Machine Learning Engineer Nanodegree

Capstone Project

Mark Bastian
May 22nd, 2019

Political Tweet Classification

I. Definition

Project Overview

Twitter is a common platform for people, including politicians, to express their views. Often tweets are highly polarized and opinionated, especially given the current political climate in the United States as well as the rise of fake "Trolling" tweets. Given this, it is valuable to have tools to answer such questions as:

- Is this a tweet from a real person?
- What are the political leanings of a tweet?
- Can I determine the party of the author of a tweet?

Besides being interesting, such a tool could be useful for doing things such as targeted advertising of candidates to twitter users in general. Users that express sentiments that correlate well to one party would be good targets for advertising for issues or candidates with similar sentiments and values.

A massive amount of related work exists, including:

- Actionable and Political Text Classification using Word Embeddings and LSTM
- Leveraging Deep Learning for Political Leaning Classification
- On Classifying the Political Sentiment of Tweets
- Text Classifiers for Political Ideologies
- Topic-centric Classification of Twitter User's Political Orientation

Problem Statement

The problem I solved is classification of the author of a tweet into one of the two major US political parties (Republican or Democrat). Specifically, the model models I developed take a tweet or tweet length text as input and produce a prediction of whether the tweet's author is a Democrat or Republican. The input is expected to be political in nature, so I do not detect non-political tweets and I limit the output categories to only the two major parties.

The dataset I trained on is sourced such that I know beforehand the party of the tweet. This can be broken up into testing and training sets to measure goodness of model fit.

As I did not know exactly what model would do best in this situation, my approach was to use a simple Naive Bayes Classifier as a baseline model and then try several different neural architectures to see if any stood out as being better than the Naive Bayes Classifier or at least had potential to do well.

Metrics

As all data is already tagged, it is appropriate to use precision, recall, accuracy, f1 score, and a confusion matrix to evaluate the quality of my solution.

These metrics are defined as:

$$Accuracy = \frac{True\ Positives + True\ Negatives}{True\ Positives + True\ Negatives + False\ Positives + False\ Negatives}$$

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Intutively, *Accuracy* says "How did I do at correctly predicting a category over my entire population?", *Precision* says "If I stated something belongs to a category, how likely am I to be correct?," *Recall* says "If something belongs to a category, how likely am I to find it?," and the *F1* score is a metric that pulls towards the worst of the two previous scores, so you must do well at both for a high F1 score.

This can be viewed with a Confusion Matrix as well, where D(0) and R(1) correspond to Democrat or Republican:

	Predicted D(0)	**Predicted R(1)**
Actual D(0)	True D(0)	False R(1)
Actual R(1)	False D(0)	True R(1)

For skewed populations it is easy to see that Accuracy is a poor metric as any model that favors that larger population will have high accuracy. In our case, the number of tweets is fairly evenly divided between the two parties (Democrat=42068, Republican=44392), so accuracy should be a good metric here.

If there is high risk or regret associated with a misclassification (I said you were a Democrat, but you were really a Republican) then we should favor models with high Precision. In our case, there is no higher penalty for one type of misclassification over another, so Precision isn't as important a metric for us.

If there is high risk or regret associated with not finding all of a given category (I was trying to find all of the Democrats, but missed some) then we should favor high Recall models. Again, there is no reason to favor one category or another in this case, so Recall is not as important either.

As Accuracy is more important than Precision or Recall in this case, F1 is also not going to be as important since it is a weighted score of our two less important metrics.

II. Analysis

Data Exploration

I used the Democrat Vs. Republican Tweets dataset found here.

This dataset provides 86,460 tweets divided roughly evenly (over 42,000 tweets per party) and is sufficiently large to produce sizeable testing and training sets. As the authors are all politicans it is implicitly categorized by the author's political affiliation.

Here is the breakdown of tweets by party:

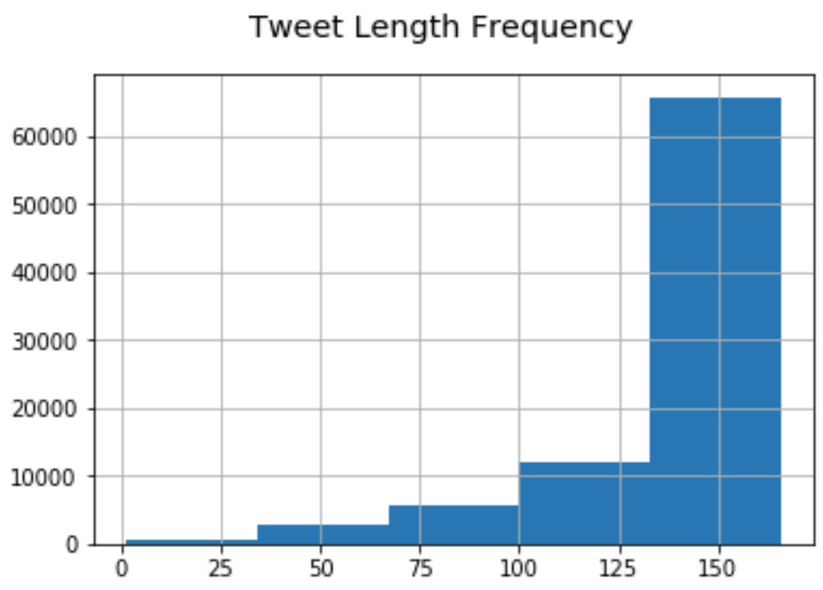
Statistic	**Value**
Democrat Tweets	42068
Republican Tweets	44392

This was loaded using the following code:

```
categories = ['Democrat', 'Republican']
tweetsdf = pd.read_csv('democratvsrepublicantweets/ExtractedTweets.csv')
handlesdf = pd.read_csv('democratvsrepublicantweets/TwitterHandles.csv')
raw_tweets = tweetsdf['Tweet']
parties = tweetsdf['Party']
y = 1.0 - np.asarray(parties == 'Democrat')
X_train, X_test, y_train, y_test = train_test_split(raw_tweets,
                                                    y,
                                                    test_size=0.2,
                                                    random_state=42)
```

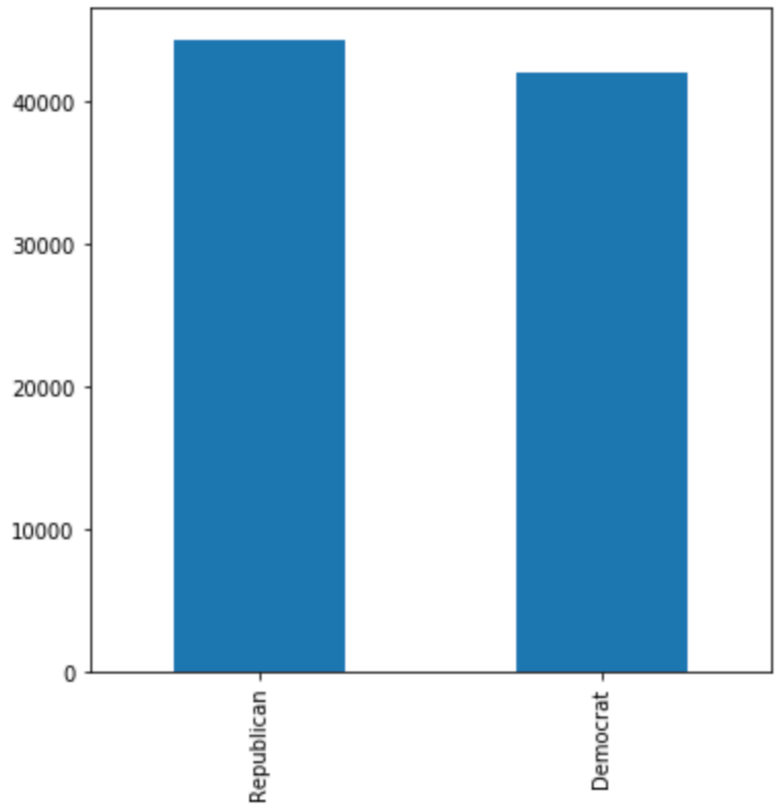
The above code listing was used for all models.

Intuitively, it would make sense for longer tweets to have more information, as there are more words to give context to the tweet. Based on this next plot, the vast majority of tweets are greater than 100 characters and most are centered at 150 characters, which is good.

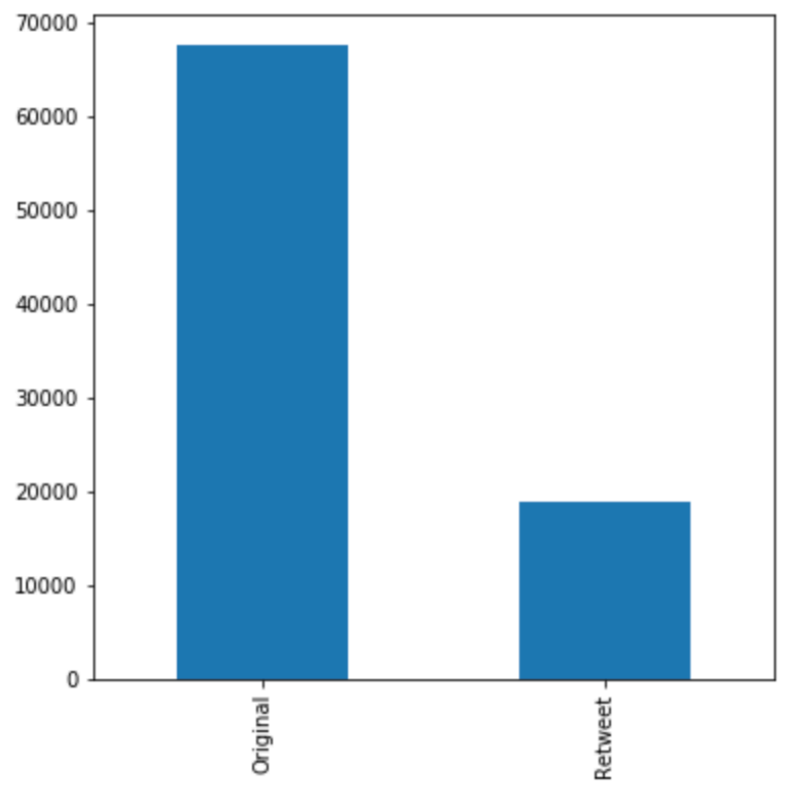


Exploratory Visualization

As can be seen in the below image, the number of tweets is fairly evenly split between parties (Republican=44392, Democrat=42068).



Futhermore, the number of original tweets is fairly high compared to the number of retweets. It is possible that a retweet might cause data duplication or be something that doesn't express the original opinion of the retweeter, but I am going to assume that if a tweet is repeated by an individual then they agree with the views expressed in the original tweet.



Algorithms and Techniques

My intent for this project was two solve the problem using two categories of algorithms and compare the results. The first would be a Naive Bayes Classifier and the second would be a Neural Network. The specifics of the latter were left open in the proposal as there are a lot of different potential architectures and I wanted to investigate several. I ended up trying our several techniques, including word and character embedding layers, 1D convolutional networks, and LSTM layers.

Key algorithms I tried are:

- Naive Bayes Classification using the sklearn MultinomialNB class. From the documentation, "The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work." Naive Bayes is a simple algorithm that computes the probability that something belongs to a class given the presence of the words in the item to be classified. This does not take into account any aspect of word ordering or the sentiment of a sentence.
- Deep Neural Networks: In all models except for the Naive Bayes Classifer, some form of neural network is used. The "Deep" aspect of the network is simply the fact that hidden layers are present in the network that must be trained. All input is in the form of a vector whose values are multiplied and summed then passed through an activation function for node of each layer in the network. There are a large number of layer types and I will highlight some of the main ones used after this.

- Embeddings (Word and Character Level): Embedding layers take a sparse input (one hot encoded words or characters) and map them into dense vectors of data. This can give context to words or character sequences and reduce the number of input parameters since you are going from a sparse to a dense representation. An excellent explanation of word embeddings can be found at The Illustrated Word2vec.
- Dense Layers: In a "traditional" deep network, many fully connected layers are used. These do not have any concept of spatial or temporal context or connectivity.
- Convolutional Layers: Convolution Layers consume arrays of data and use kernels rather than scalar weights to transform their input. These kernels can be used to detect spatial features in input. For textual data, a 1D Convolutional Layer can be used to learn relationships between words or characters. For example, encoding "I am happy" as [1,2,3] vs. "I am not happy" as [1,2,4,3] contains spatial relationships that can be learned with a kernel such that a network could determine the sentiment of happiness vs. unhappiness. Recently, there has been a lot of work in using CNNs for text despite not having the "memory" of LSTMs because CNNs are so much faster to train. For example, this blog post discuses the pros and cons of CNNs vs. RNNs at length, with the conclusion that CNNs work well for text classification and are much faster to train.
- LSTM Layers: LSTMs, or Long short-term memory layers, are recurrent layers that maintain a short term "memory" internally and are appropriate for learning sequences of data, such as word sequences in sentences. Perhaps the best resource for gaining intuition into LSTMs is Understanding LSTM Networks. One of the challenges of LSTMs is that due to their recurrent design, they cannot be trained in parallel, so training can take a very long time. For this reason, I favored architectures with convolutional layers in this project.

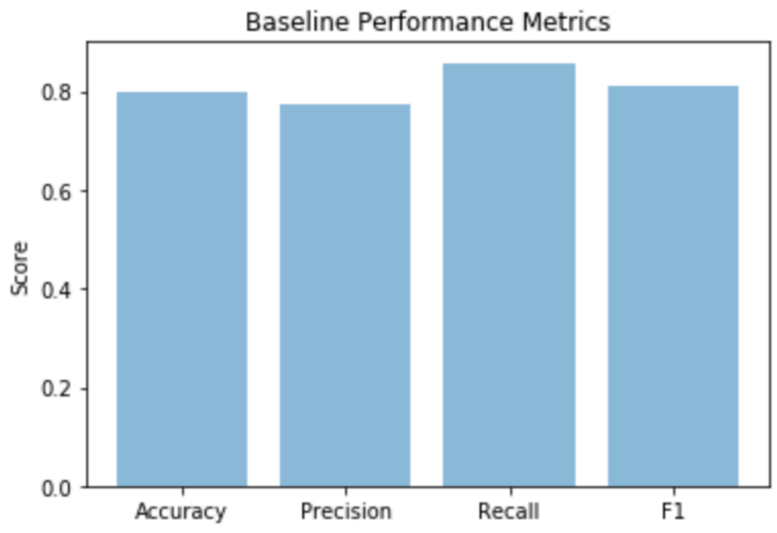
Benchmark

My baseline model was a simple Naive Bayes Classifier created using the following pipeline:

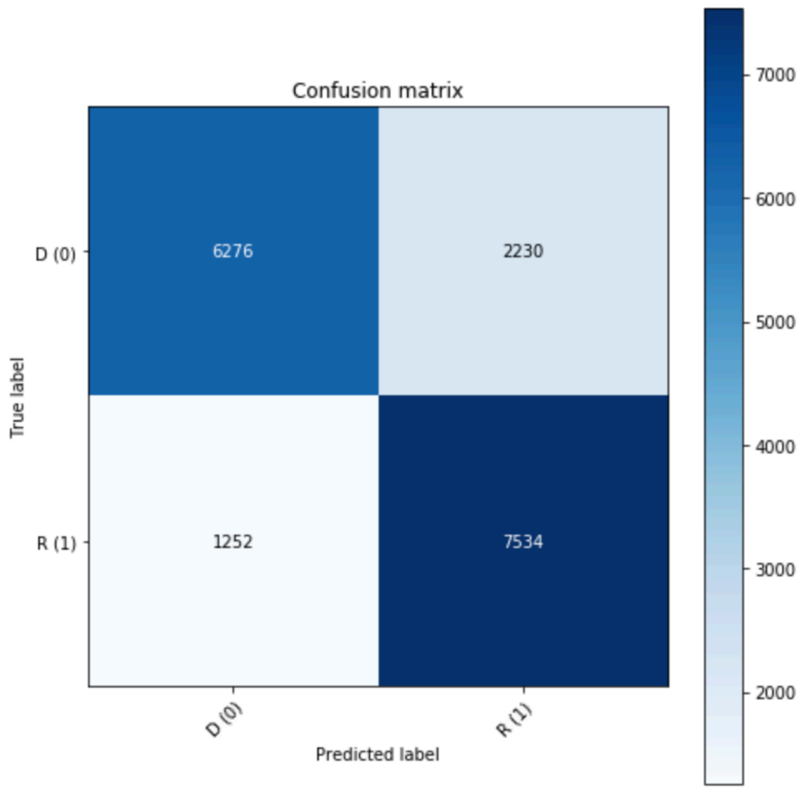
```
model = Pipeline([('vect', CountVectorizer()),
                  ('tfidf', TfidfTransformer()),
                  ('clf', MultinomialNB())])
```

As shown above, I used an 80/20 split to divide training and testing data for all experiments.

The baseline Naive Bayes Classifier produced accuracy, precision, recall, and f1 scores of 0.80, 0.77, 0.86, and 0.81, respectively, and is show graphically here:



The corresponding confusion matrix is shown here:



Since this is a fairly evenly split dataset and there is no preference for precsion or recall, accuracy is probably the most useful metric as we have an equal preference for correct classification into either class. The baseline accuracy is computed as total correctly classified items over the total population, that is (6276 + 7534) / (6276 + 7534 + 1252 + 2230), which is 80%.

III. Methodology

The goal of the remainder of this project was to compare the above model with several neural network architectures to see if the networks could be tuned to give better overall performance, with accuracy being the most interesting metric.

The following networks were tried:

1. A 1D Convolutional Network
2. A 1D Convolutional Network with dropout layers added to prevent overfitting
3. A Character Embedding Layer (dimension 256) followed by a Convolutional Layer with 256 filters.
4. A Character Embedding Layer (dimension 256) followed by two Convolutional Layers with 256 then 128 filters.
5. A Character Embedding Layer (dimension 256) followed by three Convolutional Layers with 64, 32, then 16 filters. A 512 unit Dense Layer followed the Convolutional Layers. This trial was meant to investigate if network depth did a better job of generalizing than network width.
6. A Character Embedding Layer (dimension 32) followed by a 2 unit LSTM. This was meant to see if a smaller network with different layer types would perform differently.
7. A Word Embedding Layer (dimension 100) followed by two Convolutional Layers (128 and 64 filters).
8. The same architecture as network 7, but with a different strategy for generating the vocabulary.
9. A Characer Embedding Layer (dimension 64) followed by two LSTM layers of dimension 64. This was done to see if more "memory" would assist in a better fit.

All solutions can be found here in notebooks starting with the model number (e.g. 1*political*party_classifier.ipynb for Architecture 1).

The models themselves are located in the models file and are named by their model number.

For example, here is the listing for model1:

```
def model1(input_length):
    """A convolutional network with no embeddings."""
    input_layer = Input(shape=(input_length,1))
    x = Conv1D(256, kernel_size=4, activation='relu')(input_layer)
    x = MaxPooling1D(pool_size=2)(x)
    x = Conv1D(64, kernel_size=4, activation='relu')(x)
    x = MaxPooling1D(pool_size=2)(x)
    x = Conv1D(32, kernel_size=4, activation='relu')(x)
    x = MaxPooling1D(pool_size=2)(x)
    x = Flatten()(x)
    x = Dense(2, activation='softmax')(x)
    model = Model(input_layer, x)
    optimizer = Adam(lr=0.0003)
    model.compile(loss='binary_crossentropy', optimizer=optimizer)
    return 'cnn-weights-0.0337.hdf5', model
```

All Convolutional Layers are followed by a Max Pooling layer with a pool size of 2. Each of the above networks was completed with a 2 element Dense layer (one for each final category) with softmax activation.

These different ideas were inspired by many different blog posts, including:

- How to Use Word Embedding Layers for Deep Learning with Keras
- What Are Word Embeddings for Text?
- How to Develop a Word Embedding Model for Predicting Movie Review Sentiment
- Embed, encode, attend, predict: The new deep learning formula for state-of-the-art NLP models
- How to Develop Word Embeddings in Python with Gensim
- Stacked Long Short-Term Memory Networks

Data Preprocessing

For all character-level encodings, the following transform was applied to get the data in the right format:

```
def create_encoder_decoder(all_text):
    chars = sorted(set(all_text))
    char_to_int = dict((c, i + 1) for i, c in enumerate(chars))
    int_to_char = dict((i + 1, c) for i, c in enumerate(chars))
    return char_to_int, int_to_char


def encode_string(line, char_to_int, l):
    z = np.zeros(l)
    z[0:len(line)] = [char_to_int[c] for c in line]
    return z


def encode_strings(lines, char_to_int, l):
    return np.array([encode_string(line, char_to_int, l) for line in lines])
```

This was applied like so:

```
X = encoders.encode_strings(normalized_tweets, char_to_int, max_tweet_len)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=42)
```

For the first word embedding architecture (Architecure 7, above), the following additional transformations were used to encode the words:

```
cv = CountVectorizer(stop_words='english', strip_accents='ascii')
bag_of_words = cv.fit_transform([re.sub(r'https?://[^\s]+', '', tweet) for tweet in r
aw_tweets])

vocab = {}
for feature, freq in zip(cv.get_feature_names(), bag_of_words.sum(axis=0).tolist()[0]
):
    if freq > 10:
        vocab[feature] = freq

vocabulary = list(vocab.keys())
vocabulary_size = len(vocabulary) + 1

word_to_int = {word: i + 1 for i, word in enumerate(vocabulary)}
int_to_word = {i + 1: word for i, word in enumerate(vocabulary)}

encoded_tweets =\
    [[word_to_int.get(word, 0) for word in word_tokenize(re.sub(r'https?://[^\s]+', '
', tweet).lower())
      if word in word_to_int]
     for tweet in raw_tweets]
```

For Architecture 8, the following word encoding was used:

```
def tokenize_tweet(s):
    s = s.lower()
    s = re.sub(r'https?://[^\s]+', '', s)
    s = re.sub(r'[^A-Za-z\s$#@0-9]+', '', s)
    s = re.sub(r'\s+', ' ', s)
    return [tok for tok in s.strip().split(' ') if tok not in english_stopwords]

tokenized_tweets = [tokenize_tweet(tweet) for tweet in raw_tweets]

vocab = {}
for toks in tokenized_tweets:
    for tok in toks:
        if tok in vocab:
            vocab[tok] += 1
        else:
            vocab[tok] = 1

vocab = {k:v for k,v in vocab.items() if v > 10}

vocabulary = list(vocab.keys())
vocabulary_size = len(vocabulary) + 1

word_to_int = {word: i + 1 for i, word in enumerate(vocabulary)}
int_to_word = {i + 1: word for i, word in enumerate(vocabulary)}

encoded_tweets =\
    [[word_to_int.get(tok, 0) for tok in toks if tok in word_to_int]
     for toks in tokenized_tweets]
```

This second encoding used my own tokenizer in which I removed urls and emojis, but kept hashtags and 'at' targets (e.g. @soandso). This was done to see if perhaps there was any significance to keeping those items in the vocabulary.

Implementation

For each architecture, the networks were run for at least 100 epochs and weights were saved if loss improved. Due to the challenges of keeping track of the number of runs for long time periods on local hardware (or sometimes using AWS instances) the number of specific epochs were not tracked specifically. Generally, however, losses were at a minimal or near minimal state after this time period and rarely showed additional improvement with further iteration.

The code shown above was used for each architecture and models were put in a models.py file so that most notebooks have a section that loads the files and is then generally followed by several common code blocks.

First, the models are loaded from models.py:

```
# note that the model number (e.g. model5) would change per notebook
filepath, model = models.model5(len(char_to_int) + 1, max_tweet_len)
if filepath in os.listdir():
    model.load_weights(filepath)
model.summary()
```

Next, training was done using a common function:

```
def train(X_train, y_train, model, filepath, num_epochs=100, batch_size=1000):
    checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=
True, mode='min')
    tensorboard = TensorBoard(log_dir="logs/{}".format(time()))
    callbacks_list = [checkpoint, tensorboard]
    model.fit(X_train,
              np_utils.to_categorical(y_train),
              epochs=num_epochs,
              batch_size=batch_size,
              callbacks=callbacks_list)
```

This training function would be called several times if convergence was not achieved. Generally losses did not improve with most models after 200 epochs.

Finally, results were computed and plotted as follows:

```
predictions = np.argmax(model.predict(X_test), axis=1)

metrics, confusion_matrix = models.plot_results(y_test, predictions)
(accuracy, precision, recall, f1) = metrics
print('Accuracy: %s' % accuracy)
print('Precision: %s' % precision)
print('Recall: %s' % recall)
print('F1: %s' % f1)
```

`plot_results` provided a common set of results as computed here:

```
def plot_results(y_test, predictions):
    objects = ('Accuracy', 'Precision', 'Recall', 'F1')
    y_pos = np.arange(len(objects))
    performance = [accuracy_score(y_test, predictions),
                   precision_score(y_test, predictions),
                   recall_score(y_test, predictions),
                   f1_score(y_test, predictions)]
    cm = confusion_matrix(y_test, predictions)

    plt.bar(y_pos, performance, align='center', alpha=0.5)
    plt.xticks(y_pos, objects)
    plt.ylabel('Score')
    plt.title('Baseline Performance Metrics')
    plt.show()

    plt.rcParams["figure.figsize"] = (7,7)
    classes=np.array(['D (0)', 'R (1)'])
    plot_confusion_matrix.plot_confusion_matrix(y_test.astype(int), predictions.astyp
e(int),
                                                classes=classes,
                                                title='Confusion matrix')
    plt.show()
    # This is a variation of the example found online at
    # https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_ma
trix.html
    plot_confusion_matrix.plot_confusion_matrix(y_test.astype(int), predictions.astyp
e(int),
                                                classes=classes,
                                                normalize=True,
                                                title='Normalized confusion matrix')
    plt.show()

    return performance, cm
```

Complete listings of all code and implementations can be found at https://github.com/markbastian/capstone.

Refinement

The various architectures 1-9 above were each created as a follow on to the previous architecture in an attempt to explore what could be improved or attempted as a next experiment. In many cases, the follow on architecure was "going wider" by expanding the number of units or "going deeper" by adding additional layers (but halving the units or filters).

IV. Results

Model Evaluation and Validation

As was stated previously, accuracy is the driving metric for this study as there is no preference for precision or recall and the data is divided fairly evenly. A summary of the architectures are listed here, sorted by accuracy.

Architecture	Accuracy	Precision	Recall	F1
1	0.798635	0.771610	0.857501	0.812291
8	0.758443	0.758381	0.769861	0.764078
7	0.755552	0.751906	0.774414	0.762994
4	0.732882	0.715482	0.787389	0.749716
3	0.685751	0.710712	0.643410	0.675388
6	0.603863	0.605195	0.633849	0.619191
2	0.557021	0.547247	0.742204	0.629987
5	0.531055	0.581156	0.275893	0.374161
9	0.508096	0.508096	1.000000	0.673825

A few interesting observations:

- The initial baseline model (A Naive Bayes Classifier) outperformed the other models.
- Word embeddings were the next best models.
- The next group used character embeddings with 2 convolutional layers.
- With the exception of the "complicated" architectures, a simple deep network with no embeddings did worse than the other architectures.
- Architectures 5 and 9 fared worst. These both are characterized by having more deep layers than their counterparts (3 vs. 2 Convolutional Layers or 2 large LSTM layers vs. 1 smaller LSTM layer). In fact, architectue 9 classified all tweets as Republican, so obviously did not learn much at all.

All of these models were evaluated using 20% of the original data that was held out for testing.

Regarding robustness, the Naive Bayes Classifier uses standard word counts so as long as the vocabulary of a given tweet didn't change dramatically and is of a political nature, I would expect consistent performance. Although not as good as the Naive Bayes classifier, the word embedded models do limit input via their encodings to an existing vocabulary of 8736 words and remove outliers like urls that will vary all the time and probably don't contain useful information. One potential concern might be how much of the results are based on who the target of a tweet is or on a particular hashtag. For example, if a particular politician did a lot of tweeting about a current hot button issue and that that politician left office or the issue cooled off the model might not respond as favorably to newer people or subjects. This is discussed more on the conclusions.

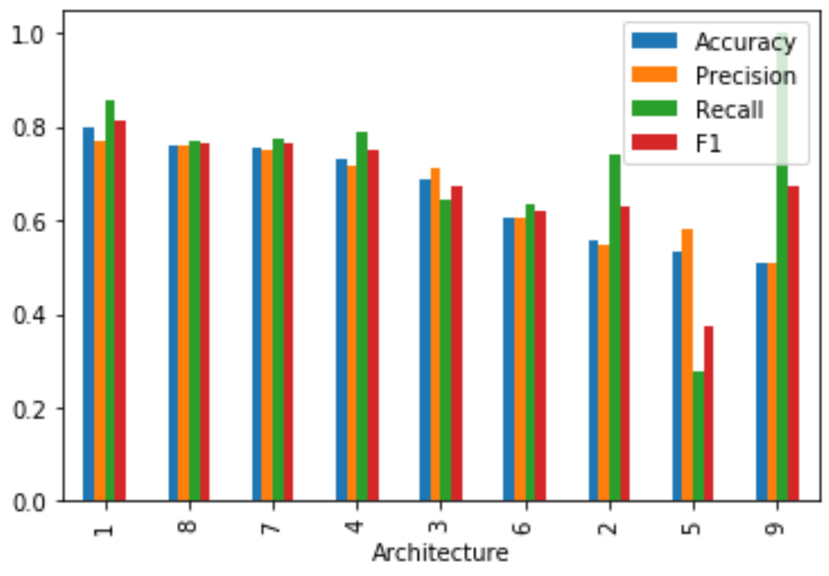
Justification

While I am a bit disappointed that none of the architectures beat a basic Naive Bayes Classifier, the "intent for this project was two solve the problem using two categories of algorithms and compare the results," and not to explicitly beat the Naive Bayes Classifier. The thing I feel needs the most defense is the final set of architectures used. I wanted to try architectures with character embeddings, word embeddings, LSTMs, and CNNs and there are effectively an infinte number of combinations of layers, activation functions, and hyperparameters (e.g. units or filters) that can be chosen from. I felt that the architectures used were fairly representative of basic architectures described online and in the class.

V. Conclusion

Free-Form Visualization

The best way to capture the results of this study is to look at the various metrics for each model. As stated earlier, Accuracy is the metric that is most important as the samples are split fairly evenly and there is no preference for precision or recall.



In general, word embeddings seem to be the best neural models after a Naive Bayes Classifier.

Reflection

This project resulted in a pipeline that created normalized text input in a small number of formats depending on the model used, trained a set of models on this input, and then used a common reporting mechanism for all of the results.

Several aspects of the results that I found interesting were:

- What I found most interesting about this study is that the neural net based approaches, especially word and character encodings, did not perform as well as a basic Bayes Classifier. However, I think this shows one of the potential strengths and weaknesses of using deep networks as a solution. I believe that with the right tuning and hyperparameters it is possible to do better than the Bayes Classifier. However, this requires a lot of patience, trial and error, intuition, and skill.
- On a related note, one thing that I found particularly challenging was hyperparameter tuning for deep networks. There are a massive number of combinations that can be tried and many infrastructures (e.g. LSTMs) can take a very long time to train.
- Embeddings, particularly word embeddings, did very well.

Improvement

Given the general observations I made on the various models I tried, I thing the following are potential areas for improvement:

- Do more investigation and work with embedding layers. In particular, I would try two approaches:
    - Try some off the shelf pre-trained models such as word2vec or Amazon's new Object2Vec model.
    - Try different permuations of the solutions that did work.
    - Investigate the sensitivity of the model to tweet targets (e.g. @soandso) and hashtag topics. If the model is highly suceptible to those items, then a newer model could remove them or do continuous learning from a data feed to always stay current.