Using Neural Networks to Classify Based on Combined Text and Image Content: An Application to Election Incident Observation[*]

Alejandro Pineda[†] Walter R. Mebane, Jr.[‡]

August 20, 2019

[*]Prepared for presentation at the 2019 Annual Meeting of the American Political Science Association, Washington, D.C., August 29–September 1, 2019. Work supported by NSF award SES 1523355.

[†]University of Michigan, ajpineda@umich.edu.

[‡]University of Michigan, wmebane@umich.edu.

Abstract

The use of social media data in political science is now commonplace. Recent work in election forensics uses Twitter data to capture people's observations of incidents during the 2016 U.S. Presidential Election. Findings suggest that partisans report different types of election incidents on Twitter and are involved in partisan communication silos. That work uses automated text-based classification, but such tools do not use all available content—in particular the machine classifiers ignore images. For instance, some Tweets feature pictures of long lines or of smiling voters wearing "I voted" stickers. Human coders use such image information, but the machine algorithms do not. In this paper we introduce machine-learning methods to automate the coding of combined text and image content. Layers in a deep neural network combine and learn from features extracted from text and, where present, images. The method is used to examine incident observations from the Twitter Election Observatory using data from both the 2016 Presidential and 2018 Midterm elections.

1 Introduction

Election forensics is the field devoted to using statistical methods to determine whether the results of an election accurately reflect the intentions of the electors. One goal for such analysis is eventually to be able to incorporate information about direct observations of election processes. To try to gather individuals' reports of their experiences in the election process across the United States, large-scale surveys can be important (Stewart 2017), but it is natural as well to turn to social media data, which may offer more extensive and granular coverage.

In particular Mebane, Pineda, Woods, Klaver, Wu and Miller (2017) gathered individual observation data from Twitter.[1] During October 1–November 8, 2016, Mebane et al. (2017) used Twitter APIs to collect slightly more than six million original Tweets (excluding retweets) from which, using supervised text classification methods, Mebane et al. (2017) recovered 315,180 Tweets that apparently reported one or more election "incidents" (Mebane, Wu, Woods, Klaver, Pineda and Miller 2018).[2] An election incident is an individual's report of their personal experience with some aspect of the election process: for example, being in a long line at a polling place or not having to wait at all to vote, successfully casting a vote, or registering or having trouble registering. Even though human coders used all the information contained in or associated with a Tweet to label it as an incident or not, classification algorithms used by Mebane et al. (2018) use only text associated with the Tweet. Even though 2,150,079 of the 4,566,333 original Tweets that have unique augmented texts have images, the classification algorithms used by Mebane et al. (2018) ignore those images.

Here we describe our implementation of deep neural network classifier methods that use both text and images associated with Tweets. The network is constructed such that its text-focused parts learn from the image inputs, and its image-focused parts learn from the text inputs. Using

[1]Section 1.1 describes data collection details that bear on the current analysis.

[2]Labeling and classification in Mebane et al. (2018) proceeds in two stages. An initial stage of binary classification finds "hits" ($n = 315,180$) among a set of 4,566,333 original Tweets that have unique augmented texts: hits might precisely be described as apparent incidents. A second stage type-of-incidents classification using a multiclass approach on the 315,180 hits finds that 89,917 of the 315,180 are not incidents according to the multiclass coding rules applied to a slightly augmented (with place and date) corpus.

such a dual-mode classifier ought to improve performance. In principle our architecture should improve performance classifying Tweets that do not include images as well as Tweets that do.

Related Work: Political communication research has established the importance of images for creating visual frames, eliciting emotion, and grabbing readers' attention (Abraham and Appiah 2006; Iyer and Oldmeadow 2006; Mendelberg 2017; Casas and Williams 2019). Methods for automated content analysis, however, focus exclusively on text while ignoring the visual cues that create additional, latent meanings (Barberá 2015; Hopkins and King 2010). Automating analysis for digital content proves difficult because the *form* of data takes so many different shapes—text, image, audio, video, et cetera. This paper offers a solution: a method for the automated classification of multi-modal content. Research in this regard is scarce; while we use Artificial Neural Networks, recent efforts to handle multi-modal content in a single machine learning system leverage Support Vector Machines and Multimodal Transformers.

Recent work by Joo and Steinert-Threlkeld (2018) review technical details of deep learning and neural networks. Joo and Steinert-Threlkeld (2018) differentiate between neural networks and convolutional neural networks, give sample configurations from the machine learning literature, and review important concepts in deep learning research like weight sharing, local connectivity, and nonlinearity. We review back-propagation and long-short term memory units (or LSTM) in this paper.

In particular, Lin and Hauptmann (2002) focus on video classification of news content. The authors combine text features from closed-captions and visual features from images to classify broadcast news video. By experimenting with the system, the authors find that combining text and image features improves precision and recall. While they use a "meta-classification" strategy – multiple classifiers that essentially vote to determine final predictions – we use a single model that trains on both text and image features. We are classifying tweets, not news content and we do not rely on a meta-classifier. Instead our model combines features using three densely connected layers and there is no second or third model ”voting” on the prediction outcome. The model takes image and text as input and outputs a single classification decision for each tweet – two inputs, one output.

Yu, Li, Yu and Huang (2019) use a Multimodal Transformer (MT) model for image captioning. This task aims to automatically generate a description of an image—thus it requires some combination of image-and-text input processing. We recognize MT as a possible avenue for future research, however, our stated goal is not image captioning; it is content classification. While some of the data processing run parallel, the larger aims diverge. For example, unlike Yu et al. (2019), we are not processing video, but videos are broken down to image stills, and then similar techniques—like pooling, convolutions, and flattening—come into play for processing visual data.

1.1 Data

Mebane et al. (2018) describes the procedures used to collect the data we draw on.[3] The source data contain 505,112 unique augmented Tweet texts that have `place` and `fullname` information. Of these 25,013 are labeled (3,689 hits and 21,324 not-hits), and of the labeled Tweets 10,574 have image files (1,697 hits and 8,877 not-hits).[4] A Tweet being labeled as a hit means that, all things considered, the Tweet was judged to report an incident, not that the image alone represents an incident. Our initial sample contains 1,278 of these labeled Tweets that have images (165 hits and 1,113 not-hits). Note that most of the labeled Tweets were selected from the set of Tweets with `place` information using active learning methods and are not a random sample (Miller, Linder and Mebane 2019). The set of 1,278 Tweets is a random sample from the set of 10,574 labeled Tweets that have image files.

[3]For each of the 16,265,633 Tweets obtained using APIs, the resource, if any, located at each URL the Tweet content contains is inspected. Any text in the `og:description` field in the resource's HTML code is captured and appended to the text. For type-of-incident classification the date (month, day and year) and `place$fullname` of the Tweet is also appended to the text. Such appendings create "augmented texts," which also have URLs and HTML and non-ASCII characters stripped. Any URL in the `og:image` field in the resource's HTML code is captured, and the URL is followed to try to obtain the referent image file (Mebane et al. 2018).

[4]If labeled Tweets that have images whose files we could not acquire are included, then 10,985 have image files (1,724 hits and 9,261 not-hits).

Some variants of our models use text features produced as part of the classification methods used in Mebane et al. (2018). Using the set of unique augmented texts (retaining stop words and without stemming), classification in Mebane et al. (2018) uses a word n-gram model for the preprocessed text and a character n-gram model for hashtags[5] to convert the Tweet corpus into a document term matrix that is TF-IDF transformed (Leopold and Kindermann 2002; Lan, Tan, Low and Sung 2005). Features are selected using the coefficients of a linear support vector machine (SVM) with ℓ_1 norm penalty: features with SVM coefficients less than the mean of all coefficients are discarded (Rakotomamonjy 2003).

2 Models

The main problem in building the text-image model was getting data processed in a way that allowed both forms to take a similar shape. Building the architecture in Keras let us perform various transformations on the data in parallel. On the image side, we used convolutions and flattening to reduce dimensions; on the text side, we loaded each word's GloVe embedding, treating each word as a vector.

Looking at Figure *Model*, we see text and image get processed separately at first, before features are concatenated and fed to three densely connected layers (one of which outputs predictions). Because error gets back-propagated across the entire network, the model learns on both text and image simultaneously. The image side of the network relies on convolutions and flattening layers to extract features and reshape image data. The text side relies on GloVe word embeddings and LSTM nodes to model word vectors sequentially. Next we provide theory underlying neural networks and back-propagation, before detailing how text and image get processed by our model.

[5]Up to 5-grams are allowed for words and 2-, 3- and 4-grams for characters in hashtags.

2.1 Neural Networks

For data $X \in \mathbf{R}^d$, assume X is separable into classes $y \in \{1, 2, ..., k\}$ where assignment to class k means x_i shares relevant *features* with other data points in k. We need a statistical model that can learn the relevant features in X that map onto the associated classes in y:

$$X \mapsto y.$$

The model needs to understand the *relationships* between data points that *best determine class*. For instance, a simple linear model might look like:

$$\hat{y}(x, w) = \sum_{j=1}^{d} w_j x_j \tag{1}$$

where w are model parameters that capture relationships between x_i and $x_\ni$ and $\hat{y}$ is the predicted classes as a function of x and w. For a binary classification task – only two classes – this model becomes:

$$\hat{y}(x, w) = \text{sign} \sum_{j=1}^{d} w_j x_j \tag{2}$$

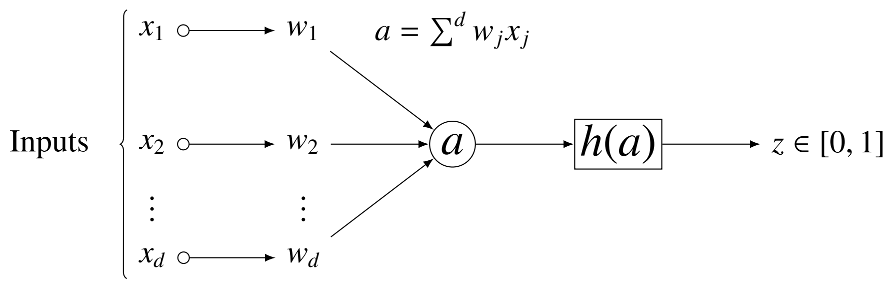
with classes defined as $[-1, 1]$ or even $[0, 1]$. For instance, social scientists working with a binary dependent variable should be familiar with logistic regression. We might use such a model for our purposes, of the form:

$$P(y = 1 |\, x, w) = h\left(\sum_{j=1}^{d} w_j x_j \right) \tag{3}$$

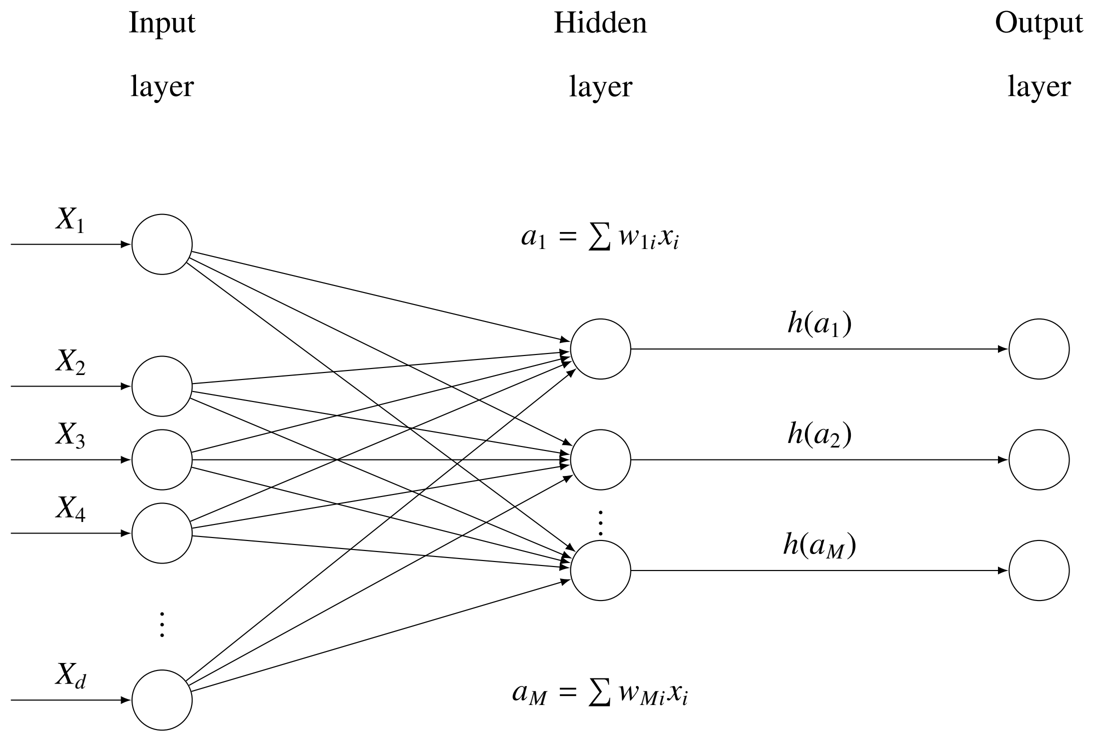
where h is the logistic function

$$h(a) = \frac{1}{1 + e^{-a}}.$$

With X in lower-dimensions, this model would be a reasonable approach, but let us assume that X has a large feature space and thus, requires a classifier that can scale well. To this end, we can represent our logistic regression graphically, as a network.



This particular form is called a *single-layer perceptron* and is the essential building block of neural networks. Note three essential components: an **input layer** (where X is fed into the model), **a node** (containing the dot product $X^T w$) and an **output layer** (re-labeled as z) that predicts class. Additionally, the logistic function used in Equation 3 is generalized to an activation function $h(a)$ that exists along the edge of the network (several different functions are used in this role). If we set $\hat{y} = a$ the single-layer perceptron approximates a logistic regression. Indeed, as we expand to more classes, we have more boundaries to consider, and we need a more powerful classifier.



We can capture more intricate relationships if we increased the number of weights in the model, as above. To handle the increase in weights, we add more nodes $\{a_1, a_2, ...a_M\}$ to the model. By increasing the number of nodes in the "hidden layer" we can approximate any boundary or function (we call the inner layers of a network hidden because they do not go directly observed). Note that we maintain the same activation function, but it gets fed different values depending on what edge we are looking at. This structure can handle high dimensional data, but it can handle even more complexity by increasing the number of hidden layers in the network, as we see in the next section.

2.2 The Back-Propagation Algorithm

This algorithm is how learning occurs. At each iteration of training, a combination of forward and backward passes make predictions, calculate error, and adjust weights to minimize said error. During the forward pass, the algorithm works from Input layers to Output layer, calculating values for each node in the network. After the output layer has produced predictions about what tweets belong to what class, the backward pass calculates total error $R(\theta)$ and how much each node from the last layer contributed to each output node's error. Let $z_{mi} = h(\alpha_m^T x_i)$ and let $z_i = (z_{1i}, z_{2i}, ..., z_{Mi})$. Then the loss function $R(\theta)$ takes the form:

$$R(\theta) \equiv \sum_{i=1}^{N} R_i \tag{4}$$

$$= \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2 \tag{5}$$

Where N is the number of observations and K is the number of classes. Then, the algorithm calculates how much each node from the previous layer contributed to the error of the last hidden layer, continuing layer-by-layer until it reaches the Input layer. This pass propagates the *error gradient* backward, computing the gradient across all connected weights in the network. The error gradient vector is composed of partial derivatives that capture how much each weight (β_{km} and α_{ml}) contribute to total error:

$$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g'_k(\beta_k^T z_i)z_{mi} \tag{6}$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} = -\sum_{k=1}^{K} 2(y_{ik} - f_k(x_i))g'_k(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i)x_{il} \tag{7}$$

Finally, the backward pass uses the error gradient to adjust the weights toward a minimum – this is called the gradient descent step. At (r+1) iteration the gradient descent updates the weights via the following expressions:

$$\beta_{km}^{(r+1)} = \beta_{km}^{r} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}} \tag{8}$$

$$\alpha_{km}^{(r+1)} = \alpha_{km}^{r} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{km}^{(r)}} \tag{9}$$

where γ_r is the learning rate – a tuning parameter that determines how "large" each gradient descent update should be – and α^r and β^r are the models' weights at the r^{th} iteration. We can rewrite equations 9 and 10 above to better understand how the algorithm propagates error backward:

$$\frac{\partial R_i}{\partial \beta_{km}} = \delta_{ki} z_{mi} \tag{10}$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} = s_{mi} x_{il} \tag{11}$$

The value of δ_{ki} is error from units in the output layer and s_{mi} is the error from units in the hidden layers. By definition, these values satisfy the following condition:

$$s_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki}. \tag{12}$$

To summarize: the forward pass computes predicted values $\hat{f}_k(x_i)$ using fixed weights at each iteration. The resulting values are compared to target values Y_k in the backward pass, and error from each unit in the output layer $\delta_k i$ are calculated. The error gradient is then back propogated to each unit in the previous layers. Weights are adjusted via gradient descent and the process starts again.

Because our model back-propagates across all trainable layers in the network, it learns on both text and image features simultaneously. We do not simply pool, or average the results of two separate models – via an ensemble learning approach. No, we use a single model with two inputs of different modes. The single model relies on back-propagation across all trainable layers in the network; the embeddings layer comes pre-trained, and thus, those weights are excluded from training.

2.3 Primary Libraries: Keras and TensorFlow

To build our neural network, we primarily relied on two libraries in Python 3: Keras and TensorFlow. Keras supports network construction by treating each layer as a function whose inputs and outputs rely on other layers. The user defines network architecture by connecting layers together during construction phase. After the construction phase, data gets input, the network is compiled and training begins. TensorFlow is the main engine doing all of the computing, it is the deep learning library developed by Google that has become a standard library for deep learning research. Keras provides the wrappers for different types of layers: convolutional, flattening, pooling, LSTM, and fully connected (dense). TensorFlow does the heavy lifting.

TensorFlow treats tensor objects as n-dimensional arrays, much like the arrays we see in other data science packages across Python3 like Numpy or Pandas. From the TensorFlow user guide: ”an object represents a partially defined computation that will eventually produce a value.” This is the construction phase: you design your network by defining different layers who will pass data forward across the network. ”TensorFlow programs work by first building a graph of `tf.Tensor` objects, detailing how each tensor is computed based on the other available tensors and then by running parts of this graph to achieve the desired results”.

Keras supports building deep network models using simple framework where each line of code becomes a layer, each layer becomes a function, and one can easily define inputs, outputs, and tuning parameters without having to design the behavior of each layer from scratch. Here, it is worth noting that a *tensor* is a generalization of a matrix, but to higher dimensions. Tensors are matrices that contain data, but also, functional relationships between data. For instance, in image analysis, tensors not only contain information about pixel values, they also hold information on how a pixel value relates to those around it. Not only do tensors contain data, they contain information about variance in the data.

2.4 Why is it difficult to train a machine on text and image simultaneously?

Fitting a model to text and image is difficult because of the differences in the shape, size, and techniques used to process each. Text can be stored in a 2-dimensional structure like a document term matrix, whereas each image comes in a 3-dimensional tensor. They are shaped differently, as data, and the types of models we work with generally require one shape or another.

Further, data entires in a document term matrix are not immediately comparable to data entries in an image tensor. In a document term matrix, each row is a document, each column a term. The entries in a 3D representation of an image represent *pixel values*, which individually, mean very little because we are interested in groups of pixel values, patterns in those groups.

Text and image vary in size. Text corpora consist of text documents, which are measured in kilobytes (if not, bytes). Images are an order of magnitude larger, and they are measured in megabytes. One cannot load a corpus of images into memory, like text documents, without fear of crashing a laptop. Working with images requires larger computers and more advanced techniques.

A lot of the advanced techniques involving image analysis are aimed at making images easier to deal with for the computer. Convolutions act as filters that learn the relevant features necessary to make classifications. These filters only retain certain pixels, downsizing the amount of data each image contains. Pooling reduces pixel count by keeping only the strongest pixel values in each square set of pixels, again downsizing the amount of data each image contains. Flattening reduces the dimensions of image matrices by one, reshaping image tensors into lower-dimensional matrices. Much of image processing is about reducing the image to its absolute barest features.

The use of graphics processing units (GPU's) is another domain where images require the more advanced computational technique. These computers are ideal for deep learning analysis of images because they perform the matrix-algebra required to perform convolutions, pooling, flattening much faster than the traditional central processing unit (CPU). Interestingly, we were able to build this model without GPU's, as we relied on a smaller dataset, we kept images small (only 28×28 pixels), and we did not pre-process the images all at once—they were loaded into memory in batches.

Within the machine learning framework, image analysis can have multiple goals, not all of which parallel that of text. Within the broad definition of "image analysis" we might be discussing object recognition and localization (where we are trying to find an object within the image), facial recognition, image classification, or even video analysis (where we treat videos like a series of images). Not to say that text analysis is any more or less broad, but you do not need text to classify faces, for instance. The substantive difference between the two forms lends themselves to different machine learning tasks.

In this paper, we are focused on classification: sorting tweets into classes based on whether or not they report election incidents.

2.5 How do the tweet-images get processed and analyzed?

Beginning with Krizhevsky, Sutskever and Hinton (2012), research has explored *convolutional neural networks* (or ConvNets) for image analysis tasks including facial detection, object and pattern recognition, and video classification. ConvNets are powerful, flexible, and scalable, making them ideal for analyzing high-dimensional data (e.g., images and video). This flexibility is both a blessing and a curse. ConvNet architectures can be configured in so many different ways that training these models is more art than science. Researchers benchmark their models using various visual recognition competitions, like the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC).

Think of a convolutional neural network as a school building: each floor is a layer and each student is a node. When students are grouped into classrooms, they work together as *filters* that slowly break images down into their core components. Classrooms on the lower floors learn basic shapes, lines, and edges, while higher levels leverage this knowledge to learn complex patterns. For this reason, convolutional neural networks tend to be very deep (10 layers or more) while we only include one convolutional layer, on the image side, to save lower computing costs.

Images are stored in directories, labeled “Hit” and “Not_Hit” with each class has having its own subfolder. Python takes each folder name as a label and any contents of said folder as belonging to that label. To qualify for the dataset, the tweet had to have both image and tweet text. Further, the image could only come as `.jpeg` or `.png`. Any files with `.gif` extensions were thrown out.

To avoid crashing the computer, we load image files in batches. Python3 loads images using a generator object, creating an “array of arrays.” We retrieve Numpy arrays of the pixel-level values that have been converted to grayscale and resized so all images are 28×28. Image pre-processing included random flips and crops. Each image was converted to grayscale. The use of only one color channel also minimized on computational cost of this basic development part of the project.[6]

Image arrays are fed into the Input Layer #2 where it becomes a Keras Tensor Object, which lets us build the model around the shape of the input and output data. Images undergo five-layers of processing on the “image side” before joining the “text side.” A convolutional layer first learns pixel-level patterns that prove relevant for classification purposes. The flattening layer, as it sounds, flattens the image from a 2D array of arrays, to a 1D array—this shape resembles how the text gets input. The dense layer is a fully connected neural network whose nodes all connect to nodes from the previous “Flatten” layer and the awaiting Dropout layer. Dropout layer randomly drops a certain percentage of waits; this prevents overfitting the model on the training set.

2.6 How does the tweet-text get processed and analyzed?

Tweet-text gets loaded as a `.csv` file and little pre-processing is done. There is no removal of stop words or punctuation, but all letters are lower-cased. The tweet-text gets Tokenized (turns each text into a sequence of integers and each integer indexes the token in a Python dictionary).

Input Layer #1 gets fed the tweet texts and instantiates a Keras Tensor Object. Remember that a Tensor is a big matrix that contains information on the shape and size of our data. The model expects two inputs of the same size—having the same number of samples.

[6]The bulk of code, training, and validation was done a Lenovo T450s with an Intel i5 processor.

Embeddings Layer Word embeddings are a natural language processing technique where words are cast into a geometric space, such that, distances between words capture how semantically similar words are to each other. The closer the words in space, the more similar they are to one another. Representing words as these global vectors means representing words by their semantic neighbors in geometric space.

The Embeddings Layer learns a GloVe embedding for each word in the training set. GloVe stands for: "Global Vectors for Word Representation." GloVe is an unsupervised learning algorithm for representing words as vectors in space. The algorithm learns global word-word co-occurrence statistics from Wikipedia, and the resulting vectors represent linear relations between words. That is, we define a word by the words that occur around it in geometric space, the words that are most closely defined. The location of that word in the space is referred to as its embedding.

LSTM Layer The LSTM Layer are a type of recurrent neural network where words in a sequence get treated like time series data and the model has a temporary window where it retains information from previous iterations of training. LSTM units are commonly applied to language modeling and sentiment analysis Raschka (2015).

LSTM units are a type of *reccurrent neural network* (or RNN) characterized by three weight matrices: W_{xh}, W_{hh}, and W_{hy}. Each matrix contains the parameters of the model, but the relationships capture by each weight matrix differ. W_{xh} is the relationship between Inputs x^t (inputs at iteration t) and the hidden layer h. W_{hh} contains relationships about the *recurrent edge* – how the hidden layers relate to themselves from one iteration to the next. Finally W_{hy} contains information about the relationships between the hidden layers h and the output layer y.

This followed by a pooling layer and a dense, fully connected layer. Both image and text tensors are then fed to a concatenate layer. The concatenate layer combines two different inputs, merging them together in a singly Tensor as output. Remember that a Tensor is a matrix that is being passed from one layer to the next across our network. The final three layers in the network are three fully connected, dense layers, with 32, 64, 64 output filters, respectively and all relying on a ReLU activation function.

Benchmarks for Training Training performance is measured using two benchmarks: accuracy and cross-entropy loss. Accuracy, defined formally in Equation 13 below, measures the percentage of images the model correctly classifies. We define y_i as the true value of subject *i's* label, and $\hat{y}_i$ as the model's predicted label. The higher the accuracy, the better the model.

$$\texttt{accuracy}(y, \hat{y}) = \frac{1}{n_s} \sum_{i=0}^{n_s-1} \mathbf{1}(\hat{y}_i = y_i) \tag{13}$$

Cross-entropy loss, defined in Equation 14 below, measures performance of binary classifiers on a scale from 0 to 1. As predicted probability p diverges from the true label, cross-entropy increases; loss closer to 1 corresponds with poor performance, while a perfectly performing model would have a loss of 0.

$$\texttt{loss}(y, p) = -(y \log(p) + (1 - y) \log(1 - p)) \tag{14}$$

In above equation, we define p as the predicted probability and y as a binary indicator (0 or 1) if the model correctly predicts a subject's class.

3 Results

Accuracy about here

Loss about here

4 Discussion

Transfer learning for Image side. Currently, we leverage word embeddings trained on Wikipedia entries to boost training on the text side. Ideally, we'd have a pre-trained convolutional neural network working on the image side in order to boost training on the image side.

More Data. Currently, our goal was to get the model to train on text and image data, no small feat, as no single machine learning framework had ever done that. Now, we need to improve the model. At present, we have a sample size of 1289 tweets, text and image for each. Only 900 are being used for training at each iteration, and 300 are being used

Simplify the Model. Sometimes, to improve performance of a deep learning model that is training on limited data, you need a simpler model with fewer parameters. The more parameters you have, the more data you need to train. As social scientists, the use of simpler models and leveraging pre-trained models makes sense as we explore deep learning methods, as it prevents us from suffering costs of buying big computers, spending hours tuning parameters and

References

Abraham, Linus and Osei Appiah. 2006. "Framing news stories: The role of visual imagery in priming racial stereotypes." *The Howard Journal of Communications* 17(3):183–203.

Barberá, Pablo. 2015. "Birds of the same feather tweet together: Bayesian ideal point estimation using Twitter data." *Political Analysis* 23(1):76–91.

Casas, Andreu and Nora Webb Williams. 2019. "Images that matter: Online protests and the mobilizing role of pictures." *Political Research Quarterly* 72(2):360–375.

Hopkins, Daniel J and Gary King. 2010. "A method of automated nonparametric content analysis for social science." *American Journal of Political Science* 54(1):229–247.

Iyer, Aarti and Julian Oldmeadow. 2006. "Picture this: Emotional and political responses to photographs of the Kenneth Bigley kidnapping." *European Journal of Social Psychology* 36(5):635–647.

Joo, Jungseock and Zachary C Steinert-Threlkeld. 2018. "Image as data: Automated visual content analysis for political science." *arXiv preprint arXiv:1810.01544* .

Krizhevsky, Alex, Ilya Sutskever and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. pp. 1097–1105.

Lan, Man, Chew-Lim Tan, Hwee-Boon Low and Sam-Yuan Sung. 2005. A Comprehensive Comparative Study on Term Weighting Schemes for Text Categorization with Support Vector Machines. In *Special interest tracks and posters of the 14th international conference on World Wide Web*. ACM pp. 1032–1033.

Leopold, Edda and Jörg Kindermann. 2002. "Text Categorization with Support Vector Machines. How to Represent Texts in Input Space?" *Machine Learning* 46:423–444.

Lin, Wei-Hao and Alexander Hauptmann. 2002. News video classification using SVM-based multimodal classifiers and combination strategies. In *Proceedings of the tenth ACM international conference on Multimedia*. ACM pp. 323–326.

Mebane, Jr., Walter R., Alejandro Pineda, Logan Woods, Joseph Klaver, Patrick Wu and Blake

Miller. 2017. "Using Twitter to Observe Election Incidents in the United States." Paper presented at the 2017 Annual Meeting of the Midwest Political Science Association, Chicago, April 6–9, 2017.

Mebane, Jr., Walter R., Patrick Wu, Logan Woods, Joseph Klaver, Alejandro Pineda and Blake Miller. 2018. "Observing Election Incidents in the United States via Twitter: Does Who Observes Matter?" Paper presented at the 2018 Annual Meeting of the Midwest Political Science Association, Chicago, April 5–8, 2018.

Mendelberg, Tali. 2017. *The race card: Campaign strategy, implicit messages, and the norm of equality*. Princeton University Press.

Miller, Blake, Fridolin Linder and Walter R. Mebane, Jr. 2019. "Active Learning Approaches for Labeling Text: Review and Assessment of the Performance of Active Learning Approaches." *Political Analysis* ??:conditionally accepted.

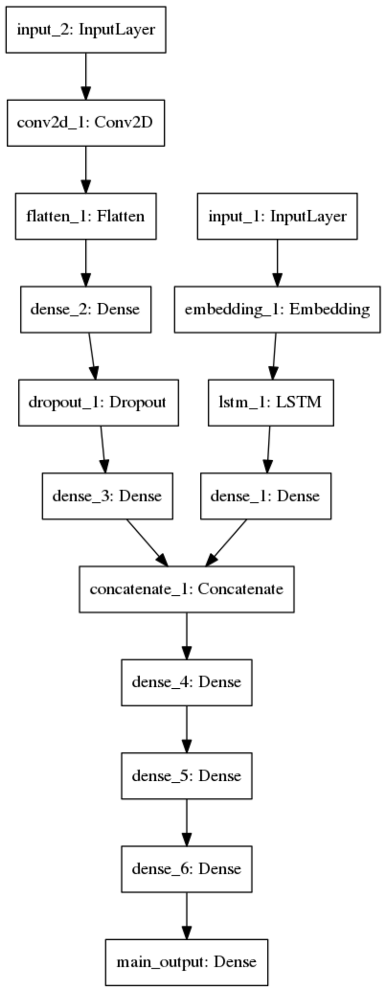
Rakotomamonjy, Alain. 2003. "Variable Selection Using SVM-based Criteria." *Journal of Machine Learning Research* 3(Mar):1357–1370.

Raschka, Sebastian. 2015. *Python machine learning*. Packt Publishing Ltd.

Stewart, Charles. 2017. "2016 Survey of the Performance of American Elections." Harvard Dataverse, V1, UNF:6:/Mol52fZ59fx6OsPWIRsWw== [fileUNF]. URL: `https://doi.org/10.7910/DVN/Y38VIQ`.

Yu, Jun, Jing Li, Zhou Yu and Qingming Huang. 2019. "Multimodal Transformer with Multi-View Visual Representation for Image Captioning." *arXiv preprint arXiv:1905.07841* .

VOTE
EARLY
HERE


input_2: InputLayer
conv2d_1: Conv2D
flatten_1: Flatten
input_1: InputLayer
dense_2: Dense
embedding_1: Embedding
dropout_1: Dropout
lstm_1: LSTM
dense_3: Dense
dense_1: Dense
concatenate_1: Concatenate
dense_4: Dense
dense_5: Dense
dense_6: Dense
main_output: Dense