CS 145, Fall 2018
Yelp Rating Prediction Midterm Report

TEAM: Omae wa mou shindeiru
Jonathan Chang
104853981

1 Problem Definition

It is worthwhile to predict how users would rate businesses based on past ratings or other implicit information that they offer up. These predictions could be used to serve more relevant content, improving user appreciation, but also driving revenue. In this case, we analyze a subset of Yelp datasets consisting of users and implicit information, business information, and a partial list of explicit ratings and reviews, which serves as a training set. Given other user-business combinations, our goal is to predict how those users would rate those businesses.

Such an undertaking requires algorithmic analysis of data. The first step is to preprocess our features and transform them into formats conducive to model use and speed. This requires basic understanding of how models work and their requirements, as well as efficient methods to manipulate large amounts of data. Next, using our intuition and understanding of the data, we must make informed choices about the models and hyperparameters used. We should also design visualizations to help us achieve better understanding of relations implicit in the data.

2 Proposed Methods

Methods will mostly divide into three tasks, which form a feedback loop:

(a) Learning the data.

(b) Data preprocessing.

(c) Tuning the model.

In this section, I will discuss the steps taken in each task

2.1 Learning the data

Machine learning and data mining are wide-spanning tasks with no well-defined process. They depend on intimate knowledge of the data. Specifically, we need to know the explicit features, the implications of those features, how they relate, what information is useful, what information is missing, etc. With that, it was immediately obvious that since our reviews only exist in the training set, they have questionable usefulness. The follow up problem would be the methodology

of how the dataset was created. If it was randomly sampled, then we want to know how many users there are, whether there are the same users in the training, validation and test set, how many friends they have, what their average rating is, etc.

This section might as well be the grunt work, since there isn't any concrete progress, yet future success depends on it. Specifically, we found that validation and test queries have up to 40% users that were not included in the training set, further limiting usefulness of reviews. We also found that the median number of friends is 2-3 per user, although some socialites greatly increase the mean. This means that if we only examine the immediate neighbors in a graph, the majority of analysis would only pertain to a small number of users – once again calling into question the usefulness of the data.

2.2 Preprocessing

Preprocessing involves figuring out what format the data needs to be in such that they are compatible to model algorithms, and feature engineering to improve their usefulness. Luckily, we don't have any missing data, or we would have had to impute the mean. However, there were still plenty to do.

Perhaps the first task is to reduce file and memory space needed by simplifying fields, and removing unnecessary ones. User IDs and business IDs take up 22 characters each, but we only have $< 10^6$ users and businesses. Converting them to a sequence of integers will not only make them easier to search, but drastically decrease file size, especially the user IDs from the friends list. Fields that we may not used could be separated into their own file, so that we could recombine them if necessary. Review text and friends list in particular take a lot of space. Fields with too fine or wide granularity to be useful can also be removed, such as addresses or states.

Most algorithms cannot deal with some feature formats well. Numerical features were normalized so as to limit bias from large distances. Categorical features needed to be one-hot or label encoded. Label encoding converts factor levels into numbers, preserving some relation between the categories, but also introducing an ordinal relation which may be disastrous. One-hot encoding does not suffer from the ordinal problem, but also eliminates all relation between different categories of the same features. They also drastically increase number of features. To reduce feature dimensions, we experimented with PCA (principle component analysis), and also a spin on Word2Vec (more like Category2Vec) to preserve relations. However, empirical testing between these sets with the XGBoost model show no significant difference.

Since there is a high level of redundancy between geographical fields like postal code, neighborhood, state, and city, we attempted to find some balance to combine the categories that were too thin, and delete the rest. For example, Canadian postal codes numbered hundreds to each city, while several states only had a few samples. A heuristic was used to combine these fields.

2.3 Tuning the model

We needed a suite of functions to be able to effectively evaluate and compare the results of different models, so we created functions that easily take predictions and ground truths, and return a report of accuracy, precision, recall of each predicted star rounded, so we could identify the strengths and weaknesses of every model. We will add graphs as necessary.

(a) Mean (baseline)

As a baseline, we experimented with total user means, means per each user, total business mean, and means per each business. Using this as predictions, we tested on the validation set, and decided that means per each user was superior. This was submitted to get an RMSE of 1.13383. This sets a starting point so we know what is and is not worth pursuing.

(b) Random forest

Random Forest, or bagging with trees, is a fairly robust starting point that gives decent results out of the bag, although we extensively grid searched all of its parameters to find something closer to the optimal value for our data set. We used the library *Ranger* for R. Training each model took several hours, so we decided to move some of the processing to the cloud with AWS EC2 instances and R Server as the production environment. The resulting models were saved to a file store and downloaded for analyses. Although there are some further fine tuning to be done, we were able to achieve and RMSE of 1.05800.

(c) XG Boost (Extreme Gradient)

XGBoost is a competition winner and somewhat algorithmically close to trees, so we decided to try it out. We did an extensive grid search. The library used is *xgboost* for R as well, and has an early stopping mechanism to find optimal iterations. Despite running through hundreds of combinations, we only achieved an RMSE of 1.172 on the validation set, and decided it was not worth submitting at this moment.

(d) Future steps

A recent variant of the gradient boost is CatBoost (Category Boost) developed by the Russian company Yandex. It seems CatBoost would be better suited for the preponderance of categorical features in the business set, since XGBoost seems to specialize in numerical data. After this, we plan to try fitting a feed-forward neural network, before moving on to recommender system models based around explicit information and a user-item matrix. We have already scouted several papers, such as one that uses an autoencoder, and another that uses low rank factorization, that we would like to try implementing. Further development would probably move toward Python to improve speed.

Further documentation are attached below.

Test the Completeness of Reviews

Jonathan Chang

October 31, 2018

The Problem

It should be immediately obvious that the data contained in 'train_reviews.csv' have limited usefulness if it doesn't pertain to users and businesses in the test or validation set. If it's not useful, we may not want to waste time processing the review text.

Load the relevant data

We remove review text to improve processing speed, since we are not working with that field at this moment.

```
reviews = read.csv('train_reviews.csv')
validate = read.csv('validate_queries.csv')
test = read.csv('test_queries.csv')
reviews$text = NULL
```

```
## There are 150232 reviews, 50077 validation queries, and 50078 test queries.
```

Non-unique Users

First, we can find the number of queries in the validation and test set by users that we do not have reviews for.

```
non_unique_val = nrow(anti_join(validate, reviews, by="user_id"))
non_unique_test = nrow(anti_join(test, reviews, by="user_id"))
```

```
## There are 19096 validation queries and 19029 test queries with users we don't have
reviews for. That's, 38.13 % and 38 % respectively.
```

Unique Users

We might also want to find how many unique users this represent.

```
validate_users = as.data.frame(unique(validate$user_id))
test_users = as.data.frame(unique(test$user_id))
colnames(validate_users) = c("user_id")
colnames(test_users) = c("user_id")
unique_val = nrow(anti_join(validate_users, reviews, by="user_id"))
unique_test = nrow(anti_join(test_users, reviews, by="user_id"))
```

```
## There are 8823 users in validation and 8707 users in test we don't have reviews fo
r. That's, 62.38 % and 62.44 % respectively.
```

We might also want to find out how many test queries lack both user and business in reviews.

```
test_business = as.data.frame(unique(test$business_id))
shutout = test[!(test$user_id %in% reviews$user_id),]
shutout = shutout[!(shutout$business_id %in% reviews$business_id),]
num_shutout = nrow(shutout)
shutout_users = length(unique(shutout$user_id))
shutout_business = length(unique(shutout$business_id))
```

```
## There are 3563 test queries without both users and businesses in the review set. T
his corresponds to 2479 ( 17.78 %) unique users and 1247 ( 11.89 %) unique businesses
.
```

Mean Modeling

Jonathan Chang

October 31, 2018

Baseline

As a baseline, before we start using machine learning models, it makes sense to predict with the mean to see what kind of ball park error we should expect.

But first we should cop a rough feel for the data.

```
users = read.csv('users.csv')
business = read.csv('business.csv')
reviews = read.csv('train_reviews.csv')
review_count = sum(users$review_count)
user_mean = mean(users$average_stars)
user_median = median(users$average_stars)
business_mean = mean(business$stars)
business_median = median(business$stars)
review_mean = mean(reviews$stars)
review_median = median(reviews$stars)
```

```
## Reviews written by all users: 1771074
```

```
## Mean stars by a user: 3.74432166826462
```

```
## Median stars by a user: 3.89
```

```
## Mean stars to a business: 3.78300713219439
```

```
## Median stars to a business: 4
```

```
## Mean stars of reviews we have: 3.76414478939241
```

```
## Median stars of reviews we have: 4
```

The distribution in each set seems pretty close. There's no major skew. We could also test variance, but I think it's pretty reasonable to assume that each set of representative overall.

Validation Error

Let's run some tests on the validation set to see what kind of RMSE we should expect.

```
validate = read.csv('validate_queries.csv')
val_users = inner_join(validate, users, by="user_id")
error = mean(val_users$average_stars) - validate$stars
rmse_overall_user = sqrt(mean(error^2))

error = val_users$average_stars - validate$stars
rmse_each_user = sqrt(mean(error^2))

business_cols = colnames(business)
business_cols[60] = "average_stars"
colnames(business) = business_cols
val_business = inner_join(validate, business, by="business_id")
error = mean(val_business$average_stars) - validate$stars
rmse_overall_business = sqrt(mean(error^2))

error = val_business$average_stars - validate$stars
rmse_each_business = sqrt(mean(error^2))
```

```
## RMSE using mean rating of all users: 1.26332327471125
```

```
## RMSE using mean rating of each user: 1.11838053741079
```

```
## RMSE using mean rating of all business: 1.26211500644174
```

```
## RMSE using mean rating of each business: 1.17600815788471
```

Prediction

Note: In hindsight, I should've used the validation in the average, but it doesn't matter. We'll get much better scores later. :)

We use the mean rating of each user as a baseline, since it scored the best.

```
test = read.csv('test_queries.csv')
test_users = inner_join(test, users, by="user_id")
avg_stars = test_users$average_stars
submit = data.frame(seq.int(0, length(avg_stars)-1), avg_stars)
colnames(submit) = c("index", "stars")
```

This received a **1.13383**.

```
write.csv(submit, 'submit_baseline.csv', row.names=FALSE, quote=FALSE)
```

Next, we'll try rounding to the nearest whole. Since RMSE is mean error, it doesn't really make sense to round from the mean, but we aren't lacking in submissions…

```
submit$stars = round(submit$stars)
```

This received a **1.16683**, which is a bit worse, as expected.

```
write.csv(submit, 'submit_mean_rounded.csv', row.names=FALSE, quote=FALSE)
```

Simplify IDs

Jonathan Chang

October 31, 2018

Motivation

Previous operations have revealed that large file sizes take up more memory, slowing down the computer significantly. Most of the files are, however, taken up by fields that are unnecessarily long. Case in point: IDs. These are 22 characters, when observations are in the 5 digits. Therefore, we will convert them to a numerical sequence starting from 1 to save space and make processing easier.

Save Mappings

First, we will create separate files with a map of the new IDs to old IDs, in case we need to revert for whatever reason.

```
users = read.csv('users.csv', stringsAsFactors=FALSE)
uid = seq.int(nrow(users))
user_id_map = data.frame(user_id=users$user_id, uid=uid)

friends = data.frame(uid=uid, users$friends)

business = read.csv('business.csv')
bid = seq.int(nrow(business))
business_id_map = data.frame(business_id=business$business_id,
                             bid=bid)
```

This data will be saved, so we can process to our hearts' content without worrying about making mistakes.

```
write.csv(user_id_map, 'user_id_map.csv', row.names=FALSE)
write.csv(business_id_map, 'business_id_map.csv', row.names=FALSE)
write.csv(friends, 'friends.csv', row.names=FALSE)
```

Need New Friends

We also want to convert all user friends into the new ID system, since that takes up a huge amount of space. However, this is too slow in R, so I've written a script in Python. Note that this is contingent on 'user_id_map.csv' and 'users.csv'. The code is as such:

```
import pandas as pd
uid_map = pd.read_csv('user_id_map.csv')
hash = {}
for index, row in uid_map.iterrows():
    hash[row['user_id']] = str(row['uid'])
print("Completed building map.")
data = pd.read_csv('users.csv')
friends_list = data['friends'].values
for index, friends in enumerate(friends_list):
    if friends != "None":
        friends = friends.split(', ')
        length = len(friends)
        f_new = []
        for i in range(length):
            try:
                if friends[i] in hash:
                    f_new.append(hash[friends[i]])
            except:
                print("Error on...\nFriends: {0}\nIndex: {1}".format(friends, i))
                exit()
        if len(f_new) == 0:
            data.at[index, 'friends'] = "None"
        else:
            data.at[index, 'friends'] = ', '.join(f_new)
    if index % 1000 == 0:
        print("On row {0}...".format(index))
data.to_csv('users_simplified.csv', index=False)
```

During this process, I noticed that a number of friends also do not exist in our user set. They are removed, since we have no information about them, so they are useless to us.

Other Files

We take this opportunity to replace user ID and business ID from reviews, validate, and test sets.

```
import pandas as pd
uid_map = pd.read_csv('user_id_map.csv')
bid_map = pd.read_csv('business_id_map.csv')
uid = {}
for index, row in uid_map.iterrows():
    uid[row['user_id']] = str(row['uid'])
print("Completed building user id map.")
bid = {}
for index, row in bid_map.iterrows():
    bid[row['business_id']] = str(row['bid'])
print("Completed building business id map.")
def simplify(infile, outfile):
    data = pd.read_csv('{0}.csv'.format(infile))
    for index in range(len(data.index)):
        data.at[index, 'user_id'] = uid[data.at[index, 'user_id']]
        data.at[index, 'business_id'] = bid[data.at[index, 'business_id']]
    data.rename(columns={'user_id': 'uid', 'business_id': 'bid'}, inplace=True)
    data.to_csv('{0}.csv'.format(outfile), index=False)
simplify('train_reviews', 'reviews_simplified')
simplify('validate_queries', 'validate_simplified')
simplify('test_queries', 'test_simplified')
```

Final Steps

Now we'll finalize the change and delete some useless fields.

```
users = read.csv('users_simplified.csv')

users$uid = uid
business$bid = bid

users$user_id = NULL
users$name = NULL

business$business_id = NULL
business$address = NULL
business$attributes = NULL
business$hours = NULL
business$is_open = NULL
business$name = NULL
```

Finally, we'll write our new version of the data files.

```
write.csv(users, 'users_simplified.csv', row.names=FALSE)
write.csv(business, 'business_simplified.csv',
          row.names=FALSE)
```

File size has decreased massively!

```
orig_user_size = round(file.size('users.csv')/1000000, 2)
new_user_size = round(file.size('users_simplified.csv')/1000000, 2)
orig_business_size = round(file.size('business.csv')/1000000, 2)
new_business_size = round(file.size('business_simplified.csv')/1000000,
                          2)
```

```
## Users: 72.7 MB --> 3.38 MB
```

```
## Business: 8.19 MB --> 8.5 MB
```

Factor Review Set

Jonathan Chang

October 31, 2018

Motivation

'train_reviews.csv' is by far the largest file. Most of it is the review text, yet it's questionable how we should use this. The review text is not present in the validation or test set, so we would have to predict the text to generate it. Otherwise, it could be used to identify user similarities, although most algorithms deal with explicit data (i.e. star rating), and the review text seems redundant with the ratings in hand.

To reduce file size, we set aside the text for now, so we could use it in the future it necessary, and it doesn't impede our current business by taking up memory unnecessarily.

```
reviews = read.csv('reviews_simplified.csv')
text = reviews$text
rid = seq.int(nrow(reviews))
text_df = data.frame(rid, text)
colnames(text_df) = c("rid", "text")
```

And we save this so we don't have to worry about spamming the delete button.

```
write.csv(text_df, 'review_text.csv', row.names=FALSE)
```

Now we can greatly simplify the reviews set by doing several things:

- Removing the text field
- Replacing review_id with a numerical sequence
- Converting dates to a numerical representation
- Normalizing large numbers

Note: The first user in our set signed up on 10/12/2004.

```
reviews$text = NULL

reviews$rid = rid
reviews$review_id = NULL

reviews$date = as.integer(as.Date(reviews$date) -
                          as.Date('2004-10-12'), units="days")

numeric_cols = c("cool", "date", "funny", "useful")
reviews[,numeric_cols] = round(scale(reviews[,numeric_cols]), 4)
```

Now we can save our perfectly simplified review set.

```
write.csv(reviews, 'reviews_clean.csv', row.names=FALSE)
```

Let’s compare file sizes. Now it should be much easier to work with.

```
orig_size = round(file.size('train_reviews.csv')/1000000, 2)
new_size = round(file.size('reviews_clean.csv')/1000000, 2)
```

```
## Users: 137.6 MB --> 7.61 MB
```

Clean Up Users

Jonathan Chang

November 1, 2018

Motivation

Preprocessing is the most tedious, but most important part of machine learning. Previously, we cleaned up the reviews set and factored out data we don't need into another file. We've also taken care of simplifying IDs and deleted some useless columns. Now we'll put the finishing touches on 'users.csv' to prepare it for fitting models.

```
users = read.csv('users_simplified.csv')
users$yelping_since = as.integer(as.Date(users$yelping_since)
                                 -as.Date('2004-10-12'), units="days")
users$elite = (users$elite != "None")
non_numbers = c("uid", "elite", "friends")
users[,!(colnames(users) %in% non_numbers)] =
  round(scale(users[,!(colnames(users) %in% non_numbers)]), 4)
users$friends[users$friends == 'None'] = NA
users$friends = factor(users$friends)
```

```
write.csv(users, 'users_clean.csv', row.names=FALSE)
```

Friendly Analysis

I want to separate out the friends list and analyze it a bit to determine its usefulness.

```
friends = data.frame(users$uid, as.character(users$friends))
colnames(friends) = c("uid", "friends")
```

```
write.csv(friends, 'friends.csv', row.names=FALSE)
```

First, let's find the amount of users with any friends.

```
has_friends = table(!is.na(friends$friends))
```

```
## 0.242 % of users have friends. 31630 users have no friends.
```

Ouch! T.T

Amongst the people who do have friends, how many do they have?

```
friends = friends[!is.na(friends$friends),]
friend_count = lengths(strsplit(as.character(friends$friends), ', '))
median_friends = median(friend_count)
total_friends = sum(friend_count)
lteq_3 = table(friend_count <= 3)[2]
lteq_5 = table(friend_count <= 5)[2]
gt_10 = table(friend_count > 10)[2]
```

```
## Median # friends: 2
```

```
## Total # friends: 61601
```

```
## 3 or less friends: 7283
```

```
## 5 or less friends: 8136
```

```
## 10 or less friends: 8961
```

```
## More than 10 friends: 1129
```

Now it's apparent that 7283 people have 3 or less friends, compared to 825 people with between 3-10 friends, and 1129 people with more than 10.

Let's say we assume that people friend other people on Yelp who they agree with, so we roughly assume that a user's friends' reviews are an extension of their own. The idea is that a fully connected user-business review network will pad our training data and give us more to work with. There are several questions:

- Can most users having less than 3 friends help?
- It seems like there's a small number of social elite users having the vast majority of friends. Is it helpful to consider the ratings from hundreds of friends of one user? It seems like a waste of processing power for a small, questionable benefit.
- How many more rows in the training set will we need to add by considering friends?

Unfortunately, only the third question can be answered at this point.

Cost-Benefit Analysis

Let's count the number of reviews each user has made in the training and validation sets. We'll then multiply these numbers by each friend ID in the friends list to see how big the respective sets will get if we consider the entire friends list as an extension.

```
reviews = read.csv('reviews_clean.csv')
validate = read.csv('validate_simplified.csv')
train_freq = reviews %>% group_by(uid) %>% summarise(n=n())
val_freq = validate %>% group_by(uid) %>% summarise(n=n())
```

```
## There are 16556 training set users, and, 14145 validation set users with reviews.
```

```
write.csv(data.frame(train_freq), 'train_freq.csv', row.names=FALSE)
write.csv(data.frame(val_freq), 'val_freq.csv', row.names=FALSE)
```

We're going to use a function that matches up all the 'uid' from the frequency tables to the friends list.

```
import pandas as pd
friends = pd.read_csv('friends.csv').loc[:, "friends"]
def ratings_count(friends, freqs, limit=None):
    hash = {}

    # Create O(1) hash of frequencies,
    # since pandas is O(n)
    for index, freq in freqs.iterrows():
        hash[str(freq['uid'])] = int(freq['n'])
    count = 0
    for friend in friends:
        if not pd.isnull(friend):
            f_list = friend.split(', ')
            # Limit friends per user
            if limit is not None:
                f_list = f_list[0:limit]
            # Count friends
            for uid in f_list:
                if uid in hash:
                    count += hash[uid]
    return count
def report(limit=None):
    train_count = ratings_count(friends, pd.read_csv('train_freq.csv'), limit)
    val_count = ratings_count(friends, pd.read_csv('val_freq.csv'), limit)
    if limit is None:
        preface = "Considering all friends"
    else:
        preface = "If we limit each user to {0} friends".format(limit)
    out = "{0}, there will be {1} and {2}".format(preface, train_count, val_count)
    out = "{0} from training and validation, respectively.".format(out)
    print(out)
report()
```

```
## Considering all friends, there will be 220527 and 77349 from training and validati
on, respectively.
```

```
report(3)
```

```
## If we limit each user to 3 friends, there will be 69683 and 23881 from training an
d validation, respectively.
```

```
report(5)
```

```
## If we limit each user to 5 friends, there will be 88776 and 30345 from training an
d validation, respectively.
```

```
report(10)
```

```
## If we limit each user to 10 friends, there will be 115607 and 40232 from training
and validation, respectively.
```

And this is considering that there are 150232 and 50077 rows in reviews and validation existing. The questions are:

- Can we afford to more than double the space taken in memory?
- Is it beneficial to more than double the space taken in memory? (i.e. Is it beneficial to consider more than *X* number of friends?)

For now, the latter may have to be a judgment call. We can answer the former when we see how much memory the training set takes up without considering any friends first. We can consider this a hyperparameter.

Cleaning Business: Opening Hours (Part 1)

Jonathan Chang

November 1, 2018

Motivation

The ‘business.csv’ data set has by far the most malformatted junk, so we’ll be splitting this up into parts:

1. Opening hours
2. PCA with categories
3. Attribute objects
4. Zip codes and neighborhoods
5. One-Hot & cleanup

What’s wrong with the times?

Business open hours are currently stored as a string of opening to closing times. Machine learning models cannot process strings, so they are usually considered factors. The problem, then, is that there are about 600 levels per each day of the week representing all the combinations of opening and closing hours. When we think about what’s being encoded in here, actually there is just the opening time and closing time. Everything in between is encoded in a dash, which gives us no information at all. Therefore, what we should do is turn every day of the week into 2 columns, integers representing the opening and closing time.

For example, the computer cannot tell the difference between *“9:0-17:30”* and *“9:0-18:0”*. It see them as two completely unrelated factors. The relations, that they open at the same time and close within half an hour of each other, are lost. By splitting them up into *9* and *9* for opening, it’s easy to see they are the same; and *17.5* and *18*, the computer can determine their closeness.

Seeing is believing

```
business = read.csv('business_simplified.csv',
                    stringsAsFactors=FALSE)
business[1:10, c("hours_Monday")]
```

```
##  [1] ""            "11:0-22:0"   "11:0-0:0"    "7:0-14:30"   "11:0-22:0"
##  [6] "9:0-17:0"    "11:0-22:0"   "11:30-22:0"  "20:0-6:0"    "16:30-21:30"
```

Nuts and bolts

```
convert_hours = function(df, day) {
  # Simplify usage by prepending tags for the column names.
  # Then, input only consists of the day of the week.
  hours_name = paste("hours_", day, sep="")
  open_name = paste("open_", day, sep="")
  close_name = paste("close_", day, sep="")
  hours = as.character(df[,c(hours_name)])

  # Split each time range by the '-'.
  hours = t(sapply(strsplit(hours, '-', fixed=TRUE),
                   function(times) {

    # Make sure time isn't blank.
    if (!is.na(times[1])) {

      # Reconstruct "hours:minutes" into
      # double: hours.(fraction of hour).
      open = unlist(strsplit(times[1], ':', fixed=TRUE))
      open = as.double(open[1]) + as.double(open[2])/60
      close = unlist(strsplit(times[2], ':', fixed=TRUE))
      close = as.double(close[1]) + as.double(close[2])/60

      # Time wraps around at midnight, but we want to preserve
      # that relation.
      if (close <= open) {
        close = close + 24
      }

      # Represent the two new columns.
      list(open=open, close=close)

    } else {
      # If time was blank, use -1 for both times.
      list(open= -1, close= -1)
    }
  }))

  # Format list into new dataframe with 2 columns.
  hours = as.data.frame(hours)
  hours$open = unlist(hours$open)
  hours$close = unlist(hours$close)
  colnames(hours) = c(open_name, close_name)

  # Replace with original format.
  df[hours_name] = NULL
  cbind(df, hours)
}

# Convert the format for each day
business = convert_hours(business, "Monday")
```

```
business = convert_hours(business, "Tuesday")
business = convert_hours(business, "Wednesday")
business = convert_hours(business, "Thursday")
business = convert_hours(business, "Friday")
business = convert_hours(business, "Saturday")
business = convert_hours(business, "Sunday")

# Show the new format
business[1:5, c("open_Monday", "close_Monday")]
```

```
##   open_Monday close_Monday
## 1          -1         -1.0
## 2          11         22.0
## 3          11         24.0
## 4           7         14.5
## 5          11         22.0
```

Scale and center

We don't require inputs to be human interpretable, and we want to normalize them so they're weighted properly in regression algorithms.

```
business[,grepl("open_|close_", names(business))] =
  round(scale(business[,grepl("open_|close_", names(business))]), 3)
```

Save

```
write.csv(business, 'business_preclean1.csv', row.names=FALSE)
```

Next we'll fix the categories.

Cleaning Business: PCA with Categories (Part 2)

Jonathan Chang

November 1, 2018

Factors are unordered

```
business = read.csv('business_preclean1.csv')
paste("Category factors:", length(unique(business$categories)), "\n",
      sep=' ') %>% cat()
```

```
## Category factors: 8943
```

I think the snippet above is indicative of a major problem, and not even the main problem we'll be discussing. Although some algorithms are better suited to deal with factors and categories (like trees), there is a problem with factors in general for algorithms which cannot deal with them (like linear regression). Internally, factors are stored as integers. While there's an ordered relation between one integer and the next, a factor and the next are not necessarily related at all. The solution to this is usually to create one-hot encoded columns, where each factor is split into a separate binary column.

But c'mon… we have 8943 factors. This will blow up our file into gigabytes. Closer inspection, however, shows that these factors are combinations of unique categories, so our first step is to find out how many unique categories there are.

Categories, not factors

In the following scripts, we will create a frequency dictionary to see how often each category appears and sort them from highest to lowest.

```
# List categories of each business.
categories = as.character(business$categories)
categories = strsplit(categories, ', ', fixed=TRUE)

# List all unique categories.
cat_list = Reduce(f = union, x = categories)
cat_list[1:10]
```

```
##  [1] "Cajun/Creole"  "Southern"      "Restaurants"   "Bars"
##  [5] "Sports Bars"   "Dive Bars"     "Burgers"       "Nightlife"
##  [9] "Sandwiches"    "Chicken Wings"
```

Next, we create a frequency dictionary.

```
hash = new.env(hash=TRUE, parent=emptyenv(), size=length(cat_list))
for (cats in categories) {
  for (cat in cats) {
    cat = trimws(cat)
    hash[[cat]] = ifelse(exists(cat, hash),
                         hash[[cat]]+1, 1)
  }
}

# Convert hash to dataframe.
cat_names = names(hash)
cat_freq = data.frame(cat_names, unlist(as.list(hash)), row.names=c())
colnames(cat_freq) = c("categories", "freq")

# Sort in descending order of frequency.
cat_freq = cat_freq[order(-cat_freq$freq),]
cat_freq[1:10,]
```

```
##                 categories freq
## 213            Restaurants 8940
## 735                   Food 2609
## 314             Nightlife 2503
## 94                    Bars 2348
## 649 American (Traditional) 1507
## 556         American (New) 1422
## 460     Breakfast & Brunch 1235
## 158             Sandwiches  963
## 95                 Mexican  879
## 281                 Italian  863
```

```
## There are 799 unique categories.
```

Ouch! First, let's save.

```
write.csv(cat_freq, 'cat_freq.csv', row.names=FALSE)
```

Now we'll split the categories into separate columns. Note that this code in R took a couple hours to run, so I'll be rewriting it in Python.

```
import pandas as pd
import numpy as np
# Determine headers.
categories = pd.read_csv('cat_freq.csv').loc[:, "categories"]
categories = list(categories)
# Determine rows and create new dataframe.
business = pd.read_csv('business_preclean1.csv')
nrow = len(business.index)
df = pd.DataFrame(0, index=np.arange(nrow), columns=categories, dtype='int64')
# Iterate businesses and count their categories.
for index, row in business.iterrows():
    if not pd.isnull(row['categories']):
        cat_list = row['categories'].split(', ')
        for cat in cat_list:
            df.loc[index, cat] += 1
    if index % 5000 == 0:
        print("Processing row {0}...".format(index))
# Change column names so they are more easily identifiable
cat_dict = {}
for cat in categories:
    new_cat = cat.replace(' ', '')
    new_cat = new_cat.replace('&', '')
    new_cat = new_cat.replace('/', '')
    new_cat = new_cat.replace('(', '')
    new_cat = new_cat.replace(')', '')
    new_cat = new_cat.replace("'", '')
    new_cat = new_cat.replace('-', '')
    new_cat = "cat_{0}".format(new_cat)
    cat_dict[cat] = new_cat
df = df.rename(columns=cat_dict)
# Save the result.
df.to_csv('catframe.csv', index=False)
print("Completed!")
```

Reduction

We could manually go through all the categories to mine for similarities and combine them, but this might still be too laborous, and I'm lazy. We can see something about the distribution of these categories. For example, let's see how many categories are claimed by over 50 businesses and 10 businesses, respectively.

```
## Categories claimed by more than 50 businesses:
```

```
##
## FALSE  TRUE
##   670   129
```

```
##
## Categories claimed by more than 10 businesses:
```

```
##
## FALSE  TRUE
##   472   327
```

But is it helpful to just chop off insignificant categories? It is to an extent, but we'd like to preserve as much information as possible. This is where **principle component analysis (PCA)** comes in. Without getting into too much math, PCA rotates the axes of the dimensions to reduce as much variance as possible. It then gives the rotated axes in terms of most variance reduced. Columns that don't have much change in variances means that they contain little data to begin with. The disadvantage here is that we lose all interpretability, which I'm willing to sacrifice, but we don't actually care which category contributes to the result, we just want to find the maximally effective columns while minimizing the data as much as possible.

PCA

First, we will train a PCA regression model. We won't be using it to predict anything, although we can. We'll just be using the rotated axes it computes as feature reduction.

```
library(stats)
cats = read.csv('catframe.csv')
pca.model = prcomp(cats, scale=TRUE)
pca.variance = summary(pca.model)$importance[3,]

# Show how much cumulative variance first 5 columns take up.
pca.variance[1:5]
```

```
##     PC1     PC2     PC3     PC4     PC5
## 0.00778 0.01485 0.02133 0.02751 0.03350
```

We might want to save these PCA rotations in case we want to use more or less of them.

```
write.csv(pca.model$rotation, 'cat_pca.csv', row.names=FALSE)
```

Now it's time to reduce the features. Recall that we have 799 categories. The general guideline is to keep 70% to 80% of the variance. I'm going to make a judgment call and err to the side of file size reduction because I have a slow computer. So I'll keep 70% of the variance.

```
num_vectors = table(pca.variance < 0.7)["TRUE"]
pca.vector = pca.model$rotation[, 1:num_vectors]
pca.final = as.data.frame(as.matrix(cats) %*% pca.vector)

# Save space
pca.final = round(pca.final, 5)
colnames(pca.final) = sapply(colnames(pca.final), function(name) {
  paste("cat", name, sep='_')
})

# Replace categories column with PCA vectors
business$categories = NULL
business = cbind(business, pca.final)
```

```
write.csv(business, 'business_preclean2.csv', row.names=FALSE)
```

Next, we'll deal with attribute objects.

Cleaning Business: Attribute Objects (Part 3)

Jonathan Chang

November 1, 2018

Objects

Objects in Python, Javascript, etc., might as well be a list of key-value pairs, where the values can be any variable, or even functions (since functions are first class citizens). Objects are often passed as JSON (Javascript Object Notation), which are string-string key-value pairs of a specific format. Since the Yelp data set was at some point obtained from its REST API, it chose to use objects to attributes with several related properties.

The problem, again, is that machine learning algorithms will recognize these as mere text or factors – equally useless, when they're expecting numbers. Treating combinations of multiple properties as text also fails to preserve the relations that they hold. So, again, we'll split these into separate columns.

To make our lives easier, we'll leverage existing libraries that split JSON into separate variables. We'll make a few changes to make these attribute objects compatible with JSON format.

```
library(jsonlite)
convert_json = function(attr) {
  txt = gsub("'", "\"", attr)
  txt = gsub("False", "false", txt)
  gsub("True", "true", txt)
}
```

Splitting

```
business = read.csv('business_preclean2.csv', stringsAsFactors=FALSE)
```

As before, we'll split these into separate columns. Let's see an example before we start.

```
## [1] "{'dj': True, 'background_music': False, 'no_music': False, 'karaoke': False,
'live': False, 'video': False, 'jukebox': False}"
```

```
# Inputs
# data - dataframe
# attr - attribute name without the "attributes_" prefix
expand_attr = function(data, attr) {

  # To keep new columns organized, we'll prepend the original
  # attribute name to each new column.
  prefix = attr

  # Add "attributes_" back in the column title, so we don't
  # have to type this over and over.
  attr = paste("attributes", attr, sep='_')

  # Convert column to JSON format.
  txt = convert_json(data[,attr])
  df = data.frame()

  for (row in 1:length(txt)) {

    # Create a list using number of columns of new dataframe.
    cols = ncol(df)
    df[row,] = rep("None", each=cols)

    # Attribute is not empty.
    if (nchar(txt[row]) > 0) {

      # Extract properties from JSON attribute object.
      js = fromJSON(txt[row])
      colns = colnames(df)

      # Assign value of each property
      # (add column if we haven't seen it before).
      for (category in 1:length(js)) {
        cat_value = js[category]
        cat_name = trimws(names(cat_value))
        if (!(cat_name %in% colns)) {
          df[cat_name] = "None"
        }
        df[row, cat_name] = ifelse(cat_value, 1, 0)
      }
    }
  }

  # Add original attribute name as prefix to all new columns.
  colns = colnames(df)
  colns = gsub('-', '', colns)
  colns = as.vector(sapply(colns, function(x) paste(prefix, x, sep='_')))
  colnames(df) = colns

  # Replace original attribute column with new columns.
```

```
  data = cbind(df, data)
  data[,attr] = NULL
  data
}

# Apply to every attribute that uses objects.
business = expand_attr(business, "Music")
business = expand_attr(business, "HairSpecializesIn")
business = expand_attr(business, "GoodForMeal")
business = expand_attr(business, "DietaryRestrictions")
business = expand_attr(business, "BusinessParking")
business = expand_attr(business, "BestNights")
business = expand_attr(business, "Ambience")
```

We now have 415 columns. But that's the cost of doing business. The next best alternative is to delete these attributes altogether rather than keeping them in a useless format. Let's see the next columns before we save.

```
business[55, grepl("Music_", names(business))]
```

```
##    Music_dj Music_background_music Music_no_music Music_karaoke Music_live
## 55        1                      0              0             0          0
##    Music_video Music_jukebox
## 55           0             0
```

Finally, let's change all blank values into "None" to keep it consistent.

```
business[business==""] = "None"
business[is.na(business)] = "None"
```

```
write.csv(business, 'business_preclean3.csv', row.names=FALSE)
```

Cleaning Business: Postal Codes and Neighborhoods (Part 4)

Jonathan Chang

November 1, 2018

Geographical Redundancy

Addresses, lattitudes and longitudes, cities, postal codes, neighborhoods, and states – they all describe the same thing in various granularities. Where the granularity is too fine, like address, it becomes useless to us, since every address is a text string, and have no obvious relation to each other. The algorithm will treat them as factors with no relation, when there is clearly a geographical relation. That's why we removed addresses earlier. Lat/long offers the same information, but since they're encoded as floats, they preserve the geographical relation. However, they don't give a cultural relation, such as demographics of an area. For that, city, postal code, or neighborhood might be the best bet. But what exactly is a neighborhood, is it finer than a postal code or coarser?

If we keep all the fields, the redundancy might skew our results since the data is somewhat dependent. Linear regression requires features be independent; otherwise we couldn't take an inverse because the matrix would not be full rank. Even if we use pseudo-inverse, it damages the model. Plus, it adds to the file without providing enough additional information.

Therefore, we will investigate how to best navigate this topic.

Which hood are you from?

Let's gather some basic information about neighborhoods so we can clarify which questions to ask further.

```
business = read.csv('business_preclean3.csv', stringsAsFactors=FALSE)
num_cities = length(unique(business$city))
num_hoods = length(unique(business$neighborhood))
num_zips = length(unique(business$postal_code))
```

```
## There are 208 cities, 213 neighborhoods, and 1574 zip codes
```

I know I don't want to add 1574 one-hot columns to this data set, so we need to find a way to reduce postal codes. Cities and neighborhoods seem to be… in the same neighborhood, so we should differentiate them. Let's group the data by city >> neighborhood >> businesses, and find out how big each neighborhood is, and how many neighborhoods are in a city.

```
cityhood = business %>% group_by(city, neighborhood) %>% summarise(n=n())
cityhood %>% filter(neighborhood != "") %>% arrange(desc(n)) %>% head(10)
```

```
## # A tibble: 10 x 3
## # Groups:   city [6]
##    city       neighborhood      n
##    <chr>      <chr>         <int>
##  1 Phoenix    None           1435
##  2 Scottsdale None            799
##  3 Las Vegas  The Strip       759
##  4 Las Vegas  None            470
##  5 Tempe      None            440
##  6 Las Vegas  Spring Valley   385
##  7 Las Vegas  Westside        360
##  8 Las Vegas  Southeast       350
##  9 Chandler   None            347
## 10 Mesa       None            322
```

There is quite a range. This result seems interesting though, so let's save it before we move on.

```
write.csv(cityhood, 'cityhood.csv', row.names=FALSE)
```

We found out that businesses in neighborhoods vary widely, but it's kind of hard to tell from the table how much variance of neighborhoods there are in cities. So let's tabulate neighborhoods per city.

```
numhoods = cityhood %>% filter(neighborhood != "") %>%
  group_by(city) %>% summarise(n=n()) %>% arrange(desc(n))
cities_with_hoods = nrow(numhoods)
cities_multi_hoods = nrow(numhoods %>% filter(n > 1))
cities_five_hoods = nrow(numhoods %>% filter(n >= 5))
```

```
## Out of 208 cities with given neighborhoods, there are 22 cities with more than 1 n
eighborhoods, and 9 cities with 5 or more neighborhoods.
```

This tells us that 191 out of 213 occur in 22 cities. That's quite a concentration.

Going Postal

Now that we have an idea of the city-hood relation, let's bring postal codes into the picture. Since we just want to concentrate on areas labeled with a neighborhood, let's filter out all businesses without a hood.

```
bhood = business[!(business$neighborhood==""),]
num_zips_with_hoods = length(unique(bhood$postal_code))
```

Out of 1574 postal codes, there are 1574 with hoods. But wait a sec! If most postal codes have hoods, and most hoods coincide with 22 cities, then that means most postal codes are from those same cities as well.

```
zips_per_hood = bhood %>% group_by(neighborhood, postal_code) %>%
  summarise(n=n())
n_zips_per_hood = nrow(zips_per_hood)
overlap = n_zips_per_hood - num_zips_with_hoods
```

```
## There are 1800 postal code-neighborhood combinations. This happens to be 226 highe
r than the number of unique postal codes with hoods. Therefore, there must be 226 hoo
ds with duplicate postal codes in more than one hood.
```

Taking Off the Hood

Let's look at businesses without given neighborhoods.

```
bnohood = business[business$neighborhood=="",]
num_zips_no_hood = length(unique(bnohood$postal_code))
over_zip = num_zips_with_hoods + num_zips_no_hood - num_zips
```

```
## There are 0 postal codes without a hood. Since postal codes without and without ho
ods exceed the total number of postal codes ( 1574 + 0 > 1574 ), there must be 0 post
al codes with both hood and no hood.
```

Previously, we found that most postal codes are associated to neighborhoods concentrated in a few cities. We want to find if postal codes are concentrated in a few cities without neighborhoods, or if this is just some phenomenon specific to cities with hoods.

```
citycode = bnohood %>% group_by(city, postal_code) %>% summarise(n=n())
numzips = citycode %>% group_by(city) %>% summarise(n=n())
cities_gt1_zip = table(numzips$n > 1)['TRUE']
cities_gt2_zip = table(numzips$n > 2)['TRUE']
cities_gt5_zip = table(numzips$n > 5)['TRUE']
cities_gt8_zip = table(numzips$n > 8)['TRUE']
cities_gt10_zip = table(numzips$n > 10)['TRUE']

most_business_per_zip = citycode %>% arrange(desc(n)) %>% head(10)
most_zips_per_city = numzips %>% arrange(desc(n)) %>% head(10)
```

Cities with multiple postal codes.

Postal codes per city	**# cities**
more than 1	NA
more than 2	NA
more than 5	NA
more than 8	NA
more than 10	NA

The following are perhaps more illuminating. The first shows that the largest postal codes are in the desert in Southwestern US, with most businesses per postal code.

```
## # A tibble: 0 x 3
## # Groups:   city [0]
## # ... with 3 variables: city <chr>, postal_code <chr>, n <int>
```

The next table shows that the smallest postal codes are in Canada (Toronto, Calgary, and Mississauga), with most postal codes per city.

```
## # A tibble: 0 x 2
## # ... with 2 variables: city <chr>, n <int>
```

Phoenix and Las Vegas also tops in number of postal codes per city.

Blame Canada

We can see where the Canadians sit on businesses per postal code.

```
citycode[citycode$city == 'Toronto',] %>% head(5)
```

```
## # A tibble: 0 x 3
## # Groups:   city [0]
## # ... with 3 variables: city <chr>, postal_code <chr>, n <int>
```

```
citycode[citycode$city == 'Calgary',] %>% head(5)
```

```
## # A tibble: 0 x 3
## # Groups:   city [0]
## # ... with 3 variables: city <chr>, postal_code <chr>, n <int>
```

```
citycode[citycode$city == 'Mississauga',] %>% head(5)
```

```
## # A tibble: 0 x 3
## # Groups:   city [0]
## # ... with 3 variables: city <chr>, postal_code <chr>, n <int>
```

In each of the Canadian cities, there are only 1-2 businesses per postal code. This is too fine granularity, and since postal codes are factors with no apparent relation between codes, it's not useful to have such little businesses per postal code.

A little bit of an aside, but during the course of this exercise, I found Aurora to be both a US and CA city name, so that has to be handled.

This quora post suggests Canadian postal codes are more precise: https://www.quora.com/What-is-the-zip-code-for-Toronto-Canada (https://www.quora.com/What-is-the-zip-code-for-Toronto-Canada)

Specifically, one post said they can cover as little as a block each. This is definitely too fine for our use.

We can develop a little more intuition.

```
## There are 12 states, 3 are in Canada.
```

```
ca_provinces = c("AB", "QC", "ON")
ca_business = business[business$state==ca_provinces,]
zips_in_ca = length(unique(ca_business$postal_code))
hoods_in_ca = length(unique(ca_business$neighborhood))
```

```
## There are only 547 businesses in Canada, out of 12058 total. They correspond to 47
1 postal codes and 71 hoods.
```

Note that this means that most of the Canadian postal codes only host 1 business, while Canadian neighborhoods host on average less than 8 businesses, if they were evenly distributed. But we've shown above that they're not.

```
cahoods = ca_business %>%
  group_by(state, neighborhood) %>%
  summarise(n=n())
cahoods_gteq5 = table(cahoods$n >= 5)['TRUE']
cahoods_gteq10 = table(cahoods$n >= 10)['TRUE']

cazips = ca_business %>% group_by(postal_code) %>%
  summarise(n=n()) %>% arrange(desc(n)) %>% head(5)
cazips
```

```
## # A tibble: 5 x 2
##   postal_code     n
##   <chr>       <int>
## 1 M5T 1L1         6
## 2 H2Y 2A3         4
## 3 L4B 3B4         4
## 4 M2N 5S1         4
## 5 M5A 3C4         4
```

```
## There are only 28 Canadian neighborhoods with 5 or more businesses, and 16 with 10
or more businesses, out of 72 total. The maximum number of businesses per Canadian po
stal code is 6 .
```

Let's keep a copy of this result out of interest.

```
write.csv(cahoods, 'cahoods.csv', row.names=FALSE)
```

Finally, we should have a baseline for how dense everything is. The following shows the number of businesses per state.

```
business %>% group_by(state) %>% summarise(n=n()) %>% arrange(n) %>%
  head(5)
```

```
## # A tibble: 5 x 2
##   state     n
##   <chr> <int>
## 1 OR        1
## 2 SC       20
## 3 IL       57
## 4 AB       71
## 5 QC      232
```

We are also interested in how many businesses there are in a city, but there are way too many to list. So we use the mean to get a ballpark.

```
b_city = business %>% group_by(city) %>% summarise(n=n(), state=first(state)) %>%
  arrange(n)
mean_business_per_city = mean(b_city$n)
```

```
## Note that there is an average of 57.9711538461538 businesses per city.
```

This can be our starting point for granularity control. Note that the mean for cities is similar to the businesses we have for the entire state of Illinois (IL).

```
small_cities = b_city[b_city$n <= 4,]
```

```
## Note that there are still 133 cities with less than 5 businesses, which is huge co
nsidering there are only 208 total cities.
```

Policies

We've come to a number of conclusions on how to proceed with the original problem of reducing geographical fields.

1. Attach state to all city names to solve duplicate names problem.
2. Cities with less than *k* businesses use only the state.
3. If a business is labeled with a neighborhood, use the hood if businesses per that hood >= *k*.
4. If no neighborhood is labeled, use postal code if businesses per that postal code >= *k*.
5. Delete state, neighborhood, and postal code columns.

k is a potential parameter that can be tuned.

Selection

```
# Inputs:
# data - dataframe
# feature - name of feature column
# k - postal code and neighbor hood frequency threshold
factors_gt_k = function(data, feature, k) {

  # Create a list of values of the feature that we will keep.
  factors = data %>% group_by(data[,feature]) %>% summarise(n=n())
  names(factors) = c(feature, "n")
  factors = factors[factors[,feature]!="None" &
                      factors[,"n"] >= k, feature]

  # Keep only the values.
  factor(factors %>% pull(feature))
}

# Inputs:
# data - dataframe
# k - postal code and neighborhood frequency threshold
reduce_business_geo = function(data, k) {

  # Combine city and state into same column.
  data$city = paste(data$city, data$state, sep=", ")

  # Use only state for cities under k threshold.
  cities = factors_gt_k(data, "city", k)
  city_mask = !(data$city %in% cities)
  data$city[city_mask] = data$state[city_mask]

  # Keep hoods with k threshold.
  hoods = factors_gt_k(data, "neighborhood", k)
  hood_mask = data$neighborhood %in% hoods
  data$city[hood_mask] = data$neighborhood[hood_mask]

  # Keep postal codes with k threshold.
  zips = factors_gt_k(data[data$neighborhood=="None",],
                      "postal_code", k)
  zip_mask = data$postal_code %in% zips
  data$city[zip_mask] = data$postal_code[zip_mask]

  # Scale lat, long.
  data$latitude = round(scale(data$latitude), 4)
  data$longitude = round(scale(data$longitude), 4)

  # Remove columns we don't need anymore.
  data$state = NULL
  data$neighborhood = NULL
  data$postal_code = NULL
  data$city = as.factor(data$city)
  data
```

```
}
```

I can’t think of a good metric to pick this parameter so this will be sort of an art for now.

```
k = 10
```

```
business = reduce_business_geo(business, k)
num_locations = length(unique(business$city))
```

```
## There are now 250 locations, compared to 208 cities we originally had.
```

We don’t want duplicate column names between any of our data, so let’s do one last bit of clean up, then save.

```
business$business_reviews = business$review_count
business$review_count = NULL
business$business_stars = business$stars
business$stars = NULL
```

```
write.csv(business, 'business_preclean4.csv', row.names=FALSE)
```

We’re almost finished with cleaning. The only thing left to do is to one-hot the attributes with dummy variables to aid regression.

That Was Random!

Jonathan Chang

November 1, 2018

Preprocessing

Random forest (and trees in general) is the one model where PCA would throw it off and make it take forever, since it must calculate probabilities for every value – and they all look different! It's also the one model that should be able to deal with numerical factors since all it has to do is make splits down each feature. Therefore, just this one time, we'll replace the PCA columns with the 799 categories, and keep all the other business attributes as factors. After this, we'll be using the processed version and one-hot encoding for most of the other models, like linear regression and neural networks, etc.

```
users = read.csv('users_clean.csv')
users$friends = NULL
business = read.csv('business_preclean4.csv')
reviews = read.csv('reviews_clean.csv')[, c("uid", "bid", "stars")]
```

We'll replace the PCA columns with the category frequencies.

```
pca_cols = grepl("cat_", colnames(business))
business[, pca_cols] = NULL
catframe = read.csv('catframe.csv')
business = cbind(business, catframe)
rm(catframe)
```

You know what, let's save this version of the business set in case we need to use it later as well.

```
write.csv(business, 'business_cats.csv', row.names=FALSE)
```

Now we will join the columns into a training set. After, we get rid of the ID columns since we do not want to include them in the training process.

```
train = inner_join(reviews, users, by="uid")
train = inner_join(train, business, by="bid")
train$uid = NULL
train$bid = NULL
rm(reviews)
```

Similarly, we will join columns for the validation set.

```
validate = read.csv('validate_simplified.csv')
val = inner_join(validate, users, by="uid")
val = inner_join(val, business, by="bid")
val$uid = NULL
val$bid = NULL
val_x = val[, -which(names(val) %in% c("stars"))]
rm(validate)
```

Beware of slow packages

WARNING: Don't run the code chunk below. The *caret* package provides a unified interface for dozens of models. It's also an easy way to run repeated cross-validation or bootstrapping across a variety of tunable parameters, and will evaluate the best model by some metric after a grid or random search.

```
library(caret)
library(ranger)
library(e1071)
fitControl = trainControl(classProbs = TRUE)
rf = train(stars ~ ., data=train, method='ranger',
           trControl=fitControl,
           importance='impurity')
rf_pred = predict(rf, newdata=val)
```

The code above ran for 3 days with no sign of ending. Although randomforest will prune features by default, With the size of the training set, each run took half a day to a day. The default values for caret was to run boostrapping 10 times with 10 repeats, and select parameters *mtry*, *min.node.size*, and *splitRule* 3 times each, with a total of 900 runs. This will give us a better model, but time restriction and processing power of my computer is limited.

Seeing the trees for the forest

Note: After training all the random forest models, I realized I had *NA*'s in my business data set that made R import them as factors instead of numbers. Since it's too involved to re-run the entire process, I am just going to train the final model on the fixed dataset. Therefore, the RMSE values hereforth presented should be slightly off, although it shouldn't be major.

Hyperparameter	Explanation
mtry	Number of features randomly selected to train each tree.

Hyperparameter	Explanation
min.node.size	Minimum samples the leaves of the trees represent. This indirectly affects the height of the trees.
importance	Criteria of feature selection.
splitrule	Criteria of making splits.

For that reason, we will run the underlying function below with a more basic grid search of recommended values, given by this paper, *Hyperparameters and Tuning Strategies for Random Forest*, by Probst, Philipp, et. al.: https://arxiv.org/pdf/1804.03515.pdf (https://arxiv.org/pdf/1804.03515.pdf).

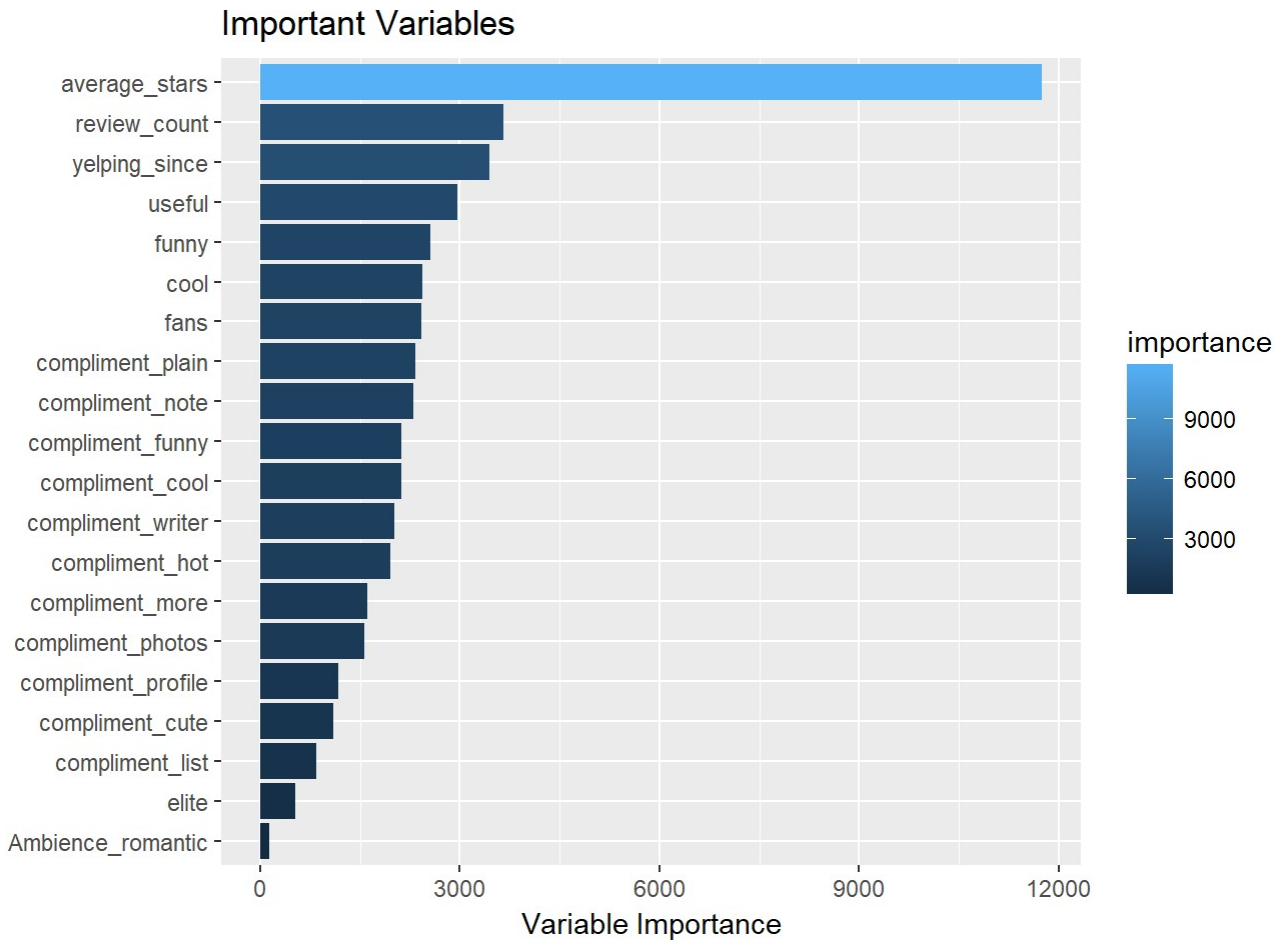
But first let's see how long a single run takes. :)

```
library(ranger)
p = ncol(train)
rf = ranger(stars ~ ., train,
            write.forest = TRUE,
            classification = FALSE,
            mtry = round(sqrt(p)),
            min.node.size = 5,
            importance = "impurity",
            splitrule = "variance",
            verbose = TRUE)
rf_pred = predict(rf, val_x)
error = rf_pred$predictions - val$stars
rmse = sqrt(mean(error^2))
saveRDS(rf, 'models/ranger_trial.rds')
```

```
## RMSE is 1.14752340004684.
```

20 minutes is better than half a day (it turns out *mtry* proportionately affects training time). Unfortunately, prediction isn't any better than before. We can at least visualize what it thinks are important features.

```
importance = data.frame(names(rf$variable.importance),
                        rf$variable.importance)
colnames(importance) = c("feature", "importance")
library(ggplot2)
ggplot(importance[1:20,], aes(x=reorder(feature, importance),
      y=importance, fill=importance)) +
      geom_bar(stat="identity", position="dodge") + coord_flip() +
      ylab("Variable Importance") +
      xlab("") +
      ggtitle("Important Variables")
```



The paper recommends $\mathrm{mtry} = \frac{p}{3}$, where *p* is the number of features, for regression trees. But a quick test showed that setting that value that large will take 12 hours to run per model. So while we might run that once at the end to test for improvement, we will not be using that value for grid search. So… let's get to it!

```
mtry = c(round(sqrt(p)), 2 * round(sqrt(p)))
min.node.size = c(5, 10)
importance = c("impurity", "permutation")
splitrule = c("variance", "extratrees")
params = expand.grid(mtry,
                     min.node.size,
                     importance,
                     splitrule,
                     stringsAsFactors = FALSE)
colnames(params) = c("mtry",
                     "min.node.size",
                     "importance",
                     "splitrule")
for (i in 1:nrow(params)) {
  paste("Trial", i, "...\n", sep=' ') %>% cat()
  rf = ranger(stars ~ ., train,
              write.forest = TRUE,
              classification = FALSE,
              mtry = params[i, "mtry"],
              min.node.size = params[i, "min.node.size"],
              importance = params[i, "importance"],
              splitrule = params[i, "splitrule"],
              respect.unordered.factors = TRUE)
  rf_pred = predict(rf, val_x)
  error = rf_pred$predictions - val$stars
  rmse = sqrt(mean(error^2))
  params[i, "rmse"] = rmse
}
best_params = params[which.min(params$rmse),]
```

Let's save the results.

```
write.csv(params, 'rf_grid_params.csv', row.names=FALSE)
```

Now for the moment of faith!

```
## **BEST PARAMS:**
##   mtry: 60
##   min.node.size: 10
##   importance: impurity
##   splitrule: variance
## RMSE: 1.08960829872114
```

We notice signs of overfitting in that a larger *min.node.size* seems to consistently lead to a better score. Let's try pushing that angle to see if it improves further. But first, let's factor out some redundant code.

```
train_rf = function(param_list, savefile) {
  row_n = nrow(params) + 1
  params[row_n, ] = best_params
  param_names = names(param_list)
  for (i in 1:length(param_list)) {
    params[row_n, param_names[i]] = param_list[i]
  }
  rf = ranger(stars ~ ., train,
              write.forest = TRUE,
              classification = FALSE,
              mtry = params[row_n, "mtry"],
              min.node.size = params[row_n, "min.node.size"],
              importance = params[row_n, "importance"],
              splitrule = params[row_n, "splitrule"],
              verbose = TRUE)
  rf_pred = predict(rf, val_x)
  error = rf_pred$predictions - val$stars
  rmse = sqrt(mean(error^2))
  params[row_n, "rmse"] = rmse
  filepath = paste('models', savefile, sep='/')
  saveRDS(rf, filepath)
  best_params = params[which.min(params$rmse),]
  write.csv(params, 'rf_grid_params.csv', row.names=FALSE)
}
```

```
train_rf(list(min.node.size=15), 'ranger_min_nodes_15.rds')
```

```
## RMSE is 1.08872784151878.
```

That was a very minor improvement. Since we know that *min.nodes.size* increases bias, we should lean towards it when we increase *mtry*, which increases variance, as a balancing mechanism. So for completeness, we'll run the best params with *mtry* recommended for regression trees to see if there's any improvement (crossing my fingers since this will take a long time).

```
p = ncol(train)
train_rf(list(min.node.size=10, mtry=round(p/3)),
         'ranger_mtry_306.rds')
```

```
## RMSE is 1.05769327788674.
```

Our best params are now **drumroll**

```
## **BEST PARAMS:**
##   mtry: 306
##   min.node.size: 15
##   importance: impurity
##   splitrule: variance
## RMSE: 1.05769327788674
```

That's what we want to see! But let me just increase *mtry* incrementally as a sanity check.

```
train_rf(list(mtry=100), 'ranger_mtry_100.rds')
```

```
## RMSE is 1.07144806776432.
```

So there is a nice logarithmic improvement from 30, 60, 100, to 306. There's a chance that an even higher *mtry* could improve RMSE more, or it might overfit. We'll try to squeeze one more drop.

```
p = ncol(train)
train_rf(list(mtry=round(2*p/3)), 'ranger_mtry_612.rds')
```

```
## **BEST PARAMS:**
##   mtry: 306
##   min.node.size: 15
##   importance: impurity
##   splitrule: variance
## RMSE: 1.05769327788674
```

So it looks like the latest run probably overfit, as it got slightly, insignificantly worse (since we've previous testing shown a bit of variation between each run). However, there's no need to waste a bunch of time for random improvement. Our random forest model is 90% there, and we've reached the point of diminishing returns.

It's the final countdown

To train our final model, we should include the validation set so we have more data to make our predictions more generalizable. We should also remove unused objects in our environment to free up memory, and hopefully have a little more swap space to make it run a little faster.

```
train = rbind(train, val)
test = read.csv('test_simplified.csv')
test = inner_join(test, users, by='uid')
test = inner_join(test, business, by='bid')
test$uid = NULL
test$bid = NULL
rm(users)
rm(business)
rm(val)
rm(val_x)
```

Choo-choo!

```
rf = ranger(stars ~ ., train,
            write.forest = TRUE,
            classification = FALSE,
            mtry = best_params$mtry,
            min.node.size = best_params$min.node.size,
            importance = best_params$importance,
            splitrule = best_params$splitrule,
            verbose = TRUE)
saveRDS(rf, 'models/ranger_final_fixed.rds')
rf_pred = predict(rf, test)
submit = data.frame(seq.int(0, nrow(test)-1), rf_pred$predictions)
colnames(submit) = c("index", "stars")
write.csv(submit, 'submit_ranger_fixed.csv',
          row.names=FALSE, quote=FALSE)
```

The score with business hours as factors is **1.05812**. With the bug fixed and business hours converted to numbers, we got a whopping **1.05800**. What a waste of time. Let's move on to a more powerful ensemble model, XGBoost, before we try our favorite linear regression. We should also design a test suite for error analysis to figure out what exactly our model did well on and what it did poorly on, so we know how to stack the model later.

Error Analysis

Jonathan Chang

November 9, 2018

Motivation

We just trained our first real model, and did pretty good. But two models with the same RMSE score are not the same. Some models might perform better for certain subsets. It is useful to analyze exactly where the models succeeded and failed for several reasons:

- To learn more about the data.
- To see what we can focus on to improve predictions.
- To have an idea the decision boundaries for stacking models for an ensemble.

The last point is a bit ambitious, but not out of hand.

We should develop a suite of functions that could easily be used by future models to contrast prediction ability.

Helpers

Since R Markdown root path normalizations don’t apply to the console, these functions might help for debugging purposes.

```
console.read_csv = function(fname, stringsAsFactors=TRUE) {
  wd = getwd()
  setwd(working_dir)
  csv = read.csv(fname, stringsAsFactors=stringsAsFactors)
  setwd(wd)
  csv
}

console.write_csv = function(obj, fname, row.names=FALSE) {
  wd = getwd()
  setwd(working_dir)
  write.csv(obj, fname, row.names=row.names)
  set(wd)
}

console.read_RDS = function(fname) {
  wd = getwd()
  setwd(working_dir)
  model = readRDS(fname)
  set(wd)
  model
}

working_dir = getwd()
files_list = list.files()
console.getwd = function() { working_dir }
console.setwd = function(wd) { setwd(wd) }
console.list_files = function() { files_list }
```

Formalizations of the objective.

```
sq_error = function(y_pred, y) {
  (y_pred - y)^2
}

rmse = function(y_pred, y) {
  sqrt(mean(sq_error(y_pred, y)))
}
```

Strengths and weaknesses

Although this turned out to be a regression problem due to the objective metric, in the real application, Yelp, users can only rate discrete number of stars between 1-5 inclusive. It might be useful to see the actual stars the model succeeds or fails at predicting. We'll say that prediction is $\hat{y}_{\text{pred},i} = y_i$ if $|y_{\text{pred},i} - y_i| \leq 0.5$.

```
rating_confusionMatrix = function(y_pred, y, rating) {
  y_pred = round(as.numeric(y_pred))
  tab = table(y_pred != rating, y != rating)
  conf = list()
  conf$TP = tab[1, 1]
  conf$FP = tab[1, 2]
  conf$FN = tab[2, 1]
  conf$TN = tab[2, 2]
  conf
}

confusion = function(y_pred, y) {
  conf = data.frame()
  for (i in 1:5) {
    cm = rating_confusionMatrix(y_pred, y, i)
    conf[i, "TP"] = cm$TP
    conf[i, "FP"] = cm$FP
    conf[i, "FN"] = cm$FN
    conf[i, "TN"] = cm$TN
  }
  conf
}

rating_accuracy = function(conf) {
  (conf$TP + conf$TN) / rowSums(conf)
}

rating_precision = function(conf) {
  conf$TP / (conf$TP + conf$FP)
}

rating_recall = function(conf) {
  conf$TP / (conf$TP + conf$FN)
}

rating_fscore = function(conf) {
  pre = rating_precision(conf)
  rec = rating_recall(conf)
  (2 * pre * rec) / (pre + rec)
}

rating_score = function(y_pred, y) {
  conf = confusion(y_pred, y)
  accuracy = rating_accuracy(conf)
  precision = rating_precision(conf)
  recall = rating_recall(conf)
  fscore = rating_fscore(conf)
  data.frame(accuracy, precision, recall, fscore)
}
```

Okay, but what if we want to compare two models? Let's do that.

```
score_dist = function(score1, score2) {
  score1-score2
}
```

That was underwhelming.

Interpretation

- **Accuracy** is the rate of correct predictions.
- **Precision** is the rate of true predictions that are actually true.
- **Recall** is the rate of actually true samples predicted true.
- **F-Score** reveals the best balance of precision and recall.

Note that low precision corresponds to high *Type I* error, or false alarm. We'd be saying something is true when it isn't. Low recall corresponds to high *Type II* error, or false negative. We'd be saying something is false when it isn't. Therefore, low precision means we're overestimating that class, and low recall means we're underestimating that class. If both are low, then we have low accuracy in terms of true predictions.

Example

Random Forest

We're going to use the functions we developed above to analyze the error of random forest predictions on the validation set (so, not our final model, but the one without validation samples). Then, we'll compare this model to the user mean baseline.

```
rf = readRDS('models/ranger_mtry_306.rds')
val = read.csv('validate_simplified.csv')
val = inner_join(val, read.csv('users_clean.csv'), by='uid')
val = inner_join(val, read.csv('business_cats.csv'), by='bid')
val$uid = NULL
val$bid = NULL
val$friends = NULL
y = val$stars
val = val[, -which(names(val) %in% c("stars"))]
rf_y_pred = predict(rf, val)$predictions
rf_score = rating_score(rf_y_pred, y)
rf_score
```

```
##    accuracy precision     recall    fscore
## 1 0.9232981 0.9356725 0.07731336 0.1428253
## 2 0.9011922 0.2045210 0.09870130 0.1331465
## 3 0.7481878 0.2377340 0.34643907 0.2819724
## 4 0.5287258 0.3437593 0.73611995 0.4686599
## 5 0.6801126 0.8328462 0.28766465 0.4276271
```

We notice first the high precision on 1 and 5. These mean most of the samples we predicted to be 1 and 5 are actually 1 and 5, respectively. However, low recall means we are missing a lot of 1 and 5 star ratings. This is likely due to that half as many numbers $x \in \mathcal{R}$ are rounded to these classes. Low recall for 2 stars means we are also missing a lot of 2 star ratings. Most likely these go to 3 stars, we can tell by the low 3-star precision showing overestimation. Likewise, many 3 stars are predicted 2 or 4 stars. With this model, all the classes are being underestimated except for 4-star ratings, which happen to be most popular. The F1 measure of 4 and 5 stars similar, so the precision in one is traded for recall of the other. In other words, a lot of the 4 stars likely belong to the 5-star class.

We can see that just looking at the accuracy, which is highest in 1 and 2 stars, is misleading. Most of that accuracy goes to false negatives. Because 1 and 2 stars make up much smaller proportion of the overall set (i.e. the classes are skewed), this favors accuracy.

All in all, random forest seems to greatly underestimate the edge numbers, and make a lot of mistakes in the middle ones.

User Mean

Let's analyze the user mean (the first submission).

```
val = read.csv('validate_simplified.csv')
unnormed_stars = read.csv('users.csv')$average_stars
users = read.csv('users_clean.csv')[,c("uid","average_stars")]
users$average_stars = unnormed_stars
val = inner_join(val, users, by='uid')
mean_score = rating_score(val$average_stars, y)
mean_score
```

```
##    accuracy precision     recall    fscore
## 1 0.9245762 0.9458128 0.09277603 0.1689769
## 2 0.9104379 0.2307040 0.07064935 0.1081726
## 3 0.7392615 0.1969852 0.26878411 0.2273507
## 4 0.4911237 0.3320303 0.79298395 0.4680736
## 5 0.6569283 0.8642397 0.20656668 0.3334368
```

There's a very similar pattern here, so let's compare the two.

```
score_dist(rf_score, mean_score)
```

```
##       accuracy   precision      recall        fscore
## 1 -0.001278032 -0.01014029 -0.01546267 -0.0261516355
## 2 -0.009245762 -0.02618300  0.02805195  0.0249738622
## 3  0.008926254  0.04074880  0.07765496  0.0546217097
## 4  0.037602093  0.01172896 -0.05686399  0.0005863832
## 5  0.023184296 -0.03139354  0.08109797  0.0941903158
```

As observed, differences are minor, but keep in mind that these minor differences are enough to increase RMSE by 0.07571. The overall positive F1 measure might be an indication. Random forest actually has slightly worse precision on the edges, so it is predicting more 1 and 5 stars when they are not. It also has worse recall

on 4 stars, or predicting less 4 stars than there actually are, compared to the user mean.

Anyways, the similarity of these models make them unsynergistic. When we produce models that differ drastically, they can be stacked together to take advantage of both their strengths.

Clean Business: Second Attempt at Feature Reduction (Part 5)

Jonathan Chang

November 10, 2018

Fixing mistakes

There was a mistake in one of the earlier sections that may have negatively impacted our random forest models. Before, we made the executive decision to replace all *NA*'s with *None*, so that they can be seen as categorical data and no samples are dropped (as many algorithms simply ignore any samples with *NA*). We neglected that some features would have been seen as numerical if not for a few *None*'s sprinkled around the dataset. Particularly, the opening and closing hours, I assumed would have been imported as numeric, but were actually imported as factors, which ruined the ordering of the actual numbers (since factors representations are arbitrarily assigned).

Since we centered and scaled all our data, it would be inconvenient to fix it here, so I fixed it in one of the prior steps (Opening Hours), and re-trained only the final random forest model.

Label Encoding

```
business = read.csv('business_cats.csv')
factor_cols = names(Filter(is.factor, business))
business[, factor_cols] = apply(business[, factor_cols], 2,
                          function(c) { match(c, unique(c)) })
dim(business)
```

```
## [1] 12058   900
```

```
write.csv(business, 'business_labels.csv', row.names=FALSE)
```

One-Hot Mess

In preparing to train for XGBoost, I discovered that the algorithm doesn't actually take factors. A popular approach, then, is to convert factors to numerical data in what's called *label encoding*, with the idea that XGBoost's ability to make multiple splits along the same feature could take care of any ordering problems (since label encoding causes implicit ordering relations between the numbers that may not exist in categorical data). However, since XGBoost uses a gradient-based approach instead of information gain like a typical decision tree, the ordering isn't completely ignored.

The second approach is *one-hot encoding*, or splitting each categorical feature into *n* features representing each of the values (i.e. categories) that it takes. However, the problem here is obvious. In addition to our 799 categories, we have 250 locations, and between 3-5 values in each of our attributes. Each one-hot feature will

mostly contain zeroes. The problem with having overly sparse data is:

1. Some algorithms will perform worse.
2. It will take up a lot of memory. This can be partially solved with a sparse matrix representation, but then it causes the overhead of converting back and forth between formats.
3. It increases training time. In trees, the algorithm has to look through each feature to find the best splits. In regression, we have more weights for the same information content.

In fact, let's create our one-hot encodings to see just how big it gets. Note that the categories are already one-hot in their current format, and the business hours, dates and counts are numeric. That means we just have to one-hot encode the attributes and location. Looking through the dataset, we find that all the attributes and location happen to be the first columns. So we'd like to find which columns to subset.

```
library(caret)
business = read.csv('business_cats.csv')
city_col = which(names(business)=="city")
dummy = dummyVars(~., data=business[,1:city_col], fullRank=TRUE)
one_hots = as.data.frame(predict(dummy, newdata=business[,1:city_col]))
business[,1:city_col] = NULL
```

Full rank regards the *dummy variable trap*. If we keep all the variables in a one-hot encoding, then since the combination of *n-1* columns fully describe the last category by process of elimination, it causes multicollinearity (i.e. dependency). So usually, we remove the last column. In this case, I want to keep all the columns so I can later manually remove the columns labeled "None", to make it neater.

```
one_hots$`city.Peoria, AZ` = NULL # only has 1 sample
one_hots[, grepl("None", names(one_hots))] = NULL
business = cbind(one_hots, business)
dim(business)
```

```
## [1] 12058  1157
```

1157 columns is actually not that bad.

```
write.csv(business, 'business_onehot.csv', row.names=FALSE)
```

Preserving Relations

The problem with one-hot encoding is that it also fails to preserve relations. For example, the hypothetical categories *music_Dj* and *music_Live* are certainly closer to each other than *music_Dj* and *Nebraska*, yet one-hot columns look all the same to the algorithm. While label encoding adds relations that don't exist, one-hot encoding eliminates all relations besides patterns intrinsic in the data. A PCA feature reduction based on one-hot encodings doesn't help since it's still based on one-hot encodings. Further, it hypothetically there is a feature that is 100% correlated to a 1-star rating, but the lack of that feature has no specific pattern, then it would appear to be low variance, eliminated by PCA.

Still, let's make a set reduced by PCA.

```
business = read.csv('business_onehot.csv')

# Set aside numerical columns.
numerical_cols = business[,340:358]
business[,340:358] = NULL

# Train PCA on categorical columns.
pca.model = prcomp(business, scale=TRUE)
pca.variance = summary(pca.model)$importance[3,]
```

This time we'll keep 0.8 variance.

```
num_vectors = table(pca.variance < 0.8)["TRUE"]
pca.vector = pca.model$rotation[, 1:num_vectors]
pca.final = as.data.frame(as.matrix(business) %*% pca.vector)

# Save space
pca.final = round(pca.final, 4)

# Save dataframe
business = cbind(numerical_cols, pca.final)
dim(business)
```

```
## [1] 12058   621
```

Now we're down to a little over a half of the one-hot columns.

```
write.csv(business, 'business_pca.csv', row.names=FALSE)
```

Let's look at a different embedding that preserves some relations.

Word2Vec

Word2Vec is typically used in NLP (peuro-linguistic programming) applications to preserve meaning between words. We'll consider the *skip-gram* variant that takes as input sentences and tries to predict the most likely word. Word2Vec does this using a shallow one-layer neural network by taking windows of size *n* and considering each word as a feature. The neural network then learns the context from each window. The resulting number of features is the size of the hidden layer.

While this isn't typically applied to categorical data, its use has been explored in academia. After all, there is an analogy here, right? If we take each category as a word, then the combination of categories representing each business forms a sentence, or a window. We then learn the context between the categories, preserving relations while reducing the feature dimension. Some have called this Category2Vec, but all the *X*2Vec imply the same algorithm with different feature types. A leading edge variant by Facebook called *FastText* breaks words down to *n*-grams to learn, also, parts of each word to predict unknown similar words – but since our data set is pretty well curated, this should not be an issue. Categorical data are rarely overlapped (e.g. "dj" is pretty distinct from "vegetarian"), and if there are overlaps, we can append prefixes to make them distinguishable if necessary.

Let's convert column names into words.

```
business = read.csv('business_onehot.csv', stringsAsFactors=FALSE)

# Set aside numerical columns.
numerical_cols = business[,340:358]
business[,340:358] = NULL

# Format column names
cols = colnames(business)
cols = gsub(' ', '', cols, fixed=TRUE)
cols = gsub(',', '', cols, fixed=TRUE)
cols = gsub('.', '', cols, fixed=TRUE)
cols = gsub('-', '', cols, fixed=TRUE)
```

Next, we convert each sample into a sentence of column names.

```
sentences = apply(business, 1, function(sample) {
  sample = ifelse(sample != 0, cols, '')
  gsub("\\s+", " ", trimws(paste(sample, collapse=' ')))
})
sentences[1]
```

```
## [1] "BusinessParking_street1 GoodForMeal_dinner1 attributes_Alcoholfull_bar attrib
utes_BikeParkingTrue attributes_BusinessAcceptsCreditCardsTrue attributes_CatersTrue
attributes_OutdoorSeatingTrue attributes_RestaurantsGoodForGroupsTrue attributes_Rest
aurantsPriceRange22 attributes_RestaurantsReservationsTrue attributes_RestaurantsTabl
eServiceTrue attributes_RestaurantsTakeOutTrue attributes_WiFino cityTorontoON cat_Re
staurants cat_Southern cat_CajunCreole"
```

Finally, we train these sentences with Word2Vec. h2o will run on a local Java server so that it's faster.

```
library(h2o)
```

```
h2o.init()
```

```
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         2 days 3 hours
##     H2O cluster timezone:       America/Los_Angeles
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.22.0.1
##     H2O cluster version age:    17 days
##     H2O cluster name:           H2O_started_from_R_jacha_ltu936
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   2.05 GB
##     H2O cluster total cores:    4
##     H2O cluster allowed cores:  4
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     H2O API Extensions:         Algos, AutoML, Core V3, Core V4
##     R Version:                  R version 3.5.0 (2018-04-23)
```

```
sentences = as.h2o(sentences)
sentences = h2o.ascharacter(sentences)
sentences = h2o.tokenize(sentences, ' ')
```

We'll use a hidden layer size of 400 because the authors of the paper recommended 300-400.

```
w2v = h2o.word2vec(sentences,
                   model_id="word2vec_400",
                   min_word_freq=1,
                   word_model="SkipGram",
                   vec_size=400,
                   window_size=7,
                   sent_sample_rate=0,
                   epochs=100)
h2o.saveModel(w2v, 'models/')
```

Applying the model to our categories, we get…

```
text.vecs = h2o.transform(w2v, sentences,
                          aggregate_method='AVERAGE')
business = cbind(numerical_cols, as.data.frame(text.vecs))
dim(business)
```

```
## [1] 12058   419
```

```
write.csv(business, 'business_wv.csv', row.names=FALSE)
```

Obviously, there are a lot of hyperparameters here that can be tuned, but it's too much work to tune hyperparameters for each method here, and on each model. Let's figure out which of these methods are competitive first before tuning them further. This is imperfect because ideally we want to do a grid search of all parameters in case one setting might greatly improve RMSE, but this is a tradeoff between time and results. To be sure, when we find that we can no longer push the frontiers with better models, we might revisit all these things that we've skipped to iron out the kinks.