Fake news detection

Academic approach using different classification methods

IS 590DT Data Mining

Information School
University of Illinois at Urbana-Champaign

Worawich (Win) Chaiyakunapruk

December 18, 2018

Date Performed:	December 4, 2018
Advisor:	Dr. Vetle Torvik

Abstract

Nowadays the news spread really fast over the internet. Sometimes, people shared it without knowing it was not true. This is not limited to only social network users, but many times even BBC or CNN had spread misleading contents to consumers. Having a reliable fake news detector would be a big step toward people's accurate data perception, which could prevent misunderstood problems and therefore more opportunities and agreement.

1 Objective

To determine whether if the news is credible and should be trusted, by creating a reliable fake news detector.

Approach by using the data set which was customly scraped from the websites that was well known for distributing fake news together with the data from big news agencies. The data was trained using Bayesian Network.

2 Defining fake news

Fake news does not need to be a news that was made up to be fake. Instead, it referred to an untruthful news, so called biased. Most of the news nowadays contain somewhat biasness, it just the matter of is it more or less. An ideal news report is the one with the most neutral perspective, or just stating the fact. The biasness reference was from the mediabiasfactcheck.com which listed the bias data of all the news websites.

However, with all this information, fake news still isn't easy to spot by human decision, as well as computer classification. From the research that was conducted, training the model by category would give less diverse of the news pattern, and therefore, give the better accuracy model.

3 Approach

From the research and the talk with many experts in the field, it was concluded that the better model will be approached by using Weka to initialize the basic model accuracy and further explore the possibilities by using other tools, such as Python, Sci-kit learn, Tensor Flow, Keras, and MXNet.

The main approaching steps of the project are

i. Scraping the data

As mentioned earlier, the fake news data source were from the list of bias websites, such as inforwars.com, along with other websites on mediabiasfactcheck.com. Python and Beautiful Soup were used as the main tools to construct the web crawler. The functions to pull the data were created by using Beautiful Soup to do the text parser, together with inspecting the html code of the page then customized the Regular Expression to pull the title and modify it using regex.sub and put into the URL, to be able to pull the webpage, and filter out the article using the same strategy as pulling the URL. Multiple webpages crawling was achieved by creating a function to change the page and modified it into the URL variable. All the pulled data were collected into the data frame for easy management.

ii. Pre-processing the data

Pre-processing is actually one of the most important and also most challenging parts of the project. The first method, which is the most popular and basic one was Word2vec. Word2vec was easily achieved in Weka, by using a series of filters, such as NominalToString, StringToWordVector, and MathExpression to normalize the data. (10-fold cross validation was done when ran model in Weka)

However, that was just the basic approach. The pre-processing model in Python was more complicated. Splitting the data into k-fold and word wrapper and tokenizer needed to be done before making it a vector, unless it will consume much more memories than it needed to. (The K-fold cross validation was done using StratifiedKFold process to get the train and test data set.) Also, using the same word2vec feature in Pyhton could be a lot more complicated[a]. In addition, from the research, it was proved that using TF-IDF matrix[b] would give a more accurate model. Furthermore, it could be used together with word2vec to give even better result[c]. (Accuracy was improved by 5-10%)

The possibility of applied Latent Semantic Analysis (LSA) also has been explored by applying after each feature extraction algorithm. The accuracy was slightly improved for most of the time[d].

iii. Analyzing the data

In Weka Explorer, the basic classifications that have potential in classifying data with correlation were used to initiate the accuracy for further adjustment in python. The model that falls into account were Naïve Bayes, Bayesian Network, Supported Vector Machine, and Convolutional Neuron Network.

The packages that were used to run the model on Python were Sci-kit Learn, Keras, Tensor Flow, and Pomegranate.

4 Results and Conclusions

i. Weka

I've sampled the data to 600 before testing a full dataset at 6,000 observations. From the experiments with 6,000 observations and 5-fold cross validation, the best accuracy was 89.84% accurate, which performed by Random Forest followed by 89.20% by support vector machine (SVM) with sequential minimal optimization (SMO) applied. Followed by stochastic gradient descent (SGD) at 88.20% accuracy and Multinomial Naïve Bayes which is very slightly less accurate at 87.97%.

Other decent model which scored about 5% lower are Bayesian Network with 2 parents at 87.71%, Bernoulli Naive Bayes at 82.30%, REPTree at 80.94% and Logistic Regression at 78.82%. I have terminated some of the process that could be really accurate, which is Multilayer Perceptron and Bayesian Network with 5 parents, because it takes much longer than other. Therefore, I'll run them on the cloud server later.

The result performed by Random Forest.

```
=== Summary ===

Correctly Classified Instances        4026               89.846  %
Incorrectly Classified Instances       455               10.154  %
Kappa statistic                          0.797
Mean absolute error                      0.2873
Root mean squared error                  0.3264
Relative absolute error                 57.4558 %
Root relative squared error             65.2819 %
Total Number of Instances             4481

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.923    0.125    0.879      0.923    0.900      0.798    0.963     0.958     FAKE
                 0.875    0.077    0.920      0.875    0.897      0.798    0.963     0.966     REAL
Weighted Avg.    0.898    0.101    0.899      0.898    0.898      0.798    0.963     0.962

=== Confusion Matrix ===

    a    b   <-- classified as
 2050  172 |    a = FAKE
  283 1976 |    b = REAL
```

The result performed by support vector machine (SVM) with sequential minimal optimization (SMO) applied.

```
=== Summary ===

Correctly Classified Instances        3997               89.1988 %
Incorrectly Classified Instances       484               10.8012 %
Kappa statistic                          0.784
Mean absolute error                      0.108
Root mean squared error                  0.3287
Relative absolute error                 21.6038 %
Root relative squared error             65.7325 %
Total Number of Instances             4481

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.898    0.114    0.886      0.898    0.892      0.784    0.892     0.846     FAKE
                 0.886    0.102    0.899      0.886    0.892      0.784    0.892     0.853     REAL
Weighted Avg.    0.892    0.108    0.892      0.892    0.892      0.784    0.892     0.850

=== Confusion Matrix ===

    a    b   <-- classified as
 1996  226 |    a = FAKE
  258 2001 |    b = REAL
```

The result performed by stochastic gradient descent (SGD).

```
=== Summary ===

Correctly Classified Instances        3979               88.7971 %
Incorrectly Classified Instances       502               11.2029 %
Kappa statistic                          0.7759
Mean absolute error                      0.112
Root mean squared error                  0.3347
Relative absolute error                 22.4072 %
Root relative squared error             66.9436 %
Total Number of Instances             4481

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.887    0.111    0.887      0.887    0.887      0.776    0.888     0.843     FAKE
                 0.889    0.113    0.889      0.889    0.889      0.776    0.888     0.846     REAL
Weighted Avg.    0.888    0.112    0.888      0.888    0.888      0.776    0.888     0.845

=== Confusion Matrix ===

    a    b   <-- classified as
 1970  252 |    a = FAKE
  250 2009 |    b = REAL
```

The result performed by Multinomial Naïve Bayes.

```
=== Summary ===

Correctly Classified Instances        3942               87.9714 %
Incorrectly Classified Instances       539               12.0286 %
Kappa statistic                          0.7595
Mean absolute error                      0.1278
Root mean squared error                  0.3314
Relative absolute error                 25.5579 %
Root relative squared error             66.2788 %
Total Number of Instances             4481

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.911    0.151    0.856      0.911    0.882      0.761    0.935     0.916     FAKE
                 0.849    0.089    0.906      0.849    0.877      0.761    0.935     0.943     REAL
Weighted Avg.    0.880    0.120    0.881      0.880    0.880      0.761    0.935     0.930

=== Confusion Matrix ===

    a    b   <-- classified as
 2024  198 |    a = FAKE
  341 1918 |    b = REAL
```

The result performed by Bayesian Network with 2 parents.

```
=== Summary ===

Correctly Classified Instances        3796               84.7132 %
Incorrectly Classified Instances       685               15.2868 %
Kappa statistic                          0.6946
Mean absolute error                      0.154
Root mean squared error                  0.3723
Relative absolute error                 30.8032 %
Root relative squared error             74.4529 %
Total Number of Instances             4481

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.914    0.219    0.804      0.914    0.856      0.701    0.933     0.932     FAKE
                 0.781    0.086    0.903      0.781    0.837      0.701    0.933     0.935     REAL
Weighted Avg.    0.847    0.152    0.854      0.847    0.847      0.701    0.933     0.933

=== Confusion Matrix ===

    a    b   <-- classified as
 2032  190 |    a = FAKE
  495 1764 |    b = REAL
```

The result performed by Bernoulli Naive Bayes.

```
=== Summary ===

Correctly Classified Instances        3688               82.3031 %
Incorrectly Classified Instances       793               17.6969 %
Kappa statistic                          0.6462
Mean absolute error                      0.1769
Root mean squared error                  0.4196
Relative absolute error                 35.3789 %
Root relative squared error             83.927  %
Total Number of Instances             4481

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.850    0.204    0.804      0.850    0.827      0.647    0.866     0.803     FAKE
                 0.796    0.150    0.844      0.796    0.819      0.647    0.872     0.837     REAL
Weighted Avg.    0.823    0.177    0.824      0.823    0.823      0.647    0.869     0.820

=== Confusion Matrix ===

    a    b   <-- classified as
 1889  333 |    a = FAKE
  460 1799 |    b = REAL
```

The result performed by REPTree.

```
=== Summary ===

Correctly Classified Instances        3627               80.9418 %
Incorrectly Classified Instances       854               19.0582 %
Kappa statistic                          0.6187
Mean absolute error                      0.2465
Root mean squared error                  0.3829
Relative absolute error                 49.3    %
Root relative squared error             76.5783 %
Total Number of Instances             4481

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.792    0.173    0.818      0.792    0.805      0.619    0.869     0.845     FAKE
                 0.827    0.208    0.801      0.827    0.814      0.619    0.869     0.853     REAL
Weighted Avg.    0.809    0.191    0.810      0.809    0.809      0.619    0.869     0.849

=== Confusion Matrix ===

    a    b   <-- classified as
 1759  463 |    a = FAKE
  391 1868 |    b = REAL
```

The result performed by Logistic Regression.

```
=== Summary ===

Correctly Classified Instances        3532               78.8217 %
Incorrectly Classified Instances       949               21.1783 %
Kappa statistic                          0.5766
Mean absolute error                      0.2123
Root mean squared error                  0.4572
Relative absolute error                 42.4529 %
Root relative squared error             91.4442 %
Total Number of Instances             4481

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.814    0.237    0.771      0.814    0.792      0.577    0.845     0.794     FAKE
                 0.763    0.186    0.807      0.763    0.784      0.577    0.842     0.815     REAL
Weighted Avg.    0.788    0.211    0.789      0.788    0.788      0.577    0.843     0.805

=== Confusion Matrix ===

    a    b   <-- classified as
 1809  413 |    a = FAKE
  536 1723 |    b = REAL
```

ii. Python

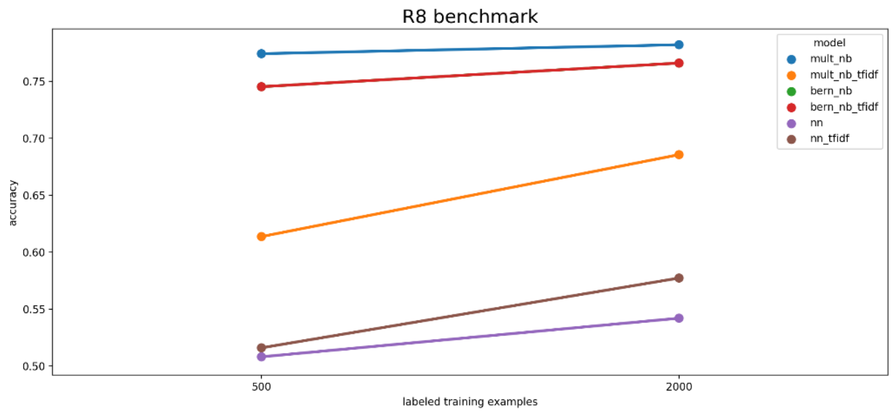
Models used names:

1) mult_nb = Multinomial Naïve Bayes
2) bern_nb = Bernoulli Naive Bayes
3) svc = linear kernel Support Vector Machine
4) SVM = RBF kernel Support Vector Machine
5) glove_small - ExtraTrees with 200 trees and vectorizer based on 50-dimensional gloVe embedding trained on 6B tokens
6) glove_big = same as above but using 300-dimensional gloVe embedding trained on 840B tokens
7) w2v = same but with using 100-dimensional word2vec embedding trained on the benchmark data itself (using both training and test examples [but not labels!])
8) nn = Neural Network (Multi-layer Perceptron)

The result from using 600 sample observations trained with 100 data points vs 500 data point

model	score
w2v_tfidf	0.8530
w2v	0.8479
glove_small_tfidf	0.7956
mult_nb	0.7815
glove_small	0.7746
bern_nb	0.7632
bern_nb_tfidf	0.7632
mult_nb_tfidf	0.7174
nn_tfidf	0.6401
nn	0.5863

Number of Folds = 5

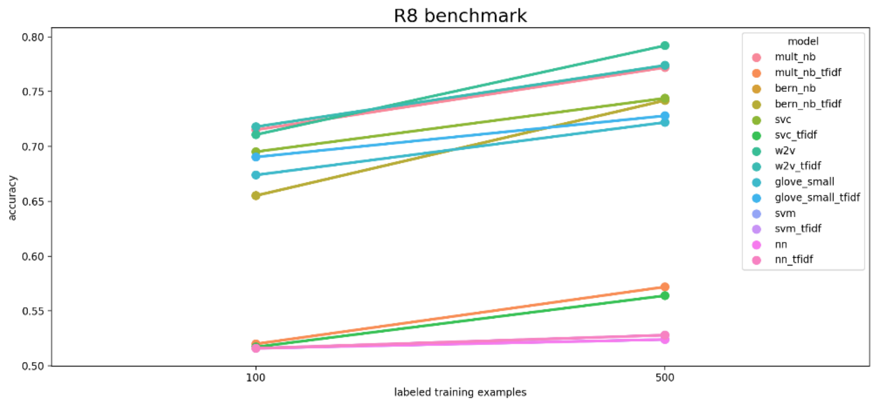


The result from using all 6,000 observations trained with 500 data points vs 2,000 data point

model	score
mult_nb	0.7900
w2v	0.7683
w2v_tfidf	0.7683
svc	0.7600
bern_nb	0.7567
bern_nb_tfidf	0.7567
glove_small_tfidf	0.7400
glove_small	0.7117
mult_nb_tfidf	0.5583
svc_tfidf	0.5467
svm	0.5200
svm_tfidf	0.5200
nn	0.5200
nn_tfidf	0.5200

Number of Folds = 5

.



iii. CNN using Apache MXNet

I have found the code from the tutorial part of mxnet.incubator.apache.org. I've set it to 50 epochs and run it on AWS.

```
optimizer rmsprop
maximum gradient 5.0
learning rate (step size) 0.0005
epochs to train for 50
Iter [0] Train: Time: 107.560s, Training Accuracy: 68.094
--- Dev Accuracy thus far: 88.000
Iter [1] Train: Time: 100.771s, Training Accuracy: 86.132
--- Dev Accuracy thus far: 90.500
Iter [2] Train: Time: 101.234s, Training Accuracy: 90.472
--- Dev Accuracy thus far: 92.300
Iter [3] Train: Time: 101.387s, Training Accuracy: 93.245
--- Dev Accuracy thus far: 93.700
Iter [4] Train: Time: 101.461s, Training Accuracy: 95.038
--- Dev Accuracy thus far: 94.100
Iter [5] Train: Time: 101.830s, Training Accuracy: 96.434
--- Dev Accuracy thus far: 93.700
Iter [6] Train: Time: 101.480s, Training Accuracy: 97.019
--- Dev Accuracy thus far: 94.500
Iter [7] Train: Time: 101.441s, Training Accuracy: 97.830
--- Dev Accuracy thus far: 94.600
Iter [8] Train: Time: 101.483s, Training Accuracy: 98.245
--- Dev Accuracy thus far: 95.000
Saved checkpoint to ./cnn-0009.params
Iter [9] Train: Time: 101.465s, Training Accuracy: 99.038
--- Dev Accuracy thus far: 95.800
Iter [10] Train: Time: 101.407s, Training Accuracy: 99.245
--- Dev Accuracy thus far: 95.200
Iter [11] Train: Time: 101.525s, Training Accuracy: 99.321
--- Dev Accuracy thus far: 95.300
Iter [12] Train: Time: 101.452s, Training Accuracy: 99.509
--- Dev Accuracy thus far: 95.500
Iter [13] Train: Time: 101.455s, Training Accuracy: 99.585
--- Dev Accuracy thus far: 95.700
```

```
Iter [14] Train: Time: 101.514s, Training Accuracy: 99.698
--- Dev Accuracy thus far: 95.300
Iter [15] Train: Time: 101.462s, Training Accuracy: 99.679
--- Dev Accuracy thus far: 96.200
Iter [16] Train: Time: 101.517s, Training Accuracy: 99.792
--- Dev Accuracy thus far: 96.400
Iter [17] Train: Time: 101.396s, Training Accuracy: 99.868
--- Dev Accuracy thus far: 96.200
Iter [18] Train: Time: 101.580s, Training Accuracy: 99.925
--- Dev Accuracy thus far: 96.500
Saved checkpoint to ./cnn-0019.params
Iter [19] Train: Time: 101.555s, Training Accuracy: 99.849
--- Dev Accuracy thus far: 96.000
Iter [20] Train: Time: 101.463s, Training Accuracy: 99.830
--- Dev Accuracy thus far: 96.100
Iter [21] Train: Time: 101.424s, Training Accuracy: 99.868
--- Dev Accuracy thus far: 96.400
Iter [22] Train: Time: 101.570s, Training Accuracy: 99.906
--- Dev Accuracy thus far: 96.100
Iter [23] Train: Time: 101.477s, Training Accuracy: 99.925
--- Dev Accuracy thus far: 95.800
Iter [24] Train: Time: 101.476s, Training Accuracy: 99.906
--- Dev Accuracy thus far: 95.900
Iter [25] Train: Time: 101.442s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 96.200
Iter [26] Train: Time: 101.567s, Training Accuracy: 99.943
--- Dev Accuracy thus far: 96.000
Iter [27] Train: Time: 101.566s, Training Accuracy: 99.943
--- Dev Accuracy thus far: 96.100
Iter [28] Train: Time: 101.646s, Training Accuracy: 99.868
--- Dev Accuracy thus far: 95.700
Saved checkpoint to ./cnn-0029.params
Iter [29] Train: Time: 101.586s, Training Accuracy: 99.925
--- Dev Accuracy thus far: 96.300
Iter [30] Train: Time: 101.552s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 95.500
Iter [31] Train: Time: 101.630s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 96.600
Iter [32] Train: Time: 101.655s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 95.700
Iter [33] Train: Time: 101.613s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 96.000
Iter [34] Train: Time: 101.588s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 96.100
Iter [35] Train: Time: 101.646s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 95.800
Iter [36] Train: Time: 101.663s, Training Accuracy: 100.000
--- Dev Accuracy thus far: 96.300
Iter [37] Train: Time: 101.664s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 95.300
Iter [38] Train: Time: 101.669s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 96.400
Saved checkpoint to ./cnn-0039.params
Iter [39] Train: Time: 101.565s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 95.800
Iter [40] Train: Time: 101.445s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 96.300
Iter [41] Train: Time: 101.526s, Training Accuracy: 100.000
--- Dev Accuracy thus far: 95.900
Iter [42] Train: Time: 101.487s, Training Accuracy: 99.962
```

```
--- Dev Accuracy thus far: 95.500
Iter [43] Train: Time: 101.549s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 96.000
Iter [44] Train: Time: 101.567s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 96.600
Iter [45] Train: Time: 101.657s, Training Accuracy: 100.000
--- Dev Accuracy thus far: 95.500
Iter [46] Train: Time: 101.609s, Training Accuracy: 100.000
--- Dev Accuracy thus far: 96.500
Iter [47] Train: Time: 101.624s, Training Accuracy: 100.000
--- Dev Accuracy thus far: 96.600
Iter [48] Train: Time: 101.659s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 95.900
Saved checkpoint to ./cnn-0049.params
Iter [49] Train: Time: 101.641s, Training Accuracy: 99.943
--- Dev Accuracy thus far: 96.700
```

5 Discussion of Experimental Result and Uncertainty

.

Since I couldn't finish writing the whole report on time. So I put this in to compensate with this missing part. Sorry, I might finish the report/project later if I do the independent study on this.

Fall 2018 IS590DT Final

Due by Tuesday, December 18, 5PM US Central Time

Answer **3 of the 5** questions, in your own words. They count equally. Upload your answers to the final assignment section of the class Moodle page as a single narrative document in pdf format. You may, and **are encouraged to, illustrate your answers using Weka**, but that's no substitute for lucid natural language explanations. To preserve the natural flow of the narrative, figures and tables should be embedded into the document near their first mention. Any supplementary files like code or data should be referenced in the text and separately uploaded. You may use books, articles, notes, search engines, or computers, but **may not solicit or receive direct assistance from other human beings**. Cite sources if you use them.

Question 1. Describe and discuss some differences between classification and clustering.

Classification we know about data. We have labels for data (supervised), but clustering is just cluster the data without giving label. (unsupervised)

Question 2. What is your favorite classification or clustering algorithm? Why?

K-mean. It's the most basic one yet powerful depends on your idea, on what you put in. It's fast, powerful, efficient, and easy to use.

Question 3. Explain some aspects that might influence the accuracy of classification?

Noise and missing values. Because they could mislead the training model and give confusion to the algorithm.

Question 4. Describe and discuss two different techniques for dealing with the "curse of dimensionality".

Question 5. What is the purpose of cross-validation?

6 Problems

There were so many problems. Most of the time it's just the model doesn't work, or the accuracy doesn't make sense. But the biggest one is just can't make the model to run or can't finish the whole step, especially CNN. Since many people said it's the best accuracy and worth give it a try. I spent so much time debugging and still could not get it to work. Another one is the Bayesian Network, I've tried many packages, but they're all failed except the one in Weka. Also another problem is Jupyter notebook doesn't have package manager and the version control need to be done manually. Many times, the packages couldn't work together. Therefore, I have migrated the code to Pycharm because of its package version manager tool.

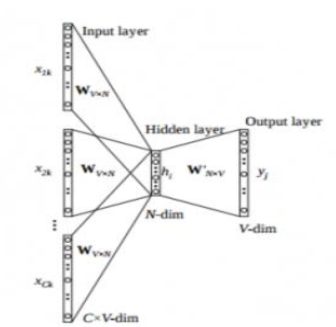
7 Further work and suggestion

To improve the accuracy of the model, the addition dependencies could be added into account, such as the website's origin and the traffic flow. As the data already contained the list of bias websites, the untruth news sources are already known, and therefore can be used to train the model. The website's registration data from whois.com and the website's traffic flow from Google Analytics together with the fake-ness from mediabiasfactcheck.com could be used to trained the model for further work exploration.

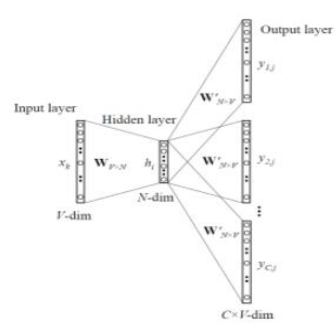
8 Answers to Definitions

a. **Word2vec** method applies shallow neural network to generate the vector of a word. There are 2 approaches for this particular algorithm: 1) Continuous Bag of Words 2) Skip-gram. In the CBOW approach, the word is predicted based on its context word, whereas in the skip-gram approach, the context words are predicted based on some input word.

CBOW



Skip-Gram



Softmax function is used in the output layer to calculate the probability of the output. Word vectors are trained using the backpropagation method. The weights obtained after the model is trained, are basically the vector elements of the corresponding word.

In a text classification problem, a document contains multiple words.

These word vectors of a particular document can be added and the average of them can be taken to come up with a single document vector.

b. **TF-IDF** stands for term frequency-inverse document frequency. This method is more wildly used in weighed information retrieval and text mining. Term frequency is basically the ratio of the number of times a term appears in a document and the total number of words in the document. Inverse document frequency is obtained by dividing the total number of documents by the number of documents containing the term and taking the logarithmic of the ratio. The idea is to give less weightage to commonly occurring words which don't convey much meaning.

c. **TF-IDF weighted Word2Vec** approach, the TF-IDF value of a particular word is multiplied by the corresponding word vector. The simple Word2Vec approach only gives the vector or the position of the word in n-dimensional space but it does not tell how important or how frequently of the words. The idea behind using this approach is to also consider the word importance along with its position in some n-dimensional space.

In a text classification problem, the word vectors of a particular document can be multiplied with the corresponding TF-IDF value and then the weighted average can be taken to come up with a single document vector.

d. **LSA (Latent Semantic Analysis)** use singular value decomposition technique (SVD) to reduce the dimensionality of the futures. This method keeps the most useful information and make classification algorithm runs faster. Another advantage of this method is it allows us to train the data with a lot more observations. Even train with the whole dataset takes less than a minute. The drawback of this method ids we lose some information which could be useful. Most of the time, the accuracy will be increased but sometimes it could drop. The reason that most of the time LSA helps because TF-IDF matrix, by measure is a very sparse matrix and contains a lot of zeros, and values near zero.

Special thanks

Dr. Vetle Torvik, *Information School, University of Illinois at Urbana-Champaign*,
Who helped made this project happened, initialized the ideas, motivated me, and gave the preliminary approaches to the project.

Visanu Chulmonkol, *Department of Industrial Engineering, Penn State University,*
For helping with the feature extracting and pre-processing methods, cloud computing explanation and AWS account, together with fixing all the Sci-kit Learn and CNN models, and most important of all, make everything up and running.

Sukrit Sriratanawilai, *Computer Engineering, Kasetsart University, Thailand*,
For helping with the CNN model and in-depth explanation of a great paper, "An Effective and Scalable Framework for Authorship Attribution Query Processing."

Warunyou Dej-Udom, *Department of Computer Science, University of Illinois Urbana–Champaign*,
For helping with the ideas and concepts of how to approach and determine the "true" fake news data.

Also, thanks to many people I didn't mention but I've sought some help from.

References

Sarwar, Raheem, et al. "An Effective and Scalable Framework for Authorship Attribution Query Processing." ***IEEE Access*****, vol. 6, 2018, pp. 50030–50048., doi:10.1109/access.2018.2869198. https://ieeexplore.ieee.org/document/8457490**

Lilleberg, Joseph, Yun Zhu, and Yanqing Zhang. "Support vector machines and word2vec for text classification with semantic features." Cognitive Informatics & Cognitive Computing (ICCI* CC), 2015 IEEE 14th International Conference on. IEEE, 2015.

Code I used
https://mxnet.incubator.apache.org/tutorials/nlp/cnn.html

How CNN works on NLP
http://www.davidsbatista.net/blog/2018/03/31/SentenceClassificationConvNets/

Explanation on LSA
http://lsa.colorado.edu/whatis.html

SVM
http://blog.aylien.com/support-vector-machines-for-dummies-a-simple/
https://data-flair.training/blogs/applications-of-svm/

Word2vec
http://nadbordrozd.github.io/blog/2016/05/20/text-classification-with-word2vec/
https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/
https://radimrehurek.com/gensim/models/word2vec.html

ROC
https://en.wikipedia.org/wiki/Receiver_operating_characteristic

Split fold
https://data-flair.training/blogs/train-test-set-in-python-ml/
https://www.kaggle.com/ogrellier/kfold-or-stratifiedkfold

CNN
https://github.com/dennybritz/cnn-text-classification-tf
https://data-flair.training/blogs/convolutional-neural-networks-tutorial/
https://machinelearningmastery.com/how-to-develop-convolutional-neural-networks-for-multi-step-time-series-forecasting/
https://medium.com/@pushkarmandot/build-your-first-deep-learning-neural-network-model-using-keras-in-python-a90b5864116d
https://stackoverflow.com/questions/43876770/pandas-dataframe-and-keras
https://circleci.com/gh/bjherger/keras-pandas/64

Bayesian Network
https://stats.stackexchange.com/questions/139728/when-to-use-bayesian-networks-over-other-machine-learning-approaches
https://www.reddit.com/r/MachineLearning/comments/2vynmn/how_do_i_implement_a_bayesian_network/

Feature extraction
https://stats.stackexchange.com/questions/239704/analysis-of-dependent-variables
https://medium.com/mindorks/what-is-feature-engineering-for-machine-learning-d8ba3158d97a
https://www.kdnuggets.com/2017/11/amazing-predictive-power-conditional-probability-bayes-nets.html
https://medium.com/@karim_ouda/tutorial-document-classification-using-weka-aa98d5edb6fa

Fake news
https://github.com/docketrun/Detecting-Fake-News-with-Scikit-Learn
http://science.sciencemag.org/content/sci/359/6380/1146.full.pdf
https://www.whois.com/whois/infowars.com
https://www.datacamp.com/community/tutorials/scikit-learn-fake-news
https://github.com/aldengolab/fake-news-detection
https://github.com/KaiDMML/FakeNewsNet
https://mediabiasfactcheck.com/right/
https://github.com/clips/news-audit
https://towardsdatascience.com/i-trained-fake-news-detection-ai-with-95-accuracy-and-almost-went-crazy-d10589aa57c
https://theconversation.com/how-to-spot-fake-news-an-experts-guide-for-young-people-88887
https://github.com/sachinruk/deepschool.io
https://arxiv.org/pdf/1705.00648.pdf
https://www.extrawatch.com/traffic-flow
https://ahrefs.com/blog/website-traffic/

Appendix I. – Python code of the web crawler

[It’s really long. Double click to open.]

```
import requests
from bs4 import BeautifulSoup
import re
import pandas as pd
```

1. Crawl single site

In [23]:

```
def get_title(url):
    res = requests.get(url) #get the website, return request.Response obj
ect
    #print(res.status_code) #statu_code: return 200(found web), 404(not f
ound)
    soup = BeautifulSoup(res.text, 'html.parser')
    us_news_div = soup.find_all('div', re.compile('article-content'))

    title_list = []
    for i in range(len(us_news_div)):
        us_news_h3 = us_news_div[i].find_all('h3', recursive=False) #head
er
        us_news_a = us_news_h3[0].find_all('a', recursive=False) #anchor
tag
        for index, item in enumerate(us_news_a[:]):
            title = item.text.strip()
            title_list.append(title)
    return title_list
```

In [28]:

```
def get_url(title_list):
    title_name = ''
    url_list = []
    pattern = re.compile('([^\s\w]|_)+')

    for i in range(len(title_list)):
        title = re.sub("[‘’]", '', title_list[i])
        #print(title)
        strippedList = pattern.sub(' ', title)
        a = strippedList.split(" ")
        empty_string = ''
        if empty_string in a:
            a = [x for x in a if x != '']
            title_name = '-'.join(a)
        else:
            title_name = '-'.join(a)
```

Appendix II. – Python code of various classification

```
#!/usr/bin/env python
"""
Run k-NN classification on the Reuters text dataset using LSA.

This script leverages modules in scikit-learn for performing tf-idf and SVD.

Classification is performed using k-NN with k=5 (majority wins).

The script measures the accuracy of plain tf-idf as a baseline, then LSA to
show the improvement.
"""

import pickle
import time
import pandas as pd

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer
from sklearn.neighbors import KNeighborsClassifier

from sklearn.naive_bayes import BernoulliNB, MultinomialNB


###############################################################################
#  Load the raw text dataset.
###############################################################################

print("Loading dataset...")

data = pd.read_csv("fake_news.csv")
print(data.head())

print(data.columns.values)

X = data['text']
y = data['label']

# print(pd.value_counts(y))
# def give_rating(x):
#     if x >1:
#         return "1"
#     elif x <= 1:
#         return "0"
#
# y = y.apply(give_rating)

                            #
                            # index_Fake_News = [idx for idx,val in enumerate(y) if val == 'FAKE']
                            # index_True_News = [idx for idx,val in enumerate(y) if val == 'REAL']
                            # from random import shuffle
                            # shuffle(index_Fake_News)
                            # sh_index_Fake_News  = index_Fake_News[:]
                            #
                            # shuffle(index_True_News)
                            # sh_index_True_News  = index_True_News[:]
                            #
                            #
                            # # Apply those Shuffles index to "X" data and "y" data
                            #
                            # under_train_after_shuffle = sh_index_Fake_News+sh_index_True_News
                            # shuffle(under_train_after_shuffle)
                            #
                            # under_train_y = [y[i] for i in under_train_after_shuffle]
                            #
                            # under_train_X = [X[i] for i in under_train_after_shuffle]
                            # y = pd.Series(under_train_y)



X_train_raw = X[:5500]
y_train_labels = y[:5500]
```

```
X_test_raw = X[-500:].reset_index()['text']
y_test_labels = y[-500:].reset_index()['label']




print("print X_train_raw[0]")
print(X_train_raw[0])
print("print y_train_raw[0]")
print(y_train_labels[0])
print("print X_test_raw[0]")
print(X_test_raw[0])
print("print y_test_raw[0]")
print(y_test_labels[0])


y_train_labels.tolist()
y_train = [str(i) for i in y_train_labels]

y_test_labels.tolist()
y_test = [str(i) for i in y_test_labels]


###############################################################################
#  Use LSA to vectorize the articles.
###############################################################################

# Tfidf vectorizer:
#   - Strips out "stop words"
#   - Filters out terms that occur in more than half of the docs (max_df=0.5)
#   - Filters out terms that occur in only one document (min_df=2).
#   - Selects the 10,000 most frequently occuring words in the corpus.
#   - Normalizes the vector (L2 norm of 1.0) to normalize the effect of
#     document length on the tf-idf values.

from nltk.corpus import stopwords
stop_words_list = stopwords.words("english")
stop_words_list += ['.','/','<','>','``','"','-','--']

vectorizer = TfidfVectorizer(min_df = 10,max_df=0.5, ngram_range=(1,3)
                             , max_features=10000,
                             stop_words=stop_words_list,
                             use_idf=True)


# from sklearn.feature_extraction.text import TfidfVectorizer
# import pandas as pd
# texts = [
#     "good movie", "not a good movie", "did not like",
#     "i like it", "good one"
# ]
# # using default tokenizer in TfidfVectorizer
# vectorizer = TfidfVectorizer(min_df=2, max_df=0.5, ngram_range=(1, 2))
# features = vectorizer.fit_transform(X_train_raw)
# features_df = pd.DataFrame(
#     features.todense(),
#     columns=vectorizer.get_feature_names()
# )

#print(features_df.head())




# Build the tfidf vectorizer from the training data ("fit"), and apply it
# ("transform").
X_train_tfidf = vectorizer.fit_transform(X_train_raw)

print(type(X_train_tfidf))
print(X_train_tfidf)

print("show the shape of TFIDF")
print(X_train_tfidf.shape)

print("  Actual number of tfidf features: %d" % X_train_tfidf.get_shape()[1])

print("\nPerforming dimensionality reduction using LSA")
t0 = time.time()
```

```
# Project the tfidf vectors onto the first N principal components.
# Though this is significantly fewer features than the original tfidf vector,
# they are stronger features, and the accuracy is higher.
svd = TruncatedSVD(1500)
lsa = make_pipeline(svd, Normalizer(copy=False))

# Run SVD on the training data, then project the training data.
X_train_lsa = lsa.fit_transform(X_train_tfidf)
print("show one element")
print(X_train_lsa[0][0])
print("show X_train_lsa matrix")
print(X_train_lsa)


print("show the shape of X_train_lsa")
print(X_train_lsa.shape)

print("  done in %.3fsec" % (time.time() - t0))

explained_variance = svd.explained_variance_ratio_.sum()
print("  Explained variance of the SVD step: {}%".format(int(explained_variance * 100)))


# Now apply the transformations to the test data as well.
X_test_tfidf = vectorizer.transform(X_test_raw)
X_test_lsa = lsa.transform(X_test_tfidf)


###############################################################################
#  Run classification of the test articles
###############################################################################

print("\nClassifying tfidf vectors...")

# Time this step.
t0 = time.time()

print(X_train_tfidf.shape)
print(len(y_train))

# Build a k-NN classifier. Use k = 5 (majority wins), the cosine distance,
# and brute-force calculation of distances.
model = KNeighborsClassifier(n_neighbors=5, algorithm='brute', metric='cosine')
model.fit(X_train_tfidf, y_train)

print("see the x_train TFIDF matrix")
print(X_train_tfidf)
print("see the dimension of the x_train TFIDF matrix")
print(X_train_tfidf.shape)

# Classify the test vectors.
p = model.predict(X_test_tfidf)

# Measure accuracy
numRight = 0;
for i in range(0,len(p)):
    if p[i] == y_test[i]:
        numRight += 1

print("  (%d / %d) correct - %.2f%%" % (numRight, len(y_test), float(numRight) / float(len(y_test)) *
100.0))

# Calculate the elapsed time (in seconds)
elapsed = (time.time() - t0)
print("  done in %.3fsec" % elapsed)


print("\nClassifying LSA vectors...")

# Time this step.
t0 = time.time()

# Build a k-NN classifier. Use k = 5 (majority wins), the cosine distance,
# and brute-force calculation of distances.
model_lsa = KNeighborsClassifier(n_neighbors=5, algorithm='brute', metric='cosine')
model_lsa.fit(X_train_lsa, y_train)
```

```
# print x_train LSA
print("X_train_lsa")
print(X_train_lsa)
# print y_train LSA
print("y_train LSA")
print(y_train)


# Classify the test vectors.
p = model_lsa.predict(X_test_lsa)

#print(p)

# Measure accuracy
numRight = 0;
for i in range(0,len(p)):
    #if p[i] != y_test[i]:
        #print(p[i])
    if p[i] == y_test[i]:
        #print(p[i])
        numRight += 1

print("  (%d / %d) correct - %.2f%%" % (numRight, len(y_test), float(numRight) / float(len(y_test)) *
100.0))

# Calculate the elapsed time (in seconds)
elapsed = (time.time() - t0)
print("    done in %.3fsec" % elapsed)
```

Appendix III. – Python code CNN model

```
In [1]: from __future__ import print_function

        from collections import Counter
        import itertools
        import numpy as np
        import re
        import csv

        try:
            # For Python 3.0 and later
            from urllib.request import urlopen
        except ImportError:
            # Fall back to Python 2's urllib2
            from urllib2 import urlopen

        def clean_str(string):
            """
            Tokenization/string cleaning.
            Original from https://github.com/yoonkim/CNN_sentence/blob/ma
        ster/process_data.py
            """
            string = re.sub(r"[^A-Za-z0-9(),!?\'\`]", " ", string)
            string = re.sub(r"\'s", " \'s", string)
            string = re.sub(r"\'ve", " \'ve", string)
            string = re.sub(r"n\'t", " n\'t", string)
            string = re.sub(r"\'re", " \'re", string)
            string = re.sub(r"\'d", " \'d", string)
            string = re.sub(r"\'ll", " \'ll", string)
            string = re.sub(r",", " , ", string)
            string = re.sub(r"!", " ! ", string)
            string = re.sub(r"\(", " \( ", string)
            string = re.sub(r"\)", " \) ", string)
            string = re.sub(r"\?", " \? ", string)
            string = re.sub(r"\s{2,}", " ", string)

            return string.strip().lower()

        def download_sentences(url):
            """
```

```
    Download sentences from specified URL.

    Strip trailing newline, convert to Unicode.
    """

    remote_file = urlopen(url)
    return [line.decode('Latin1').strip() for line in remote_file
.readlines()]

def load_data_and_labels():
    """
    Loads polarity data from files, splits the data into words an
d generates labels.
    Returns split sentences and labels.
    """

    positive_examples = download_sentences('https://raw.githubuse
rcontent.com/yoonkim/CNN_sentence/master/rt-polarity.pos')
    negative_examples = download_sentences('https://raw.githubuse
rcontent.com/yoonkim/CNN_sentence/master/rt-polarity.neg')


    with open('only_FAKE.csv', 'r') as f:
      reader = csv.reader(f)
      positive_examples = list(reader)

    positive_examples = [item for sublist in positive_examples fo
r item in sublist]

    with open('only_REAL.csv', 'r') as f:
      reader = csv.reader(f)
      negative_examples = list(reader)

    negative_examples = [item for sublist in negative_examples fo
r item in sublist]

    # Tokenize
    x_text = positive_examples + negative_examples
    x_text = [clean_str(sent).split(" ") for sent in x_text]

    # Generate labels
    positive_labels = [1 for _ in positive_examples]
```

```
    negative_labels = [0 for _ in negative_examples]
    y = np.concatenate([positive_labels, negative_labels], 0)
    return x_text, y


def pad_sentences(sentences, padding_word=""):
    """
    Pads all sentences to be the length of the longest sentence.
    Returns padded sentences.
    """
    sequence_length = max(len(x) for x in sentences)
    padded_sentences = []
    for i in range(len(sentences)):
        sentence = sentences[i]
        num_padding = sequence_length - len(sentence)
        new_sentence = sentence + [padding_word] * num_padding
        padded_sentences.append(new_sentence)

    return padded_sentences


def build_vocab(sentences):
    """
    Builds a vocabulary mapping from token to index based on the
sentences.
    Returns vocabulary mapping and inverse vocabulary mapping.
    """
    # Build vocabulary
    word_counts = Counter(itertools.chain(*sentences))

    # Mapping from index to word
    vocabulary_inv = [x[0] for x in word_counts.most_common()]

    # Mapping from word to index
    vocabulary = {x: i for i, x in enumerate(vocabulary_inv)}

    return vocabulary, vocabulary_inv


def build_input_data(sentences, labels, vocabulary):
    """
    Maps sentences and labels to vectors based on a vocabulary.
```

```
    """
    x = np.array([
            [vocabulary[word] for word in sentence]
            for sentence in sentences])
    y = np.array(labels)

    return x, y

"""
Loads and preprocesses data for the MR dataset.
Returns input vectors, labels, vocabulary, and inverse vocabulary
.
"""
# Load and preprocess data
sentences, labels = load_data_and_labels()
sentences_padded = pad_sentences(sentences)
vocabulary, vocabulary_inv = build_vocab(sentences_padded)
x, y = build_input_data(sentences_padded, labels, vocabulary)

vocab_size = len(vocabulary)

# randomly shuffle data
np.random.seed(10)
shuffle_indices = np.random.permutation(np.arange(len(y)))
x_shuffled = x[shuffle_indices]
y_shuffled = y[shuffle_indices]

# split train/dev set
# there are a total of 10662 labeled examples to train on
x_train, x_dev = x_shuffled[:-1000], x_shuffled[-1000:]
y_train, y_dev = y_shuffled[:-1000], y_shuffled[-1000:]

sentence_size = x_train.shape[1]

print('Train/Dev split: %d/%d' % (len(y_train), len(y_dev)))
print('train shape:', x_train.shape)
print('dev shape:', x_dev.shape)
print('vocab_size', vocab_size)
print('sentence max words', sentence_size)
Train/Dev split: 5302/1000
train shape: (5302, 6007)
dev shape: (1000, 6007)
```

```
vocab_size 75757
sentence max words 6007
```

In [2]:
```
import mxnet as mx
import sys,os

'''
Define batch size and the place holders for network inputs and ou
tputs
'''

batch_size = 50
print('batch size', batch_size)

input_x = mx.sym.Variable('data') # placeholder for input data
input_y = mx.sym.Variable('softmax_label') # placeholder for outp
ut label


'''
Define the first network layer (embedding)
'''

# create embedding layer to learn representation of words in a lo
wer dimensional subspace (much like word2vec)
num_embed = 300 # dimensions to embed words into
print('embedding dimensions', num_embed)

embed_layer = mx.sym.Embedding(data=input_x, input_dim=vocab_size
, output_dim=num_embed, name='vocab_embed')

# reshape embedded data for next layer
conv_input = mx.sym.Reshape(data=embed_layer, shape=(batch_size,
1, sentence_size, num_embed))
```

```
batch size 50
embedding dimensions 300
```

In [3]:
```
# create convolution + (max) pooling layer for each filter operat
ion
filter_list=[3, 4, 5] # the size of filters to use
print('convolution filters', filter_list)
```

```
num_filter=100
pooled_outputs = []
for filter_size in filter_list:
    convi = mx.sym.Convolution(data=conv_input, kernel=(filter_si
ze, num_embed), num_filter=num_filter)
    relui = mx.sym.Activation(data=convi, act_type='relu')
    pooli = mx.sym.Pooling(data=relui, pool_type='max', kernel=(s
entence_size - filter_size + 1, 1), stride=(1, 1))
    pooled_outputs.append(pooli)

# combine all pooled outputs
total_filters = num_filter * len(filter_list)
concat = mx.sym.Concat(*pooled_outputs, dim=1)

# reshape for next layer
h_pool = mx.sym.Reshape(data=concat, shape=(batch_size, total_fil
ters))
```

```
convolution filters [3, 4, 5]
```

In [4]:

```
# dropout layer
dropout = 0.5
print('dropout probability', dropout)

if dropout > 0.0:
    h_drop = mx.sym.Dropout(data=h_pool, p=dropout)
else:
    h_drop = h_pool
```

```
dropout probability 0.5
```

In [7]:

```
# fully connected layer
num_label = 2

cls_weight = mx.sym.Variable('cls_weight')
cls_bias = mx.sym.Variable('cls_bias')

fc = mx.sym.FullyConnected(data=h_drop, weight=cls_weight, bias=c
ls_bias, num_hidden=num_label)

# softmax output
```

```python
sm = mx.sym.SoftmaxOutput(data=fc, label=input_y, name='softmax')

# set CNN pointer to the "back" of the network
cnn = sm
```

In [9]:

```python
from collections import namedtuple
import math
import time

# Define the structure of our CNN Model (as a named tuple)
CNNModel = namedtuple("CNNModel", ['cnn_exec', 'symbol', 'data',
'label', 'param_blocks'])

# Define what device to train/test on, use GPU if available
ctx = mx.gpu() if mx.test_utils.list_gpus() else mx.cpu()

arg_names = cnn.list_arguments()

input_shapes = {}
input_shapes['data'] = (batch_size, sentence_size)

arg_shape, out_shape, aux_shape = cnn.infer_shape(**input_shapes)
arg_arrays = [mx.nd.zeros(s, ctx) for s in arg_shape]
args_grad = {}
for shape, name in zip(arg_shape, arg_names):
    if name in ['softmax_label', 'data']: # input, output
        continue
    args_grad[name] = mx.nd.zeros(shape, ctx)

cnn_exec = cnn.bind(ctx=ctx, args=arg_arrays, args_grad=args_grad
, grad_req='add')

param_blocks = []
arg_dict = dict(zip(arg_names, cnn_exec.arg_arrays))
initializer = mx.initializer.Uniform(0.1)
for i, name in enumerate(arg_names):
    if name in ['softmax_label', 'data']: # input, output
        continue
    initializer(mx.init.InitDesc(name), arg_dict[name])
```

```
            param_blocks.append( (i, arg_dict[name], args_grad[name], nam
        e) )

        data = cnn_exec.arg_dict['data']
        label = cnn_exec.arg_dict['softmax_label']

        cnn_model= CNNModel(cnn_exec=cnn_exec, symbol=cnn, data=data, lab
        el=label, param_blocks=param_blocks)
```

In [10]:

```
'''
Train the cnn_model using back prop
'''

optimizer = 'rmsprop'
max_grad_norm = 5.0
learning_rate = 0.0005
epoch = 50

print('optimizer', optimizer)
print('maximum gradient', max_grad_norm)
print('learning rate (step size)', learning_rate)
print('epochs to train for', epoch)

# create optimizer
opt = mx.optimizer.create(optimizer)
opt.lr = learning_rate

updater = mx.optimizer.get_updater(opt)

# For each training epoch
for iteration in range(epoch):
    tic = time.time()
    num_correct = 0
    num_total = 0

    # Over each batch of training data
    for begin in range(0, x_train.shape[0], batch_size):
        batchX = x_train[begin:begin+batch_size]
        batchY = y_train[begin:begin+batch_size]
        if batchX.shape[0] != batch_size:
            continue
```

```
        cnn_model.data[:] = batchX
        cnn_model.label[:] = batchY

        # forward
        cnn_model.cnn_exec.forward(is_train=True)

        # backward
        cnn_model.cnn_exec.backward()

        # eval on training data
        num_correct += sum(batchY == np.argmax(cnn_model.cnn_exec
.outputs[0].asnumpy(), axis=1))
        num_total += len(batchY)

        # update weights
        norm = 0
        for idx, weight, grad, name in cnn_model.param_blocks:
            grad /= batch_size
            l2_norm = mx.nd.norm(grad).asscalar()
            norm += l2_norm * l2_norm

        norm = math.sqrt(norm)
        for idx, weight, grad, name in cnn_model.param_blocks:
            if norm > max_grad_norm:
                grad *= (max_grad_norm / norm)

            updater(idx, grad, weight)

            # reset gradient to zero
            grad[:] = 0.0

    # Decay learning rate for this epoch to ensure we are not "ov
ershooting" optima
    if iteration % 50 == 0 and iteration > 0:
        opt.lr *= 0.5
        print('reset learning rate to %g' % opt.lr)

    # End of training loop for this epoch
    toc = time.time()
    train_time = toc - tic
    train_acc = num_correct * 100 / float(num_total)
```

```
    # Saving checkpoint to disk
    if (iteration + 1) % 10 == 0:
        prefix = 'cnn'
        cnn_model.symbol.save('./%s-symbol.json' % prefix)
        save_dict = {('arg:%s' % k) : v  for k, v in cnn_model.cn
n_exec.arg_dict.items()}
        save_dict.update({('aux:%s' % k) : v for k, v in cnn_mode
l.cnn_exec.aux_dict.items()})
        param_name = './%s-%04d.params' % (prefix, iteration)
        mx.nd.save(param_name, save_dict)
        print('Saved checkpoint to %s' % param_name)


    # Evaluate model after this epoch on dev (test) set
    num_correct = 0
    num_total = 0

    # For each test batch
    for begin in range(0, x_dev.shape[0], batch_size):
        batchX = x_dev[begin:begin+batch_size]
        batchY = y_dev[begin:begin+batch_size]

        if batchX.shape[0] != batch_size:
            continue

        cnn_model.data[:] = batchX
        cnn_model.cnn_exec.forward(is_train=False)

        num_correct += sum(batchY == np.argmax(cnn_model.cnn_exec
.outputs[0].asnumpy(), axis=1))
        num_total += len(batchY)

    dev_acc = num_correct * 100 / float(num_total)
    print('Iter [%d] Train: Time: %.3fs, Training Accuracy: %.3f
\
            --- Dev Accuracy thus far: %.3f' % (iteration, train_
time, train_acc, dev_acc))
```

optimizer rmsprop
maximum gradient 5.0
learning rate (step size) 0.0005
epochs to train for 50

```
Iter [0] Train: Time: 107.560s, Training Accuracy: 68.094
--- Dev Accuracy thus far: 88.000
Iter [1] Train: Time: 100.771s, Training Accuracy: 86.132
--- Dev Accuracy thus far: 90.500
Iter [2] Train: Time: 101.234s, Training Accuracy: 90.472
--- Dev Accuracy thus far: 92.300
Iter [3] Train: Time: 101.387s, Training Accuracy: 93.245
--- Dev Accuracy thus far: 93.700
Iter [4] Train: Time: 101.461s, Training Accuracy: 95.038
--- Dev Accuracy thus far: 94.100
Iter [5] Train: Time: 101.830s, Training Accuracy: 96.434
--- Dev Accuracy thus far: 93.700
Iter [6] Train: Time: 101.480s, Training Accuracy: 97.019
--- Dev Accuracy thus far: 94.500
Iter [7] Train: Time: 101.441s, Training Accuracy: 97.830
--- Dev Accuracy thus far: 94.600
Iter [8] Train: Time: 101.483s, Training Accuracy: 98.245
--- Dev Accuracy thus far: 95.000
Saved checkpoint to ./cnn-0009.params
Iter [9] Train: Time: 101.465s, Training Accuracy: 99.038
--- Dev Accuracy thus far: 95.800
Iter [10] Train: Time: 101.407s, Training Accuracy: 99.245
--- Dev Accuracy thus far: 95.200
Iter [11] Train: Time: 101.525s, Training Accuracy: 99.321
--- Dev Accuracy thus far: 95.300
Iter [12] Train: Time: 101.452s, Training Accuracy: 99.509
--- Dev Accuracy thus far: 95.500
Iter [13] Train: Time: 101.455s, Training Accuracy: 99.585
--- Dev Accuracy thus far: 95.700
Iter [14] Train: Time: 101.514s, Training Accuracy: 99.698
--- Dev Accuracy thus far: 95.300
Iter [15] Train: Time: 101.462s, Training Accuracy: 99.679
--- Dev Accuracy thus far: 96.200
Iter [16] Train: Time: 101.517s, Training Accuracy: 99.792
--- Dev Accuracy thus far: 96.400
Iter [17] Train: Time: 101.396s, Training Accuracy: 99.868
--- Dev Accuracy thus far: 96.200
Iter [18] Train: Time: 101.580s, Training Accuracy: 99.925
--- Dev Accuracy thus far: 96.500
Saved checkpoint to ./cnn-0019.params
Iter [19] Train: Time: 101.555s, Training Accuracy: 99.849
--- Dev Accuracy thus far: 96.000
```

```
Iter [20] Train: Time: 101.463s, Training Accuracy: 99.830
--- Dev Accuracy thus far: 96.100
Iter [21] Train: Time: 101.424s, Training Accuracy: 99.868
--- Dev Accuracy thus far: 96.400
Iter [22] Train: Time: 101.570s, Training Accuracy: 99.906
--- Dev Accuracy thus far: 96.100
Iter [23] Train: Time: 101.477s, Training Accuracy: 99.925
--- Dev Accuracy thus far: 95.800
Iter [24] Train: Time: 101.476s, Training Accuracy: 99.906
--- Dev Accuracy thus far: 95.900
Iter [25] Train: Time: 101.442s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 96.200
Iter [26] Train: Time: 101.567s, Training Accuracy: 99.943
--- Dev Accuracy thus far: 96.000
Iter [27] Train: Time: 101.566s, Training Accuracy: 99.943
--- Dev Accuracy thus far: 96.100
Iter [28] Train: Time: 101.646s, Training Accuracy: 99.868
--- Dev Accuracy thus far: 95.700
Saved checkpoint to ./cnn-0029.params
Iter [29] Train: Time: 101.586s, Training Accuracy: 99.925
--- Dev Accuracy thus far: 96.300
Iter [30] Train: Time: 101.552s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 95.500
Iter [31] Train: Time: 101.630s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 96.600
Iter [32] Train: Time: 101.655s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 95.700
Iter [33] Train: Time: 101.613s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 96.000
Iter [34] Train: Time: 101.588s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 96.100
Iter [35] Train: Time: 101.646s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 95.800
Iter [36] Train: Time: 101.663s, Training Accuracy: 100.000
--- Dev Accuracy thus far: 96.300
Iter [37] Train: Time: 101.664s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 95.300
Iter [38] Train: Time: 101.669s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 96.400
Saved checkpoint to ./cnn-0039.params
Iter [39] Train: Time: 101.565s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 95.800
```

```
Iter [40] Train: Time: 101.445s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 96.300
Iter [41] Train: Time: 101.526s, Training Accuracy: 100.000
--- Dev Accuracy thus far: 95.900
Iter [42] Train: Time: 101.487s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 95.500
Iter [43] Train: Time: 101.549s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 96.000
Iter [44] Train: Time: 101.567s, Training Accuracy: 99.962
--- Dev Accuracy thus far: 96.600
Iter [45] Train: Time: 101.657s, Training Accuracy: 100.000
--- Dev Accuracy thus far: 95.500
Iter [46] Train: Time: 101.609s, Training Accuracy: 100.000
--- Dev Accuracy thus far: 96.500
Iter [47] Train: Time: 101.624s, Training Accuracy: 100.000
--- Dev Accuracy thus far: 96.600
Iter [48] Train: Time: 101.659s, Training Accuracy: 99.981
--- Dev Accuracy thus far: 95.900
Saved checkpoint to ./cnn-0049.params
Iter [49] Train: Time: 101.641s, Training Accuracy: 99.943
--- Dev Accuracy thus far: 96.700
```