Janggu

Release 0.9.5

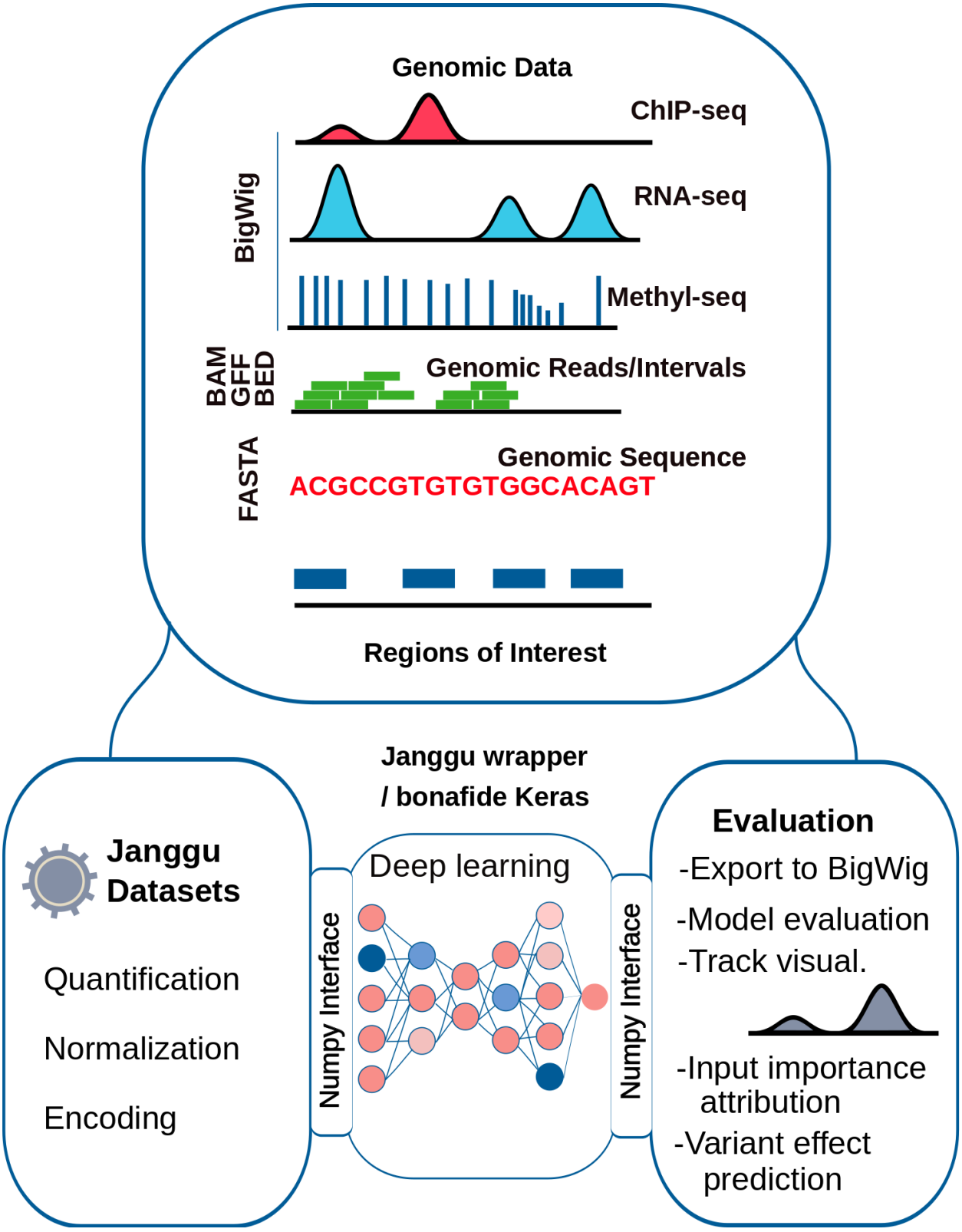
Nov 08, 2019

Contents

CHAPTER 1

Janggu - Deep learning for Genomics

Janggu is a python package that facilitates deep learning in the context of genomics. The package is freely available under a GPL-3.0 license.



In particular, the package allows for easy access to typical **Genomics data formats** and **out-of-the-box evaluation** so that you can concentrate on designing the neural network architecture for the purpose of quickly testing biological hypothesis. A comprehensive documentation is available here.

1.1 Hallmarks of Janggu:

1. Janggu provides special **Genomics datasets** that allow you to access raw data in FASTA, BAM, BIGWIG, BED and GFF file format.
2. Various **normalization** procedures are supported for dealing with of the genomics dataset, including 'TPM', 'zscore' or custom normalizers.
3. Biological features can be represented in terms of higher-order sequence features, e.g. di-nucleotide based features.
4. The dataset objects are directly consumable with neural networks for example implemented using keras or using scikit-learn (see src/examples in this repository).
5. Numpy format output of a keras model can be converted to represent genomic coverage tracks, which allows exporting the predictions as BIGWIG files and visualization of genome browser-like plots.
6. Genomic datasets can be stored in various ways, including as numpy array, sparse dataset or in hdf5 format.
7. Caching of Genomic datasets avoids time consuming preprocessing steps and facilitates fast reloading.
8. Janggu provides a wrapper for keras models with built-in logging functionality and automatized result evaluation.
9. Janggu supports input feature importance attribution using the integrated gradients method and variant effect prediction assessment.
10. Janggu provides a utilities such as keras layer for scanning both DNA strands for motif occurrences.

1.2 Why the name Janggu?

Janggu is a Korean percussion instrument that looks like an hourglass.

Like the two ends of the instrument, the philosophy of the Janggu package is to help with the two ends of a deep learning application in genomics, namely data acquisition and evaluation.

1.2.1 Installation

A list of python dependencies is defined in *setup.py*. Additionally, bedtools is required for *pybedtools* which *janggu* depends on.

The simplest way to install janggu is via the conda package management system. Assuming you have already installed conda, create a new environment and type

```
pip install janggu
```

The janggu neural network model depends on tensorflow which you have to install depending on whether you want to use GPU support or CPU only. To install tensorflow type

```
conda install tensorflow  # or tensorflow-gpu
```

Further information regarding the installation of tensorflow can be found on the official tensorflow webpage

To verify that the installation works try to run the example contained in the janggu package as follows

```
git clone https://github.com/BIMSBbioinfo/janggu
cd janggu
python ./src/examples/classify_fasta.py single
```

A model is then trained to predict the class labels of two sets of toy sequencesby scanning the forward strand for sequence patterns and using an ordinary mono-nucleotide one-hot sequence encoding. The entire training process takes a few minutes on CPU backend. Eventually, some example prediction scores are shown for Oct4 and Mafk sequences. The accuracy should be around 85% and individual example prediction scores should tend to be higher for Oct4 than for Mafk.

You may also try to rerun the training by evaluating sequences features on both strands and using higher-order sequence encoding using i.e. the command-line arguments: *dnaconv -order 2*. Accuracies and prediction scores for the individual example sequences should improve compared to the previous example.

A range of additional examples can be found in './src/examples' including some jupyter notebooks or by following the tutorial.

CHAPTER 2

Tutorial

This tutorial is split in three parts.

Part I treats the Genomic Dataset that are available through Janggu which can be directly consumed by your keras model. The tutorial illustates how to access genomics data from various widely used file formats, including FASTA, BAM, BIGWIG, BED and GFF for the purpose of using them as input to a deep learning application. It illustrates a range of parameters to adapt the read out of genomics data and it shows how the predictions or feature activities of a neural network in numpy format can be converted to a genomic coverage representation that can in turn be exported to BIGWIG file format or visualized directly via a genome browser-like plot.

Part II treats utilities for defining a neural networks based on keras.

Part III illustrates Janggu's evaluation utilities.

Complementary to this tutorial, the janggu repository contains a number of jupyter notebooks that illustrate for example with keras or sklearn:

Example notebooks
keras cnn example
sklearn example
janggu example I
janggu example II
reusing datasets with view
randomizing HDF5 data
variant effect prediction
plotting genome coverage

Furthermore, use cases for predicting JunD binding, training and adapting published genomics models as well as a regression model example are demonstrated in the supplementary repository: Janggu use cases

2.1 Part I) Introduction to Genomic Datasets

Genomic Datasets

Most of the parameters are consistent across `Bioseq` and `Cover`.

`janggu.data` provides Dataset classes that can be used for training and evaluating neural networks. Of particular importance are the Genomics-specific dataset, `Bioseq` and `Cover` which allow easy access to genomics data, including DNA sequences or coverage information. Apart from accessing raw genomics data, `Janggu` also facilitates a method for converting an ordinary numpy array (e.g. predictions obtained from a neural net) to a `Cover` object. This enables the user to export the predictions as BIGWIG format or interactively plot genome browser tracks. In this tutorial, we demonstrate some of the key functionality of Janggu. Further details are available in *Genomic Datasets* and *API*.

2.1.1 Bioseq

The `Bioseq` can be used to load nucleotide or protein sequence data from fasta files or from a reference genome along with a set of genomic coordinates defining the region of interest (ROI). The class facilitates access the *one-hot encoding* representation of the sequences. Specifically, the *one-hot encoding* is represented as a 4D array with dimensions corresponding to `(region, region_length, 1, alphabet_size)`. The Bioseq offers a number of features:

1. Strand-specific sequence extraction (if DNA sequences are extracted from the reference genome)
2. Higher-order one-hot encoding, e.g. di-nucleotide based

Sequences can be loaded in two ways: using `Bioseq.create_from_seq` or `Bioseq.create_from_refgenome`. The former constructor method can be used to load DNA or protein sequences from fasta files directly or from as list of `Bio.SeqRecord.SeqRecord` entries. An example is shown below:

```
from pkg_resources import resource_filename
from janggu.data import Bioseq

fasta_file = resource_filename('janggu',
                               'resources/sample.fa')

dna = Bioseq.create_from_seq(name='dna',
                             fastafile=fasta_file)

# there are 3897 sequences in the in sample.fa
len(dna)

# Each sequence is 200 bp of length
dna.shape  # is (3897, 200, 1, 4)

# One-hot encoding for the first 10 bases of the first region
dna[0][0, :10, 0, :]
```

Furthermore, it is possible to trim variable sequence length using the `fixedlen` option. If specfied, all sequences will be truncated or zero-padded to length *fixedlen*. For example,

```
dna = Bioseq.create_from_seq(name='dna',
                             fastafile=fasta_file,
                             fixedlen=205)

# Each sequence is 205 bp of length
dna.shape  # is (3897, 205, 1, 4)
```

(continues on next page)

(continued from previous page)

```
# the last 5 position were zero padded
dna[0][0, -6:, 0, :]
```

Alternatively, nucleotide sequences can be obtained from a reference genome directly along with a BED or GFF file that indicates the region of interest (ROI).

If each interval in the BED-file already corresponds to a 'datapoint' that shall be consumed during training, like it is the case for 'sample_equalsize.bed', the associated DNA sequences can be loaded according to

```
roi = resource_filename('janggu',
                        'resources/sample_equalsize.bed')
refgenome = resource_filename('janggu',
                              'resources/sample_genome.fa')

dna = Bioseq.create_from_refgenome(name='dna',
                                   refgenome=refgenome,
                                   roi=roi)

dna.shape  # is (4, 200, 1, 4)
# One-hot encoding of the first 10 nucleotides in region 0
dna[0][0, :10, 0, :]
```

Sometimes it is more convenient to provide the ROI as a set of variable-sized broad intervals (e.g. chr1:10000-50000 and chr3:4000-8000) which should be divided into sub-intervals of equal length (e.g. of length 200 bp). This can be achieved by explicitly specifying a desired `binsize` and `stepsize` as shown below:

```
roi = resource_filename('janggu',
                        'resources/sample.bed')

# loading non-overlapping intervals
dna = Bioseq.create_from_refgenome(name='dna',
                                   refgenome=refgenome,
                                   roi=roi,
                                   binsize=200,
                                   stepsize=200)

dna.shape  # is (100, 200, 1, 4)

# loading mutually overlapping intervals
dna = Bioseq.create_from_refgenome(name='dna',
                                   refgenome=refgenome,
                                   roi=roi,
                                   binsize=200,
                                   stepsize=50)

dna.shape  # is (394, 200, 1, 4)
```

The argument `flank` can be used to extend the intervals up and downstream by a given length

```
dna = Bioseq.create_from_refgenome(name='dna',
                                   refgenome=refgenome,
                                   roi=roi,
                                   binsize=200,
                                   stepsize=200,
                                   flank=100)
```

(continues on next page)

(continued from previous page)

```
dna.shape  # is (100, 400, 1, 4)
```

Finally, sequences can be represented using **higher-order** one-hot representation using the `order` argument. An example of a di-nucleotide-based one-hot representation is shown below

```
dna = Bioseq.create_from_refgenome(name='dna',
                                   refgenome=refgenome,
                                   roi=roi,
                                   binsize=200,
                                   stepsize=200,
                                   order=2)

# is (100, 199, 1, 16)
# that is the last dimension represents di-nucleotides
dna.shape
```

2.1.2 Cover

`Cover` can be utilized to fetch different kinds of coverage data from commonly used data formats, including BAM, BIGWIG, BED and GFF. Coverage data is stored as a 4D array with dimensions corresponding to `(region, region_length, strand, condition)`.

The following examples illustrate some use cases for `Cover`, including loading, normalizing coverage data. Additional features are described in the *API*.

Loading read count coverage from BAM files is supported for single-end and paired-end alignments. For the single-end case reads are counted on the 5'-end and and for paired-end alignments, reads are optionally counted at the mid-points or 5' ends of the first mate. The following example illustrate how to extract base-pair resolution coverage with and without strandedness.

```
from janggu.data import Cover

bam_file = resource_filename('janggu',
                             'resources/sample.bam')

roi = resource_filename('janggu',
                        'resources/sample.bed')

cover = Cover.create_from_bam('read_count_coverage',
                              bamfiles=bam_file,
                              binsize=200,
                              stepsize=200,
                              roi=roi)

cover.shape  # is (100, 200, 2, 1)
cover[0]  # coverage of the first region

# Coverage regardless of read strandedness
# sums reads from both strand.
cover = Cover.create_from_bam('read_coverage',
                              bamfiles=bam_file,
                              binsize=200,
                              stepsize=200,
                              stranded=False,
```

(continues on next page)

(continued from previous page)

```
                                roi=roi)

cover.shape  # is (100, 200, 1, 1)
```

Sometimes it is desirable to determine the read count coverage in say 50 bp bins which can be controlled by the `resolution` argument. Consequently, note that the second dimension amounts to length 4 using *binsize=200* and *resolution=50* in the following example

```
# example with resolution=200 bp
cover = Cover.create_from_bam('read_coverage',
                                bamfiles=bam_file,
                                binsize=200,
                                resolution=50,
                                roi=roi)

cover.shape  # is (100, 4, 2, 1)
```

It might be desired to aggregate reads across entire interval rather than binning the genome to equally sized bins of length `resolution`. An example application for this would be to count reads per possibly variable-size regions (e.g. genes). This can be achived by `resolution=None` which results in the second dimension being collapsed to a length of one.

```
# example with resolution=None
cover = Cover.create_from_bam('read_coverage',
                                bamfiles=bam_file,
                                binsize=200,
                                resolution=None,
                                roi=roi)

cover.shape  # is (100, 1, 2, 1)
```

Similarly, if strandedness is not relevant we may use

```
# example with resolution=None without strandedness
cover = Cover.create_from_bam('read_coverage',
                                bamfiles=bam_file,
                                binsize=200,
                                resolution=None,
                                stranded=False,
                                roi=roi)

cover.shape  # is (100, 1, 1, 1)
```

Finally, it is possible to normalize the coverage profile, e.g. to account for differences in sequencing depth across experiments using the `normalizer` argument

```
# example with resolution=None without strandedness
cover = Cover.create_from_bam('read_coverage',
                                bamfiles=bam_file,
                                binsize=200,
                                resolution=None,
                                stranded=False,
                                normalizer='tpm',
                                roi=roi)

cover.shape  # is (100, 1, 1, 1)
```

More details on alternative normalization options are discussed in *Genomic Datasets*.

Loading signal coverage from BIGWIG files can be achieved analogously:

```
roi = resource_filename('janggu',
                        'resources/sample.bed')
bw_file = resource_filename('janggu',
                            'resources/sample.bw')

cover = Cover.create_from_bigwig('bigwig_coverage',
                                 bigwigfiles=bw_file,
                                 roi=roi,
                                 binsize=200,
                                 stepsize=200)

cover.shape  # is (100, 200, 1, 1)
```

When applying signal aggregation using e.g `resolution=50` or `resolution=None`, additionally, the aggregation method can be specified using the `collapser` argument. For example, in order to represent the resolution sized bin by its mean signal the following snippet may be used:

```
cover = Cover.create_from_bigwig('bigwig_coverage',
                                 bigwigfiles=bw_file,
                                 roi=roi,
                                 binsize=200,
                                 resolution=None,
                                 collapser='mean')

cover.shape  # is (100, 1, 1, 1)
```

More details on alternative collapse options are discussed in *Genomic Datasets*.

Coverage from a BED files is largely analogous to extracting coverage information from BAM or BIGWIG files, but in addition it is possible to interpret BED files in various ways:

1. **Binary** or Presence/Absence mode interprets the ROI as the union of positive and negative cases in a binary classification setting and regions contained in `bedfiles` as positive examples.
2. **Score** mode reads out the real-valued score field value from the associated regions.
3. **Categorical** mode transforms integer-valued scores into one-hot representation. For that option, each of the *nclasses* corresponds to an integer ranging from zero to `nclasses - 1`.

Examples of loading data from a BED file are shown below

```
roi = resource_filename('janggu',
                        'resources/sample.bed')
score_file = resource_filename('janggu',
                               'resources/scored_sample.bed')

# binary mode (default)
cover = Cover.create_from_bed('binary_coverage',
                              bedfiles=score_file,
                              roi=roi,
                              binsize=200,
                              stepsize=200,
                              collapser='max',
                              resolution=None)

cover.shape  # is (100, 1, 1, 1)
```

(continues on next page)

(continued from previous page)

```
cover[4]  # contains [[[[1.]]]]

# score mode
cover = Cover.create_from_bed('score_coverage',
                              bedfiles=score_file,
                              roi=roi,
                              binsize=200,
                              stepsize=200,
                              resolution=None,
                              collapser='max',
                              mode='score')

cover.shape  # is (100, 1, 1, 1)
cover[4]  # contains the score [[[[5.]]]]

# categorical mode
cover = Cover.create_from_bed('cat_coverage',
                              bedfiles=score_file,
                              roi=roi,
                              binsize=200,
                              stepsize=200,
                              resolution=None,
                              collapser='max',
                              mode='categorical')

cover.shape  # is (100, 1, 1, 6)
cover[4]  # contains [[[[0., 0., 0., 0., 0., 1.]]]]
```

2.1.3 Dataset wrappers

In addition to the core datset `Bioseq` and `Cover`, Janggu offers convenience wrappers to transform them in various ways. For instance, `ReduceDim` can be used to convert a 4D coverage dataset into 2D table like object. That is it may be used to transform the dimensions `(region, region_length, strand, condition)` to `(region, condition)` by aggregating over the middle two dimensions.

```
from janggu.data import ReduceDim

data = ReduceDim(cover, aggregator='sum')
```

Other dataset wrappers can be used in order to perform data augmentation, including `RandomSignalScale` and `RandomOrientation` which can be used to randomly alter the signal intensity during model fitting and randomly flipping the 5' to 3' orientations of the coverage signal.

For more specialized cases, these wrappers might also be a good starting point to derive or adapt from.

2.1.4 Using the Genomic Datasets with keras or sklearn

The above mentioned datasets `Bioseq` and `Cover` are directly compatible with keras and sklearn models. An illustration of a simple convolutional neural network with keras is shown in keras cnn example. Moreover, an example of a logistic regression model from sklearn used with Janggu is shown in sklearn example.

2.1.5 Converting a Numpy array to `Cover`

After having trained and performed predictions with a model, the data is represented as numpy array. A convenient way to reassociate the predictions with the genomic coordinates they correspond to is achieved using `create_from_array`.

```
import numpy as np

# True labels may be obtained from a BED file
cover = Cover.create_from_bigwig('cov',
                                 bigwigfiles=bw_file,
                                 roi=roi,
                                 binsize=200,
                                 resolution=50)


# Let's pretend to have derived predictions from a NN
# of the same shape
predictions = np.random.randn(*cover.shape)*.1 + cover[:]

# We can reassociate the predictions with the genomic coordinates
# of a :code:`GenomicIndexer` (in this case, cover.gindexer).
predictions = Cover.create_from_array('predictions',
                                      predictions, cover.gindexer)
```

2.1.6 Exporting and visualizing `Cover`

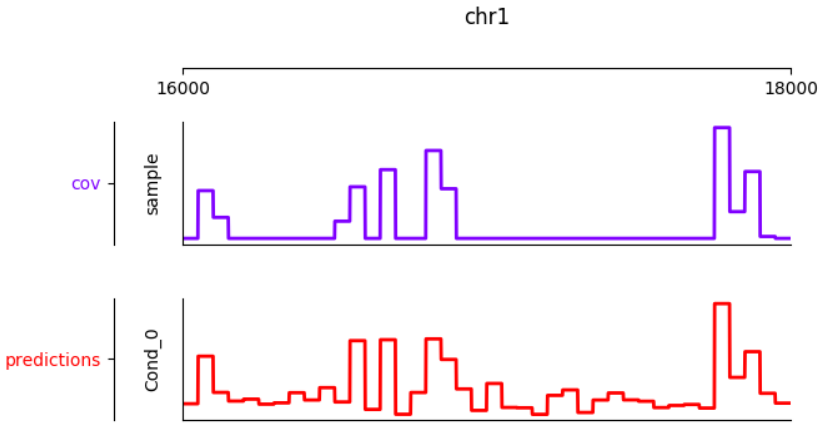
After having converted the predictions or feature activities of a neural network to a `Cover` object, it is possible to export the results as BIGWIG format for further investigation in a genome browser of your choice

```
# writes the predictions to a specified folder
predictions.export_to_bigwig(output_dir = './')
```

which should result in a file 'predictions.Cond_0.bigwig'.

Furthermore, it is possible to visualize the tracks interactively

```
from janggu.data import LineTrack
from janggu.data import plotGenomeTrack

fig = plotGenomeTrack([LineTrack(cover), LineTrack(predictions)], 'chr1', 16000,␣
↪18000).figsave('coverage.png')
```



2.2 Part II) Building a neural network with Janggu

While the Genomic Dataset may be used directly with keras, this part of the tutorial discusses the `Janggu` wrapper class for a keras model. It offers the following features:

1. Building models using automatic input and output layer shape inference
2. Built-in logging functionality
3. Automatic evaluation through the attachment of Scorer callbacks

A list of examples can be found in the *Table* at the beginning.

Datasets are named

Dataset names must match with the Input and Output layers of the neural network.

A neural network can be created by instantiating a `Janggu` object. There are two ways of achieving this:

1. Similar as with *keras.models.Model*, a `Janggu` object can be created from a set of native keras Input and Output layers, respectively.
2. Janggu offers a *Janggu.create* constructor method which helps to reduce redundant code when defining many rather similar models.

2.2.1 Example 1: Instantiate Janggu similar to keras.models.Model

Model name

Model results, e.g. trained parameters, are automatically stored with the associated model name. To simplify the determination of a unique name for the model, Janggu automatically derives the model name based on a md5-hash of the network configuration. However, you can also specify a name yourself.

```
from keras.layers import Input
from keras.layers import Dense

from janggu import Janggu

# Define neural network layers using keras
in_ = Input(shape=(10,), name='ip')
layer = Dense(3)(in_)
output = Dense(1, activation='sigmoid',
               name='out')(layer)

# Instantiate model name.
model = Janggu(inputs=in_, outputs=output)
model.summary()
```

2.2.2 Example 2: Specify a model using a model template function

As an alternative to the above stated variant, it is also possible to specify a network via a python function as in the following example

```
def model_template(inputs, inp, oup, params):
    inputs = Input(shape=(10,), name='ip')
    layer = Dense(params)(inputs)
    output = Dense(1, activation='sigmoid',
                   name='out')(layer)
    return inputs, output

# Defines the same model by invoking the definition function
# and the create constructor.
model = Janggu.create(template=model_template,
                      modelparams=3)
```

The model template function must adhere to the signature `template(inputs, inp, oup, params)`. Notice, that `modelparams=3` gets passed on to `params` upon model creation. This allows to parametrize the network and reduces code redundancy.

From the model template it is also possible to obtain a keras model directly, rather than the Janggu model wrapper if this is preferred

```
from janggu import create_model

# This will construct a keras model directly
model = create_model(template=model_template,
                     modelparams=3)
```

2.2.3 Example 3: Automatic Input and Output layer extension

A second benefit to invoke `Janggu.create` is that it can automatically determine and append appropriate Input and Output layers to the network. This means, only the network body remains to be defined.

```
import numpy as np
from janggu import inputlayer, outputdense
from janggu.data import Array

```

(continues on next page)

(continued from previous page)

```
# Some random data
DATA = Array('ip', np.random.random((1000, 10)))
LABELS = Array('out', np.random.randint(2, size=(1000, 1)))

# inputlayer and outputdense automatically
# extract dataset shapes and extend the
# Input and Output layers appropriately.
# That is, only the model body needs to be specified.
@inputlayer
@outputdense('sigmoid')
def model_body_template(inputs, inp, oup, params):
    with inputs.use('ip') as layer:
        # the with block allows
        # for easy access of a specific named input.
        output = Dense(params)(layer)
    return inputs, output

# create the model.
model = Janggu.create(template=model_body_template,
                      modelparams=3,
                      inputs=DATA, outputs=LABELS)
model.summary()
```

As is illustrated by the example, automatic Input and Output layer determination can be achieved by using the decorators `inputlayer` and/or `outputdense` which extract the layer dimensions from the provided input and output Datasets in the create constructor.

2.2.4 Fit a neural network on DNA sequences

In the previous sections, we learned how to acquire data and how to instantiate neural networks. Now let's create and fit a simple convolutional neural network that learns to discriminate between two classes of sequences. In the following example the sample sequences are of length 200 bp each. *sample.fa* contains Oct4 CHip-seq peaks and sample2.fa contains Mafk CHip-seq peaks. We shall use a simple convolutional neural network with 30 filters of length 21 bp to learn the sequence features that discriminate the two sets of sequences.

The example makes use of two more janggu utilities: First, `DnaConv2D` constitutes a keras layer wrapper that facilitates scanning of both DNA strands with the same kernels. That is it simulataneously applies a convolution and a cross-correlation and aggregates the resulting activities. Second, the example illustrates the dataset wrapper `ReduceDim` which allows to collapse 4D the signal contained in the Cover object across the sequence length and strand dimension. The result is yields a 2D table-like dataset which is used in the subsequent model fitting example.

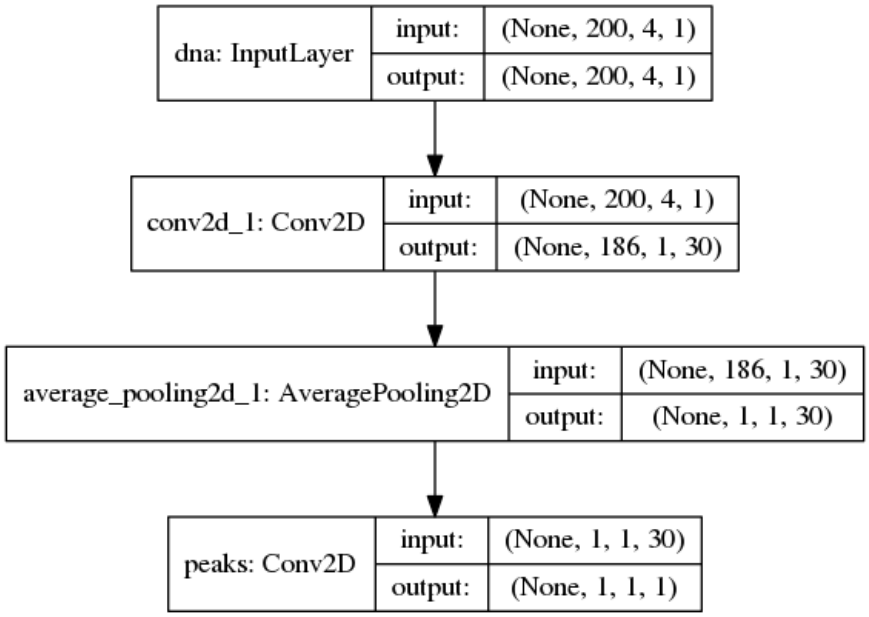
```
from keras.layers import Conv2D
from keras.layers import AveragePooling2D
from janggu import inputlayer
from janggu import outputconv
from janggu import DnaConv2D
from janggu.data import ReduceDim


# load the dataset which consists of
# 1) a reference genome
REFGENOME = resource_filename('janggu', 'resources/pseudo_genome.fa')
# 2) ROI contains regions spanning positive and negative examples
ROI_FILE = resource_filename('janggu', 'resources/roi_train.bed')
# 3) PEAK_FILE only contains positive examples
```

(continues on next page)

(continued from previous page)

```
PEAK_FILE = resource_filename('janggu', 'resources/scores.bed')

# DNA sequences are loaded directly from the reference genome
DNA = Bioseq.create_from_refgenome('dna', refgenome=REFGENOME,
                                   roi=ROI_FILE,
                                   binsize=200)

# Classification labels over the same regions are loaded
# into the Coverage dataset.
# It is important that both DNA and LABELS load with the same
# binsize, stepsize to ensure
# the correct correspondence between both datasets.
# Finally, the ReduceDim dataset wrapper transforms the 4D Coverage
# object into a 2D table like object (regions by conditions)
LABELS = ReduceDim(Cover.create_from_bed('peaks', roi=ROI_FILE,
                                 bedfiles=PEAK_FILE,
                                 binsize=200,
                                 resolution=None), aggregator='mean')


# 2. define a simple conv net with 30 filters of length 15 bp
# and relu activation.
# outputconv as opposed to outputdense will put a conv layer as output
@inputlayer
@outputdense('sigmoid')
def double_stranded_model(inputs, inp, oup, params):
    with inputs.use('dna') as layer:
        # The DnaConv2D wrapper can be used with Conv2D
        # to scan both DNA strands with the weight matrices.
        layer = DnaConv2D(Conv2D(params[0], (params[1], 1),
                                 activation=params[2]))(layer)

    output = GlobalAveragePooling2D(name='motif')(layer)
    return inputs, output


# 3. instantiate and compile the model
model = Janggu.create(template=double_stranded_model,
                      modelparams=(30, 15, 'relu'),
                      inputs=DNA, outputs=LABELS)
model.compile(optimizer='adadelta', loss='binary_crossentropy',
              metrics=['acc'])

# 4. fit the model
model.fit(DNA, ReduceDim(LABELS, epochs=100)
```

An illustration of the network architecture is depicted below. Upon creation of the model a network depiction is automatically produced in `<results_root>/models` which is illustrated below



After the model has been trained, the model parameters and the illustration of the architecture are stored in `<results_root>/models`. Furthermore, information about the model fitting, model and dataset dimensions are written to `<results_root>/logs`.

Note that in the example above the output dimensionality of the network is 4D. However, it might be more convenient at times to remove the single dimensional elements of the array. This can be achieved by wrapping the LABELS dataset using `ReduceDim`. In this case the example becomes

```
@inputlayer
@outputdense('sigmoid')
def double_stranded_model(inputs, inp, oup, params):
    with inputs.use('dna') as layer:
        # The DnaConv2D wrapper can be used with Conv2D
        # to scan both DNA strands with the weight matrices.
        layer = DnaConv2D(Conv2D(params[0], (params[1], 1),
                                 activation=params[2]))(layer)

    output = GlobalAveragePooling2D(name='motif')(layer)
    return inputs, output


# 3. instantiate and compile the model
model = Janggu.create(template=double_stranded_model,
                      modelparams=(30, 15, 'relu'),
                      inputs=DNA, outputs=ReduceDim(LABELS))
model.compile(optimizer='adadelta', loss='binary_crossentropy',
              metrics=['acc'])

# 4. fit the model
model.fit(DNA, ReduceDim(LABELS), epochs=100)
```

2.3 Part III) Evaluation and interpretation of the model

Janggu supports various methods to evaluate and interprete a trained model, including evaluating summary scores, inspecting the results in the built-in genome browser (see Part I), evaluating the integrated gradients which allows to visualized input feature importance and by offering support for variant effect predictions. In this last part we will illustrate these aspects.

2.3.1 Evaluation of summary scores

After the model has been trained, the quality of the predictions is usually summarized by its agreement with the ground truth, e.g. by evaluating the area under the ROC curve in a binary classification application or by computing the correlation between predictions and targets in a regression setting.

For some commonly used evaluation criteria, the evaluate method directly allows to determine and export the given metric results. For example, for a classification task the following line evaluates the ROC and PRC and exports a figure and a tsv file, respectively, for each measure.

```
model.evaluate(DNA_TEST, LABELS_TEST, callbacks=['roc', 'prc', 'auprc', 'auroc'])
```

The results are stored in `<results_root>/evaluation/{roc,prc}.png` as well as `<results_root>/evaluation/{auroc,auprc}.tsv`.

Furthermore, for a regression setting it is possible to invoke

```
model.evaluate(DNA_TEST, LABELS_TEST, callbacks=['cor', 'mae', 'mse', 'var_explained
↪'])
```

which evaluates the Pearson's correlation, the mean absolute error, the mean squared error and the explained variance into tsv files.

It is also possible to customize the scoring callbacks by instantiating a `Scorer` objects which can be passed to `model.evaluate` and `model.predict`. Further details about customizing the scoring callbacks are given in *Customize Evaluation*.

2.3.2 Input feature importance

In order to inspect what the model has learned, it is possible to identify the most important features in the input space using the integrated gradients method.

This is illustrated on a toy example for discriminating Oct4 and Mafk binding sites (see variant effect prediction).

2.3.3 Variant effect prediction

In order to measure the effect of single nucleotide variant on the predict network output can be tested via the `Janggu.predict_variant_effect` based on a Bioseq object and single nucleotide variants in VCF format. This method evaluates the network for each variant (using its sequence context) as well as its respective reference sequence. As a result, an hdf5 file and a bed file will be produced which contain the network predictions for each variant and the associated genomic loci. An illustration of the variant effect prediction in the notebook (see variant effect prediction).

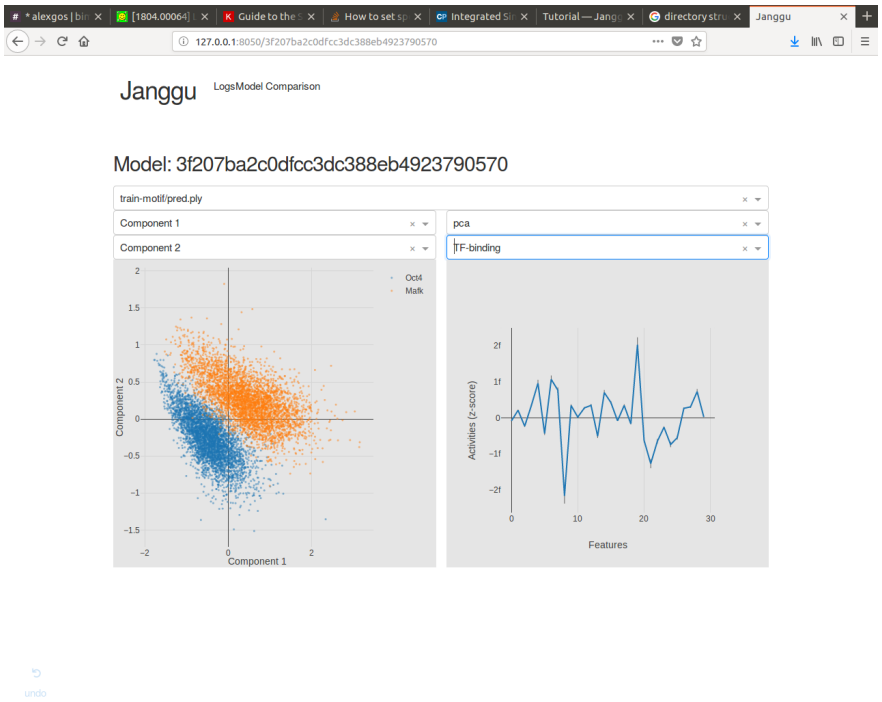
2.3.4 Browse through the results

Finally, after you have fitted and evaluated your results you can browse through the results in an web browser of your choice.

To this end, first start the web application server

```
janggu -path <results-root>
```

Then you can inspect the outputs in a browser of your choice (default: localhost:8050)



CHAPTER 3

Genomic Datasets

One of the central features of Janggu are the genomic datasets `Cover` and `Bioseq`. On the one hand, they allow quick and flexible access to genomics data, including **FASTA**, **BAM**, **BIGWIG**, **BED** and **GFF** file formats, which bridges the gap between the data being present in raw file formats and the numpy inputs required for python-based deep learning models (e.g. keras). On the other hand, predictions from a deep learning library again are in numpy format. Janggu facilitates a convertion between numpy arrays and `Cover` objects in order to associate the predictions with the respective genomic coordinates. Finally, coverage information may be exported to BIGWIG or inspected directly via genome browser-like plots.

3.1 General principle of the Genomics Datasets

Internally, the genomics datasets maintains coverage or sequence type of information along with the associated genomic intervals. Externally, the datasets behave similar to a numpy array. This makes it possible to directly consume the datasets using keras, for instance.

In this section we briefly describe the internal organisation of these datasets. The classes `Cover` and `Bioseq` maintain a `GenomicArray` and a `GenomicIndexer` object. `GenomicArray` is a general data structure that holds numeric information about the genome. For instance, read count coverage. It can be accessed via genomic coordinates (e.g. chromosome name, start and end) and returns the respective data as numpy array. The `GenomicIndexer` maintains a list of genomic coordinates, which should be traversed during training/evaluation. The GenomicIndexer can be accessed by an integer-valued index which returns the associated genomic coordinates.

When querying the 'i'-th region from `Cover` or `Bioseq`, the index is passed to the `GenomicIndexer` which yields a genomic coordinates that is passed on to the `GenomicArray`. The result is returned in numpy format. Similarly, the dataset objects also support slicing and indexing via a list of indices, which is usually relevant when using mini-batch learning.

3.2 Normalization

Upon creation of a `Cover` object, normalization of the raw data might be require. For instance, to make coverage tracks comparable across replicates or experiments. To this end, `create_from_bam`, `create_from_bigwig`

and `create_from_bed` expose a `normalizer` option. Janggu already implements various normalization methods which can be called by name, TPM (transcript per million) normalization. For instance, using

```
Cover.create_from_bam('track', bamfiles=samplefile, roi=roi, normalizer='tpm')
```

Other preprocessing and normalization options are: `zscore`, `zscorelog`, `binsizenorm` and `perctrim`. The latter two apply normalization for read depth and trimming the signal intensities at the 99%-ile.

Normalizers can also be applied via callables and/or in combination with other transformations. For instance, suppose we want to trim the outliers at the 95%-tile instead and subsequently apply the z-score transformation then we could use

```
from janggu.data import PercentileTrimming
from janggu.data import ZScore

Cover.create_from_bam('track', bamfiles=samplefile, roi=roi,
                      normalizer=[PercentileTrimming(95), ZScore()])
```

It might be necessary to evaluate the normalization parameter on one dataset and apply the same transformation on other datasets. For instance, in the case of the ZScore, we might want to keep the mean and standard deviation that was obtained from, say the training set, and reuse the to normalize the test set. This is possible by just creating a zscore object that is used multiple times. At the first invokation the mean and standard deviation are determine and the transformation is applied. Subsequently, the zscore is determined using the predetermined mean and standard deviation. For example:

```
from janggu.data import ZScore

zscore = ZScore()

# First, mean and std will be determined.
# Then zscore transformation is applied.
Cover.create_from_bam('track_train', bamfiles=samplefile, roi=roitrain,
                      normalizer=[zscore])

# Subsequently, zscore transformation is applied with
# the same mean and std determined from the training set.
Cover.create_from_bam('track_test', bamfiles=samplefile, roi=roitest,
                      normalizer=[zscore])
```

In case a different normalization procedure is required that is not contained in janggu, it is possible to define a custom_normalizer as follows:

```
def custom_normalizer(genomicarray):

   # perform normalization genomicarray

   return genomicarray
```

The currently implemented normalizers may be a good starting point for this purpose.

3.3 Granularity of the coverage

Depending on the applications, different granularity of the coverage data might be required. For instance, one might be interested in reading out nucleotide-resolution coverage for one purpose or 50 base-pair resolution bins for another. Furthermore, in some cases the signal of variable size regions might be of interest. For example, the read counts across the gene bodies, to measure gene expression levels.

These adjustments can be made when invoking `create_from_bam`, `create_from_bigwig` and `create_from_bed` using an appropriate region of interest ROI file in conjunction with specifying the `resolution` and `collapser` parameter.

First, we the resolution parameter allows to the coverage granularity. For example, base-pair and 50-base-pair resolution would be possible using

```
Cover.create_from_bam('track', bamfiles=samplefile, roi=roi,
                      resolution=1)

Cover.create_from_bam('track', bamfiles=samplefile, roi=roi,
                      resolution=50)
```

janggu-trim

When using N-based pair resolution with `n>1` in conjunction with the option `store_whole_genome=True`, then the region of interest starts and ends must be divisible by the resolution. Otherwise, undesired rounding effect might occur. This can be achieved by using `janggu-trim`. See Section command line tools.

In case the signal intensity should be summarized across the entire interval, specify `resolution=None`. For instance, if the region of interest contains a set of variable length gene bodies, the total read count per gene can be obtained using

```
Cover.create_from_bam('genes',
                      bamfiles=samplefile,
                      roi=geneannot,
                      resolution=None)
```

It is also possible to use `resolution=None` in conjunction with e.g. `binsize=200` which would have the same effect as chosing `binsize=resolution=200`.

Whenever we deal with `resolution>1`, an aggregation operation needs to be performed to summarize the signal intensity across the region. For instance, for `create_from_bam` the reads are summed within each interval.

For `create_from_bigwig` and `create_from_bed`, it is possible to adjust the collapser. For example, 'mean' or 'sum' aggregation can be applied by name or by handing over a callable according to

```
import numpy as np

Cover.create_from_bigwig('bwtrack',
                         bigwigfiles=samplefile,
                         roi=roi,
                         resolution=50,
                         collapser='mean')

Cover.create_from_bigwig('bwtrack',
                         bigwigfiles=samplefile,
                         roi=roi,
                         resolution=50,
                         collapser=np.sum)
```

Moreover, more specialized aggregations may require a custom collaper function. In that case, it is important to note that the function expects a 3D numpy array and the aggragation should be performed across the second dimension. For example

```
def custom_collapser(numpyarray):

    # Initially, the dimensions of numpyarray correspond to
    # (intervallength // resolution, resolution, strand)

    numpyarray = numpyarray.sum(axis=1)

    # Subsequently, we return the array of shape
    # (intervallength // resolution, strand)

    return numpyarray
```

3.4 Caching

The construction, including loading and preprocessing, of a genomic dataset might require a significant amount of time. In order to avoid having to create the coverage profiles each time you want to use them, they can be cached and quickly reloaded later. Caching can be activated via the options `cache=True`. When caching is required, janggu will check for changes in the file content, file composition and various dataset specific argument (e.g. binsize, resolution) by constructing a SHA256. The dataset will be loaded or reloaded from scratch if the determined hash does not exist.

Example:

```
# load hg19 if the cache file does not exist yet, otherwise
# reload it.
Bioseq.create_from_refgenome('dna', refgenome, order=1, cache=True)
```

3.5 Dataset storage

3.5.1 Storage option

Depending on the structure of the dataset, the required memory to store the data and the available memory on your machine, different storage options are available for the genomic datasets, including **numpy array**, as **sparse array** or as **hdf5 dataset**. To this end, `create_from_bam`, `create_from_bigwig`, `create_from_bed`, `create_from_seq` and `create_from_refgenome` expose the *storage* option, which may be 'ndarray', 'sparse' or 'hdf5', respectively.

'ndarray' amounts to perhaps the fastest access time, but also most memory demanding option for storing the data. It might be useful for dense datasets, and relatively small datasets that conveniently fit into memory.

If the data is sparse, the option *sparse* yields a good compromise between access time and speed. In that case, the data is stored in its compressed sparse form and converted to a dense representation when querying mini-batches. This option may be used to store e.g. genome wide ChIP-seq peaks profiles, if peaks occur relatively rarely.

Finally, if the data is too large to be kept in memory, the option *hdf5* allows to consume the data directly from disk. While, the access time for processing data from hdf5 files may be higher, it allows to processing huge datasets with a small amount of RAM in your machine.

3.5.2 Whole and partial genome storage

`Cover` and `Bioseq` further allow to maintain coverage and sequence information from the entire genome or only the part that is actively consumed during training. This option can be configured by `store_whole_genome=True/`

`False`.

In most situations, the user may find it convenient to set *store_whole_genome=False*. In that case, when loading `Cover` and `Bioseq` only information overlapping the region of interest will be gathered. The advantage of this would be not to have to store an overhead of information when only a small part of the genome is of interest for consumption.

On the other hand, *store_whole_genome=True* might be an advantage for the following purposes:

1. If a large part of the genome is consumed for training/evaluation
2. If in addition the *stepsize* for traversing the genome is smaller than *binsize*, in which case mutually overlapping intervals do not have to be stored redundantly.
3. It simplifies sharing of the same genomic array for different tasks. For example, during training and testing different parts of the same genomic array may be consumed.

3.6 Converting Numpy to Cover

When performing predictions, e.g. with a keras model, the output corresponds to an ordinary numpy array. In order to reestablish the association of the predicted values with the genomic coordinates **Cover** exposes the constructor: *create_from_array*. Upon invocation, a new `Cover` object is composed that holds the predicted values. These predictions may subsequently be illustrated via *plotGenomeTrack* or exported to a BIGWIG file.

3.7 Evaluation features

`Cover` objects may be exported as BIGWIG files. Accordingly, for each condition in the `Cover` a file will be created.

It is also possible to illustrate predictions in terms of a genome browser-like plot using *plotGenomeTrack*, allowing to interactively explore prediction scores (perhaps in comparison with the true labels) or feature activities of the internal layers of a neural net. *plotGenomeTrack* return a matplotlib figure that can be stored into a file using native matplotlib functionality.

3.8 Rearranging channel dimensions

Depending on the deep learning library that is used, the dimensionality of the tensors need to be set up in a specific order. For example, tensorflow expects the channel to be represented by the last dimension, while theano or pytorch expect the channel at the first dimension. With the option *channel_last=True/False* it is possible to configure the output dimensionality of `Cover` and `Bioseq`.

3.9 Wrapper Datasets

A Cover object is represents a 4D object. However, sometimes one or more dimensions of Cover might be single dimensional (e.g. containing only one element). These dimensions can be dropped using `ReduceDim`. For example `ReduceDim(cover)`.

3.10 Different views datasets

Suppose you already have loaded DNA sequence from a reference genome and you want to use a different parts of it for training and validating the model performance. This is achieved by the view mechanism, which allows to reuse the same dataset by instantiating views that reading out different subsets.

For example, a view constituting the training and test set, respectively.

```
# union ROI for training and test set.
ROI_FILE = resource_filename('janggu', 'resources/roi.bed')
ROI_TRAIN_FILE = resource_filename('janggu', 'resources/roi_train.bed')
ROI_TEST_FILE = resource_filename('janggu', 'resources/roi_test.bed')

DNA = Bioseq.create_from_refgenome('dna', refgenome=REFGENOME,
                                   roi=ROI_FILE,
                                   binsize=200,
                                   store_whole_genome=True)

DNA_TRAIN = view(DNA, ROI_TRAIN_FILE)
DNA_TEST = view(DNA, ROI_TEST_FILE)
```

Since underneath the actual dataset is just referenced rather than copied, the memory footprint won't increase. It just allows to read out different parts of the genome.

An example is illustrated in the using view notebook.

3.11 Randomized dataset

In order to achieve good predictive performances, it is recommended to randomize the mini-batches during model fitting. This is usually achieved by specifying *shuffle=True* in the fit method.

However, when using HDF5 dataset, this approach may be prohibitively slow due to the limitations that data from HDF5 files need to be accessed in chunks rather than in random access fashion.

In order to overcome this issue, it is possible to randomize the dataset already during loading time such that the data can be consumed later by reading coherent chunks by setting *shuffle=False*.

For example, randomization is induced by specifying an integer-valued `random_state` as in the example below

```
DNA = Bioseq.create_from_refgenome('dna', refgenome=REFGENOME,
                                   roi=ROI_TRAIN_FILE,
                                   binsize=200,
                                   storage='hdf5',
                                   cache=True,
                                   store_whole_genome=False,
                                   random_state=43)
```

For this option to be effective and correct, all datasets consumed during e.g. training need to be provided with the same `random_state` value. Furthermore, the HDF5 file needs to be stored with `store_whole_genome=False`, since data storage is not affected by the random_state when the entire genome is stored. An example is illustrated in the using hdf5 notebook.

CHAPTER 4

Output directory configuration

Optionally, janggu produces various kinds of output files, including cache files for the datasets, log files for monitoring the training / evaluation procedure, stored model parameters or summary output files about the evaluation performance.

The root directory specifying the janggu output location can be configured via setting the environment variable `JANGGU_OUTPUT`. This might be done in the following ways:

Setting the directory globally:

```
export JANGGU_OUTPUT='/output/dir'
```

on startup of the script:

```
JANGGU_OUTPUT='/output/dir' python classify.py
```

or inside your model script using

```
import os
os.environ['JANGGU_OUTPUT']='/output/dir'
```

If `JANGGU_OUTPUT` is not set, root directory will be set to `/home/user/janggu_results`.

CHAPTER 5

Customize Evaluation

Janggu facilitates automatic evaluation and scoring using the Scorer callbacks for `model.predict` and `model.evaluate`.

A number of export methods are readily available in the package. In this section, we describe how to define custom scoring and export functionality to serve specialized use cases. If you intend to implement a custom scorer or a custom exporter, some of the unit test might also serve as useful examples / starting points.

5.1 Score callable

The scoring function should be a python callable with the following signature:

```
def custom_score(ytrue, ypred):
   """Custom score to be used with model.evaluate"""
   # do some evaluation
   return score

def custom_score(ypred):
   """Custom score to be used with model.predict"""
   # do some evaluation
   return score
```

If additional parameters are required for the evaluation, you might want to use the following construct

```
class CustomScore(object):
    def __init__(self, extra_parameter):
        self.extra_parameter

    def __call__(self, ytrue, ypred):
        # do some evaluation using self.extra_parameter
        return score
```

The results returned by the custom scorer may be of variable types, e.g. list or a scalar value, depending on the use case. Therefore, it is important to choose or designe an exporter that can understand and process the score subsequently.

5.2 Exporter callable

A custom exporter can be defined as a python callable using the following construct

```
class CustomExport(object):
    def __init__(self, extra_parameter):
        self.extra_parameter

    def __call__(self, output_dir, name, results):
        # run export
        pass
```

Of course, if no extra parameters are required, a plain function may also be specified to export the results.

Upon invocation of the exporter, `output_dir`, `name` and `results` are passed. The first two arguments dictate the output location and file name to store the results in. On the other hand, `results` holds the scoring results as a python dictionary of the form: `{'date': <currenttime>, 'value': score_values, 'tags': datatags}` `score_value` denotes another dictionary whose keys are given by a tuple *(modelname, layername, conditionname)* and whose values are the returned score values from the scoring function (see above). Example exporters can be found in *API* or in the source code of the package.

5.3 Custom example scorers

A `Scorer` maintains a **name**, a **scoring function** and an **exporter function**. The latter two dictate the scoring method and how the results should be stored.

An example of using `Scorer` to evaluate the ROC curve and the area under the ROC curve (auROC) and export it as plot and into a tsv file, respectively, is shown below

```
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
from janggu import Scorer
from janggu.utils import ExportTsv
from janggu.utils import ExportScorePlot

# create a scorer
score_auroc = Scorer('auROC',
                     roc_auc_score,
                     exporter=ExportTsv())
score_roc = Scorer('ROC',
                   roc_curve,
                   exporter=ExportScorePlot(xlabel='FPR', ylabel='TPR'))
# determine the auROC
model.evaluate(DNA, LABELS, callbacks=[score_auroc, score_roc])
```

After the evaluation, you will find `auROC.tsv` and `ROC.png` in `<results-root>/evaluation/<modelname>/`.

Similarly, you can use `Scorer` to export the predictions of the model. Below, the output predictions are exported in json format.

```
from janggu import Scorer
from janggu import ExportJson

# create scorer
```

(continues on next page)

(continued from previous page)

```
pred_scorer = Scorer('predict', exporter=ExportJson())

# Evaluate predictions
model.predict(DNA, callbacks=[pred_scorer])
```

Using the Scorer callback objects, a number of evaluations can be run out of the box. For example, with different *sklearn.metrics* and different exporter options. A list of available exporters can be found in *API*.

Alternatively, you can also plug in custom functions

```
# computes the per-data point loss
score_loss = Scorer('loss', lambda t, p: -t * numpy.log(p),
                        exporter=ExportJson())
```

CHAPTER 6

Command line tools

6.1 janggu

The janggu app can be used to browse through the results of one or more models using webbrowser of your choice.

Example usage:

```
janggu -path <results-root> -port <PORT>
```

6.2 janggu-trim

janggu-trim can be used to trim the interval starts and ends of a given BED/GFF file which is intended for creating the ROI by a specified factor.

Trimming might circumvent undesired round effects when using `resolution>1` and `store_whole_genome=True` to handle covarage data with `Cover`. Therefore, we suggest to trim the ROIs that are used during training and evaluation beforehand. For the sake of convenience, we added the tool janggu-trim to do that.

Example usage:

```
janggu-trim input.bed trimmed.bed -divby 50
```

CHAPTER 7

API

7.1 janggu.data - Genomics datasets for deep learning

Bioseq.create_from_seq(name, fastafile[, ...])	Create a Bioseq class from a biological sequences.
Bioseq.create_from_refgenome(name, refgenome)	Create a Bioseq class from a reference genome.
Cover.create_from_bam(name, bamfiles[, roi, ...])	Create a Cover class from a bam-file (or files).
Cover.create_from_bigwig(name, bigwigfiles)	Create a Cover class from a bigwig-file (or files).
Cover.create_from_bed(name, bedfiles[, roi, ...])	Create a Cover class from a bed-file (or files).
Cover.create_from_array(name, array, gindexer)	Create a Cover class from a numpy.array.
plotGenomeTrack(tracks, chrom, start, end[, ...])	plotGenomeTrack shows plots of a specific interval from cover objects data.
Track(data, height)	General track
HeatTrack(data[, height])	Heatmap Track
LineTrack(data[, height, linestyle, marker, ...])	Line track
SeqTrack(data[, height])	Sequence Track

7.1.1 Main Dataset classes

Dataset(name)	Dataset interface.
Cover(name, garray, gindexer)	Cover class.
Bioseq(name, garray, gindexer, alphabet)	Bioseq class.
Array(name, array[, conditions])	Array class.
GenomicIndexer(binsize, stepsize[, flank, ...])	GenomicIndexer maps a set of integer indices to respective genomic intervals.

class janggu.data.**Dataset**(*name*)
Dataset interface.

All dataset classes in janggu inherit from the Dataset class which mimics a numpy array and can be used directly with keras.

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

Variables

- ***name*** (*str*) – Name of the dataset
- ***shape*** (*tuple*) – numpy-style shape of the dataset

name
Dataset name

shape
Shape of the dataset

class janggu.data.**Cover**(*name*, *garray*, *gindexer*)
Cover class.

This datastructure holds coverage information across the genome. The coverage can conveniently fetched from a list of bam-files, bigwig-file, bed-files or gff-files.

Parameters

- **name** (*str*) – Name of the dataset
- **garray** (GenomicArray) – A genomic array that holds the coverage data
- **gindexer** (*GenomicIndexer* or None) – A genomic indexer translates an integer index to a corresponding genomic coordinate. It can be None the genomic indexer is supplied later.

classmethod create_from_array(*name*, *array*, *gindexer*, *genomesize=None*, *conditions=None*, *resolution=None*, *storage='ndarray'*, *overwrite=False*, *cache=False*, *datatags=None*, *padding_value=0.0*, *store_whole_genome=False*, *verbose=False*)
Create a Cover class from a numpy.array.

The purpose of this function is to convert output prediction from keras which are in numpy.array format into a Cover object.

Parameters

- **name** (*str*) – Name of the dataset
- **array** (*numpy.array*) – A 4D numpy array that will be re-interpreted as genomic array.
- **gindexer** (*GenomicIndexer*) – Genomic indices associated with the values contained in array.
- **genomesize** (*dict or None*) – Dictionary containing the genome size to fetch the coverage from. If *genomesize=None*, the genome size is automatically determined from the GenomicIndexer. If *store_whole_genome=False* this option does not have an effect.
- **conditions** (*list(str) or None*) – List of conditions. If *conditions=None*, the conditions are obtained from the filenames (without the directories and file-ending).
- **storage** (*str*) – Storage mode for storing the coverage data can be 'ndarray', 'hdf5' or 'sparse'. Default: 'ndarray'.
- **overwrite** (*boolean*) – Overwrite cachefiles. Default: False.

- **datatags** (*list(str) or None*) – List of datatags. Together with the dataset name, the datatags are used to construct a cache file. If `cache=False`, this option does not have an effect. Default: None.
- **store_whole_genome** (*boolean*) – Indicates whether the whole genome or only ROI should be loaded. Default: False.
- **padding_value** (*float*) – Padding value. Default: 0.
- **verbose** (*boolean*) – Verbosity. Default: False

classmethod create_from_bam(*name*, *bamfiles*, *roi=None*, *genomesize=None*, *conditions=None*, *min_mapq=None*, *binsize=None*, *stepsize=None*, *flank=0*, *resolution=1*, *storage='ndarray'*, *dtype='float32'*, *stranded=True*, *overwrite=False*, *pairedend='5prime'*, *template_extension=0*, *datatags=None*, *cache=False*, *normalizer=None*, *zero_padding=True*, *random_state=None*, *store_whole_genome=False*, *verbose=False*)

Create a Cover class from a bam-file (or files).

This constructor can be used to obtain coverage from BAM files. For single-end reads the read will be counted at the 5 prime end. Paired-end reads can be counted relative to the 5 prime ends of the read (default) or with respect to the midpoint.

Parameters

- **name** (*str*) – Name of the dataset
- **bamfiles** (*str or list*) – bam-file or list of bam files.
- **roi** (*str, list(Interval), BedTool, pandas.DataFrame or None*) – Region of interest over which to iterate. If set to None, the coverage will be fetched from the entire genome and a genomic indexer must be attached later.
- **genomesize** (*dict or None*) – Dictionary containing the genome size. If *genomesize=None*, the genome size is determined from the bam header. If *store_whole_genome=False*, this option does not have an effect.
- **conditions** (*list(str) or None*) – List of conditions. If *conditions=None*, the conditions are obtained from the filenames (without the directories and file-ending).
- **min_mapq** (*int*) – Minimal mapping quality. Reads with lower mapping quality are filtered out. If None, all reads are used.
- **binsize** (*int or None*) – Binsize in basepairs. For binsize=None, the binsize will be determined from the bed-file. If resolution is of type integer, this requires that all intervals in the bed-file are of equal length. If resolution is None, the intervals in the bed-file may be of variable size. Default: None.
- **stepsize** (*int or None*) – stepsize in basepairs for traversing the genome. If stepsize is None, it will be set equal to binsize. Default: None.
- **flank** (*int*) – Flanking size increases the interval size at both ends by flank base pairs. Default: 0
- **resolution** (*int or None*) – If resolution represents an interger, it determines the base pairs resolution by which an interval should be divided. This requires equally sized bins or zero padding and effectively reduces the storage for coverage data. If resolution=None, the intervals will be represented by a collapsed summary score. For example, gene expression may be expressed by TPM in that manner. In the latter case, variable size intervals are permitted and zero padding does not have an effect. Default: 1.

- **storage** (*str*) – Storage mode for storing the coverage data can be 'ndarray', 'hdf5' or 'sparse'. Default: 'ndarray'.
- **dtype** (*str*) – Typecode to be used for storage the data. Default: 'int'.
- **stranded** (*boolean*) – Indicates whether to extract stranded or unstranded coverage. For unstranded coverage, reads aligning to both strands will be aggregated.
- **overwrite** (*boolean*) – Overwrite cachefiles. Default: False.
- **datatags** (*list(str) or None*) – List of datatags. Together with the dataset name, the datatags are used to construct a cache file. If `cache=False`, this option does not have an effect. Default: None.
- **pairedend** (*str*) – Indicates whether to count reads at the '5prime' end or at the 'midpoint' for paired-end reads. Default: '5prime'.
- **template_extension** (*int*) – Elongates intervals by template_extension which allows to properly count template mid-points whose reads lie outside of the interval. This option is only relevant for paired-end reads counted at the 'midpoint' and if the coverage is not obtained from the whole genome, e.g. roi is not None.
- **cache** (*boolean*) – Indicates whether to cache the dataset. Default: False.
- **zero_padding** (*boolean*) – Indicates if variable size intervals should be zero padded. Zero padding is only supported with a specified binsize. If zero padding is false, intervals shorter than binsize will be skipped. Default: True.
- **normalizer** (*None, str or callable*) – This option specifies the normalization that can be applied. If None, no normalization is applied. If 'zscore', 'zscorelog', 'rpkm' then zscore transformation, zscore transformation on log transformed data and rpkm normalization are performed, respectively. If callable, a function with signature *norm(garray)* should be provided that performs the normalization on the genomic array. Normalization is ignored when using storage='sparse'. Default: None.
- **random_state** (*None or int*) – random_state used to internally randomize the dataset. This option is best used when consuming data for training from an HDF5 file. Since random data access from HDF5 may be probibitively slow, this option allows to randomize the dataset during loading. In case an integer-valued random_state seed is supplied, make sure that all training datasets (e.g. input and output datasets) use the same random_state value so that the datasets are synchronized. Default: None means that no randomization is used.
- **store_whole_genome** (*boolean*) – Indicates whether the whole genome or only ROI should be loaded. If False, a bed-file with regions of interest must be specified. Default: False
- **verbose** (*boolean*) – Verbosity. Default: False

classmethod create_from_bed(*name*, *bedfiles*, *roi=None*, *genomesize=None*, *conditions=None*, *binsize=None*, *stepsize=None*, *resolution=1*, *flank=0*, *storage='ndarray'*, *dtype='float32'*, *mode='binary'*, *store_whole_genome=False*, *overwrite=False*, *zero_padding=True*, *normalizer=None*, *collapser=None*, *minoverlap=None*, *random_state=None*, *datatags=None*, *cache=False*, *verbose=False*)

Create a Cover class from a bed-file (or files).

Parameters

- **name** (*str*) – Name of the dataset

- **bedfiles** (*str or list*) – bed-file or list of bed files.
- **roi** (*str, list(Interval), BedTool, pandas.DataFrame or None*) – Region of interest over which to iterate. If set to None a genomesize must be supplied and a genomic indexer must be attached later.
- **genomesize** (*dict or None*) – Dictionary containing the genome size to fetch the coverage from. If *genomesize=None*, the genome size is fetched from the region of interest.
- **conditions** (*list(str) or None*) – List of conditions. If *conditions=None*, the conditions are obtained from the filenames (without the directories and file-ending).
- **binsize** (*int or None*) – Binsize in basepairs. For binsize=None, the binsize will be determined from the bed-file. If resolution is of type integer, this requires that all intervals in the bed-file are of equal length. If resolution is None, the intervals in the bed-file may be of variable size. Default: None.
- **stepsize** (*int or None*) – stepsize in basepairs for traversing the genome. If stepsize is None, it will be set equal to binsize. Default: None.
- **resolution** (*int or None*) – If resolution represents an interger, it determines the base pairs resolution by which an interval should be divided. This requires equally sized bins or zero padding and effectively reduces the storage for coverage data. If resolution=None, the intervals will be represented by a collapsed summary score. For example, gene expression may be expressed by TPM in that manner. In the latter case, variable size intervals are permitted and zero padding does not have an effect. Default: 1.
- **flank** (*int*) – Flanking size increases the interval size at both ends by flank bins. Note that the binsize is defined by the resolution parameter. Default: 0.
- **storage** (*str*) – Storage mode for storing the coverage data can be 'ndarray', 'hdf5' or 'sparse'. Default: 'ndarray'.
- **dtype** (*str*) – Typecode to define the datatype to be used for storage. Default: 'int'.
- **mode** (*str*) – Mode of the dataset may be 'binary', 'score' or 'categorical'. Default: 'binary'.
- **overwrite** (*boolean*) – Overwrite cachefiles. Default: False.
- **datatags** (*list(str) or None*) – List of datatags. Together with the dataset name, the datatags are used to construct a cache file. If `cache=False`, this option does not have an effect. Default: None.
- **store_whole_genome** (*boolean*) – Indicates whether the whole genome or only ROI should be loaded. If False, a bed-file with regions of interest must be specified. Default: False.
- **zero_padding** (*boolean*) – Indicates if variable size intervals should be zero padded. Zero padding is only supported with a specified binsize. If zero padding is false, intervals shorter than binsize will be skipped. Default: True.
- **normalizer** (*None, str or callable*) – This option specifies the normalization that can be applied. If None, no normalization is applied. If 'zscore', 'zscorelog', 'tpm' then zscore transformation, zscore transformation on log transformed data and rpkm normalization are performed, respectively. If callable, a function with signature *norm(garray)* should be provided that performs the normalization on the genomic array. Normalization is ignored when using storage='sparse'. Default: None.
- **collapser** (*None, str or callable*) – This option defines how the genomic signal should be summarized when resolution is None or greater than one. It is possible to choose a number of options by name, including 'sum', 'mean', 'max'. In addtion, a function

may be supplied that defines a custom aggregation method. If collapser is None, 'max' aggregation will be used. Default: None.

- **minoverlap** (*float or None*) – Minimum fraction of overlap of a given feature with a ROI bin. If None, any overlap (e.g. a single base-pair overlap) is considered as overlap. Default: None
- **cache** (*boolean*) – Indicates whether to cache the dataset. Default: False.
- **random_state** (*None or int*) – random_state used to internally randomize the dataset. This option is best used when consuming data for training from an HDF5 file. Since random data access from HDF5 may be probibitively slow, this option allows to randomize the dataset during loading. In case an integer-valued random_state seed is supplied, make sure that all training datasets (e.g. input and output datasets) use the same random_state value so that the datasets are synchronized. Default: None means that no randomization is used.
- **verbose** (*boolean*) – Verbosity. Default: False

classmethod create_from_bigwig(*name*, *bigwigfiles*, *roi=None*, *genomesize=None*, *conditions=None*, *binsize=None*, *stepsize=None*, *resolution=1*, *flank=0*, *storage='ndarray'*, *dtype='float32'*, *overwrite=False*, *datatags=None*, *cache=False*, *store_whole_genome=False*, *zero_padding=True*, *normalizer=None*, *collapser=None*, *random_state=None*, *nan_to_num=True*, *verbose=False*)

Create a Cover class from a bigwig-file (or files).

Parameters

- **name** (*str*) – Name of the dataset
- **bigwigfiles** (*str or list*) – bigwig-file or list of bigwig files.
- **roi** (*str, list(Interval), BedTool, pandas.DataFrame or None*) – Region of interest over which to iterate. If set to None, the coverage will be fetched from the entire genome and a genomic indexer must be attached later. Otherwise, the coverage is only determined for the region of interest.
- **genomesize** (*dict or None*) – Dictionary containing the genome size. If *genomesize=None*, the genome size is determined from the bigwig file. If *store_whole_genome=False*, this option does not have an effect.
- **conditions** (*list(str) or None*) – List of conditions. If *conditions=None*, the conditions are obtained from the filenames (without the directories and file-ending).
- **binsize** (*int or None*) – Binsize in basepairs. For binsize=None, the binsize will be determined from the bed-file. If resolution is of type integer, this requires that all intervals in the bed-file are of equal length. If resolution is None, the intervals in the bed-file may be of variable size. Default: None.
- **stepsize** (*int or None*) – stepsize in basepairs for traversing the genome. If stepsize is None, it will be set equal to binsize. Default: None.
- **resolution** (*int or None*) – If resolution represents an interger, it determines the base pairs resolution by which an interval should be divided. This requires equally sized bins or zero padding and effectively reduces the storage for coverage data. If resolution=None, the intervals will be represented by a collapsed summary score. For example, gene expression may be expressed by TPM in that manner. In the latter case, variable size intervals are permitted and zero padding does not have an effect. Default: 1.

- **flank** (*int*) – Flanking size increases the interval size at both ends by flank bins. Note that the binsize is defined by the resolution parameter. Default: 0.
- **storage** (*str*) – Storage mode for storing the coverage data can be 'ndarray', 'hdf5' or 'sparse'. Default: 'ndarray'.
- **dtype** (*str*) – Typecode to define the datatype to be used for storage. Default: 'float32'.
- **overwrite** (*boolean*) – Overwrite cachefiles. Default: False.
- **datatags** (*list(str) or None*) – List of datatags. Together with the dataset name, the datatags are used to construct a cache file. If `cache=False`, this option does not have an effect. Default: None.
- **cache** (*boolean*) – Indicates whether to cache the dataset. Default: False.
- **store_whole_genome** (*boolean*) – Indicates whether the whole genome or only ROI should be loaded. If False, a bed-file with regions of interest must be specified. Default: False.
- **zero_padding** (*boolean*) – Indicates if variable size intervals should be zero padded. Zero padding is only supported with a specified binsize. If zero padding is false, intervals shorter than binsize will be skipped. Default: True.
- **normalizer** (*None, str or callable*) – This option specifies the normalization that can be applied. If None, no normalization is applied. If 'zscore', 'zscorelog', 'rpkm' then zscore transformation, zscore transformation on log transformed data and rpkm normalization are performed, respectively. If callable, a function with signature *norm(garray)* should be provided that performs the normalization on the genomic array. Normalization is ignored when using storage='sparse'. Default: None.
- **collapser** (*None, str or callable*) – This option defines how the genomic signal should be summarized when resolution is None or greater than one. It is possible to choose a number of options by name, including 'sum', 'mean', 'max'. In addtion, a function may be supplied that defines a custom aggregation method. If collapser is None, 'mean' aggregation will be used. Default: None.
- **nan_to_num** (*boolean*) – Indicates whether NaN values contained in the bigwig files should be interpreted as zeros. Default: True
- **random_state** (*None or int*) – random_state used to internally randomize the dataset. This option is best used when consuming data for training from an HDF5 file. Since random data access from HDF5 may be probibitively slow, this option allows to randomize the dataset during loading. In case an integer-valued random_state seed is supplied, make sure that all training datasets (e.g. input and output datasets) use the same random_state value so that the datasets are synchronized. Default: None means that no randomization is used.
- **verbose** (*boolean*) – Verbosity. Default: False

class janggu.data.**Bioseq**(*name*, *garray*, *gindexer*, *alphabet*)

Bioseq class.

This class maintains a set of biological sequences, e.g. nucleotide or amino acid sequences, and determines its one-hot encoding.

Parameters

- **name** (*str*) – Name of the dataset
- **garray** (`GenomicArray`) – A genomic array that holds the sequence data.

- **gindexer** (*GenomicIndexer* or None) – A genomic index mapper that translates an integer index to a genomic coordinate. Can be None, if the Dataset is only loaded.
- **alphabet** (*str*) – String of sequence alphabet. For example, 'ACGT'.

classmethod create_from_refgenome(*name*, *refgenome*, *roi=None*, *binsize=None*, *stepsize=None*, *flank=0*, *order=1*, *storage='ndarray'*, *datatags=None*, *cache=False*, *overwrite=False*, *random_state=None*, *store_whole_genome=False*, *verbose=False*)

Create a Bioseq class from a reference genome.

This constructor loads nucleotide sequences from a reference genome. If regions of interest (ROI) is supplied, only the respective sequences are loaded, otherwise the entire genome is fetched.

Parameters

- **name** (*str*) – Name of the dataset
- **refgenome** (*str or Bio.SeqRecord.SeqRecord*) – Reference genome location pointing to a fasta file or a SeqRecord object from Biopython that contains the sequences.
- **roi** (*str, list(Interval), BedTool, pandas.DataFrame or None*) – Region of interest over which to iterate. If set to None, the sequence will be fetched from the entire genome and a genomic indexer must be attached later. Otherwise, the coverage is only determined for the region of interest.
- **binsize** (*int or None*) – Binsize in basepairs. For binsize=None, the binsize will be determined from the bed-file directly which requires that all intervals in the bed-file are of equal length. Otherwise, the intervals in the bed-file will be split to subintervals of length binsize in conjunction with stepsize. Default: None.
- **stepsize** (*int or None*) – stepsize in basepairs for traversing the genome. If stepsize is None, it will be set equal to binsize. Default: None.
- **flank** (*int*) – Flanking region in basepairs to be extended up and downstream of each interval. Default: 0.
- **order** (*int*) – Order for the one-hot representation. Default: 1.
- **storage** (*str*) – Storage mode for storing the sequence may be 'ndarray' or 'hdf5'. Default: 'ndarray'.
- **datatags** (*list(str) or None*) – List of datatags. Together with the dataset name, the datatags are used to construct a cache file. If `cache=False`, this option does not have an effect. Default: None.
- **cache** (*boolean*) – Indicates whether to cache the dataset. Default: False.
- **overwrite** (*boolean*) – Overwrite the cachefiles. Default: False.
- **store_whole_genome** (*boolean*) – Indicates whether the whole genome or only ROI should be loaded. If False, a bed-file with regions of interest must be specified. Default: False.
- **random_state** (*None or int*) – random_state used to internally randomize the dataset. This option is best used when consuming data for training from an HDF5 file. Since random data access from HDF5 may be probibitively slow, this option allows to randomize the dataset during loading. In case an integer-valued random_state seed is supplied, make sure that all training datasets (e.g. input and output datasets) use the same random_state value so that the datasets are synchronized. Default: None means that no randomization is used.

- **verbose** (*boolean*) – Verbosity. Default: False

classmethod create_from_seq(*name*, *fastafile*, *storage='ndarray'*, *seqtype='dna'*, *order=1*, *fixedlen=None*, *datatags=None*, *cache=False*, *overwrite=False*, *verbose=False*)

Create a Bioseq class from a biological sequences.

This constructor loads a set of nucleotide or amino acid sequences. By default, the sequence are assumed to be of equal length. Alternatively, sequences can be truncated and padded to a fixed length.

Parameters

- **name** (*str*) – Name of the dataset
- **fastafile** (*str or list(str) or list(Bio.SeqRecord)*) – Fasta file or list of fasta files from which the sequences are loaded or a list of Bio.SeqRecord.SeqRecord.
- **seqtype** (*str*) – Indicates whether a nucleotide or peptide sequence is loaded using 'dna' or 'protein' respectively. Default: 'dna'.
- **order** (*int*) – Order for the one-hot representation. Default: 1.
- **fixedlen** (*int or None*) – Forces the sequences to be of equal length by truncation or zero-padding. If set to None, it will be assumed that the sequences are already of equal length. An exception is raised if this is not the case. Default: None.
- **storage** (*str*) – Storage mode for storing the sequence may be 'ndarray' or 'hdf5'. Default: 'ndarray'.
- **datatags** (*list(str) or None*) – List of datatags. Together with the dataset name, the datatags are used to construct a cache file. If `cache=False`, this option does not have an effect. Default: None.
- **cache** (*boolean*) – Indicates whether to cache the dataset. Default: False.
- **overwrite** (*boolean*) – Overwrite the cachefiles. Default: False.
- **verbose** (*boolean*) – Verbosity. Default: False

class janggu.data.**Array**(*name*, *array*, *conditions=None*)

Array class.

This datastructure wraps arbitrary numpy.arrays for a deep learning application with Janggu. The main difference to an ordinary numpy.array is that Array has a name attribute.

Parameters

- **name** (*str*) – Name of the dataset
- **array** (`numpy.array`) – Numpy array.
- **conditions** (*list(str) or None*) – Conditions or label names of the dataset.

class janggu.data.**GenomicIndexer**(*binsize*, *stepsize*, *flank=0*, *zero_padding=True*, *collapse=False*, *random_state=None*)

GenomicIndexer maps a set of integer indices to respective genomic intervals.

The genomic intervals can be directly used to obtain data from a genomic array.

classmethod create_from_file(*regions*, *binsize*, *stepsize*, *flank=0*, *zero_padding=True*, *collapse=False*, *random_state=None*)

Creates a GenomicIndexer object.

This method constructs a GenomicIndexer from a given BED or GFF file.

Parameters

- **regions** (*str or list(Interval)*) – Path to a BED or GFF file.
- **binsize** (*int or None*) – Binsize in base pairs. If None, the binsize is obtained from the interval lengths in the bed file, which requires intervals to be of equal length.
- **stepsize** (*int or None*) – Stepsize in base pairs. If stepsize is None, stepsize is set to equal to binsize.
- **flank** (*int*) – flank size in bp to be attached to both ends of a region. Default: 0.
- **zero_padding** (*boolean*) – zero_padding indicate if variable sequence lengths are used in conjunction with zero-padding. If zero_padding is True, a binsize must be specified. Default: True.
- **collapse** (*boolean*) – collapse indicates that the genomic interval will be represented by a scalar summary value. For example, the gene expression value in TPM. In this case, zero_padding does not have an effect. Intervals may be of fixed or variable lengths. Default: False.
- **random_state** (*None or int*) – random_state for shuffling intervals. Default: None

7.1.2 Dataset wrappers

Utilities for reshaping, data augmentation, NaN removal.

ReduceDim(array[, aggregator, axis])	ReduceDim class.
SqueezeDim(array[, axis])	SqueezeDim class.
Transpose(array, axis)	Transpose class.
NanToNumConverter(array)	NanToNumConverter class.
RandomOrientation(array)	RandomOrientation class.
RandomSignalScale(array, deviance)	RandomSignalScale class.

class janggu.data.**ReduceDim**(*array*, *aggregator=None*, *axis=None*)

ReduceDim class.

This class wraps an 4D coverage object and reduces the middle two dimensions by applying the aggregate function. Therefore, it transforms the 4D object into a table-like 2D representation

Example

```
# given some dataset, e.g. a Cover object
# originally, the cover object is a 4D-object.
cover.shape
cover = ReduceDim(cover, aggregator='mean')
cover.shape
# Afterwards, the cover object is 2D, where the second and
# third dimension have been averaged out.
```

Parameters

- **array** (*Dataset*) – Dataset
- **aggregator** (*str or callable*) – Aggregator used for reducing the intermediate dimensions. Available aggregators are 'sum', 'mean', 'max' for performing summation, averaging or obtaining the maximum value. It is also possible to supply a callable directly that performs the operation. Default: 'sum'

- **axis** (*None or tuple(ints)*) – Dimensions over which to perform aggregation. Default: None aggregates with `axis=(1, 2)`

class janggu.data.**SqueezeDim**(*array*, *axis=None*)
SqueezeDim class.

This class wraps an 4D coverage object and reduces the middle two dimensions by applying the aggregate function. Therefore, it transforms the 4D object into a table-like 2D representation

Parameters

- **array** (*Dataset*) – Dataset
- **axis** (*None or tuple(ints)*) – Dimensions over which to perform aggregation. Default: None aggregates with `axis=(1, 2)`

class janggu.data.**Transpose**(*array*, *axis*)
Transpose class.

This class can be used to shuffle the dimensions. For example, if the channel is expected to be at a specific location.

Parameters

- **array** (*Dataset*) – Dataset
- **axis** (*tuple(ints)*) – Order to the dimensions.

class janggu.data.**NanToNumConverter**(*array*)
NanToNumConverter class.

This wrapper dataset converts NAN's in the dataset to zeros.

Example

```
# given some dataset, e.g. a Cover object
cover
cover = NanToNumConverter(cover)

# now all remaining NaNs will be converted to zeros.
```

Parameters **array** (*Dataset*) – Dataset

class janggu.data.**RandomOrientation**(*array*)
RandomOrientation class.

This wrapper randomly inverts the directionality of the signal tracks. For example a signal track is randomely presented in 5' to 3' and 3' to 5' orientation. Furthermore, if the dataset is stranded, the strand is switched as well.

Parameters **array** (*Dataset*) – Dataset object must be 4D.

class janggu.data.**RandomSignalScale**(*array*, *deviance*)
RandomSignalScale class.

This wrapper performs performs random uniform scaling of the original input. For example, this can be used to randomly change the peak or signal heights during training.

Parameters

- **array** (*Dataset*) – Dataset object

- **deviance** (*float*) – The signal is rescaled using (1 + uniform(-deviance, deviance)) x original signal.

7.1.3 Normalization and transformation

LogTransform()	Log transformation of intput signal.
PercentileTrimming(percentile)	Percentile trimming normalization.
RegionLengthNormalization([regionmask])	Normalization for variable-region length.
ZScore([mean, std])	ZScore normalization.
ZScoreLog([mean, std])	ZScore normalization after log transformation.
normalize_garray_tpm(garray)	This function performs TPM normalization for a given GenomicArray.

class janggu.data.**LogTransform**
Log transformation of intput signal.

This class performs log-transformation of a GenomicArray using log(x + 1.) to avoid NAN's from zeros.

class janggu.data.**PercentileTrimming**(*percentile*)
Percentile trimming normalization.

This class performs percentile trimming of a GenomicArray to aleviate the effect of outliers. All values that exceed the value associated with the given percentile are set to be equal to the percentile.

Parameters **percentile** (*float*) – Percentile at which to perform chromosome-level trimming.

class janggu.data.**RegionLengthNormalization**(*regionmask=None*)
Normalization for variable-region length.

This class performs region length normalization of a GenomicArray. This is relevant when genomic features are of variable size, e.g. enhancer regions of different width or when using variable length genes.

Parameters **regionmask** (*str or GenomicIndexer, None*) – A bed file or a genomic indexer that contains the masking region that is considered for the signal. For instance, when normalizing gene expression to TPM, the mask contains exons. Otherwise, the TPM would normalize for the full length gene annotation. If None, no mask is included.

class janggu.data.**ZScore**(*mean=None*, *std=None*)
ZScore normalization.

This class performs ZScore normalization of a GenomicArray. It automatically adjusts for variable interval lenths.

Parameters

- **means** (*float or None*) – Provided means will be applied for zero-centering. If None, the means will be determined from the GenomicArray and then applied. Default: None.
- **stds** (*float or None*) – Provided standard deviations will be applied for scaling. If None, the stds will be determined from the GenomicArray and then applied. Default: None.

class janggu.data.**ZScoreLog**(*mean=None*, *std=None*)
ZScore normalization after log transformation.

This class performs ZScore normalization after log-transformation of a GenomicArray using log(x + 1.) to avoid NAN's from zeros. It automatically adjusts for variable interval lenths.

Parameters

- **means** (*float or None*) – Provided means will be applied for zero-centering. If None, the means will be determined from the GenomicArray and then applied. Default: None.
- **stds** (*float or None*) – Provided standard deviations will be applied for scaling. If None, the stds will be determined from the GenomicArray and then applied. Default: None.

janggu.data.**normalize_garray_tpm**(*garray*)
This function performs TPM normalization for a given GenomicArray.

7.1.4 Visualization utilitites

janggu.data.**plotGenomeTrack**(*tracks*, *chrom*, *start*, *end*, *figsize=(10, 5)*, *plottypes=None*)
plotGenomeTrack shows plots of a specific interval from cover objects data.

It takes one or more cover objects as well as a genomic interval consisting of chromosome name, start and end and creates a genome browser-like plot.

Parameters

- **tracks** (*janggu.data.Cover, list(Cover), janggu.data.Track or list(Track)*) – One or more track objects.
- **chrom** (*str*) – chromosome name.
- **start** (*int*) – The start of the required interval.
- **end** (*int*) – The end of the required interval.
- **figsize** (*tuple(int, int)*) – Figure size passed on to matplotlib.
- **plottype** (*None or list(str)*) – Plot type indicates whether to plot coverage tracks as line plots, heatmap, or seqplot using 'line' or 'heatmap', respectively. By default, all coverage objects are depicted as line plots if plottype=None. Otherwise, a list of types must be supplied containing the plot types for each coverage object explicitly. For example, ['line', 'heatmap', 'seqplot']. While, 'line' and 'heatmap' can be used for any type of coverage data, 'seqplot' is reserved to plot sequence influence on the output. It is intended to be used in conjunction with 'input_attribution' method which determines the importance of paricular sequence letters for the output prediction.

Returns *matplotlib Figure* – A matplotlib figure illustrating the genome browser-view of the coverage objects for the given interval. To depict and save the figure the native matplotlib functions show() and savefig() can be used.

class janggu.data.**Track**(*data*, *height*)
General track

Parameters

- **data** (*Cover object*) – Coverage object
- **height** (*int*) – Track height.

class janggu.data.**HeatTrack**(*data*, *height=3*)
Heatmap Track

Visualizes genomic data as heatmap.

Parameters

- **data** (*Cover object*) – Coverage object
- **height** (*int*) – Track height. Default=3

class janggu.data.**LineTrack**(*data*, *height=3*, *linestyle='-'*, *marker='o'*, *color='b'*, *linewidth=2*)
Line track

Visualizes genomic data as line plot.

Parameters

- **data** (*Cover object*) – Coverage object
- **height** (*int*) – Track height. Default=3
- **linestyle** (*str*) – Linestyle for plot
- **marker** (*str*) – Marker code for plot
- **color** (*str*) – Color code for plot
- **linewidth** (*float*) – Line width.

class janggu.data.**SeqTrack**(*data*, *height=3*)
Sequence Track

Visualizes sequence importance.

Parameters

- **data** (*Cover object*) – Coverage object
- **height** (*int*) – Track height. Default=3

7.2 janggu - Utilities for creating, fitting and evaluating models

This section describes the interface and utilities to build build and evaluate deep learning applications with janggu.

Janggu model and utilities for deep learning in genomics.

Janggu(inputs, outputs[, name])	Janggu class
Janggu.create(template[, modelparams, ...])	Janggu constructor method.
Janggu.create_by_name(name[, custom_objects])	Creates a Janggu object by name.
Janggu.fit([inputs, outputs, batch_size, ...])	Model fitting.
Janggu.predict(inputs[, batch_size, ...])	Performs a prediction.
Janggu.evaluate([inputs, outputs, ...])	Evaluates the performance.
input_attribution(model, inputs[, chrom, ...])	Evaluates the integrated gradients method on the input coverage tracks.

7.2.1 Janggu Model

class janggu.**Janggu**(*inputs*, *outputs*, *name=None*)
Janggu class

The class *Janggu* maintains a `keras.models.Model` object, that is an instance of a neural network. Furthermore, to the outside, Janggu behaves similarly to `keras.models.Model` which allows you to create, fit, and evaluate the model.

Parameters

- **inputs** (*Input or list(Input)*) – Input layer or list of Inputs as defined by keras. See https://keras.io/.

- **outputs** (*Layer or list(Layer)*) – Output layer or list of outputs. See https://keras.io/.
- **name** (*str*) – Name of the model.

Examples

Define a Janggu object similar to keras.models.Model using Input and Output layers.

```
from keras.layers import Input
from keras.layers import Dense

from janggu import Janggu

# Define neural network layers using keras
in_ = Input(shape=(10,), name='ip')
layer = Dense(3)(in_)
output = Dense(1, activation='sigmoid', name='out')(layer)

# Instantiate a model.
model = Janggu(inputs=in_, outputs=output, name='test_model')
model.summary()
```

compile(*optimizer*, *loss*, *metrics=None*, *loss_weights=None*, *sample_weight_mode=None*, *weighted_metrics=None*, *target_tensors=None*)

Model compilation.

This method delegates the compilation to keras.models.Model.compile. See also https://keras.io/models/model/

Examples

```
model.compile(optimizer='adadelta', loss='binary_crossentropy')
```

classmethod create(*template*, *modelparams=None*, *inputs=None*, *outputs=None*, *name=None*)

Janggu constructor method.

This method instantiates a Janggu model with model template and parameters. It also allows to automatically infer and extend the correct input and output layers for the network.

Parameters

- **template** (*function*) – Python function that defines a model template of a neural network. The function signature must adhere to the signature `template(inputs, inputs, outputs, modelparams)` and is expected to return `(input_tensor, output_tensor)` of the neural network.
- **modelparams** (*list or tuple or None*) – Additional model parameters that are passed along to template upon creation of the neural network. For instance, this could contain number of neurons at each layer. Default: None.
- **inputs** (*Dataset, list(Dataset) or None*) – Input datasets from which the input layer shapes should be derived. Use this option together with the `inputlayer` decorator (see Example below).
- **outputs** (*Dataset, list(Dataset) or None*) – Output datasets from which the output layer shapes should be derived. Use this option toghether with `outputdense` or `outputconv` decorators (see Example below).

- **name** (*str or None*) – Model name. If None, a unique model name is generated based on the model configuration and network architecture.

Examples

Specify a model using a model template and parameters:

```
def test_manual_model(inputs, inp, oup, params):
    in_ = Input(shape=(10,), name='ip')
    layer = Dense(params)(in_)
    output = Dense(1, activation='sigmoid', name='out')(in_)
    return in_, output

# Defines the same model by invoking the definition function
# and the create constructor.
model = Janggu.create(template=test_manual_model, modelparams=3)
model.summary()
```

Specify a model using automatic input and output layer determination. That is, only the model body needs to be specified:

```
import numpy as np
from janggu import Janggu
from janggu import inputlayer, outputdense
from janggu.data import Array

# Some random data which you would like to use as input for the
# model.
DATA = Array('ip', np.random.random((1000, 10)))
LABELS = Array('out', np.random.randint(2, size=(1000, 1)))

# The decorators inputlayer and outputdense
# extract the layer shapes and append the respective layers
# to the network
# so that only the model body remains to be specified.
# Note that the the decorator order matters.
# inputlayer must be specified before outputdense.
@inputlayer
@outputdense(activation='sigmoid')
def test_inferred_model(inputs, inp, oup, params):
    with inputs.use('ip') as in_:
        # the with block allows for easy
        # access of a specific named input.
        output = Dense(params)(in_)
    return in_, output

# create a model.
model = Janggu.create(template=test_inferred_model, modelparams=3,
                      name='test_model',
                      inputs=DATA,
                      outputs=LABELS)
```

classmethod create_by_name(*name*, *custom_objects=None*)

Creates a Janggu object by name.

This option is used to load a pre-trained model.

Parameters

- **name** (*str*) – Name of the model.
- **custom_objects** (*dict or None*) – This allows loading of custom layers using load_model. All janggu specific layers are automatically included as custom_objects. Default: None

Examples

```
in_ = Input(shape=(10,), name='ip')
layer = Dense(3)(in_)
output = Dense(1, activation='sigmoid', name='out')(layer)

# Instantiate a model.
model = Janggu(inputs=in_, outputs=output, name='test_model')

# saves the model to <janggu_results>/models
model.save()

# remove the original model
del model

# reload the model
model = Janggu.create_by_name('test_model')
```

evaluate(*inputs=None*, *outputs=None*, *batch_size=None*, *sample_weight=None*, *steps=None*, *datatags=None*, *callbacks=None*, *use_multiprocessing=False*, *workers=1*)

Evaluates the performance.

This method is used to evaluate a given model. All of the parameters are directly delegated the evalute_generator of the keras model. See https://keras.io/models/model/#methods.

Parameters

- **inputs** (`Dataset`, list(Dataset) or Sequence (keras.utils.Sequence)) – Input Dataset or Sequence to use for evaluating the model.
- **outputs** (`Dataset`, list(Dataset) or None) – Output Dataset containing the training targets. If a Sequence is used for inputs, outputs will have no effect.
- **batch_size** (*int or None*) – Batch size. If set to None a batch size of 32 is used.
- **sample_weight** (*np.array or None*) – Sample weights. See https://keras.io.
- **steps** (*int, None.*) – Number of predict steps. If None, this value is determined from the dataset size and the batch_size.
- **datatags** (*list(str) or None*) – Tags to annotate the evaluation results. Default: None.
- **callbacks** (List(`Scorer` or str)) – Scorer instances to be applied on the predictions. Furthermore, commonly used scoring metrics can be added by name, including 'roc', 'auroc', 'prc', 'auprc' for evaluating binary classification applications and 'cor' (for Pearson's correlation), 'mae', 'mse' and 'var_explained' for regression applications.
- **use_multiprocessing** (*boolean*) – Whether to use multiprocessing for the prediction. Default: False.
- **workers** (*int*) – Number of workers to use. Default: 1.

Examples

```
model.evaluate(DATA, LABELS)

# binary classification evaluation with callbacks
model.evaluate(DATA, LABELS, callcacks=['auprc', 'auroc'])
```

fit(*inputs=None*, *outputs=None*, *batch_size=None*, *epochs=1*, *verbose=1*, *callbacks=None*, *validation_data=None*, *shuffle=True*, *class_weight=None*, *sample_weight=None*, *initial_epoch=0*, *steps_per_epoch=None*, *use_multiprocessing=False*, *workers=1*)
Model fitting.

This method is used to fit a given model. Most of parameters are directly delegated the fit_generator of the keras model.

Parameters

- **inputs** (`Dataset`, list(Dataset) or Sequence (keras.utils.Sequence)) – Input Dataset or Sequence to use for fitting the model.
- **outputs** (`Dataset`, list(Dataset) or None) – Output Dataset containing the training targets. If a Sequence is used for inputs, outputs will have no effect.
- **batch_size** (*int or None*) – Batch size. If set to None a batch size of 32 is used.
- **epochs** (*int*) – Number of epochs. Default: 1.
- **verbose** (*int*) – Verbosity level. See https://keras.io.
- **callbacks** (*List(keras.callbacks.Callback)*) – Callbacks to be applied during training. See https://keras.io/callbacks
- **validation_data** (*tuple, Sequence or None*) – Validation data can be a tuple (input_dataset, output_dataset), or (input_dataset, output_dataset, sample_weights) or a keras.utils.Sequence instance or a list of validation chromsomoes. The latter choice only works with when using Cover and Bioseq dataset. This allows you to train on a dedicated set of chromosomes and to validate the performance on respective heldout chromosomes. If None, validation is not applied.
- **shuffle** (*boolean*) – shuffle batches. Default: True.
- **class_weight** (*dict*) – Class weights. See https://keras.io.
- **sample_weight** (*np.array or None*) – Sample weights. See https://keras.io.
- **initial_epoch** (*int*) – Initial epoch at which to start training.
- **steps_per_epoch** (*int, None.*) – Number of steps per epoch. If None, this value is determined from the dataset size and the batch_size.
- **use_multiprocessing** (*boolean*) – Whether to use multiprocessing. See https://keras.io. Default: False.
- **workers** (*int*) – Number of workers to use in multiprocessing mode. Default: 1.

Examples

```
model.fit(DATA, LABELS)
```

get_config()
Get model config.

name
Model name

predict(*inputs*, *batch_size=None*, *verbose=0*, *steps=None*, *layername=None*, *datatags=None*, *callbacks=None*, *use_multiprocessing=False*, *workers=1*)
Performs a prediction.

This method predicts the targets. All of the parameters are directly delegated the predict_generator of the keras model. See https://keras.io/models/model/#methods.

Parameters

- **inputs** (`Dataset`, list(Dataset) or Sequence (keras.utils.Sequence)) – Input Dataset or Sequence to use for fitting the model.
- **batch_size** (*int or None*) – Batch size. If set to None a batch size of 32 is used.
- **verbose** (*int*) – Verbosity level. See https://keras.io.
- **steps** (*int, None.*) – Number of predict steps. If None, this value is determined from the dataset size and the batch_size.
- **layername** (*str or None*) – Layername for which the prediction should be performed. If None, the output layer will be used automatically.
- **datatags** (*list(str) or None*) – Tags to annotate the evaluation results. Default: None.
- **callbacks** (List(`Scorer`)) – Scorer instances to be applied on the predictions.
- **use_multiprocessing** (*boolean*) – Whether to use multiprocessing for the prediction. Default: False.
- **workers** (*int*) – Number of workers to use. Default: 1.

Examples

```
model.predict(DATA)
```

predict_variant_effect(*bioseq*, *variants*, *conditions*, *output_folder*, *condition_filter=None*, *batch_size=None*, *annotation=None*, *ignore_reference_match=False*)
Evaluates the performance.

Parameters

- **bioseq** (`Bioseq`) – Input sequence containing the reference genome.
- **variants** (*str*) – File name of a VCF file containg the variants under study.
- **conditions** (*list(str)*) – Condition labels for each output prediction.
- **output_folder** (*str*) – The method produces an hdf5 and a bed file as output. The bedfile contains the variant positions while the hdf5 file contains the reference and alternative variant scores for each output feature.
- **condition_filter** (*str or None*) – Regular expression filter on which conditions should be evaluated. If None, all output conditions will be returned.
- **batch_size** (*int, None.*) – Batch size. If None, a batch_size of 128 is used.
- **annotation** (*BedTool object or None*) – BedTool holding feature annotation e.g. gene annotation. The annotation may be used to perform strand-specific variant effect predictions. Each variant is intersected with the annotation in order to derive the correct strandedness.

If variants do not overlap with an annotation features or for missing annotation, the forward strand is used.

- **ignore_reference_match** (*boolean*) – Whether to ignore mismatches between the reference sequence and the reference base in the VCF file. If False, the variant will be skipped over and only matching positions are processed. Otherwise all variants will be processed. Default: False.

Returns *tuple* – Tuple containing the output filenames: an hdf5 and a bed file.

Examples

```
# Evaluate all variants and all conditions (outputs)
model.predict_variant_effect(DATA, VARIANTS, CONDITIONS,
                             'vcfoutput')

# Evaluate all variants and a subset of conditions (Ctcf output labels)
model.predict_variant_effect(DATA, LABELS, CONDITIONS,
                             'vcfoutput_subset',
                             contition_filter='Cfcf')
```

save (*filename=None*, *overwrite=True*, *show_shapes=True*)
Saves the model.

Parameters

- **filename** (*str*) – Filename of the stored model. Default: None.
- **overwrite** (*bool*) – Overwrite a stored model. Default: True.

summary ()
Prints the model definition.

7.2.2 Input feature attribution

janggu.**input_attribution** (*model*, *inputs*, *chrom=None*, *start=None*, *end=None*)
Evaluates the integrated gradients method on the input coverage tracks.

This allows to attribute feature importance values to the prediction scores. Integrated gradients have been introduced in Sundararajan, Taly and Yan, Axiomatic Attribution for Deep Networks. PMLR 70, 2017.

The method can either be called, by specifying the region of interest directly by setting chrom, start and end. Alternatively, it is possible to specify the region index. For example, the n^th region of the dataset.

Parameters

- **model** (*Janggu*) – Janggu model wrapper
- **inputs** (`Dataset`, list(Dataset)) – Input Dataset.
- **chrom** (*str or None*) – Chromosome name.
- **start** (*int or None*) – Region start.
- **end** (*int or None*) – Region end.

Examples

```
# Suppose DATA is a Bioseq or Cover object
# To query the input feature importance of a specific genomic region
# use
input_attribution(model, DATA, chrom='chr1', start=start, end=end)
```

7.2.3 Performance evaluation

Scorer.score(model, predicted[, outputs, ...])	Scoring of the predictions relative to true outputs.
Scorer.export(path, collection_name[, datatags])	Exporting of the results.

class janggu.**Scorer**(*name*, *score_fct=None*, *conditions=None*, *exporter=<janggu.utils.ExportJson object>*, *immediate_export=True*, *percondition=True*, *subdir=None*)

Scorer class.

This class implements the callback interface that is used with `Janggu.evaluate` and `Janggu.predict`. The scorer maintains a scoring callable and an exporter callable which take care of determining the desired score and writing the result into a desired file, e.g. json, tsv or a figure, respectively.

Parameters

- **name** (*str*) – Name of the score to be performed.
- **score_fct** (*None or callable*) – Callable that is invoked for scoring. This callable must satisfy the signature `fct(y_true, y_pred)` if used with `Janggu.evaluate` and `fct(y_pred)` if used with `Janggu.predict`. The returned score should be compatible with the exporter.
- **conditions** (*list(str) or None*) – List of strings describing the conditions dimension of the dataset that is processed. If None, conditions are extracted from the y_true Dataset, if available. Otherwise, the conditions are integers ranging from zero to `len(conditions) - 1`.
- **exporter** (*callable*) – Exporter function is used to export the scoring results in the desired manner, e.g. as json or tsv file. This function must satisfy the signature `fct(output_path, filename_prefix, results)`.
- **immediate_export** (*boolean*) – If set to True, the exporter function will be invoked immediately after the evaluation of the dataset. If set to False, the results are maintained in memory which allows to export the results as a collection rather than individually.
- **percondition** (*boolean*) – Indicates whether the evaluation should be performed per condition or across all conditions. The former determines a score for each output condition, while the latter first flattens the array and then scores across conditions. Default: percondition=True.
- **subdir** (*str*) – Name of the subdir to store the output in. Default: None means the results are stored in the 'evaluation' subdir.

export(*path*, *collection_name*, *datatags=None*)

Exporting of the results.

When calling export, the results which have been collected in self.results by using the score method are written to disk by invoking the supplied exporter function.

Parameters

- **path** (*str*) – Output directory.
- **collection_name** (*str*) – Subdirectory in which the results should be stored. E.g. Modelname.
- **datatags** (*list(str) or None*) – Optional tags describing the dataset. E.g. 'training_set'. Default: None

score(*model*, *predicted*, *outputs=None*, *datatags=None*)
Scoring of the predictions relative to true outputs.

When calling score, the provided score_fct is applied for each layer and condition separately. The result scores are maintained in a dict that uses `(modelname, layername, conditionname)` as key and as values another dict of the form: `{'date':<currenttime>, 'value': derived_score, 'tags':datatags}`.

Parameters

- **model** (*Janggu*) – a Janggu object representing the current model.
- **predicted** (*dict{name: np.array}*) – Predicted outputs.
- **outputs** (*dict{name: Dataset} or None*) – True output labels. The Scorer is used with `Janggu.evaluate` this argument will be present. With `Janggu.evaluate` it is absent.
- **datatags** (*list(str) or None*) – Optional tags describing the dataset, e.g. 'test_set'.

7.3 Performance score utilities

class janggu.**ExportJson**(*filesuffix='json'*, *annot=None*, *row_names=None*)
Method that dumps the results in a json file.

Parameters

- **filesuffix** (*str*) – Target file ending. Default: 'json'.
- **annot** (*None, dict*) – Annotation data. If encoded as dict the key indicates the name, while the values holds a list of annotation labels. Default: None.
- **row_names** (*None or list*) – List of row names. Default: None.

class janggu.**ExportTsv**(*filesuffix='tsv'*, *annot=None*, *row_names=None*)
Method that dumps the results as tsv file.

This class can be used to export general table summaries.

Parameters

- **filesuffix** (*str*) – File ending. Default: 'tsv'.
- **annot** (*None, dict*) – Annotation data. If encoded as dict the key indicates the name, while the values holds a list of annotation labels. For example, this can be used to store the true output labels. Default: None.
- **row_names** (*None, list*) – List of row names. For example, chromosomal loci. Default: None.

class janggu.**ExportBed**(*gindexer*, *resolution*)
Export predictions to bed.

This function exports the predictions to bed format which allows you to inspect the predictions in a genome browser.

Parameters

- **gindexer** (*GenomicIndexer*) – GenomicIndexer that links the prediction for a certain region to its associated genomic coordinates.
- **resolution** (*int*) – Used to output the results.

class janggu.**ExportBigwig**(*gindexer*)
Export predictions to bigwig.

This function exports the predictions to bigwig format which allows you to inspect the predictions in a genome browser. Importantly, gindexer must contain non-overlapping windows!

Parameters **gindexer** (*GenomicIndexer*) – GenomicIndexer that links the prediction for a certain region to its associated genomic coordinates.

class janggu.**ExportScorePlot**(*figsize=None*, *xlabel=None*, *ylabel=None*, *fform=None*)
Exporting score plot.

This class can be used for producing an AUC or PRC plot.

Parameters

- **figsize** (*tuple(int, int)*) – Used to specify the figure size for matplotlib.
- **xlabel** (*str or None*) – xlabel used for the plot.
- **ylabel** (*str or None*) – ylabel used for the plot.
- **fform** (*str or None*) – Output file format. E.g. 'png', 'eps', etc. Default: 'png'.

7.4 Decorators for network construction

janggu.**inputlayer**(*func*)
Input layer decorator

This decorator appends an input layer to the network with the correct shape and name.

janggu.**outputdense**(*activation*)
Output layer decorator

This decorator appends an output layer to the network with the correct shape, activation and layer name.

janggu.**outputconv**(*activation*)
Output layer decorator

This decorator appends an output convolution layer to the network with the correct shape, activation and layer name.

7.5 Genomics-specific keras layers

class janggu.**DnaConv2D**(*layer*, *merge_mode='max'*, ***kwargs*)
DnaConv2D layer.

This layer wraps a normal keras Conv2D layer for scanning DNA sequences on both strands using the same weight matrices.

Parameters **merge_mode** (*str or None*) – Specifies how to merge information from both strands. Options: {"max", "ave", "concat", None} Default: "max".

Examples

To scan both DNA strands for motif matches use

```
xin = Input((200, 1, 4))
dnalayer = DnaConv2D(Conv2D(nfilters, filter_shape))(xin)
```

class janggu.**Complement**(**args*, ***kwargs*)
Complement layer.

This layer can be used with keras to determine the complementary DNA sequence in one-hot encoding from a given DNA sequences. It supports higher-order nucleotide representation, e.g. dinucleotides, trinucleotides. The order of the nucleotide representation is automatically determined from the previous layer. To this end, the input layer is assumed to hold the nucleotide representation dimension 3. The layer uses a permutation matrix that is multiplied with the original input dataset in order to evaluate the complementary sequence's one hot representation.

```
forwardstrand_dna = Input((200, 1, 4))
reversestrand_dna = Complement()(forwardstrand_dna)
# this also works for higher-order one-hot encoding.
```

class janggu.**Reverse**(*axis=1*, ***kwargs*)
Reverse layer.

This layer can be used with keras to reverse a tensor for a given axis.

Parameters **axis** (*int*) – Axis which needs to be reversed. Default: 1.

class janggu.**LocalAveragePooling2D**(*window_size=1*, ***kwargs*)
LocalAveragePooling2D layer.

This layer performs window averaging along the lead axis of an input tensor using a given window_size. At the moment, it assumes data_format='channels_last'. This is similar to applying GlobalAveragePooling2D, but where the average is determined in a window of length 'window_size', rather than along the entire sequence length.

Parameters **window_size** (*int*) – Averaging window size. Default: 1.

CHAPTER 8

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

8.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

8.2 Documentation improvements

Janggu could always use more documentation, whether as part of the official Janggu docs, in docstrings, or even on the web in blog posts, articles, and such.

8.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/BIMSBbioinfo/janggu/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

8.4 Development

To set up *janggu* for local development:

1. Fork janggu (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/janggu.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

8.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

8.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 9

Authors

Main developer: Wolfgang Kopp (https://github.com/wkopp)

We are greatful for contributions from

- Annalaura Tamburrini (https://github.com/Annalaura94) for advancing the filtering options of the GenomicIndexer and for developing *plotGenomeTrack*.

CHAPTER 10

Changelog

10.1 0.9.5 (2019-10-17)

- Variant effect prediction: added annotation argument which enables strand-specific variant effect predictions using the strandedness of features in the annotation file.
- Variant effect prediction: added ignore_reference_match argument which enables ignores mismatching nucleotides between the VCF reference base and the reference genome. By default, variant effects are only evaluated if the nucleotides agree in the reference genome and the VCF file.
- Added file validity check
- Added option to control verbosity
- Improved efficiency for reading BAM and BIGWIG files
- Create a new cachefile with random_state only for not storing the whole genome
- Relaxed constraint for using resolution > 1 with ROI intervals. Still the interval starts have to be divisible by the resolution. Otherwise, weird rounding errors might occur.
- Fixed issue due to different numbers of network output layers.
- Added seperate dataversion to better control when cache files need to be reloaded from scratch.

10.2 0.9.4 (2019-07-15)

- Added SqueezeDim wrapper for compatibility with sklearn
- Added Transpose wrapper, replaces channel_last option of the datasets
- Loading paired-end bam-files with pairedend='5pend' option counts both ends now.
- resolution option added to create_from_array
- Relaxed restriction for sequence feature order

- Cover access via interval now returns nucleotide-resolution data regardless of the store_whole_genome option to ensure consistency.
- Refactoring

10.3 0.9.3 (2019-07-08)

- View mechanism added which allows to reuse the same dataset for different purposes, e.g. training set and test set.
- Added a dataset randomization which allows to internally randomize the data in order to avoid having to use shuffle=True with the fit method. This allows fetch randomized data in coherent chunks from hdf5 format files which improves access time.
- Added lazy loading mechanism for DNA and BED files, which defer the determination of the genome size to the dataset creation phase, but does not perform it when loading cached files to improve reload time.
- Caching logic improved in order to maximize the amount of reusability of dataset. For example, when the whole genome is loaded, the data can later be reloaded with different binsizes.
- Variant effect prediction functionality added.
- Improved efficiency for loading coverage from an array.
- Added axis option to ReduceDim
- Added Track classes to improve flexibility on plotGenomeTrack

10.4 0.9.2 (2019-05-04)

- Bugfix: Bioseq caching mechanism fixed.

10.5 0.9.1 (2019-05-03)

- Removed HTSeq dependence in favour of pybedtools for parsing BED, GFF, etc. This also introduces the requirement to have bedtools installed on the system, but it allows to parse BED-like files faster and more conveniently.
- Internal rearrangements for GenomicArray store_whole_genome=False. Now the data is stored as one array in a dict-like handle with the dummy key 'data' rather than storing the data in a fragmented fashion using as key-values the genomic interval and the respective coverages associated with them. This makes storage and processing more efficient.
- Bugfix: added conditions property to wrapper datasets.

10.6 0.9.0 (2019-03-20)

Added various features and bug fixes:

Changes in janggu.data

- Added new dataset wrapper to remove NaNs: NanToNumConverter
- Added new dataset wrappers for data augmentation: RandomOrientation, RandomSignalScale

- Adapted ReduceDim wrapper: added aggregator argument
- plotGenomeTrack added figsize option
- plotGenomeTrack added other plot types, including heatmap and seqplot.
- plotGenomeTrack refactoring of internal code
- Bioseq bugfix: Fixed issue for reverse complementing N's in the sequence.
- GenomicArray: condition, order, resolution are not read from the cache anymore, but from the arguments to avoid inconsistencies
- Normalization of Cover can handle a list of normalizer callables which are applied in turn
- Normaliation and Transformation: Added PercentileTrimming, RegionLengthNormalization, LogTransform
- ZScore and ZScoreLog do not apply RegionLengthNormalization by default anymore.
- janggu.data version-aware caching of datasets included
- Added copy method for janggu datasets.
- split_train_test refactored
- removed obsolete transformations attribute from the datasets
- Adapted the documentation
- Refactoring according to suggestions from isort and pylint

Changes in janggu

- Added input_attribution via integrated gradients for feature importance assignment
- Performance scoring by name for Janggu.evaluate for a number common metrices, including ROC, PRC, correlation, variance explained, etc.
- training.log is stored by default for each model
- Added model_from_json, model_from_yaml wrappers
- inputlayer decorator only instantiates Input layers if inputs == None, which makes the use of inputlayer less restrictive when using nested functions
- Added create_model method to create a keras model directly
- Adapted the documentation
- Refactoring according to suggestions from isort and pylint

10.7 0.8.6 (2019-03-03)

- Bugfix for ROIs that reach beyond the chromosome when loading Bioseq datasets. Now, zero-padding is performed for intervals that stretch over the sequence ends.

10.8 0.8.5 (2019-01-09)

- Updated abstract, added logo
- Utility: janggutrim command line tool for cutting bed file regions to avoid unwanted rounding effects. If rounding issues are detected an error is raised.

- Caching mechanism revisited. Caching of datasets is based on determining the sha256 hash of the dataset. If the data or some parameters change, the files are automatically reloaded. Consequently, the arguments overwrite and datatags become obsolete and have been marked for deprecation.
- Refactored access of GenomicArray
- Added ReduceDim wrapper to convert a 4D Cover object to a 2D table-like object.

10.9 0.8.4 (2018-12-11)

- Updated installation instructions in the readme

10.10 0.8.3 (2018-12-05)

- Fixed issues for loading SparseGenomicArray
- Made GenomicIndexer.filter_by_region aware of flank
- Fixed BedLoader of partially overlapping ROI and bedfiles issue using filter_by_region.
- Adapted classifier, license and keywords in setup.py
- Fixed hyperlinks

10.11 0.8.2 (2018-12-04)

- Bugfix for zero-padding functionality
- Added ndim for keras compatibility

10.12 0.8.1 (2018-12-03)

- Bugfix in GenomicIndexer.create_from_region

10.13 0.8.0 (2018-12-02)

- Improved test coverage
- Improved linter issues
- Bugs fixed
- Improved documentation for scorers
- Removed kwargs for scorers and exporters
- Adapted exporters to classes

10.14 0.7.0 (2018-12-01)

- First public version

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index

R

S

T

Z