Job Posting Fraud Detection

By Scott Salisbury, for CIS3715

Abstract

I built a non-sequential neural network using Keras to distinguish between legitimate and fraudulent online job postings. I trained it on an imbalanced, labelled dataset of online job postings which had 17 attributes for each posting. Text attributes which had manageable numbers of different values were treated as categorical variables and one hot encoded, while each of the remaining text attributes was embedded and then analyzed by a specialized LSTM. The LSTMs' outputs and the remaining attributes of a posting were then fed into a deep feed-forward neural network. This model had AUROC scores of 97.6% on the validation set and 98.1% on the test set, while a simple Naïve Bayes predictor using tf-idf matrices of the postings' text content only managed AUROC scores of 86.8% on the validation set and 85.4% on the test set. The model also got F1 scores of 0.583 on the validation set and 0.553 on the test set while the Naïve Bayes baseline predictor got abysmal F1 scores of 0.015 on the validation set and 0 on the test set.

Introduction

The dataset is here: https://www.kaggle.com/shivamb/real-or-fake-fake-jobposting-prediction

I'm very interested in designing machine learning models which can approximate intelligence to improve human life, and this problem of identifying fraudulent cases is related to problems I might work on in the future (in particular, my first job is at a bank).

The 17.8k example dataset was structured into 16 different attributes plus the label. 3 attributes were Boolean, 1 was a numeric range, 6 were text attributes with manageable numbers of distinct values (6-132), and 7 were other text attributes. Also, many attributes had a lot of null values and the longer text attributes had a fair bit of noise.

It seemed to me that fraudulent postings would probably have different sorts of common patterns in different text attributes (e.g. company profiles vs requirements lists or job descriptions). Therefore, I thought that it might be more effective to train a different LSTM for each text field of the job posting.

Approach

Data Preprocessing (16 hours)

First, I spent roughly 3 hours over several days researching Keras, LSTM neural networks, and text preprocessing for machine learning.

I then wrote a Python script to do basic preprocessing of the dataset and to reformat/summarize it. This took around 7 hours in total.

Each job posting example was preprocessed as it was loaded from the raw data file. For each categorical variable, I accumulated a list of the different text values for that variable and stored an index for that list in the processed version of the example rather than the actual text value. I also cleaned the longer text attributes- I eliminated stopwords, cut non-alphanumeric characters, converted everything to lower case, replaced URL's/phone numbers/email addresses with generic 'URL'/'phone'/'email' tokens, and adapted some regex operations which I found in an article (Ack. #4) to prepare the text to be tokenized in a sensible way. I also aggregated, for each text attribute, all of the text values for that attribute in the dataset (which would determine the vocabulary of that attribute's text vectorizer).

Commented [SSS1]: Oops. This means it would probably crash (or at least generalize badly to) new data because it doesn't even have logic to handle novel values of these categorical variables (and so necessarily wasn't trained to consider the 'novel' category in its prediction). I should have excluded the test set (and probably the dev set) from the categories-compiling process.

Commented [SSS2]: Oops again. This means it would probably crash (or at least generalize badly to) new data because it doesn't even have logic to handle out-of-vocabulary options (and so necessarily wasn't trained to consider the prevalence of OOV tokens in its prediction). I should have excluded the test set (and probably the dev set) from the vocabulary-building process.

The salary attribute, which was either null, a single number, or a range specified by min and max numbers, was also broken down/analyzed. I stored minimum and maximum values in variables and then made two more variables, one for the difference between the min/max values and another for the midpoint between the min/max values. All four of those numbers were included as attributes in the processed example.

I then realized that there was a somewhat significant issue in the dataset- a lot of the text contained words that had been spliced together (like "themCheerful" or "attributesClient"). I guessed that the web-scraping tool of the dataset creators might not have handled line breaks correctly in some cases. As it was, this reduced the amount of information that the LSTM's would be able to glean from the text because two spliced-together words would register as one unique (and meaningless) token rather than as two meaningful tokens which might show up in multiple text strings or be in the vocabulary of pretrained word embeddings.

I spent 6 hours adding logic to find and 'unsplice' these tokens. I used a spell-checking library to find tokens in a text string which didn't register as words. Then, for each of those tokens, I iterated over its characters, seeing whether splitting the token at a certain character produced two new tokens which were both recognized as words by the spell checker. A significant portion of the work for this feature was in reviewing the token-splits which that algorithm made & catching cases where it split a real word which it didn't recognize. To mostly fix that, I went over half of the token-splits made by the algorithm in the dataset and added any tokens which shouldn't have been split to a 'personal word list' for the spell checker. I only went over half of them because I decided at that point that the task was too time-consuming to complete and wasn't catching that many incorrect splits (~1100 in that first half of the splits). This ultimately corrected a little over 50,000 spliced-together word pairs.

Once the dataset had been processed, I wrote a simple script to split it into training, validation, and test subsets and save those subsets to files.

EDA (2 hours)

I reviewed anomalies in the text data during the latter part of the preprocessing work, but then I did somewhat more typical exploratory data analysis before starting on the neural network which took in text attributes. For each text attribute, I combined all of the string values of that attribute in the processed dataset and counted the number of distinct words which showed up. I then measured how long (i.e. number of words) each string value of that attribute was.

The number of distinct words was important for deciding on the vocabulary size of the text vectorizer (which converts a sequence of text tokens into a sequence of indices for a vocabulary).

More importantly, the string lengths (i.e. number of words) informed my decision about the length of input which the LSTM for that attribute would accept. (Since, for some attribute, the LSTM has to take the same-length sequence with each example, all values of that attribute have to be 'padded' or truncated to some chosen length.) I wanted to find a sequence length which would minimize chopping off the ends of some string values of the attribute (since that would abandon potentially-useful data), but I also didn't want to make the sequence length so long that most examples would have a huge amount of padding and slow the whole process down.

The NumPy library was extremely useful for the latter task, straightforwardly finding the 50^{th}, 60^{th}, 70^{th}, 80^{th}, 90^{th}, and even 95^{th} and 98^{th} percentile values in a list of word-lengths.

AI Model Development (12.5 hours)

I started by writing a function for each text attribute that would take cleaned string values of that attribute, convert them into sequences of numbers using a vectorizer, and pad those sequences to the standard length for that attribute. I then built a simple neural network model which fed the 'job

description' vectorized text sequences through an Embedding layer into an LSTM layer, whose output was converted into a fraud prediction by a single node with a sigmoid activation. This took 3 +/-1 hours.

After getting that single-input/single-output model working, I added additional inputs for the Boolean and categorical attributes. I then fed those values, along with the job-description-LSTM's output, into a feed-forward neural network which culminated in a second output. This part proved quite difficult, with several tricky problems around the numbers of dimensions of input arrays (specifically the labels) and of tensors in the model design. The labels vector had to be duplicated for each of the model's outputs. Meanwhile, the tensors problem involved getting the outputs of Embedding layers, the output of an LSTM layer, and the Boolean inputs to have the same number of dimensions and to have those dimensions ordered in such a way that they could all feed into a Merge (Concatenate) layer together. Getting all of this working took 3 +/-1 hours as well.

Commented [SSS3]: Embedding layers which I was using for the categorical variables. Those embedding layers were later eliminated in favor of simply one-hot encoding those categorical variables.

At that point, I realized that I was suffering from horrendous levels of overfitting- the training loss would quickly drop to near-zero while the validation loss did not decrease anywhere near as much and would often increase instead. I experimented with the severity of dropout layers throughout the model without much success. However, after some research, I realized that I needed to add regularization, increasing the loss function based on the size of the weights in the various layers. Adding that in alongside less severe settings for the Dropout layers permanently eliminated severe overfitting as an issue. This research and experimentation took 1-2 hours.

When I tried to add Input/Embedding/LSTM layers for the other text attributes, I ran into trouble, because all of the text attributes except for 'job description' had some null values. Those resulted in the LSTM receiving an embedded sequence which was entirely masked out, and it didn't handle that well at all. I solved this by adding an arbitrary 'start token' to the beginning of every value for every text attribute, so that only non-empty text attribute values were fed into LSTM's. Implementing and debugging this took 2 or 3 hours.

Commented [SSS4]: Tensorflow crashed.

Finally, the salary data was normalized and fed into the network as another input. I also added class weights so that training would focus more on the underrepresented positive (fraudulent) class. This part took about an hour.

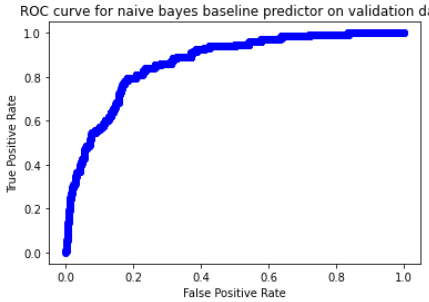
Also, to provide a baseline to compare the main model to, I set up a Naïve Bayes predictor for the same task. I wrote a function to, for each job posting in a dataset, combine all information about that posting (except the salary) into a single string. Then, I fed those strings through a tf-idf vectorizer and into a Complement Naïve Bayes classifier. I chose that Naïve Bayes variant because the dataset was imbalanced. This took an hour and a half.

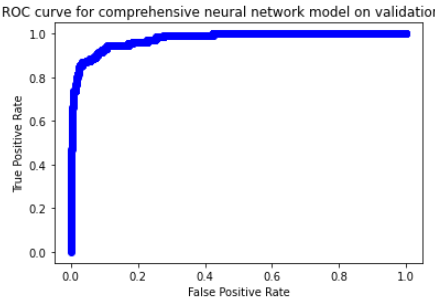
Results

The project is hosted in a GitHub repository:
https://github.com/BareBeaverBat/JobPostingFraudDetection

The deep neural network model markedly outperformed my baseline model in terms of metrics like AUROC and F1 score which measure performance on imbalanced classification tasks. See in the below ROC graphs how the neural network's curve rises very steeply, so that it might be possible to tune it to get very high true positive rates (~80%) with very low false positive rates (<5%). By contrast, the baseline model's curve rises relatively slowly, and so it has to accept a large proportion of false positives to achieve a high true positive rate.

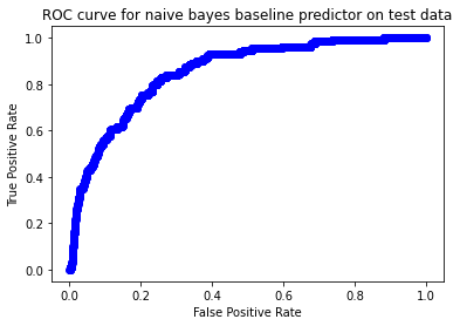
Validation Set:

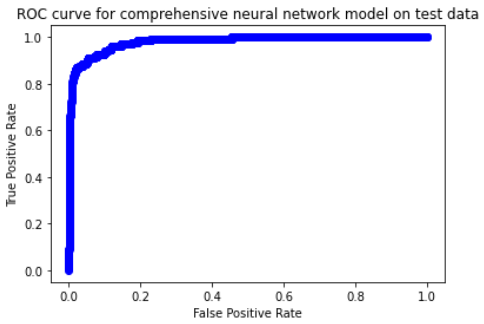




accuracy= 0.9519015659955258	accuracy= 0.9388516032811335
balanced accuracy = 0.5038461538461538	balanced accuracy = 0.9131149023390402
F1 score= 0.015267175572519085	F1 score= 0.5837563451776651
AUROC score= 0.8675608873884736	AUROC score= 0.9758258982396913
Precision= 1.0	Precision= 0.4356060606060606
Recall= 0.007692307692307693	Recall= 0.8846153846153846
ROC curve's (model output) thresholds:	ROC curve's (model output) thresholds:
(0^{th}, 10^{th}, 20^{th}…80^{th}, 90^{th}, 100^{th} percentiles):	(0^{th}, 10^{th}, 20^{th}…80^{th}, 90^{th}, 100^{th} percentiles):
4.47900961e-13 1.53573915e-09	0.01361483 0.01366846
5.70305561e-09 1.89709110e-08	0.01369501 0.01377448
6.32686222e-08 2.14758444e-07	0.01394862 0.01432123
6.66949125e-07 2.44429604e-06	0.0392849 0.06891177
1.13225455e-05 9.25950487e-05	0.15600666 0.49888635
1.96347132e+00	1.99976635

Test Set:





accuracy= 0.9511740588893031	accuracy= 0.9288110324263884
balanced accuracy = 0.49980415197806505	balanced accuracy = 0.9187893579197928
F1 score= 0.0	F1 score= 0.5526932084309133
AUROC score= 0.8543252282382717	AUROC score= 0.9811021724065202
Precision= 0.0	Precision= 0.39730639730639733
Recall= 0.0	Recall= 0.9076923076923077
ROC curve's (model output) thresholds:	ROC curve's (model output) thresholds:
(0^{th}, 10^{th}, 20^{th}…80^{th}, 90^{th}, 100^{th} percentiles):	(0^{th}, 10^{th}, 20^{th}…80^{th}, 90^{th}, 100^{th} percentiles):

3.42491749e-12	1.36253778e-09	0.01361693	0.01367211
5.95723783e-09	1.98047828e-08	0.01370592	0.01379958
7.61189590e-08	2.54045518e-07	0.01399435	0.01460198
7.91237346e-07	2.53710480e-06	0.04106319	0.06721462
1.09542463e-05	1.03022828e-04	0.16149975	0.56710256
1.86595742e+00		1.99967039	

There was a massive disparity in recall between the two models. On the validation set, the baseline model only correctly identified 7.7% of the fraudulent postings, while the neural network was able to identify 88.5% of them. While the neural network also had a lot of false positives (56.4% of positive predictions were false), that doesn't necessarily matter quite as much in an imbalanced classification task like this search for con artists on job boards. When the positive class is such a tiny minority of the whole (~1/20), it can be quite valuable to narrow things down to a set of predicted positives because the fraction of those predicted positives which are actually positive is far greater than the fraction of the whole population that're actually positive (even if the true positives are still a minority of the predicted positives, 1/2.3 is 8.7 times better than 1/20).

On the test set, that trend was even more pronounced. The neural network happened to have a slightly higher recall score (90.8%) on that than on the validation set (and only slightly worse precision, with 60.3% of the positive predictions being false). Meanwhile, on the test set the baseline model failed to make any positive predictions at all and so for the test set it had a recall score of 0.

Conclusion

This was a really interesting learning experience. It was incredible how many highly-applicable resources there were for different parts of the project (see Acknowledgments below for the most important ones). I developed a working familiarity with and at least some real understanding of important/intriguing topics like long short-term memory networks, the importance of word embeddings for natural language processing, and neural network normalization techniques like dropout. Ultimately, I was able to build a relatively sophisticated predictor which performed better and generalized far better than a straightforward machine-learning model.

A major takeaway for me is that it's important to double-check the behavior of data-transformation logic. I had to go back and modify my data processing/cleaning code 5-10 times after reviewing the output and finding unintended/problematic results in the cleaned text. Regex patterns were especially tricky.

I also experienced first-hand how improving model design/scope can only do so much with a limited amount of data. Past a certain point, even dramatic increases in the size/depth of the model made little difference in how good the model could get.

Acknowledgements

TensorFlow 2.1/Keras/NumPy/Pandas/Scikit-learn documentation

1. https://towardsdatascience.com/multi-class-text-classification-with-lstm-1590bee1bd17
2. https://towardsdatascience.com/multi-class-text-classification-with-lstm-using-tensorflow-2-0-d88627c10a35
3. https://www.kaggle.com/lystdo/lstm-with-word2vec-embeddings
4. https://datascience.stackexchange.com/a/11421
5. https://www.kdnuggets.com/2019/04/text-preprocessing-nlp-machine-learning.html
6. https://machinelearningmastery.com/keras-functional-api-deep-learning/
7. https://www.mdpi.com/1999-5903/9/1/6 (paper by the creators of the dataset)
8. https://machinelearningmastery.com/reshape-input-data-long-short-term-memory-networks-keras/