KTH Information and
Communication Technology

Label-Efficient Multi-Objective Machine Learning for e-Commerce

Exploration of transfer, multi-task and active learning
across shallow and deep neural network architectures
for e-commerce product classification.

MATTIAS ARRO

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

2018

Contents

1 Introduction **1**
- 1.1 Problem . . . 2
- 1.2 Purpose . . . 2
- 1.3 Goals . . . 2
- 1.4 Hypotheses . . . 3
- 1.5 Ethical Consideration . . . 4
- 1.6 Sustainability . . . 5
- 1.7 Delimitations . . . 5
- 1.8 Outline . . . 5

2 Background **7**
- 2.1 Input Representations . . . 7
  - 2.1.1 Categorical Input . . . 7
  - 2.1.2 Text Input . . . 8
  - 2.1.3 Image Input . . . 9
- 2.2 Models Considered . . . 9
  - 2.2.1 General Models . . . 9
  - 2.2.2 Text Models . . . 11
  - 2.2.3 Image Models . . . 13
  - 2.2.4 Downstream Models . . . 13
- 2.3 Unsupervised and Semisupervised Learning . . . 15
  - 2.3.1 Autoencoders & Variational Autoencoders . . . 15
  - 2.3.2 Conditional Variational Autoencoders . . . 16
  - 2.3.3 Generative Adversarial Networks . . . 17
- 2.4 Combining Models . . . 18
- 2.5 Active Learning . . . 19
- 2.6 General Practices in Machine Learning . . . 21
  - 2.6.1 Hyperparameter Tuning Using Bayesian Optimisation . . . 21

3 Method **23**
- 3.1 Data Preprocessing . . . 23
  - 3.1.1 Category Structure . . . 24
- 3.2 System Architecture . . . 25

3.2.1 Airflow Pipelines & Pipeline Runs 27
3.2.2 Dataflow Pipelines 28
3.2.3 TensorFlow and ML Engine 31
3.3 Experiments, Evaluation & Results 32
3.3.1 Visual Similarity 32
3.3.2 Independent Models 34
3.3.3 Multi-Objective Training 42
3.3.4 Active Learning 43

4 Discussion **45**
4.1 Visual Similarity 45
4.2 Baseline Model 45
4.3 Independent Models 45

5 Conclusion **47**

References **49**

Declaration **55**

Appendices **55**

A Screenshots **57**
A.1 Dataprep Histogram 57

1. Introduction

Machine learning (ML) has recently become a popular research area, and is increasingly applied in industry. A lot of this newfound interest and hype is directed at neural networks and deep learning. This focus is not unfounded - approaches based on deep networks continue to break benchmarks in core machine learning research areas such as computer vision [10, 43, 3], speech recognition and synthesis [20, 54], many natural language processing tasks [55, 60, 16], game play [48, 39], and even enabling novel applications such as style transfer [15] and content generation [18].

Artificial neural networks consist of layers of transformations that map multidimensional inputs to (usually multidimensional or structured) outputs. A single-layer neural network is a linear transformation of the inputs; each additional layer with a non-linear activation function enables the network to partition the output space and hence approximate more complex functions. Complex models are prone to overfitting and require more training data and heavier regularisation, yet deep models are somewhat unique in that their performance continues to increase with the size of the dataset, while the benefits of more data taper off for other models.

The naive conclusion is to use deep learning only when labeled data is abundant, but the depth of a model is not a binary "deep vs shallow" decision - one can start out with a shallow model and increase model complexity to see the effects on performance and generalisation. Logistic regression might well be the most appropriate model for a classification problem, but this is often not obvious up front, so it is beneficial to build models in a framework that also supports deep models. In addition to easy experimentation with model architectures, it is easy to do transfer learning with deep models on popular platforms: image or text models pretrained on large datasets can be utilised to increase product classification performance, and representations learned for classification could in turn be used by a downstream product recommender system. Datasets with abundant unlabeled data can benefit from unsupervised and semisupervised learning using deep generative models.

Each project has different requirements on predictive performance, label complexity, interpretability, computational resources and engineering challenges. Neural networks are a good fit for this project due to their amenability to transfer learning and flexibility in handling multi-modal inputs and multi-task objectives. As neural networks are notoriously data-hungry, the question immediately becomes - how to do it without providing many labels.

1.1 Problem

The client company gathers data from various affiliate networks (that in turn get their data from various retailers) and displays it on their website. There are millions of products belonging to roughly 1300 categories, and categories follow the usual nested tree structure. The incoming data is noisy and inconsistent: what kind of data is stored in which 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 fail 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 to replace the old way of categorising products. The predicted outputs can either be a set of independent binary classifications, or the category structure has to be re-organised to ensure classes are mutually exclusive. 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. The highest priority is low label complexity, beating requirements for high accuracy and interpretability. The system should be robust to noisy inputs; data preprocessing should not consider the idiosyncrasies of each affiliate 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 combinations, 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 sim-

ilar products.

- Build an interface for subjectively evaluating the visual similarity algorithm.
- Train baseline model to reproduce the behaviour of the rule-based system.
- Train and evaluate a number of different models on the rule-based labels.
- Define a subset of the 1300 independent categories to be exclusive (non-overlapping). Build an interface for assigning **exclusive labels** to products either through **uncertainty** or **random** sampling.[1]
- Implement the active labelling mechanism defined in 3.3.4 and train a small selection of well-performing models in the following 6 settings:
  - One training objective: **exclusive**.
  - Two training objectives: **exclusive** and **rule-based**. In addition to human-assigned exclusive labels, the model is trained on the original 1300 independent categories assigned by the rule-based system. [2]
  - Both training objectives are trained in three settings: exclusive labels are either limited to random or uncertainty sampling (to determine the efficacy of active learning), or not limited (labels from random and uncertainty sampling are combined to have the largest training set).
- Document the results as well as the technical architecture and workflow.

1.4 Hypotheses

The following hypotheses were postulated prior to running most experiments[3]:

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 ones.
3. Deep models with histogram inputs do not improve substantially on linear models.
4. Shallow models with embedding inputs generalise better, and deep models with embedding inputs generalise slightly more.
5. Pre-trained 2D CNN are consistently good predictors of clothing products, yet fail to accurately predict categories such as technology.

[1] The final version of this interface was built by an employee of the client company, though earlier versions for displaying the results of classification / item similarity were implemented by the author.

[2] We call this the rule-based rather than independent objective, as we are planning to train with independent training targets other than the rule-based labels at a later stage in the project.

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

6. Active learning substantially decreases label complexity, however uncertainty sampling might give poor results on the first round.

How these hypotheses were evaluated is described in the relevant subsections of 3.3.

1.5 Ethical Consideration

Given how pervasive machine learning is becoming, it is crucial that there is discussion about the safety, transparency and bias of machine learning systems. There have already been cases of black box algorithms being used to make decisions that severely influence a person's life - predicting reoffending probability of convicts to determine whether they are given parole - 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 convicts [29]. Even though the data did not contain explicit information about race, the algorithm was capable of implicitly detecting race 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 consistently 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.

The worst case scenario for inaccurate, opaque or biased product classification is embarrassment for the client company, therefore ethical considerations are not of much concern in our case. It is still worth reminding the reader that machine learning models always contain some kind of bias: from the data it uses as input (the way it is gathered, what is gathered), the way the data is processed, and the kinds of models used. One way to quantify the fairness of a binary classifier the p% rule, which sets bounds on the ratio between the probability of the outcome given the sensitive attribute [63]. One way to overcome a classifier that seems to rely on some "nuisance parameter" (such as gender or race - even when the parameter is removed from the training data) is adversarial training [63] to ensure the model's output distribution does not depend on the sensitive parameter.

A related concern to fairness is privacy: ensuring that a ML model does not leak its training data to an attacker that might orchestrate a large series of queries on the model. Each individual query might not reveal much, but with a large number of queries the attacker can see how the statistics of the output distribution changes; paired with an external dataset for cross-referencing or knowledge that a data point was added to the training set (e.g. someone answered a questionnaire), it is possible to recover some of the original data (though not necessarily with high fidelity or probability). A countermeasure is differential privacy, which can

be applied to the data sharing process or at the algorithm level. Algorithms that are differentially private inject noise at the input, model or output level to ensure that the output distribution of the model does not change substantially with the inclusion or exclusion of a data point.

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 variab

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.

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 [6]. 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 is 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 [14] 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 high fidelity. 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)"[14]. The projection is random because the weights for the sum of pixel intensities are

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

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, like 1-hot encoding of categorical variables. Text can also be represented as a sequence of **word embeddings**. Word embeddings can either be random vectors or representations learned with word cooccurrence algorithms such as word2vec [38] and GloVe [41]; these representations can remain fixed throughout the learning procedure, or updated as part of the stochastic gradient descent (SGD) when applicable. A simple 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]. In [37], paragraph vectors are created in a similar manner to word2vec vectors by relying on these vectors to predict the next word in a sentence.

More complex methods use recurrent neural networks (**RNNs**) to combine a sequence of word embeddings into a fixed length vector (see 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 might 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 nontrivial 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.

2.2.1 General Models

Logistic Regression

Logistic regression is a linear, discriminative classifier that is a common baseline model, 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 [31].

We have two cases: for independent categorical outputs we use a separate binomial logistic regression model for each output class, and for mutually exclusive classes we use multinomial logistic regression. This is shown formally in eq 2.1 and 2.2, where where X and W correspond to the input and weight matrices, respectively.

We follow the notational trick where the first row of X is always 1, and the first row 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.

$$p(y|X, W) = \text{sigmoid}(W^T X) \tag{2.1}$$

$$p(y|X, W) = \text{softmax}(W^T X) \tag{2.2}$$

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^*. 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 [17]. In our case, the input is some representation of categorical, text or image input - often an embedding extracted with another model or assigned randomly. 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. Similarly to logistic regression, we have two cases: sigmoid activation for independent categories and softmax layer for mutually exclusive categories.

Wide & Deep

Wide and deep model was originally proposed for recommender systems [8], but can just as well be used for classification. In this model a deep network and a linear logistic regression model are trained jointly in the same SGD learning process, rather than trained separately and then ensembled. The wide component is good at remembering "feature interactions through a wide set of cross-product feature transformation" while the deep model provides good generalisation. In our experiments we did not use feature cross products of features.

Loss Function and Optimizers

Logistic regression and neural networks are optimised with some variant of gradient descent. The loss function was the same for multi-label and multi-class training objectives: binary cross-entropy between the training data and the model distribution, with the loss value averaged across all classes. This is given in eq. 2.3, 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.

$$NLL(\theta) = \sum_{c=1}^{C} \sum_{i=1}^{N} [c_i \log P(c = 1|x, \theta) + (1 - c_i) \log(P(c = 0|x, \theta))] \tag{2.3}$$

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 [33] optimiser works very well with default hyperparameters, and is therefore used for all stochastic gradient descent updates.

Cross-entropy is a standard loss function for multi-class and multi-label classification, however in our cases it may have downsides. There is at times ambiguity as to which category a product should belong, therefore even hand-assigned labels are somewhat arbitrary. The similarity between categories is ignored by this loss function: a short coat that is misclassified as an earring has the same loss value as a coat that is classified as a jacket. Additionally, the rule-based labels are incomplete: the lack of a label does not always mean that the product does not belong to the given category, yet the model gets penalised for cases where such label is missing but the model correctly predicted a semantically similar class.

The Wasserstein distance metric (also called earth mover distance) can be used instead, which gives the cost of the optimal transport plan for moving the mass of one probability distribution to another. In our case the two distributions are the label distribution of a data point (1-hot / k-hot vector) and the class distribution predicted by the model. The optimal transport plan is weighted by a KxK distance matrix, where K is the number of classes and the entries in the matrix correspond to how dissimilar the classes are[3]. Intuitively, transporting probability mass from the category "Coats" to "Earrings" should be more expensive than from "Coats" to "Jackets".

2.2.2 Text Models

1D Convolutional Neural Networks

A simple convolutional architecture for text classification is described in [32]. 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

[3]Similarity is known *a priori*, e.g. derived from WordNet hierarchy. It would be interesting to explore similarity matrices created using cosine distance of embedding vectors of categories; embeddings could be extracted with an RNN from the description of the category

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 [17]), 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" [61]. 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 [59].

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:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}) \tag{2.4}$$

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) [24] 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 [40] and [50] describes well how this helps mitigate vanishing gradients. A simpler gating mechanism is offered by gated recurrent units (GRU) [9], 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 [51]. After a number of such repeated structures, there are normally a few fully-connected layers followed by a softmax or sigmoid layer.

Transfer learning is particularly applicable to computer vision. Successful computer vision models require a notoriously large training datasets, which would be hard to come by for every problem. Instead, models are often initialised to the values of a model that has been trained on a large dataset such as ImageNet [44], 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.

2.2.4 Downstream Models

Item Similarity

Similarity measures based on tokenised text are easily accessible to e-commerce companies: the popular open source NoSQL database ElasticSearch (which makes use of Lucene, a full-text search engine) provides document similarity metrics that are some combination of normalised TF-IDF vectors between all documents and a query (which could be another document). This gives decent results in many cases, but it does not take advantage of a product image.

Embeddings extracted with a deep network can be viewed as a distributed representation of the original data, which carries semantic meaning [23]. In the contrived examples of distributed representations, each dimension would represent the degree to which a property (e.g. of shape or colour) is present of an object, however the dimensions are not necessarily so nicely disentangled in the representations learned by deep networks[4]. Still, it is clear that such dense representations help linear models do accurate predictions, and semantically similar data points will have similar embedding vectors.

Therefore we can compare any two embedding vectors (extracted with the same neural network) using standard vector space similarity measures, such as cosine similarity (1 - normalised dot product between the vectors). Such embeddings can be extracted from product images with 2D CNNs pre-trained on another task, 1D CNNs or RNNs on product title and descriptions, or from the joint embeddings described in section 2.4. With large numbers of products, calculating pairwise similarity scores becomes intractable, therefore approximate nearest neighbour methods could be used [4].

Another interesting method of computing similarity scores between documents utilises the full-text search capability of tools such as Lucene, which would be valuable to firms already using such technologies. The embedding vectors of products could be tokenised, by converting each dimension of each embedding vector into tokens that represent the feature with some precision [45]. For example, we could consider two levels of precision and divide the normalised feature into 10 intervals of size 0.1 and 100 intervals of size 0.01; each product would get two tokens per embedding dimension[5]. For example, if the first embedding dimension is 0.46, the inserted tokens would be d1_0.4 and d1_0.46. Full-text search engines would assign higher scores to co-occurrences in the more precise tokens and lower scores to co-occurrences in more coarse token.

Recommender Systems

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

Models that deal exclusively with deep embeddings [12] 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 give to items - vectors of latent factors that represent each user's preferences and each item's "qualities"; the inner

[4]The author has not seen strong evidence in literature to support either case, but at least in [26] extra steps had to be taken to *prevent* certain latent variables from encoding properties of the data that were designed to be captured by other latent variables. This implies that by default, each latent variable contains a little bit of information about each aspect of the input, and that the whole pattern of the embedding matters for conveying semantic information

[5]It would be a good idea to also have some wider overlapping intervals as well, e.g. 0 ... 0.5

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 [56], stacked denoising autoencoders are used for unsupervised feature learning of item data, while [21] uses simply visual embeddings extracted from a 2D CNN pre-trained on ImageNet.

2.3 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 are not considered, as one of our goals is to learn good representations, but also because these methods can reinforce poor predictions or do not make full use of all available unlabelled data.

Some generic methods can still improve the representations learned by our models when applied to models that learn deep embeddings in order to make predictions. Entropy minimisation [19] can be incorporated into models trained with SGD to encourage confident predictions of each class; this encourages the deep model to learn predictions that are strong predictors of some class and avoid producing features that produce mixed predictions. More complex but very performant approach is Mean Teacher [53]. A student that gets a harder task (such as predicting from a noisy/adversarial example), and a teacher that gets an easier task (i.e. the teacher model is an ensemble, which is more accurate). The student is trained on the prediction of the teacher; the teacher's parameters are an exponential moving average of the student's, updated after each minibatch. The teacher's predictions are higher-quality than the student's (and can be applied on unlabelled data), while the student tries to continuously learn to learn a predictor that is robust to noisy and adversarial examples.

2.3.1 Autoencoders & Variational Autoencoders

Deep autoencoders (AEs) 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 sample; 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 different inputs. We would like embedding for similar inputs to also be similar, which is not always the case for ordinary autoencoders. 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. The model is optimised to minimise the reconstruction loss and KL divergence between the model's distribution z and the prior on it, which can be computed just from μ and σ if the prior is a isotropic standard normal. Enforcing the prior also means that the embeddings z will occupy a smooth, contiguous space, which allows us to draw samples from the prior $\epsilon \sim \mathcal{N}(0, 1)$ and use that as the input to the decoder - this gives us a generative model, which would not be possible with ordinary autoencoders. The output of the VAE is also probabilistic: e.g. for images, the output for a pixel would be a Gaussian distribution, which is sampled the same way as the Gaussians for z.

The VAEs described here are probabilistic models parameterised by neural networks for approximating the true posterior $p(z|x)$, since the exact posterior is intractable. In practice, VAEs add an extra loss term (KL divergence) to AEs, and additional step to determining the embedding z. There are two separate neural network layers from the bottleneck layer of the AE: 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 [35] 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.

2.3.2 Conditional Variational Autoencoders

It is easier for linear models to learn from embeddings extracted with AEs, and embeddings from VAEs make the classes even easier to separate. These are unsupervised methods that do not take advantage of labels in the training data. In the semisupervised approach introduced by [34] there are two inference networks: a discriminative classifier that outputs a categorical distribution from the input x, and a class-conditional encoder that takes as input both x and the 1-hot encoded categorical label y[6]. The model is trained on both labelled and unlabelled data. For labelled examples, the x and y are given as input to the model, which embeds it in z as described earlier, and reconstructs the original x. The value of y is unknown for unlabelled data points; therefore the model is run $|y|$ times, encoding and decoding the data point conditioned on each possible class i; loss from these runs is averaged, weighted by the discriminator's estimate of the sample belonging to class i. This approach is acceptable for small numbers of classes, but is impractical for many

[6]class-conditional encoder means that the encoder is aware of the class of the data point, i.e. the output distribution z is conditioned on the class y in addition to the original input x

multi-class problems. A solution uses the reparametrisation trick mentioned above: rather than running the procedure once per class, we take a Gumbel-softmax [28] to get a discrete estimate of y that is still differentiable.

Such class-conditional VAEs can be used in any situation where unlabelled data is abundant but labels are scarce. The discriminator's loss is optimised as part of the VAE's loss function, which means adding more unlabelled data can improve the accuracy of the discriminator. The approach was developed and tested on the MNIST dataset, which consists of 28x28 grayscale images - a very simple dataset by today's standards. Some reports have emerged that fail to reproduce this success on more complex datasets such as CIFAR-10, being outperformed even by PCA [30].

One related approach is offered by [26], where a class-conditional VAE is used to generate synthetic data for a discriminator network, which has reportedly a higher accuracy in semisupervised setting than the approach described just above. They use it for conditional text generation, but this approach of using the class-conditional VAE to synthesise training data to train a discriminator is applicable to any input modality. In fact the general approach to "dreaming up" new training samples in the *sleep phase* and training the classifier in the *wake phase* was introduced already in [22]. The current author has implemented this model and open sourced it[7].

2.3.3 Generative Adversarial Networks

Generative adversarial networks (GANs) are an interesting approach capable of synthesising realistic data, but also useful for learning good representations of data. GANs have been most successful for image synthesis, but are in fact applicable to any kind of input data including text [62] and even mixed data types such as e-commerce orders [36]. In the GAN setting, there are two networks: a generator that tries to produce realistic synthetic samples, and a discriminator whose goal is to distinguish between synthetic and actual samples. Given that the training is stable enough, both the generator's and the discriminator's performance improves with time, resulting in more realistic synthetic samples, as opposed to the relative blurry images produced by VAEs. The discriminator needs to learn good representations of the input to accurately distinguish which samples come from the true distribution, and which samples are synthetic; as with many CNNs for computer vision, these representations can be useful for other tasks as well.

It has been shown that linear models from the embeddings learned by the discriminator outperform other unsupervised feature learning techniques such as k-means [42]. At the time of writing, variants of Wasserian GANs (WGANs) achieve the best performance by improving training stability and preventing the generator from generating samples from a limited number of modes; this is achieved by minimising the Wasserian distance between the generator's and real data distributions, as opposed to minimising the JS divergence as was common before. A performant

[7] https://github.com/mattiasarro/seq2seq-cvae-tensorflow

variant of WGAN is CT-GAN, which adds a regularisation term to enforce a Lipschitz continuity condition over the manifold of the real data [58].

2.4 Combining Models

This section introduces common ensembling methods, describes joint learning and multi-objective training. 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 a 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 [5]. 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 [13].

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 [8], 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.

A related idea is **multi-objective learning**, where a neural network is trained to predict multiple aspects about the training data. In our setting we can consider the rule-based multi-label objective and the hand-assigned exclusive labels as separate training objectives; we might add additional training objectives that predict colour (multi-label) or conversion (binary). When using feedforward networks for multi-objective learning, the lower layers are usually shared among the objectives, and the top layers are specific to each training objective. It has been shown repeatedly that when the training objectives are somewhat related, then adding an additional objective increases the prediction accuracy across the other objectives. When implementing multi-objective training, care must be taken not to influence the parameters that are specific to another objective.

2.5 Active Learning

This section describes the main approaches to active learning; details analysis is beyond the scope of this report.

The goal of active learning is to reduce the number of labels needed to effectively train a machine learning model by being selective about which data points to label. The three general scenarios: membership query synthesis, stream-based selective sampling, and pool-based sampling [46]. In the first scenario, the algorithm synthesises a datapoint and asks a label for it. In stream-based selective sampling, and instance is sampled and the model decides (while having access to the data point's features) whether to acquire a label or not. In pool-based sampling, the model chooses which data point to label from the pool of all unlabelled samples. Our use case is a version of pool-based sampling, where the algorithm picks out a batch of products to be labelled. The rest of the section describes different approaches to picking the data points that are expected to help the model learn.

Uncertainty sampling is the most popular choice: if the model is uncertain about a prediction, obtaining a label for that data point would help it differentiate between different targets. The most common way to quantify uncertainty is by entropy of the model's predictions:

$$x^*_H = \underset{x}{\operatorname{argmax}} - \sum_i P_\theta(y_i|x) \log P_\theta(y_i|x), \tag{2.5}$$

where x^*_H corresponds to the most informative instance according the entropy (H) measure, and y_i corresponds to each class.

Query by committee uses a number of models trained on the labelled data which all make a prediction on the unlabelled data. Data points with highest disagreement are expected to be most informative, as by definition a larger number of models would have to be wrong in their predictions. The models in the committee do not have to be of different type like is our case, but could be e.g. a set of linear classifiers where each committee member just has different parameter values, yet each member would have to be consistent with the current set of labels (not contradict it with its predictions). Disagreement between committee members can be quantified with vote entropy [11]:

$$x^*_{VE} = \underset{x}{\operatorname{argmax}} - \sum_i \frac{V(y_i)}{C} \log \frac{V(y_i)}{C}, \tag{2.6}$$

where $V(y_i)$ is the number of votes given to class i and C is committee size. Alternatively, KL divergence between each committee member's predictions and the consensus P_ζ (average prediction of committee members) could be used:

$$x^*_{KL} = \underset{x}{\operatorname{argmax}} \frac{1}{C} \sum_i D(P_{\theta^{(C)}} \parallel P_\zeta) \tag{2.7}$$

$$P_\zeta(y_i|x) = \frac{1}{C} \sum_{c=1}^{C} P_{\theta^{(C)}}(y_i|x) \tag{2.8}$$

$$D(P_{\theta^{(C)}} \parallel P_\zeta) = \sum_i P_{\theta^{(C)}}(y_i|x) \log \frac{P_{\theta^{(C)}}(y_i|x)}{P_\zeta(y_i|x)} \tag{2.9}$$

Expected model change calculates how much a model would change if we knew its label. In gradient-based learning algorithms it is possible to compute the L2 norm of the gradient vector for each combination of labelings of the unlabelled product [47], which is a direct measure of how much the model would change.

Expected error reduction considers how much the test set error is likely to decrease with the acquisition of a given label. This is done by re-training the model once per each combination of the label values and observing how either the risk or expected log loss of the unlabelled data points decreases. This is not a suitable use case for deep networks, which are trained over several epochs of the data; there is no efficient way to incrementally train the model, and when re-training with a single additional label, the change in the model is probably higher from the stochasticity in the training procedure rather than the additional label. In any case, this method is very expensive computationally.

Variance reduction tries to pick data points which are expected to decrease the variance in predictions. To do that, the Fisher information of the model parameters should be maximised. Unfortunately this involves inverting a $K \times K$ covariance matrix, where K is the number of parameters in the model. This is impractical for deep networks with millions of parameters.

Information density measures consider data points that are not just uncertain but also representative of the underlying distribution. This avoids the problem with many other approaches that ask labels for samples that are controversial, but could otherwise be outliers and not the most useful for good generalisation. In this framework, a base informativeness measure $\phi_A(x)$ such as uncertainty or disagreement is weighted by the average distance of the data point x to other data points in the distribution:

$$x^*_{ID} = \underset{x}{\operatorname{argmax}}\, \phi_A(x) \times \left(\frac{1}{U} \sum_{u=1}^{U} sim(x, x^{(u)}) \right)^{\beta}, \tag{2.10}$$

where sim is some similarity function and β controls the importance of the density term.

2.6 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, 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 [49]; this has been interpreted as training and exponential (in the number of parameters) number of models and averaging their predictions.

Batch normalisation is known to stabilise training and hence speed up convergence by normalising each input to have unit variance and zero mean [27]. gradient clipping layer norm

2.6.1 Hyperparameter Tuning Using Bayesian Optimisation

3. Method

Several hypotheses were tested in this work. Section 3.1 gives an overview of how input data was pre-processed. 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 in section 3.3.

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. Exploring the predictions subjectively.
- Building a visual similarity feature and evaluating its results subjectively.
- Training the independent classifiers to determine best performers (3.3.2). Due to the good perforance of individual models, ensembling was not attempted due to the engineering overhead it would incur. Experimenting with different multi-objective training approaches.
- Training [TODO] 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. 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 a training objective for which 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 or 1280 dimensions extracted with a 2D CNN

3.1.1 Category Structure

At the time of writing, there were roughly 1300 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.

Categories can be considered as independent (multi-label) or mutually exclusive (multi-class). The common way to handle this is to assume categories are mutually exclusive. With exclusive categories, assigning a label to a product determines its label across all categories, whereas with independent categories a positive label

only determines the label across the category in question (and its ancestors in the category tree) - that is roughly a thousand-fold difference in labelling efficiency. The prediction task is also easier for exclusive classes - rather than needing to predict a probability score above a threshold separately for each class, the model would need to just assign the highest probability to the correct class compared to other classes.

On the other hand, some categories at the client company are inherently ambiguous or even overlapping. The rule-based labels are also independent, since a product may be labelled to belong to zero or many categories. Independent categories are also somewhat easier to handle: with exclusive categories the rule-based and hand-assigned labels would have to be considered as separate training objectives, because the activations from softmax and sigmoid layers that lead to a positive prediction will be different and we can not share the output layer across these two types of labels.

Bearing in mind the considerable efficiency gain in labelling, we mark a subset of the categories (1063) as "exclusive". Exclusive categories are usually leaves, but at times a higher-level category was marked as exclusive, denoting the more general case[1]. Therefore we have two training objectives: a multi-label prediction of independent categories labelled by the rule-based system, and a multi-class prediction of exclusive categories labelled by the employees of the client company - respectively called the **rule-based** and **exclusive** training objectives. Note that if training on the exclusive objective turns out to have downsides, it is possible to revert to considering each category as independent, and to use all the labels assigned with the assumption that the category is exclusive, while the converse is not true.

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.

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

[1] e.g. if "Books", "Books > Travel Books" and "Books > Cookbooks" are all marked as exclusive, then the semantics of "Books" is actually "Books that are not about travel or cooking". This means some categories are not strictly exclusive, but we had to work around the existing category structure.

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.
- 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 JavaScript 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 labels, top 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

[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.

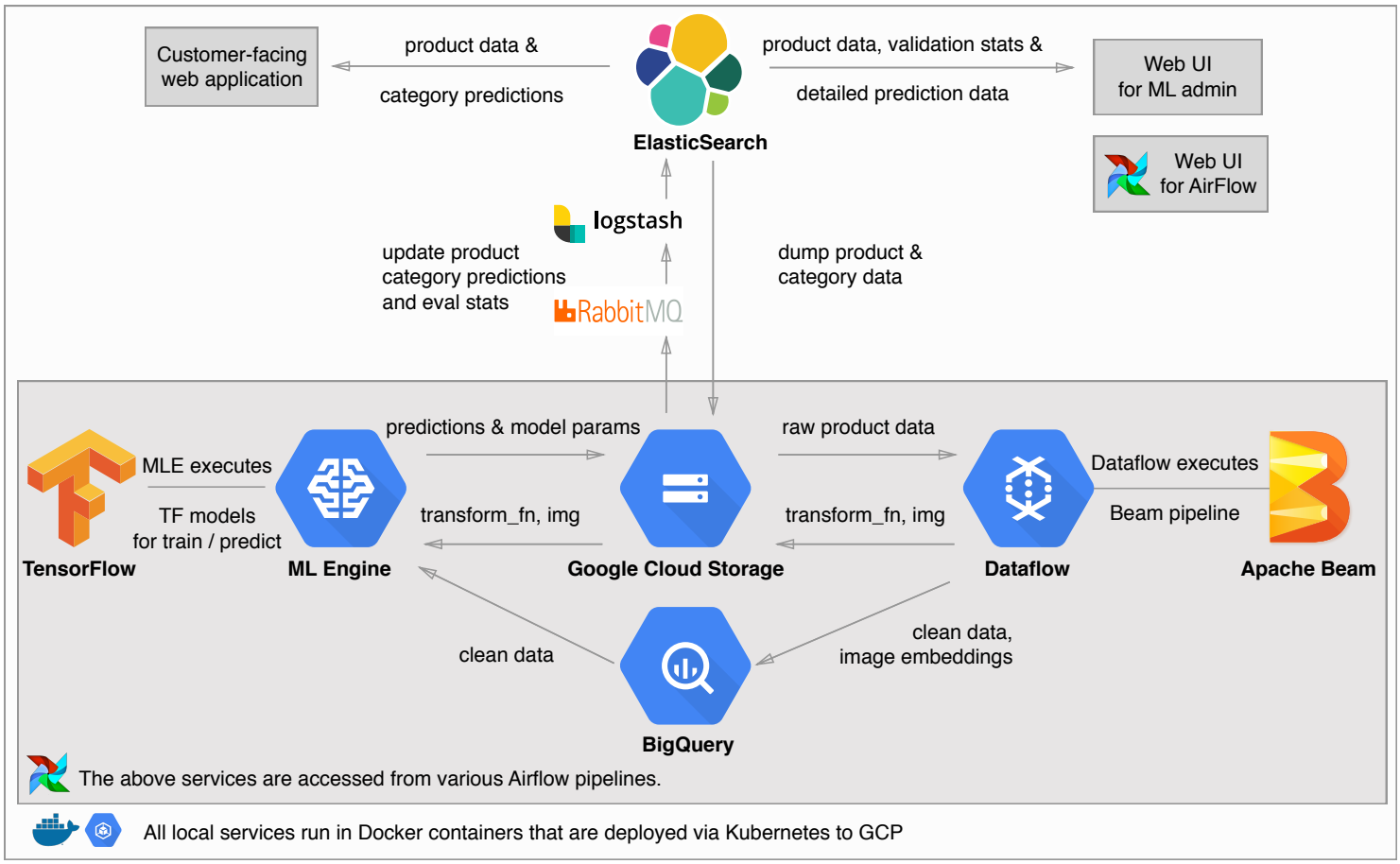

Figure 3.1: High-level system architecture of the ML pipeline

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 by network latency as well as computing the appropriate category path for each product (explained in 3.1.1).

3.2.1 Airflow Pipelines & Pipeline Runs

A group of tasks that would need to be run together repeatedly is encapsulated in an Airflow pipeline, which is a directed acyclic graph (DAG) of tasks (also referred to as an AF pipeline). A DAG is run many times, either based on a schedule or triggered manually. Figure 3.2 shows a screenshot of the most common Airflow

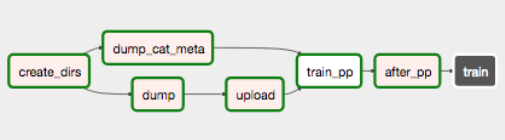

Figure 3.2: Airflow DAG: preprocess, train specific model, update statistics. All tasks except for "train" have completed, which is not yet scheduled for execution.

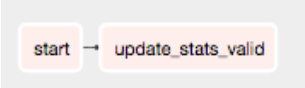

Figure 3.3: Airflow sub-DAG: train. No task has been scheduled for execution during this run.

pipeline "cat_pp_train_specific", which is short for "categorisation: preprocess and train a specific model"; figure 3.3 shows the sub-DAG of the task "train".

Most of our pipelines start by creating directories for our files. Which files are stored where is configured in a simple file that is accessed by Airflow, Dataflow and TensorFlow (locally or inside MLE); the configuration has placeholders for variables that are determined per run or are specific to a task inside a run. For example, the location of .tfrecords files is determined by "*:prefix*/runs/*:run_id*/tfrecords/*:objective_:split*.tfrecords", where *prefix* corresponds to the path to the data directory (either locally or in GCS), *run_id* is specific to the current run of the pipeline, *objective* is either *rule_based* or *exclusive*, and *split* is either *train*, *test* or *valid*. This ensures that all parts of the system look for the same file in the same place, and makes building file paths easier. All file access in the different parts work equally on a local file system and GCS, which makes it particularly convenient to switch between local experimentation and remote training; in fact, the *prefix* parameter is by default read from an environment variable, which is different in local, Docker and cloud service environments.

A training dataset is identified by its *run_id*. The processed .tfrecords, transform function (described below) and metadata is persisted under that run's directory. Several models can be trained from the same dataset. The checkpoints of a model are saved under "*:prefix*/runs/*:run_id*/checkpoints/*:hparams_id*/", where *hparams_id* identifies a model architecture, as well as what kinds of training objectives and label types it is trained on; during hyperparameter tuning, a unique index is appended to the *hparams_id* of the model. As *hparams_id* encodes only the general model architecture and training objectives, the full set of all hyperparameters is persisted along model checkpoints as a JSON file - for future reference.

3.2.2 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). Figure 3.4 shows a screenshot of this pipeline, as visualised by the GCP UI.

In this pipeline, 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. The tokenised data is stored to GCS as a sharded .tfrecords files (with one set of files per training objective), a binary file format that TF Transform helps produce; alternatively the raw data could be saved (e.g. as JSON or inside BigQuery) and the transform function inside a TF graph could re-tokenise this raw data at runtime.

The pipeline is also responsible for downloading product images, which are saved to GCS as individual files to avoid hitting the content delivery network that stores product images multiple times. Images are a major bottleneck due to network latency. To avoid a TF / MLE job from making separate requests to fetch each file individually[4], Dataflow saves the raw image content into the .tfrecords file; this increases Dataflow execution time, but reduces time and cost overall, as Dataflow is cheaper and easier to parallelise than a MLE job.

A former version of the pipeline also computed image embeddings inside the Dataflow job, but this was inefficient. We also wanted to have the flexibility to try different models for computing embeddings (e.g. Inception V3, MobileNet, AutoML models) and fine-tuning image models. Attempts to persist these pre-computed embeddings had severe overheads, and currently the approach is to re-compute these at every time inside MLE, which can leverage a GPU to do it efficiently.

Visual Similarity

The Dataflow pipeline implemented for the experiments described in 3.3.1 relied on image embeddings computed by the preprocessing pipeline. 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 the simplest case computed within products that belong to same second level category. The pipeline read the files output by the preprocessing stage, and partitioned

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

[4]Each request has latency overhead, which compounds quickly when doing million of these.

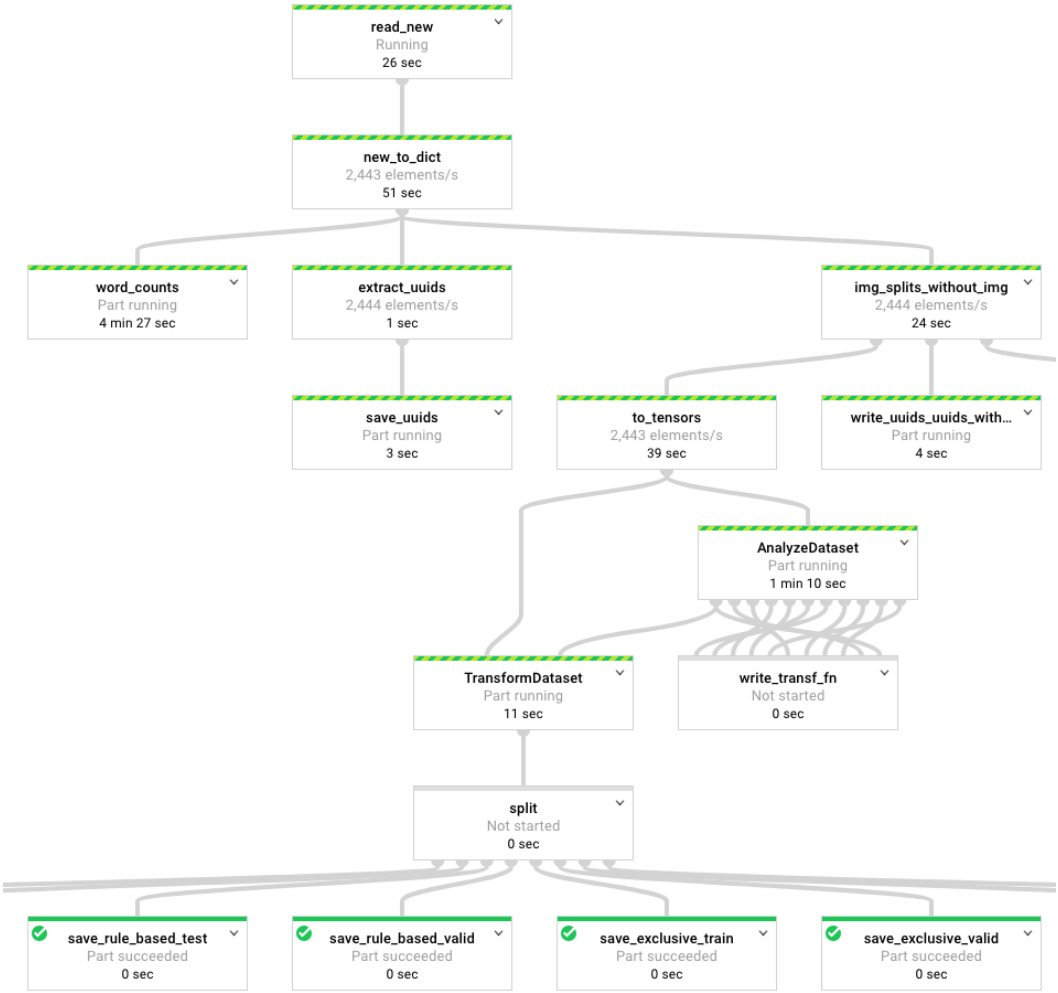

Figure 3.4: A running Dataflow pipeline of data preprocessing

it into datasets by the category predicted in the previous train run (as the new predictions would not be saved in the file). The per-category image embeddings were merged (using *reduce*) into matrices of image embeddings in a given category, and the top 100 approximate nearest neighbours were extracted for each product using nmslib [4] (as a *map* operation over all the per-category embedding matrices). The nearest neighbour UUIDs were persisted to GCS and a downstream AF task picked these up and updated ES with these.

This set-up will be replaced in the future. Computing embeddings in Dataflow is slow, and me way want to enable more flexible ways of restricting the subset of products that are considered (e.g. not just 2nd level category, but a 3rd level category or even globally). The most flexible approach is one in which nearest

neighbours are computed at runtime, i.e. when a product is viewed by a user. The biggest challenge in such a system is keeping all the image embeddings in memory, which would require 32 GB of memory for the 4 million products we currently have[5], and the number of products is likely to increase in the future.

An option worth considering is deploying this as a TensorFlow model to MLE, which reads the embedding matrix in as a sharded tf.Variable; the shards can be distributed on different machines, which is handled automatically by TF. The input to this model would be just the UUID of the product in question, and the UUID of the category to which the nearest neighbour search is limited; the "prediction" of the model would be dot product of the product embedding in question, and all the other product embeddings that are in the given category. Therefore to get the nearest neighbours of an image embedding, TF would go through all the product embeddings to check for their membership of the given category - but this is fast given all the embeddings are always kept in memory. The nearest neighbour search will also be precise, which would be prohibitively slow when pre-computing nearest neighbours but should be manageable when done online, as there will be only a handful of requests per second and the dot products are calculated in parallel by multiple workers. We can also use autoscaling that would increase the number of workers to handle high loads, which would also put all workers to sleep after 15 minutes of inactivity.

3.2.3 TensorFlow and ML Engine

All models were implemented using TensorFlow, using higher level APIs (such as tf.data, tf.estimator and tf.learn) when possible. This was somewhat challenging, as the high-level APIs were poorly documented, changed even throughout the duration of this project, and it was not clear which APIs are compatible with each other. Ultimately the only reliable way to understand a class or function was to read its source code. In many cases these APIs provided almost what was needed, but to support our use case the code was copied from the TensorFlow GitHub repository and adapted to our purpose[6].

The point of entry to the trainer program is the controller.py, which decides based on command line arguments which task to run: train, predict, train and predict, evaluate, export model, etc. There is a large number of hyperparameters that can be supplied via command-line arguments, with reasonable defaults.

The data was loaded using the tf.data.Dataset API, which provide convenient functions for reading large numbers of files, prefetching, shuffling, batching, and performing arbitrary transformations on the data (such as transforming a set of bytes representing a JPEG image into a 3D tensor of integers). This works well for simple use cases but is less flexible when for example doing multi-objective learning.

[5] $4000000 * 2048 * 32$ = number of products * embedding dimensionality * 32 bits per number

[6] For example, multi-objective learning where each objective potentially changes a subset of the model's variables was not possible due to the way loss from multiple "heads" was merged in the current TF APIs. Also some parts had to be rewritten to give us per-class evaluation metrics.

We would like to control roughly how many data points from each objective end up in a batch, or for how long a model is pre-trained on one objective before the second objective is introduced. Refer to section 3.3.3 for a description of two approaches that were tried. In general, the trainer program would use tf.data.Dataset class to build an *input function* for either training or evaluating, and the input function would be supply data points for the train loop.

TensorFlow models were built using the tf.estimator.Estimator class by providing a custom *model function*. The model function would dynamically build the model based on the *model type* (deep / linear), *input type* (these are listed in section 3.3.2), *training objectives* and *hyperparameters*. An input type determines which input features are used and how they are represented, while a training objective determined which training dataset, loss function and evaluation metrics were used. Both input types and training objectives had simple configuration files dictating their behaviour, which made adding new training objectives and experimenting with different model architectures easy.

Training was handled by the tf.estimator.train_and_evaluate function. It loads a model from the checkpoint directory if present, and trains the model for a specified number of steps (or until the input function terminates). It also periodically saves model checkpoints to GCS, and handles writing TF summary operations that are needed for data visualisations using TensorBoard[7]

3.3 Experiments, Evaluation & Results

There are many kinds of experiments conducted; the experiments, their evaluation and results are explained under the same subsection. In section 3.3.1 three visual similarity approaches are described and subjectively evaluated; in section 3.3.2 we describe all the models that were trained to reproduce the behaviour of the rule-based system; how multi-objective learning was used to improve the accuracy of individual training objectives is described in section 3.3.3; finally, our efforts to reduce label complexity using active learning is described in section 3.3.4.

3.3.1 Visual Similarity

Three approaches were tried for computing visual similarity. In the first case, an approximate nearest neighbour method [4] was used on the image embeddings (as described in section 3.2.2) and the top 100 predictions were saved to ElasticSearch. In the second approach, the embedding vectors were discretised according to [45] (as described in section 2.2.4) and inserted to ElasticSearch as strings of space-delimited tokens; standard ElasticSearch fulltext search was used on these token strings to get the nearest neighbour for a given product. The third approach com-

[7]TensorBoard is a web interface for visualising the variables and metrics a TensorFlow train run produces; many of the visualisations in the following sections are taken from Tensorboard.

bined the similarity scores of the second approach and the original ElasticSearch title matching, as a single ElasticSearch query.

Evaluation & Results

The evaluation of similarity scores is difficult, since it is inherently somewhat subjective. A common approach is to hand-label triplets of images Q, A, B with four choices: (1) both A and B are similar to query image Q, (2) both A and B are dissimilar to Q, (3) A is more similar to Q than B, and (4) B is more similar to Q than A. These labeled triplets can be ranked using with similarity precision (percentage of triplets correctly ranked) and score at top K (see [57]). Creating this dataset of triplets would be a very time-consuming process. Therefore evaluation was done entirely subjectively.

Comparing the similarity between the former ElasticSearch title (ES title) matching and the new approximate nearest neighbour (ANN) search was straightforward - one look at the results was enough to confirm that the embedding-based approach outperforms title matching. Figures 3.5 and 3.6 show the nearest neighbours according to ES title and ANN, respectively; the image embeddings nicely pick up the pattern on the shirt, as well as the general style, but also seems to prefer pictures with no fashion model. For some clothing categories this worked remarkably well, but with categories that had more varied products within it, there were occasional odd results. For example, in the "Hiking" category, the nearest neighbours of sleeping bags could be items with seemingly similar shapes, such as a flashlight that a very zoomed-in product photo that had similar contours as an unrolled sleeping bag. Another example of this is given in figure 3.8, which mostly returns chairs that are indeed visually and even functionally very similar to the chair in question - more so than the ElasticSearch title match seen in 3.7 - but also returns tables with similar legs; this is understandable, given that nearly half of the surface area of these images is covered by these foldable legs, which are visually quite unique. This problem can mostly be mitigated by having more fine-grained product categories and restricting the nearest neighbour search to those more specific categories. The nearest neighbour similarity search may pick up qualities of the image that are not related to the product in question, such as background or fashion model; this is un-intended but is acceptable, and it could be argued creates a product listing which is more homogenous and therefore more aesthetically pleasing.

The second approach of discretised and tokenised image embeddings had mixed results. The returned nearest neighbours were reasonable, but subjectively felt inferior to the ones computed from the raw image embeddings using ANN. At times it felt like some aspect of the appearance dominated, which is understandable considering a lot of precision is lost when discretising the embeddings, and there is no theoretical justification why a TF-IDF score of these arbitrarily discretised tokens should behave similarly as a dot product between the original vectors. With enough precision (using several discrete intervals per embedding dimension) this may well work, but this would incur a considerable performance overhead (when

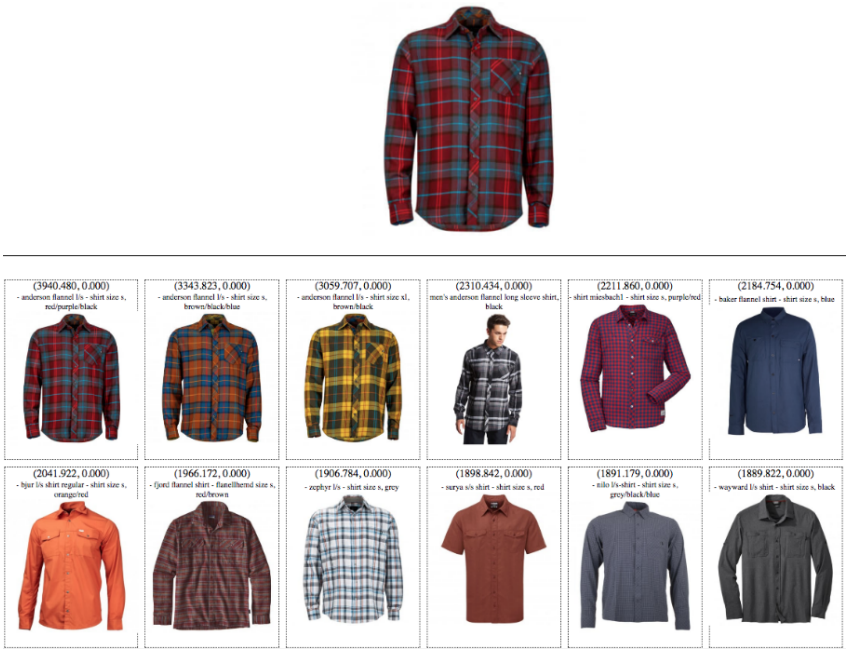

Figure 3.5: Nearest neighbours based on ElasticSearch title matching (TF-IDF-like)

indexing and querying). A hybrid approach was also attempted, where the tokenised version is used to come up with a small set of candidate nearest neighbours, which are used for a precise nearest neighbour calculation; the latter was implemented as an ElasticSearch plugin that performed dot products on the raw embedding vectors, but this was not a scaleable solution and did not subjectively lead to particularly good similarity results.

The third approach of combining the existing title matching and discretised and tokenised vectors was somewhat successful. As seen in figure 3.9, the title matching ensures that top results are at least about the same kind of items (chairs) while the tokenised embedding query returns products that share some visual similarity (mostly the legs of the chair). This approach still suffered from similar problems as the second approach - poor performance and somewhat inconsistent visual similarity - while introducing an additional complication of finding a good weighting between the title and visual scores. Given that there was no efficient way to evaluate the different weightings, it was decided to use raw image embeddings for visual similarity (either as an ANN or a precise realtime version).

3.3.2 Independent Models

Several models were trained on the rule-based labels to find out which models can best replicate its behaviour. In section 3.3.2 we describe the first model that was trained as a baseline. The different input features and how these were represented is described in section 3.3.2, along with how linear and deep modelds were trained

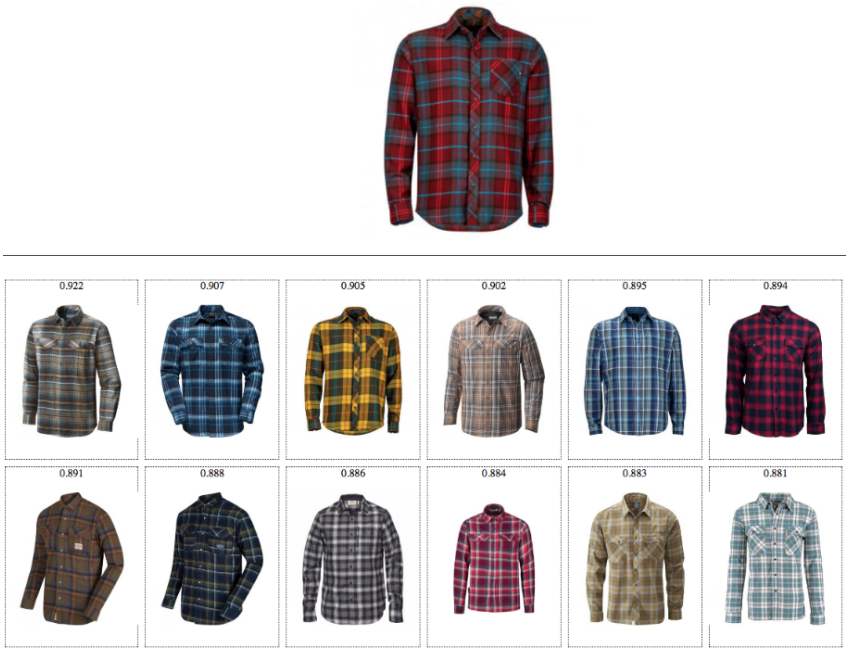

Figure 3.6: Nearest neighbours based on ANN search of image embeddings

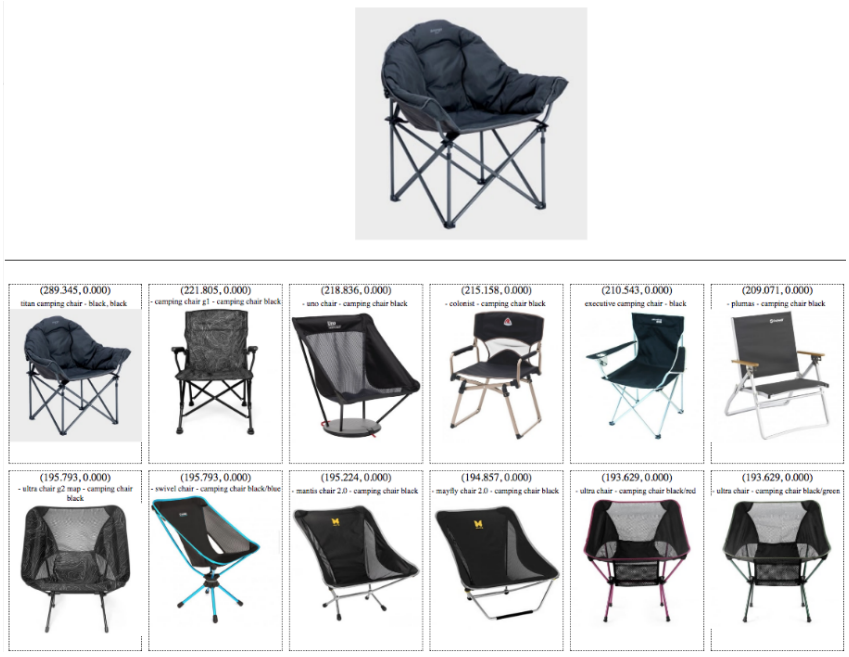

Figure 3.7: Nearest neighbours based on ElasticSearch title matching (TF-IDF-like)

Figure 3.8: Nearest neighbours based on ANN search of image embeddings

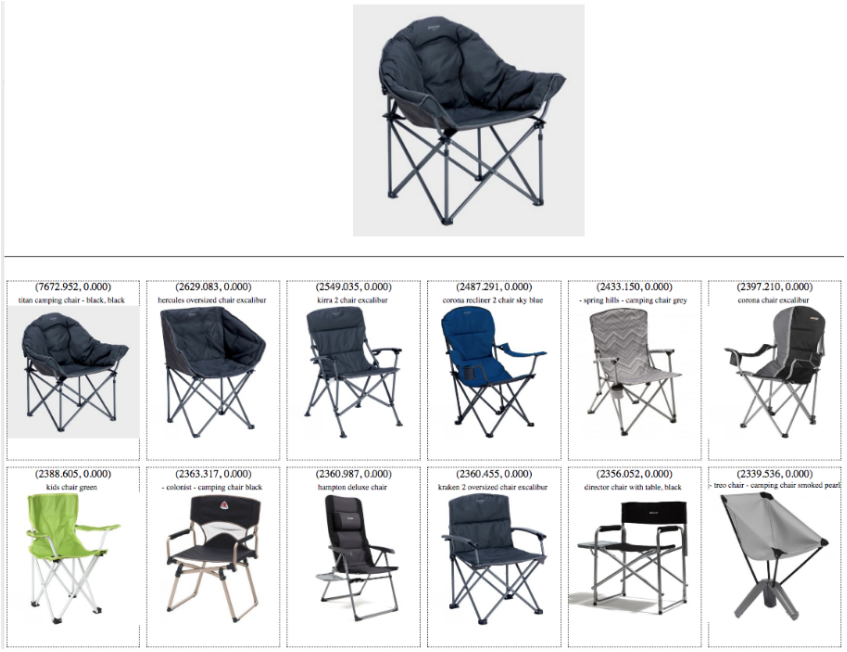

Figure 3.9: Nearest neighbours based on fulltext search on discretised/tokenised image embeddings as in [45]. Here scores from the title and image embedding fields was combined.

from these input representations. Section 3.3.3 describes how hyperparameters were selected, and two attempts at tuning these programmatically.

Evaluation

Many metrics can be used to evaluate **multi-label classification**:

- TP, TN, FP, FN: true positives, true negatives, false positives, false negatives,
- accuracy: $\frac{TP+TN}{FP+FN}$, fraction of correct predictions,
- precision: $\frac{TP}{TP+FP}$, fraction of positive predictions that were correct,
- recall: $\frac{TP}{TP+FN}$, fraction of positive labels that were correctly "recalled",
- F1: $2\frac{precision*recall}{precision+recall}$, harmonic average of precision and recall,
- TPR = precision: $\frac{TP}{TP+FP}$, true positive rate,
- FPR: $\frac{FP}{FP+FN}$, false positive rate,
- ROC AUC: the area under the curve of TPR plotted against FPR.
- PR AUC: the area under the curve of precision plotted against recall.

In our case, an overwhelming number of labels are negative, therefore a very simple predictor (that predict "negative" for every product / category) would have an accuracy near 99%; ROC AUC is not suitable for the same reason, as there would be no false positives. Precision and recall are useful measures to understand the model: it is expected to see a sharp increase of precision and decrease of recall as the model learns to predict "negative" for most classes, followed by a gradual increase of recall as the model learns to predict "positive" for the correct classes. Precision and recall are competing metrics: increased precision tends to lead to decreased recall and vice versa, therefore models which combine the two are suitable for evaluating multi-label prediction problems with unbalanced classes. While F1 score gives the harmonic mean of precision and recall at a given threshold, PR AUC paints a more complete picture by showing what the ratio would be at all the threshold values[8]. Therefore, all evaluation of multi-label classification is based on PR AUC, i.e. when choosing between model architectures or hyperparameters, a model with a higher PR AUC is preferred.

For **multi-class classification**, accuracy is still a reasonable measure. Even though classes are imbalanced ("Dresses" has more products than "Travel Books"), there is no single dominant class that could be used as a shortcut by the model. On the other hand, similar problems arise with evaluating multi-class problems where there is potentially overlap between some classes: the metric would give a low score even if the model predicted a semantically very similar class, or a more general class instead of the more specific one it was labelled as. Therefore a manual evaluation of

[8]There are infinitely many threshold values in the range 0 ... 1, so naturally we consider a small number of threshold values when approximating AUC.

the misclassified products is needed to determine whether the error rate is caused by the ambiguity of the classes or by problems with learning.

Baseline Model: Wide & Deep

As described in section 2.2.1 "Wide & Deep" consists of two models that are trained jointly using stochastic gradient descent: a deep and a shallow neural network which both predict the same set of binary outputs. The original idea of Wide & Deep was to use the wide component for feature crosses (e.g. of two categorical values), but in our experiments the wide component received mostly just the 1-hot encoded categorical inputs while the wide component received the same inputs as random embeddings (or as pre-computed image embeddings extracted with Inception V3 that is pre-trained on ImageNet).

Initial experiments with this model were done on a dataset of roughly 800 000 products which were labeled by the rule-based system[9]. The data was partitioned into a train (~80%), validation (~10%) and test (~10%) sets in the Dataflow preprocessing pipeline based on a pseudorandom number generator - assign the product in question to the dataset if the random number is in the range 0 ... 0.8, 0.8 ... 0.9, 0.9 ... 1. This allowed us to use parallel data preprocessing where workers do not need to be aware of the index of the data point it is processing or what the other workers are doing; due to the randomness the dataset split boundaries will not be exact (e.g. test and validation sets may end up with 12% and 8% of products), but this is acceptable given the relatively large dataset size. The train set was used to train the model, and the validation set was used to evaluate the model periodically as it was trained - during individual training runs and during hyperparameter tuning. The test set was held out for evaluating the model after hyperparameter tuning, but was not used in this case.

Results The PR AUC on the 800,000-item dataset for this model after hyperparameter tuning was 0.9981 (see section 3.3.3 for a full description of the set-up). A manual examination of the predictions revealed that the top predictions for most categories was high-quality, but there was a large number of products below the decision boundary that actually belonged above it. A later run of the model on 1 600 000 products gave an PR AUC of 0.8647; this was using the hyperparameters that the tuning step produced, so it is likely that the decreased AUC PR score is caused by a bug[10]. There were occasional odd results from the model once it was trained on the 1.6M dataset; it was hard to tell which part of the model was causing

[9]The database in the client company was in constant flux, so the training set sizes depended on how many labels the rule based-system had assigned to the current database

[10]Three possible explanations come to mind: (1) the initial PR AUC was artificially high as some of the test set products ended up in the training set; (2) when migrating from the 800k dataset to the 1.6M dataset, the image embeddings were transferred over as an attempted optimisation approach; (3) the model was overfitting, 10 epochs of 1.6M is effectively 20 epochs of 800k products (the fact that there were more products does not necessarily matter, because labels were assigned simplistically from keyword matches on the title).

it, or even whether all the input features are actually improving predictive power; therefore, a more systematic approach was tried, explained in the next section.

Input Representation & Model Type Combinations

To assess which features are most useful for learning the rule-based objective, different features were grouped into *input types*, shown in table 3.10. The columns represent the input feature; refer to 3.1 for the full feature names and their description. The entries in the table show how each input feature was represented: *emb* for for categorical features that are represented as random embedding vectors (which are updated during SGD), emb avg for multi-token fields where tokens are represented as random embeddings and the input feature is a weighted sum of these embeddings[11]. Feature columns marked with *u.s.enc* have been converted from raw text into dense embedding vectors of size 512 using a pre-trained model Universal Sentence Encoder [7]; cells marked with *mobilenet* and *inceptionv3* have respectively extracted image embeddings of size 1280 and 2048 using the Mobilenet [25] and InceptionV3 [52] models that were pre-trained on the ImageNet challenge; these embedding models were used via TensorFlow Hub[12]. Pre-trained models were used only as feature extractors, i.e. were not fine-tuned on our data.

Two model types were used to train with each of these 16 input types: linear and deep. Like in the case of the baseline model, the ~1.6 million data points were split into train/validation/test sets (80%/10%/10%); train and validation sets were used during individual training runs and hyperparameter tuning. All 32 combinations of input and model type were trained using hyperparameters listed in section 3.3.3; these were selected based on what worked well on the baseline model and in general "seemed reasonable". The initial plan was to use automatic hyperparameter tuning on a few model/input types, but the tuning failed to produce good results.

Table 3.11 shows the precise PR AUC and recall scores of the final time steps. The training loss function of linear and deep models with all features (but no extra embeddings extracted with another pre-trained model) is shown in figure 3.12. The PR AUC on the validation set is shown in figure 3.13, where each model is trained for 300 000 batches, which is roughly 6 epochs; similarly, figure 3.14 shows how recall changes during the course of training. The PR AUC on the final step (300k, or 220k for inceptionv3) is shown in figure 3.15; recall on the final time step was omitted, as it was very similar to the final PR AUC scores.

[11] Each embedding vector is weighted by the number of times it appears in the input feature, and divided by the L2 norm of the token counts; this gives good results for bag-of-words inputs according to TensorFlow documentation:
https://www.tensorflow.org/api_docs/python/tf/feature_column/embedding_column

[12] TensorFlow Hub is a collection of TensorFlow models that are pre-trained on some other task and can be used as a feature extractor or as an initialisation for a model that is fine-tuned: https://www.tensorflow.org/hub/

Input Type	cat	raw cat	title	desc	brand	size	gender	img
full_emb	emb	emb avg	emb avg	emb avg	emb	emb	1-hot	mobilenet
full_one_hot	1-hot	k-hot	k-hot	k-hot	1-hot	k-hot	1-hot	mobilenet
noimg_emb	emb	emb avg	emb avg		emb	emb	1-hot	
noimg_one_hot	1-hot	k-hot	k-hot		1-hot	k-hot	1-hot	
categorical_emb	emb				emb	emb	1-hot	
categorical_one_hot	1-hot				1-hot	k-hot	1-hot	
raw_cat_emb_avg		emb avg						
raw_cat_k_hot		k-hot						
title_emb_avg			emb avg					
title_k_hot			k-hot					
desc_emb_avg				emb avg				
desc_k_hot				k-hot				
title_uni_sent_enc_emb				u.s.enc				
desc_uni_sent_enc_emb				u.s.enc				
mobilenet_emb								mobilenet
inceptionv3_emb								inceptionv3

Figure 3.10: Input types. Orange: embedding representation, blue: 1-hot or k-hot, red: embedding extracted with a pretrained deep network.

	PR AUC (deep)	PR AUC (linear)	Recall (deep)	Recall (linear)
full_emb	**0.9848**	**0.6974**	0.9351	0.4613
full_one_hot	0.975	-	0.9037	-
noimg_emb	**0.9805**	0.5354	0.9243	0.3018
noimg_one_hot	0.9627	**0.7169**	0.8765	0.4492
categorical_emb	0.799	0.4936	0.6155	0.2638
categorical_one_hot	0.7748	0.5445	0.5919	0.3069
raw_cat_emb_avg	0.8808	0.4514	0.7231	0.2637
raw_cat_k_hot	0.8706	0.6211	0.705	0.358
title_emb_avg	0.9298	0.4195	0.8242	0.2388
title_k_hot	0.9215	0.5363	0.7904	0.291
desc_emb_avg	0.8388	0.417	0.6714	0.2286
desc_k_hot	0.8975	0.6688	0.7562	0.3932
title_uni_sent_enc_emb	0.749	0.4885	0.555	0.2808
desc_uni_sent_enc_emb	0.7337	-	0.5304	0.4305
mobilenet	0.7233	0.6479	0.5065	0.4152
inceptionv3 (220k steps)	0.6697	-	0.473	-

Figure 3.11: Evaluation metrics of the rule-based training objective on the final time step. Minus denotes a train run that was not scheduled (inadvertently).

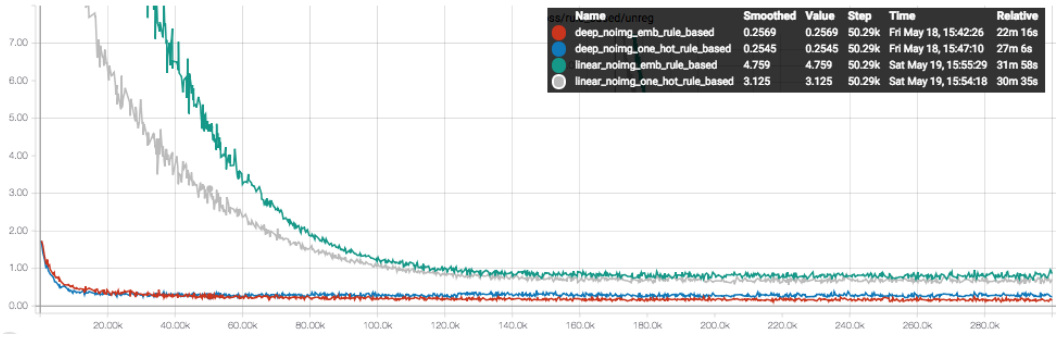

Figure 3.12: x axis = train step (batch number), y axis = training loss of rule-based objective.

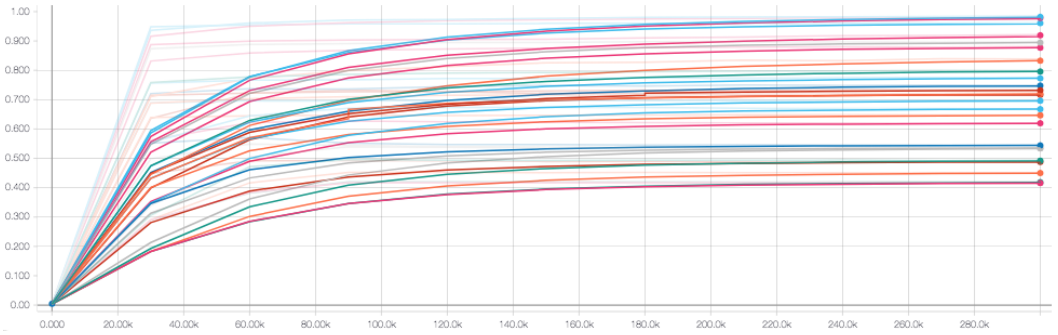

Figure 3.13: PR AUC of all combinations of input and model types; x axis = train step (batch number), y axis = PR AUC. Refer to figure ... for exact scores.

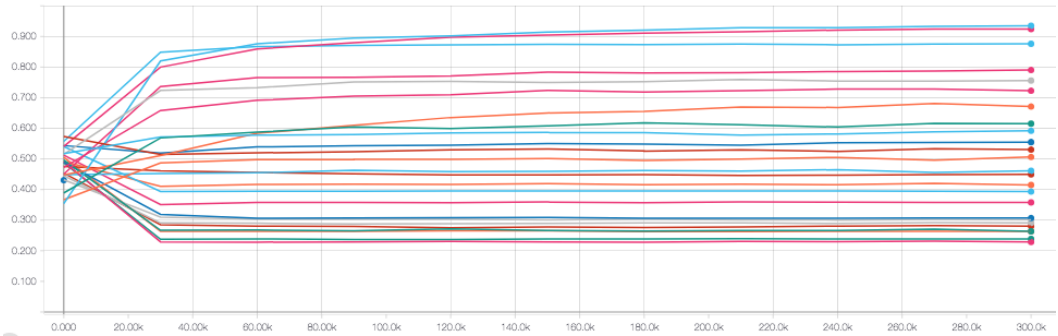

Figure 3.14: Recall of all combinations of input and model types; x axis = train step (batch number), y axis = recall..

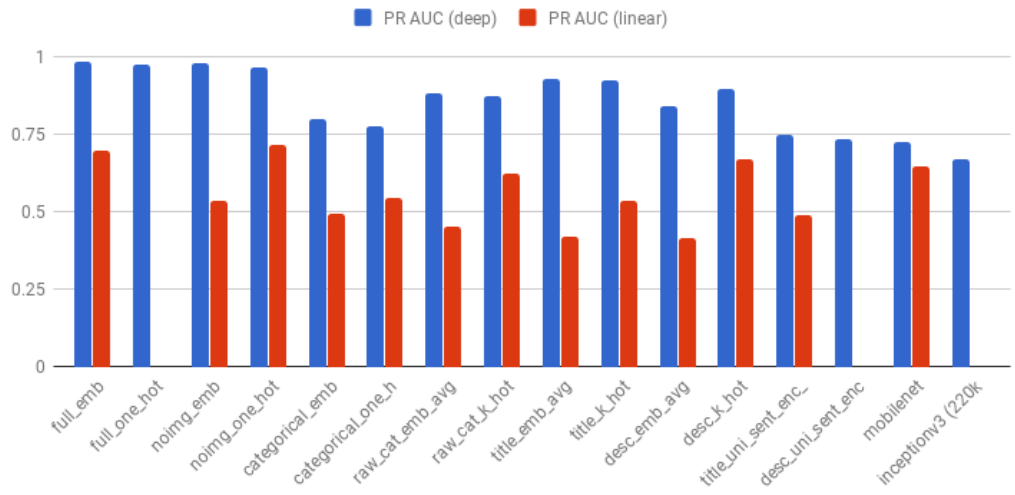

Figure 3.15: PR AUC on final timestep (300k, or 220k for inceptionv3)

3.3.3 Multi-Objective Training

pre-train vs train exclusive from the start both vs interleave from rb to ex

Hyperparameter Selection & Tuning

Hyperparameters were chosen based on gut feeling and Bayesian optimisation. The former was used to come up with reasonable ranges for hyperparameter ranges; automatic tuning was done on the GCP ML Engine, where one just needs to specify the ranges of hyperparameters, their type (categorical, integer, reverse log scale), and how many models are trained in these hyperparameter ranges. The ML Engine tuning mechanism uses Bayesian optimisation to come up with candidate hyperparameter values, which is described in section 2.6.1 and by GCP engineers[13]. The hyperparameter ranges used and the results that each of the three tuning rounds found is shown in table 3.16. In all cases 60 models were trained, with maximum 4 models trained in parallel, with early stopping.

The *L2 scale* parameter is the weighting factor on the L2 regularisation term that is applied across the weights of the linear and deep models. *Linear learning rate* is the learning rate of the Adam optimiser for the linear part of Wide & Deep, or the weights of the linear model; the other parameters of the optimiser were left as defaults (TensorFlow and the original paper); similarly for *deep learning rate*. Dropout corresponds to the likelihood of dropping out a hidden unit during training; *layers* corresponds to the number of hidden layers, and *hidden units* corresponds to the number of units in those hidden layers; *activation* corresponds to the activation

[13]https://cloud.google.com/blog/big-data/2017/08/hyperparameter-tuning-in-cloud-machine-learning-engine-using-bayesian-optimization

	Wide & Deep		Deep (rb title)		Deep (ex from rb)		Deep (rb all)
	full range	foud	full range	found	full range	found	used
batch size	32, 64, 128, 256	**64**	64	-	64	-	64
L2 scale	-	-	10^-1 ... 10^-5	**0.099379**	0.0005	-	0.0005
linear learning rate	10^-3	-	-	-	-	-	-
deep learning rate	10^-3	-	10^-2 ... 10^-5	**0.002657**	10^-4 ... 10^-6	**???**	0.0001
dropout	0, 0.1, 0.25, 0.5	**0**	0, 0.1, 0.25, 0.5	**0.5**		-	0
layers	1 ... 30	**17**	1 ... 30	**28**	4	-	4
hidden units	30 ... 1000	**784**	30 ... 1000	**85**	784	-	784
activation	relu, tanh	**relu**	relu	-	relu	-	relu
	PR AUC:	0.998	PR AUC:	0.4069	PR AUC:	???	

Figure 3.16: Hyperparameter ranges and the values found during automatic tuning. A single value in the "full range" column means the value was fixed.

function used in the hidden layers; the four hyperparameters are only applicable to the deep and Wide & Deep models.

Table 3.16 shows the the tuning rounds in different colours, as well as the PR AUC achieved by each. In the *Wide & Deep* case, tuning was performed over ~800k data points on the rule-based objective, with each model trained up to 10 epochs (e.g. 125k time steps with a batch size of 64). In the *Deep (rb title)* case, a random sample of 15% of the full 1.6M dataset was used to train on the rule-based objective for up to 50k time steps; the model used was deep_title_emb_avg, which showed good results compared to the small selection of models that was manually tried. In the *Deep (ex from rb)* case, the deep_noimg_emb model was trained on the full 1.6M dataset on the *from_rb_to_ex* objective (described in section 3.3.3) for up to 300k time steps. In the Wide & Deep and *Deep (rb title)* cases, models with a higher PR AUC score were preferred while in *Deep (ex from rb)* accuracy was used to choose between hyperparameter performance.

It is clear that the hyperparameters found by the *Deep (ex from rb)* round were not optimal, therefore the ones used in the *Deep (rb all)* was used in most experiments described in section 3.3.2.

3.3.4 Active Learning

4. Discussion

4.1 Visual Similarity

Could also try image embeddings extracted from the last non-FC layer to extract lower-level fisual features

4.2 Baseline Model

it is questionable whether adding the same data to the deep and wide components is beneficial some justification is given by highway networks & densenets; our version could be seen as a special case of a highway connection from the input to the output

model consistently underpredicted categories due to

4.3 Independent Models

Would be interesting to see results from character n-grams, gradient boosted trees, tfidf-weighted inputs the same number of tokens was used for embeddings and 1-hot encoding; it could be beneficial to take nearly all tokens for embeddings that aappear more than once, as we're not adding input dimensions

weight decay not necessarily a hyperparameter

k-fold xvalid

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] Luca Bertinetto, Jack Valmadre, João F. Henriques, Andrea Vedaldi, and Philip H. S. Torr. Fully-convolutional siamese networks for object tracking. *CoRR*, abs/1606.09549, 2016.

[4] Leonid Boytsov and Bilegsaikhan Naidan. Engineering efficient and effective non-metric space library. In *Similarity Search and Applications - 6th International Conference, SISAP 2013, A Coruña, Spain, October 2-4, 2013, Proceedings*, pages 280–293, 2013.

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

[6] 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.

[7] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. Universal sentence encoder. *CoRR*, abs/1803.11175, 2018.

[8] 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.

[9] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase repre-

sentations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[10] François Chollet. Information-theoretical label embeddings for large-scale image classification. *CoRR*, abs/1607.05691, 2016.

[11] Ido Dagan and Sean P Engelson. Committee-based sampling for training probabilistic classifiers. In *Machine Learning Proceedings 1995*, pages 150–157. Elsevier, 1995.

[12] 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.

[13] 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.

[14] 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.

[15] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015.

[16] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. *CoRR*, abs/1705.03122, 2017.

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

[18] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[19] Yves Grandvalet and Yoshua Bengio. Semi-supervised learning by entropy minimization. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 529–536. MIT Press, 2005.

[20] Kyu J. Han, Akshay Chandrashekaran, Jungsuk Kim, and Ian R. Lane. The CAPIO 2017 conversational speech recognition system. *CoRR*, abs/1801.00059, 2018.

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

[22] Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal. The" wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

[23] Geoffrey E Hinton, James L McClelland, David E Rumelhart, et al. Distributed representations. *Parallel distributed processing: Explorations in the microstructure of cognition*, 1(3):77–109, 1986.

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

[25] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[26] Zhiting Hu, Zichao Yang, Xiaodan Liang, Ruslan Salakhutdinov, and Eric P. Xing. Toward controlled generation of text. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1587–1596, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[27] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[28] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[29] Surya Mattu Jeff Larson. How We Analyzed the COMPAS Recidivism Algorithm. `https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm`, 2016. [Online; accessed 17-May-2018].

[30] Brian Keng. Semi-supervised Learning with Variational Autoencoders. `http://bjlkeng.github.io/posts/semi-supervised-learning-with-variational-autoencoders/`, 2017. [Online; accessed 1-April-2018].

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

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

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

[34] Diederik P Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. In *Advances in Neural Information Processing Systems*, pages 3581–3589, 2014.

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

[36] Ashutosh Kumar, Arijit Biswas, and Subhajit Sanyal. ecommercegan : A generative adversarial network for e-commerce. *CoRR*, abs/1801.03244, 2018.

[37] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[38] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[39] Matej Moravcík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael H. Bowling. Deepstack: Expert-level artificial intelligence in no-limit poker. *CoRR*, abs/1701.01724, 2017.

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

[41] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[42] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[43] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[44] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115(3):211–252, December 2015.

[45] Jan Rygl, Jan Pomikálek, Radim Rehurek, Michal Ruzicka, Vít Novotný, and Petr Sojka. Semantic vector encoding and similarity search using fulltext search engines. *CoRR*, abs/1706.00957, 2017.

[46] Burr Settles. Active learning literature survey. Technical report, 2010.

[47] Burr Settles, Mark Craven, and Soumya Ray. Multiple-instance active learning. In *Advances in neural information processing systems*, pages 1289–1296, 2008.

[48] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[49] 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.

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

[51] 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.

[52] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[53] Antti Tarvainen and Harri Valpola. Weight-averaged consistency targets improve semi-supervised deep learning results. *CoRR*, abs/1703.01780, 2017.

[54] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[56] 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.

[57] Jiang Wang, Yang Song, Thomas Leung, Chuck Rosenberg, Jingbin Wang, James Philbin, Bo Chen, and Ying Wu. Learning fine-grained image similarity with deep ranking. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '14, pages 1386–1393, Washington, DC, USA, 2014. IEEE Computer Society.

[58] Xiang Wei, Zixia Liu, Liqiang Wang, and Boqing Gong. Improving the improved training of wasserstein GANs. In *International Conference on Learning Representations*, 2018.

[59] 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.

[60] Adams Wei Yu, David Dohan, Quoc Le, Thang Luong, Rui Zhao, and Kai Chen. Fast and accurate reading comprehension by combining self-attention and convolution. In *International Conference on Learning Representations*, 2018.

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

[62] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. *CoRR*, abs/1609.05473, 2016.

[63] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rogriguez, and Krishna P. Gummadi. Fairness Constraints: Mechanisms for Fair Classification. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 962–970, Fort Lauderdale, FL, USA, 20–22 Apr 2017. PMLR.

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, May 27, 2018

.....
Mattias Arro

A. Screenshots

A.1 Dataprep Histogram

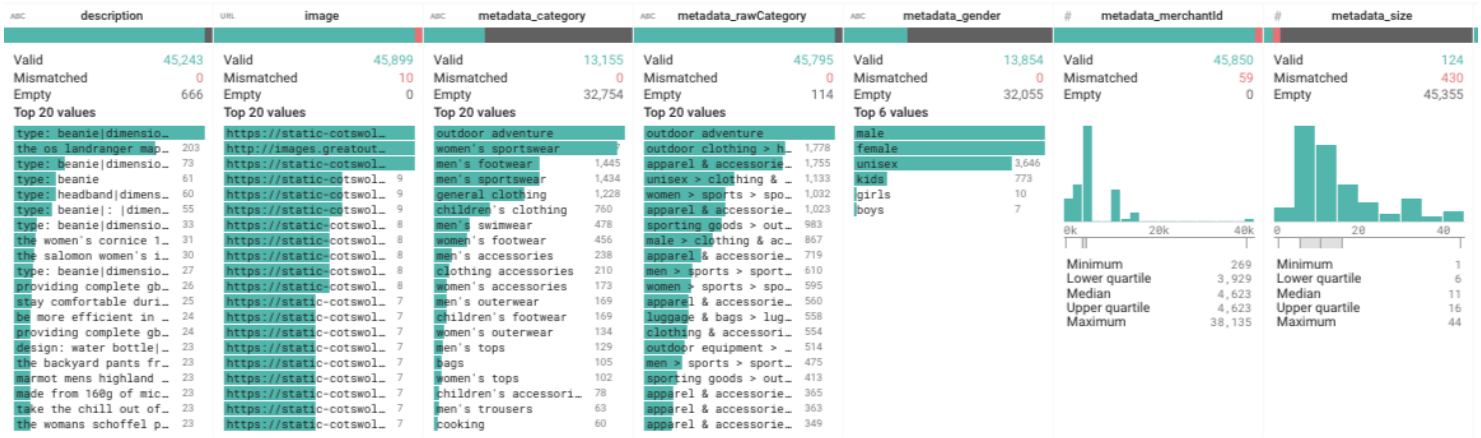

Figure A.1: Dataprep: the histograms of field values of a subset of fields.