CLASSIEfier:

From scikit-learn to SageMaker in multilabel text classification

InnovationLab

An enterprise of:
ourcommunity.com.au
Where not-for-profits go for help

Certified B Corporation

CLASSIEfier: From scikit-learn to SageMaker in multilabel text classification

An Our Community Innovation Lab paper,
January 2020

Author: Paola Oliva-Altamirano, Data Scientist

This whitepaper is produced by the Our Community Innovation Lab – the engine room for mobilising data to drive social change.

**Our team of data scientists, IT engineers and domain knowledge experts is working to bring
to life ideas to do old things better and new things first.**

www.ourcommunity.com.au/innovationlab

Summary

Tracking the flow of funding to and within the Australian social sector has historically been difficult because of inconsistencies in categorisation (or the absence of categorisation entirely!).
Our Community, a highly ranked accredited B Corporation founded in Melbourne, Australia, developed CLASSIE to address this problem. Having developed CLASSIE as a universal classification system for Australian social sector initiatives, Our Community is now developing a machine learning algorithm called CLASSIEfier to reduce the need for manual classification. In the long-term, CLASSIEfier will help answer fundamental questions: Where is the money going? What impact is it having? And is the money going to those most in need?
Here we present the results of our experiments with different model training and deployment options; a) Using classical machine learning packages such as scikit-learning (python) on a local computer and b) Using pre-built models in AWS SageMaker.

SageMaker has considerable advantages over classical training and deployment, but raises concerns over bias control and model debugging. Can the black box become even darker?

In this article we also discuss the importance of defining the right scoring technique when designing fair algorithms, and demonstrate how accuracy does not always give you the right answer.

Note: This is the second article in a series about CLASSIEfier. For information about algorithm design and data exploration please go to CLASSIEfier: Using machine learning to paint a picture of social sector trends.

CLASSIE and CLASSIEfier

CLASSIEfier is a supervised machine learning algorithm designed to classify social sector text. It was initially intended to classify grant applications but is now functional for any text, including project summaries, fundraising campaign descriptions, organisation mission statements, and others (see a whole description of the algorithm here).

The algorithm categorises text by subjects and population (the latter category is known as 'beneficiaries' in the case of grant applications) according to the Classification system for Social Sector Initiatives and Entities (CLASSIE).

CLASSIE is a hierarchical taxonomy, comprising three levels for populations and four levels for subjects. Each level provides additional detail. For example, a project designed to provide services for people with autism spectrum disorder might be classified as follows: Health (level one) > Diseases and conditions (level two) > Brain and nervous system disorders (level three) > Autism (level four). Each classification is correct, but the higher the level, the more detailed is the classification.

Often text will generate more than one accurate subject label and more than one beneficiary label; in machine learning this is known as multilabel.

These two characteristics - hierarchical and multilabel classification - create huge challenges for CLASSIEfier to overcome (see this video for more detail).

CLASSIEfier early stages – classical machine learning

The first version of CLASSIEfier was trained on my local computer using Python's most common machine learning library, scikit-learn. This library holds a variety of prebuilt supervised and unsupervised models that can be used by anyone free of charge. I found that the most suitable models for CLASSIEfier were Nearest Neighbor (known as KNN) and Support Vector Machine (known as SVC), mostly because these algorithms have in-built ways to deal with multilabel classification.

KNN gives more accurate predictions, and lets me do really fast training, which is advantageous when you are trying different parameters, but once trained the model is very slow at the moment of making predictions (it's a "lazy learner"). SVC, on the other hand, is an "eager learning" algorithm. It is very slow when training but much faster than KNN when making predictions. I used both models in the training process and used SVC as the final option for predictions.

A frequent question I get when talking about model training is: What are the best/most common values to use in model parameters (n_neighbors, in the case of KNN, and C and gamma, in the case of SVC)? You can use hyperparameter tuning to answer this question. However, the hyperparameter tuning gives you a set of parameters that

achieve higher accuracy, which is not necessarily the ultimate answer that you are looking for. What is accuracy anyway? Sometimes accuracy is not the best way to measure the success of your algorithm, it just shows how truthful the algorithm is to the data you fed in initially. What if the data is not good enough? But that is a subject for a future blog post.

With CLASSIEfier I learned (the hard way) that the "data-dependent parameters" were more influential in the final classifications than the model parameters - i.e. the maximum number of words needed per document, the minimum number of documents per label, whether or not the text should be pre-processed. In the end, I landed on a maximum of 3000 characters per document (around 500 words), a minimum of 10,000 documents per label, and concluded unprocessed text gave better context. For example, 'swimmer' refers to 'professional swimmer' (beneficiary) while 'swim' refers to a 'recreational activity' (subject). Needless to say, more words and more documents made the process slower thus I was always looking for a bare minimum.

The way you convert words into numbers (so that the model can use mathematical equations) is also really important. I used TF-IDF and found that TF-IDF parameters make a big difference in processing time, and final results. I found that using a max_df of 20% was the best option, i.e. ignore words that appear in more than 20% of the documents. Why? If I have 17 labels, and I am training with an equal number of documents per label, only 5% of the documents will have common 'health' (a label) words. However due to the multilabel nature of the text, this 5% might overlap with another 10% of documents, so I used 20% to be in a safe zone. Words that are in more than 20% of the documents are just noise.

Snapshot of CLASSIEfier's log on data and model parameter tuning (not complete)

TF-IDF max	# of app per label	length app (bits)	recall	precision	applications without class	pre-processed text	duration(min)	Sample selection	applications	max_iter	GAMMA	Kernel	
0.1	10,000	3000	0.7421	0.6989	237	non-processed	2 days			4000	0.1	linear	
0.2	10,000	3000	0.6127	0.8012	3852	non-processed	minutes			20		knn 20	log tfIDF
0.2	10,000	3000	0.599	0.7575	4096	non-processed	minutes			10		knn	norma tfIDF
	20,000	3000	0.6049	0.8161	1071	non-processed	9min			30		knn	log tfIDF

CLASSIEfier Deployment

Disclaimer: All my deployment testing was done with AWS because our in-house application, SmartyGrants, was already using AWS. Other cloud services could be as good as AWS for this kind of task but I have not tested them myself.

As expected, deploying CLASSIEfier was the most difficult task. Why? Because CLASSIEfier is an extremely complicated algorithm (see this video for more detail). It is a combination of one model per level and some keyword-matching. Most of the examples found online use a nice lambda function (e.g. here) with a clean, light model. That was not possible in my case.

In general, when deploying a model with AWS lambda you follow these steps:

1. Wrap the model and the libraries used by the model in a docker container.
2. Use Flask to create an API (code) which will read the input text, call the model, predict and send a prediction.
3. Transfer that code to AWS lambda, zappa can help with that.
4. Deploy! Lambda will create an endpoint that you can later link to AWS API Gateway. That endpoint can be plugged to an application and then you can make predictions live.

Easy! But not for me. Mostly because CLASSIEfier is a combination of two models (one for level 1 and one for level 2) and some keyword-matching (for level 3 and 4). These are some of the challenges I faced when I tried this (and these are common challenges):

- The libraries I used to train my model sometimes were not compatible with zappa or flask or docker. Really painful! **How did I overcome this?** I had to retrain the model using the libraries' versions requested by the other applications.
- Lambda is designed to deal with light models and it has space and memory limitations. CLASSIEfier is above those limits. There are some hacks but they are tedious to implement. **The alternative** was to host the model in an EC2 AWS instance and create an endpoint from there. That's a different procedure and is not free.
- Every time CLASSIEfier is updated, I have to go through all those steps all over again.

Moving to the Cloud, the beauty of SageMaker

Disclaimer: SageMaker was my first test, for the reason mentioned above. Also, in this option you need to pay for the live endpoints, it is not free.

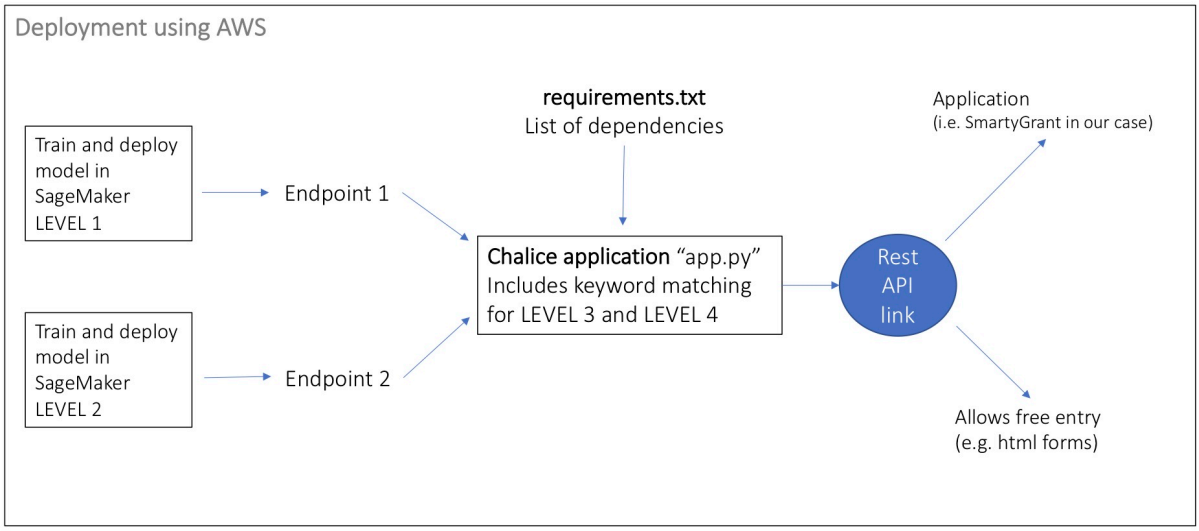
After trying the previous option over a span of two weeks it was natural to give SageMaker a go. SageMaker is a virtual machine hosted in an AWS instance. Here you can create your own Jupyter Notebooks and from the Notebooks you can train and deploy models. SageMaker in the background creates docker containers, EC2 instances and the lambda functions needed to deploy your model. You don't need to go through the hazards of matching library versions or dealing with lambda limitations.

First, I tried to bring my original scikit-learn models into SageMaker but it was easier to rebuild the model from scratch using the in-built algorithms – Blazing Text, in this case. Since I had already found the ideal "data-dependent parameters" it took me just a couple of days to convert my training data to the SageMaker format and retrain the model.

Deploying in SageMaker is literally one line of code. The steps here were:

1. SageMaker creates an endpoint, that endpoint can be plugged to your application and make predictions live. In the case of CLASSIEfier I have more than one model; therefore, more than one endpoint was created, thus extra steps were needed.
2. I created an API/wrapper that will join the endpoints and keyword-matching to a final classification. Chalice helped with that.
3. Deploying!

Snapshot of CLASSIEfier's deployment flow



Benefits of using AWS SageMaker:

- The whole experience of training, testing and deploying models is pain-free. SageMaker takes care of everything in the background, even optimising the right model parameters when training, keeping libraries consistent when creating docker containers, and optimising the size of the models.
  You only need to tell SageMaker what you want. What model? How much computer power? Where is the training data set sitting? What model parameters do you want? See the Notebook example below:

Snapshot of CLASSIEfier's training and deployment SageMaker Notebook (not complete)

```
%%time

train_channel = prefix + '/train/final'
validation_channel = prefix + '/validation'

sess.upload_data(path='data/classie.train', bucket=bucket, key_prefix=train_channel)
sess.upload_data(path='data/classie.valid', bucket=bucket, key_prefix=validation_channel)

s3_train_data = 's3://{}/{}'.format(bucket, train_channel)
s3_validation_data = 's3://{}/{}'.format(bucket, validation_channel)
```

```
CPU times: user 3.38 s, sys: 814 ms, total: 4.19 s
Wall time: 4.17 s
```

```
s3_train_data
```

```
's3://sagemaker-[redacted]/level1-sagemaker/train/final'
```

Next we need to setup an output location at S3, where the model artifact will be dumped. These artifacts are also the output of the algorithm's traning job.

```
s3_output_location = 's3://{}/{}/output'.format(bucket, prefix)
```

```
bt_model = sagemaker.estimator.Estimator(container,
                                         role,
                                         train_instance_count=1,
                                         train_instance_type='ml.c4.4xlarge',
                                         train_volume_size = 30,
                                         train_max_run = 360000,
                                         input_mode= 'File',
                                         output_path=s3_output_location,
                                         sagemaker_session=sess)
```

Please refer to algorithm documentation for the complete list of hyperparameters.

```
bt_model.set_hyperparameters(mode="supervised",
                             epochs=10,
                             min_count=2,
                             learning_rate=0.05,
                             vector_dim=10,
                             early_stopping=True,
                             patience=4,
                             min_epochs=5,
                             word_ngrams=2)
```

```
train_data = sagemaker.session.s3_input(s3_train_data, distribution='FullyReplicated',
                        content_type='text/plain', s3_data_type='S3Prefix')
validation_data = sagemaker.session.s3_input(s3_validation_data, distribution='FullyReplicated',
                             content_type='text/plain', s3_data_type='S3Prefix')
data_channels = {'train': train_data, 'validation': validation_data}
```

We have our `Estimator` object, we have set the hyper-parameters for this object and we have our data channels linked with the algorithm. The only remaining thing to do is to train the algorithm. The following command will train the algorithm. Training the algorithm involves a few steps. Firstly, the instance that we requested while creating the `Estimator` classes is provisioned and is setup with the appropriate libraries. Then, the data from our channels are downloaded into the instance. Once this is done, the training job begins. The provisioning and data downloading will take some time, depending on the size of the data. Therefore it might be a few minutes before we start getting training logs for our training jobs. The data logs will also print out Accuracy on the validation data for every epoch after training job has executed `min_epochs`. This metric is a proxy for the quality of the algorithm.

Once the job has finished a "Job complete" message will be printed. The trained model can be found in the S3 bucket that was setup as `output_path` in the estimator.

```
bt_model.fit(inputs=data_channels, logs=True)
```

```
2019-08-12 06:54:38 Starting - Starting the training job...
2019-08-12 06:54:40 Starting - Launching requested ML instances......
2019-08-12 06:55:43 Starting - Preparing the instances for training...
2019-08-12 06:56:35 Downloading - Downloading input data
2019-08-12 06:56:35 Training - Downloading the training image..
Arguments: train
[08/12/2019 06:56:51 WARNING 140640004896576] Loggers have already been setup.
[08/12/2019 06:56:51 WARNING 140640004896576] Loggers have already been setup.
[08/12/2019 06:56:51 INFO 140640004896576] nvidia-smi took: 0.02516913414 secs to identify 0 gpus
[08/12/2019 06:56:51 INFO 140640004896576] Running single machine CPU BlazingText training using supervised mode.
[08/12/2019 06:56:51 INFO 140640004896576] Processing /opt/ml/input/data/train/classie.train . File size: 340 MB
[08/12/2019 06:56:51 INFO 140640004896576] Processing /opt/ml/input/data/validation/classie.valid . File size: 17 MB
Read 10M words
```

```
text_classifier = bt_model.deploy(initial_instance_count = 1,instance_type = 'ml.m4.xlarge')
```

- The BlazingText model is performing as well as the scikit-learn local models.
- I can access the Jupyter Notebooks from any computer by using the AWS console. This is really handy when you work from home or when you travel often.
- CLASSIEfier updates are easy to apply just by re-running the Jupyter Notebooks.

Drawbacks of using AWS SageMaker:

- SageMaker takes care of everything in the background! As much as this is an advantage this can also be a disadvantage. It means that the black box is becoming darker.
  For example:

1. It is a disadvantage if you have not trained models from scratch before. A junior data scientist will not understand what is happening behind the scenes. This raises a range of questions: How do optimisation and parallelisation work? How much weight do different model parameters have over the model? How are the words being converted to number space to train the model? etc.
2. It is very difficult to identify biases and bugs in this environment. The only measurement you get is accuracy and other scores, which do not necessarily tell you if your model is biased or giving wrong predictions. If you discover that your model is wrong, it will be difficult to debug.

   The only reason why it was so simple for me to use SageMaker is because I knew the training data in detail, and when I found suspicious results I could loop back to the data and fix it.

- I couldn't find an in-build SageMaker algorithm that deals with multilabel classification. To train in Blazing text I had to repeat the multilabel documents. For example, if a text is to be classified 'health' and also 'arts and recreation', I label the text 'health' then I repeat the document and label it 'arts and recreation'. This makes the training data set heavier and the training and prediction process slower.

  It also affects the final score. In the previous example, instead of getting a 90% probability of being 'health and arts and recreation', I get 50% probability of being 'health' and 50% probability of being 'arts and recreation'.

  I fixed this by writing a scoring algorithm that normalises the data. This is not ideal but it serves the purpose.

- It is not free.

Conclusion:

SageMaker is a great tool for senior Data Scientists who know well how machine learning works and know what to expect from model training and deployment. If you are thinking of using SageMaker you need to be hyperaware of model and data biases and

know the training data in detail to avoid bad AI design. For example, if CLASSIEfier provides marginally wrong classifications, such as nesting 'kids recreational activities' under 'sport professionals' this could have serious consequences. Grantmakers might think that they are supporting too many professional sport causes and cut down the funding to those areas, when in reality they were simply funding kids. Overall, though, SageMaker, when used carefully, can save you a lot of time and give you a pain-free machine learning journey.

It's worth mentioning that besides AWS, there are other cloud providers, such as Google or Microsoft, who offer similar services to SageMaker (e.g. Google Datalab, Microsoft Azure Machine Learning). I encourage you to try as many of them as possible and pick the one most suitable to your case.

Do not forget that cloud-based machine learning services will have a monthly cost and you need to have a budget for it (or use free-trial accounts for testing).

Author

Paola Oliva-Altamirano - Data Scientist, Our Community

Looking to learn more? Visit our Innovation Lab page. Ideas? Feel free to reach out!

With the help of:

- Kathy Richardson – Executive Director, Our Community
- Sarah Barker – Director of Data Intelligence, Our Community
- Nathan Mifsud – Data scientist, Our Community