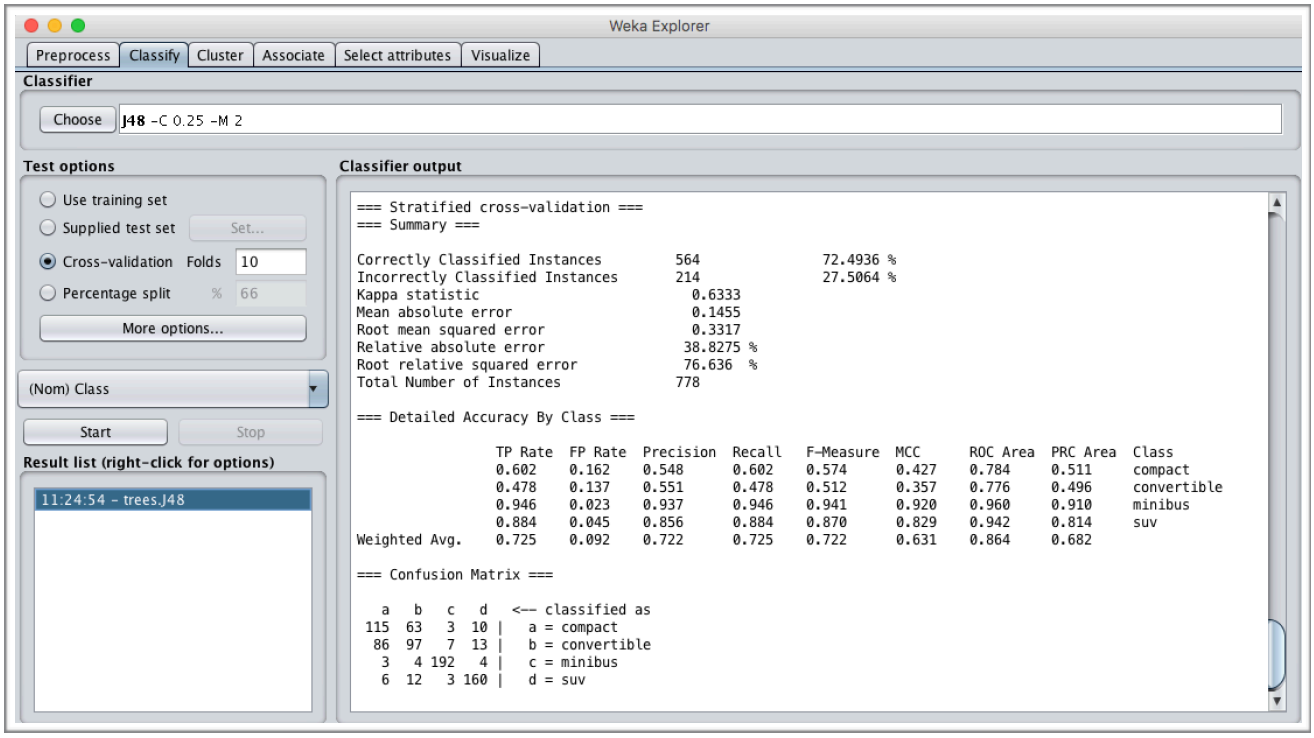
Name: Gurpreet Singh ***Assignment: 2*** ***Student No.: 17202278***

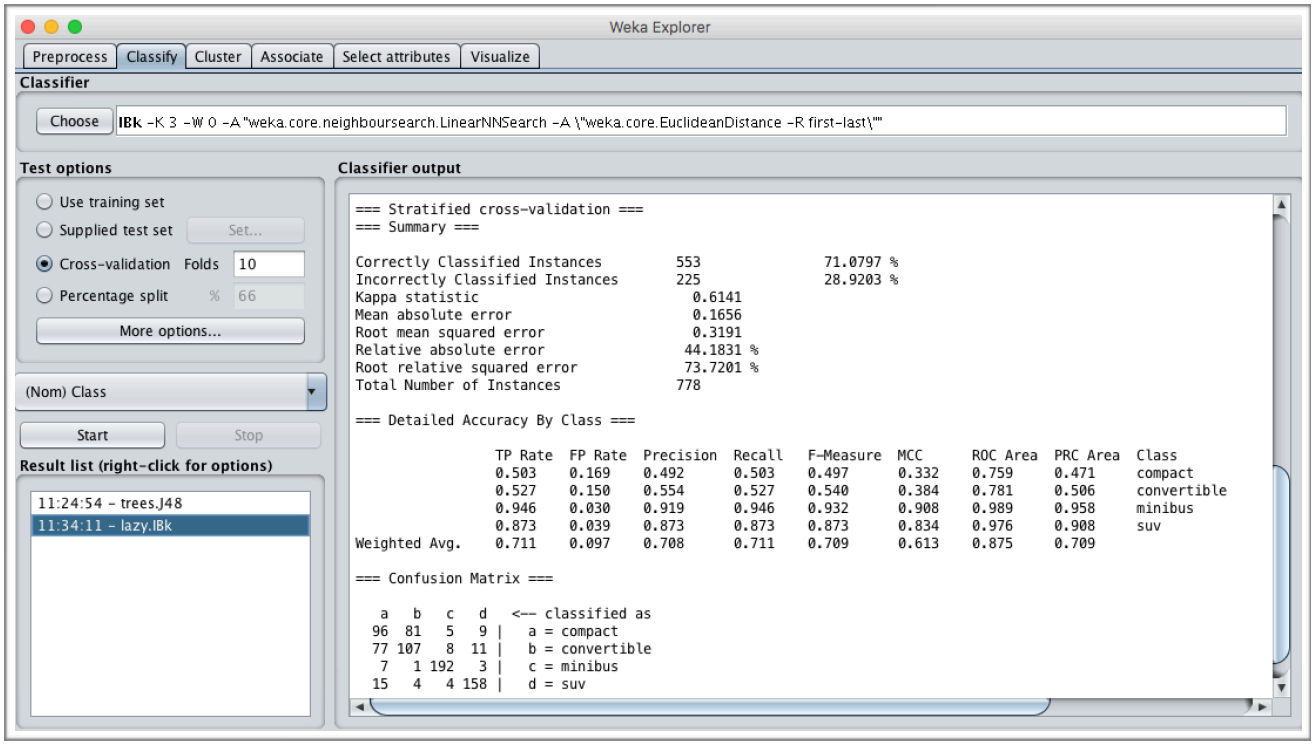
The Data Set used is:
http://claritytrec.ucd.ie/~alawlor/comp47460/datasets/ml/vehicles/17202278.arff

Q1. (a) Evaluate the performance of two basic classifiers on the data: Decision Trees (J48) and 3-NN.

Solution:
On evaluation with the Decision Trees(J48), I received the correctly classified instances as 72.49%, with cross-validation as 10.



On evaluation with the K-NN (IBK) with 3 nearest neighbours, I received the correctly classified instances as 71.07%, with cross-validation as 10.



Name: Gurpreet Singh *Assignment: 2* *Student No.: 17202278*

(b) Apply ensembles with bagging using both classifiers from Task (a). Investigate the performance of both classifiers as the ensemble size increases, in steps of 20from 20 to 100 members.
Using the best performing ensemble size, investigate, how changing the number of instances in the bootstrap samples affects classification performance (i.e. the "bag size").

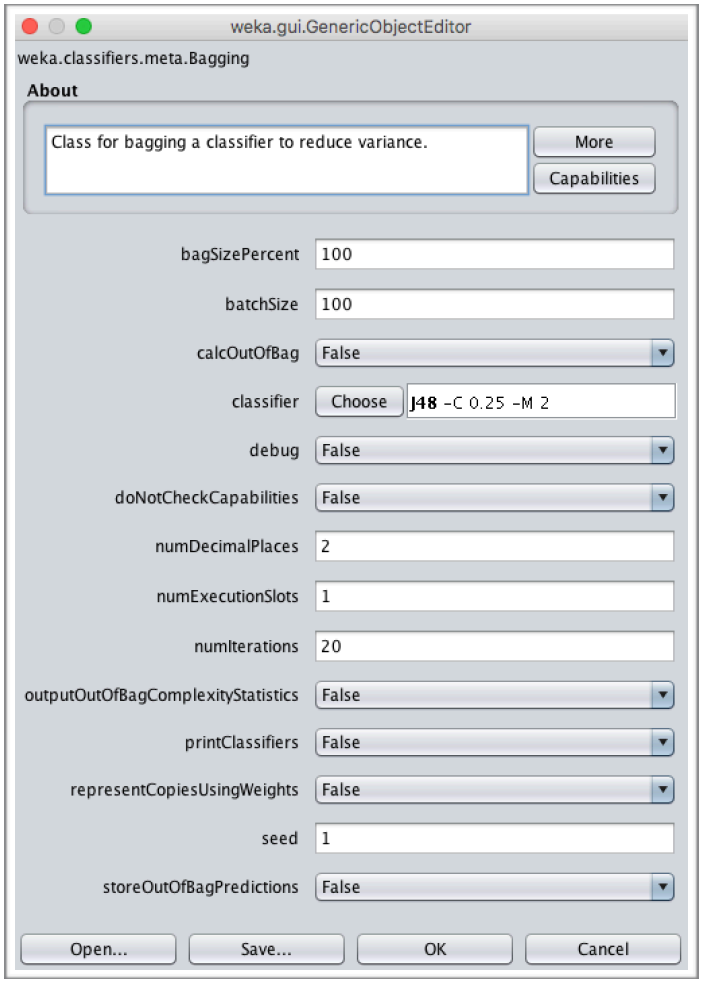
Solution:

Bagging with:

- **J48**
- **3-NN**

Steps:

1. Using Weka, click on the Classify tab.
2. Click Choose, select method classifiers->meta->Bagging.
3. Click Bagging in the box to the right. The configuration interface of the method appears.
4. Click Choose, select <J48/IBK(3-NN)>. (one at a time)
5. Set the "numIterations" to <20,40,60,80,100 (one at a time)> Click OK button.
6. Click Start button to build the ensemble.

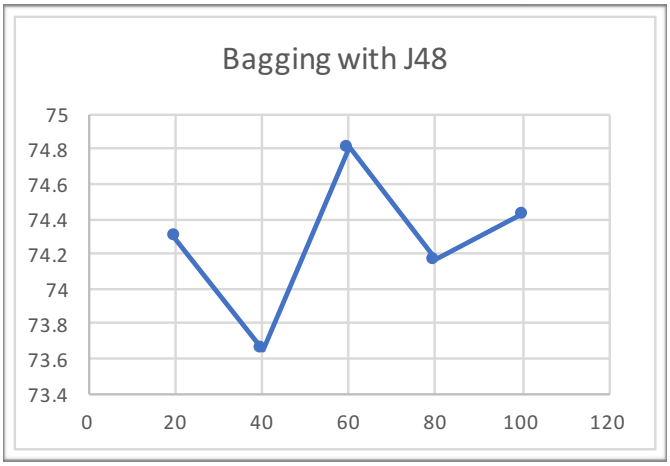


Bagging J48:

Bagging with J48

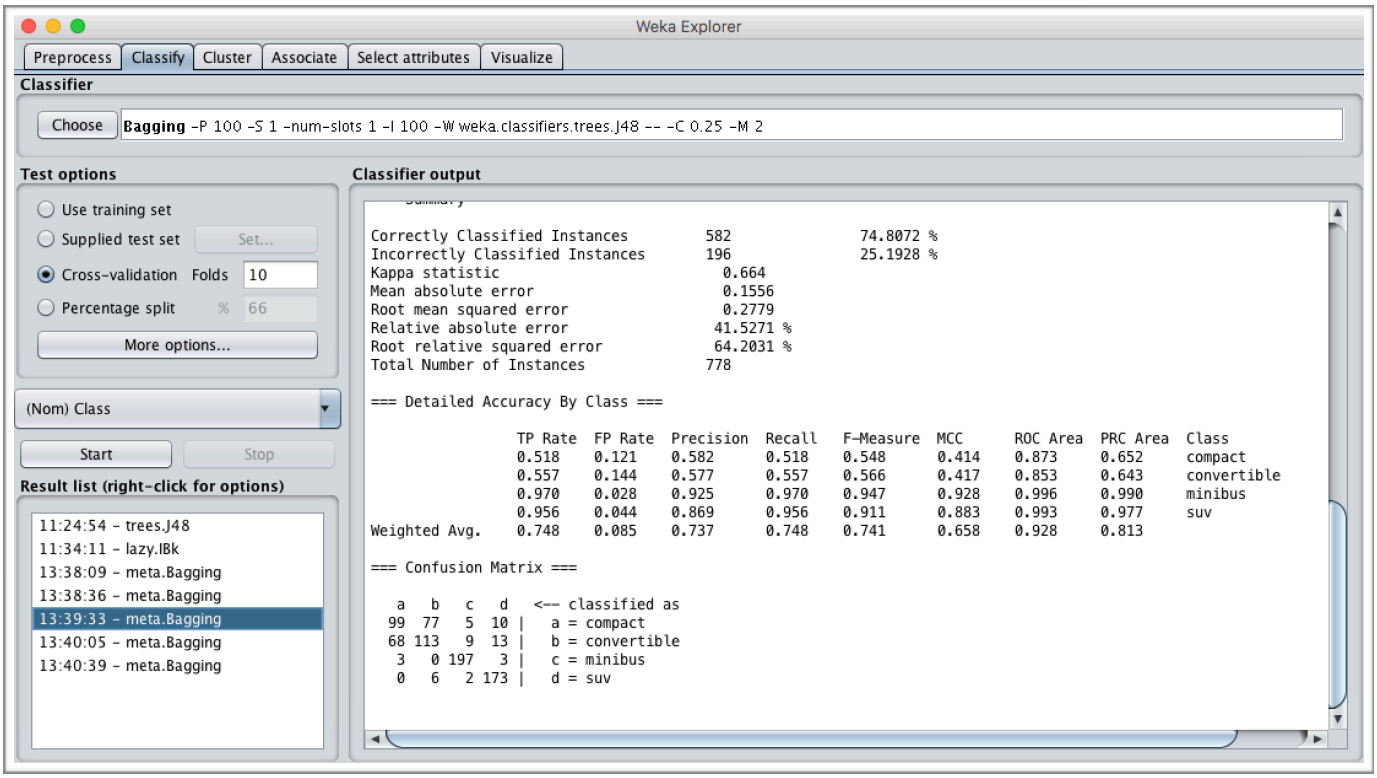
Ensemble Size	% Correct	% Incorrect
20	74.2931	25.7069
40	73.6504	26.3496
60	**74.8072**	**25.1928**
80	74.1645	25.8355
100	74.4216	25.5784

Visualisation of the performance



Name: Gurpreet Singh ***Assignment: 2*** ***Student No.: 17202278***

The best performance I received with the ensemble size of 60, with cross-validation as 10.

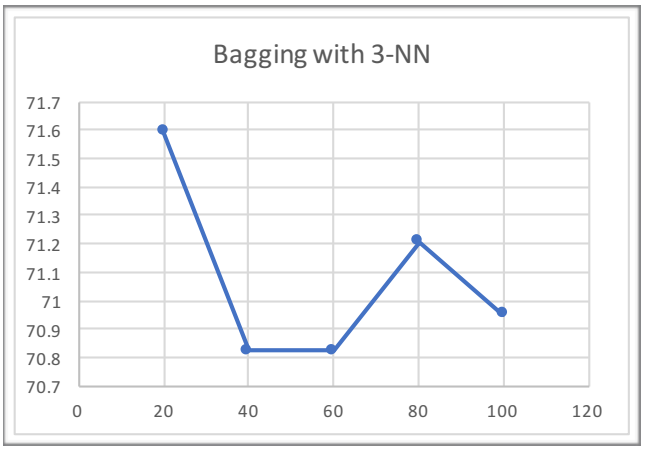


Bagging 3-NN:

Bagging with 3-NN

Ensemble Size	% Correct	% Incorrect
20	**71.5938**	**28.4062**
40	70.8226	29.1774
60	70.8226	29.1774
80	71.2082	28.7918
100	70.9512	29.0488

Visualisation of the performance



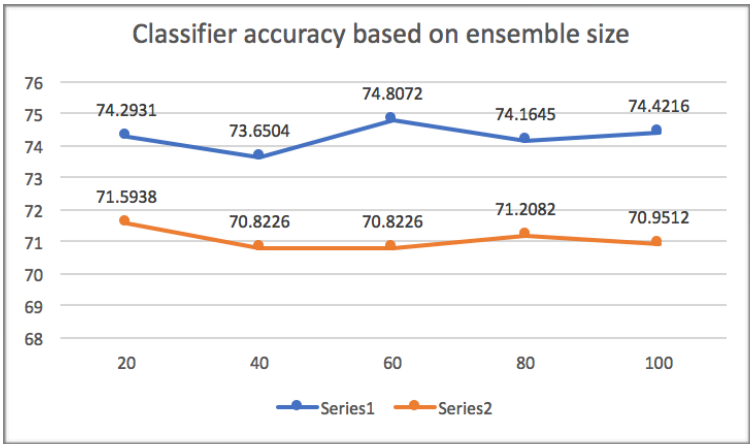
- *For both of the classifiers **the best <u>ensemble size</u> are, J48 with '60' and for 3-NN with '20'**.*
- *Therefore calculated the accuracy on **bag-size** from <u>10 to 100, with the step size of 10.</u>*
- *Right table shows the classifier's <u>accuracy on **varying** the bag size and **fixed** ensemble size as 60 for J48 and 20 for 3-NN</u>.*

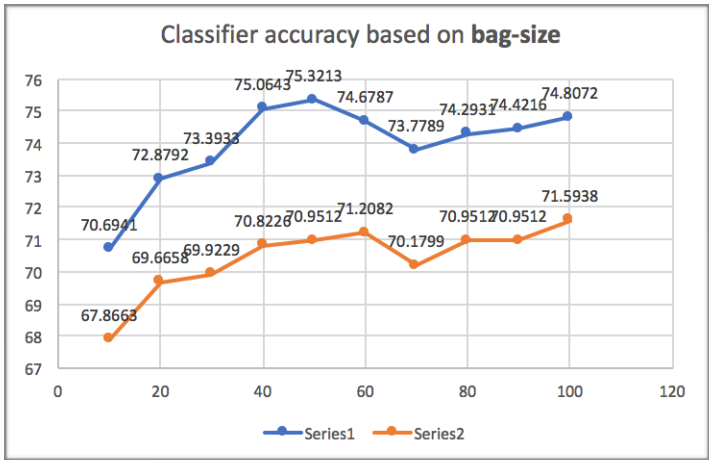
<u>Classifier (Ensemble Size)</u>

Bag Size	J48 (60)	3-NN (20)
10	70.6941	67.8663
20	72.8792	69.6658
30	73.3933	69.9229
40	75.0643	70.8226
50	**75.3213**	70.9512
60	74.6787	71.2082
70	73.7789	70.1799
80	74.2931	70.9512
90	74.4216	70.9512
100	74.8072	**71.5938**

Name: Gurpreet Singh ***Assignment: 2*** ***Student No.: 17202278***

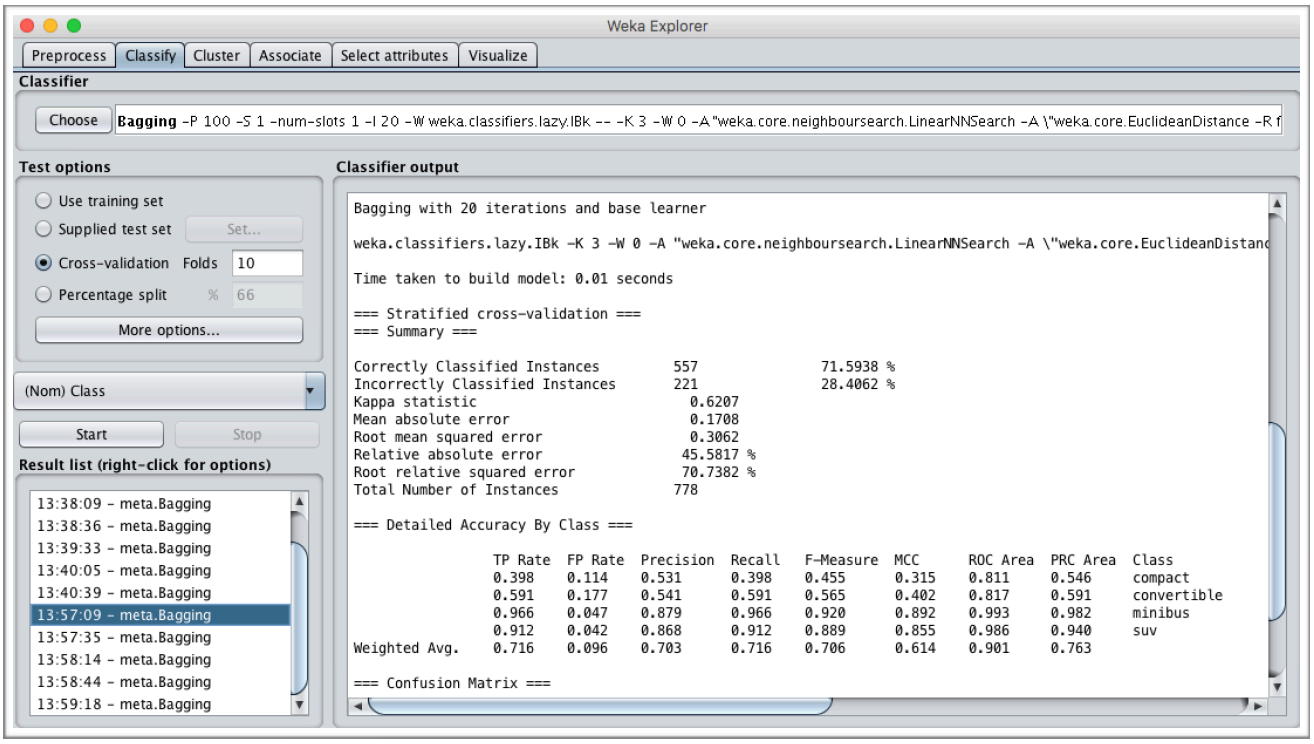
Below is the plot showing the performances on ensemble size and bag-size for best ensemble size.





Series 1: J48 & Series 2: 3-NN

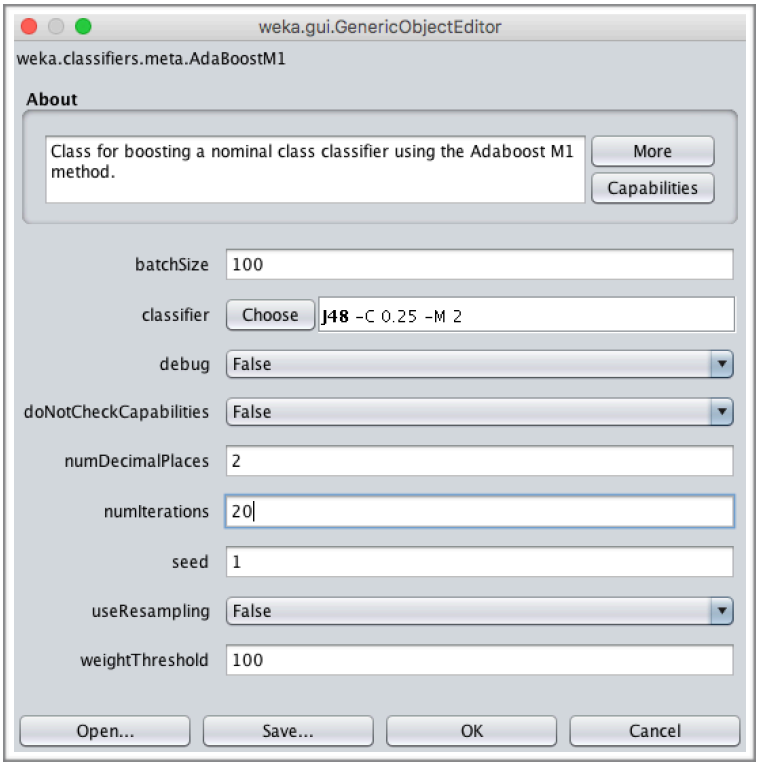
The best performance I received with the ensemble size of 20, with cross-validation as 10.



(c) Apply ensembles with boosting using both classifiers from Task (a). Investigate the performance of both classifiers as the ensemble size increases, in steps of 20 from 20 to 100 members.

Steps:

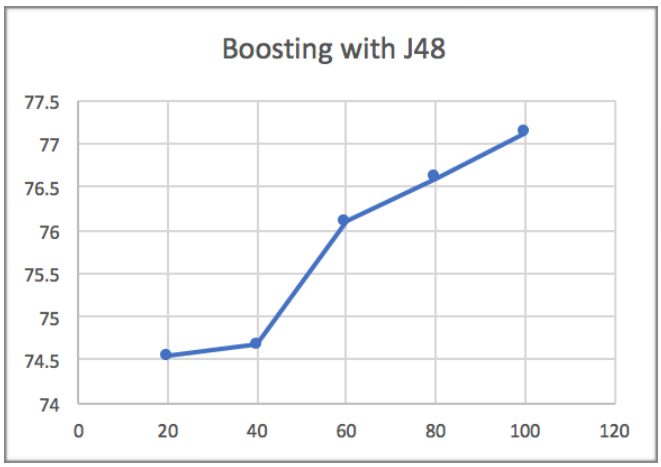
1. Using Weka, click on the Classify tab.
2. Click Choose, select method classifiers->meta->AdaBoostM1.
3. Click AdaBoostM1 in the box to the right. The configuration interface of the method appears.
4. Click Choose, select <J48/IBK(3-NN)>. (one at a time)
5. Set the numIterations to <20,40,60,80,100 (one at a time)>.
6. Click OK button.
7. Click Start button to build the ensemble.



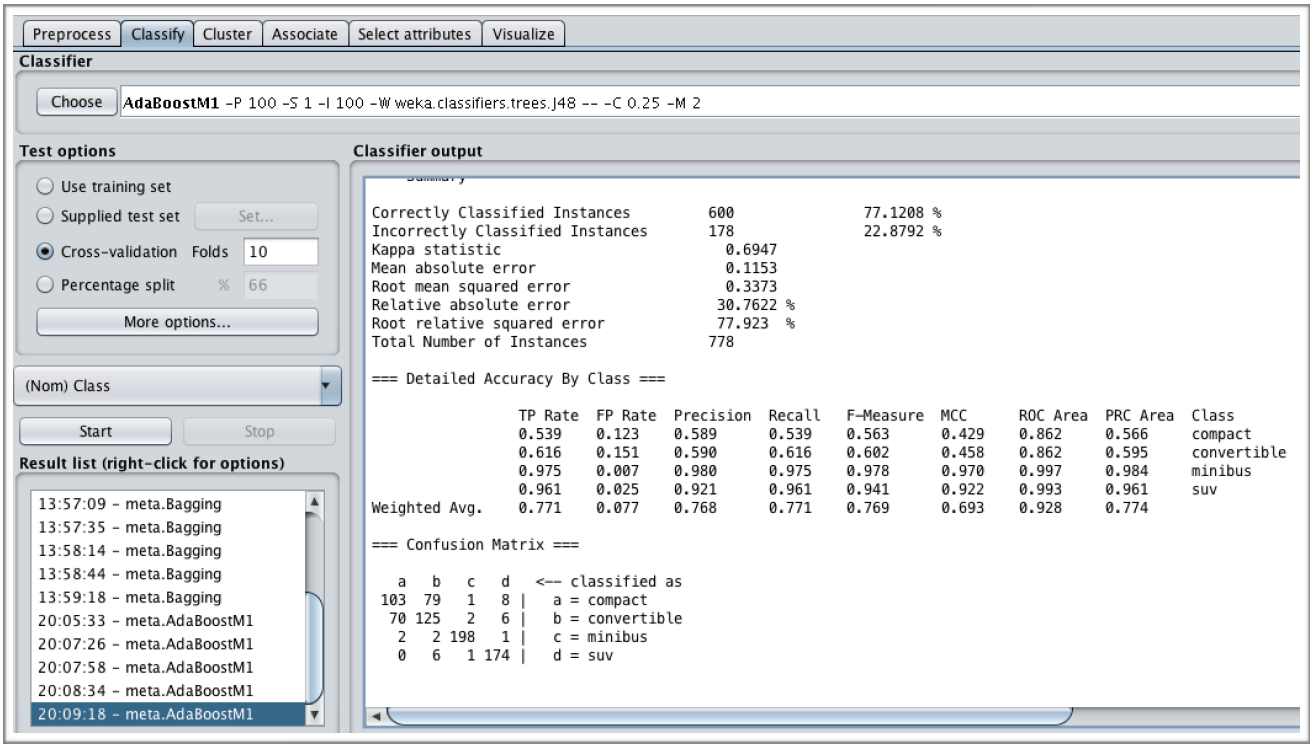
Boosting with J48:

Ensemble Size	% Correct	% Incorrect
20	74.5501	25.4499
40	74.6787	25.3213
60	76.0925	23.9075
80	76.6067	23.3933
100	**77.1208**	**22.8792**

Visualisation of the performance



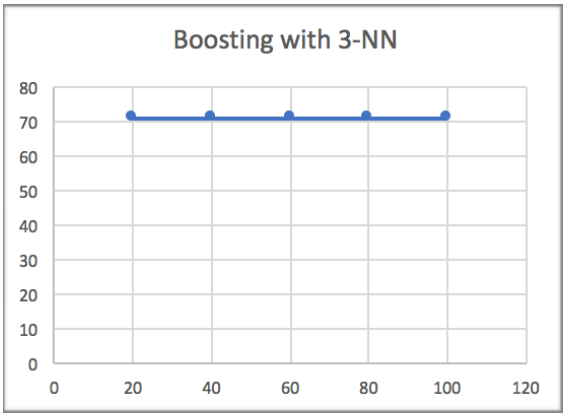
The best performance I received with the ensemble size of 100, with cross-validation as 10.



Boosting with 3-NN:

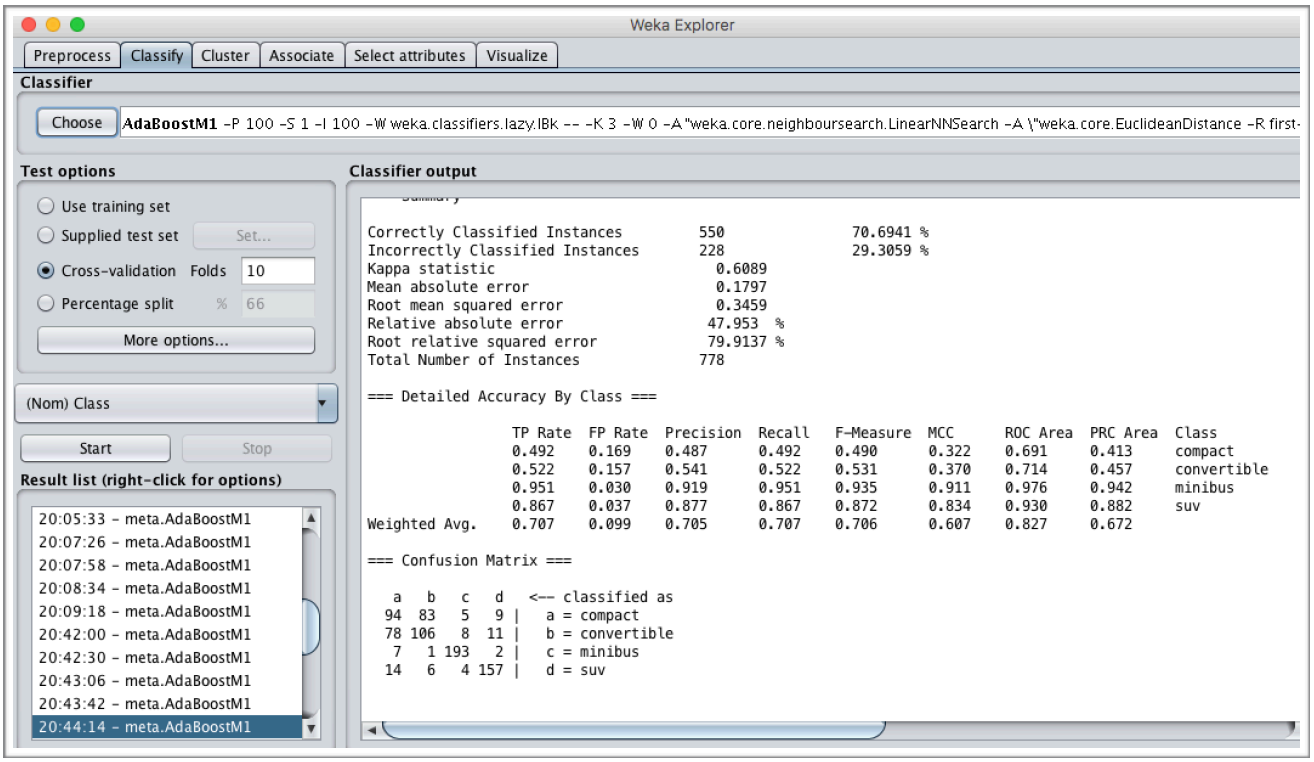
Ensemble Size	% Correct	% Incorrect
20	70.6941	29.3059
40	70.6941	29.3059
60	70.6941	29.3059
80	70.6941	29.3059
100	70.6941	29.3059

Visualisation of the performance



*Here for every 'numiterations' I received the **same** correctly and incorrectly classified instances, there by for this on researching I got to know that the direct application of boosting to k-NN (in this case 3-NN) classifiers is not available, when using re-weighting, or resampling. k-NN is in-stable with respect to inputs.[12] There-fore the performance didn't increased / decreased.*

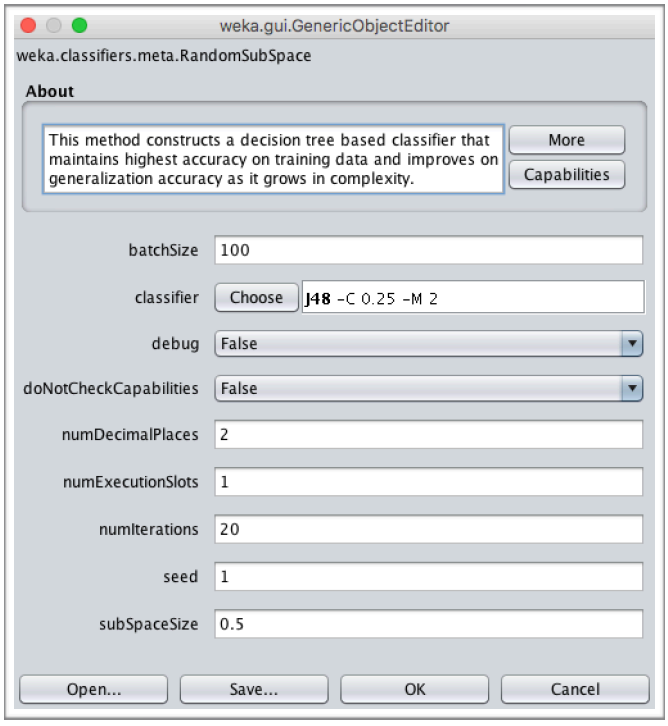
The performance for 20/40/60/80/100 as *all were same, with cross-validation as 10*:



(d) Apply ensembles with random subspacing using both classifiers from Task (a). Investigate the performance of both classifiers as the ensemble size increases, in steps of 20 from 20 to 100 members.
Using the best performing ensemble size, investigate how changing the number of features used when applying random subspacing affects classification performance (i.e. the "subspace size").

Solution:

1. Using Weka, click on the Classify tab.
2. Click Choose, select method classifiers->meta->RandomSubSpace.
3. Click RandomSubSpace in the box to the right. The configuration interface of the method appears.
4. Click Choose, select <J48/IBK(3-NN)> (one at a time).
5. Click J48/IBK(3-NN). In the configuration window, if KNN then set to 3. Close window.
6. Set the numIterations to <20,40,60,80,100 (one at a time)>.
7. Click OK button.

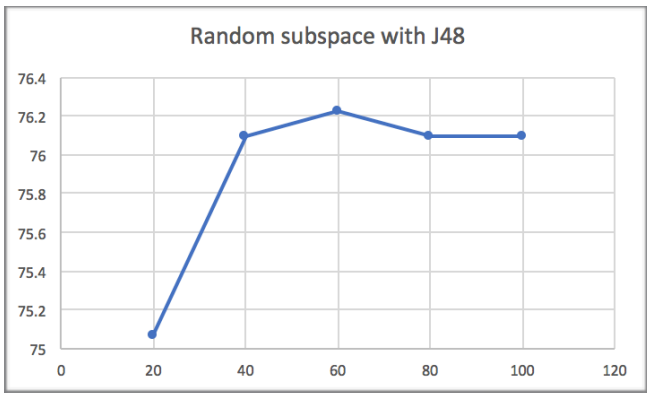


Name: Gurpreet Singh ***Assignment: 2*** ***Student No.: 17202278***

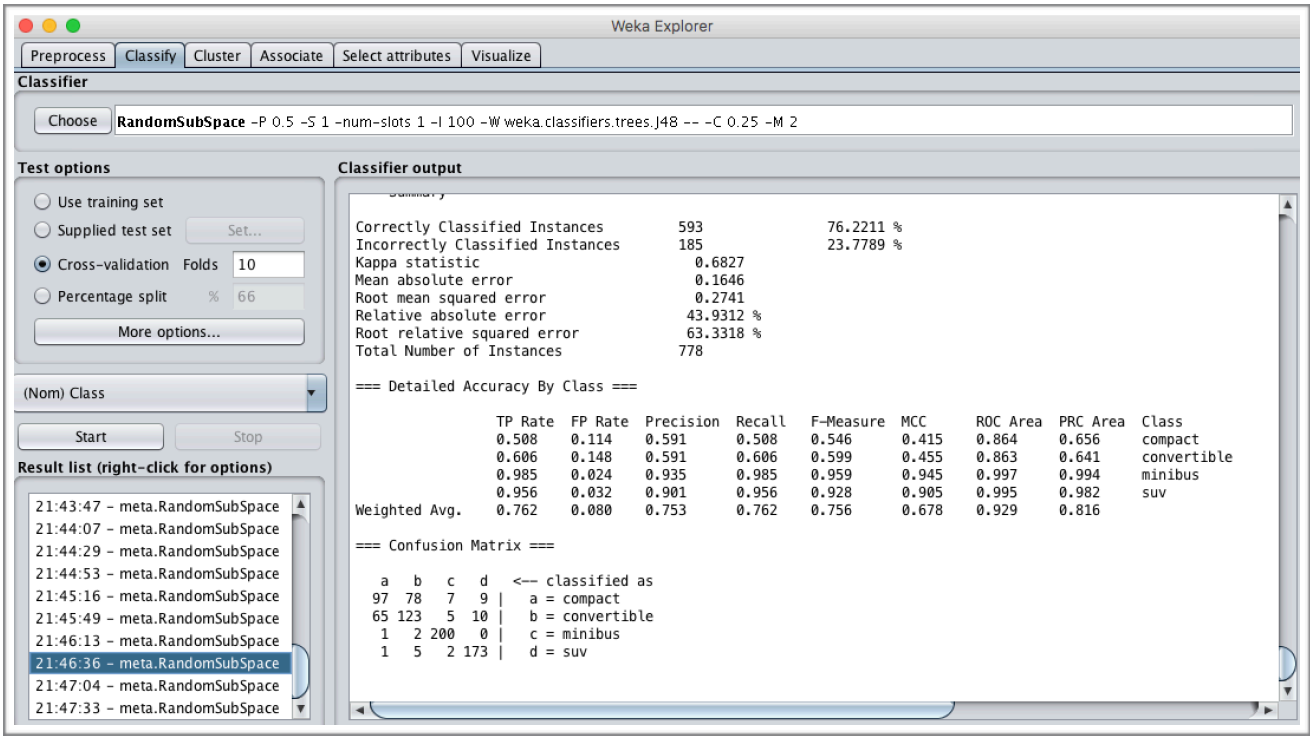
Random subspace with J48:

Ensemble Size	% Correct	% Incorrect
20	75.0643	24.9357
40	76.0925	23.9075
60	**76.2211**	**23.7789**
80	76.0925	23.9075
100	76.0925	23.9075

Visualisation of the performance



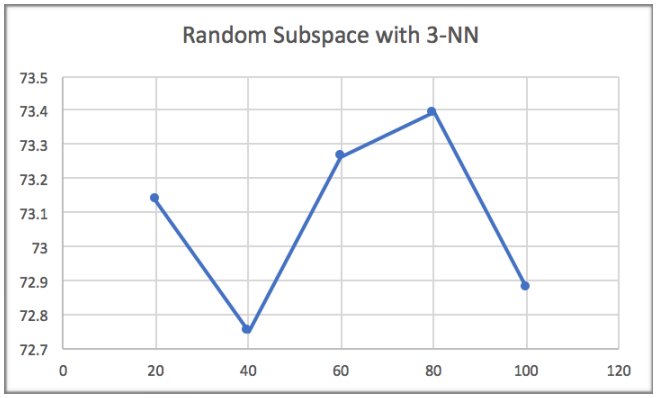
Best accuracy with numIterations as 60, with cross-validation as 10:



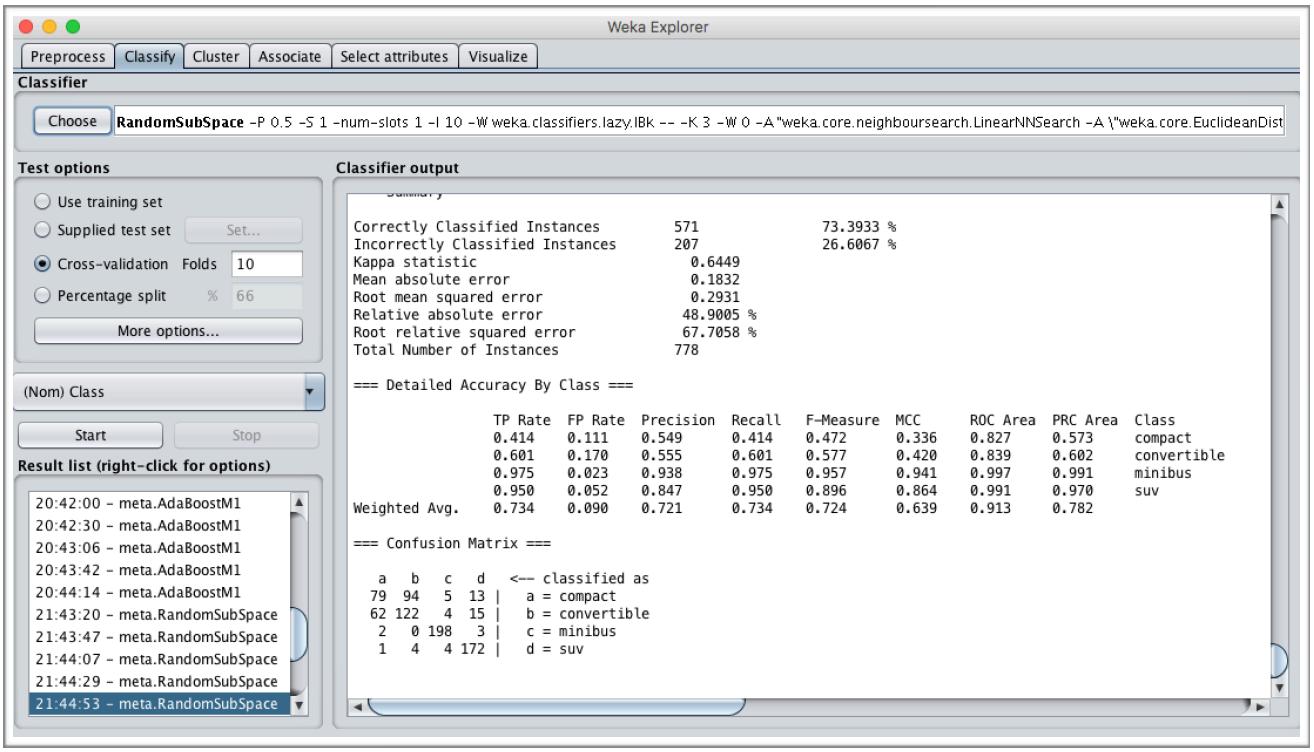
Random subspace with 3-NN:

Ensemble Size	% Correct	% Incorrect
20	73.1362	26.8638
40	72.7506	27.2494
60	73.2648	26.7352
80	**73.3933**	**26.6067**
100	72.8792	27.1208

Visualisation of the performance



Best accuracy with numIterations as 80, with cross-validation as 10:



Best Performance:

We can clearly see from the above set that "**Random Subspace with J48 Decision Tree** with **Ensemble Size** as **60**" performed best, with correct percentage as 76.2211, with cross-validation as 10.

In the tables shown below it can be seen *different subspace sizes* (0.25,0.5,0.75,0.95) to see how the value of RandomSubSpace *affect the accuracy of the models,* with the **J48 Decision Tree and 3-NN** with *Ensemble Size 20/40/60/80/100, previously*.

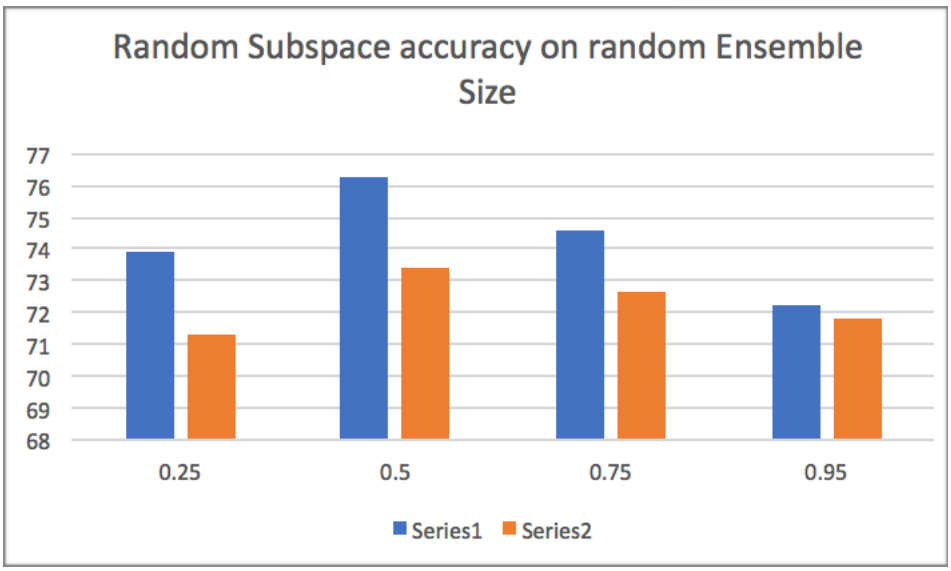
Random Subspace with J48 Decision Tree:

Subspace Ensemble size	0.25	0.5	0.75	0.95
20	**75.1928**	75.0643	74.8072	72.108
40	72.7506	76.0925	**74.9357**	**72.2365**
60	73.9075	**76.2211**	74.5501	**72.2365**
80	73.9075	76.0925	73.9075	71.9794
100	74.2931	76.0925	73.7789	72.108

Random Subspace with 3-NN:

Subspace Ensemble size	0.25	0.5	0.75	0.95
20	71.0797	73.1362	71.8509	72.108
40	70.5656	72.7506	72.2365	**72.365**
60	71.0797	73.2648	**73.0077**	71.9794
80	**71.3368**	**73.3933**	72.6221	71.8509
100	71.2082	72.8792	72.6221	72.2365

Here from the above table we can see clearly that the *best performance of J48* was with the ensemble size 60, where as on the other hand the *best accuracy with 3-NN* has with ensemble size of 80. Below is the visualisation of both of the best accuracies based on the variable



Series 1: J48 & Series 2: 3-NN

Conclusion:

- In both the above cases it can be seen that a subspace value of 0.5 performs better than 0.25.
- This is because working with 25% of the features leaves too much of important information in order to produce an accurate model.
- Hence, it can be seen that even tough with the variable ensemble size, the *subspace=0.5* has always performed better for the J48 algorithm for my dataset.
- It can be further seen that the J48 has performed better than 3-NN in almost all the ensemble size leaving 0.95.

(e) Summarise the differences in the performance results from Tasks (a)-(d), and describe some factors which might explain these differences.

Solution:

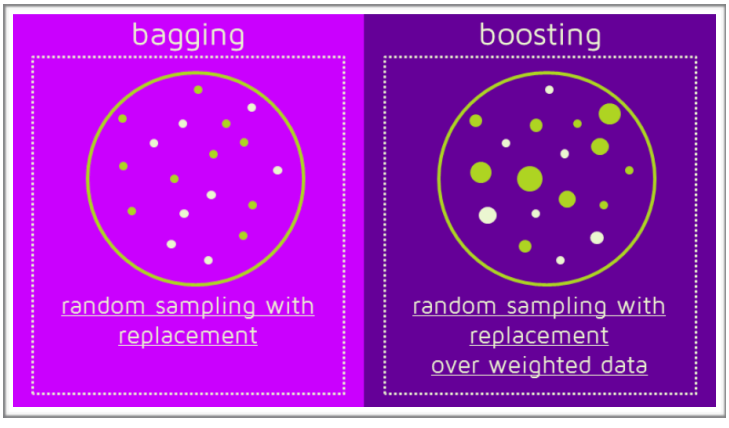
In the below table I have summarised all the above table, and we can see the best accuracy of both the J48 and 3-NN's with Bagging, Boosting and Random Subspace on its respective ensemble size (and subspace for Random subspace).

Method's Accuracy	J48	3-NN
Bagging	74.8072	71.5938
Ensemble Size	60	20
Boosting	**77.1208**	70.6941
Ensemble Size	**100**	20/40/60/80/100
Random SubSpace	76.2211	73.3933
Ensemble Size & Subspace	60 0.5	80 0.5

These differences are because:

Bagging:
Bagging is an ensemble generation method that uses variations of samples used to train base classifiers. For each classifier to be generated, Bagging selects (with replacement (No weights)) N samples from the training set with size N and train a base classifier. This is repeated until the desired size of the ensemble is reached.
Bagging should be used with unstable classifiers, that is, classifiers that are sensitive to variations in the training set such as Decision Trees.
That is why J48 is having (74.8072%) better score than 3-NN with (71.5938%).

Boosting:
Boosting generates an ensemble by adding classifiers that correctly classify "difficult samples". For each iteration, *boosting weights the samples*, so that, *misclassified samples* by the ensemble can have a *higher weight.* Hence, in next run those have *higher probability of being selected for training the new classifier.*
Boosting approach is very sensitive to noise and is only effective using weak classifiers.
Hence we can see that with J48 this outperformed than the remaining with the accuracy of 77.1208%.



Random Subspace:
Random Subspace is an approach that uses *variations in the features* instead of *variations in the samples*.

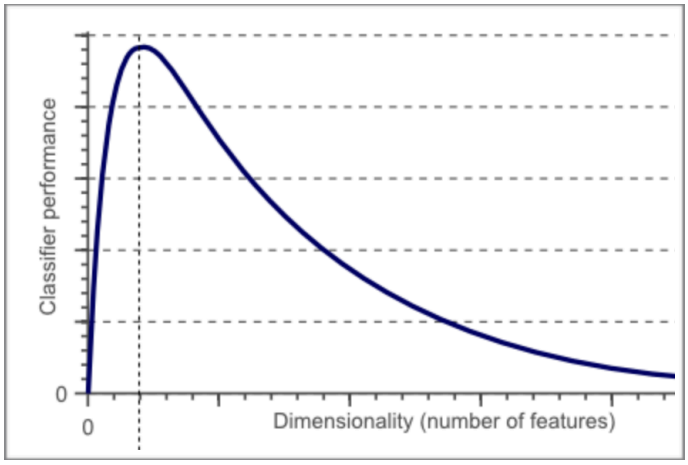
That is why we can see that the performance of random subspace is better than bagging in both the cases J48 and 3-NN, whereas for *3-NN random subspace gave the best accuracy.*

To sum up the best performance was with J48 with Boosting and having the ensemble size as 100.

Question 2

a)

Curse of dimensionality: The curse of dimensionality refers to how certain learning algorithms may perform poorly in high-dimensional data. Here below we can see that on increasing the number of features the performance increased, but after a certain interval the performance started decreasing, this behaviour is termed as curse of dimensionality.



Problems:
This problem with high dimensional data is that *because of the way distance is calculated*, there are fewer points near any given data point, and *a lot of points of roughly the same distance*.

The curse of dimensionality causes problems for many of the algorithms because most of the machine learning methods use ***distance*** as a measure:
Most of the algorithms rely on the segmentation and clustering methods by computing the distances between observations.
E.g.

- K nearest neighbours assign points to the nearest centres.
- DBSCAN & Hierarchical clustering also uses the distance metrics.

Distribution and density based **outlier detection** algorithms also make use of *distance relative to other distances to mark outliers.*
Most *common distance metrics is Euclidian Distance metric*, which is simply linear distance between two points in multi-dimensional hyper-space. **Euclidian Distance** for point i and point j in n dimensional space can be computed as:

$$d_{ij} = \sqrt{\sum_{k=1}^{n} (x_{ik} - x_{jk})^2}$$

Ways to handle curse of dimensionality:

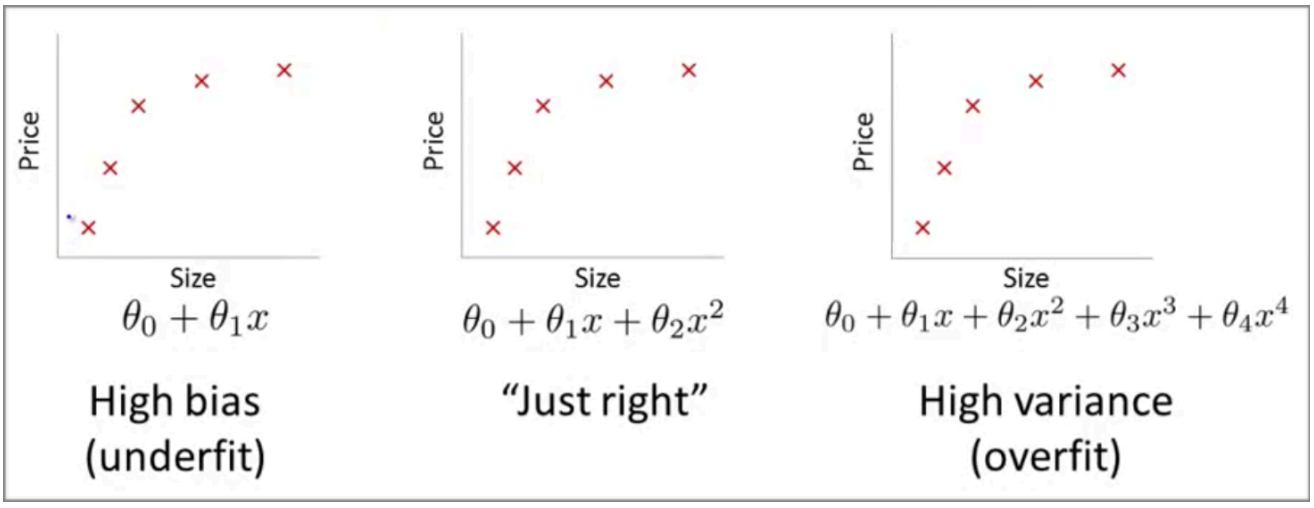
- *Applying preprocessing techniques to reduce the number of features:*
  For this we can use Feature Transformation & Feature Selection methods.
  Feature Transformation: Transforms the original features of a dataset to a completely new, smaller, more compact feature set, while retaining as much information as possible.
  e.g. Principal Components Analysis (PCA), Linear Discriminant Analysis (LDA).
  Feature Selection: Tries to find a minimum subset of the original features that optimises one or more criteria, instead of producing an entirely new set of dimensions for the data.
  e.g. Information Gain filter, Wrapper with sequential forward selection.

On further searching I found a method Auto-encoding by which the dimensions can be reduced too.

- Auto encoders are similar to PCA but PCA is *restricted to a linear map*, while auto encoders *can have nonlinear encoder/decoders*.
  - A single layer auto encoder with linear transfer function is nearly equivalent to PCA.
  - The linearity of PCA, however, places significant limitations on the kinds of feature dimensions that can be extracted.
  - Auto-encoders overcome these limitations by exploiting the inherent nonlinearity of neural networks.

(b)

Bias and variance are the types of error the classifier can make.
This usually occurs with the under-fitting or an overfitting problem, so we need to figure out which is the problem occurring so that we can improve the performance of the algorithm. But we would like to minimise each of these, but we can't do this independently as there is a trade-off.

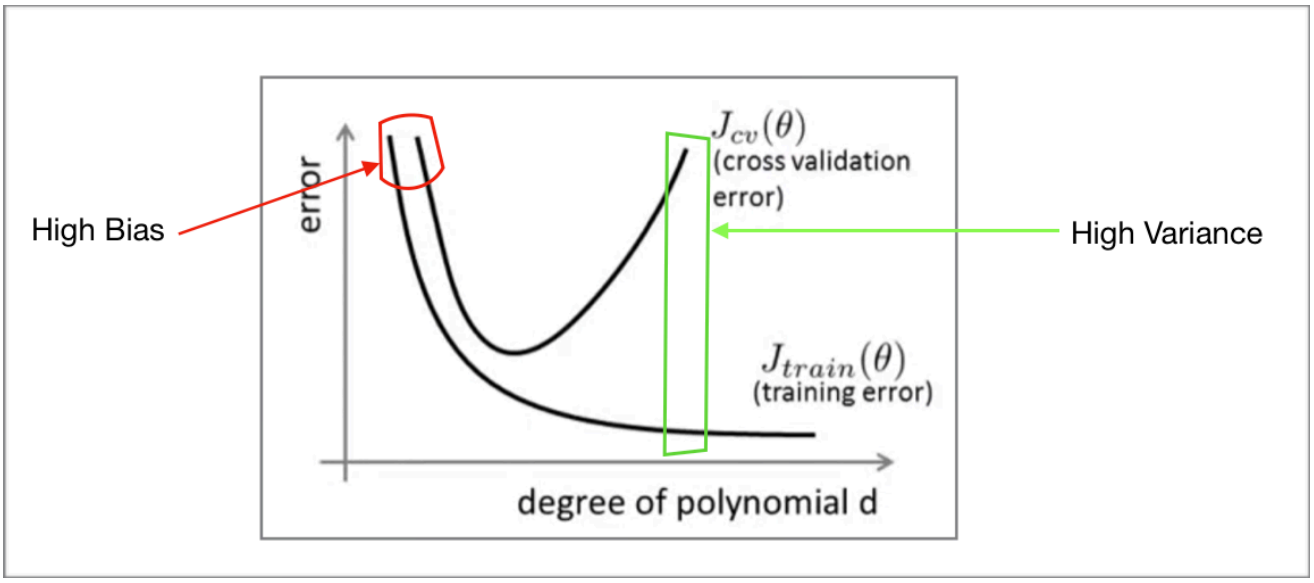


Bias:
Bias is when the training error is usually high with very less number of features where as the error rate is very low as the degree of features increases.

Variance:

On the other hand when we do the same thing with and we find that with less degree of feature the error is very high but on increasing the features the error reduces to a certain point but on further increasing of the features the error again rises, this condition is known as variance.

It can be shown as:



To Know Our Model has:
- High Bias or
- High Variance

High Bias:
- It occurs when the *training set error is high* as well as the *cross validation error* is also *high*. So when both of the errors are high then we say our model is having high bias.
- High bias means a poor match.
- Measures the accuracy or quality of the algorithm.

High Variance:
- It occurs when the *training error is very low* (i.e. the training set is well fit) but we have a *high cross validation error*. So when we get this combination we can say that we have high variance in our model.
- A high variance means a weak match.
- Measures the precision or specificity of the match.

Effects of bagging in bias and variance:

Bagging: Bootstrap **Agg**regati**ng**
So the Bagging is an ensemble generation method that uses variations of samples used to train base classifiers. For each classifier to be generated, **Bagging selects (with replacement (No weights)).**
"According to the approximate way of estimating variance, *bagging removes the variance while leaving bias unchanged."*

Explanation:
For b = 1, …, B do
Sb = bootstrap replicate of S
Apply learning algorithm to S(b) to learn h(b)
Classify new points by unweighted vote:

$$\frac{\sum_b w_b h_b}{B} > 0$$

Bagging makes predictions according to:

$$y = \frac{1}{B}\sum_{b} h_b(\mathrm{x})$$

Hence, bagging's predictions are h(x)

Now If we estimate bias and variance using the same B bootstrap samples:

- Bias = $(\underline{h} - y)$ [same as before]
- Variance = $\Sigma_k\ (\underline{h} - \underline{h})^2/(K-1) = 0$

(c)

Logistic regression model Based on the coefficient sign:

- Logistic regression models are used when the outcome of interest is binary.(There are ways to handle multi-class classification, too.)
- The predicted values, which are between zero and one, can be interpreted as probabilities for being in the positive class—they are labeled as 1**.**
- Interpreting logistic regression on coefficients amounts to calculate the odds, which *corresponds the likelihood that an event will occur, relative to it not occurring.*
- The coefficients are most useful for insertion in formulas that give *predicted probabilities* for each level of the dependent variable.

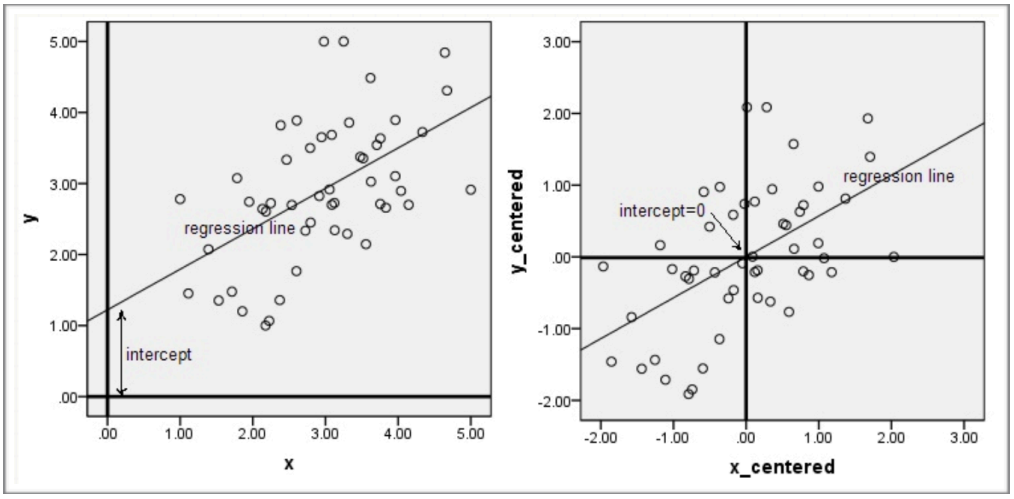
Odds ratio to interpret a logistic regression model:

- The odd of success is defined as ratio of probability of success to probability of failure.
- Odd ratio is usually used to study the effect of treatment on outcomes/result.
- The Odd ratio represents the odds with which an outcome will come up with given a particular treatment, compared to the odds of the outcome occurring in the absence of that treatment.
    - Odd Ratio = 1 Treatment doesn't affect the odds of outcome
    - Odd Ratio $\geq$ 1 Treatment increases the odd of outcome
    - Odd Ratio $\leq$ 1 Treatment decreases the odd of outcome

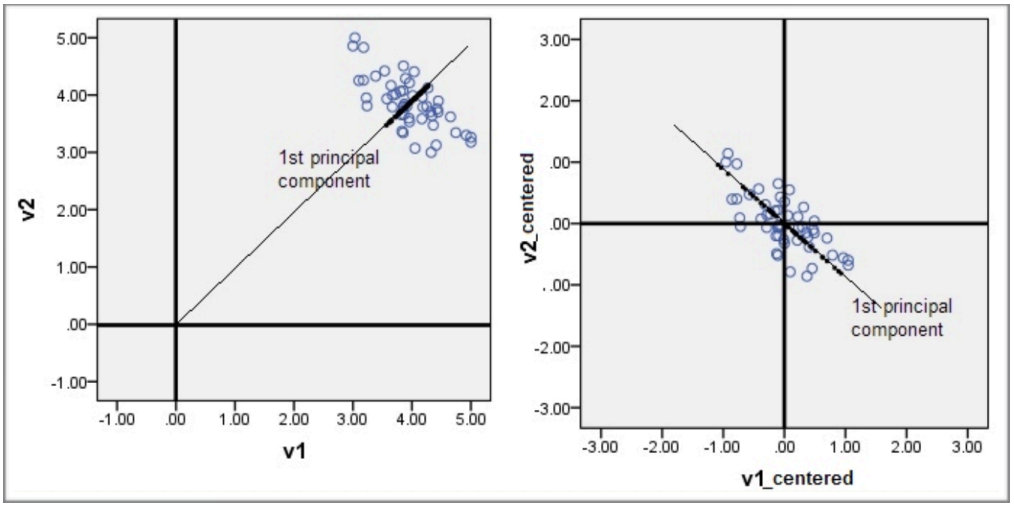
(d)

Principal component analysis (**PCA**) allows us to summarise and to visualise the information in a data set containing observations described by multiple inter-correlated quantitative variables. Each variable could be considered as a different dimension.
So firstly, to find the axes of the ellipse, we must first subtract the mean of each variable from the dataset to centre the data around the origin. Then, we compute the covariance matrix of the data. Then we will observe that this is not a very accurate formulation. PCA on non centred data. Will look like as shown below:

Here is the difference that can be seen what will happen if we don't centre the data.



If we don't centre then:
PCA is a regression model *without* intercept.
Thus, principal components inevitably come through the origin.
If we *forget to centre the data*, the 1st principal component may pierce the cloud, but not along the main direction of the cloud, and will be misleading, this can be seen below.



Normalising data before PCA:

- It is necessary to normalise data before performing PCA. The PCA calculates a new projection of the data set. And the new axis are based on the standard deviation of the variables.
- So a variable with a high standard deviation will have a higher weight for the calculation of axis than a variable with a low standard deviation.
- *If we normalise the data, all variables have the same standard deviation, thus all variables have the same weight and then PCA calculates the relevant axis.*
- It is also done to calculate the covariance matrix.
- Eigen vector decomposition is then applied to the covariance matrix, with the eigenvalues corresponding to the variance of the data along the eigenvectors.

When not to Normalise:

An alternative, but equivalent approach, is to use a *singular value decomposition (SVD)* based on singular vectors of the *data matrix*. *Then in this case Normalisation is not necessary. So, for single value decomposition we do not normalise as it is calculates through singular values using data matrix.*

References:

[1]: http://www.visiondummy.com/2014/04/curse-dimensionality-affect-classification/
[2]: http://www.edupristine.com/blog/curse-dimensionality
[3]: Lecture Notes on Feature selection.
[4]: http://nikhilbuduma.com/2015/03/10/the-curse-of-dimensionality/
[5]: https://stats.stackexchange.com/questions/120080/whatre-the-differences-between-pca-and-autoencoder
[6]: diagnosing-bias-vs-variance by Andrew Ng
[7]: Bias-Variance Tradeoff and Ensemble Methods by Tom Dietterich & Rich Maclin
[8]: http://www-scf.usc.edu/~csci567/17-18-bias-variance.pdf
[9]: https://stats.stackexchange.com/questions/22329/how-does-centering-the-data-get-rid-of-the-intercept-in-regression-and-pca
[10]: http://www.sthda.com/english/articles/31-principal-component-methods-in-r-practical-guide/112-pca-principal-component-analysis-essentials/
[11]: http://www.juanshishido.com/logisticcoefficients.html
[12]: Boosting k-nearest neighbor classifier by means of input space projection by Nicolás & Domingo.
[13]:Bagging, Boosting and the Random Subspace Method for Linear Classifiers by Marina Skurichina and Robert P. W. Duin.
[14]: https://quantdare.com/what-is-the-difference-between-bagging-and-boosting/17