WhosInMyDMs - Final Report

Alexandria Salib (as12453), Nabeel Sarwar (ns3429), Ziv Schwartz (zs1349)

December 8, 2018

Contents

1 Abstract

For our assignment, we analyzed SMS text messages to classify them as 'spam' or 'ham'. As online communication has adapted and shifted from email to various forms of direct messaging, phishers have adjusted where and how they target individuals with spam. Users want to know that their accounts and data are secure, and they do not have time to be bothered by receiving spam messages. We looked to create a classification model, using the following algorithms: Logistic Regression, Random Forest, and an LSTM Neural Network. For each of these models, we performed feature extraction using sparse vectoratization techniques, CountVectorizer and TfidfVectorizer, as well as utilizing bigrams, n-grams and word embeddings to test against a dense vector representation. After performing an ablation study on the models and tuning each with the optimal hyperparameters via cross validation, we compared a set of performance metrics: AUC, precision, recall, and F1 score to choose the best model. Within our paper, we further discuss the business value of counteracting spam and malicious messages through classification, the data sources and preprocessing, and explaining which model and feature extraction method performs best. Following this we construct a proposal of how to deploy our solution into production.

2 Business Understanding

We will look to classify text message data as legitimate, referred to as 'ham' messages, or spam messages. This is a highly relevant, significant, and topical business problem for a variety of telecommunications and messaging service companies, ranging from Verizon to Facebook. The companies' abilities to detect spam result in increased business value by providing high quality service and being considered a top service provider in this field, therefore, decreasing user churn. As technology constantly improves, phishing attempts improve, and anti-phishers must likewise adapt and improve in order to keep their messaging systems secure and users' data private. Phishing was originally an issue with emails, but has expanded to direct messaging platforms as individuals shift their primary mean of communication. As messaging systems begin to contain more and more sensitive user data, it is important to keep users' accounts and data secure from phishing attempts. If a user cannot trust that their messaging account and its corresponding personal data are safe, then they will be more likely to churn. Our business solution is to build a model to classify each message as ham or spam. Therefore, our use scenario is to alert users when a message is potentially spam, which depending on the recall of the model, can either be a simple notification below the spam message, or filtering these messages into another section of the messaging app entirely. This will be readdressed during the deployment section of this analysis.

3 Data Understanding

The data for this analysis was obtained from the UCI Machine Learning Repository. SMS Spam Collection Data Set[1] is a public ensemble of four datasets which include:

- Grumbletext, the public UK forum where users report spam SMS messages
- Caroline Tagg's PhD thesis at The University of Birmingham, A Corpus Linguistics Study of SMS Text Messaging
- The NUS SMS Corpus (NSC), a corpus of legitimate messages collected for research at the Department of Computer Science at the National University of Singapore
- SMS Spam Corpus created by Jose Maria Gomez Hidalgo

Within the collection of the 5,574 SMS messages, all documents in the corpus are labeled ham or spam. We trust that these have been correctly classified and did not verify them ourselves, therefore, original data misclassification may have occurred. To begin our data mining process, we visualized the data for a preliminary understanding and to find patterns in the differences between the ham and spam documents. As seen in Figure 1 below, ham makes up 86.6% and spam makes up 13.4% of the entire dataset. Within the spam subset of data, the word count distribution is right skewed and within the ham subset of data, the word count distribution is left skewed. It is clear that there is a difference in text data structure between the two classes.

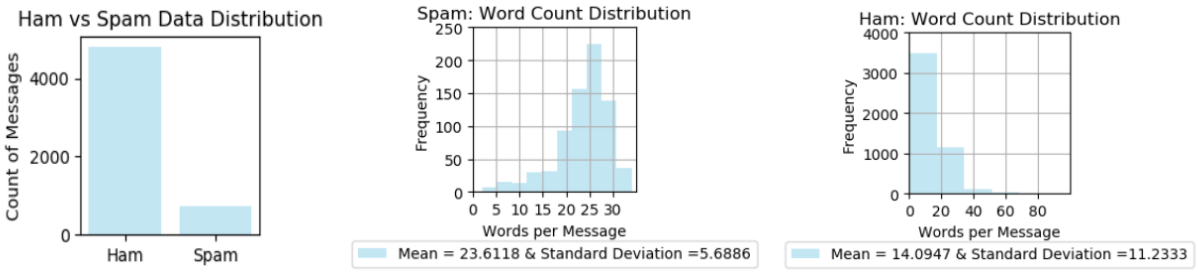

Figure 1: The plots above show the division between Ham and Spam SMS messages and the distribution of their word counts

Shown below are some of the most frequent words in the ham and spam subsets, after removing stop words:

- Ham: Dont, ill, got, know, like, come, good, ur, time, day, love, going, want, home, need, sorry, later
- Spam: Free, txt, mobile, claim, reply, prize, won, send, new, urgent, cash, win, contact, service, customer, nokia, phone, week, tone

Along with the structure of the documents, the content is also different and no top words overlap between subsets. These differences between classes lead us to believe that our baseline model will perform well.

[1] For a reference to the dataset, please see this link: http://dcomp.sor.ufscar.br/talmeida/smsspamcollection/

4 Data Preparation

After the data collection, our data was passed through a series of preprocessing techniques to prepare it for analysis, model training, and finally, model selection. Initially, each SMS text string was passed through a regular expression to remove any punctuation and set all text to lowercase. Following this, we utilized sklearn's 'preprocessing.LabelEncoder' to normalize the data labels across all of the samples in our set. We then split the data into training and testing sets, with 80% of documents used for training and 20% for testing. The target variable remained the SMS text message data label, ham or spam, since our goal is to predict this classification. We set a seed so that each algorithm used the same partitions of training and testing data.

4.1 Feature Engineering

After our basic data preparation, we leveraged multiple tokenization and feature extraction, or vectorization, methods to pass our data through for each model. Tokenization and vectorization methods were used because these processes are necessary to transform text data into a format to which we could effectively apply predictive and machine learning models. Each algorithm was run with different vector representations of our features, including: default CountVectorizer and TfidfVectorizer, bigram CountVectorizer and TfidfVectorizer, n-gram CountVectorizer and TfidfVectorizer, and custom word embeddings. These transform the text data into a sparse matrix that lists all corpus documents as rows and each word from the corpus as a column. The resulting vectors show the count of each word in that specified document. We chose the above vectorization techniques, as they offer different levels of complexity, with CountVectorizer as the most simplistic. TfidfVectorizer is more complex because it provides a weighting methodology to certain words based on frequency along with inverse document frequency. Within each feature extraction method, different parameters can be set. Along with removing stop words for all, we expanded upon single gram tokens and considered both bigrams and n-grams, where bigrams containing 2 co-occurring word tokens and n-grams consider co-occurring words of varying length (n). We fit out feature engineering pipeline only to the training data.

4.2 Custom Embeddings

For our most complex feature engineering, we formed our own word embeddings built upon our data, based on the classical n-gram model of Bengio, et al [Bengio et al., 2003][2]. In word embeddings like Word2Vec or

[2] To see where we got idea to use n-gram model to generate embeddings, see: https://pytorch.org/tutorials/beginner/nlp/word_embeddings_tutorial.html

ours, instead of having a word or sentence being represented by a sparse representation where the index of a feature corresponds to a particular word, each word has a dense representation in some embedding space.

This works due to being given a multilayer perceptron with a certain set of words (numbers of words being the context), one could predict the probability of the next word. The inputs are the concatenated sparse representations. The output is a probability for each word. In the hidden layer lie the 'representations' of these words. The size of the embedding for each word should be the same, so the hidden layer has size Context_Size $\times$ Embedding_Size.

This is essentially the methodology behind how we formed our own embeddings. Context size and embedding size were both chosen by heuristics, with the context size of one to require less computational power and the embedding size of 300 was selected iteratively until training the embeddings took less than 12 hours.

The embeddings were not trained on the whole dataset, but rather trained on the same training set as mentioned above, so it does not use the whole dictionary in order to mimic production. In addition, to prevent over fitting and to reduce noise, we required the bigrams to be present in at least two documents of the training set. As is common in multiple class predictions, we used a softmax layer with a negative log-likelihood for the last layer. The hidden layer after the embedding layer uses the ReLU activation function with 128 neurons.

We trained for 20 epochs of the upsampled training set until log-loss seemed to converge, and training did not take too long. The dataset itself is not very large, but on our computational equipment, training takes a bit long so exploration of hyperparameters was manual and very driven by heuristics and intuition.

To use the embeddings in the logistic regression and and random forest models, we summed the embeddings of all the words in an instance. For the recurrent neural network model, we fed in the embeddings sequentially. Given the performance of the models, it seems the predictive power of the embeddings for the target word is slightly overfitted.

5 Modelling and Evaluation

To address the SMS spam classification data mining problem, our study focused on testing different types of algorithms, including logistic regression, random forests, and neural networks. These models were chosen because they vary in complexity, baseline structure, and interpretability, which will all factor in to choosing our best model. Model evaluation will also be determined by the highest AUC, precision, recall, and F1 scores. We began with comparing the AUC values for all models and since they were very similar, we decided to consider precision, recall, and F1 score.

All of the potential model (except the neural network model), feature extraction, and hyperparameter combinations were run with K-fold cross validation, where K=5. K-fold cross validation is a way to measure the learning ability of any predictive or machine learning model[3]. Using this resampling procedure was especially important in this analysis, as our set of data is not very large, with only 5,572 documents. Additionally, since the balance between spam and ham classification is skewed with only 13.4% of the documents classified as spam, we upsampled this subset of data within the training folds of the K-fold cross validation and made sure to leave the validation fold of the iteration with the **original class ratio**, to prevent data leakage. Together, K-fold cross validation and upsampling reduced selection bias. Please see Table 1 for a summary of results.

5.1 Logistic Regression

The initial model exploration focused on utilizing the **Logistic Regression** algorithm with the different vector representations, previously mentioned in Sections 4.1 and 4.2, Feature Engineering and Custom Embeddings. The **baseline model** is the Logistic Regression model with CountVectorizer as the sparse vector representation for our features, as this is the most simplistic. The baseline model resulted in an AUC of .9886, 99.21% precision, 86.21% recall, and an F1 score of 0.9295. After this initial model exploration we implemented cross validation and tuned the hyperparameters to optimize each model.

To determine the best hyperparameters, we focused in on optimizing penalty, either L1 or L2, and the C-value, either 1, 5, or 10. For each of these preprocessing techniques, hyperparameters were tuned unique to each technique in order to maximize AUC. Below are the performance of all models:

The best model is the n-gram with TfidfVectorizer. This follows cross validation which optimized on the hyperparameters, penalty=L2 and C=10. The penalty refers to L2 regularization, which adds a penalty to the squared coefficients, forcing coefficients to be relatively small (but not zero). The C parameter refers to the strength of the inverse regularization, meaning that higher values of C, such as 10, allows Logistic Regression more freedom. This model resulted in an AUC of 0.9907, 99.32% precision, 87.35% recall, an F1 score of 0.9295. Compared to the baseline model, this one does slightly worse in terms of AUC but by a very small difference. However, it is important to highlight that this a minute difference and that the previous model had not undergone cross validation. It is not surprising that this model, as well as our second highest performing model, utilize the TfidfVectorizer, which is consistent with our intuition that these parameters and techniques would perform well at classifying the data due to their level of complexity.

[3] The only algorithms that did not use cross validation were the custom word embeddings and LSTM classifier. This is because of computational complexity and available computing power. Hyperparameters were chosen through iterative exploration until convergence and acceptable performance on validation set

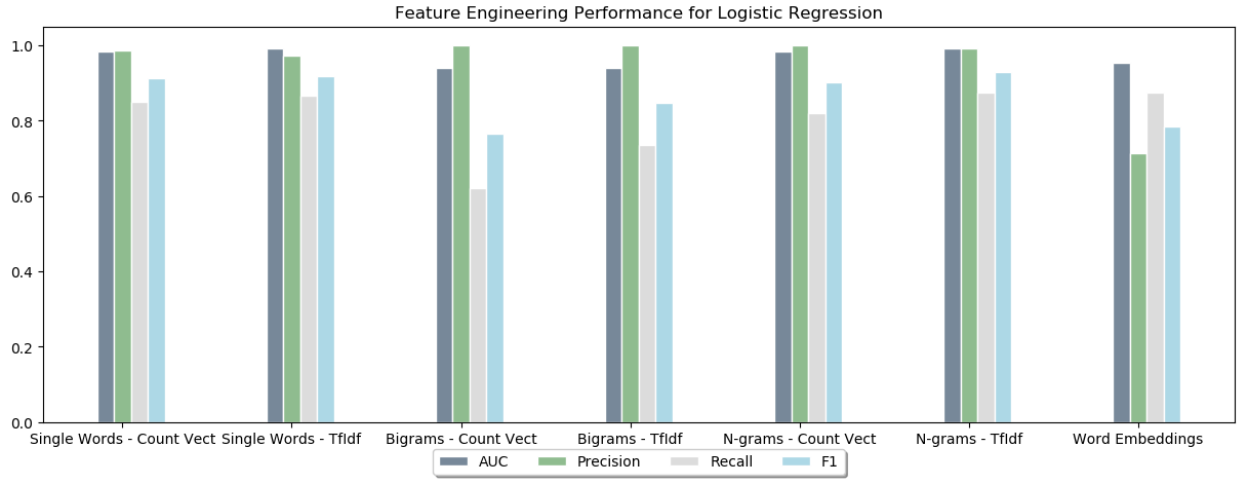

Figure 2: Logistic Regression Performance by Vectorization Method

5.2 Random Forest

Next, we explored the ensemble method, **Random Forest**, to see how a more complex algorithm handled this classification task. Similarly to the Logistic Regression process, we fit the Random Forest using all defined vector representations of our features and K-fold cross validation. For Random Forest, we tuned the following hyperparameters: n_estimators: 10, 100, max_depth: None, 1, 3, 7, 10, 50, min_samples_split: 2, 10, 50, 100, 250, 500, 750, 1000, min_samples_leaf: 1, 10, 25, 50, 75, 100, 125, 150. This range of hyperparameters was chosen based on default values, as well as what we thought would cover a significant range of options for the size of the data. We chose the combination of potential model types and hyperparameters that maximized AUC on the validation fold. The best combination of all the models is listed below:

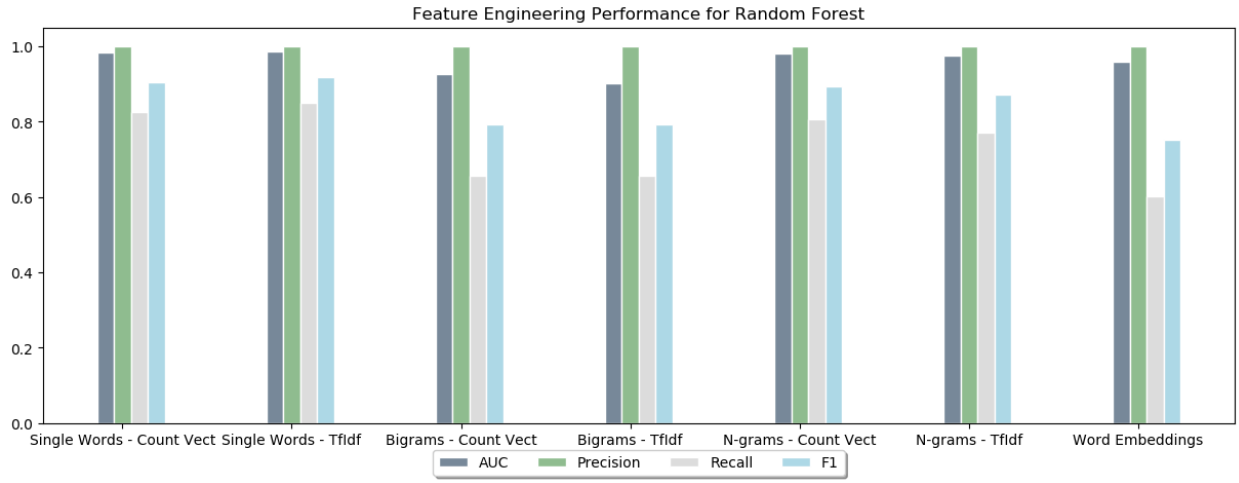

Figure 3: Random Forest Performance by Vectorization Method

The best model was the single gram TfidfVectorizater, with the following hyperparameters: n_estimators=10, max_depth=None, min_samples_split=10, min_samples_leaf=1, and resulted in an AUC of 0.9853, 100% precision, 84.94% recall, an F1 score of 0.9186. N_estimators refers to the number of trees in the forest, with higher values improving performance but heavily increasing computation time. In our case, the default value, 10, performs the best. Max_depth is the maximum depth of the tree, here 'None' refers to that until all leaves contain less than min_samples_split samples, the nodes are expanded until all leaves are pure (homogeneous labels). Min_samples_split and min_samples_leaf correspond to the number of samples required to split an internal node and the minimum number of samples required to be at a leaf node. As leaf size increases, the Random Forest model is less prone to capturing noise within the training data, which helps to explain why our best Random Forest model uses a value of 1 for min_samples_leaf.

Compared to the best performing Logistic Regression model, this one does slightly worse in terms of AUC, but by a very small difference. This model does have perfect precision, but lower recall and thus an overall lower F1 score. Similar to the Logistic Regression model, this model utilizes TF-IDF Vectorization, but with single gram tokens as opposed to n-grams.

5.3 LSTM Classifier

Language is an inherently sequential process (and sometimes the meaning propagates in two directions, though we do not consider this). Due to the sequential property, it might be useful to consider a model that explicitly shows this behavior; thus, we chose to develop a recurrent neural network model. Neural Networks are models that have layers of 'neurons' that carry their meaning from layer to layer via matrix multiplication and additive bias. They become universal approximators by making the values of the next layer go through non-linear activation functions. Recurrent neural networks also have layers that depend on the activation of not just previous layers in the current time step but also from previous timesteps.

One of the more popular recurrent neural networks are neural networks that incorporate a Long Short-Term Memory (LSTM) cell. We will call this architecture our **LSTM classifier** in short. LSTM cells have gates that allow them to remember and forget information from previous inputs. This includes a input gate, hidden state for the memory, a forgetting gate, and an output gate. [4] [5]

The LSTM model uses the following typical properties:

- Variable length sequences
- Gradient Clipping item Adam Optimizer with L2 Weight Decay
- Batch size of 10

[4] Also read this code as a tutorial: https://github.com/jiangqy/LSTM-Classification-Pytorch.We extended and modified the base classifier code extensively

[5] We used Pytorch [Paszke et al., 2017] to develop this model.

- Hidden layer of 5 neurons
- Trained on imbalanced data unlike previous models and uses class weights instead (balanced-class LSTM was not learning on either training or validation set)
- Fed the fixed custom word embeddings as inputs one by one (fixed to make the comparison fair to other models)

We use only the last layer to generate the prediction and throw away all the intermediary outputs of the LSTM sequence. Our training and testing sets were somewhat smaller than the training and testing sets used for all the other models because these sets divisible by the batch size (and we did not want to repeat particular inputs during each epoch because overfitting was already an issue).

Again, because of computational restraints, we did not grid search on the above parameters, but we did manually tweak parameters until the log-loss curve for the training and testing sets across the epochs converged to reasonable performance across re-runs. Since this dataset is relatively small, the LSTM classifier has a tendency to overfit. Normally, it is suggested to not use weight decay for RNNs, but since we overfit very quickly, this consideration did not apply to us. Regardless, even though the LSTM classifier performed among the best for the training set, it did not generalize to the test data at all, likely due to the limited amount of data and the possibly overfitted custom embeddings (as mentioned before in Section 4.2).

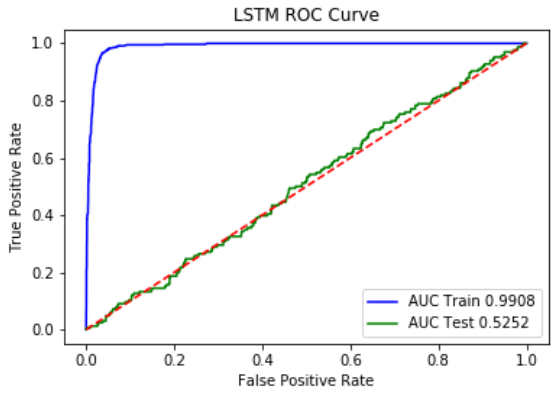

Figure 4: ROC curve of LSTM classifier on train and test sets. The discrepancy highlights overfitting because the model does extremely well on the training set but does not generalize.

5.4 Final Model Evaluation

Logistic Regression seemed to perform the best, when used with the n-gram TF-IDF vectorization. It seems that the custom embeddings were not trained on enough data and possibly overfit, so that set of feature engineering did not seem so useful[6]. The random forests models also seemed to overfit slightly, but the

[6] Not listed here, but the custom embeddings were not terrible. AUCs for Logistic Regression and Random Forest were both greater than 0.94.

Model	Type	AUC	Precision	Recall	F1 score
Logistic Regression	N-grams, TF-IDF Vect	0.9907	0.9932	0.8735	0.9295
Random Forest	Single Words, TF-IDF Vect	0.9853	1.0000	0.8494	0.9186
Neural Network	LSTM	0.5052	0.1563	0.1807	0.1676

Table 1: For each model, we list the best performing version of its hyperparameters and feature engineering. We omit the worse performing ones.

performance was comparable to the logistic regression. The LSTM classifier overfit completely.

6 Word Embeddings and Visualizations

Using the custom word embeddings model, we decided to extract the particular word embeddings and visualize them to see which words are similar. To get the embedding vectors, we utilized the default CountVectorizer with its built-in English Stop Words to build a vocabulary set. For these embeddings, we fit the method with upsampled embedded training data. Each embedding has a size of 300 and there are 8111 total words and corresponding word embeddings.

To get the data ready for visualization, the word embeddings have to be reduced dimensionally. To do this, we utilized the Uniform Manifold Approximation and Projection (UMAP) [McInnes et al., 2018] method to transform each word embedding to a two-dimensional set ready for visualization.

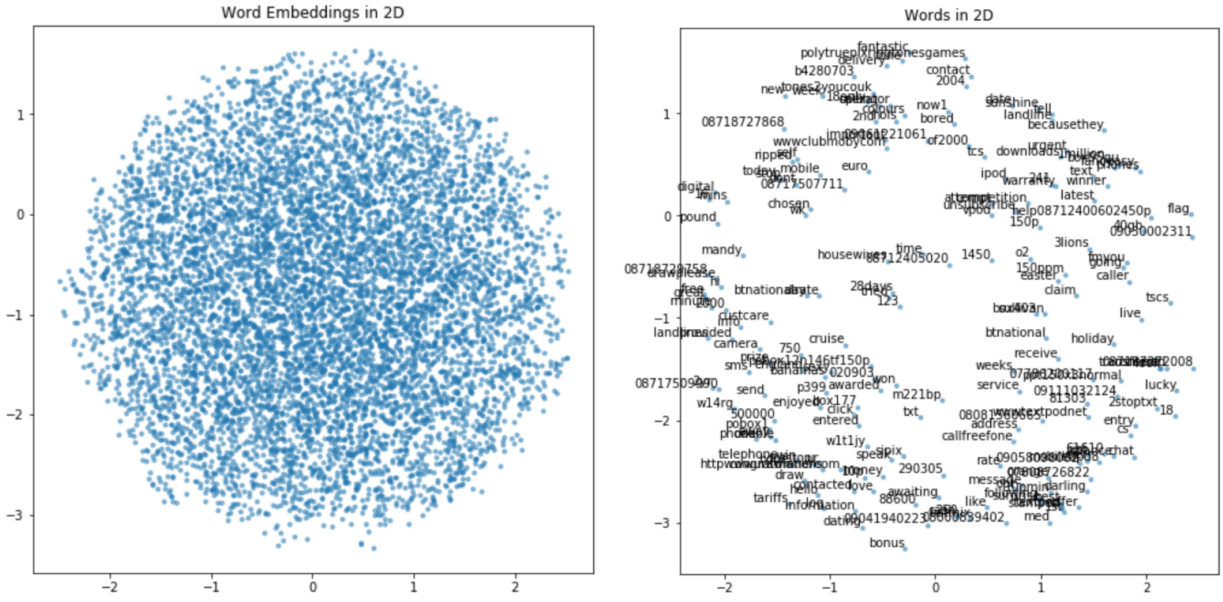

Figure 5: The left plot shows all of the 8111 words as two-dimensional embeddings. The right plot depicts the actual first 200 words

Another interesting aspect to analyze is to look at a more formal definition of word similarity. Taking

a sample of the first 1000 word entities, spaCy has a built similarity method that can give a value, between 0 and 1, for the similarity between two words[7]. Having a similarity of over 0.5 would imply that the two words are in fact similar. However, it is important to remember that similarity to humans can be a highly subjective topic and spaCy's similarity model utilizes a rather standard similarity definition. The word pairs with highest similarity from the first 1000 word sample are:

- cash and money, similarity = 0.8191,
- win and won, similarity=0.8187,
- going and just, similarity = 0.8183.

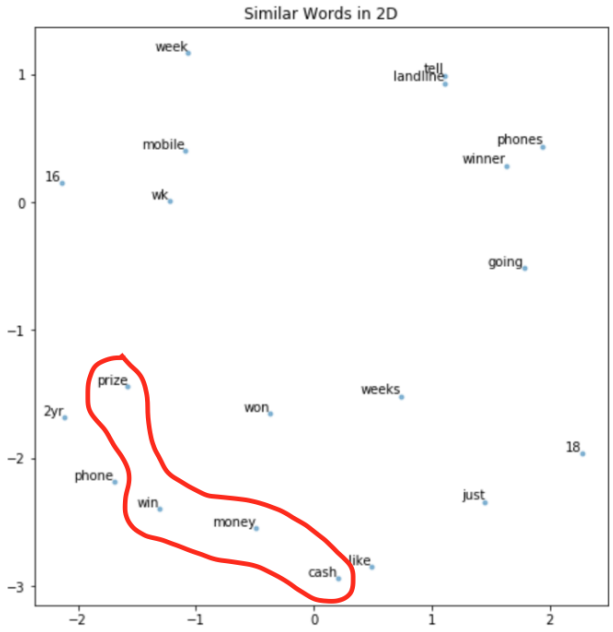

Figure 6: This plot shows how our embeddings seemed to correlate with the spaCy similarity.

7 Deployment

The results from our analysis will be deployed by passing the spam classification model with a best balance of high performance and ease of implementation, into production. Due to its high AUC, scalability, and low computation cost, the Logistic Regression with TF-IDF Vectorization and n-grams will be deployed to a messaging system. Messages flagged as spam by the model will be received in the user's inbox but marked as spam. This process will begin by passing our initial code through for further testing and engineering checks to get an optimized code that will be ready for production level deployment. This code will be streamed into

[7] To see more about spaCy similar vectors: https://spacy.io/usage/vectors-similarity

the messaging application. Once passed through to production, the classification system must be monitored by having users report if messages flagged as spam are actually spam. Reported misclassifications will help us find ways to improve the model by seeing what it struggles to correctly classify. In addition, because spam detection is such an adversarial problem, we should constantly monitor the relevancy of our models by analyzing data drift and model drift. Once we determine data drift via a hypothesis test or detect model drift because of degrading performance, we can deploy an updated version. It could also be that we periodically update the model. We could get spam and ham text messages from the deployment environment as users correct or implicitly enforce labels in production.

As we are working with sensitive user data, there are risks and ethical considerations to keep in mind. How the warning of spam messages is presented is important. For deployment, we decided to mark received messages classified as spam with a warning tag below, instead of moving the message into another section entirely. The second method would introduce a larger risk and a slightly questionable ethical situation, since it potentially leads to placing ham messages that are misclassified as spam, into a separate section. Since most people use messaging systems as their main method of communication, hiding misclassified ham messages means that people may miss important information being sent to them and ultimately lead to them becoming frustrated with the messaging platform and churning in favor for a system that will ensure they receive all their messages in a timely matter.

8 Future Work

Future work will be dependent on the deployed model performance and maintenance. Perhaps the methodology of separating spam messages will be later implemented, as long as users are made aware of this and our model continues to perform and learn well enough to minimize false positives. Regardless of the potential change in model usage, the model will have to be frequently updated to stay relevant with evolving phishing tactics.

Our custom word embeddings leave a lot to be desired. Our models would improve from using pre-trained embeddings using larger dictionaries, or we could also improve the embeddings by using more recent approaches for embeddings like the Continuous Bag of Words, Skip-gram methodologies, or GloVe embeddings.

It seems that Random Forest pipelines performed slightly worse than the best Logic Regression pipelines in terms of AUC and F1 scores. This is somewhat surprising, and it seems to indicate overfitting. We could use more data or use gradient boosted trees with heavier regularization. While gradient boosted trees tend to overfit more compared to random forests with the same number of trees, better performance

from gradient boosted trees might allow us to use fewer weak classifiers.

For the LSTM classifier, it was clear that overfitting was an issue. Our dataset was pretty small, and since neural networks have so many parameters, it is preferable to use a larger dataset. In addition, using embeddings with a lower or higher embedding size or simply better embeddings (more generalized) can improve the model. Once the overfitting issue is fixed, it might be useful to explore other recurrent neural network cells like Gated Recurrent Units.

References

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155.

[Buitinck et al., 2013] Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., and Varoquaux, G. (2013). API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9:1735–80.

[Hunter, 2007] Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95.

[Jones et al., 01] Jones, E., Oliphant, T., Peterson, P., et al. (2001–). SciPy: Open source scientific tools for Python. [Online; accessed ¡today¿].

[McInnes et al., 2018] McInnes, L., Healy, J., Saul, N., and Grossberger, L. (2018). Umap: Uniform manifold approximation and projection. *The Journal of Open Source Software*, 3(29):861.

[Paszke et al., 2017] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

A Contributions

Alexandria:

- Exploratory statistics of data and corresponding visuals
- Random Forest model using Count and TF-IDF Vectorizers with single, bigrams, and n-grams

Nabeel:

- Cross Validation framework for Logistic Regression and Random Forest models
- Custom Word2Vec model
- LSTM Classifier

Ziv:

- Logistic Regression model using Count and TF-IDF Vectorizers with single, bigrams, and n-grams
- Word Embedding visualizations and Similar Word analysis

Master Code: https://github.com/beelze-b/WhosInMyDMs