

Can you predict which company will hire you?

Aarsh Batra

October, 2017

I Introduction

In this paper, I attempt to make use of techniques from Supervised Machine Learning Literature to provide a general framework for building a ‘personalized job classifier’. The purpose of this classifier will be to take in as its input, a company’s feature vector and output a label (e.g. 0/1) that would inform its ‘user’ whether or not he gets a job in that company.^{1,2} ‘Personalized’ means that the classifier is specifically trained and tuned to a specific ‘user’ and is rendered unusable or at least unfit for anyone else to use. I will first present a theoretical framework for building such a classifier which will be followed by a coded simulation written in the R programming language.

In particular, I will start by discussing and solving a similar classification problem which once solved will provide all the tools required to tackle the original problem.³ Then, I will layout the original problem and work towards constructing an algorithm independent general framework. This framework will abstract out all the algorithmic idiosyncrasies and focus on explaining and solving the core of the problem, which once understood can be implemented using a classification algorithm of one’s choice. Next, I will discuss how one can scale up such a model to more than one user. Finally, I present a coded simulation of the entire model building process written in the R programming language. The simulation is downloadable as a R markdown (.Rmd) file such that you can see the code work for yourself (More information on how to download and run the code is provided later in this paper).

II A Similar Problem

A similar problem focuses on building a classifier for a specific company ‘ C_1 ’. This classifier would take in as its input an ‘applicant feature vector’ and output whether or not the applicant gets a job in company C_1 .^{4,5} This problem is relatively easy to tackle than the original problem. Here, the focus is not on the

¹ ‘feature vector’ for any entity ‘E’ refers to a list of attributes that characterize ‘E’. For e.g. if the entity is a ‘House’ then its feature vector includes, its price, location, number of bedrooms, size etc.

² ‘user’ refers to the person for whom the classifier is trained and tuned. I use ‘applicant’ and ‘user’ interchangeably.

³ Original problem is: Can you predict which company will hire you?(focus is on the ‘user’/applicant).

⁴ Similar problem is : Can you predict which applicant will get the job in your company?(focus is on the company).

⁵ Note that the original problem focuses on building a classifier for a specific ‘user’.

applicant but on the company. Later I will show how by repeating this process for companies C_2 to C_n we get very close to solving the original problem.

To build a classifier for C_1 that outputs a label (0/1) for an applicant requires us to create a dataset from the company's job applicants' history.

Figure 1 below displays the structure of such a dataset.⁶

Applicant Name	Feature ₁	Feature ₂	Feature _n	Label(0/1)
----------------	----------------------	----------------------	-------	----------------------	------------

Figure 1: Applicant Dataset Structure

It is crucial that this Applicant Dataset contains both positive (label = 1) and negative (label = 0) training examples. This is an important concern as we want the algorithm to learn and identify the combination of features that count towards getting a job (label = 1) and also the combination that count towards not getting a job (label = 0).⁷ In other words, the Applicant Dataset should contain both types of applicants, the ones that got the job and also the ones that didn't.

A single row in the dataset corresponds to a single applicant, for whom we have available, a ground truth label (0/1). Next, we have to construct a feature vector for the applicant, which together with the applicant's name and ground truth label will give us a single row in our dataset or in other words a single training example. By repeating this process for all applicants we will have our training set which can then be fed to the learning algorithm.

For constructing the feature vector I assume that the training set is rectangular.⁸ A rectangular training set implies that the size of the feature vector constructed for each applicant would be same. For doing that, a predefined set of categories has to be selected, on which to collect data for each applicant. Some examples of such categories are as follows; years of education, years of experience, number of connections etc. Next step requires collecting data on those pre defined categories for each applicant. This can be done in part via a business and employment oriented social networking service like LinkedIn or the like. Within a LinkedIn profile of an applicant we can leverage languages like Python which has support for web scraping packages like 'Beautiful Soup', which creates a parse tree for parsed pages that can be used to extract data from HTML and XML documents. Now we can scrape off all the data for the pre defined set of categories for each applicant one by one from their profiles.

One important thing to realize is that, it is relatively easy to construct training examples that have a ground truth label of '1'. In other words, from a data source like LinkedIn, we can only find people who 'got the job' (label = 1) in a particular company (C_1). No one specifies on his/her LinkedIn profiles, the

⁶ Labels: 0 = 'does not get the job', 1 = 'gets the job'. These labels are also referred to as ground truth labels.

⁷ One 'training example' corresponds to one applicant or in other words a single row in Figure 1: Applicant Dataset Structure. Together all such training examples form a 'training set'.

⁸ Rectangular data: collections of values that are each associated with a variable and an observation.

companies by which they were ‘turned down’ (label = 0). But, as mentioned earlier, it is crucial that our Applicant dataset for C_1 contains both positive (label = 1) and negative (label = 0) training examples.

Unlike data captured from LinkedIn profiles, where people willingly make their private information available to a wider audience. Getting data on those job applicants of company C_1 that were turned down (label = 0) is less easy. People do not usually want to share such information and the companies having such information (e.g. C_1 ’s past job applicant history) have legal constraints on sharing such information. But, there are workarounds which allows us to remain within the legal boundaries and still get access to the required information.

One workaround is to buy from company C_1 , the required dataset (e.g. records/results of past job applications), but with personally identifiable information (information that makes it possible for us to identify specific individuals) removed. For our application to work well, we need to know the combination of features that count towards getting a job and the combination of features that count towards not getting a job. It is certainly not necessary for us to know the names, home addresses, etc of the applicants (even though that information might in and itself count as interesting features, we may have to let it go). Other workarounds include repeating a similar process as above but instead of buying data from company C_1 , we could buy data from more ‘general purpose’ data brokers. These brokers might be able to offer a relatively richer (in terms of features) dataset, still remaining within the legal boundaries. Once we have the required datasets, we can then use them to construct training examples with label = 0.⁹ Next, I will focus on the process of ‘feature extraction’.

Please note that I assumed that the Applicant dataset we constructed is rectangular. This assumption is made to simplify the setting in which we construct our model. But one should know that, if we want to leverage information from datasets that are unstructured (non-rectangular), there are techniques that allows us to do that. Similarly, if we want to leverage information from different ‘types’ of data (e.g. images, video files, etc.), we can. But, throughout this essay, I will continue following the assumption that our datasets are rectangular so as to keep our primary focus on the question at hand (Can you predict which company will hire you?) and not get lost in the implementational details. But, even constructing a rectangular dataset in and itself requires a fair bit of work. Let’s go on to discuss the process of feature extraction.

The categorical data that we have collected for each applicant is not directly usable in its current state.¹⁰ It has to be first pre-processed and then normalized to be declared ‘fit’ for training. This part of the process is called ‘feature extraction’.¹¹ ‘Extraction’ means that we do not directly transition from the data

⁹ It is of ‘extreme’ importance to make sure that one has the appropriate legal clearance in trying to get access to/ creating datasets that might contain ‘Personally identifiable information’. Legal clearance might be needed from IRB’s (Institutional Review Boards) and/or equivalent institutions for applications that either directly or indirectly involves human subjects and/or makes use of information that is potentially ‘personally identifiable’.

¹⁰ At least the non-numeric categories are not directly usable.

¹¹ Normalization of data is done to bring different features into similar ranges. This improves learning and makes algorithms like gradient descent converge faster. This is usually done by subtracting off the mean of the feature from itself and then dividing the result by the feature’s standard deviation.

collection phase to the model training phase. The feature extraction phase is not entirely synonymous with the cleaning of data, as the data we collect/acquire might be clean but in a 'state' that makes it use limited.

Constructing some features require minimal work as they are already in the state we want them to be. For e.g. the 'number of connections' block in a LinkedIn profile page is directly usable as it's data type is an integer, other such examples include, 'yrs of experience', 'yrs of education' etc. Such features require minimal work like normalization and then they are good to go.

Other features might not be directly usable. For e.g., We might think that an applicant's Alma mater is a feature that is an important determinant in predicting whether or not he gets the job. But, we usually do not directly feed in the html block containing the Alma mater information into the dataset. We can do that and the machine learning algorithms will make sense of it (given enough labeled training examples). But in practice (or at least for the purposes of this essay) we establish some form of structure (e.g. a hash table, associative array, etc.) that makes learning faster, intuitive and more efficient.¹²

So, we could map each college name (value) to a number (key). This way we could construct a mapping that would help us efficiently incorporate one's Alma mater as a feature in our dataset. In some cases such mapping might not be very useful due to the hash table being very unevenly distributed (when there are either too many or too less keys).

If hash tables are not useful, a different way to capture the 'Alma mater' feature would be to use yes/no (0/1) questions. Examples of such questions are as follows, 'Did you attend Harvard as your Alma mater?', 'Did you attend MIT as your Alma mater?', 'Do you have more than a year's experience working with the R programming language?', etc. This is a very useful and fast way to construct new features that are binary (0/1) in nature. These types of features captures interesting information, take up significantly less memory and are relatively easy to construct such that a lot of these can be constructed relatively fast. Moreover, there is no need to clean or normalize them due to their binary nature. But, these features require hand designing on our part. This hand designing is based on what we think counts as interesting features (so there is an added risk of human added biases). We are forgoing benefits of techniques such as 'automatic feature selection' using train/development/test sets that the field of Machine Learning has to offer. These techniques aim at choosing the subset of features that minimize their corresponding cost function on the cross-validation set.¹³

Usually the process of feature extraction involves both hand designing features and leveraging techniques from the field of Machine Learning to automate feature selection process.¹⁴

¹² Hash table data structures are daily used in applications like spam classification to convert an email into an array of numbers, which then with a label (0/1), serves as a single training example in the training set for our spam classification dataset.

¹³ Intuitions of what one thinks are 'interesting' features is a very subjective matter. Using handpicked features (as opposed to using an algorithm to min a cost function) might result in us unknowingly adding a small bias term to our model.

¹⁴ Sometimes, carefully hand designed features saves a lot of computing time that machine learning algorithms will otherwise take to learn what subset of features are interesting. It is like leveraging the best computing machine we currently have (our brain) for a little time (maybe a few minutes) to save probably hours of computing needed to find the best subset of features.

Figure 2 summarizes the typical workflow for constructing a single training example/single applicant.¹⁵

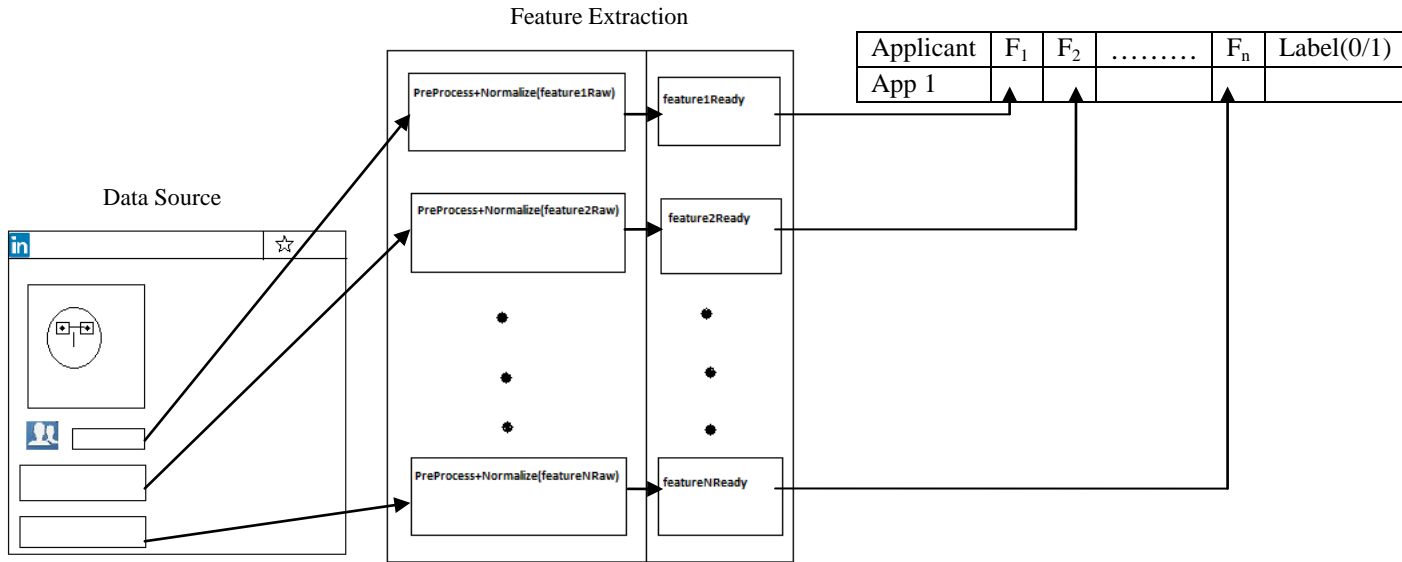


Figure 2: Typical Workflow to construct a single training example

We repeat the ‘typical workflow’ (shown in Figure 2 above) ‘m’ times (where, m is the number of training examples that we want in our Applicant dataset) to get the entire training set. We then feed that training set to an algorithm e.g. logistic regression. This algorithm will further use some optimizer like gradient descent to minimize a cost function (e.g. cross-entropy cost function) and hand out the learned parameters which can then be used to make a prediction on new unseen examples.¹⁶

This initial implementation of the model is dirty. Our objective is to be able to perform well on previously unseen examples and it is unlikely that our first implementation would be the one that serves our purpose. It is very important for us to not fall for pre mature optimization while building such models. It is recommended that we build a model, test it, rebuild it, rather than trying to optimize before building. A lot of tools are available for tuning the model once its initial implementation is ready. Some examples are, dividing the entire dataset into training, development/CrossValidation and test subsets and then using that to systematically select hyper parameters like the ‘learning rate’, or the ‘regularization parameter’ Λ (lambda).¹⁷

¹⁵ Figure 2 is just meant for illustrative purposes, i.e. we do not go through the entire workflow one training example at a time. Rather, we first collect data on all applicants from the data source, then we perform feature extraction for all applicants and then we get our entire training set, all at once.

¹⁶ Note that logistic regression for an applicant outputs $\text{Probability}(\text{label} = 1 \mid \text{training set})$, so in the above example, if suppose we input a new applicant’s feature vector we get as the output of the hypothesis function, 0.8, this means that there is an 80% chance that the applicant will get the job. Now, to output a label for the applicant we have to set a threshold e.g. 0.5, above which we predict 1 else 0.

¹⁷ We train different models only on a small part of the ‘entire dataset’, and call that part the ‘training/train’ set. Then we use the cross-validation set to choose the best performing (in terms of minimizing the cost function) model from these different models. Further, we use the chosen model and evaluate its performance by testing it on the ‘test’ set, which gives us a reasonable estimate of the model’s generalizability.

Other examples include, using techniques like regularization to address over fitting.¹⁸

Given the availability of all these techniques, premature optimization is not a good idea. So, rather than being fixated on to decide what the learning rate or the number of iterations of gradient descent should be, we should use these machine learning techniques of automatic model selection to serve that purpose.

Now, let's assume that we have trained and tuned our model such that it is giving us a high predictive accuracy and also good performance on the test set.¹⁹ We can now use it to classify whether or not a new applicant will get a job in company C_1 . Now, having solved this problem we have already come very close to solving the original problem (see pg. 1, footnote no. 3). Now, let's tackle the original problem.

III Original Problem

In section II, I provided a framework to build a model that addresses the question 'Can you predict which applicant will get the job in your company?'. There, the input to the algorithm was the applicant feature vector and the output was a label (0/1) which informed the applicant whether or not he got the job in that company. Note that this model was built for a specific company C_1 , so if an applicant wants to know whether or not he gets a job in company C_2 , he cannot use this model. To solve this problem, we have to build a general model. This model will take in as its input, not the applicant feature vector, but rather the 'company feature vector' and inform the applicant/user of his job prospects (0/1) for any company he is interested in. The original problem presented ahead aims at building such a model.

The original problem states, 'Can you predict which company will hire you?'. It aims at building a model for a particular user/applicant (see pg.1, footnote no. 2), the focus is on the user/applicant. All that the user has to do is to enter the company feature vector for the company he is interested in and have as the output of the model a label that will inform him (with certain probability; see pg. 4, footnote no. 13) whether or not he will get a job in that company. So, in this model the focus will be on a specific 'user'. Also, the model that is built will be trained and tuned to the specific user and will be rendered unusable or at least unfit for anyone else to use. I assume from now on that the model is built for a user named User1. Next, I will layout the entire process of building this model in a few lines and then expand on it.

In section II, we built the model for a specific company C_1 (let's call that model M_1). Now, we have to repeat the entire process used to construct M_1 , for companies C_2 to C_n , which gives us models M_2 to M_n (corresponding to companies C_2 to C_n). We have 'n' models each corresponding to a specific company. Now, User1 can in turn use M_1 to M_n to get 'n' labels L_1 to L_n , each of which will inform him of his job prospects (0/1) in companies C_1 to C_n . In section II, we built an applicant feature vector, one for each applicant, in a similar way now, we have to build a company feature vector, one each for companies C_1 to

¹⁸ Regularization (in a few words) makes it costly for our model to have 'big' weights/parameters. Adding the regularization term to the cost function and then minimizing it, adds an additional incentive to push 'weights' towards 'zero'. The size by which the weights are pushed towards 'zero' depends on the size of the regularization parameter Λ (lambda).

¹⁹ Predictive accuracy of the model is equal to the proportion of the training examples it classifies correctly.

C_n . Once we do that we will have ‘n’ company feature vectors (V_1 to V_n) each of which is ‘n’ dimensional. Having done that, for User1, we have labels L_1 to L_n (each of which tells him whether or not he got a job in companies C_1 to C_n). Also, we have ‘n’ feature vectors V_1 to V_n , corresponding to companies C_1 to C_n .

Now, we have all we need to create our new ‘Company Dataset’. In this dataset, each training example contains information for a single company in the form of ‘n’ features and a label. So, row_1 in the training set corresponds to C_1 which possesses a ‘n’ dimensional feature vector V_1 and a label L_1 . Similarly row_2 in the training set corresponds to C_2 which possesses a ‘n’ dimensional feature vector V_2 and a label L_2 . This continues in a similar fashion till row_n in the training set which corresponds to C_n which possesses a ‘n’ dimensional feature vector V_n and a label L_n .

Figure 3 below represents the structure of the ‘Company dataset’.

Company Name	Feature ₁	Feature ₂	Feature _n	Label(0/1)
--------------	----------------------	----------------------	-------	----------------------	------------

Figure 3: Company Dataset Structure

Note that we get labels L_1 to L_n by running a ‘specific user’s’, applicant feature vector ‘in turn’ through models M_1 to M_n . This means that the Company Dataset represented above is ‘personalized’ to a specific user and is not suitable for anyone else to use. This idea is better represented by an example.

First, let’s see what row_1 in the above Company Dataset represents. First comes the Company Name (C_1), that is followed by a ‘n’ dimensional company feature vector (V_1), which defines C_1 in terms of n features (some examples of features could be “Is the company a fortune 500”, ‘stock price’, ‘Number of employees’, etc.), after that comes a label (L_1), this label L_1 comes from running a ‘specific user’s’ (let’s assume that user is named User1) applicant feature vector through the model M_1 (remember, M_1 took as its input an applicant feature vector and outputted whether or not that applicant got the job in company C_1), which outputs a label (let’s assume that label was 1, i.e. User1 got the job). So, an informal way to read row_1 of the table is,

“As per M_1 , User1 got the job in company C_1 which has features V_1 (number of employees, stock price, etc.) “

If we follow a similar procedure for row_2 to row_n , while keeping in mind that, just like L_1 , each label L_2 to L_n comes from running User1’s applicant feature vector in turn through models M_2 to M_n . (this is what makes the model ‘personalized’ to User1, as all labels L_1 to L_n in our entire Company Dataset correspond to User1). So, we can now informally read all other rows just like we read row_1 above as follows. (For illustration purposes, I assume that row_2 has label = 1, row_3 has label = 0, row_n has label = 1).

“As per M_2 , User1 got the job in company C_2 which has features V_2 (number of employees, stock price, etc.) “

“As per M_3 , User1 didn't get the job in company C_3 which has features V_3 (number of employees, stock price, etc.) “

“As per M_n , User1 got the job in company C_n which has features V_n (number of employees, stock price, etc.) “

There are few important things to notice in the above statements. First, across all statements the ‘user’ remains the same (User1), this is what I mean by a model ‘personalized’ to User1. Second, each statement corresponds to a specific row in the Company Dataset. Third, each row corresponds to a company, a model, a feature vector and a label, e.g. row₁ corresponds to company C_1 , model M_1 , feature vector V_1 , and label L_1 . Finally, the ‘Company Dataset’ is also a rectangular dataset (see pg. 2, footnote no. 8).

Once we feed the entire Company Dataset to our learning algorithm like logistic regression or the like, it will go on to learn (begin training) what combination of company features count towards User1 getting a job and also what combinations of company features count towards User1 not getting a job. Once trained and tuned, the model serves as a ‘personalized job classifier’ for User1, which means that the model can now take any new company that it has not seen before, distill that company’s characteristics into a company feature vector, take that vector as its input and output a label (0/1) which will inform User1 (with certain probability, see pg. 4, footnote no. 13), whether or not he gets a job in that new company. Also, User1 can do this for any number of new companies, all he has to do is enter as an input to the model the company feature vector of the company he is interested and get a label as an output that will inform him of his job prospects in that company.

With that, the model is complete. I answered how someone can build a model that lets him predict whether or not a particular company will hire him. Next, I will expand on a few important details of the model and then I will summarize the entire process of building the whole model (personalized job classifier) for a specific user (User1) in a diagram

First, I will discuss, How to construct a company feature vector?. In section II (Figure 2), I showed how we can construct an applicant feature vector using the LinkedIn profile of an applicant. Like an applicant, a company may not have a LinkedIn profile. But, every company has some form of online presence, e.g. a website, blog, etc. So, the ideas discussed, to construct an application feature vector in section II, can in similar ways be leveraged to construct a company feature vector. To highlight some of the important things, first we have to decide the categories on which to collect data (from a data source, like a company website) for each company. After that we have to perform feature extraction, keeping in mind that our

dataset is rectangular so that the features we decide to extract from the categorical data we collected for ‘ C_1 ’ (e.g. stock price, number of employees) are same for all other companies C_2 to C_n . In other words, if we have two features (stock price, number of employees) for C_1 , we will have those same features for C_2 (with values corresponding to C_2 filled in), and the same for C_3 to C_n (with their corresponding values filled in). Next, we may want to leverage data structures like hash tables (as explained in section II) to construct various features. Also, we discussed ways of creating interesting binary features, in the same way, we can create binary features that capture important information about the companies, some examples are, ‘Is this company a fortune 500’, ‘Is this company listed in a stock exchange’ etc.

One advice for anyone building such a model would be to focus only on the set of companies relevant to him. For e.g. If an IT professional wants to build such a model for himself, then maybe he should streamline his training set to only include companies from the IT sector (and/or, some other sectors where he thinks there are hints of job prospects). Including companies from sectors where his job prospects are ‘nil’ might not hurt, but unnecessarily crowding the training set with the companies for which he is sure of not getting a job (as those are the companies that don’t match his skill set and are already classified with label = 0.) might not be a good idea.²⁰

Next concern is regarding the quality and quantity of features and the size of the training set. By quality features, I mean those features that strongly count towards predicting the variable we are interested in. In other words, features that explain most of the variance in the underlying variable we want to predict. For e.g. if I want to predict housing prices, some quality features might include ‘size of the house’, ‘location of the house’, etc. Other not so good features that might somehow contribute towards deciding the price of a might include ‘distance to the nearest train station’. One of the ways to choose quality features is to handpick the ones we ‘think’ are good. Other ways (that are usually complementary) include using train/CV/test sets to systematically choose the set of features that correspond to the model which minimizes the cost function on the cross validation set.

Finally, let’s discuss one more concern, the quantity of features and the size of the training set. The total number of features we choose to have, has a bearing on the total number of training examples. To illustrate, suppose we decided to collect five hundred features (i.e. five hundred columns) for each row in our training set. So, if we had a thousand training examples (rows), the size of our training set would be (1000, 500). But, now suppose we only have five training examples such that the size of the training set is (5, 500). The difference between a training set size of (1000, 500) and (5, 500) is that in the latter case, our learning algorithm has only five instances of data to decipher patterns from 500 (and much more in terms of total permutations /combinations) features, which is not nearly enough to learn the underlying general statistical patterns. Given that, it is important that our training set size (total number of instances of data) is big enough (relative to the total number of features) so that our learning algorithm could learn a

²⁰ Adding companies from sectors where job prospects are ‘nil’ can ideally be done, if someone wants to build a very ‘general purpose’ personalized job classifier. If someone considers building such a model than an important thing to remember is that, adding a few additional training examples of companies from these ‘other sectors’ might not be enough, for our algorithm to decipher meaningful patterns from it. So, we may have to provide a lot of new training examples of the ‘other sector’ type.

reasonable representation (that will perform good on the test set) of the underlying statistical patterns in the data. In other words, if we have a very large training set, then we can accommodate a lot of features into our model. Exactly how large the training set size should be relative to the number of features is debatable. But, the training set size should, at the least be equal to the total number of features (for e.g. if we have two features, than we at the least need two training examples, simply because in order to solve for two variables, at the least we need two equations.)

So, if our training set is big enough, such that we can accommodate a lot of features, than we should, as that increases the granularity of the model. For example, in our original model, if we assume that our training set size is significantly larger than the total number of features (e.g. 20x larger), than we can accommodate a lot of features so that our model is able to more finely recognize the ‘company features’ that count towards User1 getting a job and hence is able to pick very small differences between different company feature vectors. This might not be possible with a small set of features (There is an additional concern of ‘over fitting’ in models with high dimensional feature spaces, but with machine learning techniques like ‘regularization’ in our toolbox, that can be handled quite reasonably.)

Below, I summarize the entire process of building a ‘personalized job classifier’ for ‘User1’ in a diagram,

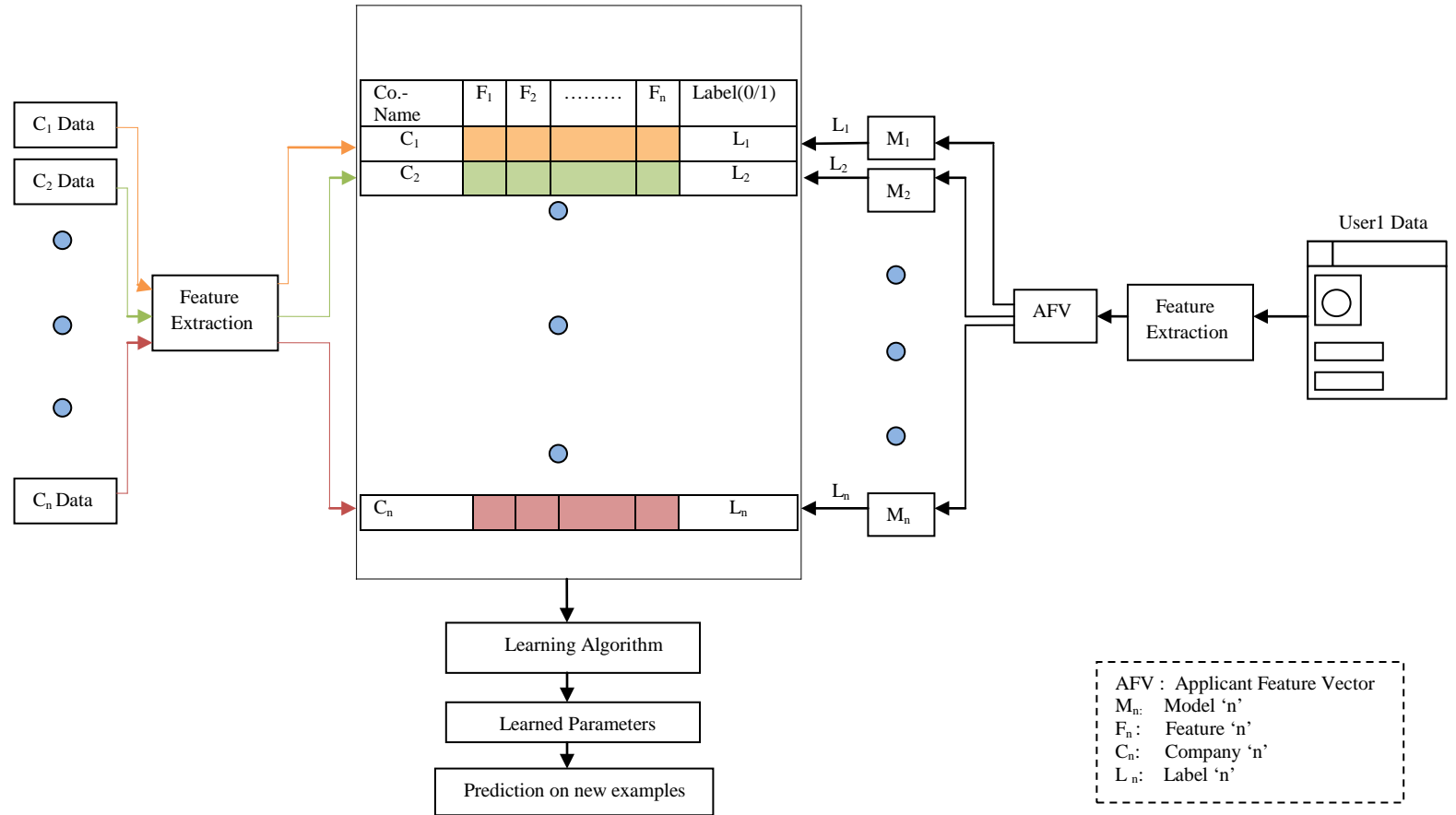


Figure 4: Process of building a personalized job classifier for User1

IV Conclusion

Figure-4 summarizes the entire process for building a personalized job classifier for a specific user (User1). Throughout in the essay, I mentioned that the model is built for a specific user (User1). Now, in these last few lines I will discuss, how we can very easily scale up this model to any number of users (assuming that these other users belong to the same profession as User1 does, and/or these other users belong to the same group of people as User1 does, e.g. the group of Mathematics Graduates).

Let's assume that User1 is a Mathematics Graduate and wants to apply for a job. Now, User1 builds for himself a personalized job classifier. To build a model for User2 (who is also a Mathematics graduate) takes very less time as we already have the infrastructure for User1's model, the only things that change while building the model for User2 is the applicant feature vector and the labels (L_1 to L_n). In other words, as User2 belongs to the same group of people as User1 (Mathematics Graduates), it is reasonable to assume that while applying for a job, User2 also aims at applying at the companies that User1 aims at, so that there is no need to reconstruct the company feature vectors. Given that, the only thing that User2 need to do is to construct his own 'personalized labels', which he will do by constructing his applicant feature vector and passing it through models M_1 to M_n (which are the same models that User1 used and hence are reutilized by User2) to get his own personalized labels (L_1 to L_n). These labels together with the company feature vectors can then be passed to the learning algorithm, which will output the learned parameters that will then be used to make prediction on new unseen examples for User2. Similarly, we can scale up this process to any number of users (given they belong to the same group/category of people).

But, what if User1 and User2 belonged to different professions?. In that case, if we assume that the companies relevant to User1 are not relevant to User2 (because they don't match User2's skill set). Then, at best, the only things that User2 can leverage from User1's already built infrastructure are its company feature vectors. For User2 that would serve as data with label = 0 (as User2 is sure of not getting a job in companies relevant to User1 such that he can safely classify them with label = 0, and add them to his training set). Note that this borrowed infrastructure might not serve as quality data for User2 as these training examples are from a sector where the job prospects of User 2 are 'nil' (see pg. 9, footnote no. 20). (Besides that, remember that User2 also has to construct his personalized labels for which he has to construct models M_1 to M_n and his applicant feature vector).

The theoretical part of the model is complete. Next, let's dive into the *Personalized Job Classifier Model Simulator*.

Personalized Job Classifier Model: Simulation

Aarsh Batra

October, 2017

About

The Personalized Job Classifier model aims at building a model personalized to a specific ‘User’ such that when feeded with information of a specific company (in form of a feature vector) the model will output a binary label (0/1) that would inform its ‘User’ of his job prospects in that company (‘0’ = didn’t get the job; ‘1’ = got the job). **Personalized** means that the classifier is specifically trained and tuned to a specific User and is rendered unusable or at least unfit for anyone else to use. Below, I provide a simulation for the entire model building process from scratch in the R programming language. This is essentially my lab notebook for the model in which I have recorded (alongside the code) all of my thoughts that went into writing the code.

Recommended Reading

Please read the paper presented above which I wrote in correspondance to the code below. The paper lays down the theoretical foundation and the terminology for the model. It is 11 pages and would take 15-20 min to read. I strongly recommend reading it as that way all the code ahead will make much more sense. I will provide explanations in good detail but moving ahead I strongly recommend that you read the paper.

Purpose

The purpose of this simulation is to serve as a guide to anyone wanting to build a Personalized Job Classifier for themselves. The code ahead covers the entire process of building such a model, which involves but is not limited to building datasets from scratch, followed by training, tuning, prediction and measuring of accuracy of the models built.

Important Considerations

Below considerations will make more sense once you have read the corresponding paper.

- Throughout the code I assume that the model is built for a user named ‘User1’.
- Total number of companies = 100; Total number of applicants per company = 100. These numbers are arbitrary and you may choose to build model with as many number of companies as you wish.
- Both ‘Applicant Datasets’ and ‘Company Dataset’ use binary features only. This is done for simulation purposes such that datasets can be generated automatically from scratch using commands like ‘runif’. But, when building a full fledged implementation of the model, one must use techniques like web scraping to generate features much richer than the binary ones.
- I have built this simulation by assuming that User1 is a person whose academic credentials lie in fields like Economics/Data Analytics/Computer Science. (You can think of User1 as an Economics Graduate or a CS graduate). Given that, I have appropriately constructed the features (columns) for the applicant dataset on which to fill in information for various applicants. Also I have assumed that companies C_1 to C_{100} are the type of companies which are looking for applicants with credentials like that of User1 (for e.g. some consultancy firm, Banks, etc). You may build classifiers by assuming different settings.
- Finally, when I mean ‘company’ that does not mean company in the ‘strict’ sense. It generally refers to any kind of employer User1 might be interested in, e.g. University, NGO, Research Group, Banks etc.

Pacakges and Version information

I recommend updating to the latest R version (3.4.2, as on October, 15, 2017) to avoid any clashes with pacakges built under different versions of R. Below is the list of pacakges that were used to build this model (The list was generated using `devtools::session_info()` command).

```
load("sessionInfoRestore.RData")
sessionInfoRestore
## $platform
## $version
## [1] "R version 3.4.2 (2017-09-28)"
##
## $system
## [1] "x86_64, mingw32"
##
## $ui
## [1] "RTerm"
##
## $language
## [1] "(EN)"
##
## $collate
## [1] "English_United States.1252"
##
## $tz
## [1] "Asia/Calcutta"
##
## $date
## [1] "2017-10-20"
##
## attr(,"class")
## [1] "platform_info"
##
## $packages
##      package * version      date      source
## 1  assertthat   0.2.0 2017-04-11 CRAN (R 3.3.3)
## 2   backports   1.1.1 2017-09-25 CRAN (R 3.3.3)
## 3     base *    3.4.2 2017-09-28      local
## 4     bindr      0.1 2016-11-13 CRAN (R 3.3.3)
## 5  bindrcpp     0.2 2017-06-17 CRAN (R 3.3.3)
## 6     caret *   6.0-77 2017-09-07 CRAN (R 3.3.3)
## 7     class     7.3-14 2015-08-30 CRAN (R 3.4.2)
## 8  codetools    0.2-15 2016-10-05 CRAN (R 3.4.2)
## 9  colorspace   1.3-2 2016-12-14 CRAN (R 3.3.2)
## 10  compiler    3.4.2 2017-09-28      local
## 11    CVST      0.2-1 2013-12-10 CRAN (R 3.3.3)
## 12  datasets *  3.4.2 2017-09-28      local
## 13   ddalpha     1.3.1 2017-09-27 CRAN (R 3.3.3)
## 14  DEoptimR    1.0-8 2016-11-19 CRAN (R 3.3.2)
## 15  devtools    1.13.3 2017-08-02 CRAN (R 3.3.2)
## 16   digest     0.6.12 2017-01-27 CRAN (R 3.3.2)
## 17   dimRed     0.1.0 2017-05-04 CRAN (R 3.3.3)
## 18    dplyr     0.7.4 2017-09-28 CRAN (R 3.3.3)
```

## 19	DRR	0.0.2	2016-09-15	CRAN	(R 3.3.3)
## 20	e1071	1.6-8	2017-02-02	CRAN	(R 3.3.3)
## 21	evaluate	0.10.1	2017-06-24	CRAN	(R 3.3.3)
## 22	foreach *	1.4.3	2015-10-13	CRAN	(R 3.3.3)
## 23	ggplot2 *	2.2.1	2016-12-30	CRAN	(R 3.3.3)
## 24	glmnet *	2.0-13	2017-09-22	CRAN	(R 3.4.2)
## 25	glue	1.1.1	2017-06-21	CRAN	(R 3.3.3)
## 26	gower	0.1.2	2017-02-23	CRAN	(R 3.3.3)
## 27	graphics *	3.4.2	2017-09-28		local
## 28	grDevices *	3.4.2	2017-09-28		local
## 29	grid	3.4.2	2017-09-28		local
## 30	gtable	0.2.0	2016-02-26	CRAN	(R 3.3.2)
## 31	highr	0.6	2016-05-09	CRAN	(R 3.3.2)
## 32	htmltools	0.3.6	2017-04-28	CRAN	(R 3.3.3)
## 33	ipred	0.9-6	2017-03-01	CRAN	(R 3.3.3)
## 34	iterators	1.0.8	2015-10-13	CRAN	(R 3.3.3)
## 35	kernlab	0.9-25	2016-10-03	CRAN	(R 3.3.2)
## 36	knitr	1.17	2017-08-10	CRAN	(R 3.3.3)
## 37	labeling	0.3	2014-08-23	CRAN	(R 3.3.2)
## 38	lattice *	0.20-35	2017-03-25	CRAN	(R 3.4.2)
## 39	lava	1.5.1	2017-09-27	CRAN	(R 3.3.3)
## 40	lazyeval	0.2.0	2016-06-12	CRAN	(R 3.3.2)
## 41	lubridate	1.6.0	2016-09-13	CRAN	(R 3.3.3)
## 42	magrittr *	1.5	2014-11-22	CRAN	(R 3.3.3)
## 43	MASS	7.3-47	2017-02-26	CRAN	(R 3.4.2)
## 44	Matrix *	1.2-11	2017-08-21	CRAN	(R 3.4.2)
## 45	memoise	1.1.0	2017-04-21	CRAN	(R 3.3.3)
## 46	methods *	3.4.2	2017-09-28		local
## 47	ModelMetrics	1.1.0	2016-08-26	CRAN	(R 3.3.3)
## 48	munsell	0.4.3	2016-02-13	CRAN	(R 3.3.2)
## 49	nlme	3.1-131	2017-02-06	CRAN	(R 3.4.2)
## 50	nnet	7.3-12	2016-02-02	CRAN	(R 3.3.3)
## 51	pkgconfig	2.0.1	2017-03-21	CRAN	(R 3.3.3)
## 52	plyr	1.8.4	2016-06-08	CRAN	(R 3.3.3)
## 53	prodlim	1.6.1	2017-03-06	CRAN	(R 3.3.3)
## 54	purrr	0.2.3	2017-08-02	CRAN	(R 3.3.3)
## 55	R6	2.2.2	2017-06-17	CRAN	(R 3.3.3)
## 56	Rcpp	0.12.13	2017-09-28	CRAN	(R 3.3.3)
## 57	RcppRoll	0.2.2	2015-04-05	CRAN	(R 3.3.3)
## 58	recipes	0.1.0	2017-07-27	CRAN	(R 3.3.3)
## 59	reshape2	1.4.2	2016-10-22	CRAN	(R 3.3.3)
## 60	rlang	0.1.2	2017-08-09	CRAN	(R 3.3.3)
## 61	rmarkdown	1.6	2017-06-15	CRAN	(R 3.3.2)
## 62	robustbase	0.92-7	2016-12-09	CRAN	(R 3.3.3)
## 63	rpart	4.1-11	2017-03-13	CRAN	(R 3.4.2)
## 64	rprojroot	1.2	2017-01-16	CRAN	(R 3.3.2)
## 65	scales	0.5.0	2017-08-24	CRAN	(R 3.3.3)
## 66	sfsmisc	1.1-1	2017-06-08	CRAN	(R 3.3.3)
## 67	splines	3.4.2	2017-09-28		local
## 68	stats *	3.4.2	2017-09-28		local
## 69	stats4	3.4.2	2017-09-28		local
## 70	stringi	1.1.5	2017-04-07	CRAN	(R 3.3.3)
## 71	stringr	1.2.0	2017-02-18	CRAN	(R 3.3.2)

```
## 72     survival      2.41-3 2017-04-04 CRAN (R 3.3.3)
## 73     tibble       1.3.4 2017-08-22 CRAN (R 3.3.3)
## 74     timeDate     3012.100 2015-01-23 CRAN (R 3.3.3)
## 75     tools        3.4.2 2017-09-28          local
## 76     utils *      3.4.2 2017-09-28          local
## 77     withr        2.0.0 2017-07-28 CRAN (R 3.3.3)
## 78     yaml         2.1.14 2016-11-12 CRAN (R 3.3.2)
##
## attr(,"class")
## [1] "session_info"
```

What is the code doing?

Figure 1 below gives an overview of the entire code. After the parameter initialization step, the entire code is divided in two parts.

```
1 Parameter initializations
2 for each company do
3   Build applicant dataset (sans labels column)
4   Assign row and column names to applicant dataset
5   Perform logic correction on applicant dataset
6   Generate labels using criteria number
7   Add labels column to the applicant dataset
8   Set Controls for model training, e.g train/test split, etc.
9   Train model using training set, save model for future ref
10  Create User1's applicant feature vector
11  Pass AFV through the trained model to get a label.
12  Save the label in 'myPersonalizedLabels' vector
13  Save other datasets for future reference.
14 end
15 Build company dataset (sans labels column)
16 Assign row and column names to company dataset
17 Perform logic correction on company dataset
18 Add 'myPersonalizedLabels' as the labels column for company dataset
19 Set Controls for training the 'Personalized Job Classifier' model
20 Train model using training set, save model.
21 Done! Use the model on 'new' unseen companies.
```

Figure 1: Code Outline

Part-1 (Steps:2-14, Figure 1) builds 100 models (M_1 to M_{100}) corresponding to 100 companies (C_1 to C_{100}). Also, in every iteration (one pass through the for loop shown in Figure 1) it accumulates the personalized labels corresponding to User1 in `myPersonalizedLabels` vector (Remember the personalized label for User1 corresponding to C_n is obtained by passing the **Applicant Feature Vector** of User1 through M_n).

Part-2 (Steps:15-21, Figure 1) builds the `companyDataset` which is used to train the Personalized Job Classifier Model.

Next, let's begin with going through the actual code. First comes the Parameter Initialization step:

Parameter initialization

```
# used later to calculate elapsed Time-----
startTime <- proc.time()

# set current working directory-----
setwd("C:/Users/pc/Dropbox/RA Docs/2018-19/PersonalizedJobClassifier")

# set seed-----
set.seed(336)

# load packages-----
library(tidyverse)
library(knitr)
library(caret)
library(magrittr) # for using pipes (%>%)

# parameter initialization-----
kTotalNumOfApp <- 100
kTotalNumOfAppDatasets <- 100
kTotalNumOfFeat <- 20
kTotalNumOfCriteriaions <- 10

# lists for storing datasets for debugging-----
# list of all applicant datasets (for companies C-1 to C-100)
# where appDatasetList[[z]] is the applicant dataset for the
# z'th company (C-z). Also, making lists for storing the train
# and test subsets of appDatasetList[[z]]
appDatasetList <- list()
appDatasetListTrainData <- list()
appDatasetListTestData <- list()

# list for storing models (M-1 to M-100, corresp to C-1 to C-100)-----
modelsList <- list()

# more on this in later code-----
criteriaNumberRecord <- c(rep(NA, times = kTotalNumOfAppDatasets))

# stores labels as returned by running User1's AFV through models M-1 to M-100
myPersonalizedLabels <- c()

# user interface-----
# print(sprintf(">>> Simulation Part-1: total iterations = %i",
#               kTotalNumOfAppDatasets), quote = FALSE)

# print("      Running...", quote = FALSE)
```

After Parameter initialization step comes Part-1 of the code,

Part-1: Code

Part-1 of the code comprises of **Steps: 2 to 14**, in Figure 1. This is a relatively big chunk of code. I recommend first reading it all in one go and then revisit specific parts as and when necessary. Please read the code **AND** the comments for best understanding. I have tried to make the comments sufficiently descriptive keeping in mind the reader who hasn't read the corresponding paper. But these comments are no substitute for the paper. To get the most out of this code, read the corresponding paper.

```
# Part-1: Build 100 models M-1 to M-100 corresp to C-1 to C-100-----
for (z in 1:kTotalNumOfAppDatasets){

  # filling applicant dataset (tmpAppDatasetConstructor) of C-z-----
  # Generate 20 random numbers (single row in the applicant dataset of a com-
  # -pany) for each of the total num of app (100) = raw applicant dataset
  # for C-z
  tmpAppDatasetConstructor <- matrix(NA, nrow = kTotalNumOfApp,
                                     ncol = kTotalNumOfFeat)

  for (i in 1:kTotalNumOfApp){
    temp <- runif(kTotalNumOfFeat, 0, 1)
    temp <- temp > (runif(1, 0, 1))
    tmpAppDatasetConstructor[i, ] <- temp
  }

  # converting raw appDataset for C-z from logical(TRUE/FALSE) to numeric (0/1)
  tmpAppDatasetConstructor <- apply(tmpAppDatasetConstructor, c(1, 2),
                                   as.integer)

  # defining row and column names for tmpAppDatasetConstructor
  tempRowNamesVec <- c(rep(NA, times = nrow(tmpAppDatasetConstructor)))
  for (p in 1:nrow(tmpAppDatasetConstructor)){
    tempRowNamesVec[p] <- sprintf("App-%i", p)
  }

  rownames(tmpAppDatasetConstructor) <- tempRowNamesVec
  colnames(tmpAppDatasetConstructor) <- c("hi sch deg?",
    "hi sch deg highest hns?", "clg deg in Econ/SocSc/CS?",
    "highest hns in any cleg deg?",
    "crsewrk in quant sub e.g Lin Alg, Calculus?",
    "strong grades (e.g >90%) in any qnt sub?", "grad sch deg?",
    "grad sch deg highest hns?", "R/MATLAB/STATA/SAS + fmlr?",
    "R/MATLAB/..+prty cmfrtble?",
    "gen purp prg lng (e.g. Python) + fmlr?",
    "gen purp prg lng (e.g. Python) + prty cmf?",
    "dtabse mng sys (e.g MySQL ..) + fmlr?",
    "dtabse mng sys (e.g MySQL ..) + pr cmf?", "Hadoop/Apache Spark+ fmlr?",
    "Hadoop/Apache Spark prty cmfrtable?",
    "exp estmtng eco'ric mod in prg lng's?", "exp in experimental design?",
    "Prev rsrch exp/ frml empl?", "rec any awards/fellowships/schlrships?")

  # logic correction in raw 'tmpAppDatasetConstructor'-----
  # each row of tmpAppDatasetConstructor contains randomly distributed 0's
```

*# and 1's. Given the colnames (features) these raw 0's and 1's will have
logical errors, each row in turn needs to be checked for possible
logical inconsistencies and be corrected if found.*

*# e.g. of logical mistake corrected: If rec hi sch deg with hns is TRUE(1)
then rec hi sc deg cannot be false(0). Similar to this all the column
numbers in 'colNumWithPotClash' has a potential for a clash with other
columns. Look at the colnames to learn more.*

```
colNumWithPotClash <- c(2, 4, 6, 8, 10, 12, 14, 20)
for (l in 1:nrow(tmpAppDatasetConstructor)){
  for (m in 1:length(colNumWithPotClash)){
    if (tmpAppDatasetConstructor[l, colNumWithPotClash[m]] == 1){
      if (tmpAppDatasetConstructor[l, (colNumWithPotClash[m] - 1)] == 0){
        tmpAppDatasetConstructor[l, (colNumWithPotClash[m] - 1)] <- 1
      }
    } else{
      next
    }
  }
  else{
    next
  }
}
```

*# another type of logical mistake corrected-----
Type of logical mistake corrected (or treat it as an assumption): If grad
sch deg rec =TRUE, then both cleg deg rec and hi sch deg rec must be TRUE*

```
for (n in 1:nrow(tmpAppDatasetConstructor)){
  if (tmpAppDatasetConstructor[n, 7] == 1){
    if ((tmpAppDatasetConstructor[n, 3]) == 1 &&
        (tmpAppDatasetConstructor[n, 1] == 1)){
      next
    }
  } else{
    tmpAppDatasetConstructor[n, 3] <- 1
    tmpAppDatasetConstructor[n, 1] <- 1
  }
}
else{
  if (tmpAppDatasetConstructor[n, 3] == 1){
    if (tmpAppDatasetConstructor[n, 1] == 0){
      tmpAppDatasetConstructor[n, 1] <- 1
    }
  } else{
    next
  }
}
else{
  next
}
```

```

}
}

# coerce from type 'matrix' to type 'data.frame', for ease in handling
tmpAppDatasetConstructor <- as.data.frame(tmpAppDatasetConstructor)

# label construction for 'tmpAppDatasetConstructor'-----
# labels construction for tmpAppDatasetConstructor using a randomly chosen
# criteria (out of a total of 10 criterions). Each criteria has a
# 'degree of selectivity' associated with it (scale ranging from 1 to 10),
# such that criteria-1 is most selective (10/10), criteria-2 (9/10) ...
# criteria-10 (1/10, least selective).

# Degree of selectivity is meant to associate a company with a particular
# selection criteria. This is done for simulation purposes, to generate
# labels for our applicant dataset. The purpose of this criterion system,
# is to try to simulate the real world differences in selection criteria,
# between companies and serve as a good representative of the total set
# of possible criterions out there.

# randomly choose a criteria number for C-z-----
criteriaNumber <- sample(c(1:kTotalNumOfCriterions), 1)

# initialize the labels column with NA's
labelsForTmpAppDataset <- as.data.frame(matrix(NA,
                                                nrow = nrow(tmpAppDatasetConstructor),
                                                ncol = 1))

# Almost all criterions below (total 10) have a general theme, which they
# use to automatically label training data for simulation purposes. So, for
# a particular training example (a single row in the applicant dataset for a
# particular company), a label of 1 is assigned if the number of "1's" in
# the row are greater or equal to a particular threshold. Next, a label of
# zero is assigned, if the number of "1's" in the row are less than equal to
# a particular threshold. For rest of the cases, a 'custom' decision making
# mechanism is employed, which if satisfied, then the row is labelled as '1'
# , else '0'. Think of '1' as representing 'presence of a quality', and of
# '0' as 'absence of a quality'.

switch(criteriaNumber,
  "1" = for (q in 1:nrow(tmpAppDatasetConstructor)){
    if (sum(tmpAppDatasetConstructor[q, ]) >= 18){
      labelsForTmpAppDataset[q, ] <- 1
    }
    else if (sum(tmpAppDatasetConstructor[q, ]) <= 14){
      labelsForTmpAppDataset[q, ] <- 0
    }
    else{
      if ((tmpAppDatasetConstructor[q, 2] == 1) &&
          (tmpAppDatasetConstructor[q, 4] == 1) &&
          (tmpAppDatasetConstructor[q, 6] == 1) &&
          (tmpAppDatasetConstructor[q, 10] == 1) &&
          (tmpAppDatasetConstructor[q, 19] == 1) &&

```

```

      (tmpAppDatasetConstructor[q, 12] == 1)){

        labelsForTmpAppDataset[q, ] <- 1
      }
      else{
        labelsForTmpAppDataset[q, ] <- 0
      }
    }
  },

  "2" = for (q in 1:nrow(tmpAppDatasetConstructor)){
    if (sum(tmpAppDatasetConstructor[q, ]) >= 17){
      labelsForTmpAppDataset[q, ] <- 1
    }
    else if (sum(tmpAppDatasetConstructor[q, ]) <= 13){
      labelsForTmpAppDataset[q, ] <- 0
    }
    else{
      if ((tmpAppDatasetConstructor[q, 2] == 1) &&
          (tmpAppDatasetConstructor[q, 4] == 1) &&
          (tmpAppDatasetConstructor[q, 6] == 1) &&
          (tmpAppDatasetConstructor[q, 10] == 1) &&
          (tmpAppDatasetConstructor[q, 12] == 1)){

        labelsForTmpAppDataset[q, ] <- 1
      }
      else{
        labelsForTmpAppDataset[q, ] <- 0
      }
    }
  },

  "3" = for (q in 1:nrow(tmpAppDatasetConstructor)){
    if (sum(tmpAppDatasetConstructor[q, ]) >= 16){
      labelsForTmpAppDataset[q, ] <- 1
    }
    else if (sum(tmpAppDatasetConstructor[q, ]) <= 12){
      labelsForTmpAppDataset[q, ] <- 0
    }
    else{
      if ((tmpAppDatasetConstructor[q, 2] == 1) &&
          (tmpAppDatasetConstructor[q, 4] == 1) &&
          (tmpAppDatasetConstructor[q, 6] == 1) &&
          (tmpAppDatasetConstructor[q, 10] == 1) &&
          (tmpAppDatasetConstructor[q, 11] == 1) &&
          (tmpAppDatasetConstructor[q, 14] == 1) &&
          (tmpAppDatasetConstructor[q, 15] == 1)){

        labelsForTmpAppDataset[q, ] <- 1
      }
      else{
        labelsForTmpAppDataset[q, ] <- 0
      }
    }
  }
}

```

```

    }
  },

  "4" = for (q in 1:nrow(tmpAppDatasetConstructor)){
    if (sum(tmpAppDatasetConstructor[q, ]) >= 15){
      labelsForTmpAppDataset[q, ] <- 1
    }
    else if (sum(tmpAppDatasetConstructor[q, ]) <= 11){
      labelsForTmpAppDataset[q, ] <- 0
    }
    else{
      if ((tmpAppDatasetConstructor[q, 2] == 1) &&
          (tmpAppDatasetConstructor[q, 4] == 1) &&
          (tmpAppDatasetConstructor[q, 6] == 1) &&
          (tmpAppDatasetConstructor[q, 10] == 1) &&
          (tmpAppDatasetConstructor[q, 11] == 1) &&
          (tmpAppDatasetConstructor[q, 14] == 1)){

        labelsForTmpAppDataset[q, ] <- 1
      }
      else{
        labelsForTmpAppDataset[q, ] <- 0
      }
    }
  },

  "5" = for (q in 1:nrow(tmpAppDatasetConstructor)){
    if (sum(tmpAppDatasetConstructor[q, ]) >= 14){
      labelsForTmpAppDataset[q, ] <- 1
    }
    else if (sum(tmpAppDatasetConstructor[q, ]) <= 10){
      labelsForTmpAppDataset[q, ] <- 0
    }
    else{
      if ((tmpAppDatasetConstructor[q, 2] == 1) &&
          (tmpAppDatasetConstructor[q, 4] == 1) &&
          (tmpAppDatasetConstructor[q, 6] == 1) &&
          (tmpAppDatasetConstructor[q, 10] == 1) &&
          (tmpAppDatasetConstructor[q, 11] == 1)){

        labelsForTmpAppDataset[q, ] <- 1
      }
      else{
        labelsForTmpAppDataset[q, ] <- 0
      }
    }
  },

  "6" = for (q in 1:nrow(tmpAppDatasetConstructor)){
    if (sum(tmpAppDatasetConstructor[q, ]) >= 13){
      if ((tmpAppDatasetConstructor[q, 4] == 1) &&
          (tmpAppDatasetConstructor[q, 6] == 1)){

```

```

        labelsForTmpAppDataset[q, ] <- 1
    }
    else{
        labelsForTmpAppDataset[q, ] <- 0
    }

}
else if (sum(tmpAppDatasetConstructor[q, ]) <= 9){
    labelsForTmpAppDataset[q, ] <- 0
}
else{
    if ((tmpAppDatasetConstructor[q, 4] == 1) &&
        (tmpAppDatasetConstructor[q, 6] == 1) &&
        (tmpAppDatasetConstructor[q, 9] == 1) &&
        (tmpAppDatasetConstructor[q, 17] == 1)){

        labelsForTmpAppDataset[q, ] <- 1
    }
    else{
        labelsForTmpAppDataset[q, ] <- 0
    }
}
},

"7" = for (q in 1:nrow(tmpAppDatasetConstructor)){
    if (sum(tmpAppDatasetConstructor[q, ]) >= 12){
        if ((tmpAppDatasetConstructor[q, 3] == 1) &&
            (tmpAppDatasetConstructor[q, 6] == 1) &&
            (tmpAppDatasetConstructor[q, 9] == 1)){

            labelsForTmpAppDataset[q, ] <- 1
        }
        else{
            labelsForTmpAppDataset[q, ] <- 0
        }
    }
    else if (sum(tmpAppDatasetConstructor[q, ]) <= 8){
        labelsForTmpAppDataset[q, ] <- 0
    }
    else{
        if ((tmpAppDatasetConstructor[q, 3] == 1) &&
            (tmpAppDatasetConstructor[q, 6] == 1) &&
            (tmpAppDatasetConstructor[q, 10] == 1)){

            labelsForTmpAppDataset[q, ] <- 1
        }
        else{
            labelsForTmpAppDataset[q, ] <- 0
        }
    }
}
},

```

```

"8" = for (q in 1:nrow(tmpAppDatasetConstructor)){
  if (sum(tmpAppDatasetConstructor[q, ]) >= 11){
    if ((tmpAppDatasetConstructor[q, 3] == 1) &&
        (tmpAppDatasetConstructor[q, 6] == 1) &&
        (tmpAppDatasetConstructor[q, 12] == 1)){

      labelsForTmpAppDataset[q, ] <- 1
    }
    else{
      labelsForTmpAppDataset[q, ] <- 0
    }
  }
  else if (sum(tmpAppDatasetConstructor[q, ]) <= 7){
    labelsForTmpAppDataset[q, ] <- 0
  }
  else{
    if ((tmpAppDatasetConstructor[q, 3] == 1) &&
        (tmpAppDatasetConstructor[q, 6] == 1) &&
        (tmpAppDatasetConstructor[q, 12] == 1) &&
        (tmpAppDatasetConstructor[q, 16] == 1)){

      labelsForTmpAppDataset[q, ] <- 1
    }
    else{
      labelsForTmpAppDataset[q, ] <- 0
    }
  }
},

"9" = for (q in 1:nrow(tmpAppDatasetConstructor)){
  if (sum(tmpAppDatasetConstructor[q, ]) >= 10){
    if ((tmpAppDatasetConstructor[q, 4] == 1) &&
        (tmpAppDatasetConstructor[q, 6] == 1) &&
        (tmpAppDatasetConstructor[q, 17] == 1)){

      labelsForTmpAppDataset[q, ] <- 1
    }
    else{
      labelsForTmpAppDataset[q, ] <- 0
    }
  }
  else if (sum(tmpAppDatasetConstructor[q, ]) <= 6){
    labelsForTmpAppDataset[q, ] <- 0
  }
  else{
    if ((tmpAppDatasetConstructor[q, 4] == 1) &&
        (tmpAppDatasetConstructor[q, 6] == 1) &&
        (tmpAppDatasetConstructor[q, 10] == 1) &&
        (tmpAppDatasetConstructor[q, 17] == 1)){

      labelsForTmpAppDataset[q, ] <- 1
    }
  }
}

```

```

    }
    else{
        labelsForTmpAppDataset[q, ] <- 0
    }
}
},

"10" = for (q in 1:nrow(tmpAppDatasetConstructor)){
    if (sum(tmpAppDatasetConstructor[q, ]) >= 9){
        if ((tmpAppDatasetConstructor[q, 3] == 1) &&
            (tmpAppDatasetConstructor[q, 6] == 1) &&
            (tmpAppDatasetConstructor[q, 12] == 1) &&
            (tmpAppDatasetConstructor[q, 16] == 1)){

            labelsForTmpAppDataset[q, ] <- 1
        }
        else{
            labelsForTmpAppDataset[q, ] <- 0
        }
    }
    else if (sum(tmpAppDatasetConstructor[q, ]) <= 5){
        labelsForTmpAppDataset[q, ] <- 0
    }
    else{
        if ((tmpAppDatasetConstructor[q, 3] == 1) &&
            (tmpAppDatasetConstructor[q, 6] == 1) &&
            (tmpAppDatasetConstructor[q, 12] == 1) &&
            (tmpAppDatasetConstructor[q, 14] == 1) &&
            (tmpAppDatasetConstructor[q, 16] == 1) &&
            (tmpAppDatasetConstructor[q, 19] == 1)){

            labelsForTmpAppDataset[q, ] <- 1
        }
        else{
            labelsForTmpAppDataset[q, ] <- 0
        }
    }
}
}) # switch statement ends here

# storing criteria number of C-z for later reference-----
criteriaNumberRecord[z] <- criteriaNumber

# stacking 'labels' column at the end of tmpAppDatasetConstructor-----
# It turns out that labels generated from the above process leads to
# 'perfect separation' in data. This happens because in all of the
# criterions we use some sort of thresholding (above which label is 1,
# below which label is 0) which leads to such an effect. So to deal with
# this issue we add regularization to our model via the 'caret' package
# using the method glmnet. This will help to reduce the extent of
# overfitting in the model and help to make the model more generalizable
# and also intuitive for simulation purposes.
tmpAppDatasetConstructor[, (kTotalNumOfFeat+1)] <- (labelsForTmpAppDataset

```



```

%>% as.data.frame()

colnames(tmpAppDatasetConstructor)[kTotalNumOfFeat+1] <- "labels"

# setting various controls for training of the model-----
indexTrainTmp <- createDataPartition(tmpAppDatasetConstructor$labels,
                                     p = 0.70, list = FALSE)
trainingSetTmp <- tmpAppDatasetConstructor[indexTrainTmp, ] # training set
testSetTmp <- tmpAppDatasetConstructor[-indexTrainTmp, ] # test set
controlsTmp <- trainControl(method = "repeatedcv", repeats = 5)

# setting up a search grid for k fold cross validation-----
# below search grid contains only one value each for 'alpha' and 'lambda',
# this is because I have already trained and tuned the model several times
# and the below values are the optimal ones as they performed best on avg
# in k fold cross validation. Note alpha = 1 (lasso penalty), alpha = 0,
# (ridge penalty). See ?glmnet for more details.
searchGridTmp <- expand.grid(alpha = 0.07206841, lambda = 0.03755844)

# training using caret package's 'glmnet' method-----
modelsList[[z]] <- train(x = trainingSetTmp[, 1:kTotalNumOfFeat],
                        y = as.factor(
                          trainingSetTmp[, (kTotalNumOfFeat+1)]),
                        method = "glmnet",
                        metric = "Accuracy",
                        trControl = controlsTmp,
                        tuneGrid = searchGridTmp
                      )

# constructing User1's applicant feature vector-----
# 'myAppFeatVec' would then be feeded to model M-z (corresponding to compa-
# -ny C-z) to get a label L-z, which will then be stored in
# 'myPersonalizedLabels' vector.
myAppFeatVec <- as.numeric(c(1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
                           1, 1, 0, 0, 1, 1, 0, 1, 0, 1))
myAppFeatVec <- as.data.frame(matrix(myAppFeatVec, 1, kTotalNumOfFeat))
colnames(myAppFeatVec) <- (colnames(tmpAppDatasetConstructor)[
  1:kTotalNumOfFeat])
lbl <- predict.train(modelsList[[z]], newdata = myAppFeatVec, type = "raw")
lbl <- as.numeric(lbl)
myPersonalizedLabels <- append(myPersonalizedLabels, lbl)

# store datasets for future reference-----
# transfer 'tmpAppDatasetConstructor' to appDatasetList[[z]] (corresponding
# to company C-z) and also storing in two seperate lists the corresponding
# train and test sets for each applicant dataset.
appDatasetList[[z]] <- tmpAppDatasetConstructor
appDatasetListTrainData[[z]] <- tmpAppDatasetConstructor[indexTrainTmp, ]
appDatasetListTestData[[z]] <- tmpAppDatasetConstructor[-indexTrainTmp, ]

# user interface-----
# if (z != kTotalNumOfAppDatasets){
#   print(sprintf(" Iteration %i/%i: Complete...", z, kTotalNumOfAppDatasets),

```

```

#           quote = FALSE)
# }elseif{
#   print(sprintf(" Iteration %i/%i: Complete", z, kTotalNumOfAppDatasets))
# }

} # top most for loop for Part 1 ends here.

# accuracy stats of the 100 models trained above-----
accuracyModelsList <- lapply(modelsList, function(x) x$results$Accuracy)
minAccuracyModelsList <- min(as.numeric(accuracyModelsList))
maxAccuracyModelsList <- max(as.numeric(accuracyModelsList))

# Part-1 ends here-----

```

Part-1: Code Discussion

Where do we stand?

Before beginning with **Part-2**, let's see where we are in the model building process. **Part-1** is complete. This means that we have constructed 100 models M_1 to M_{100} corresponding to companies C_1 to C_{100} and applicant datasets D_1 to D_{100} respectively.

These models are stored in a list called `modelsList`. We have also stored User-1's personalized labels (L_1 to L_{100}) in the vector `myPersonalizedLabels` which we constructed by passing User1's applicant feature vector named `myAppFeatVec` in turn through models (M_1 to M_{100}) respectively. These personalized labels will be used as the `labels` column for the `companyDataset` which we will construct in Part-2 of code.

Next, I will discuss in more detail a few chunks of code that needs more attention than that provided by the comments above:

Applicant Dataset: Structure

First, let's see the general structure of a typical applicant dataset for a company. Below, I display the first 4 rows and a few feature columns (I display 4 feature columns, but the dataset contains 20 feature columns) followed by a `labels` column for the first applicant dataset (`appDatasetList[[1]]`) corresponding to C_1 . Note that the `labels` column comes at the end (i.e. 21st column), but I display it here just after the first 4 columns for illustration purposes.

```

colNotToDisplay <- c(5:20)
knitr::kable(
  appDatasetList[[1]][1:4, -colNotToDisplay],
  caption = "Applicant Dataset: Structure"
)

```

Table 1: Applicant Dataset: Structure

	hi sch deg?	hi sch deg highest hns?	clg deg in Econ/SocSc/CS?	highest hns in any cleg deg?	labels
App-1	1	0	1	1	0
App-2	1	1	1	1	0
App-3	1	0	1	1	0
App-4	1	1	1	0	0

For those who have read the corresponding paper, please refer Figure 1 of the paper which displays the same structure as the table above. Each row of the applicant dataset contains information on a single applicant in form of a feature vector, followed by a label (0/1). This label tells us whether or not a particular applicant with some set of features got a job in C_1 . Together all of these rows form the applicant dataset for the company C_1 . Similar to the applicant dataset for C_1 (stored in `appDatasetList[[1]]`) there are applicant datasets for C_2 to C_{100} (stored in `appDatasetList[[2]]...appDatasetList[[100]]`).

Applicant Dataset: Column Names

Lets look at the column names for the applicant datasets in more detail.

```
print(colnames(appDatasetList[[1]]))
## [1] "hi sch deg?"
## [2] "hi sch deg highest hns?"
## [3] "clg deg in Econ/SocSc/CS?"
## [4] "highest hns in any cleg deg?"
## [5] "crsewrk in quant sub e.g Lin Alg, Calculus?"
## [6] "strong grades (e.g >90%) in any qnt sub?"
## [7] "grad sch deg?"
## [8] "grad sch deg highest hns?"
## [9] "R/MATLAB/STATA/SAS + fmlr?"
## [10] "R/MATLAB/..+prty cmfrtbl?"
## [11] "gen purp prg lng (e.g. Python) + fmlr?"
## [12] "gen purp prg lng (e.g. Python) + prty cmf?"
## [13] "dtabse mng sys (e.g MySQL ..) + fmlr?"
## [14] "dtabse mng sys (e.g MySQL ..) + pr cmf?"
## [15] "Hadoop/Apache Spark+ fmlr?"
## [16] "Hadoop/Apache Spark prty cmfrtable?"
## [17] "exp estmtng eco'ric mod in prg lng's?"
## [18] "exp in experimental design?"
## [19] "Prev rsrch exp/ frml empl?"
## [20] "rec any awards/fellowships/schlrships?"
## [21] "labels"
```

Each column (except the `labels` column) is to be interpreted as a **feature** that captures some information about the applicant. The `labels` column in the end captures one of two states; '1'- gets the job, '0'- does not get a job. Each feature is to be interpreted as a question which has one of two answers; yes(1)/ no(0).

I have abbreviated the features such that they can be easily reconstructed into full questions by the reader, e.g. feature [1] in the above list asks the following question: **Does the applicant have a high school degree?** whose answer would be either yes(1) or no(0) but not both. Similarly, another example is, feature [8] which asks the following question: **Does the applicant have a grad school degree with highest honors?** whose answer will be either yes(1) or no(0).

Please note that not all of the features may not neatly fit in (1/0) outcome state, but I have assumed that they do for simulation purposes. Also, when building a full fledged model (which we will not do here), in addition to these binary features, we can use much more interesting features using techniques like web scraping (I have discussed this in greater detail in the corresponding paper).

Applicant Dataset: Logic correction

In the code I perform logic correction on the applicant datasets. This is important because the **raw** applicant dataset contains in a given row randomly distributed 0's and 1's. But given the fact that we have specifically defined the columns for the applicant dataset, we cannot directly use the **raw** applicant dataset to carry out further analysis. We have to correct for the logical inconsistencies in the columns. For e.g. let's take the first row (Applicant-1) of the applicant Dataset corresponding to C_1 . So the applicant feature vector for Applicant-1 consists of 20 numbers (0's and 1's) each corresponding to a specific column. Now, without logic correction the AFV of Applicant-1 might have a 0 under column-1 (hi sch deg?), but 1 under column-2 (hi sch deg highest hns?) but that is not possible as if a person possesses a high school degree(with highest honors), than that implies that the person has a high school degree.

Similarly, I have corrected all possible potential clashes between columns of the applicant dataset. The vector `colNumWithPotClash` in the code contains all the columns that have a potential for clash with atleast one other column. I recommend that the reader now look at the 'logic correction' code once again (referencing the column names/number as and when needed) I hope it will make more sense now.

Applicant Dataset: Automatic Label Generation

In the code there is a big **switch** statement that is used to associate each company with a particular `criteriaNumber`(ranging from 1 to 10) whose purpose is to automatically label the applicant datasets for each company. For e.g. if `criteriaNumber` corresponding to C_1 is 1 than C_1 will refer to the "1" in the switch statement and use the instrutions to label the rows of its corresponding applicant dataset.

Choosing this "criterion" system helps in automating the labelling of the applicant datasets as manual labelling is not practical for simulation purposes. Another reason for choosing this system is to draw parallels with the real world hiring process. For e.g., we can think of having a 1 in each of the columns of the applicant datasets as possessing a quality and having a 0 as lacking a quality such that if an applicant has a lot of 1's it is highly likely that he will get a job and someone with a lot of 0's is likely to not get a job.

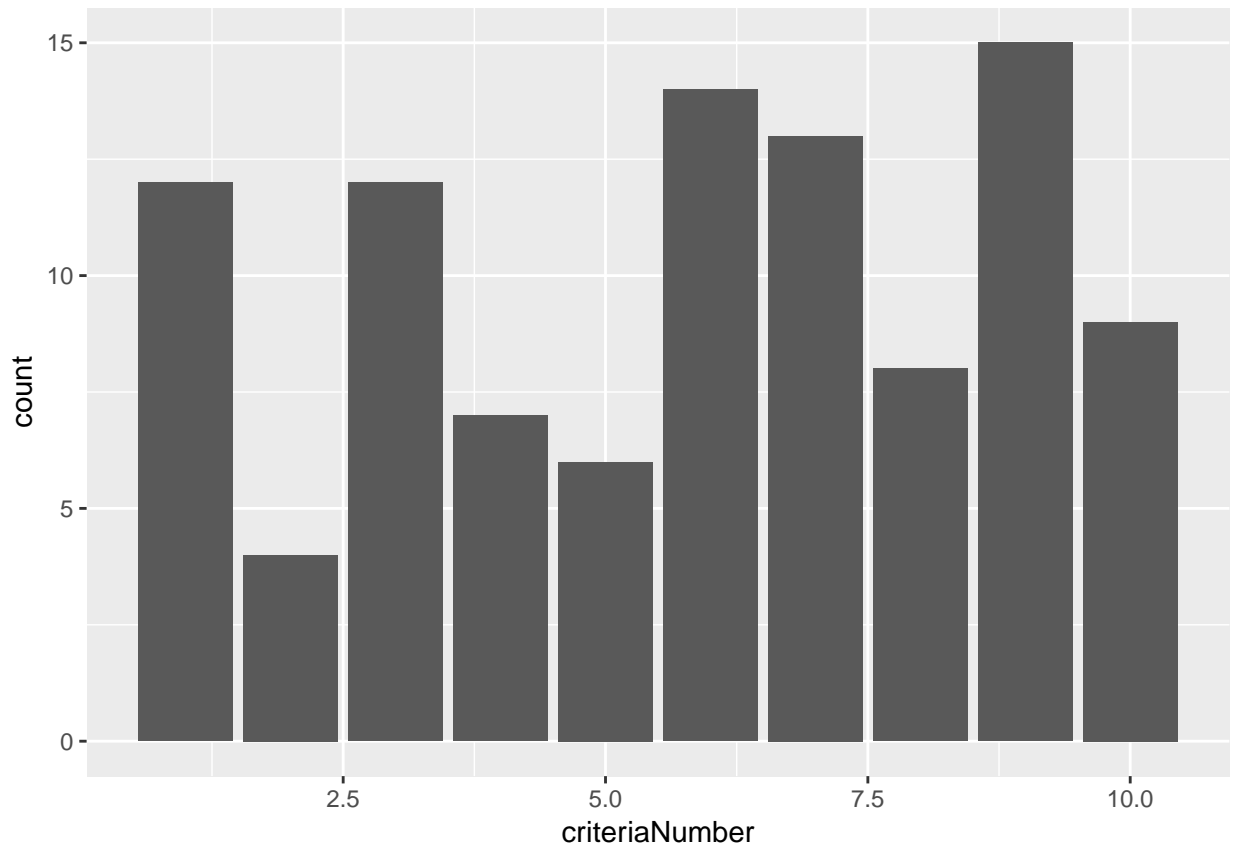
So the criterions above (ranging from 1 to 10) are a combination of this simple counting approach (e.g. if total number of 1's > 18; give the job) and besides that each criterion also utilizes a custom approach (e.g. a person with a more than eight 0's in his applicant feature vector will only get a job if he has a grad school degree with highest honors). The criterion system that I have set up can be 'roughly' thought of as decreasing in **degree of selectivity** as we move from 1 to 10 (See the code for individual criterions for more insights). For e.g. Criterion-1 can be thought of as the most strict and hence a company possessing it can be thought of as a company in which it is very hard to get in. Similarly Criterion-10 can be thought of as least strict.

The `criteriaNumberRecord` vector stores the `criteriaNumber` for each company. `criteriaNumberRecord[[z]]` represents the `criteriaNumber` corresponding to C_z . Let's look at the `criteriaNumberRecord` vector (Note that each company is assigned a `criteriaNumber` by randomly choosing a number b/w 1 to 10).

```
criteriaNumberRecord
##      [1]  8  7  3  5 10  1  7  3  2  9  7  3  3  9  6  5  6  9  3  9 10  7  7
##     [24]  1 10  1  8  2  9  1 10  2  4  3  1  7  9  6  6  4  5  7 10  6  1  6
##     [47]  6  4 10  4  8 10  5  1  2  8  3  9  3  9  1  4  7  9  5  6  6  7  8
##     [70]  8 10  9  9  8  6  4  3  4  5  9  9  1  3  7  1  7  3  7  3  8  7  6
##     [93]  6  1  6  6 10  1  9  9
```

and, here is the distribution of `criteriaNumberRecord`

```
tmpCritRecDataFrame <- as.data.frame(criteriaNumberRecord)
colnames(tmpCritRecDataFrame) <- "criteriaNumber"
ggplot(data = tmpCritRecDataFrame) + geom_bar(mapping = aes(x = criteriaNumber))
```



Model Output

Now, let's look at the output of a typical model. `modelsList[[z]]` stores the trained model corresponding to C_z . Let's see the output for `modelsList[[1]]`,

```
modelsList[[1]]
## glmnet
##
## 70 samples
## 20 predictors
## 2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 62, 63, 63, 63, 63, 63, ...
## Resampling results:
##
##   Accuracy   Kappa
## 0.9536905 0.9047867
```

```
##
## Tuning parameter 'alpha' was held constant at a value of 0.07206841
##
## Tuning parameter 'lambda' was held constant at a value of 0.03755844
```

The output contains useful summary which can be read as follows, the model was trained using the **glmnet** method in the **caret** package. The model was trained using 70 samples (in the code above I allotted 70% of the total rows = 100 to training set). Within these 70 samples the underlying caret package uses the **10 fold repeated cross validation** as its resampling method and repeat it 5 times for all the parameter settings in the search Grid. The accuracy of the model is which means that the final model chosen ($\alpha = 0.0720684$, $\lambda = 0.0375584$) correctly predicted the label 95% of the time on the cross validation set (this accuracy is an average over the total number of resampling attempts, which in the above case is 50, 10 folds repeated 5 times).

Please note that in the code above I have not set up a search grid for searching of parameters because I already did that while I was training and tuning the model and found the above values for **alpha** and **lambda** to be the best amongst the lot. You can set up a search Grid for yourself using the `expand.grid()` function if you would like to see it work for yourself.

User1 Information: AFV, Personalized Labels

To wrap up Part-1 lets look at the applicant feature vector and the personalized labels of User1,

```
matrix(myAppFeatVec, 1, 20)
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## [1,]  1    1    1    1    1    1    1    0    0    0     1     1     0
##      [,14] [,15] [,16] [,17] [,18] [,19] [,20]
## [1,]  0     1     1     0     1     0     1
```

[,1] to [,20] correspond to the [1] to [20] features mentioned above. Just like any other applicant User1 also has a feature vector which is represented above. Because we are building a model for User1 we will only focus on his AFV. In Part-1 we construct the `myPersonalizedLabels` vector one element at a time (adding one element to the end in each iteration of the main (top-most) for loop of the Part-1).

For example, to get `myPersonalizedLabels[z]` we pass User1's AFV through M_z such that the `myPersonalizedLabels[z]` column inform User1 whether or not he got a job in company C_z (as M_z is the model corresponding to company C_z). So in total User1's `myPersonalizedLabels` vector will have 100 elements, where element the z 'th element is the User1's label for C_z . The `myPersonalizedLabels` column will be used as the labels column of the `companyDataset` in Part-2 of the code.

The `myPersonalizedLabels` column for User1 is shown below,

```
matrix(myPersonalizedLabels,1,100)
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## [1,]    2    1    1    2    1    1    2    1    1    1    1    1    1
##      [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24]
## [1,]    1    2    1    1    1    1    1    1    2    1    1    1
##      [,25] [,26] [,27] [,28] [,29] [,30] [,31] [,32] [,33] [,34] [,35]
## [1,]    2    1    2    1    1    1    1    1    1    1    1
##      [,36] [,37] [,38] [,39] [,40] [,41] [,42] [,43] [,44] [,45] [,46]
```

```
## [1,]      1      1      2      2      2      1      1      2      2      1      2
##      [,47] [,48] [,49] [,50] [,51] [,52] [,53] [,54] [,55] [,56] [,57]
## [1,]      1      2      1      2      2      2      1      1      1      2      1
##      [,58] [,59] [,60] [,61] [,62] [,63] [,64] [,65] [,66] [,67] [,68]
## [1,]      1      1      1      1      2      1      1      1      2      2      1
##      [,69] [,70] [,71] [,72] [,73] [,74] [,75] [,76] [,77] [,78] [,79]
## [1,]      2      1      1      1      1      1      2      1      1      1      2
##      [,80] [,81] [,82] [,83] [,84] [,85] [,86] [,87] [,88] [,89] [,90]
## [1,]      1      1      1      1      1      1      1      1      1      1      2
##      [,91] [,92] [,93] [,94] [,95] [,96] [,97] [,98] [,99] [,100]
## [1,]      1      1      1      1      2      1      2      1      1      1
```

I hope that now there is a strong foundation in what Part-1 of the model is doing. Let's go on to **Part-2** of the model.

Part-2: Code

In Part-2 of the code we will build the `companyDataset` which will then be used to build the **Personalized Job Classifier Model**.

Let's first have a look at the entire code for Part-2 and then we will discuss specific chunks of code in more detail.

```
# user interface-----
# print("-----", quote = FALSE)
# print(">> Simulation Part-2", quote = FALSE)
# print("      Running...", quote = FALSE)

# initializing companyDataset matrix (sans labels)-----
kTotalNumOfComp <- kTotalNumOfAppDatasets
companyDataset <- matrix(NA, kTotalNumOfComp, kTotalNumOfFeat)

# filling company dataset-----
# Generate 20 random numbers (single row in the company dataset or in other
# words a feature vector for a single company) for each of 100 companies.
# This will give us the 'raw' companyDataset.
companyDataset <- apply(companyDataset, 1,
                        function(x) x <- runif(kTotalNumOfFeat, 0, 1))
companyDataset <- apply(companyDataset, 1, function(x) x <- (x > runif(1, 0, 1)))

# converting company dataset from logical(T/F) to numeric(0/1)-----
companyDataset <- apply(companyDataset, c(1, 2), as.integer)

# constructing rownames for company dataset-----
tempRowNamesVec <- c(rep(NA, times = kTotalNumOfComp))
for (b in 1:nrow(companyDataset)){
  tempRowNamesVec[b] <- sprintf("Company-%i", b)
}

# assigning names to rows and columns of the companyDataset-----
```

```

rownames(companyDataset) <- tempRowNamesVec
colnames(companyDataset) <- c("emplyrUniversity?", "emplyrPrivateComp?",
"emplyrNGO?", "emplyrListedOnAStockExchange?", "emplyrWorkTypeResearch?",
"emplyrISO9001OrOtherEqvlemtCompliant?", "emplyrLessThan1YrOldInItsSec?",
"emplyrGreaterThanOrEqualTo10YrsOldInItsSec?", "NumOfEmployeesLessThan100?",
"NumOfEmployeesGreaterThanOrEqualTo1000?", "emplyrCountAsOneOfTop50InItsSector?",
"emplyrInternationallyRecognizedViaItsWork?",
"emplyrInternationalPhysicalPresence?", "emplyrRecipientOfAwardsInRecogOfWork?",
"emplyrHireOnlyLocallyLessCulturalDiversity?",
"emplyrPerformPoorOnSocialRespIndexLikeCSR?",
"emplyrHasSignificantOnlinePresence?", "emplyrSignifSpendingInRandD?",
"emplyrSignifSpendOnGrowingTechLikeAI?", "emplyrAnyControversialHistory?")

# logic corrections in the company Dataset-----
# I will correct 1 mistake at a time for better interpretability of the code.
# This comes at the cost of using more for loops then needed.

# type of logic correction: employer for a single row in the company dataset
# can be only 'one' of the various types (see column names above for more info)
for (v in 1:nrow(companyDataset)){
  if (sum(companyDataset[v, 1], companyDataset[v, 2],
    companyDataset[v, 3]) != 1){
    if (sum(companyDataset[v, 1], companyDataset[v, 2],
      companyDataset[v, 3]) == 0){
      next
    }
    else{
      tmpVar <- sample(c(1, 2, 3), 3)
      companyDataset[v, tmpVar[1]] <- 1
      companyDataset[v, tmpVar[2]] <- 0
      companyDataset[v, tmpVar[3]] <- 0
    }
  }
  else{
    next
  }
}

# type of logic correction: If 'emplyr listed on a stock exchange == TRUE',
# then emplyr cannot be a University OR NGO.
for (w in 1:nrow(companyDataset)){
  if (companyDataset[w, 4] == 1){
    if ((companyDataset[w, 1] == 1) || (companyDataset[w, 3] == 1)){
      companyDataset[w, 1] <- 0
      companyDataset[w, 3] <- 0
    }
    else{
      next
    }
  }
  else{
    next
  }
}

```



```

}
}

# type of logic correction: employer cannot be both (less than 1yr old) AND
# (greater than 10 yrs old)
for (x in 1:nrow(companyDataset)){
  if ((companyDataset[x, 7] == 1) && (companyDataset[x, 8] == 1)){
    tmpVar <- sample(c(7, 8), 2)
    companyDataset[x, tmpVar[1]] <- 1
    companyDataset[x, tmpVar[2]] <- 0
  }
  else{
    next
  }
}

# type of logical correction: If employer has international physical presence,
# then, it is internationally recognized via its work.
for (y in 1:nrow(companyDataset)){
  if (companyDataset[y, 13] == 1){
    if (companyDataset[y, 12] == 0){
      companyDataset[y, 12] <- 1
    }
    else{
      next
    }
  }
  else{
    next
  }
}

# type of logical mistake corrected: total number of employees cannot be both
# (less than 100) AND (greater than 1000)
for (a in 1:nrow(companyDataset)){
  if ((companyDataset[a, 9] == 1) && (companyDataset[a, 10] == 1)){
    tmpVar <- sample(c(9, 10), 2)
    companyDataset[a, tmpVar[1]] <- 1
    companyDataset[a, tmpVar[2]] <- 0
  }
  else{
    next
  }
}

# coercing company dataset into type data.frame-----
companyDataset <- as.data.frame(companyDataset)

# stacking 'myPersonalizedLabels' (generated in 'Part-1') last column of the
# company dataset.
companyDataset[, (ncol(companyDataset)+1)] <- (myPersonalizedLabels %>%
  as.data.frame())
colnames(companyDataset)[kTotalNumOfFeat+1] <- "labels"

```

```

# setting various controls for training of the model-----
indexTrain <- createDataPartition(companyDataset$labels, p = 0.70,
                                  list = FALSE)
trainingSet <- companyDataset[indexTrain, ]
testSet <- companyDataset[-indexTrain, ]
controls <- trainControl(method = "repeatedcv",
                         repeats = 5)

# setting up a search grid for k fold cross validation-----
# below search grid contains only one value each for 'alpha' and 'lambda',
# this is because I have already trained and tuned the model several times
# and the below values are the optimal ones as they performed best on avg
# in k fold cross validation. Note alpha = 1 (lasso penalty), alpha = 0,
# (ridge penalty). See ?glmnet for more details.

searchGrid <- expand.grid(alpha = 0.01385693, lambda = 1.720017)

# user interface-----
# print("      Training Personalized Job Classifier...", quote = FALSE)

# training PersonalizedJobClassifier using caret package's 'glmnet' method-----
personalizedJobClassifier <- train(x = trainingSet[, 1:kTotalNumOfFeat],
                                  y = as.factor(
                                    trainingSet[, (kTotalNumOfFeat+1)]),
                                  method = "glmnet",
                                  metric = "Accuracy",
                                  trControl = controls,
                                  tuneGrid = searchGrid
                                  )

# total time elapsed-----
elapsedTime <- proc.time() - startTime
elapsedTime <- as.vector(elapsedTime)

# user interface-----
# print(">> Done!", quote = FALSE)
# print(sprintf(">> Elapsed Time (in min): %f", ((elapsedTime[3])/60)),
#        quote = FALSE)

# end of code-----

```

Part-2: Code Discussion

Where do we stand?

We have completed the model. The model is saved in the object named `personalizedJobClassifier`. User1 can now use this model by feeding it a company feature vector for a **new** company (i.e. a company that was not included in the training and/or test sets) and the model will output a label that will inform User1 of his job prospects(0/1) in the company. Next, let's discuss a few specific code chunks of Part-2 that need more attention than that provided in the comments above.

Company Dataset: Structure

First, let's see the general structure of the company dataset. Below, I display the first 5 rows and a few feature columns (I display 4 feature columns, but the company dataset contains 20 feature columns) followed by a `labels` column. Note that the `labels` column comes at the end (i.e. 21st column), but I display it here just after the first 4 columns for illustration purposes. Also, the caret package denotes the labels as 1 and 2 which corresponds to 0 and 1 respectively. In other words, it does not matter how you represent a binary state (0/1, 1/2, 3/4...), the underlying idea is the same which is that the labels column is a binary column.

```
colNotToDisplay <- c(5:20)
knitr::kable(
  companyDataset[1:5, -colNotToDisplay],
  caption = "Company Dataset: Structure")
```

Table 2: Company Dataset: Structure

	emplyrUniversity?	emplyrPrivateComp?	emplyrNGO?	emplyrListedOnAStockExchange?	labels
Company-1	0	0	0	1	2
Company-2	1	0	0	0	1
Company-3	0	0	1	0	1
Company-4	0	0	0	1	2
Company-5	1	0	0	0	1

For those who have read the corresponding paper, please refer Figure 3 of the paper which displays the same structure as the table above. I will describe the company dataset in words so as to get a better intuition of its structure and purpose.

What does row₁ in company dataset represent?: First comes the Company Name (C_1), that is followed by a 'n' dimensional company feature vector (V_1), which defines C_1 in terms of 'n' features (just like the applicant feature vector described the applicant in terms of a feature vector) after that comes a label (L_1), this label L_1 came from **Part-1** where we ran User1's applicant feature vector through the model M_1 (remember, M_1 took as its input an applicant feature vector and outputted whether or not that applicant got the job in company C_1) which outputs a label which is stored in `myPersonalizedLabels[1]` (let's assume that label was 1, i.e. User1 got the job in C_1). So, an informal way to read row-1 of the table is as follows, *As per M_1 , User1 got the job in company C_1 which has features V_1 .*

We will follow a similar procedure for row₂ to row₁₀₀, while keeping in mind that just like L_1 , each label L_2 to L_n comes from running User1's applicant feature vector in turn through models M_2 to M_n (this is what makes the model personalized to User1, as all labels L_1 to L_n in our entire `companydataset` correspond to User1). A detailed discussion of this can be found in the corresponding paper (section III).

Company Dataset: Column Names

Let's look at the column names of the company Dataset in more detail,

```
colnames(companyDataset)
## [1] "emplyrUniversity?"
## [2] "emplyrPrivateComp?"
## [3] "emplyrNGO?"
## [4] "emplyrListedOnAStockExchange?"
```

```
## [5] "emplyrWorkTypeResearch?"
## [6] "emplyrISO9001orOtherEqvleantCompliant?"
## [7] "emplyrLessThan1YrOldInItsSec?"
## [8] "emplyrGreaterThan10YrsOldInItsSec?"
## [9] "NumOfEmployeesLessThan100?"
## [10] "NumOfEmployeesGreaterThan1000?"
## [11] "emplyrCountAsOneOfTop50InItsSector?"
## [12] "emplyrInternationallyRecognizedViaItsWork?"
## [13] "emplyrInternationalPhysicalPresence?"
## [14] "emplyrRecipientOfAwardsInRecogOfWork?"
## [15] "emplyrHireOnlyLocallyLessCulturalDiversity?"
## [16] "emplyrPerformPoorOnSocialRespIndexLikeCSR?"
## [17] "emplyrHasSignificantOnlinePresence?"
## [18] "emplyrSignifSpendingInRandD?"
## [19] "emplyrSignifSpendOnGrowingTechLikeAI?"
## [20] "emplyrAnyControversialHistory?"
## [21] "labels"
```

Each column (except the `labels` column) is to be interpreted as a **feature** that captures some information about the company. The `labels` column in the end captures one of two states; '1'- gets the job, '0'- does not get a job. Each feature is to be interpreted as a question which has one of two answers; yes(1)/no(0).

I have abbreviated the features such that they can be easily reconstructed into full questions by the reader. The manner in which we interpret these columns is very similar to what we did in **Part-1** for the applicant dataset with the difference that now each column captures information (in form of 0/1) for a **company** not an **applicant**. For example `emplyrUniversity?` translates into the question **Is the employer a University?**, `emplyrInternationalPhysicalPresence?` translates into **Does the employer have international physical presence?**. I leave the rest for the reader to explore.

Please note that not all of the features may not neatly fit in (1/0) outcome state, but I have assumed that they do for simulation purposes. Also, when building a full fledged model (which we will not do here), in addition to these binary features, we can use much more interesting features using techniques like web scraping (I have discussed this in greater detail in the corresponding paper).

Personalized Job Classifier Model

Now, let's look at the output of our final model i.e. **Personalized Job Classifier Model**,

```
personalizedJobClassifier
## glmnet
##
## 70 samples
## 20 predictors
## 2 classes: '1', '2'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 63, 62, 63, 62, 63, 63, ...
## Resampling results:
##
##   Accuracy   Kappa
## 0.745      0
```

```
##  
## Tuning parameter 'alpha' was held constant at a value of 0.01385693  
##  
## Tuning parameter 'lambda' was held constant at a value of 1.720017
```

The accuracy of the Personalized Job Classifier Model crosses the 70% mark. This represents an average accuracy over 10 fold cross validation repeated 5 times. This means that, on average, 70 out of 100 times our model (with $\alpha = 0.0138569$, $\lambda = 1.720017$) correctly predicts the label when feeded a company feature vector. Again the α and λ values are trained and tuned by me. The reader can set up a search Grid using `expand.grid()` function if they want to re tune the model.

Part-2: Conclusion

Once trained and tuned, the model serves as a **Personalized Job Classifier** for User1, which means that the model can now take any **new** company that it has not seen before, distill that company's characteristics into a company feature vector, take that vector as its input and output a label (0/1) which will inform User1 (with certain probability, whether or not he gets a job in that new company. Also, User1 can do this for any number of new companies, all he has to do is enter as an input to the model the company feature vector of the company he is interested and get a label as an output that will inform him of his job prospects in that company.

The model is complete. Finally, I will wrap up this presentation by discussing some important topics concerning the performance of the model as a whole.

Wrap up

Notes on reproducibility

In many lines in the above code random numbers were generated and utilized. In particular during the training of the model, the underlying method (in our case, `method = glmnet`) utilizes random numbers during the phase of parameter estimation. Also, the resampling indices are chosen using random numbers. Given that, it is important to control randomness to assure reproducibility of results.

To serve that purpose, I have used `set.seed()` function in the beginning of the code. But it turns out that there are a few subtleties to that. When we open a new R session, Initially, there is no seed; a new one is created from the current time and the process ID when one is required. Hence different sessions will give different simulation results, by default. So this means that **within** a session you will get the exact same results, but when **switching to a new session** the results will differ by a small amount (e.g. accuracy may slightly go up or down). So, this means that there is **Intra-Session** reproducibility but not **Inter-Session** reproducibility.

It turns out that we can mitigate this issue by using `.Random.seed` which is an integer vector, containing the random number generator (RNG) state for random number generation in R. This can be saved and restored in different sessions. It seems that it solves the problem of **Inter-Session** reproducibility and it does in most cases. More information on `.Random.seed` can be found [here](#).

But, there is yet another case (our case) in which even `.Random.seed` might not help. This is the case when there is a function like the `train()` function of the **caret** package which does a lot of random number generation in form of resampling. Moreover, the `train()` function is a very general function in the sense that it has functionality for incorporating in its `method` argument 'one' of many packages (in my model above I used the **glmnet** package). How random numbers are generated is highly dependent on the package author such that even after using `.Random.seed` we might not get the exact same results due to a potential clashes with the scheme that the package author has used. You can further read about this concern [here](#) where the concern is described in the specific context of the **caret** package's `train()` function.

What does this all mean for the model above? To minimize the variance in **Inter-Session** reproducibility, I have tried and tested the model enough times such that for the final model the accuracy (given the tuned parameters) is always above 70%. There are some more ways in the **caret** package to avoid this extra manual labor that I have done which involves setting the seeds by hand for all the resampling iterations in the `trainControl()` function. I have not done that here. But, more information on that can be found [here](#).

Final bits of advice

- The entire model takes 5-8 min to train (depending on your computer), with **Part-1** taking the majority of the time. As you run the individual chunks of code (See **Run the code yourself: Instructions** section for more information) the progress will be printed on the console for your reference. So, do keep the console window open.
- I have used 10 fold repeated cross validation (`repeats = 5`) in the above models. I could have chosen less number of repeats but I didn't so as to ensure that average (over the resamples) measure of accuracy I get as the output of my model is more reliable. Please note that decreasing the number of repeats will significantly reduce the time it takes to train the model.
- Please make sure that you have all the required packages needed for the model. Also make sure that you have the latest version updates. The list of all the packages that were used to build this model can be found under the **Packages and Version Information** section.

Run the Code yourself: Instructions

Below are simple instructions on how to run the code presented in this file in RStudio:

1. Download the **.Rmd** (R Markdown) file from the **File Downloads** section below.
2. Open the file in R Studio.
3. Run the code. Proceed at your own pace by starting from the beginning and running the code chunks (by clicking ">" button in the chunk). The results are displayed right below the code (for e.g. plots, figures, etc.) and all other variables can be retrieved from the current environment and manipulated using the console. All code is contained in R code chunks which have the following structure,

```
## [1] ```{r chunkName, option1, option2,...}
## [1] code line 1...
## [1] code line 2...
## [1] ...
## [1] ```
```

Note: Running the code chunk by chunk retains the output produced in the current environment for you to manipulate. So, if you choose to do that all is good, no worries.

But, if you want to **knit** the **.Rmd** file into a PDF all in one go, no variables will be retained in the environment for you to manipulate once the knit is complete. So, if you want to knit the **.Rmd** file into a PDF **AND** retain the variables in the environment, then you have to include an additional code chunk at the end of the **.Rmd** file that uses the `save.image()` function to save the current workspace into a **.RData** file which can be reloaded later for data manipulation. Below, in the section **Save Current Workspace** I have included code that will save all of your workspace in a file named **restoreWorkspace.RData** which will reside in your current working directory (type `getwd()` and hit enter in your R Studio console to see your current working directory).

Please remove `eval = FALSE` from the code in the section **Save Current Workspace** in order to activate the code chunk, in case you want to knit this document. Next, to load the `restoreWorkspace.RData` that you have just saved into the environment use the `load()` function.

File Downloads

- Click [here](#) to download the **.Rmd** file which you can yourself run in R studio to see the code work for yourself

Useful Links

- The model is built using the **caret** package, a short and precise introduction to the package can be found [here](#) (approximate time required to read: 10 minutes).
- The code is styled (with a few tweaks) as per the Google R style guide. Click [here](#) to view the style guide.

Elapsed Time

In case you are wondering how much time it took to **knit** this R markdown file into a PDF, It took,

```
cat(elapsedTime[3]/60, "minutes")  
## 7.338833 minutes
```

Save Current Workspace

```
restoreWorkspace <- save.image()  
save(restoreWorkspace, file = "restoreWorkspace.RData")
```

End_____