Machine Learning Engineer Nanodegree

Capstone Project

Adam Malpass, April 22nd, 2017

I. Definition

Project Overview

Project Domain

This capstone project is part of the field of Natural Language Processing (NLP) (http://sentic.net/jumping-nlp-curves.pdf). NLP exists to take in words and try to convey meaning from them. This could take the form of, for example, a chat bot trying to understand the written instructions of a user, or a home assistant such as Amazon Alexa or Google Home attempting to process spoken commands. 'Natural' is really the key here - it is important that the user can communicate in a colloquial way like they could could with another human, not in a strict robotic like manner that is required, for example, by traditional computer input systems such as the command line.

NLP is a rapidly-growing sub-field within machine learning and AI, so it will be exciting to be a part of it. Furthermore, the inspiration and data for this problem comes from a website I visit almost daily, Quora, so it gives extra motivation for tackling this challenge.

Datasets and Inputs

The datasets are provided by Kaggle and consist of 404290 pairs of questions with a human-generated label to show whether or not that pair is considered to be a duplicate. Each question is also given a unique id (which may, or may not, be useful). This data is ideal for addressing the problem at hand as we have many examples that fit exactly the input and output format that the final model is required to generate, and hence can be addressed as a supervised learning problem.

As this is a real, live, competition, Kaggle also provides another dataset, this time of unlabelled question pairs, which will be used to test the models created and judge the competition. By generating predictions as to how likely it is the items in the testing set are duplicates and submitting to Kaggle's site it is possible to get an unbiased score as to how well the model worked. As we learned during the Machine Learning Nanodegree it is vital to test the model on separate data than was used to train it to avoid overfitting.

However this submission is a manual process and it is only possible to submit predictions up to five times a day. Therefore to have better control over the testing/training process I will split the original labelled training dataset into training and testing subsets. As for the original testing dataset, it will still be interesting/fun to enter the competition as well, so this can be considered as 'competition testing dataset' and I will just use this as an input to my model once regular training and testing is starting to show good results to generate results for the competition.

The original training dataset (to be split into training/testing sets) can be downloaded from Kaggle here:

https://www.kaggle.com/c/quora-question-pairs/download/train.csv.zip

The original testing data set (to be used as competition testing dataset) can be downloaded from Kaggle here:

https://www.kaggle.com/c/quora-question-pairs/download/test.csv.zip

Problem Statement

Problem Overview

The problem I intend to solve is finding pairs of similar questions on Quora (https://www.quora.com/), a Q&A website. This is currently an active challenge on Kaggle (https://www.kaggle.com/c/quora-question-pairs), a website that organises machine learning competitions. Quora is a very popular site (100+ million uniques / month) and so often the same or very similar questions get posted multiple times, making it harder for people to find exactly the right information they are looking for. To help combat this issue Quora is searching for a solution to determine whether any given pair of questions can be considered duplicates of each other, making it possible to combine the answers to both questions into one set.

Specifically, I will design a model that takes in two questions (of written text), analyses them and outputs a continuous probability as to how similar they are believed to be, whereby 0 would indicate no similarity and 1 would indicate an identical question.

Solution Overview

The data provided is the ideal format for training a supervised machine learning algorithm. Since the final output is expected to be a prediction as to how likely the questions pairs are to be duplicates, it is necessary to use a regression technique to predict a continuous output. I plan to use regressors from the Scikit-learn Python machine learning library (http://scikit-learn.org/) to build the model. However these regressors can not directly process the textual information. I must first perform feature engineering to find useful characteristics about it to feed to the learner. I expect this to be the most important part, and biggest challenge of the project. The solution is quantifiable and can be evaluated based on the evaluation metric discussed later.

First it will be necessary to preprocess the training data to, for example, find a way to deal with NaN or other unexpected values. Next it will be necessary to generate some features from the raw text inputs, since the machine learning algorithms can't work with them directly. Some initial ideas for features I have are:

- Difference in total number of characters of both strings
- Difference in number of words of both strings
- Number of common words between both strings
- % of common words compared to number of words in longest/shortest string
- TF-IDF

I will make visualisations of how these features relate to the likelihood of being a duplicate question to get an intuitive feel for how the model should work. Some additional pre-processing of the text may be required to get better results, for example by removing 'stop-words' (common words that carry little-to-no meaning) or by performing stemming so only the root of words are considered.

Once I am happy the features can produce useful insight into the data it will be time to create the training, validation and testing sub datasets. Only then I can start to feed the data to the machine learning algorithm. As previously stated since a continuous output is required I need to use regression. The most fundamental type is linear regression, so this might be good algorithm to start with. Then I can try ramp up the complexity a bit with a Support Vector Machine based regressor. Alternatively it would be possible to use a Decision Tree Regressor. If this looks promising this could be expanded further using a Random Forest Regressor. Quora said they currently use Random Forests to tackle this issue, so it could be a promising algorithm to experiment with.

Once the model has been developed using the training data and hyperparameters optimised with the validation data, I can then use the model to generate predictions on the testing data. I can then compare these against the ground truth values and calculate the log loss score as the evaluation metric to see how well the model works. At this point it would also be possible to generate predictions on the competition test data and submit results to the Kaggle competition to get further feedback on how well the model is working.

Once this whole pipeline is in place it will then be required to iterate on the data processing, feature engineering and learning algorithm optimisation stages to improve the final score further.

Metrics

In terms of quantifying the solution, Quora have requested to use the log loss between the values predicted by the model and the actual, ground truth values. Since correct labelled values are available (after splitting the training dataset into training and testing datasets) I can calculate the log loss of the predictions and see how well the model is working. Log loss is a commonly used metric when evaluating a model that assigns a probability to an output condition, rather than simply picking the mostly likely output (in our case, whether or not the questions are duplicate). It is the entropy between the actual and predicted values. In mathematical terms for a single sample (http://scikit-learn.org/stable/modules/model_evaluation.html#log-loss):

$$L_{\log}(y,p) = -\log \Pr(y|p) = -(y\log(p) + (1-y)\log(1-p))$$

The final log loss score is then the average of the above calculation across all samples.

II. Analysis

Data Exploration

The training dataset provided by Quora has 404290 items with 6 features each and a sample can be seen in Table 1:

1. id - Row number
2. qid1 - Unique id for first question
3. qid2 - Unique id for second question
4. question1 - free-form text of first question
5. question2 - free-form text of second question
6. Is_duplicate - a human-generated binary label showing whether the texts of question1 and question2 can be considered duplicates or not

Table 1 - A sample of data from the training dataset provided by Quora, giving id, question ids, the free-form text of both questions and a label showing whether or not they are duplicate

	id	***qid1***	***qid2***	***question1***	***question2***	***is_duplicate***
0	*0*	*1*	*2*	*What is the step by step guide to invest in sh...*	*What is the step by step guide to invest in sh...*	*0*
1	*1*	*3*	*4*	*What is the story of Kohinoor (Koh-i-Noor) Dia...*	*What would happen if the Indian government sto...*	*0*
2	*2*	*5*	*6*	*How can I increase the speed of my internet co...*	*How can Internet speed be increased by hacking...*	*0*

3	*3*	*7*	*8*	*Why am I mentally very lonely? How can I solve...*	*Find the remainder when [math]23^{24}[/math] i...*	*0*
4	*4*	*9*	*10*	*Which one dissolve in water quikly sugar, salt...*	*Which fish would survive in salt water?*	*0*
5	*5*	*11*	*12*	*Astrology: I am a Capricorn Sun Cap moon and c...*	*I'm a triple Capricorn (Sun, Moon and ascendan...*	*1*

It was found that two *question2* fields (row ids 105780 and 201841) have missing (NaN) values - this will have to be accounted for later.

The most important features of the data are the free-form text inputs *question1* and *question2*, and the output label *is_duplicate*. I calculated that 36.9% of samples are considered as duplicates. However it is not really possible to calculate useful statistics about the raw text fields directly. It is first necessary to do some pre-processing and generate some features from it that can later be fed to the machine learning algorithm. So I will hold-off on a more detailed data exploration until later once these steps are completed.

Exploratory Visualization

Since the main input datas are raw text it is not really possible to make useful visualisations without first doing pre-processing and feature engineering. So I include visualisations later after these steps are complete.

Algorithms and Techniques

As mentioned in the Solution Overview section since this problem requires a continuous output is requires the use of a regression learner, not a classification one. I plan to start by trying linear regression, as it is the most fundamental and simplest to use. It works by fitting a linear model to the data by minimising least squares error. It doesn't really have any parameters that need tuning to work well; this is good at the start since it should be able to give a good baseline without too much difficulty. Training time can also expected to be quick even with the large dataset. However ultimately this lack of adaptability can be a bad thing as well as it may not be able to make a complicated enough model to accurately map the underlying patterns in the data. Thus I expect to use linear regression at the start but don't expect it to become part of my final solution.

Moving on from linear regression I plan to test using a Support Vector Machine based regressor. A support vector machine uses a 'kernel trick' to map data into a higher-dimensional space and hence allow modelling of nonlinear phenomena that linear regression cannot handle. Thus in theory it is possible to make more complicated models that better reflect the data.

However with this power comes downsides too. Many parameters need optimising to create the best model and also it doesn't scale well with large datasets. According to the scikit-learn documentation the SVM regressor doesn't work well with datasets greater than 100,000 items. In this case we have over 4 time this many items, so it is possible it will not work (or take a very long time to run).

A completely different approach would be to use a Decision Tree Regressor. Once the model is created using it works like a flowchart, with inputs starting at the top, having some aspect of it questioned and then based on the answer splitting it off to one of several separate next nodes, whereby another question is asked and it is split again, etc. By traversing through the sequence of nodes it finally reaches an output condition which is the final prediction. Unlike the linear regression and SVM techniques, for a Decision Tree the interaction between variables isn't directly taken into account, it merely considers on variable at a time. As to how the flowchart (or tree) itself is created, at every node the algorithm tries to split the data in such a way as to have the greatest entropy, or create the most information possible. Ideally each node would try and split the data half-and-half (or into 1/n amounts if there are n outputs). Decisions trees are a fairly easy to understand method and simple to implement so it makes a good choice. They also have flexibility to tune many parameters to increase complexity of model and/or prevent overfitting.

Following on from this the Random Forest Regressor should be able to further expand the power of the Decision Tree. It is a type of 'ensemble method' and essentially consists of creating many different Decision Trees and then averaging their outputs to produce the final prediction. U used 20 different trees to give a good balance between preventing overfitting and not causing too excessive run time. If the Decision Tree method seems to work well then it would make good sense to go one step further and try a Random Forest as it could well give better results.

A final option to consider, and one that has proved popular winning many Kaggle competitions is XGBoost. It is a powerful algorithm and supposedly scales well to large datasets. However since it is not part of the Scikit-learn library it is not possible / not easy to plug into the other powerful features Scikit-learn provides, search as GridSearch for parameter optimising, so there are some disadvantages for using this too. Since it is a popular algorithm I would like to at least try using it, but due to challenging integration it may not prove the best choice for this particular project.

Benchmark

Kaggle has a public leaderboard showing the best log loss scores of everybody who is competing based on their predictions on ~35% of the test data (https://www.kaggle.com/c/quora-question-pairs/leaderboard). At the time of writing (21/4/17) I calculated the following statistics about the submissions so far:

- Unique submissions (i.e. total number of competitors): 1731
- Total submissions (i.e. total number of submissions by all competitors): 6971
- Min (best) score: 0.13627
- Mean score: 1.831
- Median score: 0.364
- Max score: 28.521

The min and max score should give approximate bounds as to where the final result should lie within. The distribution of results is very skewed right so the mean doesn't make much sense here; the median gives a much better idea of the typical score. I will set a target to achieve a lower score than the median (0.364) and come in the top half of the competition!

III. Methodology

Data Preprocessing

As noted in the Data Exploration section, two of the *question2* fields have missing values. Initially I thought the easiest way to deal with this would be just to drop those items. In terms of making a good model I still believe this is a good strategy. However, it turns out the Kaggle competition data set also has lots of missing values and to create a valid entry for the competition a prediction has to be made for every single pair of questions. Thus if I simply dropped missing values from the training data, when it came to using the model to predict values on the competition test data the model wouldn't know how to cope with these values. Thus rather than dropping them, I decided to deal with them in the feature engineering stage; for every feature I developed I made sure that feature could generate an appropriate output if it encountered missing values. However it turned out there were some other issues with this, which I will discuss further shortly.

Another simple pre-processing step was to convert all the text in the *question1* and *question2* fields to lowercase. It is unlikely the case of the words will provide any useful information, however if words are different cases it could prevent the algorithm from finding words that almost certainly have the same intended meaning but just written in different cases (e.g. "Neural Network" vs. "Neural network"). It could also help match words which have typos (e.g. "Udacity" vs. "UDacity"). This step was done by applying the to_lower_case() function to the text fields.

Next I implemented the strip_punctuation() function. As the name suggests this removes all special punctuation marks (e.g: &@?!:;) from the question text strings. This will help ensure common words can be identified more easily some different punctuation won't stop them from being matched (e.g. "Hello world." vs. "Hello world!"). Arguably in real natural language the choice of punctuation does make a difference in tone and hence meaning, for example "Ok!!!" vs. "Ok…" suggest very different emotions, however this would likely take a much more sophisticated system to pick up on, so for this initial work it is probably safe (and indeed better) just to remove them entirely.

Another useful NLP preprocessing trick is to remove all the ‘stop words’ from the *question1* and *question2* fields. Stop words are frequently used words that carry little-to-no useful information since they are so common (e.g. “to”, “the”, etc). Often they are just required to form a grammatically correct sentence and are unlikely to provide useful context to compare the two questions in this case. Scikit-learn has a built-in list of English language stop words that I took advantage of to build the remove_stop_words() function that I then applied to all the text fields. One complication to this which I discovered later on is that for some questions *all* the words are stop words and so this function ended up removing everything. Thus even though I had planned a strategy to deal with missing values from the start, I ended up getting some fields with missing values in part way through. This actually led me to change my entire missing values strategy. At the start rather than ignoring or dropping NaN fields, I filled them all with the empty string (“”). Then I made sure that even if the remove_stop_words() functions removes all the words, it also returns the empty string rather than None. After this I had no more problems with missing values.

Finally I converted all remaining words into their roots, via a process called stemming (e.g. Running-->run, socks-->sock, etc). This removes grammatical suffixes and plurality indicators, such as “-ing”, “-ed”, “-s” and “es” from each word in the texts. Again this is done to try and match words that likely have the same intended meaning, even if not written in exactly the same way.

Implementation

Data Acquisition and Pre-processing

First I imported all the required packages and loaded the training data .csv file into a Pandas DataFrame to allow for easier data manipulation. Next I “dropped” (removed) the three ID-related fields (id, qid1 and qid2) as I don't believe they will provide any useful information towards solving the problem. As previously alluded to I then handled the missing values by filling them with the empty string. Then I separated the ‘is_duplicate’ column into a separate data structure since this is the value we are trying to predict. Next I applied the pre-processing functions described previously: converting all text to lowercase, stripping out punctuation, removing the stop words and converting remaining words to their stems.

Feature Engineering

At this point the question fields have been converted into a format more useful to start applying feature engineering. This is the process of taking the texts and calculating some useful characteristics about them that can be used to help us predict how likely they are to be duplicates. I devised the following features: differential number of characters, differential number of words, number of common words and ratio of common words.

Differential Number of Characters Feature

This feature, calculated by the diff_num_chars() function, calculates the length of each question in terms of number of characters in each and returns the absolute value of the difference between them. In theory if the two sentences are widely different in length there is a good chance (but by no means certain) that they could be discussing different things. To check how well this feature can distinguish between duplicate and non-duplicate questions I plotted a histogram (see Figure 1) which shows how many questions pairs of each type exist in the data set relative to this feature. For question pairs with a relatively low differential number of characters (approximately those which have a differential less than 20% of maximum difference between all pairs) this feature is not so helpful. However, as expected, as the differential increases there is a sharp drop-off in those questions that can were considered duplicates. Thus this feature could help to determine some pairs that are very likely to be non-duplicates, but may not help in trying to determine those that can be considered duplicates.

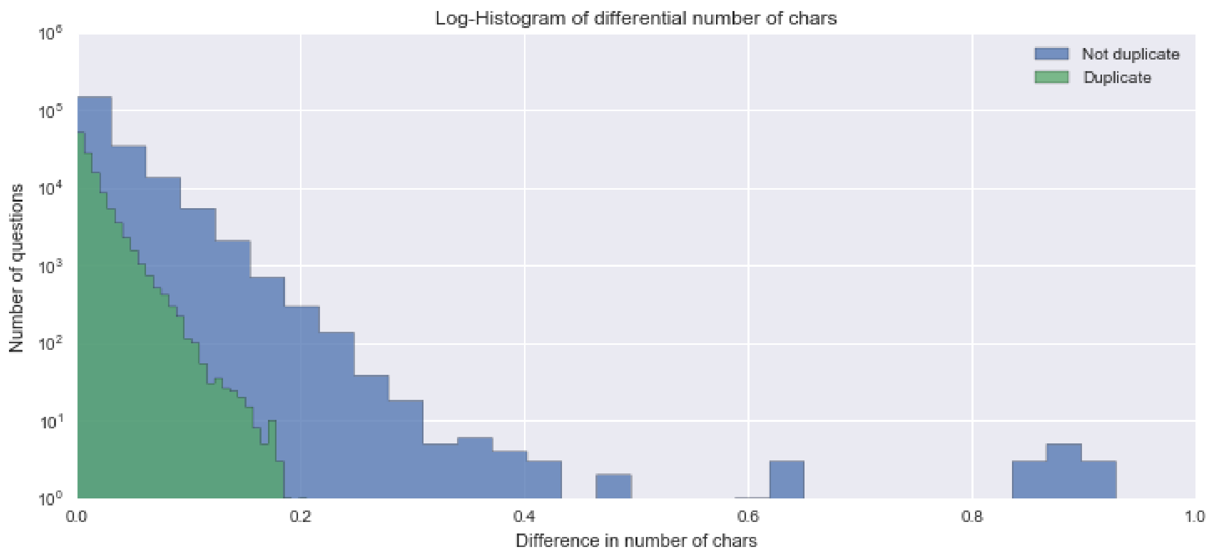

Figure 1 - Histogram showing how well the differential number of characters feature can distinguish between duplicate and non-duplicate questions

Differential Number of Words Feature

The differential number of words feature has a very similar to premise to the differential number of chars one, expect it calculates the absolute difference of length of the two questions in terms of number of words, rather than number of chars. It is calculated by the diff_num_words() function. I expect its predictive power to correlate strongly with the former feature, but it’s still worth to check. As shown in Figure 2 this is indeed the case; again the feature can clearly identify non-duplicate questions (those with large differences), but cannot really help us to determine those pairs that are similar.

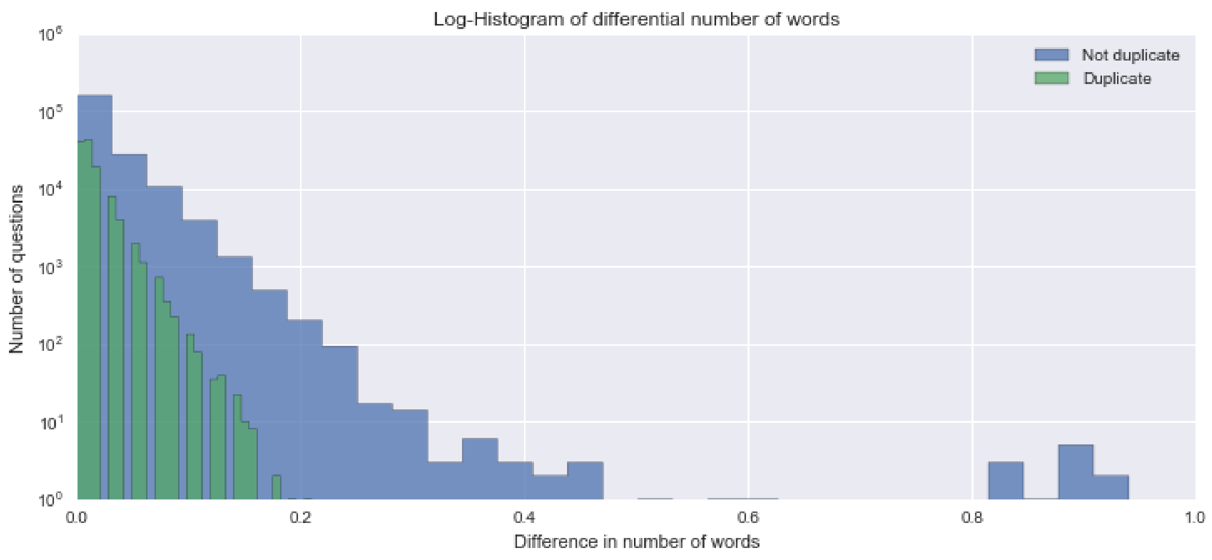

Figure 2 - Histogram showing how well the differential number of words feature can distinguish between duplicate and non-duplicate questions

Number of Common Words Feature

The number of common words feature looks at every word in the first question and sees if that word also exists in the second question, then returns the number of words that appeared in both. This is performed by the num_common_words() function. The theory is if similar words are being used by both questions then it is more likely they are discussing related things. However, as shown in Figure 3 it doesn't appear that this feature is particularly useful. There doesn't appear to be any correlation between number of common words and how likely it is for a question to be a duplicate. The reason is simple - all the questions are of different length, thus the absolute number of common words in each has little meaning. This led to the development of the next feature, the ratio of common words.

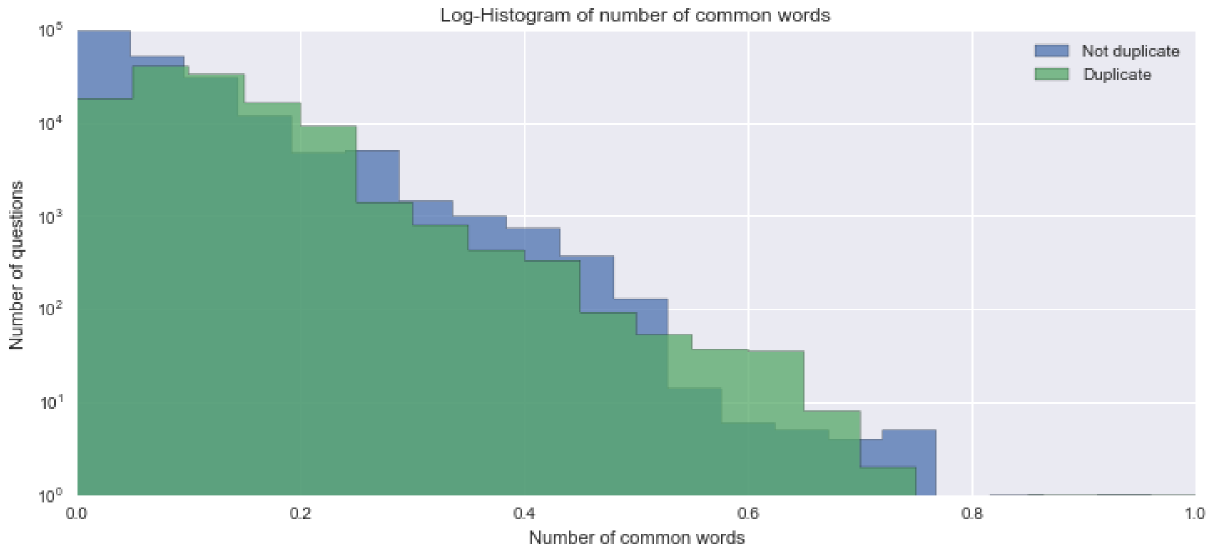

Figure 3 - Histogram showing how well the number of common words feature can distinguish between duplicate and non-duplicate questions

Ratio of Common Words Feature

The number of common words feature was not successful because it failed to consider the length of each sentence. So this feature improves upon it by dividing the total number of common words by the total length of the two sentence combined. As can be seen in Figure 4, this definitely is an improvement, with the feature well-able to distinguish a group of non-duplicate questions which have a low ratio of common words. However, similar to the earlier features it doesn't really help well to identify pairs that can be considered likely to be duplicates.

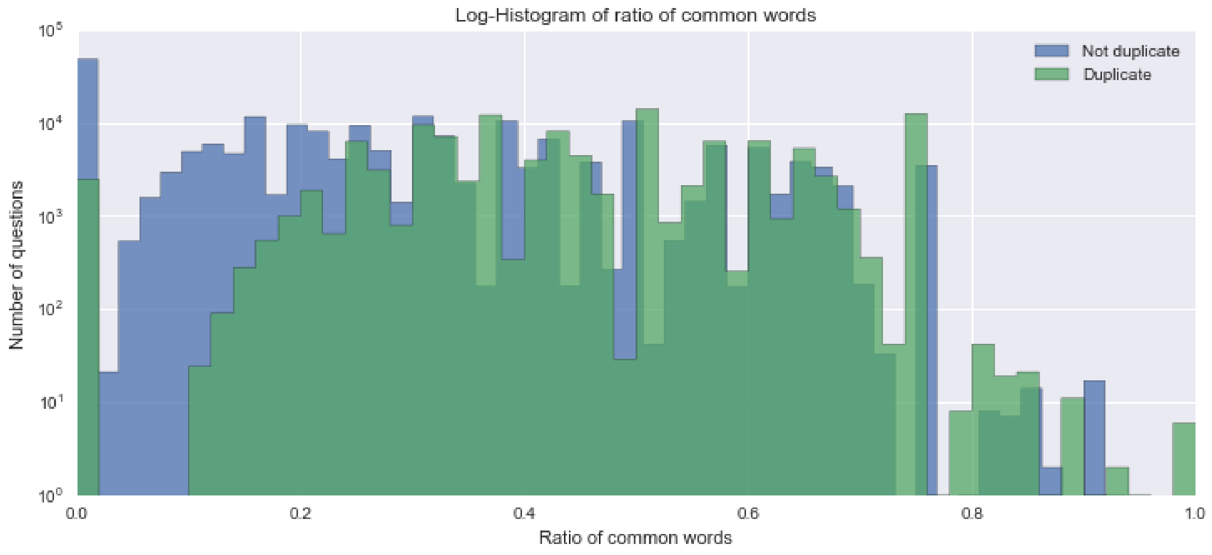

Figure 4 - Histogram showing how well the ratio of common words feature can distinguish between duplicate and non-duplicate questions

Feature Selection and Scaling

I now have four potential features that can be fed to the machine learning algorithm, however it is not necessarily wise to use them all. First, the number of common words feature seems to have no predictive power, so I dropped it. It is more likely to cause problems via overfitting than contributing anything useful. Second, the differential number of chars and differential number of words functions appear to be highly correlated to each other so it doesn't seem necessary to keep both. From the histograms I judged the differential number of words feature to be marginally better, so I dropped the differential number of chars one.

So the final two features are differential number of words and ratio of common words. For learning algorithms such as decision trees the data is now in a usable state. However since I will also try using algorithms such as linear regression it is first important to scale the data. If I didn't scale the data first the linear regressor would likely falsely think that the variable with largest absolute variation (in this case differential number of words) is more significant that the one with lower variation (ratio of common words). I applied the MinMaxScaler from Scikit-learn to do this. I initially had a misconception as to how this scaler worked, and applied independent scalers to both the training and competition testing data sets. This gave very bad final results. I realised then that they had to be a single Scaler, created from the training data, and then the issue went away.

Building the Model

Finally I have converted the data into a format useful to the machine learning algorithms. Next step is to split this data into train and testing sets using train_test_split from Scikit-learn. This is necessary so we can see an unbiased view of how well the algorithm is working, by holding back some data samples the learner doesn't see, and seeing how well it models them. To evaluate the performance of the learner I created the performance_metric() function, which returns the log loss of the difference between predicted and actual values, as discussed in the Metrics section.

Now it is simply a matter of calling the learner algorithm. Since I wanted to try several different algorithms I made it simple to switch between them by setting the 'algorithm_choice' parameter. However the implementation of them varied significantly between those in Scikit-learn (linear regression, SVR, decision tree and random forest) and that for XGBoost, which is a stand-alone library. I will describe these separately below.

Scikit-Learn Algorithms

The fit_model() functions creates a pipeline to take the training data and feed it to the chosen Scikit-learn algorithm. First it makes some cross-validation sets to help us to determine the best hyperparameters to tune the learners. Then it creates the regressor itself and the potential parameters based on the choice of algorithm. Finally it runs grid search on the cross-validation sets to choose the best hyperparameters and create the best model.

To check how well this model works, I use it to create predictions from both the training and testing data. The testing data score is most important final result, however by comparing with the training data score we can check for overfitting.

XGBoost

Since it XGBoost is not part of the Scikit-learn library, it is not so easy to integrate it with many of the useful functions it provides, such as grid search. However the algorithm still requires training and validation data sets to be passed to it. In this case, for simplicity I decided to pass the training data as training data and the testing data and the validation data. For offline testing I will then use the validation test results to get a good, but not completely unbiased way to see how well the model is performing. For a final, truly unbiased check of the model I can use the competition data set and check the log-loss on that.

Creating Kaggle Competition Entry

The final step is to create the predictions for the Kaggle competition. First I load in the .csv file of competition test data. Then I run the same pre-processing and feature engineering steps as performed on the training data. Next I use the model created in the previous step to generate the predictions. Then there is a final bit of post-processing to create an output .csv file in the format required by the competition, which is then saved to disk.

Refinement

The implementation discussed above is the final solution I developed. However it didn't start off that way and I gradually added more-and-more complexity to incrementally improve it. An abridged summary of results as the project progressed is shown in Table 2. The very first implementation used a Decision Tree with the differential number of words and number of common words features. None of the pre-processing functions were yet implemented at that time. It scored 0.52454 on the competition data set (regrettably at that stage I hadn't started keeping records of the score on training and testing data sets - something I would improve upon next time).

The best model I produced (in terms of competition test score) used XGBoost algorithm, with stop words removed and scaling implemented, but without removing punctuation and without implementing stemming, for a final result of 0.43672. However the scores are not that much worse if I included these preprocessing steps, and in theory I am sure it is the right thing to do, so it may just be for this particular sampling of the data it makes it slightly worse, but overall it could actually improve things. For this reason I decide to keep all the preprocessing steps for the final model which had a score of 0.48691.

Table 2 - A summary of training, test and competition scores obtained throughout the project after various changes to pre-processing, feature extraction and algorithm choices. Some scores not recorded after all changes.

Change	**Algorithm**	**Training Score**	**Test Score**	**Competition Test Score**	**Comment**
Initial	Decision Tree			0.52454	
-	Random Forest			0.52412	
Remove stop words	Random Forest			0.57016	Makes it worse...
Change grid search parameters	Random Forest			0.48726	
Make all words lower case	Random Forest			0.47957	Slight improvement
Add common_words_ratio feature	Random Forest		0.486668430285	0.44546	Big improvement
Make diff_num_words feature return absolute value	Random Forest		0.480668215427	0.44326	Minor improvement
Implemented minmax scaling	Random Forest		0.480969668635	-	(still random forest so shouldn't change much)

-	Linear Regression		0.551160483704	-	(NB: may include broken scaling)
-	XGBoost		0.472746 (validation score)	0.54968	(NB: may include broken scaling)
Check again random forest with scaling in case scaling caused issue	Random Forest		0.480969668635	0.93738	Didn't expect this - scaling must be working poorly
-	Decision Tree		0.491932315566	-	
Check with random forest - hopefully with fixed scaling	Random Forest		0.48167031195	1.00391	Scaling still not fixed...
Found another bug with scaling - fingers crossed	Random Forest		0.481808461668	0.44360	Scaling fixed!
Re-check now scaling fixed	XGBoost		0.472746 (validation score)	0.43672	Best entry!
Re-check linear	Linear Reg		0.551160483704	0.46455	
Re-check	Decision Tree		0.491938805072	0.45590	
Implement stemming	XGBoost		0.494345 (validation score)	0.45138	Slight worse

-	Random Forest		0.500118700284	-	
Increase hyperparams to Random Forest			0.5204	-	
Fixed handling NaN values (included after stop words removed)	XGBoost		0.503703	0.45595	
Remove punctuation	Random Forest	0.538936013406	0.548402532131	0.49217	Worse performance
-	XGBoost	0.542988	0.542235	0.48691	Worse performance

IV. Results

Model Evaluation and Validation

Using techniques inspired from the Boston Housing Project I undertook a sensitivity analysis to examine how robust the final model is to small perturbations in the training data. To do this I rebuilt the model a further 10 times using a different random split of the data into test and training buckets. I then used these new models to predict outputs on the testing subset and calculated the log-loss scores as before. I then calculated the following statistics across the 10 runs:

- Raw scores: [0.54182549758582144, 0.54287802226766191, 0.53987818699247148, 0.54015756317957053, 0.54320092540612464, 0.54334852560160252, 0.54045025604419661, 0.5426514827439366, 0.54057549019702678, 0.53924420927846661]
- Mean score: 0.54142101593
- Median score: 0.541200493891
- Standard deviation: 0.00145107899294

As can be seen, the final scores are always almost exactly the same, about 0.54. In addition the standard deviation is extremely low, at 0.001. Thus it seems the model is very robust to changes in the input data and doesn't lead to big swings in the predicted outputs.

Justification

The bench mark set earlier was based on the previous scores submitted to the Kaggle competition leaderboard. I expected my result should be somewhere between the upper and lower bounds (28.521 and 0.13627 respectively) and I challenged myself to come in the top half of the leaderboard (i.e. greater than the median score of 0.364). My best score was 0.43672 which is certainly within expected range, and close to the median, however unfortunately slightly below it. However, realistically many of the people taking part in the competition are professional data scientists with lots of experience and also many teams are taking part too. Therefore for my first Kaggle competition, and indeed first fully-independent machine learning project I think I have done well to get a score close to the median.

I have produced a solution that can well determine how likely it is the pairs of questions can be considered duplicates.

V. Conclusion

Free-Form Visualization

Once the model had been completed and successfully outputting predictions, I thought it would be interesting to see what kind of results it is predicting. A log-histogram showing the predicted likelihood (using the random forest version of the model) that the questions are duplicates for all items in both the training and testing datasets is shown in Figure 5. As can be seen the amount of question pairs predicted between around 0 and 0.6 chance of being duplicate ones is fairly consistent. However the amount of questions pairs with a prediction higher than this suddenly drop-off very quickly (especially considering this is a log plot for y-axis). Given that in the training data only 36% of the question pairs are marked as duplicates it is reasonable to expect we would see more lower predictions than higher ones. However the rate of drop-off is much higher than I would expect if the model was working well. This suggests whilst the model is able to predict questions that are not similar with good confidence, it is very indecisive about those that are duplicates and is not doing a good job of identifying them.

This makes sense given the distribution of the differential number of words and ratio of common words features shown previously in Figure 2 and Figure 4 respectively. In both cases those features do a good job of identifying non-duplicate questions, but a poor job of working out which ones really are the same. So it makes sense the final model doesn't do a good job of predicting those. This is something that would be best to improve next time.

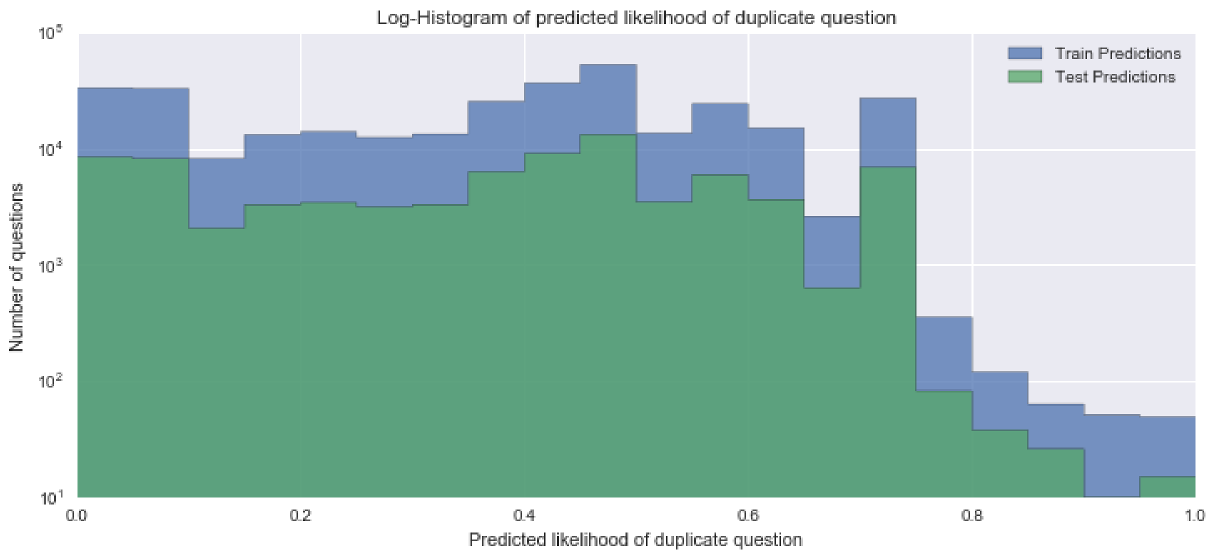

Figure 5 - Histogram showing the distribution of all predictions outputted from the Random Forest model for both training and testing sets

Reflection

This was an interesting and challenging project and it was a great experience to go from the raw data to a working model independently. I had to combine together many of the skills I had learned (and sometimes forgotten again…) over the course of the whole Nanodegree. The initial data-wrangling in Pandas was more complicated than I expected, but now I have got through it once myself I have a lot more confidence that I can do it again in the future with much fewer difficulties.

Probably the most important and challenging part to the project was coming up with useful features from the raw text. Most of the projects I did so far had many existing numerical features so it is tempting (and to some extent possible) just to throw the data at the machine learning algorithm without first really understanding it and just hope the learner can 'magically' go away and process it and come up with some kind of suitable model. However in this case that is not possible and I first really had to understand the problem and think carefully about how to tackle it. It was exciting when I had the idea for the ratio of common words feature and saw a significant improvement in the model. It really made me realise that whilst the algorithms are clever and important, the most vital thing is having data and features that can offer something useful to separate the data. Even the world's best algorithm can do nothing with data that has no helpful characteristics (for example the number of common words feature). Thus I've learned to really fully analyse the data first before sending to a machine learning algorithm, whether that data is already numerical, or text, or whatever type it is.

Another struggle I had was trying to use the Support Vector Machine Regression technique. In the end I realised that the dataset was too large for it to handle it in a realistic time, but since this was near the start of the project I assumed something else was wrong in my implementation and it led to lots of red-herrings in my debugging. Finally I tried using a massively reduced version of the training data and the model completed successfully. Of course the score was not great, but at least I realised that the actual code implementation was correct. Next time I should start by using a small version of the training data to make sure everything is working correctly, before going ahead and making the full-blown model on all the data.

Overall I think the final model worked well and produced a score I can be proud of, even though it didn't quite reach the top-half of the leaderboard. I was surprised that some of the pre-processing steps I made, such as removing punctuation or implementing stemming had very minor or even negative effects on the score. I think this is something I need to understand better next time, but overall I still feel it seems the right thing to do to include them and I am happy with the final mode.

Improvement

The XGBoost algorithm seems like it has a lot of potential as it has a good history of being used by winning Kaggle teams in the past. However since it is from a stand-alone library it is not easy to use some of the nice features of Scikit-learn such as grid search together with it. XGBoost has many hyperparameters to tune, and I could by no means yet claim to be an expert on these, so it would be good to be able to search through them automatically using a separate validation data set and get some more insight into what seems to work well. I think it must be possible to integrate it in some way, but I didn't explore this further this time. Next time I would like to try this. I expect this can help improve my final score further.

In more general terms, next time I would also be sure to keep better records of the train, test and competition test scores after every change I made. This time I only kept test score from part way through and train scores only at the very end. Keeping detailed records of every experiment right from the start would make it easier to tell exactly what effect changing certain parameters or pre-processing steps has. Finally I would like to try running the model building algorithms in the cloud, for example on AWS. I wasn't able to successfully train using the SVM regressor due to a lack of processing power, so this would be another good thing to investigate using in the future.