Opportunities for non-data scientists to apply machine learning technology

Peter Ciuffetti[*]
Director of Product Development, ProQuest, USA

Abstract. This paper is based upon a presentation given at the NFAIS conference on Artificial Intelligence that was held May 15–16, 2019 in Alexandria, VA, USA. In it, the author describes how ProQuest used AWS SageMaker to build a content recommendation system. SageMaker is a product from Amazon Web Services (AWS) that brings machine learning capabilities to a broad audience. Out-of-the-box algorithms can be deployed by users who are not career data scientists.

Keywords: Machine learning, artificial intelligence, SageMaker, recommendation engine, factorization machines, Netflix Prize, Amazon Web Services, AWS, hyperparameter optimization

1. Introduction

SageMaker, launched in Nov, 2017, is a relatively-new offering from Amazon that is attempting to bring machine learning capabilities to a broader audience of technologists. Out-of-the-box machine learning algorithms can be deployed by users who are not career data scientists. And while some technical proficiency is required, it's not as daunting as you might think to build and deploy a solution to a problem using machine learning technology.

While recommendation systems using machine learning are established concepts with advanced applications already familiar to many users, it's a relatively new concept that someone who is not a career data scientist could actually construct a working system from off-the-shelf technologies. I describe this emerging opportunity, which is of interest to many non-data scientists, and show what is possible to achieve with 'traditional' application development skills.

2. My background

I received my computer degree a long time ago, when Artificial Intelligence (AI) was an active field of academic study, but in an era when intelligent computing was mostly movie fiction. I started working in the information industry in the early 1980's as an application developer. My academic courses included the requisite math that computer science majors took at that time, but I found very little of it was knowledge required to solve the computing problems I was assigned in my early career. I soon forgot all of the higher math I learned in my 20's. I do not recall anyone at the time considering themselves to be a data scientist. In the information industry, application developers need skills in what we would call information architecture, but this is a separate skill set. I did not gain any real data science skills as the field emerged later in my career.

---

[*]E-mail: pciuffetti@alexanderstreet.com.

I guess in retrospect that I'm probably the target *persona* for SageMaker. There are a lot of programmers in my category.

3. Our use case - A recommendation engine

At Alexander Street, where I worked, we had built a 'Related Items' feature for any content viewed by the users on our platform. I had built something similar at two prior companies. The related items were computed using a "more like this" search engine algorithm which compares common index terms and other attributes of the content with all the other content in the search engine, and is able to quickly produce a ranked list of the items most like the content you are viewing. Links to the top related items were displayed below the viewed item. In a database with good indexing a more-like-this approach can work very well; the feature gets a lot of user engagement.

The main problem with a more-like-this algorithm is that its result can be rather static for a given item of content. It does not take into account context or user behavior in determining the items in which you yourself might be interested at that moment in time; it is not in any way a 'learned' response. User expectations are of a more nuanced set of suggestions.

4. The rise of recommendation…

Many of us will have used Netflix or YouTube and in these systems the user often has to do very little work to find titles of interest; usually just a bit of scrolling. In one blog post we learned that 75% of Netflix views are from recommendations. If the user has to do a search to find a title of interest, it's considered by Netflix engineers to be somewhat of a failed experience. This is partially because doing a search on your TV with your remote control is not easy, and also because Netflix has failed to predict what the user might want now and place it among the items visible through scrolling.

The big players are setting the user expectations for how a repository of content should work. Netflix and YouTube in terms of recommending videos, Google in terms of understanding your search, iTunes and Spotify in what to listen to.

Vendors like us have all these challenges within our own small systems and a fraction of the data that these big companies are collecting about user behavior and content correlations. As user expectations in a video collection become more browse-oriented and less search-oriented, we felt if we didn't introduce quality recommendations into their immediate experience it was possible that our search page might eventually become a 'bounce' page.

In another product announcement, JW Player (a popular video platform) claimed that their customers saw a 35% average increase in number of video views when they activated their recommendation service, and a 25% increase in the time spent watching.

Content providers in our industry will know that the usage trends are a major driver in what content is collected by academic institutions. Institutions have to spend their limited collection development resources wisely and declining usage can get a subscription to your product cancelled. My whole career in product development in the information industry could be categorized as efforts to help users discover the right content and use it. Increases like these in user engagement are not so easy to achieve. While our intuition told us that a personalized recommendation system would be worth building and would help with engagement, these reported benefits by others helped make the decision to proceed.

We launched a small project to make related items more personal and to present these recommendations in other key moments in the user's experience. Perhaps not surprisingly, our initial thinking on how to achieve this centered around tweaks to the more-like-this algorithm. We had a bit of a blind spot. As we had no data scientists on the project it did not initially occur to us that we could ditch the more-like-this algorithm and replace it with a machine learning algorithm, even though we knew this is how the major e-commerce sites were providing similar features.

One of our colleagues attended the annual AWS re:Invent conference in November 2017 and when he came back he recommended to our team that we look into SageMaker which was announced at that conference. I decided to do a prototype with it. I used the week between Christmas and New Year's which for me has been great time to try something new. That week provides a combination of no meeting distractions and the annual self-improvement ritual that a new year inspires. In the next section I'll describe the basic features of SageMaker and then outline our approach, the challenges we faced, and some of what we learned along the way.

5. About SageMaker

Amazon has been using machine learning algorithms behind the scenes of their web applications for decades. And more recently they started providing building block APIs powered by machine learning, for example in Amazon Lex (a tool for building conversation bots: see: https://aws.amazon.com/lex/) and Amazon Rekognition (an image analysis tool: see: https://aws.amazon.com/rekognition/). With SageMaker AWS have packaged several machine learning (ML) algorithms into an architecture and Software Development Kit (SDK) that allows developers to build ML solutions using their own data and code.

SageMaker comes with four main elements:

(1) SageMaker Notebooks provide a browser-based development environment using Jupyter
(2) The Training API submits your code and data to machine learning instances that build and test your model.
(3) The Inference API deploys your model and provides a service endpoint for performing queries against your model.
(4) GroundTruth [1] provides tools for improving the quality of your model training data.

There are three ways to use SageMaker:

(1) Develop an ML model using a provided algorithm
(2) Develop an ML model using your own algorithm
(3) Develop a model elsewhere and use SageMaker to run it in production

SageMaker achieves this flexibility with a simple architecture that packages your code into Docker containers with predefined command-line scripts for doing training and inference. The architecture permits arbitrary algorithms to be trained and deployed on allocated instances since any Docker container built and deployed by the SageMaker SDK will respond to the same two basic commands: train and inference.

SageMaker launched with ten supported algorithms and Amazon has since added several more. Most ML programming on SageMaker is done in Python (see: https://www.python.org/), though other languages like R (see: https://www.r-project.org/) are possible. Most of the programming examples are in

Python. The notebooks come with a directory of examples covering each supported algorithm and other demonstrations of the SageMaker SDK.

6. The prototype week

I had two initial challenges getting started. One was that I had not done any Python programming, but the examples helped provide a bit of a crash course. The second was that I was unsure as to how to select an algorithm from among the supported ten. After a quick email to AWS support, I received back a recommendation to try Factorization Machines. I explored this on the web and indeed the Steffen Rendle paper that introduced this algorithm [2] talked about its use in a recommendation system.

More exploration into recommendation engines revealed stories about the Netflix Prize [3], a multi-year effort with a $1M prize attached to it to improve the performance of CineMatch, Netflix's recommendation algorithm. Another helpful presentation was by Conor Duke [4], a Data Scientist at Boxever (see: https://www.boxever.com/), who presented his project using Factorization Machines at a Python conference in Ireland. This quick research gave me a sense of how data scientists approached the problem.

To get started I needed some data. Like most vendors in our space we collect playback statistics for all videos for use in COUNTER media reports. So I gathered all the playbacks for 2017 across the entire user base of our video collections. This listed which titles were played, by institution, for the whole year. I collapsed it into a single count by title by institution by date.

We had some unique differences from the other examples I researched. Our video catalog is apparently quite a bit larger than Netflix's was at the time of the Netflix Prize, about three times as large. Also, Netflix had values for user ratings of videos to include in its model training. We had only the data on which videos were watched. Our user activity is spread out among more distinct titles.

Next, I had to choose the question which I wanted my ML algorithm to predict an answer. In retrospect, this turns out to be one of the more essential steps in the process. You need to choose a question for which your data will support making a prediction. In the Netflix Prize, the solution had to predict the user's rating of a video. And one hundred thousand user ratings over seventeen thousand videos were provided. As a surrogate for rating, I had the number of times a given title was watched at each institution. A popular title might get watched one hundred or more times at a subscribing institution (most likely by different users). My model's prediction had to match that data. So my model was going to (at most) be able to predict how many times an institution would want to watch a video. Like many patterns of use, there are a small number of popular titles that get watched a lot. A wide middle of titles that get watched a few times. And a long tail of videos that don't get watched at all. The average number of times a video was watched was about 2.1.

One of the things that didn't quite sink in until later, was the importance of 'negative' data in your training set. In a five-star rating system, you might watch a film and give it a single star. That probably means you *didn't* like it as opposed to a lukewarm indication of enjoyment. The Netflix Prize considered low star ratings as a sign of dislike. And so it was important not to recommend videos in the same category as the ones the users rated with one or two stars. To achieve this, a data scientist might potentially normalize the rating into a binary recommend/don't recommend value and to group the videos into these two categories based on the ratings. Then the use of a binary classifier would discriminate which of the videos to recommend. We had no source of negative data. We had to assume that if an institution watched a video, they might want to watch others like it.

Another choice you get to make is whether to run your model as a 'binary classifier' or as a 'regressor'. A binary classifier gives a yes/no answer to an inference query. A regressor will give you a range of floating-point values between weak and strong. My assumption was that in order to predict the number of times watched that I needed to use the regressor mode. My expectation was that whichever videos came out with the strongest predicted watch counts, the top 'n' of these would be what the system would choose as a recommendation.

For all the algorithms, you have to prepare your data in the format that the algorithm wants. The example code showed that Factorization Machines wants what is called a 'sparse matrix', one that is 'one-hot encoded'. Most of the first two days on the prototype was spent getting my playback data into this format using my non-existent python skills. In essence this matrix has a row for each interaction between a user and a video. The row is a 'feature vector'. There is a column for every 'feature' with a '1' in that cell if the interaction 'has' that feature, and a '0' if the interaction does not have that feature. A feature might be something like all the directors of all the films in the data set. For example, there would be a column indicating if the film was directed by Ken Burns. All the films he directed will have a '1' in that column. All the other films he did not direct will have a 0 in that column. You make a column for each director and put the 1's in the appropriate cell for each film they've directed. You repeat this layout for every other piece of metadata you expose to the model training (I included subject, release date, publisher, and content type). Some features are derived from the metadata about the film that was watched. Other features are about the institution that watched it. After encoding your data this way, you end up with a *very* wide matrix (my data had nearly one hundred thousand columns) and a *very* tall matrix (my training data had three hundred and fifty thousand rows). Most of these cells are a zero (which is why it is called sparse).

At the end of each row (feature vector) you provide the model with a 'label'. This is the value for that interaction. In the Netflix Prize test data, this would contain the user's rating for that video. In my data set, it contained the number of times watched.

The next step is to divide your data into three distinct sets: training, test, and validation.

- The *training set* is what teaches your model about the patterns of usage among the different interactions.
- The *test set* is what you use to evaluate the model. In this set of data, you know the value that occurred (i.e. this institution watched this video five times) and you ask the model to make a prediction for this feature vector. And you measure how close the model gets to making the correct predictions for each row in the test set.
- The *validation set* is used at the end to help you understand how the model will behave in the real world and to make sure it's not 'overfitted' to make the predictions represented by the test set only.

By day three I was successfully building models and deploying them and getting the deployed model to produce a prediction. The predictions though were nowhere near the values contained in the test set. A common computation to evaluate a model's efficiency is Root Mean Square Error (RMS - see: https://www.theanalysisfactor.com/assessing-the-fit-of-regression-models/). It is a measure of the closeness of predicted-to-observed across all the tests done and its value is in the same units as the predictions. A smaller RMSE score is better. My model's RMSE in its initial test was a 4.0. Which means (as I understand it) that the average prediction was generally four greater or lesser than the observed value.

I was not deterred. I used the fact that I had a couple pages of working python code to lift my spirits. I realized that I needed to think more about my training data and understand perhaps a little more about what Factorization Machines was doing with it. And there were (at the time) some undocumented parameters on the algorithm (called 'hyperparameters') that I had no idea what to use as settings.

One of the factors of my data was a very wide distribution of the values for 'times watched'. Some films were watched one thousand times or more whereas the Netflix Prize had a much smaller ratings range of one to five. Looking at the predictions made, the model was generally unwilling to predict a 'times watched' that was much greater than the average (which was about 2.1). So I convinced myself that if an institution watched it ten times or more, that was going to be my upper limit of popularity. Anything greater than that, I made it a ten. I also realized I had to be more careful about including data from each institution in both the training and test sets. My initial random distribution meant that the test set might not have any tests for some of the institutions. So, I randomly selected some test data for each institution. Finally, I had initially included the 'month watched' as one of the features in my vector. This was an important feature of Netflix Prize results, but I convinced myself that Alexander Street video was less sensitive to the day you were watching it and I removed this factor from the data set. This collapsed the values to 'by title, by institution' and meant each 'interaction' was more focused on the institution and the video. With these and some tests where I varied the hyper parameters, I was able to get the RMSE down to 2.5.

I spent day five writing python code to perform inference queries on the validation set and then output the results into a table with video titles that I could review visually. I wanted to see how I would react visually to the titles that got the highest predictions next to the data that was actually watched.

7. Observations and conclusions of the prototype

The model tended to put the predictions into buckets. Even though I was running it in 'regressor' mode, the model assigned thousands of videos the same prediction score. By increasing numfactors (one of the hyperparameters in the algorithm) I was able to increase the number of buckets. The documentation advised against making it 'too high' because then my model would be 'overfitted' (meaning it would describe the training data more precisely, but not necessarily in a way that described real world phenomena more accurately, those that contributed to actual video watching preferences). High settings for numfactors still left me with hundreds if not thousands of videos getting the same high score for a given set of recommendations for an institution. How was I going to select which of these to actually show to the user? Factorization Machines, like most ML algorithms I guess, is trying to detect patterns in the data. If your data is a pattern-less snowstorm of white noise the algorithm won't be able to derive an ability to make predictions about where a new piece of data fits into the snow. If there are pockets of dense activity spread in various places, then its better able to detect the phenomena that cause some videos to be more popular to some users. I needed to make sure my training data had more 'relevant' features.

To help solve the bucketing problem, I decided that I would only make inference queries for unwatched videos that were in the same subjects of the videos you already watched. This greatly reduced the number of inference queries that I had to perform. But is also introduces a certain bias. This system would therefore be incapable of recommending something from a related field of study that you hadn't previously explored. It places an undue emphasis on subject tagging; many videos are justifiably about subjects with which they are not tagged.

When I evaluated the recommendations visually, I generally accessed that the recommendations were of high-quality titles, with generally good usage, that were very related to the videos you had watched. That is, it was uncovering front list material that your institution had not watched. While I was concerned that it might not dig deep enough into the 'long tail' to pull out hidden gems, there was no harm in it recommending the excellent popular films.

When I shared these conclusions with the team, we agreed that this was a successful prototype and steered our implementation plans in the direction of using SageMaker.

8. Some learning along the way

Fast forward to the present. The recommender is now implemented in our platform and is placing the recommendations into the user experience where formerly there were hand-selected 'featured titles' that tended to remain static.

To get from the prototype to a production quality deploy, our small team did get the guidance and help of data scientists at ProQuest and Amazon. This did improve the quality of the solution quite a bit. And while I stand by my initial assertion that a non-data scientist could have achieved acceptable results, it certainly helped to have access to their expertise and made the development go faster.

We learned about Hyperparameter Optimization (HPO) [5] which takes away much of the guesswork about setting the parameters to your algorithms under test. HPO builds a series of models with different parameter values, tests each one, and then adjusts the parameters up or down while it detects a convergence on the best predictions coming out of the model.

We also learned how to try different algorithms on the same training data and to choose an algorithm based on its performance. This included bringing in new algorithms that were not part of the pre-installed algorithms.

We also operationalized the process of bringing in new usage data and content metadata, updating the models and the recommendations.

There is increasing competition in this space, with Google, Microsoft, IBM and others offering solutions that might be a better fit for you. It's a fast-moving field with more models and algorithms being packaged behind APIs that don't require you to bring your own data. It's becoming increasingly likely that you could exploit machine learning solutions in an application without having to touch model training and evaluation. It's a great time to get ML solutions into your apps and build a core of expertise around it.

About the Author

Pete Ciuffetti is currently Director of Product Development at ProQuest. He has also held technical positions at information industry firms since 1984 including SilverPlatter, HW Wilson, Credo Reference and Alexander Street. Email: pciuffetti@alexanderstreet.com.

References

[1] Amazon SageMaker GroundTruth, see: https://aws.amazon.com/sagemaker/groundtruth/, accessed September 28, 2019.
[2] S. Rendle, Factorization machines, in: *Proceedings of the 2010 IEEE International Conference on Data Mining* :pp. 995–1000. Available at: https://www.csie.ntu.edu.tw/~b97053/paper/Rendle2010FM.pdf, accessed September 28, 2019.
[3] Netflix Prize, *Wikipedia*, see: https://en.wikipedia.org/wiki/Netflix_Prize#targetText=On%20September%2021%2C%202009%2C%20the,for%20predicting%20ratings%20by%2010.06%25, accessed September 28, 2019.
[4] C. Duke, Building a Social Network Content Recommendation Service Using Factorisation Machines, YouTube, available at: https://www.youtube.com/watch?v=9lifSPf1Y5o, accessed September 28, 2019.
[5] Hyperparameter Optimization, *Wikipedia,* https://en.wikipedia.org/wiki/Hyperparameter_optimization, accessed September 28, 2019.