CS5785 Image Search Engine

Anonymous Author(s)
Affiliation
Address
email

Abstract

Image search engines attempt to return the most relevant image to a given input request. Through machine learning techniques, more relevant images can be returned more frequently, improving user experience. We explore the uses of linear regression, random forest, and neural nets to better improve search engine performance. Through pre-processing techniques and a neural net, we were able to achieve a map 20 score of .3.

1 Introduction

1.1 Pre-Processing

Increasing Data

The information supplied by the "tag" files were limited, and at points empty. In order to address this issue, our team utilized a Google Vision api to parse the images and generate more key words. Our images now had significantly more tags, and no images were empty from corresponding tags.

Nltk and Stop Words

With the data input being sentences, our team wanted to extract the key words to better understand what each input was attempting to retrieve. To do this, the nltk library was utilized to remove stop words such as: punctuation, adverbs, prepositions, determiners, and to's.

Word2Vec

To better relate similar words, our team turned the pre-processed sentences and tags into word vectors. This was done by utilizing the word2vec library, which generated 300 feature vectors for each description and each set of tags.

Standardization

Machine learning regression algorithms work best when both the input and output data are standardized. In order to accomplish this, the description vectors, tag vectors, 1,000 fc1000 ResNet features, and 2,048 pool5 ResNet features were processed using sklearns "StandardScalar" function.

Input/Output Data

The input data would be the 300 word vector from the image descriptions. The output data is a concatenation of all the Resnet features and the generated tag vectors, totalling a 3348 output feature vector.

1.2 Model

The model used is a shallow neural net, containing 6696 nodes in its single hidden layer with a Relu activation function, .5 dropout rate, an Adam optimizer with a learning rate of .01, linear output

activation function, 150 epochs, and a batch size of 5000. A neural net was chosen for this task as neural nets can handle high dimensional data extremely well, especially since our dimensions for the output will be 3348.

2 Experimentation

2.1 Pre-Processing

Different pre-processing techniques were attempted, but led to less-than-optimal outcomes. The three main other pre-processing techniques attempted were:

- Normalization
- MinMax scaling
- Doubling the data
- Polynomialize the data

Normalizing the data or MinMax scaling the data between -1 to 1 led to significant performance drops in every regression algorithm, while StandardScalar improved our performance.

Our team thought we could double our data input with our brand new generated tags. The first half of our data was the original input descriptions and the outputs were the feature vectors and the tag vectors. The second half had the tag vectors and the description vectors swapped, which generated new input "sentences". Although performance was expected to dramatically increase, it resulted in no performance boost at all. This could simply be because swapping the input and the output had no effects on the weights, and therefore no effect on the performance of our model.

Polynomializing the data had insignificant effect as well.

2.2 Ridge Regression

The baseline code supplied a ridge regression technique with a map20 score of .11. By simply standardizing the data, we improved that score to .14. Through our pre-processing techniques we were able to achieve a score of .25.

One of the issues our team ran into was when running Ridge through cross-validation, it would consistently always choose the highest alpha. Alphas above 10 though under-performed, and therefore an alpha of 10 was chosen.

2.3 Random Forest

Random forest regressors were the worst performance algorithm out of all that were tested (except for a Deep Neural Net). No matter the depth or the tree amounts, their performance was always subpar and below the baseline example supplied. Due to this, our team quickly strayed away from using a random forest. Our best score was 0.1.

2.4 BERT

BERT stands for Bidirectional Encoder Representations from Transformers. This was intended to be used instead of Word2Vec to encode a sentence into a vector then check the cosine similarities or train a model. The benefit of BERT is that it can encode an entire sentence at once and uniquely encode it rather than a single word at a time. Not only was this computationally intensive, but the results were not any better than Word2Vec. The best map20 outcome was 0.19.

2.5 Siamese Network

A lot of pre-processing was necessary to train a Siamese Network. The Siamese network took 3 inputs to train, X features, Y features, and a class saying if it is a match or not. This meant each X-Y pair was assigned a class of 1 if they were a match and 0 if they were not. The data given were all

matches, meaning many 0 classes had to be created by randomly pairing X features with different Y features. This increased the training size into the millions, where matches were very sparse.

The Siamese Network would then train two neural networks in parallel with shared weights and then compare the outputs with cosine similarity, as seen in Figure 1:

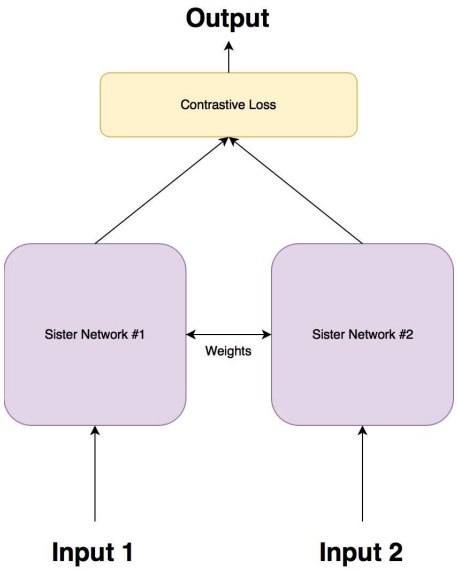

Figure 1: Siamese Network (medium.com/intel-student-ambassadors)

Siamese networks were promising, but took much longer to hyper-tune than a simple shallow neural network. Because of this, the shallow network consistently outperformed the Siamese network. The best Map20 result was 0.20.

2.6 Shallow Neural Network

A shallow neural network consists of an input layer, a single hidden layer, and an output layer. The architecture of a shallow neural net can be seen in Figure 2:

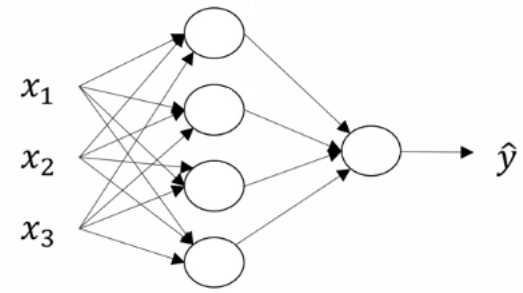

Figure 2: Shallow Neural Network Architecture (miro.medium.com)

After hyper-tuning the parameters, our shallow neural network ended up being our best performing algorithm. Although outputting the best performance, it required the most slight adjustments to its parameters,

Nodes

The amount of nodes for the hidden layer was one of the major tuning parameters. Too little nodes lead to a tremendous performance drop, while too many nodes lead to the same end. Through trial and error, 2 possible choices were observed: 3348 nodes (the same as the output features) with a

dropout layer of .2 to prevent over-fitting, or 6696 nodes with a droupout layer of .5. Both performed relatively similar, but the 6696 performed slightly better, and with a dropout layer of .5 was able to prevent over-fitting slightly better.

Activation Functions

Many activation functions were tested:

- Relu
- Elu
- Linear
- Sigmoid

Sigmoid was quickly discarded, as we were using this NN as a regressor, and sigmoid is more commonly used for classification

linear in the hidden layer would not have worked well, as continually passing values through linear functions is simply just another linear function.

elu and relu proved most useful, and gave the best performance. Overall, relu gave the best performance over elu under mostly every circumstance.

Optimizers

Two optimizers were tested:

- SGD
- Adam

Similar to elu vs Relu, the Adam optimizer gave the best results for every test, and was chosen as our main optimizer.

Layers

To determine whether a Shallow Neural Net was better than a Deep Neural Net, our team had to run tests with both few and many hidden layers. All of the tests our team ran consisted of:

- Differentiating the amount of layers
- Differentiating the amount of nodes per layer
- Differentiating the amount of epochs

Every test lead to the same result: 1 hidden layer for this data will always perform best. Adding more layers lowers the performance, no matter the hyper tuning parameters tested.

Epochs

Epochs was an important part of hyper-tuning as well. Too little epochs and the model will not converge, but too many epochs will cause the model to over-fit. In order to determine this, we would run the model over a large amount of epochs, view where our model did best on the validation set, and set our new epoch amount to the epochs seen. Epoch example can be seen in Figure 3:

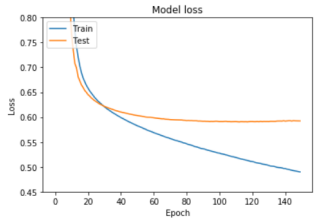

Figure 3: Loss vs Epochs

Learning Rate and Batch Size

The learning rate settled on was .01 for the Adam optimizer. When testing other learning rates we found that learning rates great than .01 (ex: 1) returned non-optimal results, while the same happened for learning rates less than .01 (ex: .005). There may be learning rates better that we simply missed, but the changes in results from slightly changing the .01 learning rate did not effect our outcome a significant amount.

Batch size did have a large effect on our outcome. We tested batch sizes of 1000, 2000, 5000, and 8000. What was found was that a batch size that was too small (ex:1000) returned results that were much lower than the higher batch sizes. That being said, batch sizes too large (ex:8000) returned a performance that was below the optimal outcome. Too Large batch size was better than the too small size, but the optimal was somewhere between both sizes. Our team found that a batch size of 5000 was able to get a map20 score of .3, while 2000 returned .25, 1000 returned .21, and 8000 returned .26.

Loss Function

Loss functions were also explored for different outcome performances. We explored two different loss functions:

- Mean Squared Error
- Mean Average Error

Our initial thought was that Mean Average Error will provide a better performance than Mean Squared Error, as it does not exponentially penalize the values when a single value is far away. After running both loss functions though, the outcomes were similar and we deemed that both were a viable option for determining the performance of our model.

2.7 Similarity Calculation

Our team tested two different distance functions:

- Euclidean
- Cosine

Euclidean distance was the first distance function tested but proved to not be the best in determining high dimensional distances. Because of this, our team switched over to using cosine distance, which is commonly used for higher dimensional data. This increased our score incredibly, when testing we had increased our score from .15 to .21 by doing this switch. Other distances we would test in the future are Manhattan distance, which has been recommended by many, and a few others which were designed to work with higher dimensions.

3 Results

Ultimately our team chose a Neural Network which contained an input layer with 300 features, 1 hidden layer with 6696 nodes with a relu activation function, a dropout layer of .5, and an output layer 3348 nodes with a linear activation function. The input data is the 300 word vector produced from the users input sentences, and the put put is the Resnet Features along with the tag word vectors. The model was then run with 150 epochs and a batch size of 5000. This lead us to our best score: .3.

To get better results, we would look into better pre-processing techniques, along with looking to expand upon our top performing models, as the top 2 or 3 models may perform much better with more pre-processing. Along with this, more hyper-tuning of the neural net parameters would have better improvement on our score.

The results for each of our models can be seen below in Table 1:

Table 1: Model Results

Algorithm	**Map20 Score**
Shallow NN	.3
Ridge	.25
Siamese NN	.2
BERT	.18
Random Forest	.1
Deep NN	.003

4 Citation

1. https://keras.io/
2. https://scikit-learn.org/stable/
3. https://numpy.org/
4. https://www.nltk.org/
5. https://pypi.org/project/bert-embedding/
6. https://medium.com/mlreview/implementing-malstm-on-kaggles-quora-question-pairs-competition-8b31b0b16a07
7. https://cloud.google.com/vision/
8. https://medium.com/@adriensieg/text-similarities-da019229c894
9. https://d4datascience.wordpress.com/2016/09/29/fbf/