Business News Classification Engine

Springboard Capstone Project 2

James Flint | mail@jamesflint.net | 2018-03-30

Requirement

Building on the work done in my first Capstone Project, use CurationCorp's labelled news database to create a neural net-based topic classification & tagging engine for news articles, and make this functionality available via a cloud-based API.

Client

My client is CurationCorp.com, a news summarisation platform. They are currently paying freelancers to select and summarise articles in this manner. This is expensive and non-scalable. As a result of this project, I hope that they will be able to automate much of their editorial process, and reserve human intervention for final edit and sign-off, instead of low-level textual processing tasks.

Data

CurationCorp has a clean, human-curated database of 43,502 summarised and labelled news articles, to which I have access. For raw source data, we'll be using a sample of data from a standard news database (LexisNexus Moreover, format: CSV) or the news aggregator dataset at http://archive.ics.uci.edu/ml/datasets/News+Aggregator.

Approach

1. Data wrangling
2. Compare Classifiers
    a. A multi-layer neural net (NN)
    b. A convolutional neural net (CNN)
    c. A long/short term memory neural net (LSTM)
    d. A very deep convolutional neural net (VDCNN)
3. Build a prediction API
4. Future research
5. Credits

1. Data wrangling

The test data has been provided by the client as a series of topic-specific JSON files, and is the same as that used in the first Capstone Project. As a result, no major data wrangling was needed, as we have already extracted the data, filtered it by content field, normalised it and saved it as individual JSONs.

The relevant data in the target files are:

- title = article title
- date = original date of publication
- source_name = name of publisher
- source_url = URL of publisher
- content = article body content

There are, in total, 43,502 files, with an average of 92.8 words of body content per file.
As this project is focussed on processing and comparison at the level of the article content, the only data that concern us during both this and the previous project are those contained in the “title” and "content" (aka “body”) fields of the training and test JSONs.

In order to handle this data and feed it to our neural nets, we converted the title and body content fields into a data frame (see **capstone_project_data_wrangling.ipynb** and/or **create_dataframe.py**), both as separate columns and as a combined column, allowing us to try the nets just on the body texts and on the body texts and titles, to see if this made a difference to the outcome.

The articles were then shuffled, all columns except title and body text dropped, and the whole dataset exported as a csv file, ready for importing by the text pre-processing functions.

```
shuffled_df = all_articles_df.sample(frac=1).reset_index(drop=True)
shuffled_df = shuffled_df.drop(["PublishedDate", "Source_name", "Source_url"],
                               axis=1)
shuffled_df.to_csv("shuffled_df.csv")
```

As we can see from the results that follow, training on both title and body text meant substantially improved results, no doubt because the title texts tend to be rich in subject-specific keywords that help to reinforce the meaning of the body texts.

A quick analysis of the topic categories revealed the following distribution of content:

cpr_articles	6846
blackswans	4837
batterytech	4092
financialservices	3986
carboneradication	3690
sharingeconomy	3574
digitalads	2920
internetofthings	2627
property	2132
digitalhealth	1943
digitalcurrency	1914
ai	1722
blockchain	1650
educationtech	1555

Clearly this is an unbalanced distribution, as there are more than four times the number of counter-party risk articles (CPR) as educationtech, with a spread of other topics in between.

We use this dataset for training the first classifiers, but later in the project created a new, more balanced dataset, by multiplying up the content samples in each category until new, more even proportions of content for each topic were achieved.

```
# the following code copies a category's content and adds it to a
# new dataset, df_append.

df_row = all_articles_df['Category'] == "batterytech"
df_append = all_articles_df[df_row]
appended = appended.append([df_append], ignore_index=True)
appended['Category'].value_counts()
```

Although this involved duplicating content, the classifiers didn't seem to mind – quite the opposite in fact. When trained on this larger, balanced, dataset the neural nets displayed far superior performance to that achieved when trained on the unbalanced data.

The distribution of content in the new, balanced dataset, was as follows:

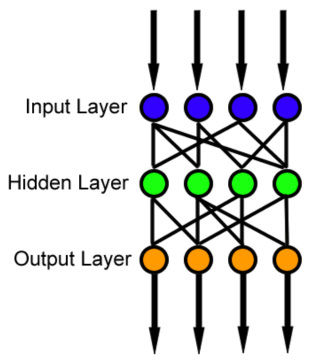
batterytech	16368
financialservices	15944
internetofthings	15762
digitalhealth	15544
digitalcurrency	15312
property	14924
carboneradication	14760
digitalads	14600
blackswans	14511
sharingeconomy	14296
educationtech	13995
ai	13776
cpr_articles	13692
blockchain	13200

2. Compare Classifiers

a. Multi-layer Neural Net

To get things started, we started by coding up a basic multi-layer neural net using TensorFlow and Keras (see **capstone_project_multi-layer-NN-body-text-only.ipynb**).

A **multilayer perceptron** (MLP) is a class of feed-forward artificial neural network, that uses at least three layers of nodes. Except for the input nodes, each node is a neuron that uses a nonlinear activation function – in our case, the "softmax" function. The network utilizes a supervised learning technique called backpropagation for training.



The feedforward neural network was the first and simplest type of artificial neural network devised. Information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops involved.

Multi-layer networks consist of multiple layers of computational units, usually interconnected in a feed-forward way. Each neuron in one layer has directed connections to the neurons of the subsequent layer. Multi-layer networks use a variety of learning techniques, the most popular being back-propagation.

Back-propagation has two phases. The first phase, propagation, involves the following steps:

1. Propagation of the training data forward through the network to generate the output value(s)
2. Calculation of the cost (i.e. the error term)
3. Propagation of the output activations back through the network using the training pattern target in order to generate the deltas (the difference between the targeted and actual output values) of all output and hidden neurons.

The second phase, weight update, has these steps:

1. The weight's output delta and input activation are multiplied to find the gradient of the weight.
2. A ratio (percentage) of the weight's gradient is subtracted from the weight.

This ratio (percentage) influences the speed and quality of learning; it is called the *learning rate*. The greater the ratio, the faster the neuron trains, but the lower the ratio, the more accurate the training is. The sign of the gradient of a weight indicates whether the error varies directly with, or inversely to, the weight. Therefore, the weight must be updated in the opposite direction, "descending" the gradient. Learning is repeated (on new batches of data) until the network performs adequately.

To train our network we prepared our data by splitting the article data set 80:20 into train and test data, and then converted from strings to vectors using Keras's built in tokenizer.

```
# Split data into train and test
train_size = int(len(shuffled_df) * .8)
train_posts = shuffled_df["Body"][:train_size]
train_tags = shuffled_df["Category"][:train_size]
test_posts = shuffled_df["Body"][train_size:]
test_tags = shuffled_df["Category"][train_size:]

# Pre-process text into vectors
import keras.preprocessing.text as text
max_words = 1000
tokenize = text.Tokenizer(num_words=max_words, char_level=False)
tokenize.fit_on_texts(train_posts) # only fit on train
x_train = tokenize.texts_to_matrix(train_posts)
x_test = tokenize.texts_to_matrix(test_posts)
```

Sklearn's LabelEncoder function was used to convert the labels to a numbered index, and Keras's to_categorical method to then turn these into one-hot representations suitable for use by the neural network.

```
# Use sklearn utility to convert label strings to numbered index
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()
encoder.fit(train_tags)
y_train = encoder.transform(train_tags)
y_test = encoder.transform(test_tags)

# Converts the labels to a one-hot representation
from tensorflow.contrib.keras.python.keras import utils
num_classes = np.max(y_train) + 1
y_train = utils.to_categorical(y_train, num_classes)
y_test = utils.to_categorical(y_test, num_classes)
```

Now we're ready to build the network:

```
from keras.layers import Dense, Activation, Dropout

batch_size = 32
epochs = 2
model = Sequential()
model.add(Dense(512, input_shape=(max_words,)))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

In the code above, once we've defined the model as Sequential(),we can add our input layer. The input layer will take the max_words arrays for each comment. We specify this as a Dense layer in Keras, which means each neuron in this layer will be fully connected to all neurons in the next layer. We pass the Dense layer two parameters: the dimensionality of the layer's output (number of neurons) and the shape of our input data.

Choosing the number of dimensions requires some experimentation, and there is a lot of discussion on the best approach for doing this. It's common to use a power of 2 as the number of dimensions, so we'll start with 512. The number of rows in our input data will be the number of posts we're feeding the model at each training step (called batch size), and the number of columns will be the size of our vocabulary. With that, we're ready to define the Dense input layer. The activation function tells our model how to calculate the output of a layer.

To prepare our model for training, we need to call the compile method with the loss function we want to use, the type of optimizer, and the metrics our model should evaluate during training and testing. We'll use the cross entropy loss function, since each of our comments can only belong to one post. The optimizer is the function our model uses to minimize loss. In this example we'll use the Adam optimizer.

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

To train our model, we'll call the fit() method, pass it our training data and labels, the number of examples to process in each batch (batch size), how many times the model should train on our entire dataset (epochs), and the validation split. validation_split tells Keras what percentage of our training data to reserve for validation.

```
history = model.fit(x_train, y_train,
                    batch_size=batch_size,
                    epochs=epochs,
                    verbose=1,
                    validation_split=0.1)
```

You can see the validation loss decreasing slowly, epoch by epoch, when you run the model. When val_loss is no longer decreasing, we stop training to prevent over-fitting.

```
Train on 31311 samples, validate on 3479 samples
Epoch 1/2 31311/31311 [==============================] - 19s 607us/step - loss: 1.038
2 - acc: 0.6674 - val_loss: 0.8556 - val_acc: 0.7160
Epoch 2/2 31311/31311 [==============================] - 18s 564us/step - loss: 0.692
4 - acc: 0.7538 - val_loss: 0.8548 - val_acc: 0.7134
```

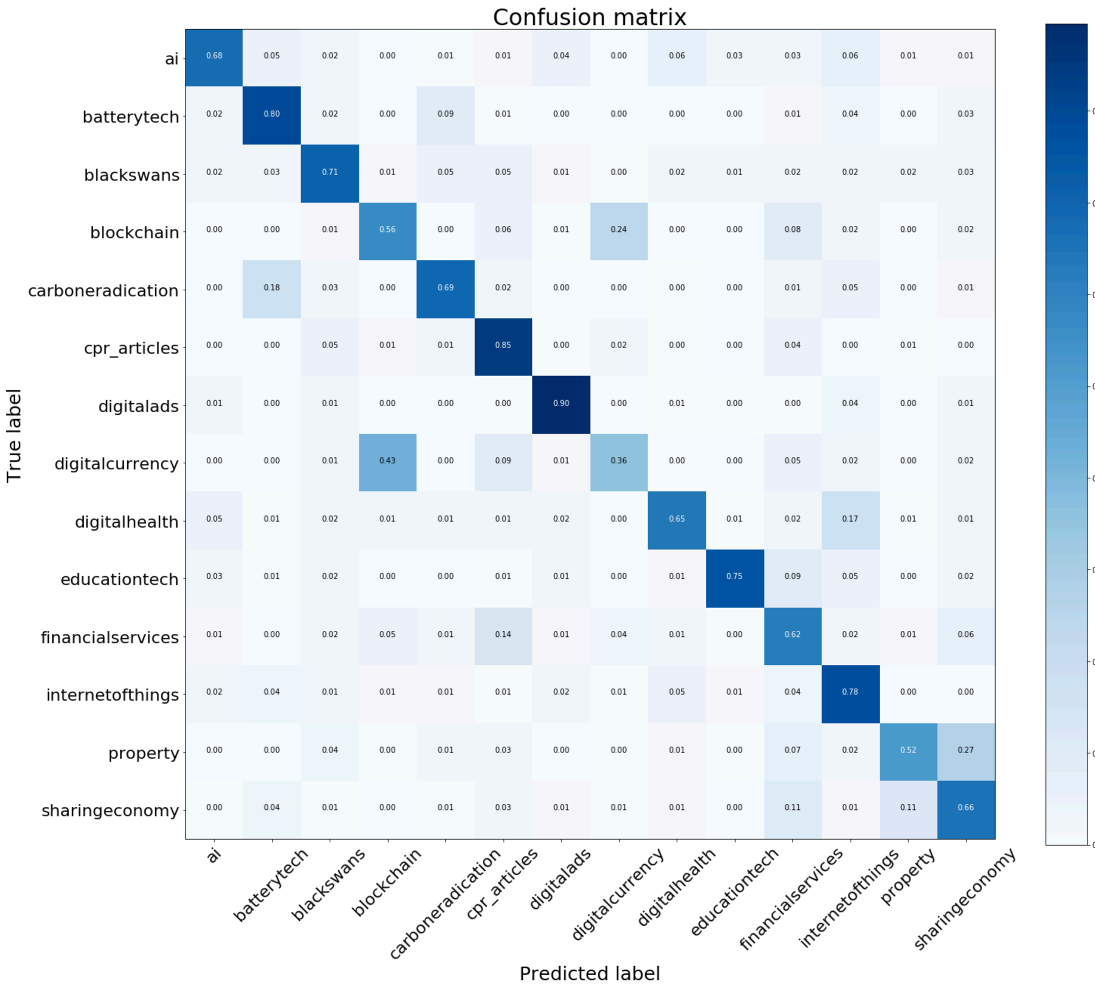
When we've trained out model, we can use it to predict stuff. The prediction accuracy of this first simple neural net (calculated using the model.evaluate method) is 0.70. Although reasonably good, this isn't as accurate as the Support Vector Machine (SVM) based on tf-idf we looked at in the first Capstone project.

Just as we did in the previous Capstone Project, we can get a sense of the overall performance of our model by using its predictions on our test data set to construct a confusion matrix.

One straightforward way to wring better performance out of this neural net might be to train it not just on the article body text, but on the title text too. Article titles tend to be rich in relevant keywords, and should help our classifier in finding relevant patterns.

We try this in the notebook **capstone_project_multi-layer-NN-body-text-and-title-text.ipynb**, and the results are immediately apparent. Just adding the Title field to the data used immediately boosts performance by nearly 2% to 0.7245. It's still, however, not as good as our SVM. Clearly more powerful techniques are required.

```
import matplotlib.pyplot as plt
import itertools
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(cm, classes,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):

    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title, fontsize=30)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45, fontsize=22)
    plt.yticks(tick_marks, classes, fontsize=22)

    fmt = '.2f'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label', fontsize=25)
    plt.xlabel('Predicted label', fontsize=25)

cnf_matrix = confusion_matrix(y_test_1d, y_pred_1d)
plt.figure(figsize=(24,20))
plot_confusion_matrix(cnf_matrix, classes=text_labels, title="Confusion matrix")
plt.show()
```



b. Convolutional Neural Net

Although fully connected feed-forward neural networks can be used to learn features as well as classify data, it is not practical to apply this architecture to images. A very high number of neurons would be necessary due to the very large input sizes associated with images, where each pixel is a relevant variable.

A more appropriate design is the Convolutional Neural Net (CNN), which apply a "convolutional" (in fact, a cross-correlation) operation to the input data in a manner that mimics the way that the convolutional neurons in the mammalian visual cortex process the data in a restricted region of the visual field known as the neurons "receptive field". The receptive fields of different neurons then partially overlap such that they cover the entire visual field, effectively tiling it with local feature recognition operations.

Convolutional networks tend to include local or global pooling layers which combine the outputs of neuron clusters at one layer into a single neuron in the next layer. For example, *max pooling* uses the maximum value from each of a cluster of neurons at the prior layer. Another example is *average pooling*, which uses the average value from each of a cluster of neurons at the prior layer

CNNs require relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage of this kind of neural network.

To prepare our text data for use in the CNN we've built for this project (**capstone_project _CNN_body_text_GloVe-preprocess_Categorical_Crossentropy.ipynb** and similar) we've used a package called GLoVe, which uses a dictionary to sift from the text data the 1000 or so words we're most interested in, put these into an array, and then ransform this, using Keras's embedding method, into an embedding layer that can be passed to the CNN.

```
# prepare GloVe word embedding vector pre-processing

import os

GLOVE_DIR = "./data/glove/glove.6B/"
embeddings_index = {}
f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

# prepare embedding matrix
# We're setting EMBEDDING_DIM (the dimension of the
# embedding vector to 100. We can tweak this later...

EMBEDDING_DIM = 100

num_words = min(MAX_NUM_WORDS, len(word_index))
embedding_matrix = np.zeros((num_words, EMBEDDING_DIM))
for word, i in word_index.items():
    if i >= MAX_NUM_WORDS:
        continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
```

We tested this CNN (code shown below) with a number of different parameters; the results are tracked in the included file **Results matrix.xlsx**, along with the results for all the other neural nets used as part of this project.

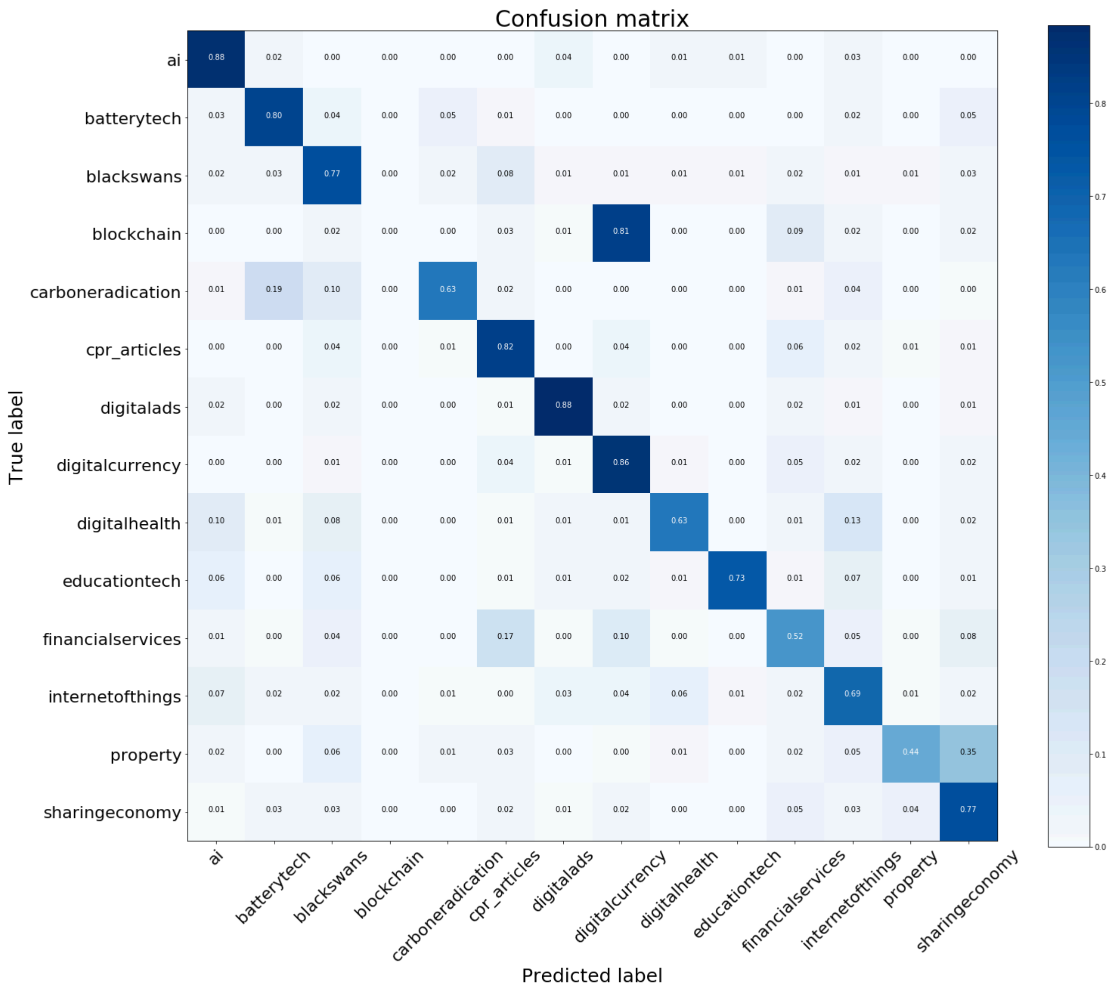
Tweaked parameters included training the CNN with just body text, and with both text and title; altering the loss function from binary to categorical (binary loss functions are designed to produce binary outcomes, rather than to sort results into fourteen topics as we require in this case, but despite that the function performed quite well); altering the length of the embedding sequence, the kernel size, and the number of filters, and the size of our GLoVe dictionary; and switching from the adam optimizer to rmsprop.

```
# load pre-trained word embeddings into an Embedding layer
# note that we set trainable = False so as to keep the embeddings fixed

from keras.layers import Embedding

embedding_layer = Embedding(num_words,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False)


# Set up and run CNN with Categorical Crossentropy loss
# function and f1 metrics

from keras.layers import Dense, Input, GlobalMaxPooling1D
from keras.layers import Conv1D, MaxPooling1D, Embedding
from keras.models import Model

sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences = embedding_layer(sequence_input)
x = Conv1D(100, 4, activation='relu')(embedded_sequences)
x = MaxPooling1D(5)(x)
x = Conv1D(100, 4, activation='relu')(x)
x = MaxPooling1D(5)(x)
x = Conv1D(100, 4, activation='relu')(x)
x = GlobalMaxPooling1D()(x)  # global max pooling
# x = Flatten()(x)
x = Dense(100, activation='relu')(x)
preds = Dense(len(labels_index), activation='softmax')(x)

model = Model(sequence_input, preds)
model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=[f1])

# happy learning!
model.fit(x_train, y_train,
          batch_size=32,
          epochs=5,
          validation_data=(x_val, y_val))
```

While some of these changes improved things, they didn't improve things that much. The best performance we managed was an accuracy of 0.7263 on a CNN with a 2000 word dictionary, a 130 word sequence length (long enough to include both title and all of the article body text), 150 filters, a kernel size of 5, and an rmsprop optimizer. But this was still quite a long way off the 0.77 accuracy of the SVM from our first Capstone Project.

One thing did make a significant difference to this performance, however – and that was switching from the unbalanced dataset to the balanced dataset we described in Section 1, above. Training the CNN on this boosted its accuracy considerably, to 0.7614. Still not quite as good as the SVM, but in the ballpark.

Here’s the confusion matrix from that model. One of the things to note here, in contrast to the situation with the multilayer in 2a, is the models insistence on classifying all blockchain articles as digital currency. Throughout both the Capstone projects all the classifiers have found it very difficult to differentiate between these two topics, a fact that strongly argues in favour of combining them into a single editorial category.



c. Long Short-Term Memory Neural Net (LSTM)

In the mid-90s, a variation of on recurrent networks – which expose part of the training data directly to the hidden layer in order to provide additional context – was proposed by the German researchers Sepp Hochreiter and Juergen Schmidhuber. They suggested adding in so-called Long Short-Term Memory units, or LSTMs, which were "gated" cells that could contain information outside the normal flow of the recurrent network in a gated cell.

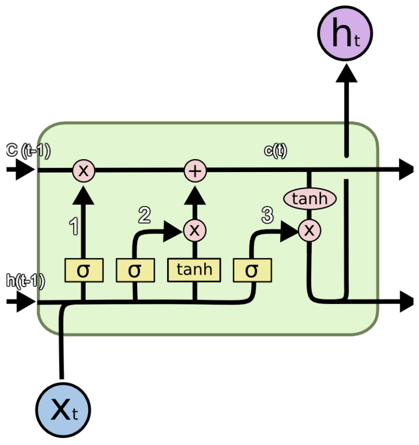
Information can be stored in, written to or read from these cells, much like data in a computer's memory. The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open and close. Unlike the digital storage on computers, however, these gates are analog and implemented with element-wise multiplication by sigmoid functions. Analog has the advantage over digital of being differentiable, and therefore suitable for backpropagation.

The gates therefore act on the signals they receive, and much like the nodes of the recurrent neural network they sit within, they block or pass on information based on its strength and import, which they filter with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are adjusted via the networks learning process. That is, the cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, back-propagating error, and adjusting weights via gradient descent.

In other words, the LSTM is capable of learning what information to store in long-term memory, and what information to get rid of. This allows them to preserve a constant error during the process of back-propagation (a problem with other kinds of neural net) and also to forget information when it might not be immediately relevant (for example when switching between documents when analysing a text corpus, as the separate documents might not have any relationship to one another, and the memory cells used to help formulate relationships in the first document should be reset to zero before it starts work on the second).

This diagram shows the logic of an LSTM unit. The various elements are:

- **X** : Scaling of information
- **+** : Adding information
- **σ** : Sigmoid layer
- **tanh**: tanh layer
- **h(t-1)** : Output of last LSTM unit
- **c(t-1)** : Memory from last LSTM unit
- **X(t)** : Current input
- **c(t)** : New updated memory
- **h(t)** : Current output

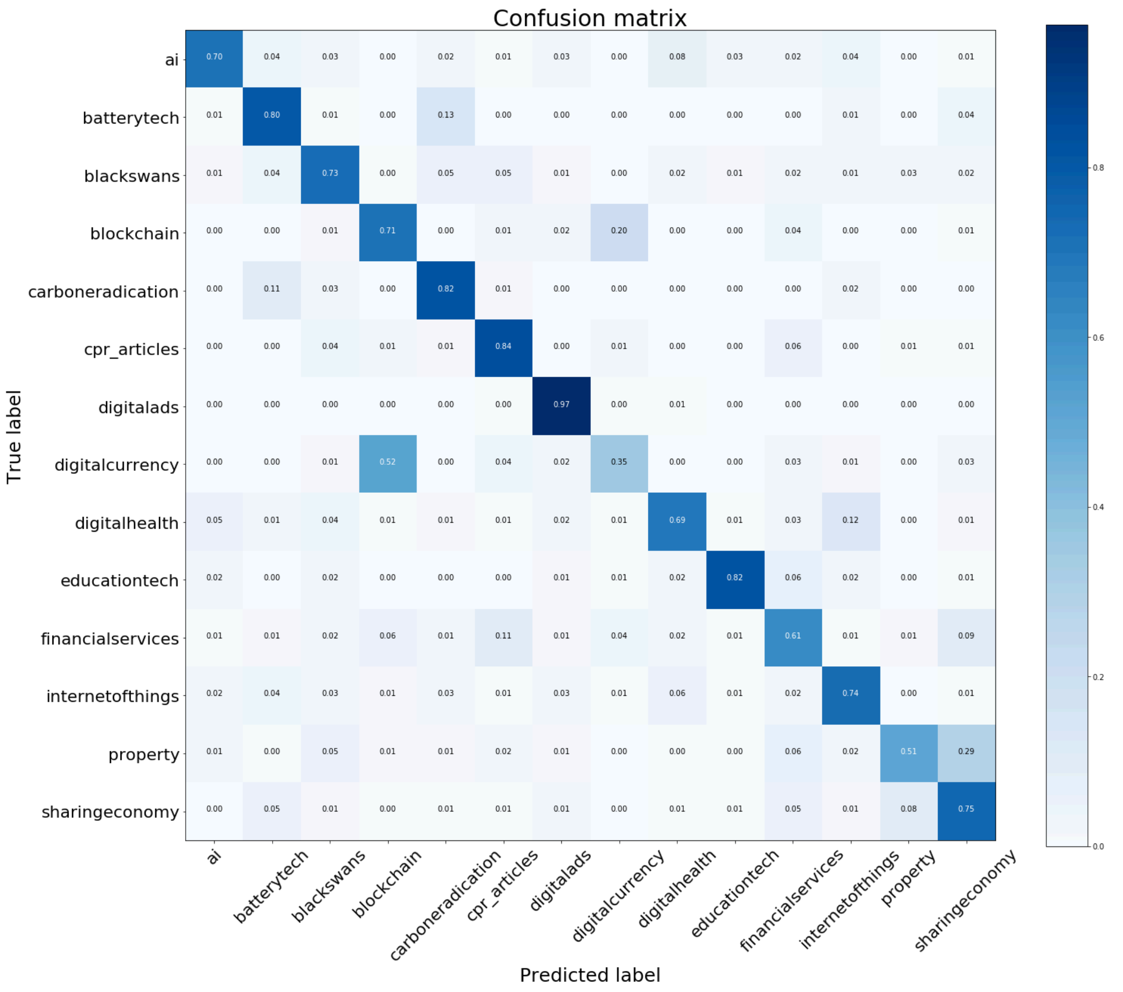


For this project, we used a relatively straight-forward LSTM hidden layer as shown in the code below (and in **capstone_project_LSTM_body_text,ipynb** and related notebooks). For pre-processing we used the same Keras tokenizer method we used for the multi-layer network in section 2a, above.

```
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(MAX_NUM_WORDS, embedding_vector_length,
input_length=MAX_SEQUENCE_LENGTH))
model.add(LSTM(150, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(14, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=[f1])
print(model.summary())
model.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=10, batch_size=64)
```

We ran several iterations of this, as catalogued in **Results matrix.xlsx**, trying similar combinations of settings to those applied to the CNN in 2b. Strangely, a set of settings that should not have worked well, combined with a mis-set maximum sequence length for the embedding layer, worked slightly more effectively than the example in which the settings were "correct". Even so, accuracy didn't climb above 0.7418, at least not while using the unbalanced dataset.

One other thing to note is that the LSTMs took two or three times the number of epochs to train before they began over-fitting and their prediction accuracy started to decline. At their peak, however, the results are good. We can see from this confusion matrix that the accuracy across topics is pretty even, and though there is some equivocation between digital currency and blockchain (and also between internetofthings, sharingeconomy and property) it's not as pronounced as with the CNN.

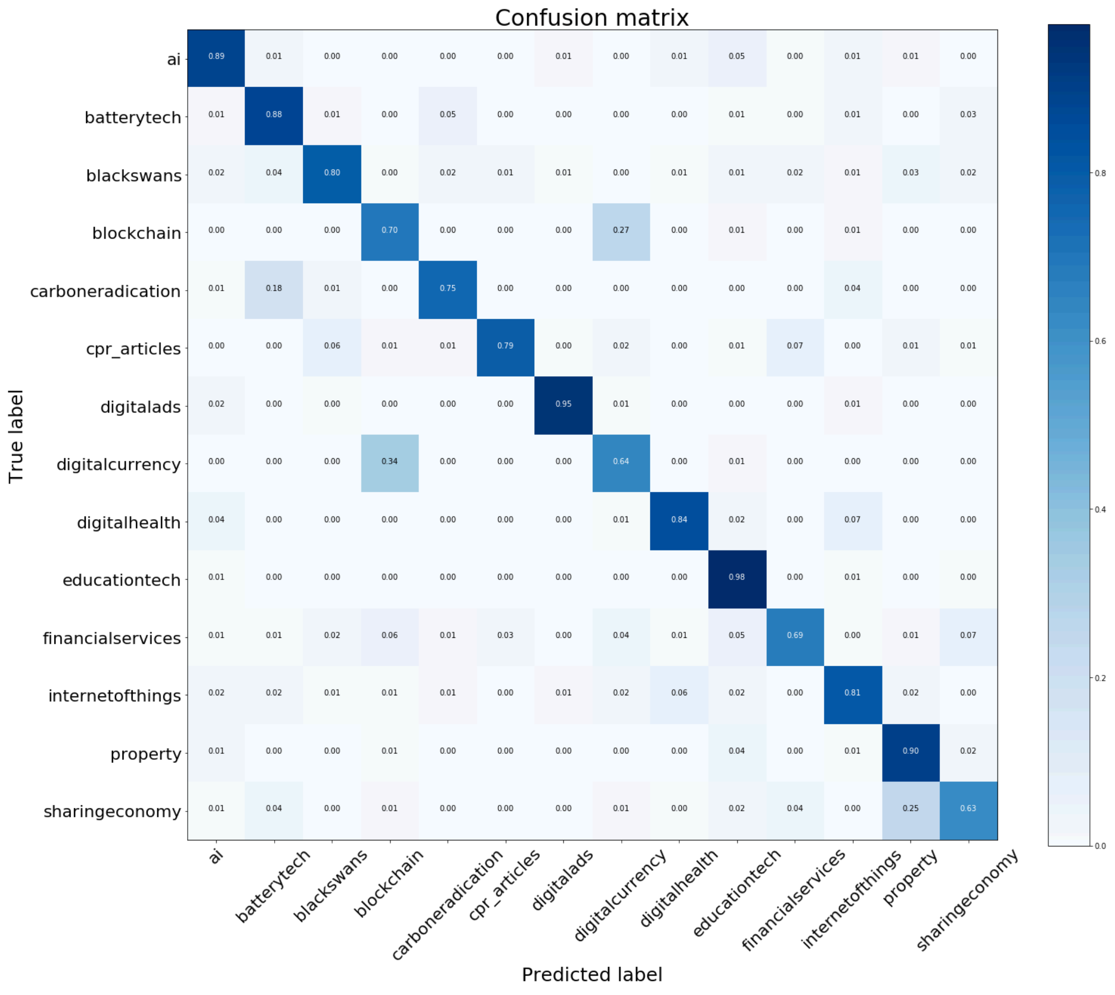


Alongside a traditional LSTM, it's worth looking at another formulation of the network, the *bidirectional* LSTM. A bidirectional layer – which can just as equally be applied to a recurrent neural network (RNN) as a LSTM – is an extension of the network that can improve model performance on sequence classification problems.

The idea was originally developed for RNNs in 1997, and involves duplicating the first recurrent layer in the network, then providing the input sequence in normal fashion to the first version and, simultaneously, a reversed version of the input sequence to the second. In this way the network is able to take account of the future elements in the sequence as well as the past ones. In theory this approximates the way in which humans anticipate future elements in a spoken sentence before they hear them, a technique that helps us make better sense of what is being said. In effect, the network is being given better context for each element, and that context leads to faster and better learning.

However, in this case, adding a bidirectional extension didn't really seem to help much. In fact, it actually reduced accuracy, to 0.7225. This may be because, while helpful in extracting meaning from conversation – for example in a chatbot application – knowing the order of words doesn't really make much difference in a topic classification situation, as what matters is the clusters of relevant keywords, not the sequence they are in.

What does make a big difference to our LSTM classifier, however, just as it did to our CNN, is training it on our expanded and balanced data set. This has a very marked affect, bumping accuracy nearly 10% from 0.7308 to 0.8127. Finally we have a model that outperforms the SVM from our first Capstone project. The confusion matrix associated with it is shown below:



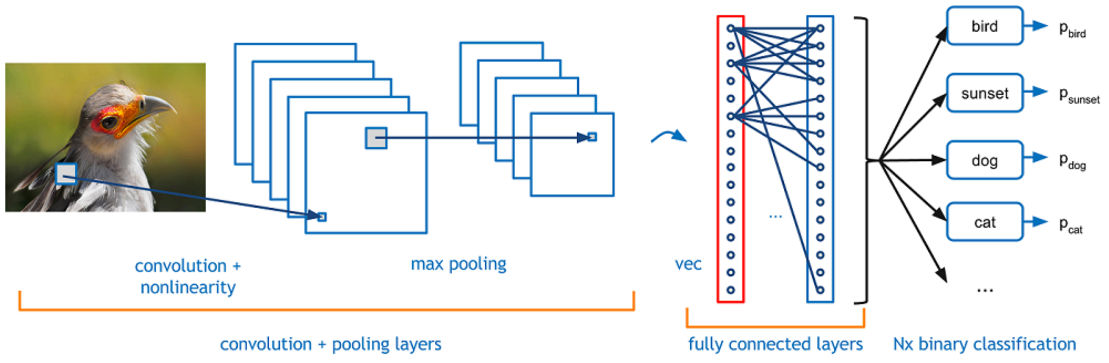
What's also particularly nice about this set of results is that for the first time we're seeing a lot of the equivocation around blockchain/digitalcurrency and property/sharingeconomy start to reduce.

d. Very Deep Convolutional Neural Net (VDCNN)

Before we abandon our search for a viable model for our topic classifier, it's worth looking at one other approach: that of the Very Deep Convolution Neural Net, as suggested in a 2016 Arvix paper by Conneau, Schwenk, Barrault and Lecun (https://arxiv.org/abs/1606.01781).

This paper suggests a novel approach to NLP and textual analysis inspired by the way in which CNNs operate on image files. Not only did Conneau et al. suggest using a CNN with 29 layers (hence the "very deep" part of the name – the number of layers can even stretch into the hundreds), but they suggested allowing the network to treat text like an image at the atomic level of the character, rather than by tokenizing words into vectors using the kinds of bag-of-words, tf-idf, GLoVe and word2vec techniques with used hitherto in this project and its predecessor.

A CNN is designed to handle bitmapped image data. It works its way around the image by taking successful snapshots of small sections (each shift of the attention window being referred to as a "stride"), converting these into embedding vectors, and using these to train the network.

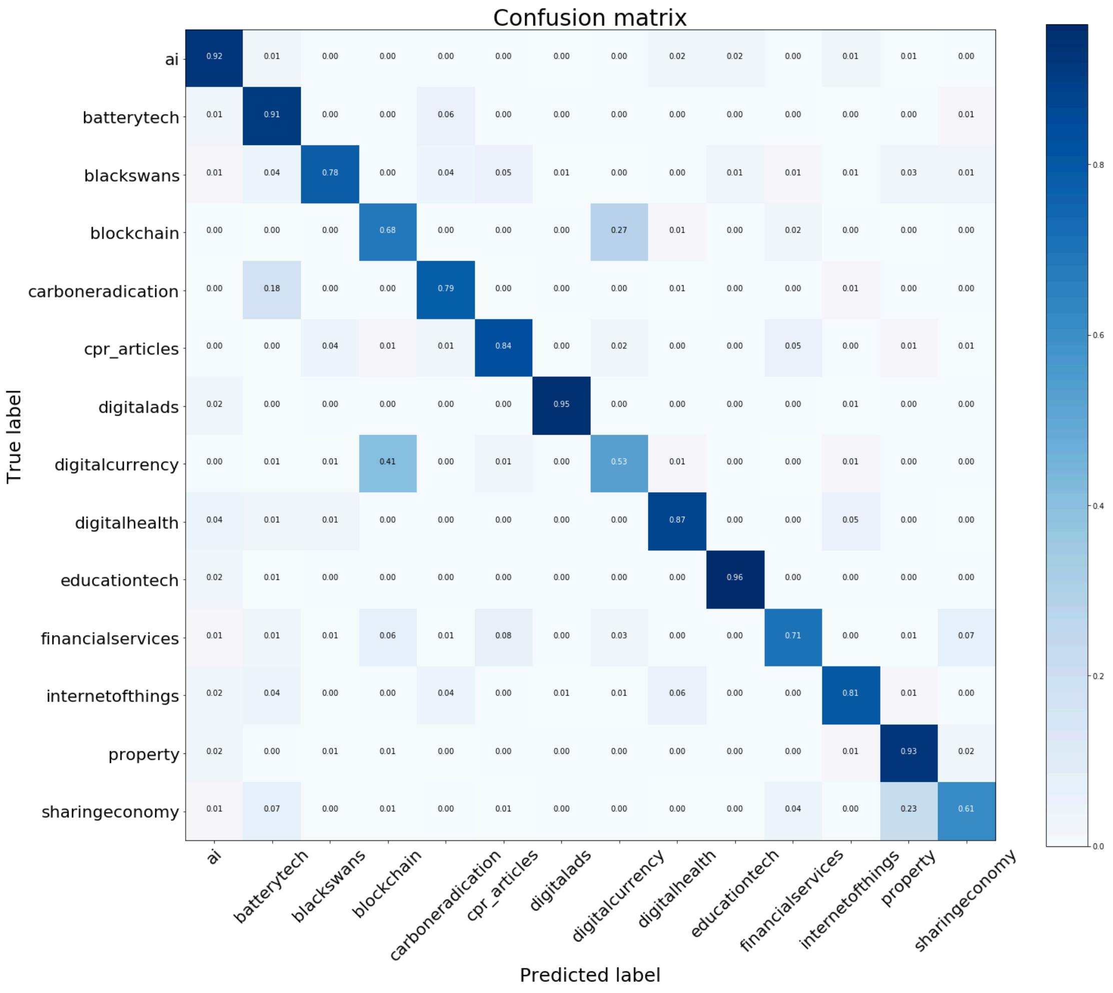


Conneau et al. suggest that a similar technique could be used for NLP, by having an attention window stride across a string and picks up patterns at the level of the letter, much as it looks at patterns at the level of the pixel when dealing with an image. For that reason we need to create an embedding vector from our texts, one that turns them into images made of letters rather than pixels – images that only have one dimension, as opposed to the two dimensions of a black and white image, and the three dimensions used to handle colour.

We set up a VDCNN and fed it with our dataset (full code is in **capstone_project_CNN_body_text+title_VDCNN.ipynb**). The smaller, unbalanced set, didn't have sufficient data for the network to generate any results at all. But the larger, balanced dataset performed extremely well, producing accuracy scores as high as 0.8055, outperforming the SVM and directly comparable to the LSTM, with a similar number of epochs required (9) in order to get optimal results. In addition, the pattern of equivocation in the contested topics (blockchain/digitalcurrency, property/sharingeconomy) was very similar across the two kinds of network.

On the downside the VDCNN was quite slow to train on account of the depth of the network requiring 2-3 days on an i7 dual core Macbook. We also had significant problems saving the model and reloading the problem for use by the prediction API, which we were not able to resolve.

```
# Pre-process text data into vectors of letters

ALPHABET = 'abcdefghijklmnopqrstuvwxyz0123456789-,;.!?:'"/|_#$%ˆ&*˜'+=<>()[]{} '
FEATURE_LEN = 1024 #maxlen

def get_char_dict():
    char_dict={}
    for i,c in enumerate(ALPHABET):
        char_dict[c]=i+1
    return char_dict

def char2vec(text, max_length=FEATURE_LEN):
    char_dict = get_char_dict()
    data=np.zeros(max_length)

    for i in range(0, len(text)):
        if i >= max_length:
            return data

        elif text[i] in char_dict:
            data[i] = char_dict[text[i]]

        else:
            data[i]=68
    return data

data=[]
for text in list_sentences_train:
    data.append(char2vec(text.lower()))
data=np.array(data)
```



	ai	batterytech	blackswans	blockchain	carboneradication	cpr_articles	digitalads	digitalcurrency	digitalhealth	educationtech	financialservices	internetofthings	property	sharingeconomy
ai	0.92	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.00	0.01	0.01	0.00
batterytech	0.01	0.91	0.00	0.00	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01
blackswans	0.01	0.04	0.78	0.00	0.04	0.05	0.01	0.00	0.00	0.01	0.01	0.01	0.03	0.01
blockchain	0.00	0.00	0.00	0.68	0.00	0.00	0.00	0.27	0.01	0.00	0.02	0.00	0.00	0.00
carboneradication	0.00	0.18	0.00	0.00	0.79	0.00	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.00
cpr_articles	0.00	0.00	0.04	0.01	0.01	0.84	0.00	0.02	0.00	0.00	0.05	0.00	0.01	0.01
digitalads	0.02	0.00	0.00	0.00	0.00	0.00	0.95	0.00	0.00	0.00	0.00	0.01	0.00	0.00
digitalcurrency	0.00	0.01	0.01	0.41	0.00	0.01	0.00	0.53	0.01	0.00	0.00	0.01	0.00	0.00
digitalhealth	0.04	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.87	0.00	0.00	0.05	0.00	0.00
educationtech	0.02	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.96	0.00	0.00	0.00	0.00
financialservices	0.01	0.01	0.01	0.06	0.01	0.08	0.00	0.03	0.00	0.00	0.71	0.00	0.01	0.07
internetofthings	0.02	0.04	0.00	0.00	0.04	0.00	0.01	0.01	0.06	0.00	0.00	0.81	0.01	0.00
property	0.02	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.93	0.02
sharingeconomy	0.01	0.07	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.00	0.04	0.00	0.23	0.61

3. Build a prediction API

We wish we could report that once the models were trained it was a relatively simple process to incorporate them into an online API, but this was not the case at all! There were various problems to overcome, some of which proved very time consuming.

As previously mentioned, the Keras library seemed unable to correctly save and reload the VDCNN model to disk. Other, simpler models worked fine, and fortunately the LSTM model that slightly improved upon the performance of the VDCNN didn't exhibit any problems, so we were able to use that for our API instead.

Please note that we're not able to call our f1 accuracy function from the loaded models, so the versions of the nets that we're saving use the built in Keras "accuracy" function. The model we trained and saved for use in the API with this altered accuracy function can be found in **capstone_project_LSTM_save_load_model_tester.ipynb**.

Running the API in a local virtual environment worked well… but only in Python 3.6. At first we tried to use the Google Cloud App Engine solution to host the API online, but that forced Python 2.7, which threw a conflict between the Keras and Numpy libraries, which again we could not resolve.

In the end we switch to Heroku and exported a local environment in Conda with the appropriate dependencies installed (including Python 3.6) to a remote server. This worked fine, and the application is now running happily at this address:

https://afternoon-shelf-15457.herokuapp.com/form

This URL hosts a webform that allows you to submit an article manually and have it classified; the application also provides a JSON API at:

https://afternoon-shelf-15457.herokuapp.com/predict

The API can be queried using the following python code, which can be found in the **API-tester.ipynb** notebook.

```
import requests
import json
url = 'https://afternoon-shelf-15457.herokuapp.com/predict'
s = {"title":"foo", "body":"bar"}
s_json = json.dumps(s)
headers = {'Content-Type': 'application/json'}
r = requests.post(url, data=s_json, headers=headers)
print(r.text)
```

Please note that "foo" and "bar" must be replaced with sample texts of a reasonable length. One-word queries can cause the application to throw an error. Sample texts for testing the model are included in the git repo in the file **Test articles for online form.rtf**.

The code for the heroku application can be found on the git repo in the **topic_api** folder. The key API code, in **main.py**, is printed on the next page. Although the executable codebase is very small, the full application is quite large – around 330mb – largely because of the extensive library of dependencies, including TensorFlow and Keras, that is required to support it.

```
 # initialize our Flask application and the Keras model
app = Flask(__name__)
model = None
tokenizer = None

# load the pre-trained Keras model
model = load_model(os.path.join(MODEL_DIR, "LSTM-body-text+title-19-epochs.h5"))

# load the saved Keras tokenizer, for text pre-processing
with open("tokenizer_LSTM.pickle", "rb") as handle:
    tokenizer = pickle.load(handle)

def remove_html_tags(data):
    '''Remove any html tags from submitted text'''
    p = re.compile(r"<.*?>")
    return p.sub("", data)

def predict_category(text_to_predict):
    '''Process text and predict category'''
    # pre-process the text and create
    # embeddable vector of the right length
    cleantext = remove_html_tags(text_to_predict)
    sequence = tokenizer.texts_to_sequences([cleantext])
    trans_text = pad_sequences(sequence, maxlen=MAX_SEQUENCE_LENGTH)
    # use the model to predict the right category
    prediction = model.predict(numpy.array(trans_text))
    # match the prediction number to the right label
    predicted_label = CATEGORIES[numpy.argmax(prediction)]
    # print(predicted_label, "\n")
    return predicted_label

@app.route("/predict", methods=["POST"])
def predict():
    '''Handle the POST api request'''
    if flask.request.method == "POST":
        if request.headers["Content-Type"] == "application/json":
            payload = request.json
            title = payload["title"]
            body = payload["body"]
            text_to_predict = str(title + " " + body)

            predicted_label = predict_category(text_to_predict)

            return flask.jsonify(predicted_label)

    else:
        return "415 Unsupported Media Type ;)"
```

4. Future research

Text Summarisation

Add in a text summarisation function, which will summarise the online source of an article and return a coherent 130-word abstract, which can then be classified using the Business Classification Engine. To do this try:

- The Gensim implementation of Textrank
- Making a call to the Skim.it online text summary tool https://docs.skimtechnologies.com/

Tag generation

Create a keyword generation tool for the classified articles, using Textrank (either the NLTK implementation or something based on the example here:

- https://github.com/davidadamojr/TextRank
- https://www.quora.com/How-can-I-use-machine-learning-to-propose-tags-for-content

Improve the model using user feedback

When returning the classification to the user, request confirmation that the classification made sense. Collate these replies and store them as new labelled data, then run an overnight cron job on the server every week to retrain the model using both original labelled data and the new labelled data, and update the saved model with the new version.

Improve presentation and enable batch processing

Using the batch processing code already developed in the project notebooks, extend both the web version and the API to handle batches of articles sent for classification.

5. Credits & Sources

Tokenizing text data in Keras
http://www.orbifold.net/default/2017/01/10/embedding-and-tokenizer-in-keras/

Very Deep Convolutional Networks for Text Classification (paper)
https://arxiv.org/abs/1606.01781

Keras implementation of a VDCNN model (code)
https://github.com/yuhsinliu1993/VDCNN

Keras API
https://blog.keras.io/building-a-simple-keras-deep-learning-rest-api.html
https://github.com/jrosebr1/simple-keras-rest-api/blob/master/run_keras_server.py

Calculating the F1 metric in Keras
https://stackoverflow.com/questions/43547402/how-to-calculate-f1-macro-in-keras

Using GloVe in Python
http://textminingonline.com/getting-started-with-word2vec-and-glove-in-python

Text Classification using CNNs
https://richliao.github.io/supervised/classification/2016/11/26/textclassifier-convolutional/

Global Vectors for Word Representation
https://nlp.stanford.edu/projects/glove/

Build a CNN in 11 lines
http://adventuresinmachinelearning.com/keras-tutorial-cnn-11-lines/

How to Develop a Bidirectional LSTM For Sequence Classification
https://machinelearningmastery.com/develop-bidirectional-lstm-sequence-classification-python-keras/

Text Generation With LSTM Recurrent Neural Networks
https://machinelearningmastery.com/text-generation-lstm-recurrent-neural-networks-python-keras/

Miguel Grinberg: The Flask Mega-Tutorial
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world

How to deploy a Python Flask app on Heroku
https://medium.com/@johnkagga/deploying-a-python-flask-app-to-heroku-41250bda27d0

Implementing a RESTful Web API with Python & Flask
http://blog.luisrei.com/articles/flaskrest.html

Adit Deshpande: A Beginner's Guide To Understanding Convolutional Neural Networks
https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/

Reuters-21578 text classification with Gensim and Keras
https://www.bonaccorso.eu/2016/08/02/reuters-21578-text-classification-with-gensim-and-keras/

Classifying Yelp Reviews
http://www.developintelligence.com/blog/2017/06/practical-neural-networks-keras-classifying-yelp-reviews/