Final Report

Introduction:

The problem that this algorithm is designed for is the identification of 'fake' tweets for journalists, this is to help prevent a journalist from using false information and spreading fake news that is severely criticising and a massive reputation loss. In the definition of the problem provided to us the definition of fake posts are:
"

> • Reposting of real multimedia, such as real photos from the past re-posted as being associated to a current event
> • Digitally manipulated multimedia
> • Synthetic multimedia, such as artworks or snapshots presented as real imagery
> "

This is how I should be aiming to identify fake posts with my algorithm. However as I mention later, this is not the case due to what data is provided.

Importing the data:
While importing the data into python in the method I used I had to tweak the formatting of certain entries as they were not being accepted by decoder. These entries were lines 573, 1812, 2547, 2675, 2714, 4240, 4441, 6637, 6677, 9*** (forgot), 10119 and 11572 I did not edit any of the data contained in these lines, they just had an extra tab on the end which confused the import, so I tidied them up.
I did however remove line 6391 id 2.63091E+17 as a part of the cleaning process as it had some character which caused conflicts with the utf8 encoding I was using meaning I couldn't import the txt file to Python with numpy. However, this entry wouldn't have been used regardless as you will find in a later point.

Data Analysis:

Straight away looking at the data we find that there are 7 fields provided in each entry namely:

tweetId	tweetText	userId	imageId(s)	username	timestamp	label

And the average entry in the training set would look as such:
4.4848E+17
Malaysian Flight MH370 has been found in bay. #PrayForMH370 http://t.co/JamMvMfJso
1407445411
malaysia_fake_13
JonasLucas19
Tue Mar 25 15:22:10 +0000 2014
fake

Looking at it one can very quickly identify that the imageId has the word ‘fake’ in it, this initially led me to believe that perhaps the method of identifying the image meta data was provided in this way. For the majority of entries in the training set the imageId either had ‘fake’ or ‘real’ in them, leading me to believe this is how I would identify ‘Digitally manipulated multimedia’ as defined by earlier.

This is however false, upon closer inspection, I searched for a case of what I thought were ‘real’ images being reposted but classified as ‘fake’. However, I found no such case, this lead to suspicions on my part, so I decided to view the test data to check that the entries where indeed similar in both the training and test data. In the test data, I found not a single entry that had an imageId that had ‘fake’ or ‘real’ in it, that was when I concluded that we had no image metadata available to us. This troubled me, as the definitions of ‘fake’ media provided above generally revolve around the image, I had no method however to determine whether an image was real or not in a general circumstance with purely supervised learning.

Looking at the data, I eventually determined that only the tweetText held relevant information for the type of algorithm that I intended to develop for this, a supervised learning algorithm. TweetID wouldn’t have provided any information as it is just a unique identifier, on the other hand I felt userId, username and ImageId could be used to keep track of reliable sources, and maybe identify reposts. I felt this was impossible as it would require too much storage to be feasible and a supervised learning algorithm wouldn’t be able to adapt to new images and users. TimeStamp could be used as a way of identifying original sources, but again, supervised learning prevents this as we would be training it on timestamps that will never happen again and again it wouldn’t be able to adapt to new data.

I did notice that the ‘humor’ labels as mentioned in the document that should be treated as ‘fake’ labels.

Algorithm design & Evaluation:

Pre-processing, feature selection and algorithm will only describe the first iteration. later iterations and the changes I performed during them will be described in individual sections afterwards, I will summarise the final algorithm I settled on as the final part of this section. At the end of each iteration I will show the results and then evaluate them. In my Code, I didn’t not dispose of any of the iterations, I just created new functions that replaced the old ones so you can perform each of these iterations in my code.

Pre-processing:
For preprocessing, I decided to use Position Of Speech (POS), my reasoning for this is that I want a general machine learning algorithm, I felt that by not using POS I would likely fall into the trap of overfitting. Certain words that may be common to use in the data provided might have become obsolete in 1 or 2 years, however, POS is much more robust to these changes as it analysis the structure of the language rather than the words being used in it. I used these together with ngrams to produce a list of trigrams for each entry. This is because I wanted to base my algorithm off the structure of language being used rather than specific words or specific types of words being used, making it more robust to changes in the words used in culture.

Originally I was intending to have a translator and spellchecker. The translator was to pick up on foregin languages and translate them quite obviously, the spellcheck was meant to pick on spelling mistakes to help the POS more correctly identify the words being used, I did away with these two dues to separate reasons.

For the translator, I tried multiple APIs for identifying and translating foreign languages however, all of them had a limit to their usages, I tried both translate and googletrans APIs, however they both had daily limits to the number of times they could be used. So I faced no issue when testing with them on smaller datasets extracted from the training data, when it came to using them on the entire data, I was quickly shut out by both APIs for the day, without having finished processing all the entries. As you can tell I decided to simply scrap foreign languages entirely, the langdetect API I now use does not have such limits as it only detects the language rather than the more intensive translating. I used it to throw out all foregin entries as I could not properly process them into POS without translating them first.

Regarding foreign languages, I'm personally of the opinion that separate algorithms should be built for them individually, my reasoning behind this, is that machine translation is not very accurate, while a human can generally grasp the meaning behind it, often machine translation leaves much to be desired. Even if I did successfully translate the foregin entries, I wouldn't feel that comfortable in processing them, knowing that they may introduce variables that go against the precedent that the english entries set due to being Machine Translated. If I were to build one for a foreign language I would actually check if there was a POS library for said language then translate it into english. Also the processing time it consumed was enormous, only to find it failed in the end due translating API limits.

I decided to remove the SpellChecker because processing the data went from around 10mins on my laptop to 3 hours. I didn't want to have to wait 3 hours every time obviously, but this also raises an issue that is very important to journalists, if they are in a rush to break the news first, then an increase in processing time by such a massive amount is simply not worth it, even if it does improve the f1-score.

At the end of pre-processing each entry would be in this form as a result of using collections.Counter() to count the occurance of each trigram in an individual entry:
[list([(('JJ', 'VBD', 'JJ'), 1), (('VBD', 'JJ', 'NNP'), 1), (('JJ', 'NNP', 'NN'), 1)]), 'fake'],

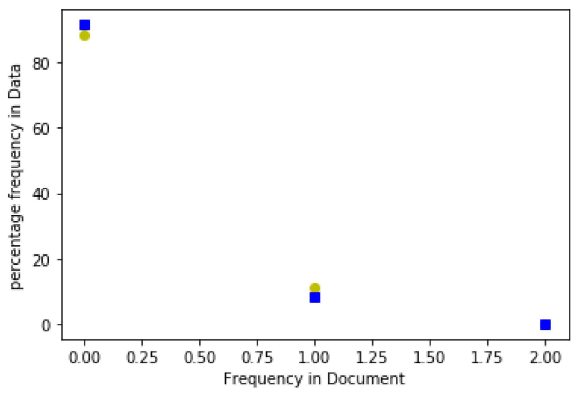
Feature Selection:
For feature selection I took the manual approach as I wasn't confident in implementing an automatic approach, I felt that I could potentially implement it in a later iteration, which I did not do in the end however.

In order to make a decision on which features to base that algorithm, I considered each possible trigram as a feature and the number of occurrences of said trigram would be the values that the algorithm would read. Using only trigrams actually created multiple issues and it was here that the first signs show up, I only picked up on them after having finished my first iteration, this issue will be addressed more in depth in the Evaluation section.

During the Pre-Processing section, I collected information on the dataset as a whole; every single trigram that appeared in the dataset was added to a list, repetitions included; I then used collection.Counter to create a collection that held a list of tuples, each tuple representing a unique trigram and the number of occurrences in the dataset. Then by using the collections method most_common() I was able to get the most 'influential' trigrams.

From the top 50 I then printed a graph of percentage occurrence against frequency in entry for both 'real' and 'fake' entries then compared them to see which trigrams had differences between their 'real' and 'fake' percentage occurrences. To ensure complete clarity as I know how awful my ability to portray meaning. What I mean by percentage occurrence in this case is the percentage of entries that used trigram y an x number of times. Originally I plotted frequency in data against frequency in entry, the issue with this, is that there are well over 1000 more 'fake' entries than there are 'real', thus they weren't equally scaled, making it very difficult to find good features. Using percentage occurrence essentially scaled them to the same level allowing me to more accurately observe the patterns in the data.

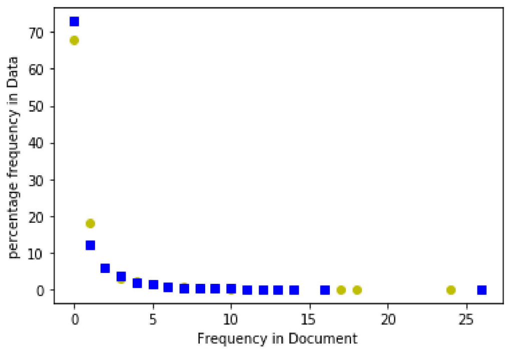
Example
trigram : ('JJ', 'NN', 'NN')



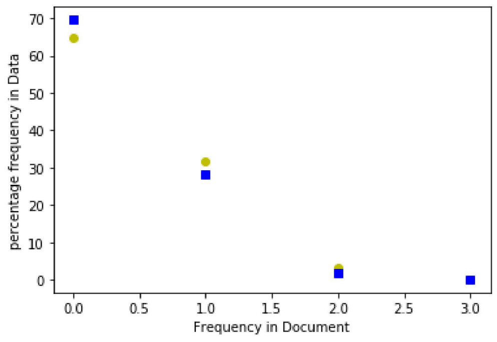
For my first Iteration I chose
('NNP', 'NNP', 'NNP'), ('NNP', 'NNP', 'NN'), ('NN', 'IN', 'NNP'), ('NNP', 'NN', 'NN'), ('JJ', 'NN', 'NN'), ('NNP', 'NN', ':'), ('.','JJ','NN')

Sample:

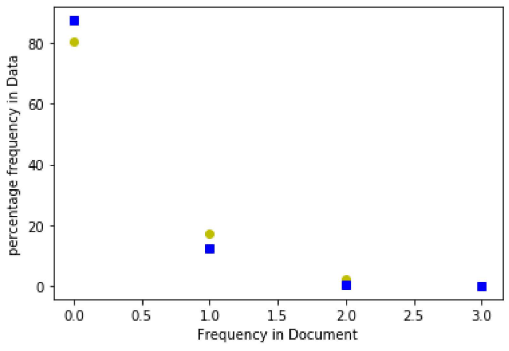
('NNP', 'NNP', 'NNP')



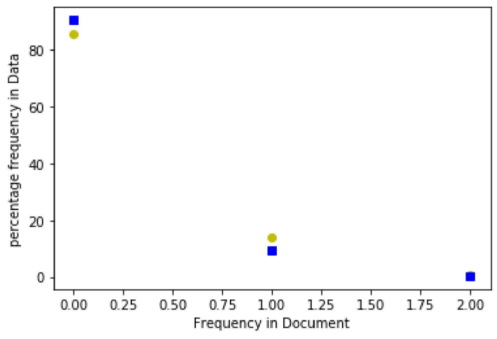
('NNP', 'NNP', 'NN')



('NN', 'IN', 'NNP')



('NNP', 'NN', 'NN')



I then took the processed data and then reformed it such that each entry was a table that held the frequency of each of the selected features, this was then used to train the algorithm

Algorithm:
I considered NRC, NBSVM and SVM[1][2] when thinking about which algorithm to go with, in the end I decided to go with Support Vector Machine as my algorithm of choice. I feel most comfortable with SVM as I have already studied this algorithm on the course and have direct code examples I can reference from lectures. This will make it easier to cope with any issues that crop up with SVM than the other algorithms, Ideally making up for my lack of experience with Machine Learning algorithms, so I would rather go with something I can easily get reference material for.

Iteration Evaluation 1:
After fitting the algorithm and setting C to 1 initially and I then tested it with 'fake' being a Positive and got the precision, recall and f1_score, results I got

```
precision:  0.5908923980903416
recall:  0.9932098765432099
f1_score:  0.7409624683398572
```

A precision was descent, f1_score was good and recall was incredible . . . a bit too good. I definitely thought that the first try shouldn't be this good in anything without optimization. I tested several times with different C values but it had little to no effect.

Didn't think too much about it. I proceeded to test an SVC with polynomialFeatures to transform it

```
precision:  0.5755926933540614
recall:  0.9141975308641975
f1_score:  0.7064154543286429
```

This deeply puzzled me as to why everything dropped
Finally I thought to directly see the number of TPs, TNs, FPs and FNs for both

```
linear:
TP:  1609 TN:  11 FP:  1114 FN:  11

polynomial:
TP:  1481 TN:  33 FP:  1092 FN:  139
```

Slight issue here, what was happening was my algorithm was just considering virtually everything as 'fake', my initial thoughts to this is that the data was too grouped up, there was virtually no difference between the 'fake' and the 'real'. I did later add Accuracy to the mix as well as a precision, recall and f1_score for if 'real' was the positive, this significantly helped me understand this issue better.

```
Accuracy:  0.5901639344262295
                fake                 real
precision:      0.5908923980903416  0.5
recall:         0.9932098765432099  0.009777777777777778
f1_score:       0.7409624683398572  0.019180470793374017
```

As you can see the while for 'fake' the results are too bad for the linearSVC they are horrible for 'real'

On the other hand for the for the polynomial SVC: note: I kept to only 2 degrees as anymore and the training time was taking too long for me to know if my computer had frozen (which did happen sometimes) or it was still training. The long processing is likely due to the number of features.

```
Accuracy:  0.5515482695810565
                fake                 real
precision:      0.5755926933540614  0.19186046511627908
```

recall:	0.9141975308641975	0.029333333333333333
f1_score:	0.7064154543286429	0.05088666152659985

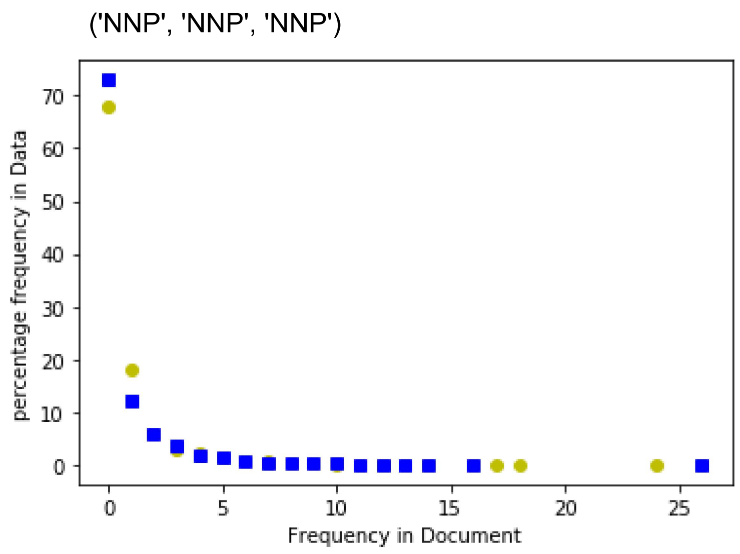
It saw a drop in accuracy and a drop in precision on both sides, there was a drop in ‘fake’ recall and f1_score but an increase in ‘real’.

My thoughts at this moment came to the conclusion that it was due to their being very little distinguishing between ‘fake’ and ‘real’. While the f1_score we were told to go by where ‘fake’ is considered positive is actually going to be highest here, however this would make the entire problem redundant. This algorithm is designed to catch fake tweets yes, but that is so journalists can verify ‘fake’ from ‘real’ which it is not doing at this point in time. In my opinion a better score might be the addition of both f1_scores to try and get a balance, of course it might just be because my algorithm is quite extreme in this case scenario.

Iteration 2:

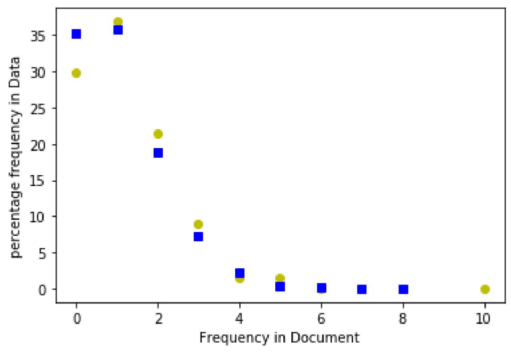
Changes:

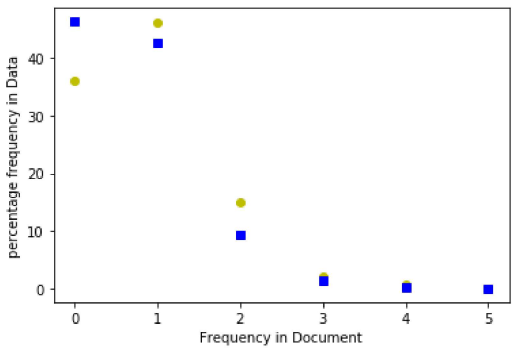
Here I thought the features were not diverse enough due to being trigrams if we go back to check on the percentage occurrences of these trigrams, as shown in fig. below, even the most frequent trigram for both ‘false’ and ‘real’ the percentage occurrence for 0 occurrences in entries is around 70% for both with only a slight difference, this likely meant that most entries only have 1 or 2 features where they are not 0.



To counteract this issue I added unigrams and bigrams to the potential features[1], they had far better variance in their use, the best unigrams have a percentage occurrence of less than 30% when frequency is 0 in entry, while the best bigrams have a percentage occurrence of less than 50% when 0 and percentage occurrence of over 40% when frequency is 1 in entry. :

Sample:





From this is picked a new list of Features
[('NNP', 'NNP'), ('NNP','NN'), ('NN','NN'), ('JJ','NN'), ('NN','IN'), ('NNP'), ('NN'), ('JJ'), ('IN'), ('.')]

I then trained and tested a polynomialfeatures SVC with C=1 and degree=2

TP: 1486 TN: 39 FP: 1088 FN: 119
Accuracy: 0.558199121522694

	fake	real
precision:	0.5773115773115773	0.2468354430379747
recall:	0.9258566978193147	0.03460514640638864
f1_score:	0.7111749222301986	0.06070038910505836

Iteration Evaluation:
In comparison to the first time polynomial SVC was run all scores have seen an increase, while the ‘fake’ scores saw a very small increase the ‘real’ scores increased by a noticeable margin. However, it's still a relatively poor result, while I would rather identify something as fake over being real to make sure that what the Journal is the thing being reported on is real, this algorithm is doing a horrible job of that because the ‘real’ precision is still only 25%, so of all entries it identifies as ‘real’ only 25% of them actually are. For the time being I’m aiming for a 50% precision rate for ‘rea’.

Iteration 3:
This time I changed from using LinearSVC to SVC with the kernel trick, the idea here is to lower the processing time in order to train with higher polynomial values. Along with this I also included a method to add a custom weight to the labels, I felt that since TP and FP are so dominant in the predicted results, its focusing too much on 'fake' so I added some more weight to 'real' entries to push back against the higher amount of 'fake' entries.
Unfortunately the time required to train at degree = 3 is still quite large thus very hard to optimise without a lot of time. Training at degree = 2 takes around a minute whereas degree 3 takes at least 10 minutes or more on my laptop, this is likely due to the many features which will be addressed in the next iteration.

After training and testing with degree = 2, r=5, C=10, weight=1.4

```
TP:  1398 TN:  95 FP:  1032 FN:  207
Accuracy:  0.5464860907759883
fake:
precision:  0.5753086419753086
recall:  0.8710280373831776
f1_score:  0.6929368029739778

real:
precision:  0.31456953642384106
recall:  0.08429458740017746
f1_score:  0.13296011196641008
```

A decrease in the 'fake' scores with a significant increase in the 'real' scores

Iteration evaluation:
During this iteration, having the weight definitely helped, the problem was that as i tried to increase the weight after a certain point the 'real' recall would continue to increase while the precision wouldn't resulting meanwhile the 'fake' scores where decreasing heading to a similar situation I started in just with 'real' being reclassified. Here I believe increase the degrees would significantly help just there are too many features to practically optimise it with the high training time.

Iteration 4:

I decreased number of features to only those with incredibly noticeable differences and very low 0 frequency probability occurrence

features = [('NNP','NN'), ('JJ','NN'), ('NN'), ('JJ'), ('IN'),]

I then increased the degree to 3 and due to the deceased number of features the processing time was significantly decreased allowing me to practically optimise it this resulted in the following:

Trained and tested with the following: degree = 3, r = 5, C =10, weight = 1.5

```
results
TP:  1199 TN:  437 FP:  690 FN:  406
Accuracy:  0.5988286969253295
fake:
precision:  0.6347273689782954
recall:  0.7470404984423676
f1_score:  0.6863194046937607

real:
precision:  0.5183867141162515
recall:  0.3877551020408163
f1_score:  0.4436548223350254
```

Final Evaluation:
As one can see there was a very large jump in the 'real' scores most importantly precision has eclipsed 50% and a large jump in both recall and f1_score. Accuracy has increased and the 'fake' precision is higher than the initial faulty fit that made everything 'fake', recall and f1_score for 'fake' has taken a hit, but overall this is definitely the best result so far. Not perfected in anyway and could certainly be improved but I feel that this algorithm as it is has passed the minimum requirement of such an algorithm, ideally I would like to increase the degree higher as well as r but I am satisfied with it for now.

Conclusion:
I can understand how having experience in Machine Learning is crucial for avoiding pitfalls. I almost didn’t realise that my algorithm was just classifying everything as ‘fake’, the f1_score we were told to use really shouldn’t be used by itself, you should have a variety of scores to keep each other in check, the highest ‘fake’ f1_score still remains to be the original fit yet that was actually the worst possible fit. This has been a great way to understand that there is never a perfect score for Machine Learning, in the future when using ML algorithms I will always make sure to have multiple scores to truly understand what my algorithms is doing.

Regarding the features, I feel I might have made some poor choices initially, I was being far too generally specific (if that makes any sense) and I rushed to a decision on that point, what I should have done is create as many possible features as I could think of within reason then look at them individually. I should have also tried my hand at some automatic features selection, I just didn’t feel comfortable doing it with such a large dataset, maybe if I experiment with some smaller datasets so that I truly understand what they do I can then apply it to larger datasets like this.

Overall the pre-processing is what I found the most challenging, maybe it seems that way due to the issues I suffered with the translators and the spellchecker, but I also believe that the preprocessing is where a lot of the thought goes in, how do you prepare the data and extract information from it. If I was making the machine learning algorithm myself this likely wouldn’t be the case but I’m not, in this practical case, the preprocessing is where I spent the vast majority of my time coding and thinking about the data. I could have tried NER, perhaps that would have resulted in better features for me to select from. For the trigrams I removed them because they were too specific, however if wildcards were to be used such as (‘NN’, *, ‘NN’) this is where trigrams would be used, without the wildcards however, I don’t think they should be used in text that is so small as these tweets. They could be used when looking at larger documents without wildcards however.

References:
[1]
“Learning Sentiment-Specific Word Embedding for Twitter Sentiment Classification”.
Duyu Tang, Furu Wei, Nan Yang, Ming Zhou, Ting Liu, Bing Qin.
Research Center for Social Computing and Information Retrieval Harbin Institute of Technology. China ‡Microsoft Research, Beijing, China \University of Science and Technology of China. Hefei, China.
Data: n/a

[2]
Automatic Text Classification: A Technical Review.
Mita K. Dalal Sarvajanik College of Engineering & Technology, Surat, India.
Mukesh A. Zaveri Sardar Vallabhbhai National Institute of Technology, Surat, India.
International Journal of Computer Applications (0975 – 8887) Volume 28– No.2, August 2011.