|  |  |
| --- | --- |
| **Capstone Project**  **Machine Learning Engineer Nanodegree**  **“Learning” from Chinese Learning Log Data** | Jed Isom  April 28th, 2016 |

**Definition**

**Project Overview**

In 2011, I was interested in learning Mandarin Chinese and decided to read a book in Mandarin. The book I decided to read had 3 columns on each page. In the 1st column, it had traditional Chinese characters, in the 2nd column it had the pinyin pronunciation of the characters found in the 1st column, and the 3rd column had the English translation of the first 2 columns. While reading, if I didn’t know a Chinese character, I would at the pinyin to see if it would jog my memory. If that didn’t work, I would look up the character in [this iOS app](https://itunes.apple.com/ca/app/pleco-chinese-dictionary/id341922306?mt=8), and then write the character, its pinyin and the English translation in a notebook.

A couple of days into this reading process, I decided to track my progress because I’m a data driven guy and thought it would be interesting. I did this by recording how long I read each day (in minutes) as well as how many Chinese characters I read. I expected the average seconds taken to read a character to follow a traditional logarithmic scale [experience curve](https://en.wikipedia.org/wiki/Experience_curve_effects). As I plotted my progress, it mostly followed the expected curve, but there were some very obvious deviations from the curve. I found that there were certain sections of the book, or periods of time, where I read much slower than I had previously. There were also instances where I read quicker than I would have expected.

In this project I sought to explain the variance from the experience curve and my actual reading speed performance. The logarithmic experience curve was the baseline performance model for this supervised learning project.

Manually entering all of this data would have been very time consuming. Luckily, the book I read can also be found and read online (without the 2nd and 3rd columns) for free [here](https://www.lds.org/scriptures/bofm?lang=zho). To create the dataset, I used my handwritten notes and this online text to create a raw dataset where each row represents a date/study session. The columns in the raw dataset include

* Date
* Time spent reading (in minutes)
* Text read; copied and pasted text from the book that represents the text read during the study session. This was recreated digitally for this project based on my detailed notes I took while reading the book.

I used the information within the ‘Text read’ field to help improve the prediction of my reading speed. Some features I explored to explain some of the error in the baseline model include

* % of characters in study session text not seen before in the book
* The number of days since each character in the text was last read
* Latent Dirichlet allocation (LDA) to create topics and use the posterior probabilities of the text being in a certain topic as a predictor
* …

I **could not** use the number of characters in the text read field or the time spent values as predictors in this project. These 2 features are what I used to create the actual output (y variable) and so would easily fit the data, but not solve the underlying question about why my reading speed did not follow the traditional experience curve rate.

**Problem Statement**

I plan to determine what underlying variables in the text that I read could have better predicted my Chinese character reading speed while reading this book. I did this by treating this project as a supervised learning problem, where my output (y) was my reading speed in seconds per Chinese character, and my inputs (X) were features I derived from the text data and the cumulative characters read.

Tasks that were required to complete this project include

1. Use hand-written notes to create the digital dataset for analysis
2. Import the MS Excel entered dataset into Python
3. Scrub the Chinese Unicode text by removing numbers, punctuation, non-Chinese characters, etc.
4. Calculate preliminary characters per study session and seconds per character metrics
5. Validate data entry against handwritten notes to make there were no big mistakes in the data entry, go back and correct #1 as necessary
6. Create X features from the dataset to predict seconds per character. Potential features to create that might help explain the variation are listed again here.
   1. % of characters in study session text not seen before in the book
   2. Get the number of days since each character in the text was last read, and then take the mean.
   3. Find the frequency of characters in the current study session text based on frequency of those characters in all previous study sessions
   4. …
7. Segregate training and test set data. This segregation will be random and static so that the same examples are always in the training set and test set. This will prevent data snooping.
   1. Use Latent Dirichlet allocation (LDA) to create topics from test set and use the posterior probabilities of the text being in a certain topic as a predictor
8. Perform supervised learning on training set with cross validation to estimate generalized model performance
9. If cross validated model performance is worse than the current best solution, then go back to #6 or #8 and try to create new/better features or learning models for predicting reading speed.
10. My original goal was to reduce the baseline error by at least 70% by doing this analysis. I worked creating new features/models until I at least reached this goal and then look for a plateau in model improvement to determine when to stop.
11. Once a final model was picked, it was used on the test set to verify how well it generalized on unseen data.

In the end, I believe that using supervised learning I will be able to determine what features of the text I read have the most predictive power for my Chinese character reading speed. This will be valuable because I can use this information to improve my study habits based on this information. In a future project, this type of information might also be used to help someone else learn to read Chinese characters faster.

**Metrics**

As mentioned previously, the output variable, y, is the seconds per character reading speed for the study session. Since this value is very large in the beginning of the book and decreases exponentially due to the experience curve effect, I plan on taking the natural logarithm of this value for model fitting purposes. This will help the model error to be more uniform across the study sessions and thus not over fit study sessions near the beginning of the book.

The project is a regression problem. Kaggle has a [list of metrics to consider using for regression problems](https://www.kaggle.com/wiki/Metrics):

* Mean absolute error
* Weighted mean absolute error
* Root mean squared error
* Root mean squared logarithmic error

Based on what I know about the problem either the mean absolute error or root mean squared error should work fine. I have no reason to believe that a more complicated metric would be required. I used root mean squared error because it is more commonly used, and also because it penalizes predictions farther from the actual values much more than predictions closer to the actual values due to the squared term. This helps incentivize the model to fit these data points farther from the predictive model better, which is essentially what this project is about; fitting the data that varies from the baseline model better.

I also wanted this model to generalize well because I wanted to use the information about what features work well in predicting reading speed to help me (or other people) understand how to learn better/faster and improve reading speed in the future. So I used the mean cross-validation root mean squared error of the model applied to my training set as my target to optimize. When I gained confidence that the model generalized well in cross-validation I ran the model on the test set and reported the root mean squared error on the test set using the model created during training set cross-validation.

**Analysis**

**Data Exploration**

As described previously, I copied and pasted text that I read during each study session into a digital dataset (Microsoft Excel format) for this project’s analysis. Below is a sample row/instance of one study session of data.

|  |  |  |
| --- | --- | --- |
| Date | Time\_Spent | Text\_Read |
| 2011/09/07 | 30 | 他​心中​愚蠢​的​幻想。   12 最年長​的​拉曼​和​雷米爾​這樣​抱怨​他們的​父親。​他們​抱怨，​因為​他們​​不​知道​創造​他們的​神​的​作為。   13 他們​也​不信​耶路撒冷​那​座​大​城 |

In Excel, I was able to create a custom date format YYYY/MM/DD that would make it easier to transfer the dataset to the Python ‘datetime’ package that takes (YYYY, MM, DD) format. The “Time\_Spent” column is the time spent (in minutes) reading during that study session. It was too much effort to record this data in a more granular form like seconds, so this will be a somewhat lumpy source of error because the seconds read during a session will jump in increments of 60 seconds. The “Text\_Read” column includes the unformatted/unfiltered text I read during the study session and copied into the dataset. Since the text data are Chinese characters, the “Text\_Read” feature is Unicode text, not ASCII text/characters. While Unicode text is the future of digital text representation, figuring out how to use this text in a machine learning problem was a bit of a challenge for me (more discussion on this point later).

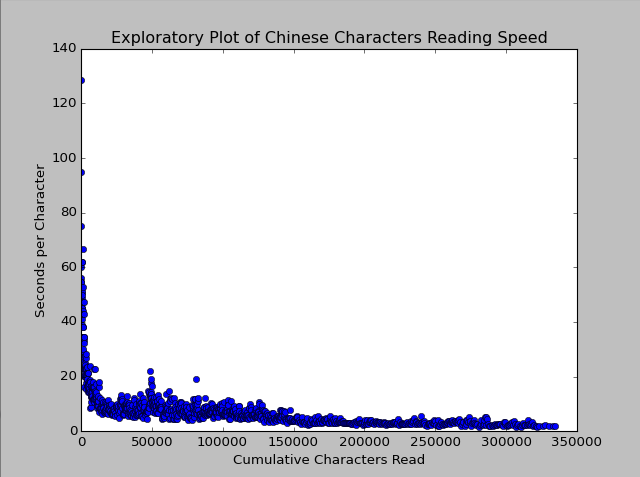
At a high level, this dataset contains 974 records (study sessions) spanning 29,227 minutes of study time. After preprocessing the text data (see the discussion on how this was done in the preprocessing section below) I was able to count 334,195 total characters read. This gives an overall average reading speed of 5.247 seconds per character when not taking the natural log of the reading speed.

The majority of the values for “Time\_Spent” are 30 minutes because of the habit I developed of reading for ½ hour in the morning before going to work. There are a few outlier days where I spent an hour, or an hour and a half studying. I am accounting for this variation by using reading speed as my output metric instead of characters read during the study session. Essentially this normalizes the output variable.

As discussed previously, because this dataset represents an experience curve process, the reading speed values are exponentially larger in the beginning of the dataset compared to the end of the dataset. To deal with this order of magnitude problem, I plan to take the natural logarithm of the reading speed and use that as my output variable for this supervised learning problem.

**Exploratory Visualization**

To explore this data, I created a simple plot with “Cumulative Characters Read” along the X axis and “Seconds per Character” on the Y axis. The reading speed reduction is dramatic in the beginning of this chart and definitely shows the characteristic pattern of an experience curve. Some of the anomalies that got me interested in this problem can also be seen in this chart. Look at the spike in the chart around 50,000 characters read. Why would I start reading ~1/2 as fast all of a sudden? I thought the answer to this and other similar questions could be found in the latent information from the text data not represented in this chart.



To visualize the experience curve baseline model I used for this project, I plotted the natural logarithm of the reading speed against the natural logarithm of both the cumulative time spent reading and the cumulative characters read. Taking the logarithm of both the X and the Y axis allows for a linear fit to most [experience curves](https://en.wikipedia.org/wiki/Experience_curve_effects). Both of these charts (see below) have roughly the same shape with a somewhat flat beginning section and then follow a linear downward trend fairly well. After viewing both of these charts, I decided to use the line fit to the natural logarithm of the “Cumulative Characters Read” as the baseline experience curve because the data fits the linear model better. Also, experience curves are normally based on “units produced” on the X axis. For my case the equivalent metric would be Chinese characters read.



**Algorithms and Techniques**

As discussed previously, this is a regression problem. Since this regression problem can be transformed into something that should be roughly linear, I started my supervised machine learning with simple linear regression. I used this technique as a quick way to determine if features I was creating had any additional predictive power when cross-validated. The only input variables to this model were the features I created and the output variable to be trained.

As I was creating features, I decided to try topic modelling with sklearn’s LatentDirichletAllocation package. This is an unsupervised learning method for finding topics in document corpora. Here’s a quick description of the method from Wikipedia.

*“*[*Latent Dirichlet Allocation*](https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation) *(LDA) is a generative statistical model that allows sets of observations to be explained by unobserved groups that explain why some parts of the data are similar. For example, if observations are words collected into documents, it posits that each document is a mixture of a small number of topics and that each word's creation is attributable to one of the document's topics.” – Wikipedia*

In my case I substituted “words” as tokens for Chinese characters/words as tokens. I wanted to be able to use the probability that a document (study session) contains a given topic as a predictor for my models. In order to do this, I mostly set the LatentDirichletAllocation to its default values. The only differences were

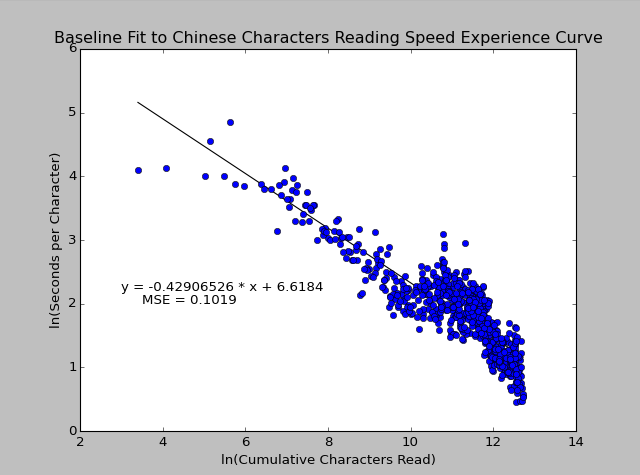
* N\_topics: I allowed the function to vary the number of topics from 10. If 10 topics didn’t predict well, maybe 3 topics would…
* Learning\_method: I set this to batch. The default/alternative is ‘online’ which uses a sampling of the training data during each EM update. I don’t have a continuous stream of data, which is what I think ‘online’ is for, and I wasn’t too concerned about processing time so I decided to use all of the data in each EM update
* Random\_state: I set this to be 0 arbitrarily. I just wanted to have a static number here so that the analysis would be reproducible.

When I had a set of features created that performed fairly well in a linear regression model, I took that same set of features and tried several more complicated models. Below is a summary of the models used, a discussion of why they were tried, and how the parameters were set in each model

1. Decision Tree
   1. Summary/Discussion
      1. Decision trees are a simple way of modeling data. Essentially splitting the dataset by the feature that reduces the mean squared error the most, and then splitting it again… This model is also very fast to fit/find because it is O(lnN) fast. For these reasons I decided to use [Occam’s Razor](https://en.wikipedia.org/wiki/Occam%27s_razor) and see if the simplest model would work best.
   2. Parameters
      1. Splitter: This is the strategy used to split the decision tree at each node. I tried 'random' and 'best'. The ‘best’ feature is the feature that would reduce the MSE the most by splitting the tree with that feature
      2. Max\_features: When creating branches in the decision trees, this is the number of features that the algorithm will consider splitting on. I used the 3 built in recommendations for this value as paramaters; ‘auto’ = sqrt(n), ‘log2’, and None.
      3. Max\_depth: This is how many layers/splits there are in the decision trees that were created. I explored 2, 3, and 4 for this parameter
      4. All other parameters I left as their [default sklearn values](http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html). While I could have explored more, this model didn’t prove to be very promising initially so I stopped trying to tune its performance.
2. Random Forest
   1. Summary/Discussion
      1. I tried this because I’ve read lots of things online about this model. Many people on Kaggle say that this model tends to perform and fit data well. This model creates many decision trees, randomizing things like what features the trees can use as predicting features, and then uses the average prediction from those many trees as the Random Forest model’s predicted output. Although this model is non-linear, it might do well explaining the non-linear effects that don’t follow the experience curve linear model.
   2. Parameters
      1. N\_estimators: This is the number of ‘decision trees’ in the random forest. Values of this parameter I tried include 3, 5, 10, 20, 30, 100, 120, 150
      2. Max\_features: When creating branches in the decision trees, this is the number of features that the algorithm will consider splitting on. I used the 3 built in recommendations for this value as paramaters; ‘auto’ = sqrt(n), ‘log2’, and None.
      3. Max\_depth: This is how many layers/splits there are in the decision trees that were created. I explored 2, 3, 4, 5, 6, 7, 9, 12, and 15 for this parameter
      4. All other parameters I left as their [default sklearn values](http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html). While I could have explored more, it would have made the GridSearchCV package run way too long to get results
3. Ridge Regression
   1. Summary/Discussion
      1. Normal linear regression can over fit data by having coefficients for features that aren’t very helpful in the models performance. Ridge regression uses a regularization parameter to penalize the models performance for large linear regression coefficients thus forcing a balance between fitting the data and model generalization. I wanted to try this model to see if it would help the model generalize better.
   2. Parameters
      1. Alpha: According to the sklearn documentation for this parameter, alpha is roughly equivalent to 1/C for models like support vector machines (SVM). The default value is 1. To explore the space I tried values of 0.1, 0.3, 1.0, 3.0, and 10.0.
      2. I read through the rest of the parameters for this model in the sklearn documentation and determined that varying the other parameters would not likely lead to better model performance. See the [sklearn Ridge documentation](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html#sklearn.linear_model.Ridge) for the other default values.
4. Bayesian Ridge Regression
   1. Summary/Discussion
      1. This model is similar to ridge regression because it has regularization parameters that can help with generalization, but these parameters are starting points and are tuned to the data during the algorithm. A good overview of the model can be found [here](http://scikit-learn.org/stable/modules/linear_model.html#bayesian-ridge-regression). I wanted to try this model to see if the regularization parameter tuning would improve performance over normal ridge regression.
   2. Parameters
      1. Alpha\_1, Alpha\_2, Lambda\_1, Lambda\_2: The default value in sklearn for all of these parameters is 1e-06. Since I’m not very familiar with this model, I tried values of 3e-07, 1e-06, and 3e-06 to see if performance would improve.
      2. I read through the rest of the parameters for this model in the sklearn documentation and determined that varying the other parameters would not likely lead to better model performance. See the [sklearn Bayesian Ridge documentation](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.BayesianRidge.html) for the other default values.
5. Support Vector Regressor (SVR)
   1. Summary/Discussion
      1. According to the [sklearn documentation for SVR](http://scikit-learn.org/stable/modules/svm.html) “The model produced by support vector classification…depends only on a subset of the training data, because the cost function for building the model does not care about training points that lie beyond the margin. Analogously, the model produced by Support Vector Regression depends only on a subset of the training data, because the cost function for building the model ignores any training data close to the model prediction.” In my past, support vector machines have been very effective at fitting data well, even if it is not clear that they fit the problem domain well. They tend to take a while to train, but I wanted to try it to see if the model showed promise for future tuning.
   2. Parameters
      1. Kernel: In my past I have tried to use this model for other supervised learning problems. The 'linear' and 'rbf' kernels have tended to be the kernels that have fit my past data the best, so I tried them. If the initial SVR model performance had been better, I might have explored more/different kernels.
      2. C: This is the penalty parameter on the error term. The default value is 1.0. In other data mining classes I’ve taken they recommended varying this parameter by rough thirds so I tried values of 0.3, 1.0, 3.0, and 10.0.
      3. Gamma: This is a kernel coefficient for the ‘rbf’ kernel. The default value is 1/n\_features. Similar to C, I tried values that were rough thirds of each other; 0.3/n\_features, 1.0/n\_features, and 3.0/n\_features.

**Benchmark**

As discussed previously, I used a traditional experience curve model as the benchmark for this project. [Experience curves](https://en.wikipedia.org/wiki/Experience_curve_effects) look non-linear on a standard scale, but *should* look linear if both the Y and the X axes are logarithmic scale. So, my baseline model is a linear regression model with X = ln(cumulative characters read) and y = ln(seconds per character). This baseline model along with its linear equation (y = -0.42906526 \* X + 6.6184) and the 10-fold cross-validated mean squared error (MSE = 0.1019) are shown in the plot below.

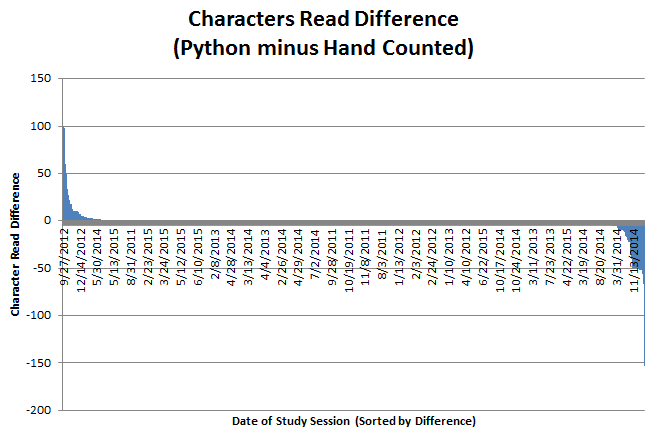


**Methodology**

**Data Preprocessing**

In order for supervised machine learning to be possible in this project several transformations from the raw dataset were required.

1. The raw data needed to be imported into a pandas dataframe from a Microsoft Excel file
   1. The ‘Date’ field needed to be parsed and formatted to be turned into a python datetime variable type
   2. The ‘Time\_Spent’ field was read in as text and transformed to an integer (only whole numbers were used in this field)
   3. The ‘Text\_Read’field was copied in as Unicode text
2. The Unicode text in the ‘Text\_Read’ field had to be stripped of all non-Chinese characters
   1. All numbers and whitespace (‘\n’ and ‘ ‘) were removed. There were no tabs (‘\t’) in this dataset
   2. Punctuation was removed
   3. A couple weird characters that were found during data entry quality checks (see 2.d.ii below) were removed. The Unicode text for these characters include
      1. u'\u200b': a zero width blank character that was throwing off character counts
      2. u'\u2500': a dash that for some reason was not part of the punctuation list used
      3. u'\xa0': a Unicode space character that was not removed during standard whitespace removal
   4. Character counts were calculated for each study session and validated against my hand written counts. During this important data entry validation process I found and corrected a couple problems.
      1. There were records where I accidentally copied some of the text into the wrong day (e.g. I actually read more text the first day, but accidentally copied some of the text into the second day). These were easy to find because the differences between the python counts and hand counts were equal and opposite (e.g. -13 and +13) on days right next to each other)
      2. As I checked other records where the python counts and hand counts didn’t match, I found the extra characters in 2.c above and modified the python code to remove them
      3. I was not able to remove all of the differences between the 2 because, as it turns out, I don’t count large numbers very well by hand. I double checked many of my manual counts and found that my original manual counts were wrong and the python counts were right. Below is a chart summarizing the discrepancies between the final python character counts and the original manual counts. You can see that there are a couple days where the counts were very off. I had a habit of counting by 50’s and at times I accidentally lost track of how many groups of 50 I had counted. You can see on the right portion of the chart that I had a bias of assuming I had read more than I actually had.



1. Features were created from the cleaned up text in order to use them for supervised learning
   1. Features representing the cumulative time spent reading and cumulative characters read were created to establish the baseline (experience curve) model.
   2. I created a feature that gives the percentage of characters in the record’s text that have previously been in the ‘text read’ field in the dataset. I had to create this feature before splitting the training and test datasets. To create this feature I did the following (see feature\_creation.py for details)
      1. Use CountVectorizer from the sklearn.feature\_extraction.text package to create a sparse document term matrix for the entire dataset corpus.
      2. Use the document term matrix to find all characters found in the text corpus up to that point, and in the current study session
      3. Use the counts in the 2 lists from step 3.b.ii. to determine how many of the characters in the current study session have already appeared in the corpus
   3. An average number of days since characters were last seen/read was created (see feature\_creation.py for details)
      1. I used the list of current characters from step 3.a.ii above to find columns in the document term matrix to search
      2. Use the python map function to get the difference between the current study session date and the date for the most recent time that character was found in the corpus
      3. Take the average (mean) of this difference and use it as a feature
   4. An average term frequency for the characters in any given study session was created. By this I mean I found the relative character frequency in the corpus before that study session for the characters found in the study session. To get this I did the following (see feature\_creation.py for details)
      1. Get the numerator of the frequency fraction by summing all of the counts of characters in the given study session that have already been seen in the corpus
      2. Get the denominator of the frequency fraction by summing all of the character counts in the corpus prior to the given study session
      3. Divide the numerator by the denominator
   5. I normalized several of the created features for future use because some supervised learning models are sensitive to scale differences in input features and I wanted to be able to try a variety of supervised learning models without this error creeping in.
   6. I created some interaction terms/features by multiplying the normalized (range between 0 and 1) versions of 2 features together (e.g. 'timeXdays\_since' = 'norm\_cum\_time' \* 'norm\_mean\_days\_since')
   7. Finally I created a function to generate n\_topic models using Latent Dirichlet Allocation (LDA).
      1. Similar to 3.b.i above I created a document term matrix
      2. Used the [sklearn.decomposition.LatentDirichletAllocation package](http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.LatentDirichletAllocation.html) to fit an LDA model to the training set text corpus
      3. Used the LatentDirichletAllocation.transform function to create features that represent the probabilities of different study sessions’ text coming from each topic model.

**Implementation**

The implementation of the code for this project started with the raw\_to\_tidy.py file. This file takes the raw Excel entered data and turns it into a tidy ‘pandas’ dataframe for future analysis. In the beginning, I thought that I would save the raw dataset as a .csv file. However, I found out that saving Unicode text in a .csv format is very complicated and getting that Unicode text into Python was more trouble than it was worth. Originally, I manually entered the dataset in Excel, so to get around the .csv problem I left Excel as the raw data format and found the [‘xl’ Python package](http://www.python-excel.org/) that allowed me to use VBA commands (which I’m already familiar with) to pull the data from Excel to Python.

As mentioned in the data preprocessing section of this report, making sure that the raw data was accurate compared to my hand-written notes was an iterative process. A lot of time was spent processing the raw digital data, digitally counting the characters, comparing the digital count to the manual count, finding and correcting discrepancies and repeating this loop as necessary until the digital an manual counts agreed.

How the model input features were created has already been discussed, but there were some technical issues in the code creation for the character count and frequency features that have not been discussed. The basis for these features was a sparse document term matrix created using [“CountVectorizer” in the “sklearn.feature\_extraction.text” package](http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html). I hadn’t used sparse matrices in python before so I went through a lot of trial and error solutions and Stack Overflow forums to get started (see commented statements throughout my code for URL references of websites that were helpful with this and other problems I encountered). Once I understood this sparse matrix, I spent a lot of time vectorizing the calculations using the map function in order to speed up the calculation and processing time needed to create the features. In the end, I did still use one ‘for’ loop to create these features.

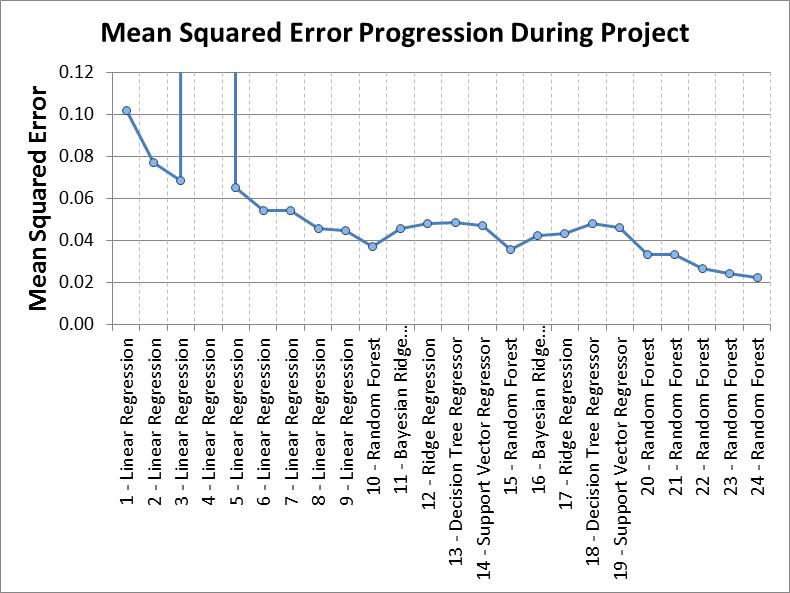
Most of the features were created from the combined X\_train and X\_test dataset, and were then split afterwards. However, the text topics needed to be created only from the training set in order to prevent snooping on the test set. This created a complication when I wanted to create these features in the test set. The Latent Dirichlet Allocation (LDA) topic model I used takes a document term matrix as input. In order for the model created with the training set to be used on the test set, the document term matrix for the training set had to have the same number of terms as the document term matrix for the test set. Because these 2 data sets are different sizes, their vocabulary/term size was different and created errors. To solve this problem, I temporarily combined/appended the training and test sets, created a combined document term matrix, then split that document term matrix back into a training matrix and a test matrix. This forced the number of terms in each matrix to be equal. With this problem resolved, I was able to create features in the test set that only used the topics learned from the training set.

One of the complications I had during the implementation of this project was the mean squared error performance metric. In the sklearn ‘mean\_squared\_error’ python package the default output is actually negative. This was very confusing for a while because it seemed impossible that the mean of something squared could end up being negative. After reading a fair amount of [documentation on this phenomenon](#https://github.com/scikit-learn/scikit-learn/issues/2439) it turns out that the package does this on purpose to allow it to be used in optimization routines that are designed to maximize their outputs; maximizing a negative number brings it closer to zero. With this understanding, the mean squared error of sklearn only needed to be multiplied by -1 in order to meet the needs of the project.

Most of the rest of the implementation was straight forward data science. I used sklearn packages for the supervised learning algorithms and used GridSearchCV to explore the parameter spaces in each learning algorithm while ensuring I wasn’t overfitting the data. The only other implementation complication was creating some of the charts. My previous data science experience was with the R language, so I had to [learn the matplotlib library](http://matplotlib.org/users/pyplot_tutorial.html) in Python to be able to create effective plots and charts.

**Refinement**

As discussed previously, I used a linear model of the experience curve where the x and y axes were on a logarithmic scale. I used 10-fold cross validation as a model performance improvement metric throughout the project. The mean cross validated MSE (mean squared error) for the baseline model was 0.1019. After getting this baseline value, I started to create features from the text in the dataset that I thought would help improve the models performance. I continued to use simple linear regression while creating new features to determine if they helped explain more of the prediction problem’s variation. You can see this sequence of trials in the chart below where “1-Linear Regression” is the baseline model performance, and “2 – Linear Regression” through “9 – Linear Regression” were linear regression models I tried with new features I was creating. The x-axis of this chart is labelled with the trial number, and then the model used in that trial. For details of all of the different features and parameters tried during this process, please refer to the table included in Appendix I of this report).



After trying a bunch of different features and seeing the plateau in the above chart around trial 6 and 7, I decided to take the features I had created and input them into several different supervised learning algorithms to see if those algorithms would fit the data better with the same/similar input features. I tried the following algorithms: random forest, Bayesian ridge regression, ridge regression, decision tree regression, support vector regression. I tried all of these models with the 2 best feature sets from the linear regression model (features from trials 8 and 9) to see which model performed the best. Again, please refer to Appendix I for more information on the parameters tried when using these models.

The random forest model performed better than the rest of the models so in trials 20 through 24 I fine-tuned the parameters of that model to minimize the cross-validated MSE. During this fine tuning process I noticed that my grid search was finding models that performed well at the edges of my parameter grid. Several of my model iterations widened and shifted the grid search parameters to see if I could find an optimum. What I found was that as I increased the n\_estimators (the number of random decision trees in the model), the cross-validated MSE improved. This makes sense because the randomness and averaging helps the model to generalize better. After a while though, increasing this number just increased the processing time without much MSE improvement. I also noticed that increasing the allowable max\_depth of the decision trees in the random forest improved MSE performance. I even allowed the max\_depth to be larger than the number of input features and still got better performance. This only makes sense for decision trees with non-Boolean input features. With continuous input data, a decision tree can split on the same input feature several times. This max\_depth parameter also had diminishing returns as I increased it. In the end, the final model performance had mean 10-fold cross-validated MSE of 0.0223; this represents a 78% reduction in MSE over the baseline model.

**Results**

**Model Evaluation and Validation**

The final model was chosen following the methodology described above in the “Refinement” section of this report. In summary, features were created and combined with the baseline linear model to see if they had any predictive power. Once a list of new features was created, those new features were used in more complex algorithms using sklearn’s GridSearchCV to explore those models parameters. The random forest model performed the best during this exploration process and was further fine-tuned to get the final model.

The final model was a random forest with the following parameters:

*RandomForestRegressor(bootstrap=True, criterion='mse', max\_depth=15,*

*max\_features='log2', max\_leaf\_nodes=None, min\_samples\_leaf=1,*

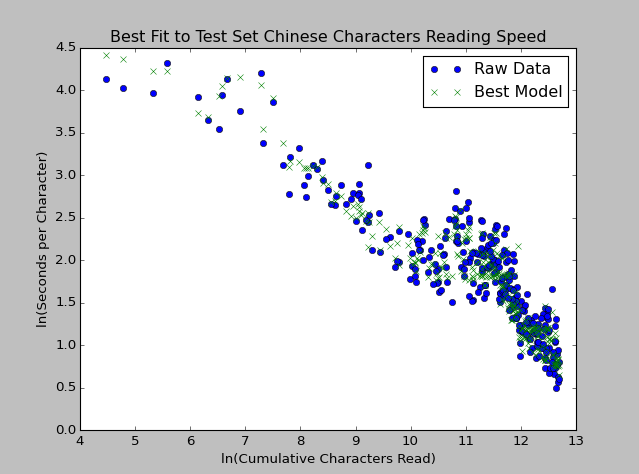
*min\_samples\_split=2, min\_weight\_fraction\_leaf=0.0,*

*n\_estimators=150, n\_jobs=1, oob\_score=False, random\_state=1,*

*verbose=0, warm\_start=False)*

Bootstrap samples used from the dataset when building decision trees is the default for random forest regressors and makes sense for this problem. The only available criterion for random forest regressors is ‘mse’, or mean squared error, so there wasn’t much choice on that variable. Maximum tree depth (max\_depth) of 15 was found through the grid search to be a good value. The same is true for the maximum number of features (max\_features) to consider when looking for the best split and the number of decision tree estimators (n\_estimators) in the forest. The other parameters like ‘max\_leaf\_nodes’ and ‘min\_samples\_leaf’ were not really explored and are discussed in the ‘Improvement’ section of this report’s conclusion as opportunities for further model improvement. Although these parameters were not explored, the default values are appropriate for this problem and do not detract from its performance; they only offer an opportunity for improvement.

The final model had a mean 10-fold cross-validated MSE of .0223, which is a 78% reduction in MSE over the baseline model. This suggests that the model is aligning with the expected final solution. When this same model was used on the ‘unseen’ test set, it had a MSE of 0.0337. This is not as good as the training set performance, but that is somewhat expected. This result still shows that the model generalizes well on unseen data. This can be seen in the chart below where the raw test set data is plotted against the best fit model prediction.



While the chart above shows that the final model is pretty good at predicting the ‘unseen’ test set reading speed, it does appear to have some weaknesses still. It appears to not be sensitive enough to some of the data points in the chart. A good example of this is near the beginning of the chart/experience curve. Those predicted data points are close to the actual values, but don’t deviate enough from linear to be as accurate as the predictions in the right side of the chart. The model also has trouble with a couple other test set data points that appear to be outliers in the right portion of the graph.

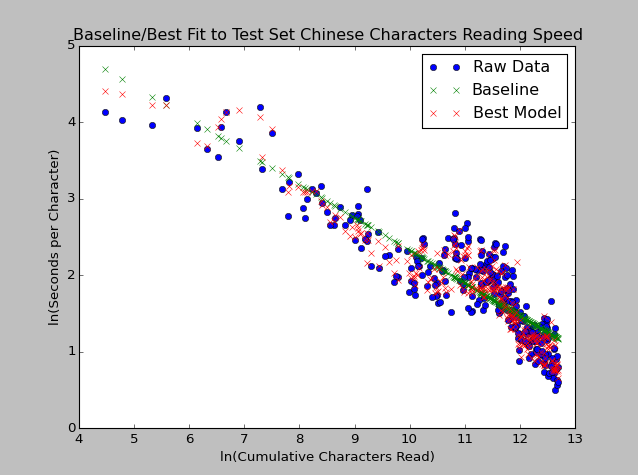
Experience curves are particularly resistant to model sensitivity problems. Taking the logarithm of the ‘y’ output and the logarithm of the main predictor prevents the input and output from getting too big (or too small) too fast. I also normalized most of the input features I created from the dataset text, so they can only take on values between 0 and 1. All of these characteristics of the model lead me to believe that the model will be fairly well behaved given a real dataset. Although it might be true that an example could be synthetically created where the ln(cumulative characters read) was very low, and the percent of characters seen was very high and the model might not predict well.

**Justification**

At this point in the project, I had cross-validated results that appeared to improve the MSE of prediction of reading speed by 78% (see the discussion of how this was reached in the refinement section above). However, this was measured on the training set that I spent a lot of time exploring. I have heard Yaser S. Abu-Mostafa, a machine learning professor from Caltech, say, “Torture the data long enough and it will confess”. Essentially I may have spent a lot of time finding a model that gets a good cross-validated error score, but might not do well on data outside of the training set.

To estimate the model’s true predictive power, I used the fine-tuned random forest model on the test set that was set aside. Without cross-validation, the baseline models MSE on the test set was 0.1107. For fine-tuned random forest model had a MSE of 0.0584 without cross-validation. That implies a 47.24% improvement in MSE, which is not as good as the result on the training dataset, but close.

I plotted the test set along with the baseline and best model fits on the plot below. Visually, you can see that the best fit model is much better at predicting the non-linear portions of the test dataset. This is what I really wanted out of this project. Predictions aren’t perfect of course; visually there appears to be more errors in the beginning of the test set. In my opinion this is acceptable because there is a lot of uncertainty in reading performance in the beginning for anyone. I think the model might be used well with people who have read for a little while (past the beginning of the curve) to be able to predict their reading performance in the future.

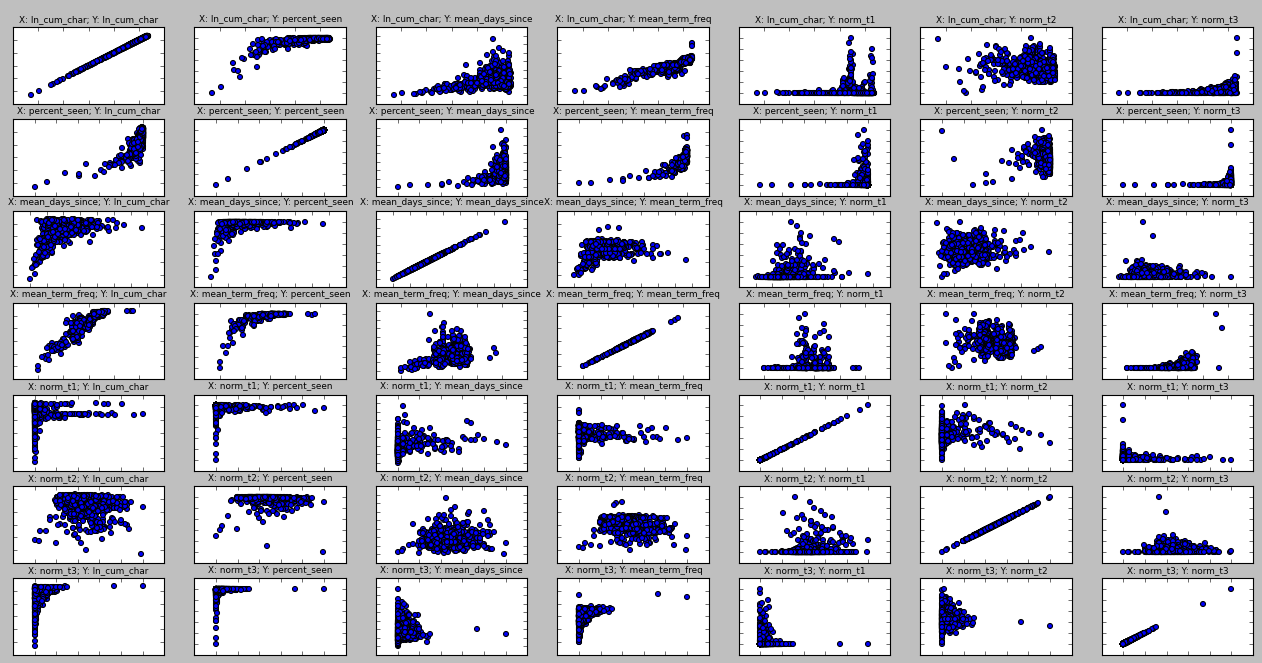


I also took the baseline model and the best model for the training set and calculated the 10-fold cross validation MSE scores on the test dataset. I checked both of these arrays of MSE scores to see if they were normally distributed. The baseline model scores had a normal test p-value of .3180 and the best model scores had a normal test p-value of 0.2545; neither of these is statistically significant with an alpha of 0.05. I also checked the Pearson correlation coefficient between these 2 series and it was 0.0074 with a p-value of 0.9838. This information made me pretty confident that these MSE score arrays were independent and identically distributed. I ran a t-test to compare their mean scores and got a p-value of 0.0000 which is statistically significant with an alpha of 0.05.

**Conclusion**

**Free-Form Visualization**

During the project, I created several different features from the text dataset. I used several of these features as inputs into the final model. Models can have more predictive power when the input features they use are linearly independent of each other. This rarely happens in reality, but it is good to visualize and understand the correlation of the input features. Below is a tiled correlation plot where I visualized the correlation between all of the input features in the final model. It is a large chart and can’t be seen well in on a portrait orientation page. Please see the landscape orientation page version of the chart in the appendix.



The x-axis and y-axis scale values have been removed on purpose because they didn’t add much understanding of what was going on in the plots. They also cluttered up the overall visualization. The subplot titles give the x-axis and y-axis features for each subplot. The x-axis is the same for all sub-plots in a given row, and the y-axis is the same for all of the subplots in any given column. The x and y axes are all scaled to include all of the data points in each correlation chart. Obviously all of the subplots along the diagonal from the top left to the bottom right are perfectly correlated because they are plots where an input feature is plotted against itself. The visualization is also symmetric along this same diagonal axis; the plots might not appear symmetric only because the aspect ratio of the subplots is not square.

As mentioned previously, I created this chart to look for correlations between the input features. Ideally all of the subplots off the diagonal would just look like blobs of data points. Some of these sub plots look like this (e.g. ‘X: mean\_days\_since; Y: norm\_t2’). There are also a couple of the input features thathave high correlation (e.g. ‘X: ln\_cum\_char; Y: mean\_term\_freq’). A visualization like this would be helpful in future work to determine which input features might be improved upon. For example, one might ask, “Is there another way of capturing information about term frequency in the text that does not correlate so strongly with the natural logarithm of the cumulative character count?” If a feature like that could be found, it would likely have better predictive power than the current ‘mean\_term\_freq’ feature.

**Reflection**

Overall this project pushed me to learn new things in a couple of key areas. As is common in data science, the biggest challenges were not implementing the algorithms or tuning them to the data, but getting the data cleaned up, and creating features that would predict the output variable well.

I learned a lot about Unicode text during this project. I’ve done machine learning on text in my past, but never with Chinese characters. This character type forced me to learn how to handle this data type and learn several new functions that helped me clean up and parse the data.

Once the data was clean the challenge was creating features from the text data that would concisely summarize a feature of a given text. I couldn’t just use a big matrix with counts of all of the characters in the text because the curse of dimensionality would have ruined my cross-validated MSE performance metric. So I thought back to struggles I had while reading the Chinese book (with its hand-written dataset) and tried to develop features of the text that represented problems I faced (e.g. long gaps of time between study sessions, reading characters that didn’t appear frequently in the text, etc…).

With a good set of predictive features in hand, I then just had to try a bunch of different supervised machine learning models and tune their parameters. After seeing a solid improvement over the baseline performance, I plotted the improved results and was pleased to see how well the new algorithm predicted my reading speed. I believe that similar techniques and input features could be used for other people learning to read Chinese, and might also work for people learning to read other languages not familiar to them.

**Improvement**

I think that there is still room for improvement on this supervised learning result. Most of the improvement is probably in finding other features to use as predictors. Here are some additional predictive features that might improve the solution:

* Clustering by Chinese radicals: Chinese characters are often made up of sub-characters called radicals. These radicals appear in many different Chinese characters and can be a source of confusion when trying to learn to read Chinese. I think that using some method of clustering the Chinese characters in the text by common radicals could be used to predict how often a new learner would get confused between 2+ similar characters, and thus better predict reading speed
* N-grams: In English there are many words that are often found together that have a specific meaning like “machine learning”. This has a different meaning from the 2 words separated “machine” and “learning”. In Chinese, this is much more common. Many words in Chinese are N-grams (N = number of characters together forming one word/idea). These n-grams might be better predictors of reading speed than the characters alone. I have done this before in other projects, but didn’t get around to it for this one.
* Part of speech tagging: Tagging certain characters/n-grams as different parts of speech might also have predictive power. It might help to distinguish between the more meaningful characters and the Chinese stop words. These stop words, like in English, are the words that are used very often (e.g. ‘the’, ‘a’, ‘and’) and don’t add much meaning, but add to the character count. Perhaps just a count/percentage of these stop words would be a good predictor
* Entropy: Entropy is a concept in “bag of words” text analysis that measures the randomness of the words in a given text based on the rest of the corpus. This entropy value for a given study session might help predict if a certain study session is non-random and therefore contains more obscure characters in it. This might improve upon just taking the average number of days since all of the characters in the study session text were seen.

As mentioned previously, there might be room for improvement by further tuning parameters in some of the models I tried. Perhaps some of the parameters in the random forest model that I did not change during my study could have positively impacted the models predictive power. These parameters, as well as parameters in some of the other models explored (and not yet explored) are an opportunity for further study.

The only algorithm that I did not implement that I would have wanted to implement in this project was a neural network. The [sklearn package for neural networks](http://scikit-learn.org/dev/modules/neural_networks_supervised.html) is in its beta version as of this writing and I didn’t want to potentially put an unstable version of sklearn in my final project. I have written a neural network algorithm by hand in my past, but didn’t see the benefit to doing that for this project when I already tried so many other algorithms.

**Appendix I: Chinese Learning Log Machine Learning Results**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Attempt # | X Features Included | Model Type | GridSearch Parameters | Mean 10-Fold Cross Validated MSE | % Improvement |
| 1 | Baseline = 'ln\_cum\_char' | Linear Regression | N/A | 0.1019 |  |
| 2 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since' | Linear Regression | N/A | 0.0770 | 24.4% |
| 3 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'timeXper\_seen', 'timeXdays\_since' | Linear Regression | N/A | 0.0686 | 32.7% |
| 4 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'timeXper\_seen', 'timeXdays\_since', 'norm\_t1', 'norm\_t2', 'norm\_t3', 'norm\_t4', 'norm\_t5', 'norm\_t6', 'norm\_t7', 'norm\_t8', 'norm\_t9', 'norm\_t10' | Linear Regression | N/A | 850728718.8 | -834866259799.9% |
| 5 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'timeXper\_seen', 'timeXdays\_since', 'norm\_t1', 'norm\_t2', 'norm\_t3' | Linear Regression | N/A | 0.0650 | 36.2% |
| 6 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'timeXper\_seen', 'timeXdays\_since', 'timeXterm\_freq' | Linear Regression | N/A | 0.0543 | 46.7% |
| 7 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'timeXper\_seen', 'timeXdays\_since', 'timeXterm\_freq', 'norm\_t1', 'norm\_t2', 'norm\_t3' | Linear Regression | N/A | 0.0541 | 46.9% |
| 8 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq' | Linear Regression | N/A | 0.0458 | 55.1% |
| 9 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq', 'norm\_t1', 'norm\_t2', 'norm\_t3' | Linear Regression | N/A | 0.0449 | 55.9% |
| 10 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq' | Random Forest | n\_estimators': [3, 5, 10, 20],  'max\_features': ['auto', 'log2', None], 'max\_depth': [2,3,4] | 0.0372 | 63% |
| 11 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq' | Bayesian Ridge Regression | alpha\_1': [3e-07, 1e-06, 3e-06], 'alpha\_2': [3e-07, 1e-06, 3e-06], 'lambda\_1': [3e-07, 1e-06, 3e-06], 'lambda\_2': [3e-07, 1e-06, 3e-06] | 0.0458 | 55% |
| 12 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq' | Ridge Regression | alpha': [0.1, 0.3, 1.0, 3.0, 10.0] | 0.0482 | 53% |
| 13 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq' | Decision Tree Regressor | splitter': ['random', 'best'], 'max\_features': ['auto', 'log2', None], 'max\_depth': [2,3,4] | 0.0486 | 52% |
| 14 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq' | Support Vector Regressor | kernel':('linear', 'rbf'), 'C':[0.3, 1.0, 3.0, 10.0] | 0.0472 | 54% |
| 15 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq', 'norm\_t1', 'norm\_t2', 'norm\_t3' | Random Forest | n\_estimators': [3, 5, 10, 20],  'max\_features': ['auto', 'log2', None], 'max\_depth': [2,3,4] | 0.0355 | 65% |
| 16 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq', 'norm\_t1', 'norm\_t2', 'norm\_t3' | Bayesian Ridge Regression | alpha\_1': [3e-07, 1e-06, 3e-06], 'alpha\_2': [3e-07, 1e-06, 3e-06], 'lambda\_1': [3e-07, 1e-06, 3e-06], 'lambda\_2': [3e-07, 1e-06, 3e-06] | 0.0424 | 58% |
| 17 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq', 'norm\_t1', 'norm\_t2', 'norm\_t3' | Ridge Regression | alpha': [0.1, 0.3, 1.0, 3.0, 10.0] | 0.0433 | 58% |
| 18 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq', 'norm\_t1', 'norm\_t2', 'norm\_t3' | Decision Tree Regressor | splitter': ['random', 'best'], 'max\_features': ['auto', 'log2', None], 'max\_depth': [2,3,4] | 0.0482 | 53% |
| 19 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq', 'norm\_t1', 'norm\_t2', 'norm\_t3' | Support Vector Regressor | kernel':('linear', 'rbf'), 'C':[0.3, 1.0, 3.0, 10.0] | 0.0459 | 55% |
| 20 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq' | Random Forest | n\_estimators': [3, 5, 10, 20, 30], 'max\_features': ['auto', 'log2', None], 'max\_depth': [2,3,4,5,6,7] | 0.0335 | 65% |
| 21 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq' | Random Forest | n\_estimators': [10, 30, 100], 'max\_features': ['auto', 'log2', None], 'max\_depth': [2,5,7,9] | 0.0335 | 67% |
| 22 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq', 'norm\_t1', 'norm\_t2', 'norm\_t3' | Random Forest | n\_estimators': [3, 5, 10, 20, 30], 'max\_features': ['auto', 'log2', None], 'max\_depth': [2,3,4,5,6,7] | 0.0265 | 74% |
| 23 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq', 'norm\_t1', 'norm\_t2', 'norm\_t3' | Random Forest | n\_estimators': [10, 30, 100], 'max\_features': ['auto', 'log2', None], 'max\_depth': [2,5,7,9] | 0.0241 | 76% |
| 24 | ln\_cum\_char', 'percent\_seen', 'mean\_days\_since', 'mean\_term\_freq', 'norm\_t1', 'norm\_t2', 'norm\_t3' | Random Forest | n\_estimators': [100, 120, 150], 'max\_features': ['auto', 'log2'], 'max\_depth': [9, 12, 15] | 0.0223 | 78% |

**Appendix II: Input Feature Correlation Visualization**

