# QuaPy: A Python-Based Framework for Quantification

Alejandro Moreo, Andrea Esuli, Fabrizio Sebastiani
Istituto di Scienza e Tecnologie dell'Informazione
Consiglio Nazionale delle Ricerche
56124, Pisa, Italy
firstname.lastname@isti.cnr.it

## ABSTRACT

QuaPy is an open-source framework for performing *quantification* (a.k.a. *supervised prevalence estimation*), written in Python. Quantification is the task of training *quantifiers* via supervised learning, where a quantifier is a predictor that estimates the *relative frequencies* (a.k.a. *prevalence values*) of the classes of interest in a sample of unlabelled data. While quantification can be trivially performed by applying a standard classifier to each unlabelled data item and counting how many data items have been assigned to each class, it has been shown that this "classify and count" method is outperformed by methods specifically designed for quantification. QuaPy provides implementations of a number of baseline methods and advanced quantification methods, of routines for quantification-oriented model selection, of several broadly accepted evaluation measures, and of robust evaluation protocols routinely used in the field. QuaPy also makes available datasets commonly used for testing quantifiers, and offers visualization tools for facilitating the analysis and interpretation of the results. The software is open-source and publicly available under a BSD-3 licence via GitHub[1], and can be installed via `pip`[2].

## CCS CONCEPTS

• **Computing methodologies** → **Supervised learning**; • **Software and its engineering** → **Development frameworks and environments**; • **Information systems** → *Data mining*.

## KEYWORDS

Quantification, Supervised Prevalence Estimation, Learning to Quantify, Supervised Learning, Python, Open Source

**ACM Reference Format:**
Alejandro Moreo, Andrea Esuli, Fabrizio Sebastiani. 2021. QuaPy: A Python-Based Framework for Quantification. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management (CIKM '21), November 1–5, 2021, Virtual Event, QLD, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3459637.3482015

[1]https://github.com/HLT-ISTI/QuaPy
[2]https://pypi.org/project/QuaPy/

## 1 INTRODUCTION

*Quantification* (variously called *learning to quantify*, or *supervised prevalence estimation*, or *class prior estimation*) is the task of training models ("quantifiers") that estimate the *relative frequencies* (a.k.a. *prevalence values*) of the classes of interest in a sample of unlabelled data items [17]. For instance, in a sample of 100,000 unlabelled tweets known to express opinions about Donald Trump, such a model may be tasked to estimate the percentage of these 100,000 tweets which display a Positive stance towards Trump (and to do the same for classes Neutral and Negative). In other words, quantification stands to classification as aggregate data stand to individual data. Quantification is of special interest in fields such as the social sciences [19], epidemiology [21], market research [11], and ecological modelling [2], since these fields are inherently concerned with aggregate data; however, quantification is also useful in applications outside these fields, such as in enforcing the fairness of classifiers [4], performing word sense disambiguation [5], allocating resources [15], and improving the accuracy of classifiers [31].

Quantification can trivially be solved via classification, i.e., by training a classifier, applying it to the unlabelled data items, and counting how many data items have been assigned to each class. However, there is by now abundant evidence [17] that this "classify and count" method delivers suboptimal quantification accuracy, and especially so in scenarios characterized by *distribution shift*, i.e., by the fact that the class prevalence values in the training set are substantially different from those in the set of unlabelled data. As a result, quantification is no more considered just a by-product of classification, and has evolved as a task in its own right; as such, quantification has its own learning methods, model selection protocols, evaluation measures, and evaluation protocols.

In this paper we present QuaPy, a framework written in Python that provides implementations of the most important tools for research, development, and experimentation, in quantification. The following script can serve as a minimal working example of how QuaPy is used. The script fetches a dataset of tweets, trains a quantifier via the *Adjusted Classify and Count* method (that is meant to improve the prevalence estimates returned by a standard classifier, here trained via logistic regression), and then evaluates the quantifier in terms of the *Absolute Error* (AE) between the estimated and the true class prevalence values of the test set.

```python
import quapy as qp
from sklearn.linear_model import LogisticRegression

data = qp.datasets.fetch_twitter('semeval16')

# create an "Adjusted Classify & Count" quantifier
model = qp.method.aggregative.ACC(LogisticRegression())
model.fit(data.training)

estim_prevalence = model.quantify(data.test.instances)
```

```
  true_prevalence  = data.test.prevalence()

  error = qp.error.ae(true_prevalence, estim_prevalence)
  print('Absolute Error (AE)', error)
```

As mentioned above, quantification is particularly useful in scenarios where distribution shift may occur. Any quantification model should thus be tested across different data samples characterized by different class prevalence values. QuaPy implements sampling procedures and evaluation protocols that automate this endeavour.

Some among the authors who have published papers on the field of quantification have also made available software packages implementing their methods and baselines. However, such software repositories are often tied to specific applicative domains[3], are limited to reproduce experimental results from specific papers[4], or lack proper documentation and wiki references[5][6][7]. While all these implementations represent valuable resources that demonstrate how to implement and use specific algorithms, to the best of our knowledge none among the existing software packages strive to define a proper framework that jointly caters for all steps of the quantification pipeline, from data preparation to the visualization of results, in a unified way. QuaPy is a flexible and extensible framework that aims at filling this gap.

The paper is structured as follows. In Section 2 we briefly describe the quantifier training methods included in QuaPy, while in Section 3 we present a number of datasets that have been previously used in quantification research and that we include in the QuaPy suite. Section 4 is devoted to quantifier evaluation, and discusses the evaluation measures and evaluation protocols that we make available within QuaPy. Section 5 turns to model selection, discussing the hyperparameter optimization protocols implemented within QuaPy, while Section 6 illustrates the tools that we make available for visualizing the results of quantification experiments. Section 7 discusses some experiments that we have carried out in order to showcase some among the features of QuaPy. In Section 8 we give some concluding remarks.

## 2 METHODS

A quantifier is defined in QuaPy as a model that can be `fit` on some training data, so that the fitted model can estimate class prevalence values for unlabelled data. More specifically, a quantifier in QuaPy must inherit from the class `BaseQuantifier`, and implement the following abstract methods:

```
  @abstractmethod
  def fit(self, data: LabelledCollection): ...

  @abstractmethod
  def quantify(self, instances): ...

  @abstractmethod
  def set_params(self, **parameters): ...

  @abstractmethod
  def get_params(self, deep=True): ...
```

The meaning of these functions should be familiar to anybody accustomed to the `scikit-learn` environment [27], since the class structure of QuaPy is directly inspired by `scikit-learn`'s "estimators".[8] Functions `fit` and `quantify` are used to train the model and to return class prevalence estimates, respectively, while functions `set_params` and `get_params` allow a model-selecting routine (see Section 5) to automate the process of hyperparameter optimization.

Quantification methods can be classified as belonging to the aggregative, *non-aggregative*, or *meta* classes. Aggregative methods are characterized by the fact that quantification is obtained as an aggregation of the outputs returned by a classification process for the individual documents. Non-aggregative methods analyse instead the sample of unlabelled documents as a whole, without resorting to the classification of individual data items. Finally, meta-quantifiers are built on top of other quantifiers, and generate their predictions by analysing the predictions made by the underlying quantifiers. We will briefly present these three classes in the next three subsections.

### 2.1 Aggregative methods

Most of the methods proposed in the literature and included in QuaPy are aggregative. QuaPy models aggregative quantifiers by means of the abstract class `AggregativeQuantifier`. This class extends `BaseQuantifier`, providing a default implementation of the `quantify` method based on the `aggregate` function, that has to be implemented, i.e.,

```
  def quantify(self, instances):
      classif_predictions = self.classify(instances)
      return self.aggregate(classif_predictions)

  @abstractmethod
  def aggregate(self, classif_predictions:np.ndarray): ...
```

Implementing an aggregative method only requires overriding the `aggregate` method. The `AggregativeQuantifier` class implements the rest of the process, and is designed to work with any `scikit-learn` estimator. Working with packages or machine learning tools other than `scikit-learn` only requires overriding the `classify` method, which takes as input the individual data items in the sample and returns the corresponding classification predictions (see Section 2.1.5).

*Probabilistic* aggregative methods are a subclass of aggregative methods, which, instead of the "crisp" decisions returned by a categorical classifier, use the posterior probabilities returned by a probabilistic classifier. Probabilistic aggregative methods inherit from the abstract class `AggregativeProbabilisticQuantifier`, which extends `AggregativeQuantifier`, by providing a default implementation of the `quantify` method as follows:

```
  def quantify(self, instances):
      classif_posteriors = self.posterior_probabilities(instances)
      return self.aggregate(classif_posteriors)
```

The method `posterior_probabilities`, similarly to the more general case, is designed to work together with the `predict_proba` method of any probabilistic classifier in `scikit-learn`. QuaPy also allows using the `scikit-learn`'s crisp estimators that do not come with an implementation of the `predict_proba` method (e.g.,

[3]https://gking.harvard.edu/va
[4]https://github.com/afonsofvaz/ratio_estimator
[5]https://gitlab.com/andregustavom/mlq_framework/
[6]https://github.com/alliedel/class_prior_estimation
[7]https://github.com/tobiasschumacher/quantification_paper

[8]QuaPy's quantifiers do not inherit from `scikit-learn`'s estimators due to one key difference that makes the two incompatible. While a `scikit-learn` estimator's `predict` method is expected to produce an array of $c$ predictions (with $c$ the number of classes) for each of the $n$ data items in the input, the quantifier's `quantify` method is instead requested to output one single vector of $c$ prevalence values for a given sample of data items.

LinearSVC). In this case, the estimator is converted into a probabilistic classifier by means of a *calibration* method [28].[9] Packages other than scikit-learn can be used as well by providing a custom implementation of the posterior_probabilities method (see Section 2.1.5).

One advantage of aggregative methods (probabilistic or not) is that the evaluation according to any sampling procedure (e.g., the artificial prevalence protocol – see Section 4) can be carried out very efficiently, since the entire set of unlabelled items can be pre-classified once for all at the beginning, and the estimation of class prevalence values for different samples can directly reuse these predictions, with no need to reclassify each individual data item every time. QuaPy takes advantage of this property to drastically speed up any routine that has to do with quantification on multiple samples drawn from the same set, as is customarily the case in quantification, both in the performance evaluation phase (Section 4) and in the model selection phase (Section 5).

#### 2.1.1 *Classify & Count and its variants.*

QuaPy provides implementations for *Classify & Count* (CC) and its variants, i.e.,

- CC (Classify & Count), the simplest aggregative quantifier, that simply relies on the label predictions of a classifier to deliver class prevalence estimates;
- ACC (Adjusted Classify & Count) [15], the "adjusted" variant of CC, that corrects the predictions of CC according to the "misclassification rates" (see below) of the classifier;
- PCC (Probabilistic Classify & Count) [3], the probabilistic variant of CC that relies on the posterior probabilities returned by a probabilistic classifier;
- PACC (Probabilistic Adjusted Classify & Count) [3], which stands to PCC as ACC stands to CC.

Note that the adjusted variants (ACC and PACC) need to estimate the parameters (the "misclassification rates") required for performing the adjustment; the estimation uses a validation set carved out of the labelled set. The specific form of parameter optimization can be set at construction time or at fitting time using the argument val_split, either by indicating a float in (0,1) specifying the fraction of the training data to be used as a held-out validation set, or by indicating an int specifying the number of folds to be used in a $k$-fold cross-validation ($k$-FCV) process, or by explicitly passing a set of instances to be used as the validation set (i.e., an instance of LabelledCollection – see Section 3).

#### 2.1.2 *Forman's variants of ACC.*

QuaPy also provides implementations of a series of binary quantification methods, proposed by Forman in [14, 15] as variations of ACC, and whose goal is to bring improved stability to the denominator of the adjustment.[10] The methods are based on different heuristics for choosing a decision threshold that would allow for more true positives and many more false positives, on the grounds this would deliver larger denominators.

QuaPy implements the methods X (which looks for the threshold that yields $\mathrm{tpr}(y) = 1 - \mathrm{fpr}(y)$), MAX (which looks for the threshold that maximizes $\mathrm{tpr}(y) - \mathrm{fpr}(y)$), T50 (which looks for the threshold that makes $\mathrm{tpr}(y)$ closest to 0.5). QuaPy also implements MS (Median Sweep), a method that generates class prevalence estimates for all decision thresholds and returns the median of them all; and MS2, a variant that computes the median only for cases in which $\mathrm{tpr}(y) - \mathrm{fpr}(y) > 0.25$.

#### 2.1.3 *The Saerens-Latinne-Decaestecker algorithm.*

The Saerens-Latinne-Decaestecker (SLD) algorithm [8, 31] (sometimes also called EMQ, for *Expectation Maximization Quantifier*) is a probabilistic quantifier-generating method. SLD consists of using the well-known Expectation Maximization algorithm to iteratively update the posterior probabilities generated by a probabilistic classifier and the class prevalence estimates obtained via maximum-likelihood estimation, in a mutually recursive way, until convergence. Although this method was originally proposed for improving the quality of the posterior probabilities returned by a probabilistic classifier, and not for improving its class prevalence estimates, SLD has proven to be among the most effective quantifiers in many experiments [24, 25, 32].

#### 2.1.4 *The HDy method.*

HDy [18] is a probabilistic method for training binary quantifiers, that models quantification as the problem of minimizing the divergence (in terms of the *Hellinger Distance*) between two cumulative distributions of posterior probabilities returned by the classifier. One of the distributions is generated from the unlabelled examples and the other is generated from a validation set. This latter distribution is defined as a mixture of the class-conditional distributions of the posterior probabilities returned for the positive and negative validation examples, respectively. The parameters of the mixture thus represent the estimates of the class prevalence values.

Since the method requires a validation set to estimate the parameters of the mixture model, the constructor and fit method of HDy receive as input the argument val_split, whose semantics is the same as in ACC and PACC.

#### 2.1.5 *Quantifiers based on Explicit Loss Minimization.*

The quantifiers based on *Explicit Loss Minimization* (ELM) represent a family of methods based on structured output learning; these quantifiers rely on classifiers that have been optimized using a quantification-oriented loss measure. QuaPy implements the following ELM-based methods, all relying on Joachims' SVM$^{\mathrm{perf}}$ structured output learning algorithm [20]:[11]

- SVM(Q), which attempts to minimize the $Q$ loss, that combines a classification-oriented loss and a quantification-oriented loss, as proposed in [1];
- SVM(KLD), which attempts to minimize the Kullback-Leibler Divergence, as proposed in [12] and as first used in [13];
- SVM(NKLD), which attempts to minimize a version of the Kullback-Leibler Divergence normalized via the logistic function, as first used in [13];

[9]In QuaPy this is automatically done by wrapping the estimator in the CalibratedClassifierCV class.

[10]In the binary case, the ACC adjustment comes down to computing $\hat{p}^{\mathrm{ACC}}(y) = \frac{\hat{p}^{\mathrm{CC}}(y) - \hat{\mathrm{fpr}}(y)}{\hat{\mathrm{tpr}}(y) - \hat{\mathrm{fpr}}(y)}$ in which $\hat{p}^{\mathrm{CC}}(y)$ is the prevalence of class $y$ as estimated by CC, and $\hat{\mathrm{tpr}}(y)$ and $\hat{\mathrm{fpr}}(y)$ stand for the *true positive rate* and *false positive rate* of the classifier, as estimated in the validation phase. The above-mentioned numerical instability arises when $\hat{\mathrm{tpr}}(y) \approx \hat{\mathrm{fpr}}(y)$.

[11]QuaPy includes the tools to automatically patch the original SVM$^{\mathrm{perf}}$ code in order to add the quantification-oriented loss functions.

- SVM(AE), which uses Absolute Error as the loss, as first used in [25];
- SVM(RAE), which uses Relative Absolute Error as the loss, as first used in [25].

All ELM-based methods can train binary quantifiers only, since they rely on SVM$^{\text{perf}}$, which is an inherently binary system. However, QuaPy allows the conversion of binary quantifiers into multi-class quantifiers (see Section 2.3).

## 2.2 Methods for training meta-quantifiers

Meta-quantifiers base their estimates on the estimates produced by other quantifiers, and are defined in the `qp.method.meta` module.

*2.2.1 Ensembles:* A quantification ensemble receives as input any quantification method (any instance of `BaseQuantifier`). QuaPy implements some among the "ensemble" variants proposed in [29, 30], that train different members of the ensemble using different samples of the original training set; in particular:

- `Averaging` (`policy='ave'`, default): computes class prevalence estimates as the average of the estimates returned by the base quantifiers.
- `Training Prevalence` (`policy='ptr'`): applies a dynamic selection to the ensemble's members by retaining only those members such that the class prevalence values in the samples they use as training set are closest to preliminary class prevalence estimates computed as the average of the estimates of all the members. The final estimate is recomputed by considering only the selected members.
- `Distribution Similarity` (`policy='ds'`): performs a dynamic selection of base members by retaining the members trained on samples whose distribution of posterior probabilities is closest, in terms of the Hellinger Distance, to the distribution of posterior probabilities in the test sample;
- `Performance` (`policy='<any-error-metric>'`): performs a static selection of the ensemble members by retaining those that minimize a quantification error measure, which is passed as an argument.

When using either dynamic or static selection policies, one has to set the `red_size` parameter, which defines the number of members that have to be retained.

*2.2.2 The QuaNet recurrent quantifier:* QuaPy provides an implementation of QuaNet, a deep-learning-based method for performing quantification on samples of textual documents, presented in [9] (although it might well be used for other types of data).[12] QuaNet processes as input a list of document embeddings (see below), one for each unlabelled document along with their posterior probabilities generated by a probabilistic classifier. The list is processed by a bidirectional LSTM that generates a sample embedding (i.e., a dense representation of the entire sample), which is then concatenated with a vector of class prevalence estimates produced by an ensemble of simpler quantification methods (CC, ACC, PCC, PACC, SLD). This vector is then transformed by a set of feed-forward layers, followed by ReLU activations and dropout, to compute the final estimations.

[12]In order to use QuaNet within QuaPy, the `torch` framework for deep learning [26] has to be installed.

QuaNet thus requires a probabilistic classifier that can provide embedded representations of the inputs. QuaPy offers a basic implementation of such a classifier, based on convolutional neural networks, that returns its next-to-last representation as the document embedding. The following is a working example showing how to index a textual dataset (see Section 3) and how to instantiate QuaNet:

```python
import quapy as qp
from quapy.method.meta import QuaNet
from classification.neural import NeuralClassifierTrainer, CNNnet

qp.environ['SAMPLE_SIZE'] = 500

# load the kindle dataset as plain text, and convert words
# to numerical indexes
dataset = qp.datasets.fetch_reviews('kindle', pickle=True)
qp.data.preprocessing.index(dataset, min_df=5, inplace=True)

cnn = CNNnet(dataset.vocabulary_size, dataset.n_classes)
learner = NeuralClassifierTrainer(cnn, device='cuda')
model = QuaNet(learner, sample_size=500, device='cuda')
```

## 2.3 Using binary quantifiers in multi-class quantification

QuaPy allows a set of binary quantifiers, one for each class, to be assembled into a single-label multi-class quantifier, by adopting a "one-vs-all" strategy. This takes the form of computing prevalence estimates independently for each class (i.e., via binary quantification) via independently trained binary quantifiers, and then normalizing the resulting vector of prevalence values (via L1-normalization) so that these values sum up to one. In QuaPy this is possible by wrapping any binary quantifier within a `OneVsAll` object. For example, a quantifier defined as `model=OneVsAll(SVMQ())` will allow `SVMQ` to work with single-label multiclass datasets.

## 3 DATASETS

QuaPy makes available a number of datasets that have been used for experimentation purposes in the quantification literature, and specifically:[13]

- **Reviews**: a collection of 3 datasets of customer reviews about (1) Kindle devices (Kindle), (2) the Harry Potter's book series (HP), both already used in [13], and (3) the well-known IMDB movie reviews dataset (IMDB) [22]. All reviews are classified according to (binary) sentiment polarity. The number of training documents range from 3821 (Kindle) to 25000 (IMDB) and present examples in which labelled data are balanced (IMDB, 50% positives), imbalanced (Kindle, 92% positives), and severely imbalanced (HP, 98% positives).
- **Twitter Sentiment**: 11 datasets of tweets labelled by sentiment, as used in [16]. The raw text of the tweets is not available due to Twitter's Terms of Service, and tweets are instead provided as *tf-idf*-weighted vectors. Similarly to the Reviews datasets, these are high-dimensional datasets, with dimensionalities ranging from 199,151 to 1,215,742. These datasets use three sentiment labels (Positive, Neutral, Negative), and are thus useful for testing non-binary quantification methods.

[13]All these datasets have a corresponding `fetch` method in QuaPy that automatically downloads the dataset from a public repository, and caches it for reuse.

- **UCI**: 33 binary datasets from the UCI Machine Learning repository [6], as used in [30].[14] Differently from the previous datasets, these non-textual datasets are low-dimensional (with dimensionalities ranging from 3 to 256), thus providing diversity, in terms of type of data, with respect to to the previous two sets of datasets.

QuaPy defines a simple `Dataset` interface that allows importing any custom dataset into the QuaPy environment. A `Dataset` object in QuaPy is essentially a pair of `LabelledCollection` objects, playing the role of the training set and of the test set, respectively. `LabelledCollection` is a data class consisting of the instances and labels. This class implements the core sampling functionality in QuaPy, which is then exploited by the evaluation tools (Section 4.2) and by the model selection tools (Section 5).

From a `LabelledCollection`, QuaPy allows to easily produce new samples at desired class prevalence values, i.e.,

```
sample_size = 100
prev = [0.4, 0.1, 0.5]  # prevalence values for 3 classes
sample = data.sampling(sample_size, *prev)
```

QuaPy supports the definition of samples consistent across runs, in order to allow testing different quantification methods on the very same samples.

## 4 EVALUATION

Evaluating a quantifier requires measuring how good it is at predicting the class prevalence values of a test sample, which may have different class prevalence values than those observed on the training data.

The evaluation of quantifiers is a complex task, since it depends on many aspects.

For example, the same difference, in absolute value, between the true and the predicted prevalence values may have a different "cost" depending on the original true prevalence value: predicting 0.5 prevalence when the true prevalence is 0.49 can be considered, in some application contexts, a less blatant error than predicting a prevalence of 0.01 when the true prevalence is 0.00. In some other application contexts, though, the two above-mentioned estimation errors may be considered equally serious [33]. This means that sometimes we may want to use a certain evaluation measure and some other times we may want to use a different one.

Additionally, for some application contexts we may be interested in measuring the quantification error only on samples whose class prevalence values do not differ too much from those of the training set, because we assume distribution shift, in practice, to always be limited in magnitude. Conversely, in some other application contexts, we may want to test our quantifiers also in situations characterized by extreme values of distribution shift, because we expect our environment to be characterized by high variability, and because we want our quantifiers to be robust also to possibly extreme amounts of shift.

As a result, an environment for experimenting with quantification must not only be endowed with several evaluation measures, but it also must allow the experimentation to be carried out according to different evaluation protocols.

[14]Some of these datasets, in the original form as made available in the UCI Machine Learning repository, are not binary, but the authors of [30] have transformed each $n$-ary dataset into $n$ binary datasets according to a "one-vs-all" policy; the datasets we make available are the binary ones as generated by the authors of [30].

### 4.1 Error measures

Several error measures have been proposed in the literature [33], and QuaPy implements a rich set of them:

- `ae`: absolute error
- `rae`: relative absolute error
- `se`: squared error
- `kld`: Kullback-Leibler Divergence
- `nkld`: normalized Kullback-Leibler Divergence

Functions `mae`, `mrae`, `mse`, `mkld`, and `mnkld` are also available, which return the average values of the same measures across different samples. For aggregative quantifiers, also the $F_1$ and "vanilla accuracy" measures are available for measuring the quality of the underlying classifiers.

Some error functions, e.g., `rae`, `kld`, and `nkld`, and their averaged versions, are undefined for extreme prevalence values (i.e., 0 and 1), and are numerically unstable for prevalence values close to these extremes. A common solution to this problem is to perform smoothing, i.e., adding to each (true or predicted) prevalence value a small amount, and then normalizing. A traditional smoothing value from the literature is $1/2T$, where $T$ is the size of the sample. QuaPy supports setting the smoothing value as an environment variable (`qp.environ['SAMPLE_SIZE']`), or passing it as an argument of the error measure.

### 4.2 Evaluation protocols

QuaPy implements both the *natural prevalence protocol* (NPP) and the *artificial prevalence protocol* (APP).

In the NPP, the test set is sampled randomly, so that most samples exhibit class prevalence values not to different from those of the test set.

In the APP, the test set is instead sampled in a controlled way, in order to generate samples characterized by different, pre-specified prevalence values, so as to cover, with uniform probability, the full spectrum of class prevalence values. In the APP the user specifies the number of equidistant points to be generated from the interval [0,1]. For example, if `n_prevs=11` then, for each class, the prevalence values [0.0, 0.1, ..., 0.9, 1.0] will be used. This means that, for two classes, the number of different sampled prevalence values will be 11 (since, once the prevalence of one class is determined, the other one is also). For 3 classes, the number of valid combinations can be obtained as 11 + 10 + ... + 1 = 66. The number of valid combinations (i.e., that sum up to one) that will be produced for a given value of `n_prevpoints` across `n_classes` can be determined by invoking `quapy.functional.num_prevalence_combinations`, e.g.:

```
import quapy.functional as F
n_prevs = 21  # [0, 0.05, 0.1, ..., 0.95, 1]
n_classes = 4
repeats = 1
n = F.num_prevalence_combinations(n_prevs, n_classes, repeats)
```

In this example, $n = 1771$. The last argument, `n_repeats`, sets the number of samples that will be generated for any valid combination (typical values are 10 or higher, in order to support the computation of standard deviations and to perform statistical significance tests).

One can instead work the other way around, i.e., set an evaluation budged so as to obtain the number of prevalence values that will generate a number of samples close but no higher than the fixed budget, e.g.:

```python
budget = 5000
n_classes = 4
repeats = 1
n_prevs = F.get_nprevpoints_approximation(budget, n_classes,
↪  repeats)
n = F.num_prevalence_combinations(n_prevs, n_classes, repeats)
```

Here the function `get_nprevpoints_approximation` determines that for the given budget and 4 classes, by setting `n_prevpoints`= 30 the number of samples will be `n`= 4960.

QuaPy implements evaluation functions that allow the user to either specify the `n_prevpoints` value or an evaluation budget. The script shown in Section 5 shows a full example in which a PCC model relying on a classifier trained via logistic regression, is tested on the HP dataset by means of the APP protocol on samples of size 500, setting a budget of 1000 test samples, in terms of various evaluation metrics (`mae`, `mrae`, `mkld`).

## 5 MODEL SELECTION

Quantification has long been regarded as a by-product of classification, which means that the model selection (i.e., hyperparameter optimization) strategies customarily adopted in quantification have simply been borrowed from classification. It has been argued in [25] that specific model selection strategies should be adopted for quantification. That is, model selection strategies for quantification should minimize quantification-oriented loss measures, and be carried out on a variety of scenarios exhibiting different degrees of distribution shift.

QuaPy supports quantification-oriented model selection by implementing, in the class `qp.model_selection.GridSearchQ`, a grid-search exploration over the space of hyperparameter combinations that evaluates each such combination by means of a given quantification-oriented error metric (see Section 4.1), and according to either the APP (the default value) or the NPP.

The following is an example of quantification-oriented model selection using `GridSearchQ`. In this example, model selection is performed with a fixed budget of 1000 evaluations for each combination of hyperparameters. The loss function to miminize is MAE, a quantification-oriented error measure, as evaluated on randomly drawn samples at equidistant prevalence values covering the entire spectrum (APP protocol) on a stratified held-out portion consisting of 40% of the training set. [15]

```python
import quapy as qp
from quapy.method.aggregative import PCC
from sklearn.linear_model import LogisticRegression
import numpy as np

# setting this environment variable allows some
# error metrics (e.g., mrae) to be smoothed
qp.environ["SAMPLE_SIZE"] = 500

dataset = qp.datasets.fetch_reviews('hp', tfidf=True, min_df=5)

# model selection with the APP
model = qp.model_selection.GridSearchQ(
    model=PCC(LogisticRegression()),
    param_grid={'C': np.logspace(-4,5,10),
                'class_weight': ['balanced', None]},
    sample_size=500,
    protocol='app',
    eval_budget=1000,
    error='mae',
    refit=True,  # retrain on the whole labelled set once done
    val_split=0.4,
).fit(dataset.training)

df = qp.evaluation.artificial_prevalence_report(
    model,  # the quantification method
    dataset.test,  # the set on which the method will be evaluated
    sample_size=500,  # indicates the size of samples to be drawn
    eval_budget=1000,  # total number of samples to generate
    n_repetitions=10,  # number of samples for each prevalence
    n_jobs=-1,  # the number of parallel workers (-1 for all CPUs)
    random_seed=42,  # allows replicating test samples across runs
    error_metrics=['mae', 'mrae', 'mkld'],  # evaluation metrics
    verbose=True  # set to True to show some standard-line outputs)

print(f'best hyper-params={model.best_params_}')
print(df)
```

[15]Classification-oriented model selection can be done in QuaPy for aggregative quantifiers by simply using `scikit-learn`'s `GridSearchCV` method on the base Estimator.

In this example, the system prints the selected hyper-parameters and a `pandas` dataframe containing the evaluation report:

```
best hyper-params={'C': 0.1, 'class_weight': 'balanced'}
       true-prev   estim-prev    mae    mrae  mkld
  0 [0.0, 1.0] [0.000, 1.000] 0.000  0.000 0.000
  1 [0.0, 1.0] [0.000, 1.000] 0.000  0.000 0.000
...   ...   ...
998 [1.0, 0.0] [0.914, 0.086] 0.086 43.243 0.086
999 [1.0, 0.0] [0.906, 0.094] 0.094 47.069 0.094
```

## 6 RESULT VISUALIZATION

QuaPy implements some plotting functions that can be useful in displaying the performance of the tested quantification methods:

- **Diagonal plot**: The diagonal plot shows a very insightful view of the quantifier's performance, i.e., it plots the predicted class prevalence (on the y-axis) against the true class prevalence (on the x-axis), averaging across all samples characterized by the same true prevalence. Unfortunately, this visualization device is inherently limited to binary quantification (one can simply generate as many diagonal plots as there are classes, though, by indicating which class should be considered the target of the plot).
- **Error-by-Shift plot**: This plot displays the quantification error made by a quantifier as a function of the distribution shift between the training set and the test sample, averaging across all samples characterized by the same amount of distribution shift. Both quantification error and distribution shift can be measured in terms of any measure among those described in Section 4, and can be computed and plotted both in the binary case and in the non-binary case.
- **Bias-Box plot**: This plot aims at displaying, by means of box plots, the bias that any quantifier exhibits with respect to the training class prevalence values. The bias can be broken down into different bins, e.g., distinguishing the bias in cases of low, medium, and high prevalence shift.

In Figure 1 we show examples of each of the above types of plot, as resulting from the experiments that we will discuss in Section 7.

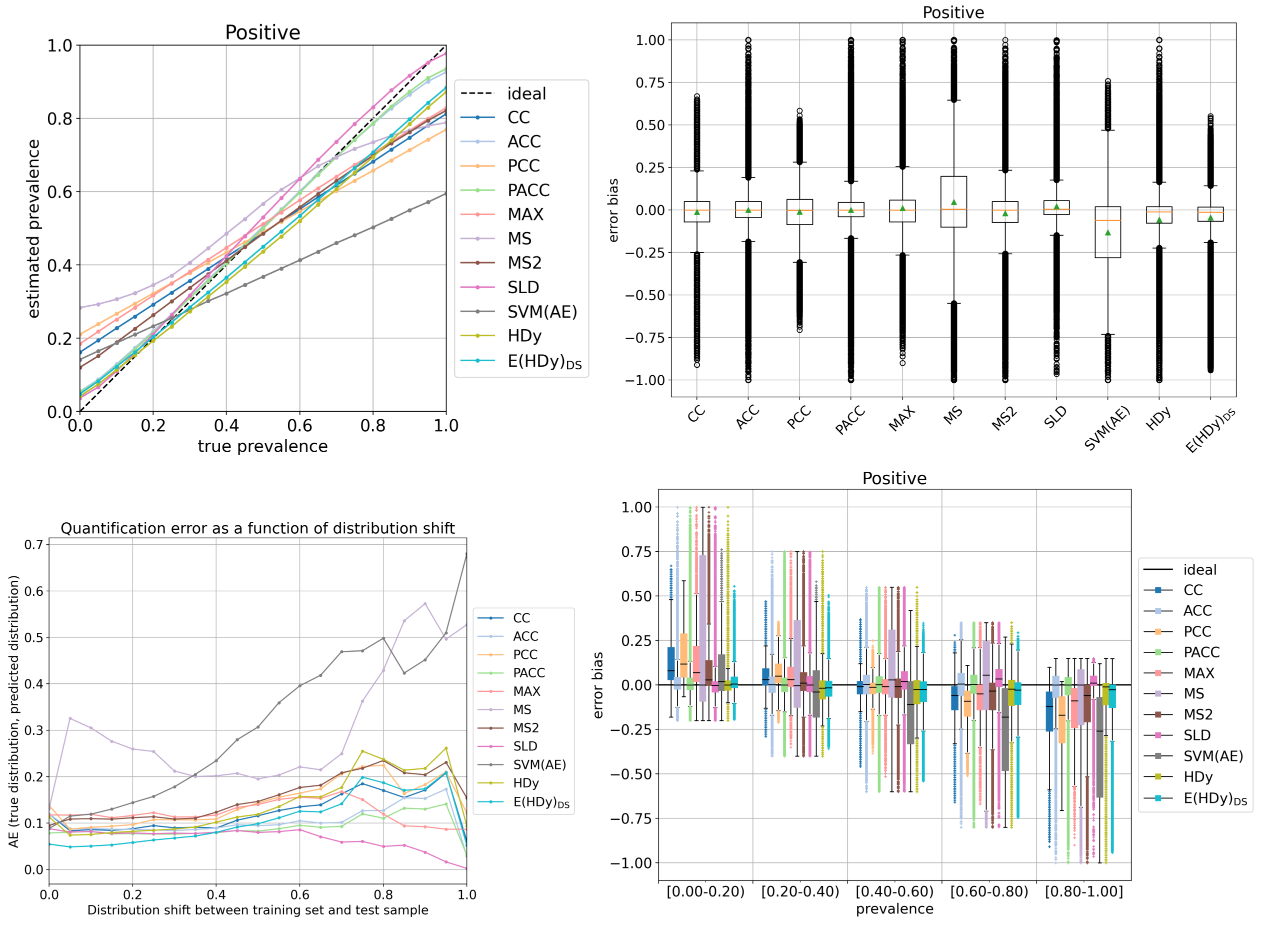

**Figure 1: Examples of plots generated by QuaPy: Diagonal plot (top left), Error-by-Shift plot (bottom left), Global Bias-Box plot (top right), and Local (5 bins) Bias-Box plot (bottom right).**

## 7 EXPERIMENTS

In this section we present some experiments that we have carried out in order to showcase some among the features of QuaPy. The code to replicate all these experiments, and to generate the relative tables and plots, can be accessed via GitHub.[16]

As the datasets, we consider the set of UCI Machine Learning datasets used in [30], consisting of 30 binary datasets (see Section 3).[17] Following [30], we remove the "frustratingly easy" datasets `acute.a`, `acute.b`, and `iris.1`, where even a trivial CC approach manages to yield zero quantification error. The datasets do not come with a predefined train/test split; we thus carry out an evaluation based on 5-fold cross-validation and report the average quantification error across the 5 test folds. Each iteration thus defines a training set $L$ (4 folds) and a test set $U$ (1 fold). We choose AE as our error metric and adopt the APP protocol for evaluation. For each method and test set $U$ we generate $m = 100$ different random samples of $q = 100$ instances each, at prevalence values in the range $[0.00, 0.05, \ldots, 0.95, 1.00]$ via selective undersampling, and report the resulting MAE value. Each MAE value we report corresponds to the average of 10,500 experiments (100 samples $\times$ 21 class prevalence values $\times$ 5 folds).

For model selection, we split the training set $L$ into a proper training set $L_{\text{Tr}}$ (consisting of 60% of $L$) and a held-out validation set $L_{\text{Va}}$ (the remaining 40%) in a stratified way. For each combination of hyperparameters we train the model using $L_{\text{Tr}}$ and evaluate the performance on $L_{\text{Va}}$ in terms of MAE by following the APP protocol [25]; in this case we use $q = 100$ and $m = 25$. Once the best values of the hyperparameters have been identified, we re-train the method using the entire training set.

All quantifiers we consider here are either aggregative quantifiers or ensembles of aggregative base quantifiers, which means that all of them rely on an underlying classifier. We consider Logistic Regression (LR) as our default classifier-training algorithm in all cases, except for the methods from the "explicit loss minimization" camp,

[16]See the files `uci_experiments.py` (runs all experiments), `uci_tables.py` (generates Table 1 directly in LaTeX), and `uci_plots.py` (generates all plots from Figure 1) included in the folder `wiki_examples/` of the repository https://github.com/HLT-ISTI/QuaPy.wiki.git

[17]In their study, [30] used 32 datasets. However, we have not been able to locate datasets "diabetes" and "phoneme" in the UCI ML repository.

which instead natively rely on SVM$^{\text{perf}}$. The set of hyperparameters to optimize include the regularization parameter $C$ (common to LR and SVMs), taking values in $\{10^{-3}, 10^{-2}, \ldots, 10^{2}, 10^{3}\}$, and the parameter `class_weight` (only for LR), which may take values `balanced` (which has the effect of giving more weight to test examples from less frequent classes) or `None` (which has the effect of giving the same weight to all test examples).

As the learning methods we consider CC, its variants PCC, ACC, PACC, Forman's variants[18] MAX, MS, MS2, the expectation-maximization-based SLD method,[19] the mixture model HDy, SVM(AE) as the representative of the "explicit loss minimization" family[20], and E(HDy)$_{\text{DS}}$ as the representative of ensemble methods (since it is the one which fared best in the experiments of [29]). For E(HDy)$_{\text{DS}}$ we set the number of base quantifiers to `size=30` and the number of members to be selected dynamically to `red_size=15`, and perform model selection independently for each base member.

Table 1 reports the AE results of our experiments. Our results are fairly consistent with those reported in [24, 25], and seem to indicate that the strongest method is SLD, which obtains the best average MAE, the best average rank, and is the best method on 13 datasets out of 30. Methods E(HDy)$_{\text{DS}}$ (8 times best method), PACC (4 times best method), and (to a lesser extent) ACC (2 times best method), also perform very well, obtaining average ranks not statistically significantly different from the best average rank (obtained by SLD). Method SVM(AE) tends to produce results that are markedly worse than the rest of competitors. In line with the observations of [32], none of the variants MAX, MS, MS2 improves over ACC. Also in line with the findings of [29], the ensemble E(HDy)$_{\text{DS}}$ clearly outperforms the base quantifier HDy it is built upon.

Figure 1 shows examples of plots (3 out of 4 plots are only for the Positive class) generated using QuaPy. The Diagonal plot (the results are averages across all samples characterized by the same true class prevalence values) reveals that, for high prevalence values of the Positive class, SLD tends to slightly overestimate these class prevalence values while most other methods tend instead to underestimate them. For low prevalence values of the Positive class, methods MAX, MS, MS2, PCC, and CC tend to overestimate these prevalence values. The Error-by-Shift plot (bottom left) displays AE as a function of the distribution shift between the training set and each of the test samples. (The results are averages across all samples characterized by the same value of distribution shift.) Here one can appreciate that E(HDy)$_{\text{DS}}$ excels at situations characterized by low distribution shift, while SLD seems the most robust in dealing with high-shift scenarios. The Bias-Box plots (top right) show the distribution of error bias (i.e., of the signed error between the estimated prevalence value and the true prevalence value) for all methods, as averaged across all datasets and test samples. This diagram reveals that PACC, SLD, and E(HDy)$_{\text{DS}}$ are the methods displaying the lowest bias overall, given that their boxes (delimiting the first and third quartiles) are the most squashed, and given that their whiskers (maximum and minimum, disregarding outliers) are the shortest. One interesting fact that is clearly revealed by this box-plot is, in line with what reported in [29], the ability of the ensemble method (E(HDy)$_{\text{DS}}$) to reduce the variance of the base quantifiers it is built upon (HDy). It is also interesting to note how the heuristic implemented in MS2 drastically reduces the variance produced by MS. The last plot (bottom right) displays error bias trends with samples binned according to their true prevalence; it clearly shows how the "unadjusted" methods (e.g., CC, PCC) display positive bias for low prevalence values (thus overestimating the true prevalence) and negative bias for high prevalence values (thus underestimating the true prevalence), while the "adjusted" versions (ACC and PACC) reduce this effect, since they tend to display box-plots centred at zero bias in those cases. This plot also clearly explains that MS tends to display a huge positive bias in the low-prevalence regime, while SVM(AE) displays a huge negative bias in the high-prevalence regime.

Note that the results presented here are just for the purposes of illustrating the functionality of QuaPy, and should not be taken as an absolute statement on the relative merits of the different quantification methods. For instance, a different batch of experiments (those reported in [24]), tell a slightly different story, since they report a much larger difference in accuracy between top-performing methods (SLD, PACC, ACC) and lesser performing ones (CC, PCC, SVM(AE), and others).[21] As always, a complete understanding of the relative merits of different learning methods can only be obtained through multiple, varied sets of experiments (see also [32]).

## 8 CONCLUSIONS

Quantification is a research topic of growing interest in machine learning, data mining, and information retrieval. We have presented QuaPy, a Python-based package that makes available a rich set of quantification methods, tools, experimental protocols, and datasets, with the goal of supporting an efficient and scientifically correct experimentation of quantification methods. We think that QuaPy will be of help to machine learning researchers that work on developing new quantification algorithms, as it provides them with many baselines to compare against, datasets to test their methods on, and tools that implement all the typical steps of quantification-based experimentation, from data preparation to the visualization of results. We think that QuaPy will be of help also to researchers and practitioners in other disciplines who simply need to apply quantification in their own work, as it provides them with a streamlined workflow, a wide choice of different approaches, and quick access to the package thanks to the support of installation based on `pip`. QuaPy is an open-source project, licensed under the BSD-3 licence; its repository will be updated following the advances

[18]To avoid clutter, we report only the three Forman's variants that have worked best in most of the experiments reported in [14]. Additional experiments that we have run, and that we do not report in this paper, confirm that T50 and X perform much worse than the other methods.

[19]Despite the fact that classifiers trained by LR are considered inherently well-calibrated (see, e.g., https://scikit-learn.org/stable/modules/calibration.html), [8] has found that re-calibrating LR brings additional benefits to SLD. In our experiments we thus instantiate SLD with a re-calibrated version of LR, and we indeed observe this to improve results noticeably. However, re-calibrating does not deliver any improvement for any other probabilistic quantifier that we test here, and instead shows a tendency to deteriorate the results. For this reason, we use a re-calibrated LR only for SLD, and a "standard" LR in all other cases.

[20]Among all ELM-based methods, we choose the one that minimizes the same loss that we adopt for evaluating the results. We do not consider other variants (SVM(Q), SVM(KLD), SVM(NKLD), SVM(RAE)) since, in recent evaluations (see, e.g., [24, 25]), they have consistently underperformed other competitors.

[21]One of the main differences between the experiments in this paper and those in [24] is that we here work on binary quantification only, while [24] tackled single-label multiclass quantification (since all datasets used there were ternary).

|  | Quantification methods |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
|  | CC | ACC | PCC | PACC | MAX | MS | MS2 | SLD | SVM(AE) | HDy | E(HDy)$_{DS}$ |
| BALANCE.1 | 0.039 | 0.032 | 0.049 | 0.037 | 0.040 | 0.046 | 0.036 | 0.025 | 0.035 | 0.022 | **0.020** |
| BALANCE.2 | 0.314 | 0.379 | **0.264** | 0.432 | 0.465 | 0.288 | 0.331 | 0.372 | 0.500 | 0.470 | 0.355 |
| BALANCE.3 | 0.039 | 0.020 | 0.045 | 0.021 | 0.040 | 0.046 | 0.036 | 0.018 | 0.064 | 0.017 | **0.014** |
| BREAST-CANCER | 0.022 | 0.025 | 0.029 | 0.023 | 0.028 | 0.021 | 0.023 | **0.020** | 0.144 | 0.029 | 0.026 |
| CMC.1 | 0.194 | 0.108 | 0.226 | 0.117 | 0.191 | 0.195 | 0.178 | **0.094** | 0.227 | 0.156 | 0.126 |
| CMC.2 | 0.178 | 0.138 | 0.220 | **0.098** | 0.271 | 0.500 | 0.427 | 0.105 | 0.449 | 0.118 | 0.103 |
| CMC.3 | 0.211 | 0.172 | 0.239 | 0.127 | 0.254 | 0.376 | 0.353 | 0.124$^{\ddagger}$ | 0.336 | 0.136 | **0.122** |
| CTG.1 | 0.037 | 0.020 | 0.050 | 0.020 | 0.041 | 0.033 | 0.035 | **0.017** | 0.094 | 0.028 | 0.018 |
| CTG.2 | 0.048 | 0.040 | 0.078 | 0.045 | 0.048 | 0.653 | 0.059 | **0.030** | 0.152 | 0.045 | 0.040 |
| CTG.3 | 0.047 | 0.044 | 0.050 | 0.043 | 0.045 | 0.649 | 0.061 | **0.022** | 0.113 | 0.053 | 0.045 |
| GERMAN | 0.151 | 0.142 | 0.191 | **0.092** | 0.154 | 0.125 | 0.134 | 0.101 | 0.262 | 0.165 | 0.113 |
| HABERMAN | 0.231 | 0.190$^{\ddagger}$ | 0.237 | 0.267 | 0.242 | 0.572 | 0.244 | **0.190** | 0.283 | 0.399 | 0.324 |
| IONOSPHERE | 0.111 | **0.074** | 0.116 | 0.084 | 0.124 | 0.209 | 0.089 | 0.075$^{\ddagger}$ | 0.256 | 0.104 | 0.082 |
| IRIS.2 | 0.201 | 0.241 | 0.195 | 0.183 | 0.251 | 0.412 | 0.256 | 0.215 | 0.461 | 0.075 | **0.056** |
| IRIS.3 | **0.019** | 0.074 | 0.044 | 0.071 | 0.054 | 0.134 | 0.024 | 0.057 | 0.205 | 0.069 | 0.047 |
| MAMMOGRAPHIC | 0.090 | 0.048 | 0.130 | 0.040 | 0.091 | 0.059 | 0.060 | 0.036 | 0.134 | 0.044 | **0.031** |
| PAGEBLOCKS.5 | 0.048 | **0.040** | 0.067 | 0.041$^{\ddagger}$ | 0.066 | 0.474 | 0.115 | 0.070 | 0.342 | 0.085 | 0.066 |
| SEMEION | 0.042 | 0.049 | 0.058 | 0.040 | 0.038 | 0.500 | 0.074 | **0.030** | 0.070 | 0.037 | 0.047 |
| SONAR | 0.135 | 0.200 | 0.163 | 0.119 | 0.145 | 0.171 | 0.159 | **0.114** | 0.346 | 0.136 | 0.131 |
| SPAMBASE | 0.042 | 0.026 | 0.066 | **0.022** | 0.049 | 0.070 | 0.037 | 0.031 | 0.196 | 0.025 | 0.024 |
| SPECTF | 0.143 | 0.155 | 0.178 | 0.133 | 0.276 | 0.620 | 0.182 | **0.105** | 0.296 | 0.420 | 0.231 |
| TICTACTOE | 0.024 | 0.019 | 0.024 | **0.014** | 0.024 | 0.136 | 0.024 | 0.019 | 0.500 | 0.018 | 0.019 |
| TRANSFUSION | 0.178 | 0.139 | 0.215 | 0.097 | 0.220 | 0.510 | 0.433 | **0.087** | 0.442 | 0.246 | 0.166 |
| WDBC | 0.034 | 0.036 | 0.034 | 0.027 | 0.038 | 0.096 | 0.029 | 0.025 | 0.056 | 0.019 | **0.015** |
| WINE.1 | 0.029 | 0.025 | 0.025 | 0.030 | 0.033 | 0.133 | 0.030 | 0.044 | 0.062 | 0.040 | **0.019** |
| WINE.2 | 0.026 | 0.048 | 0.043 | 0.052 | 0.045 | 0.088 | 0.041 | 0.046 | 0.051 | 0.032 | **0.022** |
| WINE.3 | 0.031 | 0.040 | **0.016** | 0.033 | 0.028 | 0.190 | 0.029 | 0.061 | 0.018$^{\dagger}$ | 0.018 | 0.025 |
| WINE-Q-RED | 0.140 | 0.076 | 0.183 | 0.059 | 0.141 | 0.065 | 0.099 | **0.056** | 0.222 | 0.065 | 0.058 |
| WINE-Q-WHITE | 0.150 | 0.077 | 0.194 | 0.064 | 0.149 | 0.113 | 0.124 | **0.059** | 0.247 | 0.072 | 0.066 |
| YEAST | 0.155 | 0.107 | 0.197 | 0.071 | 0.159 | 0.233 | 0.235 | **0.066** | 0.378 | 0.073 | 0.071 |
| Average | 0.104$^{\ddagger}$ | 0.093$^{\ddagger}$ | 0.121$^{\dagger}$ | 0.083$^{\ddagger}$ | 0.125$^{\dagger}$ | 0.257 | 0.132$^{\dagger}$ | **0.077** | 0.231 | 0.107$^{\ddagger}$ | 0.083$^{\ddagger}$ |
| Rank Average | 5.733 | 4.967$^{\dagger}$ | 7.033 | 3.900$^{\ddagger}$ | 7.133 | 9.033 | 6.733 | **3.133** | 9.900 | 5.233 | 3.200$^{\ddagger}$ |

**Table 1: Values of AE obtained in our experiments; each value is the average across 10,500 values, each obtained on a different sample. Boldface indicates the best method for a given dataset. Superscripts † and ‡ denote the methods (if any) whose scores are *not* statistically significantly different from the best one according to a paired sample, two-tailed t-test at different confidence levels: symbol † indicates $0.001 < p$-value $< 0.05$ while symbol ‡ indicates $0.05 \leq p$-value. For ease of readability, for each dataset we colour-code cells via intense green for the best result, intense red for the worst result, and an interpolated tone for the scores in-between.**

in quantification research, and it is open to contributions of new methods, tools, and datasets. In the near future, we plan to include implementations for further methods (e.g., Quantification Forests [23], and ReadMe [19]), ordinal quantification [7], and cross-lingual quantification [10].

## ACKNOWLEDGMENTS

The present work has been supported by the SoBigData++ project, funded by the European Commission (Grant 871042) under the H2020 Programme INFRAIA-2019-1, and by the AI4Media project, funded by the European Commission (Grant 951911) under the H2020 Programme ICT-48-2020. The authors' opinions do not necessarily reflect those of the European Commission.

## REFERENCES

[1] Barranquero, J., Díez, J., and del Coz, J. J. Quantification-oriented learning based on reliable classifiers. *Pattern Recognition 48*, 2 (2015), 591–604.

[2] Beijbom, O., Hoffman, J., Yao, E., Darrell, T., Rodriguez-Ramirez, A., Gonzalez-Rivero, M., and Hoegh-Guldberg, O. Quantification in-the-wild: Data-sets and baselines. CoRR abs/1510.04811 (2015). Presented at the NIPS 2015 Workshop on Transfer and Multi-Task Learning, Montreal, CA, 2015.

[3] Bella, A., Ferri, C., Hernández-Orallo, J., and Ramírez-Quintana, M. J. Quantification via probability estimators. In *Proceedings of the 11th IEEE International Conference on Data Mining (ICDM 2010)* (Sydney, AU, 2010), pp. 737–742.

[4] Biswas, A., and Mukherjee, S. Fairness through the lens of proportional equality. In *Proceedings of the 18th International Conference on Autonomous Agents and Multi-Agent Systems* (Montreal, CA, 2019), AAMAS 2019, pp. 1832–1834.

[5] Chan, Y. S., and Ng, H. T. Estimating class priors in domain adaptation for word sense disambiguation. In *Proceedings of the 44th Annual Meeting of the Association for Computational Linguistics (ACL 2006)* (Sydney, AU, 2006), pp. 89–96.

[6] Dua, D., and Graff, C. UCI machine learning repository, 2017.

[7] Esuli, A. ISTI-CNR at SemEval-2016 Task 4: Quantification on an ordinal scale. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval 2016)* (San Diego, US, 2016).

[8] Esuli, A., Molinari, A., and Sebastiani, F. A critical reassessment of the Saerens-Latinne-Decaestecker algorithm for posterior probability adjustment. *ACM Transactions on Information Systems 19*, 2 (2020), Article 19.

[9] Esuli, A., Moreo, A., and Sebastiani, F. A recurrent neural network for sentiment quantification. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management (CIKM 2018)* (Torino, IT, 2018), pp. 1775–1778.

[10] Esuli, A., Moreo, A., and Sebastiani, F. Cross-lingual sentiment quantification. *IEEE Intelligent Systems 35*, 3 (2020), 106–114.

[11] Esuli, A., and Sebastiani, F. Machines that learn how to code open-ended survey data. *International Journal of Market Research 52*, 6 (2010), 775–800.

[12] Esuli, A., and Sebastiani, F. Sentiment quantification. *IEEE Intelligent Systems 25*, 4 (2010), 72–75.

[13] Esuli, A., and Sebastiani, F. Optimizing text quantifiers for multivariate loss functions. *ACM Transactions on Knowledge Discovery and Data 9*, 4 (2015), Article 27.

[14] Forman, G. Quantifying trends accurately despite classifier error and class imbalance. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2006)* (Philadelphia, US, 2006), pp. 157–166.

[15] Forman, G. Quantifying counts and costs via classification. *Data Mining and Knowledge Discovery 17*, 2 (2008), 164–206.

[16] Gao, W., and Sebastiani, F. Tweet sentiment: From classification to quantification. In *Proceedings of the 7th International Conference on Advances in Social Network Analysis and Mining (ASONAM 2015)* (Paris, FR, 2015), pp. 97–104.

[17] González, P., Castaño, A., Chawla, N. V., and del Coz, J. J. A review on quantification learning. *ACM Computing Surveys 50*, 5 (2017), 74:1–74:40.

[18] González-Castro, V., Alaiz-Rodríguez, R., and Alegre, E. Class distribution estimation based on the Hellinger distance. *Information Sciences 218* (2013), 146–164.

[19] Hopkins, D. J., and King, G. A method of automated nonparametric content analysis for social science. *American Journal of Political Science 54*, 1 (2010), 229–247.

[20] Joachims, T. Transductive support vector machines. In *Semi-Supervised Learning*, O. Chapelle, B. Schölkopf, and A. Zien, Eds. The MIT Press, Cambridge, US, 2006, pp. 105–117.

[21] King, G., and Lu, Y. Verbal autopsy methods with multiple causes of death. *Statistical Science 23*, 1 (2008), 78–91.

[22] Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011)* (Portland, US, 2011), pp. 142–150.

[23] Milli, L., Monreale, A., Rossetti, G., Giannotti, F., Pedreschi, D., and Sebastiani, F. Quantification trees. In *Proceedings of the 13th IEEE International Conference on Data Mining (ICDM 2013)* (Dallas, US, 2013), pp. 528–536.

[24] Moreo, A., and Sebastiani, F. Tweet sentiment quantification: An experimental re-evaluation, 2020. arXiv 2011.08091.

[25] Moreo, A., and Sebastiani, F. Re-assessing the "Classify and Count" quantification method. In *Proceedings of the 43rd European Conference on Information Retrieval, Volume II* (Lucca, IT, 2021), pp. 75–91.

[26] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*. 2019, pp. 8024–8035.

[27] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[28] Platt, J. C. Fast training of support vector machines using sequential minimal optimization. In *Advances in kernel methods: Support vector learning*, B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds. MIT Press, Cambridge, US, 1999, pp. 185–208.

[29] Pérez-Gállego, P., Castaño, A., Quevedo, J. R., and del Coz, J. J. Dynamic ensemble selection for quantification tasks. *Information Fusion 45* (2019), 1–15.

[30] Pérez-Gállego, P., Quevedo, J. R., and del Coz, J. J. Using ensembles for problems with characterizable changes in data distribution: A case study on quantification. *Information Fusion 34* (2017), 87–100.

[31] Saerens, M., Latinne, P., and Decaestecker, C. Adjusting the outputs of a classifier to new a priori probabilities: A simple procedure. *Neural Computation 14*, 1 (2002), 21–41.

[32] Schumacher, T., Strohmaier, M., and Lemmerich, F. A comparative evaluation of quantification methods. *CoRR abs/2103.03223* (2021).

[33] Sebastiani, F. Evaluation measures for quantification: An axiomatic approach. *Information Retrieval Journal 23*, 3 (2020), 255–288.