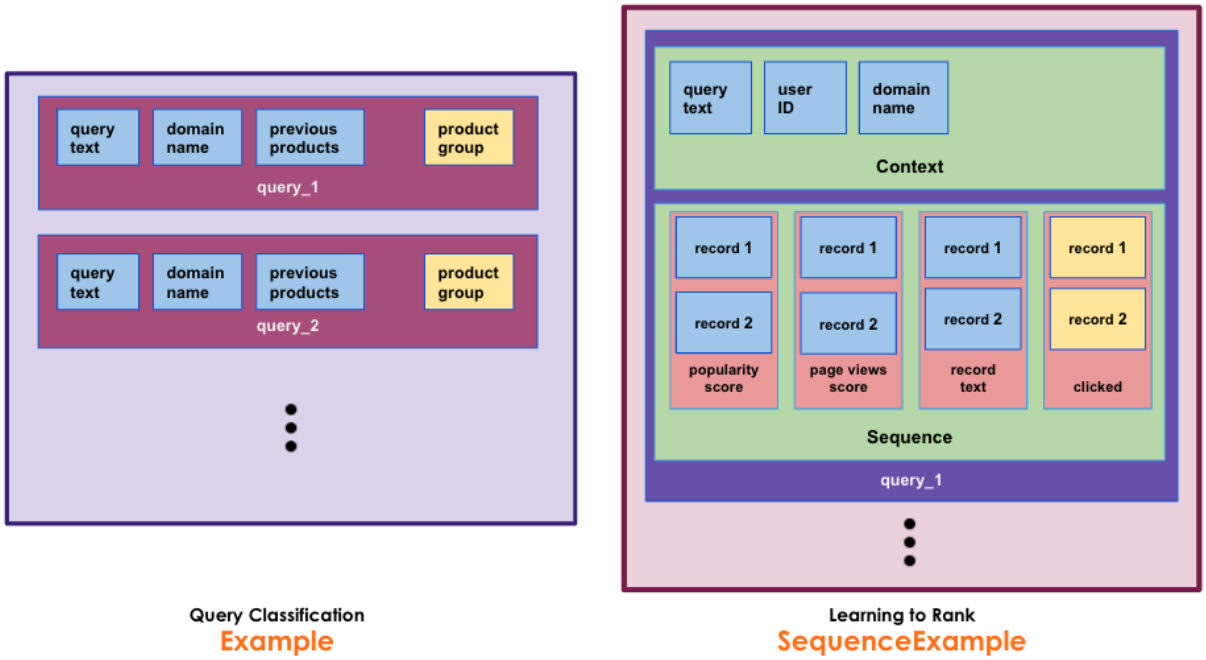
The first one that you see on the left is called Example. Here, we are using it to store sample query classification data. Each `Example` TFRecord message contains the features for a single query as key-value pairs. So the query text, domain name and previous products for a given query are stored in one single structure along with the product group, which is the classification label. We can build and store a `TFRecordDataset` as a collection of such Example messages.

The second type of protobuf message supported by TFRecords is called `SequenceExample`. `SequenceExample` contains two sub-types of features called as Context features and Sequence features (or feature lists). We use this to store data for models like Learning to Rank. We use context features to store features that are common across the query such as query text, domain name, user ID. Similar to `Example`, this is stored in one single sub-structure as key value pairs. Next we have the sequence features, which we use to store values for each feature as an ordered sequence corresponding to the documents. Here, we can see the features that are unique to each document such as popularity score, page views, record text. Finally, since the click label is also defined at a document level, we store that as a sequence feature as well.

Storing the Ranking data this way helps us achieve two things:

- Firstly, we now have a compact structured representation of data for ranking without redundant information as the query level features are stored only once per query as context features
- Secondly and more importantly, we now have a single object that contains all the query-document features for a given query. This means that we have all the information needed to learn complex ranking functions and define listwise losses for a given query without the need for any in-graph or preprocessing groupby operations.

This allows the storage to be efficient and training process to be fast.



The TFRecord data pipeline on ml4ir is configured out of the box for optimized data loading and preprocessing. The batches are lazy loaded and optimized by prefetching into memory for faster model training at scale. The serialized TFRecord messages are parsed and features are extracted based on the configuration specified in the `FeatureConfig`. ml4ir additionally applies preprocessing functions to the extracted features before feeding them as input into the model.

3.2.5 Defining the FeatureConfig

In this section, we describe how to define a feature configuration YAML file for your ml4ir application.

There are two types of feature configs that are supported in ml4ir - `ExampleFeatureConfig` and `SequenceExampleFeatureConfig` corresponding to the two types of TFRecord training and serving data format supported.

3.2.5.1 Main Keys

The feature config YAML file contains these main keys and their corresponding definitions:

- `query_key` : Feature used to uniquely identify each query (or data point)
- `label` : Feature to be used as the label
- `rank` : Feature to identify the position of the sequence record in a `SequenceExample` proto. It does not need to be specified if using `Example` data format.
- `features` : List of features that are used by the `RelevanceModel` for training and evaluation.

3.2.5.2 Feature Information

For each of the features in the FeatureConfig, we define a corresponding feature information definition. The main keys that should be specified for each feature are:

---

name | str

Name of the feature in the input dataset (CSV, TFRecord, libsvm, etc.)

---

node_name | str | default=name

Name of the feature in the tensorflow model. This will be the name of the feature in the input layer. Using the same input feature with multiple name nodes and feature transformations is supported. For example, using query text for character and word embeddings.

---

dtype | str

Tensorflow data type of the feature. Can be `string`, `int64` or `float`

---

trainable | bool | default=True

Value representing whether the feature is to be used for the scoring function. If set to False, the feature is considered a metadata feature that can be used to compute custom metrics and losses. Setting it to True, will make the transformed feature available for scoring by default.

---

tfrecord_type | str

Type of the SequenceExample feature type. Can be one of `sequence` for features unique to each sequence record or `context` for features common to all sequence records.

---

preprocessing_info | list of dicts | default=[]

List of preprocessing functions to be used on the feature. These functions will be applied in the data loading phase and will not be part of the tensorflow model. ml4ir provides an option to persist preprocessing logic as part of the SavedModel if the preprocessing functions are tensorflow compatible and serializable code.

For each preprocessing function, specify `fn`, the name of the function to be used, and `args`, a dictionary of values that are passed as arguments to the function. For example, to preprocess a text feature to remove punctuation and lower case, one can specify the preprocessing info as below

```
preprocessing_info:
  - fn: preprocess_text
    args:
      remove_punctuation: true
      to_lower: true
```

For more information on defining custom preprocessing functions and using it with ml4ir, check ***this guide***

---

feature_layer_info | dict

Definition of the feature transformation function to be applied to the feature in the model. Use this section to specify predefined or custom transformation functions to the model. Only tensorflow compatible functions can be used here as the transformation functions will be part of the `RelevanceModel` and serialized when the model is saved.

To define a feature transformation specify `fn`, the feature transformation function to be applied on the feature, and `args`, the key value pairs to be passed as arguments to the transformation function. For example, to use a text feature to learn character embeddings and produce a sequence encoding by using a bidirectional LSTM, define the feature layer as below

```
feature_layer_info:
  type: numeric
  fn: bytes_sequence_to_encoding_bilstm
  args:
    encoding_type: bilstm
    encoding_size: 128
    embedding_size: 128
    max_length: 20
```

For more information on defining custom feature transformation functions and using it with ml4ir, check ***this guide***

---

serving_info | dict

Definition of serving time feature attributes that will be used for model inference in production. Specifically, three key attributes can be specified in this section - `name`, `default_value` and `required`. `name` captures the name of the feature in production feature store that should be mapped to the model feature while constructing the input TFRecord proto. `default_value` captures the value to be used to fill the input feature tensor if the feature is absent in production. `required` is a boolean value representing if the feature is required at inference; the feature tensor will be set to default value otherwise.

---

log_at_inference | bool | default=False

Value representing if the feature should be logged when running `RelevanceModel.predict(...)`. Setting to True, returns the feature value when running inference. This can be used for error analysis on test examples and computing more complex metrics in a post processing job.

---

is_group_metric_key | bool | default=False

Value representing if the feature should be used for computing groupwise metrics when running `RelevanceModel.evaluate(...)`. The usage and implementation of the groupwise metrics is left to the user to be customized. The Ranking models come prepackaged with groupwise MRR and ACR metrics.

---

is_aux_label | bool | default=False

Value representing if the feature is used as an auxiliary label to compute failure metrics and auxiliary loss. The usage of the feature to compute the failure metrics is left to the user to be customized. The Ranking models come prepackaged with failure metrics computation that can be used, for example, to compute rate of clicks on documents without a match on the subject field.

In Ranking applications,

A secondary label is any feature/value that serves as a proxy relevance assessment that the user might be interested to measure on the dataset in addition to the primary click labels. For example, this could be used with an exact query match feature. In that case, the metric sheds light on scenarios where the records with an exact match are ranked lower than those without. This would provide the user with complimentary information (to typical click metrics such as MRR and ACR) about the model to help make better trade-off decisions w.r.t. best model selection.

The `FeatureConfig` can be extended to support additional attributes as necessary.

Example

This is an example configuration for the `query_text` feature, which will first be preprocessed to convert to lower case, remove punctuations, etc. Further we transform the feature with a sequence encoding using a bidirectional LSTM. At serving time, the feature `qtext` will be mapped from production feature store into the `query_text` feature for the model.

```
- name: query_text
  node_name: query_text
  trainable: true
  dtype: string
  log_at_inference: true
  feature_layer_info:
    fn: bytes_sequence_to_encoding_bilstm
    args:
      encoding_type: bilstm
      encoding_size: 128
      embedding_size: 128
      max_length: 20
  preprocessing_info:
    - fn: preprocess_text
      args:
        remove_punctuation: true
        to_lower: true
  serving_info:
    name: qtext
    required: true
    default_value: ""
```

3.2.6 Defining the ModelConfig

The `ModelConfig` is created from a YAML file and defines the scoring layers of the `RelevanceModel`. Specifically, the model config defines the layers to convert the transformed features output by the `InteractionModel` to the scores for the model.

Currently, ml4ir supports a dense neural network architecture (multi layer perceptron like) and a linear ranking model. Users can define the type of scoring architecture using the `architecture_key`. The layers of the neural network can be defined as a list of configurations using the `layers` attribute. For each layer, define the `type` of tensorflow-keras layer. Then for each layer, we can specify arguments to be passed to the instantiation of the layer. Finally, for each layer, we can specify a name using the `name` attribute.

Note: To train a simple linear ranking model, use the architecture_key as `linear` with a single `dense` layer.

This file is also used to define the optimizer, the learning rate schedule and calibration with temperature scaling. The current supported optimizers are: `adam`, `adagrad`, `nadam`, `sgd`, `rms_prop`. Each of these optimizers need so set the following hyper-parameter: `gradient_clip_value`. `adam` is the default optimizer if non was specified. The current supported learning rate schedules are: `exponential`, `cyclic`, `constant` and `reduce_lr_on_plateau`. `constant` is the default schedule if non was specified with learning rate = 0.01

The `exponential` learning rate schedule requires defining the following hyper-parameters: `initial_learning_rate`, `decay_steps`, `decay_rate`. For more information, see: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/ExponentialDecay

The `cyclic` learning rate schedule has three different type of policies: `triangular`, `triangular2`, `exponential`. All three types require defining the following hyper-parameters: `initial_learning_rate`, `maximal_learning_rate`, `step_size`. The `exponential` type requires and additional hyper-parameter: `gamma`. For more information, see: https://www.tensorflow.org/addons/api_docs/python/tfa/optimizers/CyclicalLearningRate and https://arxiv.org/pdf/1506.01186.pdf.

The `reduce_lr_on_plateau` reduces the learning rate by a `factor` (where `factor` < 1) when the monitor metric does not improve from one epoch to the next. Parameters that controls the scheduler: `factor`: factor by which the learning rate will be reduced `patience`: number of epochs with no improvement for the monitor metric after which learning rate will be reduced `min_lr`: The minimum value for allowed for the learning rate to reach. For more information, see: https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ReduceLROnPlateau

Calibration will be done as a separate process after possibly training or evaluating a (classification) model (currently, we do not support calibration for `RankingModel`). It implements [temperature scaling](https://github.com/gpleiss/temperature_scaling) technique to calibrate output probabilities of a classifier. It uses the `validation` set to train a `temperature` parameter, defined in the `ModelConfig` file. Then, it evaluates the calibrated model on the `test` set and stores the probability scores before and after applying calibration . After training TS, the calibrated model can be created using `relevance_model .add_temperature_layer(temp_value)` from the original `RelevanceModel` and be saved using `relevance_model.save()`. Note that for applying calibration to the Functional API model of a `RelevanceModel` it is expected that the model has an Activation layer (e.g. SoftMax) as the last layer.

Below you can see an example model config YAML using a DNN architecture to stack a bunch of dense layers with ReLU activation layers. Additionally, there are also a few dropout layers for regularization in between. A triangular2 cyclic learning rate schedule is used with adam optimizer.

```
architecture_key: dnn
layers:
  - type: dense
    name: first_dense
    units: 256
    activation: relu
  - type: dropout
```

(continues on next page)

(continued from previous page)

```
    name: first_dropout
    rate: 0.0
  - type: dense
    name: second_dense
    units: 64
    activation: relu
  - type: dropout
    name: second_dropout
    rate: 0.0
  - type: dense
    name: final_dense
    units: 1
    activation: null
optimizer:
  key: adam
  gradient_clip_value: 5.0
lr_schedule:
  key: cyclic
  type: triangular2
  initial_learning_rate: 0.001   #default value is 0.001
  maximal_learning_rate: 0.01    #default value is 0.01
  step_size: 10                  #default value is 10
calibration:
  key: temperature_scaling
  temperature: 1.5
```

Examples for defining other learning rate schedules in the ModelConfig YAML

Cyclic Learning Rate Schedule

```
lr_schedule:
  key: cyclic
  type: triangular
  initial_learning_rate: 0.001   #default value is 0.001
  maximal_learning_rate: 0.01    #default value is 0.01
  step_size: 10                  #default value is 10
```

Exponential Decay Learning Rate Schedule

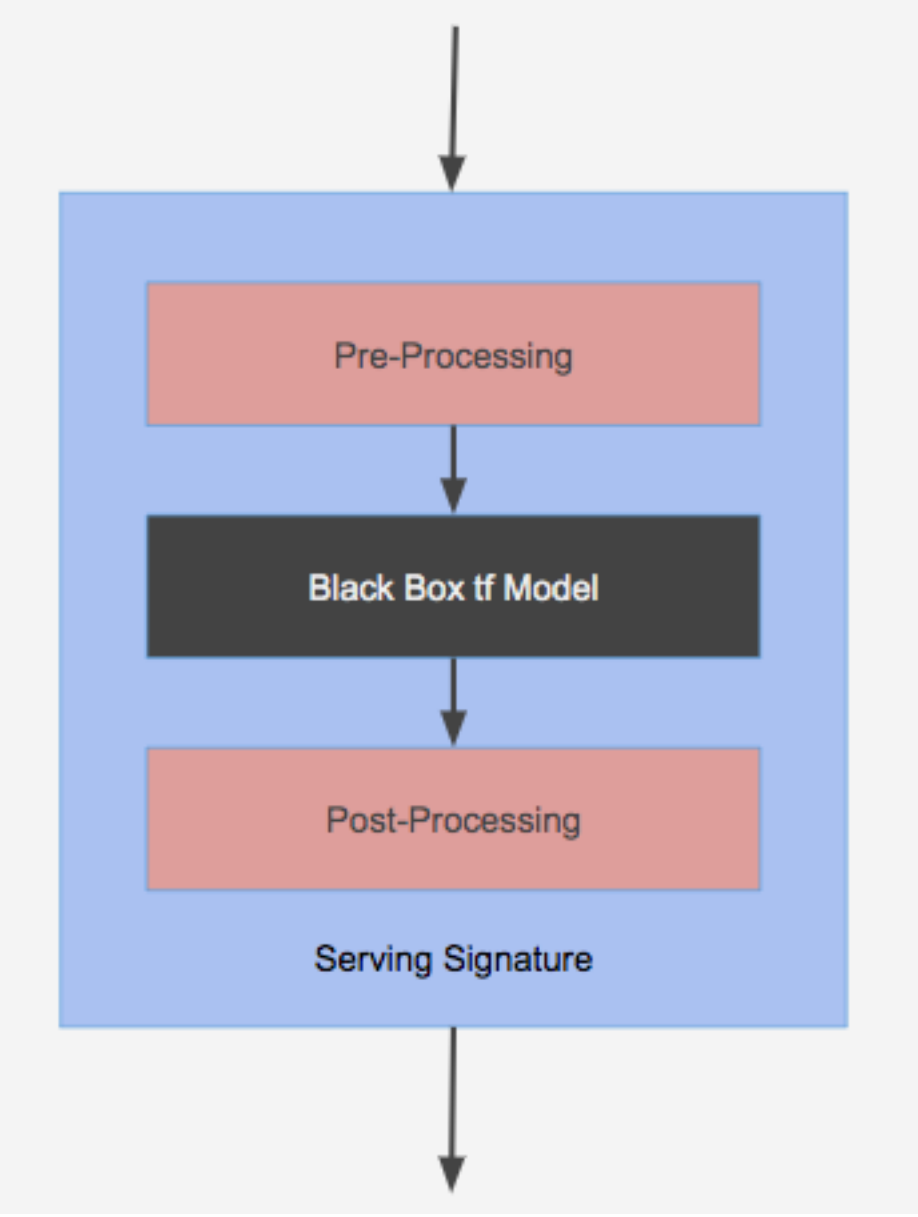
```
lr_schedule:
  key: exponential
  learning_rate: 0.01                  #default value is 0.01
  learning_rate_decay_steps: 100000   #default value is 100000
  learning_rate_decay: 0.96             #default value is 0.96
```

reduce_lr_on_plateau Learning Rate Schedule

```
lr_schedule:
  key: reduce_lr_on_plateau
  learning_rate: 1.0
  min_lr: 0.01
  patience: 1
  factor: 0.5
```

3.2.7 Saving ml4ir Models

ml4ir saves `RelevanceModel` in the SavedModel format. Doing so, allows us to add additional serving signatures to the persisted model. Serving signatures are pre and post processing wrappers for the blackbox tensorflow-keras model that is persisted along with the model. This allows us to write feature preprocessing logic at training time and be used at inference time. Additionally, these pre and post processing wrapper functions are persisted as tensorflow graph operations which allows for fast GPU executable serving time code.



Saving the models with serving signatures allows ml4ir models to be served directly on TFRecord protocol buffer messages. The model can be saved with a serving signature that accepts a TFRecord proto message as a string tensor which can then be parsed to extract features. The features can then be preprocessed and fed into the model to compute the scores. These scores can optionally be post processed before sending it back to the serving model call. For example, this can be used for converting ranking scores from each document into ranks or sort documents based on the scores.

To save a `RelevanceModel`, use

```
relevance_model.save(models_dir=MODELS_DIR,
                     preprocessing_keys_to_fns={},
                     required_fields_only=True)
```

This saves

- a SavedModel with default serving signature that accepts feature tensors as key value inputs and returns the scores
- a SavedModel with TFRecord serving signature that accepts a TFRecord proto and returns the scores
- individual layer weights that can be used for transfer learning with other ml4ir models

3.2.7.1 Saving preprocessing logic

Optionally, we can save preprocessing functions in the SavedModel as part of the serving signature as well. This requires that the preprocessing function is a `tf.function` that can be serialized as a tensorflow layer.

```
relevance_model.save(
    models_dir=MODEL_DIR,
    preprocessing_keys_to_fns=custom_preprocessing_fns,
    required_fields_only=True)
```

3.2.8 Serving ml4ir Models on the JVM

ml4ir provides Scala utilities for serving a saved model in a JVM based production environment. The utilities provide an easy way to use the `FeatureConfig` used at training time to map serving features from a production feature store or Solr into model features. These model features can then be packaged as a TFRecord protobuf message, which is then fed into the model. The utilities fetch the scores returned from the model which can then be used as necessary. For example, the scores can be used by the JVM code to

- convert ranking scores to ranks for each document per query
- sort documents based on ranking scores for each document
- convert classification scores to top label

and so on.

3.2.8.1 A high level usage of the Scala utilities

Load the FeatureConfig, saved model and create handlers to convert raw serving time features into TFRecord protos

```
val featureConfig = ModelFeaturesConfig.load(featureConfigPath)
val sequenceExampleBuilder = StringMapSequenceExampleBuilder.
↪withFeatureProcessors(featureConfig)
val rankingModelConfig = ModelExecutorConfig(inputTFNode, scoresTFNode)
val rankingModel = new SavedModelBundleExecutor(modelPath, rankingModelConfig)
```

Load serving time features from a CSV file. Replace this step with any other production feature store or Solr

```
val queryContextsAndDocs = StringMapCSVLoader.loadDataFromCSV(csvDataPath,␣
↪featureConfig)
```

Convert serving time features into a TFRecord proto message using the FeatureConfig (here, SequenceExample proto)

```
queryContextsAndDocs.map {
case q @ StringMapQueryContextAndDocs(queryContext, docs) =>
  val sequenceExample = sequenceExampleBuilder.build(queryContext, docs)
  (q, sequenceExample, rankingModel(sequenceExample))
}
```

Pass TFRecord protos to the loaded model and fetch ranking scores

```
val allScores: Iterable[
  (StringMapQueryContextAndDocs, SequenceExample, Array[Float])] =␣
↪runQueriesAgainstDocs(
    pathFor("test_data.csv"),
    pathFor("ranking_model_bundle"),
    pathFor("feature_config.yaml"),
    "serving_tfrecord_protos",
    "ranking_score"
  )
```

Sample returned scores for a query with six documents

```
0.14608994, 0.21464024, 0.1768626, 0.1312356, 0.19536583, 0.13580573
```

3.3 Advanced Guide

3.3.1 Using custom preprocessing functions

Preprocessing functions can be used with ml4ir in the data loading pipeline. Below we demonstrate how to define a custom preprocessing function and use it to load the data to train a `RelevanceModel`.

In this example, we define a preprocessing function to split a string into tokens and pad to max length.

```
@tf.function
def split_and_pad_string(feature_tensor, split_char=",", max_length=20):
    tokens = tf.strings.split(feature_tensor, sep=split_char).to_tensor()

    padded_tokens = tf.image.pad_to_bounding_box(
        tf.expand_dims(tokens[:, :max_length], axis=-1),
        offset_height=0,
        offset_width=0,
        target_height=1,
        target_width=max_length,
    )
    padded_tokens = tf.squeeze(padded_tokens, axis=-1)

    return padded_tokens
```

Define the preprocessing function in the FeatureConfig YAML:

```
- name: query_text
  node_name: query_text
  trainable: true
  dtype: string
  log_at_inference: true
  preprocessing_info:
```

(continues on next page)

(continued from previous page)

```
    - fn: split_and_pad_string
      args:
        split_char: " "
        max_length: 20
  serving_info:
    name: query_text
    required: true
```

Finally, use the custom split and pad prepreprocessing function to load a `RelevanceDataset` by passing custom functions as the `preprocessing_keys_to_fns` argument:

```
custom_preprocessing_fns = {
    "split_and_pad_string": split_and_pad_string
}

relevance_dataset = RelevanceDataset(
        data_dir=CSV_DATA_DIR,
        data_format=DataFormatKey.CSV,
        feature_config=feature_config,
        tfrecord_type=TFRecordTypeKey.EXAMPLE,
        batch_size=128,
        preprocessing_keys_to_fns=custom_preprocessing_fns,
        file_io=file_io,
        logger=logger
    )
```

Optionally, we can save preprocessing functions in the SavedModel as part of the serving signature as well. This requires that the preprocessing function is a `tf.function` that can be serialized as a tensorflow layer.

```
relevance_model.save(
    models_dir=MODEL_DIR,
    preprocessing_keys_to_fns=custom_preprocessing_fns,
    required_fields_only=True)
```

3.3.2 Using custom feature transformation functions

ml4ir allows users to define custom feature transformation functions. Here, we demonstrate how to define a function to convert text into character embeddings and then encode using a bidirectional **GRU**.

```
def bytes_sequence_to_encoding_bilstm(feature_tensor, feature_info, file_io: FileIO):
    args = feature_info["feature_layer_info"]["args"]

    # Decode string tensor to bytes
    feature_tensor = io.decode_raw(
        feature_tensor, out_type=tf.uint8, fixed_length=args.get("max_length", None),
    )

    feature_tensor = tf.squeeze(feature_tensor, axis=1)
    if "embedding_size" in args:
        char_embedding = layers.Embedding(
            name="{}_bytes_embedding".format(
                feature_info.get("node_name", feature_info.get("name"))
            ),
            input_dim=256,
            output_dim=args["embedding_size"],
```

(continues on next page)

(continued from previous page)

```
            mask_zero=True,
            input_length=args.get("max_length", None),
        )(feature_tensor)
    else:
        char_embedding = tf.one_hot(feature_tensor, depth=256)

    kernel_initializer = args.get("lstm_kernel_initializer", "glorot_uniform")
    encoding = get_bigru_encoding(
        embedding=char_embedding,
        lstm_units=int(args["encoding_size"] / 2),
        kernel_initializer=kernel_initializer,
    )
    return encoding


def get_bigru_encoding(embedding, lstm_units, kernel_initializer="glorot_uniform"):
    encoding = layers.Bidirectional(
        layers.GRU(
            units=lstm_units, return_sequences=False, kernel_initializer=kernel_
↪initializer
        ),
        merge_mode="concat",
    )(embedding)
    encoding = tf.expand_dims(encoding, axis=1)
    return encoding
```

Note: Any feature transformation function has to be a tensorflow compatible function as it is part of the tensorflow-keras `RelevanceModel`.

Define the feature transformation function to use with a text feature like query text:

```
- name: query_text
  node_name: query_text
  trainable: true
  dtype: string
  log_at_inference: true
  feature_layer_info:
    fn: bytes_sequence_to_encoding_bigru
    args:
      encoding_type: bilstm
      encoding_size: 128
      embedding_size: 128
      max_length: 20
  serving_info:
    name: query_text
    required: true
```

Finally, use the custom transformation functions with the `InteractionModel` and consecutively, create a `RelevanceModel`:

```
custom_feature_transform_fns = {
    "bytes_sequence_to_encoding_bigru": bytes_sequence_to_encoding_bigru,
}

interaction_model: InteractionModel = UnivariateInteractionModel(
                                            feature_config=feature_config,
                                            feature_layer_keys_to_fns=custom_feature_
↪transform_fns,
```

(continues on next page)

(continued from previous page)

```
                                                tfrecord_type=TFRecordTypeKey.EXAMPLE,
                                                file_io=file_io)
```

Once the `InteractionModel` has been wrapped with a `Scorer`, metrics, etc we can define a `RelevanceModel`. This model can be used for training, prediction and evaluation.

3.3.3 Predicting with a model trained on ml4ir

This sections explores how to get predictions from a model that is trained with `ml4ir`. For the sake of example, we assume that we have already trained a classification model. To train such a model, see this notebook.

The model artifacts are as follows in the `models-dir`:

```
├── checkpoint.tf
│   ├── assets
│   ├── saved_model.pb
│   └── variables
│       ├── variables.data-00000-of-00001
│       └── variables.index
└── final
    ├── default
    │   ├── assets
    │   ├── saved_model.pb
    │   └── variables
    │       ├── variables.data-00000-of-00001
    │       └── variables.index
    ├── layers
    │   ├── bidirectional.npz
    │   ├── bidirectional_1.npz
    │   ├── LAYERS as npz files
    │   ├── .
    │   ├── .
    │   └── vocab_lookup_3.npz
    └── tfrecord
        ├── assets
        ├── saved_model.pb
        └── variables
            ├── variables.data-00000-of-00001
            └── variables.index
```

The `final/default` signature is used when we hit the model with tensors. The `final/tfrecord` signature is used when we hit it with tfrecords.

3.3.3.1 Predicting with the tfrecords signature

The second case, which is easier when our data are already in tfrecords requires:

```
from tensorflow import data
import tensorflow as tf
from tensorflow.keras import models as kmodels

MODEL_DIR = "/PATH/TO/MODEL/"

model = kmodels.load_model(os.path.join(MODEL_DIR, 'final/tfrecord/'), compile=False)
infer_fn = model.signatures["serving_tfrecord"]
```

And now to construct a dataset and get predictions on it:

```
dataset = data.TFRecordDataset(glob.glob(os.path.join('/PATH/TO/DATASET', "part*")))
total_preds =  []
i = 0
# A prediction loop; to predict to one batch we can simply `infer_
↪fn(next(iter(dataset)))`
for batch in dataset.batch(1024):
    probs = infer_fn(protos=batch)
    total_preds.append(probs)
# Post processing of predictions
```

3.3.3.2 Predicting with the default signature

The default signature requires hitting the model with tensors. This, in turn, requires to do all the required preprocessing (look-ups, etc) to get these tensors. This is done with ml4ir. The code sceleton below describes the required steps.

```
# Define logger
# Define feat_config
# Define RelevanceDataset
# Defing RelevanceModel

relevance_model.predict(relevance_dataset.test)
```

This process, while much more verbose allows to do custom pre-processing on the model inputs, which can be different from the preprocessing done during training. For images, this can be artificial blurring. For text classification, using a subset of the text and many others.

Recall, pre-processing in ml4ir is controlled in the feature_config.yaml file. To do something extra during inference, we need to add it to the feature config, so that the pipeline is updated. For example, to use only the first few bytes of a text field called query that it is currently only preprocessed by lower-casing it, we need a function that achieves this and to pass the details in the config. So before, the features config could be:

```
    preprocessing_info:
      - fn: preprocess_text
        args:
          remove_punctuation: true
          to_lower: true
```

so that `preprocess_text` is the only preprocessing function. We can now do

```
    preprocessing_info:
      - fn: preprocess_text
        args:
          remove_punctuation: true
          to_lower: true
      - fn: trim_text
        args:
          keep_first: 3
```

and define trim text in the code. Assuming that:

```
@tf.function
def trim_text(inp, keep_first=3):
    """Keeps the first `keep_first` bytes of a tf.string"""
    return tf.strings.substr(inp, 0, keep_first, unit='BYTE')
```

then defining the RelevanceDataset as:

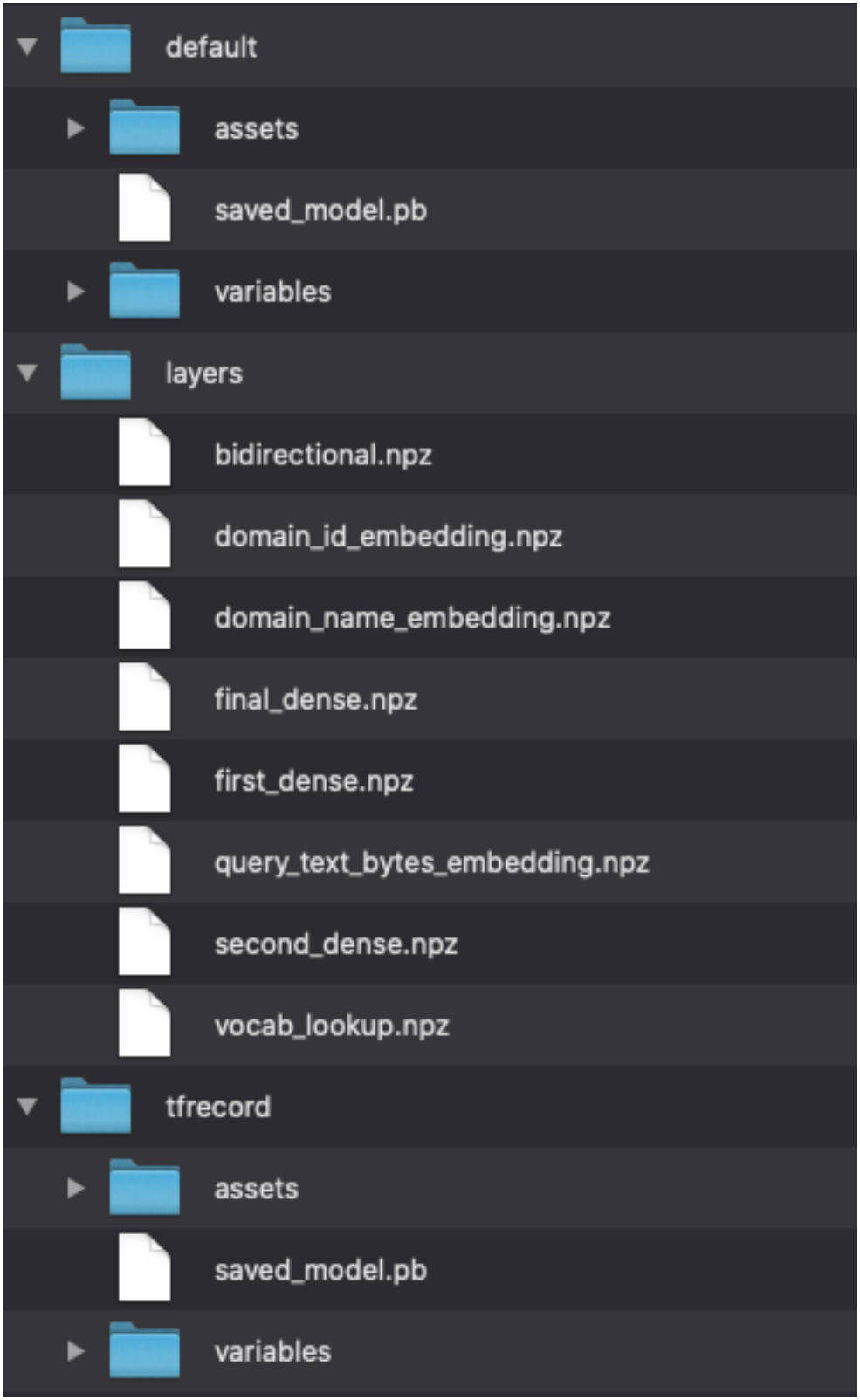
```
relevance_dataset = RelevanceDataset(
        data_dir="/tmp/dataset",
        data_format=DataFormatKey.TFRECORD,
        feature_config=feature_config,
        tfrecord_type=TFRecordTypeKey.EXAMPLE,
        batch_size=1024,
        preprocessing_keys_to_fns={'trim_text': trim_text},  # IMPORTANT!
        file_io=file_io, use_part_files=True,
        logger=logger
    )
```

will result in queries whose size is 3 bytes (as described in `trim_text`).

For more information on these, please refer to this notebook

3.3.4 Transfer Learning with ml4ir

ml4ir saves individual layer weights as part of the `RelevanceModel.save(...)` call. These layer weights can be used with other ml4ir models for transfer learning. This enables layers like embedding vectors to be shared across search tasks like ranking, classification, etc. with ease.



ml4ir provides support for loading pretrained layers and optionally freezing them. Depending on whether these layers/weights need to be fine tuned or used as is, one can freeze these layers or not.

To use pretrained embedding vectors from a `ClassificationModel` on ml4ir with a `RankingModel`:

```
initialize_layers_dict = {
    "query_text_bytes_embedding" : "models/activate_demo/bytes_embedding.npz"
}
freeze_layers_list = ["query_text_bytes_embedding"]
ranking_model: RelevanceModel = RankingModel(
                                    feature_config=feature_config,
                                    tfrecord_type=TFRecordTypeKey.SEQUENCE_EXAMPLE,
                                    scorer=scorer,
                                    metrics=metrics,
                                    optimizer=optimizer,
                                    initialize_layers_dict=initialize_layers_dict,
                                    freeze_layers_list=freeze_layers_list,
                                    file_io=file_io,
                                    logger=logger,
                                )
```

The model can be trained, evaluated and saved as usual after this step.

3.3.5 Running Kfold Cross Validation

ml4ir allows to run in a K-fold Cross validation mode. This mode reads the data the same way as the normal "non K-fold" mode and merges the data sets training, validation and test (if specified) together. Then according to the specified number of folds the merged data set is split among the training, validation and test sets.

You can control the K-fold mode by specifying three additional command line arguments.

1. kfold

The number of folds for K-fold Cross Validation. Must be > 2 if testset is included in folds and > 1 otherwise.

1. include_testset_in_kfold

Whether to merge the testset with the training and validation sets and perform kfold on the merged dataset.

1. monitor_metric

Metric to use for post Kfold CV analysis.

Example

```
--kfold 5
--kfold_analysis_metrics MRR
--include_testset_in_kfold False
```

This would split the dataset into 5 folds: f1, f2, f3, f4 and f5 then the following would be how Kfolds cross validation:
iteration 1: validation set= f1, training set=[f2,f3,f4,f5]

iteration 2: validation set= f2, training set=[f1,f3,f4,f5]

iteration 3: validation set= f3, training set=[f1,f2,f4,f5]

iteration 4: validation set= f4, training set=[f1,f2,f3,f5]

iteration 5: validation set= f5, training set=[f1,f2,f3,f4]

Example

```
--kfold 5
--kfold_analysis_metrics MRR
--include_testset_in_kfold True
```

This would split the dataset into 5 folds: f1, f2, f3, f4 and f5 then the following would be how Kfolds cross validation:
iteration 1: validation set= f1, test set = f2 , training set=[f3,f4,f5]

iteration 2: validation set= f2, test set = f3, training set=[f1,f4,f5]

iteration 3: validation set= f3, test set = f4, training set=[f1,f2,f5]

iteration 4: validation set= f4, test set = f5, training set=[f1,f2,f3]

iteration 4: validation set= f5, test set = f1, training set=[f2,f3,f4]

3.4 API Documentation

3.4.1 Pipelines

3.4.1.1 RelevancePipeline

class ml4ir.base.pipeline.**RelevancePipeline**(*args: argparse.Namespace*)
Bases: object

Base class that defines a pipeline to train, evaluate and save a RelevanceModel using ml4ir

Constructor to create a RelevancePipeline object to train, evaluate and save a model on ml4ir. This method sets up data, logs, models directories, file handlers used. The method also loads and sets up the FeatureConfig for the model training pipeline

Parameters **args** (*argparse Namespace*) – arguments to be used with the pipeline. Typically, passed from command line arguments

setup_logging() → logging.Logger
Set up the logging utilities for the training pipeline Additionally, removes pre existing job status files

set_seeds(*reset_graph=True*)
Set the random seeds for tensorflow and numpy in order to replicate results

Parameters **reset_graph** (*bool*) – Reset the tensorflow graph and clears the keras session

get_relevance_dataset(*preprocessing_keys_to_fns={}*) → ml4ir.base.data.relevance_dataset.RelevanceDataset
Create RelevanceDataset object by loading train, test data as tensorflow datasets

Parameters **preprocessing_keys_to_fns** (*dict of (str, function)*) – dictionary of function names mapped to function definitions that can now be used for preprocessing while loading the TFRecordDataset to create the RelevanceDataset object

Returns RelevanceDataset object that can be used for training and evaluating the model

Return type *RelevanceDataset* object

Notes

Override this method to create custom dataset objects

get_kfold_relevance_dataset(*num_folds*, *include_testset_in_kfold*, *read_data_sets*, *preprocessing_keys_to_fns={}*) → ml4ir.base.data.relevance_dataset.RelevanceDataset
Create RelevanceDataset object by loading train, test data as tensorflow datasets

Parameters

- **num_folds** (*int*) – number of folds in kfold

- **include_testset_in_kfold** (*bool*) – whether to include the testset in the folds
- **read_data_sets** (*bool*) – whether to call *create_dataset* which reads data from files.
- **preprocessing_keys_to_fns** (*dict of (str, function)*) – dictionary of function names mapped to function definitions that can now be used for preprocessing while loading the TFRecordDataset to create the RelevanceDataset object

Returns RelevanceDataset object that can be used for training and evaluating the model

Return type *KfoldRelevanceDataset* object

Notes

Override this method to create custom dataset objects

get_relevance_model_cls()
Fetch the class of the RelevanceModel to be used for the ml4ir pipeline

Returns

Return type RelevanceModel class

get_loss()
Get the primary loss function to be used with the RelevanceModel

Returns

Return type RelevanceLossBase object

get_aux_loss()
Get the auxiliary loss function to be used with the RelevanceModel

Returns

Return type RelevanceLossBase object

static get_metrics(*metrics_keys: List[str]*) → List[Union[keras.metrics.base_metric.Metric, str]]
Get the list of keras metrics to be used with the RelevanceModel

Parameters **metrics_keys** (*List of str*) – List of strings indicating the metrics to instantiate and retrieve

Returns

Return type list of keras Metric objects

get_relevance_model(*feature_layer_keys_to_fns={}*) → ml4ir.base.model.relevance_model.RelevanceModel
Creates a RankingModel that can be used for training and evaluating :param feature_layer_keys_to_fns: dictionary of function names mapped to tensorflow compatible

function definitions that can now be used in the InteractionModel as a feature function to transform input features

Returns RankingModel that can be used for training and evaluating a ranking model

Return type *RankingModel*

Notes

Override this method to create custom loss, scorer, model objects

create_pipeline_for_kfold(*args*)

run()

Run the pipeline to train, evaluate and save the model. It also runs the pipeline in kfold cross validation mode if specified.

Returns Experiment tracking dictionary with metrics and metadata for the run. Used for model selection and hyperparameter optimization

Return type dict

Notes

Also populates a experiment tracking dictionary containing the metadata, model architecture and metrics generated by the model

run_pipeline(*relevance_dataset=None*)

Run the pipeline to train, evaluate and save the model.

Parameters **relevance_dataset** (RelevanceDataset) – RelevanceDataset used for running the pipeline. If none, the relevance dataset will be created.

Returns Experiment tracking dictionary with metrics and metadata for the run. Used for model selection and hyperparameter optimization

Return type dict

Notes

Also populates a experiment tracking dictionary containing the metadata, model architecture and metrics generated by the model

pre_processing_step()

Performs arbitrary pre-processing steps such as copying or transforming data that the rest of the code can not accommodate. It serves as a placeholder without an explicit implementation (returns self) in the base pipeline. We expect that users can extend it in their custom pipelines.

post_training_step()

Performs arbitrary post-training steps such as copying or transforming data that the rest of the code can not accommodate. It serves as a placeholder without an explicit implementation (returns self) in the base pipeline. We expect that users can extend it in their custom pipelines.

finish(*job_status*, *job_info*)

Wrap up the model training pipeline. Performs the following actions

- save a job status file as _SUCCESS or _FAILURE to indicate job status.
- delete temp data and models directories
- if using spark IO, transfers models and logs directories to HDFS location from local directories
- log overall run time of ml4ir job

Parameters

- **job_status** (*str*) – Tuple with first element _SUCCESS or _FAILURE second element
- **job_info** (*str*) – for _SUCCESS, is experiment tracking metrics and metadata for _FAILURE, is stacktrace of failure

3.4.1.2 RankingPipeline

class ml4ir.applications.ranking.pipeline.**RankingPipeline**(*args: argparse.Namespace*)

Bases: *ml4ir.base.pipeline.RelevancePipeline*

Base class that defines a pipeline to train, evaluate and save a RankingModel using ml4ir

Constructor to create a RelevancePipeline object to train, evaluate and save a model on ml4ir. This method sets up data, logs, models directories, file handlers used. The method also loads and sets up the FeatureConfig for the model training pipeline :param args: arguments to be used with the pipeline.

Typically, passed from command line arguments

get_relevance_model_cls()
Fetch the class of the RelevanceModel to be used for the ml4ir pipeline :returns: :rtype: RelevanceModel class

get_loss()
Get the primary loss function to be used with the RelevanceModel

Returns

Return type RelevanceLossBase object

get_aux_loss()
Get the auxiliary loss function to be used with the RelevanceModel

Returns

Return type RelevanceLossBase object

static get_metrics(*metrics_keys: List[str]*) → List[Union[keras.metrics.base_metric.Metric, str]]
Get the list of keras metrics to be used with the RelevanceModel

Parameters **metrics_keys** (*List of str*) – List of strings indicating the metrics to instantiate and retrieve

Returns

Return type list of keras Metric objects

validate_args()
Validate the arguments to be used with RelevancePipeline

create_pipeline_for_kfold(*args*)
Create a RankingPipeline object used in running kfold cross validation.

kfold_analysis(*base_logs_dir*, *run_id*, *num_folds*, *pvalue_threshold=0.1*, *metrics=None*)
Aggregate results of the k-fold runs and perform t-test on the results between old(prod model) and new model's w.r.t the specified metrics. :param base_logs_dir: Total number of folds :type base_logs_dir: int :param run_id: current fold number :type run_id: int :param num_folds: Total number of folds

:type num_folds: int :param pvalue_threshold: the threshold used for pvalue to assess significance :type pvalue_threshold: float :param metrics: List of metrics to include in the kfold analysis :type metrics: list

run_kfold_analysis(*logs_dir*, *run_id*, *num_folds*, *metrics*)
Running the kfold analysis for ranking. Parameters: ———— logs_dir: str

path to logs directory

run_id: str string run_id

num_folds: int number of folds

metrics: list list of metrics to include in the kfold analysis

summary of the kfold analysis

3.4.1.3 ClassificationPipeline

class ml4ir.applications.classification.pipeline.**ClassificationPipeline**(*args: argparse.Namespace*)
Bases: *ml4ir.base.pipeline.RelevancePipeline*

Base class that defines a pipeline to train, evaluate and save a RelevanceModel for classification using ml4ir

Constructor to create a RelevancePipeline object to train, evaluate and save a model on ml4ir. This method sets up data, logs, models directories, file handlers used. The method also loads and sets up the FeatureConfig for the model training pipeline

Parameters **args** (*argparse Namespace*) – arguments to be used with the pipeline. Typically, passed from command line arguments

get_relevance_model_cls()
Fetch the class of the RelevanceModel to be used for the ml4ir pipeline

Returns

Return type RelevanceModel class

get_loss()
Get the primary loss function to be used with the RelevanceModel

Returns

Return type RelevanceLossBase object

static get_metrics(*metrics_keys: List[str]*) → List[Union[keras.metrics.base_metric.Metric, str]]
Get the list of keras metrics to be used with the RelevanceModel

Parameters **metrics_keys** (*List of str*) – List of strings indicating the metrics to instantiate and retrieve

Returns

Return type list of keras Metric objects

get_relevance_dataset(*parse_tfrecord=True*, *preprocessing_keys_to_fns={}*) → ml4ir.base.data.relevance_dataset.RelevanceDataset
Create RelevanceDataset object by loading train, test data as tensorflow datasets Defines a preprocessing feature function to one hot vectorize classification labels

Parameters **preprocessing_keys_to_fns** (*dict of (str, function)*) – dictionary of function names mapped to function definitions that can now be used for preprocessing while loading the TFRecordDataset to create the RelevanceDataset object

Returns RelevanceDataset object that can be used for training and evaluating the model

Return type *RelevanceDataset* object

Notes

Override this method to create custom dataset objects

get_kfold_relevance_dataset(*num_folds*, *include_testset_in_kfold*, *read_data_sets=False*, *parse_tfrecord=True*, *preprocessing_keys_to_fns={}*) → ml4ir.base.data.kfold_relevance_dataset.KfoldRelevanceDataset

Create KfoldRelevanceDataset object by loading train, test data as tensorflow datasets Defines a preprocessing feature function to one hot vectorize classification labels

Parameters

- **num_folds** (*int*) – Number of folds in kfold CV
- **include_testset_in_kfold** (*bool*) – Whether to include testset in the folds
- **read_data_sets** (*bool*) – Whether to read datasets from disk
- **preprocessing_keys_to_fns** (*dict of (str, function)*) – dictionary of function names mapped to function definitions that can now be used for preprocessing while loading the TFRecordDataset to create the KfoldRelevanceDataset object

Returns KfoldRelevanceDataset object that can be used for training and evaluating the model in a kfold cross validation mode.

Return type *KfoldRelevanceDataset* object

Notes

Override this method to create custom dataset objects

create_pipeline_for_kfold(*args*)

Create a ClassificationPipeline object used in running kfold cross validation.

run_kfold_analysis(*base_logs_dir*, *base_run_id*, *num_folds*, *metrics*)

3.4.2 Data Loaders and Helpers

3.4.2.1 RelevanceDataset

class ml4ir.base.data.relevance_dataset.**RelevanceDataset**(*data_dir: str*, *data_format: str*, *feature_config: ml4ir.base.features.feature_config.FeatureConfig*, *tfrecord_type: str*, *file_io: ml4ir.base.io.file_io.FileIO*, *max_sequence_size: int = 0*, *batch_size: int = 128*, *preprocessing_keys_to_fns: dict = {}*, *train_pcent_split: float = 0.8*, *val_pcent_split: float = -1*, *test_pcent_split: float = -1*, *use_part_files: bool = False*, *parse_tfrecord: bool = True*, *logger: Optional[logging.Logger] = None*, *keep_additional_info: int = 0*, *non_zero_features_only: int = 0*, *output_name: str = None*)

Bases: `object`

class to create/load TFRecordDataset for train, validation and test

Constructor method to instantiate a RelevanceDataset object Loads and creates the TFRecordDataset for train, validation and test splits

Parameters

- **data_dir** (*str*) – path to the directory containing train, validation and test data
- **data_format** (*{"tfrecord", "csv", "libsvm"}*) – type of data files to be converted into TFRecords and loaded as a TFRecordDataset
- **feature_config** (*FeatureConfig* object) – FeatureConfig object that defines the features to be loaded in the dataset and the preprocessing functions to be applied to each of them
- **tfrecord_type** (*{"example", "sequence_example"}*) – Type of the TFRecord protobuf message to be used for TFRecordDataset
- **file_io** (*FileIO* object) – file I/O handler objects for reading and writing data
- **max_sequence_size** (*int, optional*) – maximum number of sequence to be used with a single SequenceExample proto message The data will be appropriately padded or clipped to fit the max value specified
- **batch_size** (*int, optional*) – size of each data batch
- **preprocessing_keys_to_fns** (*dict of (str, function), optional*) – dictionary of function names mapped to function definitions that can now be used for preprocessing while loading the TFRecordDataset to create the RelevanceDataset object

- **train_pcent_split** (*float, optional*) – ratio of overall data to be used as training set
- **val_pcent_split** (*float, optional*) – ratio of overall data to be used as validation set
- **test_pcent_split** (*float, optional*) – ratio of overall data to be used as test set
- **use_part_files** (*bool, optional*) – load dataset from part files checked using "part-" prefix
- **parse_tfrecord** (*bool, optional*) – parse the TFRecord string from the dataset; returns strings as is otherwise
- **logger** (*Logger*, optional) – logging handler for status messages
- **output_name** (*str*) – The name of tensorflow's output node which carry the prediction score.

Notes

- Currently supports CSV, TFRecord and Libsvm data formats
- Does not support automatically splitting train, validation and test
- *data_dir* should contain *train*, *validation* and *test* directories with files within them

create_dataset(*parse_tfrecord=True*)
Loads and creates train, validation and test datasets

Parameters **parse_tfrecord** (*bool*) – parse the TFRecord string from the dataset; returns strings as is otherwise

balance_classes()
Balance class labels in the train dataset

train_val_test_split()
Split the dataset into train, validation and test

3.4.2.2 tfrecord_reader

class ml4ir.base.data.tfrecord_reader.**TFRecordParser**(*feature_config: ml4ir.base.features.feature_config.FeatureConfig*, *preprocessing_map: ml4ir.base.features.preprocessing.PreprocessingMap*, *required_fields_only: Optional[bool] = False*)

Bases: object

Base class for parsing TFRecord examples. This class consolidates the parsing and feature extraction pipeline for both Example and SequenceExample protobuf messages

Constructor method for instantiating a TFRecordParser object

Parameters

- **feature_config** (*FeatureConfig*) – FeatureConfig object defining context and sequence feature information
- **preprocessing_map** (*PreprocessingMap* object) – Object mapping preprocessing feature function names to their definitons

- **required_fields_only** (*bool, optional*) – Whether to only use required fields from the feature_config

get_features_spec()
Define the features spec from the feature_config. The features spec will be used to parse the serialized TFRecord

Returns feature specification dictionary that can be used to parse TFRecords

Return type dict

Notes

For SequenceExample messages, this method returns a pair of dictionaries, one each for context and sequence features.

extract_features_from_proto(*proto*)
Parse the serialized proto string to extract features

Parameters **proto** (*tf.Tensor*) – A scalar string tensor that is the serialized form of a TFRecord object

Returns Dictionary of features extracted from the proto as per the features_spec

Return type dict of Tensors

Notes

For SequenceExample proto messages, this function returns two dictionaries, one for context and another for sequence feature tensors. For Example proto messages, this function returns a single dictionary of feature tensors.

get_default_tensor(*feature_info*, *sequence_size=0*)
Get the default tensor for a given feature configuration

Parameters

- **feature_info** (*dict*) – Feature configuration information for the feature as specified in the feature_config
- **sequence_size** (*int, optional*) – Number of elements in the sequence of a SequenceExample

Returns Tensor object that can be used as a default tensor if the expected feature is missing from the TFRecord

Return type tf.Tensor

get_feature(*feature_info*, *extracted_features*, *sequence_size=0*)
Fetch the feature from the feature dictionary of extracted features

Parameters

- **feature_info** (*dict*) – Feature configuration information for the feature as specified in the feature_config
- **extracted_features** (*dict*) – Dictionary of feature tensors extracted by parsing the serialized TFRecord
- **sequence_size** (*int, optional*) – Number of elements in the sequence of a SequenceExample

Returns Feature tensor that is obtained from the extracted features for the given feature_info

Return type tf.Tensor

generate_and_add_mask(*extracted_features*, *features_dict*)
Create a mask to identify padded values

Parameters

- **extracted_features** (*dict*) – Dictionary of tensors extracted from the serialized TFRecord
- **features_dict** (*dict*) – Dictionary of tensors that will be used for model training/serving as inputs to the model

Returns

- **features_dict** (*dict*) – Dictionary of tensors that will be used for model training/serving updated with the mask tensor if applicable
- **sequence_size** (*int*) – Number of elements in the sequence of the TFRecord

pad_feature(*feature_tensor*, *feature_info*)
Pad the feature to the *max_sequence_size* in order to create uniform data batches for training :param feature_tensor: Feature tensor to be padded :type feature_tensor: tf.Tensor :param feature_info: Feature configuration information for the feature as specified in the feature_config :type feature_info: dict

Returns Feature tensor padded to the *max_sequence_size*

Return type tf.Tensor

preprocess_feature(*feature_tensor*, *feature_info*)
Preprocess feature based on the feature configuration

Parameters

- **feature_tensor** (*tf.Tensor*) – input feature tensor to be preprocessed
- **feature_info** (*dict*) – Feature configuration information for the feature as specified in the feature_config

Returns preprocessed tensor object

Return type *tf.Tensor*

Notes

Only preprocessing functions part of the *preprocessing_map* can be used in this function for preprocessing at data loading

Pass custom preprocessing functions while instantiating the RelevanceDataset object with *preprocessing_keys_to_fns* argument

get_parse_fn() → tensorflow.python.eager.def_function.function
Define a parsing function that will be used to load the TFRecordDataset and create input features for the model.

Returns Parsing function that takes in a serialized TFRecord protobuf message and extracts a dictionary of feature tensors

Return type *tf.function*

Notes

This function will also be used with the TFRecord serving signature in the saved model.

class ml4ir.base.data.tfrecord_reader.**TFRecordExampleParser**(*feature_config: ml4ir.base.features.feature_config.FeatureCo preprocessing_map: ml4ir.base.features.preprocessing.Preprocess required_fields_only: Optional[bool] = False*)

Bases: *ml4ir.base.data.tfrecord_reader.TFRecordParser*

Class for parsing Example TFRecord protobuf messages

Constructor method for instantiating a TFRecordParser object

Parameters

- **feature_config** (*FeatureConfig*) – FeatureConfig object defining context and sequence feature information
- **preprocessing_map** (*PreprocessingMap* object) – Object mapping preprocessing feature function names to their definitons
- **required_fields_only** (*bool, optional*) – Whether to only use required fields from the feature_config

get_features_spec()

Define the features spec from the feature_config. This will be used to parse the serialized TFRecord

Returns feature specification dictionary that can be used to parse TFRecords

Return type dict

extract_features_from_proto(*serialized*)

Parse the serialized proto string to extract features

Parameters **proto** (*tf.Tensor*) – A scalar string tensor that is the serialized form of a TFRecord object

Returns Dictionary of features extracted from the proto as per the features_spec

Return type dict of Tensors

get_default_tensor(*feature_info*, *sequence_size=0*)

Get the default tensor for a given feature configuration

Parameters

- **feature_info** (*dict*) – Feature configuration information for the feature as specified in the feature_config
- **sequence_size** (*int, optional*) – Number of elements in the sequence of a SequenceExample

Returns Tensor object that can be used as a default tensor if the expected feature is missing from the TFRecord

Return type tf.Tensor

get_feature(*feature_info*, *extracted_features*, *sequence_size=0*)

Fetch the feature from the feature dictionary of extracted features

Parameters

- **feature_info** (*dict*) – Feature configuration information for the feature as specified in the feature_config
- **extracted_features** (*dict*) – Dictionary of feature tensors extracted by parsing the serialized TFRecord
- **sequence_size** (*int, optional*) – Number of elements in the sequence of a SequenceExample

Returns Feature tensor that is obtained from the extracted features for the given feature_info

Return type tf.Tensor

generate_and_add_mask(*extracted_features*, *features_dict*)

Create a mask to identify padded values

Parameters

- **extracted_features** (*dict*) – Dictionary of tensors extracted from the serialized TFRecord
- **features_dict** (*dict*) – Dictionary of tensors that will be used for model training/serving as inputs to the model

Returns

- **features_dict** (*dict*) – Dictionary of tensors that will be used for model training/serving updated with the mask tensor if applicable
- **sequence_size** (*int*) – Number of elements in the sequence of the TFRecord

pad_feature(*feature_tensor*, *feature_info*)

Pad the feature to the *max_sequence_size* in order to create uniform data batches for training :param feature_tensor: Feature tensor to be padded :type feature_tensor: tf.Tensor :param feature_info: Feature configuration information for the feature as specified in the feature_config :type feature_info: dict

Returns Feature tensor padded to the *max_sequence_size*

Return type tf.Tensor

class ml4ir.base.data.tfrecord_reader.**TFRecordSequenceExampleParser**(*feature_config: ml4ir.base.features.feature_config.FeatureConfig*, *preprocessing_map: ml4ir.base.features.preprocessing.PreprocessingMap = <ml4ir.base.features.preprocessing.PreprocessingMap object>*, *required_fields_only: Optional[bool] = False*, *pad_sequence: Optional[bool] = True*, *max_sequence_size: Optional[int] = 25*, *output_name: Optional[str] = None*)

Bases: *ml4ir.base.data.tfrecord_reader.TFRecordParser*

Constructor method for instantiating a TFRecordParser object

Parameters

- **feature_config** (*FeatureConfig*) – FeatureConfig object defining context and sequence feature information
- **preprocessing_map** (*PreprocessingMap* object) – Object mapping preprocessing feature function names to their definitons
- **required_fields_only** (*bool, optional*) – Whether to only use required fields from the feature_config
- **pad_sequence** (*bool, optional*) – Whether to pad sequence
- **max_sequence_size** (*int, optional*) – Maximum number of sequence per query. Used for padding
- **output_name** (*str*) – The name of tensorflow's output node which carry the prediction score

get_features_spec()

Define the features spec from the feature_config. This will be used to parse the serialized TFRecord

Returns

- *dict* – Feature specification dictionary that can be used to parse Context features from the serialized SequenceExample
- *dict* – Feature specification dictionary that can be used to parse Sequence features (or feature lists) from the serialized SequenceExample

extract_features_from_proto(*serialized*)

Parse the serialized proto string to extract features

Parameters **proto** (*tf.Tensor*) – A scalar string tensor that is the serialized form of a TFRecord object

Returns

- *dict of Tensors* – Dictionary of context feature tensors extracted from the proto as per the *features_spec*
- *dict of Tensors* – Dictionary of sequence feature tensors extracted from the proto as per the *features_spec*

get_default_tensor(*feature_info*, *sequence_size*)

Get the default tensor for a given feature configuration :param feature_info: Feature configuration information for the feature as specified in the feature_config :type feature_info: dict :param sequence_size: Number of elements in the sequence of a SequenceExample :type sequence_size: int, optional

Returns Tensor object that can be used as a default tensor if the expected feature is missing from the TFRecord

Return type tf.Tensor

get_feature(*feature_info*, *extracted_features*, *sequence_size*)

Fetch the feature from the feature dictionary of extracted features :param feature_info: Feature configuration information for the feature as specified in the feature_config :type feature_info: dict :param extracted_features: Dictionary of feature tensors extracted by parsing the serialized TFRecord :type extracted_features: dict :param sequence_size: Number of elements in the sequence of a SequenceExample :type sequence_size: int, optional

Returns Feature tensor that is obtained from the extracted features for the given feature_info

Return type tf.Tensor

generate_and_add_mask(*extracted_features*, *features_dict*)

Create a mask to identify padded values

Parameters

- **extracted_features** (*dict*) – Dictionary of tensors extracted from the serialized TFRecord
- **features_dict** (*dict*) – Dictionary of tensors that will be used for model training/serving as inputs to the model

Returns

- **features_dict** (*dict*) – Dictionary of tensors that will be used for model training/serving updated with the mask tensor if applicable
- **sequence_size** (*int*) – Number of elements in the sequence of the TFRecord

pad_feature(*feature_tensor*, *feature_info*)

Pad the feature to the *max_sequence_size* in order to create uniform data batches for training :param feature_tensor: Feature tensor to be padded :type feature_tensor: tf.Tensor :param feature_info: Feature configuration information for the feature as specified in the feature_config :type feature_info: dict

Returns Feature tensor padded to the *max_sequence_size*

Return type tf.Tensor

ml4ir.base.data.tfrecord_reader.**get_parse_fn**(*tfrecord_type: str*, *feature_config: ml4ir.base.features.feature_config.FeatureConfig*, *preprocessing_keys_to_fns: dict*, *max_sequence_size: int = 0*, *required_fields_only: bool = False*, *pad_sequence: bool = True*, *output_name: str = None*) → tensorflow.python.eager.def_function.function

Create a parsing function to extract features from serialized TFRecord data using the definition from the FeatureConfig

Parameters

- **tfrecord_type** (*{"example", "sequence_example"}*) – Type of TFRecord data to be loaded into a dataset
- **feature_config** (*FeatureConfig* object) – FeatureConfig object defining the features to be extracted
- **preprocessing_keys_to_fns** (*dict of(str, function), optional*) – dictionary of function names mapped to function definitions that can now be used for preprocessing while loading the TFRecordDataset to create the RelevanceDataset object
- **max_sequence_size** (*int*) – Maximum number of sequence per query. Used for padding
- **required_fields_only** (*bool, optional*) – Whether to only use required fields from the feature_config
- **pad_sequence** (*bool*) – Whether to pad sequence
- **output_name** (*str*) – The name of tensorflow's output node which carry the prediction score

Returns Parsing function that takes in a serialized SequenceExample or Example message and extracts a dictionary of feature tensors

Return type *tf.function*

ml4ir.base.data.tfrecord_reader.**read**(*data_dir: str*, *feature_config: ml4ir.base.features.feature_config.FeatureConfig*, *tfrecord_type: str*, *file_io: ml4ir.base.io.file_io.FileIO*, *max_sequence_size: int = 0*, *batch_size: int = 0*, *preprocessing_keys_to_fns: dict = {}*, *parse_tfrecord: bool = True*, *use_part_files: bool = False*, *logger: logging.Logger = None*, ***kwargs*) → tensorflow.python.data.ops.readers.TFRecordDatasetV2

Extract features by reading and parsing TFRecord data and converting into a TFRecordDataset using the FeatureConfig

Parameters

- **data_dir** (*str*) – path to the directory containing train, validation and test data
- **feature_config** (*FeatureConfig* object) – FeatureConfig object that defines the features to be loaded in the dataset and the preprocessing functions to be applied to each of them
- **tfrecord_type** (*{"example", "sequence_example"}*) – Type of the TFRecord protobuf message to be used for TFRecordDataset
- **file_io** (*FileIO* object) – file I/O handler objects for reading and writing data

- **max_sequence_size** (*int, optional*) – maximum number of sequence to be used with a single SequenceExample proto message The data will be appropriately padded or clipped to fit the max value specified
- **batch_size** (*int, optional*) – size of each data batch
- **preprocessing_keys_to_fns** (*dict of(str, function), optional*) – dictionary of function names mapped to function definitions that can now be used for preprocessing while loading the TFRecordDataset to create the RelevanceDataset object
- **use_part_files** (*bool, optional*) – load dataset from part files checked using "part-" prefix
- **parse_tfrecord** (*bool, optional*) – parse the TFRecord string from the dataset; returns strings as is otherwise
- **logger** (*Logger*, optional) – logging handler for status messages

Returns TFRecordDataset loaded from the *data_dir* specified using the FeatureConfig

Return type *TFRecordDataset*

3.4.2.3 csv_reader

ml4ir.base.data.csv_reader.**read**(*data_dir: str*, *feature_config: ml4ir.base.features.feature_config.FeatureConfig*, *tfrecord_type: str*, *tfrecord_dir: str*, *file_io: ml4ir.base.io.file_io.FileIO*, *batch_size: int = 128*, *preprocessing_keys_to_fns: dict = {}*, *use_part_files: bool = False*, *max_sequence_size: int = 25*, *parse_tfrecord: bool = True*, *logger=None*, ***kwargs*) → tensorflow.python.data.ops.readers.TFRecordDatasetV2

Create a TFRecordDataset from directory of CSV files using the FeatureConfig

Current execution plan:

1. Load CSVs as pandas dataframes
2. Convert each query into tf.train.SequenceExample protobufs
3. Write the protobufs into a .tfrecord file
4. Load .tfrecord file into a TFRecordDataset and parse the protobufs

Parameters

- **data_dir** (*str*) – Path to directory containing csv files to read
- **feature_config** (*FeatureConfig object*) – FeatureConfig object that defines the features to be loaded in the dataset and the preprocessing functions to be applied to each of them
- **tfrecord_dir** (*str*) – Path to directory where the serialized .tfrecord files will be stored
- **batch_size** (*int*) – value specifying the size of the data batch
- **use_part_files** (*bool*) – load dataset from part files checked using "part-" prefix
- **max_sequence_size** (*int*) – value specifying max number of records per query
- **logger** (*Logger object*) – logging handler to print and save status messages

Returns tensorflow TFRecordDataset loaded from the CSV file

Return type *TFRecordDataset* object

3.4.2.4 tfrecord_writer

Writes data in Example or SequenceExample protobuf (tfrecords) format.

To use it as a standalone script, refer to the argument spec at the bottom

Notes

Setting `--keep-single-files` writes one tfrecord file for each CSV file (better performance). If not set, joins everything to a single tfrecord file.

Examples

Syntax to convert a single or several CSVs:

```
>>> python ml4ir/base/data/tfrecord_writer.py \
... sequence_example|example \
... --csv-files <SPACE_SEPARATED_PATHS_TO_CSV_FILES> \
... --out-dir <PATH_TO_OUTPUT_DIR> \
... --feature_config <PATH_TO_YAML_FEATURE_CONFIG> \
... --keep-single-files
```

or to convert all CSV files in a dir

```
>>> python ml4ir/base/data/tfrecord_writer.py \
... sequence_example|example \
... --csv-dir <DIR_WITH_CSVs> \
... --out-dir <PATH_TO_OUTPUT_DIR> \
... --feature_config <PATH_TO_YAML_FEATURE_CONFIG> \
... --keep-single-files
```

Usage example:

```
>>> python ml4ir/base/data/tfrecord_writer.py \
... sequence_example \
... --csv-files /tmp/d.csv /tmp/d2.csv \
... --out-dir /tmp \
... --feature-config /tmp/fconfig.yaml \
... --keep-single-files
```

ml4ir.base.data.tfrecord_writer.**write_from_files**(*csv_files: List[str], tfrecord_file: str, feature_config: ml4ir.base.features.feature_config.FeatureConfig, tfrecord_type: str, file_io: ml4ir.base.io.file_io.FileIO, logger: logging.Logger = None*)

Converts data from CSV files into tfrecord files

Parameters

- **csv_files** (*list of str*) – list of csv file paths to read data from
- **tfrecord_file** (*str*) – tfrecord file path to write the output

- **feature_config** (*FeatureConfig*) – FeatureConfig object that defines the features to be loaded in the dataset and the preprocessing functions to be applied to each of them
- **tfrecord_type** (*{"example", "sequence_example"}*) – Type of the TFRecord protobuf message to be used for TFRecordDataset
- **file_io** (*FileIO object*) – FileIO handler object for reading and writing files
- **logger** (*Logger*, optional) – logging handler for status messages

ml4ir.base.data.tfrecord_writer.**write_from_df**(*df: pandas.core.frame.DataFrame*, *tfrecord_file: str*, *feature_config: ml4ir.base.features.feature_config.FeatureConfig*, *tfrecord_type: str*, *logger: logging.Logger = None*)

Converts data from CSV files into tfrecord files

Parameters df : *pd.DataFrame*

pandas DataFrame to be converted to TFRecordDataset

tfrecord_file [str] tfrecord file path to write the output

feature_config [*FeatureConfig*] FeatureConfig object that defines the features to be loaded in the dataset and the preprocessing functions to be applied to each of them

tfrecord_type [{"example", "sequence_example"}] Type of the TFRecord protobuf message to be used for TFRecordDataset

logger [*Logger*, optional] logging handler for status messages

3.4.3 Relevance Models

3.4.3.1 RelevanceModel

class ml4ir.base.model.relevance_model.**RelevanceModel**(*feature_config: ml4ir.base.features.feature_config.FeatureConfig*, *tfrecord_type: str*, *file_io: ml4ir.base.io.file_io.FileIO*, *scorer: Optional[ml4ir.base.model.scoring.scoring_model.Releva = None*, *metrics: List[Union[keras.metrics.base_metric.Metric*, *str]] = []*, *optimizer: Optional[keras.optimizers.optimizer_v2.optimizer_v2.Opt = None*, *model_file: Optional[str] = None*, *initialize_layers_dict: dict = {}*, *freeze_layers_list: list = []*, *compile_keras_model: bool = False*, *output_name: str = 'score'*, *logger=None*, *eval_config: dict = {}*)

Bases: object

Constructor to instantiate a RelevanceModel that can be used for training and evaluating the search ML task

Parameters

- **feature_config** (*FeatureConfig* object) – FeatureConfig object that defines the features to be loaded in the dataset and the preprocessing functions to be applied to each of them
- **tfrecord_type** (*{"example", "sequence_example"}*) – Type of the TFRecord protobuf message used for TFRecordDataset
- **file_io** (*FileIO* object) – file I/O handler objects for reading and writing data
- **scorer** (*RelevanceScorer* object) – Scorer object that wraps an InteractionModel and converts input features into scores
- **metrics** (*list*) – List of keras Metric objects/strings that will be used for evaluating the trained model
- **optimizer** (*Optimizer*) – Tensorflow keras optimizer to be used for training the model
- **model_file** (*str, optional*) – Path to pretrained model file to be loaded for evaluation or retraining
- **initialize_layers_dict** (*dict, optional*) – Dictionary of tensorflow layer names mapped to the path of pretrained weights Use this for transfer learning with pretrained weights
- **freeze_layers_list** (*list, optional*) – List of model layer names to be frozen Use this for freezing pretrained weights from other ml4ir models
- **compile_keras_model** (*bool, optional*) – Whether the keras model loaded from disk should be compiled with loss, metrics and an optimizer
- **output_name** (*str, optional*) – Name of the output tensorflow node that captures the score
- **logger** (*Logger*, optional) – logging handler for status messages
- **eval_config** (*dict*) – A dictionary of Evaluation config parameters

is_compiled = None

Specify inputs to the model

Individual input nodes are defined for each feature Each data point represents features for all records in a single query

classmethod from_relevance_scorer(*feature_config: ml4ir.base.features.feature_config.FeatureConfig, interaction_model: ml4ir.base.model.scoring.interaction_model.InteractionM model_config: dict, loss: ml4ir.base.model.losses.loss_base.RelevanceLossBase, metrics: List[Union[keras.metrics.base_metric.Metric, str]], optimizer: keras.optimizers.optimizer_v2.optimizer_v2.OptimizerV2, tfrecord_type: str, file_io: ml4ir.base.io.file_io.FileIO, model_file: Optional[str] = None, initialize_layers_dict: dict = {}, freeze_layers_list: list = [], compile_keras_model: bool = False, output_name: str = 'score', logger=None*)

Create a RelevanceModel with default Scorer function constructed from an InteractionModel

Parameters

- **feature_config** (*FeatureConfig* object) – FeatureConfig object that defines the features to be loaded in the dataset and the preprocessing functions to be applied to each of them
- **tfrecord_type** (*{"example", "sequence_example"}*) – Type of the TFRecord protobuf message used for TFRecordDataset

- **file_io** (*FileIO* object) – file I/O handler objects for reading and writing data
- **interaction_model** (*InteractionModel* object) – InteractionModel object that converts input features into a dense feature representation
- **loss** (*RelevanceLossBase* object) – Loss object defining the final activation layer and the loss function
- **metrics** (*list*) – List of keras Metric classes that will be used for evaluating the trained model
- **optimizer** (*Optimizer*) – Tensorflow keras optimizer to be used for training the model
- **model_file** (*str, optional*) – Path to pretrained model file to be loaded for evaluation or retraining
- **initialize_layers_dict** (*dict, optional*) – Dictionary of tensorflow layer names mapped to the path of pretrained weights Use this for transfer learning with pretrained weights
- **freeze_layers_list** (*list, optional*) – List of model layer names to be frozen Use this for freezing pretrained weights from other ml4ir models
- **compile_keras_model** (*bool, optional*) – Whether the keras model loaded from disk should be compiled with loss, metrics and an optimizer
- **output_name** (*str, optional*) – Name of the output tensorflow node that captures the score
- **logger** (*Logger*, optional) – logging handler for status messages

Returns RelevanceModel object with a default scorer build with a custom InteractionModel

Return type *RelevanceModel*

classmethod from_univariate_interaction_model(*model_config, feature_config: ml4ir.base.features.feature_config.FeatureConfig, tfrecord_type: str, loss: ml4ir.base.model.losses.loss_base.RelevanceLossBase, metrics: List[Union[keras.metrics.base_metric.Metric, str]], optimizer: keras.optimizers.optimizer_v2.optimizer_v2.OptimizerV2, feature_layer_keys_to_fns: dict = {}, model_file: Optional[str] = None, initialize_layers_dict: dict = {}, freeze_layers_list: list = [], compile_keras_model: bool = False, output_name: str = 'score', max_sequence_size: int = 0, file_io: ml4ir.base.io.file_io.FileIO = None, logger=None*)

Create a RelevanceModel with default UnivariateInteractionModel

Parameters

- **feature_config** (*FeatureConfig* object) – FeatureConfig object that defines the features to be loaded in the dataset and the preprocessing functions to be applied to each of them
- **model_config** (*dict*) – dictionary defining the dense model architecture

- **tfrecord_type** (*{"example", "sequence_example"}*) – Type of the TFRecord protobuf message used for TFRecordDataset
- **file_io** (*FileIO* object) – file I/O handler objects for reading and writing data
- **loss** (*RelevanceLossBase* object) – Loss object defining the final activation layer and the loss function
- **metrics** (*list*) – List of keras Metric classes that will be used for evaluating the trained model
- **optimizer** (*Optimizer*) – Tensorflow keras optimizer to be used for training the model
- **feature_layer_keys_to_fns** (*dict*) – Dictionary of custom feature transformation functions to be applied on the input features as part of the InteractionModel
- **model_file** (*str, optional*) – Path to pretrained model file to be loaded for evaluation or retraining
- **initialize_layers_dict** (*dict, optional*) – Dictionary of tensorflow layer names mapped to the path of pretrained weights Use this for transfer learning with pretrained weights
- **freeze_layers_list** (*list, optional*) – List of model layer names to be frozen Use this for freezing pretrained weights from other ml4ir models
- **compile_keras_model** (*bool, optional*) – Whether the keras model loaded from disk should be compiled with loss, metrics and an optimizer
- **output_name** (*str, optional*) – Name of the output tensorflow node that captures the score
- **max_sequence_size** (*int, optional*) – Maximum length of the sequence to be used for SequenceExample protobuf objects
- **logger** (*Logger*, optional) – logging handler for status messages

Returns RelevanceModel object with a UnivariateInteractionModel

Return type *RelevanceModel*

build(*dataset: ml4ir.base.data.relevance_dataset.RelevanceDataset*)
Build the model layers and connect them to form a network

Parameters **dataset** (RelevanceDataset) – RelevanceDataset object used to initialize the weights and input/output spec for the network

Notes

Because we build the model using keras model subclassing API, it has no understanding of the actual inputs to expect. So we do one forward pass to initialize all the internal weights and connections

define_scheduler_as_callback(*monitor_metric*, *model_config*)
Adding reduce lr on plateau as a callback if specified

Parameters

- **monitor_metric** (*string*) – The metric to be monitored by the callback
- **model_config** (*dict*) – dictionary defining the dense model architecture

Returns The created scheduler callback object.

Return type reduce_lr

fit(*dataset: ml4ir.base.data.relevance_dataset.RelevanceDataset*, *num_epochs: int*, *models_dir: str*, *logs_dir: Optional[str] = None*, *logging_frequency: int = 25*, *monitor_metric: str = ''*, *monitor_mode: str = ''*, *patience=2*)

Trains model for defined number of epochs and returns the training and validation metrics as a dictionary

Parameters

- **dataset** (*RelevanceDataset* object) – RelevanceDataset object to be used for training and validation
- **num_epochs** (*int*) – Value specifying number of epochs to train for
- **models_dir** (*str*) – Directory to save model checkpoints
- **logs_dir** (*str, optional*) – Directory to save model logs If set to False, no progress logs will be written
- **logging_frequency** (*int, optional*) – Every #batches to log results
- **monitor_metric** (*str, optional*) – Name of the metric to monitor for early stopping, checkpointing
- **monitor_mode** (*{"max", "min"}*) – Whether to maximize or minimize the monitoring metric
- **patience** (*int*) – Number of epochs to wait before early stopping

Returns **train_metrics** – Train and validation metrics in a single dictionary where key is metric name and value is floating point metric value. This dictionary will be used for experiment tracking for each ml4ir run

Return type dict

predict(*test_dataset: tensorflow.python.data.ops.readers.TFRecordDatasetV2*, *inference_signature: str = 'serving_default'*, *additional_features: dict = {}*, *logs_dir: Optional[str] = None*, *logging_frequency: int = 25*)

Predict the scores on the test dataset using the trained model

Parameters

- **test_dataset** (*Dataset* object) – *Dataset* object for which predictions are to be made
- **inference_signature** (*str, optional*) – If using a SavedModel for prediction, specify the inference signature to be used for computing scores
- **additional_features** (*dict, optional*) – Dictionary containing new feature name and function definition to compute them. Use this to compute additional features from the scores. For example, converting ranking scores for each document into ranks for the query
- **logs_dir** (*str, optional*) – Path to directory to save logs
- **logging_frequency** (*int*) – Value representing how often(in batches) to log status

Returns pandas DataFrame containing the predictions on the test dataset made with the *RelevanceModel*

Return type *pd.DataFrame*

evaluate(*test_dataset: tensorflow.python.data.ops.readers.TFRecordDatasetV2*, *inference_signature: str = None*, *additional_features: dict = {}*, *group_metrics_min_queries: int = 50*, *logs_dir: Optional[str] = None*, *logging_frequency: int = 25*, *compute_intermediate_stats: bool = True*)

Evaluate the RelevanceModel

Parameters

- **test_dataset** (*an instance of tf.data.dataset*) –
- **inference_signature** (*str, optional*) – If using a SavedModel for prediction, specify the inference signature to be used for computing scores
- **additional_features** (*dict, optional*) – Dictionary containing new feature name and function definition to compute them. Use this to compute additional features from the scores. For example, converting ranking scores for each document into ranks for the query
- **group_metrics_min_queries** (*int, optional*) – Minimum count threshold per group to be considered for computing groupwise metrics
- **logs_dir** (*str, optional*) – Path to directory to save logs
- **logging_frequency** (*int*) – Value representing how often(in batches) to log status
- **compute_intermediate_stats** (*bool*) – Determines if group metrics and other intermediate stats on the test set should be computed

Returns

- **df_overall_metrics** (*pd.DataFrame* object) – *pd.DataFrame* containing overall metrics
- **df_groupwise_metrics** (*pd.DataFrame* object) – *pd.DataFrame* containing groupwise metrics if group_metric_keys are defined in the FeatureConfig
- **metrics_dict** (*dict*) – metrics as a dictionary of metric names mapping to values

Notes

You can directly do a *model.evaluate()* only if the keras model is compiled

Override this method to implement your own evaluation metrics.

run_ttest(*mean*, *variance*, *n*, *ttest_pvalue_threshold*)

Compute the paired t-test statistic and its p-value given mean, standard deviation and sample count :param mean: The mean of the rank differences for the entire dataset :type mean: float :param variance: The variance of the rank differences for the entire dataset :type variance: float :param n: The number of samples in the entire dataset :type n: int :param ttest_pvalue_threshold: P-value threshold for student t-test :type ttest_pvalue_threshold: float :param metrics_dict: dictionary of metrics to keep track :type metrics_dict: dict

Returns **t_test_metrics_dict** – A dictionary with the t-test metrics recorded.

Return type Dictionary

save(*models_dir: str*, *preprocessing_keys_to_fns={}*, *postprocessing_fn=None*, *required_fields_only: bool = True*, *pad_sequence: bool = False*, *sub_dir: str = 'final'*, *dataset: Optional[ml4ir.base.data.relevance_dataset.RelevanceDataset] = None*, *experiment_details: Optional[dict] = None*)

Save the RelevanceModel as a tensorflow SavedModel to the *models_dir*

There are two different serving signatures currently used to save the model:

- *default*: default keras model without any pre/post processing wrapper
- ***tfrecord*: serving signature that allows keras model to be served using TFRecord proto messages.** Allows definition of custom pre/post processing logic

Additionally, each model layer is also saved as a separate numpy zipped array to enable transfer learning with other ml4ir models.

Parameters

- **models_dir** (*str*) – path to directory to save the model
- **preprocessing_keys_to_fns** (*dict*) – dictionary mapping function names to tf.functions that should be saved in the preprocessing step of the tfrecord serving signature
- **postprocessing_fn** (*function*) – custom tensorflow compatible postprocessing function to be used at serving time. Saved as part of the postprocessing layer of the tfrecord serving signature
- **required_fields_only** (*bool*) – boolean value defining if only required fields need to be added to the tfrecord parsing function at serving time
- **pad_sequence** (*bool, optional*) – Value defining if sequences should be padded for SequenceExample proto inputs at serving time. Set this to False if you want to not handle padded scores.
- **sub_dir** (*str, optional*) – sub directory name to save the model into
- **dataset** (*RelevanceDataset* object) – RelevanceDataset object that can optionally be passed to be used by downstream jobs that want to save the data along with the model. Note that this feature is currently unimplemented and is upto the users to override and customize.
- **experiment_details** (*dict*) – Dictionary containing metadata and results about the current experiment

Notes

All the functions passed under *preprocessing_keys_to_fns* here must be serializable tensor graph operations

load(*model_file: str*) → keras.engine.training.Model
Loads model from the SavedModel file specified

Parameters **model_file** (*str*) – path to file with saved tf keras model

Returns Tensorflow keras model loaded from file

Return type *tf.keras.Model*

Notes

Retraining currently not supported! Would require compiling the model with the right loss and optimizer states

load_weights(*model_file: str*)
Load saved model with compile=False

Parameters **model_file** (*str*) – path to file with saved tf keras model

calibrate(*relevance_dataset*, *logger*, *logs_dir_local*, ***kwargs*) → Tuple[numpy.ndarray, ...]
Calibrate model with temperature scaling :param relevance_dataset: RelevanceDataset object to be used for training and evaluating temperature scaling :type relevance_dataset: RelevanceDataset :param logger: Logger object to log events :type logger: Logger :param logs_dir_local: path to save the calibration results. (zipped csv file containing original

probabilities, calibrated probabilities, . . .)

Returns

- *Union[np.ndarray, Tuple[np.ndarray, ...]]*
- *optimizer output containing temperature value learned during temperature scaling*

add_temperature_layer (*temperature: float = 1.0*, *layer_name: str = 'temperature_layer'*)

Add temperature layer to the input of last activation (softmax) layer :param self: input RelevanceModel object that its last layer inputs will be divided by a

temperature value

Parameters

- **temperature** (*float*) – a scalar value to scale the last activation layer inputs
- **layer_name** (*str*) – name of the temperature scaling layer

Returns

- *RelevanceModel*
- *updated RelevanceModel object with temperature*

3.4.3.2 RankingModel

class ml4ir.applications.ranking.model.ranking_model.**RankingModel**(*feature_config: ml4ir.base.features.feature_config.Fe tfrecord_type: str, file_io: ml4ir.base.io.file_io.FileIO, scorer: Optional[ml4ir.base.model.scoring.scor = None, metrics: List[Union[keras.metrics.base_metr str]] = [], optimizer: Optional[keras.optimizers.optimizer_v2 = None, model_file: Optional[str] = None, initialize_layers_dict: dict = {}, freeze_layers_list: list = [], compile_keras_model: bool = False, output_name: str = 'score', logger=None, eval_config: dict = {}*)

Bases: *ml4ir.base.model.relevance_model.RelevanceModel*

Constructor to instantiate a RelevanceModel that can be used for training and evaluating the search ML task

Parameters

- **feature_config** (*FeatureConfig* object) – FeatureConfig object that defines the features to be loaded in the dataset and the preprocessing functions to be applied to each of them
- **tfrecord_type** (*{"example", "sequence_example"}*) – Type of the TFRecord protobuf message used for TFRecordDataset
- **file_io** (*FileIO* object) – file I/O handler objects for reading and writing data
- **scorer** (*RelevanceScorer* object) – Scorer object that wraps an InteractionModel and converts input features into scores
- **metrics** (*list*) – List of keras Metric objects/strings that will be used for evaluating the trained model
- **optimizer** (*Optimizer*) – Tensorflow keras optimizer to be used for training the model

- **model_file** (*str, optional*) – Path to pretrained model file to be loaded for evaluation or retraining
- **initialize_layers_dict** (*dict, optional*) – Dictionary of tensorflow layer names mapped to the path of pretrained weights Use this for transfer learning with pretrained weights
- **freeze_layers_list** (*list, optional*) – List of model layer names to be frozen Use this for freezing pretrained weights from other ml4ir models
- **compile_keras_model** (*bool, optional*) – Whether the keras model loaded from disk should be compiled with loss, metrics and an optimizer
- **output_name** (*str, optional*) – Name of the output tensorflow node that captures the score
- **logger** (*Logger*, optional) – logging handler for status messages
- **eval_config** (*dict*) – A dictionary of Evaluation config parameters

predict (*test_dataset: tensorflow.python.data.ops.readers.TFRecordDatasetV2*, *inference_signature: str = 'serving_default'*, *additional_features: dict = {}*, *logs_dir: Optional[str] = None*, *logging_frequency: int = 25*)

Predict the scores on the test dataset using the trained model

Parameters

- **test_dataset** (*Dataset* object) – *Dataset* object for which predictions are to be made
- **inference_signature** (*str, optional*) – If using a SavedModel for prediction, specify the inference signature to be used for computing scores
- **additional_features** (*dict, optional*) – Dictionary containing new feature name and function definition to compute them. Use this to compute additional features from the scores. For example, converting ranking scores for each document into ranks for the query
- **logs_dir** (*str, optional*) – Path to directory to save logs
- **logging_frequency** (*int*) – Value representing how often(in batches) to log status

Returns pandas DataFrame containing the predictions on the test dataset made with the *RelevanceModel*

Return type *pd.DataFrame*

evaluate (*test_dataset: tensorflow.python.data.ops.readers.TFRecordDatasetV2*, *inference_signature: str = None*, *additional_features: dict = {}*, *group_metrics_min_queries: int = 50*, *logs_dir: Optional[str] = None*, *logging_frequency: int = 25*, *compute_intermediate_stats: bool = True*)

Evaluate the RelevanceModel

Parameters

- **test_dataset** (*an instance of tf.data.dataset*) –
- **inference_signature** (*str, optional*) – If using a SavedModel for prediction, specify the inference signature to be used for computing scores
- **additional_features** (*dict, optional*) – Dictionary containing new feature name and function definition to compute them. Use this to compute additional features from the scores. For example, converting ranking scores for each document into ranks for the query

- **group_metrics_min_queries** (*int, optional*) – Minimum count threshold per group to be considered for computing groupwise metrics
- **logs_dir** (*str, optional*) – Path to directory to save logs
- **logging_frequency** (*int*) – Value representing how often(in batches) to log status
- **compute_intermediate_stats** (*bool*) – [Currently ignored] Determines if group metrics and other intermediate stats on the test set should be computed

Returns

- **df_overall_metrics** (*pd.DataFrame* object) – *pd.DataFrame* containing overall metrics
- **df_groupwise_metrics** (*pd.DataFrame* object) – *pd.DataFrame* containing groupwise metrics if group_metric_keys are defined in the FeatureConfig
- **metrics_dict** (*dict*) – metrics as a dictionary of metric names mapping to values

Notes

You can directly do a *model.evaluate()* only if the keras model is compiled

Override this method to implement your own evaluation metrics.

save(*models_dir: str*, *preprocessing_keys_to_fns={}*, *postprocessing_fn=None*, *required_fields_only: bool = True*, *pad_sequence: bool = False*, *dataset: Optional[ml4ir.base.data.relevance_dataset.RelevanceDataset] = None*, *experiment_details: Optional[dict] = None*)

Save the RelevanceModel as a tensorflow SavedModel to the *models_dir* Additionally, sets the score for the padded records to 0

There are two different serving signatures currently used to save the model *default*: default keras model without any pre/post processing wrapper *tfrecord*: serving signature that allows keras model to be served using TFRecord proto messages.

Allows definition of custom pre/post processing logic

Additionally, each model layer is also saved as a separate numpy zipped array to enable transfer learning with other ml4ir models.

Parameters

- **models_dir** (*str*) – path to directory to save the model
- **preprocessing_keys_to_fns** (*dict*) – dictionary mapping function names to tf.functions that should be saved in the preprocessing step of the tfrecord serving signature
- **postprocessing_fn** (*function*) – custom tensorflow compatible postprocessing function to be used at serving time. Saved as part of the postprocessing layer of the tfrecord serving signature
- **required_fields_only** (*bool*) – boolean value defining if only required fields need to be added to the tfrecord parsing function at serving time
- **pad_sequence** (*bool, optional*) – Value defining if sequences should be padded for SequenceExample proto inputs at serving time. Set this to False if you want to not handle padded scores.
- **dataset** (*RelevanceDataset* object) – RelevanceDataset object that can optionally be passed to be used by downstream jobs that want to save the data along with the model. Note that this feature is currently unimplemented and is upto the users to override and customize.

- **experiment_details** (*dict*) – Dictionary containing metadata and results about the current experiment

Notes

All the functions passed under *preprocessing_keys_to_fns* here must be serializable tensor graph operations

3.4.4 Feature Configuration

3.4.4.1 FeatureConfig

class ml4ir.base.features.feature_config.**FeatureConfig**(*features_dict*, *logger: Optional[logging.Logger] = None*)

Bases: object

Class that defines the features and their configurations used for training, evaluating and serving a RelevanceModel on ml4ir.

features_dict
Dictionary of features containing the configuration for every feature in the model. This dictionary is used to define the FeatureConfig object.

Type dict

logger
Logging handler to log progress messages

Type *Logging* object

query_key
Dictionary containing the feature configuration for the unique data point ID, query key

Type dict

label
Dictionary containing the feature configuration for the label field for training and evaluating the model

Type dict

mask
Dictionary containing the feature configuration for the computed mask field which is used to identify padded values

Type dict

features
List of dictionaries containing configurations for all the features excluding query_key and label

Type list of dict

all_features
List of dictionaries containing configurations for all the features including query_key and label

Type list of dict

train_features
List of dictionaries containing configurations for all the features which are used for training, identified by *trainable=False*

Type list of dict

metadata_features
List of dictionaries containing configurations for all the features which are NOT used for training, identified by *trainable=False*. These can be used for computing custom losses and metrics.

Type list of dict

features_to_log
List of dictionaries containing configurations for all the features which will be logged when running *model.predict()*, identified using *log_at_inference=True*

Type list of dict

group_metrics_keys
List of dictionaries containing configurations for all the features which will be used to compute groupwise metrics

Type list of dict

Notes

Abstract class that is overriden by ExampleFeatureConfig and SequenceExampleFeatureConfig for the respective TFRecord types

Constructor to instantiate a FeatureConfig object

Parameters

- **features_dict** (*dict*) – Dictionary containing the feature configuration for each of the model features
- **logger** (*Logging* object, optional) – Logging object handler for logging progress messages

initialize_features()
Initialize the feature attributes with empty lists accordingly

static get_instance(*feature_config_dict: dict*, *tfrecord_type: str*, *logger: logging.Logger*)
Factory method to get *FeatureConfig* object from a dictionary of feature configurations based on the TFRecord type

Parameters

- **feature_config_dict** (*dict*) – Dictionary containing the feature definitions for all the features for the model
- **tfrecord_type** (*{"example", "sequence_example"}*) – Type of the TFRecord message type used for the ml4ir RelevanceModel
- **logger** (*Logging* object) – Logging object handler to log status and progress messages

Returns ExampleFeatureConfig or SequenceExampleFeatureConfig object computed from the feature configuration dictionary

Return type *FeatureConfig* object

extract_features()
Extract the features from the input feature config dictionary and assign to relevant FeatureConfig attributes

log_initialization()
Log initial state of FeatureConfig object after extracting all the attributes

get_query_key(*key: str = None*)
Getter method for query_key in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str*) – Value from the query_key feature configuration to be fetched

Returns Query key value or entire config dictionary based on if the key is passed

Return type str or int or bool or dict

get_label(*key: str = None*)
Getter method for label in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str*) – Value from the label feature configuration to be fetched

Returns Label value or entire config dictionary based on if the key is passed

Return type str or int or bool or dict

get_aux_label(*key: str = None*)
Getter method for label in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str*) – Value from the label feature configuration to be fetched

Returns Label value or entire config dictionary based on if the key is passed

Return type str or int or bool or dict

get_mask(*key: str = None*)
Getter method for mask in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str*) – Value from the mask feature configuration to be fetched

Returns Label value or entire config dictionary based on if the key is passed

Return type str or int or bool or dict

get_feature_by_node_name(*name: str*)
Getter method for feature by node name in FeatureConfig object

Parameters **name** (*str*) – Name of the feature node name to fetch

Returns Feature config dictionary for the name of the feature passed

Return type dict

get_feature(*name: str*)
Getter method for feature in FeatureConfig object

Parameters **name** (*str*) – Name of the feature to fetch

Returns Feature config dictionary for the name of the feature passed

Return type dict

feature_exists(*name: str*, *trainable=True*)
Check if a feature exists in FeatureConfig object

Parameters **name** (*str*) – Name of the feature to fetch

Returns If a feature exists

Return type Boolean

set_feature(*name: str*, *new_feature_info: dict*)
Setter method to set the feature_info of a feature in the FeatureConfig as specified by the name argument

Parameters

- **name** (*str*) – name of feature whose feature_info is to be updated
- **new_feature_info** (*dict*) – dictionary used to set the feature_info for the feature with specified name

get_all_features(*key: str = None*, *include_label: bool = True*, *include_mask: bool = True*)
Getter method for all_features in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters

- **key** (*str, optional*) – Name of the configuration key to be fetched. If None, then entire dictionary for the feature is returned
- **include_label** (*bool, optional*) – Include label in list of features returned
- **include_mask** (*bool, optional*) – Include mask in the list of features returned. Only applicable with SequenceExampleFeatureConfig currently

Returns Lift of feature configuration dictionaries or values for all features in FeatureConfig

Return type list

get_train_features(*key: str = None*)
Getter method for train_features in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str, optional*) – Name of the configuration key to be fetched. If None, then entire dictionary for the feature is returned

Returns Lift of feature configuration dictionaries or values for trainable features in FeatureConfig

Return type list

get_metadata_features(*key: str = None*)
Getter method for metadata_features in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str, optional*) – Name of the configuration key to be fetched. If None, then entire dictionary for the feature is returned

Returns Lift of feature configuration dictionaries or values for metadata features in FeatureConfig

Return type list

get_features_to_log(*key: str = None*)
Getter method for features_to_log in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str, optional*) – Name of the configuration key to be fetched. If None, then entire dictionary for the feature is returned

Returns Lift of feature configuration dictionaries or values for features to be logged at inference

Return type list

get_group_metrics_keys(*key: str = None*)
Getter method for group_metrics_keys in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str, optional*) – Name of the configuration key to be fetched. If None, then entire dictionary for the feature is returned

Returns Lift of feature configuration dictionaries or values for features used to compute group-wise metrics

Return type list

get_dtype (*feature_info: dict*)
Retrieve data type of a feature

Parameters **feature_info** (*dict*) – Dictionary containing configuration for the feature

Returns Data type of the feature

Return type str

get_default_value (*feature_info*)
Retrieve default value of a feature

Parameters **feature_info** (*dict*) – Dictionary containing configuration for the feature

Returns Default value of the feature

Return type str or int or float

create_dummy_protobuf (*num_records=1*, *required_only=False*)
Generate a dummy TFRecord protobuffer with dummy values

Parameters

- **num_records** (*int*) – Number of records or sequence features per TFRecord message to fetch
- **required_only** (*bool*) – Whether to fetch on fields with *required_only=True*

Returns Example or SequenceExample object with dummy values generated from the FeatureConfig

Return type protobuffer object

get_hyperparameter_dict ()
Create hyperparameter configs to track model metadata for best model selection Unwraps the feature config for each of the features to add preprocessing_info and feature_layer_info as key value pairs that can be tracked across the experiment. This can be used to identify the values that were set for the different feature layers in a given experiment. Will be used during best model selection and Hyper Parameter Optimization.

Returns Flattened dictionary of important configuration keys and values that can be used for tracking the experiment run

Return type dict

3.4.4.2 ExampleFeatureConfig

class ml4ir.base.features.feature_config.**ExampleFeatureConfig** (*features_dict*, *logger: Optional[logging.Logger] = None*)
Bases: *ml4ir.base.features.feature_config.FeatureConfig*

Class that defines the features and their configurations used for training, evaluating and serving a RelevanceModel on ml4ir for Example data

features_dict
Dictionary of features containing the configuration for every feature in the model. This dictionary is used to define the FeatureConfig object.

Type dict

logger
Logging handler to log progress messages

Type *Logging* object

query_key
Dictionary containing the feature configuration for the unique data point ID, query key

Type dict

label
Dictionary containing the feature configuration for the label field for training and evaluating the model

Type dict

features
List of dictionaries containing configurations for all the features excluding query_key and label

Type list of dict

all_features
List of dictionaries containing configurations for all the features including query_key and label

Type list of dict

train_features
List of dictionaries containing configurations for all the features which are used for training, identified by *trainable=False*

Type list of dict

metadata_features
List of dictionaries containing configurations for all the features which are NOT used for training, identified by *trainable=False*. These can be used for computing custom losses and metrics.

Type list of dict

features_to_log
List of dictionaries containing configurations for all the features which will be logged when running *model.predict()*, identified using *log_at_inference=True*

Type list of dict

group_metrics_keys
List of dictionaries containing configurations for all the features which will be used to compute groupwise metrics

Type list of dict

Constructor to instantiate a FeatureConfig object

Parameters

- **features_dict** (*dict*) – Dictionary containing the feature configuration for each of the model features
- **logger** (*Logging* object, optional) – Logging object handler for logging progress messages

create_dummy_protobuf(*num_records=1*, *required_only=False*)
Create a SequenceExample protobuffer with dummy values

3.4.4.3 SequenceExampleFeatureConfig

class ml4ir.base.features.feature_config.**SequenceExampleFeatureConfig**(*features_dict*, *logger*)

Bases: *ml4ir.base.features.feature_config.FeatureConfig*

Class that defines the features and their configurations used for training, evaluating and serving a RelevanceModel on ml4ir for SequenceExample data

features_dict
Dictionary of features containing the configuration for every feature in the model. This dictionary is used to define the FeatureConfig object.

Type dict

logger
Logging handler to log progress messages

Type *Logging* object

query_key
Dictionary containing the feature configuration for the unique data point ID, query key

Type dict

label
Dictionary containing the feature configuration for the label field for training and evaluating the model

Type dict

rank
Dictionary containing the feature configuration for the rank field for training and evaluating the model. *rank* is used to assign an ordering to the sequences in the SequenceExample

Type dict

mask
Dictionary containing the feature configuration for the mask field for training and evaluating the model. *mask* is used to identify which sequence features are padded. A value of 1 represents an existing sequence feature and 0 represents a padded sequence feature.

Type dict

features
List of dictionaries containing configurations for all the features excluding query_key and label

Type list of dict

all_features
List of dictionaries containing configurations for all the features including query_key and label

Type list of dict

context_features
List of dictionaries containing configurations for all the features which represent the features common to the entire sequence in a protobuf message

Type list of dict

sequence_features
List of dictionaries containing configurations for all the features which represent the feature unique to a sequence

Type list of dict

train_features
List of dictionaries containing configurations for all the features which are used for training, identified by *trainable=False*

Type list of dict

metadata_features
List of dictionaries containing configurations for all the features which are NOT used for training, identified by *trainable=False*. These can be used for computing custom losses and metrics.

Type list of dict

features_to_log
List of dictionaries containing configurations for all the features which will be logged when running *model.predict()*, identified using *log_at_inference=True*

Type list of dict

group_metrics_keys
List of dictionaries containing configurations for all the features which will be used to compute groupwise metrics

Type list of dict

Constructor to instantiate a FeatureConfig object

Parameters

- **features_dict** (*dict*) – Dictionary containing the feature configuration for each of the model features
- **logger** (*Logging* object, optional) – Logging object handler for logging progress messages

initialize_features()
Initialize the feature attributes with empty lists accordingly

extract_features()
Extract the features from the input feature config dictionary and assign to relevant FeatureConfig attributes

get_context_features(*key: str = None*)
Getter method for context_features in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str, optional*) – Name of the configuration key to be fetched. If None, then entire dictionary for the feature is returned

Returns Lift of feature configuration dictionaries or values for context features common to all sequence

Return type list

get_sequence_features(*key: str = None*)
Getter method for sequence_features in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str, optional*) – Name of the configuration key to be fetched. If None, then entire dictionary for the feature is returned

Returns Lift of feature configuration dictionaries or values for sequence features unique to each sequence

Return type list

log_initialization()
Log initial state of FeatureConfig object after extracting all the attributes

generate_mask()
Add mask information used to flag padded records. In order to create a batch of sequence examples from n TFRecords, we need to make sure that they all have the same number of sequences. To do this, we pad sequence records to a fixed max_sequence_size. Now, we do not want to use these additional padded sequence records to compute metrics and losses. Hence we maintain a boolean mask to tell ml4ir the sequence records that were originally present.

In this method, we add the feature_info for the above mask feature as it is not implicitly present in the data.

Returns Dictionary configuration for the mask field that captures which sequence have been masked in a SequenceExample message

Return type dict

get_rank(*key: str = None*)
Getter method for rank in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str*) – Value from the rank feature configuration to be fetched

Returns Rank value or entire config dictionary based on if the key is passed

Return type str or int or bool or dict

get_mask(*key: str = None*)
Getter method for mask in FeatureConfig object Can additionally be used to only fetch a particular value from the dict

Parameters **key** (*str*) – Value from the mask feature configuration to be fetched

Returns Mask value or entire config dictionary based on if the key is passed

Return type str or int or bool or dict

create_dummy_protobuf(*num_records=1*, *required_only=False*)
Generate a dummy TFRecord protobuffer with dummy values

Parameters

- **num_records** (*int*) – Number of records or sequence features per TFRecord message to fetch
- **required_only** (*bool*) – Whether to fetch on fields with *required_only=True*

Returns Example or SequenceExample object with dummy values generated from the FeatureConfig

Return type protobuffer object

3.4.5 Losses

3.4.5.1 RelevanceLossBase

class ml4ir.base.model.losses.loss_base.**RelevanceLossBase**(*trainable=True*, *name=None*, *dtype=None*, *dynamic=False*, ***kwargs*)

Bases: keras.engine.base_layer.Layer

Abstract class that defines the loss and final activation function used to train a RelevanceModel

call(*inputs*, *y_true*, *y_pred*, *training=None*)
Compute the loss using predicted probabilities and expected labels

Parameters

- **inputs** (*dict of dict of tensors*) – Dictionary of input feature tensors
- **y_true** (*tensor*) – True labels
- **y_pred** (*tensor*) – Predicted scores
- **training** (*boolean*) – Boolean indicating whether the layer is being used in training mode

Returns Resulting loss tensor after applying comparing the y_pred and y_true values

Return type tensor

final_activation_op(*inputs*, *training=None*)
Final activation layer that is applied to the logits tensor to get the scores

Parameters

- **inputs** (*dict of dict of tensors*) – Dictionary of input feature tensors with scores
- **training** (*boolean*) – Boolean indicating whether the layer is being used in training mode

Returns Resulting score tensor after applying the function on the logits

Return type tensor

get_config()
Return layer config that is used while serialization

3.4.5.2 SigmoidCrossEntropy

class ml4ir.applications.ranking.model.losses.pointwise_losses.**SigmoidCrossEntropy**(*loss_key='*, *scoring_type='*, *output_name=*, ***kwargs*)

Bases: ml4ir.applications.ranking.model.losses.loss_base.PointwiseLossBase

call(*inputs*, *y_true*, *y_pred*, *training=None*)
Get the sigmoid cross entropy loss Additionally can pass in record positions to handle positional bias

Parameters

- **inputs** (*dict of dict of tensors*) – Dictionary of input feature tensors
- **y_true** (*tensor*) – True labels
- **y_pred** (*tensor*) – Predicted scores
- **training** (*boolean*) – Boolean indicating whether the layer is being used in training mode

Returns Scalar sigmoid cross entropy loss tensor

Return type tensor

Notes

Uses *mask* field to exclude padded records from contributing to the loss

final_activation_op(*inputs*, *training=None*)
Get sigmoid activated scores on logits

Parameters **inputs** (*dict of dict of tensors*) – Dictionary of input feature tensors

Returns sigmoid activated scores

Return type tensor

3.4.5.3 RankOneListNet

class ml4ir.applications.ranking.model.losses.listwise_losses.**RankOneListNet**(*loss_key: str = 'rank_one_listnet', scoring_type: str = 'listwise', output_name: str = 'score', **kwargs*)

Bases: ml4ir.applications.ranking.model.losses.listwise_losses.SoftmaxCrossEntropy

Parameters

- **loss_key** (*str*) – Name of the loss function as specified by LossKey
- **scoring_type** (*str*) – Type of scoring function - pointwise, pairwise, groupwise
- **output_name** (*str*) – Name of the output node for the predicted scores

call(*inputs*, *y_true*, *y_pred*, *training=None*)
Define a masked rank 1 ListNet loss. This loss is useful for multi-label classification when we have multiple click labels per document. This is because the loss breaks down the comparison between y_pred and y_true into individual binary assessments. Ref -> https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2007-40.pdf

Parameters

- **inputs** (*dict of dict of tensors*) – Dictionary of input feature tensors
- **y_true** (*tensor*) – True labels
- **y_pred** (*tensor*) – Predicted scores
- **training** (*boolean*) – Boolean indicating whether the layer is being used in training mode

Returns Scalar sigmoid cross entropy loss tensor

Return type tensor

Notes

Uses *mask* field to exclude padded records from contributing to the loss

3.4.5.4 CategoricalCrossEntropy

class ml4ir.applications.classification.model.losses.categorical_cross_entropy.**CategoricalC**

Bases: *ml4ir.base.model.losses.loss_base.RelevanceLossBase*

Initialize categorical cross entropy loss

Parameters **output_name** (*str*) – Name of the output node after final activation op

call(*inputs*, *y_true*, *y_pred*, *training=None*)
Define a categorical cross entropy loss

Parameters

- **inputs** (*dict of dict of tensors*) – Dictionary of input feature tensors
- **y_true** (*tensor*) – True labels
- **y_pred** (*tensor*) – Predicted scores
- **training** (*boolean*) – Boolean indicating whether the layer is being used in training mode

Returns Categorical cross entropy loss

Return type function

final_activation_op(*inputs*, *training=None*)
Get softmax activated scores on logits

Parameters **inputs** (*dict of dict of tensors*) – Dictionary of input feature tensors

Returns Softmax activated scores

Return type tensor

get_config()
Return layer config that is used while serialization

3.4.6 Metrics

Any keras supported Metric class can be used with ml4ir. ml4ir comes prepackaged with the following popular search metrics.

3.4.6.1 MeanReciprocalRank

class ml4ir.applications.ranking.model.metrics.metrics_impl.**MRR**(*name='mean'*, *dtype=None*)

Bases: ml4ir.applications.ranking.model.metrics.metrics_impl.MeanRankMetric

Custom metric class to compute the Mean Reciprocal Rank.

Calculates the mean of the reciprocal ranks of the clicked records from a list of queries.

Examples

```
>>> `y_true` is [[0, 0, 1], [0, 1, 0]]
>>> `y_pred` is [[0.1, 0.9, 0.8], [0.05, 0.95, 0]]
>>> then the MRR is 0.75
```

3.4.6.2 AverageClickRank

class ml4ir.applications.ranking.model.metrics.metrics_impl.**ACR**(*name='mean'*, *dtype=None*)

Bases: ml4ir.applications.ranking.model.metrics.metrics_impl.MeanRankMetric

Custom metric class to compute the Average Click Rank.

Calculates the mean of the ranks of the clicked records from a list of queries.

Examples

```
>>> `y_true` is [[0, 0, 1], [0, 1, 0]]
>>> `y_pred` is [[0.1, 0.9, 0.8], [0.05, 0.95, 0]]
>>> then the ACR is 1.50
```

3.4.6.3 CategoricalAccuracy

3.4.6.4 Top5CategoricalAccuracy

class ml4ir.applications.classification.model.metrics.metrics_impl.**Top5CategoricalAccuracy**

Bases: keras.metrics.metrics.TopKCategoricalAccuracy

Custom metric class to compute the Top K Categorical Accuracy.

Currently a wrapper around tf.keras.metrics.TopKCategoricalAccuracy that squeezes one dimension. It maintains consistency of arguments to __init__

Creates a CategoricalAccuracy instance

Parameters **name** (*str*) – Name of the metric

update_state(*y_true*, *y_pred*, *sample_weight=None*)

Squeeze the second dimension(axis=1) and compute top K categorical accuracy

Parameters

- **y_true** (*Tensor object*) – Tensor containing true class labels Shape : [batch_size, 1, num_classes]
- **y_pred** (*Tensor object*) – Tensor containing predicted scores for the classes Shape : [batch_size, 1, num_classes]
- **sample_weight** (*dict*) – Dictionary containing weights for the classes to measure weighted average metric

Returns Top K categorical accuracy computed on y_true and y_pred

Return type Tensor object

Notes

Input shape is a 3 dimensional tensor of size (batch_size, 1, num_classes). We are squeezing the second dimension to follow the API of tf.keras.metrics.TopKCategoricalAccuracy

Axis 1 of y_true and y_pred must be of size 1, otherwise *tf.squeeze* will throw error.

3.4.7 Feature Processing

ml4ir.base.features.preprocessing.**preprocess_text**

String preprocessing function that removes punctuation and converts strings to lower case based on the arguments.

Parameters feature_tensor : Tensor object

input feature tensor of type tf.string

remove_punctuation [bool] Whether to remove punctuation characters from strings

to_lower [bool] Whether to convert string to lower case

punctuation [str] Punctuation characters to replace (a single string containing the character to remove

replace_with_whitespace [bool] if True punctuation will be replaced by whitespace (i.e. used as separator), note that leading and trailing whitespace will also be removed, as well as consecutive whitespaces.

Returns Processed string tensor

Return type Tensor object

Examples

Input:

```
>>> feature_tensor = "ABCabc123,,,"
>>> remove_punctuation = True
>>> to_lower = True
```

Output:

```
>>> "abcabc123"
```

ml4ir.base.features.preprocessing.**get_one_hot_label_vectorizer**(*feature_info*, *file_io: ml4ir.base.io.file_io.FileIO*)

Returns a tf function to convert categorical string labels to a one hot encoding.

Parameters

- **feature_info** (*dict*) – Dictionary representing the configuration parameters for the specific feature from the FeatureConfig. See categorical_indicator_with_vocabulary_file, here it is used to read a vocabulary file to create the one hot encoding.
- **file_io** (*FileIO required to load the vocabulary file.*) –

Returns Function that converts labels into one hot vectors

Return type function

Examples

Input:

```
>>> feature_tensor = ["abc", "xyz", "abc"]
>>> vocabulary file
>>>    abc -> 0
>>>    xyz -> 1
>>>    def -> 2
```

Output:

```
>>> [[1, 0, 0], [0, 1, 0], [1, 0, 0]]
```

ml4ir.base.features.preprocessing.**split_and_pad_string**

String preprocessing function that splits and pads a sequence based on the max_length.

Parameters

- **feature_tensor** (*Tensor object*) – Input feature tensor of type tf.string.
- **split_char** (*str*) – String separator to split the string input.
- **max_length** (*int*) – max length of the sequence produced after padding.

Returns processed float tensor

Return type Tensor object

Examples

Input:

```
>>> feature_tensor = "AAA,BBB,CCC"
>>> split_char = ","
>>> max_length = 5
```

Output:

```
>>> ['AAA', 'BBB', 'CCC', '', '']
```

ml4ir.base.features.preprocessing.**natural_log**

Compute the signed log of the feature_tensor

Parameters

- **feature_tensor** (*Tensor object*) – input feature tensor of type tf.float32
- **shift** (*int*) – floating point shift that is added to the feature tensor element wise before computing natural log (used to handle 0 values)

Examples

Input:

```
>>> feature_tensor = [10, 0]
>>> shift = 1
```

Output:

```
>>> [2.39, 0.]
```

3.4.8 Feature Transformation

3.4.8.1 Categorical Feature Transformations

class ml4ir.base.features.feature_fns.categorical.**CategoricalEmbeddingWithHashBuckets**(*feature_dict*, *file_io*, *ml4ir.b*, ***kwa*

Bases: ml4ir.base.features.feature_fns.base.BaseFeatureLayerOp

Converts a string feature tensor into a categorical embedding. Works by first converting the string into num_hash_buckets buckets each of size hash_bucket_size, then converting each hash bucket into a categorical embedding of dimension embedding_size. Finally, these embeddings are combined either through mean, sum or concat operations to generate the final embedding based on the feature_info.

Initialize the layer to get categorical embedding with hash buckets

Parameters

- **feature_info** (*dict*) – Dictionary representing the configuration parameters for the specific feature from the FeatureConfig
- **file_io** (*FileIO object*) – FileIO handler object for reading and writing

Notes

Args under feature_layer_info:

num_hash_buckets [int] number of different hash buckets to convert the input string into

hash_bucket_size [int] the size of each hash bucket

embedding_size [int] dimension size of the categorical embedding

merge_mode [str] can be one of "mean", "sum", "concat" representing the mode of combining embeddings from each categorical embedding

LAYER_NAME = 'categorical_embedding_with_hash_buckets'

NUM_HASH_BUCKETS = 'num_hash_buckets'

HASH_BUCKET_SIZE = 'hash_bucket_size'

EMBEDDING_SIZE = 'embedding_size'

MERGE_MODE = 'merge_mode'

call(*inputs*, *training=None*)
Defines the forward pass for the layer on the inputs tensor

Parameters

- **inputs** (*tensor*) – Input tensor on which the feature transforms are applied
- **training** (*boolean*) – Boolean flag indicating if the layer is being used in training mode or not

Returns Resulting tensor after the forward pass through the feature transform layer

Return type tf.Tensor

class ml4ir.base.features.feature_fns.categorical.**CategoricalEmbeddingWithIndices**(*feature_info*, *dict*, *file_io*: *ml4ir.base.io*, ***kwargs*)

Bases: ml4ir.base.features.feature_fns.base.BaseFeatureLayerOp

Converts input integer tensor into categorical embedding. Works by converting the categorical indices in the input feature_tensor, represented as integer values, into categorical embeddings based on the feature_info.

Initialize feature layer to convert categorical feature into embedding based on indices

Parameters

- **feature_info** (*dict*) – Dictionary representing the configuration parameters for the specific feature from the FeatureConfig
- **file_io** (*FileIO object*) – FileIO handler object for reading and writing

Notes

Args under feature_layer_info:

num_buckets [int] Maximum number of categorical values

default_value [int] default value to be assigned to indices out of the num_buckets range

embedding_size [int] dimension size of the categorical embedding

LAYER_NAME = 'categorical_embedding_with_indices'

NUM_BUCKETS = 'num_buckets'

DEFAULT_VALUE = 'default_value'

EMBEDDING_SIZE = 'embedding_size'

call(*inputs*, *training=None*)

Defines the forward pass for the layer on the inputs tensor

Parameters

- **inputs** (*tensor*) – Input tensor on which the feature transforms are applied
- **training** (*boolean*) – Boolean flag indicating if the layer is being used in training mode or not

Returns Resulting tensor after the forward pass through the feature transform layer

Return type tf.Tensor

class ml4ir.base.features.feature_fns.categorical.**CategoricalEmbeddingToEncodingBiLSTM**(*featu*, *dict*, *file_i*, *ml4i*, ***kw*)

Bases: ml4ir.base.features.feature_fns.base.BaseFeatureLayerOp

Encode a string tensor into categorical embedding. Works by converting the string into a word sequence and then generating a categorical/char embedding for each words based on the List of strings that form the vocabulary set

of categorical values, defined by the argument vocabulary_file. The char/byte embeddings are then combined using a biLSTM.

Initialize the layer to convert input string tensor into an encoding using categorical embeddings

Parameters

- **feature_info** (*dict*) – Dictionary representing the configuration parameters for the specific feature from the FeatureConfig
- **file_io** (*FileIO object*) – FileIO handler object for reading and writing

Notes

Args under feature_layer_info:

vocabulary_file [string] path to vocabulary CSV file for the input tensor containing the vocabulary to look-up. uses the "key" named column as vocabulary of the 1st column if no "key" column present.

max_length: int

max number of rows to consider from the vocabulary file. if null, considers the entire file vocabulary.

embedding_size [int]

dimension size of the embedding; if null, then the tensor is just converted to its one-hot representation

encoding_size [int] dimension size of the sequence encoding computed using a biLSTM

The input dimension for the embedding is fixed to 256 because the string is converted into a bytes sequence.

LAYER_NAME = 'categorical_embedding_to_encoding_bilstm'

VOCABULARY_FILE = 'vocabulary_file'

MAX_LENGTH = 'max_length'

EMBEDDING_SIZE = 'embedding_size'

ENCODING_SIZE = 'encoding_size'

LSTM_KERNEL_INITIALIZER = 'lstm_kernel_initializer'

call(*inputs*, *training=None*)

Defines the forward pass for the layer on the inputs tensor

Parameters

- **inputs** (*tensor*) – Input tensor on which the feature transforms are applied
- **training** (*boolean*) – Boolean flag indicating if the layer is being used in training mode or not

Returns Resulting tensor after the forward pass through the feature transform layer

Return type tf.Tensor

class ml4ir.base.features.feature_fns.categorical.**CategoricalEmbeddingWithVocabularyFile**(*fe*
di
fil
m
***

Bases: ml4ir.base.features.feature_fns.base.BaseFeatureLayerOp

Converts a string tensor into a categorical embedding representation. Works by using a vocabulary file to convert the string tensor into categorical indices and then converting the categories into embeddings based on the feature_info.

Initialize layer to define a categorical embedding using a vocabulary file

Parameters

- **feature_info** (*dict*) – Dictionary representing the configuration parameters for the specific feature from the FeatureConfig
- **file_io** (*FileIO object*) – FileIO handler object for reading and writing

Notes

Args under feature_layer_info:

vocabulary_file [string]

path to vocabulary CSV file for the input tensor containing the vocabulary to look-up. uses the "key" named column as vocabulary of the 1st column if no "key" column present.

max_length [int]

max number of rows to consider from the vocabulary file. if null, considers the entire file vocabulary.

num_oov_buckets [int]

number of out of vocabulary buckets/slots to be used to encode strings into categorical indices

embedding_size [int] dimension size of categorical embedding

The vocabulary CSV file must contain two columns - key, id, where the key is mapped to one id thereby resulting in a many-to-one vocabulary mapping. If id field is absent, a unique whole number id is assigned by default resulting in a one-to-one mapping

LAYER_NAME = 'categorical_embedding_with_vocabulary_file'

VOCABULARY_FILE = 'vocabulary_file'

MAX_LENGTH = 'max_length'

NUM_OOV_BUCKETS = 'num_oov_buckets'

NUM_BUCKETS = 'num_buckets'

EMBEDDING_SIZE = 'embedding_size'

DEFAULT_VALUE = 'default_value'

call(*inputs*, *training=None*)

Defines the forward pass for the layer on the inputs tensor

Parameters

- **inputs** (*tensor*) – Input tensor on which the feature transforms are applied
- **training** (*boolean*) – Boolean flag indicating if the layer is being used in training mode or not

Returns Resulting tensor after the forward pass through the feature transform layer

Return type tf.Tensor

class ml4ir.base.features.feature_fns.categorical.**CategoricalEmbeddingWithVocabularyFileAnd**

Bases: ml4ir.base.features.feature_fns.base.BaseFeatureLayerOp

Converts a string tensor into a categorical embedding representation. Works by using a vocabulary file to convert the string tensor into categorical indices and then converting the categories into embeddings based on the feature_info. Also uses a dropout to convert categorical indices to the OOV index of 0 at a rate of dropout_rate

Parameters

- **feature_info** (*dict*) – Dictionary representing the configuration parameters for the specific feature from the FeatureConfig
- **file_io** (*FileIO object*) – FileIO handler object for reading and writing

Notes

Args under feature_layer_info:

vocabulary_file [str] path to vocabulary CSV file for the input tensor

dropout_rate [float] rate at which to convert categorical indices to OOV

embedding_size [int] dimension size of categorical embedding

The vocabulary CSV file must contain two columns - key, id, where the key is mapped to one id thereby resulting in a many-to-one vocabulary mapping. If id field is absent, a unique natural number id is assigned by default resulting in a one-to-one mapping

OOV index will be set to 0 num_oov_buckets will be 0

LAYER_NAME = 'categorical_embedding_with_vocabulary_file_and_dropout'

VOCABULARY_FILE = 'vocabulary_file'

DROPOUT_RATE = 'dropout_rate'

EMBEDDING_SIZE = 'embedding_size'

NUM_BUCKETS = 'num_buckets'

DEFAULT_VALUE = 'default_value'

call(*inputs*, *training=None*)

Defines the forward pass for the layer on the inputs tensor

Parameters

- **inputs** (*tensor*) – Input tensor on which the feature transforms are applied
- **training** (*boolean*) – Boolean flag indicating if the layer is being used in training mode or not

Returns Resulting tensor after the forward pass through the feature transform layer

Return type tf.Tensor

class ml4ir.base.features.feature_fns.categorical.**CategoricalIndicatorWithVocabularyFile**(*fe*
di
fil
m
***

Bases: ml4ir.base.features.feature_fns.base.BaseFeatureLayerOp

Converts a string tensor into a categorical one-hot representation. Works by using a vocabulary file to convert the string tensor into categorical indices and then converting the categories into one-hot representation.

Parameters

- **feature_info** (*dict*) – Dictionary representing the configuration parameters for the specific feature from the FeatureConfig
- **file_io** (*FileIO object*) – FileIO handler object for reading and writing

Notes

Args under feature_layer_info:

vocabulary_file [string] path to vocabulary CSV file for the input tensor containing the vocabulary to look-up. uses the "key" named column as vocabulary of the 1st column if no "key" column present.

max_length [int] max number of rows to consider from the vocabulary file. if null, considers the entire file vocabulary.

num_oov_buckets [int, optional] number of out of vocabulary buckets/slots to be used to encode strings into categorical indices. If not specified, the default is 1.

The vocabulary CSV file must contain two columns - key, id, where the key is mapped to one id thereby resulting in a many-to-one vocabulary mapping. If id field is absent, a unique whole number id is assigned by default resulting in a one-to-one mapping

LAYER_NAME = 'categorical_indicator_with_vocabulary_file'

VOCABULARY_FILE = 'vocabulary_file'

MAX_LENGTH = 'max_length'

NUM_OOV_BUCKETS = 'num_oov_buckets'

call(*inputs*, *training=None*)

Defines the forward pass for the layer on the inputs tensor

Parameters

- **inputs** (*tensor*) – Input tensor on which the feature transforms are applied
- **training** (*boolean*) – Boolean flag indicating if the layer is being used in training mode or not

Returns Resulting tensor after the forward pass through the feature transform layer

Return type tf.Tensor

3.4.8.2 Sequence Feature Transformations

class ml4ir.base.features.feature_fns.sequence.**BytesSequenceToEncodingBiLSTM**(*feature_info: dict*, *file_io: ml4ir.base.io.file_io.FileIO*, ***kwargs*)

Bases: ml4ir.base.features.feature_fns.base.BaseFeatureLayerOp

Encode a string tensor into an encoding. Works by converting the string into a bytes sequence and then generating a categorical/char embedding for each of the 256 bytes. The char/byte embeddings are then combined using a biLSTM

Initialize a feature layer to convert string tensor to bytes encoding

Parameters

- **feature_info** (*dict*) – Dictionary representing the feature_config for the input feature
- **file_io** (*FileIO object*) – FileIO handler object for reading and writing

Notes

Args under *feature_layer_info*:

max_length [int] max length of bytes sequence

embedding_size [int] dimension size of the embedding; if null, then the tensor is just converted to its one-hot representation

encoding_size [int] dimension size of the sequence encoding computed using a biLSTM

The input dimension for the embedding is fixed to 256 because the string is converted into a bytes sequence.

LAYER_NAME = 'bytes_sequence_to_encoding_bilstm'

MAX_LENGTH = 'max_length'

EMBEDDING_SIZE = 'embedding_size'

LSTM_KERNEL_INITIALIZER = 'lstm_kernel_initializer'

ENCODING_SIZE = 'encoding_size'

call(*inputs*, *training=None*)

Defines the forward pass for the layer on the inputs tensor

Parameters

- **inputs** (*tensor*) – Input tensor on which the feature transforms are applied
- **training** (*boolean*) – Boolean flag indicating if the layer is being used in training mode or not

Returns Resulting tensor after the forward pass through the feature transform layer

Return type tf.Tensor

class ml4ir.base.features.feature_fns.sequence.**Global1dPooling**(*feature_info: dict*, *file_io: ml4ir.base.io.file_io.FileIO*, ***kwargs*)

Bases: ml4ir.base.features.feature_fns.base.BaseFeatureLayerOp

1D pooling to reduce a variable length sequence feature into a scalar value. This method optionally allows users to add multiple such pooling operations to produce a fixed dimensional feature vector as well.

Initialize a feature layer to apply global 1D pooling operation on input tensor

Parameters

- **feature_info** (*dict*) – Dictionary representing the feature_config for the input feature
- **file_io** (*FileIO object*) – FileIO handler object for reading and writing

Notes

Args under *feature_layer_info*:

fns [list of str] List of string pooling operations that should be applied. Must be one of ["sum", "mean", "max", "min", "count_nonzero"]

padded_val [int/float] Value to be ignored from the pooling operations.

masked_max_val [int/float] Value used to mask the padded values for computing the max and min pooling operations. This allows us to ignore these values in the min and max pool operations. For example, if all the values in the tensor are in [0., 1.], then a masked_max_val of > 1. will make sure we do not pick padded values in the min/max pooling ops. Default value: 2

LAYER_NAME = 'global_1d_pooling'

FNS = 'fns'

PADDED_VAL = 'padded_val'

MASKED_MAX_VAL = 'masked_max_val'

DEFAULT_MASKED_MAX_VAL = 2.0

call(*inputs*, *training=None*)
Defines the forward pass for the layer on the inputs tensor

Parameters

- **inputs** (*tensor*) – Input tensor on which the feature transforms are applied
- **training** (*boolean*) – Boolean flag indicating if the layer is being used in training mode or not

Returns Resulting tensor after the forward pass through the feature transform layer

Return type tf.Tensor

3.4.8.3 Tensorflow Native Operations

class ml4ir.base.features.feature_fns.tf_native.**TFNativeOpLayer**(*feature_info: dict*, *file_io: ml4ir.base.io.file_io.FileIO*, ***kwargs*)

Bases: ml4ir.base.features.feature_fns.base.BaseFeatureLayerOp

Run a series of tensorflow native operations on the input feature tensor. The functions will be applied in the order they are specified.

Initialize the feature layer

Parameters

- **feature_info** (*dict*) – Dictionary representing the feature_config for the input feature
- **file_io** (*FileIO object*) – FileIO handler object for reading and writing

Notes

Args under feature_layer_info:

ops: list of dict List of function specifications with associated arguments

Arguments under ops:

fn [str] Tensorflow native function name. Should start with tf. Example: tf.math.log or tf.clip_by_value

args [dict] Keyword arguments to be passed to the tensorflow function

LAYER_NAME = 'tf_native_op'

OPS = 'ops'

call(*inputs*, *training=None*)

Defines the forward pass for the layer on the inputs tensor

Parameters

- **inputs** (*tensor*) – Input tensor on which the feature transforms are applied
- **training** (*boolean*) – Boolean flag indicating if the layer is being used in training mode or not

Returns Resulting tensor after the forward pass through the feature transform layer

Return type tf.Tensor

3.4.9 Interaction Model

3.4.9.1 InteractionModel

class ml4ir.base.model.scoring.interaction_model.**InteractionModel**(*feature_config: ml4ir.base.features.feature_config.Fe tfrecord_type: str, feature_layer_keys_to_fns: dict = {}, max_sequence_size: int = 0, file_io: ml4ir.base.io.file_io.FileIO = None, **kwargs*)

Bases: keras.engine.training.Model

InteractionModel class that defines tensorflow layers that act on input features to convert them into numeric features to be fed into further neural network layers

Constructor for instantiating a base InteractionModel

Parameters

- **feature_config** (*FeatureConfig* object) – FeatureConfig object that defines list of model features and the feature transformation functions to be used on each
- **tfrecord_type** (*{"example", "sequence_example"}*) – Type of TFRecord protobuf being used for model training
- **feature_layer_keys_to_fns** (*dict*) – Dictionary of custom feature transformation functions to be applied on the input features
- **max_sequence_size** (*int, optional*) – Maximum size of the sequence in SequenceExample protobuf
- **file_io** (*FileIO object*) – *FileIO* object that handles read write operations

3.4.9.2 UnivariateInteractionModel

class ml4ir.base.model.scoring.interaction_model.**UnivariateInteractionModel**(*feature_config: ml4ir.base.features.fe tfrecord_type: str, feature_layer_keys_to_f dict = {}, max_sequence_size: int = 0, file_io: ml4ir.base.io.file_io.F = None, **kwargs*)

Bases: *ml4ir.base.model.scoring.interaction_model.InteractionModel*

Keras layer that applies in-graph transformations to input feature tensors

Constructor for instantiating a UnivariateInteractionModel

Parameters

- **feature_config** (*FeatureConfig* object) – FeatureConfig object that defines list of model features and the feature transformation functions to be used on each
- **tfrecord_type** (*{"example", "sequence_example"}*) – Type of TFRecord protobuf being used for model training
- **feature_layer_keys_to_fns** (*dict*) – Dictionary of custom feature transformation functions to be applied on the input features
- **max_sequence_size** (*int, optional*) – Maximum size of the sequence in SequenceExample protobuf
- **file_io** (*FileIO object*) – *FileIO* object that handles read write operations

call(*inputs*, *training=None*)

Apply the feature transform op to each feature

Parameters

- **inputs** (*dict of tensors*) – List of tensors that can be found in the FeatureConfig key-d with their node_name
- **training** (*boolean*) – Boolean specifying if the layer is used in training mode or not

Returns

train: dict of tensors List of transformed features that are used for training

metadata: dict of tensors List of transformed features that are used as metadata

Return type dict of dict of tensors

3.4.9.3 feature_layer

class ml4ir.base.features.feature_layer.**FeatureLayerMap**

Bases: object

Class defining mapping from keys to feature layer functions

Define ml4ir's predefined feature transformation functions

add_fn(*key*, *fn*)

Add custom new function to the FeatureLayerMap

Parameters

- **key** (*str*) – name of the feature transformation function
- **fn** (*tf.function*) – tensorflow function that transforms input features

add_fns(*keys_to_fns_dict*)

Add custom new functions to the FeatureLayerMap

Parameters **keykeys_to_fns_dict** (*dict*) – Dictionary with name and definition of custom tensorflow functions that transform input features

get_fns()

Get all feature transformation functions

Returns Dictionary of feature transformation functions

Return type dict

get_fn(*key*)

Get feature transformation function using the key

Parameters **key** (*str*) – Name of the feature transformation function to be fetched

Returns Feature transformation function

Return type tf.function

pop_fn(*key*)

Get feature transformation function using the key and remove from FeatureLayerMap

Parameters **key** (*str*) – Name of the feature transformation function to be fetched

Returns Feature transformation function

Return type tf.function

3.4.10 Scorer

3.4.10.1 ScorerBase

3.4.10.2 RelevanceScorer

class ml4ir.base.model.scoring.scoring_model.**RelevanceScorer**(*model_config: dict*, *feature_config: ml4ir.base.features.feature_config.FeatureC* *interaction_model: ml4ir.base.model.scoring.interaction_model* *loss: ml4ir.base.model.losses.loss_base.Relevanc* *file_io: ml4ir.base.io.file_io.FileIO*, *aux_loss: Optional[ml4ir.base.model.losses.loss_base.Re* = *None*, *aux_loss_weight: float* = *0.0*, *aux_metrics: Optional[List[Union[keras.metrics.base_metri* *str]]]* = *None*, *output_name: str* = *'score'*, *logger: Optional[logging.Logger]* = *None*, *logs_dir: Optional[str]* = *''*, ***kwargs*)

Bases: keras.engine.training.Model

Base Scorer class that defines the neural network layers that convert the input features into scores

Defines the feature transformation layer(InteractionModel), dense neural network layers combined with activation layers and the loss function

Notes

- This is a Keras model subclass and is built recursively using keras Layer instances
- This is an abstract class. In order to use a Scorer, one must define and override the *architecture_op* and the *final_activation_op* functions

Constructor method for creating a RelevanceScorer object

Parameters

- **model_config** (*dict*) – Dictionary defining the model layer configuration
- **feature_config** (*FeatureConfig* object) – FeatureConfig object defining the features and their configurations
- **interaction_model** (*InteractionModel* object) – InteractionModel that defines the feature transformation layers on the input model features

- **loss** (*RelevanceLossBase* object) – Relevance loss object that defines the final activation layer and the loss function for the model
- **file_io** (*FileIO* object) – FileIO object that handles read and write
- **aux_loss** (*RelevanceLossBase* object) – Auxiliary loss to be used in conjunction with the primary loss
- **aux_loss_weight** (*float*) – Floating point number in [0, 1] to indicate the proportion of the auxiliary loss in the total final loss value computed using a linear combination total loss = (1 - aux_loss_weight) * loss + aux_loss_weight * aux_loss
- **aux_metrics** (*List of keras.metrics.Metric*) – Keras metric list to be computed on the aux label
- **output_name** (*str, optional*) – Name of the output that captures the score computed by the model
- **logger** (*Logger, optional*) – Logging handler
- **logs_dir** (*str, optional*) – Path to the logging directory

Notes

logs_dir [Used to point model architectures to local logging directory,] primarily for saving visualizations.

classmethod from_model_config_file(*model_config_file: str*, *interaction_model: ml4ir.base.model.scoring.interaction_model.InteractionModel*, *loss: ml4ir.base.model.losses.loss_base.RelevanceLossBase*, *file_io: ml4ir.base.io.file_io.FileIO*, *aux_loss: Optional[ml4ir.base.model.losses.loss_base.RelevanceLossBase] = None*, *aux_loss_weight: float = 0.0*, *output_name: str = 'score'*, *feature_config: Optional[ml4ir.base.features.feature_config.FeatureConfig] = None*, *logger: Optional[logging.Logger] = None*, ***kwargs*)

Get a Scorer object from a YAML model config file

Parameters

- **model_config_file** (*str*) – Path to YAML file defining the model layer configuration
- **feature_config** (*FeatureConfig* object) – FeatureConfig object defining the features and their configurations
- **interaction_model** (*InteractionModel* object) – InteractionModel that defines the feature transformation layers on the input model features
- **loss** (*RelevanceLossBase* object) – Relevance loss object that defines the final activation layer and the loss function for the model
- **file_io** (*FileIO* object) – FileIO object that handles read and write
- **aux_loss** (*RelevanceLossBase* object) – Auxiliary loss to be used in conjunction with the primary loss
- **aux_loss_weight** (*float*) – Floating point number in [0, 1] to indicate the proportion of the auxiliary loss in the total final loss value computed using a linear combination total loss = (1 - aux_loss_weight) * loss + aux_loss_weight * aux_loss

- **output_name** (*str, optional*) – Name of the output that captures the score computed by the model
- **logger** (*Logger, optional*) – Logging handler

Returns RelevanceScorer object that computes the scores from the input features of the model

Return type *RelevanceScorer* object

plot_abstract_model()
Visualize the model architecture if defined by the architecture op

call(*inputs: Dict[str, tensorflow.python.framework.ops.Tensor], training=None*)
Compute score from input features

Parameters **inputs** (*dict of tensors*) – Dictionary of input feature tensors

Returns **scores** – Tensor object of the score computed by the model

Return type dict of tensor object

get_architecture_op()
Get the model architecture instance based on the configs

compile(***kwargs*)
Compile the keras model and defining a loss metrics to track any custom loss

train_step(*data*)
Defines the operations performed within a single training step. Called implicitly by tensorflow-keras when using model.fit()

Parameters **data** (*tuple of tensor objects*) – Tuple of features and corresponding labels to be used to learn the model weights

Returns Dictionary of metrics and loss computed for this training step

Return type dict

test_step(*data*)
Defines the operations performed within a single prediction or evaluation step. Called implicitly by tensorflow-keras when using model.predict() or model.evaluate()

Parameters **data** (*tuple of tensor objects*) – Tuple of features and corresponding labels to be used to evaluate the model

Returns Dictionary of metrics and loss computed for this evaluation step

Return type dict

metrics
Get the metrics for the keras model along with the custom loss metric

3.4.11 File I/O Utilities

3.4.11.1 FileIO

class ml4ir.base.io.file_io.**FileIO**(*logger: Optional[logging.Logger] = None*)
Bases: object

Abstract class defining the file I/O handler methods

Constructor method to create a FileIO handler object

Parameters **logger** (*Logger* object, optional) – logging handler object to instantiate FileIO object with the ability to log progress updates

set_logger(*logger: Optional[logging.Logger] = None*)
Setter method to assign a logging handler to the FileIO object

Parameters **logger** (*Logger* object, optional) – logging handler object to be used with the FileIO object to log progress updates

log(*string*, *mode=20*)
Write specified string with preset logging object using the mode specified

Parameters

- **string** (*str*) – string text to be logged
- **mode** (*int, optional*) – One of the supported logging message types. Currently supported values are logging.INFO, DEBUG, ERROR

make_directory(*dir_path: str*, *clear_dir: bool = False*) → str
Create directory structure specified recursively

Parameters

- **dir_path** (*str*) – path for directory to be create
- **clear_dir** (*bool, optional*) – clear contents on existing directory

Returns path to the directory created

Return type str

read_df(*infile: str*, *sep: str = ', '*, *index_col: int = None*, ***kwargs*) → Optional[pandas.core.frame.DataFrame]
Load a pandas dataframe from a file

Parameters

- **infile** (*str*) – path to the csv input file
- **sep** (*str, optional*) – separator to use for loading file
- **index_col** (*int, optional*) – column to be used as index

Returns pandas dataframe loaded from specified path

Return type *pandas.DataFrame*

read_df_list(*infiles*, *sep=', '*, *index_col=None*, ***kwargs*) → pandas.core.frame.DataFrame
Load a pandas dataframe from a list of files by concatenating the individual dataframes from each file

Parameters

- **infiles** (*list of str*) – list of paths to the csv input files
- **sep** (*str, optional*) – separator to use for loading file
- **index_col** (*int, optional*) – column to be used as index
- **Returns** –
- **pandas.DataFrame** – pandas dataframe loaded from specified path

write_df(*df*, *outfile: str = None*, *sep: str = ', '*, *index: bool = True*)
Write a pandas dataframe to a file

Parameters

- **df** (*pandas.DataFrame*) – dataframe to be written

- **outfile** (*str. optional*) – path to the csv output file
- **sep** (*str, optional*) – separator to use for loading file
- **index** (*bool, optional*) – boolean specifying if index should be saved

read_text_file (*infile*) → str
Read text file and return as string

Parameters **infile** (*str*) – path to the text file

Returns file contents as a string

Return type str

read_json (*infile*) → dict
Read JSON file and return a python dictionary

Parameters **infile** (*str*) – path to the json file

Returns python dictionary loaded from JSON file

Return type dict

read_yaml (*infile*) → dict
Read YAML file and return a python dictionary

Parameters **infile** (*str*) – path to the YAML file

Returns python dictionary loaded from JSON file

Return type dict

write_json (*json_dict: dict*, *outfile: str*)
Write dictionary to a JSON file

Parameters

- **json_dict** (*dict*) – dictionary to be dumped to json file
- **outfile** (*str*) – path to the output file

path_exists (*path: str*) → bool
Check if a file path exists

Parameters **path** (*str*) – check if path exists

Returns True if path exists; False otherwise

Return type bool

get_files_in_directory (*indir: str*, *extension='.csv'*, *prefix=''*)
Get list of files in a directory

Parameters

- **indir** (*str*) – input directory to search for files
- **extension** (*str, optional*) – extension of the files to search for
- **prefix** (*str, optional*) – string file name prefix to narrow search

Returns list of file path strings

Return type list of str

clear_dir (*dir_path: str*)
Clear contents of existing directory

Parameters **dir_path** (*str*) – path to directory to be cleared

rm_dir (*dir_path: str*)
Delete existing directory

Parameters **dir_path** (*str*) – path to directory to be removed

rm_file (*file_path: str*)
Deletes existing file_path

Parameters **file_path** (*str*) – path to file to be removed

3.4.11.2 LocalIO

class ml4ir.base.io.local_io.**LocalIO**(*logger: Optional[logging.Logger] = None*)
Bases: *ml4ir.base.io.file_io.FileIO*

Class defining the file I/O handler methods for the local file system

Constructor method to create a FileIO handler object

Parameters **logger** (*Logger* object, optional) – logging handler object to instantiate FileIO object with the ability to log progress updates

make_directory (*dir_path: str*, *clear_dir: bool = False*)
Create directory structure specified recursively

Parameters

- **dir_path** (*str*) – path for directory to be create
- **clear_dir** (*bool, optional*) – clear contents on existing directory

read_df (*infile: str*, *sep: str = ','*, *index_col: int = None*, ***kwargs*) → Optional[pandas.core.frame.DataFrame]
Load a pandas dataframe from a file

Parameters

- **infile** (*str*) – path to the csv input file; can be hdfs path
- **sep** (*str, optional*) – separator to use for loading file
- **index_col** (*int, optional*) – column to be used as index

Returns pandas dataframe loaded from file

Return type *pandas.DataFrame*

read_df_list (*infiles*, *sep=','*, *index_col=None*, ***kwargs*) → pandas.core.frame.DataFrame
Load a pandas dataframe from a list of files

Parameters

- **infiles** (*list of str*) – paths to the csv input files; can be hdfs paths
- **sep** (*str, optional*) – separator to use for loading file
- **index_col** (*int, optional*) – column to be used as index

Returns pandas dataframe loaded from file

Return type *pd.DataFrame*

write_df (*df*, *outfile: str = None*, *sep: str = ','*, *index: bool = True*) → str
Write a pandas dataframe to a file

Parameters

- **df** (*pandas.DataFrame*) – dataframe to be written
- **outfile** (*str*) – path to the csv output file; can NOT be hdfs path currently
- **sep** (*str*) – separator to use for loading file
- **index** (*int*) – boolean specifying if index should be saved

Returns dataframe in csv form if outfile is None

Return type str

read_json(*infile*) → dict
Read JSON file and return a python dictionary

Parameters **infile** (*str*) – path to the json file; can be hdfs path

Returns python dictionary loaded from file

Return type dict

read_yaml(*infile*) → dict
Read YAML file and return a python dictionary

Parameters **infile** (*str*) – path to the json file; can be hdfs path

Returns python dictionary loaded from file

Return type dict

write_json(*json_dict: dict*, *outfile: str*)
Write dictionary to a JSON file

Parameters

- **json_dict** (*dict*) – dictionary to be dumped to json file
- **outfile** (*str*) – path to the output file

path_exists(*path: str*) → bool
Check if a path exists

Parameters **path** (*str*) – check if path exists

Returns True if path exists; False otherwise

Return type bool

get_files_in_directory(*indir: str*, *extension='.csv'*, *prefix=''*)
Get list of files in a directory

Parameters

- **indir** (*str*) – input directory to search for files
- **extension** (*str, optional*) – extension of the files to search for
- **prefix** (*str, optional*) – string file name prefix to narrow search

Returns list of file path strings

Return type list of str

clear_dir_contents(*dir_path: str*)
Clear contents of existing directory

Parameters **dir_path** (*str*) – path to directory to be cleared

rm_dir(*dir_path: str*)
Delete existing directory

Parameters **dir_path** (*str*) – path to directory to be removed

rm_file(*file_path: str*)
Delete existing file_path

Parameters **file_path** (*str*) – path to file to be removed

save_numpy_array(*np_array*, *file_path: str*, *allow_pickle=True*, *zip=True*, ***kwargs*)
Save a numpy array to disk

Parameters

- **np_array** (*numpy array or list of numpy arrays*) – Array like numpy object to be saved
- **file_path** (*str*) – file path to save the object to
- **allow_pickle** (*bool, optional*) – Allow pickling of objects while saving
- **zip** (*bool, optional,*) – use np.savez to save the numpy arrays, allows passing in python list

Notes

Used to save individual model layer weights for transfer learning.

If using zip=True, the np_array has to be a python list tensorflow layer weights are lists of arrays. np.save() can not be used for saving list of numpy arrays directly as it tries to manually convert the list into a numpy array, leading to errors with numpy shape. savez allows us to save each list item in separate files and abstracts this step for end user.

load_numpy_array(*file_path*, *allow_pickle=True*, *unzip=True*, ***kwargs*)
Load a numpy array from disk

Parameters

- **file_path** (*str*) – file path to load the numpy object from
- **allow_pickle** (*bool, optional*) – Allow pickling of objects while loading
- **unzip** (*bool, optional*) – To unzip the numpy array saved as a zip file. Used when saved with zip=True

Returns python list of numpy arrays

Return type list of numpy arrays

Notes

Used to load individual model layer weights for transfer learning

3.4.11.3 SparkIO

class ml4ir.base.io.spark_io.**SparkIO**(*logger: Optional[logging.Logger] = None*)
Bases: *ml4ir.base.io.file_io.FileIO*

Class defining the file I/O handler methods for the HDFS file system using spark

Constructor method to create a FileIO handler object and set up spark session and hadoop file system handlers

Parameters **logger** (*Logger* object, optional) – logging handler object to instantiate FileIO object with the ability to log progress updates

get_path_from_str(*file_path: str*)
Get Path object from string

Parameters **file_path** (*str*) – string file path

Returns Hadoop Path object

Return type hadoop path

read_df(*infile: str*, *sep: str = ', '*, *index_col: int = None*, ***kwargs*) → Optional[pandas.core.frame.DataFrame]
Load a pandas dataframe from a file

Parameters

- **infile** (*str*) – path to the csv input file; can be hdfs path
- **sep** (*str, optional*) – separator to use for loading file
- **index_col** (*int, optional*) – column to be used as index

Returns pandas dataframe loaded from file

Return type *pandas.DataFrame*

read_df_list(*infiles*, *sep=', '*, *index_col=None*, ***kwargs*) → pandas.core.frame.DataFrame
Load a pandas dataframe from a list of files

Parameters

- **infiles** (*list of str*) – paths to the csv input files; can be hdfs paths
- **sep** (*str, optional*) – separator to use for loading file
- **index_col** (*int, optional*) – column to be used as index

Returns pandas dataframe loaded from list of files

Return type *pandas.DataFrame*

Notes

sep and *index_col* are not used in SparkIO

read_text_file(*infile*) → str
Read text file and return as string

Parameters **infile** (*str*) – path to the text file

Returns file contents as a string

Return type str

read_json(*infile*) → dict
Read JSON file and return a python dictionary

Parameters **infile** (*str*) – path to the json file; can be hdfs path

Returns python dictionary loaded from file

Return type dict

read_yaml (*infile*) → dict
Read YAML file and return a python dictionary

Parameters **infile** (*str*) – path to the json file; can be hdfs path

Returns python dictionary loaded from file

Return type dict

path_exists (*path: str*) → bool
Check if a path exists

Parameters **path** (*str*) – check if path exists

Returns True if path exists; False otherwise

Return type bool

rm_dir (*dir_path: str*)
Delete existing directory

Parameters **dir_path** (*str*) – path to directory to be removed

rm_file (*file_path: str*)
Deletes existing file_path

Parameters **file_path** (*str*) – path to file to be removed

copy_from_hdfs (*src: str*, *dest: str*)
Copy a directory/file from HDFS to local filesystem

Parameters

- **src** (*str*) – String path to source(on HDFS)
- **dest** (*str*) – String path to destination(on local file system)

copy_to_hdfs (*src: str*, *dest: str*, *overwrite=True*)
Copy a directory/file to HDFS from local filesystem

Parameters src : str

String path to source(on local file system)

dest [str] String path to destination(on HDFS)

overwrite [bool, optional] Boolean to specify whether existing destination files should be overwritten

3.5 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

3.5.1 [0.1.16] - 2023-02-06

3.5.1.1 Added

- RankMatchFailure metric for evaluation
- Statistical significance and power analysis utilities

- Stat analysis for groupwise metrics in Ranking

3.5.2 [0.1.15] - 2023-01-20

3.5.2.1 Changed

- Upgrading from tensorflow 2.0.x to 2.9.x
- Moving from Keras Functional API to Model Subclassing API for more customization capabilities
- Auxiliary loss is reimplemented as part of ScoringModel

3.5.2.2 Added

- AutoDAGNetwork which allows for building flexible connected architectures using config files
- SetRankEncoder keras Layer to train SetRank like Ranking models
- Support for using tf-models-official deep learning garden library
- RankMatchFailure metric for validation

3.5.3 [0.1.14] - 2022-11-18

3.5.3.1 Changed

- Ability to pass custom RelevanceModel class in Pipeline.

3.5.4 [0.1.13] - 2022-10-17

3.5.4.1 Fixed

- Bug in metrics_helper when used without secondary_labels

3.5.4.2 Added

- RankMatchFailure metric for evaluation
- RankMatchFailure auxiliary loss

3.5.5 [0.1.12] - 2022-04-26

3.5.6 [0.1.11] - 2021-01-18

3.5.6.1 Changed

- Adding rank feature to serving parse fn by default and removing dependence on required serving_info attribute

3.5.7 [0.1.10] - 2021-12-29

3.5.7.1 Changed

- Adding all trained features to serving parse fn by default

3.5.8 [0.1.9] - 2021-11-29

3.5.8.1 Changed

- Refactored secondary label metrics computation for ranking and added unit tests
- Added NDCG metric for secondary labels

3.5.9 [0.1.8] - 2021-10-21

3.5.9.1 Added

- New argument to model.save()

3.5.10 [0.1.7] - 2021-09-30

3.5.10.1 Added

- SoftmaxCrossEntropy loss for ranking models

3.5.11 [0.1.6] - 2021-07-16

3.5.11.1 Fixed

- Fixing required arguments in setup.py

3.5.12 [0.1.5] - 2021-07-15

3.5.12.1 Added

- Adding support for performing post-training steps (such as copying data) by custom class inheriting RelevancePipeline.

3.5.13 [0.1.4] - 2021-06-30

3.5.13.1 Changed

- Performing pre-processing step in `__init__()` to be able to copy files before model_config and feature_config are initiated.

3.5.14 [0.1.3] - 2021-06-24

3.5.14.1 Changed

- Making pyspark an optional dependency to install ml4ir

3.5.15 [0.1.2] - 2021-06-16

3.5.15.1 Added

- Support for performing pre-processing steps (such as copying data) by custom class inheriting RelevancePipeline.

3.5.16 [0.1.1] - 2021-05-20

3.5.16.1 Added

- Support for using native tf/keras feature functions from the feature config YAML

3.5.17 [0.1.0] - 2021-03-01

3.5.17.1 Changed

- TFRecord format changed for SequenceExample to earlier implementation.
- Removed support for `max_len` attribute for SequenceExample features.
- No effective changes for Example TFRecords.
- TFRecord implementation on python (training) and jvm (inference) side are now in sync.

3.5.18 [0.0.5] - 2021-02-17

3.5.18.1 Added

- Changelog file to track version updates for ml4ir.
- `build-requirements.txt` with all python dependencies needed for developing on ml4ir and the CircleCI autobuilds.
- Updated CircleCI builds to use `build-requirements.txt`

3.5.18.2 Fixed

- Removed build requirements from the base ml4ir `requirements.txt` allowing us to keep the published whl file dependencies to be minimal.

3.6 License

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/
```

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such

license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You

may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

```
To apply the Apache License to your work, attach the following
boilerplate notice, with the fields enclosed by brackets "[]"
replaced with your own identifying information. (Don't include
the brackets!)  The text should be enclosed in the appropriate
comment syntax for the file format. We also recommend that a
file or class name and description of purpose be included on the
same "printed page" as the copyright notice for easier
identification within third-party archives.
```

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

```
http://www.apache.org/licenses/LICENSE-2.0
```

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

3.7 Contact Us

For any issues with ml4ir, please file an issue on github **here**

For further questions, please contact one of the following contributors:

- Ashish Bharadwaj Srinivasa - ashish.srinivasa@salesforce.com
- Jake Mannix - jmannix@salesforce.com

Python Module Index

m

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W