KTH Information and
Communication Technology

Data-Efficient Machine Learning for Multi-Label Classification

Exploration of various models - shallow, deep, tree-structured, and ensebles thereof - for multi-label product classification in multi-modal and transfer learning settings.

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

our models do not explicitly model the correlation between output variables

1.8 Outline

2. Background

2.1 Input Representations

2.1.1 Categorical Input

The simplest option for representing categorical input is 1-hot encoding, where input has as many dimensions as there are distinct categorical values (vocabulary size); a single dimension is set to 1, with all other dimensions set to 0. This is a straightforward representation: for a simple model such as logistic regression, we can clearly interpret the model parameters and see how the presence or absence of a given category increases or decreases the likelihood of a given output. However, in cases where vocabulary sizes get large, and the number of outputs (the number of units in the next layer in a deep network, or the number of output units in a shallow one) increases, this kind of encoding can really blow up the number of parameters of the model. this has many negative consequences: more memory, more required train data - curse of dimensionality

An alternative is to use random embeddings: dense, low-dimensional representations of a high-dimensional vectors. It has been shown in compressed sensing literature that if a high-dimensional signal in effect lies on a low-dimensional manifold, then the original signal can be reconstructed from a small number of linear measurements [4]. This has a useful implication for machine learning: categorical variables with large vocabularies of size d can be represented with random embedding vectors of size $M = \log(d)$. This because it is possible to reconstruct any d-dimensional k-sparse signal using at most $klog\frac{d}{k}$ dimensional vectors [2], and 1-hot vectors are 1-sparse (only one dimension is nonzero).

An intuitive example is given in [9] about natural images: a 1-million-pixel image could have roughly 20,000 edges[1], i.e. it lies in a roughly 20,000-dimensional manifold from which the original image can be constructed with little loss in quality. The 1-million-dimensional image could be projected into a M-dimensional space "by taking M measurements of the image, where each measurement consists of a weighted sum of all the pixel intensities, and allowing the weights themselves to be chosen randomly (for example, drawn independently from a Gaussian distribution)"[9].

[1] technically: wavelet coefficients with a significant power, which roughly corresponds to a superimposition of edges

The projection these random because the weights for the sum of pixel intensities are chosen randomly. Why the projections need to be random is motivated by another intuitive example: when light is shined on a 3-dimensional wireframe, its shadow is a 2-dimensional projection of it. The projection could lose some important information about the regional object if the direction of the light source is chosen poorly, however random directions are likely to result in projections where every link of the wire will have a corresponding nonzero length of shadow.

2.1.2 Text Input

The simplest way to represent text is bag words (**BoW**), which is simply the histogram of word occurrences in the text. BoW representations give equal weight to each of the words in the text, and the model has to learn the relative importance of each of these. A commonly used representation in information retrieval (IR) is **TF-IDF**, which stands for "term frequency - inverse document frequency". TF-IDF multiplies the term frequency (number of times the word occurred in the text) with its inverse document frequency (the inverse of how often a occurs in all documents). This has the effect of giving lower scores for common words, and higher scores for words which appear often in a document but not too frequently in the whole corpus. Another common representation is bag of n-grams (**BonG**), which is a histogram of character n-grams.

The above representations are sparse: there are vector representations grow linearly with the vocabulary size. Text can also be represented as a sequence of **word embeddings**. Words and betting can either be random vectors or representations learned with word cooccurrence algorithms such as word2vec [] and GloVe []; these representations can remain fixed throughout the learning procedure, or updated as part of the stochastic gradient descent (SGD) when applicable. A simple yet surprisingly powerful way to represent text is to average the word embedding contained in it []. The average could also be weighted by the TF-IDF score of each word[2].

More complex methods use recurrent neural networks (RNNs) to combine a sequence of word embeddings into a **fixed length vector**, which are described in more detail in section 2.2.2. These kinds of models are good at disambiguating the potentially many meanings a word might have. Exactly what gets persisted in the final vector representation of text depends on how the model is trained - two RNNs trained on different tasks (such as sentiment analysis and named entity recognition) would need to store different information in its hidden state to successfully perform their relevant tasks, hence the encoding for the same sentence would be different for either model. Still, an RNN that is trained on one task could provide useful features for another. Encoding text as fixed length vectors using RNNs has been interpreted as compressed sensing [1], with such vectors being "provably at least as powerful on classification tasks, up to small error, as a linear classifier over BonG vectors".

[2]the author is unaware whether this has been tried before, but it seems a promising approach

2.1.3 Image Input

Images can be represented as dense 3-dimensional tensor. A 28x28 RGB image could be represented as a tensor with shape [28, 28, 3], with the final dimension corresponding to the green, red, and blue intensity values at a given x/y coordinate. These intensities are usually in the range 0 ... 255, but are normalised before input to machine learning model. Similarly to RNNs encoding a sequence of words to a vector, 2D convolutional neural networks (2D CNNs) can be used to extract dense feature vectors from raw images (image embedding), further discussed in section 2.2.3.

2.2 Models Considered

The following sections describes a selection of models that could be used for our classification task. This is by no means an exhaustive list, nor is it a selection that is expected to have the highest predictive performance individually. These models will later become part of an ensemble, where diversity matters much more than the performance of any individual model. Engineering considerations and time constraints also influence the selection of models: we did not want to spend time implementing models from scratch, or to use many frameworks for training models. TensorFlow is a widely used machine learning library with a good selection of open source models, in particular a selection of pre-trained models for computer vision and NLP, and has arguably the most mature deployment ecosystem. Therefore models that did not have an open source TensorFlow implementation were automatically excluded.

Section 2.2.1 describes models that can take categorical and text inputs, section 2.2.2 looks at some neural models that take only text inputs, and section 2.2.3 describes 2D CNNs that can classify or extract features from images. In all cases the models are described in terms of how they could be used for solving the problem of multi-label classification.

2.2.1 General Models

Logistic Regression

Logistic regression is a linear, discriminative classifier that is often the baseline model of choice, since it is both interpretable and efficient to train. There are methods that take time linear in the number of non-zeros in the dataset, which is the smallest amount possible, and it can be made to handle non-linear decision boundaries by using kernels [13]. Our case is the binomial logistic regression, with a separate logistic regression model for each output class. This is shown formally in eq 2.1, where where x and w correspond to the input and weight vectors, respectively. We follow the notational trick where the first element of x is always 1, and the first element of w corresponds to the bias term. The loss function of this model as well

as how it is optimised is described in section 2.2.1.

$$p(y|x, w) = \text{Ber}(y|\text{sigm}(w^T x)) \tag{2.1}$$

Feedforward Neural Network

Feedforward neural networks (also called deep feedforward networks, multi-layer perceptrons) have been described as function approximator, whose goal is to approximate some functions f^* [10]. We expect some familiarity with neural networks from the reader; for an excellent overview of the various use cases and models of deep learning, refer to. In our case, the input is some representation of categorical and text input: sparse BoW or TF-IDF of text, 1-hot encoded or random embedding vectors of categorical variables, or embedding is extracted from text or images using pre-trained recurrent or convolution on neural networks. The hidden layers of our network are homogenous: they use the same activation function (ReLu, sigmoid, or tanh) and have the same number of units; activation functions and the number of units in hidden layers are determined through hyperparameter tuning. The output layer uses sigmoid activation function and has the same number of units as there are classes.

Wide & Deep

Loss Function and Optimizers

Logistic regression and neural networks are optimised with some variant of gradient descent. For our task the loss function binary cross-entropy between the training data and the model distribution, with the loss value averaged across all classes. This is given in eq. 2.2, where θ corresponds to the model parameters such as the weights and biases of the model, C corresponds to the number of classes, and $P(c = 1|x, \theta)$ gives the predicted probability that the item belongs to class c given the feature vector x.

$$NLL(\theta) = \sum_{c=1}^{C} \sum_{i=1}^{N} [c_i \log P(c = 1|x, \theta) + (1 - c_i) \log(P(c = 0|x, \theta))] \tag{2.2}$$

We are minimising the negative log likelihood (NLL) rather than maximising the product of likelihoods. Multiplying large numbers of probabilities could result in numerical underflow and rounding errors, therefore it is pragmatic to work with sums of log probabilities instead. The Adam [15] optimiser works very well with default hyperparameters, and is therefore used for all stochastic gradient descent updates.

2.2.2 Text Models

1D Convolutional Neural Networks

A simple convolutional architecture for text classification is described in [14]. Pre-trained k-dimensional word vectors are concatenated to represent a sentence, and a filter is $w \in R^{hk}$ is applied to a window of h words at each possible window of words in a sentence; this gives a single variable-length feature vector representing the sentence. Several such filters are learned (each with potentially a different width h), and max-over-time pooling is applied to these feature maps; this gives a vector of the highest activations from each filter, which is then passed to a fully-connected softmax layer for final classification (in the case of multi-class classification). For multi-label classification, the final layer would be a fully-connected layer of sigmoid activations.

This simple architecture should be able to predict output classes reasonably. Each filter can indicate the presence of a a sequence of h words; max-pooling discards information about where exactly in the text it appeared, and ensures the output is of a fixed length. The same filter w is used across all possible word windows in the sentence, which can be seen as a form of parameter sharing (and as an infinitely strong prior over the parameters of the model [10]), and enables processing variable-length sequences. The relatively small number of shared parameters requires less training data than an equivalent fully-connected architecture. Using pre-trained word embeddings further simplifies the task, as these already carry some information about the meaning or syntactic role of words.

More sophisticated architectures can be built using 1D CNNs, although not used in our experiments. Most notably, several layers of convolutions (each optionally followed by pooling) can be stacked, where the following layer works on the feature maps output by the previous layer. The model described above uses a stride and dilation of 1, but multi-layer architectures where the higher layers have use exponentially increasing dilation are able to expand the receptive field of the model, and as a result aggregate "contextual information from multiple scales" [23]. Dilated convolutions were beneficial for semantic segmentation of images with 2D CNNs, but this increased receptive field has been useful for sequence-to-sequence models in NLP [22].

Recurrent Neural Networks

Recurrent neural networks (RNNs) are a family of models for processing sequential data. The sequence of inputs $x^{(1)} \dots x^{(t)}$ in our case are vectors of word embeddings, but there are other options for input representations (e.g. sequence of 1-hot encoded tokens, or vectors representing a multivariate time series at a given time step t). Vanilla RNNs maintain a hidden state vector h which contains some information about the sequence it has seen so far, and is calculated from the input at the current

time step t, and the previous hidden state:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}) \tag{2.3}$$

Vanilla RNNs suffer from the vanishing and exploding gradient problem in long sequences and are notoriously difficult to train. Probably the most widely used variant, the long short-term memory networks (LSTM) [12] overcome this limitation by augmenting the network with a memory cell c; there is a learned gating mechanism that controls what part of (and the extent to which) the cell is forgotten, what gets persisted in the cell state, and what gets output at a given time step. An intuitive overview of the gating mechanisms is given by [17] and [19] describes well how this helps mitigate vanishing gradients. A simpler gating mechanism is offered by gated recurrent units (GRU) [6], which has fewer parameters and works well on simpler tasks.

We are interested in using RNNs as text encoders, though they are often used for sequence to sequence modelling, or for predicting an output at each time step. In the simplest case, the concatenation of the cell state c and the hidden state h at the last time step could be used as a representation of the sequence.

2.2.3 Image Models

2D CNNs are important historically - excellent results in computer vision revived interest and funding in deep learning - but also applicable to a wide range of problems. Computer vision CNNs tend to be complex models with millions of parameters and dozens (or even hundreds) of layers. It is very compute-intensive and time-consuming to search for new architectures, and the analysis of these is beyond the scope of this work. Our goal is to use a CNN that has a reasonable performance without incurring unreasonable overhead.

The general architecture of a 2D CNN is as follows. Patches of feature extractors are convolved over the x and y axis of the image; these are often small, e.g. 3x3xc patches that span all the c channels of the image ($c = 3$ for RGB images), and convolution could be strided. Former models used larger patch sizes, but stacking several smaller patches could achieve similar receptive fields while allowing for a larger number of non-linearities for the same number of parameters. One or more layers of convolutions would be followed by a (often max) pooling layer. Computer vision architectures tend to re-use the same sub-structure repeatedly: either the same kinds of convolutions followed by pooling, or the "network in a network" structure popularised by GoogLeNet [20]. After a number of such repeated structures, there are normally a few fully-connected layers followed by a softmax layer. In our case, the final layer would be a fully-connected layer of sigmoid activations.

Transfer learning has shown to be particularly useful in computer vision. Successful computer vision models require a notoriously large training dataset, which would be hard to come by for every distinct computer vision task. Instead, models are often initialised to the values of a model that has been trained on a large dataset such as Imagenet [], excluding only the last layer(s) of the model that has

learned features that are specific to the original problem. Representations learned at the lower layers are remarkably universal and useful for other tasks. Therefore pre-trained 2D CNNs can be used as feature extractors for various other tasks, or the models could be fine-tuned to learn representations that are even more useful to the new task.

2.3 Downstream Models

2.4 Item Similarity

2.5 Recommender Systems

Several methods have been proposed that use dense embeddings learned by deep models that are used as part of a recommender system. These are briefly described here to motivate our focus on models that obtain dense embeddings of products.

Models that deal exclusively with deep embeddings [7] require huge amounts of user-product feedback pairs, therefore the more interesting solutions are ones which work together with classical methods such as matrix factorisation (MF). MF finds - purely from the implicit or explicit feedback users five to items - vectors of latent factors that represent each user's preferences and each item's "qualities"; the inner product between a user's and item's latent factors gives a score of compatibility between the two. Deep models can augment the item latent factors by injecting the embeddings learned for another task The main difference in these methods is how the deep embeddings are merged into MF, and how the deep embeddings are obtained in the first place. In [21], stacked denoising autoencoders are used for unsupervised feature learning of item data, while [11] uses simply visual embeddings extracted from a 2D CNN pre-trained on ImageNet.

2.6 Unsupervised and Semisupervised Learning

This section explores ways in which unsupervised and semi-supervised learning can learn good feature representations and improve sample complexity. Generic semi-supervised methods such as self-training and co-training are not considered, as one of our goals is to learn good representations.

Deep **autoencoders** can be used to learn low-dimensional representations of inputs. Many variants exist, but the general pattern is to have an "hourglass-structured" neural network where the first half of the model shrinks the input, and the second half of the network reconstructs the original input. Activations in the middle layer correspond to a dense embedding of the input; the shrinking layers need to throw away some of the detail in the input, yet persists enough for the expanding layers to be able to reconstruct the original input with some fidelity. The embedding contains information that is mostly unique to the data point, while

the parameters of the encoding and decoding layers ensure that the reconstructed input is realistic.

It is common to add noise (Gaussian, dropout) to the input or the intermediate layers to increase resilience to noisy inputs and to prevent the autoencoder simply memorising each data point. If the bottleneck layer is unconstrained, it will use a wide range of values to represent of different inputs. However, we would like embedding for similar inputs to also be similar. Variational autoencoders (**VAEs**) impose a prior distribution on the values of the embedding, often a multivariate Gaussian distribution with a diagonal covariance matrix, and the model is regularised during training to respect this prior. In practice this means that there is an additional step to determining the embedding z of the input. There are two separate neural network layers from the bottleneck layer of the autoencoder: one for determining μ (the vector of means of the multivariate Gaussian), and one for determining σ (the vector of standard deviations of the multivariate gaussian). Given μ and σ, we can sample the final item embedding $z \sim \mathcal{N}(\mu, \sigma^2)$. Note that sampling of z is a discrete decision that would normally stop gradient from propagating past this step, but we can use the reparametrisation trick introduced by [16] that turns the discrete decision into a deterministic function of $z = \mu + \sigma\epsilon$, where $\epsilon \sim \mathcal{N}(0, 1)$ is a random auxiliary noise parameter.

2.7 General Practices in Machine Learning

We briefly list some techniques and approaches that could be used for many kinds of machine learning problems. Some of these will be used in our experiments, yet some may not be pragmatic due to long train times and time constraints.

Bootstrap sampling, where data points are sampled from training data with replacement, can be used to repeatedly train the same model, and to observe the variance of its predictions. Bumping can be used to train several models (of the same family) on bootstrap samples to move around in the model space, and will pick the model that best fits the training data. This helps it explore a wider selection of models, but given that the original training data often also included as one of the bootstrap samples, this method can still pick the model original model if it happens to have the best accuracy.

There are some versatile methods that can help regularise a neural network (deep or shallow), or to speed up convergence by improving gradient updates. L2 regularisation should always be used to decrease moral complexity and impose prior on the model parameters, which implies that very high and very low values are unlikely. Dropout can mitigate overfitting by preventing neurons from co-adapting, encouraging each to learn representations that are useful independently [18]; this has been interpreted as training and exponential (in the number of parameters) number of models and averaging their predictions.

gradient clipping layer norm

2.8 Combining Models

This section introduces common ensembling methods. Some of these (bagging, boosting) are expected to consist of weak learners or models from the same model family, while others (committee methods, BIC, stacked generalisation) or more appropriate for ensembles of strong models.

Bootstrap aggregation (a.k.a bagging) draws several bootstrap samples from the training data, trains are separate model per bootstrap sample, and averages the predictions of these individual models. This reduces the variance of predictions without increasing its bias and generally leads to more accurate predictors [3]. Boosting is another common meta-algorithm that iteratively trains an ensemble of weak models, such that the data points that incurred a higher error on the previous ensemble are weighted more heavily, and each new weak model is added to the ensemble with a weight proportional to the weak learners accuracy. A common boosting algorithm is AdaBoost [8].

We are interested in ways of combining models that are separately trained and combined into a single predictor. Simple solutions are voting or (weighted) averaging, where the weights can be found with k-fold cross-validation. A more appealing approach is stacked generalisation, where a meta-learner is trained on the outputs of the individual models. This meta-learner could be a relatively simple model, such as logistic regression or a decision tree, and the results of the meta-learner could be therefore quite interpretable.

One orthogonal approach to combining models is joint training of model components. All models described here are trained using stochastic gradient descent, so in principle it would be possible to learn the parameters of all components (1D CNN, 2D CNN, RNN, DNN, logistic regression) as part of a single optimisation loop. This would be an engineering challenge due to the different ways these model require their input to be represented and the large size of the model, and optimisation would surely be difficult. In our case this would be clearly an overkill, but it would be an interesting research problem to examine whether joint training has advantages over ensembling in cases where we have different views of the data (in terms of considering different input dimensions or processing the inputs differently). As mentioned in the Wide & Deep paper [5], the wide component of the model has to just handle cases where the deep component falls short, and can therefore be simpler than it would need to be in an ensemble; analogously, jointly training different kinds of models neural networks would allow each component to be simpler.

One of the motivations for picking mostly deep neural networks as our ensemble components was their transfer learning capability: they all produce a dense feature vector (embedding) from which separate logistic regressions predict the class. While it is useful to have separate embeddings for images, text and categorical variables, we would also like to have a compound embedding of each product. A simple concatenation of all individual embeddings could be a useful (albeit a quite high-dimensional) representation to be used in downstream models. Alternatively, we could use this concatenation as an input to a neural network, whose goal is to (1)

reduce its dimensionality by removing redundant information and (2) predict the output classes from this compressed embedding. The latter case could act as a kind of a (replacement for) the meta-learner, with the difference that it takes the penultimate (as opposed to the final) layer of each individual model as input.

2.9 Active Learning

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.

Note that the decision to consider each of the outputs as an independent binary variable in favour of mutually exclusive categories may later change. There are some benefits to this: in some cases categories are inherently ambiguous, and allowing overlapping categories would enable us to use the same model to learn other attributes of products which might be more subjective. For example, we might be able to train a model to distinguish certain styles within a category, which would not have clear boundaries. Predicting binary independent outputs means that when a product is given a label (that it belongs to a category X), this label does not immediately exclude it from other categories; with mutually exclusive categories labelling a product would immediately determine whether or not it belongs to any of the categories.

3.1.2 Class and Label Imbalance

The labels provided by the rule-based system are only positive: some products are labelled to belong to a given category, but many products that ought to belong to a category are not labelled accordingly - but their label still appears to be negative. This (which we call the label imbalance problem) only exacerbates the class imbalance problem (that most products do not belong to most categories). As a result the model trained on rule-based labels will certainly underestimate the likelihood of any product belonging to any category.

Still, most positive labels provided by the rule-based system are correct, which means the model should assign higher likelihoods to products that should actually belong to the given class. Our active learning strategy outlined in section 3.3.4 ensures that products around the decision boundary would receive a label - which during the initial rounds of active labelling would be overwhelmingly positive examples[1]. The first rounds of active labelling should therefore counterbalance the underestimating nature of the pre-trained model; as the model starts to make more "optimistic" predictions, the products near the decision boundary should become a more balanced mix of positive and negative examples.

Class imbalance may or may not be an issue after label imbalance has been accounted for. The worst case scenario is that our model continues to underestimate the likelihood of products belonging to classes, particularly if these are rare classes. False negatives are not much of our problem in our setting. There are millions of products that affiliate networks provide, and in any case we can show the average

[1] this is confirmed by our experiments, described in section [ref]

user a tiny fraction of those products, so it is okay if some are left out from some categories and their chance to be seen decreases. Conversely, false positives will appear unprofessional and reduce user satisfaction.

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[2]). 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[3], 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 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.

[2]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.

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

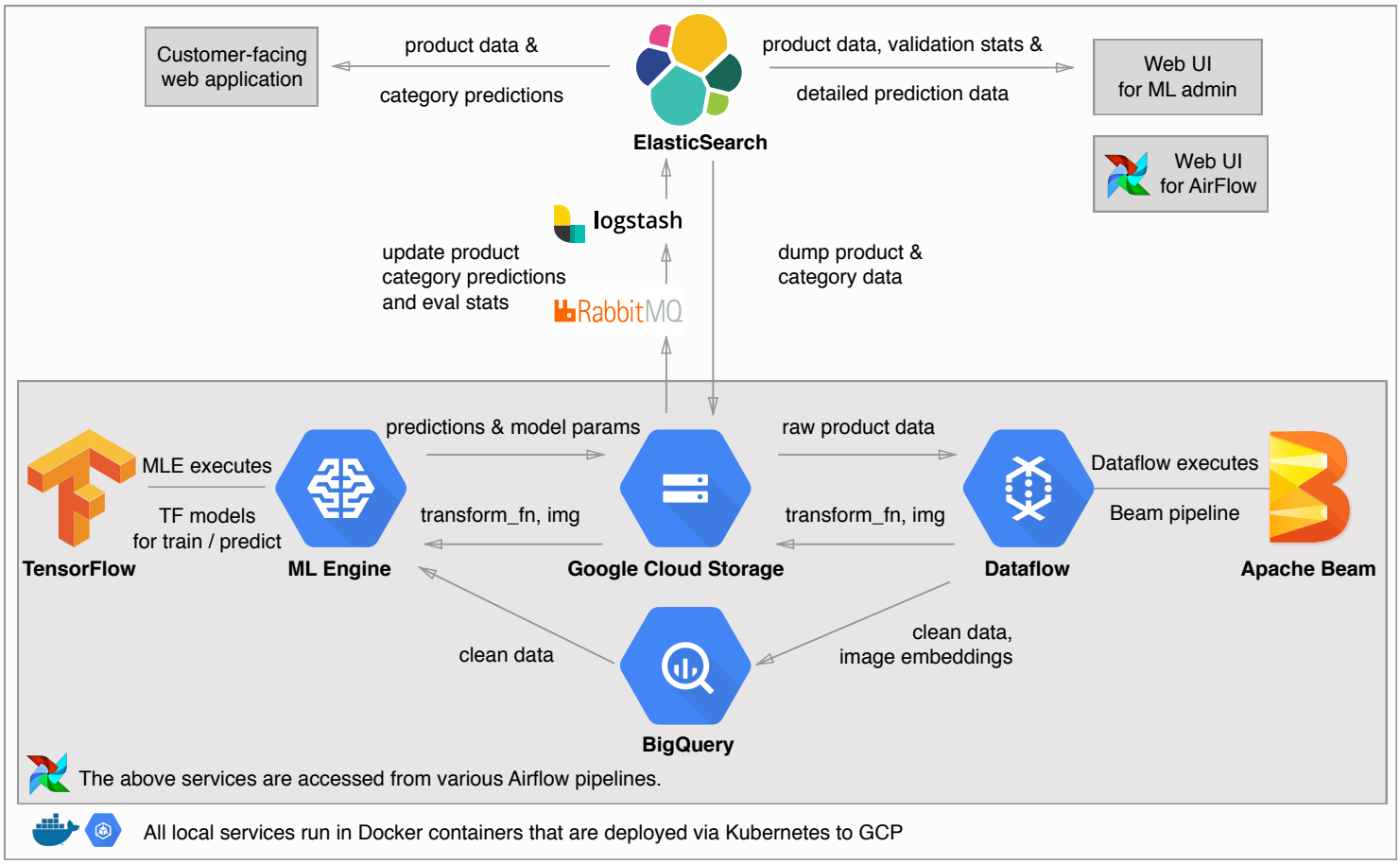

Figure 3.1. High-level system architecture of the ML pipeline

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[4]. 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).

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

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

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

[1] Sanjeev Arora, Mikhail Khodak, Nikunj Saunshi, and Kiran Vodrahalli. A compressed sensing view of unsupervised text embeddings, bag-of-n-grams, and LSTMs. In *International Conference on Learning Representations*, 2018.

[2] Richard G. Baraniuk. Compressive sensing. In *42nd Annual Conference on Information Sciences and Systems, CISS 2008, Princeton, NJ, USA, 19-21 March 2008*, 2008.

[3] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[4] Emmanuel J. Candès, Justin Romberg, and Terence Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *Information Theory, IEEE Transactions on*, 52(2):489–509, 2006.

[5] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. *CoRR*, abs/1606.07792, 2016.

[6] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[7] Ali Mamdouh Elkahky, Yang Song, and Xiaodong He. A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 278–288, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee.

[8] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of online learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[9] Surya Ganguli and Haim Sompolinsky. Compressed sensing, sparsity, and dimensionality in neuronal information processing and data analysis. *Annual review of neuroscience*, 35:485–508, 2012.

[10] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[11] Ruining He and Julian McAuley. Vbpr: Visual bayesian personalized ranking from implicit feedback. In *AAAI*, pages 144–150, 2016.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[13] Murphy Kevin. Machine learning: a probabilistic perspective, 2012.

[14] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[17] Christopher Olah. Understanding LSTM Networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, 2015. [Online; accessed 30-March-2018].

[18] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[19] Harini Suresh. Vanishing Gradients & LSTMs. `http://harinisuresh.com/2016/10/09/lstms/`, 2016. [Online; accessed 30-March-2018].

[20] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[21] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1235–1244. ACM, 2015.

[22] Zichao Yang, Zhiting Hu, Ruslan Salakhutdinov, and Taylor Berg-Kirkpatrick. Improved variational autoencoders for text modeling using dilated convolutions. *arXiv preprint arXiv:1702.08139*, 2017.

[23] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *CoRR*, abs/1511.07122, 2015.

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, April 1, 2018

.....
Mattias Arro

A. Screenshots

A.1 Dataprep Histogram

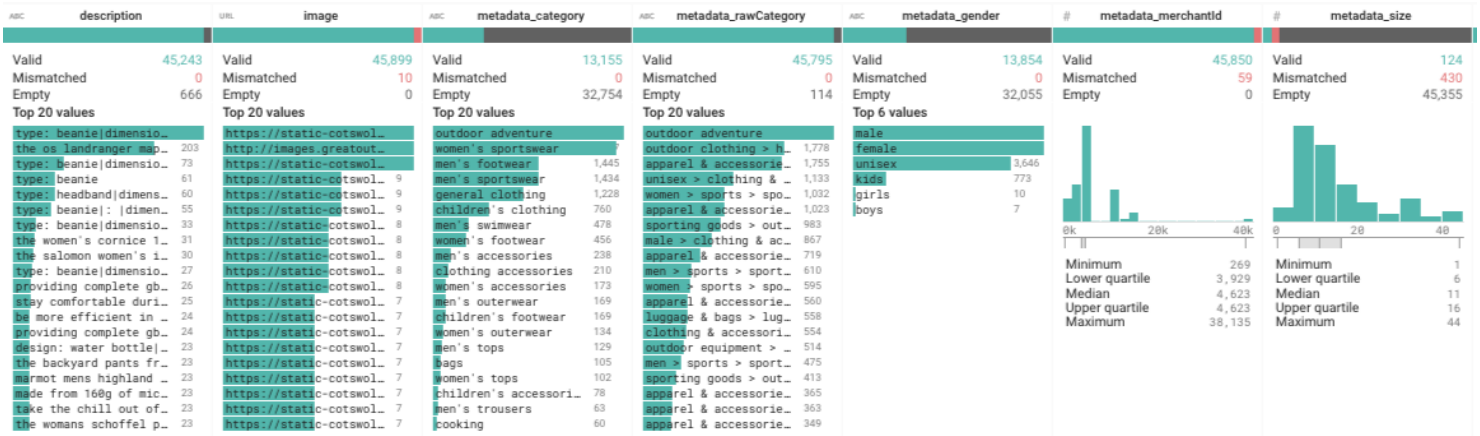

Figure A.1. Dataprep: the histograms of field values of a subset of fields.