Final Project

INTL/QMBU450/550:

Advanced Data Analysis in Python

Abstract

Commercially available and open-source spam filtering solutions use Naive Bayes Binary Classifiers. The following work explores an alternative, aiming to build a spam filter classifier using a deep-learning approach by using logistic regression analysis with TensorFlow.

Contents

Introduction

Writing, checking and reading emails is now a daily activity in most people's lives (Lagrana 2016: pp. 14-16). Reaching out to people over email for different kinds of reasons is now a standard strategy for marketers and sales persons among others. Often though, receiving unsolicited emails with at best dubious and at worst downright malicious content that seeks to extract personal and financial information ranges from annoying to dangerous, and fills up mostly limited Inbox storage space (Fallows 2007).

While the first solution to this problem was simply maintaining a list of IP addresses that sent out spam email and not accepting email from them (an approach known as Real-Time Blackhole List, or RBL), as the number of email sending computers exploded a content-based solution was needed. The first popular generic spam filtering software, called SpamAssassin, was uploaded to SourceForge, a then-popular open source code sharing website, in April 2001.

Although machine learning was not a buzzword at that time, spam filtering systems were one of the first applications of machine learning to consumer computing technology, as these systems learned to recognise what spam looked like from examples of such emails, with the aim of correctly classifying previously unseen email.

Industry standard spam filtering systems still use the technique developed by SpamAssassin to this day. By training the filter with a large corpus of emails, the following values become known:

- Probability that any given email is spam, denoted as $P(spam)$
- Probability that a certain word appears in any email, denoted as $P(word)$
- Probability that a certain word appears in an email that is spam, denoted as $P(word|spam)$

Given this data, the probability that a certain email is spam given that a certain word appears in the email, denoted as $P(spam|word)$ can be found out by a simple application of the Bayes Theorem:

$$P(spam|word) = \frac{P(word|spam)P(spam)}{P(word)}$$

This is computationally very inexpensive to obtain, and additionally with every new email that is classified, the system incrementally learns by updating the three probabilities on the right side of the equation. (Cornell University Blog 2018, Sahami et al. 1998)

Spam Filtering: A Binary Classification Problem

Machine Learning is usually used to solve two kinds of problems: *regression*, which tries to predict the future value of a continuous variable using previous values, and *classification,* which tries to classify items into categories based on *features*, or predetermined characteristics (Silva et al. 2016: pp.1).

Spam filtering is a classic classification problem, because we're trying to *classify* a message as either spam or ham (not spam), based on certain *features* of the email, such as what words and phrases it contains, whether it contains images and links, and so on and so forth.

There are two groups of classification problems - binary classification and multi-class classification. Binary classification tries to classify an item into one of two categories, while multi-class classification tries to classify items into more than two categories. As our problem deals with classifying an email into either spam or ham, which are exactly two categories, any solution that works for binary classification problems will work for our purpose.

Moreover, machine learning techniques can be further classified into two kinds of techniques: supervised learning and unsupervised learning. Supervised learning aims to teach a neural network to classify items based on a number of features by showing the network a lot of example items and telling it which item falls into which category, thus letting the system learn which combination of features conforms to which category. Unsupervised learning, on the other hand, does not teach the network anything, instead feeding the network a lot of unstructured data and letting it figure out by itself which items are similar and can be grouped into a category by itself (Silva et al. 2016: pp.71-80).

Both supervised and unsupervised learning will produce valid models which can predict whether an email is spam or not, but a binary classification model trained using supervised learning makes the most intuitive sense for this problem.

Logistic Regression

The term logistic regression is a misnomer, because logistic regressions are not used for regression analysis, but rather for classification problems. Logistic regressions are helpful when the output or the independent variable to be analysed is nominally scaled. There are three different types of logistic regressions: the binary logistic regression (two categories), the multinomial logistic regression (three or more categories without ordering) and ordinal logistic regression (three or more categories with ordering) (Kopp et al. 2012: pp. 159-181). As mentioned in the paragraph before, we want our model to be able to simply classify emails as either spam or ham. So, our output only has two categories and we use a binary logistic regression model.

The following form is being calculated:

$$y = \varsigma(\omega_1 x_1 + \omega_2 x_2 + \ldots + \omega_n x_n)$$

where x is the input parameter, ω is the weight given to the parameter, and n is the number of input parameters.

The summation of the weighted features need to be passed through a logistic function in order to convert them to a probability - a value between 0 and 1 - and in our case, we use ς to denote the Sigmoid function. This makes it possible to separate the input in two groups, with the value 0.5 demarcating the line of separation (Manduk 2019). In our case, values greater than or equal to 0.5 will classify the incoming email as spam, while values below 0.5 will classify the email as ham.

Introduction to TensorFlow

TensorFlow is an open-source Python library created by Google that implements dataflow programming (hence the Flow in the name), with the data itself being represented as tensors (implemented as multidimensional arrays, hence the name Tensor). As it turns out, dataflow

programming with data being represented as tensors lends itself really well to machine learning applications, which is why TensorFlow has seen an immense amount of uptake in the field of machine learning.

TensorFlow can be used to build very complex computation graphs in a simple declarative manner, with higher level libraries such as Keras making it easy to build such graphs that model neural networks, repeatedly executing these graphs to train the neural network. Additionally, TensorFlow can take advantage of advanced vector mathematics capabilities, including making use of vector instructions in the processor, or utilising the GPU via CUDA or OpenCL to accelerate mathematical operations, without the programmer having to explicitly tailor their code to vectorise further.

Code Structure

The theory behind using neural network based approaches to machine learning is simple. Using a technique known as supervised learning, we show a computer many examples of items while simultaneously telling it what that item is. The artificial neural network "gets an idea" of what constitutes a certain item, so that when it sees an example of an item it has never seen before, it can correctly categorize that example.

Data Structures

To get our neural network to make sense of our data (emails), we need to convert each email into a set of *features*, so that the neural network can assign a certain *weight* to each feature.

Now what to use as a feature is the work of a trained data scientist to figure out. A simple feature could just be whether a certain word exists or not, and complicated models could operate on anything from given word combinations, distances between certain words, whether or not an email contains attachments, whether those attachments are encrypted and so on and so forth.

For this project, we are going to use a very simple model - our features are simply going to be "how many times a given word occurs in an email".

To train the model, we need to produce two matrices. The first is called the *feature matrix*, and this contains a matrix of which email contains how many occurrences of which words. It looks like this:

	dog	**bank**	**money**	...
Email 1	0	2	6	...
Email 2	3	2	0	...
Email 3	4	0	2	...
...	...	...	...	...

Table: Feature Matrix

The second matrix we need to produce is called the *label matrix*, and this simply tells the neural network whether a given email is ham or spam. This matrix looks like this:

	Ham	**Spam**
Email 1	1	0
Email 2	0	1
Email 3	0	1
...	...	...

Table: Label Matrix

The label matrix is simple - each column can only contain either a 0 or 1, and both of the columns cannot be the same value (after all, the email is either ham or spam).

An important point to note is that as presented here, the tables contain column and row labels, but when fed to the neural network, the first row and first column will be removed, meaning that the table will only contain our word counts or classification confidence.

It is also important to remember two terms: *feature vector* and *label vector*. These are simply one row in the feature matrix and label matrix respectively. These terms will come in useful when we describe how to test unknown emails - as we shall have to convert them into feature vectors, and after prediction by the model we will get a label vector back.

Data Preparation

The code for preparing our data is in a file called `ProcessData.py`, located in a class called `DataProcessor`.

The most important function is called `processSingleMail`. This function turns a single mail into a dictionary of word counts. For example, given a text:

```
My baked cake is better than your bought cake
```

It will return a dictionary in the form:

```
{
	"cake": 2,
	"better": 1,
	"baked": 1,
	"bought: 1
}
```

You will notice that some of the words, namely "my", "is", "than" and "your" are missing. This is because these are known as "stopwords" - common words that exist in pretty much all sentences and thus doesn't add any identifying information about whether an email is ham or spam. The `processSingleMail` function has 4 steps:

1. Split the email into words. We use NLTK, the Natural Language Tool Kit, a Python library for working with natural language text to do this conversion, also called tokenization.
2. Normalise all words by turning them into lowercase
3. Remove all punctuation and stopwords
4. Count the number of occurrences of every unique word and return our dictionary

The next important function is `buildFeatureVector`. Once we have all emails in our corpus analysed and stored in our ham and spam lists, we need to figure out how many unique words there are in all the vectors. The function `buildFeatureVector` does just that by looping over every dictionary in both the ham and spam lists and adding their keys (which are our words) to a set. It then converts the set to a list, and builds a secondary data structure that maps every word to its index in this list, for faster lookups.

The third important function is `buildMailVector`. This function looks at every email (in our pre-processed dictionary form), builds an array that is the size of our feature vector and is initially

filled with zeros, and then goes over every word in the email and fills up the corresponding position in the feature vector with the word count.

When we are building the mail vector of one of the emails we used in the training sets, all the words in the email are guaranteed have an index in the feature vector, because we've seen all those words before when we were building the feature vector (in the `buildFeatureVector` function). However, when we are testing arbitrary emails, all the words in the email may not have an index in our feature vector. Therefore, we have a try-catch block to ignore errors if a word cannot be found in our feature vector index map.

The last pieces of this puzzle are the functions `saveFeatureVector` and `loadFeatureVector`. When we are testing arbitrary emails, we need to convert them to a mail vector, and of course it is essential that the vector be of the same shape as the ones we used in training, and that the words have the same indexes. So `saveFeatureVector` simply saves the contents of the unique word array in order to a text file, and `loadFeatureVector` loads this text file and reconstructs the index map.

With this, we are now ready to produce training datasets. The code expects training data to be arranged like so:

```
---+ trainingData/
   +--+ ham/
   |  +--- mail1.txt
   |  +--- mail2.txt
   |  +--- mail3.txt
   |  +--- ...
   +--+ spam/
      +--- mail1.txt
      +--- mail2.txt
      +--- mail3.txt
      +--- ...
```

Now we can simply open up a Python console and type in:

```
from ProcessData import DataProcessor

oDP = DataProcessor()
oDP.processAllMails("trainingData") # the directory with the ham and spam folders
oDP.buildFeatureVector()
oDP.saveFeatureVector("features.txt") # file to save the feature vector to
oDP.buildFeatureMatrix("spam.csv", "ham.csv") # files to save our data in
```

And this will produce 3 files - features.txt, spam.csv and ham.csv - the first of which is required to test arbitrary emails later, and the latter two csv files are required to train our neural network.

Neural Network

The code for the Neural Network itself is located in a file called `Regression.py`, inside the `LogisticRegressionPredictor` class.

TensorFlow 2.0 makes it incredibly easy to specify a neural network model, as it includes a library called Keras, which is a higher level abstraction that allows us to declaratively define a model by specifying its layers, the nature of connections between the nodes within each layer (and between layers), and the activation function. It completely abstracts away all our housekeeping, including the number of nodes in the layer, and even maintaining our bias vector.

The model is defined in just one line of code (split into multiple lines for code readability purposes). The line is:

```
oModel = tf.keras.Sequential([
      tf.keras.layers.Dense(iNumLabels, input_shape = (iNumFeatures,),
                            activation = "sigmoid")
])
```

As is evident by the code, we create a model with just a single layer, with data flowing sequentially through each layer. The single layer consists of a densely connected set of nodes, which has the output dimensionality of the number of labels (in our case 2), the input dimensionality of however many words our feature matrix contains, and the sigmoid activation function.

The API documentation for Tensorflow gives us the following explanation for this layer: *`Dense` implements the operation: `output = activation(dot(input, kernel) + bias)` where `activation` is the element-wise activation function passed as the `activation` argument, `kernel` is a weights matrix created by the layer, and `bias` is a bias vector created by the layer*. Dense by default uses a vector of biases initialised to 0.

This is exactly what we need, as dictated by the equation for Logistic Regression. Our activation function of choice is the Sigmoid function, which is dictated by its primary purpose (mapping predicted values to probabilities) and the following advantages:

1. Logistic regression is really a classification approach, and the Sigmoid function works well for classification problems.
2. Sigmoid produces a value normalised between [0, 1], which is what we need.

At this point, we need to compile and train our model. This is achieved by the following two lines of code:

```
oModel.compile(optimizer = "sgd", loss = "mean_squared_error",
               metrics = ["accuracy"])
oModel.fit(x = aFeatures, y = aLabels, shuffle = True, epochs = 27000,
           batch_size = 10)
```

The choice of the optimiser and learning rate is obtained by trial-and-error, and their efficacy not only differs between models but also datasets, and thus there is no right or wrong choice for which ones to use. The choice of a Stochastic Gradient Descent optimiser was because it was "good enough" with both a very small and a medium sized dataset, and the mean squared error loss function choice was made based on it being mathematically the most compatible.

The model is fitted using our feature matrix as the X variable and our label matrix as the Y variable. The *epochs* is the number of iterations to train, and the value 27,000 is definitely really large for the small synthetic dataset provided with the source code; however with slightly larger

datasets (random samplings of the Enron Spam dataset) this number was where we started seeing diminishing returns for our accuracy, hitting about 96.5%.

The loadData function in Regression.py is an important function. It simply loads the ham.csv and spam.csv files into memory, and splits them into two chunks: it keeps 75% of the data for training purposes, and 25% for evaluation.

To train our model, we can open up a Python console and type the following:

```
from Regression import LogisticRegressionPredictor

oLRP = LogisticRegressionPredictor()
oLRP.loadData()
oLRP.trainModel("model.hd5") # save model into this folder after training
```

Predicting Arbitrary Emails

Predicting the category of an arbitrary email is a 2-step process, and it's fast and easy compared to training the model.

In the first step, we need to convert the email text into a mail feature vector:

```
from ProcessData import DataProcessor

sMail = """Subject: COVID-19 Donation

Dear Sir,
You have been selected to receive the sum of $ 5,000,000.00 due to COVID-19.
Please send us your bank details and a copy of your passport to facilitate the
funds transfer."""

oDP = DataProcessor()
oDP.loadFeatureVector("features.txt") # we need the index of words
aMailVector = oDP.buildAdhocMailVector(sMail)
```

`aMailVector` now contains a single email as a feature vector, contained in a NumPy `ndarray`. Tensorflow actually expects inputs to also be a matrix; as a result `aMailVector` is actually a 2D array but with only 1 row.

The second step is to actually use our model to predict the category of this email:

```
from Regression import LogisticRegressionPredictor

oLRP = LogisticRegressionPredictor()
oLRP.predictMail("model.hd5", aMailVector)
```

We will get back a label vector in an `ndarray`, and the first element will be the predicted spamminess of the email while the second element will be the predicted hamminess.

Synthetic Test Results

As a synthetic test, the model was trained using a collection of 5 spam emails and 5 ham emails. All the sample emails are available as part of the source code, in the synthetic directory.

The test machine was a 2020 MacBook Air, with the following specifications:

- 10th generation dual-core Intel Core i3, 1.1 GHz base clock, 3.2 GHz boost clock
- 8 GB 3733MHz LPDDR4 memory
- Intel Iris Plus Graphics, no GPGPU acceleration available

In addition, a generic binary build of TensorFlow was used, which does not optimise for the use of AVX2 and AVX512 vector instructions which are available on the 10th generation Intel processors.

The ham emails came from publicly available mailing lists of the KDE Open Source Project, which can be publicly accessed at https://mail.kde.org/mailman/listinfo/. The spam emails were harvested from the author's personal email.

Model Training

The preprocessing of data into the CSV files and the feature vector index text file took an average of 2.97 seconds over 5 runs. NLTK data was pre-downloaded and hence their download did not impact running times.

Training the model with just 8 emails (with 2 saved for evaluation later) and 27,000 epochs took an average of 1m 04.54s over 5 runs. The accuracy and loss data is not given here as they are meaningless training over just 8 emails 27,000 times.

Prediction

To test whether this minimally trained model is any good at providing valid predictions, one each of spam and ham email were used to see if any usable predictions could be obtained. The spam email used to test was:

```
Subject: Donation Information

Hi, My name is Mr. Charles Koch, an elder brother to late
Mr. David Hamilton Koch a philanthropist and the founder
of Koch Industries, one of the largest private foundations
in the world. Mr. David Hamilton Koch believe strongly in
giving while living and had one idea that never changed
in his mind, that you should use your wealth to help
people and he decided to give USD2,000,000.00 Million
Dollars to randomly selected individuals worldwide
before his death on the 23rd of August 2019.

On receipt of this email, you should count yourself
as the lucky individual. Kindly get back to me at your
earliest convenience, so that I will know your email
```

```
address is valid. Email me at (charleskoch131@gmail.com)
you can also visit the web page of late Mr. David
Hamilton Koch to know more about the Hamilton Foundation
and this grant: http://en.wikipedia.org/wiki/David_H._Koch

Regards,
Mr. Charles Koch
```

This produced the following label vector at less than 0.05s of runtime:

```
array([[0.65761214, 0.38953513]], dtype=float32)
```

Remarkably, this minimally trained model correctly classifies the email as spam with a confidence of roughly 66%.

We tried to test whether it identified ham correctly with the following email (again, taken from the kde-devel mailing list):

```
Subject: Framework for encrypted storage for apps

Hi,

One important question here is how do you want the secrets to be unlocked.
Is the user going to type in some master password every time the otpclient is
started?

If that is the case, you can use QCA to encrypt and decrypt the data you need
to store securely.

If that is not the case - if you don't want to always ask the user for the
password, then you can expect quite a complicated story of how to make
something like kwallet safe.
```

This produced the following label vector:

```
array([[0.3075037, 0.6994512]], dtype=float32)
```

Again, it correctly identifies the email as ham with a confidence of roughly 70%. These results were much better than expected, given that the entire corpus of emails that this model was trained with totalled 8 emails.

Conclusions

This experiment has shown that a machine-learning based approach to spam filtering is completely impractical for all but the most resourceful of organizations. This is because of the following reasons:

1. Training a neural network or machine learning model requires immense amounts of computing resources, and makes it impossible to train anything bigger than a synthetic dataset composed of only 10 emails in a reasonable amount of time on a personal laptop. In comparison, open-source solutions like rspamd and SpamAssassin, which use Naive Bayes classifiers, run perfectly well on single-core x86 processors with only a few gigabytes of memory.
2. The data structures used in machine learning, in particular large and extremely sparse matrices, are inefficient and require additional effort in optimisation. In particular TensorFlow has no support for optimising the storage of sparse matrices yet.
3. Incremental learning is in general very difficult for machine learning models, and while TensorFlow does offer a few methods for saving the learning state of a model and incrementally continuing at a later stage, this is far more difficult than incrementally training Naive Bayes classifiers like SpamAssassin.
4. Machine learning is still an inexact science, compared to established Naive Bayes approaches. Hyperparameters need to be tuned to each model and the incoming training dataset, and most of this tuning is still based on trial and error.
5. And finally, even though none of the above are insurmountable challenges, machine learning based approaches do not yet offer a compelling advantage, either in terms of speed or accuracy, over Naive Bayes based approaches that justified the cost and effort investment in this novel approach. Naive Bayesian spam filters have been the mainstay of the industry and are still good enough.

That said, for an organization with the appropriate amount of resources, investing in machine learning approaches to spam filtering seems to pay off. Google says that they now block an additional 100 million spam messages that slip by their Naive Bayesian filters by using TensorFlow to prototype machine learning models and a companion application called TensorBoard to quickly evaluate and decide which of the models are expected to be useful (Neil).

In conclusion, while machine learning offers exciting possibilities for more advanced forms of spam filtering, it is currently not a viable option to replace existing and established Naive Bayesian spam filters as a first-line spam filtering agent owing to their simplicity and speed.

Bibliography

Cornell University Blog 2018:
http://blogs.cornell.edu/info2040/2018/10/27/bayes-theorem-in-email-spam-filtering/

Fallows, Deborah (2007): Spam 2007. Pew Research Center.
https://www.pewresearch.org/internet/2007/05/23/spam-2007/

Kopp, Johannes & Lois, Daniel (2012): Sozialwissenschaftliche Datenanalyse. Eine Einführung. Wiesbaden: Springer VS.

Kumaran, Neil (2019): Spam does not bring us joy—ridding Gmail of 100 million more spam messages with TensorFlow.
https://cloud.google.com/blog/products/g-suite/ridding-gmail-of-100-million-more-spam-messages-with-tensorflow

Lagrana, Fernando (2016): E-Mail and Behavioral Changes. Uses and Misuses of Electronic Communications. Hoboken: John Wiley & Sons, Incorporated.

Manduk, Szymon (2019):
https://aigeekprogrammer.com/binary-classification-using-logistic-regression-and-keras/

Sahami, Mehran, Dumais, Susan, Heckerman, David, Horvitz, Eric (1998): A Bayesian Approach to Filtering Junk E-Mail. https://www.aaai.org/Papers/Workshops/1998/WS-98-05/WS98-05-009.pdf

Silva, Thiago Christiano, Zhao, Liang (2016): Machine Learning in Complex Networks. Cham: Springer.