Question 3.1 (a)
Using the same data set (credit_card_data.txt or credit_card_data-headers.txt) as in Question 2.2, use the `ksvm` or `kknn` function to find a good classifier: using cross-validation (do this for the k-nearest-neighbors model; SVM is optional)

(Please note that my R code for this question is contained in the file 'homework_2_Q3.1a.R', and if you wish to run the code, you will need to change line 8 of my R code to your local directory which contains the credit card application data, referred to in this document as *cc_data*.)

Response
It is a good idea to implement a model validation technique such as cross-validation when assessing the results of a model. When a model is trained and then tested on the same set of data, the predicted accuracy of the model will generally be greater than the true performance (accuracy) of the model when it is applied to independent datasets. k-fold cross-validation is one method which can be used to mitigate this problem. Cross-validation can tell you how a model will generalize to an independent set of data and can provide a good estimation of the true performance of the model as it is applied to independent data sets. Cross-validation has the benefit over other validation techniques of being able to be applied to data sets which are too small to be permanently partitioned into separate training and validation sets without losing modeling capability. A detailed explanation of the inner workings of the k-fold cross-validation algorithm is provided below.

In my solution to the problem of finding a good classifier for the credit card approval data, I first read in the credit card approval data and saved it in a matrix called *cc_data*. I used k-fold cross-validation to test the performance of the k-nearest neighbors algorithm as a method of finding a classifier for the problem, and I tested all numbers of neighbors *k* in the set *k = {1, 2, 3, ... ,100}*, where *k* is defined as the number of neighbors.

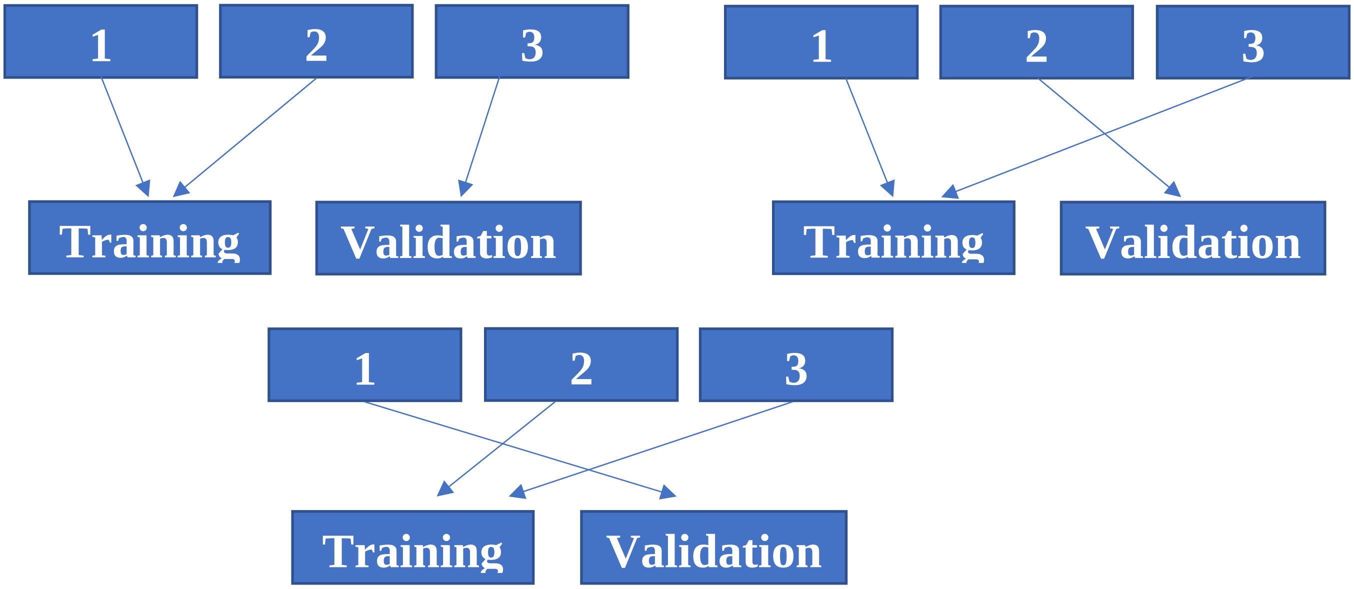
To begin, I created a function called *cc_kknn_function. cc_kknn_function* utilizes R's *kknn* (k nearest neighbors) function, and is designed specifically to work with *cc_data*. *cc_kknn_function* takes in two parameters called *val_data* and *test_data*. *val_data* and *test_data* are both created from subsets of the data points (rows) of *cc*_data. The precise method of creation and required specifications of the parameters *val_data* and *train_data* are described in detail below in the discussion on the implementation of k-fold cross-validation to determine the optimal parameters in the k-nearest neighbors model.

Within the function *cc_kknn*_function, nested for loops are used to generate *100* different k nearest neighbor models using R's *kknn* function, for every number of neighbors *k* in the set *k = {1, 2, 3, ... ,100}*. These *100* models are trained using the input parameter *train_data*. Next, each of the *100* models are applied to the *10* independent variables in each of the data points in the input parameter *val_data*. Finally, the accuracy of each of the *100* models is determined by comparing the predicted results with the actual results (the dependent variable in *val_data*). Using two separate sets of data to train and then validate the accuracy of a model is a critical part of k-fold cross-validation, as will be described in detail later. The percentage of correct results for each of the tested values of *k* is returned by *cc_kknn_function* in the form of a matrix in which each row in the first column represents the value of *k* (number of neighbors) used in creating a k-nearest neighbors model, and each row in the second column represents the corresponding accuracy of the model.

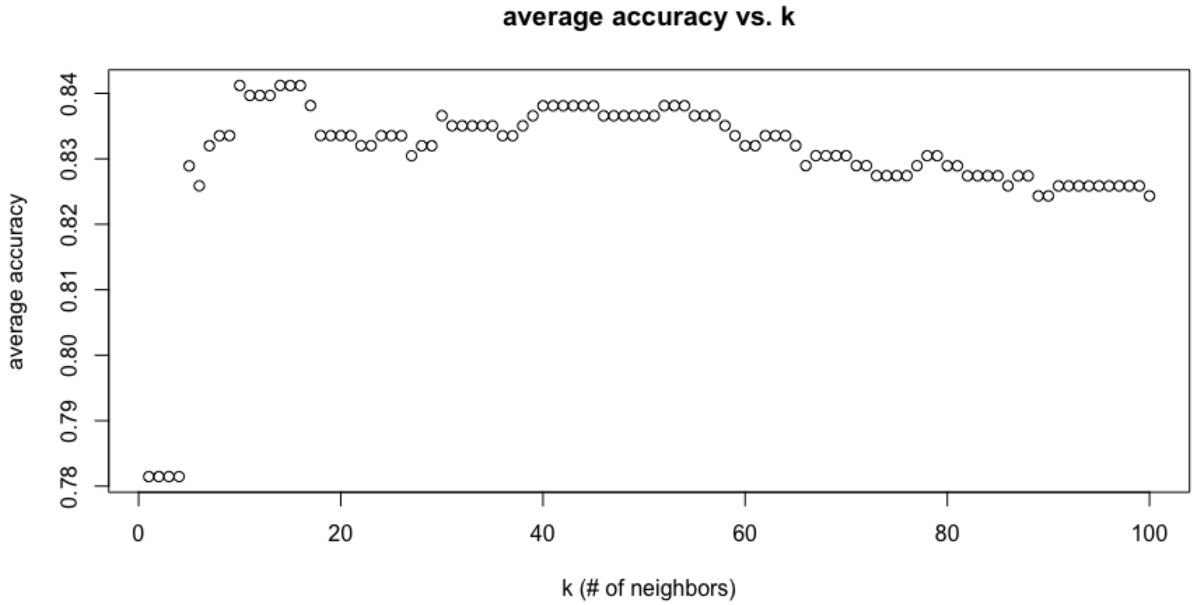
It should be pointed out that in the *kknn* function it is specified that each data point *i* being tested is specifically excluded from being counted as one of the "neighbors", so that the predicted outcome will not be tainted by self-counting. Also, the R function *kknn* returns the fraction of k nearest neighbors that are equal to *1*. Since *kknn* returns a continuous a response, and we are attempting to generate a binary classification, the output is rounded to the nearest whole number (either *0* or *1*) in order to be able make a prediction and then verify the validity of the prediction. Additionally, data scaling was taken care of within the *ksvm* function, by setting *scaled = TRUE*.

After creating the function *cc_kknn_function* I next created a function called *kknn_cross_validate*. *kknn_cross_validate* uses k-fold cross-validation in order to find a good classifier for *cc_data* by applying the technique of k-fold cross-validation to the function *cc_kknn_function* (which implements R's kknn function). *kknn_cross_validate* calls the function *cc_kknn_function* in an iterative manner, and more importantly generates the inputs of *cc_kknn_function* in accordance with the definition of k-fold cross-validation in each iteration. The function *kknn_cross_validate* takes in two parameters called *input_data* and *num_folds*. In the context of this paper, *input_data* will always be assumed to *cc_data*. *num_folds* is an integer, and is the number of equal length complementary sets which *cc_data* will be split into, in order to implement k-fold cross-validation.

kknn_cross_validate implements k-fold cross-validation in the following manner: The input parameter *input_data* (*cc_data*) is randomly shuffled and then split into equal length complementary sets. As mentioned in the previous paragraph, the number of equal sets which *input_data* is partioned into is equal to the integer *num_folds*. *num_folds* is commonly referred to as *k* in the context of k-fold cross-validation, but is instead referred to here as *num_folds*, so that it is not confused with the definition of *k* above, in which *k* is used to designate the number of neighbors considered in the k-nearest neighbors algorithm. *kknn_cross_validate* calls the function *cc_kknn_function* *num_folds* number of times. Each time *cc_kknn_function* is called, the model created is applied to a different subset of *cc_data* in order to measure the accuracy of the k-nearest neighbor models created by the function *kknn*. This set of data is referred to as the *validation* portion of the data. All of the remaining sets of *cc_data* are used to train the k-nearest neighbors model. This portion of the data is therefore referred to as the *training* portion of the data. Each subset of the data will therefore be used *num_folds* – 1 times for training, and 1 time for validation. This may be visualized in the case of *num_folds = 3* in the figure below:



Each time *cc_kknn_function* is called within *kknn_cross_validate*, the output of *cc_kknn_function* is recorded and stored in a variable called *cumulative_average_accuracies*. As mentioned above, the percentage of correct results for each of the tested values of *k* (as applied to *val_data*) is returned by *cc_kknn_function* in the form of a matrix in which each row in the first column represents the value of *k* (number of neighbors) used in creating a k-nearest neighbors model, and each row in the second column represents the corresponding accuracy of the model. The row-wise mean of *cumulative_average_accuracies* is found for every number of neighbors *k* tested, where *k* is the set *k = {1, 2, 3, ... ,100}*. These average accuracies for each value of *k* may be observed in the following figure:



It should be noted that if you choose to run my code, your results will likely be slightly different from mine, since the R random number generator called *sample* is used in the partitioning of the validation and training sets which are used in the k-fold cross-validation process.

As may be observed in the figure, for every value of *k* in the set *{1, 2, 3, … ,100*}, the average model accuracy produced through the process of k-fold cross-validation process was between *.78* and *.85*. Thus, we can see that there is not a significant variance in the accuracy of each model for each given value of *k* in the set *{1, 2, 3, … ,100.*

In order to determine the optimal value of *k* for use in our model, I found the value(s) of *k* which resulted in the highest accuracy. This resulted in the determination (for this particular implementation of k-fold cross-validation) that a maximum average prediction accuracy of *0.8412016* occurs when *k = 10.*

In my implementation of applying k-fold cross-validation to the task of finding a good classifier for *cc_data* I chose to let number partitions, *num_folds*, be equal to *4.* I chose to set *num_folds* equal to *4* for several reasons. First, 4 is a large enough number of partitions to ensure that variability of the model's predictions arising from the specific data points which are chosen in each round of cross-validation should be minimized after averaging the results. Additionally, and perhaps more importantly, 4 is a small enough number of partitions to ensure that the algorithm will run relatively quickly (choosing a very large number of partitions could drastically increase the run-time of the algorithm).

As mentioned above, it was determined that the best model was produced by letting *k = 10*. It should however be pointed out that *k = 10* produced only a nominally better model than several other values of *k*, as may be observed in the figure above. Therefore, while *k = 10* is the recommended choice, other values of *k* exist which will likely produce similar results when the model is applied to other independent datasets.

Question 3.1 (b)
Using the same data set (credit_card_data.txt or credit_card_data-headers.txt) as in Question 2.2, use the `ksvm` or `kknn` function to find a good classifier: (b) splitting the data into training, validation, and test data set

(Please note that my R code for this question is contained in the file 'homework_2_Q3.1b.R', and if you wish to run the code, you will need to change line 8 of my R code to your local directory which contains the credit card application data, referred to in this document as *cc_data*.)

Response
As mentioned in the previous question, it is always a good idea to implement a model validation technique when assessing the results of a model. When comparing multiple models as candidates to be used in solving a given problem, it is best to do this as a three-part process. The

set of data which is being analyzed should be partitioned into three complementary sets. 50-70% of the data should be partitioned to create a *training* set and the rest of the data should be split evenly into *validation* and *testing* sets. In the first step of the validation process, each model is trained using the *training* set of data. In the second step each model is applied to the *validation* set of data to in order to assess each model's performance. This is done because if a model is trained and then tested on the same set of data, it will generally exhibit greater performance than what may be expected when the model is later applied to an independent dataset. After testing each model's performance on the *validation* set of data, the best model is selected. However, after we have selected the best model we are not yet finished. As part of the third and final step, the performance of the best model is tested using the *testing* set of data, and then compared to the performance exhibited when the model was applied to the *validation* set of data. The reason we do this is because of the following idea:

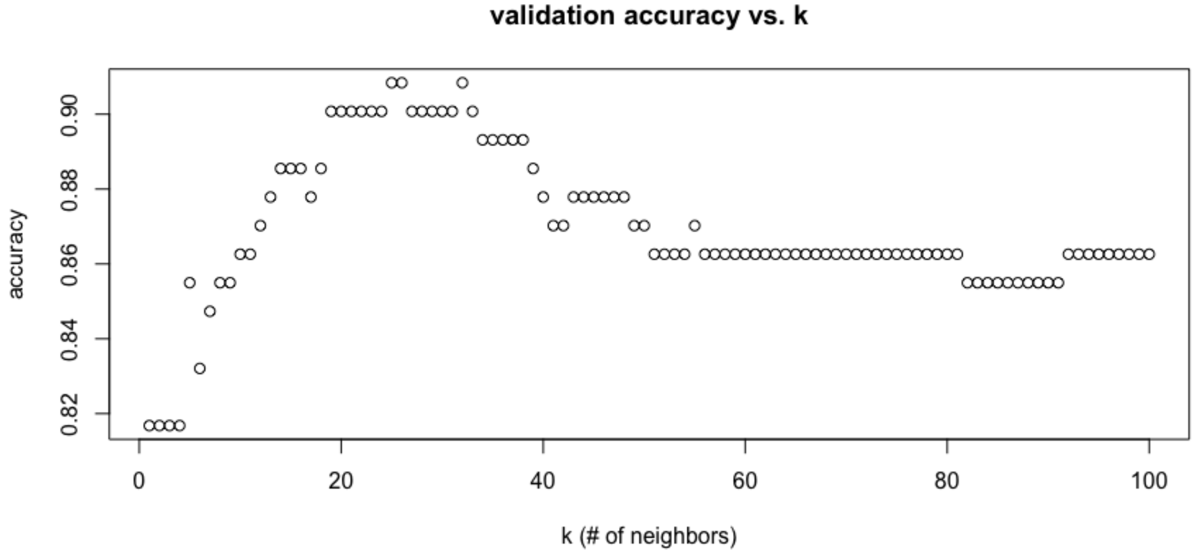
Observed performance = real quality + random effects

Because of the notion that observed performance is due in part to random effects, we can say that high-performing models are the most likely to have benefited from above average random effects, and low-performing models are the most likely to have suffered from below average random effects, assuming all models had a similar likelihood of success. Thus, we can in turn say that the performance of the best model tested using the *validation* set of data was likely overly optimistic, and the performance observed when this model is then applied to the *test* set of data is likely a more realistic representation of how the model will perform when applied in the future to other independent data sets.

In my solution to the problem of finding a good classifier for the credit card approval data, I first read in the credit card approval data and saved it in a matrix called *cc_data*. I used a three step validation process as described above to test the performance of the k-nearest neighbors algorithm as a method of finding a classifier, and I tested all numbers of neighbors *k* in the set *k = {1, 2, 3, ... ,100}.*

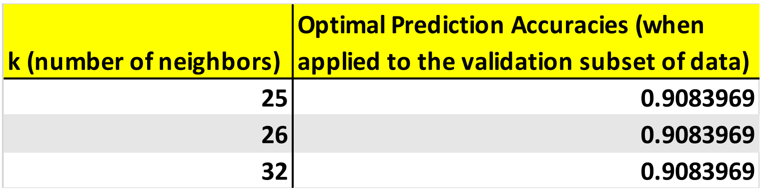
In order to implement the three-step validation process, I first needed to decide how to partition the data into complementary *training, validation,* and *testing* sets. The decision of how to partition the data is not trivial. In order to decide how to split the data, we must consider that there are two competing concerns which must be balanced when determining how the data should be partitioned. First, as the number of data points in the *training* set decreases, the variance of the parameters generated by the models will increase. Second, as the number of data points in the *validation* or *testing* sets decrease, the variance of the performance statistics produced by the applying the models to these sets will increase. In my implementation of validation testing for this particular problem, I chose to allocate *60%* of *cc_data* to the *training* set and to split the remaining *40%* of the data evenly between the *validation* and *testing* sets. It is most important to ensure that variance of the parameters of the models which are trained is minimized, and this is why I chose to allocate *60%* of the data to the *training* set. Allocating *20%* of *cc_data* to the *validation* set and *20%* of *cc_data* to the *testing* set ensured that each of these sets contains approximately *130* data points (*cc_data* contains 654 data points; *0.2 * 654 ≈ 130*). 130 data points is a sufficient number to ensure that the outcome of our model's validation and testing is not significantly affected by random effects resulting from variance caused by the selection of various individual points which are assigned to the *validation* and *testing* sets. The selection of which points should be allocated to the *training*, *validation*, and *testing* sets was determined by using R's random number generator *sample*, to randomly allocate *60%* of the data points to the *training* set, *20%* of the data points to the *validation* set, and *20%* of the data points to the *testing* set. It should therefore also be pointed out that if you choose to run my code, your results will likely be slightly different than the results contained within this paper, due to a random number generator is used in the partitioning process.

In my solution to this problem, I re-used the function cc_*kknn_function* from question 3.1(a). As mentioned above, *cc_kknn_function* utilizes R's *kknn* (k nearest neighbors) function, and is designed specifically to work with *cc_data*. *cc_kknn_function* takes in two parameters called *val_data* and *test_data*. I implemented the three step validation method described in the previous paragraph by utilizing *cc_kknn_function* in the following manner: First, I used the *taining* and *validation* and subsets of *cc_data* as input paramenters in *cc_kknn_function*. I trained *100* different models for all values *k* in the set *k = {1, 2, 3, ... ,100}*, where *k* is defined as the number of neighbors, using the *training* subset of data. Next, I applied the *100* models trained using the *training* subset of data to the *validation* subset of data. The percentage of correct results for each of the *100* trained models (as applied to the *validation* subset of data) was returned by *cc_kknn_function* in the form of a matrix in which each row in the first column represents the value of *k* (number of neighbors) used in creating a k-nearest neighbors model, and each row in the second column represents the corresponding accuracy of the model. The resulting accuracies for each value of *k* may be observed in the following figure:



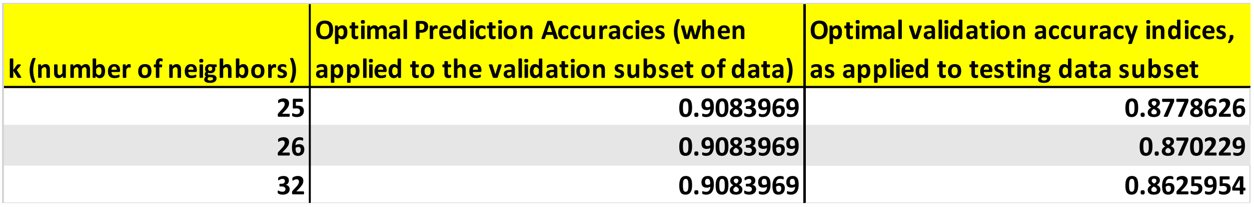
As may be observed in the figure, for every value of *k* in the set {1, 2, 3, … ,100}, the accuracy of every model fell between *0.81* and *0.92* . Thus, we can see that there is not a large variance in the accuracy of the models tested.

In order to determine the optimal model to use with regard to *k*, I found the value(s) of *k* from the figure above which resulted in the highest accuracy. This resulted in the determination that there are 3 separate values of *k* which each produce a model that results in an identical maximum accuracy, as may be observed in the following table:



k (number of neighbors)	Optimal Prediction Accuracies (when applied to the validation subset of data)
25	0.9083969
26	0.9083969
32	0.9083969

In accordance with the 3-step procedure defined above for model validation, I next applied the same models trained using the *training* subset of data to the *testing* subset of data. I began by analyzing the accuracy of the models applied to the *testing* set of data for the values of *k* which were found to be optimal when the model was applied to the *validation* subset of data. This analysis may be observed in the following figure:

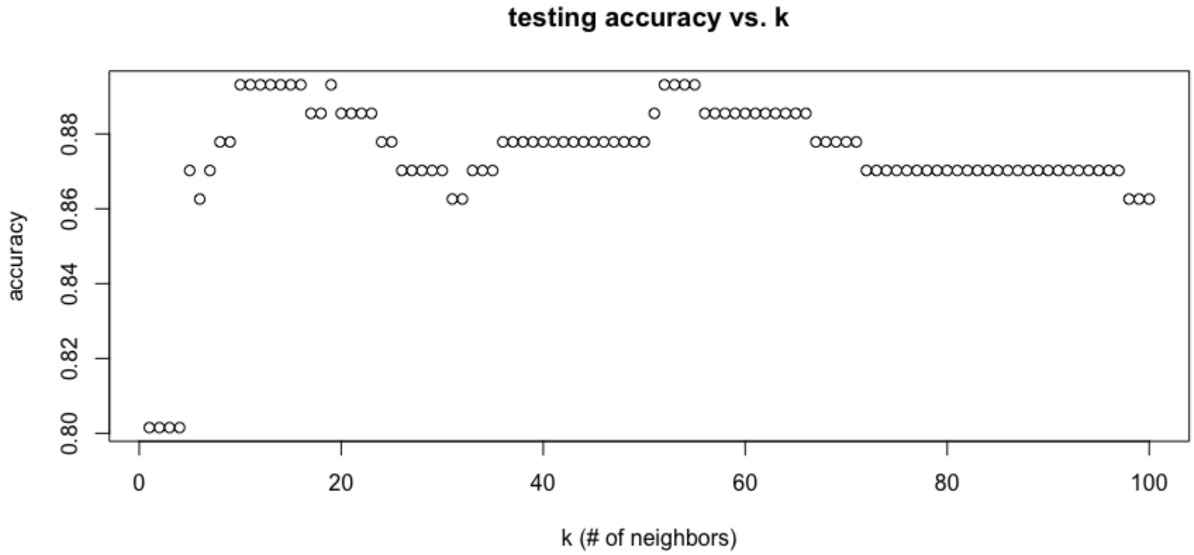


k (number of neighbors)	Optimal Prediction Accuracies (when applied to the validation subset of data)	Optimal validation accuracy indices, as applied to testing data subset
25	0.9083969	0.8778626
26	0.9083969	0.870229
32	0.9083969	0.8625954

As expected, when the optimal models, as determined by the *validation* subset of data, were applied to the *testing* subset of data, the resulting level of accuracy decreased slightly, as may observed in the above table. The accuracy levels observed when the models are applied to the *testing* data subset are likely a better indication of the level of performance that will be observed when the models are applied to other independent datasets.

Based upon the above analysis, I would suggest that the model in which the number of neighbors is given by *k = 25* should be selected as our choice of the optimal model. I suggest this choice because it is one of the three models which produced the highest level of accuracy when applied to the *validation* subset of data, and of the three best models determined using the *validation* subset of data, it produced the highest level of accuracy when applied to the *testing* subset of data. It should however be pointed out that other values of *k* will likely produce similar results when applied to other independent datasets, and in some cases other models may even produce more accurate results, simply due to random effects.

Normally, model validation would be complete after the above step. However, I dug deeper into the results produced by testing on the *validation* subset relative to the results produced by testing on the *training* subset. I applied each of the *100* models trained using the *training* subset of data, for the values *k* in the set *k = {1, 2, 3, ... ,100}*, where *k* is defined as the number of neighbors, to the *testing* subset of data, as was done above with using *validation* subset of data. The resulting accuracies for each value of *k* may be observed in the following figure:



As may be observed in the figure, for every value of *k* in the set {1, 2, 3, … ,100}, the accuracy of each model fell between *0.80* and *0.90* . Thus, we can see that there is not a large variance in the accuracy of each model tested.

It is of interest to note how the same models applied to two mutually exclusive subsets of the same data set can result in the determination that different models perform the best. This result may be attributed to the above described rule that:

Observed performance = real quality + random effects

Random effects due to how the data is divided into *training, validation*, and *testing* subsets may lead to the indication that different models are likely to perform the best, each time model validation is performed. These random effects may be measured in the following way:

Variability due to random effects = mean (with regard to k) of the absolute value of (validation accuracy for all k – testing accuracy for all k)

The random effects due to the way *cc_data* was partitioned into subsets resulted in a mean absolute difference in accuracy of *0.01572519* or about *1.6%* when the same 100 models were applied to the *validation* and *testing* subsets, for each value *k* in the set *k = {1,2,3, … ,100}*.

Question 4.1

Describe a situation or problem from your job, everyday life, current events, etc., for which a clustering model would be appropriate. List some (up to 5) predictors that you might use.

Response

I have worked in the renewable energy industry as a data scientist. One project that I have worked on which utilized clustering involved creating a model to predict which homes are likely candidates for photovoltaic (solar) panel installation in the future. Some predictors that can be used to cluster homes based on the likelihood of future PV installation include: household income, type of home (detached, townhouse, etc.), age of occupants, geographic location, orientation (cardinal direction) of the roof of the home, proximity to other PV installations, and average sunshine duration (how sunny the area is where is the home is located). Based upon these factors, a clustering model can effectively help to predict whether a given house should be considered a likely candidate for a future PV installation or not.

Question 4.2

The *iris* data set iris.txt contains 150 data points, each with four predictor variables and one categorical response. The predictors are the width and length of the sepal and petal of flowers and the response is the type of flower. The data is available from the R library datasets and can be accessed with iris once the library is loaded. It is also available at the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Iris). *The response values are only given to see how well a specific method performed and should not be used to build the model.*

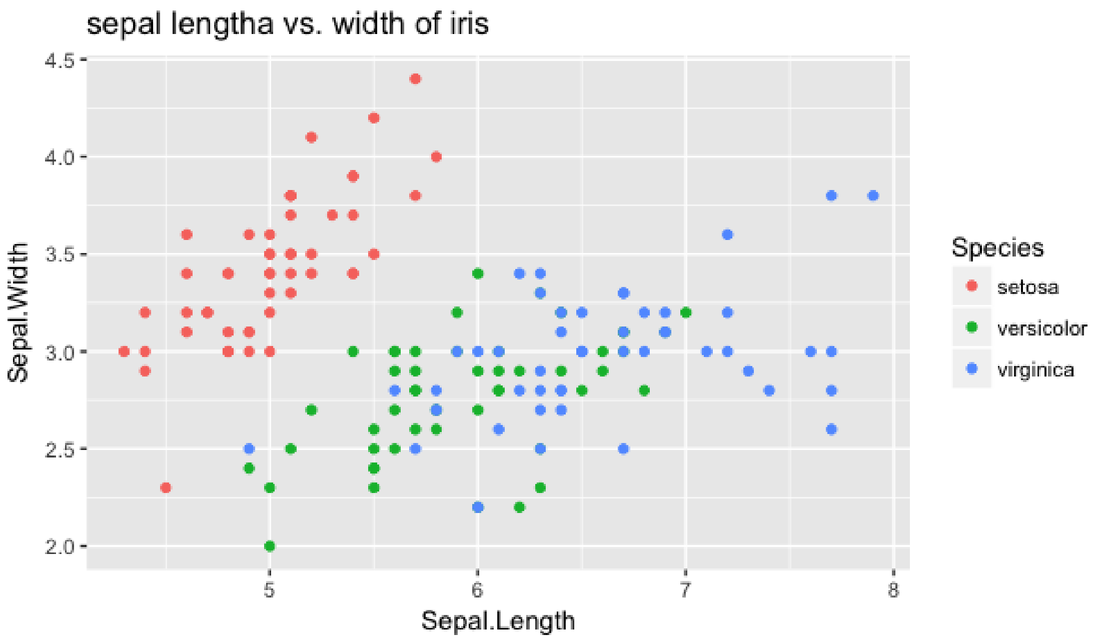
Use the R function `kmeans` to cluster the points as well as possible. Report the best combination of predictors, your suggested value of k, and how well your best clustering predicts flower type.

(Please note that my R code for this question is contained in the file 'homework_2_Q4.2.R')

Response

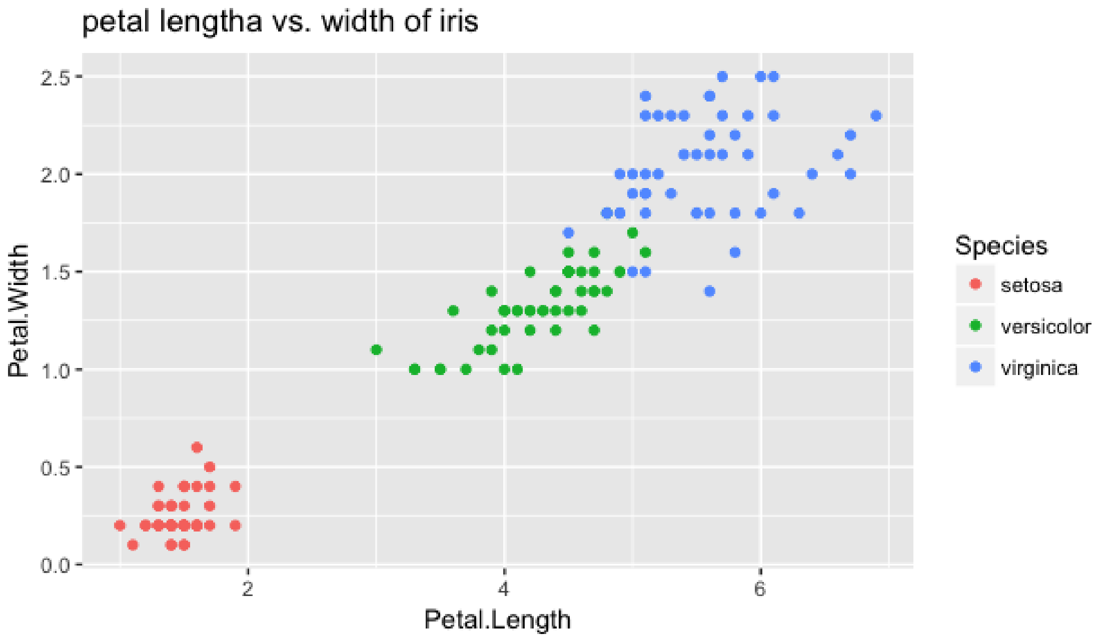
The k-means algorithm is a method of clustering (grouping) data points together based on similar features when the correct classification of each data point is not known. It is therefore considered an unsupervised machine learning algorithm. In the iris dataset, the correct classification of each data point is known, however it will be ignored when implementing k-means clustering, and only used to measure the performance of the implemented k-means algorithm.

To begin solving this problem, I started by plotting the *iris* data in order to have a better understanding of the dataset that I was working with. Iris contains 4 numerical predictor variables called *Sepal.Length, Sepal.Width, Petal.Length,* and *Petal.Width*, and 1 categorical response variable, called *Species*, which is comprised of the categories *setosa, versicolor*, and *virginica*. I began by plotting *Sepal.Length* against *Sepal.Width* and displayed each of the categorical response variables using three separate colors, as may be observed in the following figure:



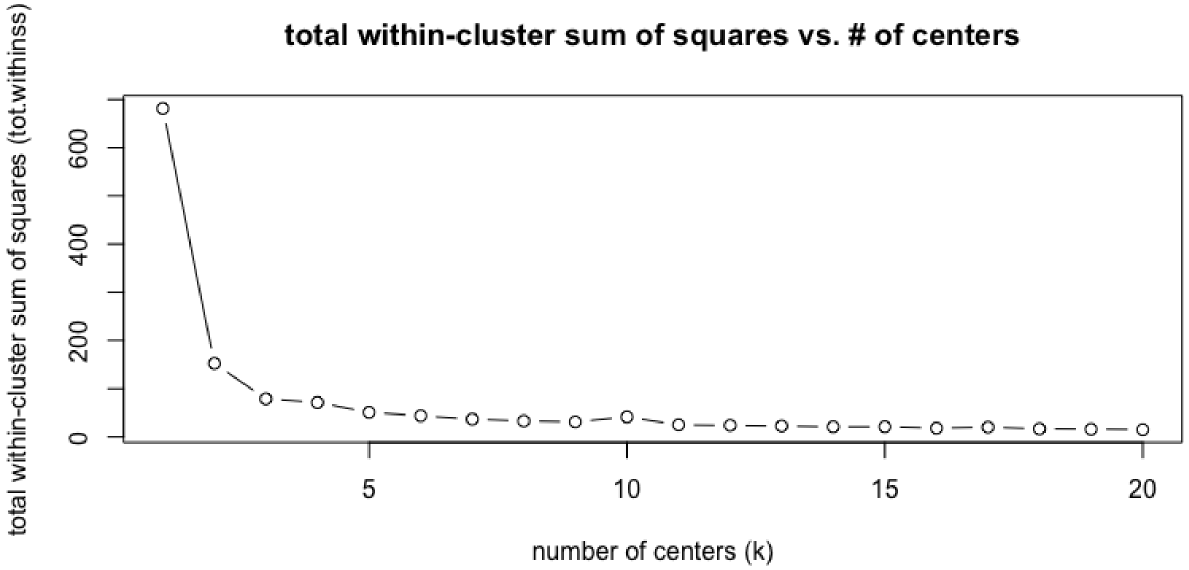
As it may be observed in the plot above, the *setosa* species of the iris plant may be separated from the *versicolor* and *virginica* species of the iris plant based on *Sepal.Length* and *Sepal.Width*, however the *versicolor* and *virginica* species of the iris plant cannot be distinguished from one another based only on this information.

I next plotted *Petal.Length* against *Petal.Width* and displayed each of the categorical response variables using three separate colors, as may be observed in the following figure:



As it may be seen in the plot above, *Petal.Length* and *Petal.Width* have similar values within each species, but vary significantly between species. This indicates that these values will be a useful combination to consider when using the k-means algorithm to cluster the *iris* dataset.

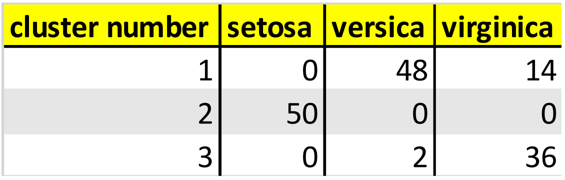
In k means clustering, one of the main challenges is determining the appropriate number of clusters. We already know that the *iris* data set contains three types of flowers, however as mentioned above, k-means is an unsupervised machine learning algorithm, and therefore the number of clusters would generally not be known prior implementing the k-means algorithm. I therefore wanted to see if I could independently determine the appropriate number of clusters. In order to do this, I ran R's *kmeans* algorithm 20 times within a for loop. Each time I ran the *kmeans* algorithm, I used a different number of cluster centers *k* and tested all values of *k* in the set of integers *k = {1, 2, 3, ... ,20}*. Each of the *20* times I ran the *kmeans* algorithm, I recorded the variable generated by R's *kmeans* function called *tot.withinss*. *tot.withinss* is simply the sum of *withinss*, and *withinss* is defined by R as a 'vector of within-cluster sum of squares, one component per cluster'. Therefore, *tot.withinss* is simply sum of squares within all clusters. Of course *tot.withinss* will generally decrease as the number of clusters increases, and a smaller value of *tot.withinss* indicates a smaller sum of squares, which indicates a better fitted model. However, we must remember that one of our primary goals in implementing the *k-means* algorithm is to determine the correct number of clusters. In order to do achieve this goal, I chose to create an elbow plot of *tot.withinss* versus the number of cluster centers. When using an elbow plot to determine the appropriate number of clusters, we choose the point at which adding another cluster doesn't significantly improve model performance. The elbow plot I created may be observed below



As it may observed from analyzing the above plot, we see a very substantial improvement in performance (i.e. *tot.withinss* decreases) as we move from the point *k =1* to the point *k = 2*, and again we can see that performance improves significantly as we move from *k = 2* to *k = 3* .

However, beyond the point *k = 3*, we see only a very minimal increase in model performance as *k* increases. This therefore indicates that *3* is the ideal number of clusters. Thus, we have rigorously shown, independent of our prior knowledge that the *iris* data set contains *3* types of species, that *3* is indeed the number species likely present within the *iris* dataset. It may therefore be recommend, independent of our prior knowledge, that we let the number of clusters *k* be equal to *3* when running the *kmeans* clustering algorithm on the *iris* dataset.

Finally, since we have determined that *3* is the optimal number of clusters, we will analyze this particular case in more detail. I chose to begin by making a table which shows how many members of each species were grouped into each of the 3 clusters. That table may be observed below:



cluster number	setosa	versica	virginica
1	0	48	14
2	50	0	0
3	0	2	36

As it may be observed in the table above, all *50* members of the *setosa* species were grouped together in *cluster 2*, which means that *100%* of the *setosa* species members were correctly classified. *48* members of the *versica* species were grouped together in *cluster 1*, and *2* members were placed incorrectly in *cluster 3*, implying that *96%* of the *versica* species members were correctly classified. Finally, *36* members of the *virginica* species were grouped together in *cluster 3*, and *14* members were placed incorrectly in *cluster 1*, implying that *72%* of the *versica* species members were correctly classified. It is perhaps not surprising that all of the *setosa* species were classified correctly if we recall that in the plots above, the *setosa* species of the iris plant appeared distinct from the other species with regard to the values contained in the predictor variables *Sepal.Length*, *Sepal.Width*, *Petal.Length*, and *Petal.* In total, *134* out of the *150* of the *iris* data points (approximately *89.3%*) were correctly classified. This is an impressive percentage of correct classifications, given that the *kmeans* algorithm determined this result in an unsupervised manner. It should be pointed out that if you choose to run my code, your results will likely vary slightly, since the starting points of each of the clusters is determined randomly by R’s *kmeans* function.