Search Keras documentation...

» Keras API reference / Metrics / Classification metrics based on True/False positives & negatives

Classification metrics based on True/False positives & negatives

`AUC` class

[source]

```
tf.keras.metrics.AUC(
    num_thresholds=200,
    curve="ROC",
    summation_method="interpolation",
    name=None,
    dtype=None,
    thresholds=None,
    multi_label=False,
    num_labels=None,
    label_weights=None,
    from_logits=False,
)
```

Approximates the AUC (Area under the curve) of the ROC or PR curves.

The AUC (Area under the curve) of the ROC (Receiver operating characteristic; default) or PR (Precision Recall) curves are quality measures of binary classifiers. Unlike the accuracy, and like cross-entropy losses, ROC-AUC and PR-AUC evaluate all the operational points of a model.

This class approximates AUCs using a Riemann sum. During the metric accumulation phrase, predictions are accumulated within predefined buckets by value. The AUC is then computed by interpolating per-bucket averages. These buckets define the evaluated operational points.

This metric creates four local variables, `true_positives`, `true_negatives`, `false_positives` and `false_negatives` that are used to compute the AUC. To discretize the AUC curve, a linearly spaced set of thresholds is used to compute pairs of recall and precision values. The area under the ROC-curve is therefore computed using the height of the recall values by the false positive rate, while the area under the PR-curve is the computed using the height of the precision values by the recall.

This value is ultimately returned as `auc`, an idempotent operation that computes the area under a discretized curve of precision versus recall values (computed using the aforementioned variables). The `num_thresholds` variable controls the degree of discretization with larger numbers of thresholds more closely approximating the true AUC. The quality of the approximation may vary dramatically depending on `num_thresholds`. The `thresholds` parameter can be used to manually specify thresholds which split the predictions more evenly.

For a best approximation of the real AUC, `predictions` should be distributed approximately uniformly in the range [0, 1] (if `from_logits=False`). The quality of the AUC approximation may be poor if this is not the case. Setting `summation_method` to 'minoring' or 'majoring' can help quantify the error in the approximation by providing lower or upper bound estimate of the AUC.

If `sample_weight` is `None`, weights default to 1. Use `sample_weight` of 0 to mask values.

Arguments

- **num_thresholds**: (Optional) Defaults to 200. The number of thresholds to use when discretizing the roc curve. Values must be > 1.
- **curve**: (Optional) Specifies the name of the curve to be computed, 'ROC' [default] or 'PR' for the Precision-Recall-curve.
- **summation_method**: (Optional) Specifies the Riemann summation method used. 'interpolation' (default) applies mid-point summation scheme for `ROC`. For PR-AUC, interpolates (true/false) positives but not the ratio that is precision (see Davis & Goadrich 2006 for details); 'minoring' applies left summation for increasing intervals and right summation for decreasing intervals; 'majoring' does the opposite.
- **name**: (Optional) string name of the metric instance.

- **dtype**: (Optional) data type of the metric result.
- **thresholds**: (Optional) A list of floating point values to use as the thresholds for discretizing the curve. If set, the `num_thresholds` parameter is ignored. Values should be in [0, 1]. Endpoint thresholds equal to {-epsilon, 1+epsilon} for a small positive epsilon value will be automatically included with these to correctly handle predictions equal to exactly 0 or 1.
- **multi_label**: boolean indicating whether multilabel data should be treated as such, wherein AUC is computed separately for each label and then averaged across labels, or (when False) if the data should be flattened into a single label before AUC computation. In the latter case, when multilabel data is passed to AUC, each label-prediction pair is treated as an individual data point. Should be set to False for multi-class data.
- **num_labels**: (Optional) The number of labels, used when `multi_label` is True. If `num_labels` is not specified, then state variables get created on the first call to `update_state`.
- **label_weights**: (Optional) list, array, or tensor of non-negative weights used to compute AUCs for multilabel data. When `multi_label` is True, the weights are applied to the individual label AUCs when they are averaged to produce the multi-label AUC. When it's False, they are used to weight the individual label predictions in computing the confusion matrix on the flattened data. Note that this is unlike class_weights in that class_weights weights the example depending on the value of its label, whereas label_weights depends only on the index of that label before flattening; therefore `label_weights` should not be used for multi-class data.
- **from_logits**: boolean indicating whether the predictions (`y_pred` in `update_state`) are probabilities or sigmoid logits. As a rule of thumb, when using a keras loss, the `from_logits` constructor argument of the loss should match the AUC `from_logits` constructor argument.

Standalone usage:

```
>>> m = tf.keras.metrics.AUC(num_thresholds=3)
>>> m.update_state([0, 0, 1, 1], [0, 0.5, 0.3, 0.9])
>>> # threshold values are [0 - 1e-7, 0.5, 1 + 1e-7]
>>> # tp = [2, 1, 0], fp = [2, 0, 0], fn = [0, 1, 2], tn = [0, 2, 2]
>>> # tp_rate = recall = [1, 0.5, 0], fp_rate = [1, 0, 0]
>>> # auc = ((((1+0.5)/2)*(1-0)) + (((0.5+0)/2)*(0-0))) = 0.75
>>> m.result().numpy()
0.75
```

```
>>> m.reset_state()
>>> m.update_state([0, 0, 1, 1], [0, 0.5, 0.3, 0.9],
...                sample_weight=[1, 0, 0, 1])
>>> m.result().numpy()
1.0
```

Usage with `compile()` API:

```
# Reports the AUC of a model outputting a probability.
model.compile(optimizer='sgd',
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=[tf.keras.metrics.AUC()])

# Reports the AUC of a model outputting a logit.
model.compile(optimizer='sgd',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=[tf.keras.metrics.AUC(from_logits=True)])
```

`Precision` class

[source]

```
tf.keras.metrics.Precision(
    thresholds=None, top_k=None, class_id=None, name=None, dtype=None
)
```

Computes the precision of the predictions with respect to the labels.

The metric creates two local variables, `true_positives` and `false_positives` that are used to compute the precision. This value is ultimately returned as `precision`, an idempotent operation that simply divides `true_positives` by the sum of `true_positives` and `false_positives`.

If `sample_weight` is `None`, weights default to 1. Use `sample_weight` of 0 to mask values.

If `top_k` is set, we'll calculate precision as how often on average a class among the top-k classes with the highest predicted values of a batch entry is correct and can be found in the label for that entry.

If `class_id` is specified, we calculate precision by considering only the entries in the batch for which `class_id` is above the threshold and/or in the top-k highest predictions, and computing the fraction of them for which `class_id` is indeed a correct label.

Arguments

- **thresholds**: (Optional) A float value or a python list/tuple of float threshold values in [0, 1]. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is `true`, below is `false`). One metric value is generated for each threshold value. If neither thresholds nor top_k are set, the default is to calculate precision with `thresholds=0.5`.
- **top_k**: (Optional) Unset by default. An int value specifying the top-k predictions to consider when calculating precision.
- **class_id**: (Optional) Integer class ID for which we want binary metrics. This must be in the half-open interval `[0, num_classes)`, where `num_classes` is the last dimension of predictions.
- **name**: (Optional) string name of the metric instance.
- **dtype**: (Optional) data type of the metric result.

Standalone usage:

```
>>> m = tf.keras.metrics.Precision()
>>> m.update_state([0, 1, 1, 1], [1, 0, 1, 1])
>>> m.result().numpy()
0.6666667
```

```
>>> m.reset_state()
>>> m.update_state([0, 1, 1, 1], [1, 0, 1, 1], sample_weight=[0, 0, 1, 0])
>>> m.result().numpy()
1.0
```

```
>>> # With top_k=2, it will calculate precision over y_true[:2] and y_pred[:2]
>>> m = tf.keras.metrics.Precision(top_k=2)
>>> m.update_state([0, 0, 1, 1], [1, 1, 1, 1])
>>> m.result().numpy()
0.0
```

```
>>> # With top_k=4, it will calculate precision over y_true[:4] and y_pred[:4]
>>> m = tf.keras.metrics.Precision(top_k=4)
>>> m.update_state([0, 0, 1, 1], [1, 1, 1, 1])
>>> m.result().numpy()
0.5
```

Usage with `compile()` API:

```
model.compile(optimizer='sgd',
              loss='mse',
              metrics=[tf.keras.metrics.Precision()])
```

`Recall` class

[source]

```
tf.keras.metrics.Recall(
    thresholds=None, top_k=None, class_id=None, name=None, dtype=None
)
```

Computes the recall of the predictions with respect to the labels.

This metric creates two local variables, `true_positives` and `false_negatives`, that are used to compute the recall. This value is ultimately returned as `recall`, an idempotent operation that simply divides `true_positives` by the sum of `true_positives` and `false_negatives`.

Classification metrics based on True/False positives & negatives

If `sample_weight` is `None`, weights default to 1. Use `sample_weight` of 0 to mask values.

If `top_k` is set, recall will be computed as how often on average a class among the labels of a batch entry is in the top-k predictions.

If `class_id` is specified, we calculate recall by considering only the entries in the batch for which `class_id` is in the label, and computing the fraction of them for which `class_id` is above the threshold and/or in the top-k predictions.

Arguments

- **thresholds**: (Optional) A float value or a python list/tuple of float threshold values in [0, 1]. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is `true`, below is `false`). One metric value is generated for each threshold value. If neither thresholds nor top_k are set, the default is to calculate recall with `thresholds=0.5`.
- **top_k**: (Optional) Unset by default. An int value specifying the top-k predictions to consider when calculating recall.
- **class_id**: (Optional) Integer class ID for which we want binary metrics. This must be in the half-open interval `[0, num_classes)`, where `num_classes` is the last dimension of predictions.
- **name**: (Optional) string name of the metric instance.
- **dtype**: (Optional) data type of the metric result.

Standalone usage:

```
>>> m = tf.keras.metrics.Recall()
>>> m.update_state([0, 1, 1, 1], [1, 0, 1, 1])
>>> m.result().numpy()
0.6666667
```

```
>>> m.reset_state()
>>> m.update_state([0, 1, 1, 1], [1, 0, 1, 1], sample_weight=[0, 0, 1, 0])
>>> m.result().numpy()
1.0
```

Usage with `compile()` API:

```
model.compile(optimizer='sgd',
              loss='mse',
              metrics=[tf.keras.metrics.Recall()])
```

`TruePositives` class

[source]

```
tf.keras.metrics.TruePositives(thresholds=None, name=None, dtype=None)
```

Calculates the number of true positives.

If `sample_weight` is given, calculates the sum of the weights of true positives. This metric creates one local variable, `true_positives` that is used to keep track of the number of true positives.

If `sample_weight` is `None`, weights default to 1. Use `sample_weight` of 0 to mask values.

Arguments

- **thresholds**: (Optional) Defaults to 0.5. A float value or a python list/tuple of float threshold values in [0, 1]. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is `true`, below is `false`). One metric value is generated for each threshold value.
- **name**: (Optional) string name of the metric instance.
- **dtype**: (Optional) data type of the metric result.

Standalone usage:

```
>>> m = tf.keras.metrics.TruePositives()
>>> m.update_state([0, 1, 1, 1], [1, 0, 1, 1])
>>> m.result().numpy()
2.0
```

```
>>> m.reset_state()
>>> m.update_state([0, 1, 1, 1], [1, 0, 1, 1], sample_weight=[0, 0, 1, 0])
>>> m.result().numpy()
1.0
```

Usage with `compile()` API:

```
model.compile(optimizer='sgd',
              loss='mse',
              metrics=[tf.keras.metrics.TruePositives()])
```

`TrueNegatives` class [source]

```
tf.keras.metrics.TrueNegatives(thresholds=None, name=None, dtype=None)
```

Calculates the number of true negatives.

If `sample_weight` is given, calculates the sum of the weights of true negatives. This metric creates one local variable, `accumulator` that is used to keep track of the number of true negatives.

If `sample_weight` is `None`, weights default to 1. Use `sample_weight` of 0 to mask values.

Arguments

- **thresholds**: (Optional) Defaults to 0.5. A float value or a python list/tuple of float threshold values in [0, 1]. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is `true`, below is `false`). One metric value is generated for each threshold value.
- **name**: (Optional) string name of the metric instance.
- **dtype**: (Optional) data type of the metric result.

Standalone usage:

```
>>> m = tf.keras.metrics.TrueNegatives()
>>> m.update_state([0, 1, 0, 0], [1, 1, 0, 0])
>>> m.result().numpy()
2.0
```

```
>>> m.reset_state()
>>> m.update_state([0, 1, 0, 0], [1, 1, 0, 0], sample_weight=[0, 0, 1, 0])
>>> m.result().numpy()
1.0
```

Usage with `compile()` API:

```
model.compile(optimizer='sgd',
              loss='mse',
              metrics=[tf.keras.metrics.TrueNegatives()])
```

`FalsePositives` class [source]

```
tf.keras.metrics.FalsePositives(thresholds=None, name=None, dtype=None)
```

Calculates the number of false positives.

If `sample_weight` is given, calculates the sum of the weights of false positives. This metric creates one local variable, `accumulator` that is used to keep track of the number of false positives.

If `sample_weight` is `None`, weights default to 1. Use `sample_weight` of 0 to mask values.

Arguments

- **thresholds**: (Optional) Defaults to 0.5. A float value or a python list/tuple of float threshold values in [0, 1]. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is `true`, below is `false`). One metric value is generated for each threshold value.
- **name**: (Optional) string name of the metric instance.
- **dtype**: (Optional) data type of the metric result.

Standalone usage:

```
>>> m = tf.keras.metrics.FalsePositives()
>>> m.update_state([0, 1, 0, 0], [0, 0, 1, 1])
>>> m.result().numpy()
2.0
```

```
>>> m.reset_state()
>>> m.update_state([0, 1, 0, 0], [0, 0, 1, 1], sample_weight=[0, 0, 1, 0])
>>> m.result().numpy()
1.0
```

Usage with `compile()` API:

```
model.compile(optimizer='sgd',
              loss='mse',
              metrics=[tf.keras.metrics.FalsePositives()])
```

`FalseNegatives` class

[source]

```
tf.keras.metrics.FalseNegatives(thresholds=None, name=None, dtype=None)
```

Calculates the number of false negatives.

If `sample_weight` is given, calculates the sum of the weights of false negatives. This metric creates one local variable, `accumulator` that is used to keep track of the number of false negatives.

If `sample_weight` is `None`, weights default to 1. Use `sample_weight` of 0 to mask values.

Arguments

- **thresholds**: (Optional) Defaults to 0.5. A float value or a python list/tuple of float threshold values in [0, 1]. A threshold is compared with prediction values to determine the truth value of predictions (i.e., above the threshold is `true`, below is `false`). One metric value is generated for each threshold value.
- **name**: (Optional) string name of the metric instance.
- **dtype**: (Optional) data type of the metric result.

Standalone usage:

```
>>> m = tf.keras.metrics.FalseNegatives()
>>> m.update_state([0, 1, 1, 1], [0, 1, 0, 0])
>>> m.result().numpy()
2.0
```

```
>>> m.reset_state()
>>> m.update_state([0, 1, 1, 1], [0, 1, 0, 0], sample_weight=[0, 0, 1, 0])
>>> m.result().numpy()
1.0
```

Usage with `compile()` API:

```
model.compile(optimizer='sgd',
              loss='mse',
              metrics=[tf.keras.metrics.FalseNegatives()])
```

`PrecisionAtRecall` class

[source]

```
tf.keras.metrics.PrecisionAtRecall(
    recall, num_thresholds=200, class_id=None, name=None, dtype=None
)
```

Computes best precision where recall is >= specified value.

This metric creates four local variables, `true_positives`, `true_negatives`, `false_positives` and `false_negatives` that are used to compute the precision at the given recall. The threshold for the given recall value is computed and used to evaluate the corresponding precision.

If `sample_weight` is `None`, weights default to 1. Use `sample_weight` of 0 to mask values.

If `class_id` is specified, we calculate precision by considering only the entries in the batch for which `class_id` is above the threshold predictions, and computing the fraction of them for which `class_id` is indeed a correct label.

Arguments

- **recall**: A scalar value in range `[0, 1]`.
- **num_thresholds**: (Optional) Defaults to 200. The number of thresholds to use for matching the given recall.
- **class_id**: (Optional) Integer class ID for which we want binary metrics. This must be in the half-open interval `[0, num_classes)`, where `num_classes` is the last dimension of predictions.
- **name**: (Optional) string name of the metric instance.
- **dtype**: (Optional) data type of the metric result.

Standalone usage:

```
>>> m = tf.keras.metrics.PrecisionAtRecall(0.5)
>>> m.update_state([0, 0, 0, 1, 1], [0, 0.3, 0.8, 0.3, 0.8])
>>> m.result().numpy()
0.5
```

```
>>> m.reset_state()
>>> m.update_state([0, 0, 0, 1, 1], [0, 0.3, 0.8, 0.3, 0.8],
...                sample_weight=[2, 2, 2, 1, 1])
>>> m.result().numpy()
0.33333333
```

Usage with `compile()` API:

```
model.compile(
    optimizer='sgd',
    loss='mse',
    metrics=[tf.keras.metrics.PrecisionAtRecall(recall=0.8)])
```

`SensitivityAtSpecificity` class

[source]

```
tf.keras.metrics.SensitivityAtSpecificity(
    specificity, num_thresholds=200, class_id=None, name=None, dtype=None
)
```

Computes best sensitivity where specificity is >= specified value.

the sensitivity at a given specificity.

`Sensitivity` measures the proportion of actual positives that are correctly identified as such (tp / (tp + fn)). `Specificity` measures the proportion of actual negatives that are correctly identified as such (tn / (tn + fp)).

This metric creates four local variables, `true_positives`, `true_negatives`, `false_positives` and `false_negatives` that are used to compute the sensitivity at the given specificity. The threshold for the given specificity value is computed and used to evaluate the corresponding sensitivity.

If `sample_weight` is `None`, weights default to 1. Use `sample_weight` of 0 to mask values.

If `class_id` is specified, we calculate precision by considering only the entries in the batch for which `class_id` is above the threshold predictions, and computing the fraction of them for which `class_id` is indeed a correct label.

For additional information about specificity and sensitivity, see [the following].

Arguments

- **specificity**: A scalar value in range `[0, 1]`.
- **num_thresholds**: (Optional) Defaults to 200. The number of thresholds to use for matching the given specificity.
- **class_id**: (Optional) Integer class ID for which we want binary metrics. This must be in the half-open interval `[0, num_classes)`, where `num_classes` is the last dimension of predictions.
- **name**: (Optional) string name of the metric instance.
- **dtype**: (Optional) data type of the metric result.

Standalone usage:

```
>>> m = tf.keras.metrics.SensitivityAtSpecificity(0.5)
>>> m.update_state([0, 0, 0, 1, 1], [0, 0.3, 0.8, 0.3, 0.8])
>>> m.result().numpy()
0.5
```

```
>>> m.reset_state()
>>> m.update_state([0, 0, 0, 1, 1], [0, 0.3, 0.8, 0.3, 0.8],
...                sample_weight=[1, 1, 2, 2, 1])
>>> m.result().numpy()
0.333333
```

Usage with `compile()` API:

```
model.compile(
    optimizer='sgd',
    loss='mse',
    metrics=[tf.keras.metrics.SensitivityAtSpecificity()])
```

`SpecificityAtSensitivity` class

[source]

```
tf.keras.metrics.SpecificityAtSensitivity(
    sensitivity, num_thresholds=200, class_id=None, name=None, dtype=None
)
```

Computes best specificity where sensitivity is >= specified value.

`Sensitivity` measures the proportion of actual positives that are correctly identified as such (tp / (tp + fn)). `Specificity` measures the proportion of actual negatives that are correctly identified as such (tn / (tn + fp)).

This metric creates four local variables, `true_positives`, `true_negatives`, `false_positives` and `false_negatives` that are used to compute the specificity at the given sensitivity. The threshold for the given sensitivity value is computed and used to evaluate the corresponding specificity.

If `sample_weight` is `None`, weights default to 1. Use `sample_weight` of 0 to mask values.

If `class_id` is specified, we calculate precision by considering only the entries in the batch for which `class_id` is above the threshold predictions, and computing the fraction of them for which `class_id` is indeed a correct label.

For additional information about specificity and sensitivity, see the following.

Arguments

- **sensitivity**: A scalar value in range `[0, 1]`.
- **num_thresholds**: (Optional) Defaults to 200. The number of thresholds to use for matching the given sensitivity.
- **class_id**: (Optional) Integer class ID for which we want binary metrics. This must be in the half-open interval `[0, num_classes)`, where `num_classes` is the last dimension of predictions.
- **name**: (Optional) string name of the metric instance.
- **dtype**: (Optional) data type of the metric result.

Standalone usage:

```
>>> m = tf.keras.metrics.SpecificityAtSensitivity(0.5)
>>> m.update_state([0, 0, 0, 1, 1], [0, 0.3, 0.8, 0.3, 0.8])
>>> m.result().numpy()
0.66666667
```

```
>>> m.reset_state()
>>> m.update_state([0, 0, 0, 1, 1], [0, 0.3, 0.8, 0.3, 0.8],
...                sample_weight=[1, 1, 2, 2, 2])
>>> m.result().numpy()
0.5
```

Usage with `compile()` API:

```
model.compile(
    optimizer='sgd',
    loss='mse',
    metrics=[tf.keras.metrics.SpecificityAtSensitivity()])
```