Analyzing relative effectiveness of different help types in Project LISTEN data

Jacqueline Gutman

May 15, 2015

What is Project LISTEN?

Project LISTEN (Literacy Innovation that Speech Technology Enables) is an automated and adaptive online reading tutor that displays sentences as part of a story on screen for children to read aloud. The tutor has automatic speech recognition to detect whether the child is reading the sentence correctly, and whether there is any hesitation between words. The program has been evaluated on hundreds of children and has been demonstrated to improve reading comprehension. It aims to challenge students just enough to keep them engaged and learning new words without frustrating them too much ("zone of proximal development").

Motivation for current research problem

The children have the ability to request help, and the Reading Tutor can provide help automatically. There are 13 different types of help available, but not all help types are available for all words. When help is requested, the Reading Tutor chooses randomly from among the types of help available for that word, and provides help of that type. So, an important question for the researchers developing the tutor is: Is there a better way to decide what type of help should be provided instead of choosing randomly? That is, can we predict, for a given trial and word, which type of help is the most likely to result in the student reading the word successfully the next time the student encounters that same word?

Obtaining the data

Because the Reading Tutor randomly chooses from available help types, it creates a kind of natural experiment–the best proxy we have to a true randomized controlled experiment. From the database of Project LISTEN data, we can search for instances where a student received help on a particular word, and extract the data for the trial containing their next encounter of that same word (at least one day later).

The eligible trials have already been extracted from a MySQL database as part of a prior research project, so we will begin directly with a spreadsheet containing only these trials (the first encounter of a particular word at least 1 day subsequent to the same student receiving help on the same word).

The dependent variable of interest is whether or not they read that word successfully–without hesitation and without requesting help.

Analytic goals and thought process

1. Our first task is to build a model that will allow us to predict whether or not the student succeeding in reading a word after previously receiving help on that same word.
2. From there, we can perform feature selection in order to analyze what features are critical in determining whether a word is read successfully or not.
3. Use the model and selected features to determine, for any given trial, what help type would produce the greatest predicted success for that particular trial.
4. For a new set of data, determine the help type that is expected to yield the highest predicted probability of success. Compare the success rate for trials where the actual help type provided matches the recommended help type provided, to the overall success rate for trials with randomly chosen help types. In this pseudo-experiment, can we conclude that our model-based recommendations for help type appear to have a positive "effect" on the reading success rate?
5. If we can develop a model for making help type recommendations, the next logical step would be to implement that model in the Reading Tutor as a real experiment, and see how much the recommendations improve performance in practice.

Exploratory data analysis

Let's start by reading in the data and getting familiar with the features contained in the data.

```
require(pander)
require(plyr)
require(glmnet)
require(ROCR)
require(rpart)
require(partykit)
require(caret)
require(knitr)
require(tree)
```

Load in the primary raw data file.

```
folder <- "2012_Fluency_Evolution"
subfolder <- "Documentation - Experimentation help success"
file <- "Alldata join helpgiven and lexview, freq, prev, levels.csv"
loc <- paste(".", folder, subfolder, file, sep = "/")
data <- read.csv(loc)

#names(data)
#dim(data)
#str(data, strict.width = "cut")
```

We begin with 60917 observations in 10 different features.

feature	class	num.unique.values	range.of.numeric	any.missing	description

user_id	factor: 532 levels	532	N/A	no	unique identifier for each student
target_word_number	integer	41	2 to 49	no	position of the target word in the sentence
target_word	factor: 3237 levels	3237	N/A	no	target word to be read
latency	numeric	208	0.01 to 4.54	no	hesitation in seconds before student begins reading the word
helped	integer	binary	0 to 1	no	binary indicator of whether the student received help for this word in this trial
Type	factor: 13 levels	13	N/A	no	type of hint student received on same word in previous encounter (at an earlier date)
Reading_Level	factor: 8 levels	8	N/A	no	reading level of the student
Story_Level	factor: 10 levels	10	N/A	no	level of difficulty of the story (students at higher level get harder stories
frequency	numeric	281	1.15 to 6.07	no	measure of frequency of the word in the English language (higher is less frequent)

PREV	integer	37	0 to 36	no	how many times the student encountered the word previously

There are a few things we'd like to change about the dataset before we begin our analyses.

1. Let's create a binary target variable, called success, that encodes whether or not the student spoke the word with no hesitation and without receiving any help.
2. Let's change all the target words to lowercase. This way we don't count "the" and "The" as two separate words.
3. Let's add features for the length of the word and for whether it's just a letter or number word.
4. Finally, we want to get some kind of a base rate performance, based on the average success or latency over all other trials featuring the same target word, and the average success or latency over all other trials for the same student. I expect that the average difficulty of a word, or the average reading ability of a particular student, might be very predictive of the success of that trial. (It might make sense to also compute these means conditional on the reading level or story level.)

Feature engineering

```
# Let's begin by defining a function that takes a dataset, a variable to averag
e over, and a
# variable to condition on, and then loops through the dataset, computing the c
onditional
# mean for each observation. So for example, we can condition on the student (u
ser_id), and
# then take the average of their latency variable.
# Importantly, we want to return this conditional mean as a vector in the same
shape as the
# original data, and we need to exclude the current observation from the mean t
o avoid
# issues with leakage.

############# GET_MEAN_OTHER FUNCTION DEFINITION #############
get_mean_other <- function(levels.var, mean.var, data) {
  # Add a feature that gives the mean and number of trials over all trials in t
he data
  # for that particular value of the levels variable

  # TODO update get_mean_other function to accommodate list of levels.var
  # i.e. get mean for each interaction of student and reading level variable
  all_trials <- ldply(levels(data[[levels.var]]), function(id) { #loop over eac
h unique student, word, etc.
    subset <- data[[mean.var]][data[[levels.var]] == id] #all trials for a part
icular student, word, etc.
    data.frame(mean_all_trials=mean(subset), num_trials=length(subset))
  })
  rownames(all_trials) <- levels(data[[levels.var]])
  # returns a data frame with a row for each student, word, etc. and a column f
or the mean
  # second column for the number of trials, NOT SAME SHAPE AS ORIGINAL DATA

  # We need to make sure we exclude the current value for that particular trial
from the mean
  num_excluding_current <- all_trials$num_trials[data[[levels.var]]] - 1
  mean_excluding_current <- (all_trials$mean_all_trials[data[[levels.var]]] *
    all_trials$num_trials[data[[levels.var]]] - data[[mean.var]]) /
    num_excluding_current
  return(list(mean = mean_excluding_current, num = num_excluding_current))
  # returns an object containing two vectors, both same dimension as the origin
al data
  # mean over all trials except current trial
  # may contain NAs if there are no other trials for that value of the student,
word, etc.
}

############# GET_MEAN_OTHER_MULTICAT FUNCTION DEFINITION #############
get_mean_other_multicat <- function(levels.var, mean.var, data) {
  # this function is just like get_mean_other but it should work
  # when levels.var is a vector of features we want to condition on
  # also should work when mean.var is a vector of features we want to avg over

  x <- split(data[, c(levels.var, mean.var)], data[, levels.var])
  y <- sapply(seq(length(x)), function(i) {dim(x[[i]])[1] > 0})
  x <- x[y] # x is a list of dataframes
  # the length of x is the number of non-empty interactions of the levels varia
ble
  # i.e. we drop any combinations of the levels variables that don't contain da
ta

  # create a dataframe with 1 column for each variable in mean.var
  # containing the mean over all other observations
  # in the same subgroup formed by conditioning on the levels variable
  z <- ldply(names(x), function(cat) {
    index <- rownames(x[[cat]])
    target <- x[[cat]][mean.var]
    means <- ldply(seq(dim(target)[1]), function(x) {colMeans(target[-x,])})
    rownames(means) <- index
    return(means)
    # NA in rows for subcategories with only one entry
  })
  return(z) }

############ FORMAT_DATA FUNCTION DEFINITION ############
format_data <- function(data) {
  # format_data uses get_mean_other functions defined above
  # change all words to lowercase
  data$target_word <- factor(tolower(data$target_word))
  # regular expression to find the words that are just numbers
  # could be something like "1st", "14th", etc. but not "first", "zero", etc.
  # there's something easier about reading "1000" than "zero"
  data$is.number <- grepl("[0-9]+",as.character(data$target_word))

  # Add a feature that gives the length of the word
  data$word_length <- nchar(as.character(data$target_word))
  data$is.letter <- (data$word_length == 1) & !(data$is.number) # 1 if word = a
-z, A-Z
  data$is.letter <- as.numeric(data$is.letter)
  data$is.number <- as.numeric(data$is.number)

  # Call the trial a success if there is no latency or hesitation (min value =
.01) and no help given
  data$success <- ifelse(data$latency == .01, 1, 0)
  data$success <- ifelse(data$helped == 1, 0, data$success)

  # Add a feature that gives the mean latency over all other trials in the data
for that student
  latency_by_student <- get_mean_other("user_id", "latency", data)
  data$mean_latency <- latency_by_student$mean
  data$num_student_trials <- latency_by_student$num

  # Add a feature that gives the mean success rate over all other trials in the
data for that student
  success_by_student <- get_mean_other("user_id", "success", data)
  data$mean_success <- success_by_student$mean

  # Add a feature that gives the mean latency and success rate over all other t
rials for that word
  latency_by_word <- get_mean_other("target_word", "latency", data)
  data$mean_word_latency <- latency_by_word$mean
  data$num_word_trials <- latency_by_word$num
  success_by_word <- get_mean_other("target_word", "success", data)
  data$mean_word_success <- success_by_word$mean

  # Add a feature that gives the mean latency and success rate over all other t
rials
  # for the trials of that student at that story level
  latency_success_by_student_story <- get_mean_other_multicat(levels.var =
          c("user_id", "Story_Level"), mean.var = c("latency", "success"), da
ta=data)
  data$mean_student_level_latency <- latency_success_by_student_story$latency
  data$mean_student_level_success <- latency_success_by_student_story$success

  # Any other features we want to generate here?
  return(data) }
```

Okay, now we can call the format_data function on our raw data frame and explore the new features that have been generated.

- Possible change for future analyses: It may be more appropriate to only take the mean over trials that occurred earlier in time than the current trial, instead of taking the mean over past and future trials, excluding the current trial. Averaging over past events better approximates the information that the Reading Tutor might actually have access to at run time.

```
data.2 <- format_data(data)
names(data.2)
```

```
     [1] "user_id"                     "target_word_number"
     [3] "target_word"                 "latency"
     [5] "helped"                      "Type"
     [7] "Reading_Level"               "Story_Level"
     [9] "frequency"                   "PREV"
    [11] "is.number"                   "word_length"
    [13] "is.letter"                   "success"
    [15] "mean_latency"                "num_student_trials"
    [17] "mean_success"                "mean_word_latency"
    [19] "num_word_trials"             "mean_word_success"
    [21] "mean_student_level_latency" "mean_student_level_success"
```

```
#dim(data.2)
#str(data.2, strict.width = "cut")
```

Adding possible help types

Not every help type is available for every word. Unfortunately we don't have data about what help types are actually possible for specific words, but we can do our best to approximate this information by looking at what help types are observed over all occurrences of a given word in our dataset.

For future analyses, it would be better to get an accurate list of what help types are actually possible for each word, rather than inferring this information from the help types observed for that word in our data. If this is not possible, a better approximation of the help types available might be to observe the help types provided over all occurrences in the full database of trials, rather than restricting our view to the trials we have extracted from the database as relevant to this analysis.

For these analyses, however, we may be able ignore help types that are possible for a word but not observed, since we will have no data on whether the unobserved help types were associated with improved probability of success.

```
# Let's define a function that takes in a dataset, and creates a new data frame
, with a row
# for each unique word in the dataset. For each word, we want this new datafram
e to tell us
# how many times the word occurs, the number of different help types observed f
or that word,
# as well as a binary indicator for each help type, indicating whether that hel
p type was
# observed for that particular word. We will also create a factor variable that
gives a label
# to each unique subset of help types-- for example, help class 1 may mean that
Autophonics
# and StartsLike are available for this word, and help class 34 may mean that A
utophonics,
# RhymesWith, and Recue are available for this word.

############# ADD_POSSIBLE_HELP_TYPES FUNCTION DEFINITION #############
add_possible_help_types <- function(data.2) {
  # create a list of lists, where the outer list entries are the unique words i
n the data
  # and the inner list gives the actual help types observed for that word
  # this is essentially a ragged array with 1-13 entries in each row
  help_types_by_word <- sapply(levels(data.2$target_word), function(word)
    sort.int(unique(data.2$Type[data.2$target_word == word])))
  help_type_subsets <- unique(help_types_by_word)
  # letters.and.numbers <- unique(data.2$target_word[
  #      (data.2$is.letter == 1) | (data.2$is.number == 1)])
  # help_types_letters_and_numbers <- sapply(letters.and.numbers, function(word
)
  #      sort.int(unique(data.2$Type[data.2$target_word == word])))

  # count the number of times each word is observed
  count_trials_by_word <- ldply(levels(data.2$target_word), function(word)
    count(data.2$target_word[data.2$target_word == word]))
  # assign a unique label (we'll just use integers) to each observed combinatio
n of
  # help types available. Returns a dataframe containing all the observed
  # combinations of help types in the data
  types <- ldply(seq(to=length(help_type_subsets)),
                 function(i) {
                   category = help_type_subsets[[i]]
                   size = length(category)
                   types = sapply(levels(category), function(type) type %in% ca
tegory)
                   c(index=i, len=size, types) })

  # dataframe containing 1 entry for each word in the dataset
  # add columns for the observed count of the word, the label for the subset of
available
  # help types it belongs to, the length of the word in characters, and binary
indicators
  # for each help type -- was that help type observed for this word or not?
  words <- ldply(seq(to=length(help_types_by_word)),
                 function(i) {
                   word = names(help_types_by_word[i])
                   freq = count_trials_by_word$freq[i]
                   word_help = help_types_by_word[[i]]
                   size = length(word_help)
                   type_words = sapply(levels(word_help), function(type) type %
in% word_help)
                   a = TRUE
                   for(i in seq(to=length(levels(word_help)))) {
                     b = types[i+2] == type_words[i]
                     a = a & b}
                   index_type = types$index[a]
                   c(word=word, num_occurrences=freq, help_class=index_type,
                     num_help_types = size, type_words)
                 })
  # convert chr to boolean to numeric : 0 <- "FALSE" and 1 <- "TRUE"
  types = colnames(words)[5:length(words)]
  for(type in types) {
    words[[type]] <- as.numeric(as.logical(words[[type]]))
  }
  # convert numbers back to numeric (all numbers got cast to chr by the c funct
ion)
  words$num_occurrences <- as.numeric(words$num_occurrences)
  words$help_class <- as.numeric(words$help_class)
  words$num_help_types <- as.numeric(words$num_help_types)
  return(words)
}
```

In further analysis, it may be helpful to have coded categories indicating the subset of help types which are available for a particular word. We will use the add_possible_help_types function to create these codes, and then from these we can generate indicator variables for whether a particular help type was observed to be possible for the given word.

```
# Let's now run the add_possible_help_types function to categorize each word in
the data.
words <- add_possible_help_types(data.2)
# List of words that only appear once in the dataset
singletons <- words$word[words$num_occurrences == 1]
# set the word to be the row index
rownames(words) <- words$word
words$word <- NULL
tail(words)
```

```
       num_occurrences help_class num_help_types Autophonics OnsetRime
youre                6         15              2           0         0
youth                1          2              1           0         0
youve                3         15              2           0         0
z                   47          1              2           0         0
zephyr               1          2              1           0         0
zero               184        167              6           1         1
       PlaybackWord Recue RhymesWith SayWord ShowPicture SoundEffect
youre             0     0          0       1           0           0
youth             0     0          0       1           0           0
youve             0     0          0       1           0           0
z                 0     1          0       1           0           0
zephyr            0     0          0       1           0           0
zero              0     1          0       1           0           0
       SoundOut SpellOut StartsLike Syllabify WordInContext
youre         0        0          0         0             1
youth         0        0          0         0             0
youve         0        0          0         0             1
z             0        0          0         0             0
zephyr        0        0          0         0             0
zero          1        0          0         0             1
```

- So we have 2785 unique words in the dataset, and of these, there are 900 words that only appear once.

Filling in missing values

In generating these features, we’ve also generated some missing values in our dataset. For example, we computed the average over all other trials with the same word in the data, but for the 900 words that only appear once, there are no other trials to average over, so the mean is NA. These might be less common words, so these singletons might actually be more difficult on average than the full set of words, so for now let’s just use basic conditional mean imputation to fill in the mean success variable with the average over all words that appear only once (this is a little bit like assuming these words are all the same word–it isn’t clear that this is the best way to generate the feature, but for me it was the most reasonable and computationally feasible solution that I was comfortable with.)

```
# Fill in NAs with the average latency and success over all words in singletons
singleton.index <- data.2$target_word %in% singletons
data.2$mean_word_latency[singleton.index] <- mean(data.2$latency[singleton.inde
x])
data.2$mean_word_success[singleton.index] <- mean(data.2$success[singleton.inde
x])
```

```
num.imputed.students <- sum(is.na(data.2$mean_latency))
num.imputed.students.by.story.level <- sum(is.na(data.2$mean_student_level_late
ncy))
total.trials <- dim(data.2)[1]
```

We need to do a similar thing for imputing over the means of students where we only have one trial from that particular student. Fortunately, there are not very many of these; in fact there are only 23 trials out of 60917 total trials with 532 different students where we only have 1 data point for a particular student.

```
# For these 23 students, let's just impute their missing values with the mean o
f the mean
# That is, let's take the average of the mean performance of the other 509 stud
ents
data.2$mean_latency <- replace(data.2$mean_latency, which(is.na(data.2$mean_lat
ency)),
                               mean(data.2$mean_latency, na.rm=TRUE))
data.2$mean_success <- replace(data.2$mean_success, which(is.na(data.2$mean_suc
cess)),
                               mean(data.2$mean_success, na.rm=TRUE))

# We also have 396 trials where that trial was the only available trial for tha
t particular
# student at that particular story level. So what should we do with these missi
ng values?
# If we have no data for that student at that story level, then our best guess
is probably
# just the overall average performance of that student.

# If mean_student_level_latency is NA, replace with mean_latency
data.2$mean_student_level_latency<- replace(data.2$mean_student_level_latency,
      which(is.na(data.2$mean_student_level_latency)), data.2$mean_latency)
data.2$mean_student_level_success<- replace(data.2$mean_student_level_success,
      which(is.na(data.2$mean_student_level_success)), data.2$mean_success)
```

If the above mean imputations make you nervous, keep in mind that the first imputation involved 23 observations out of 60917, or less than 0.04 percent of the data. The second imputation involved 396 observations out of 60917, or less than 0.65 percent of the data. With less than 1 percent of data filled in, the method of imputation won't make much of a difference.

```
# Let's double check that we have filled in all the missing data
any.nas <- colSums(is.na(data.2))
sum(any.nas) == 0
```

```
    [1] TRUE
```

```
# any.nas
```

Now we need to merge in the information we compiled about each word and its available help types into the main dataframe containing all individual trials.

```
# In order to use the join function, we need a column with the same name to app
ear in each dataset
words$target_word <- rownames(words)
data.2 <- join(data.2, words, by="target_word", match="first", type="left")
# Help class is the numeric code indicating which help types are available for
that word
# The number itself is meaningless, but it encodes which category the word belo
ngs to
# This feature may end up getting dropped in the analysis, but we'll include it
here for now
data.2$help_class <- as.factor(data.2$help_class)
# Indicator of whether the word appears only once in the dataset
# This feature also tells us that the mean_word_latency and mean_word_success c
olumns have been imputed
data.2$is.singleton <- as.numeric(data.2$num_occurrences == 1)
```

We have now generated more features than we probably will need for our model, and some of our features are now redundant. Where we could, we transformed categorical variables with a very large number of levels (e.g. the identifiers for the student, the target word, and even the class of help types availables) into sets of real-valued continuous and binary features that will be much easier for our models to work with. So we can discard the original categorical features for now.

```
data <- data.2
# identifier for the student, now replaced by mean_success, mean_latency, mean_
student_level_latency,
# mean_student_level_success, and num_student_trials (531 features -> 5 feature
s)
data$user_id <- NULL
# identifier for the word, now replaced by is.letter, is.number, word_length, m
ean_word_latency,
# mean_word_success, num_word_trials, is.singleton, plus help types 1-13 (2784
features -> 20 features)
data$target_word <- NULL
data$num_occurrences <- NULL # this one gets discarded because it's nearly iden
tical to num_word_trials
data$help_class <- NULL # 324 levels of this factor variable, might want to cir
cle back to this later

#names(data)
#dim(data)

# Let's take one last look at the data now that additional features have been a
dded.
#str(data.2, strict.width = "cut")
```

Building the model

Plan for the analysis:

1. Let's begin by looking at a basic logistic regression with all features included, and get a sense of how well that model generalizes to new data.

2. Next, let's compare different methods for feature selection. We might try, as a few possible approaches:

- stepwise logistic regression, with AIC as a metric to compare the models
- using a regularized logistic regression with a regularization penalty that will tend to induce sparsity (lasso/L1 penalty has the characteristic of setting some feature coefficients to zero, and these zeroed out features are ones we might consider unimportant or redundant)
- the varImp function in the caret package will take a model and compute a metric of variable importance for each feature included in the model, and we can sort these in descending order to get a sense of the most informative features

3. Since we are particularly interested in help type, we can directly compare the model fit for the models with and without help type information included. Does dropping those features significantly hurt performance on new data?

Feature engineering is done and we are now finally ready to split the data into training and test sets to try out different models.

```
# Let's first define a function that takes a dataset, the name of a target vari
able, and the
# name of alternative target variables. The function will discard the alternati
ve target variables
# to prevent issues of data leakage, and returns a list, where X is a dataframe
of all the predictors,
# and y is a vector of the target values.

# A little late now, but note that the caret package has a function createDataP
artition to create balanced splits of the data--by default the random sampling
occurs within each class and will preserve the overall class distribution for u
nbalanced data.

# caret package also has createFolds function for k-fold cross-validation and c
reateTimeSlices for splitting timeseries data by time point

############ split_X_y FUNCTION DEFINITION ############
split_X_y <- function(data, target_var, alternative_targets) {
  # target_var should be a single character string
  # alternative targets can be a single character string or a character vector
  # X will contain all features in data that are not in target_var or in altern
ative_targets
```

```
  target_indices <- names(data) %in% c(target_var, alternative_targets)
  X <- data[!target_indices] # dataframe
  y <- data[[target_var]] # vector
  return(list(X=X, y=y)) # list object
}

# Now we want to be able to take an object in the format returned by the split_
X_y function
# and split it into a training set and a test set. Specify the size of training
set as a
# percentage of total dataset size, default size = 80% of total data.

# However, because we are dealing with unbalanced binary data, where successful
trials are much more
# common than unsuccessful trials (in this dataset, success = 1 is about three
times as common as
# success = 0, but this function should be applicable for very sparse datasets
as well).
# Note that this function will only work properly if target variable is numeric
and binary.

############# stratified_train_test_split FUNCTION DEFINITION #############
stratified_train_test_split <- function(data, split.proportion = .80, seed=4850
) {
  binary <- length(unique(data$y)) == 2
  numeric <- is.numeric(data$y)
  if (!(binary & numeric))
    stop("Target variable is not in correct format. Must be numeric and binary.
")

  pos.cases <- which(data$y == 1) # indices of all cases where target variable
is 1
  neg.cases <- which(data$y == 0) # indices of all cases where target variable
is 0
  pos.size <- round(split.proportion * length(pos.cases)) # number of pos cases
desired in the training set

  set.seed(seed)
  pos.train <- sample(pos.cases, pos.size) # indices of pos cases in the traini
ng set
  neg.size <- round(split.proportion * length(neg.cases)) # number of neg cases
desired in the training set
  set.seed(seed)
  neg.train <- sample(neg.cases, neg.size) # indices of neg cases in the traini
ng set
```

```
  train_indices <- c(pos.train, neg.train)
  set.seed(seed)
  train_indices <- sample(train_indices, length(train_indices)) # combine and s
huffle pos and neg cases
  trainX <- data$X[train_indices, ]
  testX <- data$X[-(train_indices), ]
  trainY <- data$y[train_indices]
  testY <- data$y[-train_indices ]
  # return a list object with 4 fields containing train and test predictors (as
dataframes)
  # and train and test values of target (as binary numeric vectors)
  return(list(train.X = trainX, train.y = trainY,
              test.X = testX, test.y = testY))
}
```

Now we can use these functions to create our training and test data. The stratified_train_test_split function takes an optional seed parameter for reproducibility of analyses. For now we will use the default seed.

```
# Remember that the success variable was created deterministically from the val
ues of
# latency and helped. So we need to be sure to exclude both of those variables
from the
# set of predictors.
data.3 <- split_X_y(data, "success", c("latency", "helped"))
data.3 <- stratified_train_test_split(data.3) # split proportion is 80% by defa
ult
str(data.3, max.level=1)
```

```
    List of 4
     $ train.X:'data.frame':    48733 obs. of  32 variables:
     $ train.y: num [1:48733] 1 0 0 1 1 0 1 0 1 0 ...
     $ test.X :'data.frame':    12184 obs. of  32 variables:
     $ test.y : num [1:12184] 1 1 0 1 0 1 1 1 1 1 ...
```

Let’s start by building a standard logistic regression to get a sense of our ability to predict the target.

```
# Build our logistic regression
train = cbind(data.3$train.X, success = data.3$train.y)
predictors = colnames(data.3$train.X) # use all features in the X dataframe as
potential predictors
fit  <- glm(reformulate(predictors, "success"), data = train, family =
binomial)
glmfit <- summary(fit)$coef[,c(1,2,4)]
# sort the coefficients in order of their significance level
pander(glmfit[order(glmfit[,3]),], digits = 3)
```

	Estimate	Std. Error	Pr(>|z|)
mean_success	3.69	0.17	9.66e-105
(Intercept)	-2.5	0.235	2.34e-26
mean_word_success	0.994	0.115	4.41e-18
PREV	0.0343	0.00522	4.96e-11
Story_LevelC	0.17	0.0357	1.95e-06
Reading_LevelD	0.235	0.0581	5.36e-05
Story_LevelG	0.214	0.0635	0.00075
Reading_LevelC	0.131	0.0399	0.00108
StartsLike	-0.122	0.0412	0.00307
word_length	-0.0332	0.0121	0.00617
num_word_trials	9.14e-05	3.96e-05	0.021
num_help_types	0.0721	0.0322	0.025
SoundEffect	-0.308	0.138	0.0254
is.letter	-0.15	0.0672	0.0255
TypeSayWord	-0.0932	0.0423	0.0274
SoundOut	-0.103	0.0572	0.0706
mean_word_latency	0.615	0.345	0.0751
Story_LevelE	-0.067	0.0447	0.134
mean_latency	-1.05	0.729	0.152
OnsetRime	-0.0773	0.0542	0.154
Story_LevelB	-0.0471	0.0349	0.178

mean_student_level_latency	0.564	0.429	0.189
num_student_trials	-5.99e-05	4.64e-05	0.197
Reading_LevelB	0.0415	0.0326	0.203
Reading_LevelK	-0.0382	0.0307	0.213
Reading_LevelE	0.0617	0.0521	0.236
Story_LevelD	0.0524	0.0466	0.261
TypeSoundEffect	0.396	0.353	0.262
Story_LevelF	0.054	0.0588	0.358
is.number	0.124	0.136	0.36
SpellOut	0.52	0.634	0.413
target_word_number	-0.00207	0.00278	0.455
TypeSyllabify	0.0631	0.0848	0.456
Autophonics	-0.0368	0.0511	0.471
TypeShowPicture	-0.127	0.178	0.477
Story_LevelK	0.0279	0.0409	0.495
Story_LevelH	-0.324	0.481	0.5
TypeWordInContext	-0.0348	0.0547	0.525
TypeRhymesWith	0.0347	0.0592	0.557
RhymesWith	0.03	0.0518	0.563
TypeSoundOut	0.0267	0.0469	0.569
TypeRecue	0.0289	0.0538	0.592
Story_LevelP	0.125	0.24	0.601
TypeOnsetRime	0.0281	0.0548	0.609
is.singleton	-0.0318	0.0968	0.742
frequency	-0.00549	0.0178	0.758
Reading_LevelF	0.027	0.155	0.862
ShowPicture	-0.0136	0.0814	0.867
Recue	0.00925	0.0556	0.868

TypePlaybackWord	-0.188	1.16	0.871
TypeSpellOut	7.86	50.8	0.877
mean_student_level_success	0.017	0.11	0.878
SayWord	-0.0107	0.0771	0.889
TypeStartsLike	0.00656	0.0543	0.904
Syllabify	-0.00243	0.057	0.966
PlaybackWord	0.00371	0.102	0.971
Reading_LevelG	-0.000173	0.106	0.999

```
aic.all.predictors <- fit$aic
```

```
# Let's get the top ten most important features from the glm
glm.feature.imp <- varImp(fit)
sorted <-  with(glm.feature.imp, order(-Overall)) #index of most imp features
mostImportant <- rownames(glm.feature.imp)[sorted] #names of most imp features
pander(head(cbind(features=mostImportant,
                  feature.importance=glm.feature.imp[mostImportant,]), 10), dig
its = 3)
```

features	feature.importance
mean_success	21.7346067430956
mean_word_success	8.6678058504497
PREV	6.57224012390364
Story_LevelC	4.7585782031147
Reading_LevelD	4.03931896655357
Story_LevelG	3.37071927036194
Reading_LevelC	3.26961271852486
StartsLike	2.96048905921609
word_length	2.73861632613107
num_word_trials	2.3086859784046

Let’s not try to directly interpret these coefficients. There’s a few initial impressions we might have from looking at the significance level of the tests (conclusions to be taken with some caution).

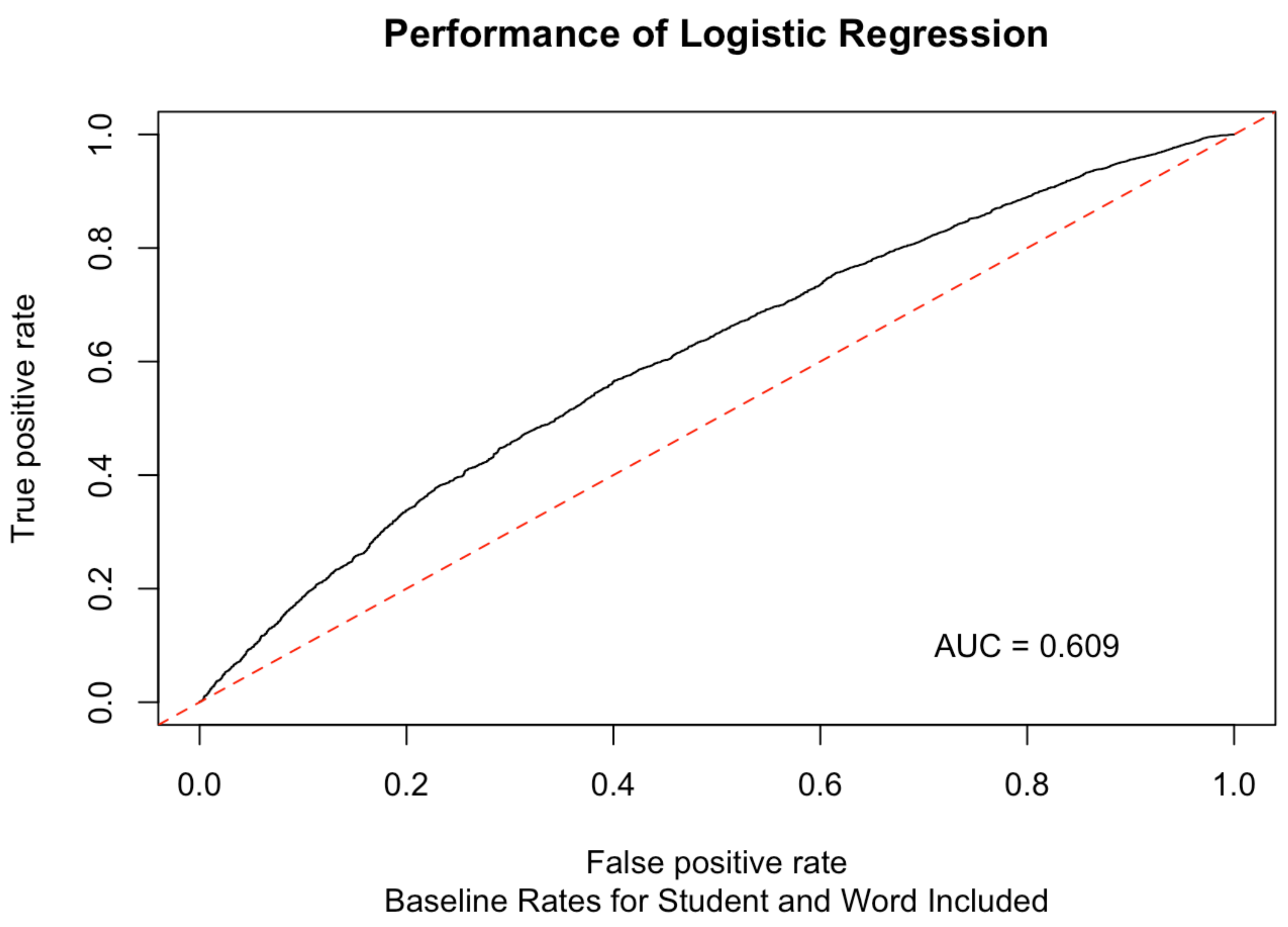
1. Help type doesn't really seem to be significant in predicting the success of a trial. We will explore this further.
2. Number of times a student previously encountered a word seems to be significantly predictive of the success of a trial (the more times a word was encountered, the more likely we are to predict success).
3. The average success rate of that student appears to be significantly predictive of the success of a trial (the higher the student's average success rate across all other trials, the more likely we are to predict success on this trial).
4. The average success rate of that target word appears to be significantly predictive of the success of a trial (the higher the average success rate of all students on that particular word across all other trials, the more likely we are to predict success on this trial).

Instead of focusing on significance testing and interpretability of coefficients, let's get a sense of how good our predictions are overall. (It doesn't matter much which features the model considers important if the model is a poor fit to the data.)

One way to do this is by plotting the ROC curve for our logistic regression model. The more bowed out the curve is, and the further away the area under the curve is from 0.5, the more confident we can be in our model's predictive power. (If we are very close to the diagonal and our AUC is .5, it means that for each trial we predicted to be a success, we were just as likely to be wrong as right.)

This is where our test data comes in.

```
# Let's use the logit model to predict probability of success on our test set
test = cbind(data.3$test.X, success = data.3$test.y)
yhat <- predict(fit, newdata = test, type = "response") # predicted target valu
es on the test set
pred <- prediction(yhat, test$success)
perf <- performance(pred, "tpr", "fpr") # build and plot an ROC curve for the t
est data
plot(perf, main="Performance of Logistic Regression",
  sub="Baseline Rates for Student and Word Included")
abline(0, 1, lty=2, col=2)
print_perf = paste("AUC =", round(performance(pred, "auc")@y.values[[1]], digit
s=3))
text(0.8, 0.1, print_perf) # displays the AUC on the ROC plot
```



```
AUC.glm <- performance(pred, "auc")@y.values[[1]]
```

Our predictive ability doesn't look like it's very good! We're predicting above the level of chance, but not well above chance. Let's see what we can do to improve performance.

Let's try stepping back to the full dataset for a moment, and let's look at a regularized logistic regression. Here, instead of manually splitting the data into training and test sets, we'll use 10 fold cross-validation in the cv.glmnet package. This will allow every observation to be used 9 times in training and once in testing, and since we are averaging over the results of 10 different models, we can decrease some of the variability caused by just randomly picking a bad sample to be used as the test data.

```
# Let's try this on the full dataset, but remember we still need to get rid of
helped and latency
data.full <- data
data.full$helped <- NULL
data.full$latency <- NULL
non.target.index <- -which(names(data.full)=="success")
predictors <- names(data.full)[non.target.index]

# In order to use the glmnet package, we need to convert our factor variables,
which we can
# do with the model.matrix function
# attach the dataset for convenience
# we can remember to detach later if we want to clear up the namespace
attach(data.full)

# Note that now these features which our stepwise logistic regression treated a
s 1 feature each (either keep or drop all levels) are now being treated as dist
inct variables--the lasso can drop some levels of a feature without dropping th
em all, making direct comparisons difficult

factor.vars <- c("Type", "Reading_Level", "Story_Level")
factor.indices <- which(names(data.full) %in% factor.vars)
# this will transform our factor variables into a matrix of binary variables
xfactors <- model.matrix(reformulate(factor.vars, "success"))[, -1]
# Let's just check that we have the right number of indicator variables
dim(xfactors)[2] == (length(levels(Type)) - 1 +
                    length(levels(Reading_Level)) - 1 +
                    length(levels(Story_Level)) - 1)
```

```
    [1] TRUE
```

```
data.formatted <- cbind(subset(data.full, select = -c(factor.indices,
                                 -non.target.index)), xfactors)
data.formatted <- as.matrix(data.formatted)
y <- as.factor(success)
detach(data.full)
#dim(data.formatted)
#length(y)
```

So now after transforming our three categorical variables into sets of indicator variables, it looks like we now have 57 features in the data.

The data is now in a format that can be used by the glmnet package, which fits a generalized linear model (here just a logistic regression) via a penalized maximum likelihood. Here, we are starting with a basic lasso regression. This will add a penalty term to the logistic regression that is based on the sum of the absolute values of the weights fitted to the regression. Lasso has a characteristic of inducing sparsity into the model (in the model with the best fit, we will typically see many of the features fitted with weight zero, so these features get dropped).

```
# Let's fit a GLM based on logistic regression with lasso penalty
# Data is standardized by default (response variable is not standardized)
# By convention, we also standardize the indicator variables, which has been
# shown to be good/okay for performance but bad for interpretability of coeffic
ients

# We will use cv.glmnet, which does cross-validated logistic regression
# Automatically chooses a range of lambda values, with 10 fold-cross validation
lasso.log.regression <- cv.glmnet(data.formatted, y, family="binomial",
                                  type.measure= "auc")
```

Here, we use AUC as a measure of how well the model fit the data. The cv.glm package searches over values of the regularization parameter, with a larger regularization parameter tending to push more feature coefficients towards zero.

```
# Let's get the top ten most important features from the glm
lasso.feature.imp <- varImp(lasso.log.regression$glmnet.fit,
                            lambda = lasso.log.regression$lambda.min)
sorted <-  with(lasso.feature.imp, order(-Overall)) #index of most imp features
mostImportant <- rownames(lasso.feature.imp)[sorted] #names of most imp feature
s
pander(head(cbind(features=mostImportant,
                  feature.importance=lasso.feature.imp[mostImportant,]), 10), d
igits = 3)
```

features	feature.importance
mean_success	3.7276213436384
mean_word_success	0.911172832320429
mean_student_level_latency	0.210530981650319
mean_word_latency	0.163098447431091
Reading_LevelD	0.162391602081006
Story_LevelG	0.158352918817848
Story_LevelC	0.126790407891631

TypeSpellOut	0.11728336058272
Reading_LevelC	0.0750493113826054
is.number	0.0673174114016073

Quick note: let’s ignore the fact that Type of help = SpellOut appears to be an important feature. It perfectly predicts success, but only because there happen to be 2 instances of SpellOut in our dataset of size 60917.

```
best.lambda <- lasso.log.regression$lambda.min
index.best.lambda <- which(lasso.log.regression$lambda == best.lambda)
# This will give us the number of non-zero features in the regression fit
# with the maximum mean AUC across the 10 folds
num.non.zero.features <- lasso.log.regression$nzero[index.best.lambda]
# coefficients of each feature using the optimal value of lambda (including zer
os)
coefficients <- coef(lasso.log.regression, s="lambda.min")
index.non.zero.features <- summary(coefficients)$i
# get the names of all features not set to zero coefficient in the lasso regres
sion
non.zero.features <- rownames(coefficients)[index.non.zero.features]

# get the names of all features set to zero coefficient
pred <- colnames(data.formatted)
cv.glm.dropped <- pred[!(pred %in% non.zero.features)]
auc <- lasso.log.regression$cvm[index.best.lambda] # get AUC of optimal lasso m
odel
```

```
# List of non-zero features in the cross-validated lasso regression:
# non.zero.features
# List of features dropped by the cross-validated lasso regression:
# cv.glm.dropped

in.model <- ldply(pred, function(x) ifelse(x %in% non.zero.features, "kept", "d
ropped"))
colnames(in.model) <- "kept.in.model"
df <- cbind(features=pred, in.model)
kable(df[order(df$kept.in.model, decreasing=TRUE),], row.names=FALSE, align='c'
)
```

features	kept.in.model
frequency	kept
PREV	kept

is.number	kept
word_length	kept
is.letter	kept
mean_latency	kept
num_student_trials	kept
mean_success	kept
mean_word_latency	kept
num_word_trials	kept
mean_word_success	kept
mean_student_level_latency	kept
mean_student_level_success	kept
num_help_types	kept
Autophonics	kept
Recue	kept
RhymesWith	kept
SoundEffect	kept
StartsLike	kept
Syllabify	kept
WordInContext	kept
is.singleton	kept
TypeRecue	kept
TypeSayWord	kept
TypeShowPicture	kept
TypeSoundOut	kept
TypeSpellOut	kept
TypeSyllabify	kept
Reading_LevelB	kept
Reading_LevelC	kept

Reading_LevelD	kept
Reading_LevelE	kept
Reading_LevelK	kept
Story_LevelB	kept
Story_LevelC	kept
Story_LevelE	kept
Story_LevelG	kept
Story_LevelH	kept
Story_LevelP	kept
target_word_number	dropped
OnsetRime	dropped
PlaybackWord	dropped
SayWord	dropped
ShowPicture	dropped
SoundOut	dropped
SpellOut	dropped
TypeOnsetRime	dropped
TypePlaybackWord	dropped
TypeRhymesWith	dropped
TypeSoundEffect	dropped
TypeStartsLike	dropped
TypeWordInContext	dropped
Reading_LevelF	dropped
Reading_LevelG	dropped
Story_LevelD	dropped
Story_LevelF	dropped
Story_LevelK	dropped

So using cross-validated regularized logistic regression, the model fit gave us 39 non-zero features out of 0 features. The AUC for this model is 0.617. This is hardly an improvement over the AUC of the non-regularized logistic regression, with an AUC of 0.609.

```
help.type.index <- grep("Type+", colnames(data.formatted))
help.available.index <- which(colnames(data.formatted) %in% levels(data.full$Ty
pe))
data.formatted.minus.help <- data.formatted[, -c(help.type.index,
                help.available.index)]

lasso.no.help.types <- cv.glmnet(data.formatted.minus.help, y, family="binomial
", type.measure= "auc")
best.lambda2 <- lasso.no.help.types$lambda.min
index.best.lambda2 <- which(lasso.no.help.types$lambda == best.lambda2)
# This will give us the number of non-zero features in the regression fit
# with the maximum mean AUC across the 10 folds
num.non.zero.features2 <- lasso.no.help.types$nzero[index.best.lambda2]
coefficients2 <- coef(lasso.no.help.types, s="lambda.min")
index.non.zero.features2 <- summary(coefficients2)$i
non.zero.features.without.help <- rownames(coefficients2)[index.non.zero.featur
es2]
auc.2 <- lasso.no.help.types$cvm[index.best.lambda2]

# List of non zero features when help type is not included in the model
#non.zero.features.without.help
# Let's make a clean table showing the features that have been kept versus drop
ped
pred <- colnames(data.formatted.minus.help)
in.model <- ldply(pred, function(x) ifelse(x %in% non.zero.features.without.hel
p, "kept", "dropped"))
colnames(in.model) <- "kept.in.model"
df <- cbind(features=pred, in.model)
kable(df[order(df$kept.in.model, decreasing=TRUE),], row.names=FALSE, align='c'
)
```

features	kept.in.model
frequency	kept
PREV	kept
is.number	kept
word_length	kept
is.letter	kept

mean_latency	kept
num_student_trials	kept
mean_success	kept
mean_word_latency	kept
num_word_trials	kept
mean_word_success	kept
mean_student_level_latency	kept
mean_student_level_success	kept
num_help_types	kept
is.singleton	kept
Reading_LevelB	kept
Reading_LevelC	kept
Reading_LevelD	kept
Reading_LevelE	kept
Reading_LevelF	kept
Reading_LevelK	kept
Story_LevelB	kept
Story_LevelC	kept
Story_LevelE	kept
Story_LevelG	kept
Story_LevelH	kept
Story_LevelP	kept
target_word_number	dropped
Reading_LevelG	dropped
Story_LevelD	dropped
Story_LevelF	dropped
Story_LevelK	dropped

So using the same type of model but excluding the features related to help type, the model fit gave us 27 non-zero features out of 32 features. The AUC for this model is 0.617. Compare this to the AUC for the model with help type features included: AUC = 0.617.

```
intercept <- glm(success ~ 1, data=data.full, family=binomial)
full <- glm(reformulate(predictors, "success"), data=data.full, family=binomial
)
total.scope <- list(lower=formula(intercept), upper=formula(full))
backwards <- step(full, direction="backward", trace=0, scope=total.scope)
forwards <- step(intercept, direction="forward", trace=0, scope=total.scope)
```

```
# Note that we can also use stepAIC in the MASS library to do stepwise regressi
on
# However, stepwise regression is really slow and statistically terrible

backwards$formula
```

```
    success ~ Reading_Level + Story_Level + PREV + word_length +
        is.letter + num_student_trials + mean_success + mean_word_latency +
        num_word_trials + mean_word_success + mean_student_level_latency +
        num_help_types + OnsetRime + SoundEffect + SoundOut + StartsLike
```

```
forwards$formula
```

```
    success ~ mean_success + mean_word_success + PREV + num_help_types +
        Story_Level + Reading_Level + num_word_trials + StartsLike +
        word_length + is.letter + num_student_trials + SoundOut +
        SoundEffect + OnsetRime + mean_word_latency + mean_student_level_latenc
y
```

```
backwards.predictors <- attributes(terms(backwards$formula))$term.labels
forwards.predictors <- attributes(terms(forwards$formula))$term.labels
# Are the predictors chosen by the forwards and backwards stepwise regression t
he same?
all(backwards.predictors %in% forwards.predictors) &
  length(backwards.predictors) == length(forwards.predictors)
```

```
    [1] TRUE
```

```
# List of all the predictors that were dropped in the stepwise logistic regress
ion
# Note that this regression is not cross-validated in any way
dropped.stepwise <- predictors[!(predictors %in% backwards.predictors)]
dropped.stepwise
```

```
     [1] "target_word_number"         "Type"
     [3] "frequency"                  "is.number"
     [5] "mean_latency"               "mean_student_level_success"
     [7] "Autophonics"                "PlaybackWord"
     [9] "Recue"                      "RhymesWith"
    [11] "SayWord"                    "ShowPicture"
    [13] "SpellOut"                   "Syllabify"
    [15] "WordInContext"              "is.singleton"
```

```
# Let's fit a GLM based on logistic regression with ridge penalty
# Automatically chooses a range of lambda values, with 10 fold-cross validation
ridge.log.regression <- cv.glmnet(data.formatted, y, family="binomial",
                                  type.measure= "auc", alpha = 0)
# Let's just compare to be sure that the ridge regression (which doesn't have t
he advantage of
# inducing sparsity) isn't greatly outperforming our lasso regression model.
best.lambda.ridge <- ridge.log.regression$lambda.min
index.best.lambda <- which(ridge.log.regression$lambda == best.lambda.ridge)
auc.ridge <- ridge.log.regression$cvm[index.best.lambda]
# Does ridge regression do better than lasso here?
print(ifelse(auc.ridge > auc, "Ridge (L2) penalized logistic regression outperf
orms lasso (L1)", "Lasso (L1) penalized logistic regression outperforms ridge")
)
```

```
    [1] "Lasso (L1) penalized logistic regression outperforms ridge"
```

Because we've generated a lot of features, many of which might not be meaningful/predictive, it's helpful to have a regularization penalty that tends to induces sparsity in our feature set, and if we get better performance (or at least no worse), then we can be comfortable using this kind of penalty for complexity.

```
xType <- ifelse(length(grep("Type+", cv.glm.dropped)) ==
         (length(levels(data.full$Type)) - 1), "Type", NA)
xReading <- ifelse(length(grep("Reading_Level+", cv.glm.dropped)) ==
         (length(levels(data.full$Reading_Level)) - 1), "Reading_Level", NA)
xStory <- ifelse(length(grep("Story_Level+", cv.glm.dropped)) ==
        (length(levels(data.full$Story_Level)) - 1), "Story_Level", NA)
cv.glm.dropped.reformatted <- cv.glm.dropped
find.factors <- c(grep("Type+", cv.glm.dropped),
                  grep("Reading_Level+", cv.glm.dropped), grep("Story_Level+",
cv.glm.dropped))
cv.glm.dropped.reformatted <- cv.glm.dropped.reformatted[-(find.factors)]
cv.glm.dropped.reformatted <- c(cv.glm.dropped.reformatted, xType, xReading, xS
tory)
cv.glm.dropped.reformatted <- cv.glm.dropped.reformatted[!is.na(cv.glm.dropped.
reformatted)]

# features dropped from the stepwise regression
# dropped.stepwise
replacement <- paste("Type",levels(data$Type), sep="")
dropped.stepwise[which(dropped.stepwise == "Type")] <- replacement[1]
dropped.stepwise <- c(dropped.stepwise, replacement[-1])
# features dropped from the lasso regression (including indicator variables)
# cv.glm.dropped
# features dropped from the lasso regression, where factor variables are only c
onsidered
# dropped if all levels of the indicator variables for that factor are dropped
# cv.glm.dropped.reformatted
pred <- colnames(data.full)
dropped <- ldply(pred, function(x) {lasso=ifelse(x %in% cv.glm.dropped, "droppe
d", "kept")
  stepwise=ifelse(x %in% dropped.stepwise, "dropped", "kept")
  cbind(lasso, stepwise)})
rownames(dropped) <- pred
pander(dropped)
```

	lasso	stepwise
target_word_number	dropped	dropped
Type	kept	kept
Reading_Level	kept	kept
Story_Level	kept	kept
frequency	kept	dropped

PREV	kept	kept
is.number	kept	dropped
word_length	kept	kept
is.letter	kept	kept
success	kept	kept
mean_latency	kept	dropped
num_student_trials	kept	kept
mean_success	kept	kept
mean_word_latency	kept	kept
num_word_trials	kept	kept
mean_word_success	kept	kept
mean_student_level_latency	kept	kept
mean_student_level_success	kept	dropped
num_help_types	kept	kept
Autophonics	kept	dropped
OnsetRime	dropped	kept
PlaybackWord	dropped	dropped
Recue	kept	dropped
RhymesWith	kept	dropped
SayWord	dropped	dropped
ShowPicture	dropped	dropped
SoundEffect	kept	kept
SoundOut	dropped	kept
SpellOut	dropped	dropped
StartsLike	kept	kept
Syllabify	kept	dropped
WordInContext	kept	dropped
is.singleton	kept	dropped

```
# Do we want to drop some of these uninformative features from the dataset comp
letely?
# Should we try PCA or some other dimensionality reduction instead?
```

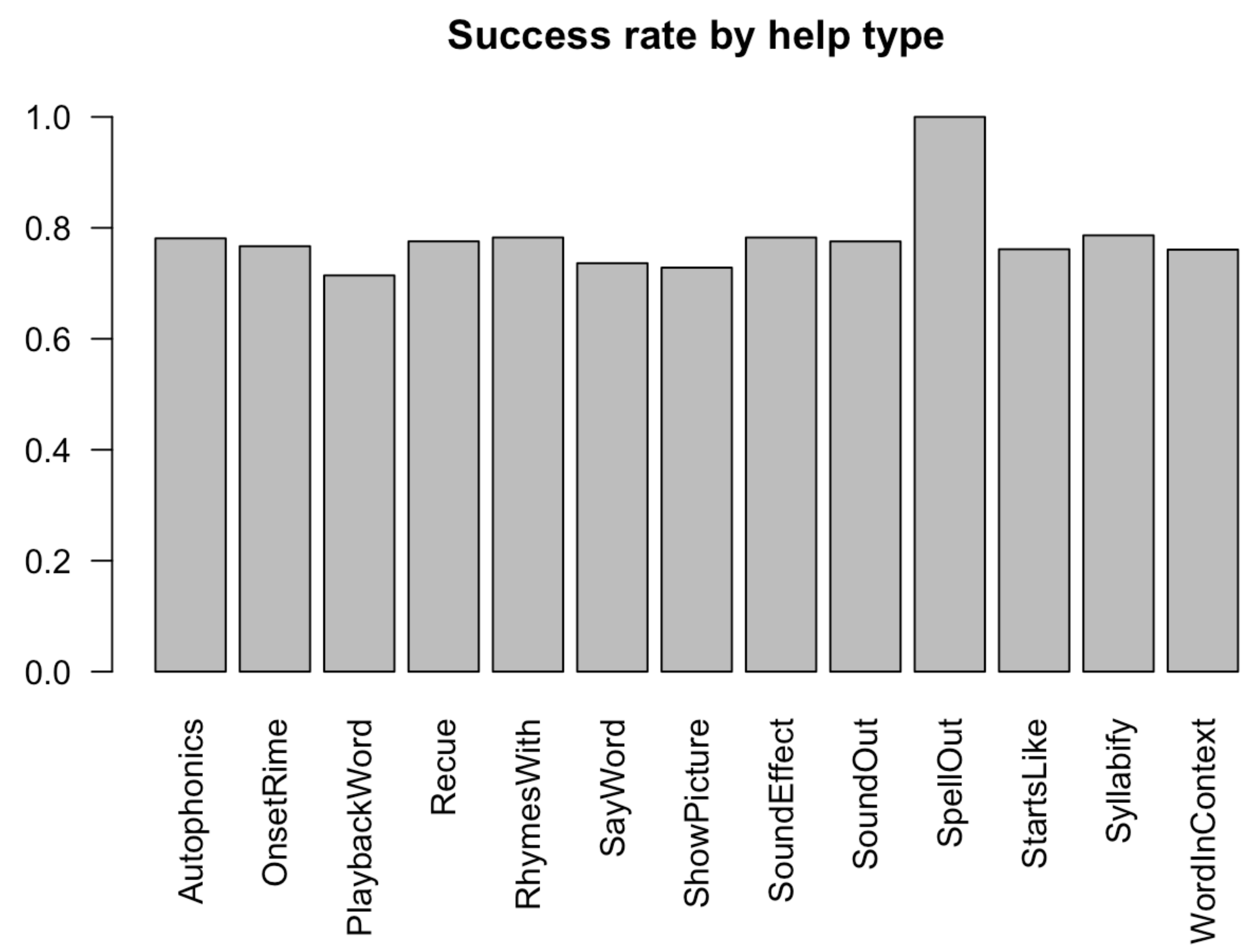
It’s important to take a moment and remember the ultimate goal of our analyses. We don’t really care about our ability to predict the success variable. We do care about understanding if we can improve performance by assigning specific help types in specific contexts, and the first step in answering this question boils down to understanding the relationship between help types and the success variable. Looking at the stepwise logistic regression and cross-validated lasso logistic regression, we can examine the features dropped by those models to understand which features don’t really provide us any new information about the success of a trial. We don’t want to overinterpret–the stepwise regression is not the most methodologically sound choice and the lasso regression will drop features that are predictive of the target variable if their information overlaps too much with another feature–but it does appear to be the case that in all of these candidate models, our conclusions indicate that the features related to the help types are not terribly useful in predicting whether a trial was successful or not.

So it may be the case that overall, no one help type is more or less conducive to the success of a reading trial (even after controlling for other features of the data), but we might be able to find some subsets of the data where help type matters (there could be some complex latent interaction terms that call for the use of a non-linear model).

```
# Let's take a quick look at the average success rate within each help type.
x <- split(data.full, data.full$Type)
summary(data.full$Type)
```

```
  Autophonics    OnsetRime  PlaybackWord         Recue   RhymesWith
         6515         4771             7          4703         3865
      SayWord  ShowPicture   SoundEffect      SoundOut     SpellOut
        21673          254            69          8203            2
   StartsLike    Syllabify WordInContext
         4708         1574          4573
```

```
help_means <- sapply(x, function(type) mean(type$success))
par(mar=c(7,4,4,2) + 0.1)
barplot(help_means, main="Success rate by help type", xlab = "", las=2)
```



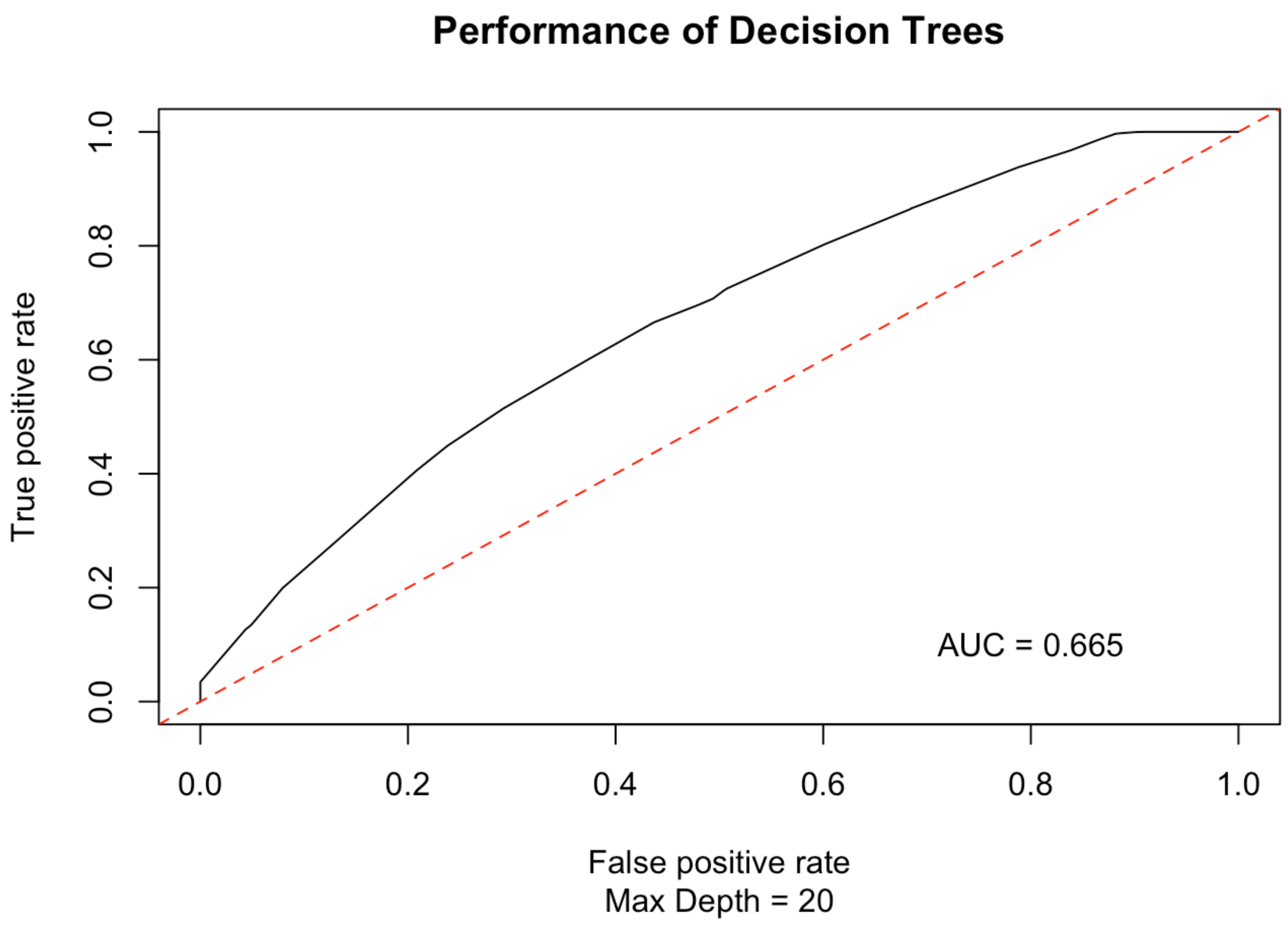
```r
n = length(levels(data.full$Type))
```

Let’s try building a non-linear partitioning model to see if we can fit the data better than a GLM. Ideally, we would like to see whether help type is important in constructing the decision tree, and as a next step in my analysis, I would like to try fitting a different decision tree to the data from each help type, to understand if the help type allows us to differentiate which features are most important to predicting the success of a trial– maybe there is a feature that is strongly related to success in when the help type provided was Autophonics, but doesn’t make a difference in predicting success when the help type was RhymesWith (or maybe the feature has a relationship of the opposite direction).

So we eventually want to build 13 different decision tree models, to represent the subsets of data created by each help type. As a first pass through the data, let’s create a single model, including all help types together.

```
get.tree.by.depth <- function(depth, train, test) {
  tree <- ctree(success ~ ., data=train, control = ctree_control(maxdepth =
depth))
  tree.pred <- predict(tree, newdata = test, type = "prob")
  predict.tree <- prediction(unlist(tree.pred), test$success)
  return(predict.tree)
}

pred.tree <- get.tree.by.depth(20, train, test)
perf.ROC <- performance(pred.tree, "tpr", "fpr") # build an ROC curve for the t
est data
plot(perf.ROC, main="Performance of Decision Trees", sub="Max Depth = 20")
abline(0, 1, lty=2, col=2)
print_perf = paste("AUC =", round(performance(pred.tree, "auc")@y.values[[1]],
digits=3))
text(0.8, 0.1, print_perf) # displays the AUC on the ROC plot
```



```
# TODO try random forest
# try cross-validating tree depth parameter
# try manually altering tree depth parameter
```

```
# Note: varimp and varimpAUC are in party package, only work with random forest
(cforest objects)
# varImp from the caret package works with rpart objects but not ctree object
#simple.tree.feature.imp <- varimpAUC(default.forest)
tree.depth.20 <- rpart(success ~ ., data=train, control = rpart.control(maxdept
h = 20,
                                    usesurrogate = 0))
simple.tree.feature.imp <- varImp(tree.depth.20)
sorted <-  with(simple.tree.feature.imp, order(-Overall)) #index of most imp fe
atures
mostImportant <- rownames(simple.tree.feature.imp)[sorted] #names of most imp f
eatures
pander(head(cbind(features=mostImportant,
                  feature.importance=simple.tree.feature.imp[mostImportant,]),
5), digits = 3)
```

features	feature.importance
mean_success	0.0199883586039385
mean_latency	0.0125724439982522
mean_word_success	0.00822168341250701
Reading_Level	0.00493183328217905
mean_word_latency	0.00429155523129568

- Not entirely sure how the rpart tree is working – it may not be doing the same thing as the party::ctree. Need to look into this further

```
# TODO
# Does it make sense to try:
# Cross validated decision trees/ random forests?
# Merge in skills data - skills for each phoneme of each word - think about how
to
# incorporate into the current data set
# idea: use grapheme to phoneme mappings to cluster words and build a new model
for each word

# Clustering (model based) to try to define clusters in the data based on...
# Find the help type with highest success rate within each cluster
# Exclude help type data when clustering?
# Discretization of continuous variables
```

CONCLUSIONS:

There is still a lot of analysis to be done. We haven't yet answered the question of whether help types matter for specific students or in specific situational contexts. The next step will be to incorporate a cognitive skills model of what specific cognitive skills and what grapheme to phoneme mappings are involved in each utterance– it is possible this makes a substantial difference in differentiating between different help types in how strongly they are associated with later successful outcomes for the student users.

We can conclude that on the whole, help types are not particularly associated with successful reading encounters the next time the student reads a word. That is, it does not appear that certain help types are useful or not useful overall, for all students or for all words. However, it's difficult to draw too many conclusions about feature importance from the range of candidate models we have tried, since the predictive performance of our model suggests that there is certainly room for improvement in fitting our model.

Moving forward, it could be interesting to try fitting 13 different models, one for each different help type, to predict success from the remaining features. Then, for new data, we could make predictions from each of the help type models that are possible for the given word, and select the help type that gives the greatest predicted probability of success.