Hub Miner User Manual

Hubness-aware Machine Learning for Intrinsically High-dimensional Data

Written and Maintained by:
Nenad Tomašev
Institute Jožef Stefan
Artificial Intelligence Laboratory
Jamova 39, 1000 Ljubljana, Slovenia
nenad.tomasev@gmail.com

October 2014

Short contents

Contents

List of Figures

List of Tables

One

Preface: What is Hub Miner?

Hub Miner is a machine learning library that focuses on experimentation with various data representations and distance functions for effective high-dimensional data analysis. The main emphasis of the library is on the novel hubness-aware machine learning techniques, that have been proposed in order to deal with the phenomenon of *hubness*[RNI09][RNI10a][RNI10b][Rad11] in intrinsically high-dimensional data, a well known aspect of the curse of dimensionality.

Figure 1.1: Hub Miner logo.

In Section 2.1, the hubness phenomenon is discussed in more detail. It has to do with the distribution of relevance within the (kNN) models in high-dimensional data, which assumes a scale-free shape, as most decisions are being dominated by the influence of a small number of prominent hub points. This entails a significant information loss, as well as a questionable semantic consistency of the emerging hubs, as they can be highly detrimental in certain cases.

Hub Miner implements hubness-aware methods for classification [RNI09][TRMI11b] [TRMI13a][TRMI11a][TM12b][TM13b], clustering [TRMI11c][TRMI13b][TRMI14], instance selection [BNST11a], metric learning [ZmP04][JHS07][SFSW12][TM12a][TM14a], re-ranking [TLM13] and other types of machine learning and data mining tasks.

Hubness visualization is possible via Image Hub Explorer [TM13c], a tool that was built for image feature representation experimentation, metric learning and visualization of various aspects of the kNN graph and the induced topology. It is possible to apply Image Hub Explorer not only to image data but also to other data types, though some specific functions might not be available in that case. A demo of Image Hub Explorer is available at the following link: `https://www.youtube.com/watch?v=LB9ZWuvm0qw`. Additional information can be found here: `http://ailab.ijs.si/nenad_tomasev/image-hub-explorer/`. This manual contains a chapter devoted to Image Hub Explorer and how it can be used in various visualization tasks.

Despite its focus in terms of current method implementation, Hub Miner has been designed as a general purpose machine learning library and can be used for other types of analysis as well, as it offers an extensive experimental framework and supports various data types. It has been implemented purely in Java, so it is easily portable to different platforms.

Java implementations have their ups and downs, but it is actually a decent trade-off when it comes to scientific computing, where experimentation under strict deadlines is the norm. Compared to C++, Java code requires much less effort to produce, de-bug and deploy. This can be quite beneficial for time-constrained projects that are typically seen in research. It is not as easy as Python, but it falls somewhere in between. This library being in Java, the entry point to extending it or re-using its approaches is probably not as high as it would have been with C++.

This user manual will enable Hub Miner users to quickly get the gist of the basic system functionality and the experimental framework. It also provides a basic overview of the algorithms themselves and the interfaces and implementation details. Many code examples are used to show how easy it is to extend the system to cover new use cases and include more baselines or apply the implemented techniques for solving practical machine learning tasks.

For things not explicitly covered in the manual, it is possible to get information from the source files themselves, as Hub Miner is fully documented and a lot of effort has been put into improving code presentation and style. The goal of Hub Miner is to become a hub (no pun intended) for hubness-aware machine learning, so it is important to make the code easy to use and understand. Expect the future releases to contain even more algorithms and more flexibility.

Hub Miner implements an OpenML-compatible experimental framework (`http://openml.org/`)[vRBT+13], which enables cross-system method comparisons to implementations in other OpenML-compatible machine learning libraries, such as Weka and RapidMiner.

Two

Motivation: Why yet another library?

Hub Miner is a general machine learning library, but its implementations are focused on one particular problem - the problem of hubness in instance-based learning in intrinsically high-dimensional data. It is, to our knowledge, the largest and most complete collection of hubness-aware methods.

Curse of dimensionality is a well-known phrase among the machine learning community and it's used to denote various types of problems that frequently arise when learning from high-dimensional feature representations. This includes sparsity, redundancy, distance concentration, problematic density estimates, less meaningful nearest neighbors, as well as hubness - the power-law-like distribution of relevance in the (kNN) models. Among all the listed issues, hubness was the latest to be observed [RNI09][Rad11]. Because of this, many instance-based systems and libraries are deployed without any safeguards and we will see why this might prove to be a problem in certain cases. In fact, any system that involves working with top-K result sets or top-K most similar items (as in collaborative filtering) can potentially be compromised if no measures are taken to assure the semantic consistency of hubs within the model. The music retrieval and recommendation community has seen a lot of recent research advances in hubness-aware metric learning and selection.

The extensive experimental framework in Hub Miner enables researchers to evaluate various data representations, primary or secondary metrics and kernels, classification and clustering algorithms, prototype selection strategies, under various circumstances. Multiple performance measures are supported and automated support for result summarization and statistical significance testing is included. The details of the testing environment will be provided in Chapter 5.

Since the word *hubness* has been used a few times already with no clarification and it is the main issue addressed by Hub Miner, the following Section explains why the hubness phenomenon matters and where it is expected to occur.

2.1 Relevance of Data Hubness for Data Analysis

Readers already deeply familiar with hubness can skip this Section and proceed to more technical instructions that follow.

As already mentioned, hubness as a phenomenon has to do with a skewed distribution of relevance in high-dimensional models, where a small number of hub points dominates the predictive models and influences most decisions, often in a detrimental way. An illustration of the emerging hubness is given in Figure 2.1, where the dimensionality of synthetic Gaussian mixtures is increased and the distribution of neighbor occurrence frequencies assumes a long-tailed shape.

The most natural question to ask is: 'Ok, so what? This is interesting, but why is this a bad thing?'. In most practical cases (in our experience) this distribution of neighbor occurrence frequencies is not scale-free, but rather converges towards a scale-free distribution as the dimensionality is

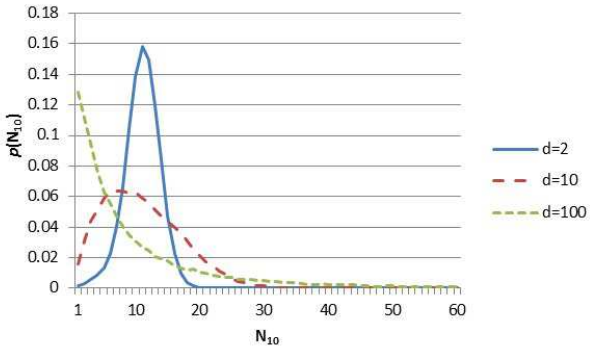


Figure 2.1: The change in the neighbor occurrence distribution shape with increasing dimensionality, in case of Gaussian mixture data. The increasing skewness results in most data points becoming *orphans* and a small number of *hubs* in the long tail of the distribution dominates the analysis.

increased. Yet, even if it were entirely scale-free, would this be a bad thing? Many scale-free networks exist in the real world and they seem to operate normally. A typical example is the Internet or social network popularity like the distribution of the number of Twitter followers or in-degrees in co-authorship research networks. Scale-free networks arise in the natural world as well, like protein-protein-interaction networks.

The main difference between real-world hub-harbouring networks and the kNN networks that we are considering is in the distribution of noise and inconsistencies and their influence on the systems. Scale-free networks have been shown to be more robust to uniformly random noise, but **substantially more sensitive to any inconsistency contained within the hubs**. Hub-centered errors can propagate quickly throughout the system, with dire consequences. Naturally occurring networks arise through self-organizing mechanisms that assure that hubs are at least as 'safe' as other points and often much more so. They are the Achilles heel, so they must be protected from noise (or malicious attacks, in case of the Internet).

However, the consistency of hubs in kNN graphs is not guaranteed, as their distribution depends on high-dimensional geometry and the particular choice of data representation and metric. Different pairs of metrics and data representations yield different degrees of overall hubness and a different selection of hub-points. In fact, it has been shown [Rad11][TRMI13b] that points that lie closer to local cluster means in high-dimensional data are much more likely to become hubs and they have a much higher expected occurrence frequency. As most high-dimensional data lies approximately on hyper-spheres around such cluster means, points closer to the center become close to many points on the hyper-sphere surface and this small difference coupled with distance concentration results in the emergence of hub points. In borderline regions, the emerging hubs often become semantic singularities and occur as neighbors to points from many different classes. This can be highly detrimental. An example is shown in Figure 9.6.

Most neighbor occurrences in high-dimensional data are hub occurrences and most bad occurrences are hub occurrences as well. The influence of a particular hub point can be either good or bad or - most likely - something in between. If hubs were strictly good or bad, a natural solution to the problem would be to try eliminating bad hubs entirely during data pre-processing. However, this is not the case - and even if it were - removing any highly influential neighbor point creates

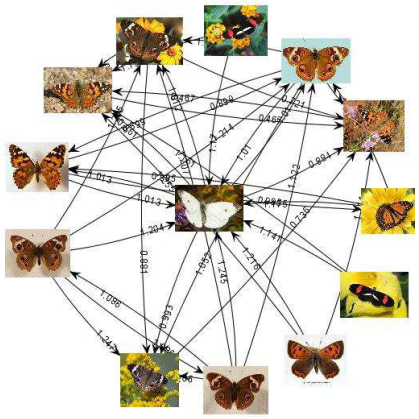


Figure 2.2: An example of a bad hub in the quantized SIFT feature representation, a detrimental center of influence. Neither of the reverse neighbors of the selected image belongs to the same class as the image itself, so its occurrences induce label mismatches and are semantically inconsistent.

holes in kNN sets that are filled in with new neighbor occurrences and some of these in turn might lead to the emergence of new bad hubs. The problem of finding the minimal set of prototype that properly covers all the training data has been shown to be NP-complete [BNST11b]. So, things are not as easy as they might initially seem.

Given that hubness arises naturally in textual data representation, image representations, time series and many other frequently encountered and intrinsically high-dimensional data types, it is an important issue in robust data analysis. As mentioned before, even systems that don't explicitly rely on kNN methods as such might be affected, especially if they use some internal ranking or queries for the most relevant results. While these functions often do not use explicit distance functions, implicitly it makes little difference.

Therefore, there are two things that we can try to do, as researchers or developers. We can try to somehow reduce the hubness of the data by a careful choice of data representations and metrics or we can try to design novel analytic methods that are intrinsically hubness-aware and robust. Hub Miner library contains both types of approaches and it can be shown that these two strategies are in fact not mutually exclusive, especially since metric learning and/or dimensionality reduction often merely reduce the hubness of the data instead of eliminating it completely. A complete reduction in hubness can usually be achieved only by reducing the intrinsic dimensionality of the data representation, which entails information loss and possibly lower predictive or descriptive system performance.

Have in mind that we are talking about the *intrinsic* dimensionality of the data here, not the number of features used to describe the data in the explicit representation, as many features are usually correlated in some way and they partially encode the same information. In fact, it is possible to observe very high-dimensional data with a low intrinsic dimensionality, that has no significant hubness. This is especially the case for some time series that do not vary much and consecutive sensor measurements are highly correlated.

Three

Building Hub Miner: Dependencies

Hub Miner is written in Java, so it shouldn't be difficult to build. You *will* need an up-to-date version of Java, though. Hub Miner has been developed under jdk1.7.0_21, but new features will be added that are compliant with newer versions as well, in future releases, so try having an up-to-date JRE or SDK on your machine if running or building Hub Miner. You should include the following dependencies in the CLASSPATH variable in order to be able to build and use Hub Miner code:

```
apiconnector-fat.jar
collections-generic-4.01.jar
colt-1.2.0.jar
commons-codec-1.3.jar
commons-httpclient-3.0.1.jar
commons-logging-1.1.jar
concurrent-1.3.4.jar
gson-2.3.jar
guice-3.0.jar
iText-2.1.7_mx-1.0.jar
Jama-1.0.2.jar
jcommon-1.0.17.jar
jdom.jar
jetty-6.1.1.jar
jetty-util-6.1.1.jar
jfreechart-1.0.14.jar
jgraph.jar
jgraphx.jar
json.jar
jsoup-1.7.2.jar
jtidy-r7.jar
jung-algorithms-2.0-beta1.jar
jung-api-2.0-beta1.jar
jung-graph-impl-2.0-beta1.jar
jung-jai-samples-2.0-beta1.jar
jung-visualization-2.0-beta1.jar
junit-4.7.jar
mdsj.jar
mxgraph-all.jar
rome-0.8.jar
```

```
servlet-api-2.5-6.1.1.jar
servlet.jar
swing-layout-1.0.3.jar
swingx-1.6.jar
swingx-beaninfo-1.6.jar
swingx-ws-1.0.jar
TGGraphLayout.jar
xercesImpl.jar
xmlunit1.0.jar
```

A dependency on OpenML is apiconnector-fat.jar and it can be downloaded from http://openml.org/downloads/apiconnector-fat.jar .

That being said, there are some limited experimental parts of the code that rely on some other external dependencies. Image feature extraction components use SIFT features and expect to extract them via the SiftWin binary that is to be accessible from the command line so it has to be contained within the PATH variable. The same goes for Image Magick, which is used mainly for conversions of JPG images to PGM format that SiftWin expects. In almost all cases, you will not be invoking these functions since Hub Miner is not about image feature extraction and does not intend to become a pipeline for that. However, expect to see this dependency disappear in some later Hub Miner release, as I intend to switch to Java-based image feature extraction libraries and also introduce more compatibility with OpenCV.

Other than the mentioned image feature extraction dependency, the rest of Hub Miner code is currently portable and should be usable on different platforms without any additional difficulty.

Four

Supported Data Formats

Hub Miner supports working with data files in the ARFF, CSV and TSV formats. It also offers support for configuration file load/save and cross-validation folds import via JSON. Of course, since JSON is very strict and sometimes difficult to read for larger configuration files, there exists a custom human-readable easy-to-use default format for specifying the experimental configurations. The configuration file format will be reviewed in the following section.

For readers not already familiar with the ARFF file format (also used by Weka and OpenML), here is an example of a small ARFF data file:

```
@RELATION Ionosphere3DProjectionSample
@ATTRIBUTE fAtt0 real
@ATTRIBUTE fAtt1 real
@ATTRIBUTE fAtt2 real
@ATTRIBUTE class string
@DATA
0.08901424,-0.10969148,0.0032827153,'1'
-0.044016507,-0.1379753,-0.10553418,'0'
0.10856112,-0.039936334,0.044553082,'1'
-0.066826984,0.07276285,-0.039159104,'0'
-9.745792E-4,-0.09848936,0.017975774,'1'
-0.13074261,-0.022526562,-0.101753585,'0'
0.054415386,-0.20338419,-0.0049496056,'1'
-0.041923035,-0.0017737036,-0.17944402,'0'
0.15486263,-0.17194971,0.012484646,'1'
-0.13705792,-7.132998E-4,-0.10510041,'0'
0.13835455,-0.21733429,0.018616,'1'
0.24783619,-0.10117917,0.0027347172,'0'
0.17932422,-0.14469129,0.03308493,'1'
0.21689402,-0.049687877,0.020258931,'0'
0.2064358,0.10299684,0.031745467,'1'
0.006842114,-0.055629272,-0.14137895,'0'
```

There is a header that defines the feature names and feature types - and this is followed by a data section, that presents data in a comma-separated fashion, with label information at the end. In Hub Miner there is a convention that in the ARFF files the label attribute is named 'class' and of type 'string', so make sure to follow the same convention in the data files that you will use in the experiments.

As for CSV files, here is an example of a sample of the classical Iris data presented in a Hub Miner - compatible CSV form.

```
4.4,3.0,1.3,0.2,0
5.1,3.4,1.5,0.2,0
5.0,3.5,1.3,0.3,0
4.5,2.3,1.3,0.3,0
4.4,3.2,1.3,0.2,0
5.0,3.5,1.6,0.6,0
5.1,3.8,1.9,0.4,0
4.8,3.0,1.4,0.3,0
5.1,3.8,1.6,0.2,0
4.6,3.2,1.4,0.2,0
5.3,3.7,1.5,0.2,0
5.0,3.3,1.4,0.2,0
7.0,3.2,4.7,1.4,1
6.4,3.2,4.5,1.5,1
6.9,3.1,4.9,1.5,1
5.5,2.3,4.0,1.3,1
6.5,2.8,4.6,1.5,1
5.7,2.8,4.5,1.3,1
6.3,3.3,4.7,1.6,1
4.9,2.4,3.3,1.0,1
6.6,2.9,4.6,1.3,1
5.2,2.7,3.9,1.4,1
5.0,2.0,3.5,1.0,1
5.6,2.9,3.6,1.3,1
```

The label is the last value in each line. It should be noted here, though - that it does not need to be a number, strings containing class names are also acceptable. In case of CSV files, all features are loaded as float features. This is somewhat less flexible than in the ARFF case.

Also, one should keep in mind that while the last value is treated as class value by default, the loaded has both a supervised and an unsupervised save/load modes - and that it doesn't know in principle which one should be used in which occasion. Therefore, take care when working with CSV files to use the proper mode. Supervised is default in most cases, even in Clustering evaluation, since labeled data is also used there, where labels are used for calculating cluster configuration homogeneity.

Five

A Quick Guide to the Experimental Framework

Hub Miner supports batch experimentation by running multiple algorithms on multiple datasets under varying conditions in a multi-threaded way with optimized object sharing for distance matrices and kNN sets to avoid redundant calculations. Such batch task processing is available for classification, clustering and exploratory hubness data analysis.

5.1 Batch Classifier Evaluation

The class that runs batch classifier evaluation in Hub Miner is available at learning.supervised.evaluation.cv.BatchClassifierTester and most logic is implemented in learning.supervised.evaluation.cv.MultiCrossValidation.

For classifier evaluation, Hub Miner uses X-time Y-fold cross validation. I have personally always used 10-times 10-fold cross-validation, though other set-ups are possible for large datasets and initial exploratory runs. The general design of the experimental framework is shown in Figure 5.1.

The classification evaluation framework supports metric learning, instance selection, learning with label noise, learning with feature noise, varying neighborhood sizes, biased and unbiased prototype occurrence modeling, etc. Despite its many options, the basic process is quite simple.

The data is loaded and feature normalization is performed, if specified. The primary distance matrix is calculated once for the entire data and divided up into sub-matrices during cross-validation, based on the generated data splits. In case of primary kNN sets (when no metric learning is used), the k-nearest neighbor sets are also calculated once for all data and a slightly larger neighborhood size and are then sub-sampled and restricted on the training data in each experimental iteration. In most cases, no additional re-calculations are required, though rarely the system has to run a few fast additional queries. Instance selection is easily integrated with this framework and doesn't complicate things at all.

The calculated objects are distributed to all the algorithms that require them, which is declared by implementing appropriate interfaces in the code. In case no algorithm required distances or k-nearest neighbor sets, none are calculated.

Each algorithm runs in its own thread and all threads are synchronized. The experiment ends when the last algorithm finishes.

It is possible also to run experiments with the approximate kNN sets and recursive Lanczos bisections are used as a default approach for this in the current implementation, though this will probably change in future versions, where I intend to introduce more flexibility.

In case of metric learning, it is not possible to apply the exact same framework as above, as the global distance matrix would then be a result of using both training and test data, which would

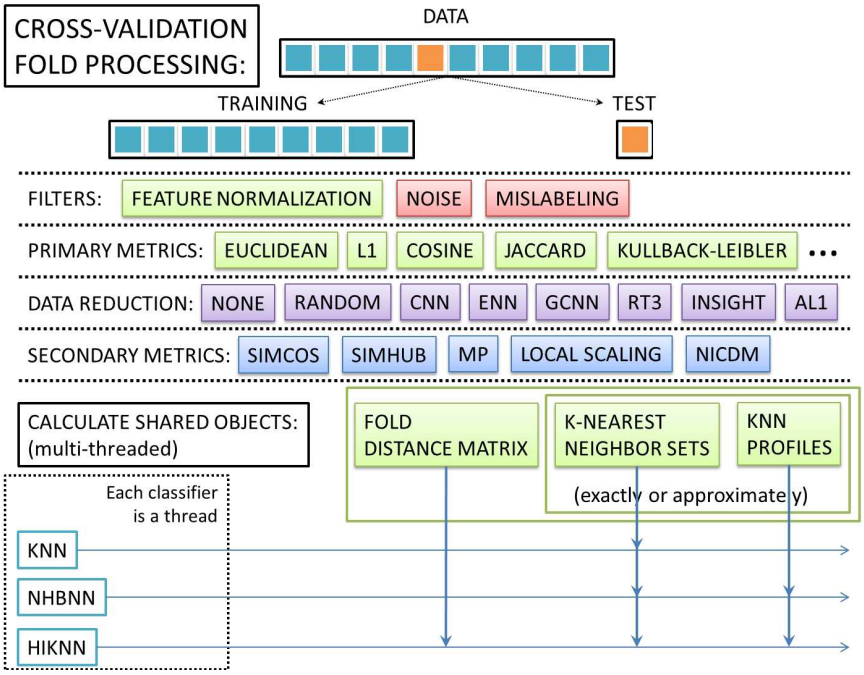


Figure 5.1: Hub Miner experimental set-up for classification experiments.

constitute an information leak. The experiments with metric learning are therefore usually much slower, as both the distance matrix and the k-nearest neighbor sets need to be calculated on each training iteration separately.

Each test run in each CV iteration produces a learning.supervised.evaluation.ClassificationEstimator instance. These classification estimators calculate the quality measures like accuracy, precision, recall, f-score, Matthews correlation coefficient and the likes (micro and macro-averaged where necessary). These results are then persisted to the specified files and summarized automatically to produce brief and readable experimental summaries. Statistical significance testing is done on-the-fly with the corrected re-sampled t-test.

An example configuration file for running the evaluation is shown in Figure 5.2.

Users can specify the desired fold scheme, the number of common threads to use when calculating the distance matrices and kNN sets, the directory structure, instance selection strategies (optional), secondary distances (optional), a list of classification algorithms to compare, the range of neighborhood sizes to test for, approximate kNN set quality (optional, exact kNN sets are default), ranges of feature and label noise rates to test for, weights for weight-proportional noise (optional), the feature normalization strategy and finally, a list of datasets from the data directory to run the experiments on, paired with the appropriate metric objects.

Let us go through all options one by one.

- **@cross_validation** is an option that specifies how many folds and runs to use in the cross-validation procedure. It is pretty self-explanatory. If omitted, 10-times 10-fold cross-validation is used as default.

```
@cross_validation 10 10
@common_threads 8
@in_directory K:\DATA_MINING\DATA
@out_directory K:\EXPERIMENTS\experiment_name\tests
@summary_directory K:\EXPERIMENTS\experiment_name\summaries
@distances_directory K:\EXPERIMENTS\DMAT
@fold_directory K:\EXPERIMENTS\DMAT\CVFolds
@instance_selection GCNN
@protoHubness unbiased
@secondary_distance simhub 50
@algorithm h-FNN
@algorithm KNN
@algorithm NHBNN
@algorithm LWNB
@algorithm DNaiveBayes
@algorithm CBWkNN
@k_range 1 10 1
@approximateNN 0.7
@noise_range 0 0 1
@mislabeled_range 0 0.5 0.1
@mislabeling_weights_dir K:\EXPERIMENTS\mlWeights
@normalization tfidf
@dataset allSubSamples\threeSimsetSample.arff null distances.primary.Manhattan
@dataset uci-data\sonar-num.csv null distances.primary.MinkowskiMetric
@dataset uci-data\diabetes-num.csv null distances.primary.MinkowskiMetric
@dataset GaussianMixtures\datasets\dataset0.arff null distances.primary.MinkowskiMetric
@dataset TimeSeries\tSerARFFDummies\Coffee.arff null distances.primary.DTW
@dataset TimeSeries\tSerARFFDummies\ECG200.arff null distances.primary.DTW
```

Figure 5.2: An example of a classification configuration file. While most options are 'on' in this example, in practice a user would normally only use some of them. The example merely shows that they can all in principle be combined and that all stages can be plugged in and operate together.

- **@common_threads** is used for specifying the number of threads to use for calculating shared objects in the experimental framework, like primary and secondary distance matrices and primary and secondary k-nearest neighbor sets.
- **@in_directory** is used for specifying the data directory. The easiest approach is to simply have all data in one directory, in appropriate subdirectories. The following @dataset commands accept relative paths w.r.t. this specified top directory.
- **@out_directory** specifies the primary evaluation output target, where most raw classification results will be persisted. These detailed results are useful for later analysis, but the users will usually first review the classification summaries for each dataset (see below).
- **@summary_directory** specifies the directory where classification summaries will be generated from the raw test data, for each dataset. Statistical tests are also run at this point, to detect statistical significance in the differences between the algorithms.
- **@distances_directory** specifies where the distance matrices are to be saved or loaded from. Pre-calculated distance matrices can be loaded into the framework if provided at the expected location and the calculated distance matrices can also be saved for a later load. For instance, assume that I specify the distance directory as: K:\EXPERIMENTS\DMAT.

The distance matrix for dataset0 if no normalization is done will be expected at K:\EXPERIMENTS\DMAT\dataset0\NO\distances.primary.Manhattan location. The type of normalization is included in the path, so take care which feature normalization you specify. This is especially useful if you need to 'hack' the library to load a distance matrix for a distance that is not actually implemented in the library - but you need to use it in some experiments and you can calculate it externally. It is recommended to define a dummy class in the primary metrics directory and to specify that dummy metric for the datasets in question. You could in principle try to trick the system and present the distance matrix as originating from some of the standard metrics - but that is not a good approach, as you might also want to use standard metrics on the same data and you might forget about the hack in time and confuse the distance matrices in a few weeks when you get back to the experiment. Therefore, try to be as consistent and precise as possible. Another important detail is that - since the distance matrices are loaded if the system detects that they exist at the expected path - it is possible to encounter problems if you have datasets that share the exact same name, so try to avoid that scenario by either using separate distance directories in those cases (recommended) or deleting the matrices after a while. If the distance directory is left unspecified, the distances will be calculated on the fly and will not be saved for later use.

- **@fold_directory** Data folds can be saved and loaded, in JSON format - and this option specifies the target directory that contains the data folds for all datasets.
- **@instance_selection** specifies the instance selection approach to use, if needed in the experiments. Note that you do not have to specify the full path to the class (you can, of course). You can check which abbreviations and synonyms are used for different methods in the preprocessing.instance_selection.ReducersFactory class.
- **@proto_hubness** has two options: biased and unbiased. This one might be a bit difficult at first, but it is used only with instance selection and only if hubness-aware classification methods are present. Namely, instance selection implements a certain selection bias, so the neighbor occurrence models learned from the reduced data also reflect that bias. In order to correct the bias in the models, an in-between step that re-calculates the kNN sets on the entire training data with using only the selected prototypes as potential neighbors can be inserted and this is what the unbiased option in @proto_hubness does, if selected. It leads to a slightly better classification performance.
- **@secondary_distance** specifies which secondary distance type to use, if doing metric learning in form of secondary distances in the experiments. The options are as follows: simcos, simhub, mp, ls, nicdm. If you are unsure about this option or options in general, you can look at configuration.BatchClassifierConfig to see how the parsing is done and which values are expected. This option takes an optional second parameter that is the secondary neighborhood size to use for calculating the secondary distances, as some of them explicitly require this parameter. If omitted, a default value is used, which may or may not be appropriate for your data, so it is recommended to always specify it explicitly.
- **@algorithm** specifies an algorithm to use in the testing and multiple occurrences of the option specify multiple algorithms to test at the same time, in a multi-threaded way, each algorithm being a single thread. You can look at all the abbreviations and synonyms for specifying classification algorithms in learning.supervised.ClassifierFactory class. If you specify an algorithm by its class path, that should also work. Additional parameters can be set for each algorithm via a JSON string, as in the following example.

```
@algorithm DWHFNN {"mValue":3.5,"thetaCutoff":3}
```

- **@k_range** specifies the range of neighborhood sizes to test for. It has three parameters - the minimal value, the maximal value and the increment.
- **@noise_range** specifies the range of feature noise rates to test for. It has three parameters - the minimal value, the maximal value and the increment.
- **@mislabeled_range** specifies the range of label noise rates to test for. It has three parameters - the minimal value, the maximal value and the increment.
- **@approximateNN** announces that recursive Lanczos bisections will be used for approximating the k-nearest neighbor sets. It also takes a quality parameter between 0 and 1. This option is currently not used for actual scalability, but rather to evaluate the robustness of certain kNN-based approaches under approximate kNN sets of varying quality. Omit this option in order to use the exact kNN sets - recommended and default.
- **@mislabeling_weights_dir** is an optional parameter used to specify weights for weight-proportional random label noise. This option has been used for testing hubness-proportional random label noise in past experiments.
- **@normalization** specifies which normalization scheme to use. The permitted values are: no, normalizeTo01, standardize, tfidf.
- **@dataset** specifies a dataset to run the experiments on. Multiple occurrences of this option are used to specify multiple datasets for the experiments, so that they run in a batch mode. The first parameter is the relative path to the dataset in ARFF or CSV or TSV format, w.r.t. to the global data directory and the second and third parameter are metrics to be used for integer and float variables, respectively, given by their class path. A CombinedMetric object is then formed from these two feature-type specific metric objects. All features from CSV files are loaded into float feature arrays as default, but features from ARFF files can also be loaded as integers specifically, so this is why there are two separate metrics here. Also, some categorical variables that are represented as integers make no sense to use in standard float metrics, so this is a useful separation.

You are encouraged to look at configuration.BatchClassifierConfig for more details on the configuration file and its parsing. Note that it is possible to save the configuration object to JSON and load it from JSON. This is useful, since JSON is much easier to generate automatically and this allows the experimentation protocols to be invoked by other classes in the library or remotely. The framework accepts a configuration object, so it can also be instantiated from within the code. As for manual experimentation, the custom format that was presented above is much easier and more readable.

Any line in the configuration file that doesn't start with a valid option will be ignored. You can therefore easily comment out the options that you are currently not using (but will be using later) by using any standard comment notation. I use this all the time to comment out datasets or algorithms and then bring them back in, while doing initial tests.

5.1.1 OpenML Compatibility

Hub Miner is an OpenML-compatible library and it is possible to perform cross-validation classification experiments according to OpenML standards. OpenML `http://openml.org/` is a networked science project that aims to bridge the gap between scattered method evaluations across different studies. This service stores data and the fold train and test splits and offers it at request. The library then runs the experiments on the specified splits and uploads the point-wise predictions during all runs, for all tested algorithms. Algorithm registration at OpenML is performed automatically, which also includes a set of possible parameters, parameter descriptions, types and default

values. The exact values used in the run are uploaded along with the predictions, so that it is possible to compare different parametrizations online.

After all the predictions are uploaded, OpenML calculates various performance measures and stores all this into a database. It is therefore possible to compare algorithms implemented in different libraries and OpenML offers many visualizations of such comparisons. Therefore, algorithms implemented in Hub Miner can now be easily compared with algorithm implementations in Weka, R or Rapid Miner. This is very useful for all future Hub Miner users and developers, as they can easily compare the implemented approaches to baselines that are not yet contained within Hub Miner.

There are plans in OpenML to introduce support for experiments that involve learning with label or feature noise, though these experiments are not currently supported (October 2014). Also, clustering flows will become available soon and this will be supported in Hub Miner as well.

So, how does one specify an OpenML data source in a Hub Miner batch classification configuration file? This involves a very minor modification of what was already shown. It is necessary to authenticate for using the OpenML service, so all users that want to use this feature have to open an OpenML account and receive credentials. The use of these credentials is to be specified in the following way.

```
@openml_authentication username password
```

OpenML data sources are specified similarly to local datasets. Here is an example, with some common UCI classification tasks.

```
@openml_task openml-data\iris.arff null distances.primary.MinkowskiMetric 1939
@openml_task openml-data\mfeat-zernike.arff null distances.primary.MinkowskiMetric 1902
@openml_task openml-data\mfeat-morphological.arff null distances.primary.MinkowskiMetric 1898
@openml_task openml-data\sonar.arff null distances.primary.MinkowskiMetric 1919
@openml_task openml-data\diabetes.arff null distances.primary.MinkowskiMetric 1917
@openml_task openml-data\breast-w.arff null distances.primary.MinkowskiMetric 1895
@openml_task openml-data\ionosphere.arff null distances.primary.MinkowskiMetric 1937
@openml_task openml-data\mfeat-fourier.arff null distances.primary.MinkowskiMetric 1894
@openml_task openml-data\glass.arff null distances.primary.MinkowskiMetric 1920
@openml_task openml-data\ecoli.arff null distances.primary.MinkowskiMetric 1918
```

Instead of **@dataset**, the **@openml_task** command is used. What follows is the path where the data will be saved locally after the download, as well as the metrics to use for integers and floats. The final item on each line is the task ID. Make sure that you use the task ID here, not the data ID, otherwise, this won't work. The task ID has to correspond to a cross-validation task and these tasks shown in the example are for 10-times 10-fold cross-validation. If you are using the **@cross_validation** command in the configuration file, make sure that the number of repeats and folds in that specification matches the one in the task that you are specifying. Mismatches will cause errors.

Figure 5.3 shows an example of the uploaded run information for a specific algorithm. Along with the uploaded predictions, it is possible to view the performance measures, as shown in Figure 5.4.

It is easy to compare algorithm runs, as shown in Figure 5.5.

In order to obtain the desired task IDs for your experiments, go to the dataset list at OpenML data page: `http://openml.org/d`. Browse for the desired dataset and select the appropriate task and copy its task ID into the Hub Miner batch classification configuration file.

Figure 5.3: An example of the uploaded run information in OpenML.

5.1.2 Viper Charts for Visualizing Classification Results

Hub Miner can be used for visualizing the generated classification results in its experimental framework by invoking the Web API of Viper charts `http://viper.ijs.si/`, a great tool that can be used to produce various type of classification performance charts and comparisons.

The API connector is located in visualization.ViperChartAPICall and it exposes a command line interface for providing it with the selected algorithm result directories, in order to visual selected algorithms on the examined data and produce the specified chart type.

For instance, here is a simple call that I have used locally to generate a lift chart comparison between kNN and NHBNN on the standard ionosphere UCI dataset. It has been broken down into multiple lines for readability.

```
java visualization.ViperChartAPICall
-inDataFile::K:\DATA_MINING\DATA\uci-data\ionosphere-num.csv
-inAlgorithmDir::K:\EXPERIMENTS\viperDemoTests\ionosphere-num\k5\ml0.0\noise0.0\KNN
-inAlgorithmDir::K:\EXPERIMENTS\viperDemoTests\ionosphere-num\k5\ml0.0\noise0.0\NHBNN
-positiveClassIndex::1
-chartTypeStringCode::lift
```

Evaluations

area_under_roc_curve	0.9731	[0.991775,0.984514,0.990966,0.96103
confusion_matrix		[[1939,0,1,10,0,12,0,0,38,0],[0,1918,0,22 [10,20,32,1789,10,90,21,8,0,20],[0,45,0,([0,2,0,3,27,2,1354,14,10,588],[0,7,0,32,0 [0,10,0,21,5,45,610,11,0,1298]]
f_measure	0.8896	[0.970956,0.953754,0.968082,0.89876
kappa	0.8772	
kb_relative_information_score	16820.3425	
mean_absolute_error	0.0485	
mean_prior_absolute_error	0.1800	
number_of_instances	20000.0000	[2000,2000,2000,2000,2000,2000,200
precision	0.8899	[0.972417,0.948566,0.973219,0.90307
predictive_accuracy	0.8895	
prior_entropy	3.3219	
recall	0.8895	[0.9695,0.959,0.963,0.8945,0.953,0.88
relative_absolute_error	0.2696	
root_mean_prior_squared_error	0.3000	
root_mean_squared_error	0.1368	
root_relative_squared_error	0.4562	

Figure 5.4: An example of the uploaded run performance measures in OpenML.

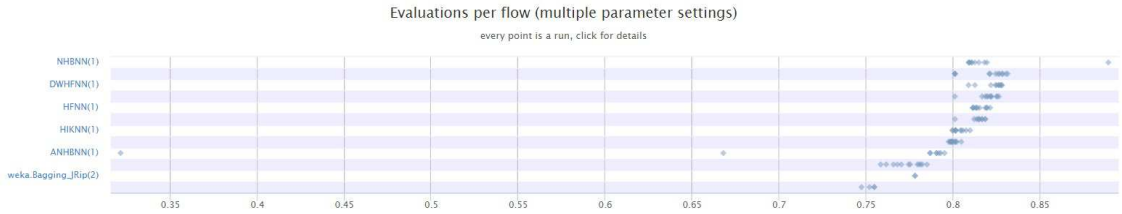


Figure 5.5: An example of between-classifier comparisons in OpenML.

Notice that the -inAlgorithmDir can be specified multiple times and this is how multiple algorithms are compared. It is a feature in the CommandLineParser class, that allows multi-valued parameters if it is specified in the parsing configuration.

The original data is also provided, in order to obtain the correct label assignments, the ground truth. The user also needs to specify which class is to be used as the positive class. This is especially important in the multi-class case, as most charts were built with binary classification problems in mind. Finally, the chart type is also specified by its code, in the same way as in the Viper Charts online documentation. This information is also easily readable from the source file, as there is a dictionary of String codes that correspond to different chart types.

The lift chart that was generated for the given command line call is given in Figure 5.6.

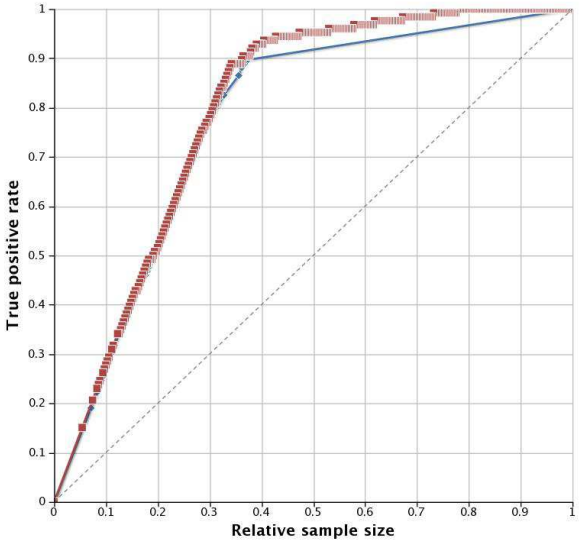


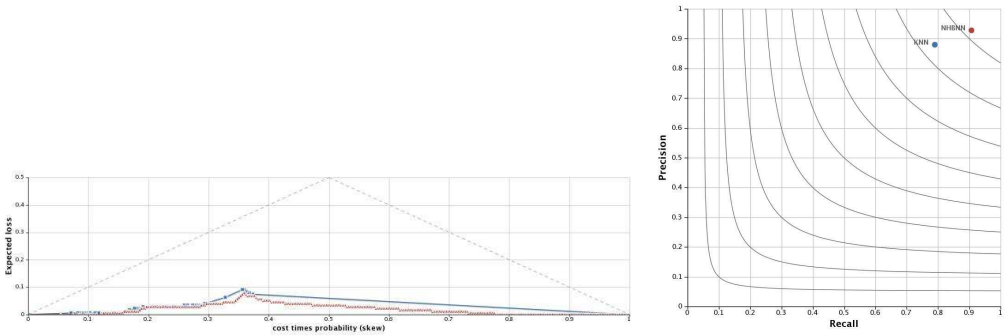
Figure 5.6: An example of a visualization of classification results in form of a lift chart, as generated by Viper Charts. These results show the classification performance of kNN and NHBNN on the ionosphere UCI dataset, for $k = 5$. NHBNN is given in red and kNN in blue.

It is possible to configure to show legend and grind lines or isolines for every chart, while working on the generated URL in the browser. There is an option in the ViperChartAPICall class command line call named "-openInBrowser" which opens the URL in the default browser automatically if set to true. False value is currently default, as the users might want to call the Viper API automatically from within other classes and experimental protocols, so it would not make sense to do this in those cases.

Other visualizations for different chart types on the same data and for the same algorithms are given in Figure 5.7.

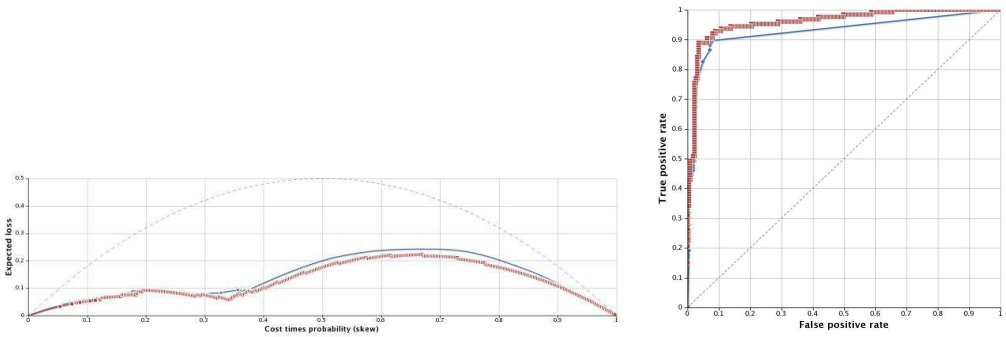
The comparisons clearly show that NHBNN outperforms kNN for the selected neighborhood size on ionosphere, as it has a lower expected loss, which is shown in Kendall and rate-driven curves, a higher AUC score which can be seen from the ROC chart, NHBNN is located beyond a higher F-isoline in PR-space, etc.

Since the charts are generated in Javascript and a URI is provided on the Viper Charts web page with the visualization, users can easily share their findings with others by just sharing the link.



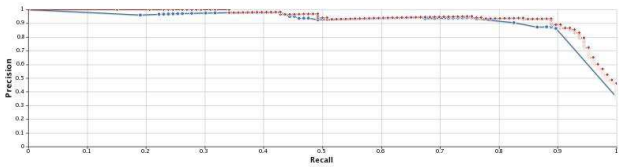
(a) Kendall curve

(b) PR Space



(c) Rate-driven curve

(d) ROC Curve



(e) Recall-Precision curve

Figure 5.7: Example visualizations on ionosphere UCI data for $k = 5$ of classification performance of kNN and NHBNN from within Hub Miner framework. NHBNN is given in red and kNN in blue.

There is a daily usage limit for Viper Charts, though. It is currently set to 100 per user (as identified by IP). However, you will probably not be in need of more than 100 charts a day anyway, so this is not very restrictive.

5.2 Batch Clustering Evaluation

The Hub Miner batch clustering experimentation framework is quite similar to the classification framework in terms of configuration syntax and also in terms of internal optimization and speed-ups. We will review all the options one by one. Descriptions for options that we have already

mentioned previously will be repeated so that you do not have to scroll back up to look for them. A bit redundant, but more readable.

- **@times** The number of times a clustering is to be done on a single dataset by an algorithm. Since some clustering methods use random initialization and can converge to multiple optima, it is necessary to test the clustering algorithm performance by repeating the clustering process multiple times.
- **@iter** The maximal number of iterations for iteration-based algorithms. This prevents some very long and slow converging processes from taking too much time. Use with care.
- **@num_clusters** Some clustering algorithms need a pre-defined number of clusters to cluster for, while other try to discover the optimal number of clusters automatically. For those that require an explicit parameter, this option is used to set it. It is also possible to specify "natural" (without the enclosing quotes) as an option value here, which will set the number of clusters to the number of classes in the data defined by the loaded labels, automatically.
- **@split_training** specifies whether to split the data into training and test data or not. Optional and the system does not perform any training/test splits for clustering evaluation, naturally. Some people prefer to do the split (and this was actually implemented in response to a reviewer's request for one of the papers on hubness-based clustering) and this allows it to be specified. The problem with this is that it is not altogether clear how to assign test points to a trained clustering model. For some algorithms, like K-means, this might be obvious and straightforward - but more complex algorithms with complex models do not always permit such easy solutions for incrementally adding test points.
- **@common_threads** is used for specifying the number of threads to use for calculating shared objects in the experimental framework, like primary and secondary distance matrices and primary and secondary k-nearest neighbor sets.
- **@in_directory** is used for specifying the data directory. The easiest approach is to simply have all data in one directory, in appropriate subdirectories. The following @dataset commands accept relative paths w.r.t. this specified top directory.
- **@out_directory** specifies the primary evaluation output target, where clustering results will be persisted. There is no additional summary directory for clustering.
- **@distances_directory** specifies where the distance matrices are to be saved or loaded from. Pre-calculated distance matrices can be loaded into the framework if provided at the expected location and the calculated distance matrices can also be saved for a later load. For instance, assume that I specify the distance directory as: K:\EXPERIMENTS\DMAT. The distance matrix for dataset0 if no normalization is done will be expected at K:\EXPERIMENTS\DMAT\dataset0\NO\distances.primary.Manhattan location. The type of normalization is included in the path, so take care which feature normalization you specify. While it was possible to 'hack' the batch classification experimental framework by using distance matrices from non-implemented metrics, it is **not possible to use distances from non-implemented metrics** in clustering. The reason is simple, since algorithms like K-means calculate centroids and need to calculate the distance in the specified metric from the centroid to all other points in the data. If a proper metric object is not provided, this is simply not going to work. Another important detail is that - since the distance matrices are loaded if the system detects that they exist at the expected path - it is possible to encounter problems if you have datasets that share the exact same name, so try to avoid that scenario by either using separate distance directories in those cases (recommended) or deleting the matrices after a while. If the distance directory is left unspecified, the distances will be calculated on the fly and will not be saved for later use.

- **@secondary_distance** specifies which secondary distance type to use, if doing metric learning in form of secondary distances in the experiments. The options are as follows: simcos, simhub, mp, ls, nicdm. If you are unsure about this option or options in general, you can look at configuration.BatchClusteringConfig to see how the parsing is done and which values are expected. This option takes an optional second parameter that is the secondary neighborhood size to use for calculating the secondary distances, as some of them explicitly require this parameter. If omitted, a default value is used, which may or may not be appropriate for your data, so it is recommended to always specify it explicitly.
- **@algorithm** specifies an algorithm to use in the testing and multiple occurrences of the option specify multiple algorithms to test at the same time, in a multi-threaded way, each algorithm being a single thread. You can look at all the abbreviations and synonyms for specifying classification algorithms in learning.unsupervised.ClustererFactory class. If you specify an algorithm by its class path, that should also work. Direct JSON parametrization is possible, as in classification.
- **@k_range** specifies the range of neighborhood sizes to test for. It has three parameters - the minimal value, the maximal value and the increment.
- **@noise_range** specifies the range of feature noise rates to test for. It has three parameters - the minimal value, the maximal value and the increment.
- **@mislabeled_range** specifies the range of label noise rates to test for. It has three parameters - the minimal value, the maximal value and the increment.
- **@approximateNN** announces that recursive Lanczos bisections will be used for approximating the k-nearest neighbor sets. It also takes a quality parameter between 0 and 1. This option is currently not used for actual scalability, but rather to evaluate the robustness of certain kNN-based approaches under approximate kNN sets of varying quality. Omit this option in order to use the exact kNN sets - recommended and default.
- **@mislabeling_weights_dir** is an optional parameter used to specify weights for weight-proportional random label noise. This option has been used for testing hubness-proportional random label noise in past experiments.
- **@normalization** specifies which normalization scheme to use. The permitted values are: no, normalizeTo01, standardize, tfidf.
- **@kernel** specifies the kernel to use in kernel clustering methods. Provide the class path to the desired kernel class. Optional.
- **@dataset** specifies a dataset to run the experiments on. Multiple occurrences of this option are used to specify multiple datasets for the experiments, so that they run in a batch mode. The first parameter is the relative path to the dataset in ARFF or CSV or TSV format, w.r.t. to the global data directory and the second and third parameter are metrics to be used for integer and float variables, respectively, given by their class path. A CombinedMetric object is then formed from these two feature-type specific metric objects. All features from CSV files are loaded into float feature arrays as default, but features from ARFF files can also be loaded as integers specifically, so this is why there are two separate metrics here. Also, some categorical variables that are represented as integers make no sense to use in standard float metrics, so this is a useful separation.

The achieved clustering configuration quality for each algorithm on each dataset is calculated via several clustering quality indexes. While more quality indexes are implemented in the library, a default set of quality indexes used in the batch clustering evaluation framework is as follows: Rand quality index, Rand stability index, isolation index, Dunn index, Silhouette index, average squared

error and average cluster entropy. Silhouette index 'a' and 'b' values are also given for hubs, anti-hubs and regular points separately. This is done in order to better evaluate how different types of points in intrinsically high-dimensional data are clustered.

As with classification, this configuration specification can be saved to JSON and loaded from JSON, in order to make it easy to automatically invoke the experimentation framework from a different class or remotely. A configuration object is also acceptable.

5.3 BATCH HUBNESS ANALYSIS

Apart from doing classification and clustering experimentation and evaluation, it is sometimes useful to do some exploratory analysis of the data and extract useful statistics.

When it comes to analyzing hubness, these statistics have to do with the k-nearest neighbor sets and the neighbor occurrence frequencies. There is a class that does batch data analysis of this type, with a similar configuration syntax to that of classification or clustering. These are the allowed options.

- **@in_directory** is used for specifying the data directory. The easiest approach is to simply have all data in one directory, in appropriate subdirectories. The following @dataset commands accept relative paths w.r.t. this specified top directory.
- **@out_directory** specifies the primary evaluation output target, where results of the exploratory analysis are to be persisted.
- **@k_max** specifies the maximum neighborhood size to consider. All values in the range $\{1 \dots k_{max}\}$ will be tried. Essentially, k_{max}-neighbor sets will be calculated initially and then the stats will be dynamically re-calculated for all smaller neighborhoods.
- **@noise_range** specifies the range of feature noise rates to test for. It has three parameters - the minimal value, the maximal value and the increment.
- **@mislabeled_range** specifies the range of label noise rates to test for. It has three parameters - the minimal value, the maximal value and the increment.
- **@mislabeling_weights_dir** is an optional parameter used to specify weights for weight-proportional random label noise. This option has been used for testing hubness-proportional random label noise in past experiments.
- **@common_threads** is used for specifying the number of threads to use for calculating the primary and secondary distance matrices and primary and secondary k-nearest neighbor sets.
- **@secondary_distance** specifies which secondary distance type to use, if doing metric learning in form of secondary distances in the experiments. The options are as follows: simcos, simhub, mp, ls, nicdm. If you are unsure about this option or options in general, you can look at configuration.BatchHubnessAnalysisConfig to see how the parsing is done and which values are expected. This option takes an optional second parameter that is the secondary neighborhood size to use for calculating the secondary distances, as some of them explicitly require this parameter. If omitted, a default value is used, which may or may not be appropriate for your data, so it is recommended to always specify it explicitly.
- **@normalization** specifies which normalization scheme to use. The permitted values are: no, normalizeTo01, standardize, tfidf.
- **@distances_directory** specifies where the distance matrices are to be saved or loaded from. Pre-calculated distance matrices can be loaded into the framework if provided at the expected location and the calculated distance matrices can also be saved for a later load. For instance, assume that I specify the distance directory as: K:\EXPERIMENTS\DMAT. The distance matrix for dataset0 if no normalization is done will be expected at

K:\EXPERIMENTS\DMAT\dataset0\NO\distances.primary.Manhattan location. The type of normalization is included in the path, so take care which feature normalization you specify. This is especially useful if you need to 'hack' the library to load a distance matrix for a distance that is not actually implemented in the library - but you need to use it in some experiments and you can calculate it externally. It is recommended to define a dummy class in the primary metrics directory and to specify that dummy metric for the datasets in question. You could in principle try to trick the system and present the distance matrix as originating from some of the standard metrics - but that is not a good approach, as you might also want to use standard metrics on the same data and you might forget about the hack in time and confuse the distance matrices in a few weeks when you get back to the experiment. Therefore, try to be as consistent and precise as possible. Another important detail is that - since the distance matrices are loaded if the system detects that they exist at the expected path - it is possible to encounter problems if you have datasets that share the exact same name, so try to avoid that scenario by either using separate distance directories in those cases (recommended) or deleting the matrices after a while. If the distance directory is left unspecified, the distances will be calculated on the fly and will not be saved for later use.

- **@dataset** specifies a dataset to run the experiments on. Multiple occurrences of this option are used to specify multiple datasets for the experiments, so that they run in a batch mode. The first parameter is the relative path to the dataset in ARFF or CSV or TSV format, w.r.t. to the global data directory and the second and third parameter are metrics to be used for integer and float variables, respectively, given by their class path. A CombinedMetric object is then formed from these two feature-type specific metric objects. All features from CSV files are loaded into float feature arrays as default, but features from ARFF files can also be loaded as integers specifically, so this is why there are two separate metrics here. Also, some categorical variables that are represented as integers make no sense to use in standard float metrics, so this is a useful separation.

As in other cases, configuration.BatchHubnessAnalysisConfig can be saved to JSON and loaded from a JSON string or file and the configuration object is then passed on to the batch hubness analysis experimental framework. This can automate some types of analysis. For manual experimentation, the human-readable format that we have described works better, as it is easier to modify and comments are also possible.

An example of the output of a hubness analysis on a dataset is given below. Apart from the basic stats like the number of instances and categories in the data, the number of zero vectors (to look for feature extraction problems), class priors, relative class imbalance and the number of dimensions, the analyzer calculates kNN-related stats. Items under stDevArray, skewArray, kurtosisArray correspond to the neighbor occurrence frequency for the specified k-range and 'bad hubness' lists the mislabeling percentages for different neighborhood sizes. Direct and reverse neighbor set entropy distributions follow, along with the percentage of points occurring above some small threshold, diameters of top hub sets, within-cluster distances of hub points, top hub occurrence frequencies, as well as a list of class-to-class neighbor occurrence matrices for each k value.

```
dataset: dataset0.arff
k_max: 10
noise: 0.0
ml: 0.0
instances: 1244
numCat: 10
nZeroVects: 0
class priors: 0.152 0.06 0.202 0.057 0.182 0.147 0.064 0.023 0.081 0.031
```

```
RelativeImbalance 0.20435232
dim: 100
-------------------------------------
stDevArray:
2.831,5.11,7.432,9.518,11.532,13.552,15.323,17.183,19.071,21.005
-------------------------------------
skewArray:
8.723,8.174,8.132,7.797,7.441,7.419,7.05,6.921,6.764,6.679
-------------------------------------
kurtosisArray:
105.12,93.226,92.541,86.821,80.346,80.084,72.869,70.639,67.442,65.616
-------------------------------------
bad hubness:
0.491,0.501,0.508,0.514,0.518,0.519,0.524,0.527,0.532,0.535
-------------------------------------
kEntropyMeans:
0.0,0.401,0.595,0.698,0.774,0.825,0.868,0.9,0.934,0.957
-------------------------------------
kEntropyStDevs:
0.0,0.49,0.528,0.545,0.547,0.549,0.543,0.55,0.557,0.562
-------------------------------------
kEntropySkews:
0.0,0.403,0.068,0.029,0.004,-0.047,-0.089,-0.069,-0.069,-0.107
-------------------------------------
kEntropyKurtosis:
0.0,-1.837,-1.254,-0.932,-0.647,-0.637,-0.552,-0.522,-0.473,-0.529
-------------------------------------
kRNNEntropyMeans:
0.149,0.283,0.374,0.457,0.523,0.583,0.645,0.693,0.737,0.766
-------------------------------------
kRNNEntropyStDevs:
0.445,0.578,0.673,0.737,0.771,0.797,0.812,0.828,0.838,0.849
-------------------------------------
kRNNEntropySkews:
2.908,1.916,1.529,1.298,1.129,0.974,0.849,0.762,0.687,0.652
-------------------------------------
kRNNEntropyKurtosis:
8.195,2.772,1.322,0.565,0.134,-0.263,-0.467,-0.643,-0.749,-0.793
-------------------------------------
kEnt - khEnt avgs:
-0.149,0.118,0.221,0.241,0.251,0.242,0.223,0.207,0.197,0.191
-------------------------------------
Hubness above zero percentage Array:
0.37,0.51,0.6,0.65,0.69,0.73,0.76,0.78,0.8,0.82
-------------------------------------
Hubness above one percentage Array:
0.19,0.32,0.42,0.48,0.53,0.57,0.61,0.64,0.66,0.68
-------------------------------------
Hubness above two percentage Array:
0.11,0.22,0.3,0.37,0.43,0.47,0.51,0.55,0.57,0.59
-------------------------------------
Hubness above three percentage Array:
0.06,0.15,0.22,0.29,0.34,0.38,0.42,0.46,0.49,0.52
-------------------------------------
Hubness above four percentage Array:
0.04,0.11,0.17,0.23,0.28,0.32,0.36,0.39,0.42,0.45
-------------------------------------
Top ten hubs diam:
47.75,46.13,46.13,46.13,46.13,46.13,46.13,39.45,33.78,33.78
-------------------------------------
Top ten hubs avg within cluster dist:
36.02,34.25,34.25,34.25,34.25,34.25,34.25,32.94,32.23,32.23
-------------------------------------
Top five hubs diam:
32.08,32.08,32.08,32.08,32.08,32.42,32.42,32.42,32.42,32.42
-------------------------------------
Top five hubs avg within cluster dist:
29.89,29.89,29.89,29.89,29.89,29.61,29.61,29.61,29.61,29.61
-------------------------------------
Highest occurrence frequencies (each line is for one k value, lines go from zero to k_max):
```

```
k: 1:: 10.0,11.0,11.0,11.0,12.0,13.0,16.0,16.0,17.0,18.0,24.0,25.0,37.0,41.0,44.0
k: 2:: 18.0,19.0,22.0,22.0,23.0,27.0,29.0,29.0,31.0,34.0,37.0,50.0,68.0,70.0,76.0
k: 3:: 28.0,29.0,32.0,34.0,35.0,37.0,38.0,42.0,44.0,48.0,52.0,83.0,89.0,99.0,115.0
k: 4:: 35.0,38.0,39.0,42.0,43.0,50.0,52.0,55.0,56.0,60.0,69.0,104.0,104.0,127.0,147.0
k: 5:: 45.0,45.0,51.0,52.0,55.0,61.0,61.0,65.0,67.0,80.0,86.0,114.0,121.0,152.0,177.0
k: 6:: 55.0,57.0,60.0,60.0,61.0,68.0,72.0,75.0,81.0,89.0,95.0,128.0,151.0,185.0,203.0
k: 7:: 63.0,65.0,69.0,70.0,73.0,76.0,77.0,85.0,92.0,101.0,107.0,139.0,169.0,209.0,220.0
k: 8:: 72.0,73.0,80.0,80.0,81.0,81.0,84.0,99.0,99.0,113.0,128.0,154.0,184.0,234.0,245.0
k: 9:: 84.0,85.0,85.0,86.0,89.0,92.0,95.0,108.0,119.0,127.0,138.0,170.0,204.0,256.0,268.0
k: 10:: 91.0,93.0,93.0,94.0,99.0,100.0,106.0,120.0,136.0,138.0,161.0,177.0,229.0,280.0,291.0
---------------------------------------
Global class to class hubness matrices for all K-s:
k = 1
0.519 0.065 0.165 0.057 0.06 0.051 0.029 0.011 0.028 0.015
0.0 0.701 0.0 0.0 0.146 0.076 0.014 0.007 0.056 0.0
0.005 0.0 0.922 0.005 0.005 0.002 0.0 0.0 0.005 0.056
0.01 0.0 0.019 0.951 0.0 0.0 0.0 0.0 0.01 0.01
0.0 0.004 0.0 0.0 0.724 0.143 0.081 0.028 0.021 0.0
0.0 0.0 0.0 0.0 0.0 0.875 0.0 0.0 0.125 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.999 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.029 0.0 0.969 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.008 0.0 0.0 0.991 0.0
0.0 0.0 0.023 0.0 0.0 0.0 0.0 0.0 0.0 0.975

k = 2
0.437 0.072 0.189 0.065 0.076 0.06 0.035 0.012 0.031 0.023
0.0 0.603 0.0 0.0 0.177 0.105 0.034 0.014 0.067 0.0
0.007 0.004 0.882 0.014 0.005 0.002 0.002 0.002 0.007 0.075
0.008 0.0 0.039 0.938 0.0 0.0 0.0 0.0 0.008 0.008
0.0 0.005 0.0 0.0 0.65 0.185 0.092 0.04 0.028 0.0
0.0 0.0 0.0 0.0 0.0 0.814 0.003 0.0 0.183 0.0
0.0 0.0 0.0 0.0 0.018 0.0 0.981 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.028 0.028 0.0 0.942 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.007 0.0 0.0 0.993 0.0
0.0 0.0 0.043 0.0 0.0 0.0 0.0 0.0 0.021 0.934

k = 3
0.402 0.078 0.194 0.07 0.092 0.058 0.036 0.013 0.036 0.023
0.004 0.529 0.0 0.0 0.205 0.108 0.05 0.025 0.079 0.0
0.01 0.004 0.859 0.014 0.005 0.005 0.003 0.001 0.01 0.09
0.013 0.0 0.057 0.911 0.0 0.0 0.006 0.0 0.006 0.006
0.0 0.005 0.0 0.0 0.606 0.21 0.105 0.042 0.031 0.0
0.0 0.0 0.0 0.0 0.0 0.795 0.004 0.0 0.2 0.0
0.0 0.0 0.0 0.0 0.017 0.0 0.983 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.025 0.025 0.025 0.923 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.005 0.0 0.0 0.994 0.0
0.0 0.0 0.059 0.0 0.0 0.0 0.0 0.0 0.02 0.92

k = 4
0.381 0.082 0.193 0.072 0.097 0.062 0.036 0.014 0.041 0.021
0.003 0.485 0.003 0.0 0.214 0.125 0.054 0.021 0.095 0.0
0.01 0.007 0.835 0.02 0.007 0.009 0.004 0.001 0.011 0.098
0.011 0.0 0.057 0.909 0.0 0.0 0.006 0.0 0.011 0.006
0.0 0.004 0.0 0.0 0.584 0.221 0.114 0.044 0.033 0.0
0.0 0.0 0.0 0.0 0.0 0.787 0.004 0.0 0.209 0.0
0.0 0.0 0.0 0.0 0.0 0.023 0.0 0.976 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.022 0.045 0.022 0.909 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.005 0.0 0.0 0.995 0.0
0.0 0.0 0.053 0.0 0.0 0.0 0.0 0.0 0.035 0.911

k = 5
0.369 0.08 0.197 0.073 0.102 0.062 0.037 0.016 0.042 0.022
0.002 0.464 0.002 0.0 0.24 0.119 0.056 0.017 0.1 0.0
0.013 0.007 0.826 0.018 0.006 0.01 0.006 0.001 0.013 0.1
0.024 0.005 0.056 0.873 0.005 0.005 0.005 0.0 0.009 0.019
0.0 0.005 0.0 0.0 0.564 0.234 0.118 0.045 0.034 0.0
0.0 0.0 0.0 0.0 0.0 0.769 0.003 0.002 0.226 0.0
0.0 0.0 0.0 0.0 0.035 0.0 0.965 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.021 0.042 0.021 0.894 0.021 0.0
0.0 0.0 0.0 0.0 0.0 0.009 0.0 0.0 0.991 0.0
```

```
0.0 0.0 0.05 0.0 0.0 0.0 0.0 0.0 0.033 0.915

k = 6
0.36 0.081 0.194 0.074 0.104 0.067 0.038 0.016 0.043 0.024
0.006 0.447 0.002 0.0 0.24 0.125 0.058 0.017 0.102 0.002
0.013 0.006 0.82 0.02 0.008 0.011 0.006 0.002 0.017 0.097
0.033 0.004 0.058 0.859 0.004 0.008 0.004 0.0 0.008 0.021
0.0 0.005 0.0 0.0 0.555 0.237 0.12 0.047 0.037 0.0
0.0 0.0 0.0 0.0 0.003 0.754 0.005 0.002 0.236 0.0
0.0 0.0 0.0 0.0 0.032 0.0 0.961 0.006 0.0 0.0
0.0 0.0 0.0 0.0 0.019 0.058 0.019 0.864 0.039 0.0
0.0 0.0 0.0 0.0 0.0 0.012 0.0 0.0 0.988 0.0
0.0 0.0 0.075 0.0 0.0 0.0 0.0 0.0 0.03 0.894

k = 7
0.353 0.084 0.197 0.073 0.106 0.068 0.038 0.016 0.043 0.022
0.009 0.41 0.004 0.0 0.254 0.143 0.059 0.014 0.106 0.002
0.014 0.007 0.807 0.021 0.008 0.01 0.006 0.001 0.021 0.104
0.033 0.004 0.066 0.854 0.004 0.007 0.007 0.0 0.007 0.018
0.0 0.005 0.0 0.0 0.546 0.241 0.123 0.047 0.039 0.0
0.0 0.0 0.0 0.0 0.004 0.744 0.005 0.001 0.245 0.0
0.0 0.0 0.0 0.0 0.03 0.006 0.952 0.006 0.006 0.0
0.0 0.0 0.0 0.0 0.018 0.053 0.018 0.876 0.035 0.0
0.0 0.0 0.0 0.0 0.0 0.015 0.0 0.0 0.985 0.0
0.0 0.0 0.071 0.0 0.0 0.0 0.0 0.0 0.029 0.899

k = 8
0.348 0.084 0.197 0.072 0.109 0.068 0.039 0.016 0.043 0.023
0.008 0.395 0.003 0.0 0.25 0.15 0.062 0.014 0.11 0.006
0.014 0.007 0.801 0.023 0.009 0.011 0.006 0.002 0.024 0.104
0.042 0.007 0.068 0.837 0.003 0.007 0.007 0.0 0.013 0.016
0.0 0.005 0.0 0.0 0.539 0.243 0.125 0.049 0.039 0.0
0.0 0.0 0.0 0.0 0.005 0.738 0.005 0.001 0.251 0.0
0.0 0.0 0.0 0.0 0.028 0.006 0.954 0.006 0.006 0.0
0.0 0.0 0.0 0.0 0.017 0.05 0.033 0.849 0.05 0.0
0.0 0.0 0.0 0.0 0.0 0.014 0.0 0.0 0.986 0.0
0.0 0.0 0.069 0.0 0.0 0.0 0.0 0.0 0.028 0.902

k = 9
0.341 0.083 0.197 0.072 0.112 0.069 0.04 0.017 0.045 0.023
0.009 0.388 0.003 0.0 0.251 0.156 0.062 0.014 0.112 0.006
0.016 0.008 0.791 0.025 0.011 0.012 0.007 0.002 0.023 0.104
0.039 0.009 0.069 0.834 0.006 0.006 0.006 0.0 0.015 0.015
0.0 0.007 0.0 0.0 0.531 0.248 0.127 0.048 0.04 0.0
0.0 0.0 0.0 0.0 0.009 0.729 0.006 0.001 0.255 0.0
0.0 0.0 0.0 0.0 0.028 0.011 0.95 0.006 0.006 0.0
0.0 0.0 0.0 0.0 0.016 0.046 0.062 0.83 0.046 0.0
0.0 0.0 0.0 0.0 0.0 0.013 0.0 0.0 0.987 0.0
0.0 0.0 0.079 0.0 0.0 0.0 0.0 0.0 0.026 0.894

k = 10
0.337 0.083 0.198 0.072 0.113 0.07 0.04 0.017 0.046 0.025
0.008 0.369 0.003 0.0 0.253 0.169 0.063 0.017 0.114 0.005
0.017 0.009 0.787 0.027 0.01 0.014 0.007 0.002 0.024 0.102
0.042 0.008 0.076 0.823 0.006 0.008 0.006 0.0 0.014 0.017
0.0 0.006 0.0 0.0 0.527 0.249 0.127 0.048 0.042 0.0
0.0 0.0 0.0 0.0 0.009 0.727 0.005 0.001 0.258 0.0
0.0 0.0 0.0 0.0 0.026 0.01 0.953 0.005 0.005 0.0
0.0 0.0 0.0 0.0 0.015 0.045 0.06 0.835 0.045 0.0
0.0 0.0 0.0 0.0 0.0 0.015 0.0 0.0 0.984 0.0
0.0 0.0 0.076 0.0 0.0 0.0 0.0 0.0 0.025 0.898
```

Six

The Data Model

The current Hub Miner data model is simple, yet flexible. At the same time, it is not too complicated to use. In case that additional representational capacity is needed for an application, it is always possible to extend the corresponding classes in a straightforward way. The basic class hierarchy for data representation is shown in Figure 6.1. Additional classes in data.representation.images extend the basic dense data holders.

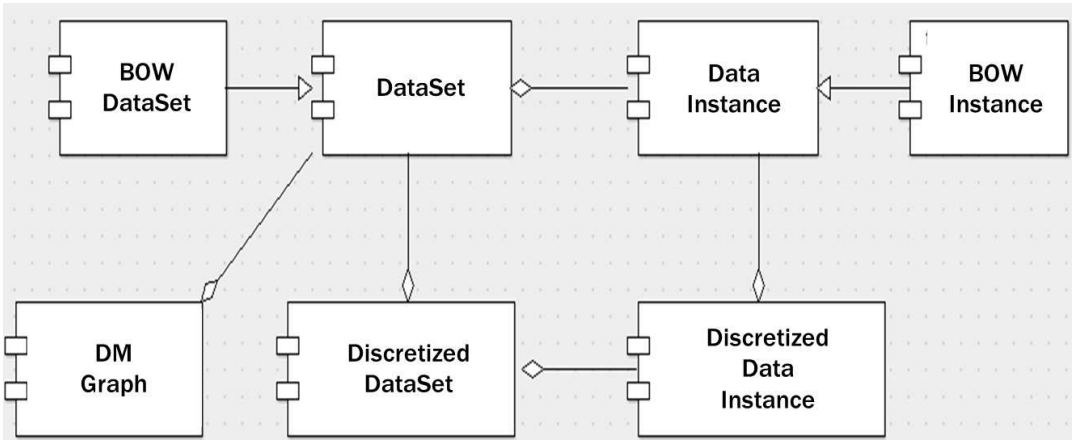


Figure 6.1: The basic Hub Miner data model.

The idea is simple. DataSet objects represent dense datasets as lists of DataInstance objects. Also, DataSet objects hold the representation definition in terms of feature names. What follows are the current variable declarations in DataSet objects:

```
public class DataSet implements Serializable {

    private static final long serialVersionUID = 1L;
    private String name;
    // The corresponding DataSet object holding the identifiers of these
    // feature representations, represented as DataInstances. This way of
    // representing the data is optional, but allows for more complex keys and
    // splitting the features that machine learning is to be based upon from
    // the auxiliary data features that can still be contained within the
    // identifier context.
    public DataSet identifiers = null;
```

```
// Feature names.
public String[] iAttrNames = null;
public String[] fAttrNames = null;
public String[] sAttrNames = null;
// A map between feature names and their indexes.
private HashMap<String, Integer> attNameMappings;
// A list of data instances.
public ArrayList<DataInstance> data;
private static final int DEFAULT_INIT_CAPACITY = 1000;
private int initCapacity;
```

The basic data holders can handle three different feature types: float features, integer features and nominal features. Of course, categorical values can also be represented as integers, but then special care should be taken with regards to how distance measures are used and which can be applied to which feature. The Hub Miner version of ARFF file format supports the keyword **integer** for a feature type and these features are then loaded into the integer part of the data representation. This is a slight extension of the original ARFF format, as all numbers are treated equally there.

DataSet class implements many useful methods for data handling. Method addDataInstance inserts a new data instance into the DataSet. A copy of the current data definition in terms of feature names can be produced via the cloneDefinition method, which outputs an empty DataSet object with the same feature specification. If the data is to be copied as well, the copy method is available. It is possible to normalize features via normalizeFloats and standardizeAllFloats methods. Checking for missing values in the data is trivial, as there is a hasMissingValues method. Introducing label noise into the data for experiments can be done by invoking the induceMislabeling method. Embedding the data into a single cluster can be achieved by makeClusterObject. Merging two disjoint views of the same data, two different feature representations, can be done via the mergeDisjointRepresentations method. It is possible to directly calculate the distance matrix by invoking calculateDistMatrix and there is also a multi-threaded variant readily available. There are many more methods implemented within DataSet and also many auxiliary methods for handling DataSet objects that are available in other Hub Miner classes.

Creating a DataInstance object that conforms to a particular feature representation defined in the DataSet object is easily done by using the proper DataInstance constructor, as follows.

```
DataSet dset = new DataSet();
dset.fAttrNames = new String[2];
dset.fAttrNames[0] = "Height";
dset.fAttrNames[1] = "Width";
DataInstance instance = new DataInstance(dset);
```

This creates a DataSet feature representation that consists of two float features: height and width. The subsequent DataInstance constructor then generates a DataInstance object that has a float feature value array of length two, corresponding to the feature representation of the DataSet. However, the instance is not automatically included in the collection and the data context in the DataInstance object is not automatically set. If one wishes to include the instance, the following code should be included.

```
// Set the data context.
instance.embedInDataSet(dset);
```

```
// Include the instance into the dataset list.
dset.addDataInstance(instance);
```

DataInstance objects have the following variable declarations.

```
public class DataInstance implements Serializable {

    private static final long serialVersionUID = 1L;
    // The DataSet that defines the feature types for this data instance.
    private DataSet dataContext = null;
    // The label of the data instance.
    private int category = 0;
    // Support for fuzzy labels is slowly being added throughout the code. Most
    // methods work with crisp labels, as is customary.
    public float[] fuzzyLabels = null;
    // Feature values.
    public int[] iAttr = null;
    public float[] fAttr = null;
    public String[] sAttr = null;
    // Identifier, which can be composed of multiple values and is thus also
    // represented as a data instance.
    private DataInstance identifier = null;
```

Three feature value arrays correspond to the three feature name arrays in the DataSet object. The dataContext variable holds a link to the parent DataSet object. It can be used either to signal where the feature representation definition is or to signal that the instance is contained within the object. The feature arrays in the DataInstance need to correspond to those in the DataSet object that is the dataContext and that contains the instance. If there is a mismatch in lengths, errors may occur - since this is not a valid object state.

Instance label is contained in the category field. There is also a fuzzyLabels field if fuzzy labels are to be defined, but this is still experimental and not fully supported in the current Hub Miner release. Users are encouraged to include it in their own applications and extend the basic Hub Miner functionality, if they require fuzzy training labels in their own experiments.

The identifier field is also of DataInstance type and it is used to hold multi-values primary keys for the instances, in cases when such keys exist, when instances are pulled from a database. If you are simply analyzing a UCI dataset, this field will not be utilized. It might, however, be useful in some practical applications.

DataInstance class implements many useful methods for data handling. These include equality checks, checks for missing values and noise, consistency checks, as well as other utility methods. It is also possible to add or average a list of DataInstance objects, which is useful in many calculations. There is also a toString() method, which means that DataInstance objects can easily be printed to any standard output. This is very useful during de-bugging, though it can be used to easily persist the results. In general, for results I/O, classes in the ioformat package are to be preferred, as they enable save/load in/from ARFF, CSV and TSV formats.

While DataSet and DataInstance objects make convenient representations for dense data, they would not be appropriate for sparse data representations, like for instance bag-of-words in text mining. Therefore, Hub Miner offers BOWDataSet and BOWInstance objects for this particular purpose. For instance, here are the declarations in case of BOWDataset:

```
public class BOWDataSet extends DataSet {
```

```
    public static final int DEFAULT_INIT_WORDS = 20000;
    // The number of words in the vocabulary.
    private int numWords = -1;
    // Map of words to their indexes in the vocabulary.
    private HashMap<String, Integer> vocabularyHash;
    // Vocabulary of all the represented words.
    private ArrayList<String> vocabulary = null;
    // Word frequencies.
    private ArrayList<Float> wordFrequencies = null;
```

This class extends the DataSet class, so mixed data can be easily represented, as multimedia data in particular can often contain both sparse and dense representation parts. Consider images with their captions or associated comments and descriptions. The textual part of the representation is usually sparse (unless we consider tags), but image features are usually not. The sparse representation is simple. A single HashMap could also have been used, but it is often useful to have immediate access to the vocabulary, so there is one HashMap that maps a word to an index in the frequency and vocabulary list - and these lists are maintained separately. Methods for easily inserting words and instances into the sparse dataset are available, so users do not have to think about doing these updates manually. BOWInstance objects also extend DataInstance objects.

```
public class BOWInstance extends DataInstance {

    public static final int INIT_HASH_SIZE = 500;
    // This map stores the mapping between word indexes from the vocabulary
    // and their weights or counts in the current document.
    private HashMap<Integer, Float> wordIndexHash =
            new HashMap<>(INIT_HASH_SIZE);
    // The data context variable here is named corpus.
    public BOWDataSet corpus;
    // Name or path of the document from which the data was extracted, if
    // available.
    public String documentName;
```

Each BOWInstance contains a HashMap that maps word indexes to their weight, which is the standard bag-of-words representation. For readability, an additional data context variable named corpus that is of the BOWDataSet type is introduced and returned in the context query methods. An additional String value is present to mark document names, when necessary. BOWInstance objects do not store raw text, as this is rarely needed at this stage in the analysis. Such data can be stored separately.

While it is certainly true that integer features can be used to represent value ranges when float features are discretized according to pre-computed divisions, this is not the most fortunate generic solution. This is why Hub Miner allows for a more fine-grained control over discretized features and this is achieved in DiscretizedDataSet and DiscretizedDataInstance. Unlike their sparse counterparts, the design decision here was not to let them directly extend DataSet and DataInstance - since the discretized data is not usable in same types of methods as non-discretized data, which was not the case with sparse data. Namely, it makes little sense to apply kNN to discretized data (while it is still possible with some carefully chosen metric), but other types of methods like decision trees expect discretized data as input. This rather clear separation has lead to separate pipelines for discretized and non-discretized data within Hub Miner. Separate types of classification algorithms implement the predictive interfaces for these different types of instances and datasets.

```
public class DiscretizedDataSet implements Serializable {

    private static final long serialVersionUID = 1L;

    // Original DataSet from which this one was created.
    private DataSet originalData;
    // Data points.
    public ArrayList<DiscretizedDataInstance> data = null;
    // Discretization structures.
    private HashMap<String, Integer>[] nominalHashes = null;
    private int[] hashSizes = null;
    private ArrayList<String>[] nominalVocabularies = null;
    // By convention, they define [) intervals and are supposed to be ordered.
    // IMPORTANT: Float/Integer -MAX_VALUE / MAX_VALUE are supposed to be at the
    // front and the back of the interval definitions. This is implicitly
    // assumed in the methods defined below.
    private int[][] intIntervalDivisions = null;
    private float[][] floatIntervalDivisions = null;
```

The original DataSet reference is present in DiscretizedDataSet variables, as all discretized dataset representations in Hub Miner are expected to be derived from some specific continuous representation. If this is not the case and the data was externally loaded, this can be generically filled or left as null. Like DataSet, DiscretizedDataSet also contains a list of instances, only DiscretizedDataInstance objects in this case. It also holds the discretization definitions, the value ranges for float and integer variables - and a HashMap for nominal values, which are replaced by the corresponding indexes in their feature vocabularies.

```
public class DiscretizedDataInstance implements Serializable {

    private static final long serialVersionUID = 1L;

    private DiscretizedDataSet dataContext = null;
    private DataInstance originalDataInstance;
    private int category = -1;
    private float[] fuzzyLabels = null;
    // Discrete arrays all have discrete index values pointing to certain
    // ranges of values in the original continous spectrum. The definition of
    // the ranges can be found in the embedding dataContext.
    public int[] integerIndexes = null;
    public int[] floatIndexes = null;
    public int[] nominalIndexes = null;
```

DiscretizedDataInstance object are designed in a similar fashion to DataInstance objects. They contain separate discretized value index arrays for floats, integers and nominal variables, as well as crisp and fuzzy labels. A reference to the embedding discrete data context and a reference to the original DataInstance that the DiscretizedDataInstance was derived from are also present.

So, how does one automatically generate a DiscretizedDataInstance from a DataInstance object? In fact, there are methods in Hub Miner that enable the users to automatically discretize entire datasets.

For instance, consider the class EntropyMDLDiscretizer that is located in data.representation.discrete.tranform package.

```
DataSet dset = ... \\ Some data load code here ;
```

```
DiscretizedDataSet dsetDisc = new DiscretizedDataSet(dset);
EntropyMDLDiscretizer discretizer = new EntropyMDLDiscretizer(dset, dsetDisc, 10);
// Use the discretizer to discretize the data.
discretizer.discretizeAll();
dsetDisc.discretizeDataSet(dset);
```

Like the discretizedAll method in EntropyMDLDiscretizer, there is also discretizeAllBinary, as well as separate discretization methods for each feature type. The former tries to find the optimal number of splits for each variable, while the latter always splits the value range into two interval sub-ranges. Certainly, the entropy minimum-description-length discretized implemented in Hub Miner is not the only such transformation filter that could be used and additional discretization filters might be introduced in the future.

Hub Miner offers basic support for operations on graphs, as hubness is a phenomenon on k-nearest neighbor graphs in intrinsically high-dimensional data. While these graphs are rarely explicitly represented in hubness-aware analysis, it is sometimes useful to be able to visualize them and analyze them as graphs. The support for graph data representations in Hub Miner is currently basic and it will be modified and extended in future releases, even though it is not the top priority since graph analysis is a vast field with many methods - and these methods are not the focus of the Hub Miner library nor the researchers performing research on hubness and high-dimensional data mining.

```
public class DMGraph {

    public DataSet vertices;
    public DMGraphEdge[] edges;
    public String networkName;
    public String networkDescription;
    // Copying a DMGraph object will not copy the JGraph reference, by default.
    // JGraph is only used when the results are to be displayed on the screen.
    // JGraph is not always used for graph drawing in this library, so this is
    // more of a backward-compatibility thing.
    public JGraph visGraph = null;
```

The basic supported static graph representation is very simple. DataInstance objects are used as vertices, which makes it easy to find correspondence between graph nodes and the underlying representation, in case of k-nearest neighbor graphs. DMGraphEdge is the name of the class that represents weighted edges between different nodes. Graphs can also contain names and descriptions, as well as a reference to a JGraph visualization context, in case that JGraph is used for visualization. Image Hub Explorer relies on JUNG, for instance.

```
public class DMGraphEdge implements Serializable {

    public double weight = 0;
    public int first;
    public int second;
    // Edges are maintained as linked lists.
    public DMGraphEdge next = null;
    public DMGraphEdge previous = null;
```

DMGraphEdge objects are very simple. They are connected as doubly linked lists and contain start and end vertex indices, as well as the associated weight. In DMGraph, each element in the

edges array corresponds to a start of a list holding the edges that start at a given node. This makes it easy to access all edges corresponding to a specified vertex.

While not the optimal graph representation, these basic graph classes in Hub Miner enable for some analysis to be done along with the visualization. The graph.subgraphs package contains classes that enable users to find the connected components in a graph, perform cuts and select subgraphs. The graph.calc.GraphStatistics class contains methods for calculating degree and closeness centrality for vertices in the graph.

Since this initial support may not suffice for all applications, it is possible to export DMGraph objects in Pajek-compatible format, which means that it is possible to use Pajek (`http://pajek.imfm.si/doku.php?id=pajek`), a well-known network analysis tool, for subsequent analysis of kNN graphs on intrinsically high-dimensional data. This export was the original intent of introducing the DMGraph structure in Hub Miner.

Seven

Hubness-aware Implementations

Since Hub Miner is primarily a library for hubness-aware machine learning and data analysis, special attention is given to hubness-aware implementations. This chapter covers hubness-aware methods for classification, clustering, metric learning and instance selection. It does not cover the exploratory framework for analysing hubness in intrinsically high-dimensional data, that is also part of Hub Miner.

The current draft of this manual omits the equations used to infer hubness-aware models, as they are available in the cited papers where the methods were first proposed. Instead, this chapter gives a brief overview of the ideas behind each algorithm, as well their location in the Hub Miner class hierarchy. Interested readers are encouraged to look up the details in external material, which is freely available online.

7.1 CLASSIFICATION

Several hubness-aware classification methods have recently been proposed [RNI09][TRMI11b] [TRMI13a][TRMI11a][TM12b][TM13b] and Hub Miner contains their implementations.

The initial way of dealing with detrimental hub points in kNN votes was to assign them a lower voting weight. This was proposed in hw-kNN [RNI09], a simple yet effective algorithm that was a proof of concept that hubness-aware methods can be made and that the negative effects of hubness in the data can be reduced. This algorithm is implemented in HwKNN class in learning.supervised.methods.knn package. Other hubness-aware kNN methods are contained within the same package.

The initial hw-kNN algorithm did not take into account class-conditional neighbor occurrences and this was rectified in the hubness-aware extension of the fuzzy k-nearest neighbor framework, h-FNN [TRMI11b][TRMI13a]. The weighted counterpart of h-FNN, dwh-FNN, usually achieves comparable, though slightly higher accuracy. These algorithms are implemented in HFNN and DWHFNN classes in the learning.supervised.methods.knn package. Since it is impossible to form hubness-based fuzzy votes for orphan and anti-hub points directly, a special anti-hub handling mechanism needs to be employed in order to properly define votes for such points, since they occasionally do occur as neighbors on the test data, despite the fact that they were never observed as neighbors on the training data.

A later extension of h-FNN and dwh-FNN took into account the differences in the information content of different neighbor occurrences, as hubs have been judged as less informative in general. This algorithm was named hubness-information k-nearest neighbor (HIKNN) [TM12b] and it included absolute and relative surprise factors for forming neighbor votes without the anti-hub θ threshold of h-FNN and dwh-FNN. Two variants of the HIKNN approach are present in learning.supervised.methods.knn package. One variant does not employ additional distance-based

weighting and it is given in the class HIKNNNonDw. The weighted counterpart is implemented in HIKNN class.

The naive Bayesian re-interpretation of k-nearest neighbor sets was shown to yield very good results in intrinsically high-dimensional data under the assumption of hubness, especially in presence of class imbalance. This method was named NHBNN (naive hubness-Bayesian k-nearest neighbor) [TRMI11a] and it is present in the NHBNN class in the same learning.supervised.methods.knn package.

An extension of NHBNN was proposed that takes neighbor co-occurrences into account and that is able to outperform NHBNN on high-hubness datasets for larger neighborhood sizes. This extension was named the augmented naive hubness-Bayesian k-nearest neighbor [TM13b]. Across a wider range of datasets, though, NHBNN still proves more robust in our experiments. However, this has to do with the current ANHBNN design and it is possible to learn different types of hub co-occurrence models, some of which might be more robust in absence of co-occurrence information for most neighbor pairs. Namely, high data hubness actually increases the number of pairs for which the algorithm is able to derive meaningful class-conditional co-occurrence probabilities and therefore calculate proper mutual information and related measures. It might also be the case that better handling of those low-or-no-information cases for pair co-occurrences could improve ANHBNN performance on low-hubness datasets. Nevertheless, as it was designed for classification under the assumption of hubness, the algorithm is quite useful in those cases where it is applicable, in its current form. The algorithm is given in ANHBNN class, in the same package as the rest.

It is possible to perform neighbor re-ranking in k-nearest neighbor sets based on their hubness and their bad hubness in particular. This was exploited in RRKNN (re-ranked k-nearest neighbor) [TM14b]. RRKNN employs secondary re-ranking of k-nearest neighbor sets that was first proposed for improved bug report duplicate detection [TLM13] in presence of high hubness in textual bug report data. The algorithm performs secondary re-ranking on the original k-nearest neighbor set and then learn a set of secondary local distances that are used to re-rank the neighbors. Since this does not change the content of the original k-nearest neighbor set (the distances are only re-computed locally for the neighbors), a smaller neighborhood size (usually $k/2$) is used to perform the actual voting. The algorithm is available in RRKNN class in the same learning.supervised.methods.knn package.

In principle, RRKNN approach could be applied not only to kNN but also to other hubness-aware methods and we have experimented with this, but - the accuracy gains were mostly insignificant. These prototype classes were therefore not included in the current Hub Miner release.

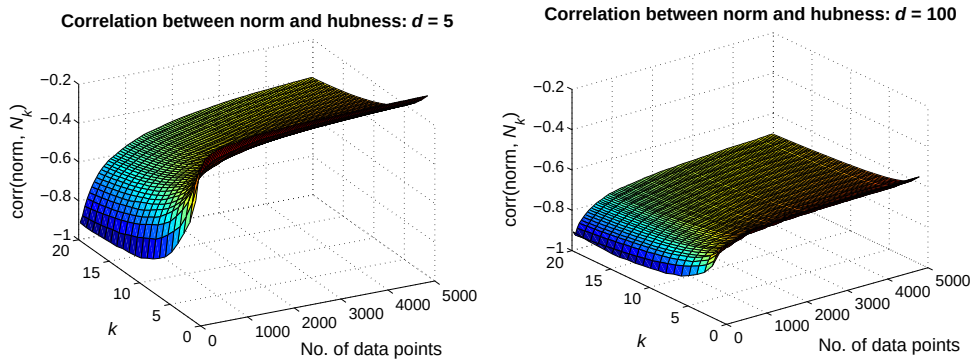
It was recently demonstrated that it is possible to boost hubness-aware classifiers and that certain types of boosting result in implicit inner ensembles in a sense that the resulting classifier has the same model form as the base hubness-aware approaches. While they are not weak classifiers, it is still possible to effectively boost hubness-aware approaches due to the fact that the k-nearest neighbor sets need only be calculated once on the training data and boosting can be done via iterative instance re-weighting without re-sampling [Tom14]. These initial experiments were performed within the boosting framework of Adaboost.M2, which is present as AdaBoostM2 in learning.supervised.meta.boosting package. Boostable implementations of hubness-aware base learners that support instance weighting and can therefore be used within the framework are given in learning.supervised.meta.boosting.baselearners. In particular, classes HFNNBoostable, DWHFNNBoostable, HIKNNBoostable and HwKNNBoostable.

Classification experiments in Hub Miner can be performed under introduced random label noise. This type of experiments is used to determine the algorithm robustness in noisy data under controlled noise rates. While the default protocol is to use the uniform random label noise in such experiments,

it is also possible to use weight-proportional random label noise and this was used to conduct experiments with hubness-proportional random label noise in one recent publication. This is a somewhat unique feature of Hub Miner, to have this option already included and available in the default classification evaluation protocol and configuration files. Hubness-proportional random label noise can be used to test for predicted worst-case classifier performance in noisy data, as hub-centered noise induces a much higher misclassification rate than uniform noise [TB14].

7.2 CLUSTERING

It was experimentally shown that point-wise hubness (neighbor occurrence frequency) is highly correlated with local cluster centrality in intrinsically high-dimensional data and this has been exploited for clustering. An example is shown in Figure 7.1, for synthetic zero-centered i.i.d. Gaussian data. Not only does point-wise hubness become a good indicator of closeness to local cluster centers, but density becomes less correlated with centrality with increasing data dimensionality. In a sense, hubness is used as a replacement for density estimates in many dimensions.



(a) Correlation between norm and hubness for $d = 5$ in Gaussian i.i.d. data.

(b) Correlation between norm and hubness for $d = 100$ in Gaussian i.i.d. data.

Figure 7.1: Interaction between norm and hubness, in low- and high-dimensional scenarios.

Hubs have been shown to make promising candidates for cluster representatives and selecting a set of hubs instead of centroids or medoids can lead either to faster convergence or better cluster configurations, in certain high-dimensional settings [TRMI11c][TRMI13b][TRMI14]. An illustrative low-dimensional example of why hubs might be preferable as prototype points is shown in Figure 7.2.

Hub Miner implements several hubness-based clustering methods that exploit this newly discovered property in many dimensions. These methods are located in the learning.unsupervised.methods package. LKH (Local K-hubs) and GKH (Global K-hubs) are the simplest hubness-based methods, proof-of-concept-like. They are not very effective in practice, as they sometimes get stuck in suboptimal hub configurations and they converge very quickly. These methods are simple extensions of K-means. The stochastic variants of the two methods, LHPC (local hubness-proportional clustering) and GHPC (global hubness-proportional clustering) perform much better, as they avoid local cluster configuration optima. GHPKM is a further exension of GHPC, where centroids are selected in deterministic iterations and hubs in stochastic iteration, via a square-hubness-proportional stochastic

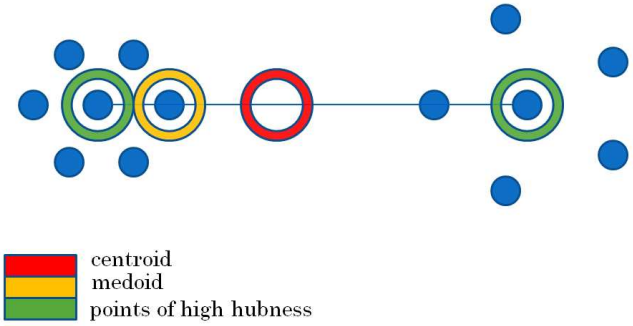


Figure 7.2: Illustrative example. The red dashed circle marks the centroid (C), yellow dotted circle the medoid (M), and green circles denote two elements of highest hubness (H_1, H_2), for neighborhood size 3. In this particular example, it is clear that selecting hubs as cluster prototypes would go directly to the centers of local sub-groups and speed up convergence.

framework. This hybrid method combines the advantages of two different types of approaches and outperforms both in many real-world examples.

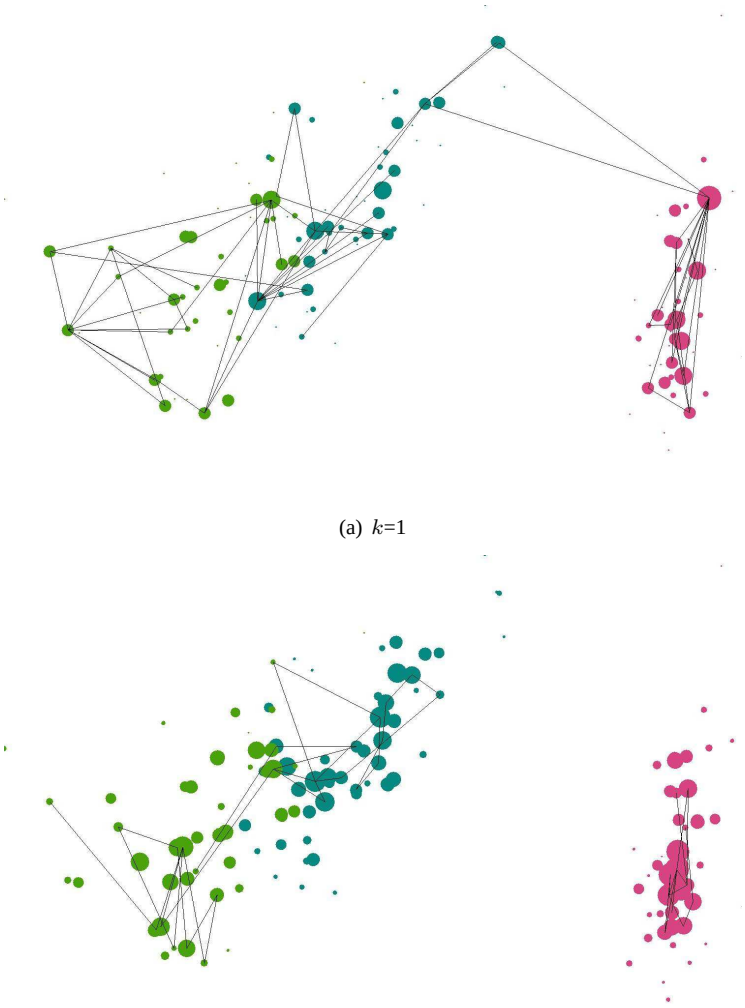
LKH, GKH, LHPC, GHPC and GHPKM have been shown to perform well in many cases and to be quite robust to high rates of uniform noise. However, they are incapable of detecting non-hyper-spherical clusters by design. Therefore, in order to enable clusters of more arbitrary shapes to be formed in a hubness-based framework, an extension of kernel K-means was proposed, kernel-GHPKM [TRMI14]. This algorithm is (logically) implemented in the KernelGHPKM class in the learning.unsupervised.methods package. In the clustering evaluation experimental framework, users can experiment with different types of kernels in order to test and evaluate both kernel-GHPKM and kernel K-means.

Hub Miner also implements some tools for clustering progress visualization in case of stochastic hubness-proportional clustering methods. An example is given in the learning.unsupervised.visualization package, in the HPCVisualizer class. This class generates visualizations like the one shown in Figure 7.3. It can be seen that the cluster prototype search goes mostly through central cluster regions (even in this low-dimensional case) and that many configurations are tested in the process.

7.3 Metric Learning

Hubness arises in many commonly used metrics in high-dimensional data. Euclidean, Manhattan and cosine are among the most commonly used distance/similarity measures and they are known to exhibit substantial hubness in many dimensions. Fractional distances might lead to slight improvements occasionally, but not enough to really be a game-changer. Also, dimensionality reduction does not really help. Reducing the number of dimensions enough to suppress hubness in the data entails substantial information loss and the effectiveness of the methods is affected in different ways.

While different feature representations and different primary distance or similarity measures might induce varying degrees of hubness in the data - it is often impractical to test all possible



(a) k=1

(b) k=10

Figure 7.3: Hubness-guided search for the best cluster hub-configuration in global hubness-proportional clustering on Iris data.

feature extraction pipelines and distance measures in order to reach the optimal system configuration. Furthermore, sometimes a satisfactory configuration would not even be available among the tested primary approaches. This is why metric learning in form of secondary distance measures has been proposed as a way of reducing hubness in the data and handling high-hubness data in similarity-based systems in general [ZmP04][JHS07][SFSW12][HKK$^+$10][TM12a][TM14a].

Hub Miner implements several secondary metrics that help with learning in intrinsically high-dimensional data, in presence of hubness. Two local techniques, local scaling and NICDM, are

implemented in the distances.secondary package, in classes LocalScalingCalculator and NICDMCalculator. Mutual Proximity is a global scaling approach based on mutual neighbor relation probabilities and it is given in MutualProximityCalculator class in the same package. These calculator classes permit both single-threaded and multi-threaded calculations. Mutual proximity is an especially promising approach that was shown to substantially reduce hubness in many real-world high-dimensional datasets.

Shared-neighbor secondary similarities are available in the distances.secondary.snd package, though shared neighbor calculations are actually performed by the SharedNeighborFinder class in data.neighbors. Both $simcos_s$ and $simhub_s$ are represented by the SharedNeighborCalculator class. Since $simhub_s$ is a weighted extension of $simcos_s$, it was natural to group the two implementations together in the same class. The effectiveness of $simhub_s$ and $simcos_s$ in not only reducing the hubness in the data but also improving the semantical consistency among the neighbor sets can be glimpsed from an illustrative example in Figure 7.4.

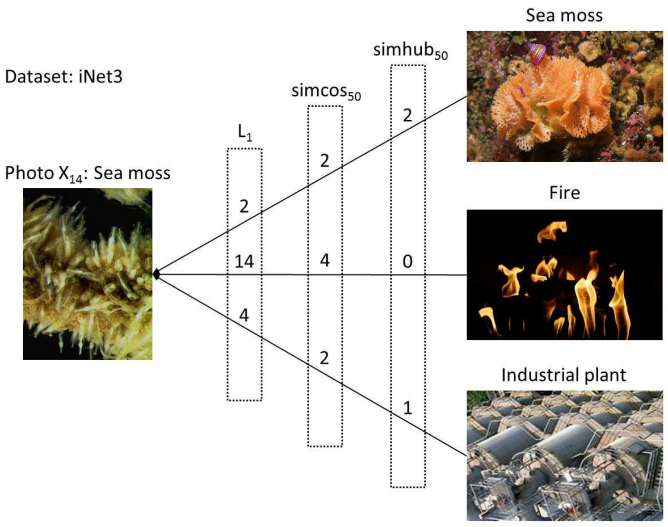


Figure 7.4: An illustrative example of how secondary distances ($simcos_s$ and $simhub_s$) affect the consistency of the reverse k-nearest neighbor sets in image data and the consistency of hub occurrences in particular.

Underwater sea moss images are often confused with fire images taken in the dark and the Example in Figure 7.4 shows how the sea moss photo X_{14} acts as a neighbor. In the primary L_1 distance, it is a detrimental hub image, a neighbor to many points in the fire images class. Application of $simcos_{50}$ and $simhub_{50}$ in turn reduces both the neighbor occurrence frequency of X_{14} as well as the frequency of semantically incorrect occurrences.

7.4 INSTANCE SELECTION

Instance selection is often used in conjunction with k-nearest neighbor classification, as kNN methods do not scale well unless approximate kNN calculations are used. Also, careful instance selection strategies can sometimes even improve kNN performance.

Neighbor occurrence frequency has been pinpointed as a useful instance selection criterion in high-dimensional data, which gave rise to some reverse-neighbor-based selection methods [DH11], as well as hubness-aware instance selection [BNST11a]. The former is implemented in the RNNR_AL1 class and the latter in INSIGHT, both of which are contained in the preprocessing.instance_selection package.

Apart from the two selection methods that are directly based on hubness, Hub Miner also implements a general hubness-aware instance selection pipeline that extends the functionality of all implemented instance selection methods, whether they were conceived as hubness-aware or not.

Namely, past experiments have shown that combining certain hubness-aware classification methods with certain instance selection methods can be quite promising, but that the instance selection bias of the selection methods negatively affects the class-conditional neighbor occurrence models learned on the reduced data. Therefore, in order to improve the performance of hubness-aware classifiers and train unbiased models, an in-between step was inserted that calculates the class-conditional selected prototype occurrence frequencies on all training data. This is shown in Figure 7.5.

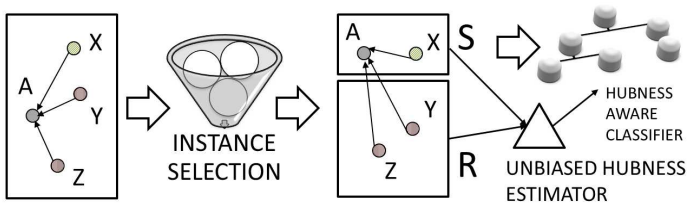


Figure 7.5: The modified instance selection pipeline. An unbiased prototype occurrence profile estimator is included between the instance selector and a huness-aware classifier. It ought to provide more reliable hubness estimates to the hubness-aware occurrence models. In the example we see that point A is a neighbor to three other points (X,Y,Z), but only one of them gets selected. Hence, some occurrence information is irretrievably lost.

The unbiased hubness estimation pipeline has been shown to yield much better results and this approach is applicable regardless of the underlying instance selection strategy.

7.5 OUTLIER DETECTION

A method called anti-hub has recently been proposed for outlier detection [RNI14] and is hubness-based. It is implemented in Hub Miner in the learning.unsupervised.outliers package, AntiHub class. Like with hubness-based clustering, it revolves around the observation that local cluster centrality tends to become highly correlated with neighbor occurrence frequency in intrinsically high-dimensional data. Therefore, non-central points will have a very low neighbor occurrence frequency, on average. These anti-hubs can be selected as outliers and it has been shown that this is a very promising idea. In order to achieve better contrast and a higher discriminative power, a somewhat larger neighborhood size is required in intrinsically high-dimensional data, though smaller neighborhood sizes can be used in lower-dimensional data.

Eight

Code Examples: Using Hub Miner for Data Analysis

Hub Miner is very easy to use for various analytic tasks. This is simplest to demonstrate in case of classification. Assume we have some data and we would like to see if we can perform learning and classification by some approach on this data. An illustrative examples is given in the class learning.supervised.example.ClassifierUsageExample. The code is copied below (without the copyright statement). An explanation follows immediately afterwards.

```
package learning.supervised.example;

import data.representation.DataSet;
import distances.primary.CombinedMetric;
import ioformat.SupervisedLoader;
import java.io.File;
import learning.supervised.evaluation.ClassificationEstimator;
import learning.supervised.methods.knn.AKNN;
import util.CommandLineParser;

/**
 * This class gives a usage example for classification - how to load the data,
 * train a classification model and save the results to a file. As an example,
 * the adaptive k-nearest neighbor classifier (AKNN) is used.
 *
 * @author Nenad Tomasev <nenad.tomasev at gmail.com>
 */
public class ClassifierUsageExample {

    public static void main(String[] args) throws Exception {
        // Specify the command line parameters. While it is possible to write
        // custom command line parsing methods for each class, the utility
        // CommandLineParser class makes it easy in HubMiner.
        CommandLineParser clp = new CommandLineParser(true);
        clp.addParam("-inFileTrain", "Path to the input training data file.",
                CommandLineParser.STRING, true, false);
        clp.addParam("-inFileTest", "Path to the input test data file.",
                CommandLineParser.STRING, true, false);
        clp.addParam("-outFile", "Path to the output file.",
                CommandLineParser.STRING, true, false);
        // The parser parses the command line to extract the parameter values.
        clp.parseLine(args);
        // We assign the in/out train and test file path values to the
        // respective variables.
        File inFileTrain =
```

```
                new File((String) clp.getParamValues("-inFileTrain").get(0));
        File inFileTest =
                new File((String) clp.getParamValues("-inFileTest").get(0));
        File outFile = new File((String) clp.getParamValues("-outFile").get(0));
        // Data load is simple when done via the SupervisedLoader class. It
        // can handle .arff, .csv and .tsv files. It can also load data from the
        // sparse modifications of the .arff format that are used in HubMiner.
        // It detects and loads the proper format automatically.
        DataSet datasetTrain = SupervisedLoader.loadData(inFileTrain, false);
        DataSet datasetTest = SupervisedLoader.loadData(inFileTest, false);
        // We use a default metric here, the Euclidean distance.
        CombinedMetric cmet = CombinedMetric.EUCLIDEAN;
        // We choose a desired neighborhood size.
        int k = 5;
        // Initialization of the classifier.
        AKNN classifier = new AKNN(datasetTrain, cmet, k);
        // Model training.
        classifier.train();
        // An aggregate ClassificationEstimator object is generated when the
        // predictions are compared on the test set.
        ClassificationEstimator estimator = classifier.test(datasetTest);
        // The estimator values are output to a file.
        estimator.printEstimatorToFile(outFile);
    }
}
```

The script takes the paths to the input training data, the input test data and the output file target as its command line parameters. These parameters are conveniently parsed by the util.CommandLineParser class. Users can use this class to specify the expected parameter and parameter types, which is especially useful for type checking. Parameters are allowed to accept multiple values.

Once the command line parameters have been parsed, we load the train and test data into corresponding DataSet objects by a single call of ioformat.SupervisedLoader.

Since we are using a k-nearest neighbor approach, we need to select a metric and here we take Euclidean distance as default, as a constant predefined object within distances.primary.CombinedMetric class. This object will handle both integer and float features properly. If we wanted to ignore integer features and calculate distances only from the floats, we would have used CombinedMetric.FLOAT_EUCLIDEAN instead.

The adaptive k-nearest neighbor algorithm [WNC07] is initialized by passing in the training data, the desired neighborhood size and the metric object.

Training is performed by a single call to the classifier, classifier.train(). We could also have inserted a pre-calculated distance matrix or the k-nearest neighbor sets, if we had them ready from another context. If they are not provided to the classifier, it calculates them implicitly. Take note that if we calculate them externally, we can set them to multiple classifiers at no additional cost, since they do not modify these structures. This achieves a considerable speed-up and is the way in which the experimental framework is currently implemented.

Classifier testing and evaluation is performed compactly in a single line: ClassificationEstimator estimator = classifier.test(datasetTest). The ClassificationEstimator object contains all the necessary performance measures, as well as methods for saving and loading this data. This is exactly how we reach the desired output, by calling estimator.printEstimatorToFile(outFile).

It is also possible to save and load learned models via serialization and it is very simple.

```
// Save a model.
classifier.save(ourFile);
// Load a model.
Classifier loadedModel = Classifier.load(inFile);
```

Of course, you would need to cast the loaded model into its proper type if you need to do something other than basic prediction with it, but that's not an issue.

Similarly, assume we wanted to perform clustering and that we have already parsed the parameters, similar to the previous example. We can cluster the data and output the results as shown below.

```
CombinedMetric cmet = CombinedMetric.EUCLIDEAN;
DataSet dset = SupervisedLoader.loadData(inFile, false);
int numClusters = dset.countCategories();
ClusteringAlg = new FastKMeans();
clust.setCombinedMetric(cmet);
clust.setDataSet(dset);
clust.setNumClusters(numClusters);
clust.cluster();
Cluster[] clusters = clust.getClusters();
Cluster.writeConfigurationToFile(outFile, clusters, dset);
```

Of course, the shown code snippet could be presented even more compactly, since the data, the metric and the number of clusters can be passed in the constructor of FastKMeans. Again we see that there are existing methods for writing output to a file, which saves us the time of having to write some I/O code for each use case.

This example can easily be extended to include some evaluation of the produced data clustering. Let us use the well-known Silhouette index to quantify the quality of the produced clustering. We would continue by doing the following:

```
QIndexSilhouette silIndex = new QIndexSilhouette(
                        numClusters, clust.getClusterAssociations(),
                        dset);
float clusteringQuality = silIndex.validity();
```

If we had an externally calculated distance matrix, we could set it to the quality index object. This way, it is calculated implicitly. Either way, we obtain an estimate of our clustering quality in two lines of code.

Outlier detection is often used prior to clustering or independently in order to perform anomaly detection or detect data pre-processing errors. Hub Miner implements several outlier detection approaches and using them for analysis is quite trivial. Here is an example of outlier detection with the local distance-based outlier factor. Similar simple code can be used for other implemented approaches. While some method automatically determine the number of outlier points in the data, some methods have user-specified thresholds or user-specified proportions of data to be marked as outliers. This can be viewed either as flexible (since it gives more control to the users) or bothersome (since users have to think about the threshold or percentages to set). In any case, both types of approaches are implemented in Hub Miner, so users can choose the approach that suits them (and the data) best.

```
CombinedMetric cmet = CombinedMetric.EUCLIDEAN;
DataSet dset = SupervisedLoader.loadData(inFile, false);
// Let's say we want to mark 5% of data points as outliers.
float outlierRatio = 0.05f;
// Local distance-based outlier factor.
LDOF detector = new LDOF(dset, 10, outlierRatio);
detector.detectOutliers();
ArrayList<Integer> outlierIndexes = detector.getOutlierIndexes();
SOPLUtil.printArrayList(outlierIndexes);
```

Let us suppose that we are not that interested in prediction. How do we analyze data? In principle, BatchHubnessAnalyzer is a useful tool for batch analysis across various datasets. However, a user might want to do some custom analysis. Let us set up a couple of examples. First of all, what if a user simply want to see what the major hubs in the data are.

```
CombinedMetric cmet = CombinedMetric.EUCLIDEAN;
DataSet dset = SupervisedLoader.loadData(inFile, false);
float[][] dMat = dset.calculateDistMatrix(cmet);
HubFinder hFinder = new HubFinder(dset, dMat, cmet);
// We will look at 1-neighbor sets, so only nearest neighbors.
int k = 1;
ArrayList<Integer> hubIndexes = hFinder.findHubsForK(k);
SOPLUtil.printArrayList(hubIndexes);
```

There is already a class responsible for extracting hubs. HubFinder implicitly calculates the kNN sets via NeighborSetFinder, observes the neighbor occurrence frequencies, calculates the mean and the standard deviation and outputs those hub points that have occurrence frequencies that exceed mean by more than two standard deviations. SOPLUtil is a useful utility class for quickly printing out some stuff to the command line and/or Writer objects.

Metric learning is often used for improving the performance of similarity-based methods and Hub Miner implements support for secondary distances that have been demonstrated as useful in high-dimensional data. Learning secondary distances in Hub Miner is fairly simple. Let us look at an example showing how to calculate $simhub_s$ for $s = 50$ and $k = 10$ on all training data and initialize a metric object for use in calculating future distances according to the query results against the training set.

```
// Initialize the primary metric object.
CombinedMetric cmet = CombinedMetric.EUCLIDEAN;
// Load the data from a specified input file (assume this as given).
DataSet dset = SupervisedLoader.loadData(inFile, false);
// Calculate the primary distance matrix.
float[][] dMat = dset.calculateDistMatrix(cmet);
int numClasses = dset.countCategories();
// Specify the secondary neighborhood size for calculating shared neighbors.
int secondaryK = 50;
// Initialize the kNN finder object.
NeighborSetFinder nsfSecK = new NeighborSetFinder(dset, dMat, cmet);
// Calculate the kNN sets for k = 50.
nsfSecK.calculateNeighborSets(secondaryK);
// Specify the target neighborhood size to be used in classification.
int kValue = 10;
// Initialize the object that does the shared neighbor calculations.
SharedNeighborFinder snf = new SharedNeighborFinder(nsfSecK, kValue);
```

```
snf.setNumClasses(numClasses);
// Specify that hubness information weights are to be used, defining simhub and not simcos.
snf.obtainWeightsFromHubnessInformation(0);
// Calculate the shared neighbor sets.
snf.countSharedNeighborsMultiThread(numCommonThreads);
// First fetch the similarities.
float[][] dMatSec = snf.getSharedNeighborCounts();
// Then transform them into distances.
for (int indexFirst = 0; indexFirst < dMatSec.length; indexFirst++) {
    for (int indexSecond = 0; indexSecond < dMatSec[indexFirst].length; indexSecond++) {
        dMatSec[indexFirst][indexSecond] = secondaryK - dMatSec[indexFirst][indexSecond];
    }
}
// Initialize the secondary metric object for later use.
SharedNeighborCalculator snc = new SharedNeighborCalculator(
        snf, SharedNeighborCalculator.WeightingType.HUBNESS_INFORMATION);
```

Calculating other secondary distance types is even easier, as demonstrated on an example involving mutual proximity.

```
// Initialize the primary metric object.
CombinedMetric cmet = CombinedMetric.EUCLIDEAN;
// Load the data from a specified input file (assume this as given).
DataSet dset = SupervisedLoader.loadData(inFile, false);
// Calculate the primary distance matrix.
float[][] dMat = dset.calculateDistMatrix(cmet);
int numClasses = dset.countCategories();
// Specify the secondary neighborhood size for calculating shared neighbors.
int secondaryK = 100;
// Initialize the kNN finder object.
NeighborSetFinder nsfSecK = new NeighborSetFinder(dset, dMat, cmet);
// Calculate the kNN sets for k = 100.
nsfSecK.calculateNeighborSets(secondaryK);
// Initialize the mutual proximity calculator.
MutualProximityCalculator calc = new MutualProximityCalculator(
        nsfSecK.getDistances(), nsfSecK.getDataSet(), nsfSecK.getCombinedMetric());
// Calculate the secondary distance matrix.
dMatSec = calc.calculateSecondaryDistMatrixMultThr(secondaryK, 8);
```

Ok, let's look at something a bit more mundane. We will use Hub Miner to calculate the information value of different features in the data, used for calculating information gain.

```
DataSet dset = SupervisedLoader.loadData(inFile, false);
// We will look at an integer attribute.
int featureType = DataMineConstants.INTEGER;
// We will look at the first float feature;
int featureIndex = 0;
int numCategories = dset.countCategories();
DiscretizedDataSet discDSet = new DiscretizedDataSet(dset);
EntropyMDLDiscretizer discretizer =
        new EntropyMDLDiscretizer(
        dset, discDSet, numCategories);
discretizer.discretizeAll();
discDSet.discretizeDataSet(dset);
DiscreteAttributeValueSplitter splitter = new DiscreteAttributeValueSplitter(discDSet);
DiscreteAttributeEvaluator evaluator = new Info(splitter, numCategories);
float informationValue = evaluator.evaluate(featureType, featureIndex);
```

This kind of analysis is done in the decision tree implementation in Hub Miner.

We have seen that it is possible to do analysis in Hub Miner in a few lines of code. However, as we will see, in order to perform useful data analysis with Hub Miner, it is not necessary to write any lines of code, there are many useful tools and scripts and existing frameworks. One such tool that we will have a closer look at in the following chapter is Image Hub Explorer.

Nine

Image Hub Explorer

When handling image data, there is a wide choice of possible feature representations and processing pipelines, as well as a wide choice of metrics that can be used to measure image similarity and run queries on the database. This is partly due to the semantic gap and the fact that it is not that easy to choose the optimal representation, within a given context.

Images are very high-dimensional in nature, as many features are required in order to properly encode all the objects in the scene and their properties. A usual approach consists of extracting a set of local image features from each image, creating a codebook vocabulary, and then generating histogram representations to describe each individual image, as bag-of-visual-words, similar to text. In practice, large vocabularies are used, so there are potentially hundreds of features in the representation.

In presence of captions, tags and comments, it is not that unusual to form an associated textual image meta-description as well and possibly concatenate it with the dense image feature part of the representation.

It has been experimentally shown that quantized image representations are highly susceptible to hubness under commonly used similarity measures and normalization approaches [TBMN11][PTR$^+$11]. Therefore, hubness is expected to greatly impact image-based and description-based image querying, as well as certain types of object recognition from images.

In order to help users with visualizing hubness in their image databases in order to choose the most appropriate representation and similarity measure, Image Hub Explorer was built and it is included in the gui.images package in Hub Miner. A typical Image Hub Explorer use case is shown in Figure 9.1.

A **demo video** of how Image Hub Explorer is used in practice is available at `http://youtu.be/LB9ZWuvm0qw`.

Image Hub Explorer has been designed to help with analyzing close-to-scale-free distributions of image relevance in k-nearest neighbor graphs of large processed image datasets. Image Hub Explorer can also be applied to other data types, with most of its original functionality. This is the first publicly available tool for hubness visualization and exploration. An overview of the essential types of functions that Image Hub Explorer provides is shown in Figure 9.2.

Image Hub Explorer enables the users to experiment with several state-of-the-art hubness-aware metric learning techniques [ZmP04][JHS07][SFSW12][HKK$^+$10][TM12a][TM14a], hubness-aware classification methods [RNI09][TRMI13a][TRMI11a][TM12b], standard kNN baselines [FH51][KGG85][WNC07][Tan05] and a recently proposed query result re-ranking procedure [TLM13].

The users need to provide the images for visualization and their feature representation or a precomputed distance matrix. The system then calculates the k-nearest neighbor sets in the specified metric for a range of k-values and calculates the most important hubness-related stats. It also per-

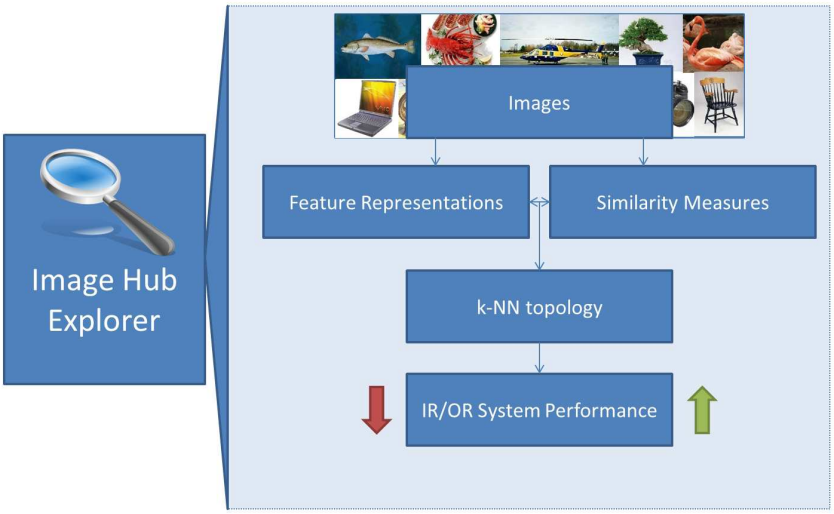


Figure 9.1: The typical Image Hub Explorer use case.

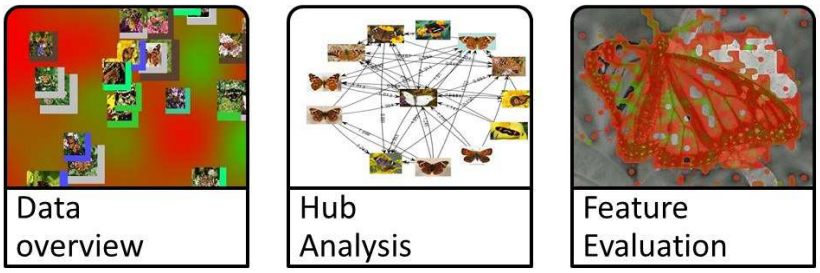


Figure 9.2: An overview of several basic Image Hub Explorer functions.

forms class-specific hubness analysis and estimates which classes are the most critical sources of detrimental influence and which classes suffer most semantic inconsistency. Users are able to focus on certain parts of the kNN graph and explore the local similarity structure. In order to be able to interpret why some images act as hubs and which parts of the image are responsible for this influence being beneficial or detrimental, feature visualization and analysis is also available. We will go through all these functions individually and also discuss the file structure and how it should be set up in order to use Image Hub Explorer properly.

While most of Image Hub Explorer's functionality is based on Hub Miner, there are also some external dependencies. Multi-dimensional scaling for data overview is performed by the MDSJ library developed at the University of Konstanz [Pic09]. The JUNG library (`http://jung.sourceforge.net/`) is used for graph drawing. Charts that are used to illustrate certain data properties are displayed via JFreeChart (`http://www.jfree.org/jfreechart/`).

There are several types of data views in Image Hub Explorer and they all hold the references to the same underlying set of data structures. These views are updated automatically when some of

the shared objects are modified.

The shared objects include the currently selected image, browsing history, the primary and secondary distance matrices, feature representations (if available), the list of kNN graphs over a range of different neighborhood sizes, as well as k-dependent lists of hubness-related statistics and charts. The images for display are loaded in batches from the disk.

Custom JPanel classes are used to interactively display the image content. The Image Hub Explorer GUI does not in itself contain any explicit data mining code. All modeling is performed by invoking the appropriate classes and methods in the underlying Hub Miner library.

The following examples were computed on images taken from the Leeds Butterfly dataset [WME09] (`http://www.comp.leeds.ac.uk/scs6jwks/dataset/leedsbutterfly/`).

9.1 PREPARING THE DATA FOR VISUALIZATION

Each dataset that is assigned a separate workspace and the user gets to select the workspace directory from the drop-down menu. This is what the workspace directory structure needs to look like:

```
codebook
distancesNNSets
photos
representation
thumbnails
tmp

classNames.txt
```

The classNames.txt file should contain a comma-separated list of class names in a single line, like for instance:

```
Danaus plexippus,Heliconius charitonius,Heliconius erato, Junonia coenia,
Junonia phlaeas, Nymphalis antiopa, Papilio cresphontes, Pieris rapae,
Vanessa atalanta, Vanessa cardui
```

The codebook directory stores the codebook used to generate the quantized image representation that is being analyzed. This is used for feature assessment and is not necessary otherwise. Two files are to be stored in the directory, codebook.txt and codebookProfile.txt. Both of these files are loaded manually by the user from the drop-down menu in the UI. The file formats correspond to Hub Miner codebook and codebook profile file formats and are quite simple. The codebook file has a single header line like: "codebook_size:400", indicating the size of the vocabulary, in this case 400. What follows afterwards is (in this case) 400 lines, each line corresponding to a single codebook vector, comma-separated. As for the codebook profile file, on the first header line it has a single number, also 400 in this case. This is followed again by 400 lines, each corresponding to the class-conditional occurrence profile of the respective codebook vector (so the number of items in the line equals the number of classes in the data), comma-separated.

The distancesNNSets directory contains a sub-directory for each metric that the users experiment with. It can be empty in the beginning, as Image Hub Explorer can automatically calculate the distance matrix and the kNN sets. If the users have them pre-computed, they can also be loaded if placed in an appropriate place in the directory structure. For instance, the directory distancesNNSets\distances.primary.CosineMetric contains the distance matrix for the cosine distance

and the associated kNN sets. The two files follow the standard distance matrix and kNN set file format in Hub Miner. Interested users can easily discern the specifics from the corresponding I/O classes. However, unless the users want to load their own distances and neighbor sets, this is not necessary. It is system-internal.

The photos directory contains the actual full-size photos, in the appropriate class directories. The same goes for the thumbnails directory. In case of the Leeds butterfly dataset, this is the class directory structure:

```
DanausPlexippus
HeliconiusCharitonius
HeliconiusErato
JunoniaCoenia
JunoniaPhlaeas
NymphalisAntiopa
PapilioCresphontes
PierisRapae
VanessaAtalanta
VanessaCardui
```

Make sure that the directory names correspond to the class names provided in the classNames.txt file.

The representation directory has two sub-directories, as follows:

```
raw_representation
quantized
```

The raw_representation directory contains the raw features extracted from the images. The directory structure is the same as with photos or thumbnails directories, image feature files are contained in the corresponding class directories. The system supports both OpenCV and SiftWin feature file formats. In case of SiftWin, it is one file per image. In case of OpenCV, one keypoint *.kp file and one descriptor *.desc file. Based on the extension, the system invokes the appropriate load mechanism.

The quantized directory contains the actual quantized image representation to use in Image Hub Explorer exploratory analysis and visualization. A single ARFF file is expected here and it will be loaded regardless of its name.

9.2 Visualization and Interactive Analysis

Image Hub Explorer has four main screens: Data Overview, Class View, Neighbor View and Search. The Feature Assessment panel can be invoked for individual images through the menus above.

9.2.1 Data Overview Screen

The Data Overview screen gives a high-level overview of the data and its main properties under the current feature representation and metric.

The Projection Panel shows a 2D visualization of the image data and allows the users to browse through the central data points. The projection is currently achieved by multi-dimensional scaling (MDS) [BG05]. An example can be seen in Figure 9.3.

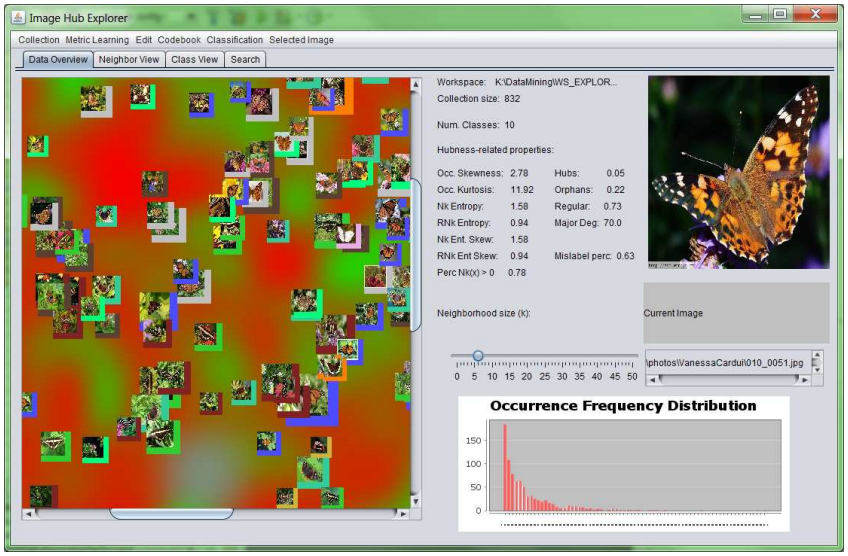


Figure 9.3: The Data Overview screen of Image Hub Explorer: Visualizing the major image hubs via multi-dimensional scaling.

After calculating the neighbor occurrence frequencies for all images, a fixed number of hub images is displayed in the Projection Panel. Only the most influential images are shown, those that have the potentially highest impact on system performance.

The background landscape is calculated based on the average good and bad hubness of different regions in the projected feature space. The green color corresponds to good hubness and the red one to bad hubness. The landscape is generated in two steps. The first step is a sort of a Gaussian blur as implemented in [FGM05] and the second step is a two-pass box blur. For more details on how the landscape is calculated, see [TM14b].

One such landscape is generated for each neighborhood size k, as it depends on good and bad hubness that are k-dependent quantities. The users can use the *slider-selector* for neighborhood size to quickly change among different k-values and observe the differences in all quantities and all tabular views of the application.

All images are shown within the frames that are colored according to their class. This makes distinguishing between different classes easier for small displayed thumbnails in various screens. All images are selectable by a simple mouse click.

9.2.2 Class View

The Class View (Figure 9.4) enables the users to inspect different classes separately. A comparison of class-specific point type distributions [NS12] often reveals why some classes are more susceptible to misclassification in the current metric and feature representations. Lists of major good and bad hubs are also computed and shown to the user.

Some pairs of classes are more difficult to distinguish than others and this can be observed in the class-to-class k-neighbor occurrence matrix, which is shown on the right side of the Class View. The cells in the table are colored according to the type and intensity of the pairwise interaction. Red cells mark the principal gradients of misclassification.

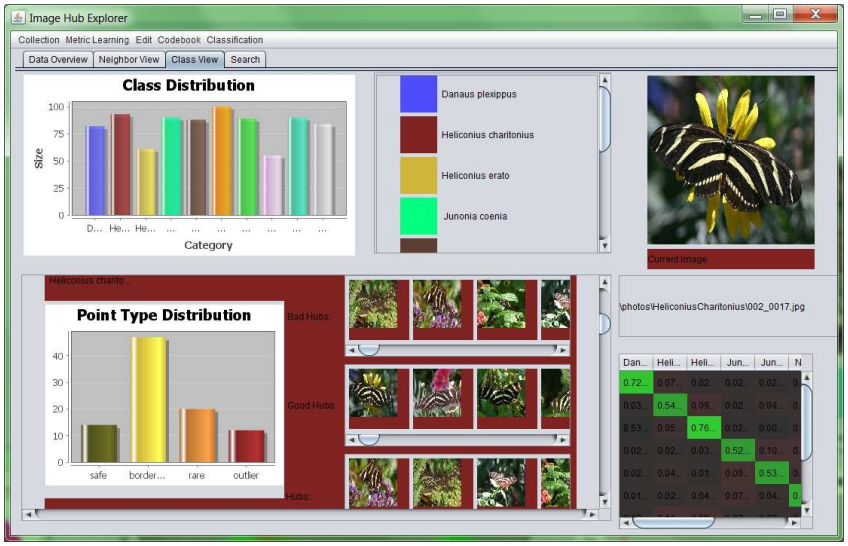


Figure 9.4: The Class View of Image Hub Explorer: Examining point type distributions and centers of influence for each class separately.

9.2.3 Neighbor View

User can quickly pinpoint the critical subsets of hub points in the Neighbor View. An example is shown in Figure 9.5.

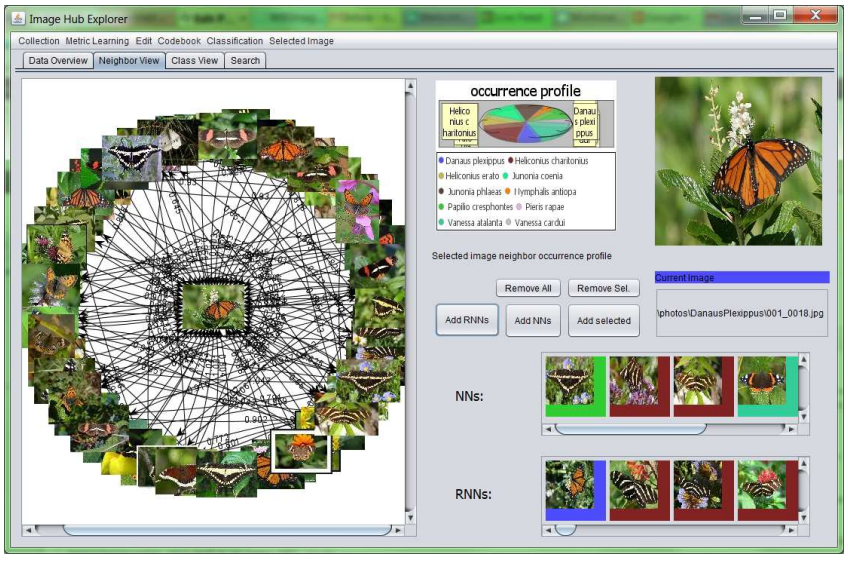
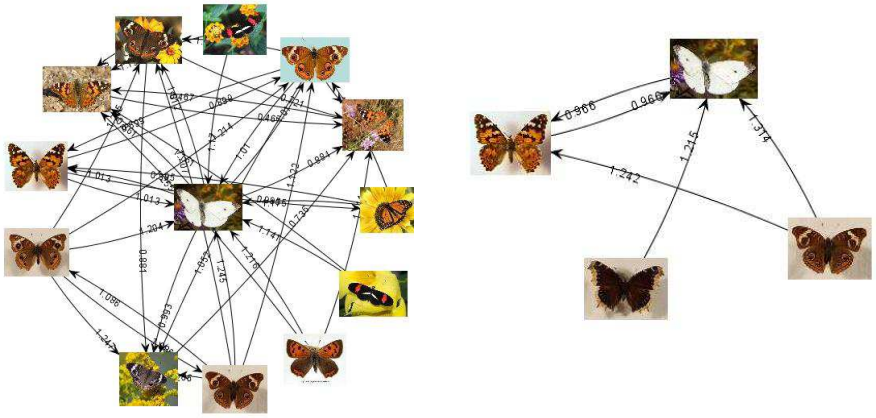


Figure 9.5: The Neighbor View of Image Hub Explorer: Exploring the nearest neighbor (NN) and reverse nearest neighbor (RNN) lists and visualizing local kNN subgraphs.

Any selected image can be inserted into the local visualized subgraph of the kNN graph of the data. The visualization is automatically updated in case of changes in neighborhood size selection. It is possible to batch-insert all the neighbors or reverse neighbors of the selected image. The weights on the edges correspond to the distance between the selected points in the selected metric.

In order to be able to decide whether to include the current selection in the view, its neighbor occurrence profile is shown in the upper right corner, as well as the lists of direct and reverse k-nearest neighbors.

The Neighbor View helps in visualizing the influence of hub points, as shown in Figure 9.6, where one bad hub image is shown, along with a set of its reverse k-nearest neighbors. In this case the *Artogeia rapae* image that is shown in the middle acts as a neighbor only to points that are not from its own class (species), which is obviously detrimental to kNN-based analysis. The comparison between two different feature representations reveals that the influence of images changes drastically when the underlying feature representation changes. This shows that the induced pseudo-relevance of images does not correspond well to their actual relevance in the considered semantic context.



(a) SIFT (b) SURF

Figure 9.6: An example of a bad hub in the quantized SIFT feature representation, a detrimental center of influence. Neither of the reverse neighbors of the selected image belongs to the same class as the image itself, so its occurrences induce label mismatches and are semantically inconsistent. The same image has an equally inconsistent occurrence profile in the quantized SURF feature representation, but it is not a hub there, as it does not occur very often. On the other hand, the displayed image never occurs as a neighbor in the quantized BRIEF feature representation, for the same neighborhood size of $k = 5$.

9.2.4 Feature Visualization and Assessment Panel

Not all features are equally informative and it is possible to use Image Hub Explorer for feature assessment in quantized feature representations. Within Image Hub Explorer, users can inspect individual visual words and their class-conditional occurrence profiles, that are displayed in form of pie charts. More importantly, Image Hub Explorer offers a possibility to visualize the distribution of informativeness on each image individually. For details on how this is actually calculated, see the original paper [TM14b].

Figure 9.7 shows an example of feature informativeness visualization in an image. The green color in the informativeness landscape is used to denote regions with high discriminative information content and the red one for the regions that do not contribute to object recognition.

(a) A regularly displayed selected image.

(b) An overall visualization of the critical feature regions.

(c) A visualization of a single visual word, one that is most beneficial for object recognition of this image type.

Figure 9.7: Individual visual words are displayed on top of the selected image and colored according to their overall usefulness and semantic consistency. This helps in identifying the critical regions in the images, those that contribute to making good class distinctions and those that represent textural patterns that might occur in many different image classes.

Figure 9.7 shows how the feature assessment and visualization components works for SIFT features in case of recognizing *Danaus plexippus* butterfly specimens. The textural regions around the black veins on the butterfly's wings are judged to be the most informative by the system. This is indeed a highly distinctive feature of the particular species. Similarly, for *Heliconius charitonius* the system determines that the white stripes on otherwise black butterfly's wings carry highly

discriminative visual information.

9.2.5 Search and Ranking

It is possible to use Image Hub Explorer for querying the image database. This is currently set up to work with SIFT features. SIFT features are extracted for the query image and a histogram representation is formed. An overview of the search interface is shown in Figure 9.8.

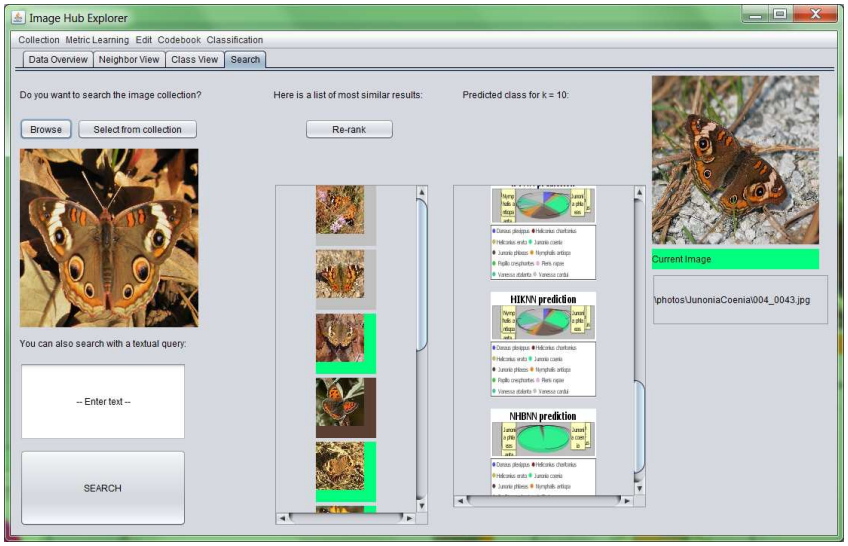


Figure 9.8: The Search screen of Image Hub Explorer. Apart from supporting the basic query functionality, the system offers label suggestions based on the output of several kNN classification models, as well as a hubness-aware secondary re-ranking procedure.

In order to use this function, the codebook needs to be loaded and SiftWin needs to be in the system path, as well as ImageMagick, for JPG to PGM conversion prior to SIFT extraction.

The search panel also makes an attempt to predict the label of the image query, based on several k-nearest neighbor models trained on the loaded images: kNN [FH51], FNN [KGG85], NWKNN [Tan05], AKNN [WNC07], hw-kNN [RNI09], h-FNN [TRMI13a], HIKNN [TM12b] and NHBNN [TRMI11a]. This allows the users to compare how different classification approaches handle certain types of points, in order to select the most appropriate approach for future deployment in the IR/OR system.

It is also possible to re-rank the images based on a recently proposed hubness-aware self-adaptive secondary re-ranking method [TLM13]. This procedure can improve the semantic consistency of the results and move the images from the same class closer to the query.

Ten

Overview of Hub Miner Packages

10.0.6 configuration

This package contains the classes that represent the configuration files for Hub Miner's experimentation framework for classification, clustering and exploratory hubness-related statistical data analysis. The configuration classes are BatchClassifierConfig, BatchClusteringConfig and BatchHubnessAnalysisConfig.

The experimental configuration classes contains I/O methods for parsing configuration files, as well as serializing/deserializing the configuration information to/from JSON.

10.0.7 data.generators

The data.generators package contains the logic for automatically generating synthetic datasets for experimentation. All generators are to implement the DataGenerator interface that contains the methods for generating an array of float or integer values based on some underlying protocol.

It is possible to combine multiple generators for generating a single dataset, as enabled by MixtureOfFloatGenerators class.

The data.generators.util subpackage contains several concrete implementations that were used for generating some data in our past experiments.

Apart from the generators, there is also the BasicGaussianDatasetExtender class that builds a Gaussian model for each category in the data and extends existing data by generating synthetic data instances.

10.0.8 data.imbalance

The initial implementation of the data.imbalance package contains a script for analyzing class-imbalanced data and also evaluating the performance of some implemented hubness-aware techniques on such data. Hubness-aware metric and hubness-aware classification algorithms are compared, in several regards. A detailed comparison of the point type (safe, borderline, rare, outlier) distribution is given for each employed metric and algorithm precision.

Hubness-aware methods work very well on class-imbalanced high-dimensional datasets [TM13a]. More types of analysis of the class imbalance problem from the perspective of high-dimensional data classification are therefore going to be introduced in future Hub Miner releases.

10.0.9 data.neighbors

This package deals with the extraction of k-nearest neighbor sets and their analysis in high-dimensional data. It also has two sub-packages, data.neighbors.approximate and data.neighbors.hubness.

NeighborSetFinder is one of the most used classes in Hub Miner. It contains a simple implementation of exact k-nearest neighbor set calculations and can be extended to indexed or approximate implementations. No index is used by default, since it is difficult to set up universally good NN-search indexes in very high-dimensional data and also different index structures might be preferable in lower-dimensional datasets or different data domains. NeighborSetFinder class therefore offers the 'vanilla' implementation for k-nearest neighbor set extraction and should be extended for more efficient extractions in cases when a better approach is known.

Apart from extracting the k-nearest neighbor sets, NeighborSetFinder objects store them and store some basic statistics. There are methods for inferring the class-conditional occurrence probabilities for the neighbor occurrence models, as well as methods for generating hubness-based weighting that is used in multiple places in the library. As for kNN search itself, NeighborSetFinder can also perform tabu-search, where some instances are not considered as neighbors.

NeighborSetFinder objects implement the logic for varying the neighborhood size and taking sub-kNN sets, calculating their stats and producing new objects to represent the restricted information.

NeighborSetFinder object is the main unit for sharing kNN information between algorithms in Hub Miner.

SyntheticKNNExtender is a class that can be used to extend the data with some synthetic examples in order to better estimate the neighbor occurrence frequencies of the given data points.

NeighborSetUserInterface declares methods for setting and getting NeighborSetFinder objects and is used to set the kNN information to all algorithms that require it during classification, clustering or instance selection.

The data.neighbor.approximate.AppKNNGraphLanczosBisection class implements approximate kNN set calculations based on recursive Lanczos bisections and has been used for testing the robustness of hubness-aware approaches to approximate kNN sets.

10.0.10 data.neighbors.hubness

This is the package with exploratory methods for establishing the level of hubness in high-dimensional data and uncovering important properties of the k-nearest neighbor occurrence frequency distribution and the k-nearest neighbor graph in general.

BatchHubnessAnalyzer is a class that enables batch-analysis of hubness across many datasets over a range of neighborhood sizes, for the specified metrics. It operates based on the provided configuration file. More details on this have already been given in the previous chapters. There is also the MultiLabelBatchHubnessAnalyzer, for datasets with multiple classification tasks defined on them, so that all the distances and neighbor sets are only calculated once for each data representation and then different label assignments are considered in turn.

BucketedOccDistributionGetter can be used to obtain a histogram of the neighbor occurrence frequency distribution.

HubFinder can be used to quickly output a list of hubs in the data, based on the calculated k-nearest neighbor sets, for the desired neighborhood size.

HubOrphanRegularPercentagesCalculator calculates the percentages of hubs, anti-hubs and orphans, as well as regular points, among the training data. The higher the hubness in the data, the lower the percentage of regular points.

HubnessAboveThresholdExplorer is similarly used to analyze the percentages of points above or below some pre-defined neighbor occurrence frequency threshold. This is especially useful for estimating the influence of anti-hub handling strategies in some hubness-aware implementations that contain a special anti-hub handling case for a specified threshold value.

HubnessExtremesGrabber is similar to HubFinder. It calculates and returns a pre-defined number of most frequent neighbors, over a range of neighborhood sizes. The difference is that this list does not necessarily contain all hubs in the data, but it could contain something like top-5 hubs. Also, if the specified number of points to return is large enough, some non-hub points might be contained as well.

HubnessSkewAndKurtosisExplorer calculates the third and fourth standard moment of the neighbor occurrence frequency distribution (skew and kurtosis) in a batch way, over a range of neighborhood sizes. Similarly, HubnessVarianceExplorer calculates the variance of the occurrence frequency distribution over a range of specified neighborhood sizes.

KNeighborEntropyExplorer calculates the average entropy of k-nearest neighbor sets and the average entropy of the reverse k-nearest neighbor sets. This helps with estimating the semantic consistency of the direct and reverse k-nearest neighbor relation.

TopHubsClusterUtil implements the methods for batch-calculating the diameters and average intra-cluster distances of top hub clusters over a range of neighborhood sizes. This enables us to determine how compact the hubs in the data are, whether they are all close to each other or dispersed.

The data.neighbors.hubness.experimental sub-package implements several experiment scripts that reflect what can be done with the exploratory hubness framework. GaussianHubnessLocalizer was used to determine the correlation between point-wise hubness and local cluster centrality in intrinsically high-dimensional Gaussian data. This is further extended in the MultiGaussianLocalityExplorer. The two HubnessRiskEstimator classes were used to determine hubness risk over multiple samplings from the same underlying distribution.

The data.neighbors.hubness.util sub-package offers some more exploratory scripts, for quickly getting the neighbor occurrence frequency arrays, reverse neighbor lists or frequent neighbor pairs on output.

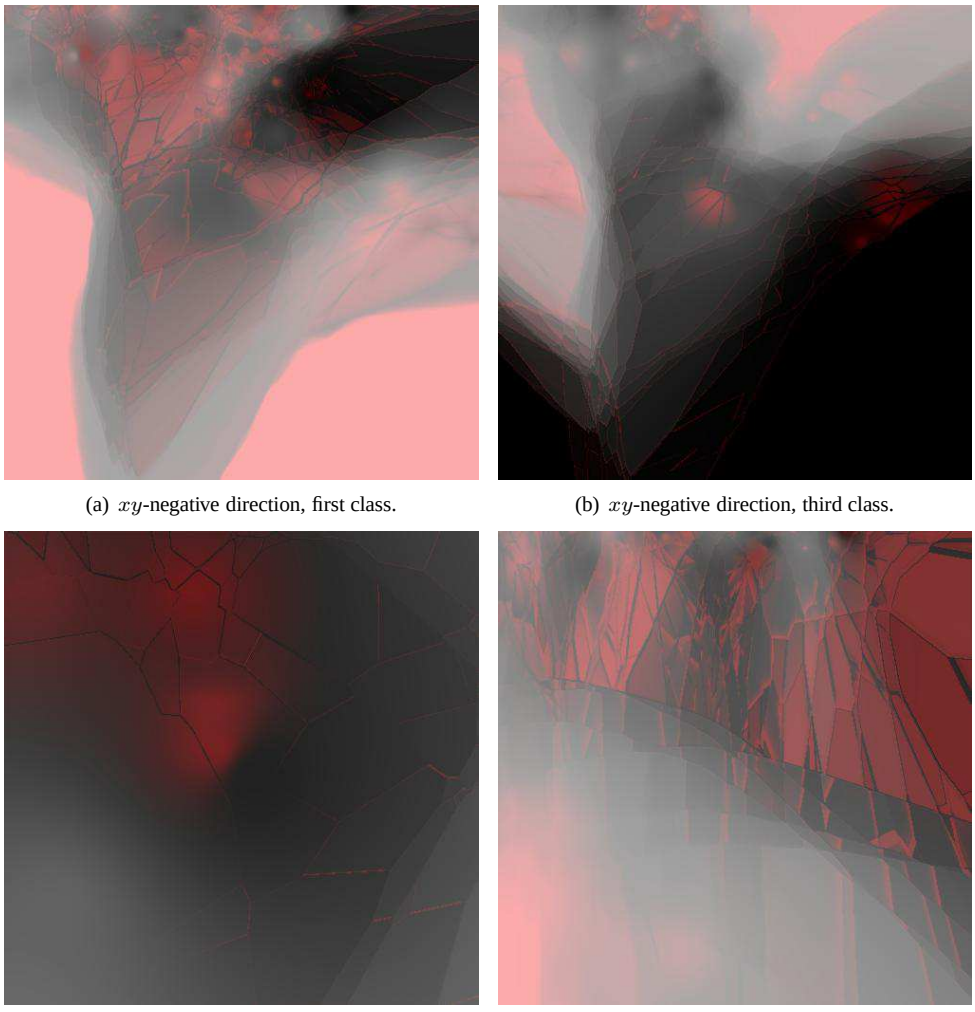
The data.neighbors.hubness.visualization package implements several default ways for visualizing hubness in synthetic and real-world data, as well as classifier performance under hubness. Some examples are shown in Figure 10.1 and Figure 10.2, though different types of visualizations are also possible and implemented in the corresponding classes.

The point of the 3D visualization in Figure 10.1, apart from it just looking awesome, is that it is very difficult to give illustrative examples in 2D, since no hubness can be observed in 2D as the maximal neighbor occurrence frequency is geometrically very constrained. Well, it's not like 3D is much better, but it might be somewhat easier to see some consequences of hubness visualized there when projecting the original spaces via MDS or PCA, since more of the original structure is preserved. Visualizing data in high-dimensional spaces is never easy, so any help is welcome. As for Figure 10.2, it shows some basic ways of visualizing the emerging hubs in the data with increasing dimensionality and the distribution and localization of neighbor occurrence frequency.

10.0.11 data.representation

This package contains all data representation classes in Hub Miner. It implements the basic support for dense, sparse and discretized data instances and datasets. More details on each of these can be found in Chapter 6.

DataSet and DataInstance classes are the basic data holders in Hub Miner and are used throughout the library. DiscretizedDataSet and DiscretizedDataInstance objects from the data.representation.discrete package are the discretized versions of the default data holders and are used for algorithms that operate on discrete values and value ranges, such as decision trees. The data.representation.discrete.transform sub-package contains the discretization methods. The data.representation.sparse sub-package contains BOWDataSet and BOWInstance classes that are



(a) xy-negative direction, first class.

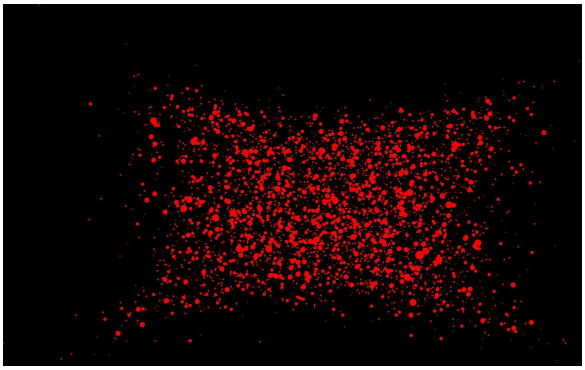
(b) xy-negative direction, third class.

(c) xy-positive direction, third class.

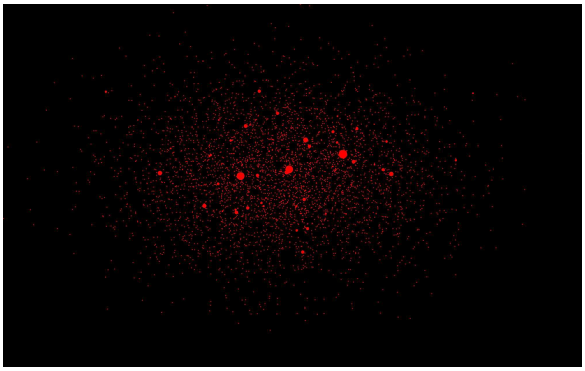
(d) zx-negative direction, first class.

Figure 10.1: Visualizing HIKNN prediction landscape in UCI Vehicle data, in 3 dimensions. For each class, two views are generated for each axis, one for each side of the cube that contains the projected data.

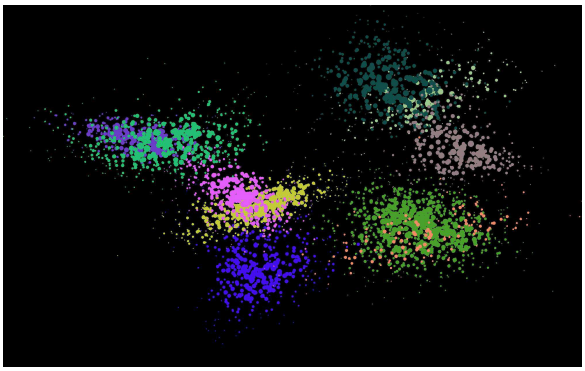
used for representing bag-of-words sparse data. The data.representation.images package holds classes used for representing quantized image data, SIFT features, as well as color histograms. While these image representations could easily by put into the default representational framework, there are some benefits to constraining the structure and ascertaining that it conforms to the defined image representation.



(a) Single cluster, $d = 3, k = 1$.



(b) Single cluster, $d = 100, k = 1$.



(c) Multiple clusters, $d = 5, k = 100$.

Figure 10.2: Basic hub visualizations where node size corresponds to the neighbor occurrence frequency. When comparing the two given single-cluster synthetic Gaussian examples, consequences of high data dimensionality become apparent, as a small number of dominant hub points emerge.

10.0.12 data.structures

This package is meant to contain all the auxiliary data structures that are used in Hub Miner implementations. Currently it holds a K-D tree implementation. It will be extended in future releases.

10.0.13 dimensionality_reduction

Dimensionality reduction is commonly used in high-dimensional data analysis and this package offers two standard approaches to dimensionality reduction - principal component analysis and random projections, which are implemented in the PrincipalComponentAnalysis and RandomProjection classes, respectively.

10.0.14 distances.primary

The most important class to note in the distance.primary package is CombinedMetric. Data in Hub Miner can have float or integer features - and this combined metric object allows us to use different metrics for integers and floats and combine them in some smart way. Initially this was implemented to also consider nominal features, but has been simplified in the meantime and it is, of course, possible to extend the class in the future, if needed for some concrete applications and projects.

The remaining classes in this package are all distance measures that can be applied to integer and float parts of the feature representation. There are many metrics in the package, standard metrics and less standard metrics. Users can find anything from Euclidean and Manhattan, Canberra, Tanimoto or Bray-Curtis to symmetrized Kullback-Leibler divergence. There are also two dummy metrics (placeholders) for some metrics that we have used in past projects, namely dynamic time-warping and Mandel-Ellis. The purpose of these placeholders is to use them when loading the appropriate distance type externally. Dummy metric objects will be initialized, but external distances will be used instead. In future releases, an implementation will probably be included for these, at least for DTW. So, stay tuned.

10.0.15 distances.secondary

Secondary distances are a metric learning approach to dealing with high hubness in intrinsically high-dimensional data and this package implements several state-of-the art secondary similarity/distance approaches. Local scaling [ZmP04], NICDM [JHS07] and mutual proximity [SFSW12] are implemented in distances.sparse and $simcos_s$ and $simhub_s$ [TM12a][TM14a] in distances.sparse.snd, though the shared neighbor sets themselves are calculated via the SharedNeighborFinder class in data.neighbors.

10.0.16 distances.sparse

This package contains the SparseCombinedMetric class that corresponds to the CombinedMetric class in distances.primary. It is an extension of that class, so it combines distances calculated on dense and sparse parts of a data representation. The package also contains implementations of standard metrics, for sparse data.

10.0.17 distances.kernel

Kernels allow for non-linear types of learning to be performed and Hub Miner offers a wide spectrum of kernel functions to use in data analysis and learning. The classes in this package include ChiSquaredKernel, ANOVAKernel, CauchyKernel, ExponentialKernel, GaussianKernel, MultiQuadraticKernel, RBF, PolynomialKernel, SigmoidKernel - and many others.

10.0.18 distances.concentration

The class ConcentrationCalculator can be used to examine distance concentration in the data. Distance concentration is related to hubness and is yet another aspect of the well known curse of dimensionality. This class implements methods for calculating distance mean, variance and relative contrast, es well estimating the intrinsic dimensionality of the data.

10.0.19 distances.analysis

MetricsAnalyzer class in this package implements two important methods for evaluating new distance measures, including the newly proposed secondary distance measures. Some of these proposed distances are actually pseudo-metrics in a sense that the triangle inequality might occasionally be breached. One of the methods in this class calculates the percentage of triangle inequality breaches. Another method calculates the Goodman-Kruskal concordance index that enables the users to estimate how concordant the distances are w.r.t. the classes in the data.

10.0.20 draw

Hub Miner contains some visualization and data exploration components and the draw package contains some of the basic building blocks used in those visualizations. This includes the BoxBlur class that is used in MDS landscape calculations in Image Hub Explorer, as well as RotatedEllipse used in SIFT feature cluster visualizations. PieRenderer class in draw.charts enables easy drawing of pie charts in Hub Miner.

10.0.21 feature

This package implements classes that enable basic feature evaluation and assessment. The feature.correlation package contains implementations of Pearson and Spearman correlation coefficients, as well as distance correlation. For discrete features, there is MutualInformation class in feature.correlation.discrete. The feature.evaluation package contains information gain (IG) and gain ratio (GR) implementations. ApplyWeights class in feature.weighting makes it easy to apply feature weights to a DataSet.

10.0.22 filters

Sometimes it is necessary to apply a transformation to a DataSet and this is what the filters package is for. It currently offers some basic filtering implementations, like TF-IDF (term frequency - inverse document frequency), shuffling and sub-sampling.

10.0.23 graph

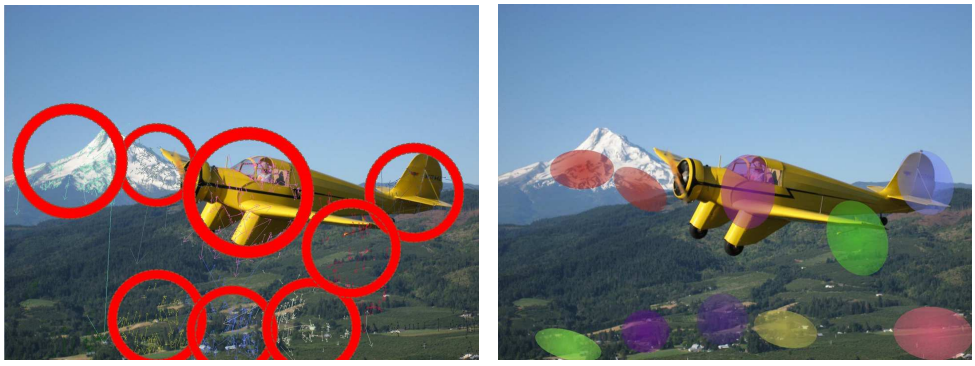
Hub Miner offers some basic support for working with graph data representations, as hubness analysis is in fact based on working with k-nearest neighbor graphs, even if mostly implicitly by considering the node degree distribution. A simple graph representation is available in graph.basic, in DMGraph, DMGraphEdge and VertexInstance classes. Some basic graph properties can be derived by applying methods from GraphGeodisic and GraphStatistics in graph.calc sub-package. Node placement for visualization can be deduced by using several methods from graph.drawing, implemented in the following classes: BarycentricCoordinateFinder, FRCoordinateFinder and RandomCoordinateFinder. PajekFormatIO class in graph.io allows the users to export DMGraph objects into Pajek data format (`http://pajek.imfm.si/doku.php?id=pajek`). Since Pajek is a well-known environment for graph/network analysis, this means that Hub Miner users will be able to delegate some of the analysis to Pajek, in case more than what is currently available in Hub Miner itself is

needed. Indeed, many types of analysis could be conducted on k-nearest neighbor graphs in high-dimensional data and are looking forward to seeing more interesting results in the future. Selecting subgraphs or calculating the connected components is possible by invoking methods implemented in classes from the graph.subgraphs package.

10.0.24 gui.images

Image Hub Explorer [TM13c][TM14b] is a great tool for exploration of hubness in high-dimensional data. Its primary purpose it to be used for analyzing different quantized image feature representations, but it can also be applied to different data types. For details, see Chapter 9. It allows for experimentations with feature representations and metrics and enables the users to study the consequences of their choices in great detail.

Apart from Image Hub Explorer, gui.images package contains two basic image handling GUI-s. ImageCollectionHandler allows for batch SIFT feature extraction via SiftWin, followed by codebook calculations via K-means clustering and quantization. ImageManipulator allows for visualizing SIFT feature clusters in images, as well as visualizing SRM segmentation. A partial example is shown in Figure 10.3.



(a) SIFT features in the image, clustered. (b) SIFT features in the image, represented as ellipses.

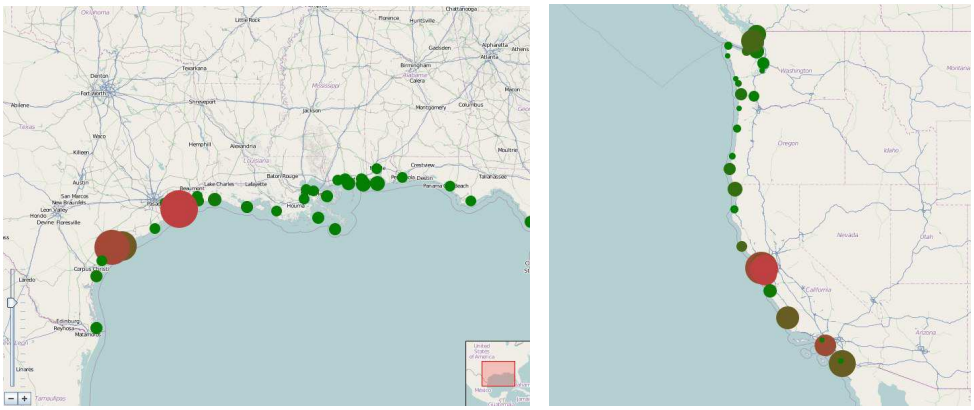
Figure 10.3: Visualization of SIFT feature clusters in Image Manipulator. SIFT features on an image are clustered and the clusters are drawn in different colors. Clusters can be represented as ellipses, where the axes follow the principal components of the clusters.

10.0.25 gui.maps

Hubness can sometimes be exploited for semi-automatic anomaly detection and gui.maps shows an application of hubness analysis for anomaly detection in oceanographic sensor data. Hub points with spatially inconsistent profiles were marked as potentially anomalous. The GeospatialSensorHubnessDrawer UI has then been used to generate images representing the anomalous sensor locations, with node size being proportional to the hubness of the measurement arrays. An example is shown in Figure 10.4.

10.0.26 gui.synthetic

Visual2DdataGenerator class can be used for manually generating 2D datasets, as examples for application of some data mining and machine learning methods. It is possible to either insert the



(a) Wind speed anomalous hub measurements.

(b) Water temperature anomalous hub measurements.

Figure 10.4: Visualization of spatially inconsistent and potentially anomalous hub sensor measurements via GeospatialSensorHubnessDrawer. The redness of a node corresponds to the spatial inconsistency.

points manually or to insert a sample from a Gaussian distribution of specified mean and variance. After insertion, it is possible to generate images of predictive performance of different classification methods on the generated data. An example is shown in Figure 10.5.

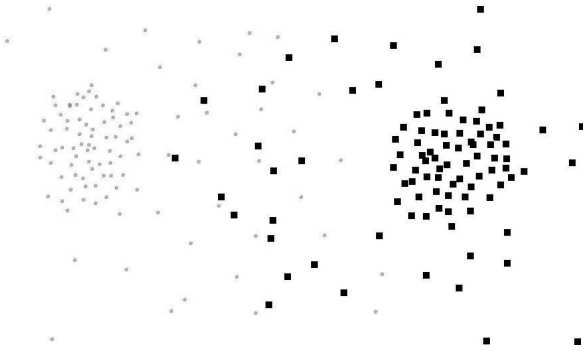
It can be seen that hubness-aware approaches generate smoother probability maps in borderline regions between different classes and that they are less prone to over-fitting in presence of label noise.

10.0.27 images.mining

This package contains the basic logic for handling quantized image feature representations. Codebook calculations are done by classes in the images.mining.codebook sub-package, either for SIFT or generic codebooks. There is also logic for calculating the visual word entropy distribution. Class-conditional codebook occurrence profiles can be calculated by using the CodebookProfileCalc class in images.mining.calc. Average colors in neighborhoods of certain points can be calculated by AverageColorGrabber within the same package. In images.mining.clustering, there are some experimental classes that aim to optimize inta-image SIFT clustering by optimizing the coefficients so that they conform to SRM image segmentation as much as possible. The remaining classes in images.mining are utility classes for quick data processing.

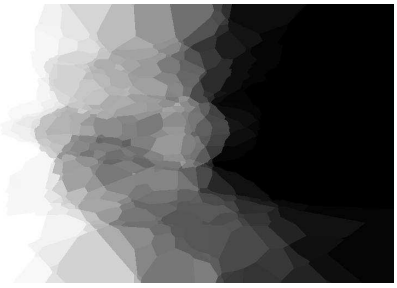
10.0.28 ioformat

Hub Miner operates with various data formats for dealing with input data and intermittent results. Classes needed to properly load and store all such data are located in the ioformat package. IOARFF class is used for dealing with ARFF data formats, dense and sparse. IOARFFDiscretized saves and loads discretized datasets in ARFF-like format, specific to Hub Miner. IOCSV is used for saving and loading CSV files. SupervisedLoader combines all data loads in a single interface and attempts to automatically guess the underlying data format during the load.

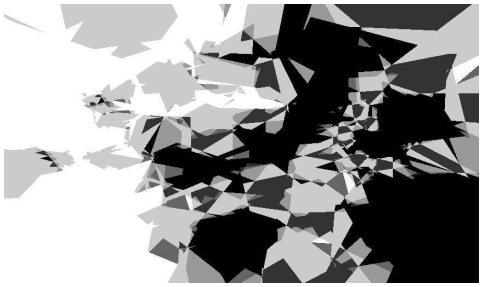


(a) The synthetic data set

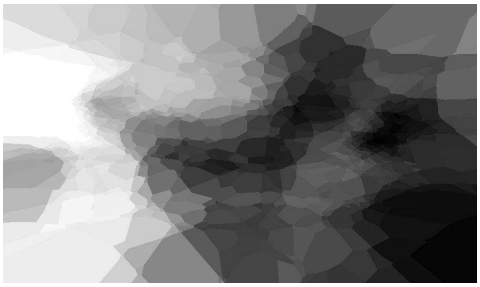
(b) kNN probability map



(c) HIKNN probability map



(d) kNN probability map, with label noise.



(e) HIKNN probability map, with label noise.

Figure 10.5: Probability maps inferred from kNN and HIKNN on synthetic data, for $k = 5$. Each pixel was classified by the algorithms and assigned a probability value of belonging to each of the two classes. Visualization was generated by Visual2DdataGenerator from gui.synthetic package.

FileUtil class implements some utility file methods, like for instance creating a file in a path that does not yet exist. The method then recursively goes up the abstract path hierarchy until it finds an existing directory and generates all the directories in between, including the target file.

DistanceMatrixIO includes methods for distance matrix save and load. Distance matrices in Hub Miner are represented as upper triangular matrices, so only the part above the diagonal is stored into the file. Each consecutive row is therefore one item shorter.

In ioformat.images, there are various utility classes for handling image data, embedding some frequently invoked functions. This includes the ThumbnailMaker, ImageFromRaster and SiftUtil.

Users can automatically summarize their experiments by invoking the BatchStatSummarizer class from ioformat.results on the command line. This class is automatically invoked at the end of the experimental run of BatchClassifierTester.

Most researchers prefer to use LaTeX for preparing their submissions to journals and conferences and Hub Miner contains some basic result summarizing capabilities for automatically generating LaTeX result tables for classification accuracy. In particular, LatexTableClassificationSummarizer and InstanceSelectionLatexTableSummarizer can be used to this end. In case the users need additional flexibility, they are free to either extend the existing framework or request certain updates in the future.

10.0.29 learning.supervised

Category and DiscreteCategory classes represent categories in the data and refer to collections of DataInstance objects and DiscretizedDataInstance objects, respectively. Classifier and DiscreteClassifier are abstract classes that classification algorithm implementations need to extend.

Several interfaces that allow for experimental optimizations in terms of requesting certain objects from the environment are available in learning.supervised.interfaces. These include DistMatrixUserInterface for algorithms that require distance matrices on the training data, NeighborPointsQueryUserInterface for algorithms that run kNN queries of test data against the training data, etc.

10.0.30 learning.supervised.evaluation

For details about running experimental evaluation in Hub Miner, see Chapter 5. The learning.supervised.evaluation package contains classes that implement most of the logic behind classifier evaluation. ValidateableInterface declares methods that classes need to implement in order to be eligible for evaluation in the framework. ClassificationEstimator calculates and stores the classifier performance metrics and the confusion matrix. ClassifierParametrization deals with listing and setting parameter lists and parameter-value maps for classification algorithms.

CVFoldsIO is responsible for loading and saving data splits for all iterations. ExternalExperimentalContext holds distance matrices and kNN sets to be used by algorithms in MultiCrossValidation while performing grid search over specified environment parameter ranges. The stored objects are the primary distance matrix and primary kNN sets. Secondary distance matrices are training split dependent and are calculated within MultiCrossValidation. The same goes for secondary kNN graphs. Distance matrices previously calculated for the same data under the same feature normalization scheme are loaded from the disk. BatchClassifierTester iterates over the environment parameter ranges and invokes cross validation runs in each tested case.

10.0.31 learning.supervised.meta

This package contains an implementation of AdaBoost.M2, as well as boostable hubness-aware base classifier implementations. Boosting does not always improve hubness-aware classification, but it can potentially lead to classification performance improvements in intrinsically high-dimensional data.

10.0.32 learning.supervised.methods

Since the main focus of the library is on evaluating the consequences of hubness in the data, most of the available classifiers are different types of k-nearest neighbor classifiers. However, other standard baselines are also implemented, for comparisons. For those baselines that are not currently supported, statistically correct comparisons are possible via OpenML (`http://openml.org/`)[vRBT+13].

As for kNN methods, this package contains many methods, including: kNN [FH51], dw-kNN, PNN [HA02], FNN [KGG85], NWKNN [Tan05], CBWKNN [DP13], AKNN [WNC07], hw-kNN [RNI09], h-FNN [TRMI13a], dwh-FNN [TRMI13a], HIKNN [TM12b], nondw-HIKNN [TM12b], NHBNN [TRMI11a], ANHBNN [TM13b], RRKNN [TM14b]. These implementations include both the recently proposed hubness-aware k-nearest neighbor classification approaches as well as some standard and less standard hubness-unaware kNN baselines.

Apart from k-nearest neighbor classifiers, Hub Miner offers some other standard classifiers like Naive Bayes, OneRule, KNNNB, LWNB, ID3 decision trees and robust stochastic learning vector quantization (RSLVQ).

10.0.33 learning.unsupervised

Clustering configurations in Hub Miner are often represented as integer cluster assignment arrays, but there is also the Cluster class that contains a list of indexes belonging to the cluster and implements many useful methods that make cluster processing easier, so that clustering configurations are often also presented as Cluster lists or arrays. For instance, within the Cluster class, there are methods for calculating the cluster diameter, the average intra-cluster distance and the centroid. ClusteringAlg class is an abstract class that the clustering algorithm implementations are to extend in order to be properly handled within the experimental framework.

10.0.34 learning.unsupervised.evaluation

The class responsible for running the clustering experiments is the BatchClusteringTester that runs a grid search over a batch of datasets for a list of clustering algorithms in a multi-threaded way. An automated approach for determining the optimal number of clusters in cluster range tests is available in LMethod implementation of the well known L-method approach for finding a 'knee' in the clustering quality index curve over the cluster number range [SC04]. BasicClusteringEvaluator is also available for initial testing of new implementations.

The learning.unsupervised.evaluation.quality sub-package contains implementations of various clustering quality indices. It is possible to evaluate the resulting cluster configurations by any of the following classes: QIndexCIndex, QIndexDaviesBouldin, QIndexDunn, QIndexGoodmanKruskal, QIndexIsolation, QIndexJaccard, QIndexRS, QIndexRand, QIndexSD, QIndexSilhouette. All of these classes extend the ClusteringQualityIndex class. OptimalConfigurationFinder helps with finding the best cluster configuration over multiple runs.

After the clustering has already been performed, it is possible to improve the final assignments by performing clustering refinement. PantSAStar algorithm is available for cluster refinements in learning.unsupervised.refinement [IER10].

Several one-off experimental scripts are included, like for instance clustering in presence of uniform noise. These scripts are included not only for result reproducibility, but also in order to demonstrate how similar scripts can be put together by Hub Miner users in their own experiments for their own research purposes.

10.0.35 learning.unsupervised.methods

Many clustering algorithms are available in learning.unsupervised.methods package, though this list is currently being extended by including even more implementations.

Several K-means variants are implemented in this package, including K-means, K-means++ [AV07], K-means-pruning [Als98], Harmonic K-means [Zha01], Kernel K-means [DGK04] and K-medoids. These partitional clustering approaches are commonly used in practice and make for good baselines for comparisons with more complex approaches.

DBScan [EpKSX96] is also implemented in the package, in order to enable users to perform density-based clustering. Density-based methods do not perform as well in intrinsically high-dimensional data, though it is possible to use smarter density estimates and reach reasonable performance.

Hub Miner also offers several recently proposed hubness-based clustering approaches for effective clustering in intrinsically high-dimensional data. These approaches are mostly extensions of the K-means partitional iterative framework and revolve around a recent observation that neighbor occurrence frequencies tend to be highly correlated with local cluster centrality when clustering in many dimensions. Therefore, hubs can be taken as prototypes during the iterations or used to guide the centroid search to more promising regions of the feature space. In either case, this usually improves clustering performance and has been shown to be much more robust to noise and capable of detecting the underlying structure of the data even in presence of large quantities of noise.

The hubness-based clustering implementations in this package include LKH, GKH, LHPC, GHPC [TRMI11c], GHPKM [TRMI13b] and Kernel-GHPKM [TRMI14]. These algorithms do not perform well in absence of hubness, but excel in high-hubness data, contrary to standard clustering approaches. There are more ways to exploit hubness in clustering and more new approaches will be included in future releases.

10.0.36 learning.unsupervised.outliers

All outlier detection approach implementations extend the abstract class OutlierDetector. The currently available implementations include the iterative clustering outlier detection, local outlier factor [BKNS00], local distance-based outlier factor [ZHJ09], local correlation integral [PKGF03], angle-based outlier detection [KShZ08] and anti-hub [RNI14], a recently proposed hubness-based outlier detection method.

10.0.37 linear

This package contains basic support for linear operations, linear subspaces and matrix decomposition. It also declares a DataMatrixInterface that allow the users to implicitly represent other object types as matrices, like DataSet objects, for instance.

10.0.38 networked_experiments

In order to support networked experiments via OpenML (`http://openml.org/`)[vRBT+13], this package implements classes and methods for connecting to OpenML services, requesting data and training/test splits, registering implementations and uploading run descriptions and experimental results. HMOpenMLConnector handles authentication issues, DataFromOpenML holds the fetched data about the number of splits, repetitions, training and test split indexes for all repetitions, as well as the fetched DataSet from the ARFF stream. ClassifierRegistrationOpenML performs implementation registration and fetches the implementation ID for the used algorithms. ClassificationResultHandler deals with preparing the results for upload to OpenML, along with the meta-data.

10.0.39 optimization.stochastic

Stochastic optimization is a useful tool in various optimization tasks and can also be used for data mining, whether for clustering or feature selection or instance selection. Genetic and evolutionary approaches have been successfully applied to these problems in the past, in various forms.

Hub Miner supports several types of stochastic optimization algorithms and these are available in the optimization.stochastic package. Most of these approaches revolve around the notion of solution fitness and a fitness function in Hub Miner needs to satisfy the FitnessEvaluator interface that

is located in optimization.stochastic.fitness. Another important feature is the ability to mutate the current solution or solution population into the next iteration. Hub Miner includes several types of mutation operators and these need to satisfy the appropriate mutation interfaces, like for instance: MutationInterface, RecombinationInterface, TwoDevsMutationInterface, HeterogenousMutationInterface.

In terms of stochastic optimization algorithms, several standard approaches are available in the optimization.stochastic.algorithms package. This includes simulated annealing, hill climbing, different types of genetic algorithms, differential evolution and predator-prey particle swarm optimization. Each algorithm is implemented in a separate class.

10.0.40 preprocessing.instance_selection

Instance selection is often used in conjunction with k-nearest neighbor classification. This has to do with scalability, as well as sensitivity of the basic kNN classifier to noise, due to its high specificity bias. Various prototype selection strategies can be used in order to filter out noise from the data and/or reduce the size of the training set.

Several standard instance selection techniques are available in the preprocessing.instance_selection package. All approaches extend the abstract InstanceSelector class. The implemented approaches include ENN [Wil72], CNN [PE68], GCNN [CKC06], RT3 [WM97], AL1 [DH11] and INSIGHT [BNST11b]. Random selection is also available, as a baseline for comparisons.

All of the implemented instance selectors can be used from within the batch classification experimentation framework in BatchClassifierTester and MultiCrossValidation in learning.supervised.evaluation.cv. They can be used in a *biased* and *unbiased* mode, w.r.t. prototype hubness estimation in hubness-aware classifiers.

10.0.41 probability

The probability package contains implementations of some basic probabilistic inference approaches.

A simple mixture model is available in the GaussianMixtureModel class. It combines several individual GaussianModel classes. Kullback-Leibler divergence can be calculated via the KLDivergence class. Perplexity class contains methods for calculating the model perplexity on the test data. NormalDistributionCalculator can be used for calculations regarding the normal distribution. VectorQuantization class implements methods for a vector quantization approximation of the underlying probability distribution.

10.0.42 sampling

Classes in the sampling package can be used to quickly sample the data.

10.0.43 statistics

The statistics package currently implements the logic for calculating distribution moments, feature variance and data covariance. CorrelationRatio class can be used to measure the relationship between the statistical dispersion within individual categories and the dispersion across the entire data sample. The package also contains an implementation of the corrected re-sampled t-test for statistical testing in cross-validation.

10.0.44 util

Many utility classes are grouped together in the util package. This includes CommandLineParser that is used for lots of command line parameter parsing throughout the library. DataSetJoiner can

be used to quickly join arrays of DataSet objects. SOPLUtil can quickly print out arrays so it is also very useful in de-bugging. HTTPUtil can be used for HTTP requests. ReaderToStringUtil can take a Reader and output a single String representing the content, which is very useful in JSON parsing. AuxSort contains sorting methods that return the index permutation along with sorting the arrays and ArrayList objects. ArrayUtil implements the binary search on arrays, as well as min/max operations and standardization.

The util.fileFilters sub-package contains some common filename filters for quick selections in the file system.

The most important class in util.text is IncrementalNGramBuilder that reads text and incrementally generates the n-gram vocabulary and builds a representation for each newly processed document or textual object.

10.0.45 visualization

This package contains the class that handles ViperCharts (`viper.ijs.si`) API calls for visualizing classification evaluation results, ViperChartAPICall. There are many visualization types available and the users can specify any chart type that they would like to see.

Eleven

Portability

Hub Miner is entirely implemented in Java and was developed with Java 7, so Java 7 or newer is required in order to properly build and run this project. All file paths in the code use platform-independent file separators, so there should be no file system specific problems on different platforms.

A small portion of image feature extraction specific code is currently tied to the Windows platform, in cases when SiftWin is used for SIFT feature extraction and when ImageMagick is used for image type conversion for this particular extraction. This dependency does not affect any of the major parts of the code and is not a requirement for running any of the experimental evaluation frameworks in Hub Miner nor for performing hubness-aware data analysis. This dependency will soon be entirely removed by switching to a purely Java-based image feature extraction library and increasing the support for OpenCV feature formats.

In the current release, it is possible to use Hub Miner for experimenting with hubness-aware classification, clustering, metric learning and instance selection or for performing data analysis and visualization, on all platforms assuming all Java dependencies are stated and present in the CLASSPATH variable. This makes for easy deployment and portability should not be an issue.

Twelve

Scalability

The implementation of Hub Miner's experimentation framework was made with speed in mind, so algorithms were made to re-use and share certain types of objects like kNN sets and distance matrices. There are also multiple internal optimizations for secondary distances and grid search over a range of possible parameter values. This is further improved by the multi-threaded experimental design.

While the experiments on small-to-medium scale problems run quite fast, it is not yet possible to run Hub Miner on genuinely large-scale datasets that contain hundreds of thousands or millions of examples. The bottleneck of most hubness-aware approaches in terms of computational complexity is the kNN graph construction on the training data. If the exact kNN sets are to be computed, storing the entire distance matrix in-memory can also be quite troublesome.

Hub Miner implements some initial support for large-scale experiments in terms of a generic fast approximate kNN graph construction via recursive Lanczos bisections. However, this is not really enough and future Hub Miner releases (See Chapter 13 for details) are to include several different approaches for scalable kNN search and kNN graph construction. Scalability is among the top priorities in terms of future implementation work.

While large-scale problems are obviously quite challenging and important, Hub Miner is very useful for dealing with another challenging issue: sparse high-dimensional data. Small and medium-sized datasets of this sort arise frequently in the biomedical domain, where generating labeled examples is either expensive or constrained by physical processes. Most of these problems are difficult and have a genuine real-world impact.

That being said, Hub Miner users can expect to see large-scale experimental support in future releases.

Thirteen

Plans for Future Releases

Hub Miner currently contains implementations of many useful machine learning and data mining algorithms, data processing techniques, data visualization UIs and an extensive experimental framework. However, I consider the current release to be merely a first step towards a more complete library geared for instance-based learning in intrinsically high-dimensional data. Of course, high-dimensional data mining is a vast field and there are many directions to consider. What follows is a list of features that the users can expect to see in future Hub Miner releases, hopefully very soon.

Improve scalability. Changes will be introduced throughout Hub Miner in order to enable large-scale data analysis, at least in certain contexts. Hubness-aware techniques have not yet been applied to large-scale datasets and enabling this should be one of the priorities in future implementation work.

Include indexing techniques for fast kNN search. In order to improve scalability, Hub Miner will include various fast approximate kNN graph construction and query approaches in future releases. This might be a substantial effort, but it would allow users to perform large-scale hubness-aware data analysis.

Include more intrinsic dimensionality estimators. Hub Miner currently implements one approach to estimating the intrinsic dimensionality of the data, but this is a vibrant field where many advances have happened in recent years and more intrinsic dimensionality estimators should be included in future releases.

Additional clustering implementations. Future Hub Miner releases will include some more complex and less standard clustering approaches.

Additional classifier implementations. While comparisons to other classifier implementations can easily be done through OpenML and the same will soon hold for clustering, the experimental framework of Hub Miner directly allows some types of experiments that are not present in other libraries, so this can not entirely eliminate the need for having more baselines. For instance, comparisons would not be that easy when secondary distances are used or when label and feature noise is to be introduced, especially non-uniform noise. Therefore, more baselines will be introduced to Hub Miner, in order to increase its experimental potential and usefulness.

Support for clustering via OpenML. While writing this, the clustering task is slowly being included in OpenML and will currently become available. As soon as it becomes available, it will be supported in Hub Miner as well.

Additional instance selection implementations. There are currently many instance selection methods that are supported in Hub Miner, but more will be added in the future as well.

Remove all SiftWin and ImageMagick dependencies. In order to make Hub Miner fully portable, all SiftWin and ImageMagick dependencies will be removed. They are currently

only present in non-central and less used parts of the code, but having any such dependencies is not a good thing, so this will be corrected. As for hubness-aware analysis itself and experimental evaluation for classification, clustering, instance selection, metric learning, etc. - none of these depend on image processing, so Hub Miner can already be used on different platforms. However, switching to a Java image feature extraction library will make the image processing pipeline portable as well, which would be an added value to the current library release.

Support various image feature extraction pipelines. Hub Miner is not an image feature extraction pipeline and image.mining packages are there more as convenience and examples of how it can be used to handle image data. Nevertheless, since images are a prime example of high-hubness data in many feature representations, Hub Miner support for handling different image feature types will be significantly extended in future releases.

Support semi-supervised classification. Semi-supervised classification is not currently explicitly supported in the experimental framework and this is soon going to change.

Support for ensemble methods in classification. Explicit support for building classifier ensembles will be included.

Include more boosting approaches. Future Hub Miner releases will include more boosting techniques, including some recently proposed approaches.

Support fuzzy methods. Learning from fuzzy labels is currently not supported in classification and clustering and fuzzy classification and clustering methods will be included, as well as a fuzzy experimental framework.

Bibliography

[Als98] Khaled Alsabti. An efficient k-means clustering algorithm. In *In Proceedings of IPPS/SPDP Workshop on High Performance Data Mining*, 1998.

[AV07] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1027–1035, Philadelphia, PA, USA, 2007. SIAM.

[BG05] I. Borg and P.J.F. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer-Verlag, Berlin, Germany, 2005.

[BKNS00] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: Identifying density-based local outliers. *SIGMOD Rec.*, 29(2):93–104, May 2000.

[BNST11a] Krisztian Buza, Alexandros Nanopoulos, and Lars Schmidt-Thieme. Insight: efficient and effective instance selection for time-series classification. In *Proceedings of the 15th Pacific-Asia conference on Advances in knowledge discovery and data mining - Volume Part II*, PAKDD'11, pages 149–160, Berlin, Germany, 2011. Springer-Verlag.

[BNST11b] Krisztian Buza, Alexandros Nanopoulos, and Lars Schmidt-Thieme. Insight: efficient and effective instance selection for time-series classification. In *Proceedings of the 15th Pacific-Asia conference on Advances in knowledge discovery and data mining - Volume Part II*, PAKDD'11, pages 149–160, Berlin, Heidelberg, 2011. Springer-Verlag.

[CKC06] Chien-Hsing Chou, Bo-Han Kuo, and Fu Chang. The generalized condensed nearest neighbor rule as a data reduction method. In *Proceedings of the 18th International Conference on Pattern Recognition - Volume 02*, ICPR '06, pages 556–559, Washington, DC, USA, 2006. IEEE Computer Society.

[DGK04] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. Kernel k-means: spectral clustering and normalized cuts. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 551–556, 2004.

[DH11] Bi-Ru Dai and Shu-Ming Hsu. An instance selection algorithm based on reverse nearest neighbor. In *Proceedings of the 15th Pacific-Asia conference on Advances in knowledge discovery and data mining - Volume Part I*, PAKDD'11, pages 1–12, Berlin, Heidelberg, 2011. Springer-Verlag.

[DP13] Harshit Dubey and Vikram Pudi. Class based weighted k-nearest neighbor over imbalance dataset. In Jian Pei, VincentS. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu, editors, *Advances in Knowledge Discovery and Data Mining*, volume 7819 of *Lecture Notes in Computer Science*, pages 305–316. Springer Berlin Heidelberg, 2013.

[EpKSX96] Martin Ester, Hans peter Kriegel, Jörg S, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[FGM05] Blaz Fortuna, Marko Grobelnik, and Dunja Mladenić. Visualization of text document corpus. *Informatica*, pages 497–502, 2005.

[FH51] E. Fix and J. Hodges. Discriminatory analysis, nonparametric discrimination: consistency properties. Technical report, USAF School of Aviation Medicine, Randolph Field, 1951.

[HA02] C. C. Holmes and N. M. Adams. A probabilistic nearest neighbor method for statistical pattern recognition. *Journal of the Royal Statistical Society: Series B*, 64:295–306, 2002.

[HKK+10] Michael E. Houle, Hans-Peter Kriegel, Peer Kröger, Erich Schubert, and Arthur Zimek. Can shared-neighbor distances defeat the curse of dimensionality? In *Proc. of the 22nd int. conf. on Scientific and statistical database management*, SSDBM'10, pages 482–500. Springer-Verlag, 2010.

[IER10] Diego Ingaramo, Marcelo Errecalde, and Paolo Rosso. A general bio-inspired method to improve the short-text clustering task. In *Proceedings of the 11th International Conference on Computational Linguistics and Intelligent Text Processing*, CICLing'10, pages 661–672, Berlin, Heidelberg, 2010. Springer-Verlag.

[JHS07] H. Jegou, H. Harzallah, and C. Schmid. A contextual dissimilarity measure for accurate and efficient image search. In *Computer Vision and Pattern Recognition*, pages 1–8, New York, NY, USA, 2007. IEEE.

[KGG85] James E. Keller, Michael R. Gray, and James A. Givens. A fuzzy k-nearest-neighbor algorithm. *IEEE Transactions on Systems, Man and Cybernetics*, pages 580–585, 1985.

[KShZ08] Hans-Peter Kriegel, Matthias S hubert, and Arthur Zimek. Angle-based outlier detection in high-dimensional data. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 444–452, New York, NY, USA, 2008. ACM.

[NS12] Krystyna Napierala and Jerzy Stefanowski. Identification of different types of minority class examples in imbalanced data. In Emilio Corchado, Vaclav Snasel, Ajith Abraham, Michal Wozniak, Manuel Graña, and Sung-Bae Cho, editors, *Hybrid Artificial Intelligent Systems*, volume 7209 of *Lecture Notes in Computer Science*, pages 139–150. Springer-Verlag, Berlin / Heidelberg, Germany, 2012.

[PE68] Hart PE. The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, 14:515–516, 1968.

[Pic09] Christian Pich. Mdsj: Java library for multidimensional scaling (version 0.2), 2009.

[PKGF03] S. Papadimitriou, H. Kitagawa, P.B. Gibbons, and C. Faloutsos. Loci: fast outlier detection using the local correlation integral. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 315–326, March 2003.

[PTR+11] Doni Pracner, Nenad Tomašev, Miloš Radovanović, Dunja Mladenić, and Mirjana Ivanović. WIKImage: Correlated Image and Text Datasets. In *SiKDD: Information Society*, 2011.

[Rad11] Miloš Radovanović. *Representations and Metrics in High-Dimensional Data Mining*. Izdavačka knjižarnica Zorana Stojanovića, Novi Sad, Serbia, 2011.

[RNI09] Miloš Radovanović, Alexandros Nanopoulos, and Mirjana Ivanović. Nearest neighbors in high-dimensional data: The emergence and influence of hubs. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, pages 865–872, San Francisco, CA, USA, 2009. Morgan Kaufmann.

[RNI10a] Miloš Radovanović, Alexandros Nanopoulos, and Mirjana Ivanović. Hubs in space: Popular nearest neighbors in high-dimensional data. *Journal of Machine Learning Research*, 11:2487–2531, 2010.

[RNI10b] Miloš Radovanović, Alexandros Nanopoulos, and Mirjana Ivanović. On the existence of obstinate results in vector space models. In *Proceedings of the 33rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 186–193, New York, NY, USA, 2010. ACM.

[RNI14] Miloš Radovanović, Alexandros Nanopoulos, and Mirjana Ivanović. Reverse nearest neighbors in unsupervised distance-based outlier detection. *IEEE Transactions on Knowledge and Data Engineering*, 2014.

[SC04] Stan Salvador and Philip Chan. Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pages 576–584. IEEE, 2004.

[SFSW12] Dominik Schnitzer, Arthur Flexer, Markus Schedl, and Gerhard Widmer. Local and global scaling reduce hubs in space. *The Journal of Machine Learning Research*, 13(1):2871–2902, 2012.

[Tan05] Songbo Tan. Neighbor-weighted k-nearest neighbor for unbalanced text corpus. *Expert Systems with Applications*, 28:667–671, May 2005.

[TB14] Nenad Tomašev and Krisztian Buza. Neighbor occurrencemodels for learning with label noise in high-dimensional data. *Neurocomputing, Special Issue on Learning with Label Noise*, pages 1–22, 2014.

[TBMN11] N. Tomašev, R. Brehar, D. Mladenić, and S. Nedevschi. The influence of hubness on nearest-neighbor methods in object recognition. In *Proceedings of the 7th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 367–374, New York, NY, USA, 2011. IEEE.

[TLM13] N. Tomašev, , G. Leban, and D. Mladenić. Exploiting hubs for self-adaptive secondary re-ranking in bug report duplicate detection. In *Proceedings of the ITI conference*, ITI 2013, Zagreb, Croatia, 2013. SRCE.

[TM12a] N. Tomašev and D. Mladenić. Hubness-aware shared neighbor distances for high-dimensional k-nearest neighbor classification. In *Proceedings of the 7th International Conference on Hybrid Artificial Intelligence Systems*, HAIS '12, pages 116–127, Berlin, Germany, 2012. Springer-Verlag.

[TM12b] N. Tomašev and D. Mladenić. Nearest neighbor voting in high dimensional data: Learning from past occurrences. *Computer Science and Information Systems*, 9:691–712, 2012.

[TM13a] Nenad Tomašev and Dunja Mladenić. Class imbalance and the curse of minority hubs. *Knowledge-Based Systems*, 53(0):157 – 172, 2013.

[TM13b] Nenad Tomašev and Dunja Mladenić. Hub co-occurrence modeling for robust high-dimensional knn classification. In *Proceedings of the ECML conference*, Berlin, Germany, 2013. Springer-Verlag.

[TM13c] Nenad Tomašev and Dunja Mladenić. Image hub explorer: Evaluating representations and metrics for content-based image retrieval and object recognition. In *Proceedings of the ECML conference*, Berlin, Germany, 2013. Springer-Verlag.

[TM14a] Nenad Tomašev and Dunja Mladenić. Hubness-aware shared neighbor distances for high-dimensional k-nearest neighbor classification. *Knowledge and Information Systems*, 39(1):89–122, 2014.

[TM14b] Nenad Tomašev and Dunja Mladenić. Image hub explorer: evaluating representations and metrics for content-based image retrieval and object recognition. *Multimedia Tools and Applications*, pages 1–30, 2014.

[Tom14] Nenad Tomašev. Boosting for vote learning in high-dimensional knn classification. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2014.

[TRMI11a] N. Tomašev, M. Radovanović, D. Mladenić, and M. Ivanović. A probabilistic approach to nearest neighbor classification: Naive hubness bayesian k-nearest neighbor. In *Proceeding of the CIKM conference*, pages 2173–2176, New York, NY, USA, 2011. ACM.

[TRMI11b] Nenad Tomašev, Miloš Radovanović, Dunja Mladenić, and Mirjana Ivanović. Hubness-based fuzzy measures for high-dimensional k-nearest neighbor classification. In *Proceedings of the MLDM Conference*, pages 16–30, Berlin, Germany, 2011. Springer-Verlag.

[TRMI11c] Nenad Tomašev, Miloš Radovanović, Dunja Mladenić, and Mirjana Ivanović. The role of hubness in clustering high-dimensional data. In *Advances in Knowledge Discovery and Data Mining*, volume 6634, pages 183–195, Berlin, Germany, 2011. Springer-Verlag.

[TRMI13a] Nenad Tomašev, Miloš Radovanović, Dunja Mladenić, and Mirjana Ivanović. Hubness-based fuzzy measures for high-dimensional k-nearest neighbor classification. *International Journal of Machine Learning and Cybernetics*, 2013.

[TRMI13b] Nenad Tomašev, Miloš Radovanović, Dunja Mladenić, and Mirjana Ivanović. The role of hubness in clustering high-dimensional data. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints):1, 2013.

[TRMI14] Nenad Tomašev, Miloš Radovanović, Dunja Mladenić, and Mirjana Ivanović. Hubness-based clustering of high-dimensional dataa. In *Partitional Clustering Algorithms*. Springer-Verlag, Berlin, Germany, 2014.

[vRBT+13] JanN. van Rijn, Bernd Bischl, Luis Torgo, Bo Gao, Venkatesh Umaashankar, Simon Fischer, Patrick Winter, Bernd Wiswedel, MichaelR. Berthold, and Joaquin Vanschoren. Openml: A collaborative science platform. In Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Őelezný, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 8190 of *Lecture Notes in Computer Science*, pages 645–649. Springer Berlin Heidelberg, 2013.

[Wil72] D. R. Wilson. Asymptotic properties of nearest neighbor rules using edited data. *IEEE Transactions on Systems, Man and Cybernetics*, 2:408–421, 1972.

[WM97] D. Randall Wilson and Tony R. Martinez. Instance pruning techniques. In *Proceedings of the fourteenth International Conference on Machine Learning (ICML)*, pages 404–411, San Francisco, CA, USA, 1997. Morgan Kaufmann.

[WME09] Josiah Wang, Katja Markert, and Mark Everingham. Learning models for object recognition from natural language descriptions. In *Proceedings of the British Machine Vision Conference*, London, UK, 2009. BMVA Press.

[WNC07] Jigang Wang, Predrag Neskovic, and Leon N. Cooper. Improving nearest neighbor rule with a simple adaptive distance measure. *Pattern Recognition Letters*, 28:207–213, January 2007.

[Zha01] Bin Zhang. Generalized k-harmonic means - dynamic weighting of data in unsupervised learning. In *First SIAM International Conference on Data Mining*, 2001.

[ZHJ09] Ke Zhang, Marcus Hutter, and Huidong Jin. A new local distance-based outlier detection approach for scattered real-world data. In *Proceedings of the 13th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, PAKDD '09, pages 813–822, Berlin, Heidelberg, 2009. Springer-Verlag.

[ZmP04] Lihi Zelnik-manor and Pietro Perona. Self-tuning spectral clustering. In *Advances in Neural Information Processing Systems 17*, pages 1601–1608, Cambridge, MA, USA, 2004. MIT Press.