KTH Information and
Communication Technology

Data-Efficient Machine Learning for Binary Outputs

Exploration of importance-weighted active learning, ensembling, joint training and class imbalance correction to reduce label complexity and training time in affiliate e-commerce product clasification

MATTIAS ARRO

Master's Thesis at KTH Information and Communication Technology
MSc Data Science (EIT Digital track)

Academic Examiner: Magnus Boman
Academic Supervisor: Jim Dowling
Industrial Supervisor: Abubakrelsedik Karali

2018

Contents

1. Introduction

Machine learning (ML) has become hugely successful over the past few years, with a lot of this newfound interest, hype, and hysteria directed at neural networks and deep learning. This focus is not unfounded - deep learning approaches continue to break benchmarks in core machine learning research areas such as computer vision [cite], speech recognition [cite], and an increasing number of natural language processing tasks [cite NMT]. Deep learning has also revolutionised reinforcement learning, achieving superhuman performance in complex games and driving vehicles in real-world situations. There are even limited results in beating human at highly uncertain games with various actors such as [Texas Holdem] poker [cite]. Understandably, academics and industry are are scrambling to apply panacea to their field.

While their superficial resemblance to natural brains might be a good topic for a sensationalist article, artificial neural networks are simply layers of non-linear transformations capable of learning complex mappings from multidimensional inputs to (usually multidimensional or structured) outputs. The building blocks of neural networks are relatively simple and the algorithms for training them are universal; this makes neural networks applicable to a variety of domains, and opens up fascinating opportunities of multimodal and transfer learning. Being able to arbitrarily increase model complexity by increasing its depth or width allows the same neural network approximate more complex functions. Increased model complexity increases training time and requires more labeled training data, yet deep models are somewhat unique in that their performance continues to increase when the dataset size increases, whereas the benefits of more data taper off for many other kinds of models. This does not automatically mean neural networks can only be used with large datasets - after all a single layer neural network can be equivalent to a logistic regression model - but that model complexity should increase with the amount of data available.

It is not immediately obvious which kind of model should one use for a given task and dataset. A data scientist can consider the following factors: how many labelled and unlabelled data points do we have, how much do we value predictive performance, interpretability, and whether we want to do some transfer learning or joint training with the models. Labelling is often expensive, so in many real world use cases a lower label complexity (number of labels needed to obtain the desired

accuracy) is preferred over slightly better performance. Deep learning seems to have a disadvantage in this aspect, but as we see in section [ref] in cases where unlabelled data is also abundant, semi-supervised and generative models can overcome low label complexity while increasing computation time. In cases where the ability of neural networks to learn features that can be used in downstream models (e.g. features learned for classification could later be used as part of a recommended system) this increased computation and engineering complexity might be justifiable.

In this thesis, we explore three orthogonal ways of efficiently learning on a proprietary dataset for product classification, where initial labels are abundant but noisy. We first evaluate different kinds of models (shallow, deep, tree-structured, convolutional, recurrent) that are trained on different modalities / input dimensions (image, text, categorical, numerical) of the same data. After determining the performance of these baseline models, the strongest models are trained as an ensemble that outperforms each individual baseline model. Finally we fine-tune the ensemble via an active learning strategy described in section [ref], where a combination of uncertainty and disagreement sampling determines a batch of products to be labeled for the next training iteration. This overcomes the noisiness and incompleteness of the initial labels without requiring much manual labeling.

1.1 Problem

The client company gathers data from various affiliate networks (that in turn give their data from various retailers) and displays the data on their online store. There are millions of products belonging to roughly 800 categories, and categories follow the usual nested tree structure. The incoming data is extremely noisy and inconsistent: what kind of data is stored in what kind of feature column varies across affiliate networks, across retailers within an affiliate network, and the data within a retailer can have lots of missing values, noisy text, missing images, etc. There is currently a rule-based system for assigning products to categories: all products matching a condition (e.g. title contains the word “trousers”) will be assigned to that category, i.e. categories are not mutually exclusive. This way of categorising products works relatively well on some categories, but such a rule-based system has several drawbacks: these rules are cumbersome to define, their evaluation is manual, they failed to match a large fraction of products that in principle should be in a given category, it is hard to trace back the rule that caused a false positive, and such rules are limited to textual data.

The client needs a classification system of binary independent outputs to replace the old way of categorising products. The system should be able to learn from the output of the old system, and if possible produce models that can be used in downstream tasks such as recommend systems and product similarity models (transfer learning). The highest priority is low label complexity, beating requirements for high accuracy and transfer learning capabilities. The system should be robust to noisy inputs; data preprocessing should not consider the idiosyncrasies of each af-

filiate network. There is an additional feature the client requires: given a product image, the visitor should be shown products that are visually similar.

1.2 Purpose

The academic purpose of this work is to (1) assess the relative strengths of different kinds of models and their ensembles, and (2) to determine whether an active labelling strategy reduces label complexity on a real-world dataset. Analogously, the commercial purpose is to (1) obtain a model with powerful predictive capabilities, and to (2) reduce costs by using an efficient labelling strategy, and (3) obtain a high-quality product similarity score.

1.3 Goals

The goals of the work, in chronological order, is to:

- Use a pre-trained 2-dimensional convolutional neural network (2D CNN) to extract features for an approximate nearest-neighbour search of visually similar products.
- Build an interface for subjectively evaluating the visual similarity algorithm.
- Train baseline model to reproduce the behaviour of the rule-based system.
- Build an interface for labelling products and obtain small dataset with ground truth labels (further referred to as the "ground truth dataset").
- Train a number of different models on the rule-based labels. Evaluate these models on the rule-based test set as well as the ground truth dataset.
- Train a selection of models that had good performance as an ensemble, preferring model diversity over good performance. Evaluate this model on the rule-based test set as well as the ground truth dataset.
- Implement the active labelling mechanism defined in [ref] and fine-tune the ensemble (that is pre-trained on rule-based labels) in 10 labelling rounds.
- Document the results as well as the technical architecture and workflow.

1.4 Hypotheses

The following hypotheses were postulated prior to running most experiments[1]:

1. Pre-trained 2D CNNs perform reasonably for visual item similarity, but fine-tuning might be needed.
2. Linear models are relatively good at predicting higher-level categories, worse at predicting lower-level categories.

[1] the Wide&Deep baseline model had been trained on rule-based labels

3. Deep models for histogram / tf-idf inputs do not improve substantially on linear models.
4. Shallow models with embedding inputs generalise better to the ground truth data, and deep models with embedding inputs generalise slightly more.
5. XGBoost is the best performing model for textual and categorical data.
6. Convolutional neural networks (CNNs) that are pre-trained on Imagenet data are consistently good predictors of clothing products, yet failed to accurately predict categories such as technology.
7. Fine-tuning CNNs will give little, if any, performance improvement due to inefficient training data.
8. Ensembling gives better performance than any individual model.
9. A meta-learner with a few layers obtains better results than ensembling by averaging.
10. Active learning substantially decreases label complexity. This will be evaluated subjectively, given the difficulties outlined in section [ref].

How these hypotheses were evaluated is described in section 3.4.

1.5 Ethical Consideration

Given how pervasive machine learning is becoming across areas of society, it is good that discussions are happening about safety, transparency and bias in machine learning systems that have the ability to affect lives. The worst case scenario for inaccurate, opaque or biased product classification is embarrassment for the client company, therefore ethical considerations not of much concern for the client company. It is still worth reminding the reader, who might use some of the ideas in this work, that machine learning models always contain some kind of bias: from the data users as input (the way it is gathered, what is gathered), the way the data is processed, and the kinds of models used.

Already at this early stage of machine learning adoption there are cases where blackbox algorithms have been used to decide decisions that severely influence persons life - predicting reoffending probability of convicts to determine whether they are given parole or not - with terrible results. The model that was touted to be and objective substitute to a subjective judge has simply learnt that biases inherent to the data (the prejudices and racism of the judges at the time): it consistently underestimated reoffending rate of white convicts and overestimated the reoffending rate for black convict [cite]. Even though the data did not contain explicit information about race, the algorithm was capable of implicitly detecting grace through some other variable that was correlated with race. Most data scientists will never build models with such severe consequences, but even more subtle decisions made autonomously by algorithms can prolong the inequalities of our society by consis-

tently favouring certain characteristics such as race, gender, place of origin, etc. Prime examples are algorithms that determine whether one gets a mortgage or not, what the premium on their insurance would be, whose CV gets shortlisted for a position, and so on.

1.6 Sustainability

The reality of any e-commerce company is that increase revenue almost by definition means more waste and increased carbon released into the atmosphere as a result of production and transport. Efforts to mitigate these unwelcome side effects can be successful at a government level, though the author firmly believes there ought to be stronger intergovernmental regulation to tax carbon emissions and the consumption of materials, as currently there is absolutely no incentive for retailers or consumers to reduce waste, and few incentives to recycle it. The best a data scientist working for a retailer can hope for is poor performance of his models, which would not lead to increased sales.

1.7 Delimitations

1.8 Outline

2. Background

3. Method

Several hypotheses were tested in this work. Section 3.1 gives an overview of how input data was pre-processd. Section 3.2 describes the technical set up required for running all these experiments and deploying the model to production before delving into the specific experiments and their evaluation methods in sections [ref].

The experiments were conducted in four distinct stages:

- Training a baseline model on the rule-based labels to get a sense of the difficulty of this problem. Since there is no fundamental difference in the way this model was trained and evaluated compared to the other classifiers, this is described in section 3.3.2 with the others. After this step, employees of the client company labeled products that would become the ground truth dataset. The visual comparison feature was also subjectively evaluated at this stage.
- Training the independent classifiers to determine best performers (3.3.2).
- Training an ensemble of the independent classifiers (3.3.3).
- Training up to 10 iterations of active learning on the strong predictor (3.3.4).

3.1 Data Preprocessing

There were around a dozen product features that affiliate networks provided. Most of these features were either categorical or textual, with just a single numerical feature (price). Initially, the data was analysed using Dataprep, a Google Cloud Platform (GCP) product for data wrangling, which at the time of use was in beta stage. Dataprep was used to process a sample of 800 000 products; it produced histograms of the values present in each feature column (see appendix A.1).

The histograms revealed that a lot of the input features were mostly empty, but also that many of the inputs that would naturally be considered categorical had much more unique value in them then one might expect. For example, each affiliate gives us the textual representation what they consider to be the category of the product, but rather than containing a small number of unique tokens, these contained all the full category paths along with the category delimiters, which varied retailer by retailer (e.g. it was common to see both "Shoes > Sneakers" and "Shoes // Sneakers"). Representing these as categorical variables would have blown up the

input space, which would have resulted in more parameters, each parameter having fewer examples to learn from. Therefore, many such "categorical" were actually represented as text, which were tokenised and cleaned appropriately, allowing for better generalisation and smaller models.

There was a single numerical field: price. This could have been min-max normalised to the range 0 ... 1, however there was a small number of very high values that would have squash nearly all the other prices. Rather than carefully considering how to mitigate this, the input dimension was dropped, because it is not likely to have much predictive value for product classification. It would be trivial to bring this feature back, for example when building a recommended system, where it would be much more useful.

A trickier question was how many distinct tokens or categorical values to keep per input column. Keeping all of them would not have been sensible: there were still large numbers of tokens that appeared only once, often because there were some unwanted formatting characters, misspellings, or incorrect punctuation that caused a token to be considered a separate entity. There was a single configuration file that dictated which models used which features as input, whether those inputs were represented as textual, categorical, or dense values; it also determined the maximum number of unique values/tokens, and the dimensionality of the embedding. This configuration file was read by Dataflow during pre-processing and by TensorFlow during inference and training, which made experimentation with different types of models and input representations considerably easier.

Below is a list of input features with information about how they were represented; it also lists the dimensionality of embeddings for the models which encoded categorical variables as embedding.

- title - text - max 8000 unique tokens
- brand - categorical - max 5000 unique values - 10 embedding dimensions
- category - categorical - max 950 unique values - 6 embedding dimensions
- rawCategory - text - max 1000 unique tokens
- description - text - max 8000 unique tokens
- gender - categorical - take all unique tokens
- size - categorical - max 100 unique tokens
- image - dense vector of 2048 features extracted with a 2D CNN

3.1.1 Category Structure

At the time of writing, there were roughly 1000 categories defined in the client database. Categories were structured in a way that is typical of e-commerce: categories can have child categories, which in turn can have child categories, etc. In our case, the typical depth of the tree structure was five, i.e. a leaf category often had four parents; naturally, the tree structure was not balanced, so many branches ended at depth three or four.

This structure has to be taken into account when labelling and predicting categories for unseen products. When a product is labelled to belong to a lower level

category, it should also be labelled to belong to all of its ancestor categories. This does not work the other way around, though: if a product is labelled not to be a Trainer (lower level category), this does not imply it is not a Shoe (ancestor category). Similarly, when the model predicts that a product belongs to and lower level category, it is considered that it also predicted the product to belong to all of its ancestor categories; again, negative predictions at lower levels do not overwrite positive predictions at high levels.

3.2 System Architecture

The following technologies were used to build the system which had to interact with existing services at the client company:

- Apache Airflow (AF) - a Python framework for defining workflows of long-running tasks and dependencies between these tasks.
- TensorFlow (TF) - ML framework for Python, capable of defining many kinds of models as a computation graph, and executing this graph locally or in a distributed manner.
- ML Engine (MLE) - a GCP service for running TensorFlow models (training, hyperparameter tuning, inference).
- Apache Beam - a data processing engine akin akin to Apache Spark and Apache Flink.
- Dataflow - a GCP service for executing Apache Beam workloads.
- Tensorflow Transform - a Python library with a small set of operations for data preprocessing that can run inside a TensorFlow graph as well as an Apache Beam pipeline.
- Google Cloud Storage (GCS) - Google Cloud Platform (GCP) object storage similar to Amazon S3.
- ElasticSearch (ES) - a NoSQL database with powerful full-text search and querying capabilities.
- RabbitMQ - a message queue, used for transferring data among our microservices (using the Logstash adapter, that can read from and write to (among other things) ElasticSearch and RabbitMQ).
- Flask - a simple backend web framework for Python.
- Node.js - a JacaScript backend web framework.
- React.js - a JavaScript front-end framework for JavaScript.
- Redux - a framework for persisting user interface (UI) state and application data in single page applications.
- GraphQL - a query language for building flexible APIs

Figure 3.1 shows the how data is passed between the main services, and how services and technologies interact.

All product data is stored in ElasticSearch (ES): the rule-based labels, the predictions of the ML system, and evaluation metrics from various train runs. ES is accessed from the public web application via GraphQL and the ML administration web UI (further referred to just as web UI[1]). Data is pulled into the ML pipelines by dumping the results of an ES query to a local file, which is uploaded to GCS. Updates to the ES index are not done directly, since indexing the updated products is computationally expensive; instead, updates are put on a RabbitMQ queue, which is consumed by Logstash, which in turn updates products in ES at a rate that will not overburden the servers.

All ML training and prediction happens in a batch-oriented way, encapsulated as Airflow pipelines. Each pipeline is a directed acyclic graph of tasks, where a task can be a shell command or Python function; a pipeline defines dependencies of task execution, which allows us co-ordinate a series of operations that could be executed locally (inside the Docker container running AF) or remotely (such as in a GCP service). A typical pipeline dumps data locally, uploads it to GCS, schedules a Dataflow job to preprocess data, polls the Dataflow service until the Dataflow job is finished, schedules an ML engine job, polls the MLE service until it has completed, and runs an update process that reads the predictions and evaluation statistics from GCS and sends the updates to the RabbitMQ queue. Reading updates and sending these to RabbitMQ is done in a parallelised manner (using multiprocessing), since the update process is bottlenecked due to network latency as well as computing the appropriate category path for each product (explained in 3.1.1).

3.2.1 Dataflow Pipelines

There are two types of Dataflow pipelines: for preprocessing training data and for calculating product-product visual similarity. Omitting various details, dead-ends and workarounds that were needed due to technical limitations and prior system architecture choices[2], the pipelines had the following tasks:

Preprocessing

This pipeline loads the products dumped from ES (as JSON) and preprocesses data according to a configuration file (see section 3.1). All fields are cleaned of obvious noise and superfluous whitespace. Text and categorical fields are tokenised and mapped to integer indices, keeping only the top k values and also computing TF-IDF scores for text fields. This is handled by TensorFlow Transform, which persists this token-to-index mapping in a *transform function* that is saved to GCS

[1]The web UI was initially built with Flask and React by the author as a quick way to get insight about the model, and then re-written as a more feature-rich version by an employee of the client company with Node.js, GraphQL, React and Redux.

[2]which were completely reasonable at the time, when the ML system was not a consideration

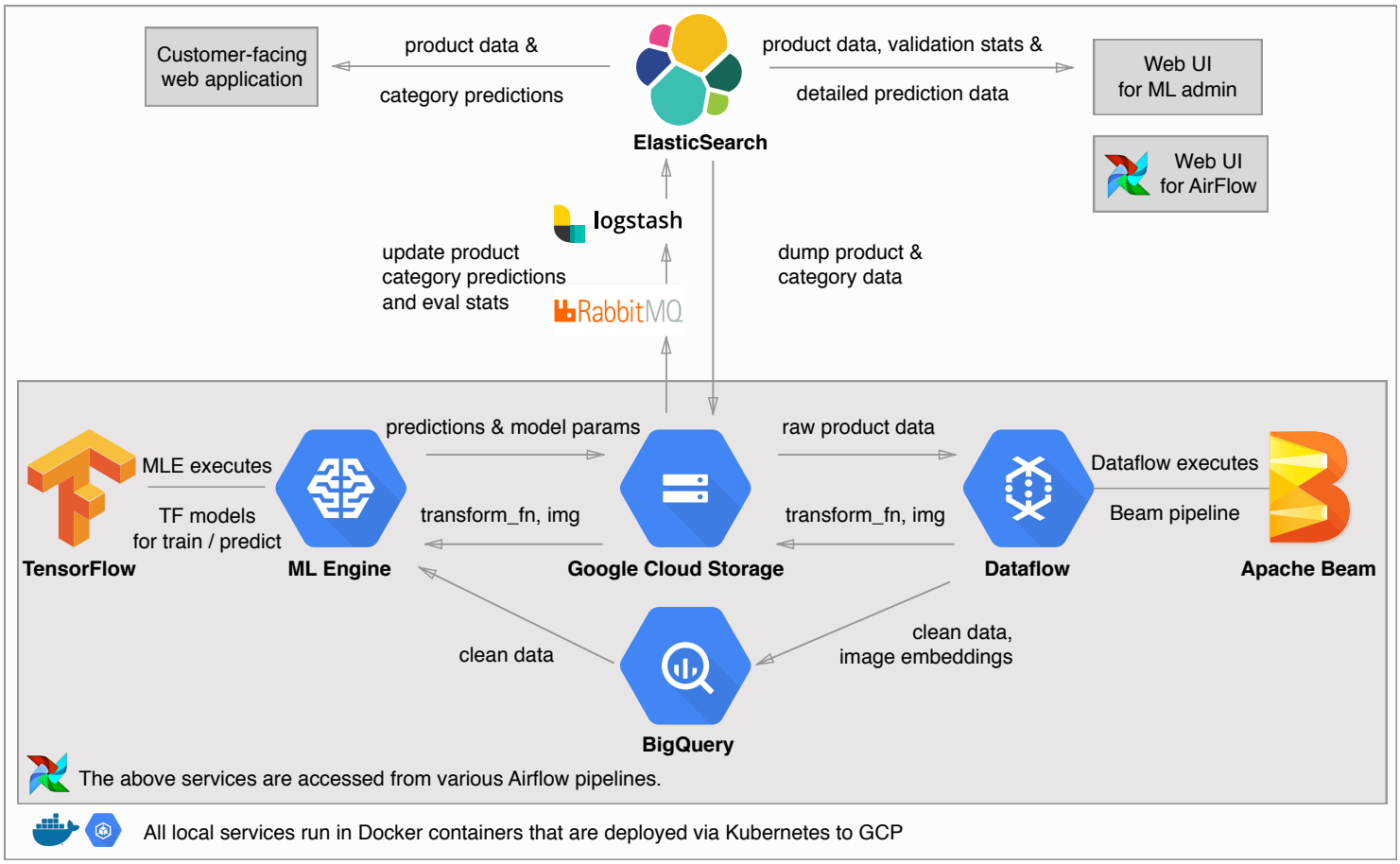

Figure 3.1. High-level system architecture of the ML pipeline

at the end of the pipeline. The transform function is used by a TensorFlow model to convert raw text inputs to a sparse inputs; separate pre-processing run will generate a different transform function, with mostly the same tokens mapping to different indices, as the order in which they will be encountered will be different.

The pipeline is also responsible for downloading product images and using a pre-trained 2D CNN to extract a dense feature vector for each product from the penultimate layer of the CNN.

The clean data and image embeddings are inserted into a new BigQuery table after each run[3]. Note that the data is not inserted in its tokeniser form. Storing tokeniser data would reduce the space it takes marginally, but keeping raw data makes the system easier to debug, and enables the option to do inference on raw data that is not tokenised (e.g. in a streaming banner, when we might not have access to the transform function).

[3]BigQuery is inefficient in updating existing records and limits the number of update per day

Visual Similarity

This Dataflow pipeline uses the data in BigQuery as input. As explained in section 3.3.1, it needs to find the *k=100* nearest neighbours of each product based on the cosine similarity of their image embeddings. The product-product similarities are computed within products that belong to same second level category, therefore the pipeline only needs to extract the categories previously predicted and image embedding from BigQuery. The resulting predictions (top 100 product UUIDs per product, ordered by similarity) are saved to GCS.

3.3 Experiments

3.3.1 Visual Similarity

ann tfidf-based

3.3.2 Independent Models

3.3.3 Ensembling

3.3.4 Active Learning

3.4 Evaluation

At the beginning of running all these experiments, the dataset was divided into development and test set (90/10%). The development set was used for

we can evaluate the performance of the model on three types of datasets:

- the test or validation set as labelled by the rule-based system (referred to as "rule-based test/validation set"),
- the ground truth dataset gathered before running most experiments,
-

4. Discussion

5. Conclusion

References

Declaration

I hereby certify that I have written this thesis independently and have only used the specified sources and resources indicated in the bibliography.

London, UK, March 29, 2018

.....
Mattias Arro

A. Screenshots

A.1 Dataprep Histogram

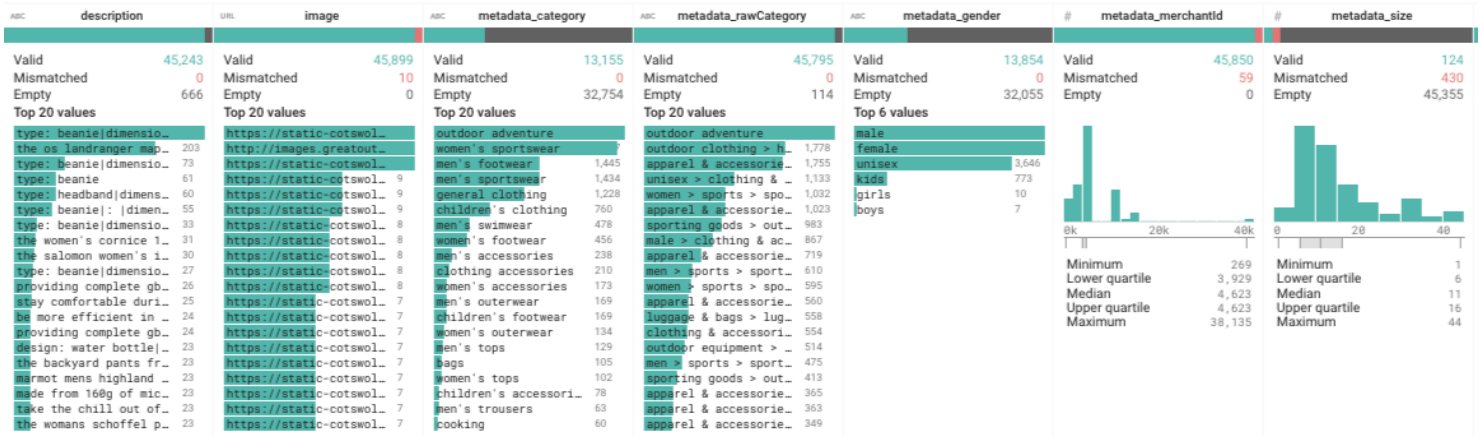

Figure A.1. Dataprep: the histograms of field values of a subset of fields.