4th International Conference on Predictive Applications and APIs

Volume 82:

PAPIs 2017

Claire Hardgrove, Louis Dorard and Keiran Thompson

Preface

We introduce the proceedings of advanced talks presented at the 4th International Conference on Predictive Applications and APIs (PAPIs '17), in Boston, United States on October 24-25, 2017.

The conference featured 25 talks discussing various aspects of the integration of Machine Learning in real-world applications, processes and businesses. Speakers presented techniques, tools and lessons learned to an audience of researchers and industry practitioners with a wide range of experience levels. These proceedings contain papers from five of the talks that presented the most advanced and novel content.

The first paper demonstrates how to generate large training datasets by extreme augmentation of a small collection of coloured cel-style cartoon images, without overfitting, and with applications to automatic colouring of cartoon images in art, design and animation.

The second paper introduces MMLSpark, an open source library that combines the deep learning library Microsoft Cognitive Toolkit (CNTK) with Apache Spark, adding several memory and performance enhancing optimizations. The paper introduces java bindings for CNTK which enable deep learning on a large class of new devices running on the Java Virtual Machine. These bindings enable a SparkML transformer that distributes, loads, and executes arbitrary CNTK computation graphs inside SparkML pipelines. OpenCV is integrated as a SparkML transformer, and an automated wrapper generation system is introduced which generates Python wrappers for PySpark.

The third paper describes an architecture for fitting many related but independent predictive models, where models are similar enough to warrant investment in shared infrastructure, but each model requires specific customizations and insight. The authors focus is on Repeated Domain-Specific Modeling systems for specific business problems, and the paper provides a set of abstractions for thinking about this type of model, and an implementation in R.

The fourth paper presents an open-source platform, Marvin, designed to help data scientists manage the lifecycle of an artificial intelligence project. Marvin can be used either on-premise or in the cloud with any algorithm that can be implemented in general-purpose languages and provides more comprehensive tools than other comparable solutions. It provides tools for data exploration, versioning, tools for sampling development data, a unit test framework, skeleton project generation, artifact versioning, parallel and distributed computing integration, training pipeline interface, a feedback server to allow models to receive external signals, and a predictor server to maintain trained models in persistent memory storage.

The fifth paper demonstrates how to predict airline flight delays using a Variational Long Short-Term Memory model. It differs from previous work using Multi-Layer Perceptron and General Regression Neural Net, traditional machine-learning techniques, and a domain-specific technique in using Monte Carlo Dropout to provide robust uncertainty metrics required for a risk management setting.

Acknowledgments

Program Committee and Session Chairs:
Keiran Thompson (Stanford), Ikaro Silva (Philips Research), Paula Fadul (Telefonica), Yan Zhang (Microsoft), Sabrina Feder (Comma Soft), Vincent Van Steenbergen, Erran Li (Uber), Jeremy Achin (DataRobot), Sudarshan Raghunathan (Microsoft), Eli Amesefe (Scribd), Ruben Glatt (USP), Matt Higgs (University of Glasgow) and Sunanda Koduvayur (Wayfair)

PAPIs Team:
Nikola Mojic, Iñigo Martinez, Brice Blanloeil, Ruben Glatt and Leonardo Noleto

Sponsors:
Microsoft, Wayfair, DataRobot, ElementAI, Unity, BigML, Dataiku, VersaMe, Nexosis, Adobe, Oracle

Finally, we wish to thank the authors for their time and effort in creating such a fine program.

October 2017

The Editorial Team:

Claire Hardgrove, PAPIs '17 Publications Chair
IBM Research Australia
chardgro@au1.ibm.com

Keiran Thompson, PAPIs '17 Program Co-chair
Stanford University
keiran@stanford.edu

Louis Dorard, PAPIs.io General Chair
UCL School of Management
l.dorard@ucl.ac.uk

Table of Contents

Proceedings of Machine Learning Research 82 (2017) 1-9, 4th International Conference on Predictive Applications and APIs

DragonPaint: Rule Based Bootstrapping for Small Data with an Application to Cartoon Coloring

K. Gretchen Greene KGRETCHENGREENE@GMAIL.COM
Greene Analytics
Industry Lab
288 Norfolk St.
Cambridge, MA 02139, USA

Abstract

In this paper, we confront the problem of deep learning's big labeled data requirements, offer a rule based strategy for extreme augmentation of small data sets and apply that strategy with the image to image translation model by Isola et al. (2016) to automate cel style cartoon coloring with very limited training data. While our experimental results using geometric rules and transformations demonstrate the performance of our methods on an image translation task with industry applications in art, design and animation, we also propose the use of rules on partial data sets as a generalizable small data strategy, potentially applicable across data types and domains.

Keywords: rules, augmentation, image to image translation, style transfer, synthetic data

1. Introduction

A big problem with supervised machine learning is the need for huge amounts of labeled data. At least it's a big problem if you don't have the labeled data and even now, in a world awash with big data, most of us don't. While a few companies have access to enormous quantities of certain kinds of labeled data, for most organizations and many applications, the creation of sufficient quantities of exactly the kind of labeled data desired is cost prohibitive or impossible. Sometimes the domain is one where there just isn't much data. It might be the diagnosis of a rare disease or determining whether a signature matches a few known exemplars. Other times the volume of data needed and the cost of human labeling by Amazon Turkers or summer interns is just too high. Paying to label every frame of a movie length video adds up fast, even at a penny a frame.

We confront the problem of deep learning's big labeled data requirements, offer a rule based strategy for extreme augmentation of small data sets and apply that strategy with an image to image translation model to automated cartoon coloring with very limited training data with industry applications in art, design and animation.

Figure 1: Kid's rules: "Color by body part."

2. A Small Data Problem - Automating Cartoon Coloring

We consider the problem of automating the consistent coloring of a cartoon character type, in a flat color style like old Disney cel animation or The Simpsons. Is animated character coloring a small data problem? In many cases it's not, but sometimes, it is.

If we were actually trying to color Bart Simpson or Mickey Mouse and had access to the Fox or Disney film archives, we might well have sufficient data for standard machine learning methods or at least enough footage to make it. There are 625 Simpsons episodes, nearly 20 million frames (Wikipedia, 2017). Even for relatively rare characters in film or video, we might be able to leverage frame by frame continuity to color subsequent or intermediate frames, and when choosing production methods for new projects we could model 3D CG characters and color each only once using a 3D shader that imitates old style 2D cel coloring.

Here, we explicitly want to study a case of scarcity, when a large body of work doesn't exist. When we have just a few dozen drawings, unconnected across time, only one of them colored, what can we do then?

If we are to get from such a small data set to a machine learning model that can consistently paint the rest of the training drawings and the same artist's future drawings of the same type(s), we'll need extreme augmentation without overfitting.

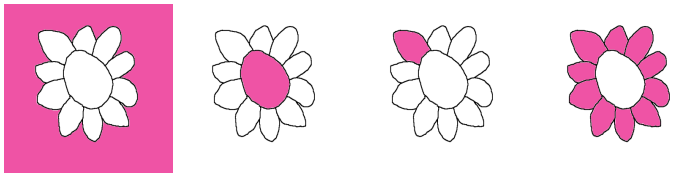
3. The Geometry of Dragons: a Rule Based Solution for 80 Percent

Faced with a shortage of training data, we should be asking if there is a good non machine learning based approach to our problem. If there's not a complete solution, is there a partial solution and would a partial solution do us any good? Do we even need machine learning to color flowers and dragons or can we specify geometric rules for coloring?

3.1 Tell a Kid How to Color

When you don't know what rules to tell a computer, one way to start is to ask what rules you'd tell a person. If you were telling a kid how to color a flower or dragon, what would you tell them? You could probably give them one that is colored and just say, "color the others like this." But to be more explicit about what "like this" means, you could state rules in terms of body or flower parts: Make the center orange. Make the petals yellow. Make the body green. Make the spikes yellow. Make the eyes white (Fig. 1).

For flowers or simple dragons, we'd be done with just two or three rules. For fancy dragons, the rule list would be longer but still doable. We could write down a dozen "color by body part" rules to color wings, ears, belly scales, arms, legs, claws, etc. We'd tell the



(a) Not flower (b) Center (c) One petal (d) All petals

Figure 2: Color flower by component

kid the extra rules or provide a couple of examples with the extra body parts and we'd expect our flowers and dragons would get colored the way we wanted them.

But for a computer that doesn't know what a "body" or a "petal" is, it seems like we're as far away as ever. We could train a model to recognize body and flower parts but we'd need a huge amount of labeled training data which we don't have.

3.2 From Body Parts to Geometric Rules

What we'd like to do is directly translate our body part rules into geometric rules that are easy to program. The original coloring task was inspired by examples colored with a "paint bucket" tool which colors all the pixels in the same connected component as the clicked pixel. If we find the connected components in a drawing (which is done for us in existing libraries in e.g. Python), we can write down geometric rules using area and distance to label the parts of flowers (Fig. 2) and simple dragons and then use our original kids' color by body part rules. These geometric labeling rules don't work for all of our initial drawings but they give us a solid partial solution that works for about 80 percent.

Rules for flowers (or simple dragons):

1. "background"= biggest white component
2. "center" (or "body") = second biggest white component
3. "petals" (or "spikes+") = the remaining white components
4. (For dragons, add "eye" vs. "spikes" rule using distance to background)

For at least some of the fancy dragon features, we can keep going, finding geometric rules that distinguish between claws and spikes or between eyes and eyelids. But even if we had geometric rules for every feature, our basic geometric rules for labeling body parts are fundamentally flawed. They only work for most of the drawings. For every rule I've written, I have a drawing it doesn't work for. Where do the rules break down? Why don't they work for all drawings? And if they aren't reliable, what good are they?

4. From 80 Percent to a Full Training Set With Rule Breaking Transformations and Extreme Augmentation

Our geometric rules work to color about 80 percent of the original drawings. We'll call those drawings "rule conforming", write a program to color them and make those AB pairs the beginning of our training set. Then what?

Figure 3: Where rules break down: line gaps, sizes, back limbs

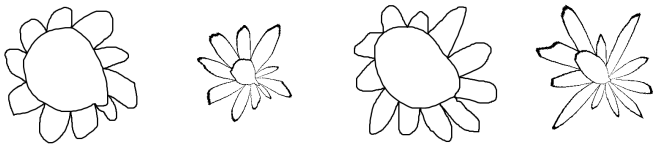


Figure 4: Synthetic daisy creation with $f(r, \theta) = (r^3, \theta)$

4.1 Where the Rules Break Down - the Other 20 Percent

Let's examine the question of where the rules break down (Fig. 3). Writing geometric rules, we are formalizing assumptions that aren't true for all of our drawings: The background is connected. The background is the biggest component. Flower centers are bigger than all the petals. There are no gaps in drawn lines. Body and limbs are connected.

It's these cases that we hope to train our model to take care of because our geometric rules don't. But because they couldn't be rule colored, they aren't in our training set. We need to find a way to add them.

4.2 Rule Breaking Transfomations

For each of the assumptions/rules we made, we look for rule breaking transformations that we can apply to a rule conforming drawing A or to a drawing/colored AB pair (by applying the same transformation to A and to B) so that we can augment our AB training set with pairs that break all the rules. We use domain knowledge where appropriate to choose the best functions and parameters.

Background is connected and is the biggest component. We cropped, scaled or translated to disconnect and/or shrink the background.

Center of flower smaller than petals. We assumed our flowers all looked like a sunflower with a big center and smaller petals but what about the ones that look more like a daisy with a small center and big petals or petals of various sizes?

We'd like a transformation that can turn our sunflowers into daisies. Since the flowers have an approximate radial symmetry, we switched to polar coordinates, scaled to embed our image square in a unit disk and looked at radially symmetric homeomorphisms of the disk to find a function which would disproportionately shrink the center of our image and make a synthetic daisy. $f(r, \theta) = (r^3, \theta)$ worked perfectly to create daisies (Fig. 4) but distorted dragons well beyond natural variation and almost beyond recognition.

Gaps in lines. We expect some test drawings will have poorly connected lines where there's a gap; where, for example, the petal doesn't quite connect to the center in a hastily drawn line. To create gaps in lines in training data, we made erasures on a drawing A from

Figure 5: Elastic distortion/Gaussian blur

an AB pair, either by hand or in an automated way using a squiggle as a mask, and then paired the new A′ with its old colored version B to get a new A′B pair with line gaps for the training set. Other functions occasionally created gaps by stretching/thinning lines.

4.3 Additional Augmentation

Having filled in the training set with many of the kinds of data we knew we were missing, what else can we do to add useful variation to the data while avoiding overfitting?

Affine transformations. We included the usual affine transformation augmentations: translation, rotation, scale, skew and mirror flip. (See Bloice et al., 2017).

Elastic distortions/Gaussian blur. We extended an idea from Simard et al. (2003) to use Gaussian blur to change the drawn line. Simard et al. used it to augment the MNIST OCR training set, comparing it to the natural oscillations of the hand. (For an implementation see Junior, 2017). A very useful transformation type, elastic distortions preserved pose and major features but changed characters in interesting and believable ways. For example, two distortions gave the same dragon either a fat, short snout or a long, pointy one and changed the bend in its tail (Fig. 5).

Composition of functions. For each original AB pair, we composed multiple sequences of transformations to get new AB pairs. Two composition dangers to watch out for are half dragons in space (cropping an edge and then moving that cut edge back into the middle) and unintended duplicates (beware of commutativity.)

5. Identify Target ML Model and Training Needs

As a general problem solving strategy, when the outline of our proposed solution has multiple parts, we should have some level of confidence that we'll be able to solve the other parts before we invest too much effort into one. To the extent that our approach to one part affects another, we should go beyond likelihood estimates and have an idea what the subpart solutions might look like.

Before beginning our quest for rule breaking transformations and extreme augmentation, we should have had an idea of the kind of machine learning model that should work if we could build a training set and what kind and how much training data we'd likely need. Our target number for a deep learning model might have been in the tens of thousands or hundreds of thousands. It was only the identification of a suitable existing model with comparatively modest data needs that made sufficient augmentation seem possible.

We have a natural pairing of images in two styles (uncolored drawing and its colored version) and we want to go from one of the first kind to one of the second kind, which made our cartoon coloring application look like an excellent candidate for applying "Image-to-

Figure 6: Every hand drawn picture that was used in making the 32Dragons training set

Image Translations with Conditional Adversarial Nets" (Isola et al., 2016). Their facades and city maps applications needed only 400-1100 AB training pairs. So we made that our target range. That's still an ambitious order of magnitude more than our original drawing data and two orders of magnitude more than our hand colored data, but far less than we might have expected.

6. Experiments, Results and Future Research

We had excellent results with uncropped flowers and promising results with dragons from only a few dozen original drawings colored and augmented as described above (all trained with orange/yellow scheme). The models handled several issues our geometric rules could not - flower line gaps, small centers and coloring dragon back limbs but fell short of our aspirations on croppings, background areas near close parts, all yellow spikes and fancy dragon parts.

We ran three experiments on a single AWS GPU instance using Hesse's TensorFlow (Hesse, 2017) implementation of Isola et al.'s Pix2Pix image translation model. We trained on all flowers, all dragons and both dragons and flowers for 200 epochs. We then tested each trained model on a mixed test set of new flowers, dragons and a few others.

Our original 40 uncolored "flower" and 32 uncolored "dragon" (Fig. 6) 400x400 px drawings were made in a standard computer Paint program by the author. We trained for an orange/yellow coloring scheme for both flowers and dragons to see if the similarity between flower centers/petals and dragon bodies/spikes would make a mixed training set useful for dragons.

6.1 Experiment 1: 40Flowers

Training data: 40 original flower drawings, rule colored and augmented, including erasures to 506 AB training pairs.

Results: 40Flowers was very good at coloring uncropped flowers. It could handle all the flowers our geometric rules could and several types they could not - line gaps, small centers and various size petals. With dragons, strongly cropped flowers and "other", it didn't color according to our intended scheme but it often (but not always) recognized lines as color boundaries (Fig. 7).

Next steps: The relative simplicity of the character type and the promising family of homeomorphisms of the disk suggest we could successfully shrink the original flower training set even further and/or improve cropping performance.

Figure 7: Colored by model 40Flowers (trained only on flowers)

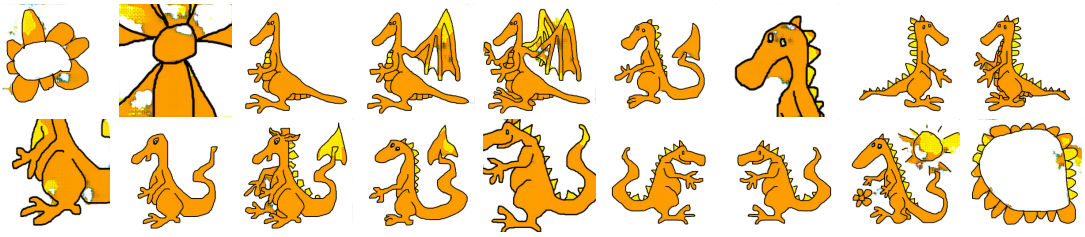


Figure 8: Colored by model 32Dragons (trained only on dragons)

6.2 Experiment 2: 32Dragons

Training data: 32 original dragon drawings, rule colored and augmented to 469 AB training pairs. We added hand erasures or "paint bucket" fills to a half dozen colored drawings to add properly colored back limbs to the training set.

Results: 32Dragons had quite good results on simple dragons, but often colored a few spikes orange and muddled orange and yellow at the tail tip or tail spike, possibly confused by orange ears and the inconsistency of a yellow tail spike's existence in the training set. It did well with the simpler eye style and struggled with the other. It colored background where the gap between body parts was small. It did a lot right with wings, back limbs and "other" pictures but didn't color them quite according to our scheme. It struggled a bit with gaps, going a bit outside the line gap and introducing unwanted black in other places. 32Dragons colored flowers in an unexpected way with mostly white centers and petals mostly orange with white and yellow mixed in. There was often some color in the background. Croppings were worse. Dragon eyes may explain the coloring of flowers since the the round eye is white surrounded by orange body (Fig. 8).

Next steps: Several research directions seem promising. We could use transfer learning to build from simpler to more complex characters (both across character types or within a single type adding body part features) and to incorporate problematic transformations (e.g erasures or croppings.) We could augment a purpose built data set by saving in progress drawings at multiple stages. We could look for additional distortion transformations and investigate methods for drawing automation. Relevant works include Hauberg et al. (2016) for learned diffeomorphisms and Ha and Eck (2017) for sketching with neural nets.

6.3 Experiment 3: MixedFD

Training Data: Combined 40Flowers and 32Dragons training sets 975 AB training pairs.

Results: MixedFD had quite good results on simple dragons, but not as good as 32Dragons. MixedFD miscolored the eyes, probably confused by orange flower centers. MixedFD colored spikes yellow, but at the expense of miscolored ears. MixedFD was quite good on

Figure 9: AI T-shirt designs: unique design for every customer

dragon line gaps and it was a little better than 32Dragons on crops but it introduced even more unwanted black. MixedFD was good on uncropped flowers, including line gaps and various center/petal sizes, but had unwanted black.

7. Industry Applications and Further Research

This research is the first step in a broader inquiry into how to generate new works in the style of an individual artist from limited original data, with industry applications in art, design and animation. Automating the coloring of cartoon characters with a consistent color scheme has potential applications in cel style animation for film and game production and in comics for periodical and book publication, but our target application is more ambitious.

In fashion, interior design and architecture, from high end photography printers to modern looms to CNC steel fabrication equipment, wherever there exists a manufacturing process where it is cheap to switch patterns, there is the possibility of offering products incorporating unique to each customer designs at mass market prices if there is an inexpensive way to create new designs. The challenge is to maintain design style and quality across a large number of items while introducing automation and variation.

For a first use case in wearables, we are currently using the work in this paper together with further work on drawing creation to produce a family of designs for T-shirts where each shirt will have a unique but closely related design. Shirt mock ups shown here (Fig. 9) use test image results from the 32Dragons experiment.

8. Conclusion

We are exploring ways to use machine learning for automated design generation, with applications across multiple industries. To automate cartoon coloring, we suggested a generalizable strategy for going from a very small data set to a deep learning training set, bootstrapping with geometric rules to get to a partial solution and using strategic rule-breaking transformations and other augmentations to go the rest of the way.

The study of a specific application has allowed for a useful discussion of implementation details and specific outcomes, but the underlying ideas generalize beyond the problem of coloring dragons. Up a few levels of abstraction, the strategy's formulation is equally applicable to image, language or numerical data across a wide range of domains. Given any unlabeled data set, if there is a substantial subset with a method for automated labeling, a method for transforming that labeled subset into labeled data like the data we couldn't automate labeling for and other application appropriate, label preserving augmentation techniques (if the original data set was small), then we can combine these methods to create a large labeled training set representing the variation in our original small unlabeled training set (and maybe well beyond).

References

Marcus Bloice, Christof Stocker, and Andreas Holzinger. Augmentor: An image augmentation library for machine learning. *arXiv:1708.03477 [cs.NE]*, 2017.

David Ha and Douglas Eck. A neural representation of sketch drawings. *arXiv:1704.03477 [cs.NE]*, 2017.

Søren Hauberg, Oren Freifeld, Anders Boesen Lindbo Larsen, John W. Fisher III, and Lars Kai Hansen. Dreaming more data: Class-dependent distributions over diffeomorphisms for learned data augmentation. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, volume 51, pages 342–350, Cadiz, Spain, May 2016. PMLR.

Christopher Hesse. Github tensorflow implementation of isola et al.'s pix2pix model, 2017. URL `https://github.com/affinelayer/pix2pix-tensorflow`.

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. *arXiv:1611.07004 [cs.CV]*, 2016.

Ernie Junior. Github implementation of Simard's elastic transformations, 2017. URL `https://gist.github.com/erniejunior/601cdf56d2b424757de5`.

Patrice Simard, Dave Steinkraus, and John Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition*, pages 958–963, Edinburgh, Scotland, UK, August 2003. IEEE.

Wikipedia. List of The Simpsons episodes, 2017. URL `https://en.wikipedia.org/wiki/List_of_The_Simpsons_episodes`.

Proceedings of Machine Learning Research 82 (2017) 11-22, 4th International Conference on Predictive Applications and APIs

Flexible and Scalable Deep Learning with MMLSpark

Mark Hamilton MARHAMIL@MICROSOFT.COM
Sudarshan Raghunathan
Akshaya Annavajhala[*], **Danil Kirsanov**[*], **Eduardo de Leon**[*], **Eli Barzilay**[*], **Ilya Matiach**[*], **Joe Davison**[*], **Maureen Busch**[*], **Miruna Oprescu**[*], **Ratan Sur**[*], **Roope Astala**[*], **Tong Wen**[*], **ChangYoung Park**[*]
Azure Machine Learning
Microsoft New England Research and Development
Boston, MA 02139, USA

Editor: Claire Hardgrove, Louis Dorard and Keiran Thompson

Abstract

In this work we detail a novel open source library, called MMLSpark, that combines the flexible deep learning library Cognitive Toolkit, with the distributed computing framework Apache Spark. To achieve this, we have contributed Java Language bindings to the Cognitive Toolkit, and added several new components to the Spark ecosystem. In addition, we also integrate the popular image processing library OpenCV with Spark, and present a tool for the automated generation of PySpark wrappers from any SparkML estimator and use this tool to expose all work to the PySpark ecosystem. Finally, we provide a large library of tools for working and developing within the Spark ecosystem. We apply this work to the automated classification of Snow Leopards from camera trap images, and provide an end to end solution for the non-profit conservation organization, the Snow Leopard Trust.

Keywords: Spark, Microsoft Cognitive Toolkit, Distributed Computing, Deep Learning, Snow Leopard Conservation

1. Introduction

Deep learning has recently flourished in popularity due to several key factors: superb performance in a variety of domains, quick training time due to the rich counter-factual information contained in the gradient, low memory training footprint of stochastic gradient descent, and the proliferation of automatic differentiation frameworks that allow users to easily define a large space of models without implementing gradient computations. As datasets grow in size, and models become larger and more complex, it becomes increasingly necessary to develop methods to easily and efficiently parallelize the training and evaluation of these models.

In the space of high performance parallel computing, Apache Spark has recently become one of the industry favorites and standards. Its success can be attributed partly to its rapid in-memory architecture, its guaranteed fault tolerance, and its high level functional APIs for machine learning, graph processing, and high-throughput streaming. However,

0. All authors significantly contributed to the library, remaining names in alphabetical order. For more detail see `aka.ms/mmlspark`

Spark, and its machine learning library, SparkML, provide almost no support for deep learning and existing models such as SparkML's word2vec and logistic regression do not leverage modern computational architectures such as Graphics Processing Units (GPUs) or Automatic Differentiation. In this work we aim to fill this gap by integrating the performant and flexible deep learning framework, The Microsoft Cognitive Toolkit (formerly CNTK).

In this work we first provide background on deep learning with the Cognitive Toolkit (section 2.1), Apache Spark (section 2.2), and its machine learning library, SparkML (section 2.3). We then outline the contributions of MMLSpark and documenting the work required to integrate the two frameworks. Section 3.1 documents Java language bindings for the toolkit, and section 3.2 describes the integration with SparkML, and our performance optimizations (section 3.3). Next, we briefly overview our integration of OpenCV (section 3.5), and our tool to automatically integrate this work into PySpark and SparklyR (section 3.6). In section 4 we investigate the scalability of this method. In section 5 we document our work with the Snow Leopard Trust, and demonstrate a large increases in accuracy and speed over traditional ML methods for automated Snow Leopard identification from remote camera trap images.

2. Background

2.1 Deep Learning with the Cognitive Toolkit

Deep learning has steadily gained popularity throughout the past several years due to a variety of complementary reasons. First and foremost, deep learning has shown incredible performance in almost all domains it has been applied to from speech processing Deng et al. (2013), translation Bahdanau et al. (2014), image classification He et al. (2016a), and probabilistic modeling Denton et al. (2015). This stellar performance has arose in part to significant advances in network optimization, primarily arising from advances in gradient based methods and computations. Networks often take the form of differentiable functions with latent parameters, optimized using empirical risk minimization. This gradient acts as rich source of counter-factual information, effectively allowing the optimizer to explore and understand the local neighborhood of a network's parameter space with only a single function and gradient computation. In recent deep learning frameworks such as the Cognitive Toolkit, Tensorflow, and Caffe, the calculation of the gradient has been abstracted away. This significantly simplifies code and simultaneously allows for automatic compilation of gradient computations to GPU architectures. It is now easier than ever to develop performant and novel network architectures using these automatic differentiation libraries.

The Cognitive Toolkit (CNTK) is an open source deep learning framework created and maintained by Microsoft. It is written in C++, but has "bindings" or interfaces in Python, C♯, and a domain specific language called BrainScript. In the Cognitive Toolkit, users create their network using a lazy, symbolic language. This symbolic representation of the network, and its gradients, can be compiled to performant machine code, on either a CPU or a GPU. Users can then execute and update parameters of this symbolic computation graph, usually using gradient based methods. The toolkit differentiates itself from other deep learning frameworks as it supports dynamic networks through sequence packing Seide (2017), and model compression for low latency operations. We have chosen the Cognitive Toolkit as our first deep learning framework to integrate with Spark because of its superior

performance relative to other deep learning frameworks, and its first class treatment of parallel processing. Furthermore, the Cognitive Toolkit has well formed abstractions for training and reading data from a variety of sources that we plan to leverage in future optimizations of the work.

2.2 Apache Spark

Apache Spark is an open source distributed computing platform that generalizes and combines map-reduce and SQL style parallelism into a single a functional language. Apache spark vastly outperforms conventional map-reduce algorithms on platforms like Hadoop, because of its heavy usage of in-memory computing, and its sophisticated Catalyst query optimizer Armbrust et al. (2015). Like CNTK, Spark exposes its core functionality to other languages using bindings. Currently, Spark has bindings in Python (PySpark), Java, R (SparklyR), C♯ and F♯ (Mobius). Users of Spark construct lazy, symbolic representations of their parallel jobs using a functional language built in Scala. This symbolic representation can then be optimized and compiled to custom parallel code that minimizes inter-node communication and time intensive IO operations. Spark operations are completely fault-tolerant and large numbers of nodes can be killed without effecting the overall status of the job. Furthermore, Spark clusters support elastic workloads with a feature called "Dynamic Allocation". In this mode, nodes can be added or removed from a job depending on cluster utilization. This allows for computations to scale adaptively to take full advantage of a cluster if necessary or to free up costly unused resources.

A central abstraction in the Spark ecosystem is the "DataFrame", or a distributed table of "Rows". Rows are type safe containers that can hold various forms of data. Each DataFrame has a pre-defined schema, or collection of meta-data that can be used to verify the correctness of a computation before execution, similar to a type system. Roughly, each node of the cluster contains a small subset, or "partition" of a DataFrame's rows, and computations are performed on each machine's subset of the data. For more complex computations, Spark can perform "shuffles" or transfers of data between nodes. Furthermore, within the same DataFrame API, Spark supports high throughput streaming work-flows, allowing for computations on thousands of records per second with sub-second latencies.

2.3 SparkML

SparkML is a library built on top of Spark's DataFrame API that gives users access to a large number of machine learning, and data processing algorithms. SparkML is similar to other large machine learning libraries such as python's SciKit-learn, but is significantly more flexible. For example unlike SciKit-learn, SparkML supports multiple typed columns and has a rich type system allowing for static validation of code. Furthermore, it can scale almost indefinitely.

SparkML's central abstraction is a PipelineStage, or a self contained unit of a data science workload that can be composed with other units to build large custom Pipelines for a given machine learning task. This central class is split into two subclasses: Estimators, and Transformers. Estimators abstract the components of the machine learning work-flow that learn, or extract state, from a dataset. Estimators can be fit to a DataFrame to produce a Transformer. A Transformer is a referentially transparent function that can transform a

DataFrame into another DataFrame, often using the state learned from the fitting process. This neatly separates the tasks of machine learning into training and evaluation and can be useful to ensure that one is not learning from a testing dataset. As a concrete example, one can consider PCA as an Estimator that finds the principal components using a training dataset, and returns a PCA transformer, or an object that can transform a testing dataset into the space whose basis is the principal component vectors learned from the training dataset.

SparkML supports conventional classifiers and regressors such as Logistic Regression (LR), Support Vector Machines (SVMs), Boosted Decision Trees (BDTs), and Generalized Linear Models (GLMs). Spark also supports unsupervised methods such as clustering, principal component analysis (PCA), Latent Dirichlet Allocation (LDA), word2vec, and TF-IDF weighting. However, SparkML currently does not support deep networks, or Kernel methods, which poses a significant challenge to those looking to use the powerful methods from these fields.

3. Methods

3.1 The Cognitive Toolkit in Java

In order to integrate the Cognitive Toolkit and Spark, we first need them to "speak the same language". One way to accomplish this is to use CNTK's python bindings and Spark's python bindings called PySpark. However, this approach suffers from the unnecessary baggage of the python interpreter, which can considerably slow operations as input and output data needs to be transferred from the JVM to the python interpreter, and the compute context needs to be switched from Scala to python and back again after execution.

To avoid the overhead of the python interpreter, we focused on bringing CNTK to Scala by creating Java language bindings for CNTK. To bring CNTK into Java, we used the Simple Wrapper and Interface Generator (SWIG) Beazley et al. (1996). This tool exposes C++ code in a wide variety of languages such as Java, C♯, OCaml, python and many more. SWIG first inspects a C++ file, and translates each programming construct to an equivalent construct in the target language. For Java, each C++ function translates to a corresponding Java function, and each C++ class translates to a Java class. This mapping is controlled by a specification called an "interface file". In this file one can define custom maps from one type system to another, inject custom code for usability of the generated bindings, and programmatically control translations with a basic macro system. To translate C++ to Java, many of the C++ functions are wrapped using the Java Native Interface (JNI), a language feature that allows Java code to call into native C and C++ code efficiently. This results in performant C++ code that has look and feel of a normal Java function. The default bindings SWIG produces are generally not very elegant, as complex language features such as generics are lost in translation. To mitigate this we have contributed a custom interface file that makes these Java bindings as idiomatic as possible and have contributed this binding generation process back to the Cognitive Toolkit. We also publish a Java Archive (JAR) of the bindings, complete with automatic loading of native libraries, to Maven Central for every CNTK release. This is the first time CNTK has been brought into a virtualized language, and will enable deep learning on a large class of new devices running on the JVM. Furthermore, this work enables research at the intersection of type

theory and deep learning, allowing for the possibility of richer abstractions when working with CNTK computation graphs.

3.2 The Cognitive Toolkit in Spark

To bring our CNTK Java bindings into Spark's Scala ecosystem, we relied on Scala's seamless inter-operation with Java. More specifically, every class in Java is treated as a type in Scala, and Scala preserves all of Java's complex language features. Effectively, we get CNTK Scala bindings for free! We hope that other languages on the JVM will be able to leverage these bindings similarly. This brings CNTK to Scala in an idiomatic and efficient way.

To bring CNTK to Spark, we leverage Spark's ability to distribute and execute custom Scala code inside of a mapping operation. More specifically, we use the "mapPartitions" function that enables parallel evaluation of a custom Scala function on each partition. This is strictly more general than Spark's "map" operation as the function can depend on *all* of the rows in a partition. It is common to use this pattern to move slow, setup-type, operations outside of the main loop of the map to improve performance. Using MapPartitions we can take advantage of CNTK's Same Instruction, Multiple Data (SIMD) optimizations that perform computations on a large number of data concurrently, potentially on a GPU. This speeds performance significantly, on both the CPU and GPU.

We built on this MapPartitions operation and create a SparkML transformer, called the "CNTKModel", that distributes, loads, and executes arbitrary CNTK computation graphs inside of SparkML pipelines. We expose parameters that give users control over the CNTK model object, mini-batch size, and evaluation of subgraphs of the CNTK graph. The ultimate is useful for transfer learning, where one is interested in inner activations of a network Bengio (2012)

3.3 Optimizations

In order to improve performance in streaming and repeated evaluation scenarios we add several memory and performance enhancing optimizations. Initially, we loaded the CNTK model on each partition, and evaluated the model on every transformation. In Spark, each node can hold multiple partitions, so this wastes time and memory on loading the model multiple times on each node. To improve this, we used Spark's system for broadcasting data from the driver node to each of the worker nodes using bit-torrent communication to minimize overhead.

3.4 Multi-GPU Training

In addition to parallel evaluation of deep networks, we also provide several solutions for rapid training of CNTK models. CPU Spark clusters excel at large scale extract, transform, and load (ETL) operations. However these clusters lack GPU acceleration and InfiniBand support, which degrades the performance of high communication network training. As a result, we have investigated hybrid systems consisting of dedicated multi-gpu machines, tethered to the Spark cluster over a virtual network. These systems use several CNTK

distributed training modes such as the highly optimized 1-bit SGD Seide et al. (2014). We provide Azure Resource Manager templates so that users can deploy these solutions.

3.5 Integrating OpenCV

SparkML also lacks support for complex datatypes such as images. We have worked with the Spark community to contribute a schema for image data-types, and a collection of readers for ingesting and streaming images and binary files to the core Spark library. Building on this work, we have integrated the popular image processing library OpenCV Bradski and Kaehler (2000) as a SparkML transformer. Like the CNTKModel, the OpenCV transformer uses Java bindings for OpenCV that call into fast native C++ code. Our OpenCV transformer allows us to perform most OpenCV operations in parallel on all nodes of the cluster. Furthermore integration automatically chains all evaluations of native code so that data does not need to be Marshalled to and from the JVM between adjacent calls to OpenCV.

3.6 Automated Generation of PySpark

Spark supports a wide variety of ecosystems through language bindings in Python, R, Java, C♯ and F♯. However, wrapper code for these libraries is mainly written "by hand". This a difficult task involving detailed knowledge of both the source and target ecosystems, and their communication platforms. Furthermore, this results in a large amount of boilerplate code, that impedes adding estimators or transformers to the spark ecosystem. To eliminate the need to manually create and maintain these language bindings, we have created an automated wrapper generation system. The system loads and inspects the types and parameters of any transformer or estimator using Scala reflection. It then generate a usable and idiomatic python wrapper for the class that can be used in PySpark. We also use this tool to create language bindings for the SparklyR ecosystem. We expose all of our estimators and transformers to these languages, proving the generality and reliability of this system. A schematic overview of the process is shown in Figure 1.

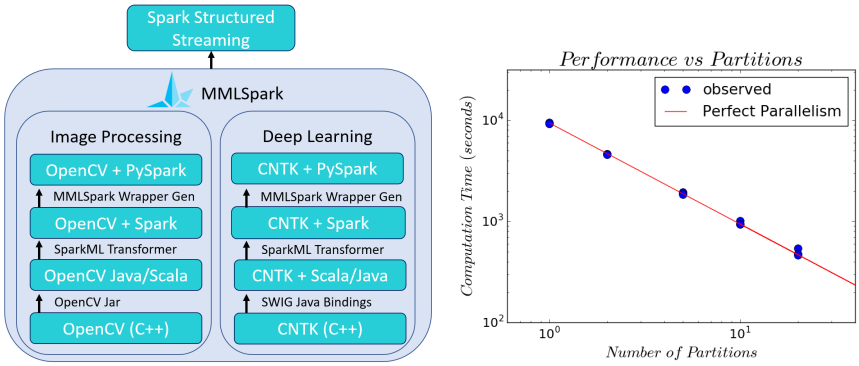


Figure 1: Left: Architectural overview of some salient features of MMLSpark. Right: Performance of the "ImageFeaturizer" transformer that contains an OpenCV image resize, and deep featurization with ResNet-50

3.7 Additional Contributions

In addition to the aforementioned contributions to the Spark and CNTK libraries, we provide the following additional contributions to SparkML:

- An estimator for automatic featurization, so that users can immediately begin experimenting with learning algorithms.
- An infrastructure for the automated testing of SparkML estimators for serialization, fuzzing, python wrapper behavior, and good design practices.
- A single estimator for high performance text featurization.
- A transformer for deep network transfer learning on images.
- A high availability content delivery network of state-of-the-art pre-trained deep networks. This delivery system has a Scala and Python API for querying and downloading with caching and checksumming.
- Several new pipeline stages for performing Spark SQL operations, repartitioning, caching. These help close many gaps in the SparkML pipeline AP.
- A high performance vector assembler that considerably outperforms SparkML's.
- A series of Jupyter notebooks documenting and demonstrating our contributions on real datasets.
- An estimator for image dataset augmentation with OpenCV transformations.

4. Performance and Scalability

We test our experiment with on a dataset $82,382$ real images of various sizes (upwards of 400×600 pixels) encoded as JPEGs. The experiment reads data from "WASB" storage, a HDFS client backed by Azure Blob storage, then passes them through our ImageFeaturizer transformer. The ImageFeaturizer internally pipelines the data into OpenCV to scale it down to the proper size and unroll the scaled image into a vector. The transformer then feeds the result through ResNet-50, a 50 layer deep state of the art convolutional network with recurrent connections. The computation time as a function of the number of executors in the spark cluster is shown in Figure 1. We have tested this pipeline with up to 20 executors, but have observed similar scaling behavior beyond that point. As a rule of thumb, scaling behavior tends to improve with larger datasets.

5. Experimental Validation

We have collaborated with the non-profit organization The Snow Leopard Trust to help them develop an automated system for organizing their extremely large database of camera trap images. This is not only a novel scientific experiment, but also validates the library and ensures that our abstractions work well in practice.

5.1 Snow Leopard Conservation

Snow leopards are a rare species of large cat found throughout the mountains of northern and central Asia. These large cats are one of the top predators of this ecosystem, but are incredibly rare with only $3,000 - 7,000$ cats spread out across 1.5 million square kilometers.

These creatures are endangered, and are constantly threatened by loss of habitat through mining and climate change, poachers, and retribution killing. Despite their endangered status, relatively little is known about snow leopard behavior, ecology, population numbers, and population trend. These data are incredibly important for understanding the risk of snow leopard extinction and where to target conservation efforts. Furthermore, robust habitat information is necessary in order to lobby for legal protection of habitat.

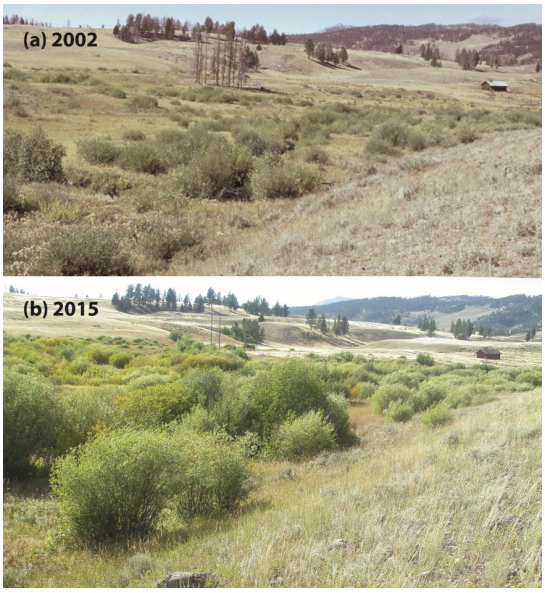


Figure 2: A comparison of vegetation in Yellowstone National Park before (above) and after (below) the reintroduction of the Grey wolf. Figure originally from Beschta and Ripple (2016)

Snow leopards occupy a critical niche in the Himalayan ecosystem as one of the apex predators. Apex predators have been shown to help stabilize ecosystems, by naturally controlling populations of secondary lifeforms in the ecosystem. This allows for primary life forms, such as plant life, to thrive and actively improves soil stability. Added vegetation can introduce several new niches, increasing biodiversity and ecosystem stability. Stunning evidence of this effect can be seen in Figure 2, which shows the effect of re-introducing the Grey wolf to the Yellowstone ecosystem in the early 2000's. The health and stability of the Himalayan ecosystem is critically important as it provides water for for over 1.4 billion people Immerzeel et al. (2010).

Surveying snow leopards is an extraordinarily difficult task. This task is made difficult because leopards are incredibly rare, live in remote and difficult to reach areas, and are mainly nocturnal. As a result, the Snow Leopard Trust uses remote camera traps to survey snow leopard locations over 1,700km^2 of potential habitat. These camera traps are equipped with regular and night vision settings, and take photographs when an attached motion sensor is triggered. These cameras generate a large number of images that need to be manually sorted to find the leopards. For many images, it is difficult for a human to identify hidden leopards, as they are often partially obscured and highly camouflaged. This task can take up to 20 thousand man hours per each million images. This task is insurmountable for most small nonprofits like the Snow Leopard Trust. Hence, it is critically important to automate this task.

5.2 Dataset

We use a dataset from Snow Leopard Trust that contains roughly $8,000$ labeled images of snow leopards and non snow leopards as well as roughly $300,000$ unlabeled images. These images are from their most recent set of surveys in the last 5 years. The images are primarily black and white night vision and optical images. A small number 5% of images are in full

color. Only around 10% of the images are Snow Leopards. The majority (90%) of the images are of grass, goats, foxes, and occasionally people. We split the labeled dataset into a training (80%) and testing (20%) set based on the cameras used in the analysis. Splitting the cameras into training and test sets is necessary, as subsequent frames in a camera burst are highly correlated. If one splits a single burst into both sets, information will leak from the training set, and the final testing performance will be overestimated. We did not have to "clean" the dataset, or remove any images, as anything that did not contain a snow leopard was marked as a negative image. At the time of publication, we use MMLSpark version 0.10, Spark version 2.2 and CNTK version 2.3.

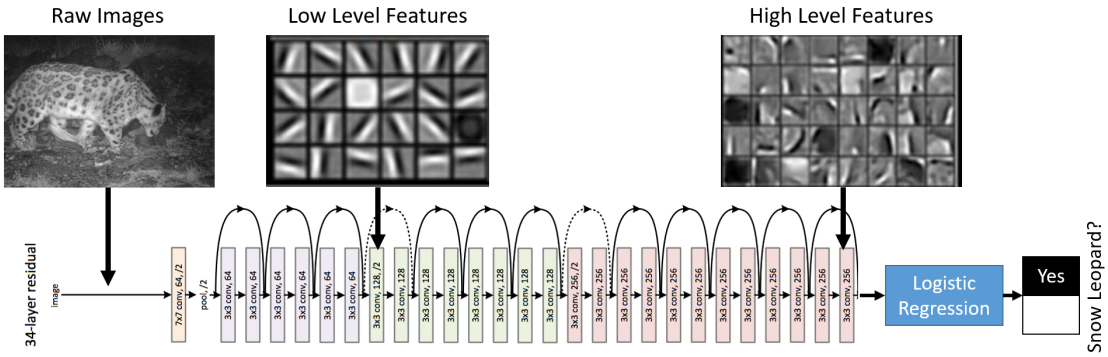


Figure 3: Schematic overview of basic transfer learning algorithm. We use ResNet50 and truncate the last layer(s). We then append a SparkML logistic regression estimator and train the model using the features from the preceding network. Feature visualizations originate from a similar convolution architecture by Lee et al. (2011).

5.3 Model and Results

To learn an automated classification system, we leverage deep transfer learning to learn a classifier that can successfully identify snow leopards with high precision and recall. We improve the classifier through the use of dataset augmentation and ensembling leopard probabilities over bursts of images. The simplest model uses features from ResNet50, trained on the ImageNet corpus. ResNet50 is a state of the art network that combines convolutional and recurrent components to capitalize on the spatial regularity of natural images and effectively learn the optimal depth of the network He et al. (2016b). We truncate the last, image-net specific layer(s) but keep a majority of the early layers. This lets us leverage the domain-general low to mid level features and adapt the network to the new classification task. To adapt the network, we append and train a SparkML logistic regressor that maps the features to a binary label. This effectively adds an additional fully connected layer to the truncated network. A schematic overview of the algorithm is provided in Figure 3. We refer to these algorithms as "RN1" and "RN2" for ResNet50 with 1 and 2 truncated layers respectively.

To improve performance of the algorithm, we augment the dataset using MMLSpark's "ImageSetAugmenter" transformer. This Transformer enriches the dataset by flipping the images about the horizontal axis. The ImageSetAugmenter doubles the training data, which

improves the learned classifier. At test time, the scores for both image parities are averaged together. This reduces the variance of the final estimate and improves performance. We refer to this algorithm as "RN2+A" for the base algorithm with dataset augmentation.

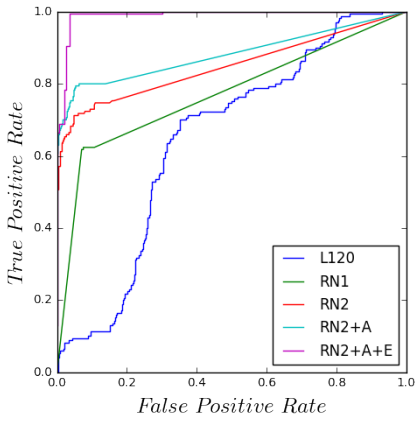


Figure 4: Receiver Operator Characteristic (ROC) curves for each algorithm. Note the superior performance of "RN2+A+E" to all other methods tested

Finally, we leverage the natural grouping of camera trap images into in small temporally contiguous segments we refer to as "bursts". We build on Spark's primitives to implement more complex inter-node operations like grouping images by keys. These bursts originate from camera traps that take a series of images until no motion is detected. We extract this burst information from image metadata, and use it to inform our predictions. This effectively turns a single image classifier into a video classifier. We note that labels are often the same for all frames in camera bursts. By averaging the predictions of each image in a burst, we dramatically reduce the variance of the classifier and allow the algorithm to use easy classifications to inform neighboring difficult classifications. This substantially reduces the number of false negatives, and improves overall accuracy. We refer to this algorithm as "RN2+A+E" for the base algorithm with dataset augmentation and sequence ensembling. We compare these models to logistic regression trained on the raw pixels of the image after scaling to 120×120 pixels. We refer to this baseline model as "LR120". Figures 5 and 4 show the dramatic performance increases from deep featurization, augmentation, and ensembling.

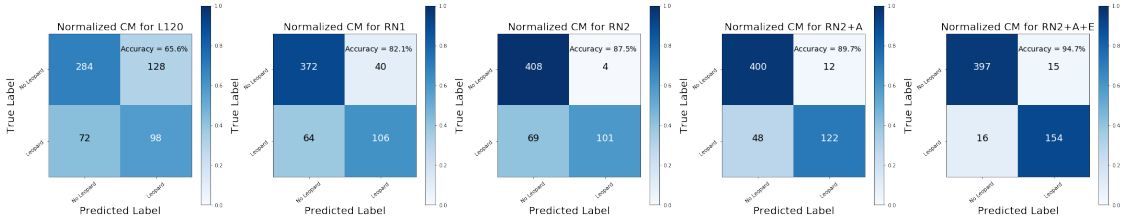


Figure 5: Normalized confusion matrices for each algorithm.

6. Future Work

MMLSpark is a growing software ecosystem with several large ongoing areas of work. In the future, we plan to extend the CNTK Java bindings to allow for training in Java. This will allow users to construct CNTK computation graphs in Java and Scala, and will provide a way to leverage the parallelism of the spark cluster for CNTK model training in addition to evaluation.

Furthermore, we plan to implement a recent advance in low communication parallel stochastic gradient descent called SymSGD Maleki et al. (2017). This will dramatically re-

duce communication overhead during parallel training of networks. In addition this improve the scalability of parallel training systems as SymSGD's method of gradient combination approximates a sequential computation. This contrasts other methods like AllReduce, ParameterServer, or Hogwild! that principally benefit from reducing the noise of the gradient. These methods see rapidly diminishing marginal returns as the number of workers increases Maleki et al. (2017).

7. Conclusions

We have created an architecture for embedding arbitrary CNTK computation graphs and OpenCV computations in Spark computation graphs. This architecture inter-operates with the majority of the Spark ecosystem and enables fault tolerant deep learning at massive scales. We also successfully translate this and the rest of the MMLSpark ecosystem into PySpark and SparkR, enabling scalable deep learning in a variety of languages. We leverage this framework to create a state-of-the art snow leopard detection system for the Snow Leopard Trust. This work lays the groundwork for future deep learning extensions to the Spark ecosystem and enables convenient cluster based solutions for a broader class of machine learning problems.

Acknowledgments

The authors would like to thank the Snow Leopard Trust, specifically Rhetick Sengupta and Koustubh Sharma, for their continued cooperation, inspiration, and dataset. Thanks to Patrick Buehler for connecting the MMLSpark team with the Snow Leopard Trust and for creating an initial POC of a snow leopard detector. In addition, we thank the CNTK team, specifically Mark Hillebrand, Wolfgang Manousek, Zhou Wang, and Liqun Fu for their continued support around SWIG and the CNTK build system. Finally, without the hard work and resources of the Azure Machine Learning team and Microsoft, this project would not have been possible.

References

Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

David M Beazley et al. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Tcl/Tk Workshop*, 1996.

Yoshua Bengio. Deep learning of representations for unsupervised and transfer learning. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, pages 17–36, 2012.

Robert L Beschta and William J Ripple. Riparian vegetation recovery in yellowstone: The first two decades after wolf reintroduction. *Biological Conservation*, 198:93–103, 2016.

Gary Bradski and Adrian Kaehler. Opencv. *Dr. Dobbs Journal of Software Tools*, 3, 2000.

Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, et al. Recent advances in deep learning for speech research at microsoft. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8604–8608. IEEE, 2013.

Emily L Denton, Soumith Chintala, Rob Fergus, et al. Deep generative image models using a laplacian pyramid of adversarial networks. In *Advances in neural information processing systems*, pages 1486–1494, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016b.

Walter W Immerzeel, Ludovicus PH Van Beek, and Marc FP Bierkens. Climate change will affect the asian water towers. *Science*, 328(5984):1382–1385, 2010.

Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. Unsupervised learning of hierarchical representations with convolutional deep belief networks. *Communications of the ACM*, 54(10):95–103, 2011.

Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Parallel stochastic gradient descent with sound combiners. *arXiv preprint arXiv:1705.08030*, 2017.

Frank Seide. Keynote: The computer science behind the microsoft cognitive toolkit: An open source large-scale deep learning toolkit for windows and linux. In *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*, pages xi–xi. IEEE, 2017.

Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

Proceedings of Machine Learning Research 82 (2018) 23-32, 4th International Conference on Predictive Applications and APIs

An Architecture and Domain Specific Language Framework for Repeated Domain-Specific Predictive Modeling

Harlan D. Harris HARLAN@HARRIS.NAME
New York, NY

Editor: Claire Hardgrove, Louis Dorard and Keiran Thompson

Abstract

When repeatedly fitting related predictive models within the same domain, for similar problems, it's helpful to have tools to support an efficient, high-quality workflow. This paper describes a theory of the architecture for such tools and for the interfaces among predictive models and other aspects of a software system. Additionally, it describes an open-source reference implementation of this design, written in R, focusing on a Domain Specific Language for one specific repeated predictive modeling task.

Keywords: Domain Specific Languages, Machine Learning, Software Engineering, Software Architecture

1. Introduction

For data scientists in certain industry roles, especially in SaaS firms, a common problem is the need to fit not just one predictive model, but many related, yet independent models. For the purposes of this paper, consider the hypothetical Acme Systems, a Software as a Service (SaaS) firm that provides applications used by large retail enterprises. One such applications incorporates a model of sales at chain stores—the Acme Chain Model. This model has been sold to many dozens of clients, with variants on the same basic model template.

Although the general problem is similar—sales are probably due to trends, business cycles, pricing changes, etc.—each company the firm works with will have different data, different inventory, and different patterns of cause and effect. Acme's data scientists, then, have to build and integrate custom predictive models for each client. To maintain quality, consistency, and timely fitting and re-fitting of these models, Acme data scientists use a set of domain-specific workflow tools designed to allow the team to efficiently build and deploy models. Importantly, a well-designed workflow can let them apply their growing domain knowledge to solve the problem better and more efficiently at each iteration.

This pattern applies under a specific set of circumstances, which I'll call Repeated Domain-Specific Modeling:

1. The predictive model is a component of a broader software product, with predictions likely being exposed to end users to help them make better decisions.

2. The same general problem is being solved many times, as when a SaaS firm builds separate models per customer.

3. The models are similar enough for investment in shared infrastructure makes sense, but are different enough to require data scientist customization and insight.

This paper has two primary contributions. First, I describe a set of abstractions for thinking about Repeated Domain-Specific Modeling. These abstractions clarify important design choices about representations and interfaces. Second, I provide and describe an implementation of this pattern as an open-source software repository, written in R (R Core Team, 2017). The code[1], available at `https://github.com/HarlanH/featgen-demo`, shows how to use a problem-specific Domain Specific Language (DSL) to describe individual instantiations of a predictive modeling problem, allowing the domain expertise of data scientists to be incorporated, while yielding a highly efficient workflow.

1.1 Related Work

Aspects of this work have been informed by contributions from several sources. There are many workflow tools for Predictive Modeling, both in general and for specific problems. Those tool sets often include APIs for integration with other systems. Commercial examples include SAS, Domino Data Lab, and Azure Machine Learning Studio, while many organizations have built their own solutions using open-source and in-house technologies.

There are several prominent examples of DSLs for Machine Learning *in general*, such as `scikit-learn` and its Pipeline for Python (Buitinck et al., 2013), `mlr` and `caret` for R (Bischl et al., 2016; Kuhn, 2008) and several of the deep-learning frameworks such as TensorFlow (Abadi et al., 2016). (See also Portugal et al., 2016) Although these provide a clean, powerful short-hand for creating predictive models, they are DSLs for the general problem, typically of supervised learning, not for specific business problems.

One public example of Repeated Domain-Specific Modeling was described recently by Uber. Their Michaelangelo framework (Li et al., 2017) allows their team to define high-level feature extractors (what I'll describe as Feature Transformer Generators, below) using a Scala-based DSL, add them to a common Feature Store, and easily re-use and extend them. This framework is specific to Uber and their domain, but shares commonalities with the work presented here.

Some of the theory described below extends work originally presented by (Morra, 2016). Their recently-described and open-sourced Aloha framework (Deak and Morra, 2018) is also a system for Repeated Domain-Specific Modeling. Aloha includes a feature-definition DSL with abstraction layers that separate data extraction from transformation, and a strong focus on the appropriate boundaries between systems.

2. Theory

In theory, machine learning systems are mathematical models mapping between arbitrary representations; in practice, machine learning systems are continuously interacting with

1. Some of the patterns described here were explored by me and my colleagues when employed by The Advisory Board Company (Washington, DC). None of the details of the specific model are relevant here, and all of the open-source code was written entirely from scratch. In practice, the system we built allowed us to effectively create customized models at the rate of one per business day, which had a substantial positive effect on the business.

other systems, human and technical. These interactions, and the interfaces between machine learning systems and other systems, are critically important to the overall effectiveness and value of what's been built.

To build a predictive model, you have to be able to represent entities in the world, such as the properties of the object or event you're predicting, and the outcome or label you wish to predict. Additionally, as a data scientist, you must represent and incorporate your own knowledge into the system. The No Free Lunch theorem (Wolpert and Macready, 1997), which proves that no predictive model can be optimal at all problems, implies that algorithm choice alone, by aligning the underlying problem with the algorithm's Inductive Bias, will have a substantial impact on model quality. Further, feature engineering, or the process of transforming data to make it more usable and useful to the learning algorithm, is a key art in predictive model development (Domingos, 2012).

Repeated Domain-Specific Modeling systems include several parts—the application, the predictive model, the data scientists who will be operationally responsible for keeping the predictive model tuned and accurate, and the users of the application. (See Figure 1) Communication among these parts, be they human-machine interfaces or machine-machine interfaces, must be efficient and consistent for the system as a whole to run smoothly.

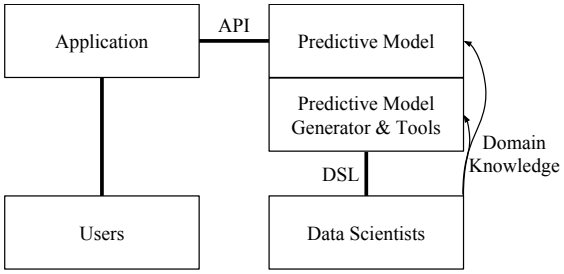


Figure 1: Important components of a Repeated Domain-Specific Modeling system include a well-designed API interface between the model and the main application, a well-designed DSL for efficient expression of the model to the model generator by data scientists, and a modeling framework that allows data scientists to insert their domain knowledge into both the model generating framework and individual models.

2.1 Representations at the Machine Learning and Application Interface

Consider the interface between a machine learning system and a broader software system that interacts with it. The application provides data and entities to be predicted to the machine learning system, then receives a prediction or score, to be provided to a user or used for a decision.

Several well-known ideas from software engineering and systems design apply. First, *Separation of Concerns* says that subsystems should do one thing well, encapsulating processing, and hiding irrelevant details from other subsystems. Second, the *Standard Service Contract* principle from the theory of Service Oriented Architectures (SOAs, Erl, 2007) says that subsystems need to define a commonly-understood language for communication, and

should be clear about what each subsystem should expect from another. Third, Conway's Law (Wikipedia, 2017) says that system structure will inevitably reflect team structure. When data scientists are building predictive systems, this very likely means that a predictive service will be a separate module or service in the overall system.

An implication of these ideas is that the application need not, and should not, know about the internal data processing of the predictive model. Specifically, the application should not know about *feature engineering*, the process of re-representing an entity to be scored in such a way that a predictive model can most effectively make predictions.

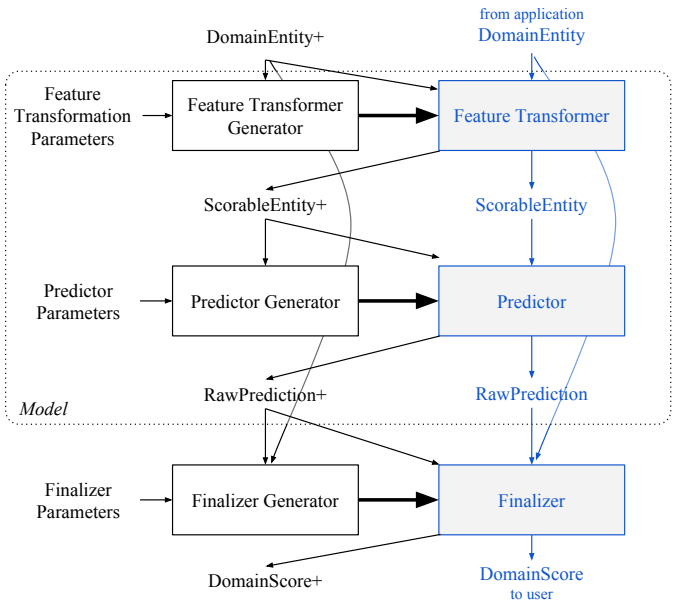


Figure 2: Representations for Repeated Domain-Specific Modeling systems. (Left) Training data flows through a training pipeline, generating both training predictions and a scoring system. (Right) Data to be scored goes through the same translations steps, using the scoring system inferred by the training data. The components within the dotted box are part of the predictive model; outside the box is the application.

Figure 2 describes a framework for thinking about the changes in representation involved in predictive modeling. (See also Morra 2016 and Deak and Morra 2018.) Predictive modeling is essentially two linked processes, Training and Scoring. Importantly, they share a parallel structure with regards to the representations that flow through their processes. And equally importantly, the Training process can be thought of as a system, or a higher-order function, that *generates* the Scoring process as an output of its data processing.

To maintain encapsulation, the application should provide Training and Scoring data to the model in an (agreed upon by contract) format that reflects the semantics and structure used by the application—a `DomainEntity` in Figure 2. The application might provide an API for serving `DomainEntity` objects, while the model might provide an API for real-time

scoring. There are a number of frameworks, commercial and open source, for exposing predictive models as API services. See the demo code for a simple approach.

2.2 Efficiently Contributing Expertise

Most readers will be aware of Drew Conway's influential Venn Diagram of Data Science (Conway, 2013). The oft-neglected third set in that diagram is "substantial expertise." When building a Repeated Domain-Specific Modeling system, it is important that data scientists be able to leverage and incorporate their substantial expertise about the problem as effectively as possible. This need exists at three levels of system building.

1. Data Scientists must be able to *contribute to their own workflow.* Nobody is more qualified to design and build tools and systems for predictive model generation than data scientists. This includes critical components such as tools for data and model validation. (See Harris 2016; Parker 2017.)

2. Data Scientists must be able to *add their domain expertise to the model-generation process.* In a repeated model-building scenario, it's important that general facts that data scientists learn about the domain be represented in the model-generation process. For instance, if you learn that seasonality and holidays are relevant for retail sales, the model-generation framework should directly represent those concepts in the system.

3. Data Scientists must be able to *add their domain expertise to individual models.* In a repeated model-building scenario, it's also important that facts that data scientists learn about specific examples of the domain be easily incorporated into the model. These facts might include data-quality issues or exceptions to general rules.

3. Implementation

As noted above, I have developed an open-source example of what this framework might look like. Most of the steps that might be used in a production environment are included: defining a model with a DSL, fitting a model, viewing a model archive file, generating a standard report about the model, scoring the model via web service, and inspecting the model in production. Two other important steps, tools for interactively validating and exploring the source data, and interactively validating and comparing candidate models, have not been implemented. But see (Harris, 2016) for approaches to those tools.

I will focus the rest of this paper on how a model is defined with the DSL. Readers interested in the other components are encouraged to explore the provided software.

3.1 Domain Specific Language

Listing 1 shows a sample model definition file for a company called Rossman Stores (See Kaggle.com, 2015). The model definition is written using an internal DSL (Fowler, 2010), leveraging the syntax and execution model of an existing language, in this case R. To train a model, the user uses the command line: `acm.R train rossman.R rossman.Rout`. The `acm` executable loads the `rossman.R` script shown and executes it in a context where functions such as `acme_chain_model` and `promo` are available. This is the only code written specifically for Rossman.

```
1  acme_chain_model("rossman") %>%
2    store_type(collapse=list(ab=c("a", "b"))) %>%
3    competition_distance(trans='boxcox', na='median') %>%
4    one_week_ago_sales(cap_to=c(2000,15000)) %>%
5    sales_trend(days_back=4*7, trunc_to=c(-100, 100), na='mean') %>%
6    promo %>%
7    current_sales %>%
8    get_data %>%
9    train(target="current_sales")
```

Listing 1: Sample model-definition file, `rossman.R` in the sample code, for an instantiation of the "Acme Chain Model" for client "Rossman." Defines a human-computer interface allowing efficient specification and customization of the model template for individual clients. The `%>%` pipeline operator, defined in the `magrittr` package (Bache and Wickham, 2014), unrolls nested function calls, passing the results of the previous function as the first argument to the next.

Line 1 executes a function that generates an object, initially with only the name of the customer. (See Figure 3A.) Each subsequent line calls a function with that object, adding additional elements to the object, such as feature generators (Lines 2–7), or the data (Line 8) or the model itself (Line 9). The result of running this script is a model object that gets saved to a file for further processing, such as inspection or deployment.

There are many details that make this DSL powerful, including a mix of feature-specific and general data transformations. (Note that the code in Figure 3B is specific to the Acme Chain Model, while C, D, and E are not.) Next, I'll describe the Feature Transformer Generator (FTG) pattern (see Section 2.1, above), and a lower-level but analogous pattern used to handle missing data.

3.2 Feature Transformer Generators

Figure 3B shows the implementation of the `promo` feature. Importantly, the design of this Acme Chain Model DSL uses FTGs that are based on the semantics of the problem, rather than on the structure of the data provided by the application. This allows the data scientist tuning this model for Rossman Stores to think about their promotion history and policies, avoiding implementation details.

On the other hand, the fact that this FTG is short, and is written in the same language as data scientists use day-to-day, means that people can switch easily from operations—using the framework to fit models—to development—extending the framework based on their accumulation of domain knowledge. This is an example of how this framework allows data scientists to incorporate their expertise at various points in the process.

Figure 3B illustrates the basic FTG pattern. A feature object ("`feat`") is created with standard slots. Line 9 allows the user to add standard parameters to the object, such as missing data handling rules. Line 10 adds the new Feature Transformer (FT) to the model object's list of features, then the function returns with an updated version of the model object. The key logic is Lines 6–8, defining an FT that extracts a `promo` ScorableEntity

```
# A -- the model object structure
> str(model,1)
 $ custname: chr "rossman"
 $ features:List of 6
 $ data    :Classes tbl_df, tbl and 'data.frame': 1115 obs. of 6 variables:
 $ cv_preds:'data.frame': 1115 obs. of  5 variables:
 $ metrics :List of 1
 $ target  : chr "current_sales
 $ model   :List of 8
```

```
# B -- an example of a Feature Transformer Generator
promo <- function(x, ...) {
    feat <- list(
        name = "promo",
        pretty_name = "Promo",
        extract = function(self, data, ...) {
            data_frame(promo=data$activity[[length(data$activity)]]$Promo)
        })
    feat <- list_modify(feat, ...)
    x$features <- list_modify(x$features, promo = feat)
    x
}
# C -- generating ScorableEntity objects for Training
get_data <- function(x) {
    raw_data <- readRDS(glue('{x$custname}.Rdata'))
    x$data <- map_dfc(x$features, function(feat) {
        map_dfr(raw_data, ~ feat$extract(feat, .)
    }
    x
}
# D -- storing NA-handing in Training
if (anyNA(new_cols)) {
    x$features[[pos]]$na_info <<- infer_missing(new_cols, feat$na)
    new_cols <- apply_missing(new_cols, x$features[[pos]]$na_info)
}
# E -- generating Scorable Entity objects, and applying NA rule, in Scoring
newdata <- imap_dfc(predictor_features, function(feat, pos) {
    new_cols <- feat$extract(feat, obj)
    if (anyNA(new_cols))
        new_cols <- apply_missing(new_cols, x$features[[pos]]$na_info)
    new_cols
})
```

Figure 3: DSL implementation code illustrating several key patterns. Extracted from code at `https://github.com/HarlanH/featgen-demo`

from the DomainEntity object. In this case, the logic is easy—just extract a Boolean from an object and create a single-column, single-row data frame. Other features might do much more complex processing (see the `sales_trend` feature for an example), or generate multiple columns.

The `promo` FT can now be used to extract features from a stream of data. Figure 3C shows a substantially simplified version of this process in the implementation of the `get_data` function. In the accompanying demo, the data is stored in an R archive file, but in practice, you might query a web service for a stream of training data. Lines 16–18 iterate over all combinations of features and training data points, generating a data frame that can be used directly by the actual machine learning algorithm.

Not shown is the implementation of `train`, which implements the `Predictor Generator` method, creating a set of scored training data, and a `Predictor` method than can be used for Scoring.

It's also worth noting the role of the `Finalizer`, a component of the *application* that translates the raw prediction to something that an end-user can use to make decisions. It's common, for instance, to translate a raw probability score to a recommended action, or a red/yellow/green risk category. These are User Interface and Design decisions, however, and the code that implements them should be part of the application. Data scientists can and should certainly provide advice on this mapping.

3.3 Missing Data

Figure 3, parts D and E, show aspects of the implementation of a critical component of this pattern — how missing data should be handled. In a SaaS context, the patterns of missing data per client may be quite different, and the best transformations may vary substantially. At a high level, a policy specified in the DSL (or in some cases by default rules set in a FTG) is applied to a new column with missing data by a standard `infer_missing` function. That function does not actually impute the missing data, but instead returns a value that is used for imputation later. In the case of mean imputation, this would be the value of the mean of the column. The identical `apply_missing` function, with identical parameters, is then applied in both Training (D) and Scoring (E), ensuring that both Training and Scoring data have the same distributions.

The demo implementation supports mean, median, min, max, and constant imputation of missing data. The same pattern is used for continuous transformations, such as Box-Cox, which requires a parameter to be fit, stored, and used in production. Other transformations such as collapsing rare categories, or Windsorizing data at percentiles, can be implemented the same way.

4. Discussion

Using the Repeated Domain-Specific Modeling pattern, an organization can create a workflow for creating and deploying many related models, maintaining flexibility while maximizing efficiency and reliability, and ensuring clean separation of concerns and the ability of data scientists to apply their domain knowledge throughout the process.

There are a number of improvements to this framework that could be made. The open source code provided is an example of how to start implementing the pattern, but it is not

a package; anybody wishing to use the code would have to discard the Acme Chain Model pieces, and rewrite those components with FTGs for their own domain. It might be possible to extract general pieces of the code into an actual package that might reduce time-to-value. Also, although writing new FTGs is not generally difficult, additional tooling for doing so and setting up test cases would make it even easier and faster for data scientists to extend the framework as they learn.

Acknowledgments

Thanks to my former colleagues at EAB for helping me think through and implement an earlier version of this pattern, and to Jon Morra and others for their thoughts on earlier versions of the theory components.

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. URL `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`.

Stefan Milton Bache and Hadley Wickham. *magrittr: A Forward-Pipe Operator for R*, 2014. URL `https://CRAN.R-project.org/package=magrittr`. R package version 1.5.

Bernd Bischl, Michel Lang, Lars Kotthoff, Julia Schiffner, Jakob Richter, Erich Studerus, Giuseppe Casalicchio, and Zachary M. Jones. mlr: Machine learning in R. *Journal of Machine Learning Research*, 17(170):1–5, 2016. URL `http://jmlr.org/papers/v17/15-066.html`.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

Drew Conway. The data science venn diagram, 2013. URL `http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram`.

Ryan M. Deak and Jonathan H. Morra. Aloha: A machine learning framework for engineers. In *Proceedings of the SysML Conference*, 2018.

Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

T. Erl. *SOA Principles of Service Design*. The Prentice Hall Service Technology Series from Thomas Erl. Pearson Education, 2007. ISBN 9780132715836. URL https://books.google.com/books?id=mkQJvjR2sX0C.

M. Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. ISBN 9780131392809. URL https://books.google.com/books?id=ri1muolw_YwC.

Harlan D. Harris. Workflow tools for helping with model-fitting, 2016. URL https://www.rstudio.com/resources/videos/workflow-tools-for-helping-with-model-fitting/.

Kaggle.com. Rossmann store sales, 2015. URL https://www.kaggle.com/c/rossmann-store-sales.

Max Kuhn. Building predictive models in r using the caret package. *Journal of Statistical Software, Articles*, 28(5):1–26, 2008. ISSN 1548-7660. doi: 10.18637/jss.v028.i05. URL https://www.jstatsoft.org/v028/i05.

Li Erran Li, Eric Chen, Jeremy Hermann, Pusheng Zhang, and Luming Wang. Scaling machine learning as a service. In Claire Hardgrove, Louis Dorard, Keiran Thompson, and Florian Douetteau, editors, *Proceedings of The 3rd International Conference on Predictive Applications and APIs*, volume 67 of *Proceedings of Machine Learning Research*, pages 14–29, 2017. URL http://proceedings.mlr.press/v67/li17a.html.

Jonathan Morra. Data science at eharmony: A generalized framework for personalization. Strata + Hadoop World, 2016. URL https://conferences.oreilly.com/strata/strata-ny-2016/public/schedule/detail/51731.

Hilary Parker. Opinionated analysis development, 2017. URL https://www.slideshare.net/hilaryparker/opinionated-analysis-development. rstudio::conf.

Ivens Portugal, Paulo S. C. Alencar, and Donald D. Cowan. A survey on domain-specific languages for machine learning in big data. *CoRR*, abs/1602.07637, 2016. URL http://arxiv.org/abs/1602.07637.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2017. URL https://www.R-project.org.

Wikipedia. Conway's law, 2017. URL https://en.wikipedia.org/w/index.php?title=Conway%27s_law&oldid=814575211.

David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

Journal of Machine Learning Research 82 (2017) 33-44, 4th International Conference on Predictive Applications and APIs

Marvin - Open source artificial intelligence platform

Lucas B. Miguel LUCAS.BONATTO@B2WDIGITAL.COM
Daniel Takabayashi DANIEL.TAKABAYASHI@B2WDIGITAL.COM
Jose R. Pizani JOSE.PIZANI@B2WDIGITAL.COM
Tiago Andrade TIAGO.ANDRADE@B2WDIGITAL.COM
Brody West BRODYW@MIT.EDU

Editor: Claire Hardgrove, Keiran Thompson and Louis Dorard

Abstract

Marvin is an open source project that focuses on empowering data science teams to deliver industrial-grade applications supported by a high-scale, low-latency, language agnostic and standardized architecture platform, while simplifying the process of exploration and modeling. Building model-dependent applications in a robust way is not trivial, one is required to have knowledge in advanced areas of sciences like computing, statistics and math. Marvin aims at abstracting the complexities in the creation process of scalable, highly available, interoperable and maintainable predictive software.

Keywords: Machine Learning, Platform, Predictive Analytics, Artifical Intelligence

1. Introduction

Being able to quickly identify hidden patterns in datasets, wisely choose the best model to train from historical data, and making predictions are the biggest advantages of data-driven organizations against their competitors. Knowing the customer demand for a product before buying it (Chen et al., 2000) or being able to detect a fraud before charging the customer's credit card (Chan and Stolfo, 1998) with a certain level of confidence are examples of how companies doing businesses on the internet are making better decisions and maximizing their earnings.

Capturing and storing large amounts of data is a commonplace for most companies these days. Having more data available is often a positive aspect in order to train models with a lower error rate. However, writing code to effectively process terabytes of data and provide near-real-time predictions supporting high throughput is not a trivial task. One is required to have knowledge in advanced areas of science, such as computing, statistics and math. High scale data processing frameworks (Zaharia et al., 2010) fulfill their role by abstracting some of the complexities related to distributed computing and process orchestration. Libraries like MLLib (Meng et al., 2015) and scikit-learn (Pedregosa et al., 2011) facilitate it by providing high level interfaces to common machine learning algorithms. Even so, building robust model-dependent applications is tricky and requires specialized knowledge. There is an open space for a platform that empowers the data scientist with the tools

and abstractions needed to create scalable, highly available, interoperable and maintainable predictive software.

In this paper we present Marvin (*https://github.com/marvin-ai*), an open source platform that aims to help data scientists with several tasks during the life cycle of an artificial intelligence project. In section 2 we describe the platform's main features, section 3 contains implementation details, section 4 has a sample application and section 5 shows or experiment results. Finally in section 6 we talk about future work and summarize our findings in section 7.

2. Marvin Overview

A model-dependent application is described by an application that processes historical data and train a mathematical model such that the output Y can be explained by the different values of the input X. Some categories of algorithm in the artificial intelligence area fits in this description, e.g. support vector machines, statistical AI, neural networks. Marvin provide tools that will help in different phases of a project with such characteristics. The tools can be executed both on-premise or in a cloud infrastructure, giving the flexibility that is needed for different scenarios. Marvin is different from ML PaaS (AzureMLTeam, 2016), since it can be executed on-premise and allow the use of any algorithm that can be implemented in general-purpose languages (Van Rossum and Drake, 2003) (Odersky et al., 2004). In fact, Marvin could be used as the back-end of such type of platforms.

Marvin was built to enforce the pillars of the basic phases of a model development project, see Figure 1.

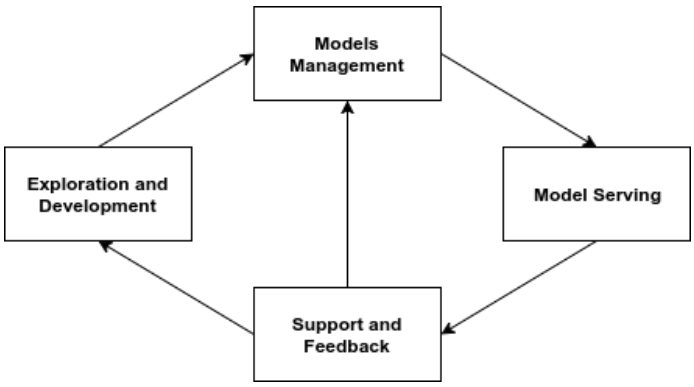


Figure 1: Model application development cycle.

To achieve that, Marvin provide several features, those include:

1. Toolbox - Provide a great set of common tools that are commonly used during the exploration phase of a data science projects and will eventually be carried up to production. The data scientist can take advantage of notebooks, plotting libraries, data frames and so on.

2. Experiment Versioning - During the exploration and exploitation of the problem it is common to test several hypothesis and eventually change approach. Keeping the history of experiments is useful and may serve to explain the final solution.

3. Data Sync CLI - During the model development data is intensively accessed, either to perform feature engineering or backtest the model. In complex environments it's often not a good approach to access production databases during development phase. To solve that Marvin provide a tool to sample data from the official dataset and work locally.

4. Unit Test Framework - In order to ensure a testable application, Marvin provides a built-in unit test framework, encouraging the data scientist to avoid bugs being introduced in the model application in the future.

5. Project Generator - Marvin introduces a design pattern, see Figure 5, to ensure that applications will be built in a decoupled manner. The project generator utility creates the base skeleton for applications, the data scientist is required to only populate the skeleton files with their logic.

6. Artifact Versioning - Marvin keeps track of artifacts generated during the training pipeline. When running the application in production Marvin allows the user to restore the system to a previous state, i.e. publishing a model trained last week because the current model is presenting greater error rate.

7. Large Dataset Processing - Integration with main frameworks for parallel and distributed computing to allow the effective handling of large datasets.

8. Training Pipeline Interface - The phases involved in the training pipeline (data acquisition, data preparation, training) can be started through the CLI or via REST HTTP calls. Allowing the application to be executed by external agents.

9. Feedback Server - In order to allow the model to receive external signals, Marvin provides a feedback server. User and applications can send feedback data to the model, which can be interpreted and perharps start a new training.

10. Predictor Server - When finishing the training pipeline Marvin persists the serialized model in a persistent memory storage. This model is loaded afterwards by the predictor server and is accessible via REST HTTP calls. Users and applications can then make predictions taking advantage of the model.

Marvin is composed by three main components, see Figure 2. The toolbox is both a command line interface and a library that contains a set of utilities to help data scientists during the phase of exploration and development. Using the toolbox the user will build his application, which in the context of Marvin is called an engine. The engine must be built following a design pattern proposed by Marvin, the toolbox will generate a scaffold with the classes corresponding to this pattern. Each phase in the pattern will produce an artifact, that can be persisted and reloaded. Lastly, the engine-executor is the component responsible of orchestrating the execution of engines and also by deploying servers to allow external interaction with it.

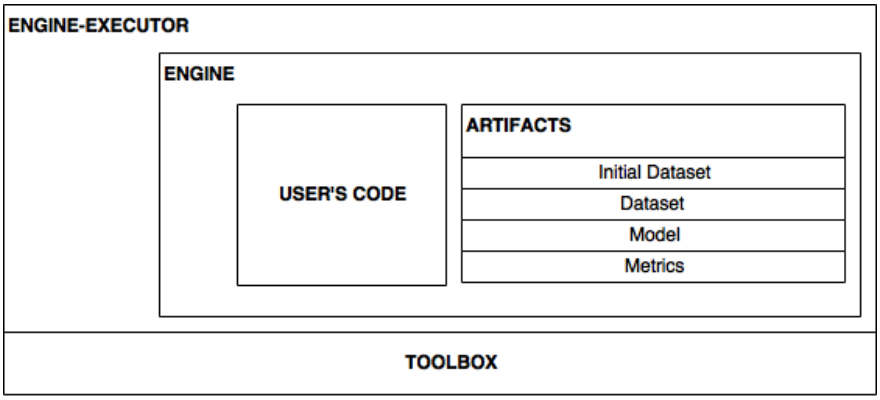


Figure 2: Marvin components.

Several previous work already help on the task of implementing artificial intelligence applications able to deal with large datasets and achieve good performance. SystemML (Ghoting et al., 2011) provide a high level declarative interface to implement machine learning applications that can process massive amounts of data. Pregel (Malewicz et al., 2010) introduces a computational model suitable for large-scale graph processing. OptiML (Sujeeth et al., 2011) is a domain-specific language (DSL) to achieve implicit parallelism on machine learning applications. MLI (Sparks et al., 2013) offers an API for distributed machine learning that helps turning prototypes into industry-grade ML software. Although these works do a great job abstracting complexity in the batch processing phase, they do not intend to offer tools to help during problem exploration and model serving. Table 1 shows the comparison of Marvin, SystemML and a market solution (Cloudera).

	Marvin	**SystemML**	**Cloudera DSW**
Multi-language support	x		x
HDFS support	x	x	x
Distributed CPU ML API	x	x	
Single node capability	x	x	
Toolbox (development environment)	x		x
Templates	x		x
Integrated notebooks	x		x
Models management	x		x
Data versioning control	x		
Pipeline scaffolding	x		
Unit test integration	x		
REST API	x		
Pre-process optimization		x	
Feedback server	x		

Table 1: Functionality comparison

3. Implementation Details

One of Marvin's main objective is to optimize execution of it's users' applications in order to process large datasets and allow a high number of concurrent model access without penalizing performance. To help users see the boundary of different executions flow within the application we propose the separation of actions in two categories:

- batch - e.g. train, evaluate, prepare
- online - e.g. predict, feedback

Batch actions are executed asynchronously and the result of it's execution will be an artifact, i.e. binary or plain text data. The generated artifacts can be persisted and re-used in the same application or with other applications instances. A practical use of this functionality can be to share the initial dataset artifact between different models, avoiding unnecessary duplicated computation. These characteristics help to perform effective long running and data-intensive jobs.

In order to achieve parallelism in different levels, Marvin applications run on top of common data processing frameworks, see Figure 3. The Marvin context is a component that frees the data scientist of setting up and optimizing the framework, it wraps some methods from the original framework library but also expose the main features of it.

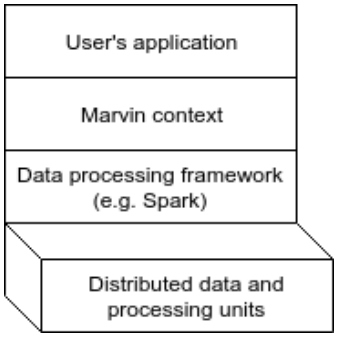


Figure 3: Multi-tier architecture for batch processing.

On the other hand, online actions are executed synchronously and they may generate a valid result. Their result will usually be interpreted by other application, therefore to ensure simple scalability, interoperability, availability and the needed consistency we adopted a microservice-based architecture (Brewer, 2000) (Fowler and Lewis, 2014).

All application's execution are orchestrated by the engine-executor component. This is a configurable component that can be deployed in different formats, depending on the environment complexity. One may want to deploy an engine-executor instance for each pipeline phase (data acquisition, data preparation, model training, etc.), it would avoid a single point of failure and allow independent scalability for each phase. However, projects on a smaller scale may prefer to deploy all the phases and the predictor server in the same instance. To achieve safe and effective concurrent execution, the underlying of engine-executor is implemented according to the Actor model (Hewitt et al., 1973).

As we understand that there is no mother programming language for artificial intelligence applications, we built Marvin under the assumption that it should be language agnostic. It means that users are free to implement their applications using their preferred language, provided that its support is implemented. The first Marvin's version supports Python language. Figure 4 shows how Marvin is using the RPC (Srinivasan, 1995) protocol to execute code written in different languages.

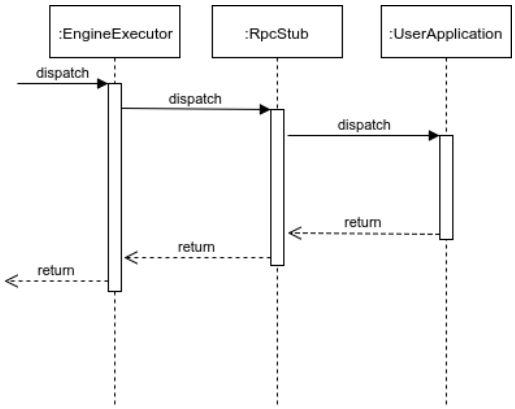


Figure 4: Simplified sequence of engine-executor executing online action on user's code.

4. Sample Engine

Marvin applications are also labeled as engines. An engine is composed by the application's source code, a file containing parameters for the application and the metadata file. We encourage users to implement decoupled engines that are easy to maintain and less bug prone. To induce that we propose the DASFE design pattern (Figure 5). This pattern is strongly based on the DASE pattern (Chan et al., 2013), however we added the feedback phase to it. This evolution intends to enable the engine to receive input from external applications or users, this kind of feature allows the model to be modified online or add hooks to start a new training pipeline.

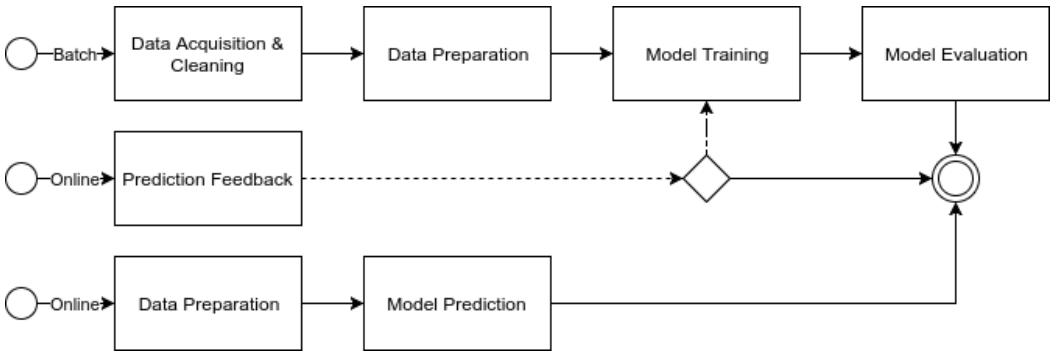


Figure 5: DASFE pattern pipeline.

We provide a project generator utility that generates the necessary base code for an engine, by doing that we reduced the complexity of the data scientist's job, requesting them to just populate these base files with the program logic. It is not necessary to care about passing the data to the next phase, or serializing the artifact in some persistence.

The next paragraphs will present a simplified sample engine as reference. The sample is a Python engine able to classify products using a linear classifier with stochastic gradient descent (SGD). The engine uses a mix of Spark framework data structures and Pandas dataframe.

The code to obtain data from data sources and remove unnecessary rows, i.e. data cleaning, should be placed in the execute method of AcquisitorAndCleaner class:

```
class AcquisitorAndCleaner(EngineBaseDataHandler):

    def execute(self, **kwargs):
        data = self.spark.sql("""select p.bscprd_desc as name,
        h.misphr_line as tag from core.mis_product_hierarchy as h,
        core.bsc_product as p
        where h.misphr_id_product = p.bscprd_id_product
        and h.misphr_line in ('SMARTPHONE', 'TABLETS')""")
        self.initial_dataset = data.toPandas()
```

Then it is necessary to prepare the acquired data before training. The execute method on TrainingPreparator should contain preparation logic, e.g. inputation of missing values and data type transformation:

```
class TrainingPreparator(EngineBaseDataHandler):

    def execute(self, **kwargs):
        data = self.initial_dataset
        vectorizer = TfidfVectorizer(encoding='utf-8')
        vectorizer.fit(data['name'])
        X_train = vectorizer.transform(data['name'][10:])
        y_train = data['tag'][10:]
        X_test = vectorizer.transform(data['name'][0:10])
        y_test = data['tag'][0:10]
        self.dataset = {
            "vectorizer": vectorizer,
            "X": (X_train, X_test),
            "y": (y_train, y_test)
        }
```

The model is finally trained at the Trainer class, when the execute method completes Marvin will serialize the model in the configured persistence. The data scientist do not need to implement the serialization logic, Marvin has serialization mechanisms implemented in all supported languages. If it is necessary to use custom serialization it can be achieved by extending Marvin application programming interface. The Trainer code will look like:

```
class Trainer(EngineBaseTraining):

    def execute(self, **kwargs):
        data = self.dataset
        clf = SGDClassifier(**self.params).fit(data['X'][0], data['y'][0])
        self.model = {"clf": clf, "vectorizer": self.dataset["vectorizer"]}
```

The MetricsEvaluator class is the appropriated class to place code related with model evaluation, in this example we're computing model error in a confusion matrix:

```
class MetricsEvaluator(EngineBaseTraining):

    def execute(self, **kwargs):
        pred = self.model['clf'].predict(self.dataset['X'][1])
        m1 = classification_report(self.dataset['y'][1], pred)
        m2 = confusion_matrix(self.dataset['y'][1], pred)
        self.metrics = [m1, m2]
```

At PredictionPreparator the sample engine is just performing data transformation:

```
class PredictionPreparator(EngineBasePrediction):

    def execute(self, input_message, **kwargs):
        return self.model['vectorizer'].transform([input_message['msg']])
```

Finally the Predictor class contains the logic that will be called for every valid HTTP request made to the Predictor server. Marvin's engine-executor will load the serialized object from the configured persistence and inject the trained model in the self.model variable. The code will be as follows:

```
class Predictor(EngineBasePrediction):

    def execute(self, input_message, **kwargs):
        return np.array_str(self.model["clf"].predict(input_message))
```

Deploying this code on Marvin's engine-executor will provide control over the training pipeline execution, dataframes and model serialization and versioning, metrics evaluation, model serving and feedback input interface.

5. Performance Assessment

Aiming to evaluate the parallelism ability of the predictor server, we conducted a set of experiments predicting classes of iris in a Support Vector Machine (SVM) (Hearst et al., 1998) model trained using the classical Fisher's iris dataset (Fisher, 1936) for classification. The overall objective of this experiment was to ensure that the predictor server is able to take advantage of multi-core architectures while not impacting significantly negative on the response time of predictions or the consistency of the results due to many queued threads or timeout exhaustion.

The test setup consisted of two dedicated machines, one simulating the users and other running Marvin's engine-executor with the SVM model that was previously trained. The client machine had 24 cores with Hyper-threading available and 48GB of RAM. The server machine also contained 24 cores with Hyper-threading and 64GB of RAM. Both machines were running Debian GNU/Linux.

The experiment strategy was to keep the amount of resources available and increase the throughput of concurrent requests being sent to the server until a significantly increase on the response time or failed requests was observed. The requests from client to server were made through the REST HTTP protocol and the machines were located in the same physical data center. As it is possible to see in Figures 6 and 7, we achieved 500 predictions

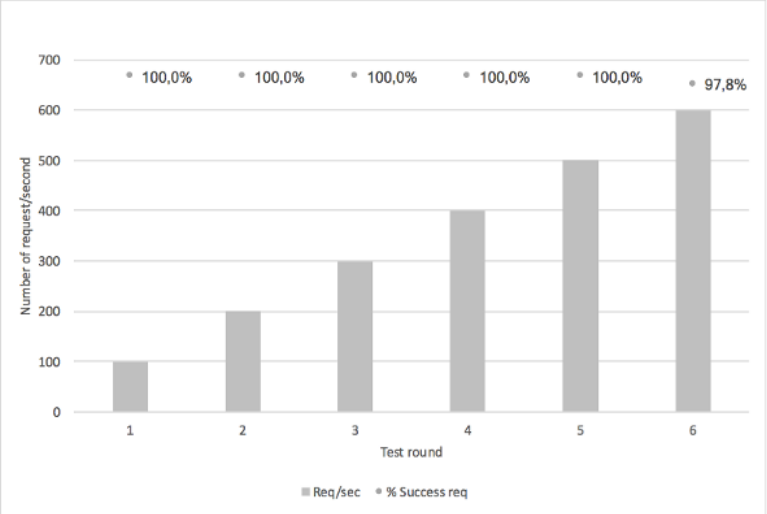


Figure 6: Predictor load test (reqs/second).

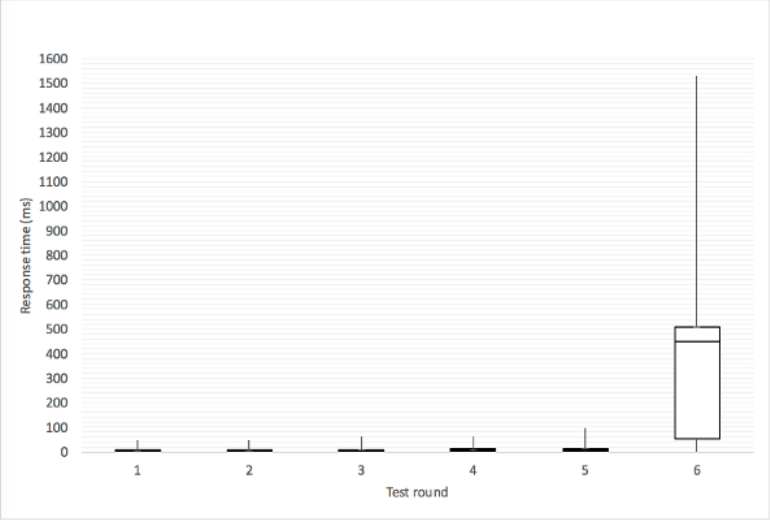


Figure 7: Predictor load test (Response time box plot).

per second maintaining a stable response time and none failed requests, at round 6 the mean response time increased significantly and the error rate has raised. The behavior of test round 6 indicates that we crossed the edge of efficient parallelism of Marvin's engine-executor predictions. Although we consider this a good number, we encourage administrators to deploy several instances of engine-executors behind a load-balancer when more than 500 predictions per second must be served. The current performance result is proved to be enough for large scale e-commerce platforms, like B2W Digital.

6. Future Work

Although Marvin is already being used in production setups, several improvements can be made to turn it into the de facto choose for data science teams which need to build production-facing models. The current version of Marvin has independent setups for each engine, it means that the user is responsible for having a layer on top of it if he desires to have a single management console of his engines. To build a cluster of engines the user needs to make specific configurations, like set the persistence folder for each engine under the same parent folder, and also maintain engine's parameters per instance. These could be challenging when it becomes to maintaining dozens of engines. Thus there is space to build a cluster admin on top of Marvin's engines.

Marvin platform was built on an architecture that allows the engine-executor to run engines implemented in different programming languages through the RPC protocol. As the date of this paper, there is a Python toolbox that facilitates the work of a user implementing engines in this language. In the near future we plan to focus our efforts on implementing toolboxes for different languages, e.g. R, Julia, Scala, Go and Java.

7. Conclusion

Implementing artificial intelligence applications with enterprise software characteristics is a hard task. Several contributions were made in libraries offering algorithms implementation and frameworks for distributed computing of data-intensive applications. Marvin adds tools and integrate with libraries and data frameworks to support the exploration and model development of such kind of applications, it introduces a framework that speeds up the task of turning model prototypes into industry-grade software. Lastly Marvin's engine-executor is a model server that takes care of pipeline execution, artifacts serialization and offers a standard interface to allow other applications to access the model, it takes into account non-functional requirements to allow safe concurrency and effective parallelism on shared and distributed memory. The experiments demonstrated that engine-executor is able to serve 500 predictions per second while maintaining stable response time and 0 failed requests.

Marvin engines are helping companies to be data-driven organizations, serving algorithms that can automate decisions such as optimizing its products prices to increase revenue, detecting fraud at the earliest stage and customizing sorting of many offers of the same product to customer clusters. The platform meshes the necessary components to empower data scientists pursuing to deliver production level applications that can support high throughput and process large datasets.

References

AzureMLTeam. Azureml: Anatomy of a machine learning service. In Louis Dorard, Mark D. Reid, and Francisco J. Martin, editors, *Proceedings of The 2nd International Conference on Predictive APIs and Apps*, volume 50 of *Proceedings of Machine Learning Research*, pages 1–13, Sydney, Australia, 06–07 Aug 2016. PMLR. URL `http://proceedings.mlr.press/v50/azureml15.html`.

Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

Philip K Chan and Salvatore J Stolfo. Toward scalable learning with non-uniform class and cost distributions: A case study in credit card fraud detection. In *KDD*, volume 1998, pages 164–168, 1998.

Simon Chan, Thomas Stone, Kit Pang Szeto, and Ka Hou Chan. Predictionio: a distributed machine learning server for practical software development. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 2493–2496. ACM, 2013.

Frank Chen, Zvi Drezner, Jennifer K Ryan, and David Simchi-Levi. Quantifying the bullwhip effect in a simple supply chain: The impact of forecasting, lead times, and information. *Management science*, 46(3):436–443, 2000.

Cloudera. Cloudera Data Science Workbench (DSW). `https://www.cloudera.com/products/data-science-and-engineering/data-science-workbench.html`. Accessed: 2017-10-10.

Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of human genetics*, 7(2):179–188, 1936.

Martin Fowler and James Lewis. Microservices. *ThoughtWorks. http://martinfowler.com/articles/microservices. html [last accessed on February 17, 2015]*, 2014.

Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 231–242. IEEE, 2011.

Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. Support vector machines. *IEEE Intelligent Systems and their applications*, 13(4):18–28, 1998.

Carl Hewitt, Peter Bishop, and Richard Steiger. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

Xiangrui Meng, Joseph Bradley, Evan Sparks, and Shivaram Venkataraman. Ml pipelines: a new high-level api for mllib. *Databricks blog, https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib. html*, 2015.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

Evan R Sparks, Ameet Talwalkar, Virginia Smith, Jey Kottalam, Xinghao Pan, Joseph Gonzalez, Michael J Franklin, Michael I Jordan, and Tim Kraska. Mli: An api for distributed machine learning. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 1187–1192. IEEE, 2013.

Raj Srinivasan. Rpc: Remote procedure call protocol specification version 2. 1995.

Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. Optiml: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 609–616, 2011.

Guido Van Rossum and Fred L Drake. *Python language reference manual*. Network Theory, 2003.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

Proceedings of Machine Learning Research 82 (2017) 45-51, 4th International Conference on Predictive Applications and APIs

Prediction and Uncertainty Quantification of Daily Airport Flight Delays

Thomas Vandal TJ.VANDAL@GETFREEBIRD.COM
Max Livingston MAX@GETFREEBIRD.COM
Camen Piho CAMEN@GETFREEBIRD.COM
Sam Zimmerman SAM@GETFREEBIRD.COM

Abstract

One in four commercial airline flights is delayed, inconveniencing travelers and causing large financial losses for carriers. The ability to accurately predict delays would make travelers' lives easier and save airlines money. In this work, we approach the problem of predicting flight delays using a Variational Long Short-Term Memory (LSTM) model. The model is trained to predict aggregate daily delays for U.S. airports using a combination of continuous and discrete variables, including weather, airport characteristics, and congestion. Monte Carlo Dropout, a Bayesian Deep Learning technique based on variational inference, is incorporated to provide planners with a well-calibrated prediction interval. We show that our Variational LSTM results in an average median absolute error of 5.8 minutes per day across 123 airports in the United States. Moreover, results show that predictive uncertainty is well explained through a calibration analysis.

1. Introduction

Commercial flight delays in the U.S. airspace are a pervasive and expensive problem. A large-scale study by the U.S. military found approximately 25% of flights were delayed in the late 2000s (Fleming, 2009). Recent analysis done by a large travel technology company estimates the economic impact of these disruptions to be $60B and growing (Gershkof, 2016), representing over 8% of global airline revenue. With ridership in the U.S. expected to double or triple in the next 15 years and few definitive plans for infrastructure improvements, this economic impact will likely increase.

In this paper, we focus on our work predicting aggregate per-airport daily delays. A forecast of the average delay intensity at an airport for a given day can help travelers, air traffic controllers, airport operations, and travel agents stay informed about airport operations and make proactive decisions (Demuth et al., 2011). This strategy is also consistent with the literature, which typically analyzes aggregate delays (Gopalakrishnan and Balakrishnan, 2017).

There are two pieces of existing research in this domain that are similar to ours. The first, (Gopalakrishnan and Balakrishnan, 2017), compares performance of two different neural network techniques (a Multi-Layer Perceptron (MLP) and a General Regression Neural Net (GRNN)), traditional machine learning techniques, and a domain-specific technique for the problem of predicting aggregate origin-destination level and origin level delays.

Using a dataset of the top 30 airports (the FAA Core 30 list), they train the model to predict average outbound delays at an airport two hours ahead, given outbound delays now, outbound delays in the previous two hours, and the time of the day. By training an MLP with those features, they achieve a median absolute error of around 7 minutes. The second, Kim et al. (2016), uses a LSTM-RNN on daily flight delays. They focus on classification, setting a threshold for the average daily delay for an airport and training the RNN to predict whether a day's average delay would be above that threshold. They achieve an average classification accuracy of between 80 to 90%.

Our primary advancement over previous work is the modeling technique used, a Variational LSTM. This is a key feature of our approach and is crucial for Freebird's applied business aims. Because Freebird is interested in using these models in a risk management setting, we require a model that provides robust uncertainty metrics, not just point estimates. This requirement has limited the application of deep learning to the problem thus far. We leverage recent advances from (Gal and Ghahramani, 2016; Gal, 2016; Gal et al., 2017), using Monte Carlo dropout to obtain parameter variance estimates. To our knowledge, we are the first to apply Bayesian deep learning to the problem of predicting and quantifying flight delays in the U.S. airspace and the first to apply a Variational LSTM to any industrial problem.

2. Data Description

Predicting average daily flight delays for a given airport is a challenging task with many factors in play. However, many of these factors, such as operational malfunctions, are unpredictable in advance, which limits our feature set to airport level characteristics and weather variables. The features we use are as follows:

Continuous Features: Max Wind Gust, Avg. Wind Speed, Avg. Heat Index, Total Precipitation, Avg. Pressure, Total Snow, Avg. Temperature, Avg Visibility, # of Arrival Flights, # of Departing Flights, Previous Day's Delay
Categorical Features: Month, Day of Week, and Airport

The weather features were extracted from historical data provided by The Weather Company. The data we were provided included hourly statistics such as temperature, wind, snow, pressure, and many others. Variables were then aggregated from hourly to daily by either averaging, computing the max, or taking the sum. These features are then normalized across all airports before training.

Daily average flight delays and number of departing and arrival flight were extracted from a large proprietary data feed containing flight statuses for essentially all flights in the U.S. The number of arriving and departing flights were normalized per airport in order to preserve well distributed features. We note that these features are available approximately 200 days in advance given flight schedules and weather forecasts. We use data from July 1 2012 to July 31 2016, with a train/test split of 80%/20% . The final dataset consists of 123 airports over 4 years of daily samples, totalling 168,387 samples for training.

3. Method

In this section we describe the Bayesian LSTM architecture we developed to predict daily average flight delays per airport using both continuous and categorical features. Including an airport indicator variable allows us to leverage similar delay effects between airports, similar to a multi-task model. The model consists of 3-Layer LSTM network where data from the day of departure and the four previous days is used to predict average delay for the day of departure. First, we embed the categorical variables from 7 days to 3 features, 12 months to 3 features, and 123 airports to 5 features. The weights for embedding are learned during training. Using a grid search with a range of dimensions, we found that these embedding dimensions provided a good trade-off between complexity and over-fitting. Categorical variables embedded to dimensions 3, 3, and 5 are concatenated with the 11 continuous features, including weather and airport congestion, resulting in 22 total features that are fed into the LSTM.

Following (Gal and Ghahramani, 2016; Gal, 2016), we assume $\mathbf{y} \sim \mathcal{N}\big(\mu(\mathbf{x}), \sigma^2(\mathbf{x})\big)$ such that $[\mu(\mathbf{x}), \sigma(\mathbf{x})] = f^{\boldsymbol{\omega}}(\mathbf{x})$ where f is an LSTM network with two hidden layers of 128 units each. Approximate variational inference is applied over all the weights, $\boldsymbol{\omega}$, using Monte Carlo Dropout. As is crucial in Variational LSTMs, the same dropout mask is used at each step and for all weights (rather than dropping different weights at each time step). The corresponding negative log-likelihood is then written as:

$$\begin{aligned}\mathcal{L}(\theta) &= -\sum_{i\in\mathcal{S}} \log p\big(\mathbf{y}_i | f^{\hat{\boldsymbol{\omega}}}(x)\big) + \mathrm{KL}(q(\boldsymbol{\omega})||p(\boldsymbol{\omega})) \\ &= \frac{1}{D}\sum_{i\in\mathcal{S}} \frac{1}{2}\exp(-s_i)^{-1}||\mathbf{y}_i - \mu(\mathbf{x}_i)|| + \frac{1}{2}s_i + \frac{1-p}{2N}||\boldsymbol{\omega}||_2\end{aligned} \tag{1}$$

with $s_i = \log(\sigma^2)$ to enforce a positive variance where $\mathcal{S}$ consists of a sample batch of size D, θ denoting all parameters, and the dropout probability p. Using this approach, the first two moments of the predictive distribution have the following unbiased estimates:

$$\begin{aligned}E[\mathbf{y}] &\approx \frac{1}{K}\sum_{k=1}^{K} \mu^{\hat{\boldsymbol{\omega}}_k}(\mathbf{x}) \\ Var[\mathbf{y}] &\approx \frac{1}{K}\sum_{k=1}^{K} \mu^{\hat{\boldsymbol{\omega}}_k}(\mathbf{x})^2 + \frac{1}{K}\sum_{k=1}^{K} \sigma^{\hat{\boldsymbol{\omega}}_k}(\mathbf{x})^2 - \Big(\frac{1}{K}\sum_{k=1}^{K} \mu^{\hat{\boldsymbol{\omega}}_k}(\mathbf{x})\Big)^2\end{aligned} \tag{2}$$

where $\hat{\boldsymbol{\omega}}_k$ refers to weight realizations of $\boldsymbol{\omega}$ after applying dropout while K denotes the number of Monte Carlo samples (we set $K = 30$ in our experiments). A constant dropout rate of 0.25 is used in all layers. We refer readers to section 4 in Gal and Ghahramani (2016) for a more detailed analysis of dropout and uncertainty quantification in recurrent neural networks.

4. Results

To test our Variational LSTM, as described above, we study its ability to produce accurate daily average delay predictions per airport as well as corresponding predictive uncertainty.

	MAE				RMSE			
	1	3	5	7	1	3	5	7
Airport								
ATL	3.60	4.07	4.54	4.67	7.90	8.11	8.32	8.37
DFW	3.88	4.66	5.16	5.82	11.13	11.44	11.61	11.82
JFK	4.36	5.31	5.27	5.40	13.85	13.85	13.82	13.89
LAX	2.95	3.44	3.58	3.77	4.59	4.92	5.07	5.25
ORD	6.20	7.63	8.35	9.14	13.03	13.93	14.47	15.01
All	5.84	6.60	7.22	7.71	10.64	11.41	12.03	12.56

Table 1: Predictive ability at major airports for 1, 3, 5, and 7-days ahead measured in minutes per day by Median Absolute Error (MAE) and Root Mean Square Error (RMSE).

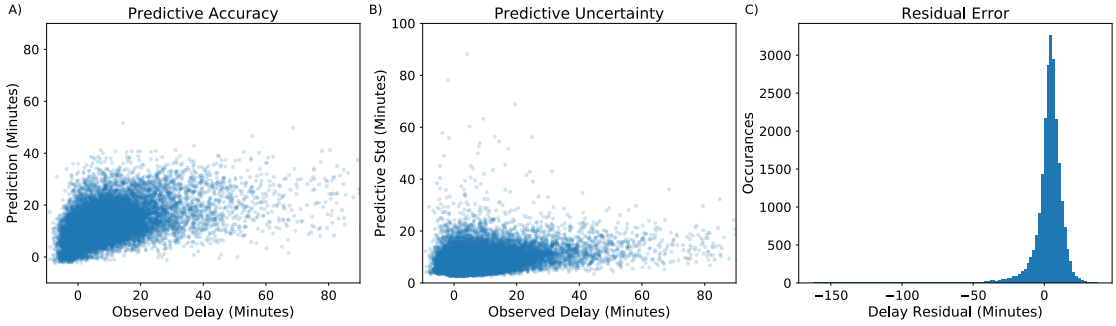


Figure 1: These results include test data only. (a) Observed average daily delay on the x-axis, which is truncated to 90 minutes) and predicted on the y-axis for 1 step ahead. (b) Standard deviation of the prediction by the observed delay (also truncated to 90 minutes). (c) Histogram of model residuals (Predicted - Observed)

All analyses are performed on a held-out test set (the last 20% of observations). To understand overall predictability, we visually compare the predictions and observations for each airport-date in Figure 1(a) and more quantitatively in Table 1. The long-tail distribution of delays results in a skewed error distribution, shown in Figure 1(a). In general, the model is not able to predict rare extreme delay events. For instance, the third largest average daily delay in our test set, 136 minutes, was on July 4th 2016 at John F. Kennedy (JFK) Airport in New York. These delays were caused by ISIS threats to three major international airports, including JFK. While this is just a single example, similar events happen, however infrequently, which are unreasonable to capture in the predictive means (as we show later on, we do reasonably well at accounting for the uncertainty via the predicted variance). If we remove observations with daily delay averages of over an hour, our RMSE for the test set drops from 10.64 to 9.21. To help understand the sensitivity to outliers, we report the Median Absolute Error (MAE) alongside the root mean squared error (RMSE) in Table 1 for predictions 1, 3, 5, and 7 days ahead for the top five airports by passenger count, as well as the validation dataset as a whole.

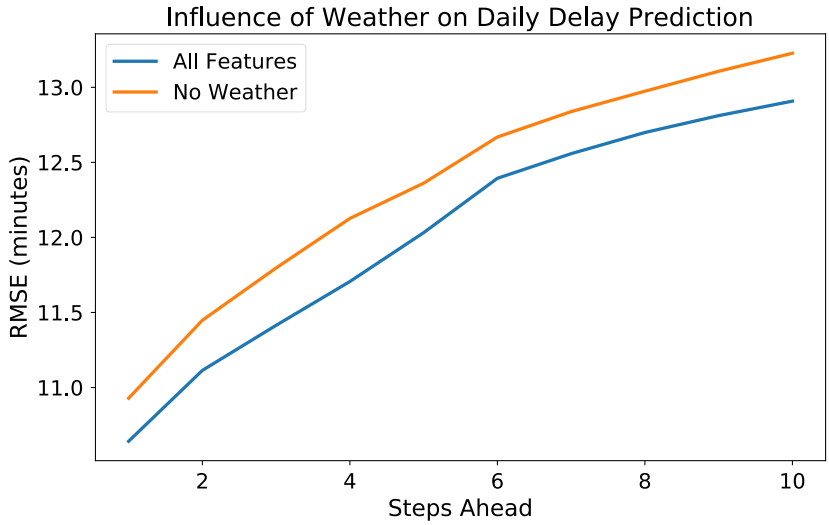


Figure 2: Comparison between model with (blue) and without weather features (orange) for 1 to 10 days ahead.

To gain insights into the importance of weather on delay prediction, we experiment with and without weather features. The model with weather outperforms the model without, with a decrease in RMSE of about 1 minute, shown in Figure 2, with roughly consistent improvement across prediction windows.

Lastly, we study the width and quality of predictive uncertainty. Figure 3 presents the calibration quality by measuring the frequency of observations within a given probability interval. For example, we expect 20% of the observations to fall between the estimated 40^{th} and 60^{th} percentiles. Too many observations outside of that percentile range would result in the curve being above the 45-degree line at 0.2. We clearly see that the uncertainty calibration is good at nearly all the airports. In contrast to the predictive mean, Figure 1 shows that predictive uncertainty measured by standard deviation does not increase dramatically with the observed delay. This all results in well-calibrated uncertainty with reasonable predictability.

5. Conclusion

This paper presented a novel Bayesian RNN-LSTM to predict aggregate flight delays in the US by airport. In addition, we also extended these deep learning models to extract stable uncertainty quantification for each prediction using Monte-Carlo Dropout techniques. Our analysis of the model showed that even at the daily aggregate level, flight delays are quite challenging to predict, with an average RMSE above 10 minutes. This furthermore motivates the need for uncertainty quantification. The approximate predictive posterior of our model was shown to be strong with little error in calibration.

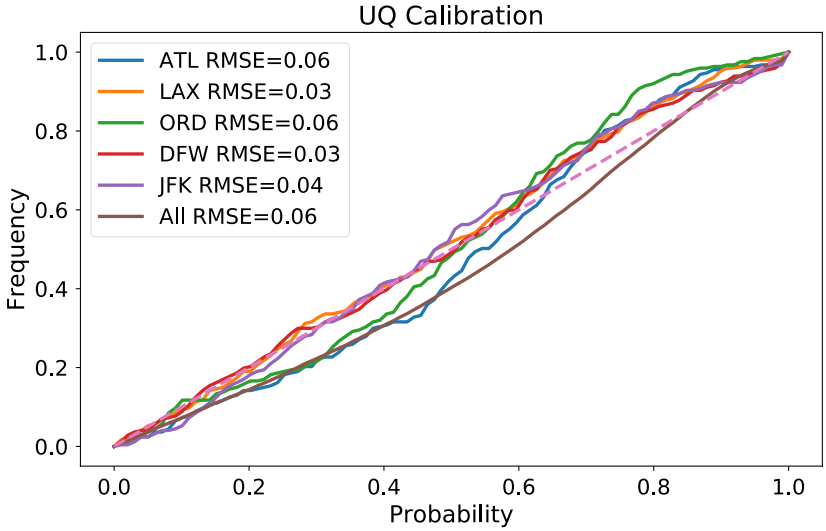


Figure 3: Uncertainty calibration measured by the frequency (y-axis) of observations within a given predictive interval (x-axis). The ideal calibration is when $y = x$, represented by the dashed line. RMSE is the error between calibration and ideal.

Future work could improve the training process to allow for more granular predictions, such as hourly forecasts or even flight-by-flight forecasts informed by the daily predictive distribution. Augmenting the RNN with additional features to capture network structure would allow us to model phenomena such as cascading delays across connected airports, represented using properties of the network. Furthermore, incorporating more domain informative features such as the Homeland Security Advisory System threat level and natural language processing feeds of relative news and social media platforms. An additional avenue for future work is to extend this model to better characterize different types of flight delays using mixed-density models, and to meaningfully distinguish between different types of uncertainty (Kendall and Gal, 2017).

References

Julie L. Demuth, Jeffrey K. Lazo, and Rebecca E. Morss. Exploring variations in people's sources, uses, and perceptions of weather forecasts. *Weather, Climate, and Society*, 3(3): 177–192, 2011. doi: 10.1175/2011WCAS1061.1.

S. Fleming. *National Airspace System: DoT and FAA Actions Will Likely Have a Limited Effect on Reducing Delays During Summer 2008 Travel Season: Congressional Testimony.* DIANE Publishing Company, 2009. ISBN 9781437908244.

Yarin Gal. *Uncertainty in deep learning.* PhD thesis, PhD thesis, University of Cambridge, 2016.

Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems*, pages 1019–1027, 2016.

Yarin Gal, Jiri Hron, and Alex Kendall. Concrete dropout. *Advances in Neural Information Processing Systems*, 2017.

Ira Gershkof. Shaping the future of airline disruption management (IROPS). Commissioned by Amadeus, 2016.

Karthik Gopalakrishnan and Hamsa Balakrishnan. A comparative analysis of models for predicting delays in air traffic networks. In *Twelfth USA/Europe Air Traffic Management Research and Development Seminar (ATM2017)*, 2017.

Alex Kendall and Yarin Gal. What uncertainties do we need in bayesian deep learning for computer vision? *Advances in Neural Information Processing Systems*, 2017.

Y. J. Kim, S. Choi, S. Briceno, and D. Mavris. A deep learning approach to flight delay prediction. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–6, Sept 2016. doi: 10.1109/DASC.2016.7778092.